EXPERT INSIGHT

Artificial Intelligence By Example

Acquire advanced AI, machine learning, and deep learning design skills





Second Edition

Denis Rothman

Packt

Artificial Intelligence By Example

Second Edition

Acquire advanced AI, machine learning, and deep learning design skills

Denis Rothman



BIRMINGHAM - MUMBAI

Artificial Intelligence By Example

Second Edition

Copyright © 2020 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Producer: Tushar Gupta

Acquisition Editor – Peer Reviews: Divya Mudaliar

Content Development Editor: Dr. Ian Hough

Technical Editor: Saby D'silva

Project Editor: Kishor Rit

Proofreader: Safis Editing

Indexer: Tejal Daruwale Soni

Presentation Designer: Pranit Padwal

First published: May 2018

Second edition: February 2020

Production reference: 1270220

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-83921-153-9

www.packt.com



packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Learn better with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.Packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.Packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and

offers on Packt books and eBooks.

Contributors

About the author

Denis Rothman graduated from Sorbonne University and Paris-Diderot University, writing one of the very first word2matrix embedding solutions. He began his career authoring one of the first AI cognitive **natural language processing** (**NLP**)chatbots applied as a language teacher for Moët et Chandon and other companies. He authored an AI resource optimizer for IBM and apparel producers. He then authored an **advanced planning and scheduling** (**APS**) solution used worldwide.

"I want to thank the corporations who trusted me from the start to deliver artificial intelligence solutions and share the risks of continuous innovation. I also thank my family, who believed I would make it big at all times."

About the reviewers

Carlos Toxtli is a human-computer interaction researcher who studies the impact of artificial intelligence in the future of work. He studied a Ph.D. in Computer Science at the University of West Virginia and a master's degree in Technological Innovation and Entrepreneurship at the Monterrey Institute of Technology and Higher Education. He has worked for some international organizations such as Google, Microsoft, Amazon, and the United Nations. He has also created companies that use artificial intelligence in the financial, educational, customer service, and parking industries. Carlos has published numerous research papers, manuscripts, and book chapters for different conferences and journals in his field.

"I want to thank all the editors who helped make this book a masterpiece."

Kausthub Raj Jadhav graduated from the University of California, Irvine, where he specialized in intelligent systems and founded the Artificial Intelligence Club. In his spare time, he enjoys powerlifting, rewatching Parks and Recreation, and learning how to cook. He solves hard problems for a living.

Table of Contents

- Preface
  - Who this book is for
  - What this book covers
  - To get the most out of this book
  - Get in touch

1. Getting Started with Next-Generation Artificial Intelligence through Reinforcement Learning
   - Reinforcement learning concepts
   - How to adapt to machine thinking and become an adaptive thinker
   - Overcoming real-life issues using the three-step approach
     - Step 1 – describing a problem to solve: MDP in natural language
       - Watching the MDP agent at work
     - Step 2 – building a mathematical model: the mathematical representation of the Bellman equation and MDP
       - From MDP to the Bellman equation
     - Step 3 – writing source code: implementing the solution in Python
   - The lessons of reinforcement learning
     - How to use the outputs
       - Possible use cases
     - Machine learning versus traditional applications
   - Summary
   - Questions
   - Further reading
2. Building a Reward Matrix – Designing Your Datasets
   - Designing datasets – where the dream stops and the hard work begins
     - Designing datasets

        - Using the McCulloch-Pitts neuron
        - The McCulloch-Pitts neuron
        - The Python-TensorFlow architecture
    - Logistic activation functions and classifiers
        - Overall architecture
        - Logistic classifier
        - Logistic function
        - Softmax
    - Summary
    - Questions
    - Further reading
3. Machine Intelligence – Evaluation Functions and Numerical Convergence
    - Tracking down what to measure and deciding how to measure it
        - Convergence
            - Implicit convergence
            - Numerically controlled gradient descent convergence
    - Evaluating beyond human analytic capacity
    - Using supervised learning to evaluate a result that surpasses human analytic capacity
    - Summary
    - Questions
    - Further reading
4. Optimizing Your Solutions with K-Means Clustering
    - Dataset optimization and control
        - Designing a dataset and choosing an ML/DL model
            - Approval of the design matrix
    - Implementing a k-means clustering solution
        - The vision
            - The data
            - The strategy

- The k-means clustering program
  - The mathematical definition of k-means clustering
  - The Python program
- Saving and loading the model
- Analyzing the results
  - Bot virtual clusters as a solution
  - The limits of the implementation of the k-means clustering algorithm
- Summary
- Questions
- Further reading

5. How to Use Decision Trees to Enhance K-Means Clustering
   - Unsupervised learning with KMC with large datasets
     - Identifying the difficulty of the problem
       - NP-hard – the meaning of P
       - NP-hard – the meaning of non-deterministic
     - Implementing random sampling with mini-batches
     - Using the LLN
     - The CLT
       - Using a Monte Carlo estimator
     - Trying to train the full training dataset
     - Training a random sample of the training dataset
     - Shuffling as another way to perform random sampling
     - Chaining supervised learning to verify unsupervised learning
       - Preprocessing raw data
     - A pipeline of scripts and ML algorithms
       - Step 1 – training and exporting data from an unsupervised ML algorithm
       - Step 2 – training a decision tree
       - Step 3 – a continuous cycle of KMC chained to a decision

tree
Random forests as an alternative to decision trees
Summary
Questions
Further reading
6. Innovating AI with Google Translate
Understanding innovation and disruption in AI
Is AI disruptive?
AI is based on mathematical theories that are not new
Neural networks are not new
Looking at disruption – the factors that are making AI disruptive
Cloud server power, data volumes, and web sharing of the early 21st century
Public awareness
Inventions versus innovations
Revolutionary versus disruptive solutions
Where to start?
Discover a world of opportunities with Google Translate
Getting started
The program
The header
Implementing Google's translation service
Google Translate from a linguist's perspective
Playing with the tool
Linguistic assessment of Google Translate
AI as a new frontier
Lexical field and polysemy
Exploring the frontier – customizing Google Translate with a Python program
k-nearest neighbor algorithm

Implementing the KNN algorithm
The knn_polysemy.py program
Implementing the KNN function in Google_Translate_Customized.py
Conclusions on the Google Translate customized experiment
The disruptive revolutionary loop
Summary
Questions
Further reading
7. Optimizing Blockchains with Naive Bayes
Part I – the background to blockchain technology
Mining bitcoins
Using cryptocurrency
PART II – using blockchains to share information in a supply chain
Using blockchains in the supply chain network
Creating a block
Exploring the blocks
Part III – optimizing a supply chain with naive Bayes in a blockchain process
A naive Bayes example
The blockchain anticipation novelty
The goal – optimizing storage levels using blockchain data
Implementation of naive Bayes in Python
Gaussian naive Bayes
Summary
Questions
Further reading
8. Solving the XOR Problem with a Feedforward Neural Network
The original perceptron could not solve the XOR function
XOR and linearly separable models

        - Linearly separable models
        - The XOR limit of a linear model, such as the original perceptron
    - Building an FNN from scratch
        - Step 1 – defining an FNN
        - Step 2 – an example of how two children can solve the XOR problem every day
        - Implementing a vintage XOR solution in Python with an FNN and backpropagation
            - A simplified version of a cost function and gradient descent
            - Linear separability was achieved
    - Applying the FNN XOR function to optimizing subsets of data
    - Summary
    - Questions
    - Further reading
9. Abstract Image Classification with Convolutional Neural Networks (CNNs)
    - Introducing CNNs
        - Defining a CNN
        - Initializing the CNN
        - Adding a 2D convolution layer
            - Kernel
            - Shape
            - ReLU
        - Pooling
        - Next convolution and pooling layer
        - Flattening
        - Dense layers
            - Dense activation functions
    - Training a CNN model
        - The goal

- Compiling the model
  - The loss function
  - The Adam optimizer
  - Metrics
- The training dataset
  - Data augmentation
  - Loading the data
- The testing dataset
  - Data augmentation on the testing dataset
  - Loading the data
- Training with the classifier
- Saving the model
  - Next steps
- Summary
- Questions
- Further reading and references

10. Conceptual Representation Learning
    - Generating profit with transfer learning
      - The motivation behind transfer learning
        - Inductive thinking
        - Inductive abstraction
        - The problem AI needs to solve
      - The Γ gap concept
      - Loading the trained TensorFlow 2.x model
        - Loading and displaying the model
        - Loading the model to use it
        - Defining a strategy
        - Making the model profitable by using it for another problem
    - Domain learning
      - How to use the programs

The trained models used in this section
The trained model program
Gap – loaded or underloaded
Gap – jammed or open lanes
Gap datasets and subsets
Generalizing the Γ (the gap conceptual dataset)
The motivation of conceptual representation learning metamodels applied to dimensionality
The curse of dimensionality
The blessing of dimensionality
Summary
Questions
Further reading
11. Combining Reinforcement Learning and Deep Learning
Planning and scheduling today and tomorrow
A real-time manufacturing process
Amazon must expand its services to face competition
A real-time manufacturing revolution
CRLMM applied to an automated apparel manufacturing process
An apparel manufacturing process
Training the CRLMM
Generalizing the unit training dataset
Food conveyor belt processing – positive pγ and negative nγ gaps
Running a prediction program
Building the RL-DL-CRLMM
A circular process
Implementing a CNN-CRLMM to detect gaps and optimize
Q-learning – MDP
MDP inputs and outputs

- The optimizer
  - The optimizer as a regulator
  - Finding the main target for the MDP function
- A circular model – a stream-like system that never starts nor ends
- Summary
- Questions
- Further reading

12. AI and the Internet of Things (IoT)
    - The public service project
    - Setting up the RL-DL-CRLMM model
      - Applying the model of the CRLMM
        - The dataset
        - Using the trained model
    - Adding an SVM function
      - Motivation – using an SVM to increase safety levels
      - Definition of a support vector machine
      - Python function
    - Running the CRLMM
      - Finding a parking space
      - Deciding how to get to the parking lot
        - Support vector machine
        - The itinerary graph
        - The weight vector
    - Summary
    - Questions
    - Further reading
13. Visualizing Networks with TensorFlow 2.x and TensorBoard
    - Exploring the output of the layers of a CNN in two steps with TensorFlow
      - Building the layers of a CNN

Processing the visual output of the layers of a CNN

Analyzing the visual output of the layers of a CNN

Analyzing the accuracy of a CNN using TensorBoard

Getting started with Google Colaboratory

Defining and training the model

Introducing some of the measurements

Summary

Questions

Further reading

14. Preparing the Input of Chatbots with Restricted Boltzmann Machines (RBMs) and Principal Component Analysis (PCA)

Defining basic terms and goals

Introducing and building an RBM

The architecture of an RBM

An energy-based model

Building the RBM in Python

Creating a class and the structure of the RBM

Creating a training function in the RBM class

Computing the hidden units in the training function

Random sampling of the hidden units for the reconstruction and contractive divergence

Reconstruction

Contrastive divergence

Error and energy function

Running the epochs and analyzing the results

Using the weights of an RBM as feature vectors for PCA

Understanding PCA

Mathematical explanation

Using TensorFlow's Embedding Projector to represent PCA

Analyzing the PCA to obtain input entry points for a chatbot

- Summary
- Questions
- Further reading

15. Setting Up a Cognitive NLP UI/CUI Chatbot
    - Basic concepts
        - Defining NLU
        - Why do we call chatbots "agents"?
        - Creating an agent to understand Dialogflow
        - Entities
        - Intents
        - Context
    - Adding fulfillment functionality to an agent
        - Defining fulfillment
        - Enhancing the cogfilmdr agent with a fulfillment webhook
        - Getting the bot to work on your website
    - Machine learning agents
        - Using machine learning in a chatbot
        - Speech-to-text
        - Text-to-speech
        - Spelling
        - Why are these machine learning algorithms important?
    - Summary
    - Questions
    - Further reading
16. Improving the Emotional Intelligence Deficiencies of Chatbots
    - From reacting to emotions, to creating emotions
        - Solving the problems of emotional polysemy
            - The greetings problem example
            - The affirmation example
            - The speech recognition fallacy

The facial analysis fallacy
Small talk
Courtesy
Emotions
Data logging
Creating emotions
RNN research for future automatic dialog generation
RNNs at work
RNN, LSTM, and vanishing gradients
Text generation with an RNN
Vectorizing the text
Building the model
Generating text
Summary
Questions
Further reading
17. Genetic Algorithms in Hybrid Neural Networks
Understanding evolutionary algorithms
Heredity in humans
Our cells
How heredity works
Evolutionary algorithms
Going from a biological model to an algorithm
Basic concepts
Building a genetic algorithm in Python
Importing the libraries
Calling the algorithm
The main function
The parent generation process
Generating a parent

Fitness
Display parent
Crossover and mutation
Producing generations of children
Summary code
Unspecified target to optimize the architecture of a neural network with a genetic algorithm
A physical neural network
What is the nature of this mysterious S-FNN?
Calling the algorithm cell
Fitness cell
ga_main() cell
Artificial hybrid neural networks
Building the LSTM
The goal of the model
Summary
Questions
Further reading
18. Neuromorphic Computing
Neuromorphic computing
Getting started with Nengo
Installing Nengo and Nengo GUI
Creating a Python program
A Nengo ensemble
Nengo neuron types
Nengo neuron dimensions
A Nengo node
Connecting Nengo objects
Visualizing data
Probes

- Applying Nengo's unique approach to critical AI research areas
- Summary
- Questions
- References
- Further reading

19. Quantum Computing
    - The rising power of quantum computers
        - Quantum computer speed
        - Defining a qubit
        - Representing a qubit
        - The position of a qubit
            - Radians, degrees, and rotations
            - The Bloch sphere
        - Composing a quantum score
            - Quantum gates with Quirk
            - A quantum computer score with Quirk
            - A quantum computer score with IBM Q
    - A thinking quantum computer
        - Representing our mind's concepts
        - Expanding MindX's conceptual representations
        - The MindX experiment
            - Preparing the data
            - Transformation functions – the situation function
            - Transformation functions – the quantum function
            - Creating and running the score
            - Using the output
    - Summary
    - Questions
    - Further reading
20. Answers to the Questions

Chapter 1 – Getting Started with Next-Generation Artificial Intelligence through Reinforcement Learning

Chapter 2 – Building a Reward Matrix – Designing Your Datasets

Chapter 3 – Machine Intelligence – Evaluation Functions and Numerical Convergence

Chapter 4 – Optimizing Your Solutions with K-Means Clustering

Chapter 5 – How to Use Decision Trees to Enhance K-Means Clustering

Chapter 6 – Innovating AI with Google Translate

Chapter 7 – Optimizing Blockchains with Naive Bayes

Chapter 8 – Solving the XOR Problem with a Feedforward Neural Network

Chapter 9 – Abstract Image Classification with Convolutional Neural Networks (CNNs)

Chapter 10 – Conceptual Representation Learning

Chapter 11 – Combining Reinforcement Learning and Deep Learning

Chapter 12 – AI and the Internet of Things

Chapter 13 – Visualizing Networks with TensorFlow 2.x and TensorBoard

Chapter 14 – Preparing the Input of Chatbots with Restricted Boltzmann Machines (RBMs) and Principal Component Analysis (PCA)

Chapter 15 – Setting Up a Cognitive NLP UI/CUI Chatbot

Chapter 16 – Improving the Emotional Intelligence Deficiencies of Chatbots

Chapter 17 – Genetic Algorithms in Hybrid Neural Networks

Chapter 18 – Neuromorphic Computing

Chapter 19 – Quantum Computing

Other Books You May Enjoy

Index

Preface

This second edition of *Artificial Intelligence By Example* will take you through the main aspects of present-day **artificial intelligence** (**AI**) and beyond!

This book contains many revisions and additions to the key aspects of AI described in the first edition:

- The theory of machine learning and deep learning including hybrid and ensemble algorithms.
- Mathematical representations of the main AI algorithms including natural language explanations making them easier to understand.
- Real-life case studies taking the reader inside the heart of e-commerce: manufacturing, services, warehouses, and delivery.
- Introducing AI solutions that combine IoT, **convolutional neural networks** (**CNN**), and **Markov decision process** (**MDP**).
- Many open source Python programs with a special focus on the new features of TensorFlow 2.x, TensorBoard, and Keras. Many modules are used, such as scikit-learn, pandas, and more.
- Cloud platforms: Google Colaboratory with its free VM, Google Translate, Google Dialogflow, IBM Q for quantum computing, and more.
- Use of the power of **restricted Boltzmann machines** (**RBM**) and **principal component analysis** (**PCA**) to generate data to create a meaningful chatbot.
- Solutions to compensate for the emotional deficiencies of chatbots.
- Genetic algorithms, which run faster than classical algorithms in specific cases, and genetic algorithms used in a hybrid deep learning neural network.
- Neuromorphic computing, which reproduces our brain activity with models of selective spiking ensembles of neurons in models that reproduce our biological reactions.
- Quantum computing, which will take you deep into the tremendous

calculation power of qubits and cognitive representation experiments.

This second edition of *Artificial Intelligence By Example* will take you to the cutting edge of AI and beyond with innovations that improve existing solutions. This book will make you a key asset not only as an AI specialist but a visionary. You will discover how to improve your AI skills as a consultant, developer, professor, a curious mind, or any person involved in artificial intelligence.

Who this book is for

This book contains a broad approach to AI, which is expanding to all areas of our lives.

The main machine learning and deep learning algorithms are addressed with real-life Python examples extracted from hundreds of AI projects and implementations.

Each AI implementation is illustrated by an open source program available on GitHub and cloud platforms such as Google Colaboratory.

Artificial Intelligence By Example, Second Edition is for developers who wish to build solid machine learning programs that will optimize production sites, services, IoT and more.

Project managers and consultants will learn how to build input datasets that will help the reader face the challenges of real-life AI.

Teachers and students will have an overview of the key aspects of AI, along with many educational examples.

Artificial Intelligence By Example, Second Edition will help anybody interested in AI to understand how systems to build solid, productive Python programs.

What this book covers

Chapter 1, *Getting Started with Next-Generation Artificial Intelligence through Reinforcement Learning*, covers reinforcement learning through the Bellman equation based on the MDP. A case study describes how to solve a delivery route problem with a human driver and a self-driving vehicle. This chapter shows how to build an MDP from scratch in Python.

Chapter 2, *Building a Reward Matrix – Designing Your Datasets*, demonstrates the architecture of neural networks starting with the McCulloch-Pitts neuron. The case study describes how to use a neural network to build the reward matrix used by the Bellman equation in a warehouse environment. The process will be developed in Python using logistic, softmax, and one-hot functions.

Chapter 3, *Machine Intelligence – Evaluation Functions and Numerical Convergence*, shows how machine evaluation capacities have exceeded human decision-making. The case study describes a chess position and how to apply the results of an AI program to decision-making priorities. An introduction to decision trees in Python shows how to manage decision-making processes.

Chapter 4, *Optimizing Your Solutions with K-Means Clustering*, covers a k-means clustering program with Lloyd's algorithm and how to apply it to the optimization of automatic guided vehicles. The k-means clustering program's model will be trained and saved.

Chapter 5, *How to Use Decision Trees to Enhance K-Means Clustering*, begins with unsupervised learning with k-means clustering. The output of the k-means clustering algorithm will provide the labels for the supervised decision tree algorithm. Random forests will be introduced.

Chapter 6, *Innovating AI with Google Translate*, explains the difference between a revolutionary innovation and a disruptive innovation. Google Translate will be described and enhanced with an innovative k-nearest neighbors-based Python program.

Chapter 7, *Optimizing Blockchains with Naive Bayes*, is about mining

blockchains and describes how blockchains function. We use naive Bayes to optimize the blocks of **supply chain management** (**SCM**) blockchains by predicting transactions to anticipate storage levels.

Chapter 8, Solving the XOR Problem with a Feedforward Neural Network, is about building a **feedforward neural network** (**FNN**) from scratch to solve the XOR linear separability problem. The business case describes how to group orders for a factory.

Chapter 9, Abstract Image Classification with Convolutional Neural Networks (CNNs), describes CNN in detail: kernels, shapes, activation functions, pooling, flattening, and dense layers. The case study illustrates the use of a CNN using a webcam on a conveyor belt in a food-processing company.

Chapter 10, Conceptual Representation Learning, explains **conceptual representation learning** (**CRL**), an innovative way to solve production flows with a CNN transformed into a **CRL metamodel** (**CRLMM**). The case study shows how to use a CRLMM for transfer and domain learning, extending the model to other applications.

Chapter 11, Combining Reinforcement Learning and Deep Learning, combines a CNN with an MDP to build a solution for automatic planning and scheduling with an optimizer with a rule-based system.

The solution is applied to apparel manufacturing showing how to apply AI to real-life systems.

Chapter 12, AI and the Internet of Things (IoT), explores a **support vector machine** (**SVM**) assembled with a CNN. The case study shows how self-driving cars can find an available parking space automatically.

Chapter 13, Visualizing Networks with TensorFlow 2.x and TensorBoard, extracts information of each layer of a CNN and displays the intermediate steps taken by the neural network. The output of each layer contains images of the transformations applied.

Chapter 14, Preparing the Input of Chatbots with Restricted Boltzmann Machines (RBM) and Principal Component Analysis (PCA), explains how to produce valuable information using an RBM and a PCA to transform raw data into chatbot-input data.

Chapter 15, Setting Up a Cognitive NLP UI/CUI Chatbot, describes how to build a Google Dialogflow chatbot from scratch using the information provided by an RBM and a PCA algorithm. The chatbot will contain entities, intents, and meaningful responses.

Chapter 16, Improving the Emotional Intelligence Deficiencies of Chatbots, explains the limits of a chatbot when dealing with human emotions.
The **Emotion** options of Dialogflow will be activated along with **Small Talk** to make the chatbot friendlier.

Chapter 17, Genetic Algorithms in Hybrid Neural Networks, enters our chromosomes, finds our genes, and helps you understand how our reproduction process works. From there, it is shown how to implement an evolutionary algorithm in Python, a **genetic algorithm** (**GA**). A hybrid neural network will show how to optimize a neural network with a GA.

Chapter 18, Neuromorphic Computing, describes what neuromorphic computing is and then explores Nengo, a unique neuromorphic framework with solid tutorials and documentation.

This neuromorphic overview will take you into the wonderful power of our brain structures to solve complex problems.

Chapter 19, Quantum Computing, will show quantum computers are superior to classical computers, what a quantum bit is, how to use it, and how to build quantum circuits. An introduction to quantum gates and example programs will bring you into the futuristic world of quantum mechanics.

Appendix, Answers to the Questions, provides answers to the questions listed in the *Questions* section in all the chapters.

To get the most out of this book

Artificial intelligence projects rely on three factors:

- **Understanding the subject the AI project will be applied to**. To do so, go through a chapter to pick up the key ideas. Once you understand the key ideas of a case study described in the book, try to see how an AI solution can be applied to real-life examples around you.
- **The mathematical foundations of the AI algorithms**. Do not skip the mathematics equations if you have the energy to study them. AI relies heavily on mathematics. There are plenty of excellent websites that explain the mathematics used in this book.
- **Development**. An artificial intelligence solution can be directly used on an online cloud platform machine learning site such as Google. We can access these platforms with APIs. In the book, Google Cloud is used several times. Try to create an account of your own to explore several cloud platforms to understand their potential and their limits. Development remains critical for AI projects.

Even with a cloud platform, scripts and services are necessary. Also, sometimes, writing an algorithm is mandatory because the ready-to-use online algorithms are insufficient for a given problem. Explore the programs delivered with the book. They are open source and free.

Technical requirements

The following is a non-exhaustive list of the technical requirements for running the codes in this book. For a more detailed chapter-wise list, please refer to this link: https://github.com/PacktPublishing/Artificial-Intelligence-By-Example-Second-Edition/blob/master/Technical%20Requirements.csv.

Package	Website
Python	https://www.python.org/

NumPy	https://pypi.org/project/numpy/
Matplotlib	https://pypi.org/project/matplotlib/
pandas	https://pypi.org/project/pandas/
SciPy	https://pypi.org/project/scipy/
scikit-learn	https://pypi.org/project/scikit-learn/
PyDotPlus	https://pypi.org/project/pydotplus/
Google API	https://developers.google.com/docs/api/quickstart/python
html	https://pypi.org/project/html/
TensorFlow 2	https://pypi.org/project/tensorflow/
Keras	https://pypi.org/project/Keras/
Pillow	https://pypi.org/project/Pillow/
Imageio	https://pypi.org/project/imageio/
Pathlib	https://pypi.org/project/pathlib/
OpenCV-Python	https://pypi.org/project/opencv-python/
Google Dialogflow	https://dialogflow.com/
DEAP	https://pypi.org/project/deap/

bitstring	https://pypi.org/project/bitstring/
nengo	https://pypi.org/project/nengo/
nengo-gui	https://pypi.org/project/nengo-gui/
IBM Q	https://www.research.ibm.com/ibm-q/
Quirk	http://algassert.com/2016/05/22/quirk.html

Download the example code files

You can download the example code files for this book from your account at `www.packt.com/`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `http://www.packt.com`.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the on-screen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at

https://github.com/PacktPublishing/Artificial-Intelligence-By-Example-Second-Edition. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781839211539_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example; "The decision tree program, `decision_tree.py`, reads the output of the KMC predictions, `ckmc.csv`."

A block of code is set as follows:

```
# load dataset
col_names = ['f1', 'f2','label']
df = pd.read_csv("ckmc.csv", header=None, names=col_names)
if pp==1:
    print(df.head())
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
for i in range(0,1000):
    xf1=dataset.at[i,'Distance']
    xf2=dataset.at[i,'location']
```

```
    X_DL = [[xf1,xf2]]
    prediction = kmeans.predict(X_DL)
```

Any command-line input or output is written as follows:

```
Selection: BnVYkFcRK Fittest: 0 This generation Fitness: 0
Time Difference: 0:00:00.000198
```

Bold: Indicates a new term, an important word, or words that you see on the screen, for example, in menus or dialog boxes, also appear in the text like this. For example: "When you click on **SAVE**, the **Emotions** progress bar will jump up."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book we would be grateful if you would report this to us. Please visit, `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1

Getting Started with Next-Generation Artificial Intelligence through Reinforcement Learning

Next-generation AI compels us to realize that machines do indeed think. Although machines do not think like us, their thought process has proven its efficiency in many areas. In the past, the belief was that AI would reproduce human thinking processes. Only neuromorphic computing (see *Chapter 18, Neuromorphic Computing*), remains set on this goal. Most AI has now gone beyond the way humans think, as we will see in this chapter.

The **Markov decision process** (**MDP**), a **reinforcement learning** (**RL**) algorithm, perfectly illustrates how machines have become intelligent in their own unique way. Humans build their decision process on experience. MDPs are memoryless. Humans use logic and reasoning to think problems through. MDPs apply random decisions 100% of the time. Humans think in words, labeling everything they perceive. MDPs have an unsupervised approach that uses no labels or training data. MDPs boost the machine thought process of self-driving cars (SDCs), translation tools, scheduling software, and more. This memoryless, random, and unlabeled machine thought process marks a historical change in the way a former human problem was solved.

With this realization comes a yet more mind-blowing fact. AI algorithms and hybrid solutions built on IoT, for example, have begun to surpass humans in strategic areas. Although AI cannot replace humans in every field, AI combined with classical automation now occupies key domains: banking, marketing, supply chain management, scheduling, and many other critical areas.

As you will see, starting with this chapter, you can occupy a central role in this new world as an adaptive thinker. You can design AI solutions and implement

them. There is no time to waste. In this chapter, we are going to dive quickly and directly into reinforcement learning through the MDP.

Today, AI is essentially mathematics translated into source code, which makes it difficult to learn for traditional developers. However, we will tackle this approach pragmatically.

The goal here is not to take the easy route. We're striving to break complexity into understandable parts and confront them with reality. You are going to find out right from the outset how to apply an adaptive thinker's process that will lead you from an idea to a solution in reinforcement learning, and right into the center of gravity of the next generation of AI.

Reinforcement learning concepts

AI is constantly evolving. The classical approach states that:

- AI covers all domains
- Machine learning is a subset of AI, with clustering, classification, regression, and reinforcement learning
- Deep learning is a subset of machine learning that involves neural networks

However, these domains often overlap and it's difficult to fit neuromorphic computing, for example, with its sub-symbolic approach, into these categories (see *Chapter 18, Neuromorphic Computing*).

In this chapter, RL clearly fits into machine learning. Let's have a brief look into the scientific foundations of the MDP, the RL algorithm we are going to explore. The main concepts to keep in mind are the following:

- **Optimal transport**: In 1781, Gaspard Monge defined transport optimizing from one location to another using the shortest and most cost-effective path; for example, mining coal and then using the most cost-effective path to a factory. This was subsequently generalized to any form of path from point A to point B.
- **Boltzmann equation and constant**: In the late 19th century, Ludwig

Boltzmann changed our vision of the world with his probabilistic distribution of particles beautifully summed up in his entropy formula:

$$S = k * \log W$$

S represents the entropy (energy, disorder) of a system expressed. *k* is the Boltzmann constant, and *W* represents the number of microstates. We will explore Boltzmann's ideas further in *Chapter 14*, *Preparing the Input of Chatbots with Restricted Boltzmann Machines (RBMs) and Principal Component Analysis (PCA)*.

- **Probabilistic distributions advanced further**: Josiah Willard Gibbs took the probabilistic distributions of large numbers of particles a step further. At that point, probabilistic information theory was advancing quickly. At the turn of the 19th century, Andrey Markov applied probabilistic algorithms to language, among other areas. A modern era of information theory was born.
- **When Boltzmann and optimal transport meet**: 2011 Fields Medal winner, Cédric Villani, brought Boltzmann's equation to yet another level. Villani then went on to unify optimal transport and Boltzmann. Cédric Villani proved something that was somewhat intuitively known to 19th century mathematicians but required proof.

Let's take all of the preceding concepts and materialize them in a real-world example that will explain why reinforcement learning using the MDP, for example, is so innovative.

Analyzing the following cup of tea will take you right into the next generation of AI:



Figure 1.1: Consider a cup of tea

You can look at this cup of tea in two different ways:

1. **Macrostates**: You look at the cup and content. You can see the volume of tea in the cup and you could feel the temperature when holding the cup in your hand.
2. **Microstates**: But can you tell how many molecules are in the tea, which ones are hot, warm, or cold, their velocity and directions? Impossible right?

Now, imagine, the tea contains 2,000,000,000+ Facebook accounts, or 100,000,000+ Amazon Prime users with millions of deliveries per year. At this level, we simply abandon the idea of controlling every item. We work on trends and probabilities.

Boltzmann provides a probabilistic approach to the evaluation of the features of our real world. Materializing Boltzmann in logistics through optimal transport means that the temperature could be the ranking of a product, the velocity can be linked to the distance to delivery, and the direction could be the itineraries we will study in this chapter.

Markov picked up the ripe fruits of microstate probabilistic descriptions and applied it to his MDP. Reinforcement learning takes the huge volume of elements (particles in a cup of tea, delivery locations, social network accounts) and defines the probable paths they take.

The turning point of human thought occurred when we simply could not analyze the state and path of the huge volumes facing our globalized world, which generates images, sounds, words, and numbers that exceed traditional software approaches.

With this in mind, we can start exploring the MDP.

How to adapt to machine thinking and become an adaptive thinker

Reinforcement learning, one of the foundations of machine learning, supposes learning through trial and error by interacting with an environment. This sounds familiar, doesn't it? That is what we humans do all our lives—in pain! Try things, evaluate, and then continue; or try something else.

In real life, you are the agent of your thought process. In reinforcement learning, the agent is the function calculating randomly through this trial-and-error process. This thought process function in machine learning is the MDP agent. This form of empirical learning is sometimes called Q-learning.

Mastering the theory and implementation of an MDP through a three-step method is a prerequisite.

This chapter will detail the three-step approach that will turn you into an AI expert, in general terms:

1. Starting by describing a problem to solve with real-life cases
2. Then, building a mathematical model that considers real-life limitations
3. Then, writing source code or using a cloud platform solution

This is a way for you to approach any project with an adaptive attitude from the outset. This shows that a human will always be at the center of AI by explaining how we can build the inputs, run an algorithm, and use the results of our code. Let's consider this three-step process and put it into action.

Overcoming real-life issues using the three-step approach

The key point of this chapter is to avoid writing code that will never be used. First, begin by understanding the subject as a subject matter expert. Then, write the analysis with words and mathematics to make sure your reasoning reflects the subject and, most of all, that the program will make sense in real life. Finally, in step 3, only write the code when you are sure about the whole project.

Too many developers start writing code without stopping to think about how the results of that code are going to manifest themselves within real-life situations. You could spend weeks developing the perfect code for a problem, only to find out that an external factor has rendered your solution useless. For instance, what if you coded a solar-powered robot to clear snow from the yard, only to discover that during winter, there isn't enough sunlight to power the robot!

In this chapter, we are going to tackle the MDP (Q function) and apply it to reinforcement learning with the Bellman equation. We are going to approach it a little differently to most, however. We'll be thinking about practical application, not simply code execution. You can find tons of source code and examples on the web. The problem is, much like our snow robot, such source code rarely considers the complications that come about in real-life situations. Let's say you find a program that finds the optimal path for a drone delivery. There's an issue, though; it has many limits that need to be overcome due to the fact that the code has not been written with real-life practicality in mind. You, as an adaptive thinker, are going to ask some questions:

- What if there are 5,000 drones over a major city at the same time? What happens if they try to move in straight lines and bump into each other?
- Is a drone-jam legal? What about the noise over the city? What about tourism?
- What about the weather? Weather forecasts are difficult to make, so how is this scheduled?
- How can we resolve the problem of coordinating the use of charging and parking stations?

In just a few minutes, you will be at the center of attention among theoreticians who know more than you, on one hand, and angry managers who want solutions they cannot get on the other. Your real-life approach will solve these problems. To do that, you must take the following three steps into account, starting with really getting involved in the real-life subject.

In order to successfully implement our real-life approach, comprised of the three steps outlined in the previous section, there are a few prerequisites:

- **Be a subject matter expert (SME)**: First, you have to be an SME. If a theoretician geek comes up with a hundred TensorFlow functions to solve a drone trajectory problem, you now know it is going to be a tough ride in which real-life parameters are constraining the algorithm. An SME knows the subject and thus can quickly identify the critical factors of a given field. AI often requires finding a solution to a complex problem that even an expert in a given field cannot express mathematically. Machine learning sometimes means finding a solution to a problem that humans do not know how to explain. Deep learning, involving complex networks, solves even more difficult problems.
- **Have enough mathematical knowledge to understand AI concepts**: Once you have the proper natural language analysis, you need to build your abstract representation quickly. The best way is to look around and find an everyday life example and make a mathematical model of it. Mathematics is not an option in AI, but a prerequisite. The effort is worthwhile. Then, you can start writing a solid piece of source code or start implementing a cloud platform ML solution.
- **Know what source code is about as well as its potential and limits**: MDP is an excellent way to go and start working on the three dimensions that will make you adaptive: describing what is around you in detail in words, translating that into mathematical representations, and then implementing the result in your source code.

With those prerequisites in mind, let's look at how you can become a problem-solving AI expert by following our practical three-step process. Unsurprisingly, we'll begin at step 1.

Step 1 – describing a problem to solve: MDP in

natural language

Step 1 of any AI problem is to go as far as you can to understand the subject you are asked to represent. If it's a medical subject, don't just look at data; go to a hospital or a research center. If it's a private security application, go to the places where they will need to use it. If it's for social media, make sure to talk to many users directly. The key concept to bear in mind is that you have to get a "feel" for the subject, as if you were the real "user."

For example, transpose it into something you know in your everyday life (work or personal), something you are an SME in. If you have a driver's license, then you are an SME of driving. You are certified. This is a fairly common certification, so let's use this as our subject matter in the example that will follow. If you do not have a driver's license or never drive, you can easily replace moving around in a car by imagining you are moving around on foot; you are an SME of getting from one place to another, regardless of what means of transport that might involve. However, bear in mind that a real-life project would involve additional technical aspects, such as traffic regulations for each country, so our imaginary SME does have its limits.

Getting into the example, let's say you are an e-commerce business driver delivering a package in a location you are unfamiliar with. You are the operator of a self-driving vehicle. For the time being, you're driving manually. You have a GPS with a nice color map on it. The locations around you are represented by the letters **A** to **F**, as shown in the simplified map in the following diagram. You are presently at **F**. Your goal is to reach location **C**. You are happy, listening to the radio. Everything is going smoothly, and it looks like you are going to be there on time. The following diagram represents the locations and routes that you can cover:

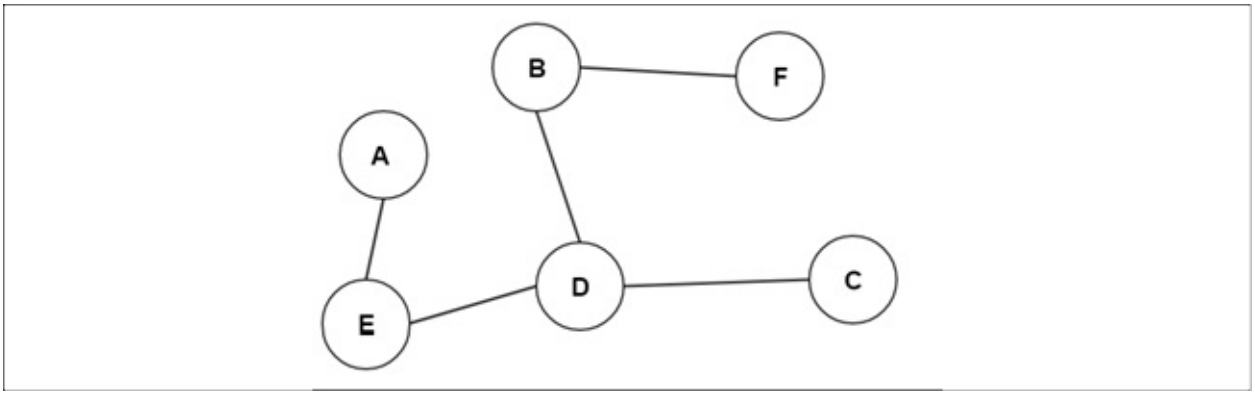


Figure 1.2: A diagram of delivery routes

The guidance system's state indicates the complete path to reach **C**. It is telling you that you are going to go from **F** to **B** to **D**, and then to **C**. It looks good!

To break things down further, let's say:

- The present state is the letter *s*. *s* is a variable, not an actual state. It can be one of the locations in *L*, the set of locations:

  $$L = \{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}\}$$

  We say *present state* because there is no sequence in the learning process. The memoryless process goes from one present state to another. In the example in this chapter, the process starts at location **F**.

- Your next action is the letter *a* (action). This action *a* is not location **A**. The goal of this action is to take us to the next possible location in the graph. In this case, only **B** is possible. The goal of *a* is to take us from *s* (present state) to *s′* (new state).
- The action *a* (not location **A**) is to go to location **B**. You look at your guidance system; it tells you there is no traffic, and that to go from your present state, **F**, to your next state, **B**, will take you only a few minutes. Let's say that the next state **B** is the letter **B**. This next state **B** is *s′*.

At this point, you are still quite happy, and we can sum up your situation with the following sequence of events:

$$s, a, s'$$

The letter *s* is your present state, your present situation. The letter *a* is the action you're deciding, which is to go to the next location; there, you will be in another state, *s′*. We can say that thanks to the action *a*, you will go from *s* to *s′*.

Now, imagine that the driver is not you anymore. You are tired for some reason. That is when a self-driving vehicle comes in handy. You set your car to autopilot. Now, you are no longer driving; the system is. Let's call that system the **agent**. At point **F**, you set your car to autopilot and let the self-driving agent take over.

Watching the MDP agent at work

The self-driving AI is now in charge of the vehicle. It is acting as the MDP agent. This now sees what you have asked it to do and checks its **mapping environment**, which represents all the locations in the previous diagram from **A** to **F**.

In the meantime, you are rightly worried. Is the agent going to make it or not? You are wondering whether its strategy meets yours. You have your **policy** *P*—your way of thinking—which is to take the shortest path possible. Will the agent agree? What's going on in its machine mind? You observe and begin to realize things you never noticed before.

Since this is the first time you are using this car and guidance system, the agent is **memoryless**, which is an MDP feature. The agent doesn't know anything about what went on before. It seems to be happy with just calculating from this state *s* at location **F**. It will use machine power to run as many calculations as necessary to reach its goal.

Another thing you are watching is the total distance from **F** to **C** to check whether things are OK. That means that the agent is calculating all the states from **F** to **C**.

In this case, state **F** is state 1, which we can simplify by writing s_1; B is state 2, which we can simplify by writing s_2; D is s_3; and C is s_4. The agent is calculating all of these possible states to make a decision.

The agent knows that when it reaches **D**, **C** will be better because the reward will be higher for going to C than anywhere else. Since it cannot eat a piece of cake to reward itself, the agent uses numbers. Our agent is a real number cruncher. When it is wrong, it gets a poor reward or nothing in this model. When it's right, it gets a reward represented by the letter *R*, which we'll encounter during step 2. This action-value (reward) transition, often named the Q function, is the core of many reinforcement learning algorithms.

When our agent goes from one state to another, it performs a *transition* and gets a reward. For example, the transition can be from **F** to **B**, state 1 to state 2, or s_1 to s_2.

You are feeling great and are going to be on time. You are beginning to understand how the machine learning agent in your self-driving car is thinking. Suddenly, you look up and see that a traffic jam is building up. Location **D** is still far away, and now you do not know whether it would be good to go from **D** to **C** or **D** to **E**, in order to take another road to **C**, which involves less traffic. You are going to see what your agent thinks!

The agent takes the traffic jam into account, is stubborn, and increases its reward to get to **C** by the shortest way. Its policy is to stick to the initial plan. You do not agree. You have another policy.

You stop the car. You both have to agree before continuing. You have your opinion and policy; the agent does not agree. Before continuing, your views need to **converge**. **Convergence** is the key to making sure that your calculations are correct, and it's a way to evaluate the quality of a calculation.

A mathematical representation is the best way to express this whole process at this point, which we will describe in the following step.

Step 2 – building a mathematical model: the mathematical representation of the Bellman equation and MDP

Mathematics involves a whole change in your perspective of a problem. You are going from words to functions, the pillars of source coding.

Expressing problems in mathematical notation does not mean getting lost in academic math to the point of never writing a single line of code. Just use mathematics to get a job done efficiently. Skipping mathematical representation will fast-track a few functions in the early stages of an AI project. However, when the real problems that occur in all AI projects surface, solving them with source code alone will prove virtually impossible. The goal here is to pick up enough mathematics to implement a solution in real-life companies.

It is necessary to think through a problem by finding something familiar around us, such as the itinerary model covered early in this chapter. It is a good thing to write it down with some abstract letters and symbols as described before, with *a*

meaning an action, and *s* meaning a state. Once you have understood the problem and expressed it clearly, you can proceed further.

Now, mathematics will help to clarify the situation by means of shorter descriptions. With the main ideas in mind, it is time to convert them into equations.

From MDP to the Bellman equation

In step 1, the agent went from **F**, or state 1 or *s*, to **B**, which was state 2 or *s′*.

A strategy drove this decision—a policy represented by *P*. One mathematical expression contains the MDP state transition function:

$$P_a(s, s')$$

P is the policy, the strategy made by the agent to go from **F** to **B** through action *a*. When going from **F** to **B**, this state transition is named the **state transition function**:

- *a* is the action
- *s* is state 1 (**F**), and *s′* is state 2 (**B**)

The reward (right or wrong) matrix follows the same principle:

$$R_a(s, s')$$

That means *R* is the reward for the action of going from state *s* to state *s′*. Going from one state to another will be a random process. Potentially, all states can go to any other state.

Each line in the matrix in the example represents a letter from **A** to **F**, and each column represents a letter from **A** to **F**. All possible states are represented. The `1` values represent the nodes (vertices) of the graph. Those are the possible locations. For example, line 1 represents the possible moves for letter **A**, line 2 for letter **B**, and line 6 for letter **F**. On the first line, **A** cannot go to **C** directly, so a `0` value is entered. But, it can go to **E**, so a `1` value is added.

Some models start with `-1` for impossible choices, such as **B** going directly to

C, and `0` values to define the locations. This model starts with `0` and `1` values. It sometimes takes weeks to design functions that will create a reward matrix (see *Chapter 2*, *Building a Reward Matrix – Designing Your Datasets*).

The example we will be working on inputs a reward matrix so that the program can choose its best course of action. Then, the agent will go from state to state, learning the best trajectories for every possible starting location point. The goal of the MDP is to go to **C** (line 3, column 3 in the reward matrix), which has a starting value of 100 in the following Python code:

```
# Markov Decision Process (MDP) - The Bellman equations ada
pted to
# Reinforcement Learning
import numpy as ql
# R is The Reward Matrix for each state
R = ql.matrix([ [0,0,0,0,1,0],
                [0,0,0,1,0,1],
                [0,0,100,1,0,0],
                [0,1,1,0,1,0],
                [1,0,0,1,0,0],
                [0,1,0,0,0,0] ])
```

Somebody familiar with Python might wonder why I used `ql` instead of `np`. Some might say "convention," "mainstream," "standard." My answer is a question. Can somebody define what "standard" AI is in this fast-moving world! My point here for the MDP is to use `ql` as an abbreviation of "Q-learning" instead of the "standard" abbreviation of NumPy, which is `np`. Naturally, beyond this special abbreviation for the MDP programs, I'll use `np`. Just bear in mind that conventions are there to break so as to set ourselves free to explore new frontiers. Just make sure your program works well!

There are several key properties of this decision process, among which there is the following:

- **The Markov property**: The process does not take the past into account. It is the memoryless property of this decision process, just as you do in a car with a guidance system. You move forward to reach your goal.
- **Unsupervised learning**: From this memoryless Markov property, it is safe to say that the MDP is not supervised learning. Supervised learning would mean that we would have all the labels of the reward matrix *R* and learn

from them. We would know what **A** means and use that property to make a decision. We would, in the future, be looking at the past. MDP does not take these labels into account. Thus, MDP uses unsupervised learning to train. A decision has to be made in each state without knowing the past states or what they signify. It means that the car, for example, was on its own at each location, which is represented by each of its states.

- **Stochastic process**: In step 1, when state **D** was reached, the agent controlling the mapping system and the driver didn't agree on where to go. A random choice could be made in a trial-and-error way, just like a coin toss. It is going to be a heads-or-tails process. The agent will toss the coin a significant number of times and measure the outcomes. That's precisely how MDP works and how the agent will learn.
- **Reinforcement learning**: Repeating a trial-and-error process with feedback from the agent's environment.
- **Markov chain**: The process of going from state to state with no history in a random, stochastic way is called a Markov chain.

To sum it up, we have three tools:

- $P_a(s, s')$: A **policy**, *P*, or strategy to move from one state to another
- $T_a(s, s')$: A *T*, or stochastic (random) **transition**, function to carry out that action
- $R_a(s, s')$: An *R*, or **reward**, for that action, which can be negative, null, or positive

T is the transition function, which makes the agent decide to go from one point to another with a policy. In this case, it will be random. That's what machine power is for, and that is how reinforcement learning is often implemented.

Randomness

Randomness is a key property of MDP, defining it as a stochastic process.

The following code describes the choice the **agent** is going to make:

```
next_action = int(ql.random.choice(PossibleAction,1))
return next_action
```

The code selects a new random action (state) at each episode.

The Bellman equation

The Bellman equation is the road to programming reinforcement learning.

The Bellman equation completes the MDP. To calculate the value of a state, let's use *Q*, for the *Q* action-reward (or value) function. The pseudo source code of the Bellman equation can be expressed as follows for one individual state:

$$Q(s) = R(s) + \gamma * \max(s')$$

The source code then translates the equation into a machine representation, as in the following code:

```
# The Bellman equation
    Q[current_state, action] = R[current_state, action] +
        gamma * MaxValue
```

The source code variables of the Bellman equation are as follows:

- *Q*(*s*): This is the value calculated for this state—the total reward. In step 1, when the agent went from **F** to **B**, the reward was a number such as 50 or 100 to show the agent that it's on the right track.
- *R*(*s*): This is the sum of the values up to that point. It's the total reward at that point.
- $\gamma = \mathbf{gamma}$: This is here to remind us that trial and error has a price. We're wasting time, money, and energy. Furthermore, we don't even know whether the next step is right or wrong since we're in a trial-and-error mode. **Gamma** is often set to 0.8. What does that mean? Suppose you're taking an exam. You study and study, but you don't know the outcome. You might have 80 out of 100 (0.8) chances of clearing it. That's painful, but that's life. The **gamma** penalty, or learning rate, makes the Bellman equation realistic and efficient.
- max(*s'*): *s'* is one of the possible states that can be reached with $P_a(s, s')$; max is the highest value on the line of that state (location line in the reward matrix).

At this point, you have done two-thirds of the job: understanding the real-life (process) and representing it in basic mathematics. You've built the mathematical model that describes your learning process, and you can implement that solution in code. Now, you are ready to code!

Step 3 – writing source code: implementing the solution in Python

In step 1, a problem was described in natural language to be able to talk to experts and understand what was expected. In step 2, an essential mathematical bridge was built between natural language and source coding. Step 3 is the software implementation phase.

When a problem comes up—and rest assured that one always does—it will be possible to go back over the mathematical bridge with the customer or company team, and even further back to the natural language process if necessary.

This method guarantees success for any project. The code in this chapter is in Python 3.x. It is a reinforcement learning program using the Q function with the following reward matrix:

```
import numpy as ql
R = ql.matrix([ [0,0,0,0,1,0],
                [0,0,0,1,0,1],
                [0,0,100,1,0,0],
                [0,1,1,0,1,0],
                [1,0,0,1,0,0],
                [0,1,0,0,0,0] ])
Q = ql.matrix(ql.zeros([6,6]))
gamma = 0.8
```

`R` is the reward matrix described in the mathematical analysis.

`Q` inherits the same structure as `R`, but all values are set to `0` since this is a learning matrix. It will progressively contain the results of the decision process. The `gamma` variable is a double reminder that the system is learning and that its decisions have only an 80% chance of being correct each time. As the following code shows, the system explores the possible actions during the process:

```
agent_s_state = 1
# The possible "a" actions when the agent is in a given sta
te
def possible_actions(state):
    current_state_row = R[state,]
    possible_act = ql.where(current_state_row >0)[1]
    return possible_act
# Get available actions in the current state
PossibleAction = possible_actions(agent_s_state)
```

The agent starts in state 1, for example. You can start wherever you want because it's a random process. Note that the process only takes values > 0 into account. They represent possible moves (decisions).

The current state goes through an analysis process to find possible actions (next possible states). You will note that there is no algorithm in the traditional sense with many rules. It's a pure random calculation, as the following `random.choice` function shows:

```
def ActionChoice(available_actions_range):
    if(sum(PossibleAction)>0):
        next_action = int(ql.random.choice(PossibleAction,1
))
    if(sum(PossibleAction)<=0):
        next_action = int(ql.random.choice(5,1))
    return next_action
# Sample next action to be performed
action = ActionChoice(PossibleAction)
```

Now comes the core of the system containing the Bellman equation, translated into the following source code:

```
def reward(current_state, action, gamma):
    Max_State = ql.where(Q[action,] == ql.max(Q[action,]))[1
]
    if Max_State.shape[0] > 1:
        Max_State = int(ql.random.choice(Max_State, size = 1
))
    else:
        Max_State = int(Max_State)
    MaxValue = Q[action, Max_State]

    # Q function
```

```
    Q[current_state, action] = R[current_state, action] +
        gamma * MaxValue
# Rewarding Q matrix
reward(agent_s_state,action,gamma)
```

You can see that the agent looks for the maximum value of the next possible state chosen at random.

The best way to understand this is to run the program in your Python environment and `print()` the intermediate values. I suggest that you open a spreadsheet and note the values. This will give you a clear view of the process.

The last part is simply about running the learning process 50,000 times, just to be sure that the system learns everything there is to find. During each iteration, the agent will detect its present state, choose a course of action, and update the Q function matrix:

```
for i in range(50000):
    current_state = ql.random.randint(0, int(Q.shape[0]))
    PossibleAction = possible_actions(current_state)
    action = ActionChoice(PossibleAction)
    reward(current_state,action,gamma)

# Displaying Q before the norm of Q phase
print("Q :")
print(Q)
# Norm of Q
print("Normed Q :")
print(Q/ql.max(Q)*100)
```

The process continues until the learning process is over. Then, the program will print the result in `Q` and the normed result. The normed result is the process of dividing all values by the sum of the values found. `print(Q/ql.max(Q)*100)` norms `Q` by dividing `Q` by `q1.max(Q)*100`. The result comes out as a normed percentage.

You can run the process with `mdp01.py`.

The lessons of reinforcement learning

Unsupervised reinforcement machine learning, such as the MDP-driven Bellman equation, is toppling traditional decision-making software location by location. Memoryless reinforcement learning requires few to no business rules and, thus, doesn't require human knowledge to run.

Being an adaptive next-generation AI thinker involves three prerequisites: the effort to be an SME, working on mathematical models to think like a machine, and understanding your source code's potential and limits.

Machine power and reinforcement learning teach us two important lessons:

- **Lesson 1**: Machine learning through reinforcement learning can beat human intelligence in many cases. No use fighting! The technology and solutions are already here in strategic domains.
- **Lesson 2**: A machine has no emotions, but you do. And so do the people around you. Human emotions and teamwork are an essential asset. Become an SME for your team. Learn how to understand what they're trying to say intuitively and make a mathematical representation of it for them. Your job will never go away, even if you're setting up solutions that don't require much development, such as AutoML. AutoML, or automated machine learning, automates many tasks. AutoML automates functions such as the dataset pipeline, hyperparameters, and more. Development is partially or totally suppressed. But you still have to make sure the whole system is well designed.

Reinforcement learning shows that no human can solve a problem the way a machine does. 50,000 iterations with random searching is not an option for a human. The number of empirical episodes can be reduced dramatically with a numerical convergence form of gradient descent (see *Chapter 3, Machine Intelligence – Evaluation Functions and Numerical Convergence*).

Humans need to be more intuitive, make a few decisions, and see what happens, because humans cannot try thousands of ways of doing something. Reinforcement learning marks a new era for human thinking by surpassing

human reasoning power in strategic fields.

On the other hand, reinforcement learning requires mathematical models to function. Humans excel in mathematical abstraction, providing powerful intellectual fuel to those powerful machines.

The boundaries between humans and machines have changed. Humans' ability to build mathematical models and ever-growing cloud platforms will serve online machine learning services.

Finding out how to use the outputs of the reinforcement learning program we just studied shows how a human will always remain at the center of AI.

How to use the outputs

The reinforcement program we studied contains no trace of a specific field, as in traditional software. The program contains the Bellman equation with stochastic (random) choices based on the reward matrix. The goal is to find a route to **C** (line 3, column 3) that has an attractive reward (`100`):

```
# Markov Decision Process (MDP) – The Bellman equations ada
pted to
# Reinforcement Learning with the Q action-value(reward) ma
trix
import numpy as ql
# R is The Reward Matrix for each state
R = ql.matrix([ [0,0,0,0,1,0],
                [0,0,0,1,0,1],
                [0,0,100,1,0,0],
                [0,1,1,0,1,0],
                [1,0,0,1,0,0],
                [0,1,0,0,0,0] ])
```

That reward matrix goes through the Bellman equation and produces a result in Python:

```
Q :
[[ 0. 0. 0. 0. 258.44 0. ]
 [ 0. 0. 0. 321.8 0. 207.752]
 [ 0. 0. 500. 321.8 0. 0. ]
```

```
 [ 0. 258.44 401. 0. 258.44 0. ]
 [ 207.752 0. 0. 321.8 0. 0. ]
 [ 0. 258.44 0. 0. 0. 0. ]]
Normed Q :
[[ 0. 0. 0. 0. 51.688 0. ]
 [ 0. 0. 0. 64.36 0. 41.5504]
 [ 0. 0. 100. 64.36 0. 0. ]
 [ 0. 51.688 80.2 0. 51.688 0. ]
 [ 41.5504 0. 0. 64.36 0. 0. ]
 [ 0. 51.688 0. 0. 0. 0. ]]
```

The result contains the values of each state produced by the reinforced learning process, and also a normed `Q` (the highest value divided by other values).

As Python geeks, we are overjoyed! We made something that is rather difficult work, namely, reinforcement learning. As mathematical amateurs, we are elated. We know what MDP and the Bellman equation mean.

However, as natural language thinkers, we have made little progress. No customer or user can read that data and make sense of it. Furthermore, we cannot explain how we implemented an intelligent version of their job in the machine. We didn't.

We hardly dare say that reinforcement learning can beat anybody in the company, making random choices 50,000 times until the right answer came up.

Furthermore, we got the program to work, but hardly know what to do with the result ourselves. The consultant on the project cannot help because of the matrix format of the solution.

Being an adaptive thinker means knowing how to be good in all steps of a project. To solve this new problem, let's go back to step 1 with the result. Going back to step 1 means that if you have problems either with the results themselves or understanding them, it is necessary to go back to the SME level, the real-life situation, and see what is going wrong.

By formatting the result in Python, a graphics tool, or a spreadsheet, the result can be displayed as follows:

	A	**B**	**C**	**D**	**E**	**F**

A	-	-	-	-	258.44	-
B	-	-	-	321.8	-	207.752
C	-	-	500	321.8	-	-
D	-	258.44	401.	-	258.44	-
E	207.752	-	-	321.8	-	-
F	-	258.44	-	-	-	-

Now, we can start reading the solution:

- Choose a starting state. Take **F**, for example.
- The **F** line represents the state. Since the maximum value is 258.44 in the **B** column, we go to state **B**, the second line.
- The maximum value in state **B** in the second line leads us to the **D** state in the fourth column.
- The highest maximum of the **D** state (fourth line) leads us to the **C** state.

Note that if you start at the **C** state and decide not to stay at **C**, the **D** state becomes the maximum value, which will lead you back to **C**. However, the MDP will never do this naturally. You will have to force the system to do it.

You have now obtained a sequence: **F->B->D->C**. By choosing other points of departure, you can obtain other sequences by simply sorting the table.

A useful way of putting it remains the normalized version in percentages, as shown in the following table:

	A	**B**	**C**	**D**	**E**	**F**
A	-	-	-	-	51.68%	-
B	-	-	-	64.36%	-	41.55%

C	-	-	100%	64.36%	-	-
D	-	51.68%	80.2%	-	51.68%	-
E	41.55%	-	-	64.36%	-	-
F	-	51.68%	-	-	-	-

Now comes the very tricky part. We started the chapter with a trip on the road. But I made no mention of it in the results analysis.

An important property of reinforcement learning comes from the fact that we are working with a mathematical model that can be applied to anything. No human rules are needed. We can use this program for many other subjects without writing thousands of lines of code.

Possible use cases

There are many cases to which we could adapt our reinforcement learning model without having to change any of its details.

Case 1: optimizing a delivery for a driver, human or not

This model was described in this chapter.

Case 2: optimizing warehouse flows

The same reward matrix can apply to go from point **F** to **C** in a warehouse, as shown in the following diagram:

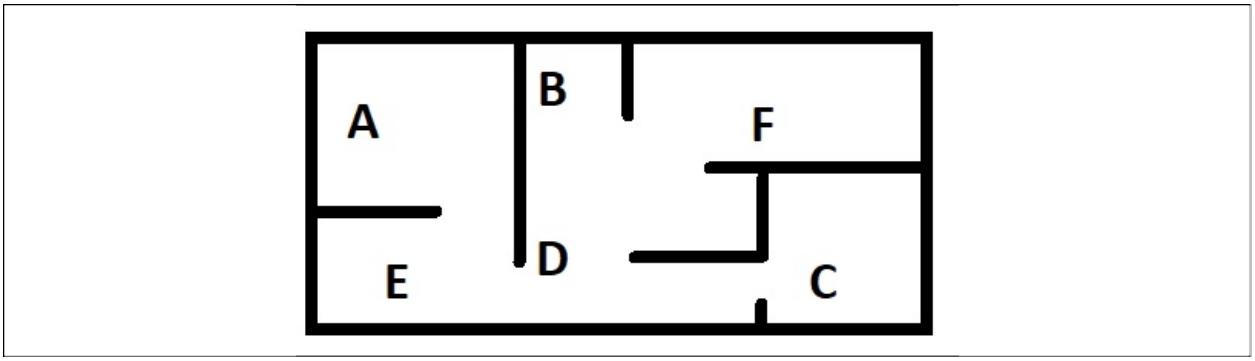


Figure 1.3: A diagram illustrating a warehouse flow problem

In this warehouse, the **F**->**B**->**D**->**C** sequence makes visual sense. If somebody goes from point **F** to **C**, then this physical path makes sense without going through walls.

It can be used for a video game, a factory, or any form of layout.

Case 3: automated planning and scheduling (APS)

By converting the system into a scheduling vector, the whole scenery changes. We have left the more comfortable world of physical processing of letters, faces, and trips. Though fantastic, those applications are social media's tip of the iceberg. The real challenge of AI begins in the abstract universe of human thinking.

Every single company, person, or system requires automatic planning and scheduling (see *Chapter 12*, *AI and the Internet of Things (IoT)*). The six **A** to **F** steps in the example of this chapter could well be six tasks to perform in a given unknown order represented by the following vector *x*:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}$$

The reward matrix then reflects the weights of constraints of the tasks of vector *x* to perform. For example, in a factory, you cannot assemble the parts of a product before manufacturing them.

In this case, the sequence obtained represents the schedule of the manufacturing process.

Cases 4 and more: your imagination

By using physical layouts or abstract decision-making vectors, matrices, and tensors, you can build a world of solutions in a mathematical reinforcement learning model. Naturally, the following chapters will enhance your toolbox with many other concepts.

Before moving on, you might want to imagine some situations in which you could use the A to F letters to express some kind of path.

To help you with these mind experiment simulations, open `mdp02.py` and go to line 97, which starts with the following code that enables a simulation tool. `nextc` and `nextci` are simply variables to remember where the path begins and will end. They are set to `-1` so as to avoid 0, which is a location.

The primary goal is to focus on the expression "concept code." The locations have become any concept you wish. A could be your bedroom, and C your kitchen. The path would go from where you wake up to where you have breakfast. A could be an idea you have, and F the end of a thinking process. The path would go from A (How can I hang this picture on the wall?) to E (I need to drill a hole) and, after a few phases, to F (I hung the picture on the wall). You can imagine thousands of paths like this as long as you define the reward matrix, the "concept code," and a starting point:

```
"""# Improving the program by introducing a decision-making
 process"""
nextc=-1
nextci=-1
conceptcode=["A","B","C","D","E","F"]
```

This code takes the result of the calculation, labels the result matrix, and accepts an input as shown in the following code snippet:

```
origin=int(input(
    "index number origin(A=0,B=1,C=2,D=3,E=4,F=5): "))
```

The input only accepts the label numerical code: `A=0`, `B=1` … `F=5`. The function then runs a classical calculation on the results to find the best path. Let's takes an example.

When you are prompted to enter a starting point, enter `5`, for example, as follows:

```
index number origin(A=0,B=1,C=2,D=3,E=4,F=5): 5
```

The program will then produce the optimal path based on the output of the MDP process, as shown in the following output:

```
Concept Path
-> F
-> B
-> D
-> C
```

Try multiple scenarios and possibilities. Imagine what you could apply this to:

- An e-commerce website flow (visit, cart, checkout, purchase) imagining that a user visits the site and then resumes a session at a later time. You can use the same reward matrix and "concept code" explored in this chapter. For example, a visitor visits a web page at 10 a.m., starting at point A of your website. Satisfied with a product, the visitor puts the product in a cart, which is point E of your website. Then, the visitor leaves the site before going to the purchase page, which is C. D is the critical point. Why didn't the visitor purchase the product? What's going on?

  You can decide to have an automatic email sent after 24 hours saying: "There is a 10% discount on all purchases during the next 48 hours." This way, you will target all the visitors stuck at D and push them toward C.

- A sequence of possible words in a sentence (subject, verb, object). Predicting letters and words was one of Andrey Markov's first applications 100+ years ago! You can imagine that B is the letter "a" of the alphabet. If D is "t," it is much more probable than F if F is "o," which is less probable in the English language. If an MDP reward matrix is built such as B leads to D or F, B can thus either go to D or to F. There are thus two possibilities, D

or F. Andrey Markov would suppose, for example, that B is a variable that represents the letter "a," D is a variable that represents the letter "t" and F is a variable that represents the letter "o." After studying the structure of a language closely, he would find that the letter "a" would more likely be followed by "t" than by "o" in the English language. If one observes the English language, it is more likely to find an "a-t" sequence than an "a-o" sequence. In a Markov decision process, a higher probability will be awarded to the "a-t" sequence and a lower one to "a-o." If one goes back to the variables, the B-D sequence will come out as more probable than the B-F sequence.

- And anything you can find that fits the model that works is great!

Machine learning versus traditional applications

Reinforcement learning based on stochastic (random) processes will evolve beyond traditional approaches. In the past, we would sit down and listen to future users to understand their way of thinking.

We would then go back to our keyboard and try to imitate the human way of thinking. Those days are over. We need proper datasets and ML/DL equations to move forward. Applied mathematics has taken reinforcement learning to the next level. In my opinion, traditional software will soon be in the museum of computer science. The complexity of the huge volumes of data we are facing will require AI at some point.

An artificial adaptive thinker sees the world through applied mathematics translated into machine representations.

Use the Python source code example provided in this chapter in different ways. Run it and try to change some parameters to see what happens. Play around with the number of iterations as well. Lower the number from 50,000 down to where you find it fits best. Change the reward matrix a little to see what happens. Design your reward matrix trajectory. This can be an itinerary or decision-making process.

Summary

Presently, AI is predominantly a branch of applied mathematics, not of neurosciences. You must master the basics of linear algebra and probabilities. That's a difficult task for a developer used to intuitive creativity. With that knowledge, you will see that humans cannot rival machines that have CPU and mathematical functions. You will also understand that machines, contrary to the hype around you, don't have emotions; although we can represent them to a scary point in chatbots (see *Chapter 16, Improving the Emotional Intelligence Deficiencies of Chatbots*).

A multi-dimensional approach is a prerequisite in an AI/ML/DL project. First, talk and write about the project, then make a mathematical representation, and finally go for software production (setting up an existing platform or writing code). In real life, AI solutions do not just grow spontaneously in companies as some hype would have us believe. You need to talk to the teams and work with them. That part is the real fulfilling aspect of a project—imagining it first and then implementing it with a group of real-life people.

MDP, a stochastic random action-reward (value) system enhanced by the Bellman equation, will provide effective solutions to many AI problems. These mathematical tools fit perfectly in corporate environments.

Reinforcement learning using the Q action-value function is memoryless (no past) and unsupervised (the data is not labeled or classified). MDP provides endless avenues to solve real-life problems without spending hours trying to invent rules to make a system work.

Now that you are at the heart of Google's DeepMind approach, it is time to go to *Chapter 2*, *Building a Reward Matrix – Designing Your Datasets*, and discover how to create the reward matrix in the first place through explanations and source code.

Questions

The answers to the questions are in *Appendix B*, with further explanations:

1. Is reinforcement learning memoryless? (Yes | No)
2. Does reinforcement learning use stochastic (random) functions? (Yes | No)
3. Is MDP based on a rule base? (Yes | No)
4. Is the Q function based on the MDP? (Yes | No)
5. Is mathematics essential to AI? (Yes | No)
6. Can the Bellman-MDP process in this chapter apply to many problems? (Yes | No)
7. Is it impossible for a machine learning program to create another program by itself? (Yes | No)
8. Is a consultant required to enter business rules in a reinforcement learning program? (Yes | No)
9. Is reinforcement learning supervised or unsupervised? (Supervised | Unsupervised)
10. Can Q-learning run without a reward matrix? (Yes | No)

Further reading

- Andrey Markov: https://www.britannica.com/biography/Andrey-Andreyevich-Markov
- The Markov process: https://www.britannica.com/science/Markov-process

2

Building a Reward Matrix – Designing Your Datasets

Experimenting and implementation comprise the two main approaches of artificial intelligence. Experimenting largely entails trying ready-to-use datasets and black box, ready-to-use Python examples. Implementation involves preparing a dataset, developing preprocessing algorithms, and then choosing a model, the proper parameters, and hyperparameters.

Implementation usually involves white box work that entails knowing exactly how an algorithm works and even being able to modify it.

In *Chapter 1*, *Getting Started with Next-Generation Artifcial Intelligence through Reinforcement Learning*, the MDP-driven Bellman equation relied on a reward matrix. In this chapter, we will get our hands dirty in a white box process to create that reward matrix.

An MDP process cannot run without a reward matrix. The reward matrix determines whether it is possible to go from one cell to another, from A to B. It is like a map of a city that tells you if you are allowed to take a street or if it is a one-way street, for example. It can also set a goal, such as a place that you would like to visit in a city, for example.

To achieve the goal of designing a reward matrix, the raw data provided by other systems, software, and sensors needs to go through **preprocessing**. A machine learning program will not provide efficient results if the data has not gone through a **standardization** process.

The reward matrix, *R*, will be built using a McCulloch-Pitts neuron in TensorFlow. Warehouse management has grown exponentially as e-commerce has taken over many marketing segments. This chapter introduces automated guided vehicles (AGVs), the equivalent of an SDC in a warehouse to store and

retrieve products.

The challenge in this chapter will be to understand the preprocessing phase in detail. The quality of the processed dataset will influence directly the accuracy of any machine learning algorithm.

This chapter covers the following topics:

- The McCulloch-Pitts neuron will take the raw data and transform it
- Logistic classifiers will begin the neural network process
- The logistic sigmoid will squash the values
- The softmax function will normalize the values
- The one-hot function will choose the target for the reward matrix
- An example of AGVs in a warehouse

The topics form a list of tools that, in turn, form a pipeline that will take raw data and transform it into a reward matrix—an MDP.

Designing datasets – where the dream stops and the hard work begins

As in the previous chapter, bear in mind that a real-life project goes through a three-dimensional method in some form or other. First, it's important to think and talk about the problem in need of solving without jumping onto a laptop. Once that is done, bear in mind that the foundation of machine learning and deep learning relies on mathematics. Finally, once the problem has been discussed and mathematically represented, it is time to develop the solution.

> First, think of a problem in **natural language**. Then, make a **mathematical description** of a problem. Only then should you begin the **software implementation**.

Designing datasets

The reinforcement learning program described in the first chapter can solve a variety of problems involving unlabeled classification in an unsupervised decision-making process. The Q function can be applied to drone, truck, or car deliveries. It can also be applied to decision making in games or real life.

However, in a real-life case study problem (such as defining the reward matrix in a warehouse for the AGV, for example), the difficulty will be to produce an efficient matrix using the proper **features**.

For example, an AGV requires information coming from different sources: daily forecasts and real-time warehouse flows.

The warehouse manages thousands of locations and hundreds of thousands of inputs and outputs. Trying to fit too many features into the model would be counterproductive. Removing both features and worthless data requires careful consideration.

A simple neuron can provide an efficient way to attain the **standardization** of the input data.

> Machine learning and deep learning are frequently used to preprocess input data for standardization purposes, normalization, and feature reduction.

Using the McCulloch-Pitts neuron

To create the reward matrix, *R*, a robust model for processing the inputs of the huge volumes in a warehouse must be reduced to a limited number of features.

In one model, for example, the thousands to hundreds of thousands of inputs can be described as follows:

- Forecast product arrivals with a low priority weight: $w_1 = 10$
- Confirmed arrivals with a high priority weight: $w_2 = 70$
- Unplanned arrivals decided by the sales department: $w_3 = 75$

- Forecasts with a high priority weight: $w_4 = 60$
- Confirmed arrivals that have a low turnover and so have a low weight: $w_5 = 20$

The weights have been provided as constants. A McCulloch-Pitts neuron does not modify weights. A perceptron neuron does as we will see beginning with *Chapter 8, Solving the XOR Problem with a Feedforward Neural Network*. Experience shows that modifying weights is not always necessary.

These weights form a vector, as shown here:

$$x = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \end{bmatrix} = \begin{bmatrix} 10 \\ 70 \\ 75 \\ 60 \\ 20 \end{bmatrix}$$

Each element of the vector represents the weight of a feature of a product stored in optimal locations. The ultimate phase of this process will produce a reward matrix, *R*, for an MDP to optimize itineraries between warehouse locations.

Let's focus on our neuron. These weights, used through a system such as this one, can attain up to more than 50 weights and parameters per neuron. In this example, 5 weights are implemented. However, in real-life case, many other parameters come into consideration, such as unconfirmed arrivals, unconfirmed arrivals with a high priority, confirmed arrivals with a very low priority, arrivals from locations that probably do not meet security standards, arrivals with products that are potentially dangerous and require special care, and more. At that point, humans and even classical software cannot face such a variety of parameters.

The reward matrix will be size 6×6. It contains six locations, A to F. In this example, the six locations, `l1` to `l6`, are warehouse storage and retrieval locations.

A 6×6 reward matrix represents the target of the McCulloch-Pitts layer implemented for the six locations.

When experimenting, the reward matrix, *R*, can be invented for testing purposes.

In real-life implementations, you will have to find a way to build datasets from scratch. The reward matrix becomes the output of the preprocessing phase. The following source code shows the input of the reinforcement learning program used in the first chapter. The goal of this chapter describes how to produce the following reward matrix that we will be building in the next sections.

```
# R is The Reward Matrix for each location in a warehouse (
or any other problem)
R = ql.matrix([ [0,0,0,0,1,0],
                [0,0,0,1,0,1],
                [0,0,100,1,0,0],
                [0,1,1,0,1,0],
                [1,0,0,1,0,0],
                [0,1,0,0,0,0] ])
```

For the warehouse example that we are using as for any other domain, the McCulloch-Pitts neuron sums up the weights of the input vector described previously to fill in the reward matrix.

Each location will require its neuron, with its weights.

INPUTS -> WEIGHTS -> BIAS -> VALUES

- Inputs are the flows in a warehouse or any form of data.
- Weights will be defined in this model.
- Bias is for stabilizing the weights. Bias does exactly what it means. It will tilt weights. It is very useful as a referee that will keep the weights on the right track.
- Values will be the output.

> There are as many ways as you can imagine to create reward matrices. This chapter describes one way of doing it that works.

The McCulloch-Pitts neuron

The McCulloch-Pitts neuron dates back to 1943. It contains inputs, weights, and an activation function. Part of the preprocessing phase consists of selecting the

right model. The McCulloch-Pitts neuron can represent a given location efficiently.

The following diagram shows the McCulloch-Pitts neuron model:

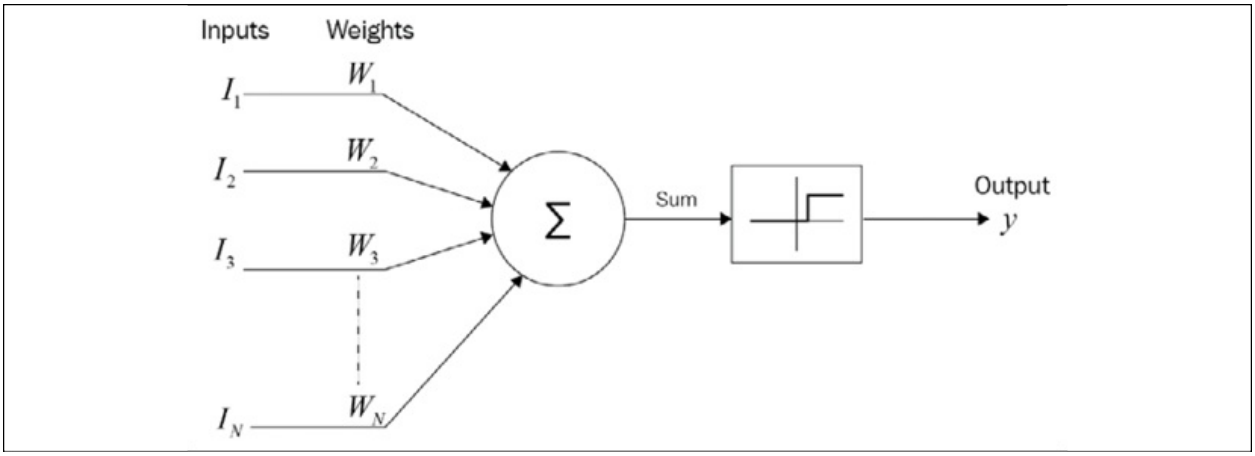


Figure 2.1: The McCulloch-Pitts neuron model

This model contains several input *x* weights that are summed to either reach a threshold that will lead, once transformed, to the output, *y* = 0, or 1. In this model, *y* will be calculated in a more complex way.

`MCP.py` written with TensorFlow 2 will be used to illustrate the neuron.

In the following source code, the TensorFlow variables will contain the input values (`x`), the weights (`W`), and the bias (`b`). Variables represent the structure of your graph:

```
# The variables
x = tf.Variable([[0.0,0.0,0.0,0.0,0.0]], dtype = tf.float32
)
W = tf.Variable([[0.0],[0.0],[0.0],[0.0],[0.0]], dtype =
    tf.float32)
b = tf.Variable([[0.0]])
```

In the original McCulloch-Pitts artificial neuron, the inputs (*x*) were multiplied by the following weights:

$$w_1 x_1 + \cdots + w_n x_n = \sum_{j=1}^{n} w_j x_j$$

The mathematical function becomes a function with the neuron code triggering a logistic activation function (sigmoid), which will be explained in the second part of the chapter. Bias (`b`) has been added, which makes this neuron format useful even today, shown as follows:

```
# The Neuron
def neuron(x, W, b):
    y1=np.multiply(x,W)+b
    y1=np.sum(y1)
    y = 1 / (1 + np.exp(-y1))
    return y
```

Before starting a session, the McCulloch-Pitts neuron (1943) needs an operator to set its weights. That is the main difference between the McCulloch-Pitts neuron and the perceptron (1957), which is the model for modern deep learning neurons. The perceptron optimizes its weights through optimizing processes. *Chapter 8*, *Solving the XOR Problem with a Feedforward Neural Network*, describes why a modern perceptron was required.

The weights are now provided, and so are the quantities for each input value, which are stored in the *x* vector at l_1, one of the six locations of this warehouse example:

$$x = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \end{bmatrix} = \begin{bmatrix} 10 \\ 70 \\ 75 \\ 60 \\ 20 \end{bmatrix}$$

The weight values will be divided by 100, to represent percentages in terms of 0 to 1 values of warehouse flows in a given location. The following code deals with the choice of *one* location, l_1 **only**, its values, and parameters:

```
# The data
x_1 = [[10, 2, 1., 6., 2.]]
```

```
w_t = [[.1, .7, .75, .60, .20]]
b_1 = [1.0]
```

The neuron function is called, and the weights (`w_t`) and the quantities (`x_1`) of the warehouse flow are entered. Bias is set to `1` in this model. No session needs to be initialized; the neuron function is called:

```
# Computing the value of the neuron
value=neuron(x_1,w_t,b_1)
```

The neuron function `neuron` will calculate the value of the neuron. The program returns the following value:

```
value for threshold calculation:0.99999
```

This value represents the activity of location l_1 at a given date and a given time. This example represents only one of the six locations to compute. For this location, the higher the value, the closer to 1, the higher the probable saturation rate of this area. This means there is little space left to store products at that location. That is why the reinforcement learning program for a warehouse is looking for the **least loaded** area for a given product in this model.

Each location has a probable **availability**:

A = Availability = 1 – load

The probability of a load of a given storage point lies between 0 and 1.

High values of availability will be close to 1, and low probabilities will be close to 0, as shown in the following example:

```
>>> print("Availability of location x:{0:.5f}".format(
...       round(availability,5)))
Availability of location x:0.00001
```

For example, the load of l_1 has a probable rounded load of 0.99, and its probable *availability* is 0.002 maximum. The goal of the AGV is to search and find the closest and most available location to optimize its trajectories. l_1 is not a good candidate at that day and time. **Load** is a keyword in production or service

activities. The less available resources have the highest load rate.

When all six locations' availabilities have been calculated by the McCulloch-Pitts neuron—each with its respective quantity inputs, weights, and bias—a location vector of the results of this system will be produced. Then, the program needs to be implemented to run all six locations and not just one location through a recursive use of the one neuron model:

$$A(L) = \{a(l_1),a(l_2),a(l_3),a(l_4),a(l_5),a(l_6)\}$$

The availability, 1 – *output value of the neuron*, constitutes a six-line vector. The following vector, l_v, will be obtained by running the previous sample code on **all** six locations.

$$l_v = \begin{bmatrix} 0.0002 \\ 0.2 \\ 0.9 \\ 0.0001 \\ 0.4 \\ 0.6 \end{bmatrix}$$

As shown in the preceding formula, l_v is the vector containing the value of each location for a given AGV to choose from. The values in the vector represent availability. 0.0002 means little availability; 0.9 means high availability. With this choice, the MDP reinforcement learning program will optimize the AGV's trajectory to get to this specific warehouse location.

The l_v is the result of the weighing function of six potential locations for the AGV. It is also a vector of transformed inputs.

The Python-TensorFlow architecture

Implementation of the McCulloch-Pitts neuron can best be viewed as shown in the following graph:

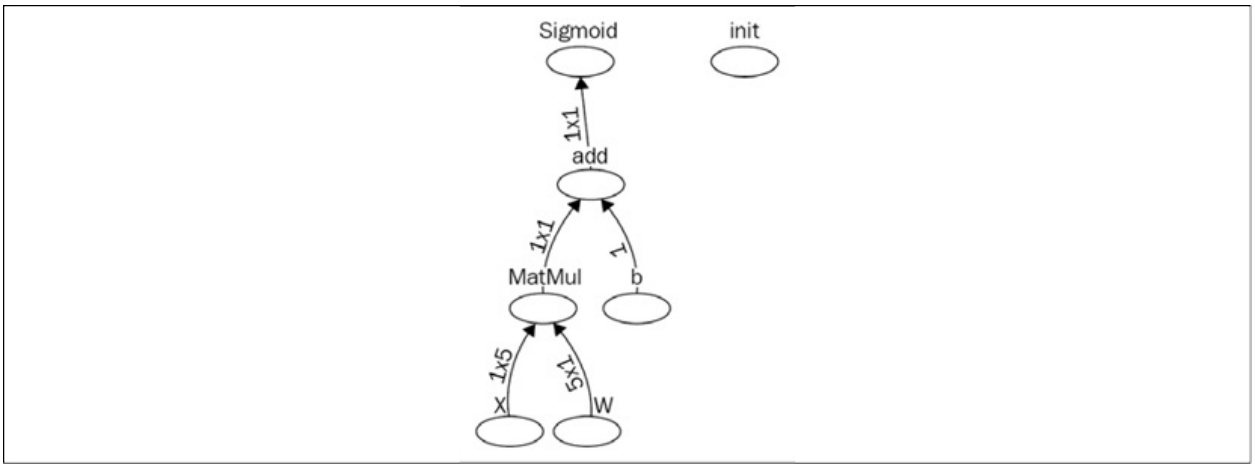


Figure 2.2: Implementation of the McCulloch-Pitts neuron

A data flow graph will also help optimize a program when things go wrong as in classical computing.

Logistic activation functions and classifiers

Now that the value of each location of $L = \{l_1, l_2, l_3, l_4, l_5, l_6\}$ contains its availability in a vector, the locations can be sorted from the most available to the least available location. From there, the reward matrix, *R*, for the MDP process described in *Chapter 1*, *Getting Started with Next-Generation Artifcial Intelligence through Reinforcement Learning*, can be built.

Overall architecture

At this point, the overall architecture contains two main components:

1. **Chapter 1**: A reinforcement learning program based on the value-action Q function using a reward matrix that will be finalized in this chapter. The reward matrix was provided in the first chapter as an experiment, but in the implementation phase, you'll often have to build it from scratch.
   It sometimes takes weeks to produce a good reward matrix.

2. **Chapter 2**: Designing a set of 6×1 neurons that represents the flow of products at a given time at six locations. The output is the availability probability from 0 to 1. The highest value indicates the highest availability. The lowest value indicates the lowest availability.

At this point, there is some real-life information we can draw from these two main functions through an example:

- An AGV is automatically moving in a warehouse and is waiting to receive its next location to use an MDP, to calculate the optimal trajectory of its mission.
- An AGV is using a reward matrix, *R*, that was given during the experimental phase but needed to be designed during the implementation process.
- A system of six neurons, one per location, weighing the real quantities and probable quantities to give an availability vector, l_v, has been calculated. It is almost ready to provide the necessary reward matrix for the AGV.

To calculate the input values of the reward matrix in this reinforcement learning warehouse model, a bridge function between l_v and the reward matrix, *R*, is missing.

That bridge function is a logistic classifier based on the outputs of the *n* neurons that all perform the same tasks independently or recursively with one neuron.

At this point, the system:

- Took corporate data
- Used *n* neurons calculated with weights
- Applied an activation function

The activation function in this model requires a logistic classifier, a commonly used one.

Logistic classifier

The logistic classifier will be applied to l_v (the six location values) to find the best location for the AGV. This method can be applied to any other domain. It is based on the output of the six neurons as follows:

$$input \times weight + bias$$

What are logistic functions? The goal of a logistic classifier is to produce a probability distribution from 0 to 1 for each value of the output vector. As you have seen so far, artificial intelligence applications use applied mathematics with probable values, not raw outputs.

The main reason is that machine learning/deep learning works best with standardization and normalization for workable homogeneous data distributions. Otherwise, the algorithms will often produce underfitted or overfitted results.

In the warehouse model, for example, the AGV needs to choose the best, most probable location, l_i. Even in a well-organized corporate warehouse, many uncertainties (late arrivals, product defects, or some unplanned problems) reduce the probability of a choice. A probability represents a value between 0 (low probability) and 1 (high probability). Logistic functions provide the tools to convert all numbers into probabilities between 0 and 1 to *normalize* data.

Logistic function

The logistic sigmoid provides one of the best ways to normalize the weight of a given output. The activation function of the neuron will be the logistic sigmoid. The threshold is usually a value above which the neuron has a $y = 1$ value; or else it has a $y = 0$ value. In this model, the minimum value will be 0.

The logistic function is represented as follows:

$$\frac{1}{1 + e^{-x}}$$

- e represents Euler's number, or 2.71828, the natural logarithm.
- x is the value to be calculated. In this case, s is the result of the logistic sigmoid function.

The code has been rearranged in the following example to show the reasoning process that produces the output, `y`, of the neuron:

```
y1=np.multiply(x,W)+b
y1=np.sum(y1)
y = 1 / (1 + np.exp(-y1)) #logistic Sigmoid
```

Thanks to the logistic sigmoid function, the value for the first location in the model comes out squashed between 0 and 1 as 0.99, indicating a high probability that this location will be full.

To calculate the availability of the location once the 0.99 value has been taken into account, we subtract the load from the total availability, which is 1, as follows:

Availability = 1 – probability of being full (value)

Or

availability = 1 – value

As seen previously, once all locations are calculated in this manner, a final availability vector, l_v, is obtained.

$$l_v = \begin{bmatrix} 0.0002 \\ 0.2 \\ 0.9 \\ 0.0001 \\ 0.4 \\ 0.6 \end{bmatrix}$$

When analyzing l_v, a problem has stopped the process. Individually, each line appears to be fine. By applying the logistic sigmoid to each output weight and subtracting it from 1, each location displays a probable availability between 0 and 1. However, the sum of the lines in l_v exceeds 1. That is not possible. A probability cannot exceed 1. The program needs to fix that.

Each line produces a [0, 1] solution, which fits the prerequisite of being a valid probability.

In this case, the vector l_v contains more than one value and becomes a probability distribution. The sum of l_v cannot exceed 1 and needs to be normalized.

The *softmax* function provides an excellent method to normalize l_v. Softmax is widely used in machine learning and deep learning.

Bear in mind that *mathematical tools are not rules*. You can adapt them to your problem as much as you wish as long as your solution works.

Softmax

The softmax function appears in many artificial intelligence models to normalize data. Softmax can be used for classification purposes and regression. In our example, we will use it to find an optimized goal for an MDP.

In the case of the warehouse example, an AGV needs to make a probable choice between six locations in the l_v vector. However, the total of the l_v values exceeds 1. l_v requires normalization of the softmax function, *S*. In the source code, the l_v vector will be named `y`.

$$S(y_i) = \frac{e^{y_i}}{\sum_{j=1}^{n} e^{y_i}}$$

The following code used is `SOFTMAX.py`.

1. `y` represents the l_v vector:

```
# y is the vector of the scores of the lv vector in the
 warehouse example:
y = [0.0002, 0.2, 0.9,0.0001,0.4,0.6]
```

2. e^{y_i} is the *exp*(*i*) result of each value in `y` (l_v in the warehouse example), as follows:

```
y_exp = [math.exp(i) for i in y]
```

3. $\sum_{j=1}^{n} e^{y_i}$ is the sum of e^{y_i} as shown in the following code:

```
sum_exp_yi = sum(y_exp)
```

Now, each value of the vector is normalized by applying the following function:

```
softmax = [round(i / sum_exp_yi, 3) for i in y_exp]
```

$$l_v = \begin{bmatrix} 0.0002 \\ 0.2 \\ 0.9 \\ 0.0001 \\ 0.4 \\ 0.6 \end{bmatrix} \rightarrow \text{softmax}(l_v) \rightarrow \begin{bmatrix} 0.111 \\ 0.135 \\ 0.273 \\ 0.111 \\ 0.165 \\ 0.202 \end{bmatrix}$$

softmax(l_v) provides a normalized vector with a sum equal to 1, as shown in this compressed version of the code. The vector obtained is often described as containing logits.

The following code shows one version of a softmax function:

```
def softmax(x):
    return np.exp(x) / np.sum(np.exp(x), axis=0)
```

l_v is now normalized by softmax(l_v) as follows.

$$l_v = \begin{bmatrix} 0.0002 \\ 0.2 \\ 0.9 \\ 0.0001 \\ 0.4 \\ 0.6 \end{bmatrix} \rightarrow \text{softmax}(l_v) \rightarrow \begin{bmatrix} 0.111 \\ 0.135 \\ 0.273 \\ 0.111 \\ 0.165 \\ 0.202 \end{bmatrix}$$

The last part of the softmax function requires softmax(l_v) to be rounded to 0 or 1. The higher the value in softmax(l_v), the more probable it will be. In clear-cut transformations, the highest value will be close to 1, and the others will be closer to 0. In a decision-making process, the highest value needs to be established as follows:

```
print("7C.
Finding the highest value in the normalized y vector : ",oh
```

```
ot)
```

The output value is `0.273` and has been chosen as the most probable location. It is then set to 1, and the other, lower values are set to 0. This is called a one-hot function. This one-hot function is extremely helpful for encoding the data provided. The vector obtained can now be applied to the reward matrix. The value 1 probability will become 100 in the *R* reward matrix, as follows:

$$l_v = \begin{bmatrix} 0.0002 \\ 0.2 \\ 0.9 \\ 0.0001 \\ 0.4 \\ 0.6 \end{bmatrix} \rightarrow \text{softmax}(l_v) \rightarrow \begin{bmatrix} 0.111 \\ 0.135 \\ 0.273 \\ 0.111 \\ 0.165 \\ 0.202 \end{bmatrix} \rightarrow \text{one-hot}(l_v) \rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \rightarrow R \rightarrow \begin{bmatrix} 0 \\ 0 \\ 100 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The softmax function is now complete. Location l_3 or **C** is the best solution for the AGV. The probability value is multiplied by 100, and the reward matrix, *R*, can now receive the input.

> Before continuing, take some time to play around with the values in the source code and run it to become familiar with the softmax function.

We now have the data for the reward matrix. The best way to understand the mathematical aspect of the project is to draw the result on a piece of paper using the actual warehouse layout from locations **A** to **F**.

```
Locations={l1-A, l2-B, l3-C, l4-D, l5-E, l6-F}
```

Line **C** of the reward matrix ={0, 0, 100, 0, 0, 0}, where **C** (the third value) is now the target for the self-driving vehicle, in this case, an AGV in a warehouse.

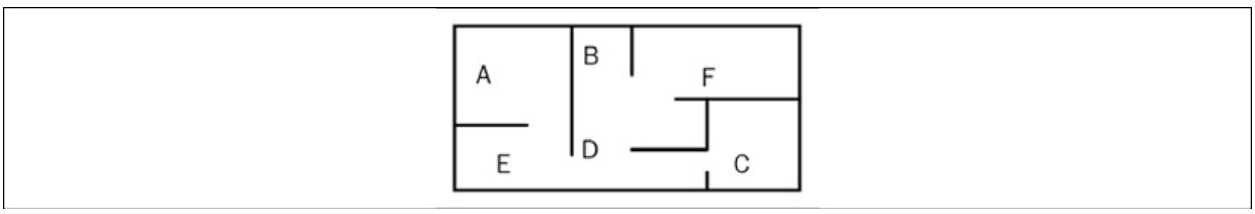


Figure 2.3: Illustration of a warehouse transport problem

We obtain the following reward matrix, *R*, described in *Chapter 1*, *Getting Started with Next-Generation Artificial Intelligence through Reinforcement Learning*:

State/values	A	B	C	D	E	F
A	-	-	-	-	1	-
B	-	-	-	1	-	1
C	-	-	100	1	-	-
D	-	1	1	-	1	-
E	1	-	-	1	-	-
F	-	1	-	-	-	-

This reward matrix is exactly the one used in the Python reinforcement learning program using the Q function from *Chapter 1*. The output of this chapter is thus the input of the *R* matrix. The 0 values are there for the agent to avoid those values. The 1 values indicate the reachable cells. The 100 in the C×C cell is the result of the softmax output. This program is designed to stay close to probability standards with positive values, as shown in the following *R* matrix taken from the `mdp01.py` of *Chapter 1*:

```
R = ql.matrix([ [0,0,0,0,1,0],
                [0,0,0,1,0,1],
                [0,0,100,1,0,0],
                [0,1,1,0,1,0],
                [1,0,0,1,0,0],
                [0,1,0,0,0,0] ])
```

At this point:

- The output of the functions in this chapter generated a reward matrix, *R*, which is the input of the MDP described in *Chapter 1*, *Getting Started with*

Next-Generation Artificial Intelligence through Reinforcement Learning.

- The MDP process was set to run for 50,000 episodes in *Chapter 1*.
- The output of the MDP has multiple uses, as we saw in this chapter and *Chapter 1*.

The building blocks are in place to begin evaluating the execution and performances of the reinforcement learning program, as we will see in *Chapter 3, Machine Intelligence – Evaluation Functions and Numerical Convergence*.

Summary

Using a McCulloch-Pitts neuron with a logistic activation function in a one-layer network to build a reward matrix for reinforcement learning shows one way to preprocess a dataset.

Processing real-life data often requires a generalization of a logistic sigmoid function through a softmax function, and a one-hot function applied to logits to encode the data.

Machine learning functions are tools that must be understood to be able to use all or parts of them to solve a problem. With this practical approach to artificial intelligence, a whole world of projects awaits you.

This neuronal approach is the parent of the multilayer perceptron that will be introduced starting in *Chapter 8, Solving the XOR Problem with a Feedforward Neural Network*.

This chapter went from an experimental black box machine learning and deep learning to white box implementation. Implementation requires a full understanding of machine learning algorithms that often require fine-tuning.

However, artificial intelligence goes beyond understanding machine learning algorithms. Machine learning or deep learning require evaluation functions. Performance or results cannot be validated without evaluation functions, as explained in *Chapter 3, Machine Intelligence – Evaluation Functions and Numerical Convergence*.

In the next chapter, the evaluation process of machine intelligence will be illustrated by examples that show the limits of human intelligence and the rise of machine power.

Questions

1. Raw data can be the input to a neuron and transformed with weights. (Yes | No)
2. Does a neuron require a threshold? (Yes | No)
3. A logistic sigmoid activation function makes the sum of the weights larger. (Yes | No)
4. A McCulloch-Pitts neuron sums the weights of its inputs. (Yes | No)
5. A logistic sigmoid function is a log10 operation. (Yes | No)
6. A logistic softmax is not necessary if a logistic sigmoid function is applied to a vector. (Yes | No)
7. A probability is a value between –1 and 1. (Yes | No)

Further reading

- The original McCulloch-Pitts neuron 1943 paper: http://www.cse.chalmers.se/~coquand/AUTOMATA/mcp.pdf
- TensorFlow variables: https://www.tensorflow.org/beta/guide/variables

3

Machine Intelligence – Evaluation Functions and Numerical Convergence

Two issues appear when a reward matrix (*R*)-driven MDP produces results. These issues can be summed up in two principles.

Principle 1: AI algorithms often surpass humans in classification, prediction, and decision-making areas.

The key executive function of human intelligence, decision-making, relies on the ability to evaluate a situation. No decision can be made without measuring the pros and cons and factoring the parameters.

Humanity takes great pride in its ability to evaluate. However, in many cases, a machine can do better. Chess represents our pride in our thinking ability.
A chessboard is often present in movies to symbolize human intelligence.

Today, not a single chess player can beat the best chess engines. One of the extraordinary core capabilities of a chess engine is the evaluation function; it takes many parameters into account more precisely than humans.

Principle 2: Principle 1 leads to a very tough consequence. It is sometimes possible and other times impossible for a human to verify the results that an AI algorithm produces, let alone ensemble meta-algorithms.

Principle 1 has been difficult to detect because of the media hype surrounding face and object recognition. It is easy for a human to check whether the face or object the ML algorithm was supposed to classify was correctly classified.

However, in a decision-making process involving many features, principle 2

rapidly appears. In this chapter, we will identify what results and convergence to measure and decide how to measure it. We will also explore measurement and evaluation methods.

This chapter covers the following topics:

- Evaluation of the episodes of a learning session
- Numerical convergence measurements
- An introduction to numerical gradient descent
- Decision tree supervised learning as an evaluation method

The first thing is to set evaluation goals. To do this, we will decide what to measure and how.

Tracking down what to measure and deciding how to measure it

We will now tackle the tough task of finding the factors that can make a system go wrong.

The model built in the previous chapters can be summed up as follows:

$$l_v = \begin{bmatrix} 0.0002 \\ 0.2 \\ 0.9 \\ 0.0001 \\ 0.4 \\ 0.6 \end{bmatrix} \rightarrow \text{softmax}(l_v) \rightarrow \begin{bmatrix} 0.111 \\ 0.135 \\ 0.273 \\ 0.111 \\ 0.165 \\ 0.202 \end{bmatrix} \rightarrow \text{one-hot} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \rightarrow R \rightarrow \begin{bmatrix} 0 \\ 0 \\ 100 \\ 0 \\ 0 \\ 0 \end{bmatrix} \rightarrow \text{gamma} \rightarrow Q \rightarrow \text{Results}$$

From l_v, the availability vector (capacity in a warehouse, for example), to R, the process creates the reward matrix from the raw data (*Chapter 2*, *Building a Reward Matrix – Designing Your Datasets*) required for the MDP reinforcement learning program (*Chapter 1*, *Getting Started with Next-Generation Artificial Intelligence through Reinforcement Learning*). As described in the previous chapter, a softmax(l_v) function is applied to l_v. In turn, a one-hot(softmax(l_v)) is

applied, which is then converted into the reward value *R*, which will be used for the *Q* (Q-learning) algorithm.

The MDP-driven Bellman equation then runs from reading *R* (the reward matrix) to the results. Gamma is the learning parameter, *Q* is the Q-learning function, and the results represent the final value of the states of the process.

The parameters to be measured are as follows:

- The company's input data. Ready-to-use datasets such as MNIST are designed to be efficient for an exploration phase. These ready-made datasets often contain some noise (unreliable data) to make them realistic. The same process must be achieved with raw company data. The only problem is that you cannot download a corporate dataset from somewhere. You have to build time-consuming datasets.
- The weights and biases that will be applied.
- The activation function (a logistic function or other).
- The choices to make after the one-hot process.
- The learning parameter.
- Episode management through convergence.
- A verification process through interactive random checks and independent algorithms such as supervised learning to control unsupervised algorithms.

In real-life company projects, a system will not be approved until tens of thousands of results have been produced. In some cases, a corporation will approve the system only after hundreds of datasets with millions of data samples have been tested to be sure that all scenarios are accurate. Each dataset represents a scenario that consultants can work on with parameter scripts. The consultant introduces parameter scenarios that are tested by the system and measured. In decision-making systems with up to 200 parameters, a consultant will remain necessary for months in an industrial environment. A reinforcement learning program will be on its own to calculate events. However, even then, consultants are needed to manage the hyperparameters. In real-life systems, with high financial stakes, quality control will always remain essential.

Measurement should thus apply to generalization more than simply applying to a single or few datasets. Otherwise, you will have a natural tendency to control the

parameters and overfit your model in a too-good-to-be-true scenario.

For example, say you wake up one morning and look at the sky. The weather is clear, the sun is shining, and there are no clouds. The next day, you wake up and you see the same weather. You write this down in a dataset and send it off to a customer for weather prediction. Every time the customer runs the program, it predicts clear sunny skies! That what overfitting leads to! This explains why we need large datasets to fully understand how to use an algorithm or illustrate how a machine learning program works.

Beyond the reward matrix, the reinforcement program in the first chapter had a learning parameter $\lambda = 0.8$, shown in `mdp03.py`, which is used for this section:

```
# Gamma: It's a form of penalty or uncertainty for learning
# If the value is 1, the rewards would be too high.
# This way the system knows it is learning.
gamma = 0.8
```

The λ learning parameter in itself needs to be closely monitored because it introduces uncertainty into the system. This means that the learning process will always remain a probability, never a certainty. One might wonder why this parameter is not just taken out. Paradoxically, that will lead to even more global uncertainty. The more the λ learning parameter tends to 1, the more you risk overfitting your results. Overfitting means that you are pushing the system to think it's learning well when it isn't. It's exactly like a teacher who gives high grades to everyone in the class all the time. The teacher would be overfitting the grade-student evaluation process, and nobody would know whether the students have learned anything.

The results of the reinforcement program need to be measured as they go through episodes. The range of the learning process itself must be measured.

All of these measurements will have a significant effect on the results obtained.

The best way to start is by measuring the quality of convergence of the system.

If the system provides good convergence, you might avoid the headache of having to go back and check everything.

Convergence

Convergence measures the distance between the present state of a training session and the goal of the training session. In a reinforcement learning program, an MDP, for example, there is no training data, so there is no target data to compare to.

However, two methods are available:

1. **Implicit convergence**: In this case, we run the training for a large number of episodes, 50,000, for example. We know through trial and error that the program will reach a solution by then.
2. **Numerically controlled gradient descent**: We measure the training progress at each episode and stop when it is safe to do so.

Implicit convergence

In the last part of `mdp01.py` in the first chapter, a range of 50,000 was implemented. In this chapter, we will run `mdp03.py`.

In the last part of `mdp01.py`, the idea was to set the number of episodes at such a level that meant that convergence was certain. In the following code, the range (`50000`) is a constant:

```
for i in range(50000):
    current_state = ql.random.randint(0, int(Q.shape[0]))
    PossibleAction = possible_actions(current_state)
    action = ActionChoice(PossibleAction)
    reward(current_state,action,gamma)
```

Convergence, in this case, will be defined as the point at which no matter how long you run the system, the `Q` result matrix will not change anymore.

By setting the range to `50000`, you can test and verify this. As long as the reward matrices remain homogeneous, this will work. If the reward matrices strongly vary from one scenario to another, this model will produce unstable results.

Try to run the program with different ranges. Lower the ranges until you see that

the results are not optimal.

Numerically controlled gradient descent convergence

In this section, we will use `mdp03.py`, a modified version of `mdp01.py` explored in *Chapter 1*, with an additional function: numerically controlled gradient descent.

Letting the MDP train for 50,000 will produce a good result but consume unnecessary CPU. Using a numerically controlled gradient descent evaluation function will save a lot of episodes. Let's see how many.

First, we need to define the gradient descent function based on a derivative. Let's have a quick review of what a derivative is.

$$\frac{f(x+h)-f(x)}{h}$$

h is the value of the step of the function. Imagine that *h* represents each line of a bank account statement. If we read the statement line by line, *h* = 1. If we read two lines at a time, *h* = 2.

Reading the present line of the bank account statement = *f*(*x*) = *a certain amount*.

When you read the next line of the bank account, the function is (*f* + *h*) = *the amount after f*(*x*). If you had 100 units of currency in your bank account at *f*(*x*) and spent 10 units of currency, on the next line, *x* + *h*, you would have *f*(*x* + *h*) = *90 units of currency* left.

The gradient provides the direction of your slope: up, down, or constant. In this case, we can say that the slope, the **gradient**, is doing downward, as shown in the following graph, which illustrates the decreasing values of *y*(cost, loss) as *x* increases (training episodes):

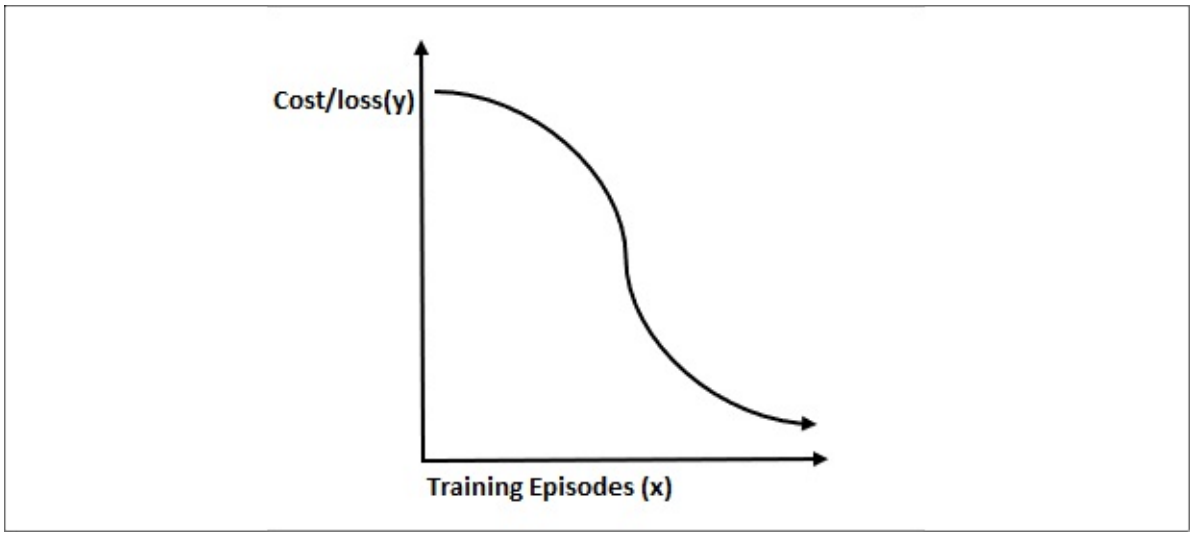


Figure 3.1: Plotting the decreasing cost/loss values as training episodes increase

We also need to know by how much your bank account is changing – how much the **derivative** is worth. In this case, derivative means by how much the balance of your bank account is changing on each line of your bank statement. In this case, you spent 10 units of currency in one bank statement line, so the derivative at this value of *x* (line in your bank account) = –10.

In the following code of the Bellman equation as seen in *Chapter 1*, *Getting Started with Next-Generation Artifcial Intelligence through Reinforcement Learning*, the step of the loop is 1 as well:

```
for i in range(sec):
```

Since $i = 1$, $h = 1$ in our gradient descent calculation can be simplified:

$$\frac{f(x+1) - f(x)}{1} = f(x+1) - f(x)$$

We now define $f(x)$ in the following code:

```
conv=Q.sum()
```

`conv` is the sum of the 6×6 `Q` matrix that is slowly filling up as the MDP training progresses. Thus $f(x)$ = `conv=Q.sum()` = sum of `Q`. The function adds up all the values in `Q` to have a precise value of the state of the system at each *i*.

$f(x)$ = the state of the system at $i - 1$

$f(x + 1)$ is the value of the system at i:

```
Q.sum()
```

We must remember that the `Q` matrix is increasing progressively as the MDP process continues to train. We measure the distance between two steps, h. This distance will decrease. Now we have:

$$f(x + 1) - f(x) = \texttt{-Q.sum()+conv}$$

- First we implement additional variables for our evaluation function, which uses gradient descent at line 83 of `mdp01.py`:

```
ci=0          # convergence counter which counts the nu
mber of episodes
conv=0        # sum of Q at state 1 and then every x ep
isodes
nc=1          # numerical convergence activated to perf
orm numerical-controlled gradient descent
xi=100        # xi episode optimizer: stop as soon as c
onvergence reached + xi-x(unknown)
sec=2500      # security number of episodes for this ma
trix size brought down from 50,000 to 2,500
cq=ql.zeros((2500, 1))
```

- `nc=1` activates the evaluation function, and `ci` begins to count the episodes it will take with this function:

```
for i in range(sec):
    current_state = ql.random.randint(0, int(Q.shape[0]
))
    PossibleAction = possible_actions(current_state)
    action = ActionChoice(PossibleAction)
    reward(current_state,action,gamma)
    ci+=1                              # convergence counte
r incremented by 1 at each state
    if(nc==1):                          # numerical converg
ence activated
```

- At the first episode, `i==1`, $f(x)$= `Q.sum()` as planned:

```
        if(i==1):                  # at state one, conv is
 activated
```

```
            conv=Q.sum()    # conv= the sum of Q
```

- $f(x + 1)$ = `-Q.sum()+conv` is applied:

```
        print("Episode",i,"Local derivative:",-Q.sum()+
conv,...
```

- The distance, the absolute value of the derivative, is displayed and stored because we will be using it to plot a figure with Matplotlib:

```
        print(... "Numerical Convergence value estimato
r",
            Q.sum()-conv)
                cq[i][0]=Q.sum()-conv
```

- `xi=100` plays a critical role in this numerically controlled gradient descent function. Every `xi`, the process stops to check the status of the training process:

```
    if(ci==xi):   # every 100 episodes the system check
s to see...
```

There are two possible cases: **a)** and **b)**.

Case a) As long as the local derivative is >0 at each episode, the MDP continues its training process:

```
            if(conv!=Q.sum()): # if the sum of Q changes...
                conv=Q.sum()   # ...the training isn't over
, conv is updated
                ci=0           # ...the convergence counter
 is set to 0
```

The output will display varying local derivatives:

```
Episode 1911 Local derivative: -9.094947017729282e-13 Numer
ical Convergence value estimator 9.094947017729282e-13
Episode 1912 Local derivative: -9.094947017729282e-13 Numer
ical Convergence value estimator 9.094947017729282e-13
Episode 1913 Local derivative: -1.3642420526593924e-12 Nume
rical Convergence value estimator 1.3642420526593924e-12
```

Case b) When the derivative value reaches a constant value for `xi` episodes, the MDP has been trained and the training can now stop:

```
            if(conv==Q.sum()):        # ...if the sum of Q
has changed
                print(i,conv,Q.sum()) # ...if it hasn't the
 training is over
                break                 # ...the system stops
 training
```

The output will display a constant derivate, `xi`, before the training stops:

```
Episode 2096 Local derivative: 0.0 Numerical Convergence va
lue estimator 0.0
Episode 2097 Local derivative: 0.0 Numerical Convergence va
lue estimator 0.0
Episode 2098 Local derivative: 0.0 Numerical Convergence va
lue estimator 0.0
Episode 2099 Local derivative: 0.0 Numerical Convergence va
lue estimator 0.0
```

When the training is over, the number of training episodes is displayed:

```
number of episodes: 2099
```

2,099 is a lot less than the 50,000 implicit convergence episodes, which proves the efficiency of this numerically controlled gradient descent method.

At the end of the learning process, you can display a Matplotlib figure containing the convergence level of each episode that we had stored in `cq=ql.zeros((2500, 1))`:

```
            cq[i][0]=Q.sum()-conv
```

The figure is displayed with a few lines of code:

```
import matplotlib.pyplot as plt
plt.plot(cq)
plt.xlabel('Episodes')
plt.ylabel('Convergence Distances')
plt.show()
```

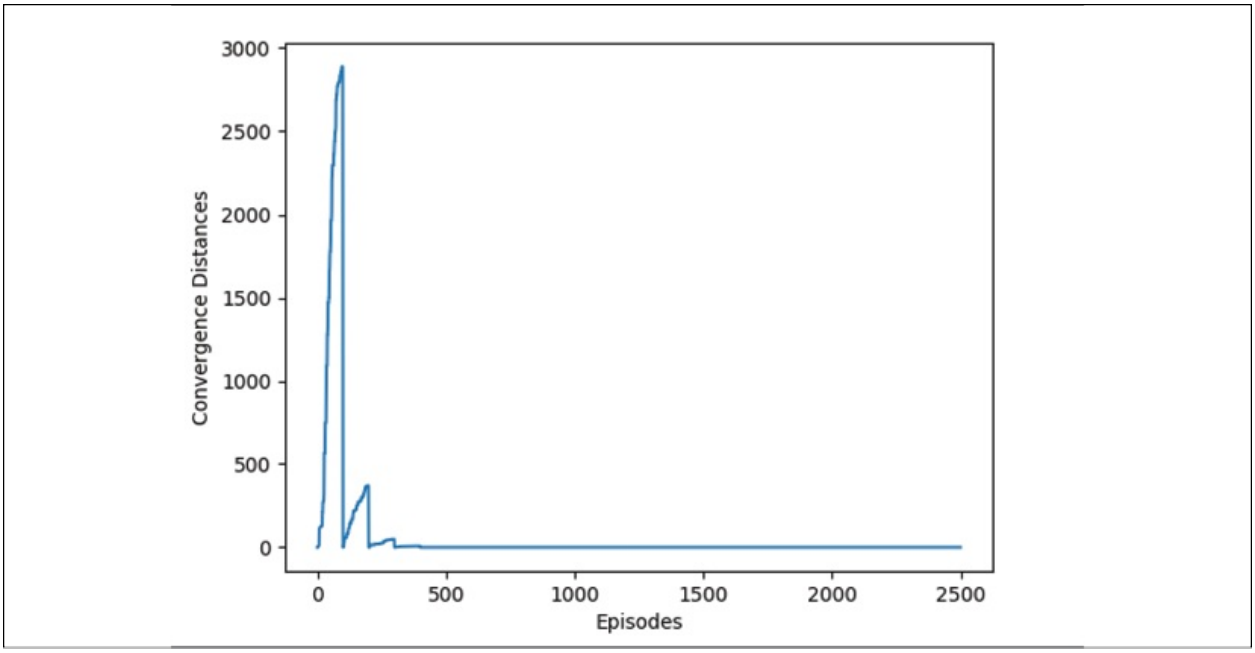


Figure 3.2: A plot demonstrating numerical convergence

This graph shows the numerical convergence. As you can see in the graph, the cost or loss decreases as the number of training episodes increases, as explained earlier in this chapter.

Please note the following properties of this gradient descent method:

- The number of episodes will vary from one training session to another because the MDP is a random process.
- The training curve at local episodes is sometimes erratic because of the random nature of the training process. Sometimes, the curve will go up instead of down locally. In the end, it will reach 0 and stay there.
- If the training curve increases locally, there is nothing you can do. An MDP does no backpropagation to modify weights, parameters, or strategies, as we will see when we look at artificial neural networks (ANNs), for example, in *Chapter 8*, *Solving the XOR Problem with a Feedforward Neural Network*. No action is required in an MDP process. You can try to change the learning rate or go back and check your reward matrix and the preprocessing phase implemented on the raw datasets.
- If the training curve does not reach 0 and stay there, check the learning parameters, the reward matrix, and the preprocessing phase implemented on the raw datasets. You might even have to go back and check the noise

(defective data or missing data) in the initial datasets.

Once the MDP training is over, do some random tests using the functionality provided at line 145 and explained in *Chapter 1*:

```
origin=int(input("index number origin(A=0,B=1,C=2,D=3,E=4,F
=5): "))
```

For example, when prompted for an input, enter `1` and see if the result is correct, as shown in the following output:

```
index number origin(A=0,B=1,C=2,D=3,E=4,F=5): 1
…/…
print("Path:")
-> B
-> D
-> C
```

This random test verification method will work efficiently with a relatively small reward matrix.

However, this approach will prove difficult with a size 25×25 reward matrix, for example. The machine easily provides a result. But how can we evaluate it? In that case, we have reached the limit of human analytic capacity. In the preceding code, we entered a starting point and obtained an answer. With a small reward matrix, it is easy to visually check and see if the answer is correct. When analyzing 25 × 25 = 625 cells, it would take days to verify the results. For the record, bear in mind that when Andrey Markov invented his approach over 100 years ago, he used a pen and paper! However, we have computers, so we must use an evaluation algorithm to evaluate the results of our MDP process.

The increasing volumes of data and parameters in a global world have made it impossible for humans to outperform the ever-growing intelligence of machines.

Evaluating beyond human analytic capacity

An efficient manager has a high evaluation quotient. A machine often has a better one in an increasing number of fields. The problem for a human is to understand the evaluation machine intelligence has produced.

Sometimes a human will say "that's a good machine thinking result" or "that's a bad result," without being able to explain why or determine whether there is a better solution.

Evaluation is one of the major keys to efficient decision-making in all fields: from chess, production management, rocket launching, and self-driving cars to data center calibration, software development, and airport schedules.

We'll explore a chess scenario to illustrate the limits of human evaluation.

Chess engines are not high-level deep learning-based software. They rely heavily on evaluations and calculations. They evaluate much better than humans, and there is a lot to learn from them. The question now is to know whether any human can beat a chess engine or not. The answer is no.

To evaluate a position in chess, you need to examine all the pieces, their quantitative value, their qualitative value, the cooperation between pieces, who owns each of the 64 squares, the king's safety, the bishop pairs, the knight positioning, and many other factors.

Evaluating a position in a chess game shows why machines are surpassing humans in quite a few decision-making fields.

The following scenario is after move 23 in the Kramnik-Bluebaum 2017 game. It cannot be correctly evaluated by humans. It contains too many parameters to analyze and too many possibilities.



Figure 3.3: Chess example scenario

It is white's turn to play, and a close analysis shows that both players are lost at this point. In a tournament like this, they must each continue to keep a poker face. They often look at their position with a confident face to hide their dismay. Some even shorten their thinking time to make their opponent think they know where they are going.

These unsolvable positions for humans are painless to solve with chess engines, even for cheap, high-quality chess engines on a smartphone. This can be generalized to all human activity that has become increasingly complex, unpredictable, and chaotic. Decision-makers will increasingly rely on AI to help them make the right choices.

No human can play chess and evaluate the way a chess engine does by simply calculating the positions of the pieces, their squares of liberty, and many other parameters. A chess engine generates an evaluation matrix with millions of calculations.

The following table is the result of an evaluation of only one position among many others (real and potential).

Position evaluated	**0,3**					
White	**34**					
	Initial	**Position**	**Value**		**Quality**	**Total**

	position				**Value**	**Value**
Pawn	a2	a2	1	a2-b2 small pawn island	0,05	1,05
Pawn	b2	b2	1	a2-b2 small pawn island	0,05	1,05
Pawn	c2	x	0	Captured	0	0
Pawn	d2	d4	1	Occupies center, defends Be5	0,25	1,25
Pawn	e2	e2	1	Defends Qf3	0,25	1,25
Pawn	f2	x	0	Captured	0	0
Pawn	g2	g5	1	Unattacked, attacking 2 squares	0,3	1,3
Pawn	h2	h3	1	Unattacked, defending g4	0,1	1,1
Rook	a1	c1	5	Occupying c-file, attacking b7 with Nd5-Be5	1	6
Knight	b1	d5	3	Attacking Nb6, 8 squares	0,5	3,5
BishopDS	c1	e5	3	Central position, 10 squares, attacking c7	0,5	3,5
Queen	d1	f3	9	Battery with Bg2, defending Ne5, X-Ray b7	1	11
King	e1	h1	0	X-rayed by Bb6 on a7-g1 diagonal	-0,5	-0,5
BishopWS	f1	g2	3	Supporting Qf3 in defense and attack	0,5	3,5

Knight	g1	x	0	Captured	0	0
Rook	h1	x	0	Captured	0	0
			29		5	34
						White: 34

The value of the position of white is 34.

White	**34**					
Black	**33,7**					
	Initial position	**Position**	**Value**		**Quality Value**	**Total Value**
Pawn	a7	a7	1	a7-b7 small pawn island	0,05	1,05
Pawn	b7	b7	1	a7-b7 small pawn island	0,05	1,05
Pawn	c7	x	0	Captured	0	0
Pawn	d7	x	0	Captured	0	0
Pawn	e7	f5	1	Doubled, 2 squares	0	1
Pawn	f7	f7	1		0	1
Pawn	g7	g6	1	Defending f5 but abandoning Kg8	0	1
Pawn	h7	h5	1	Well advanced with f5,g6	0,1	1,1
				Semi-open d-file attacking		

Rook	a8	d8	5	Nd5	2	7
Knight	b8	x	0	Captured	0	0
BishopDS	c8	b6	3	Attacking d4, 3 squares	0,5	3,5
Queen	d8	e6	9	Attacking d4,e5, a bit cramped	1,5	10,5
King	e8	g8	0	f6,h6, g7,h8 attacked	-1	-1
BishopWS	f8	x	0	Captured, white lost bishop pair	0,5	0,5
Knight	g8	e8	3	Defending c7,f6,g7	1	4
Rook	h8	f8	5	Out of play	-2	3
			31		2,7	Black: 33,7

The value of black is 33.7.

So white is winning by 34 – 33.7 = 0.3.

The evaluation system can easily be represented with two McCulloch-Pitts neurons, one for black and one for white. Each neuron would have 30 weights = $\{w_1, w_2 \ldots w_{30}\}$, as shown in the previous table. The sum of both neurons requires an activation function that converts the evaluation into 1/100th of a pawn, which is the standard measurement unit in chess. Each weight will be the output of squares and piece calculations. Then the MDP can be applied to Bellman's equation with a random generator of possible positions.

Present-day chess engines contain this type of brute calculation approach. They don't need more to beat humans.

No human, not even world champions, can calculate these positions with this accuracy. The number of parameters to take into account overwhelms them each time they reach positions like these. They then play more or less randomly with a possibly good idea in mind. The chances of success against a chess engine resemble a lottery sometimes. Chess experts discover this when they run human-played games with powerful chess engines to see how the game plays out. The players themselves now tend to reveal their incapacity to provide a deep analysis when asked why they made a questionable move. It often takes hours to go through a game, its combinations and find the reasons of a bad move. In the end, the players will often use a machine to help them understand what happened.

The positions analyzed here represent only one possibility. A chess engine will test millions of possibilities. Humans can test only a few.

Measuring a result like this has nothing to do with natural human thinking. Only machines can think like that. Not only do chess engines solve the problem, but they are also impossible to beat.

Principle 1: At one point, there are problems humans face that only machines can solve.

Principle 2: Sometimes, it will be possible to verify the result of an ML system, sometimes not. However, we must try to find ways to check the result.

One solution to solve the problem of principle 2 is to verify an unsupervised algorithm with a supervised algorithm through random samples.

Using supervised learning to evaluate a result that surpasses human analytic capacity

More often than not, an AI solution exceeds a human's capacity to analyze a situation in detail. It is often too difficult for a human to understand the millions of calculations a machine made to reach a conclusion and explain it. To solve that problem, another AI, ML, or DL algorithm will provide assisted AI

capability.

Let's suppose the following:

- The raw data preprocessed by the neural approach of *Chapter 2, Building a Reward Matrix – Designing Your Datasets*, works fine. The reward matrix looks fine.
- The MDP-driven Bellman equation provides good reinforcement training results.
- The convergence function and values work.
- The results on this dataset look satisfactory but the results are questioned.

A manager or user will always come up with a killer question: how can you prove that this will work with other datasets in the future and confirm 100% that the results are reliable?

The only way to be sure that this whole system works is to run thousands of datasets with hundreds of thousands of product flows.

The idea now is to use supervised learning to create an independent way of checking the results. One method is to use a decision tree to visualize some key aspects of the solution and be able to reassure the users and yourself that the system is reliable.

Decision trees provide a white box approach with powerful functionality. In this section, we will limit the exploration to an intuitive approach. In *Chapter 5, How to Use Decision Trees to Enhance K-Means Clustering*, we will go into the theory of decision trees and random trees and explore more complex examples.

In this model, the features of the input are analyzed so that we can classify them. The analysis can be transformed into decision trees depending on real-time data, to create a distribution representation to predict future outcomes.

For this section, you can run the following program:

`Decision_Tree_Priority_classifier.py`

Or the following Jupyter notebook on Google Colaboratory:

`DTCH03.ipynb`

Google Colaboratory might have the two following packages installed:

```
import collections        # from Python library container da
tatypes
import pydotplus          # a Python Interface to Graphviz's
 Dot language.(dot-V command line
```

This could help you avoid installing them locally, which might take some time if you get a Graphviz requirement message.

Both programs produce the same decision tree image:

`warehouse_example_decision_tree.png`

The intuitive description of this decision tree approach runs in 5 steps:

Step 1: Represent the features of the incoming orders to store in a warehouse – for example:

```
features = [ 'Priority/location', 'Volume', 'Flow_optimizer'
 ]
```

In this case, we will limit the model to three properties:

- Priority/location, which is the most important property in a warehouse flow in this model
- Volumes to transport
- Optimizing priority – the financial and customer satisfaction property

Step 2: Provide priority labels for the learning dataset:

```
Y = ['Low', 'Low', 'High', 'High', 'Low', 'Low']
```

Step 3: Providing the dataset input matrix, which is the output matrix of the reinforcement learning program. The values have been approximated but are enough to run the model. They simulate some of the intermediate decisions and transformations that occur during the decision process (ratios applied, uncertainty factors added, and other parameters). The input matrix is `X`:

```
X = [[256, 1,0],
     [320, 1,0],
     [500, 1,1],
     [400, 1,0],
     [320, 1,0],
     [256, 1,0]]
```

The features in step 1 apply to each column.

The values in step 2 apply to every line.

The values of the third column [0,1] are discrete indicators for the training session.

Step 4: Run a standard decision tree classifier. This classifier will distribute the representations (distributed representations) into two categories:

- The properties of high-priority orders
- The properties of low-priority orders

There are many types of algorithms. In this case, a standard `sklearn` function is called to do the job, as shown in the following source code:

```
classify = tree.DecisionTreeClassifier()
classify = classify.fit(X,Y)
```

Step 5: Visualization separates the orders into priority groups. Visualizing the tree is optional but provides a trendy white box approach. You will have to use:

- `import collections`, a Python container library.
- `import pydotplus`, a Python interface to Graphviz's dot language. You can choose to use Graphviz directly with other variations of this source code.

The source code will take the nodes and edges of the decision tree, draw them, and save the image in a file as follows:

```
info = tree.export_graphviz(classify,feature_names=features
,
    out_file=None, filled=True,rounded=True)
graph = pydotplus.graph_from_dot_data(info)
```

```
edges = collections.defaultdict(list)
for edge in graph.get_edge_list():
    edges[edge.get_source()].append(int(edge.get_destinatio
n()))
for edge in edges:
    edges[edge].sort()
    for i in range(2):
        dest = graph.get_node(str(edges[edge][i]))[0]
graph.write_png(<your file name here>.png)
```

The file will contain this intuitive decision tree:

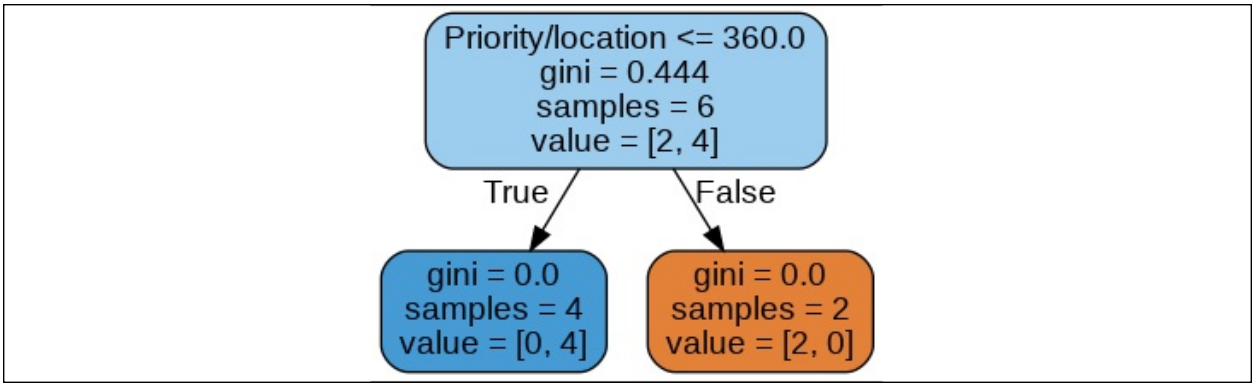


Figure 3.3: A decision tree

The image produces the following information:

- A decision tree represented as a graph that has nodes (the boxes) and edges (the lines).
- When *gini*=0, this box is a **leaf**; the tree will grow no further.
- *gini* means **Gini impurity**. At an intuitive level, Gini impurity will focus on the highest values of Gini impurity to classify the samples. We will go into the theory of Gini impurity in *Chapter 5, How to Use Decision Trees to Enhance K-Means Clustering*.
- *samples* = 6. There are six samples in the training dataset:
    - Priority/location <=360.0 is the largest division point that can be visualized:

```
X = [[256, 1,0],
     [320, 1,0],
     [500, 1,1],
     [400, 1,0],
```

```
        [320, 1,0],
        [256, 1,0]]
```

- The false arrow points out the two values that are not <=360. The ones that are classified as `True` are considered as low-priority values.

After a few runs, the user will get used to visualizing the decision process as a white box and trust the system.

Each ML tool suits a special need in a specific situation. In the next chapter, *Optimizing Your Solutions with K-Means Clustering*, we will explore another machine learning algorithm: *k-means clustering*.

Summary

This chapter drew a distinction between machine intelligence and human intelligence. Solving a problem like a machine means using a chain of mathematical functions and properties. Machine intelligence surpasses humans in many fields.

The further you get in machine learning and deep learning, the more you will find mathematical functions that solve the core problems. Contrary to the astounding amount of hype, mathematics relying on CPUs is replacing humans, not some form of mysterious conscious intelligence.

The power of machine learning reaches beyond human *mathematical reasoning*. It makes ML generalization to other fields easier. A mathematical model, without the complexity of humans entangled in emotions, makes it easier to deploy the same model in many fields. The models of the first three chapters of this book can be used for self-driving vehicles, drones, robots in a warehouse, scheduling priorities, and much more. Try to imagine as many fields you can apply these to as possible.

Evaluation and measurement are at the core of machine learning and deep learning. The key factor is constantly monitoring convergence between the results the system produces and the goal it must attain. The door is open to the constant adaptation of the parameters of algorithms to reach their objectives.

When a human is surpassed by an unsupervised reinforcement learning algorithm, a decision tree, for example, can provide invaluable assistance to human intelligence.

The next chapter, *Optimizing Your Solutions with K-Means Clustering*, goes a step further into machine intelligence.

Questions

1. Can a human beat a chess engine? (Yes | No)
2. Humans can estimate decisions better than machines with intuition when it comes to large volumes of data. (Yes | No)
3. Building a reinforcement learning program with a Q function is a feat in itself. Using the results afterward is useless. (Yes | No)
4. Supervised learning decision tree functions can be used to verify that the result of the unsupervised learning process will produce reliable, predictable results. (Yes | No)
5. The results of a reinforcement learning program can be used as input to a scheduling system by providing priorities. (Yes | No)
6. Can artificial intelligence software think like humans? (Yes | No)

Further reading

- For more on decision trees: https://youtu.be/NsUqRe-9tb4
- For more on chess analysis by experts such as Zoran Petronijevic, with whom I discussed this chapter: https://chessbookreviews.wordpress.com/tag/zoran-petronijevic/, https://www.chess.com/fr/member/zoranp
- For more on AI chess programs: https://deepmind.com/blog/article/alphazero-

shedding-new-light-grand-games-chess-shogi-and-go

4

Optimizing Your Solutions with K-Means Clustering

No matter how much we know, the key point is the ability to deliver an **artificial intelligence** (**AI**) solution. Implementing a **machine learning** (**ML**) or **deep learning** (**DL**) program remains difficult and will become more complex as technology progresses at exponential rates.

There is no such thing as a simple or easy way to design AI systems. A system is either efficient or not, beyond being either easy or not. Either the designed AI solution provides real-life practical uses, or it builds up into a program that fails to work in various environments beyond the scope of its training sets.

This chapter doesn't deal with how to build the most difficult system possible to show off our knowledge and experience. It faces the hard truth of real-life delivery and ways to overcome obstacles. For example, without the right datasets, your project will never take off. Even an unsupervised ML program requires reliable data in some form or other.

Transportation itineraries on the road, on trains, in the air, in warehouses, and increasingly in outer space require well-tuned ML algorithms. The staggering expansion of e-commerce generates huge warehouse transportation needs with automated guided vehicles (AGVs), then endless miles on the road, by train, or by air to deliver the products. Distance calculation and optimization is now a core goal in many fields. An AGV that optimizes its warehouse distance to load or unload trucks will make the storage and delivery processes faster for customers that expect their purchases to arrive immediately.

This chapter provides the methodology and tools needed to overcome everyday AI project obstacles with k-means clustering, a key ML algorithm.

This chapter covers the following topics:

- Designing datasets
- The design matrix
- Dimensionality reduction
- Determining the volume of a training set
- k-means clustering
- Unsupervised learning
- Data conditioning management for the training dataset
- Lloyd's algorithm
- Building a Python k-means clustering program
- Hyperparameters
- Test dataset and prediction
- Saving and using an ML model with Pickle

We'll begin by talking about how to optimize and manage datasets.

Dataset optimization and control

At one point, a manager or customer will inevitably ask an AI expert about the exact data required for an ML project, and in which format it's needed. Providing an answer will take some hard thinking and work.

One might wonder why the data is not open, like when we download ready-to-use datasets to learn AI algorithms. In corporate life, there are security rules and processes. Often the data required is on one or more servers. You will not be granted permission to do anything you want. You will have to specify your needs and requests. Obtaining data in the way required for AI comes at a cost for a corporation. You will have to justify your requests.

Start by properly designing the dataset and choosing the right ML model. The dataset and ML model will fit the fundamental requirement to optimize AGV distances. Each AGV in a given warehouse must reduce the distance it takes to go from a pier (where boats might drop off cargo) to a storage area, from one storage area to another area (packaging, for example), and from a storage area to

a pier. This will bring the overall costs of a warehouse down and maximize profit.

Designing a dataset and choosing an ML/DL model

On paper, finding a good model for an AGV comes down to minimizing the distance it takes to move something from point A to point B. Let's consider a scenario where we want to move products from a warehouse over to a pier. The sum of the distances *D* could be a way to measure the process:

$$D = \sum_{p_1}^{p_n} f(p)$$

f(*p*) represents the action of going from a warehouse location to a pier and the distance it represents. It takes you from the location in a shop where you picked something up, (*p*), and the door of the shop on your way out. You can imagine that if you go straight from that location, pay, and go out, then that is the shortest way. But if you pick the product up, wander around the shop first, and then go out, the distance (and time) is longer. The sum of all of the distances of all the people wandering in this shop, for example, is *D*.

The concept of the problem to solve in any self-guided bot system can be summed up as follows:

find the wanderers

How can a bot wander? It is automatic and is often guided by efficient ML programs. But a bot, like any other form of transportation, often encounters obstacles.

When a bot encounters an obstacle, either it stops, waits, or takes another route.

We now have a paradigm to investigate and implement:

- Detect the wanderers
- Optimize the choice of bots

We all know that the shortest point from location A to location B is a straight line. Right? Well, in a warehouse, as in life, it is not always true! Suppose you are in your car going in a straight line from A to B but there is a huge traffic jam. It could take a very long time to go a relatively short distance. If you took a right turn and drove around the jam, you might save a lot of time. You end up consuming more gas, and the cost of driving from A to B goes up. You are a wanderer.

In real-life traffic, there is not much you can do. You cannot decide that cars can only drive at certain hours on a road going from A to B. You cannot decide to send the cars to the locations that minimize traffic. In real life, this would mean telling the driver to go to another mall, another restaurant, or any other similar location to avoid building up traffic. That would not work!

But if you are a warehouse manager that can control all of the AGVs, there is a lot you can do about this. You can make sure that AGVs make it quickly to their locations over a very short distance and come back to make room for other AGVs and thus reduce costs. You can detect the wanderers and configure your schedule and AGVs so that they minimize cost and maximize profit. No profit, no warehouse.

Approval of the design matrix

The plan is to first obtain as much data as possible and then choose an ML/DL model. The dataset must contain all locations the bots (the AGVs) come from on their way to the piers on a given day. It's a location-to-pier analysis. Their distances are recorded in their system to provide the basis of an excellent design matrix. A design matrix contains a different example on each row, and each column is a feature. The following format fits the need:

Index	**Bot # (AGV)**	**Start (from location) Timestamp: yyyy,mm,dd,hh,mm**	**End (at the pier) Timestamp: yyyy,mm,dd,hh,mm**	**Location**	**Pier number**	**Distance Meters**
001	1	year-month-day-hour-minute	year-month-day-hour-minute	80	7	92

002	2					
003	3					
004	4					
005	5					

The design matrix is one of the best ways to design an ML solution. In this case:

- **Index**: The mission number of the bot
- **Bot #**: Identifies the vehicle
- **Start**: Timestamp when the bot left a location
- **End**: Timestamp when the bot reaches a pier where a truck is waiting to be loaded
- **Location**: The location in the warehouse where a product should be retrieved
- **Distance**: The distance from the location to the pier in meters

Distance is expressed in meters in the metric system. The metric system is the world's most reliable measurement system because it works in base 10 without having to resort to conversions.

A yard has to be divided by 3 to obtain feet. A foot has to be divided into 12 to obtain inches. To work in smaller units, a 1/16th of an inch may be necessary.

A meter is 100 centimeters, and a centimeter is 10 millimeters, then we can use 1/100 of a millimeter, and so on.

Run your calculations with the metric system even if you have to produce reports in other units of measurement.

Getting approval on the format of the design matrix

Real-life implementations differ from experimenting with ready-to-use downloadable datasets. Information does not come easy in corporations.

In this example, in a real-life situation, let's suppose:

- The bot number is not stored in the mainframe, but in the local system that manages the AGVs.
- In the mainframe, there is a start time, which is when an AGV picks up its load at the location, and an end time when it reaches the pier.
- The location can be obtained in the mainframe as well as the pier.
- No distance is recorded.

It would be necessary to have access to the data in the AGV's local system to retrieve the AGV number and correlate it with the data in the mainframe.

However, things are not so simple in a large organization. For example:

- Retrieving data from the AGV guiding system might not be possible this fiscal year. Those vehicles are expensive, and no additional budget can be allocated.
- Nobody knows the distance from a location to a pier. As long as the AGVs deliver the right products on time at the right piers, nobody so far has been interested in distances.
- The AGV mission codes in the mainframe are not the same as in the local AGV guiding system, so they cannot be merged into a dataset without development.

An AI project, like any other project, can slip away in no time. If a project comes to a standstill, it might just be shelved. Designing datasets requires imagination and responsiveness.

Keeping an AI project alive means moving quickly.

If the project does not move quickly, it will lose momentum. The actors of the project will turn to other projects that are moving more quickly for their company and their careers.

Suppose your project stops because nobody can provide the distances you need to build your model. If you have the start time, end time, and speed, then you can

work around the problem and calculate the distances yourself. If your team does not find this solution quickly, then the project will be at a standstill. The top management will say that the team costs too much to be focused on a project that is not moving ahead, no matter what. The project can be shelved right then and there.

Dimensionality reduction will not only help the AI model; it will also make it easier to gather information.

Dimensionality reduction

Dimensionality reduction can be applied to reduce the number of features in, for example, an image. Each pixel of a 3D image, for example, is linked to a neuron, which in turn brings the representation down to a 2D view with some form of function. For example, converting a color image into shades of a now-gray image can do the trick. Once that is done, simply reducing the values to, for example, 1 (light) or 0 (dark), makes it even easier for the network. Using an image converted to 0 and 1 pixels makes some classification processes more efficient, just like when we avoid a car on the road. We just see the object and avoid it.

We perform dimensionality reduction all day long. When you walk from one office to another on the same floor of a building requiring no stairs or an elevator, you are not thinking that the Earth is round and that you're walking over a slight curve.

You have performed a **dimensionality reduction**. You are also performing a **manifold** operation. It means that locally, on that floor, you do not need to worry about the global roundness of the Earth. Your manifold view of the Earth in your dimensionality reduction representation is enough to get you from your office to another one on that floor.

When you pick up your cup of coffee, you focus on not missing it and aiming for the edges of it. You don't think about every single **feature** of that cup, such as its size, color, decoration, diameter, and the exact volume of coffee in it. You identify the edge of the cup and pick it up. That is dimensionality reduction. Without dimensionality reduction, nothing can be accomplished. It would take you 10 minutes to analyze the cup of coffee and pick it up in that case!

When you pick that cup of coffee up, you test to see whether it is too hot, too cold, or just fine. You don't put a thermometer in the cup to obtain the precise temperature. You have again performed a dimensionality reduction of the features of that cup of coffee. Furthermore, when you picked it up, you computed a manifold representation by just observing the little distance around the cup, reducing the dimension of information around you. You are not worrying about the shape of the table, whether it was dirty on the other side, and other features.

> ML and DL techniques such as dimensionality reduction can be viewed as tools and can be used in any field to speed up calculation times.

Although we often relate dimensionality reduction to ML and DL, dimensionality reduction is as old as mathematics and even humanity! Somebody, long ago, went to a beach and saw that the sun was beautiful. That person, for the first time in humanity, drew a circle in the sand. The circle was not in 3D like the sun, nor did it have color, but the humans around that person were astonished:

- A group of humans were watching the sun
- A human took the color out
- The human also took the 3D view out
- They represented the sun with a circle in a much smaller dimensional space
- The first mathematician was born!

A k-means clustering algorithm provides an efficient way to represent the bot example we are dealing with in this chapter. Each location will form a cluster, as explained in the next section.

A workaround to the missing data problem would be to run the k-means clustering algorithm with the following data format:

Index	**Location**	**Start from location: Timestamp: yyyy,mm,dd,hh,mm**	**End at location: Timestamp: yyyy,mm,dd,hh,mm**
001			

The volume of a training dataset

In this model, we will focus on six locations to analyze. Six locations are chosen in the main warehouse with a group of truck loading points. Taking around 5,000 examples into account, that should represent the work of all the 25 AGVs purchased by **AGI-AI** running 24 hours a day.

Now that we've talked about optimizing and controlling a dataset, let's move on to coming up with actual solutions. In the next section, we'll talk about implementing k-means clustering.

Implementing a k-means clustering solution

The dataset requires preprocessing to be converted into a prototype to prove the financial value of the project.

> Never implement an ML solution in a corporate environment without knowing how much profit it represents and the cost of getting the job done. Without profit, a company will not survive. ML, like any other investment, must provide a return on investment (ROI). In our case, ML will reduce the cost of transportation in the warehouse by reducing AGV distances.

The vision

The primary goal of an ML project involving bots can be summed up in one sentence: finding profit by optimizing bot activity. Achieving that goal will lead to obtaining a budget for a full-scale project.

The data provided does not contain distances. However, an estimation can be made per location as follows:

$$\text{distance} = (\text{end time} - \text{start time})/\text{average speed of a bot}$$

The start location is usually a loading point near a pier in this particular warehouse configuration.

The data

The data provided contains start times s_t, end times end_t, and delivery locations. To calculate distances, we can use the following equation:

$$d_i(ll) = (end_t - s_t)/v$$

- v = velocity of the AGV per minute
- $end_t - s_t$ is expressed in minutes
- d_i = estimated distance an AGV has gone in a given time

A preprocessing program will read the initial file format and data and output a new file, `data.csv`, in the following reduced dimensionality format with two features:

Distance	Location
55	53
18	17

Conditioning management

Data conditioning means preparing the data that will become the input of the system. **Poor conditioning** can have two outcomes:

- Bad data containing noise that makes no difference (large volumes and minor errors)
- Bad data containing noise that makes a difference (regardless of volumes, the data influences the outcome)

In this particular case, let's suppose that out of the 5,000 records provided for the dataset, 25 distances are not reliable. A 0.005% noise level should not be a problem. The amount of acceptable noise depends on each project. It cannot be

an arbitrary figure.

Sometimes, noise will have a profound effect, and sometimes it will not. Suppose 2,500 records out of 5,000 records contained noise. Maybe the 2,500 remaining records provide a sufficient variety of samples to produce a reliable result. In another case, maybe, 10 missing samples out of 5,000 records will stop a project because those 10 samples were the only ones of a special kind that could critically change the calculations.

You will have to experiment and determine the level of acceptable noise for a given project.

Let's get our hands dirty and analyze the data.

The location numbers start at #1. #1 is near the loading point of the products. The bot has to bring the products to this point. To be more precise, in this warehouse configuration, the bot goes and gets a box (or crate) of products and brings them back to location 1. At location 1, humans check the products and package them. After that, humans carefully load the products in the delivery trucks.

The distance from one location to the next is about 1 meter. For example, from location 1 to location 5, the distance is about 5 meters, or 5 m. Also, since all locations lead to location 1 for the AGVs in this model, the theoretical distances will be calculated from location 1. To generalize the rule and define a distance d_i for a location lj the calculation can be simplified:

$$d_i(lj)=lj$$

d_i is expressed in meters. Since the locations start at number 1 through n, the location number is equal to the approximate distance from the first location from which the bots depart.

Let us suppose that, looking at the data, it quickly appears that many distances are superior to their location numbers. That is strange because the distance should be about equal to the location. Thus, by reducing the number of dimensions and focusing on approximations of the main features, key concepts can be represented.

The time has come to build a strategy and a program.

The strategy

As in all ML projects, there are key standard corporate guidelines that should not be avoided:

- Quickly write a proof of concept (POC). A POC will prove that the ML solution will be efficient. In this case, bot activity will be visualized.
- Check the results in detail.
- Calculate the potential optimized profit with a solution that is yet to be found. Profit will justify the investment. The cost can be an indicator. But then cost reduction must be sufficient to increase profit by a significant rate for a given corporation.
- Obtain approval with a solid case and obtain a green light for the project.

Now that our strategy is clear, we can choose a model. k-means clustering is a good algorithm to start with for this project. It will create clusters that almost literally represent the areas in which the AGVs should be located. By choosing simple dimensions, the visual representation is close enough to reality for a user to understand the calculations.

The k-means clustering program

k-means clustering is a powerful unsupervised learning algorithm. We often perform k-means clustering in our lives. Take, for example, a lunch you want to organize for a team of about 50 people in an open space that can just fit those people.

Your friend and another friend first decide to set up a table in the middle. Your friend points out that the people in that room will form a big cluster *k*, and with only one table in the geometric center (or centroid) *c*, it will not be practical. The people near the wall will not have access to the main table, as shown in the following figure.

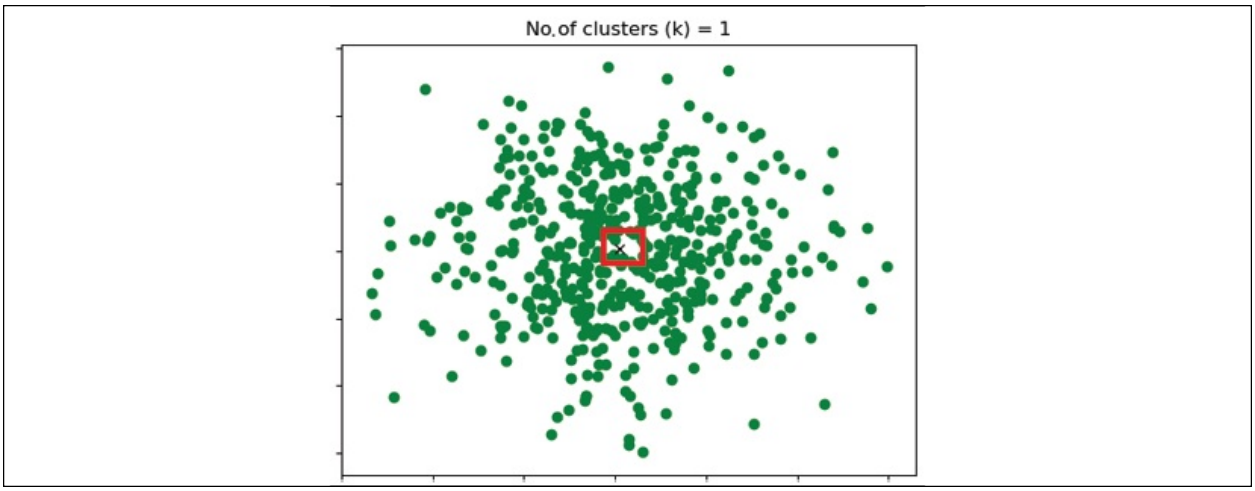


Figure 4.1: A scenario where people try to cluster around a single table

The people not close to the table (the rectangle in the middle) will not have easy access to the table.

You now try two tables (centroids) c_1 and c_2 in various places for two clusters of people k_1 and k_2.

The people x_1 to x_n form a dataset X. When imagining X, it appears that the table is not in the right place.

The best thing to do is to move a table c, and then estimate that the mean distance of the people (a subset of X) to the table will be about the same in their group or cluster k. The same is done for the other table. You draw a line with chalk on the floor to make sure that each group or cluster is at about the mean distance from its table.

This intuitive approach to k-means clustering can be summed up as follows:

- **Step 1**: You have been drawing lines with chalk to decide which group (cluster k) each person x will be in, by looking at the mean distance from the table c.
- **Step 2**: You have been moving the tables around accordingly to optimize step 1.

A Python program simulating a three-table model computed using k-means clustering would produce the following result:

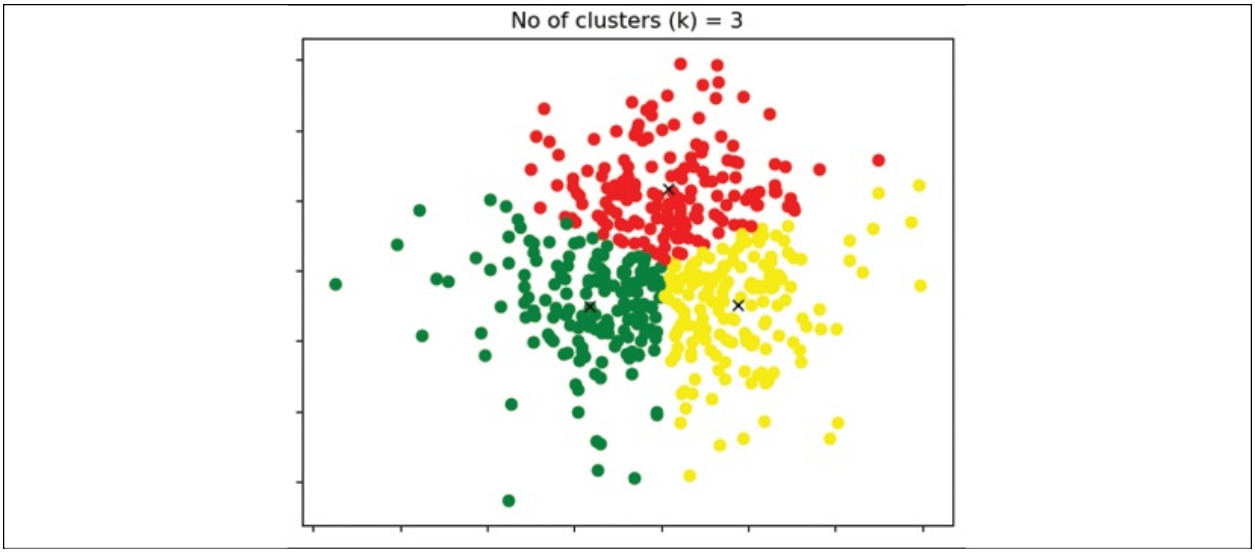


Figure 4.2: A three-table model computed by k-means clustering

Having provided an intuitive example, let's talk about the mathematical definition of k-means clustering.

The mathematical definition of k-means clustering

Dataset *X* provides *N* points. These points or data points are formed by using distance as the *x*-axis in a Cartesian representation and the location as the *y*-axis in a Cartesian representation. This low-level representation is a white box approach, even if the data is processed and transformed by the algorithm. A white box approach is when the process is transparent, and we can actually see what the algorithm is doing. A black box is when an input goes into a system, and we will not be able to understand what the system did by just looking at the result.

However, high-level representations are required to represent more features through clusters. In that case, it will not be possible to see a direct link between the actual meanings and their ML representation. We will explore these high-dimensional representations in the chapters to come.

If you have one bot in location 1 as the first record of your file, it will be represented as *x* axis = 1 and *y* axis = 1 by the black dot, which is the data point, as shown in the following diagram:

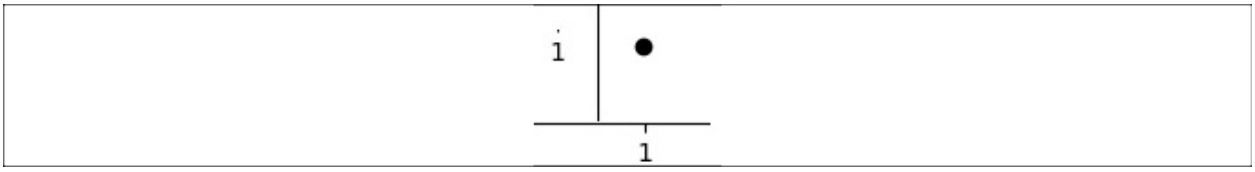


Figure 4.3: Cartesian representation

In this example, 5,000 records are loaded from `data.csv`, which is in the same directory as the program. The data is unlabeled with no linear separation. The goal is to allocate the *X* data points to *K* clusters. The number of clusters is an input value. Each cluster will have its geometric center or centroid. If you decide to have three clusters *K*, then the result will be as follows:

- Three clusters *K* in three colors in a visual representation
- Three geometric centers or centroids representing the center of the mean of the sum of distances of *x* data points of that cluster

If you decide on six clusters, then you will obtain six centroids, and so on.

Described in mathematical terms, the formula in respect of K_k, μ_k is as follows:

$$\min \sum_{k=1}^{K} \sum_{x_n \in K_k} \|x - \mu_k\|^2$$

The sum of each *k* (cluster) from 1 to the number of clusters *K* of the sum of all distances of members x_i to x_n of each cluster *K* from their position to the geometric center (centroid) μ must be minimized.

The smaller the distance from each member *x* to centroid μ, the more the system is optimized. Note that the distance is squared each time because this is a Euclidean distance in this version of the algorithm.

The Euclidean distance, in one dimension, is the distance between two points, *x* and *y*, for example, expressed as follows:

$$\sqrt[2]{(x-y)^2}$$

The distance between *x* and *y* expressed in Euclidean distance is not the real

distance that an AGV will actually travel inside a warehouse. The model in the chapter was built so that the distances would remain sufficiently realistic to make good clusters and improve the organization of the warehouse. It's sufficient because an AGV will often go in nearly straight lines from a pier to the closest aisle and then to a storage point, for example.

To calculate the actual distances, we often use Manhattan distances. Manhattan distances are taxi-cab distances. You calculate the distance up a block then to the left, for example, another block and so on, adding the distances along the way. This is because you can't drive through the buildings.

In our case, it would be like saying that a taxi-cab can only, more or less, drive up and down a given avenue, like a bus, and avoid turning right or left.

We will use Lloyd's algorithm with Euclidean distances to estimate the clusters that AGVs will have to stay in to avoid wandering.

Lloyd's algorithm

There are several variations of Lloyd's algorithm. But all of them follow a common philosophy.

For a given x_n (data point), the distance from the centroid μ in its cluster must be less than going to another center, just like how a person in the lunch example wants to be closer to one table rather than having to go far to get a sandwich because of the crowd.

The best centroid μ for a given x_n is as follows:

$$x_n : \|x_n - \mu_k\|$$

This calculation is done for all μ (centroids) in all the clusters from k_1 to K.

Once each x_i has been allocated to a K_k, the algorithm recomputes μ by calculating the means of all the points that belong to each cluster and readjusts the centroid μ_k.

We've now covered all of the concepts that we need to begin coding. Let's get into the Python program!

The Python program

`k-means_clustering_1.py`, the Python program, uses the `sklearn` library, `pandas` for data analysis (only used to import the data in this program), and `matplotlib` to plot the results as data points (the coordinates of the data) and clusters (data points classified in each cluster with a color). First, the following models are imported:

```
from sklearn.cluster import KMeans
import pandas as pd
from matplotlib import pyplot as plt
```

Next, we'll go through the stages of implementing k-means clustering.

1 – The training dataset

The training dataset consists of 5,000 lines. The first line contains a header for maintenance purposes (data checking), which is *not* used. k-means clustering is an **unsupervised learning** algorithm, meaning that it classifies unlabeled data into cluster-labeled data to make future predictions. The following code displays the dataset:

```
#I. The training Dataset
dataset = pd.read_csv('data.csv')
print(dataset.head())
print(dataset)
```

The `print(dataset)` line can be useful (though not necessary) to check the training data during a prototype phase or for maintenance purposes. The following output confirms that the data was correctly imported:

```
'''Output of print(dataset)
Distance location
0 80 53
1 18 8
2 55 38
...
'''
```

2 – Hyperparameters

Hyperparameters determine the behavior of the computation method. In this case, two hyperparameters are necessary:

- The `k` number of clusters that will be computed. This number can and will be changed during the case study meetings to find out the best organization process, as explained in the next section. After a few runs, we will intuitively set `k` to `6`.
- The *f*-number of features that will be taken into account. In this case, there are two features: distance and location.

The program implements a k-means function, as shown in the following code:

```
#II.Hyperparameters
# Features = 2
k = 6
kmeans = KMeans(n_clusters=k)
```

Note that the `Features` hyperparameter is commented. In this case, the number of features is implicit and determined by the format of the training dataset, which contains two columns.

3 – The k-means clustering algorithm

`sklearn` now does the job using the training dataset and hyperparameters in the following lines of code:

```
#III.k-means clustering algorithm
kmeans = kmeans.fit(dataset) #Computing k-means clustering
```

The `gcenters` array contains the geometric centers or centroids and can be printed for verification purposes, as shown in this snippet:

```
gcenters = kmeans.cluster_centers_
print("The geometric centers or centroids:")
print(gcenters)
'''Ouput of centroid coordinates
[[ 48.7986755 85.76688742]
[ 32.12590799 54.84866828]
```

```
[ 96.06151645 84.57939914]
[ 68.84578885 55.63226572]
[ 48.44532803 24.4333996 ]
[ 21.38965517 15.04597701]]
'''
```

These geometric centers need to be visualized with labels for decision-making purposes.

4 – Defining the result labels

The initial unlabeled data can now be classified into cluster labels, as shown in the following code:

```
#IV.Defining the Result labels
labels = kmeans.labels_
colors = ['blue','red','green','black','yellow','brown','or
ange']
```

Colors can be used for semantic purposes beyond nice display labels. A color for each top customer or leading product can be assigned, for example.

5 – Displaying the results – data points and clusters

To make sense to a team or management, the program now prepares to display the results as **data points** and **clusters**. The data will be represented as coordinates and the clusters as colors with a **geometric center** or **centroid**, as implemented in this code:

```
#V.Displaying the results : datapoints and clusters
y = 0
for x in labels:
    plt.scatter(dataset.iloc[y,0], dataset.iloc[y,1],color=
colors[x])
    y+=1
for x in range(k):
    lines = plt.plot(gcenters[x,0],gcenters[x,1],'kx')
title = ('No of clusters (k) = {}').format(k)
plt.title(title)
plt.xlabel('Distance')
plt.ylabel('Location')
plt.show()
```

The dataset is now ready to be analyzed. The data has been transformed into data points (Cartesian points) and clusters (the colors). The *x* points represent the geometric centers or centroids, as shown in the following screenshot:

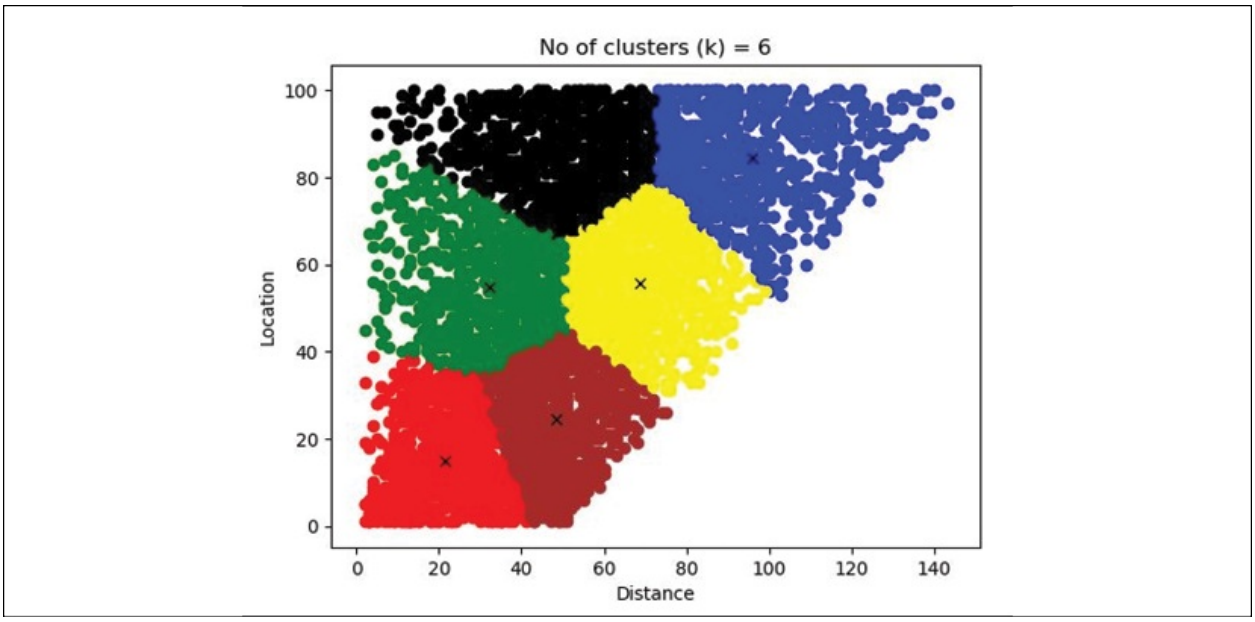


Figure 4.4: Output (data points and clusters)

Test dataset and prediction

In this case, the test dataset has two main functions. First, some test data confirms the **prediction** level of the trained and now-labeled dataset. The input contains random distances and locations. The following code implements the output that predicts which cluster the data points will be in:

```
#VI.Test dataset and prediction
x_test = [[40.0,67],[20.0,61],[90.0,90],
          [50.0,54],[20.0,80],[90.0,60]]
prediction = kmeans.predict(x_test)
print("The predictions:")
print (prediction)
'''
Output of the cluster number of each example
[3 3 2 3 3 4]
'''
```

The second purpose, in the future, will be to enter data for **decision-making** purposes, as explained in the next section.

Saving and loading the model

In this section, `k-means_clustering_1.py` will save the model using Pickle. Pickle, a Python library, saves the model in a serialized file, as shown at the end of the program:

```
# save model
filename="kmc_model.sav"
pickle.dump(kmeans, open(filename, 'wb'))
```

The Python Pickle module is imported in the header of the program:

```
import pickle
```

Now, the model, `kmeans`, is saved in a file named `kmc_model.sav`.

To test this model, we will now open `k-means_clustering_2.py` to load the model without any training involved and make predictions:

```
#load model
filename="kmc_model.sav"
kmeans = pickle.load(open(filename, 'rb'))
```

`kmc_model.save` is loaded and plugged into a classifier called `kmeans`.

`x_test` same test data as in `k-means_clustering_1.py`:

```
#test data
x_test = [[40.0,67],[20.0,61],[90.0,90],
          [50.0,54],[20.0,80],[90.0,60]]
```

We now run and display the predictions:

```
#prediction
prediction = kmeans.predict(x_test)
```

```
print("The predictions:")
print (prediction)
```

The predictions are the same as in `k-means_clustering_1.py`:

```
The predictions:
[0 0 4 0 0 1]
```

Each prediction is an output cluster number from 0 to 5 of the corresponding coordinates used as input. For example, [40.0,67] is a part of cluster #0.

The next step is to analyze the results.

Analyzing the results

The following image shows the gain zone. The gain zone is the zone in which the distances exceed the value 80.

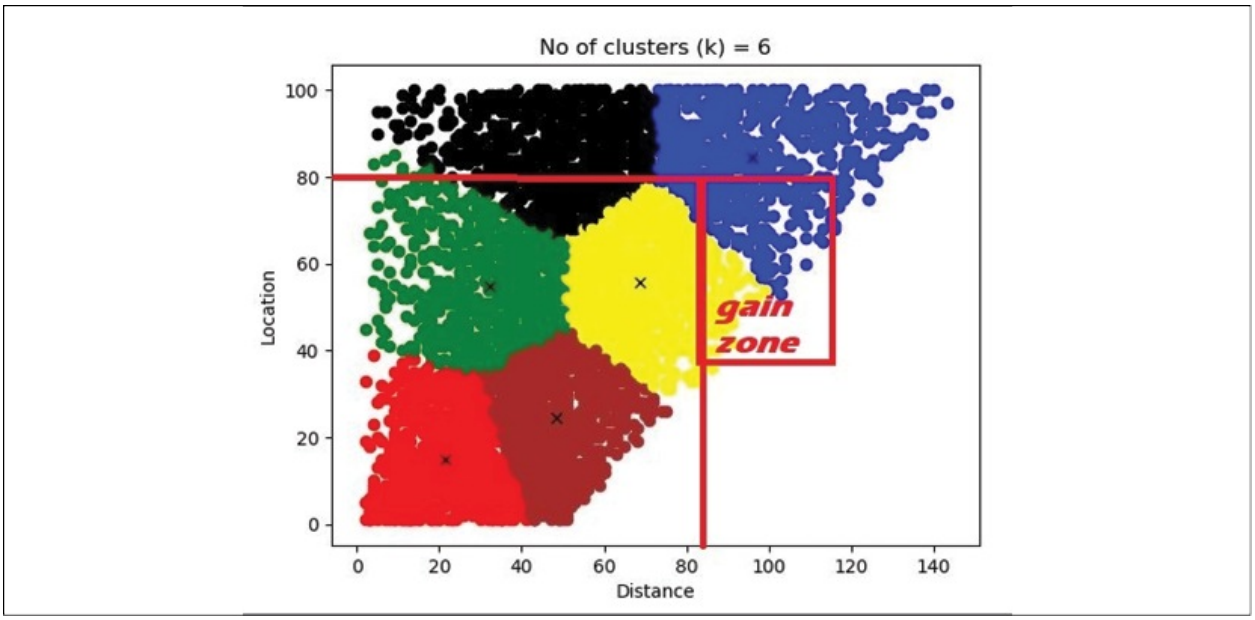


Figure 4.5: Gain zone area

The gain zone area provides useful information.

From the computations made, that gain zone shows the losses made on the locations displayed. It takes a sample of the possible locations into account. 10%

of the total distance could be saved.

The cause is that the bots are not going directly to the right locations, but are *wandering* around unplanned obstacles.

The average distance from one location to another is 1 meter. The AGVs all start from location 0 or 1. So the distance is strictly proportional to the locations in this particular example.

To find the gain zone of a location, you draw a red horizontal line from location 80, for example, and a vertical line from a distance 80 (add a couple of meters to take small variances into account).

Data analysis is made easier by data visualization. Visualizing the clusters makes it easier for management to understand the outputs and make decisions.

None of the data points on the 80-location line should be beyond the maximum limit. The limit is 80 meters + a small variance of a few meters. Beyond that line, on the right-hand side of the figure, is where the company is losing money, and something must be done to optimize the distances. This loss zone is the gain zone for a project. The gain zone on the k-means cluster results shows that some of the locations of 40 to 60 exceed a distance of 80 meters.

Bot virtual clusters as a solution

Planners anticipate bot tasks. They send them to probable locations from which they will have to pick up products and bring them back to the truck-loading points.

In the following example, we will take the example of AGVs that are assigned to a cluster area for locations 40 to 60. If an AGV goes further, up to location 70, for example, a penalty of 10 virtual (estimated) meters is added to its performance. It is easy to check. If an AGV is detected at location 70, it is out of its area.

The business rule for the AGV assigned to locations 40 to 60 is that it must never exceed location 60. If the software planning the events works well, it will never assign the AGV to an area exceeding location 60. Business rules must thus be provided to the planners.

One of the solutions is to provide AGV virtual clusters as a business rule, as shown in the following screenshot:

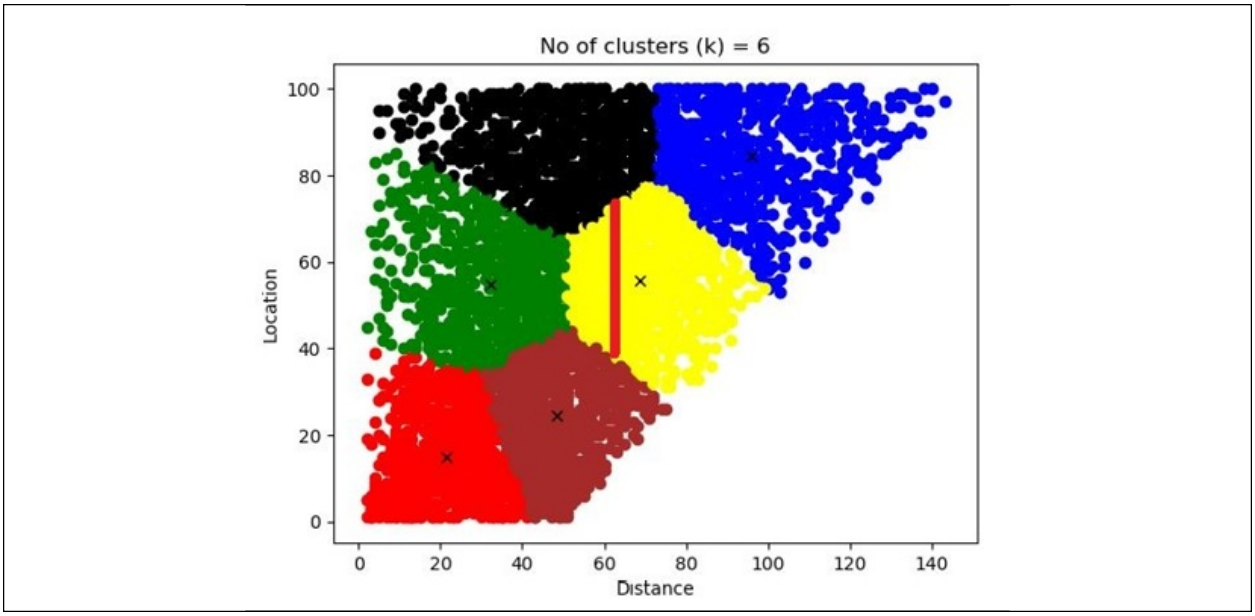


Figure 4.6: AGV virtual clusters

The rules are as follows:

- **Rule 1**: The line in the middle represents a new business rule. In phase 1 of the project, an AGV used for locations 40 to 60 cannot go beyond 60 meters plus a small variance line.
- **Rule 2**: A cluster will represent the pick-up zone for an AGV. The centroid will now be its parking zone. Distances will be optimized until all the clusters respect rule 1. If rule 1 is not followed, the AGVs will travel unnecessary distances, increasing the overall cost of transporting goods in the warehouse.

The limits of the implementation of the k-means clustering algorithm

In this chapter, an example was explored. When the volumes increase, the features reach high-level abstract representations, and noise pollutes the data, humans face several problems:

- How do we analyze a result that surpasses human analytical capacity?

- Is the algorithm reliable for larger datasets that may contain features that were overlooked?

In *Chapter 5*, *How to Use Decision Trees to Enhance k-Means Clustering*, we will explore these problems and find solutions.

Summary

Up to this point, we have explored Python with the NumPy, TensorFlow, scikit-learn, pandas, and Matplotlib libraries. More platforms and libraries will be used in this book. In the months and years to come, even more languages, libraries, frameworks, and platforms will appear on the market.

However, AI is not only about development techniques. Building a k-means clustering program from scratch requires careful planning. The program relies on data that is rarely available as we expect it. That's where our imagination comes in handy to find the right features for our datasets.

Once the dataset has been defined, poor conditioning can compromise the project. Some small changes in the data will lead to incorrect results.

Preparing the training dataset from scratch takes much more time than we would initially expect. AI was designed to make life easier, but that's after a project has been successfully implemented. The problem is that building a solution requires major dataset work and constant surveillance.

Then comes the hard work of programming a k-means clustering solution that must be explained to a team. Lloyd's algorithm comes in very handy to that effect by reducing development time.

In the next chapter, *When and How to Use Artificial Intelligence*, we will seek solutions to the limits of k-means clustering problems through dataset techniques. We will also explore random forests and enter the world of ensemble meta-algorithms, which will provide assisted AI to humans to analyze machine thinking.

Questions

1. Can a prototype be built with random data in corporate environments? (Yes | No)
2. Do design matrices contain one example per matrix? (Yes | No)
3. AGVs can never be widespread. (Yes | No)
4. Can k-means clustering be applied to drone traffic? (Yes | No)
5. Can k-means clustering be applied to forecasting? (Yes | No)
6. Lloyd's algorithm is a two-step approach. (Yes | No)
7. Do hyperparameters control the behavior of the algorithm? (Yes | No)
8. Once a program works, the way it is presented does not matter. (Yes | No)
9. k-means clustering is only a classification algorithm. It's not a prediction algorithm. (Yes | No)

Further reading

- The scikit-learn website contains additional information on k-means clustering: http://scikitlearn.org/stable/modules/generated/sklearn.cluster.KMeans.html
- You can find Python's data analysis library here: https://pandas.pydata.org/

5

How to Use Decision Trees to Enhance K-Means Clustering

This chapter addresses two critical issues. First, we will explore how to implement k-means clustering with dataset volumes that exceed the capacity of the given algorithm. Second, we will implement decision trees that verify the results of an ML algorithm that surpasses human analytic capacity. We will also explore the use of random forests.

When facing such difficult problems, choosing the right model for the task often proves to be the most difficult task in ML. When we are given an unfamiliar set of features to represent, it can be a somewhat puzzling prospect. Then we have to get our hands dirty and try different models. An efficient estimator requires good datasets, which might change the course of the project.

This chapter builds on the k-means clustering (or KMC) program developed in *Chapter 4*, *Optimizing Your Solutions with K-Means Clustering*. The issue of large datasets is addressed. This exploration will lead us into the world of the law of large numbers (LLN), the central limit theorem (CLT), the Monte Carlo estimator, decision trees, and random forests.

Human intervention in a process such as the one described in this chapter is not only unnecessary, but impossible. Not only does machine intelligence surpass humans in many cases, but the complexity of a given problem itself often surpasses human ability, due to the complex and ever-changing nature of real-life systems. Thanks to machine intelligence, humans can deal with increasing amounts of data that would otherwise be impossible to manage.

With our toolkit, we will build a solution to analyze the results of an algorithm without human intervention.

This chapter covers the following topics:

- Unsupervised learning with KMC
- Determining if AI must or must not be used
- Data volume issues
- Defining the NP-hard characteristic of KMC
- Random sampling concerning LLN, CLT, and the Monte Carlo estimator
- Shuffling a training dataset
- Supervised learning with decision trees and random forests
- Chaining KMC to decision trees

This chapter begins with unsupervised learning with KMC. We will explore methods to avoid running large datasets through random sampling. The output of the KMC algorithm will provide the labels for the supervised decision tree algorithm. The decision tree will verify the results of the KMC process, a task no human could do with large volumes of data.

> All the Python programs and files in this chapter are available at https://github.com/PacktPublishing/Artificial-Intelligence-By-Example-Second-Edition/tree/master/CH05.
>
> There is also a Jupyter notebook named `COLAB_CH05.ipynb` that contains all of the Python programs in one run. You can upload it directly to Google Colaboratory (https://colab.research.google.com/) using your Google Account.

Unsupervised learning with KMC with large datasets

KMC takes unlabeled data and forms clusters of data points. The names (integers) of these clusters provide a basis to then run a supervised learning algorithm such as a decision tree.

In this section, we will see how to use KMC with large datasets.

When facing a project with large unlabeled datasets, the first step consists of evaluating if machine learning will be feasible or not. Trying to avoid AI in a book on AI may seem paradoxical. However, in AI, as in real life, you should use the right tools at the right time. If AI is not necessary to solve a problem, do not use it.

Use a **proof of concept** (**POC**) approach to see if a given AI project is possible or not. A POC costs much less than the project itself and helps to build a team that believes in the outcome. Or, the POC might show that it is too risky to go forward with an ML solution. Intractable problems exist. It's best to avoid spending months on something that will not work.

The first step is exploring the data volume and the ML estimator model that will be used.

If the POC proves that a particular ML algorithm will solve the problem at hand, the next thing to do is to address data volume issues. The POC shows that the model works on a sample dataset. Now, the implementation process can begin.

Anybody who has run a machine learning algorithm with a large dataset on a laptop knows that it takes some time for a machine learning program to train and test these samples. A machine learning program or a deep learning convolutional neural network consumes a large amount of machine power. Even if you run an ANN using a **GPU** (short for **graphics processing unit**) hoping to get better performance than with CPUs, it still takes a lot of time for the training process to run through all the learning epochs. An epoch means that we have tried one set of weights, for example, to measure the accuracy of the result. If the accuracy is not sufficient, we run another epoch, trying other weights until the accuracy is sufficient.

If you go on and you want to train your program on datasets exceeding 1,000,000 data points, for example, you will consume significant local machine power.

Suppose you need to use a KMC algorithm in a corporation with hundreds of millions to billions of records of data coming from multiple SQL Server instances, Oracle databases, and a big data source. For example, suppose that you are working for the phone operating activity of a leading phone company. You must apply a KMC program to the duration of phone calls in locations

around the world over a year. That represents millions of records per day, adding up to billions of records in a year.

A machine learning KMC training program running billions of records will consume too much CPU/GPU and take too much time even if it works. On top of that, a billion records might only represent an insufficient amount of features. Adding more features will dramatically increase the size of such a dataset.

The question now is to find out if KMC will even work with a dataset this size. A KMC problem is **NP-hard**. The **P** stands for **polynomial** and the **N** for **non-deterministic**.

The solution to our volume problem requires some theoretical considerations. We need to identify the difficulty of the problems we are facing.

Identifying the difficulty of the problem

We first need to understand what level of difficulty we are dealing with. One of the concepts that come in handy is NP-hard.

NP-hard – the meaning of P

The P in NP-hard means that the time to solve or verify the solution of a P problem is polynomial (poly=many, nomial=terms). For example, x^3 is a polynomial. The N means that the problem is non-deterministic.

Once x is known, then x^3 will be computed. For x = 3,000,000,000 and only 3 elementary calculations, this adds up to:

$$\log x^3 = 28.43$$

It will take $10^{28.43}$ calculations to compute this particular problem.

It seems scary, but it isn't that scary for two reasons:

- In the world of big data, the number can be subject to large-scale randomized sampling.

- KMC can be trained in mini-batches (subsets of the dataset) to speed up computations.

Polynomial time means that this time will be more or less proportional to the size of the input. Even if the time it takes to train the KMC algorithm remains a bit fuzzy, as long as the time it will take to verify the solution remains proportional thanks to the batch size of the input, the problem remains a polynomial.

An exponential algorithm increases with the amount of data, not the number of calculations. An exponential function of this example would be $f(x) = 3^x = 3^{3,000,000,000}$ calculations. Such functions can often be broken down into multiple classical algorithms. Functions of this type exist in the corporate world, but they are out of the scope of this book.

NP-hard – the meaning of non-deterministic

Non-deterministic problems require a heuristic approach, which implies some form of heuristics, such as a trial and error approach. We try a set of weights, for example, evaluate the result, and then go on until we find a satisfactory solution.

The meaning of hard

NP-hard can be transposed into an NP problem with some optimization. This means that NP-hard is as hard or harder than an NP problem.

For example, we can use batches to control the size of the input, the calculation time, and the size of the outputs. That way, we can bring an NP-hard problem down to an NP problem.

One way of creating batches to avoid running an algorithm on a dataset that will prove too large for it is to use random sampling.

Implementing random sampling with mini-batches

A large portion of machine learning and deep learning contains random sampling

in various forms. In this case, a training set of billions of elements will prove difficult, if not impossible, to implement without random sampling.

Random sampling is used in many methods: Monte Carlo, stochastic gradient descent, random forests, and many algorithms. No matter what name the sampling takes, they share common concepts to various degrees, depending on the size of the dataset.

Random sampling on large datasets can produce good results, but it requires relying on the LLN, which we will explore in the next section.

Using the LLN

In probability, the LLN states that when dealing with very large volumes of data, significant samples can be effective enough to represent the whole dataset. For example, we are all familiar with polls that use small datasets.

This principle, like all principles, has its merits and limits. But whatever the limitations, this law applies to everyday machine learning algorithms.

In machine learning, sampling resembles polling. A smaller number of individuals represent a larger overall dataset.

Sampling mini-batches and averaging them can prove as efficient as calculating the whole dataset as long as a scientific method is applied:

- Training with mini-batches or subsets of data
- Using an estimator in one form or another to measure the progression of the training session until a goal has been reached

You may be surprised to read "until a goal has been reached" and not "until the optimal solution has been reached."

The optimal solution may not represent the best solution. All the features and all the parameters are often not expressed. Finding a good solution will often be enough to classify or predict efficiently.

The LLN explains why random functions are widely used in machine learning

and deep learning. Random samples provide efficient results if they respect the CLT.

The CLT

The LLN applied to the example of the KMC project must provide a reasonable set of centroids using random sampling. If the centroids are correct, then the random sample is reliable.

> A centroid is the geometrical center of a set of datasets, as explained in *Chapter 4*, *Optimizing Your Solutions with K-Means Clustering*.

This approach can now be extended to the CLT, which states that when training a large dataset, a subset of mini-batch samples can be sufficient. The following two conditions define the main properties of the CLT:

- The variance between the data points of the subset (mini-batch) remains reasonable.
- The normal distribution pattern with mini-batch variances remains close to the variance of the whole dataset.

A Monte Carlo estimator, for example, can provide a good basis to see if the samples respect the CLT.

Using a Monte Carlo estimator

The name Monte Carlo comes from the casinos in Monte Carlo and gambling. Gambling represents an excellent memoryless random example. No matter what happens before a gambler plays, prior knowledge provides no insight. For example, the gambler plays 10 games, losing some and winning some, creating a distribution of probabilities.

The sum of the distribution of $f(x)$ is computed. Then random samples are extracted from a dataset, for example, x_1, x_2, x_3,..., x_n.

$f(x)$ can be estimated through the following equation:

$$\hat{e} = \frac{1}{n}\sum_{i=1}^{n} f(x_i)$$

The estimator $\hat{e}$ represents the average of the result of the predictions of a KMC algorithm or any implemented model.

We have seen that a sample of a dataset can represent a full dataset, just as a group of people can represent a population when polling for elections, for example.

Knowing that we can safely use random samples, just like in polling a population for elections, we can now process a full large dataset directly, or preferably with random samples.

Trying to train the full training dataset

In *Chapter 4*, *Optimizing Your Solutions with K-Means Clustering*, a KMC algorithm with a six-cluster configuration produced six centroids (geometric centers), as shown here:

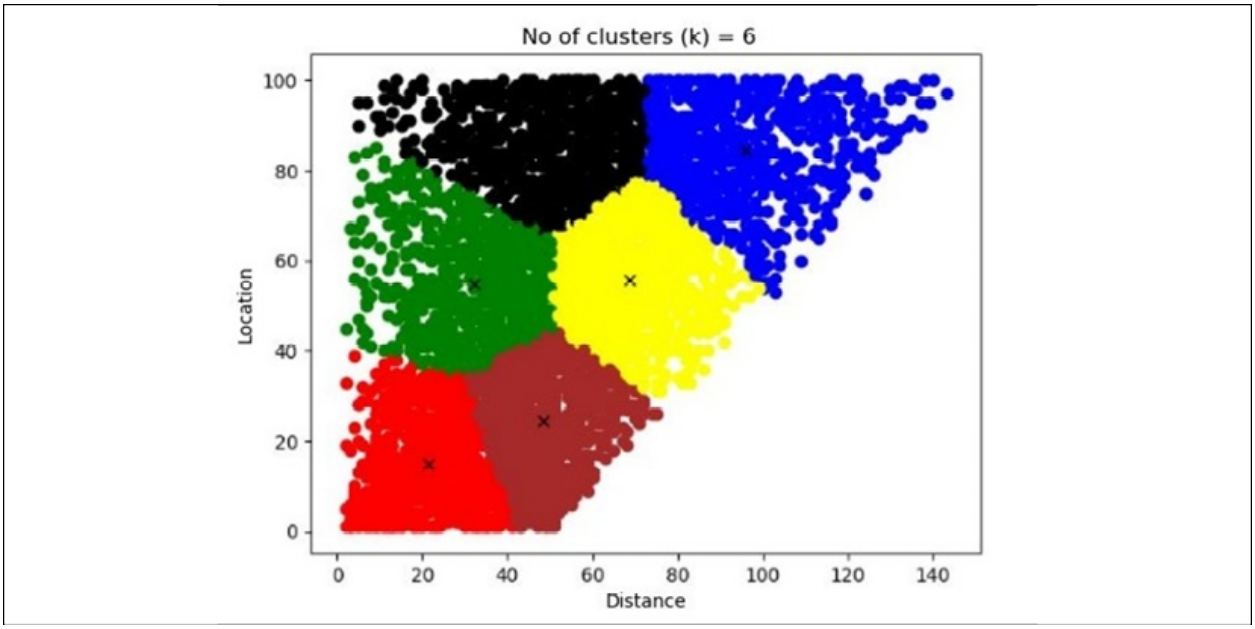


Figure 5.1: Six centroids

The problem is now how to avoid costly machine resources to train this KMC dataset when dealing with large datasets. The solution is to take random samples from the dataset in the same way polling is done on a population for elections, for example.

Training a random sample of the training dataset

The `sampling/k-means_clustering_minibatch.py` program provides a way to verify the mini-batch solution.

The program begins by loading the data in the following lines of code:

```
dataset = pd.read_csv('data.csv')
print (dataset.head())
print(dataset)
```

Loading the dataset might create two problems:

- **The dataset is too large to be loaded in the first place.** In this case, load the datasets batch by batch. Using this method, you can test the model on many batches to fine-tune your solution.
- **The dataset can be loaded, but the KMC algorithm chosen cannot absorb the volume of data.** A good choice for the size of the mini-batch will solve this problem.

Once a dataset has been loaded, the program will start the training process.

A mini-batch dataset called `dataset1` will be randomly created using Monte Carlo's large data volume principle with a mini-batch size of 1,000. Many variations of the Monte Carlo method apply to machine learning. For our example, it will be enough to use a random function to create the mini-batch:

```
n=1000
dataset1=np.zeros(shape=(n,2))
for i in range (n):
    j=randint(0,4998)
    dataset1[i][0]=dataset.iloc[j,0]
    dataset1[i][1]=dataset.iloc[j,1]
```

Finally, the KMC algorithm runs on a standard basis, as shown in the following snippet:

```
#II.Hyperparameters
# Features = 2 :implicit through the shape of the dataset (
2 columns)
k = 6
kmeans = KMeans(n_clusters=k)
#III.K-means clustering algorithm
kmeans = kmeans.fit(dataset1) #Computing k-means clustering
gcenters = kmeans.cluster_centers_ # the geometric centers
or centroids
print("The geometric centers or centroids:")
print(gcenters)
```

The following screenshot, displaying the result produced, resembles the full dataset trained by KMC in *Chapter 4*, *Optimizing Your Solutions with K-Means Clustering*:

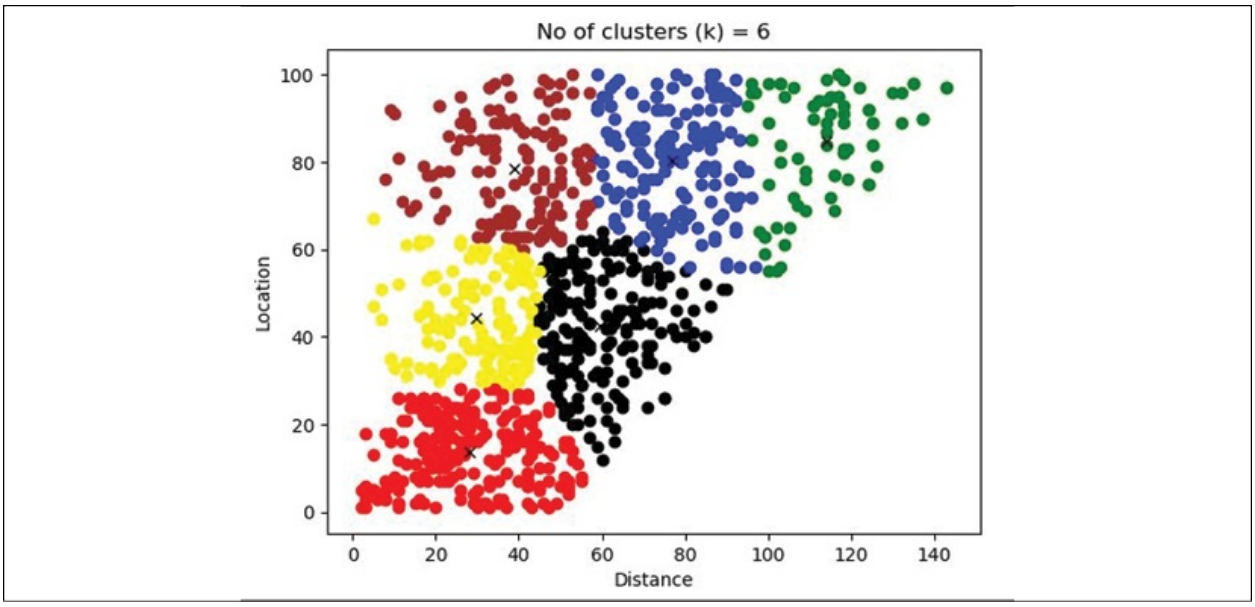


Figure 5.2: Output (KMC)

The centroids obtained are consistent, as shown here:

```
The geometric centers or centroids:
[[ 19.6626506 14.37349398]
 [ 49.86619718 86.54225352]
 [ 65.39306358 54.34104046]
```

```
 [ 29.69798658 54.7852349 ]
 [ 48.77202073 23.74611399]
 [ 96.14124294 82.44067797]]
```

The output will vary slightly at each run since this is a stochastic process. In this section, we broke the dataset down into random samples to optimize the training process. Another way to perform random sampling is to shuffle the dataset before training it.

Shuffling as another way to perform random sampling

The `sampling/k-means_clustering_minibatch_shuffling.py` program provides another way to solve a random sampling approach.

KMC is an unsupervised training algorithm. As such, it trains *unlabeled* data. A single random computation does not consume a large amount of machine resources, but several random selections in a row can.

Shuffling can reduce machine consumption costs. Proper shuffling of the data before starting training, just like shuffling cards before a poker game, will avoid repetitive and random mini-batch computations. In this model, the loading data phase and training phase do not change. However, instead of one or several random choices for `dataset1`, the mini-batch dataset, we shuffle the complete dataset once before starting the training. The following code shows how to shuffle datasets:

```
sn=4999
shuffled_dataset=np.zeros(shape=(sn,2))
for i in range (sn):
    shuffled_dataset[i][0]=dataset.iloc[i,0]
    shuffled_dataset[i][1]=dataset.iloc[i,1]
```

Then we select the first 1,000 shuffled records for training, as shown in the following code snippet:

```
n=1000
dataset1=np.zeros(shape=(n,2))
```

```
for i in range (n):
    dataset1[i][0]=shuffled_dataset[i,0]
    dataset1[i][1]=shuffled_dataset[i,1]
```

The result in the following screenshot corresponds to the one with the full dataset and the random mini-batch dataset sample:

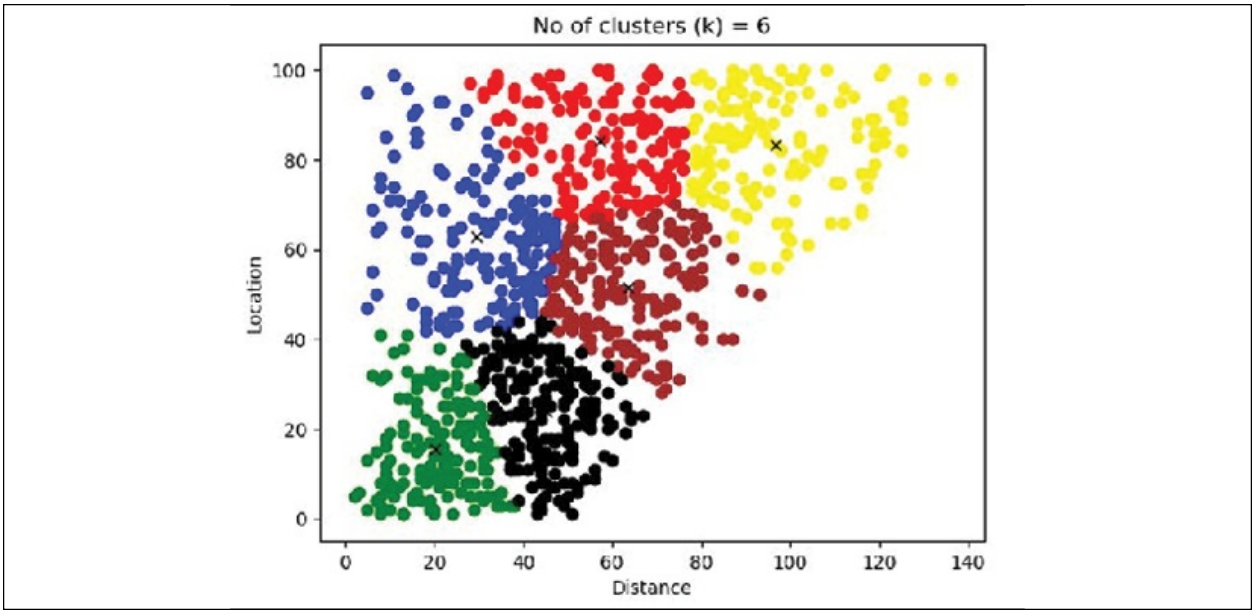


Figure 5.3: Full and random mini-batch dataset sample

The centroids produced can provide first-level results to confirm the model, as shown in the following output.

The geometric centers or centroids:

```
[[ 29.51298701 62.77922078]
 [ 57.07894737 84.21052632]
 [ 20.34337349 15.48795181]
 [ 45.19900498 23.95024876]
 [ 96.72262774 83.27737226]
 [ 63.54210526 51.53157895]]
```

Using shuffling instead of random mini-batches has two advantages: limiting the number of mini-batch calculations and preventing training the same samples twice. If your shuffling algorithm works, you will only need to shuffle the dataset once. If not, you might have to go back and use random sampling, as explained in the previous section.

Random sampling and shuffling helped to solve one part of the dataset volume problem.

However, now we must explore the other aspect of the implementation of a large dataset ML algorithm: verifying the results.

Chaining supervised learning to verify unsupervised learning

This section explores how to verify the output of an unsupervised KMC algorithm with a supervised algorithm: a decision tree.

KMC takes an input with no labels and produces an output with labels. The unsupervised process makes sense out of the chaos of incoming data.

The example in this chapter focuses on two related features: location and distance. The clusters produced are location-distance subsets of data within the dataset. The input file contains two columns: distance and location. The output file contains three columns: distance, location, and a label (cluster number).

The output file can thus be chained to a supervised learning algorithm, such as a decision tree. The decision tree will use the labeled data to produce a visual, white-box, machine thought process. Also, a decision tree can be trained to verify the results of the KMC algorithm. The process starts with preprocessing raw data.

Preprocessing raw data

Earlier, it was shown that with large datasets, mini-batches will be necessary. Loading billions of records of data in memory is not an option. A random selection was applied in `sampling/k-means_clustering_minibatch.py` as part of the KMC algorithm.

However, since we are chaining our algorithms in a pipeline and since we are not training the model, we could take the random sampling function from `sampling/k-means_clustering_minibatch.py` and isolate it:

```
n=1000
dataset1=np.zeros(shape=(n,2))
li=0
for i in range (n):
    j=randint(0,4999)
    dataset1[li][0]=dataset.iloc[j,0]
    dataset1[li][1]=dataset.iloc[j,1]
    li+=1
```

The code could be applied to datasets extracted in a preprocessing phase from packs of data retrieved from a big data environment, for example. The preprocessing phase would be repeated in cycles. We will now explore the pipeline that goes from raw data to the output of the chained ML algorithms.

A pipeline of scripts and ML algorithms

An ML **pipeline** will take raw data and perform dimension reduction or other preprocessing tasks that are not in the scope of this book. Preprocessing the data sometimes requires more than ML algorithms such as SQL scripts. Our exploration starts right after when ML algorithms such as KMC take over. However, a pipeline can run from raw data to ML using classical non-AI scripting as well.

The pipeline described in the following sections can be broken down into three major steps, preceded by classical preprocessing scripting:

- **Step 0**: A standard process performs a random sampling of a **training dataset** with classical preprocessing scripts before running the KMC program. This aspect is out of the scope of the ML process and this book. By doing this, we avoid overloading the ML Python programs. The training data will first be processed by a KMC algorithm and sent to the decision tree program.
- **Step 1**: A KMC multi-ML program, `kmc2dt_chaining.py`, reads the training dataset produced by step 0 using a **saved model** from *Chapter 4*, *Optimizing Your Solutions with K-Means Clustering*. The KMC program takes the unlabeled data, makes predictions, and produces a labeled output in a file called `ckmc.csv`. The output label is the cluster number of a line of the dataset containing a distance and location.

- **Step 2**: The decision tree program, `decision_tree.py`, reads the output of the KMC predictions, `ckmc.csv`. The decision tree algorithm trains its model and saves the trained model in a file called `dt.sav`.
    - **Step 2.1**: The training phase is over. The pipeline now takes raw data retrieved by successive equal-sized datasets. This **batch** process will provide a fixed amount of data. Calculation time can be planned and mastered. This step is out of the scope of the ML process and this book.
    - **Step 2.2**: A random sampling script processes the batch and produces a prediction dataset for the **predictions**.
- **Step 3**: `kmc2dt_chaining.py` will now run a KMC algorithm that is chained to a decision tree that will verify the results of the KMC. The KMC algorithm produces predictions. The decision tree makes predictions on those predictions. The decision tree will also provide a visual graph in a PNG for users and administrators.

Steps 2.1 to 3 can run on a twenty-four seven basis in a continuous process.

It is important to note that random forests are an interesting alternative to the decision tree component. It can replace the decision tree algorithm in `kmc2dt_chaining.py`. In the next section, we will explore random forests in this context with `random_forest.py`.

Step 1 – training and exporting data from an unsupervised ML algorithm

`kmc2dt_chaining.py` can be considered as the **chaining** of a KMC program to a decision tree program that will verify the results. Each program forms a link of the chain.

From a decision tree project's perspective, `kmc2dt_chaining.py` can be considered as a **pipeline** taking unlabeled data and labeling it for the supervised decision tree program. A pipeline takes raw data and transforms it using more than one ML program.

During the training phase of the chained model, `kmc2dt_chaining.py` runs to provide datasets for the training of the decision tree. The parameter `adt=0` limits the process to the KMC function, which is the first link of the chain.

The decision tree in this program will thus not be activated in this phase.

`kmc2dt_chaining.py` will load the dataset, load the saved KMC mode, make predictions, and export the labeled result:

- **Load the dataset**: `data.csv`, the dataset file, is the same one used in *Chapter 4, Optimizing Your Solutions with K-Means Clustering*. The two features, `location` and `distance`, are loaded:

```
dataset = pd.read_csv('data.csv')
```

- **Load the KMC model**: The k-means cluster model, `kmc_model.sav`, was saved by `k-means_clustering_2.py` in *Chapter 4, Optimizing Your Solutions with K-Means Clustering*. It is now loaded using the `pickle` module to save it:

```
kmeans = pickle.load(open('kmc_model.sav', 'rb'))
```

- **Make predictions**: No further training of the KMC model is required at this point. The model can run predictions on the mini-batches it receives. We can use an **incremental** process to verify the results on a large scale.

  If the data is not sufficiently scaled, other algorithms could be applied. In this case, the dataset does not require additional scaling. The KMC algorithm will make predictions on the sample and produce an output file for the decision tree.

  For this example, the predictions will be generated line by line:

```
    for i in range(0,1000):
        xf1=dataset.at[i,'Distance']
        xf2=dataset.at[i,'location']
        X_DL = [[xf1,xf2]]
        prediction = kmeans.predict(X_DL)
```

  The results are stored in a NumPy array:

```
        p=str(prediction).strip('[]')
        p=int(p)
        kmcpred[i][0]=int(xf1)
        kmcpred[i][1]=int(xf2)
        kmcpred[i][2]=p;
```

- **Export the labeled data**: The predictions are saved in a file:

```
np.savetxt('ckmc.csv', kmcpred, delimiter=',', fmt='%d'
)
```

  This output file is special; it is now labeled:

```
80,53,5
18,8,2
55,38,0
```

  The output file contains three columns:

  - Column 1 = feature 1 = location; `80`, for example, on the first line
  - Column 2 = feature 2 = distance; `53`, for example, on the first line
  - Column 3 = label = cluster calculated; `5`, for example, on the first line

  In our chained ML algorithms, this output data will become the input data of the next ML algorithm, the decision tree.

Step 2 – training a decision tree

In *Chapter 3, Machine Intelligence – Evaluation Functions and Numerical Convergence*, a decision tree was described and used to visualize a priority process. `Decision_Tree_Priority.py` produced the following graph:

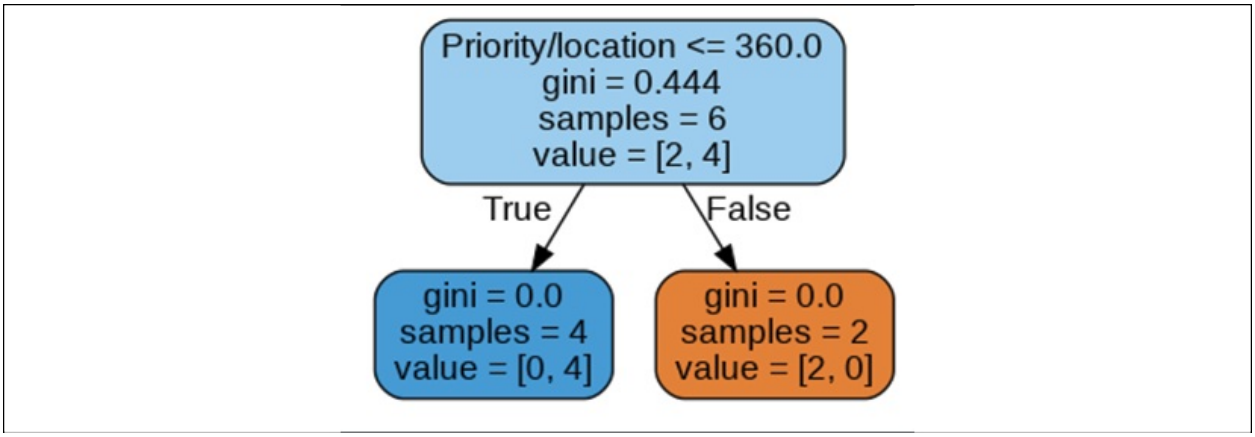


Figure 5.4: Decision tree priorities

The tree starts with a node with a high Gini value. The node is split into two, and

each node below is a "leaf" in this case because Gini=0.

The decision tree algorithm implemented in this book uses Gini impurity.

Gini impurity represents the probability of a data point being incorrectly classified.

A decision tree will start with the highest impurities. It will split the probability into two branches after having calculated a threshold.

When a branch reaches a Gini impurity of 0, it reaches its **leaf**.

Let's state that *k* is the probability of a data point being incorrectly classified.

The dataset *X* from *Chapter 3, Machine Intelligence – Evaluation Functions and Numerical Convergence,* contains six data points. Four data points are low, and two data points are high:

$$X = \{\text{Low, Low, High, High, Low, Low}\}$$

The equation of Gini impurity calculates the probability of each feature occurring and multiplies the result by 1—the probability of each feature occurring on the remaining values—as shown in the following equation:

$$G(k) = \sum_{i=1}^{i=n} P_i * (1 - P_i)$$

Applied to the *X* dataset with four lows out of six and two highs out of six, the result will be:

$$G(k) = (4/6) * (1 – 4/6) + (2/6) * (1 – 2/6)$$

$$G(k)=(0.66 * 0.33) + (0.33 * 0.66)$$

$$G(k)=0.222 + 0.222=0.444$$

The probability that a data point will be incorrectly predicted is 0.444, as shown in the graph.

The decision train is built on **the gain of information** on the features that contain the highest Gini impurity value.

We will now explore the Python implementation of a decision tree to prepare it to be chained to the KMC program.

Training the decision tree

To train the decision tree, `decision_tree.py` will load the dataset, train the model, make predictions, and save the model:

- **Load the dataset**: Before loading the dataset, you will need to import the following modules:

```
import pandas as pd #data processing
from sklearn.tree import DecisionTreeClassifier #the dt
 classifier
from sklearn.model_selection import train_test_split #s
plit the data into training data and testing data
from sklearn import metrics #measure prediction perform
ance
import pickle #save and load estimator models
```

> The versions of the modules will vary as the editors produce them and also depend on how often you update the versions and the code. For example, you might get a warning when you try to unpickle a KMC model from version 0.20.3 when using version 0.21.2. As long as it works, it is fine for educational purposes. However, in production, an administrator should have a database with the list of packages used and their versions.

The dataset is loaded, labeled, and split into training and test datasets:

```
#loading dataset
col_names = ['f1', 'f2','label']
df = pd.read_csv("ckmc.csv", header=None, names=col_names)
print(df.head())
#defining features and label (classes)
feature_cols = ['f1', 'f2']
X = df[feature_cols] # Features
y = df.label # Target variable
print(X)
print(y)
```

```
# splitting df (dataset) into training and testing data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=1) # 70% training and
 30% test
```

The data in this example will contain two features (location and distance) and a label (cluster number) provided by the output of the KMC algorithm. The following header shows how the dataset is structured:

```
   f1  f2  label
0  80  53      5
1  18   8      2
2  55  38      0
3  74  74      5
4  17   4      2
```

- **Training the model**: Once the datasets are ready, the decision tree classifier is created and the model is trained:

```
# create the decision tree classifier
dtc = DecisionTreeClassifier()
# train the decision tree
dtc = dtc.fit(X_train,y_train)
```

- **Making predictions**: Once the model is trained, predictions are made on the test dataset:

```
#predictions on X_test
print("prediction")
y_pred = dtc.predict(X_test)
print(y_pred)
```

  The predictions use the features to predict a cluster number in the test dataset, as shown in the following output:

```
prediction
[4 2 0 5 0...]
```

- **Measuring the results with metrics**: A key part of the process is to measure the results with metrics. If the accuracy approaches 1, then the KMC output chained to the decision tree algorithm is reliable:

```
# model accuracy
print("Accuracy:",metrics.accuracy_score(y_test, y_pred
```

```
))
```

In this example, the accuracy was 0.97. The model can predict the cluster of a distance and location with a 0.97 probability, which proves its efficiency. The training of the chained ML solution is over, and a prediction cycle begins.

You can generate a PNG file of the decision tree with `decision_tree.py`. Uncomment the last paragraph of the program, which contains the export function:

```
from sklearn import tree
import pydotplus
graph=1
if(graph==1):
    # Creating the graph and exporting it
    dot_data = tree.export_graphviz(dtc, out_file=None,
                                    filled=True, rounded=Tr
ue,
                                    feature_names=feature_c
ols,
                                    class_names=['0','1','2'
,
                                                 '3','4','5'
])
    #creating graph
    graph = pydotplus.graph_from_dot_data(dot_data)
    #save graph
    image=graph.create_png()
    graph.write_png("kmc_dt.png")
```

Note that once you have implemented this function, you can activate or deactivate it with the `graph` parameter.

The following image produced for this example can help you understand the thought process of the whole chained solution (KMC and the decision tree).

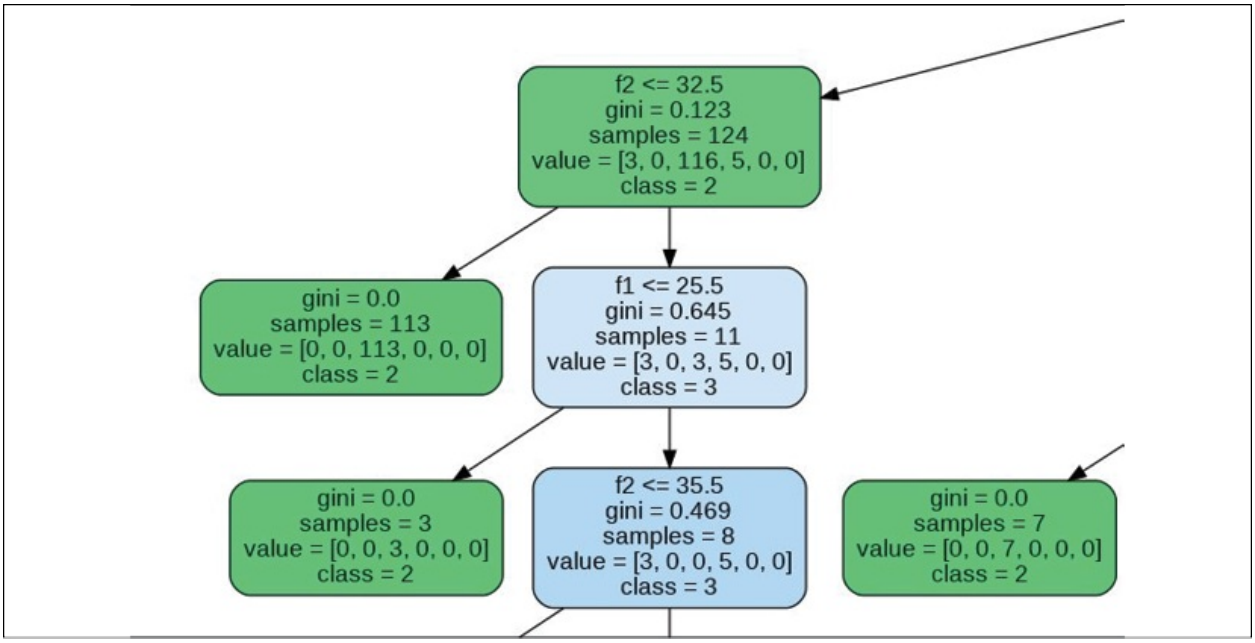


Figure 5.5: Image output from the code example

The image file, `dt_kmc.png`, is available on GitHub in `CH05`.

Step 3 – a continuous cycle of KMC chained to a decision tree

The training of the chained KMC algorithm chained to a decision tree algorithm is over.

The preprocessing phase using classical big data batch retrieval methods will continuously provide randomly sampled datasets with a script.

`kmc2dt_chaining.py` can focus on running KMC predictions and passing them on to decision tree predictions for white-box checking. The continuous process imports a dataset, loads saved models, predicts, and measures the results at the decision tree level.

The chained process can run on a twenty-four seven basis if necessary.

The modules used are required for both the KMC and the decision trees implemented in the training programs:

```
from sklearn.cluster import KMeans
```

```
import pandas as pd
from matplotlib import pyplot as plt
import pickle
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics
```

Let's run through the process:

- **Loading the KMC dataset and model**: The KMC dataset has been prepared in the preprocessing phase. The trained model has been previously saved. The dataset is loaded with pandas and the model is loaded with `pickle`:

```
#I.KMC. The prediction dataset and model
dataset = pd.read_csv('data.csv')
kmeans = pickle.load(open('kmc_model.sav', 'rb'))
```

- **Predicting and saving**: The goal is to predict the batch dataset line by line and save the result in an array in a white-box approach that can be verified by the administrator of the system:

```
    for i in range(0,1000):
        xf1=dataset.at[i,'Distance']
        xf2=dataset.at[i,'location']
        X_DL = [[xf1,xf2]]
        prediction = kmeans.predict(X_DL)
        #print (i+1, "The prediction for",X_DL," is:",
            str(prediction).strip('[]'))
        #print (i+1, "The prediction for",
            str(X_DL).strip('[]'), " is:",
            str(prediction).strip('[]'))
        p=str(prediction).strip('[]')
        p=int(p)
        kmcpred[i][0]=int(xf1)
        kmcpred[i][1]=int(xf2)
        kmcpred[i][2]=p
np.savetxt('ckmc.csv', kmcpred, delimiter=',', fmt='%d'
)
```

  The `ckmc.csv` file generated is the entry point of the next link of the chain: the decision tree.

  The two lines containing the `print` instruction are commented for standard

runs. However, you may wish to explore the outputs in detail if your code requires maintenance. That is why I recommend adding maintenance lines in the code.

- **Loading the dataset for the decision tree**: The dataset for the decision tree is loaded in the same way as in `decision_tree.py`. A parameter activates the decision tree part of the code: `adt=1`. The white-box quality control approach can thus be activated or deactivated.

  The program loads the dataset, loads the model, and splits the data:

```
if adt==1:
    #I.DT. The prediction dataset and model
    col_names = ['f1', 'f2','label']
    # load dataset
    ds = pd.read_csv('ckmc.csv', header=None,
                     names=col_names)
    #split dataset in features and target variable
    feature_cols = ['f1', 'f2']
    X = ds[feature_cols] # Features
    y = ds.label # Target variable
    # Split dataset into training set and test set
    X_train, X_test, y_train, y_test = train_test_split
(
        X, y, test_size=0.3, random_state=1) # 70% trai
ning and 30% test
    # Load model
    dt = pickle.load(open('dt.sav', 'rb'))
```

  Although the dataset has been split, only the test data is used to verify the predictions of the decision tree and the quality of the KMC outputs.

- **Predicting and measuring the results**: The decision tree predictions are made and the accuracy of the results is measured:

```
    #Predict the response for the test dataset
    y_pred = dt.predict(X_test)
    # Model Accuracy
    acc=metrics.accuracy_score(y_test, y_pred)
    print("Accuracy:",round(acc,3))
```

  Once again, in this example, as in `decision_tree.py`, the accuracy is 0.97.

- **Double-checking the accuracy of the predictions**: In the early days of a project or for maintenance purposes, double-checking is recommended. The function is activated or deactivated with a parameter named `doublecheck`.

  The prediction results are checked line by line against the original labels and measured:

```
#Double Check the Model's Accuracy
    doublecheck=1 #0 deactivated, 1 activated
    if doublecheck==1:
        t=0
        f=0
        for i in range(0,1000):
            xf1=ds.at[i,'f1']
            xf2=ds.at[i,'f2']
            xclass=ds.at[i,'label']
            X_DL = [[xf1,xf2]]
            prediction =clf.predict(X_DL)
            e=False
            if(prediction==xclass):
                e=True
                t+=1
            if(prediction!=xclass):
                e=False
                f+=1
            print (i+1,"The prediction for",X_DL," is:"
,
                str(prediction).strip('[]'),
                "the class is",xclass,"acc.:",e)
        print("true:", t, "false", f, "accuracy",
            round(t/(t+f),3))
```

  The program will print out the predictions line by line, stating if they are `True` or `False`:

```
995 The prediction for [[85, 79]]  is: 4 the class is 4
 acc.: True
996 The prediction for [[103, 100]]  is: 4 the class is
4 acc.: True
997 The prediction for [[71, 82]]  is: 5 the class is 1
 acc.: False
998 The prediction for [[50, 44]]  is: 0 the class is 0
 acc.: True
999 The prediction for [[62, 51]]  is: 5 the class is 5
```

```
acc.: True
```

In this example, the accuracy is 0.99, which is high. Only 9 out of 1,000 predictions were `False`. This result is not the same as the metrics function because it is a simple calculation that does not take mean errors or other factors into account. However, it shows that the KMC produced good results for this problem.

Decision trees provide a good approach for the KMC. However, random forests take machine learning to another level and provide an interesting alternative, if necessary, to the use of decision trees.

Random forests as an alternative to decision trees

Random forests open mind-blowing ML horizons. They are ensemble meta-algorithms. As an ensemble, they contain several decision trees. As meta-algorithms, they go beyond having one decision tree making predictions.

To run an ensemble (several decision trees) as a meta-algorithm (several algorithms training and predicting the same data), the following module is required:

```
from sklearn.ensemble import RandomForestClassifier
```

To understand how a random forest works as a meta-algorithm, let's focus on three key parameters in the following classifier:

```
clf = RandomForestClassifier(n_estimators=25, random_state=
None, bootstrap=True)
```

- `n_estimators=25`: This parameter states the number of **trees** in the **forest**. Each tree will run its prediction. The main method to reach the final prediction is obtained by averaging the predictions of each tree.

  At each split of each tree, features are randomly permuted. The trees use

different feature approaches.

- `bootstrap=True`: When bootstrap is activated, a smaller sample is bootstrapped from the sample provided. Each tree thus bootstraps its own samples, adding more variety.
- `random_state=None`: When `random_state=None` is activated, the random function uses `np.random`.

  You can also use the other methods by consulting the scikit-learn documentation (see *Further reading* at the end of the chapter). My preference is to use `np.random`. Note that to split the training state, I use scikit-learn's default random generator example with `random_state=0`.

  This shows the importance of these small changes in parameters. After many tests, this is what I preferred. But maybe, in other cases, other `random_state` values are preferable.

Beyond these key concepts and parameters, `random_forest.py` can be built in a clear, straightforward way.

`random_forest.py` is built with the same structure as the KMC or decision tree program. It loads a dataset, prepares the features and target variables, splits the dataset into training and testing sub-datasets, predicts, and measures. `random_forest.py` also contains a custom double-check function that will display each prediction, its status (`True` or `False`), and provide an independent accuracy rate.

- **Loading the dataset**: `ckmc.csv` was generated by the preceding KMC program. This time, it will be read by `random_forest.py` instead of `decision_tree.py`. The dataset is loaded, and the features and target variables are identified. Note the `pp` variable, which will trigger the `print` function or not. This is useful for switching from production mode to maintenance mode with a single variable change. Change `pp=0` to `pp=1` if you wish to switch to maintenance mode. In this case `pp` is activated:

```
pp=1 # print information
# load dataset
col_names = ['f1', 'f2','label']
df = pd.read_csv("ckmc.csv", header=None, names=col_nam
es)
if pp==1:
```

```
    print(df.head())
#loading features and label (classes)
feature_cols = ['f1', 'f2']
X = df[feature_cols] # Features
y = df.label # Target variable
if pp==1:
    print(X)
    print(y)
```

The program prints the labeled data:

```
   f1  f2  label
0  80  53      5
1  18   8      2
2  55  38      0
3  74  74      5
4  17   4      2
```

The program prints the target cluster numbers to predict:

```
[1 5 5 5 2 1 3 3…]
```

- **Splitting the data, creating the classifier, and training the model**: The dataset is split into training and testing data. The random forest classifier is created with 25 estimators. Then the model is trained:

```
#Divide the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=0)
#Creating Random Forest Classifier and training
clf = RandomForestClassifier(n_estimators=25,
    random_state=0)
clf.fit(X_train, y_train)
```

The program is ready to predict.

- **Predicting and measuring the accuracy of the trained model**:

```
#Predictions
y_pred = clf.predict(X_test)
if pp==1:
    print(y_pred)
#Metrics
ae=metrics.mean_absolute_error(y_test, y_pred)
```

```
print('Mean Absolute Error:', round(ae,3))
```

The output results and metrics are satisfactory:

```
predictions:
[1 5 5 5 2 1 3 3 0 5 3 5 3 2 2 4 3 1 3 2 2 …]
Mean Absolute Error: 0.165
```

A mean error approaching 0 is an efficient result.

- **Double-checking**: A double-checking function is recommended in the early stages of an ML project and for maintenance purposes. The function is activated by the `doublecheck` parameter as in `kmc2dt_chaining.py`. Set `doublecheck=1` if you wish to activate the maintenance mode. It is in fact, the same template:

```
#Double Check the Model's Accuracy
doublecheck=0  # 1=yes, 0=no
if doublecheck==1:
    t=0
    f=0
    for i in range(0,1000):
        xf1=df.at[i,'f1']
        xf2=df.at[i,'f2']
        xclass=df.at[i,'label']
        X_DL = [[xf1,xf2]]
        prediction =clf.predict(X_DL)
        e=False
        if(prediction==xclass):
            e=True
            t+=1
        if(prediction!=xclass):
            e=False
            f+=1
        if pp==1:
            print (i+1,"The prediction for",X_DL," is:"
,
                str(prediction).strip('[]'),"the class
is",
                xclass,"acc.:",e)
    acc=round(t/(t+f),3)
    print("true:",t,"false",f,"accuracy",acc)
    print("Absolute Error",round(1-acc,3))
```

The accuracy of the random forest is efficient. The output of the measurement is:

```
Mean Absolute Error: 0.085
true: 994 false 6 accuracy 0.994
Absolute Error 0.006
```

The absolute error is a simple arithmetic approach that does not take mean errors or other factors into account. The score will vary from one run to another because of the random nature of the algorithm. However, in this case, 6 errors out of 1,000 predictions is a good result.

Ensemble meta-algorithms such as random forests can replace the decision tree in `kmc2dt_chaining.py` with just a few lines of code, as we just saw in this section, tremendously boosting the whole chained ML process.

Chained ML algorithms using ensemble meta-algorithms are extremely powerful and efficient. In this chapter, we used a chained ML solution to deal with large datasets and perform automatic quality control on machine intelligence predictions.

Summary

Although it may seem paradoxical, try to avoid AI before jumping into a project that involves millions to billions of records of data (such as SQL, Oracle, and big data). Try simpler classical solutions like big data methods. If the AI project goes through, LLN will lead to random sampling over the datasets, thanks to CLT.

A pipeline of classical and ML processes will solve the volume problem, as well as the human analytic limit problem. The random sampling function does not need to run a mini-batch function included in the KMC program. Batches can be generated as a preprocessing phase using classical programs. These programs will produce random batches of equal size to the KMC NP-hard problem, transposing it into an NP problem.

KMC, an unsupervised training algorithm, will transform unlabeled data into a labeled data output containing a cluster number as a label.

In turn, a decision tree, chained to the KMC program, will train its model using the output of the KMC. The model will be saved just as the KMC model was saved. The random forests algorithm can replace the decision tree algorithm if it provides better results during the training phase of the pipeline.

In production mode, a chained ML program containing the KMC trained model and the decision tree trained model can make classification predictions on fixed random sampled batches. Real-time metrics will monitor the quality of the process. The chained program, being continuous, can run twenty-four seven, providing reliable real-time results without human intervention.

The next chapter explores yet another ML challenge: the increasing amount of language translations generated by global business and social communication.

Questions

1. The number of k clusters is not that important. (Yes | No)
2. Mini-batches and batches contain the same amount of data. (Yes | No)
3. K-means can run without mini-batches. (Yes | No)
4. Must centroids be optimized for result acceptance? (Yes | No)
5. It does not take long to optimize hyperparameters. (Yes | No)
6. It sometimes takes weeks to train a large dataset. (Yes | No)
7. Decision trees and random forests are unsupervised algorithms. (Yes | No)

Further reading

- Decision trees: https://scikit-learn.org/stable/modules/tree.html
- Random forests: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

6

Innovating AI with Google Translate

In this chapter, we will illustrate how to innovate existing AI through Google Translate. First, we will start by understanding the difference between inventions, innovations, disruption, high-priced products, and revolutions, and how we can create an impact on an AI innovation.

Sometimes, a developer will confuse an invention with innovation, leading to a major failure in a key AI project. Some AI designers take a revolution to mean a disruption, leading to early sales and then nothing.

Once we have defined the key concepts of innovation, we will use Google Translate to understand the linguistic principles that surround **natural language processing** (**NLP**).

Google Translate entered the market in 2006 and was enhanced by neural networks in 2016, but still, it often produces bad answers. Is that good news or bad news? We will implement Google's API in a Python program to find out how to uncover false translations from one language to another.

Once we have found Google Translate's limits, we will finally find out how to transcend those limits with our own adaptations, by exploring Google's API in a Python program, adding a **k-nearest neighbors** (**KNN**) algorithm, and measuring the results statistically.

Contrary to media hype, artificial intelligence has only just begun to innovate human processes. A huge amount of work remains to be done. To achieve progress, everybody must get involved, as we'll discuss in this chapter. Even if Google, Amazon, Microsoft, IBM, and others offer a solution, this does not mean it cannot be improved by third parties as add-ons to existing solutions or new standalone products. After all, once Ford invented the Model T over a

hundred years ago, this did not preclude the development of even better cars. On the contrary, look around you!

To take advantage of the AI adventure, we will go from understanding disruption in AI to Google Translate, and then innovate.

The following topics will be covered in this chapter:

- Understanding the key concepts of AI innovation before starting to implement Google Translate
- The difference between inventions and innovations
- The difference between revolutionary and disruptive AI
- Google Translate API implementation in Python
- Introducing linguistics as a prerequisite to building any **natural language processing** (**NLP**) algorithm
- The KNN algorithm
- How to customize Google Translate with a KNN in Python

We will start by exploring the key concepts of AI innovation and disruption.

> All the Python programs and files in this chapter are available at https://github.com/PacktPublishing/Artificial-Intelligence-By-Example-Second-Edition/tree/master/CH06.
>
> There is also a Jupyter notebook named `COLAB_Translate.ipynb` that contains all of the Python programs in one run. You can upload it directly to Google Colaboratory using your Google account: https://colab.research.google.com/.

Understanding innovation and disruption in AI

The first question we must ask ourselves when starting a project such as a translation solution is to find out where we fit in. Is what we are going to do an

invention or an innovation? Is our work disruptive or revolutionary? We will explore these concepts in this chapter to understand the broad pragmatic picture before going any further.

Is AI disruptive?

The word "disruptive" is more often than not associated with artificial intelligence. Media hype tells us that AI robots will soon have replaced humans around the world. Although media hype made it much easier to obtain AI budgets, we need to know where we stand if we want to implement an AI solution. If we want to innovate, we need to find the cutting edge and build new ideas from there.

Why not just plunge into the tools and see what happens? Is that a good idea or a risky one? Unlike corporations with huge budgets, a single human has limited resources. If you spend time learning in the wrong direction, it will take months to gather enough experience in another, better direction to reach your goal.
For example, suppose you have trouble classifying large amounts of data, as we explored in *Chapter 4, Optimizing Your Solutions with K-means Clustering*. You will spend months on a project in your company and fail. It could cost you your job. Conversely, if you find the right model and learn the key concepts, your project can take off in a few days.

Before diving into a project, find out where we stand in terms of innovation and disruption. This doesn't seem important at the start of a project, but it will mean a lot during the production and distribution phases. This section will clarify these concepts.

Nothing is new under the sun—not even when considering AI. Most AI theory and mathematical tools have been around for decades, if not centuries. We often tend to think that since something appears new to us, it has just been invented or discovered. This mistake can prove fatal in many projects. If you know that a theory or function has been around for decades or centuries, you can do some deep research and use solutions found 100+ years ago to solve your present problems. If you do, you will save a lot of time using equations that have been proven and are reliable. If you do not, you might spend useless vital time reinventing equations that exist.

Finding out what is new and what is not will make a major difference in your personal or professional AI projects.

AI is based on mathematical theories that are not new

AI theory presently relies heavily on applied mathematics. In *Chapter 1, Getting Started with Next-Generation Artifcial Intelligence through Reinforcement Learning*, the **Markov decision process** (**MDP**), a **reinforcement learning** (**RL**) approach was described. Google has been successfully combining RL with neural networks in AlphaGo Zero.

Andrey Markov was a Russian mathematician born in 1856 who invented the MDP. He successfully applied the algorithm to letter predictions in a word sequence in a given language, for example. Richard Bellman published an enhanced version of the MDP in 1957.

Bellman also coined the expression "curse of dimensionality" and published books on mathematical tools widely used today in AI. It is now well known that dimensionality reduction can be performed to avoid facing thousands of dimensions (features, for example) in an algorithm.

The logistic function (see *Chapter 2, Building a Reward Matrix – Designing Your Datasets*) can be traced back to Pierre François Verhulst (1844-1845), a Belgian mathematician. The logistic function uses *e*, the natural logarithm base, which is also named Euler's number. Leonhard Euler (1707-1783) is a Swiss mathematician who worked on this natural logarithm.

Thomas Bayes (1701-1761) invented the theorem that bears his name: Bayes' Theorem. It is widely used in AI. We will be using it in *Chapter 7, Optimizing Blockchains with Naive Bayes*.

Almost all of the applied mathematics in artificial intelligence, machine learning, and deep learning can be traced from 17th century to 20th century mathematicians. We must look elsewhere for 21st century AI innovations. We need to find what is truly new in AI, which is also what helped it expand so rapidly in the early 21st century.

Neural networks are not new

Neural networks, as described by contemporary experts, date back to the 1940s and 1950s. Even **convolutional neural networks** (**CNNs**) date back to the 20th century. Yann LeCun, a French computer scientist, laid down the basics of a CNN (see *Chapter 9, Abstract Image Classifcation with Convolutional Neural Networks (CNNs)*) in the 1980s; he successfully applied them as we know them today in the 1990s.

We must again look elsewhere for 21st century AI innovations.

If neural networks are not new either, we must find the real new factors in our environment that produced the success of present-day AI.

Looking at disruption – the factors that are making AI disruptive

Although the foundations of AI find their roots long before computers existed or were widespread, it is only recently that we have seen AI truly begin to cause waves within our society. In the following sections, we'll look at factors that have come together to make AI a powerful force of disruption in recent years.

Cloud server power, data volumes, and web sharing of the early 21st century

It's important to understand what has driven the emergency of AI in recent years.

Data volumes drive the emergence of AI in the 21st century. Processing data, classifying data, and making predictions and decisions would be impossible without AI driving the entire computer science market.

If you leave the fantasy surrounding AI behind you and bear this key necessity for AI in mind, you will perfectly understand why AI has emerged and is here to stay.

The first sign of AI's innovative disruption appeared between the years 2000 and 2010. Before then, the internet existed, and servers existed. But, starting from around 2005, cloud servers were made widely available. With that kind of computing power, developers around the world could try using the highly greedy

resources required by machine learning and deep learning. They could finally solve otherwise impossible big data problems using AI.

At the same time, as powerful servers became available, the internet provided the largest library of knowledge in the history of humanity.

On top of that, social networking became widely used. Sharing discoveries and source code became commonplace. The World Wide Web (WWW) encouraged open source software, boosting local research and development.

The era of artificial intelligence became possible for local experts starting from the middle of the first decade of the 21st century.

What makes AI appear as an innovation today is the conjunction of more powerful machines and the availability of intellectual resources.

Public awareness

Public awareness of AI remained dim for several years after the cloud architectural revolution occurred from around 1995 to 2005.

AI hit us hard by around 2015 when we all woke up realizing that AI could massively replace humans and create job displacement at levels never before seen in human history.

Worse, we realized that machines could beat us in fields we took pride in, such as chess (*Chapter 3*, *Machine Intelligence – Evaluation Functions and Numerical Convergence*), the game of Go, and video games. We see manufacturing jobs increasingly performed by robots, office jobs being done by bots, and more fields that are appearing every day.

For the first time in human history, the human species can be surpassed by a new "species": smart bots. As developers, we thought we were safe until Google presented AutoML, a solution that could create machine learning solutions better than humans. At the same time, ready-to-use machine learning platforms have spread that can reduce and even replace AI software development.

Artificial intelligence inspires both awe and fear. How far will machines go? Will we simply experience job displacement, or will it go as far as species

replacement?

Could this be an opportunity for many? Who knows? In any case, this chapter provides some guidance to help you to think in a way that drives you to be constantly innovating and being useful. In the age of Big Data, where we are often faced with huge datasets, AI is here to stay; we won't be able to cope without it. Let's make the most of it!

Before we begin to look into the incredible opportunities provided by AI, let's clarify in our minds what exactly the differences are, first between *invention* and *innovation*, and then *revolution* versus *disruption*. It is important to understand the impact that what we are developing and implementing will have on the AI market.

Inventions versus innovations

Some AI programs, especially deep learning algorithms, remained inventions and not innovations until Google and other major players used them on a large scale.

If you have invented a better algorithm than Google for some applications, it remains an invention until it actually *changes* something in your corporation or on the web.

Suppose you find a quicker way to recognize an image through an optimized number of neurons and a new activation function. If nobody uses it, then that invention remains a personal theoretical finding no matter how good it appears.

When others begin to use this new algorithm, then it becomes an innovation. An invention becomes an innovation only when it changes a process within a company or by a sufficient number of private users.

Revolutionary versus disruptive solutions

Suppose a new image recognition algorithm becomes an innovation in a significant corporation. This new algorithm has gone from being an **invention** (not used) to an **innovation** (a solution making a difference).

The corporation now widely uses the algorithm. Every subsidiary has access to it. For this corporation, the new image recognition algorithm has attained a revolutionary status. Corporate sales have gone up, and profit margins have as well.

But maybe this corporation does not dominate the market, and nobody has followed its example. The innovation remains revolutionary but has not become disruptive.

Then, let's say the person who created the algorithm decides to leave the company and start a business with the algorithm. It appears on GitHub as an open source program. Everybody wants it and the number of downloads increases daily until 1,000,000+ users have begun to implement it. Some very low-priced add-ons are provided on the company website. Within a year, it becomes a new way of recognizing images. All companies must follow suit or lag behind. The solution has become **disruptive** because it has changed its market on a global scale.

Where to start?

We have now explored the basic concepts of creating AI. The first step in a translation project using Google Translate is to take the algorithm as far as possible using Google's API. As we will see in the next section, we will explore the limits of Google Translate. Once the limit is found, creativity kicks in when we customize Google Translate using AI.

We will discover that even if a solution exists, it has limits and can be improved, customized, packaged, and sold. *If there is a limit, there is a market.*

Never criticize the flaws you find in an AI solution; they are gold mines!

Where there is a limit, there is an opportunity.

Let's go to the cutting edge and then over the border into uncharted territory using Google Translate to illustrate this.

Discover a world of opportunities with Google Translate

Starting with Google Translate to explore NLP is a good way to prepare to use NLP in web solutions. Any disruptive web-based solution must be able to run in at least a few languages. You will need to master NLP in several languages to implement a chatbot, a translation solution, and online information sites such as Wikipedia.

Google provides many resources to use, explore, or improve Google Translate. Getting a Python code to run and then assess the quality of the results will prove vital before implementing it for crucial translations in a company. Let's get Google Translate running.

Getting started

Google's developers' API client library page is as follows: https://developers.google.com/api-client-library/. On this page, you will see libraries for many languages: Java, PHP, .NET, JavaScript, Objective-C, Dart, Ruby and more.

Then, go to the Python resources and follow the instructions to sign up, create a project in the Google API Console, and install the library.

If you encounter problems doing this part or do not wish to install anything yet, this chapter is self-contained. The source code is described in the chapter.

You are now ready to go, irrespective of whether you installed the tools.

The program

The goal of this section is to implement Google Translate functionality. You can implement and run the program or first simply read the self-contained section.

The header

The standard Google header provided by Google should be enough to get the API to work, as shown in the following code:

```
from googleapiclient.discovery import build
```

Considering the many languages Google manages, special characters are a major problem to handle. Many forums and example programs on the web struggle with the `UTF-8` header when using Google Translate. Many solutions are suggested, such as the following source code header.

```
# -*- coding: utf-8 -*-
```

Then, when Google Translate returns the result, more problems occur, and many develop their own functions. They work fine, but I was looking for a straightforward one-line solution. The goal here was not to have many lines of code but focus on the limit of Google Translate to discover the cutting-edge interpreting languages in AI.

So, I did not use the `UTF-8` header, but implemented it using the HTML library.

```
import html
```

When confronted with a result, the following one-line HTML parser code did the job.

```
print("result:", html.unescape(result))
```

It works well because Google will return an HTML string or a text string depending on what option you implement. This means that the HTML module can do the job.

Implementing Google's translation service

Google's translation service needs at least three values to return a result:

- `developerKey`: This is the API key obtained at the end of the getting-started process described previously.
- `q="text to translate"`: In my code, I used `source`.
- `target="abbreviation of the translated text"`: `en` for English, `fr` for French, and so on.

More options are available, as described in the following sections.

With this in mind, the translation function will work as follows:

```
def g_translate(source,targetl):
    service = build('translate', 'v2',developerKey='your Ke
y')
    request = service.translations().list(q=source,
        target=targetl)
    response = request.execute()
    return response['translations'][0]['translatedText']
```

In the `Google_translate.py` program, `q` and `target` will be sent to the function to obtain a parsed result:

```
source="your text"
targetl="abbreviation of the target language"
result = g_translate(source,targetl)
print(result)
```

To sum up the program, let's translate Google Translate into French, which contains accents parsed by the HTML parser:

```
from googleapiclient.discovery import build
import html
def g_translate(source,targetl):
    service = build('translate', 'v2',developerKey='your ke
y')
    request = service.translations().list(q=source,
        target=targetl)
    response = request.execute()
    return response['translations'][0]['translatedText']
source='Google Translate is great!'
targetl="fr"
result = g_translate(source,targetl)
```

```
print("result:", html.unescape(result))
```

`Google_Translate.py` works fine. The result will come out with the correct answer and the parsed accent:

```
Google Translate est génial!
```

At this point, Google Translate satisfies a black box exploration approach. It is disruptive, has changed the world, and can replace translators in many corporations for all corporate needs.

In fact, we could end the chapter here, go to our favorite social network, and build some hype on our translation project.

Happy ending?

Well, not yet!

We need to explore Google Translate from a linguist's perspective.

Google Translate from a linguist's perspective

A linguist's approach to the program will involve a deeper, white box sort of exploration. The method will reveal many areas to improve.

By the time this book is published, perhaps Google will have improved the examples in this chapter. But don't worry; in this case, you will quickly find hundreds of other examples that are incorrect. The journey has just begun!

Playing with the tool

Playing with a tool with random examples can lead to surprising results. This exploratory source code is saved as `Google_translate_a_few_test_expressions.py`.

The program simulates a dialog created by a person named Usty as follows:

```
source='Hello. My name is Usty!'
```

```
 >>>result:Bonjour. Je m'appelle Usty!
source='The weather is nice today'
 >>>result: Le temps est beau aujourd'hui
source='Ce professor me chercher des poux.'
 >>>result: This professor is looking for lice!
```

The first two examples look fine in French, although the second translation is a bit strange. But in the third test, the expression *chercher des poux* means *looking for trouble* in English, and not looking for lice, as translated into French.

A linguistic assessment of Google Translate will now be made.

Linguistic assessment of Google Translate

Assessing Google Translate correctly will lead directly to its limits.

Limits are the boundaries researchers crave! We are frontiersmen!

An expert-level assessment will lead the project team to the frontier and beyond. To do this, we will first explore some linguistic methods.

Lexical field theory

Lexical fields describe word fields. A word only acquires its full meaning when interpreted within a context. This context often goes beyond a few other words or even a sentence.

Chercher des poux translated as such means *look for lice*. But in French, it can mean *looking for trouble* or literally *looking for lice*. The result that Google Translate comes up with contains three basic problems.

```
source='chercher des poux'
>>result: look for lice
```

Problem 1 – the lexical field: There is no way of knowing whether this means looking for lice or looking for trouble without a context.

Problem 2 – metaphors or idiomatic expressions: Suppose you have to translate *this is giving you a headache*. There is no way of knowing whether it is a physical problem or a metaphor meaning *this is driving you crazy*. These two

idiomatic expressions happen to have the same metaphors when translated into French. But the *lice* metaphor in French means nothing in English.

Problem 3: *chercher* is an infinitive in French, and the result should have been *looking* for lice in English. But entering *chercher des limites est intéressant* provides the right verb form, which is *looking for*:

```
source='Chercher des limites est intéressant.'
>>>result:Looking for boundaries is interesting.
```

The answer is correct because *is* splits the sentence into two, making it easier for Google Translate to identify *chercher* as the first part of a sentence, thus using *looking* in English.

Lexical fields vary from language to language, but so does jargon.

Jargon

Jargon arises when fields specialize. In AI, the expression *hidden neurons* is jargon. This expression means nothing to a lawyer, for example. A lawyer may think you have hidden intelligence on the subject somewhere or are hiding money in a cryptocurrency named hidden neuron.

In the same way, if somebody asks an AI expert to explain the exact meaning of *filing a motion*, that would prove difficult.

In a corporate legal environment, beyond using Google Translate as a dictionary, translating sentences might be risky if only a random number of results prove to be correct.

If we add the jargon variations to the lexical variations from one language to another, we can see that word-to-word translation does not work when a word is in a context. Translating is thus more than just finding the most similar words in the language we are translating to.

Translating is not just translating but interpreting

Sometimes, translating requires interpreting, as shown with the following sentence taken from a standard description of French commercial law:

```
source='Une SAS ne dispense pas de suivre les recommandatio
ns en vigueur autour des pratiques commerciales.'
>>>result:An SAS does not exempt from following the recomme
ndations in force around commercial practices.
```

The French sentence refers to a type of company; SAS is similar to company types like inc., ltd., and so on. In English, SAS means Special Air Service. Then comes the grammar, which does not sound right.

A translator would write better English and also specify what an SAS is:

An SAS (a type of company in France) must follow the recommendations that cover commercial practices.

Translating often means interpreting, and not simply translating words.

In this case, a legal translator may interpret the text in a contract and go as far as writing:

The COMPANY must respect the legal obligation to treat all customers fairly.

The legal translator will suggest that *COMPANY* be defined at the beginning of the contract to avoid confusion, such as the one Google Translate just made.

When reading about NLP, chatbots, and translation, everything seems easy. However, working on Google Translate can easily turn into a nightmare!

Let's take one last example:

```
The project team is all ears
```

Google Translate provides the output in French:

```
source:"The project team is all ears".
>>>result: L'équipe de projet est tout ouïe.
```

In French, as in English, it is better to say *project team* and not use *of* to say the *team of the project*. In French, we have *équipe projet* (équipe (team) appears before project).

From our examples so far, we can see that Google Translate is:

- Sometimes correct
- Sometimes wrong
- Sometimes partly correct and partly wrong

The problem now is how to know which category a translation is in.

How to know whether a translation is correct

How can you check a translation if you do not know the language?

Be careful. If Google Translate provides randomly correct answers in another language, then you have no way of knowing whether the translation is reliable or not.

If you can't be confident that Google Translate is going to be correct, you may find yourself in difficult situations; even sending the opposite of what you mean to somebody important to you. You may misunderstand a sentence you are trying to understand.

In a transportation company, for example, you could write an email stating that the coach stopped and people were complaining:

```
source='The coach stopped and everybody was complaining.'
```

Google Translate, for lexical field reasons, got it wrong and translated *coach* as a sports trainer in French, which would give a completely different meaning to the sentence:

```
result: L'entraîneur s'est arrêté et tout le monde se plaig
nait..
```

Now, the situation can get worse. To help Google Translate, let's add some context.

```
source='The coach broke down and stopped and everybody was
complaining.'
```

This answer is worse. Google Translate translates *broke down* correctly with the French expression *en panne* but still translates *coach* as *entraineur* (trainer) in French, meaning the *trainer* broke down, not the *coach* (bus).

```
result: L'entraîneur est tombé en panne et s'est arrêté et
tout le monde se plaignait.
```

Google will no doubt continue to improve the program as it has done since 2006. For now, however, a human translator will find hundreds of expressions Google Translate cannot deal with yet.

Understanding a sentence in your native language can prove difficult when a word or an expression has several meanings. Adding a translation function to the issue makes it even more difficult to provide a reliable answer.

And that is where we reach the frontier, just beyond the cutting edge, as we have established the limits of Google Translate.

In the next section, we'll look at some ways we could improve standard Google Translate results. Although no silver bullet exists to verify a translation, we will explore methods to improve the process. We will find a way to improve Google Translate and implement it.

AI as a new frontier

Google has a great, but limited, translation program. Use the flaws to innovate! AI research and development has just scratched the surface of the innovations to come.

First, implement an AI solution. Then, use it for what it is. But don't accept its limits. Don't be negative about it. Innovate! Imagine ideas or listen to other ideas you like and build solutions in a team! Google might even publish your solutions!

Improving Google Translate for any translation is impossible. A realistic approach is to focus on customizing Google Translate for a given domain, such as the transportation company in this example. In the next section, we will focus

on ways to customize Google Translate.

Lexical field and polysemy

`Google_Translate_Customized.py` will provide ideas on how to improve Google Translate in a specific area. This section focuses on the transportation vocabulary error Google Translate made. Once again, Google may rapidly correct this error, but the method can be applied to the many remaining errors.

A **lexical field** contains words that form sets and subsets. They differ from one language to another. A language itself forms a set and contains subsets of lexical fields.

Colder countries have more words describing water in its frozen form than tropical countries where snow hardly ever falls. A lexical field of cold could be a subset of *C*:

C = {*ice*, *hail*, *snowflakes*, *snowman*, *slushy*, *powder*, *flake*, *snowball*, *blizzard*, *melting*, *crunch* … *n*}

The curse of dimensionality applies here. Words contain an incredible number of dimensions and definitions. To translate certain expressions, Google Translate suppresses their dimensions and reduces them.

Google Translate often uses n-grams to translate. An n-gram is a fixed-length sequence of tokens. Tokens can be a word, a character, or even a numerical representation of words and characters.

The probability that token *n* means something is calculated given the preceding/following $n - x$ or $n + x$ tokens. x is a variable depending on the algorithm applied.

For example, *slushy* has a special meaning in the expression *slushy snow*. The snow is partly melting, it's watery and making a *slushing* sound when we walk through it. Melting is only one component of the meaning of *slush*.

Google Translate, at this point, will only translate *slushy snow* in French by:

neige (snow) *fondante* (melting)

Google Translate will also translate *melting snow* by:

neige (snow) *fondante* (melting)

To translate *slushy* into French, you have to use a phrase. To find that phrase, you need some imagination or have some parsed (searched) novels or other forms of speech representations. That takes time and resources. It will most probably take years before Google Translate reaches an acceptable native level in all the languages publicized.

Another dimension to take into account is polysemy.

Polysemy means a word can have several very different meanings in a language. The equivalent word in another language may simply have one meaning or other, very different meanings.

"Go + over" in English can mean *go over a bridge* or *go over some notes*. At this point (hopefully it will improve by the time you read this book), it is translated in both cases in French by *aller sur*. This means to go on (not over), which is incorrect in both cases. Prepositions in English constitute a field in themselves, generating many meanings with the same word. The verb *go* can have a wide list of meanings: *go up* (upstairs), *go up* (stock market), *go down* (downstairs), *go down* (fall apart), and many more possibilities besides.

The prototype customized program starts with defining `X`. A small dataset to translate that will be more than enough to get things going:

```
X=['Eating fatty food can be unhealthy.',
   'This was a catch-22 situation.',
   'She would not lend me her tote bag',
   'He had a chip on his shoulder',
   'The market was bearish yesterday',
   'That was definitely wrong',
   'The project was compromised but he pulled a rabit out o
f his hat',
   'So just let the chips fall where they may',
   'She went the extra mile to satisfy the customer',
   'She bailed out when it became unbearable',
   'The term person includes one or more individuals, labor
 unions, partnerships, associations, corporations, legal re
presentatives, mutual companies, joint-stock companies, tru
sts, unincorporated organizations, trustees, trustees in ba
```

```
nkruptcy, or receivers.',
   'The coach broke down, stopped and everybody was complai
ning']
```

> If you find spelling mistakes or minor mistakes, do not correct them during the training phase. Some amount of noise is required to reproduce human and machine errors to avoid overfitting.

Google Translate will automatically translate these sentences.

`X1`, as implemented in the following code, defines some keywords statistically related to the sentences; it applies the n-gram probability theory described previously.

```
X1=['grasse',
    'insoluble',
    'sac',
    'aggressif',
    'marché',
    'certainement',
    'chapeau',
    'advienne',
    'supplémentaire',
    'parti',
    'personne',
    'bus']
```

Each line in `X1` goes with the corresponding line in `X`. As explained, this only remains a probability and may not be correct.

We are not seeking perfection at this point but an improvement.

Let's explore how we can improve Google Translate by customizing translations by implementing a KNN in a Python program.

Exploring the frontier – customizing Google Translate with a Python program

Now it's time to add some customized novelties. The use of the vectors in this section will be explained in the next section, again through the source code that uses them.

A trigger vector will force the program to try an alternate method to translate a mistranslated sentence. When the sentence has been identified, and if its value in `X2` is equal to `1`, it triggers a deeper translation function, as implemented here:

```
X2=[0,0,0,1,0,0,0,0,0,0,0,1]
```

`0` and `1` are flags. Each value represents a line in `X`.

> **Note for developers**: To use this method correctly, all the values of this vector should be set to `1`. That will automatically trigger several alternate methods to translate Google Translate errors. A lot of work remains to be done here!

The example is taken from a transportation business. A transportation phrase dictionary should be built. In this case, a general `phrase_translation` dictionary has been implemented with one expression, as shown in the following array.

```
phrase_translation=['','','','Il est agressif','','','','',
'','','','']
```

What remains to be done in order to fill up this dictionary?

- Scan all the documents of the company—emails, letters, contracts, and every form of written documents.
- Store the embedded words and sentences.
- Train the team to use the program to improve it by providing feedback (the right answer) in a learning interface when the system returns incorrect answers.

What Google Translate cannot do on a global scale, you can implement at a local scale to improve the system significantly.

Now that we have defined a method, we will dig into the KNN algorithm.

k-nearest neighbor algorithm

No matter how you address a linguistic problem, it will always boil down to the concept of **context**. When somebody does not understand somebody else, they say: "you took my words out of their context," or "that is not what I meant; let me explain."

As explained before, in many cases, you cannot translate a word or expression without a lexical field. The difficulty remains proportional to the polysemy property, as the program will show.

Using the KNN algorithm as a classification method can prove extremely useful. Any language interpretation (translation or chatbot) will have to use a context-oriented algorithm.

By finding the words closest (neighbors) to each other, KNN will create the lexical fields required to interpret a language. Even better, when provided with the proper datasets, it will solve the polysemy problem, as shown in the upcoming sections.

Implementing the KNN algorithm

Generally, a word requires a context to mean something. Looking for "neighbors" close by provides an efficient way to determine where the word belongs.

KNN is supervised because it uses the labels of the data provided to train its algorithm. KNN, in this case, is used for classification purposes. For a given point *p*, KNN will calculate the distances to all other points. Then, *k* represents the k-nearest neighbors to take into account.

Let's clear this up by means of an example. In English, the word "coach" can mean a trainer on a football field, a bus, or a railroad passenger car. In a transportation company, "coach" will mostly be a bus that should not be confused with a trainer:

- **Step 1**: Parsing (examining in a detailed manner) texts with "coach" as a bus and "coach" as a trainer. Thus, the program is searching for three target words: trainer, bus, and coach.
- **Step 2**: Finding some words that appear close to the target words we are searching for. As recommended, do the following:
    - Parse all the company documents you can use with a standard program.
    - Use a Python function such as `if(n-gram in the source)` then store the data.

In this case, the `V1.csv` file shown in the following output excerpt provided with the source code contains the result of such a parsing function:

```
broke,road,stopped,shouted,class
1,3.5,6.4,9,trainer
1,3.0,5.4,9,trainer
1,3.2,6.3,9,trainer
...
6.4,6.2,9.5,1.5,bus
2,3.2,9,1,bus
6.4,6.2,9.5,1.5,bus
...
3.3,7.3,3.0,2.5,coach
4.7,5.7,3.1,3.7,coach
2.0,6.0,2.7,3.1,coach
```

Generating files such as `V1.csv` is not in the scope of this chapter or book. However, you can start, among other sites, by exploring scikit-learn's text document functionality at the following link:

https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html

The program parsed emails, documents, and contracts. Each line represents the result of parsing one document. The numbers represent the occurrences (number of times the word was present). The numbers have been "squashed" (divided again and again) to remain small and manageable. For more on how to work with text data, please click on the scikit-learn link in the previous paragraph.

Progressively, the words that came out with "trainer" were "shouted" more than "stopped." For a bus, "broke" (broken down as in breaking down), "road," and "stopped" appeared more than "shout."

"Coach" appeared on an average of "shouted," "stopped," "road," and "broke" because it could be either a trainer or a bus, hence the problem we face when translating this word. The polysemy (several meanings) of "coach" can lead to poor translations.

The KNN algorithm loaded the `V1.csv` file that contains the data to be trained and finds the following result:

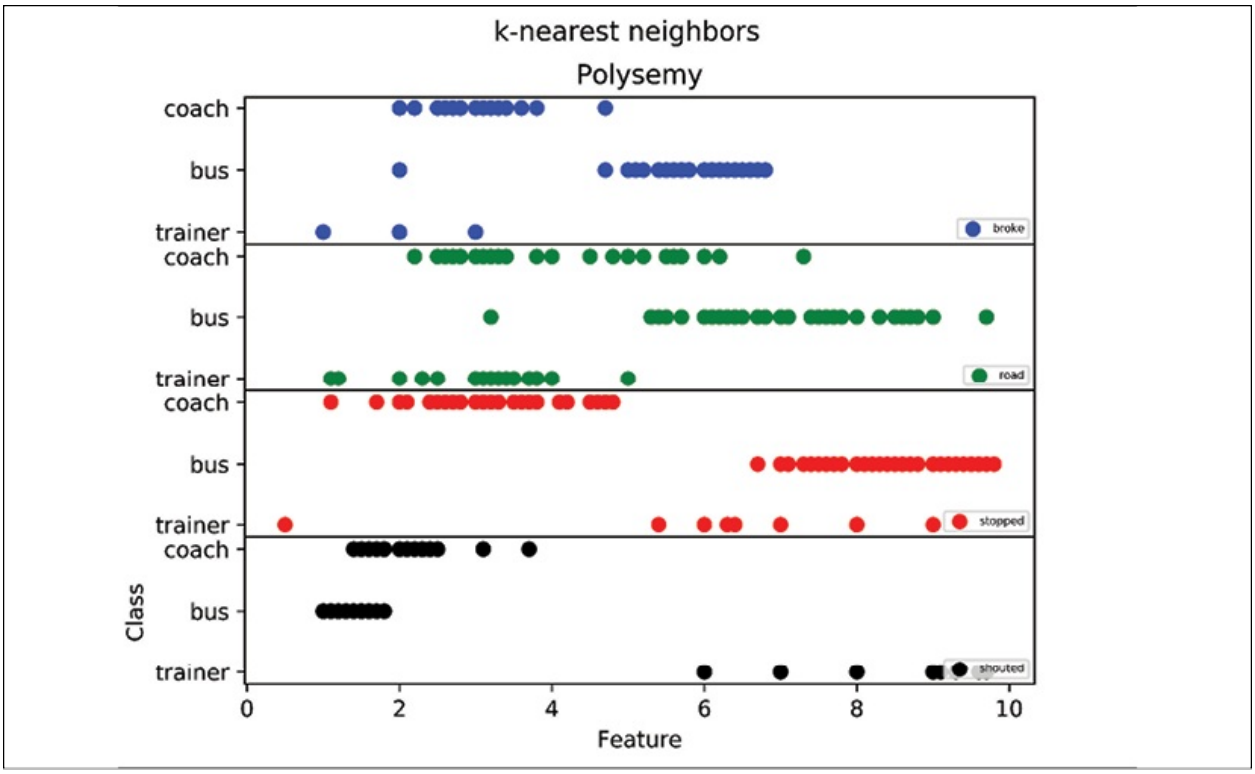


Figure 6.1: Result from the KNN algorithm

The `knn_polysemy.py` program determined the following:

- The verb "broke" in blue has a better chance of applying to a bus (*x* axis value > 6) than to a trainer (*x* axis value < 4). However, "coach" remains above "trainer" and below "bus" because it can be both.
- The word "road" follows the same logic as the blue chart.
- The verb "stopped" can apply to a trainer and more to a bus. "Coach"

remains undecided again.

- The verb "shouted" applies clearly to a trainer more than a bus. "Coach" remains undecided again.

Note that the coordinates of each point in these charts are as follows:

- **y axis**: bus = 1, coach = 2, and trainer = 3.
- **x axis**: The value represents the "squashed" occurrence (the number of times the word appeared) values.

This is the result of the search for those words in many sources.

When a new point, a data point named P_n is introduced into the system, it will find its nearest neighbor(s) depending on the value of k.

The KNN algorithm will calculate the Euclidean distance between P_n and all the other points from P_1 to P_{n-1} using the Euclidean distance formula. The k in KNN represents the number of "nearest neighbors" the algorithm will take into account for classification purposes. The Euclidean distance (d_1) between two given points, for example, between $P_n(x_1, y_1)$ and $P_1(x_2, y_2)$, is:

$$d_1(P_n, P_1) = \sqrt[2]{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Considering the number of distances to calculate, a function such as the one provided by `sklearn.neighbors` proves necessary.

The knn_polysemy.py program

The program imports the `V1.csv` file described previously, prints a few lines, and prepares the labels in the correct arrays in their respective x axis and y axis, as shown in this source code example:

```
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
# Import data
df = pd.read_csv('V1.csv')
print (df.head())
```

```
# KNN Classification labels
X = df.loc[:,'broke':'shouted']
Y = df.loc[:,'class']
```

Then the model is trained, as shown in the following code:

```
# Trains the model
knn = KNeighborsClassifier()
knn.fit(X,Y)
```

Once the model is trained, a prediction is requested and is provided by the following code:

```
# Requesting a prediction
#broke and stopped are
#activated to see the best choice of words to fit these fea
tures.
# brock and stopped were found in the sentence to be interp
reted.
# In X_DL as in X, the labels are : broke, road, stopped,sh
outed.
X_DL = [[9,0,9,0]]
prediction = knn.predict(X_DL)
print ("The prediction is:",str(prediction).strip('[]'))
```

This is the result displayed:

```
The prediction is: 'bus'
```

The initial data is plotted for visualization purposes, as implemented in the following code:

```
#Uses the same V1.csv because the parsing has
# been checked and is reliable as "dataset lexical rule bas
e".
df = pd.read_csv('V1.csv')
# Plotting the relation of each feature with each class
figure,(sub1,sub2,sub3,sub4) = plt.subplots(
    4,sharex=True,sharey=True)
plt.suptitle('k-nearest neighbors')
plt.xlabel('Feature')
plt.ylabel('Class')
```

```
X = df.loc[:,'broke']
Y = df.loc[:,'class']
sub1.scatter(X, Y,color='blue',label='broke')
sub1.legend(loc=4, prop={'size': 5})
sub1.set_title('Polysemy')
X = df.loc[:,'road']
Y = df.loc[:,'class']
sub2.scatter(X, Y,color='green',label='road')
sub2.legend(loc=4, prop={'size': 5})
X = df.loc[:,'stopped']
Y = df.loc[:,'class']
sub3.scatter(X, Y,color='red',label='stopped')
sub3.legend(loc=4, prop={'size': 5})
X = df.loc[:,'shouted']
Y = df.loc[:,'class']
sub4.scatter(X, Y,color='black',label='shouted')
sub4.legend(loc=4, prop={'size': 5})
figure.subplots_adjust(hspace=0)
plt.show()
```

A compressed version of this program has been introduced in `Google_Translate_Customized.py`, as shown here:

```
def knn(polysemy,vpolysemy,begin,end):
    df = pd.read_csv(polysemy+'.csv')
    X = df.loc[:,'broke':'shouted']
    Y = df.loc[:,'class']
    knn = KNeighborsClassifier()
    knn.fit(X,Y)
    prediction = knn.predict(vpolysemy)
    return prediction
```

The description is as follows:

- `polysemy` is the name of the file to read because it can be any file.
- `vpolysemy` is the vector that needs to be predicted.
- In future, in the to-do list, `begin` should replace `broke` and `end` should replace `shouted` so that the function can predict the values of any vector.
- The KNN classifier is called and the prediction returned.

Now that we have prepared the KNN classifier function, we can customize Google Translate.

Implementing the KNN function in Google_Translate_Customized.py

This program requires more time and research because of the concepts of linguistics involved. The best way to grasp the algorithm is to run it in order.

Google Translate offers various translation methods. We will focus on two of them in the following code:

```
#print('Phrase-Based Machine Translation(PBMT)model:base'):
 #m='base'
print('Neural Machine Translation model:nmt')
```

These are explained as follows:

- **Phrase-based machine translation (PBMT)**: This translates the whole sequence of words. The phrase, or rather phraseme (multi-word expression), is not always quite a sentence.
- **Neural machine translation (NMT)**: This uses neural networks such as a **recurrent neural network** (**RNN**), which will be detailed later in this book. This method goes beyond the phraseme and takes the whole sentence into account. In terms of the dataset presented in this chapter, this neural network method provides slightly better results.

Both methods and Google's other approaches are interesting, but Google Translate still requires additional customized algorithms to reach an acceptable level of quality in many cases. In this chapter, we are exploring one approach with a KNN, but you can use others as long as they work.

As you have seen so far, the subject is extremely complex if you take the lexical fields and structures of the many languages, their regional variations, and jargon into account.

Step 1 – translating the X dataset line by line from English into French

The following code calls the translation function:

```
for xi in range(len(X)):
```

```
    source=X[xi]
    targetl="fr";m='nmt'
    result = g_translate(source,targetl,m)
```

The code is explained as follows:

- `xi` is the line number in `X`.
- `source` is the `xi` line in `X`.
- `targetl` is the target language, in this case, `fr` (French).
- `m` is the method (PBMT or NMT), as described previously. In this case, `nmt` is applied.
- Then, the Google Translate function is called as described earlier in this chapter. The result is stored in the `result` variable.

Step 2 – backtranslation

How can somebody know the correctness of a translation from language L_1 to language L_2 if L_1 is the person's native language, and L_2 is a language the person does not understand at all?

This is one of the reasons, among others, that translators often use backtranslation to check translations:

$$Translation = Initial\ translation\ from\ L_1\ to\ L_2$$

$$Backtranslation = Translation\ back\ from\ L_2\ to\ L_1$$

If the initial text is not obtained, then there is probably a problem. In this case, the length of the initial sentence L_1 can be compared to the length of the same sentence translated back to L_1. The following code calls backtranslation:

```
    back_translate=result
    back_translate = g_translate(back_translate,targetl,m)
    print("source:",source,":",len(source))
    print("result:",result)
    print("target:",back_translate,":",len(back_translate))
```

Length comparison can be used to improve the algorithm:

Length of the initial n-gram = Length of the backtranslation

If it's equal, then the translation may be correct. If not, it could be incorrect. Of course, more methods must be applied during each translation. However, a method that leads to improvement is already a good step. In this case, the source (initial sentence) is compared to the backtranslation in the following code:

```
    if(source == back_translate):
        print("true")
        if((term not in words)and (xi!=4)):
            t+=1
    else:
    f+=1;print("false")
```

- `t` is a `True` counter.
- `f` is a `False` counter.

The first line of `X` runs as follows:

```
source: Eating fatty food can be unhealthy. : 35
result: Manger de la nourriture grasse peut être malsain.
target: Eating fat food can be unhealthy. : 33
false
```

Eating fatty food is backtranslated as *eating fat food*, which is slightly wrong. Something may be wrong.

The French sentence sounds wrong, too. Fatty food cannot be translated as such. Usually, the common sentence is *manger gras*, meaning *eating* (manger) *fatty* (gras), which cannot be translated into English as such.

The `X` array referred to in this section starts at line 8:

```
X=['Eating fatty food can be unhealthy.',
   'This was a catch-22 situation.',
   'She would not lend me her tote bag',
   'He had a chip on his shoulder',
....]
```

Several phrases come back with a false translation, for example, `X[4]`, `'He had`

`a chip on his shoulder'`. I programmed a phrase-based translation using a trigger in the `False` condition in the following code.

```
    else:
        f+=1;print("false")
    if(X2[xi]>0):
        DT=deeper_translate(source,xi)
        dt+=1
```

Since I did not write a complete application for this book, but just some examples that can be extended in the future, I used `X2` as a trigger. If `X2[x1]>0`, then the `deeper_translate` function is activated.

Step 3 – deeper translation with phrase-based translations

`deeper_translate` has two arguments:

- `source`: The initial sentence to translate
- `x1`: The target sentence

In this case, the problem to solve is an idiomatic expression that exists in English but does not exist in French:

```
source: He had a chip on his shoulder : 29
result: Il avait une puce sur son épaule
target: He had a chip on his shoulder : 29
false
```

To have *a chip on the shoulder* means to have an issue with something or somebody. It expresses some form of tension.

Google translated *chip* by assuming computer chip, or *puce* in French, which means both *computer chip* and *flea*. The translation is meaningless.

Chip enters three categories and should be labeled as such:

- Idiomatic expression
- Jargon
- Polysemy

At this point, the following function I created simulates the phrase-based solution to implement deeper translations.

```
def deeper_translate(source,index):
    dt=source
    deeper_response=phrase_translation[index]
    if(len(deeper_response)<=0):
        print("deeper translation program result:",
            deeper_response,":Now true")
```

The `deeper_translate` function looks for the translated sentence containing *chip* in the following `phrase_translation` array (list, vector, or whatever is necessary).

```
phrase_translation=['','','','Il est agressif','','','','',
'','','','']
```

The final result comes out with a translation, backtranslation, term search, and phrase translation. The following is the result produced, with comments added here before each line:

```
Initial sentence:
source: He had a chip on his shoulder : 29
Wrong answer:
result: Il avait une puce sur son épaule
The back-translation works:
target: He had a chip on his shoulder : 29
term: aggressif
false
deeper translation program result: Il est agressif
```

The question is, where did `term` come from?

`term` comes from `X1`, a list of keywords that should be in the translation. `X1` has been entered manually, but it should be a list of possibilities resulting from an automatic search conducted on the words in the sentence viewed as classes. This means that the sentence to be translated should have several levels of meaning, not just the literal one that is being calculated.

The actual `True` / `False` conditions contain the following deeper translation-level words to check:

```
    if(source == back_translate):
        print("true")
        if((term not in words) and (xi!=4)):
            t+=1
    else:
        f+=1;print("false")
        if(X2[xi]>0):
            DT=deeper_translate(source,xi)
            dt+=1
```

In the present state of the prototype, only example four activates a phrase-based translation. Otherwise, `True` is accepted. If `False` is the case, the deeper translation is only activated for two cases in this sample code. The flag is in `X2` (`0` or `1`).

`deeper_translate` is called for either the phrase-based translation (described previously) or the KNN routine, which is activated if the phrase-based translation did not work.

If the translation did not work, an n-gram is prepared for the KNN algorithm, as shown in the following code:

```
    if(len(deeper_response)<=0):
        v1=0
        for i in range(4):
            ngram=V1[i]
            if(ngram in source):
                vpolysemy[0][i]=9
                v1=1
```

`V1[i]` contains the keywords (n-grams) described in the preceding KNN algorithm for the transport lexical field, as shown in the following code:

```
V1=['broke','road','stopped','shouted','coach','bus','car',
    'truck','break','broke','roads','stop']
```

The source (sentence to translate) is parsed for each n-gram. If the n-gram is found, the polysemy vector is activated for that n-gram. The initial values are set to `0`, as shown in the following code:

```
vpolysemy=[[0,0,0,0]]
```

The variable `v1` is activated, which informs the program that `V1.csv` must be read for this case. An unlimited number of KNN references should be automatically created, as described previously in the KNN section.

In this case, only `v1` is activated. But after several months of working on the project for the company to customize their local needs, many other files should be created.

In this case, when `v1` is activated, it fills out the variables as follows.

```
    if(v1>0):
        polysemy='V1'
        begin=str(V1[0]).strip('[]');end=str(V1[3]).strip('
[]')
        sememe=knn(polysemy,vpolysemy,begin,end)
```

- `polysemy` indicates the KNN file to open.
- `begin` is the first label of the `V1` vector and `end` is the last label of the `V1` vector.
- `sememe` is the prediction we expect.

Now, a condensed version of the KNN algorithm is called, as described previously for `knn_polysemy.py`, in the following code:

```
def knn(polysemy,vpolysemy,begin,end):
    df = pd.read_csv(polysemy+'.csv')
    X = df.loc[:,begin:end]
    Y = df.loc[:,'class']
    knn = KNeighborsClassifier()
    knn.fit(X,Y)
    prediction = knn.predict(vpolysemy)
    return prediction
```

The example, in this case, is the polysemy feature of *a coach*, as explained in the KNN section. The output will be produced as follows:

```
Source: The coach broke down, stopped and everybody was com
plaining : 59
result: L'entraîneur est tombé en panne, s'est arrêté et to
```

```
ut le monde se plaignait
target: The coach broke down, stopped, and everyone was com
plaining : 59
term: bus
false
```

The translation is false because Google Translate returns *trainer* instead of *bus*.

The term *bus* is identical in English and French.

The KNN routine returned *bus* in English as the correct word to use when *broke down* and *stopped* were found, as shown in the KNN section.

The goal of the rest of the source code in the `deeper_translate` function is to replace *coach*—the word increasing the polysemy feature to translate—with a better word (limited polysemy) to translate: `sememe`.

The `sememe` variable is initialized by the KNN function in the following code:

```
            sememe=knn(polysemy,vpolysemy,begin,end)
            for i in range(2):
                if(V1_class[i] in source):
                    replace=str(V1_class[i]).strip('[]')
                    sememe=str(sememe).strip('[]')
                    dtsource = source.replace(replace,semem
e)
                    targetl="fr";m='base'
                    result = g_translate(dtsource,targetl,m
)
                    print('polysemy narrowed result:',resul
t,
                          ":Now true")
```

The function replaces *coach* by *bus* found by the KNN algorithm in the English sentence and then asks Google Translate to try again. The correct answer is returned.

Instead of trying to translate a word with too many meanings (polysemy), the `deeper_translate` function first replaces the word with a better word (less polysemy). Better results will often be attained.

Step 3.1 – adding a frequentist error probability function

A frequentist error probability function is added to measure performance, as shown in the following code:

```
def frequency_p(tnumber,cnumber):
    ff=cnumber/tnumber #frequentist interpretation and prob
ability
    return ff
```

- `cnumber` is the number of false answers returned by Google Translate.
- `tnumber` is the number of sentences translated.
- `ff` gives a straightforward error (translation) probability, ETP.

The function is called when a translation is false, or `f>0`, as implemented in the following code:

```
    if(f>0):
        B1=frequency_p(xi+1,f) #error detection probability
 before deep translation
        B2=frequency_p(xi+1,f-dt) #error detection probabil
ity after deep translation
    if(f>0):
        print("ETP before DT",round(B1,2),
            "ETP with DT",round(B2,2))
    else:
        print('Insufficient data in probability distributio
n')
```

- `B1` is the error (translation) probability (ETP) before the `deeper_translate` function is called.
- `B2` is the ETP after the `deeper_translate` function is called.

At the end of the program, a summary is displayed, as shown in the following output:

```
print("------Summary------")
print('Neural Machine Translation model:nmt')
print('Google Translate:',"True:",t,"False:",f,'ETP',round(
f/len(X),2))
print('Customized Google Translate:',"True:",t,"False:",f-d
t,'ETP',round((f-dt)/len(X),2))
a=2.5;at=t+a;af=f-a #subjective acceptance of an approximat
```

```
e result
print('Google Translate acceptable:',"True:",at,"False:",af,
'ETP',round(af/len(X),2))
#The error rate should decrease and be stabilized as the KN
N knowledge base increases
print('Customized Google Translate acceptable:',"True:",at,
"False:",af-dt,'ETP',round((af-dt)/len(X),2))
```

- A subjective acceptance of an approximate result has been added to increase the true probability.
- The error rate should decrease as the quality of the KNN knowledge base increases. In frequent probability theory, this means that a stabilized prediction rate should be reached.

We've come to the end of our attempts to improve Google Translate. Let's consider some of the conclusions following our experiment.

Conclusions on the Google Translate customized experiment

The final error (translation) probability produced is interesting, as shown in the following output:

```
>>------Summary------
>>Neural Machine Translation model:nmt
>>Google Translate: True: 2 False: 8 ETP 0.67
>>Customized Google Translate: True: 2 False: 7 ETP 0.58
>>Google Translate acceptable: True: 4.5 False: 5.5 ETP 0.46

>>Customized Google Translate acceptable: True: 4.5 False:
4.5 ETP 0.38
```

Even with its NMT model, Google Translate is still struggling.

This provides great opportunities for AI linguists, as shown with some of the methods presented to improve Google Translate at a local level that could go even further.

This experiment with Google Translate shows that Google has just scratched the

surface of real-life translations that sound right to the native speakers that receive these translations. It would take a real company project to get this on track with a financial analysis of its profitability before consuming resources.

The disruptive revolutionary loop

As you can now see, Google Translate, like all AI solutions, has its limits. Once this limit has been reached, you are at the cutting edge.

Cross the border into AI Frontierland; innovate on your own or with a team.

If you work for a corporation, you can create a revolutionary customized solution for hundreds of users. It does not have to go public. It can remain a strong asset to your company.

At some point or other, the revolutionary add-on will reach beyond the company, and others will use it. It will become disruptive.

Finally, others will reach the limit of your now-disruptive solution. They will then innovate and customize it in their corporation as a revolutionary solution. This is what I call the disruptive revolutionary loop. It is challenging and exciting because it means that AI developers will not all be replaced in the near future by AutoAI bots!

Designing a solution does not mean it will be an invention, an innovation, revolutionary, or disruptive. But that does not really matter. What a company earns with a solution represents more than the novelty of what it sells as long as it is profitable. That is rule number 1. That said, without innovating in its market, that company will not survive through the years.

If a product requires quality for security reasons, it should remain in its invention state as long as necessary. If a product can produce sales at the low end of the market before its total completion, then the company should sell it. The company will acquire a reputation for innovation, get more money to invest, and take over the territory of its competitors.

Summary

Google Translate is a good example of disruptive marketing. As shown, the theory, the model, and even the cloud architecture are over 10 years old. But each time one of the hundreds of millions of users stumbles across it, it creates more disruption by hooking the user onto Google solutions. The user will come back again and again to view more advertisements, and everyone is happy!

AI has only just begun. Google Translate has been around since 2006. However, the results still leave room for developers, linguists, and mathematicians to improve upon. Google has added a neural network and offers other models to improve translations by analyzing whole sentences. How long will it take to be really reliable? In the meantime, the world community is moving AI forward beyond the cutting edge into Frontierland.

In this chapter, we first carefully explored the difference between inventing and innovation. An innovation has an impact on the rest of the market. An invention is just the starting point of an innovation. We saw that a revolutionary solution could be a technical breakthrough. But that revolutionary solution will only become disruptive when it spreads out to the rest of the market.

We then studied some basic linguistics principles that could help us understand Google Translate, its limits, and how to improve translation errors.

We finally implemented a customized translation tool using a KNN to work around Google Translate errors.

In the next chapter, *Chapter 7*, *Optimizing Blockchains with Naive Bayes*, we will go further in our investigation of the new frontier of AI by using blockchains to make predictions in corporate environments.

Questions

1. Is it better to wait until you have a top-quality product before putting it on the market? (Yes | No)

2. Considering the investment made, a new product should always be priced high to reach the top segment of the market. (Yes | No)
3. Inventing a new solution will make it known in itself. (Yes | No)
4. AI can solve most problems without using standard non-learning algorithms. (Yes | No)
5. Google Translate can satisfactorily translate all languages. (Yes | No)
6. If you are not creative, it is no use trying to innovate. (Yes | No)
7. If you are not a linguist, it is no use bothering with trying to improve Google Translate. (Yes | No)
8. Translating is too complicated to understand. (Yes | No)
9. AI has already reached its limits. (Yes | No)

Further reading

- The Harvard Business Review on disruptive innovations can be found here: https://hbr.org/2015/12/what-is-disruptive-innovation
- Google Translate documentation can be found here: https://cloud.google.com/translate/docs/
- Google AlphaGo Zero: https://deepmind.com/blog/article/alphago-zero-starting-scratch
- KNN documentation: http://scikit-learn.org/stable/modules/neighbors.html#neighbors
- Insights on translation ANNs: https://research.googleblog.com/2016/09/a-neural-network-for-machine.html
- Insights on English-French translations: http://www.oneskyapp.com/blog/french-translation-challenges/
- More on how to work with text data to build datasets for your algorithms: https://scikit-

learn.org/stable/tutorial/text_analytics/working_with_text_data.html

7

Optimizing Blockchains with Naive Bayes

In this three-part chapter, we will use blockchains to optimize a supply chain using naive Bayes. To achieve this goal, we will first start by understanding how a blockchain is generated using cryptocurrency as an example.

Blockchains have entered corporations and are here to stay. Hundreds of major corporations have implemented IBM Hyperledger. Suppliers of these corporations will gradually join the network. Corporate blockchains will provide work for many years to come thanks to the millions of lines of code to update with new features and maintain.

Mining cryptocurrency represents the most known use of blockchains. Cryptocurrencies are growing around the world. This chapter starts by explaining how the mining aspect of blockchains works, using bitcoins as an example.

We will then move on and analyze how to use blockchains for a different purpose than generating cryptocurrency. Corporations use blockchains to record transactions between companies, for example.

IBM was founded in 1911, making it the most experienced company in its field today. Google, Amazon, and Microsoft also offer history-making machine learning platforms. However, IBM offers machine learning platforms that are supported by the 100+ years of experience that the company can bring to bear.

Some of IBM's many years in the computer and software market were bumpy. Some terrible decisions caused the company a lot of problems across the world. IBM learned from those mistakes and now offers robust solutions, including IBM Hyperledger for blockchain solutions. IBM advocates using blockchains for corporate transactions.

We will finally move to the third part of this chapter, which explains what blockchains mean for companies around the world and how to use the information blockchains to provide optimizing algorithms with artificial intelligence. Naive Bayes will be applied to a blockchain sample to optimize stock levels.

The following topics will be covered in this chapter:

- The background of blockchain
- Using blockchains to mine bitcoins
- Using blockchains for business transactions
- How the blocks of a blockchain provide a unique way to share information between companies
- Applying artificial intelligence to the blocks of a blockchain to predict and suggest transactions
- Naive Bayes
- How to use naive Bayes on blocks of a blockchain to predict further transactions and blocks

Let's begin with a short introduction to blockchain.

Part I – the background to blockchain technology

In this section, we will go through cryptocurrency mining with blockchains. Producing bitcoins with blockchains made the technology disruptive. The purpose of this section is to understand how the blockchain adventure started before moving on to subsequent uses of blockchain technology.

Blockchain technology will transform transactions in every field. Blockchains appeared in 2008. Nobody knows for sure who invented them. Each block contains an encrypted hash of its predecessor (previous block), the DateTime (timestamp) data, and the information regarding the transaction.

For more than 1,000 years, transactions have been mostly local book-keeping systems. For the past 100 years, even though the computer age changed the way information was managed, things did not change that much. Each company continued to keep its transactions to itself, only sharing some information through tedious systems.

Blockchain makes a transaction **block** visible to the global network it has been generated in.

The fundamental concepts to keep in mind are **sharing** and **privacy control**. The two ideas seem to create a cognitive dissonance, something impossible to solve. Yet it has been solved, and it will change the world.

When a block (a transaction of any kind) is generated, it is shared with the entire network. Permissions to read the information within that block remain manageable and thus private if the regulator of that block wants the information to stay private.

Whether the goal is to mine bitcoins through blocks or use blocks for transactions, artificial intelligence will enhance this innovation shortly.

In the coming sections, we'll talk about blockchain and its applications in some

further detail. Understanding how to produce blocks for cryptocurrency will enable us to grasp blockchain technology. Once we understand blockchain technology, it is easier to see how this secure encrypted method can be applied to any type of business transaction beyond cryptocurrencies. Let's go mining first!

Mining bitcoins

Creating a block in a blockchain does not necessarily have to generate a bitcoin, which is a form of transaction like any other. But understanding how to mine cryptocurrency provides a good way to understand blockchains and how to apply them to many other fields.

Mining a bitcoin means creating a mathematical block for a valid transaction and adding this block to the chain; the blockchain:

$$Blockchain = \{block_1, block_2, the\ block\ just\ added\ \ldots\ block_n\}$$

The blockchain cannot go back in time. It is like a time-dating feature in life. At minute *m*, you do something, at minute $m + 1$ something else, at minute $m + n$ something else, and so on. You cannot travel back in time. *What is done is done*.

When a block is added to the bitcoin chain, there is no way of undoing the transaction.

The global network of bitcoin mining consists of **nodes**. With the appropriate software, you leave a port open, allocate around 150+ GB of disk space, and generate new blocks. The nodes communicate with each other, and the information is relayed to the other nodes around the whole network.

For a node to be a miner, it must solve complex mathematical puzzles that are part of the bitcoin program.

To solve the puzzle, the software must find a number that fits in a specific range when combined with the data in the block being generated. The number is passed through a hash function.

You can call the number a **nonce**, and it is used only once. For example, an integer between 0 and 4,294,967,296 for a bitcoin must be generated.

The process is random. The software generates a number, passes it through the hash function, and sends it out to the network. The first **miner** who produces a number in the expected range informs the whole network that that particular block has been generated. The rest of the network stops working on that block and moves on to another one.

The reward for the miner is naturally paid out in bitcoins. It represents a lot of money, but it's hard to get, considering the competition in the network and the cost required (CPU, electricity, disk space, and time) to produce correct results.

> A constant balance must be maintained between the cost of mining a bitcoin, or any other cryptocurrency, and the amount received for mining it.

Having talked about mining bitcoin, we'll briefly discuss actually using cryptocurrency.

Using cryptocurrency

Be very careful with cryptocurrency. There are now 1,500+ cryptocurrencies. The concept sounds fascinating, but the result remains currency. Currency can be volatile, and you can lose your life's savings in less than an hour if a crash occurs.

> **Golden rule**: If you cannot resist investing in cryptocurrencies, do not invest more than you can afford to lose.

That being said, to use cryptocurrency, first set up a wallet to store your bitcoins, for example. The wallet can be online, through a provider, or even offline.

Once that is done, you will be able to purchase bitcoins as you wish in hard cash or using credit cards, debit cards, and transfers.

Remember, you are buying these currencies like any other currency with all the potential, but also all the risks involved.

In this section, we saw how the blockchain era began with bitcoin production to understand how to mine cryptocurrencies through bitcoins using a blockchain. With the original way blockchains are created in mind, we can apply blockchains to many other fields.

In the second part of our three-part chapter, we will see how to use blockchains beyond cryptocurrencies. We will apply them to supply chains.

PART II – using blockchains to share information in a supply chain

In *Part I – the background to blockchain technology*, we saw how to use blockchains to mine cryptocurrencies. That prerequisite to entering the world of blockchains having been achieved, this section will show how to use blockchains in a supply chain. This will not involve cryptocurrencies. It opens the path to innovating blockchains with AI.

A supply chain is a chain of production and service that gets a product from a starting point to the consumer. For example, take a roll of fabric (cloth) that is shipped from India to Tunisia. In Tunisia, the fabric is cut into patterns and assembled as clothing. Then the clothing is shipped to France where it is packaged in a box with a brand printed on it. It then goes on to be stored in a warehouse to be shipped to a physical shop or directly to an online customer. We can sum this supply chain up as follows:

Cloth from India -> cut and assembled in Tunisia -> shipped to France -> packaged -> stored -> shipped to a shop or directly to a consumer

A supply chain process such as an apparel production and delivery system involves thousands of people along the way: production sites, transport personnel, warehouse employees, and management teams for each step of the chain.

This is where a modern blockchain technology comes in handy to track all of the transactions down in one system.

In the following example, we will take six companies named A, B, C, D, E, and F. In *Chapter 1, Getting Started with Next-Generation Artificial Intelligence through Reinforcement Learning*, we used a size six reward matrix. In that chapter, we used A, B, C, D, E, and F to represent locations in a Markov decision process for reinforcement learning. At the end of the chapter, we saw

that the reward matrix could be applied to many different domains. In our case, we will refer to the six items (A, B, C, D, E, and F) as companies and their locations (one per company in this example) in a supply chain process.

Each company, A to F, that is a member of the supply chain using blockchains can optimize its activity. Instead of each company having separate transaction ledgers, one central blockchain will contain all of the transactions (blocks):

- How much and when the cloth left India
- When it got on a ship and how long it took
- When the ship got to Tunisia
- … all of the intermediate activities right down to the consumer

With such data, the blocks (records of the transactions) of a blockchain have become machine learning goldmines to detect information and make predictions. One profitable reason for using this system is to reduce stock levels. Piling up unsold goods to anticipate sales is costly and can ruin a company's profit. On the contrary, if each company in the supply chain can see the stock levels and real needs of the other partners through the blocks of the blockchain, they can fit their stock levels to the exact need and save huge amounts of money!

Let's see how this works using IBM software as an example.

IBM Blockchain, based on Hyperledger Fabric, provides a way for companies around the world to share a blockchain transaction network without worrying about mining or using cryptocurrencies.

The system is based on the Linux Foundation project. Hyperledger is an open source collaborative project hosted by the Linux Foundation.

At this level, Hyperledger uses blockchains to guarantee secure transactions without trying to optimize the cryptocurrency aspect. The software generates blocks in a blockchain network shared by all the parties involved, but they do not have to purchase cryptocurrencies in the currency sense—only in the technological sense.

In the following graph, we will use the six nodes (companies A to F) to illustrate how the Markov decision process we studied in *Chapter 1*, *Getting Started with Next-Generation Artificial Intelligence through Reinforcement Learning*, can be

applied to a blockchain:

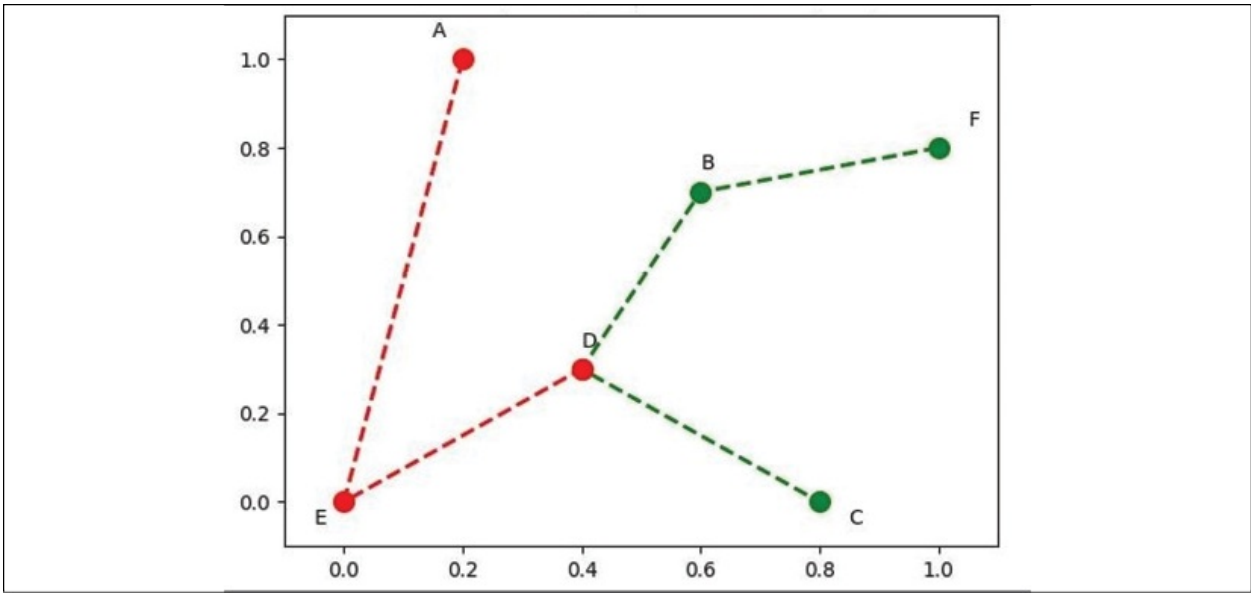


Figure 7.1: Markov decision process graph

Each node represents a company that takes part in an IBM Hyperledger network set up for those six companies, as described in the following table:

Company	**Activity**	**ML weight**
A buys and sells clothing and also other products in this network.	Provide goods to the network but keep stock levels down	Stock levels
B buys and sells fabric and also other products in this network.	Provide goods to the network but keep stock levels down	Stock levels
C buys and sells buttons and also other products in this network.	Provide goods to the network but keep stock levels down	Stock levels
D buys and sells printed fabric and also other products in this network.	Provide goods to the network but keep stock levels down	Stock levels
E buys and sells accessories (belts, bracelets) and also other products in this network.	Provide goods to the network but keep stock levels down	Stock levels

F buys and sells packaging boxes and also other products in this network.	Provide goods to the network but keep stock levels down	Stock levels

The structure of the table is as follows:

- **Company** contains six companies, {A, B, C, D, E, F}, that have different activities. Each company, in this example, only has one location, so A to F are company locations as well.
- **Activity**: Part of this group of companies to supply their members, but making sure the costly stock levels are kept down.
- **ML weight** represents a classification by ML of the stock levels to make predictions. The key to profit generation, in our example, is to track the stock levels of each member (A to F) of the supply chain. If the stock levels of company F go down, for example, the other members of the network (A to E) can anticipate and deliver only the limited necessary amount for F's stock levels to reach an acceptable level again.

With millions of commercial transactions per year with a huge amount of transportation (truck, train, boat, air), it is increasingly complicated to manage this type of network effectively in the 21st century without a solution like IBM Hyperledger.

> With IBM Hyperledger, the companies have **one** online transaction ledger with smart contracts (online) and real-time tracking.

The transactions are secure; they can be private or public among the members of the network, and they provide real-time optimization information for an artificial intelligence solution.

Using blockchains in the supply chain network

IBM Hyperledger provides artificial intelligence developers with a unique advantage over any other dataset—a 100% reliable dataset updated in real time.

Each company member (A to F) of the network will create a block for each transaction so that an AI analyst can have access to the data to make forecasts.

To make the system work, each member of the network (A to F) will create blocks in the blockchain so that others can view the information recorded and use the information to make decisions.

Creating a block

A block is formed using the method described for mining a bitcoin, except that this time, currency is not a goal. The goal is a secure transaction with a smart contract when necessary. The following screenshot is a standard IBM interface that can be customized:

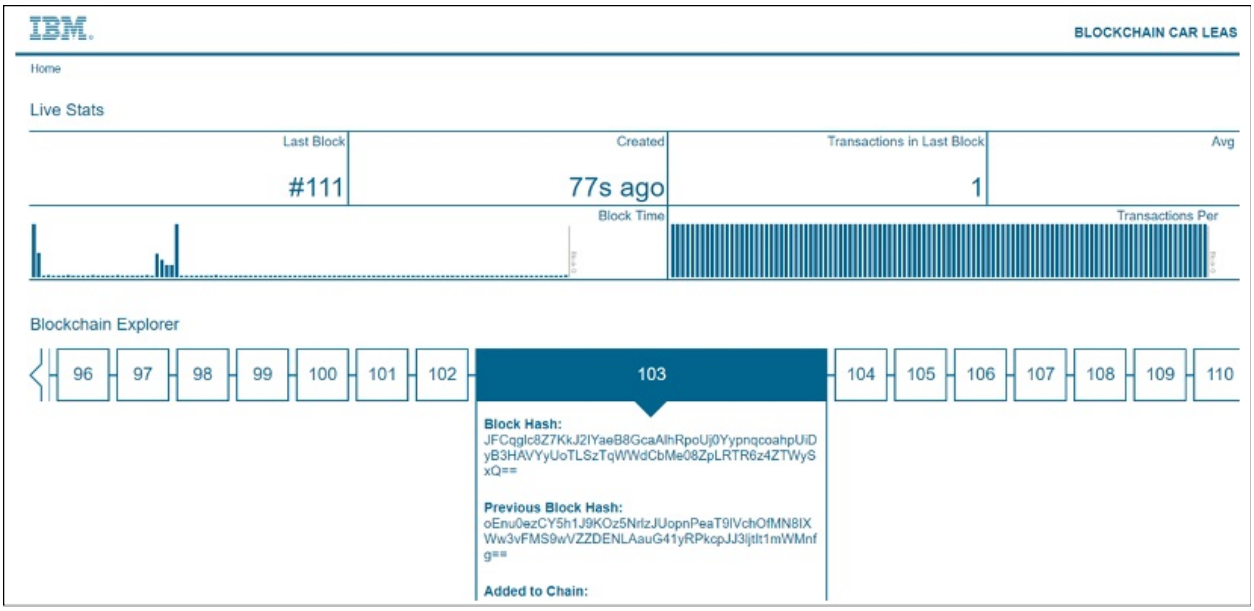


Figure 7.2: Standard IBM interface

You can see the individual and unique blocks that make up a blockchain. Each block in IBM Hyperledger has a unique number. In this example, it is **111** with a zoom on block **103**.

A block in the supply chain network in our graph (A to F) can be the purchase of a product X with a contract. The transaction can be between companies A and B, for example. The next block can be the transportation of that product to another location from A to B, for example, within the blockchain network.

The information attached to that block is in the Hyperledger repository: company name (A to F), address, phone number, transaction description, and any

other type of data required by the network of companies. Each block can be viewed by all or some depending on the permission properties attached to it.

Exploring the blocks

Exploring the blocks provides an artificial intelligence program with a gold mine: a real-life, and 100% reliable, dataset.

The interesting part for AI optimization is the block information, as described in the following screenshot. The present block was added to the chain along with the previous block and the transaction code:

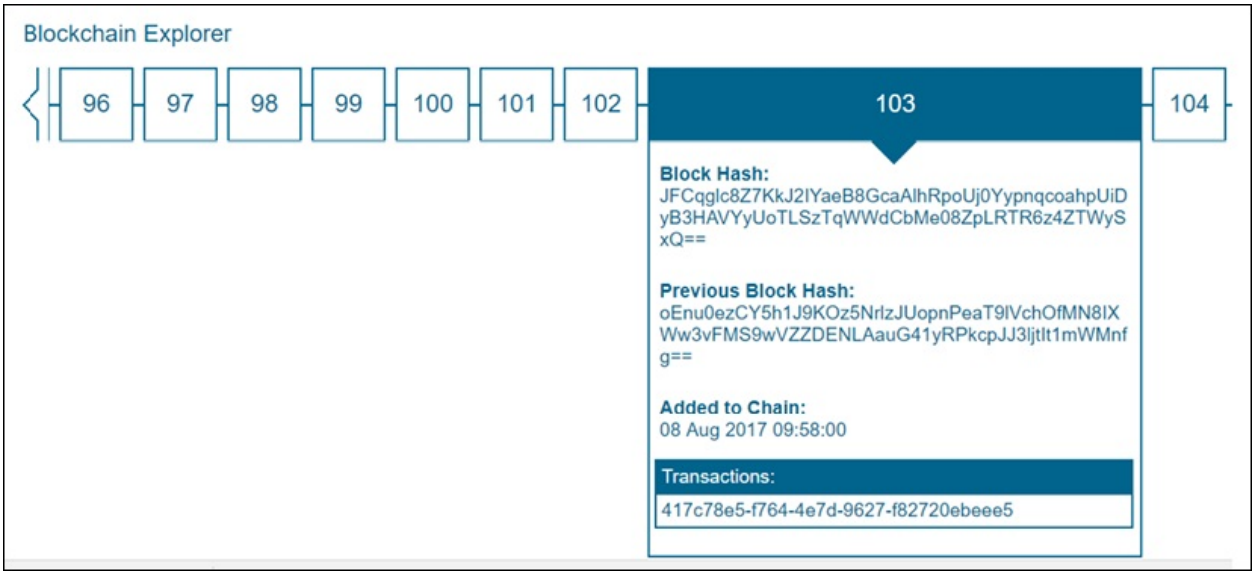


Figure 7.3: Exploring the blockchain

Notice that the block hash (see the preceding mining section) of a given block is linked to the previous block hash and possesses a unique transaction code.

Once we have our set of blocks as data (the type of transaction, date, amount), we can start to build an AI prediction algorithm. IBM provides extraction possibilities through scripts and other tools.

> Once we have a dataset containing blocks of the blockchain, the goal of the prediction will be to determine the stock level category to see whether a fellow member of the supply chain network (A to F) needs to be replenished.

The blocks of the blockchain provide data that can be used to predict the type of stock levels that the network will be facing after each transaction. Each company will be able to run a prediction algorithm that uses the existing data to predict the potential replenishing required. For example, company A might detect that company B needs more of its cloth.

Considering the amount of data, a prediction algorithm such as naive Bayes can do the job.

Part III – optimizing a supply chain with naive Bayes in a blockchain process

Naive Bayes will use some of the critical information as **features** to optimize warehouse storage and product availability in a real-time process.

The Naive Bayes learning function will learn from the previous blocks on how to predict the next blocks (supplying another company that needs more stock) that should be inserted in the blockchain. The blocks will be inserted in a dataset just like any other form of timestamped data to make predictions.

Naive Bayes is based on Bayes' theorem. Bayes' theorem applies conditional probability, defined as follows:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

- $P(A|B)$ is a **posterior probability**, the probability of A after having observed some events (B). It is also a **conditional probability**: the likelihood of A happening given B has already happened.
- $P(B|A)$ is the probability of B given the prior observations A. It is also a conditional probability: the likelihood of B happening given A has already happened.
- $P(A)$ is the probability of A prior to the observations.
- $P(B)$ is the probability of the predictions.

Naive Bayes, although based on Bayes' theorem, assumes that the features in a class are **independent** of each other. In many cases, this makes predictions more practical to implement. The statistical presence of features, related or not, will produce a prediction. As long as the prediction remains sufficiently efficient, naive Bayes provides a good solution.

A naive Bayes example

In this section, we will first illustrate naive Bayes with a mathematical example before writing the Python program. The goal of this section is just to understand the concepts involved in naive Bayes.

This section is not a development chapter but a chapter to understand the underlying concepts of the real-life example and the mathematics used to make predictions.

The blockchains in this chapter represent information on the stock levels based on manufactured goods in the apparel industry. For more, read *Chapter 12, AI and the Internet of Things (IoT)*, which describes an AI-optimized apparel manufacturing process in detail.

In this section, we will focus on storage. The load of a sewing station in the apparel industry is expressed in quantities in **stock keep units** (**SKUs**). An SKU, for example, can be product P: a pair of jeans of a given size.

Once the garment has been produced, it goes into storage. At that point, a block in a blockchain can represent that transaction with two useful features for a machine learning algorithm:

- The day the garment was stored
- The total quantity of that SKU garment now in storage

A block in a blockchain contains both the day (timestamp) and quantity stored in the transaction information.

Since the blockchain contains the storage blocks of all A, B, C, D, E, and F corporate locations that are part of the network, a machine learning program can access the data and make predictions.

The goal is to spread the stored quantities of the given product evenly over the six locations, as represented in the following histogram:

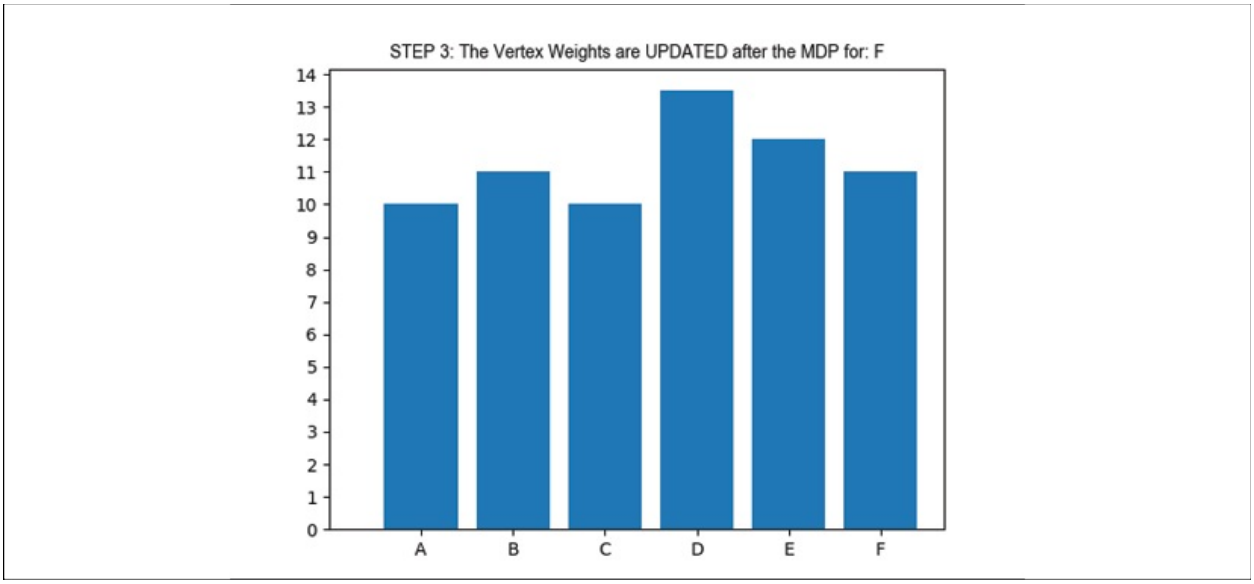


Figure 7.4: Histogram of the storage level of the product distributed over six locations

This screenshot shows the storage level of product P distributed over six locations. Each location in this blockchain network is a **hub**. A hub in **supply chain management** (**SCM**) is often an intermediate storage warehouse. For example, to cover the area of these six locations, the same product will be stored at each location. This way, local trucks can come and pick the goods for delivery.

The goal is to have an available product P at point location L (A to F) when needed.

For example, the delivery time from a location L to a location point a_1 (a store or home) will only take a few hours. If A did not store P, then the finer consumer would have to wait for the product to travel from C to A, for example.

If the blockchain network is well organized, one location can specialize in producing product P (best production costs) and evenly distribute the quantity stored in the six locations, including itself.

By having evenly balanced storage quantities at all locations, the system will flow in a continuous delivery process.

In our example, a company = its location (A to F). Using the blocks of the blockchain of these six members, we can predict when a given storage point (A to F) requires replenishing to reduce waiting times, for example.

Using blockchains to optimize the storage levels is an efficient new approach to reducing costs while delivering to customers faster.

The blockchain anticipation novelty

In former days, all the warehouses at those six locations (A to F) had to ensure the minimum storage level for each location. Since they did not know what was happening in their supply chain network in real time, they often stored more products than required, which increased their costs, or did not store enough, leading to delivery problems.

In a world of real-time production and selling, distributors need to predict **demand**. The system needs to be **demand-driven**. Naive Bayes can solve that problem.

It will take the first two features into account:

- **DAY**: The day the garment was stored
- **STOCK**: The total quantity of that SKU garment now in storage

Then it will add a novelty—*the number of blocks related to product P*.

A high number of blocks at a given date means that this product was in demand in general (production, distribution). The more blocks there are, the more transactions there are. Also, if the storage levels (the STOCK feature) are diminishing, this is an indicator; it means storage levels must be replenished. The DAY feature timestamps the history of the product.

The block feature is named **BLOCKS**. Since all share the blockchain, a machine learning program can access reliable global data in seconds. The dataset reliability provided by blockchains constitutes a motivation in itself to optimize storage levels using the blocks of the blockchains as datasets.

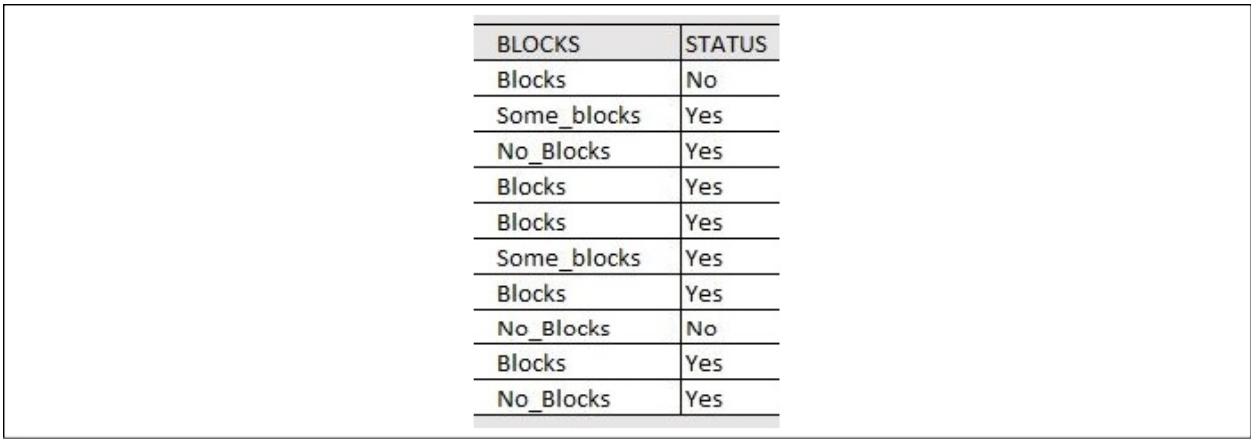
The goal – optimizing storage levels using blockchain data

The goal is to *maintain stock at low levels* by providing fast delivery when product requests are made. To make a decision, the ML solution will analyze the blocks of a blockchain in real time.

The program will take the DAY, STOCK, and BLOCKS (number of) features for a given product P and produce a result. The result predicts whether this product P will be in demand. If the answer is yes (or 1), the demand for this product requires anticipation.

Step 1 – defining the dataset

The dataset contains raw data from prior events in a sequence, which makes it perfect for prediction algorithms. Blocks can be extracted using scripts from IBM Hyperledger, for example. This constitutes a unique opportunity to see the data of all companies without having to build a database. The raw dataset will look like the following list:



BLOCKS	STATUS
Blocks	No
Some_blocks	Yes
No_Blocks	Yes
Blocks	Yes
Blocks	Yes
Some_blocks	Yes
Blocks	Yes
No_Blocks	No
Blocks	Yes
No_Blocks	Yes

Figure 7.5: Raw dataset

This dataset contains the following:

- `Blocks` of product P present in the blockchain on day `x` having scanned the blockchain back by 30 days. `No` means no significant amounts of blocks have been found. `Yes` means a significant number of blocks have been found. If blocks have been found, this means that there is a demand for this product somewhere along the blockchain.
- `Some_blocks` means that blocks have been found, but they are too sparse to be taken into account without overfitting the prediction. However, `Yes` will contribute to the prediction as well as `No`.
- `No_blocks` means there is no demand at all, sparse or otherwise (`Some_blocks`), numerous (`Blocks`) or not. This means trouble for this product, P.

The goal is to avoid predicting demand on sparse (`Some_blocks`) or absent (`No_blocks`) products. This example is trying to predict a potential `Yes` for numerous blocks for this product P. Only if `Yes` is predicted can the system trigger the automatic demand process (see the *Implementation of naive Bayes in Python* section later in the chapter).

Step 2 – calculating the frequency

Looking at the following frequency table provides additional information:

STATUS:	No	No	No	Yes	Yes	Yes
	Some_blocks	No_Blocks	Blocks	Some_blocks	No_Blocks	Blocks
			1			
				1		
					1	
						1
						1
				1		
						1
		1				
						1
					1	
FREQUENCY	0	1	1	2	2	4

Figure 7.6: Frequency table

The `Yes` and `No` statuses of each feature (`Blocks` , `Some_blocks` , or `No_blocks`) for a given product P for a given period (past 30 days) have been grouped by frequency.

The sum is on the bottom line for each `No` feature and `Yes` feature. For example, `Yes` and `No_blocks` add up to 2.

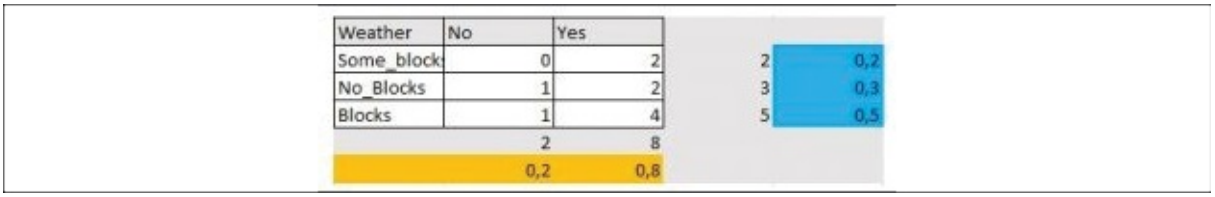
Some additional information will prove useful for the final calculation:

- The total number of samples = 10.
- The total number of `Yes` samples = 8.
- The total number of `No` samples = 2.

Step 3 – calculating the likelihood

Now that the **frequency** table has been calculated, the following **likelihood** table

is produced using that data:



Weather	No	Yes
Some_block	0	2
No_Blocks	1	2
Blocks	1	4
	2	8
	0,2	0,8

2	0,2
3	0,3
5	0,5

Figure 7.7: Likelihood table

The table contains the following statistics:

- `No` = 2 = 20% = 0.2
- `Yes` = 8 = 80%=0.8
- `Some_blocks` = 2 = 20%=0.2
- `No_blocks` = 3 = 30%=0.3
- `Blocks` = 5 = 50%=0.5

`Blocks` represent an important proportion of the samples, which means that along with `Some_blocks`, the demand looks good.

Step 4 – applying the naive Bayes equation

The goal now is to represent each variable of the Bayes' theorem in a naive Bayes equation to obtain the probability of having **demand** for product P and trigger a purchase scenario for the blockchain network. Bayes' theorem can be expressed as follows:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

- $P(Yes|Blocks) = P(Blocks|Yes) * P(Yes)/P(Blocks)$
- $P(Yes) = 8/10 = 0.8$
- $P(Blocks) = 5/10 = 0.5$
- $P(Blocks|Yes) = 4/8 = 0.5$
- $P(Yes|Blocks) = (0.5*0.8)/0.5 = 0.8$

The demand looks acceptable. However, penalties are necessary, and other

factors must be considered as well (transportation availability through other block exploration processes).

This example and method show the concept of the naive Bayes approach. This example is meant to be nothing other than a simplified mathematical explanation of the philosophy of Bayes' theorem.

By doing this calculation from scratch, we now know the basic concepts of Bayes' theorem: using prior values to predict future events taking several features into account.

We will now move from this theoretical approach to the implementation phase. We will build a program using naive Bayes in Python.

Implementation of naive Bayes in Python

This section shows how to use a stock level optimizer version of naive Bayes. Blockchains provide exceptionally reliable datasets for ML miners looking for areas to optimize and generate profit. Choosing the right model remains the key challenge.

Gaussian naive Bayes

We will be implementing Gaussian naive Bayes because it fits the apparel industry. For example, if you sell dresses, there will be a target middle size *S*. The marketing department knows that this size *S* will represent most of the sales. The larger *S* + *n* sizes and small *S* – *n* will generate fewer sales, creating a Gaussian curve.

In implementation mode, a dataset with raw data from the blockchain will be used without the feature interpretation function of naive Bayes in the following table:

DAY	STOCK	BLOCKS	DEMAND
10	1455	78	1
11	1666	67	1

12	1254	57	1
14	1563	45	1
15	1674	89	1
10	1465	89	1
12	1646	76	1
15	1746	87	2
12	1435	78	2

Each line represents a block:

- `DAY`: The day of the period analyzed in the dataset. In this case, we are analyzing the days of a given month, which represents a financial period. No other information is required to run a calculation. In other cases, a dd/mm/yyyy format can be used. You can also just use a counter (1 to *n*) from day 1 of the start of a period and run it over several weeks, months, or years.
- `STOCK`: The total inputs in a given location (A, B, or … F) found in the blocks and totaled on that day. Since this represents the inputs of one location and only one, no location information is required. In other cases, the location can be added.
- `BLOCKS`: The number of blocks containing product P for location A, for example.

  A high number of blocks in the `BLOCK` column and a low number of quantities in the `STOCK` column mean that demand is high.

- `DEMAND = 1`. The proof of demand is a transaction block that contains a purchase in the past. These transaction blocks provide vital information.

A low number of blocks in the `BLOCK` column, and a high number of quantities in the `STOCK` column, mean that the demand is low.

- `DEMAND = 2`. Proof that no transaction was found.

> **The limit of naive Bayes**
>
> In some cases, `DEMAND = 1` when the stock is high and the blocks are low. That's why strict correlation is not so useful. This would be the limit of naive Bayes, which just analyzes the statistics and learns how to predict, ignoring the actual conditional probabilities, the interactions between features.

In this section, we described the dataset and features that will be taken into account to write a naive Bayes program with ready-to-use algorithms.

The Python program

We now know what a blockchain is and how to use the blocks of a blockchain to optimize the stock levels of the locations of a network of companies (A to F). We also know the basic concepts of Bayes' theorem and naive Bayes.

With this in mind, we can build a Python program to predict the stock level categories of incoming blocks of a blockchain. These predictions will help local managers increase their stock levels to meet the demand of their partners (companies A to F) in the supply chain they are part of.

The Python `naive_bayes_blockchains.py` program uses a `sklearn` class. Consider the following snippet:

```
import numpy as np
import pandas as pd
from sklearn.naive_bayes import GaussianNB
```

It reads the dataset into the data structure. The following code reads `data_BC.csv` into `df`:

```
#Reading the data
df = pd.read_csv('data_BC.csv')
print("Blocks of the Blockchain")
```

```
print(df.head())
```

It prints the top of the file in the following output:

```
Blocks of the Blockchain
DAY STOCK BLOCKS DEMAND
0 10 1455 78 1
1 11 1666 67 1
2 12 1254 57 1
3 14 1563 45 1
4 15 1674 89 1
```

It prepares the training set, using `X` to find and predict `Y` in the following code:

```
# Prepare the training set
X = df.loc[:,'DAY':'BLOCKS']
Y = df.loc[:,'DEMAND']
```

It chooses the class and trains the following `clfG` model:

```
#Choose the class
clfG = GaussianNB()
# Train the model
clfG.fit(X,Y)
```

The program then takes some blocks of the blockchain, makes predictions, and prints them using the following `clfG.predict` function:

```
# Predict with the model(return the class)
print("Blocks for the prediction of the A-F blockchain")
blocks=[[14,1345,12],
        [29,2034,50],
        [30,7789,4],
        [31,6789,4]]
print(blocks)
prediction = clfG.predict(blocks)
for i in range(4):
    print("Block #",i+1," Gauss Naive Bayes Prediction:",
          prediction[i])
```

The blocks are displayed, and the following predictions are produced. `2` means

no demand for the moment; `1` will trigger a purchase block:

```
Blocks for the prediction of the A-F blockchain
[[14, 1345, 12], [29, 2034, 50], [30, 7789, 4], [31, 6789, 4
]]
Block # 1 Gauss Naive Bayes Prediction: 1
Block # 2 Gauss Naive Bayes Prediction: 2
Block # 3 Gauss Naive Bayes Prediction: 2
Block # 4 Gauss Naive Bayes Prediction: 2
```

This is a replenishment program. It will mimic the demand. When no demand is found, nothing happens; when demand is found, it triggers a purchase block. Some chain stores know the number of garments purchased on a given day (or week or another unit) and automatically purchase that amount. Others have other purchasing rules. Finding business rules is part of the consulting aspect of a project.

In this section, we implemented naive Bayes to predict the categories of the incoming blocks. If the demand is high, then a supply chain manager will know that more products need to be stored.

If the demand is low, the manager will avoid storing more products. Blocks in a blockchain provide reliable datasets for a program that scans the blocks 24/7 and generates recommendations in real time.

Summary

The reliable sequence of blocks in a blockchain has opened the door to endless machine learning algorithms. Naive Bayes appears to be a practical way to start optimizing the blocks of a blockchain. It calculates correlations and makes predictions by learning the independent features of a dataset, irrespective of whether the relationship is conditional or not.

This freestyle prediction approach fits the open-minded spirit of blockchains that are being propagated by the millions today with limitless resources.

IBM Hyperledger takes blockchain's "Frontierland" development to another

level with the Linux Foundation project. IBM also offers a cloud platform and services.

IBM, Microsoft, Amazon, and Google provide cloud platforms with an arsenal of disruptive machine learning algorithms. This provides a smooth approach to your market or department, along with the ability to set up a blockchain prototype online in a short space of time. With this approach, you can enter some additional prototype data in the model, export the data, or use an API to read the block sequences. Then, you will be able to apply machine learning algorithms to these reliable datasets. The only limit is our imagination.

The next chapter will lead us into more AI power as we explore the world of neural networks.

Questions

1. Cryptocurrency is the only use of blockchains today. (Yes | No)
2. Mining blockchains can be lucrative. (Yes | No)
3. Blockchains for companies cannot be applied to sales. (Yes | No)
4. Smart contracts for blockchains are more accessible to write than standard offline contracts. (Yes | No)
5. Once a block is in a blockchain network, everyone in the network can read the content. (Yes | No)
6. A block in a blockchain guarantees that absolutely no fraud is possible. (Yes | No)
7. There is only one way of applying Bayes' theorem. (Yes | No)
8. Training a naive Bayes dataset requires a standard function. (Yes | No)
9. Machine learning algorithms will not change the intrinsic nature of the corporate business. (Yes | No)

Further reading

- For more on naive Bayes on scikit-learn's website: https://scikit-learn.org/stable/modules/naive_bayes.html
- To explore IBM's Hyperledger solution: https://www.ibm.com/blockchain/hyperledger.html

8

Solving the XOR Problem with a Feedforward Neural Network

In the course of a corporate project, there always comes the point when a problem that seems impossible to solve hits you. At that point, you try everything you've learned, but it doesn't work for what's asked of you. Your team or customer begins to look elsewhere. It's time to react.

In this chapter, an impossible-to-solve business case regarding material optimization will be resolved successfully with a hand-made version of a **feedforward neural network** (**FNN**) with backpropagation.

Feedforward networks are one of the key building blocks of deep learning. The battle around the XOR function perfectly illustrates how deep learning regained popularity in corporate environments. XOR is an exclusive OR function that we will explore later in this chapter. The XOR FNN illustrates one of the critical functions of neural networks: **classification**. Once information becomes classified into subsets, it opens the doors to **prediction** and many other functions of neural networks, such as representation learning.

An XOR FNN will be built from scratch to demystify deep learning from the start. A vintage, start-from-scratch method will be applied, blowing the deep learning hype off the table.

The following topics will be covered in this chapter:

- Explaining the XOR problem
- How to hand-build an FNN
- Solving XOR with an FNN
- Classification

- Backpropagation
- A cost function
- Cost function optimization
- Error loss
- Convergence

Before we begin building an FNN, we'll first introduce XOR and its limitations in the first artificial neural model.

The original perceptron could not solve the XOR function

The original perceptron was designed in the 1950s and improved in the late 1970s. The original perceptron contained one neuron that could not solve the XOR function.

An XOR function means that you have to choose an exclusive OR (XOR).

This can be difficult to grasp, as we're not used to thinking about the way in which we use *or* in our everyday lives. In truth, we use *or* interchangeably as either inclusive or exclusive all of the time. Take this simple example:

If a friend were to come and visit me, I may ask them, "Would you like tea or coffee?" This is basically the offer of tea XOR coffee; I would not expect my friend to ask for both tea and coffee! My friend will choose one or the other.

I may follow up my question with, "Would you like milk or sugar?" In this case, I would not be surprised if my friend wanted both. This is an inclusive *or*.

XOR, therefore, means "You can have one or the other, but not both."

We will develop these concepts in the chapter through more examples.

To solve this XOR function, we will build an FNN.

Once the feedforward network for solving the XOR problem is built, it will be applied to an optimization example. The material optimizing example will choose the best combinations of dimensions among billions to minimize the use of corporate resources with the generalization of the XOR function.

First, a solution to the XOR limitation of a perceptron must be clarified.

XOR and linearly separable models

In the late 1960s, it was mathematically proven that a perceptron could *not* solve an XOR function. Fortunately, today, the perceptron and its neocognitron version form the core model for neural networking.

You may be tempted to think, *so what?* However, the entire field of neural networks relies on solving problems such as this to classify patterns. Without pattern classification, images, sounds, and words mean nothing to a machine.

Linearly separable models

The McCulloch-Pitts 1943 neuron (see *Chapter 2*, *Building a Reward Matrix – Designing Your Datasets*) led to Rosenblatt's 1957-59 perceptron and the 1960 Widrow-Hoff adaptive linear element (Adaline).

These models are linear models based on an $f(x, w)$ function that requires a line to separate results. A perceptron cannot achieve this goal and thus cannot classify many objects it faces.

A standard linear function can separate values. **Linear separability** can be represented in the following graph:

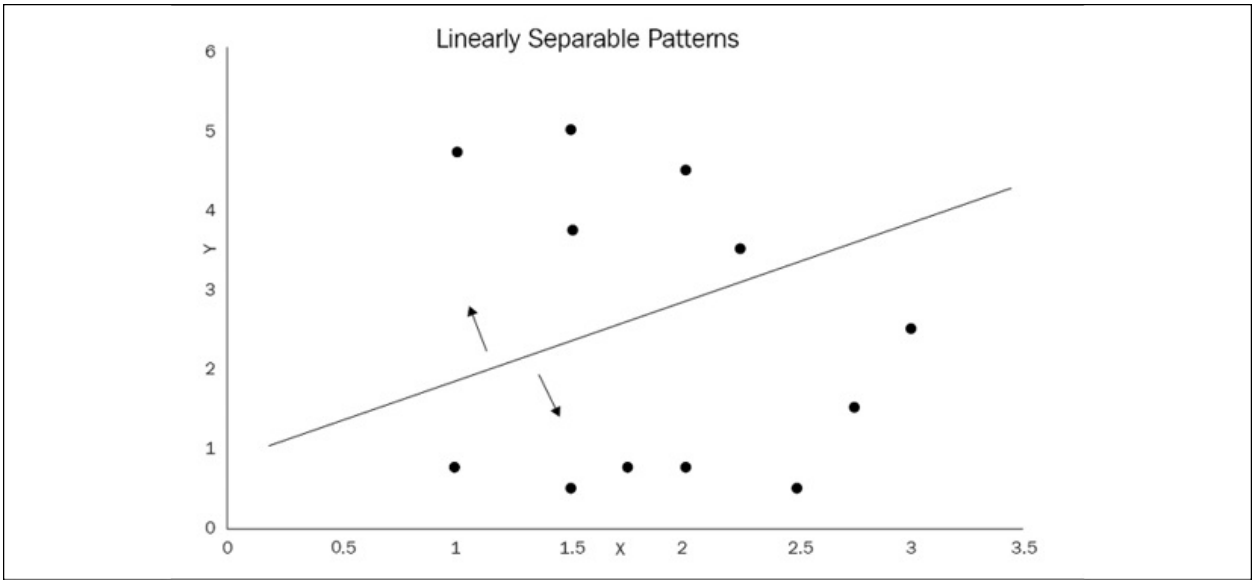


Figure 8.1: Linearly separable patterns

Imagine that the line separating the preceding dots and the part under it represent a picture that needs to be represented by a machine learning or deep learning application. The dots above the line represent *clouds* in the sky; the dots below the line represent *trees* on a hill. The line represents the slope of that hill.

To be linearly separable, a function must be able to separate the *clouds* from the *trees* to classify them. The prerequisite to classification is **separability** of some sort, linear or nonlinear.

The XOR limit of a linear model, such as the original perceptron

A linear model cannot solve the XOR problem expressed as follows in a table:

Value of x1	Value of x2	Output
1	1	0
0	0	0
1	0	1

0	1	1

Lines 3 and 4 show an exclusive OR (XOR). Imagine that you are offering a child a piece of cake OR a piece of candy (1 or 1):

- **Case 1**: The child answers: "I want candy or nothing at all!" (0 or 1). That's exclusive OR (XOR)!
- **Case 2**: The child answers: "I want a cake or nothing at all!" (1 or 0). That's an exclusive OR (XOR) as well!

The following graph shows the linear inseparability of the XOR function represented by one perceptron:

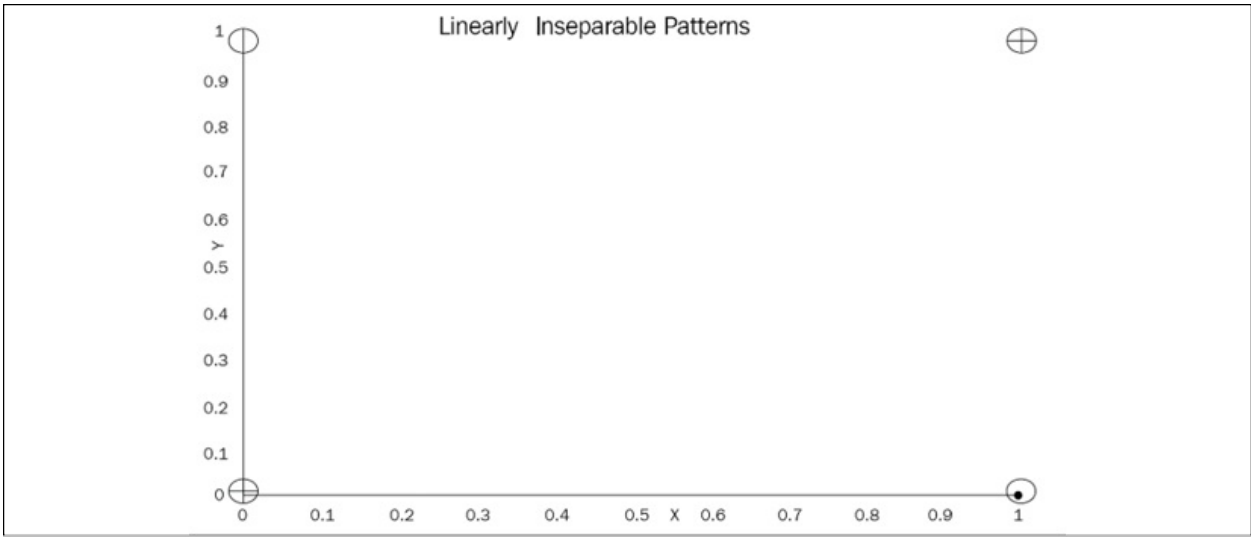


Figure 8.2: Linearly inseparable patterns

The values of the table represent the Cartesian coordinates in this graph. The circles with a cross at (1, 1) and (0, 0) cannot be separated from the circles at (1, 0) and (0, 1). That's a huge problem. It means that Frank Rosenblatt's *f*(*x*, *w*) perceptron cannot separate, and thus can not classify, these dots into *clouds* and *trees*. Thus, in many cases, the perceptron cannot identify values that require linear separability.

Having invented the most powerful neural concept of the twentieth century—a neuron that can learn—Frank Rosenblatt had to bear with this limitation through the 1960s.

As explained with the preceding cake-OR-candy example, the absence of an XOR function limits applications in which you must choose exclusively between two options. There are many "it's-either-that-or-nothing" situations in real-life applications. For a self-driving car, it could be either turn left or turn right, but don't swerve back and forth while making the decision!

We will solve this limitation with a vintage solution, starting by building, and later implementing, an FNN.

Building an FNN from scratch

Let's perform a mind experiment. Imagine we are in 1969. We have today's knowledge but nothing to prove it. We know that a perceptron cannot implement the exclusive OR function XOR.

We have an advantage because we now know a solution exists. To start our experiment, we only have a pad, a pencil, a sharpener, and an eraser waiting for us. We're ready to solve the XOR problem from scratch on paper before programming it. We have to find a way to classify those dots with a neural network.

Step 1 – defining an FNN

We have to be unconventional to solve this problem. We must forget the complicated words and theories of the twenty-first century.

We can write a neural network layer in high-school format. A hidden layer will be:

$$h_1 = x * w$$

OK. Now we have one layer. A layer is merely a function. This function can be expressed as:

$$f(x, w)$$

In which x is the input value, and w is some value to multiply x by. Hidden means that the computation is not visible, just as $x = 2$ and $x + 2$ is the hidden layer that leads to 4.

At this point, we have defined a neural network in three lines:

- Input x.
- Some function that changes its value, like $2 \times 2 = 4$, which transformed 2. That is a layer. And if the result is superior to 2, for example, then great! The output is 1, meaning yes or true. Since we don't see the computation, this is the *hidden* layer.
- An output.

$f(x, w)$ is the building block of any neural network. "Feedforward" means that we will be going from layer 1 to layer 2, moving forward in a sequence.

Now that we know that basically any neural network is built with values transformed by an operation to become an output of something, we need some logic to solve the XOR problem.

Step 2 – an example of how two children can solve the XOR problem every day

An example follows of how two children can solve the XOR problem using a straightforward everyday example. I strongly recommend this method. I have taken very complex problems, broken them down into small parts to a child's level, and often solved them in a few minutes. Then, you get the sarcastic answer from others such as "Is that all you did?" But, the sarcasm vanishes when the solution works over and over again in high-level corporate projects.

First, let's convert the XOR problem into a candy problem in a store. Two children go to the store and want to buy candy. However, they only have enough money to buy one pack of candy. They have to agree on a choice between two packs of different candy. Let's say pack one is chocolate and the other is chewing gum. Then, during the discussion between these two children, 1 means yes, 0 means no. Their budget limits the options of these two children:

- Going to the store and not buying any chocolate **or** chewing gum = (no, no) = (0, 0). That's not an option for these children! So the answer is false.
- Going to the store and buying both chocolate **and** chewing gum = (yes, yes) = (1, 1). That would be fantastic, but that's not possible. It's too expensive. So, the answer is, unfortunately, false.
- Going to the store and either buying chocolate **or** chewing gum = (1, 0 or 0, 1) = (yes or no) or (no or yes). That's possible. So, the answer is true.

Imagine the two children. The eldest one is reasonable. The younger one doesn't really know how to count yet and wants to buy both packs of candy.

We express this on paper:

- x_1 (eldest child's decision, yes or no, 1 or 0) * w_1 (what the elder child thinks). The elder child is thinking this, or:

  $$x_1 * w_1 \text{ or } h_1 = x_1 * w_1$$

  The elder child weighs a decision like we all do every day, such as purchasing a car (x = 0 or 1) multiplied by the cost (w_1).

- x_2 (the younger child's decision, yes or no, 1 or 0) * w_3 (what the younger child thinks). The younger child is also thinking this, or:

  $$x_2 * w_3 \text{ or } h_2 = x_2 * w_3$$

> **Theory**: x_1 and x_2 are the inputs. h_1 and h_2 are neurons (the result of a calculation). Since h_1 and h_2 contain calculations that are not visible during the process, they are hidden neurons. h_1 and h_2 thus form a hidden layer. w_1 and w_3 are weights that represent how we "weigh" a decision, stating that something is more important than something else.

Now imagine the two children talking to each other.

Hold it a minute! This means that now, each child is communicating with the other:

- x_1 (the elder child) says w_2 to the younger child. Thus, w_2 = this is what I think and am telling you:

$$x_1 * w_2$$

- x_2 (the younger child) says, "please add my views to your decision," which is represented by $w_{4:}$

$$x_2 * w_4$$

We now have the first two equations expressed in high-school-level code. It's what one thinks plus what one says to the other, asking the other to take that into account:

```
h1=(x1*w1)+(x2*w4) #II.A.weight of hidden neuron h1
h2=(x2*w3)+(x1*w2) #II.B.weight of hidden neuron h2
```

`h1` sums up what is going on in one child's mind: personal opinion + the other child's opinion.

`h2` sums up what is going on in the other child's mind and conversation: personal opinion + the other child's opinion.

> **Theory**: The calculation now contains two input values and one hidden layer. Since, in the next step, we are going to apply calculations to `h1` and `h2`, we are in a feedforward neural network. We are moving from the input to another layer, which will lead us to another layer, and so on. This process of going from one layer to another is the basis of deep learning. The more layers you have, the deeper the network is. The reason `h1` and `h2` form a hidden layer is that their output is just the input of another layer.

For this example, we don't need complicated numbers in an activation function such as logistic sigmoid, so we state whether the output values are less than 1 or not:

if $h_1 + h_2 >= 1$ then $y_1 = 1$

if $h_1 + h_2 < 1$ then $y_2 = 0$

> **Theory**: y_1 and y_2 form a second hidden layer. These variables can be scalars, vectors, or matrices. They are neurons.

Now, a problem comes up. Who is right? The elder child or the younger child?

The only way seems to be to play around, with the weights *W* representing all the weights. Weights in a neural network work like weights in our everyday lives. We *weigh* decisions all the time. For example, there are two books to purchase, and we will "weigh" our decisions. If one is interesting and cheaper, it will weigh more or less in our decision, for example.

The children in our case agree on purchasing at least something, so from now on, $w_3 = w_2$, $w_4 = w_1$. The younger and elder child will thus share some of the decision weights.

Now, somebody has to be an influencer. Let's leave this hard task to the elder child. The elder child, being more reasonable, will continuously deliver the bad news. You have to subtract something from your choice, represented by a minus (–) sign.

Each time they reach the point h_i, the eldest child applies a critical negative view on purchasing packs of candy. It's $–w$ of everything comes up to be sure not to go over the budget. The opinion of the elder child is biased, so let's call the variable a bias, b_1. Since the younger child's opinion is biased as well, let's call this view a bias too, b_2. Since the eldest child's view is always negative, $–b_1$ will be applied to all of the eldest child's thoughts.

When we apply this decision process to their view, we obtain:

$$h_1 = y_1 * –b_1$$

$$h_2 = y_2 * b_2$$

Then, we just have to use the same result. If the result is >=1, then the threshold has been reached. The threshold is calculated as shown in the following function:

$$y = h_1 + h_2$$

We will first start effectively finding the weights, starting by setting the weights and biases to 0.5, as follows:

$$w_1 = 0.2;\ w_2 = 0.5;\ b_1 = 0.5$$

$$w_3 = w_2;\ w_4 = w_1;\ b_2 = b_1$$

It's not a full program yet, but its theory is done.

Only the communication going on between the two children is making the difference; we will focus on only modifying w_2 and b_1 after a first try. It works on paper after a few tries.

We now write the basic mathematical function, which is, in fact, the program itself on paper:

```
#Solution to the XOR implementation with
#a feedforward neural network(FNN)
#I.Setting the first weights to start the process
w1=0.5;w2=0.5;b1=0.5
w3=w2;w4=w1;b2=b1
#II.hidden layer #1 and its output
h1=(x1*w1)+(x2*w4) #II.A.weight of hidden neuron h1
h2=(x2*w3)+(x1*w2) #II.B.weight of hidden neuron h2
#III.threshold I, hidden layer 2
if(h1>=1): h1=1
if(h1<1): h1=0
if(h2>=1): h2=1
if(h2<1): h2=0
h1= h1 * -b1
h2= h2 * b2
#IV.Threshold II and Final OUTPUT y
y=h1+h2
if(y>=1): y=1
if(y<1): y=0
#V.Change the critical weights and try again until a soluti
on is found
w2=w2+0.5
b1=b1+0.5
```

Let's go from the solution on paper to Python.

> Why wasn't this deceivingly simple solution found in 1969? Because *it seems simple today but wasn't so at that time*, like all inventions found by our genius predecessors. Nothing is easy at all in artificial intelligence and mathematics.

In the next section, we'll stick with the solution proposed here, and implement it in Python.

Implementing a vintage XOR solution in Python with an FNN and backpropagation

To stay in the spirit of a 1969 vintage solution, we will not use NumPy, TensorFlow, Keras, or any other high-level library. Writing a vintage FNN with backpropagation written in high-school mathematics is fun.

If you break a problem down into very elementary parts, you understand it better and provide a solution to that specific problem. You don't need to use a huge truck to transport a loaf of bread.

Furthermore, by thinking through the minds of children, we went against running 20,000 or more episodes in modern CPU-rich solutions to solve the XOR problem. The logic used proves that both inputs can have the same parameters as long as one bias is negative (the elder reasonable critical child) to make the system provide a reasonable answer.

The basic Python solution quickly reaches a result in a few iterations, approximately 10 iterations (epochs or episodes), depending on how we think it through. An epoch can be related to a try. Imagine looking at somebody practicing basketball:

- The person throws the ball toward the hoop but misses. That was an epoch (an episode can be used as well).
- The person thinks about what happened and changes the way the ball will be thrown.

This improvement is what makes it a learning epoch (or episode). It is not a simple memoryless try. Something is really happening to improve performance.

- The person throws the ball again (next epoch) and again (next epochs) until the overall performance has improved. This is how a neural network improves over epochs.

`FNN_XOR_vintage_tribute.py` contains (at the top of the code) a result matrix with four columns.

Each element of the matrix represents the status (`1` = correct, `0` = false) of the four predicates to solve:

```
#FEEDFORWARD NEURAL NETWORK(FNN) WITH BACK PROPAGATION SOLU
TION FOR XOR
result=[0,0,0,0] #trained result
train=4 #dataset size to train
```

The `train` variable is the number of predicates to solve: (0, 0), (1, 1), (1, 0), (0, 1). The variable of the predicate to solve is `pred`.

The core of the program is practically a copy of the sheet of paper we wrote, as in the following code:

```
#II hidden layer 1 and its output
def hidden_layer_y(epoch,x1,x2,w1,w2,w3,w4,b1,b2,pred,result
):
    h1=(x1*w1)+(x2*w4) #II.A.weight of hidden neuron h1
    h2=(x2*w3)+(x1*w2) #II.B.weight of hidden neuron h2
#III.threshold I,a hidden layer 2 with bias
    if(h1>=1):h1=1;
    if(h1<1):h1=0;
    if(h2>=1):h2=1
    if(h2<1):h2=0
    h1= h1 * -b1
    h2= h2 * b2

#IV. threshold II and OUTPUT y
    y=h1+h2
    if(y<1 and pred>=0 and pred<2):
        result[pred]=1
```

```
    if(y>=1 and pred>=2 and pred<4):
        result[pred]=1
```

`pred` is an argument of the function from `1` to `4`. The four predicates are represented in the following table:

Predicate (pred)	x_1	x_2	Expected result
0	1	1	0
1	0	0	0
2	1	0	1
3	0	1	1

That is why *y* must be <1 for predicates 0 and 1. Then, *y* must be >=1 for predicates 2 and 3.

Now, we have to call the following function limiting the training to 50 epochs, which are more than enough:

```
#I Forward and backpropagation
for epoch in range(50):
    if(epoch<1):
        w1=0.5;w2=0.5;b1=0.5
    w3=w2;w4=w1;b2=b1
```

At the first epoch, the weights and biases are all set to `0.5`. No use thinking! Let the program do the job. As explained previously, the weight and bias of `x2` are equal.

Now, the hidden layers and `y` calculation function are called four times, one for each predicate to train, as shown in the following code snippet:

```
#I.A forward propagation on epoch 1 and IV.backpropagation
starting epoch 2
```

```
    for t in range (4):
        if(t==0):x1 = 1;x2 = 1;pred=0
        if(t==1):x1 = 0;x2 = 0;pred=1
        if(t==2):x1 = 1;x2 = 0;pred=2
        if(t==3):x1 = 0;x2 = 1;pred=3
        #forward propagation on epoch 1
        hidden_layer_y(epoch,x1,x2,w1,w2,w3,w4,b1,b2,pred,r
esult)
```

Now, the system must train. To do that, we need to measure the number of predictions, 1 to 4, that are correct at each iteration and decide how to change the weights/biases until we obtain proper results. We'll do that in the following section.

A simplified version of a cost function and gradient descent

Slightly more complex gradient descent will be described in the next chapter. In this chapter, only a one-line equation will do the job. The only thing to bear in mind as an unconventional thinker is: *so what?* The concept of gradient descent is minimizing loss or errors between the present result and a goal to attain.

First, a cost function is needed.

There are four predicates (0-0, 1-1, 1-0, 0-1) to train correctly. We need to find out how many are correctly trained at each epoch.

The cost function will measure the difference between the training goal (4) and the result of this epoch or training iteration (result).

When 0 convergence is reached, it means the training has succeeded.

`result[0,0,0,0]` contains a `0` for each value if none of the four predicates have been trained correctly. `result[1,0,1,0]` means two out of the four predicates are correct. `result[1,1,1,1]` means that all four predicates have been trained and that the training can stop. `1`, in this case, means that the correct training result was obtained. It can be `0` or `1`. The `result` array is the result counter.

The cost function will express this training by having a value of `4`, `3`, `2`, `1`, or

`0` as the training goes down the slope to 0.

Gradient descent measures the value of the descent to find the direction of the slope: up, down, or 0. Then, once you have that slope and the steepness of it, you can optimize the weights. A derivative is a way to know whether you are going up or down a slope.

Each time we move up or down the slope, we check to see whether we are moving in the right direction. We will assume that we will go one step at a time. So if we change directions, we will change our pace by one step. That one step value is our **learning rate**. We will measure our progression at each step. However, if we feel comfortable with our results, we might walk 10 steps at a time and only check to see if we are on the right track every 10 steps. Our learning rate will thus have increased to 10 steps.

In this case, we hijacked the concept and used it to set the learning rate to `0.05` with a one-line function. Why not? It helped to solve gradient descent optimization in one line:

```
    if(convergence<0):w2+=training_step;b1=w2
```

By applying the vintage children-buying-candy logic to the whole XOR problem, we found that only `w2` needed to be optimized. That's why `b1=w2`. That's because `b1` is doing the tough job of saying something negative (`-`) all the time, which completely changes the course of the resulting outputs.

The rate is set at `0.05`, and the program finishes training in 10 epochs:

```
epoch: 10 optimization 0 w1: 0.5 w2: 1.0 w3: 1.0 w4: 0.5 b1
: -1.0 b2: 1.0
```

This is a logical *yes* or *no* problem. The way the network is built is pure logic. Nothing can stop us from using whatever training rates we wish. In fact, that's what gradient descent is about. There are many gradient descent methods. If you invent your own and it works for your solution, that is fine.

This one-line code is enough, in this case, to see whether the slope is going down. As long as the slope is negative, the function is going downhill to *cost* = 0:

```
    convergence=sum(result)-train #estimating the direction
of the slope
    if(convergence>=-0.00000001): break
```

The following diagram sums up the whole process:

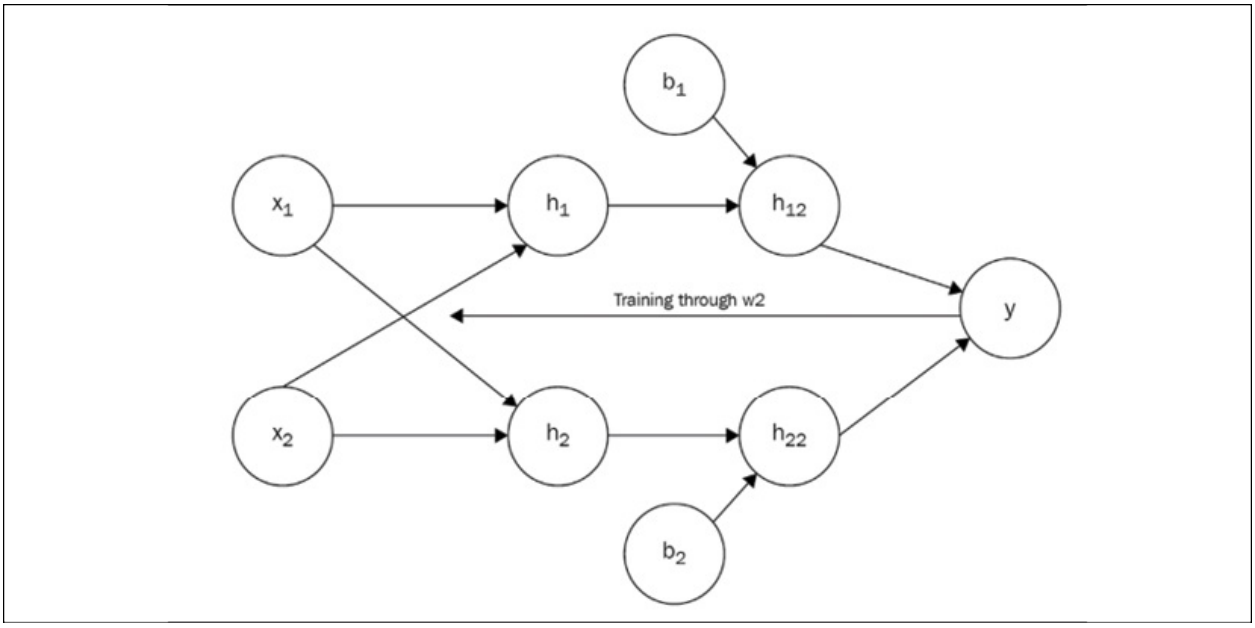


Figure 8.3: A feedforward neural network model (FNN)

We can see that all of the arrows of the layers go forward in this "feedforward" neural network. However, the arrow that stems from the *y* node and goes backward can seem confusing. This line represents a change in weights to train the model. This means that we go back to changing the weights and running the network for another epoch (or episode). The system is adjusting its weights epoch by epoch until the overall result is correct.

Too simple? Well, it works, and that's all that counts in real-life development. If your code is bug-free and does the job, then that's what matters.

Finding a simple development tool means nothing more than that. It's just another tool in the toolbox. We can get this XOR function to work on a neural network and generate income.

> Companies are not interested in how smart you are but how efficient (profitable) you can be.

A company's survival relies on multiple constraints: delivering on time, offering good prices, providing a product with a reasonable quality level, and many more factors besides.

When we come up with a solution, it is useless to show how smart we can be writing tons of code. Our company or customers expect an efficient solution that will run well and is easy to maintain. In short, focus on efficiency. Once we have a good solution, we need to show that it works. In this case, we proved that linear separability was achieved.

Linear separability was achieved

Bear in mind that the whole purpose of this feedforward network with backpropagation through a cost function was to transform a linear non-separable function into a linearly separable function to implement the classification of features presented to the system. In this case, the features had a `0` or `1` value.

One of the core goals of a layer in a neural network is to make the input make sense, meaning to be able to separate one kind of information from another.

`h1` and `h2` will produce the Cartesian coordinate linear separability training axis, as implemented in the following code:

```
    h1= h1 * -b1
    h2= h2 * b2
    print(h1,h2)
```

Running the program provides a view of the nonlinear input values once the hidden layers have trained them. The nonlinear values then become linear values in a linearly separable function:

```
linearly separability through cartesian training -1.0000000
000000004 1.0000000000000004
linearly separability through cartesian training -0.0 0.0
linearly separability through cartesian training -0.0 1.000
0000000000004
linearly separability through cartesian training -0.0 1.000
0000000000004
epoch: 10 optimization 0 w1: 0.5 w2: 1.0 w3: 1.0 w4: 0.5 b1
```

```
: -1.0 b2: 1.0
```

The intermediate result and goal are not a bunch of numbers on a screen to show that the program works. The result is a set of Cartesian values that can be represented in the following linearly separated graph:

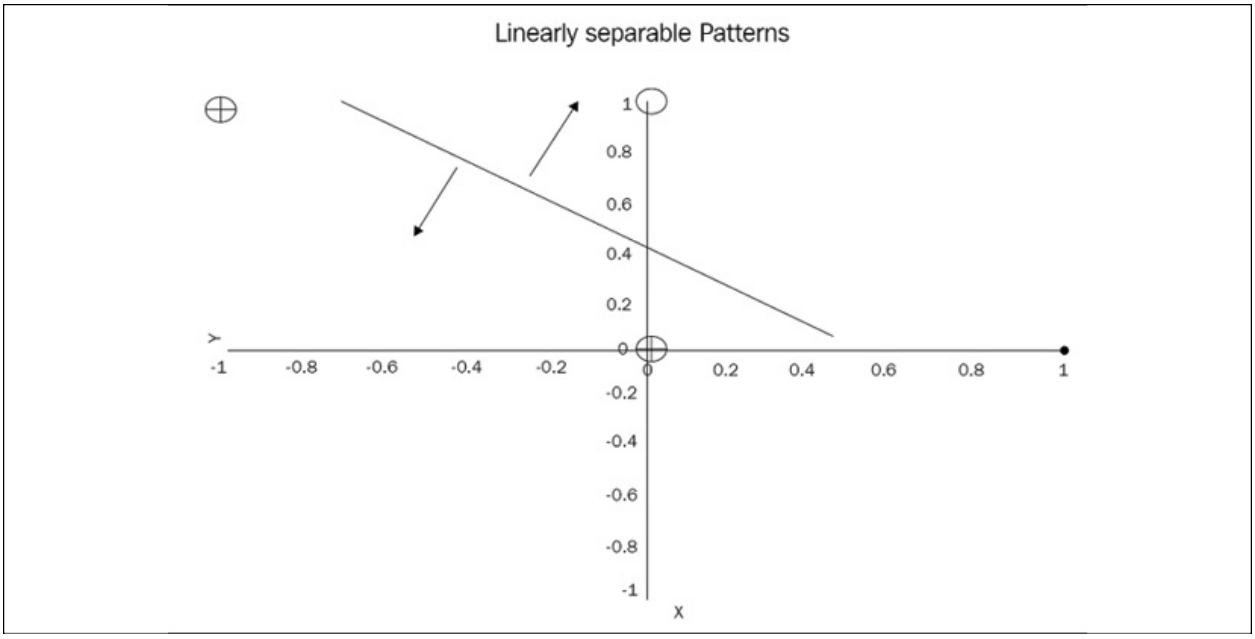


Figure 8.4: Linearly separable patterns

We have now obtained a separation between the top values, representing the intermediate values of the (1, 0) and (0, 1) inputs, and the bottom values, representing the (1, 1) and (0, 0) inputs. The top values are separated from the bottom values by a clear line. We now have *clouds* on top and *trees* below the line that separates them.

The layers of the neural network have transformed nonlinear values into linearly separable values, making classification possible through standard separation equations, such as the one in the following code:

```
#IV. threshold II and OUTPUT y
    y=h1+h2 # logical separation
    if(y<1 and pred>=0 and pred<2):
        result[pred]=1
    if(y>=1 and pred>=2 and pred<4):
        result[pred]=1
```

The ability of a neural network to make non-separable information separable and classifiable represents one of the core powers of deep learning. From this technique, many operations can be performed on data, such as subset optimization.

In the next section, we'll look at a practical application for our FNN XOR solution.

Applying the FNN XOR function to optimizing subsets of data

There are more than 7.5 billion people breathing air on this planet. In 2050, there might be 2.5 billion more of us. All of these people need to wear clothes and eat. Just those two activities involve classifying data into subsets for industrial purposes.

Grouping is a core concept for any production. Production relating to producing clothes and food requires grouping to optimize production costs. Imagine not grouping and delivering one T-shirt at a time from one continent to another instead of grouping T-shirts in a container and grouping many containers (not just two on a ship). Let's focus on clothing, for example.

A chain of stores needs to replenish the stock of clothing in each store as the customers purchase their products. In this case, the corporation has 10,000 stores. The brand produces jeans, for example. Their average product is a faded jean. This product sells a slow 50 units a month per store. That adds up to 10,000 stores × 50 units = 500,000 units or stock-keeping units (SKUs) per month. These units are sold in all sizes, grouped into average, small, and large. The sizes sold per month are random.

The main factory for this product has about 2,500 employees producing those jeans at an output of about 25,000 jeans per day. The employees work in the following main fields: cutting, assembling, washing, lasering, packaging, and warehousing.

The first difficulty arises with the purchase and use of fabric. The fabric for this

brand is not cheap. Large amounts are necessary. Each pattern (the form of pieces of the pants to be assembled) needs to be cut by wasting as little fabric as possible.

Imagine you have an empty box you want to fill up to optimize the volume. If you only put soccer balls in it, there will be a lot of space. If you slip tennis balls in the empty spaces, space will decrease. If, on top of that, you fill the remaining empty spaces with ping pong balls, you will have optimized the available space in the box.

> Building optimized subsets can be applied to containers, warehouse flows and storage, truckload optimizing, and almost all human activities.

In the apparel business, if 1% to 10% of the fabric is wasted while manufacturing jeans, the company will survive the competition. At over 10%, there is a real problem to solve. Losing 20% of all the fabric consumed in manufacturing jeans can bring the company down and force it into bankruptcy.

> The main rule is to combine larger pieces and smaller pieces to make optimized cutting patterns.

Optimization of space through larger and smaller objects can be applied to cutting the forms, which are the patterns of the jeans, for example. Once they are cut, they will be assembled at the sewing stations.

The problem can be summed up as:

- Creating subsets of the 500,000 SKUs to optimize the cutting process for the month to come in a given factory
- Making sure that each subset contains smaller sizes and larger sizes to minimize the loss of fabric by choosing 6 sizes per day to build 25,000 unit subsets per day
- Generating cut plans of an average of 3 to 6 sizes per subset per day for a production of 25,000 units per day

In mathematical terms, this means trying to find subsets of sizes among 500,000

units for a given day.

The task is to find 6 well-matched sizes among 500,000 units, as shown in the following combination formula:

$$C(n,r) = \frac{n!}{r!\,(n-r)!} = \frac{500000!}{6!\,(500000-6)!} = \log_{10} 10^{31.33}$$

At this point, most people abandon the idea and find some easy way out of this, even if it means wasting fabric.

The first reaction we all have is that this is more than the number of stars in the universe and all that hype. However, that's not the right way to look at it at all. The right way is to look exactly in the opposite direction.

The key to this problem is to observe the particle at a microscopic level, at the **bits of information** level. Analyzing detailed data is necessary to obtain reliable results. This is a fundamental concept of machine learning and deep learning. Translated into our field, it means that to process an image, ML and DL process pixels.

So, even if the pictures to process represent large quantities, it will come down to small units of information to analyze:

yottabyte (YB)	10^{24}	yobibyte (YiB)	2^{80}

It might be surprising to see these large numbers appear suddenly! However, when trying to combine thousands of elements, the combinations become exponential. When you extend this to the large population that major apparel brands have to deal with, it becomes rapidly exponential as well.

Today, Google, Facebook, Amazon, and others have yottabytes of data to classify and make sense of. Using the term **big data** doesn't mean much. It's just a lot of data, and so what?

You do not need to analyze the individual positions of each data point in a dataset but use the probability distribution.

To understand that, let's go to a store to buy some jeans for a family. One of the parents wants a pair of jeans, and so does a teenager in that family. They both go and try to find their size in the pair of jeans they want. The parent finds 10 pairs of jeans in size *x*. All of the jeans are part of the production plan. The parent picks one at *random*, and the teenager does the same. Then they pay for them and take them home.

Some systems work fine with random choices: random transportation (taking jeans from the store to home) of particles (jeans, other product units, pixels, or whatever is to be processed), making up that fluid (a dataset).

Translated into our factory, this means that a stochastic (random) process can be introduced to solve the problem.

All that was required is that small and large sizes were picked at random among the 500,000 units to produce. If 6 sizes from 1 to 6 were to be picked per day, the sizes could be classified as follows in a table:

$$\textit{Smaller sizes} = S = \{1, 2, 3\}$$

$$\textit{Larger sizes} = L = \{4, 5, 6\}$$

Converting this into numerical subset names, $S = 1$ and $L = 6$. By selecting large and small sizes to produce at the same time, the fabric will be optimized, as shown in the following table:

Size of choice 1	Size of choice 2	Output
6	6	0
1	1	0
1	6	1
6	1	1

You will notice that the first two lines contain the same value. This will not optimize fabric consumption. If you put only large size 6 products together, there

will be "holes" in the pattern. If you only put small size 1 products together, then they will fill up all of the space and leave no room for larger products. Fabric cutting is optimal when large and small sizes are present on the same roll of fabric.

Doesn't this sound familiar? It looks exactly like our vintage FNN, with 1 instead of 0 and 6 instead of 1.

All that has to be done is to stipulate that subset *S* = *value* 0, and subset *L* = *value* 1; and the previous code can be generalized.

`FFN_XOR_generalization.py` is the program that generalizes the previous code, as shown in the following snippet.

If this works, then smaller and larger sizes will be chosen to send to the cut planning department, and the fabric will be optimized. Applying the randomness concept of Bellman's equation, a stochastic process is applied, choosing customer unit orders at random (each order is one size and a unit quantity of 1):

```
w1=0.5;w2=1;b1=1
w3=w2;w4=w1;b2=b1
s1=random.randint(1,500000)#choice in one set s1
s2=random.randint(1,500000)#choice in one set s2
```

The weights and bias are now constants obtained by the result of the XOR training FNN. The training is over; the FNN is now used to provide results. Bear in mind that the word *learning* in machine learning and deep learning doesn't mean you have to train systems forever. In stable environments, training is run only when the datasets change. At one point in a project, you are hopefully using deep *trained* systems and not simply exploring the training phase of a deep *learning* process. The goal is not to spend all corporate resources on learning but on using trained models.

> Deep learning architecture must rapidly become deep trained models to produce a profit.

For this prototype validation, the size of a given order is random. `0` means the order fits in the *S* subset; `1` means the order fits in the *L* subset. The data

generation function reflects the random nature of consumer behavior in the following six-size jeans consumption model:

```
    x1=random.randint(0, 1)#property of choice:size smaller=
0
    x2=random.randint(0, 1)#property of choice :size bigger=
1
    hidden_layer_y(x1,x2,w1,w2,w3,w4,b1,b2,result)
```

Once two customer orders have been chosen at random in the correct size category, the FNN is activated and runs like the previous example. Only the `result` array has been changed since we are using the same core program. Only a yes (`1`) or no (`0`) is expected, as shown in the following code:

```
#II hidden layer 1 and its output
def hidden_layer_y(x1,x2,w1,w2,w3,w4,b1,b2,result):
    h1=(x1*w1)+(x2*w4) #II.A.weight of hidden neuron h1
    h2=(x2*w3)+(x1*w2) #II.B.weight of hidden neuron h2
#III.threshold I,a hidden layer 2 with bias
    if(h1>=1):h1=1
    if(h1<1):h1=0
    if(h2>=1):h2=1
    if(h2<1):h2=0
    h1= h1 * -b1
    h2= h2 * b2
#IV. threshold II and OUTPUT y
    y=h1+h2
    if(y<1):
        result[0]=0
    if(y>=1):
        result[0]=1
```

The number of subsets to produce needs to be calculated to determine the volume of positive results required.

The choice is made of 6 sizes among 500,000 units. But, the request is to produce a daily production plan for the factory. The daily production target is 25,000. Also, each subset can be used about 20 times. There is always, on average, 20 times the same size in a given pair of jeans available.

Six sizes are required to obtain good fabric optimization. This means that after

three choices, the result represents one subset of potential optimized choices:

$$R = 120 \times 3 \text{ subsets of two sizes} = 360$$

The magic number has been found. For every 3 choices, the goal of producing 6 sizes multiplied by a repetition of 20 will be reached.

The production-per-day request is 25,000:

$$\text{The number of subsets requested} = 25000/3=8333.\ 333$$

The system can run 8,333 products as long as necessary to produce the volume of subsets requested. In this case, the range is set to a sample of 1,000,000 products. It can be extended or reduced when needed. The system is filtering the correct subsets through the following function:

```
for element in range(1000000):
    ...(a block of code is here in the program)...
    if(result[0]>0):
        subsets+=1
        print("Subset:",subsets,"size subset #",x1," and ",
"size subset #",x2,"result:",result[0],"order #"," and ",s1,
"order #",s2)
    if(subsets>=8333):
        break
```

When the 8,333 subsets have been found respecting the smaller-larger size distribution, the system stops, as shown in the following output:

```
Subset: 8330 size subset # 1 and size subset # 0 result: 1
order # and 53154 order # 14310
Subset: 8331 size subset # 1 and size subset # 0 result: 1
order # and 473411 order # 196256
Subset: 8332 size subset # 1 and size subset # 0 result: 1
order # and 133112 order # 34827
Subset: 8333 size subset # 0 and size subset # 1 result: 1
order # and 470291 order # 327392
```

This example proves the point. *Simple solutions can solve very complex problems.*

Two main functions, among some minor ones, must be added:

- After each choice, the orders chosen must be removed from the 500,000-order dataset. When an order has been selected, processing it again will generate errors in the global results. This will preclude choosing the same order twice and reduce the number of choices to be made.
- An optimization function to regroup the results for production purposes, for example. The idea is not to run through the records randomly, but to organize them by sets. This way, each set can be controlled independently.

Application information:

- The core calculation part of the application is fewer than 50 lines long.
- With a few control functions and arrays, the program might reach 200 lines maximum. The goal of the control functions is to check and see whether the results reach the overall goal. For example, every 1,000 records, a local result could be checked to see whether it fits the overall goal.
- This results in easy maintenance for a team.

Optimizing the number of lines of code to create a powerful application can prove to be very efficient for many business problems.

Summary

Building a small neural network from scratch provides a practical view of the elementary properties of a neuron. We saw that a neuron requires an input that can contain many variables. Then, weights are applied to the values with biases. An activation function then transforms the result and produces an output.

Neural networks, even one- or two-layer networks, can provide real-life solutions in a corporate environment. A real-life business case was implemented using complex theory broken down into small functions. Then, these components were assembled to be as minimal and profitable as possible.

It takes talent to break a problem down into elementary parts and find a simple, powerful solution. It requires more effort than just typing hundreds to thousands

of lines of code to make things work. A well-thought through algorithm will always be more profitable, and software maintenance will prove more cost-effective.

Customers expect quick-win solutions. Artificial intelligence provides a large variety of tools that satisfy that goal. When solving a problem for a customer, do not look for the best theory, but the simplest and fastest way to implement a profitable solution, no matter how unconventional it seems.

In this case, an enhanced FNN perceptron solved a complex business problem. In the next chapter, we will explore a convolutional neural network (CNN). We will build a CNN with TensorFlow 2.x, layer by layer, to classify images.

Questions

1. Can the perceptron alone solve the XOR problem? (Yes | No)
2. Is the XOR function linearly non-separable? (Yes | No)
3. One of the main goals of layers in a neural network is classification. (Yes | No)
4. Is deep learning the only way to classify data? (Yes | No)
5. A cost function shows the increase in the cost of a neural network. (Yes | No)
6. Can simple arithmetic be enough to optimize a cost function? (Yes | No)
7. A feedforward network requires inputs, layers, and an output. (Yes | No)
8. A feedforward network always requires training with backpropagation. (Yes | No)
9. In real-life applications, solutions are only found by following existing theories. (Yes | No)

Further reading

- Linear separability: http://www.ece.utep.edu/research/webfuzzy/docs/kk-thesis/kk-thesis-html/node19.html

9

Abstract Image Classification with Convolutional Neural Networks (CNNs)

The invention of **convolutional neural networks** (**CNNs**) applied to vision represents by far one of the most innovative achievements in the history of applied mathematics. With their multiple layers (visible and hidden), CNNs have brought artificial intelligence from machine learning to deep learning.

In *Chapter 8*, *Solving the XOR Problem with a Feedforward Neural Network*, we saw that $f(x, w)$ is the building block of any neural network. A function f will transform an input x with weights w to produce an output. This output can be used as such or fed into another layer. In this chapter, we will generalize this principle and introduce several layers. At the same time, we will use datasets with images. We will have a dataset for training and a dataset for validation to confirm that our model works.

A CNN relies on two basic tools of linear algebra: kernels and functions, applying them to convolutions as described in this chapter. These tools have been used in mathematics for decades.

However, it took the incredible imagination of Yann LeCun, Yoshua Bengio, and others—who built a mathematical model of several layers—to solve real-life problems with CNNs.

This chapter describes the marvels of CNNs, one of the pillars of **artificial neural networks** (**ANNs**). A CNN will be built from scratch, trained, and saved. The classification model described will detect production failures on a food-processing production line. Image detection will go beyond object recognition and produce abstract results in the form of concepts.

A Python TensorFlow 2 program will be built layer by layer and trained. Additional sample programs will illustrate key functions.

The following topics will be covered in this chapter:

- The differences between 1D, 2D, and 3D CNNs
- Adding layers to a convolutional neural network
- Kernels and filters
- Shaping images
- The ReLU activation function
- Kernel initialization
- Pooling
- Flattening
- Dense layers
- Compiling the model
- The cross-entropy loss function
- The Adam optimizer
- Training the model
- Saving the model
- Visualizing the PNG of a model

We'll begin by introducing CNNs and defining what they are.

Introducing CNNs

This section describes the basic components of a CNN. `CNN_SRATEGY_MODEL.py` will illustrate the basic CNN components used to build a model for abstract image detection. For machines, as for humans, concepts are the building blocks of cognition. CNNs constitute one of the pillars of deep learning (multiple layers and neurons).

In this chapter, TensorFlow 2 with Python will be running using Keras libraries

that are now part of TensorFlow. If you do not have Python or do not wish to follow the programming exercises, the chapter is self-contained, with graphs and explanations.

Defining a CNN

A convolutional neural network processes information, such as an image, for example, and makes sense out of it.

For example, imagine you have to represent the sun with an ordinary pencil and a piece of paper. It is a sunny day, and the sun is shining very brightly—too brightly. You put on a special pair of very dense sunglasses. Now you can look at the sun for a few seconds. You have just applied a color reduction filter, one of the first operations of a convolutional network.

Then, you try to draw the sun. You draw a circle and put some gray in the middle. You have just applied an edge filter. Finally, you go over the circle several times to make it easy to recognize, progressively reducing what you saw into a representation of it. Now, with the circle, some gray in the middle, and a few lines of rays around it, anybody can see you drew the sun. You smile; you did it! You took a color image of the sun and made a mathematical representation of it as a circle, which would probably look something like this:

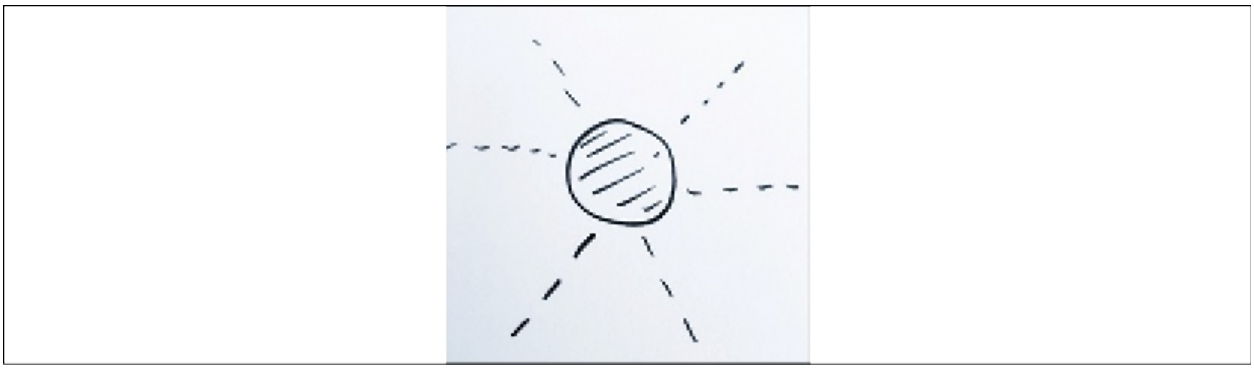


Figure 9.1: Mathematical representation of a circle

You just went through the basic processes of a convolutional network.

The word **convolutional** means that you transformed the sun you were looking at into a drawing, area by area. But, you did not look at the whole sky at once.

You made many eye movements to capture the sun, area by area, and you did the same when drawing. If you made a mathematical representation of the way you transformed each area from your vision to your paper abstraction, it would be a kernel. You can see that the convolutional operation converts an object into a more abstract representation. This is not limited to images but can apply to any type of data (words, sounds and video) we want to draw patterns from.

With that concept in mind, the following graph shows the successive mathematical steps to follow in this chapter's model for a machine to process an image just as you did. A convolutional network is a succession of steps that will transform what you see into a classification status.

In the graph, each box represents a layer. Each layer has an input that comes from the previous layer. Each layer will then transform the input and then produce an output that will become the input of the next layer. At each layer, the key features that are necessary to classify the images will be isolated.

In your example, it would serve to find out whether your drawing represents the sun or not. This falls under a binary classification model (yes or no, or 1 or 0).

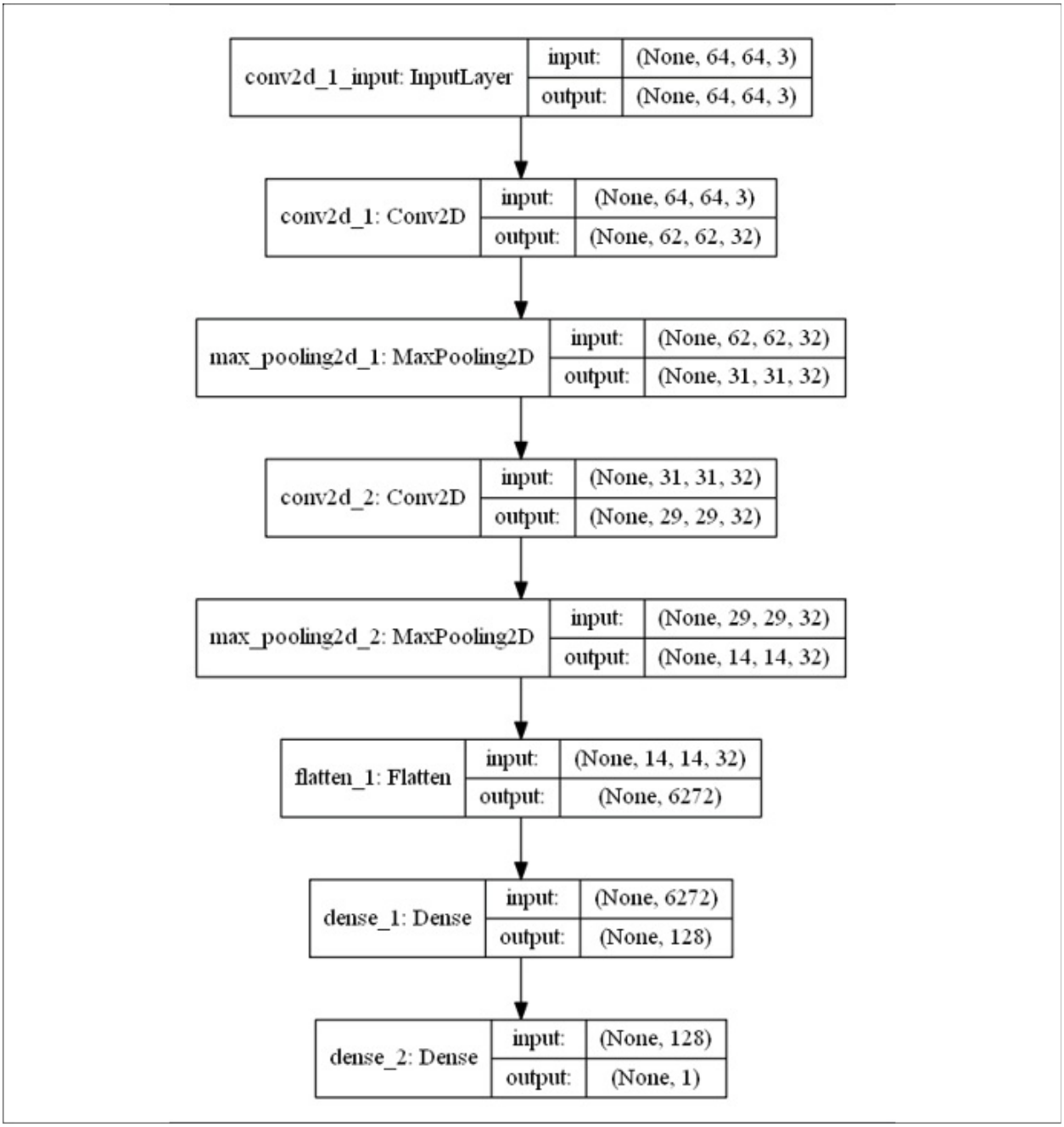


Figure 9.2: Architecture of a CNN

Notice that the size of the outputs diminishes progressively until the outputs reach 1, the binary classification status that will return (1 or 0). These successive steps, or layers, represent what you did when you went from observing the sun to drawing it. In the end, if we draw poorly and nobody recognizes the sun, it means that we'll have to go back to step 1 and change some parameters (weights in this case). That way, we train to represent the sun better until somebody says,

"Yes, it is a sun!" That is probability = 1. Another person may say that it is not a sun (probability = 0). In that case, more training would be required.

If you carry out this experiment of drawing the sun, you will notice that, as a human, you transform one area at a time with your eye and pencil. You repeat the way you do it in each area. The mathematical repetition you perform is your **kernel**. Using a kernel per area is the fastest way to draw. For us humans, in fact, it is the only way we can draw. A CNN is based on this process.

In this section, we looked at some key aspects of a CNN model, using the analogy of representing the sun as a drawing. This is just one way to start a convolutional neural network, and there are hundreds of different ways to do so. However, once you understand one model, you will have the understanding necessary to implement other variations.

In the following section, we'll see how to initialize and build our own CNN.

Initializing the CNN

`CNN_SRATEGY_MODEL.py` builds the CNN using TensorFlow 2. TensorFlow 2 has made tremendous improvements in terms of development. The Keras datasets, layers, and models are now part of the TensorFlow instance:

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
```

The CNN only requires two lines of headers to build the layers! In TensorFlow 2, for each layer, we simply have to call `layers.<add your layer here>` and that's it!

The model used is a Keras `sequential()` called from the TensorFlow `from tensorflow.keras` instance:

```
classifier = models.Sequential()
```

And that's it. We have just started to build our own CNN in just a few lines of code. TensorFlow 2 has simplified the whole process of creating a CNN, making it an easy, intuitive process, as we will see throughout this chapter.

Let's begin to build upon the foundations of our CNN in the following section and add a convolutional layer.

Adding a 2D convolution layer

In this chapter, we will be using a two-dimensional model as our example. Two-dimensional relationships can be real-life images and also many other objects, as described in this chapter. This chapter describes a two-dimensional network, although others exist:

- A one-dimensional CNN mostly describes a temporal mode, for example, a sequence of sounds (phonemes = parts of words), words, numbers, and any other type of sequence.
- A volumetric module is a 3D convolution, such as recognizing a cube or a video. For example, for a self-driving car, it is critical to recognize the difference between a 2D picture of a person in an advertisement near a road and a real 3D image of a pedestrian that is starting to cross the same road!

In this chapter, a spatial 2D convolution module will be applied to images of different kinds. The main program, `CNN_STRATEGY_MODEL.py`, will describe how to build and save a model.

`classifier.add` will add a layer to the model. The name **classifier** does not represent a function but simply the arbitrary name that was given to this model in this particular program. The model will end up with *n* layers. Look at the following line of code:

```
classifier.add(layers.Conv2D(32, (3, 3),input_shape = (64,
64, 3), activation = 'relu'))
```

This line of code contains a lot of information: the filters (applied with kernels), the input shape, and an activation function. The function contains many more options. Once you understand these in-depth, you can implement other options one by one, as you deem necessary, for each project you have to work on.

Kernel

Just to get started, intuitively, let's take another everyday model. This model is a bit more mathematical and closer to a CNN's kernel representation. Imagine a floor of very small square tiles in an office building. You would like each floor tile to be converted from dirty to clean, for example.

You can imagine a cleaning machine capable of converting 3×3 small tiles (pixels) one at a time from dirty to clean. You would laugh if you saw somebody come with one enormous cleaning machine to clean all of the 32×32 tiles (pixels) at the same time. You know it would be very bulky, slow, and difficult to use, intuitively. On top of that, you would need one big machine per surface size! Not only is a kernel an efficient way to filter, but a kernel convolution is also a time-saving resource process. The small cleaning machine is the kernel (dirty-to-clean filter), which will save you time performing the convolution (going over all of the tiles to clean a 3×3 area), transforming the floor from dirty to clean.

In this case, 32 different filters have been added with 3×3 sized kernels:

```
classifier.add(layers.Conv2D(32, (3, 3)...
```

The use of kernels as filters is the core of a convolutional network. `(32, (3,3))` means `(number of filters, (size of kernels))`.

An intuitive approach

To understand a kernel intuitively, keep the sun and cleaning tiles examples in mind. In this section, a photograph of a cat will show how kernels work.

In a model analyzing cats, the initial photograph would look like this:

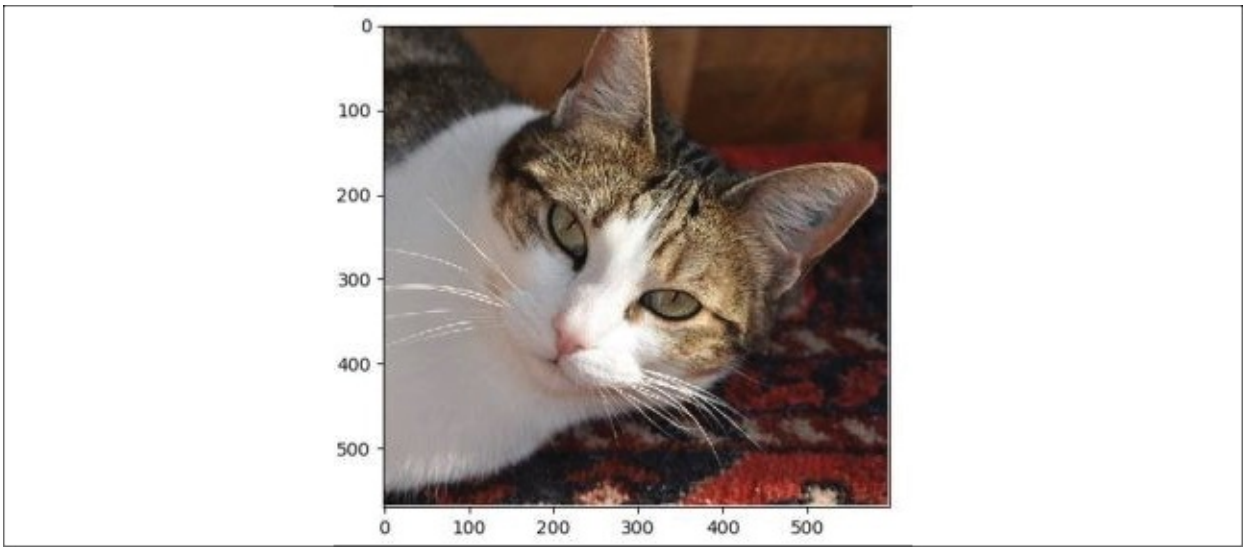


Figure 9.3: Cat photograph for model analysis

On the first run of this layer, even with no training, an untrained kernel would transform the photograph:

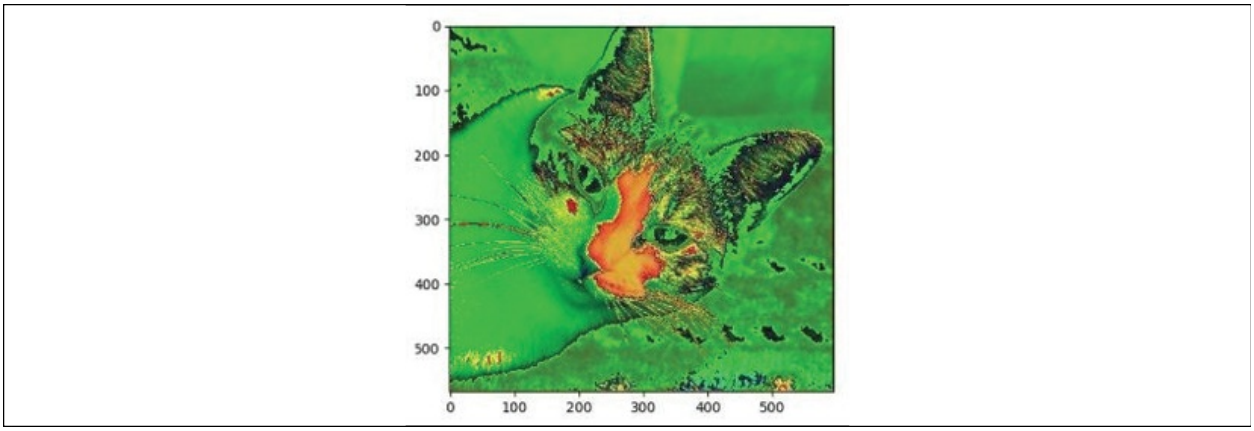


Figure 9.4: Cat photograph transformation

The first layer has already begun isolating the features of the cat. The edges have begun to appear: the cat's body, ears, nose, and eyes. In itself, this first filter (one of 32) with a size 3×3 kernel—in the first layer and with no training—already produces effective results. The size of a kernel can vary according to your needs. A 3×3 kernel will require a larger number of weights than a 1×1 kernel, for example. A 1×1 kernel will have only one weight, which restricts the size of the features to represent. The rule is that the smaller the kernel, the fewer weights we have to find. It will also perform a feature reduction. When the size of the kernel increases, the number of weights and features to find increases as well as

the number of features represented.

Each subsequent layer will make the features stand out much better, with smaller and smaller matrices and vectors, until the program obtains a clear mathematical representation.

Now that we have an intuitive view of how a filter works, let's explore a developer's approach.

The developers' approach

Developers like to see the result first to decide how to approach a problem.

Let's take a quick, tangible shortcut to understand kernels through `Edge_detection_Kernel.py` with an edge detection kernel:

```
#I.An edge detection kernel
kernel_edge_detection = np.array([[0.,1.,0.],
[1.,-4.,1.],
[0.,1.,0.]])
```

The kernel is a 3×3 matrix, like the cat example. But the values are preset, and not trained with weights. There is no learning here; only a matrix needs to be applied. The major difference with a CNN is that it will learn how to optimize kernels itself through weights and biases.

`img.bmp` is loaded, and the 3×3 matrix is applied to the pixels of the loaded image, area by area:

```
#II.Load image and convolution
image=mpimg.imread('img.bmp')[:,:,0]
shape = image.shape
```

The image before the convolution applying the kernel is the letter **A** (letter recognition):



Figure 9.5: The letter "A"

Now the convolution transforms the image, as shown in the following code:

```
#III.Convolution
image_after_kernel = filter.convolve(image,kernel_edge_dete
ction,mode='constant', cval=0)
```

The edges of A now appear clearly in white, as shown in the following graph:

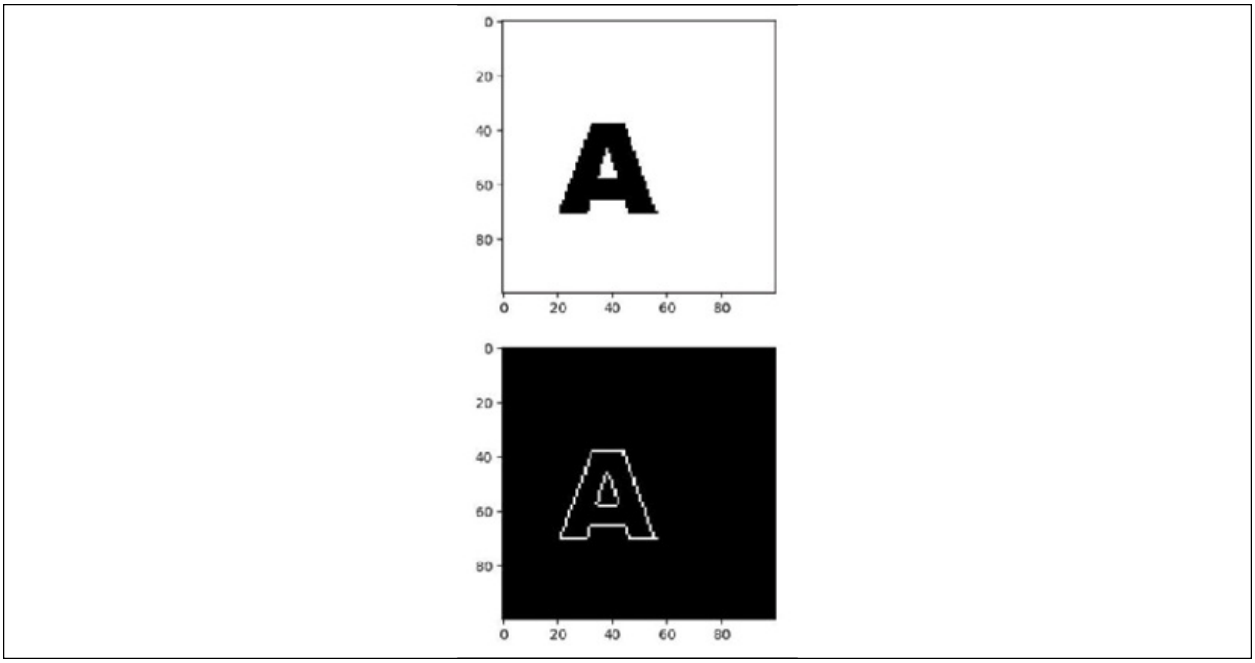


Figure 9.6: The white edges of A are visible

The original image on top displayed a very thick A. The preceding graph displays a thin, identifiable A feature through thin edges that a neural network can classify within a few mathematical operations. The first layers of a convolutional network train to find the right weights to generate the right kernel automatically.

Now that we have an intuitive and practical developer's view of a filter, let's add some mathematics to our approach.

A mathematical approach

The initial image has a set of values you can display, as follows:

```
#II.Load image
image=mpimg.imread('img.bmp')[:,:,0]
shape = image.shape
print("image shape",shape)
```

The code will print a numerical output of the image, as follows:

```
image shape (100, 100)
image before convolution
[[255 255 255 ..., 255 255 255]
 [255 255 255 ..., 255 255 255]
 [255 255 255 ..., 255 255 255]
 ...,
```

The convolution filter is applied using `filter.convolve`, a mathematical function, to transform the image and filter it.

The convolution filter function uses several variables:

- The spatial index for the 3×3 kernel to apply; in this case, it must know how to access the data. This is performed through a spatial index, *j*, which manages data in grids. Databases also use spatial indexes to access data. The axes of those grids determine the density of a spatial index. Kernels and the image are convolved using *j* over *W*, the weights kernel.
- *W* is the weights kernel.
- *I* is the input image.
- *k* is the coordinate of the center of *W*. The default value is 0 in this case.

These variables then enter the `filter.convolve` function as represented by the following equation:

$$C_i = \sum_j I_{i=j-k} W_j$$

A CNN relies on kernels. Take all the time you need to explore convolutions through the three dimensions required to master AI: an intuitive approach, development testing, and mathematical representation.

Now that we have a mathematical idea on how a convolutional filter works, let's

determine the shape and the activation function to the convolutional layer.

Shape

`input_shape` defines the size of the image, which is 64×64 pixels (height×width), as shown here:

```
classifier.add(...input_shape = (64, 64, 3)...)
```

`3` indicates the number of channels. In this case, `3` indicates the three parameters of an RGB color. Each channel can have a given value of 0 to 255.

ReLU

Activation functions provide useful ways to influence the transformation of weighted data calculations. Their output will change the course of classification, a prediction, or whatever goal the network was built for. This model applies a **rectified linear unit** (**ReLU**), as shown in the following code:

```
classifier.add(..., activation = 'relu'))
```

ReLU activation functions apply variations of the following function to an input value:

$$f(x) = \max\{0, x\}$$

The function returns 0 for negative values; it returns positive values as *x*; it returns 0 for 0 values. Half of the domain of the function will return zeros. This means that when you provide positive values, the derivative will always be 1. ReLU avoids the squashing effect of the logistic sigmoid function, for example. However, the decision to use one activation function rather than another will depend on the goal of each ANN model.

In mathematical terms, a **rectified linear unit (ReLU)** function will take all the negative values and apply 0 to them. And all the positive values remain unchanged.

The `ReLU.py` program provides some functions, including a NumPy function, to

test how ReLU works.

You can enter test values or use the ones in the source code:

```
import numpy as np
nx=-3
px=5
```

`nx` expects a negative value, and `px` expects a positive value for testing purposes for the `relu(x)` and `lrelu(x)` functions. Use the `f(x)` function if you wish to include zeros in your testing session.

The `relu(x)` function will calculate the ReLU value:

```
def relu(x):
    if(x<=0):ReLU=0
    if(x>0):ReLU=x
    return ReLU
```

In this case, the program will return the following result:

```
negative x= -3 positive x= 5
ReLU nx= 0
ReLU px= 5
```

The result of a negative value becomes 0, and a positive value remains unchanged. The derivative or slope is thus always 1, which is practical in many cases and provides good visibility when debugging a CNN or any other ANN.

The NumPy function, defined as follows, will provide the same results:

```
def f(x):
    vfx=np.maximum(0.1,x)
    return vfx
```

Through trial and error, ANN research has come up with several variations of ReLU.

One important example occurs when many input values are negative. ReLU will constantly produce zeros, making gradient descent difficult, if not impossible.

A clever solution was found using a leaky ReLU. A leaky ReLU does not return 0 for a negative value but a small value you can choose, 0.1 instead of 0, for example. See the following equation:

$$f(x) = \max\{0.1, x\}$$

The leaky ReLU fixes the problem of "dying" neurons. Suppose you have a layer that keeps returning negative values when activating neurons. The ReLU activation will always return 0 in this case. That means that these neurons are "dead." They will never be activated. To avoid these "dying" neurons, a leaky ReLU provides the small positive value seen previously (0.1) that makes sure that a neuron does not "die."

Now gradient descent will work fine. In the sample code, the function is implemented as follows:

```
def lrelu(x):
    if(x<0):lReLU=0.01
    if(x>0):lReLU=x
    return lReLU
```

Although many other variations of ReLU exist, with this in mind, you have an idea of what it does.

Enter some values of your own, and the program will display the results, as shown here:

```
print("negative x=",nx,"positive x=",px)
print("ReLU nx=",relu(nx))
print("ReLU px=",relu(px))
print("Leaky ReLU nx=",lrelu(nx))
print("f(nx) ReLu=",f(nx))
print("f(px) ReLu=",f(px))
print("f(0):",f(0))
```

The results will display the ReLU results as follows:

```
negative x= -3 positive x= 5
ReLU nx= 0
ReLU px= 5
Leaky ReLU nx= 0.01
```

We have processed a large representation of the input image. We now need to reduce the size of our representation to obtain a better, more abstract representation. By pooling some of the pixels we will also reduce the calculations of the subsequent layers.

Pooling

A CNN contains hidden layers. The input is visible. Then as the layers work to transform the data, "hidden" work goes on. The output layer is visible again. Let's continue to explore the "hidden" layers! Pooling reduces the size of an input representation, in this case, an image. Max pooling consists of applying a max pooling window to a layer of the image:

```
classifier.add(layers.MaxPooling2D(pool_size = (2, 2)))
```

This `pool_size` 2×2 window will first find the maximum value of the 2×2 matrix at the top left of the image matrix. This first maximum value is 4. It is thus the first value of the pooling window on the right.

Then, the max pooling window hops over 2 squares and finds that 5 is the highest value. 5 is written in the max pooling window. The hop action is called a **stride**. A stride value of 2 will avoid overlapping, although some CNN models have strides that overlap. It all depends on your goal. Look at the following diagram:

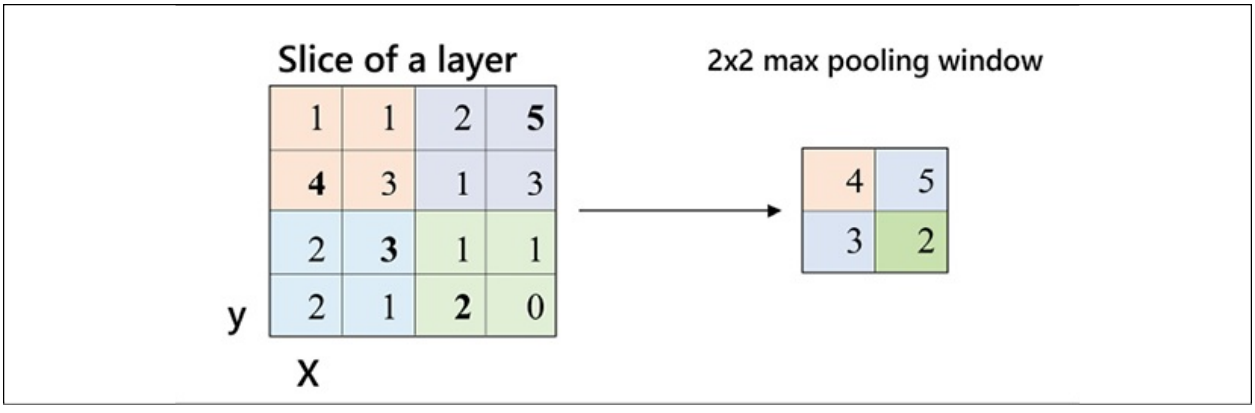


Figure 9.7: Pooling example

The output size has now gone from a 62×62×32 (number of filters) to a 31×31×32, as shown in the following diagram:

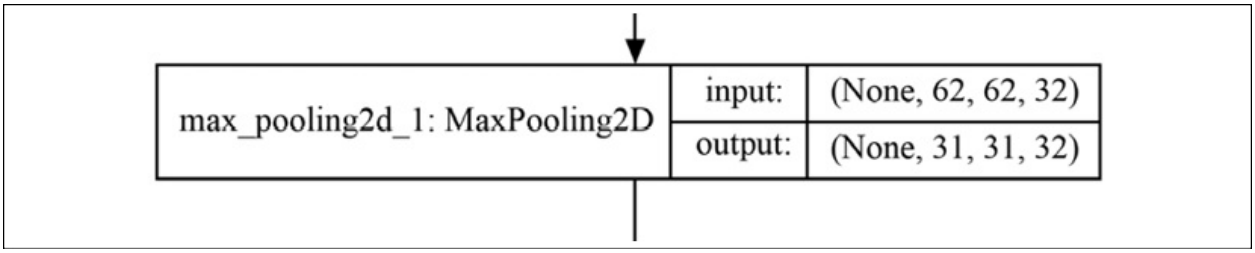


Figure 9.8: Output size changes (pooling)

Other pooling methods exist, such as average pooling, which uses the average of the pooling window and not the maximum value. This depends on the model and shows the hard work that needs to be put in to train a model.

Next convolution and pooling layer

The next two layers of the CNN repeat the same method as the first two described previously, and it is implemented as follows in the source code:

```
# Step 3 Adding a second convolutional layer and pooling la
yer
print("Step 3a Convolution")
classifier.add(layers.Conv2D(32, (3, 3), activation = 'relu'
))
print("Step 3b Pooling")
classifier.add(layers.MaxPooling2D(pool_size = (2, 2)))
```

These two layers have drastically downsized the input to 14×14×32, as shown in this diagram:

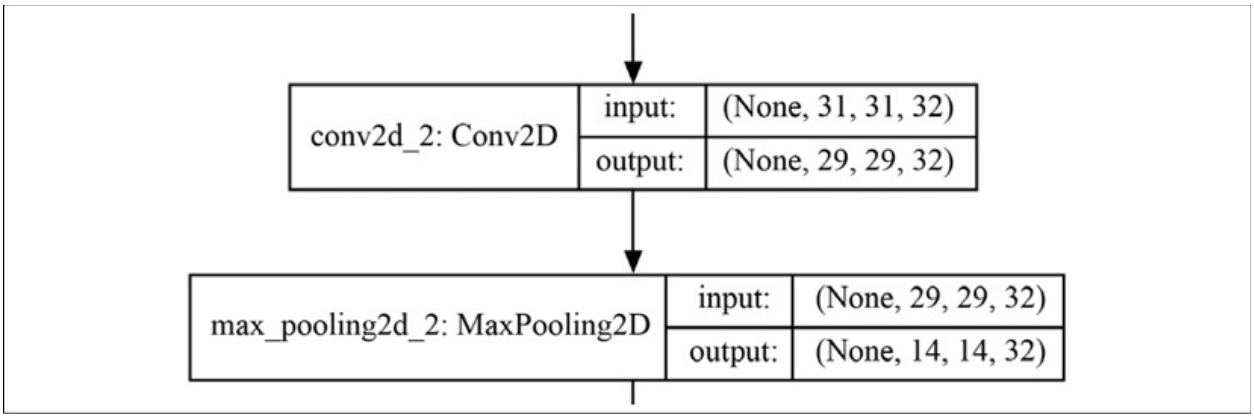


Figure 9.9: Convolution and pooling layer

It is possible to insert a padding layer on a CNN. As we shrink our image layer by layer, the filters in a convolutional network will impact the center pixels more than the outer pixels. Suppose you start drawing on a piece of paper. You tend to fill the center of the paper and avoid the edges. The edges of the piece of paper contain less information. If you decide to apply padding to the edges, the image will be more complete. In a neural network, padding has the same function. It makes sure the edges are taken into account by adding values. Padding can be implemented before or after pooling, for example. We will implement an example of padding in *Chapter 13, Visualizing Networks with TensorFlow 2.x and TensorBoard*.

The next layer can apply flattening to the output of the pooling of this section, as we'll see in the next section.

Flattening

The flattening layer takes the output of the max pooling layer and transforms the vector of size *x * y * z* into a flattened vector, as shown in the following code:

```
# Step 4 - Flattening
print("Step 4 Flattening")
classifier.add(layers.Flatten())
```

In this case, the layer vector will be 14 × 14 × 32 = 6,272, as shown in the following diagram:

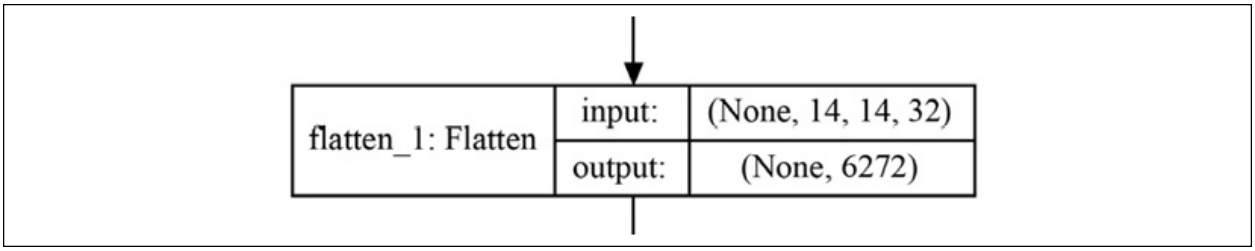


Figure 9.10: Flattening layer

This operation creates a standard layer with 6,272 very practical connections for the dense operations that follow. After flattening has been carried out, a fully connected dense network can be implemented.

Dense layers

Dense layers are fully connected. Full connections are possible through the size reductions calculated so far, as shown before.

The successive layers in this sequential model have brought the size of the image down enough to use dense layers to finish the job. `dense_1` comes first, as shown here:

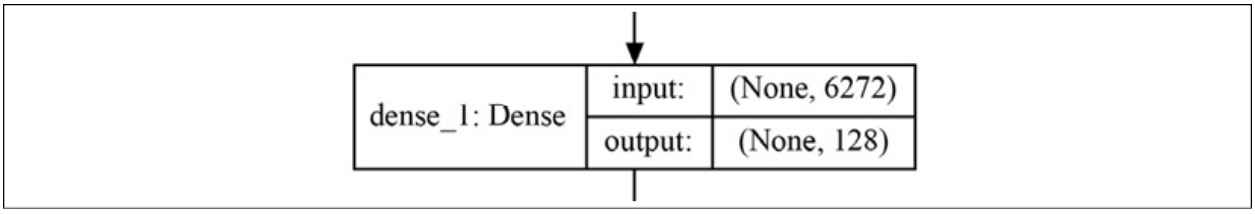


Figure 9.11: Dense layer

The flattening layer produced a 14×14×32 size 6,272 layer with a weight for each input. If it had not gone through the previous layers, the flattening would have produced a much larger layer, slowing feature extractions down. The result would produce nothing effective.

With the main features extracted by the filters through successive layers and size reduction, the dense operations will lead directly to a prediction using ReLU on the dense operation and then the logistic sigmoid function to produce the final result:

```
print("Step 5 Dense")
classifier.add(layers.Dense(units = 128, activation = 'relu
'))
classifier.add(layers.Dense(units = 1, activation = 'sigmoid
'))
```

Now that we have the dense layer, let's explore the dense layer's activation functions.

Dense activation functions

The ReLU activation function can be applied to a dense layer as in other layers.

The domain of the ReLU activation function is applied to the result of the first dense operation. The ReLU activation function will output the initial input for values >=0 and will output 0 for values <0:

$$f(\text{input\_value}) = \max\{0, \text{input\_value}\}$$

The logistic activation function is applied to the second dense operation, as described in *Chapter 2*, *Building a Reward Matrix – Designing Your Datasets*.

It will produce a value between 0 and 1:

$LS(x)=\{0,1\}$

We have now built the last dense layer after the *LS* activation function.

The last dense layer is of size 1 and will classify the initial input—an image in this case:

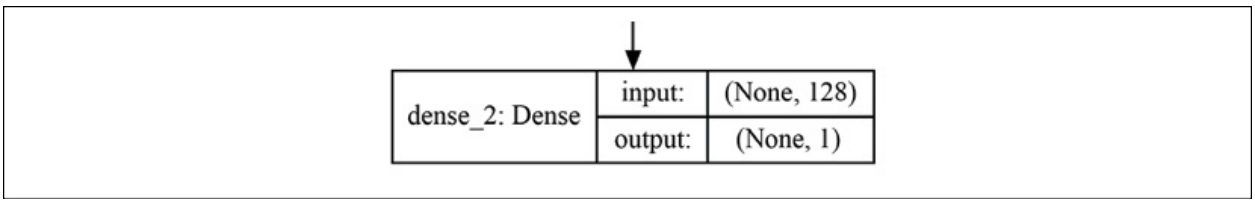


Figure 9.12: Dense layer 2

The layers of the model have now been added. Training can begin.

Training a CNN model

Training a CNN model involves four phases: compiling the model, loading the training data, loading the test data, and running the model through epochs of loss evaluation and parameter-updating cycles.

In this section, the choice of theme for the training dataset will be an example from the food-processing industry. The idea here is not only to recognize an object but to form a concept. We will explore concept learning neural networks further in *Chapter 10, Conceptual Representation Learning*. For the moment, let's train our model.

The goal

The primary goal of this model consists of detecting production efficiency flaws on a food-processing conveyor belt. The use of CIFAR-10 (images) and MNIST (a handwritten digit database) proves useful to understand and train some models. However, in this example, the goal is not to recognize objects but a concept.

The following image shows a section of the conveyor belt that contains an acceptable level of products, in this case, portions of chocolate cake:

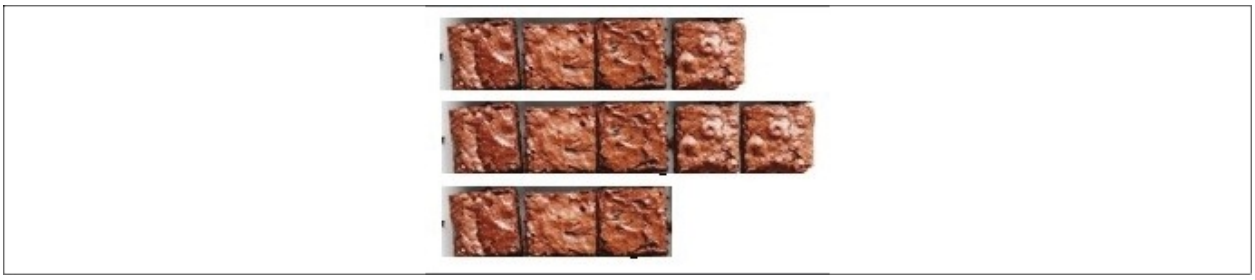


Figure 9.13: Portions of chocolate cake example

The images can represent anything from chocolate cakes, cars on a road or any other type of object. The main point is to detect when there are "gaps" or "holes" in the rows of objects. To train the CNN, I used images containing objects I sometimes drew just to train the system to "see" gaps.

However, sometimes production slows down, and the output goes down to an alert level, as shown in the following photograph:

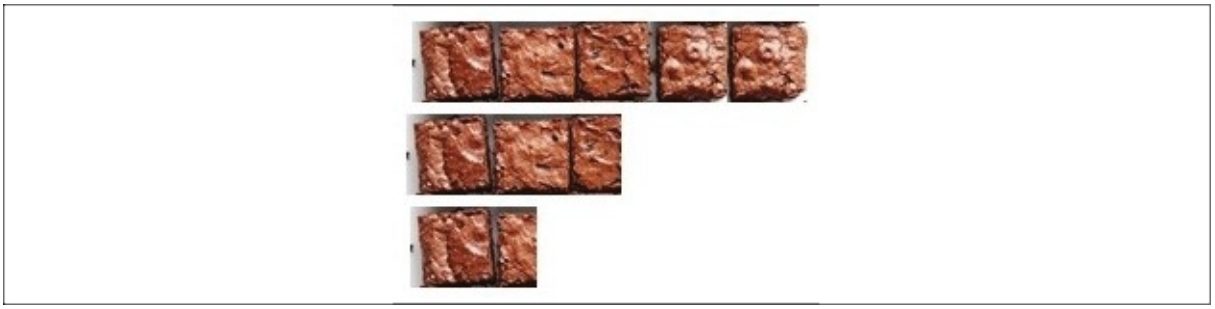


Figure 9.14: Portions of chocolate cake example

The alert-level image shows a gap that will slow down the packaging section of the factory dramatically. There are three lines of objects in the preceding image. On line one, you can see five little objects (here pieces of cake), on line two, only three. On line three, you can only see two objects. There are thus objects missing on lines two and three. This constitutes a gap in this case that is a real problem on production lines. The level of the number of acceptable objects in a frame is a parameter.

Now that we have our goal, let's begin compiling the model.

Compiling the model

Compiling a TensorFlow 2 model requires a minimum of two options: a loss function and an optimizer. You evaluate how much you are losing and then optimize your parameters, just as in real life. A metric option has been added to measure the performance of the model. With a metric, you can analyze your losses and optimize your situation, as shown in the following code:

```
classifier.compile(optimizer = 'adam', loss = 'binary_cross
entropy', metrics = ['accuracy'])
```

Let's take a look at some specific aspects of model compiling, starting with the loss function.

The loss function

The loss function provides information on how far the state of the model y_1 (weights and biases) is from its target state y.

A description of the quadratic loss function precedes that of the binary cross-entropy functions applied to the case study model in this chapter.

The quadratic loss function

Let's refresh the concept of gradient descent. Imagine you are on a hill and want to walk down that hill. Your goal is to get to y, the bottom of the hill. Presently, you are at location a. Google Maps shows you that you still have to go a certain distance:

$$y - a$$

That formula is great for the moment. But now suppose you are almost at the bottom of the hill, and the person walking in front of you has dropped a coin. You have to slow down now, and Google Maps is not helping much because at this zoom level.

You must then zoom into smaller distances with a quadratic objective (or cost) function:

$$O = (y - a)^2$$

To make it more comfortable to analyze, O is divided by 2, producing a standard quadratic cost function:

$$\text{Cost} = \frac{(y - a)^2}{2}$$

y is the goal. a is the result of the operation of applying the weights, biases, and finally, the activation functions.

With the derivatives of the results, the weights and biases can be updated. In our hill example, if you move one meter (y) per step (x), that is much more than moving 0.5 meters (y) per step. Depending on your position on the hill, you can see that you cannot apply a constant learning rate (conceptually, the length of your step); you adapt it just like Adam, the optimizer, does.

Binary cross-entropy

Cross-entropy comes in handy when the learning slows down. In the hill example, it slowed down at the bottom. But, remember, a path can lead you sideways, meaning you are momentarily stuck at a given height. Cross-entropy solves that by being able to function well with very small values (steps on the hill).

Suppose you have the following structure:

- Inputs = $\{x_1, x_2, \ldots, x_n\}$
- Weights = $\{w_1, w_2, \ldots, w_n\}$
- A bias (or sometimes more) is b
- An activation function (ReLU, logistic sigmoid, or other)

Before the activation, z represents the sum of the classical operations:

$$z = \sum_{x_i} w_i x_i + b$$

Now the activation function is applied to z to obtain the present output of the model.

$$y_1 = act(z)$$

With this in mind, the cross-entropy loss formula can be explained:

$$\text{Loss} = \frac{1}{n} \sum_{x} [y \log y_1 + (1 - y) \log(1 - y_1)]$$

In this function:

- n is the total number of items of the input training, with multiclass data. The choice of the logarithm base (2, e, 10) will produce different effects.
- y is the output goal.
- y_1 is the present value, as described previously.

This loss function is always positive; the values have a minus sign in front of them, and the function starts with a minus. The output produces small numbers that tend to zero as the system progresses.

The loss function uses this basic concept with more mathematical inputs to update the parameters.

A binary cross-entropy loss function is a binomial function that will produce a probability output of 0 or 1 and not a value between 0 and 1 as in standard cross-entropy. In the binomial classification model, the output will be 0 or 1.

In this case, the sum $\sum$ is not necessary when M (number of classes) = 2. The binary cross-entropy loss function is then as follows:

$$\text{Loss} = -y \log y_1 + (1 - y) \log (1 - y_1)$$

The whole concept of this loss function method is for the CNN network to provide information for the optimizer to adapt the weights accordingly and automatically.

The Adam optimizer

In the hill example, you first walked with big strides down the hill using momentum (larger strides because you are going in the right direction). Then, you had to take smaller steps to find the object. You adapted your estimation of your moment to your need; hence, the name **adaptive moment estimation** (**Adam**).

Adam constantly compares mean past gradients to present gradients. In the hill example, it compares how fast you were going.

The Adam optimizer represents an alternative to the classical gradient descent method. Adam goes further by applying its optimizer to random (stochastic) mini-batches of the dataset. This approach is an efficient version of stochastic gradient descent.

Then, with even more inventiveness, Adam adds **root-mean-square deviation** (**RMSprop**) to the process by applying per-parameter learning weights. It analyzes how fast the means of the weights are changing (such as the gradients

in our hill slope example) and adapts the learning weights.

Metrics

Metrics are there to measure the performance of your model during the training process. The metric function behaves like a loss function. However, it is not used to train the model.

In this case, the `accuracy` parameter was this:

```
...metrics = ['accuracy'])
```

Here, a value that descends toward 0 shows whether the training is on the right track and moves up to 1 when the training requires Adam function optimizing to set the training on track again.

With this, we have compiled our model. We can now consider our training dataset.

The training dataset

The training dataset is available on GitHub. The dataset contains the image shown previously for the food-processing conveyor belt example. I created a training dataset with a repetition of a few images that I used to illustrate the architecture of a CNN simply. In a real-life project, it will take careful designing with a trial-and-error approach to create a proper dataset that represents all of the cases that the CNN will face.

The class `A` directory contains the acceptable level images of a production line that is producing acceptable levels of products. The class `B` directory contains the alert-level images of a production line that is producing unacceptable levels of products.

The number of images in the dataset is limited because of the following:

- For experimental training purposes, the images produced good results
- The training-testing phase runs faster to study the program

The goal of the model is to detect the alert levels, an abstract conceptual application of a CNN.

Data augmentation

Data augmentation increases the size of the dataset by generating distorted versions of the images provided.

The `ImageDataGenerator` function generates batches of all images found in tensor formats. It will perform data augmentation by distorting the images (shear range, for example). Data augmentation is a fast way to use the images you have and create more virtual images through distortions:

```
train_datagen =
tf.compat.v2.keras.preprocessing.image.ImageDataGenerator(
rescale = 1./255,shear_range = 0.2,zoom_range = 0.2,
horizontal_flip = True)
```

The code description is as follows:

- `rescale` will rescale the input image if not `0` (or `None`). In this case, the data is multiplied by 1/255 before applying any other operation.
- `shear_range` will displace each value in the same direction, determined in this case by the `0.2`. It will slightly distort the image at one point, giving some more virtual images to train.
- `zoom_range` is the value of zoom.
- `horizontal_flip` is set to `True`. This is a Boolean that randomly flips inputs horizontally.

`ImageDataGenerator` provides many more options for real-time data augmentation, such as rotation range, height shift, and others.

Loading the data

Loading the data goes through the `train_datagen` preprocessing image function (described previously) and is implemented in the following code:

```
print("Step 7b training set")
```

```
training_set = train_datagen.flow_from_directory(directory+
'training_set',
target_size = (64, 64),
batch_size = batchs,
class_mode = 'binary')
```

The flow in this program uses the following options:

- `flow_from_directory` sets the directory + `'training_set'` to the path where the two binary classes to train are stored.
- `target_size` will be resized to that dimension. In this case, it is 64×64.
- `batch_size` is the size of the batches of data. The default value is 32, and it's set to `10` in this case.
- `class_mode` determines the label arrays returned: `None` or `'categorical'` will be 2D one-hot encoded labels. In this case, `'binary'` returns 1D binary labels.

Having looked at the training dataset, let's move on to the testing dataset.

The testing dataset

The testing dataset flow follows the same structure as the training dataset flow described previously. However, for testing purposes, the task can be made easier or more difficult, depending on the choice of the model. To make the task more difficult, add images with defects or noise. This will force the system to train more and the project team to do more hard work to fine-tune the model. I chose to use a small dataset to illustrate the architecture of a CNN. In a real-life project choosing the right data that contains all of the cases that the CNN will face takes time and a trial-and-error approach.

Data augmentation provides an efficient way of producing distorted images without adding images to the dataset. Both methods, among many others, can be applied at the same time when necessary.

Data augmentation on the testing dataset

In this model, the data only goes through rescaling. Many other options could be

added to complicate the training task to avoid overfitting, for example, or simply because the dataset is small:

```
print("Step 8a test")
test_datagen = tf.compat.v2.keras.preprocessing.image.Image
DataGenerator(rescale = 1./255)
```

Building datasets is one of the most difficult tasks in an artificial intelligence project. Data augmentation can be a solution if the results are efficient. If not, other techniques must be used. One technique is to gather very large datasets when that is possible and then use data augmentation just to distort the data a bit for training purposes.

Loading the data

Loading the testing data remains limited to what is necessary for this model. Other options can fine-tune the task at hand:

```
print("Step 8b testing set")
test_set = test_datagen.flow_from_directory(directory+'test
_set',
target_size = (64, 64),
batch_size = batchs,
class_mode = 'binary')
```

Never underestimate dataset fine-tuning. Sometimes, this phase can last weeks before finding the right dataset and arguments.

Once the data is loaded, the CNN classifier is ready to be trained. Let's now see how this is done.

Training with the classifier

The classifier has been built and can be run:

```
print("Step 9 training")
print("Classifier",classifier.fit_generator(training_set,
steps_per_epoch = estep,
epochs = ep,
```

```
validation_data = test_set,
validation_steps = vs,verbose=2))
```

You will notice that, in this chapter, we have a directory for the training data and a separate directory for the test data. In *Chapter 5, How to Use Decision Trees to Enhance K-Means Clustering*, we split the datasets into training subsets and testing subsets. This can be applied to a CNN's dataset as well. This is a decision you will need to make, depending on the situation.

For example, sometimes, the test set will be more difficult than the training set, which justifies a separate directory. In other cases, splitting the data can be the most efficient method.

The `fit_generator` function, which fits the model generated batch by batch, contains the main hyperparameters to run the training session through the following arguments in this model. The hyperparameter settings determine the behavior of the training algorithm:

- `training_set` is the training set flow described previously.
- `steps_per_epoch` is the total number of steps (batches of samples) to yield from the generator. The variable used in the following code is `estep`.
- `epochs` is the variable of the total number of iterations made on the data input. The variable used is `ep` in the preceding code.
- `validation_data=test_set` is the testing data flow.
- `validation_steps=vs` is used with the generator and defines the number of batches of samples to test as defined by `vs` in the following code at the beginning of the program:

```
estep=100 #10000
vs=1000 #8000->100
ep=3 #25->2
```

While the training runs, measurements are displayed: loss, accuracy, epochs, information on the structure of the layers, and the steps calculated by the algorithm.

Here is an example of the loss and accuracy data displayed:

```
Epoch 1/2
```

```
 - 23s - loss: 0.1437 - acc: 0.9400 - val_loss: 0.4083 - va
l_acc: 0.5000
Epoch 2/2
 - 21s - loss: 1.9443e-06 - acc: 1.0000 - val_loss: 0.3464
- val_acc: 0.5500
```

Now that we have built and trained the model, we need to save it. Saving the model will avoid having to train the model each time we wish to use it.

Saving the model

By saving the model, we will not have to train it again every time to use it. We will only go back to training when it's required to fine-tune it.

TensorFlow 2 provides a method to save the structure of the model and the weights in a single line of code and a single serialized file:

`model3.h5` saved in the following code, contains serialized data with the model structure and weights. It contains the parameters and options of each layer. This information is very useful to fine-tune the model:

```
print("Step 10: Saving the model")
classifier.save(directory+"model/model3.h5")
```

The model has been built, trained, and saved.

Next steps

The model has been built and trained. In *Chapter 10, Conceptual Representation Learning*, we will explore how to load and run it with no training.

Summary

Building and training a CNN will only succeed with hard work, choosing the model, the right datasets, and hyperparameters. The model must contain

convolutions, pooling, flattening, dense layers, activation functions, and optimizing parameters (weights and biases) to form solid building blocks to train and use a model.

Training a CNN to solve a real-life problem can help sell AI to a manager or a sales prospect. In this case, using the model to help a food-processing factory solve a conveyor belt productivity problem takes AI a step further into everyday corporate life.

A CNN that recognizes abstract concepts within an image takes deep learning one step closer to powerful machine thinking. A machine that can detect objects in an image and extract concepts from the results represents the true final level of AI.

Once the training is over, saving the model provides a practical way to use it by loading it and applying it to new images to classify them. This chapter concluded after we had trained and saved the model.

Chapter 10, Conceptual Representation Learning, will dive deeper into how to design symbolic neural networks.

Questions

1. A CNN can only process images. (Yes | No)
2. A kernel is a preset matrix used for convolutions. (Yes | No)
3. Does pooling have a pooling matrix, or is it random?
4. The size of the dataset always has to be large. (Yes | No)
5. Finding a dataset is not a problem with all the available image banks on the web. (Yes | No)
6. Once a CNN is built, training it does not take much time. (Yes | No)
7. A trained CNN model applies to only one type of image. (Yes | No)
8. A quadratic loss function is not very efficient compared to a cross-entropy function. (Yes | No)
9. The performance of a deep learning CNN does not present a real issue with

modern CPUs and GPUs. (Yes | No)

Further reading and references

- TensorFlow 2: https://www.tensorflow.org/beta

10

Conceptual Representation Learning

Understanding cutting-edge machine learning and deep learning theory only marks the beginning of your adventure. The knowledge you have acquired should help you become an AI visionary. Take everything you see as opportunities and see how AI can fit into your projects. Reach the limits and skydive beyond them.

This chapter focuses on decision-making through visual representations and explains the motivation that led to **conceptual representation learning** (**CRL**) and **metamodels** (**MM**), which form **CRLMMs**.

Concept learning is our human ability to partition the world from chaos to categories, classes, sets, and subsets. As a child and young adult, we acquire many classes of things and concepts. For example, once we understand what a "hole" is, we can apply it to anything we see that is somewhat empty: a black hole, a hole in the wall, a hole in a bank account if money is missing or overspent, and hundreds of other cases.

By performing concept learning, we humans do not have to learn the same concept over and over again for each case. For example, a hole is a hole. So when we see a new situation such as a crater, we know it's just a "big" hole. I first registered a word2vector patent early in my career. Then I rapidly applied it to concept2vector algorithms. I then designed and developed the CRLMM method successfully for **automatic planning and scheduling** (**APS**) software, cognitive chatbots, and more, as we will see in the following chapters. The metamodel term means that I applied one single model to many different domains, just as we humans do.

Conceptual representations also provide visual images of concepts. To plan, humans need to visualize necessary information (events, locations, and so on)

and more critical *visual dimensions* such as *image concepts*. A human being thinks in *mental images*. When we think, mental images flow through our minds with numbers, sounds, odors, and sensations, transforming our environment into fantastic multidimensional representations similar to video clips.

The following topics will be covered in this chapter:

- An approach to CRLMM in three steps:
  - Transfer learning to avoid developing a new program for each variation of a similar case
  - Domain learning to avoid developing a new program each time the domain changes
  - The motivation for using CRLMM

  Over the years, I've successfully implemented CRL in C++, Java, and logic programming (Prolog) in various forms on corporate sites. In this chapter, I'll use Python to illustrate the approach with TensorFlow 2.x with the **convolutional neural network (CNN)** built in *Chapter 9, Abstract Image Classification with Convolutional Neural Networks (CNNs)*.

- Transfer learning using a CNN model trained to generalize image recognition
- Domain learning to extend image recognition trained in one field to another field

We'll begin this chapter by looking at the benefits of transfer learning and how concept learning can boost this process.

Generating profit with transfer learning

Transfer learning means that we can use a model we designed and trained in another similar case. This will make the model very profitable since we do not have to design a new model and write a new program for every new case. You

will thus generate profit for your company or customer by lowering the cost of new implementations of your trained model. Think of a good AI model as a reusable tool when applied to similar cases. This is why concept learning, being more general and abstract, is profitable. That is how we humans adapt.

When it comes to reasoning and thinking in general, we use mental images with some words. Our thoughts contain concepts, on which we build solutions.

The trained model from *Chapter 9, Abstract Image Classification with Convolutional Neural Networks (CNNs)*, can now classify images of a certain type. In this section, the trained model will be loaded and then generalized through transfer learning to classify similar images.

> You will notice that I did not use many images for the example. My goal was to explain the process, not to go into building large datasets, which is a task in itself. The primary goal is to *understand* CNNs and conceptual learning representations.

The motivation behind transfer learning

Transfer learning provides a cost-effective way of using trained models for other purposes within the same company, such as the food processing company described in *Chapter 9*, *Abstract Image Classification with Convolutional Neural Networks (CNNs)*.

This chapter describes how the food processing company used the model for other similar purposes.

The company that succeeds in doing this will progressively generalize the use of the solution. By doing so, inductive abstraction will take place and lead to other AI projects, which will prove gratifying to the management of a corporation and the teams providing the solutions.

Inductive thinking

Induction uses inferences to reach a conclusion. For example, a food processing conveyor belt with missing products will lead to packaging productivity problems. If an insufficient amount of products reaches the packaging section,

this will slow down the whole production process.

By observing similar problems in other areas of the company, inferences from managers will come up, such as *if insufficient amounts of products flow through the process, production will slow down*.

Inductive abstraction

The project team in charge of improving efficiency in any company needs to find an *abstract representation* of a problem to implement a solution through organization or software. This book deals with the AI side of solving problems. Organizational processes need to define how AI will fit in, with several on-site meetings.

The problem AI needs to solve

In this particular example, each section of the factory has an **optimal production rate** (**OPR**) defined per hour or per day, for example. The equation of an OPR per hour can be summed up as follows:

$$\text{OPR} : \min(p(s)) <= \text{OPR} <= \max(p(s))$$

Where:

- p is the production rate of a given section (the different production departments of a factory) s.
- $p(s)$ is the production rate of the section.
- $\min(p(s))$ is the historical minimum (trial and error over months of analysis). Under that level, the whole production process will slow down.
- $\max(p(s))$ is the historical maximum. Over that level, the whole production process will slow down as well.
- OPR is the optimal production rate.

The first time somebody sees this equation, it seems difficult to understand. The difficulty arises because you have to visualize the process, which is the goal of this chapter. Every warehouse, industry, and service uses production rates as a constraint to reach profitable levels.

Visualization requires representation at two levels:

- Ensuring that if a packaging department is not receiving enough products, it will have to slow down or even stop sometimes.
- Ensuring that if a packaging department receives too many products, it will not be able to package them. If the input is a conveyor belt with no intermediate storage (present-day trends), then it will have to be slowed down, slowing down or stopping the processes before that point.

In both cases, slowing down production leads to bad financial results and critical sales problems through late deliveries.

In both cases, an OPR gap is a problem. To solve this problem, another level of abstraction is required. First, let's break down the OPR equation into two parts:

$$\text{OPR} >= \min(p(s))$$

$$\text{OPR} <= \max(p(s))$$

Now let's find a higher control level through variance variable v:

$$v_{min} = |\text{OPR} - \min(p(s))|$$

$$v_{max} = |\text{OPR} - \max(p(s))|$$

v_{min} and v_{max} are the absolute values of the variance in both situations (not enough products to produce and too many to produce respectively).

The final representation is through a single control, detection, and learning rate (the Greek letter gamma):

$$\Gamma = \max(v_{min}, v_{max})$$

The variance between the optimal production rate of a given section of a company and its minimum speed (products per hour) will slow the following section down. If too few cakes (v_{min}), for example, are produced, then the cake packaging department will be waiting and will have to stop. If too many cakes are produced (v_{max}), then the section will have to slow down or stop. Both variances would create problems in a company that cannot manage intermediate

storage easily, which is the case with the food processing industry.

With this single Γ concept, introduced in *Chapter 9, Abstract Image Classification with Convolutional Neural Networks (CNNs)*, the TensorFlow 2.x CNN can start learning a fundamental production concept: what a physical gap is. Let's go back to humans. Once we understand that a gap is some kind of hole or empty space, we can identify and represent thousands of situations with that one gap concept that is here converted into a parameter named gamma (Γ). Let's explore the concept and then implement it.

The Γ gap concept

Teaching the CNN the gap concept will help it extend its thinking power to many fields:

- A gap in production, as explained before
- A gap in a traffic lane for a self-driving vehicle to move into
- Any incomplete, deficient area
- Any opening or window

Let's teach a CNN the Γ gap concept, or simply, Γ. The symbol Γ of a gap is the Greek letter "gamma," so it is simply pronounced "gamma." We thus lead to teaching a CNN how to recognize a gap we will call gamma (Γ). The goal is for a CNN to understand the abstract concept of an empty space, a hole represented by the word *gap* and the Greek letter gamma (Γ).

To achieve that goal, the CNN model that was trained and saved in *Chapter 9, Abstract Image Classification with Convolutional Neural Networks (CNNs)*, now needs to be loaded and used. To grasp the implications of the Γ concept, imagine the cost of not producing enough customer orders or having piles of unfinished products everywhere. The financial transposition of the physical gap is a profit **variance** on set goals. We all know the pain those variances lead to.

Loading the trained TensorFlow 2.x model

The technical goal is to load and use the trained CNN model and then use the

same model for other similar areas. The practical goal is to teach the CNN how to use the **Γ concept** to enhance the thinking abilities of the scheduling, chatbot, and other applications.

Loading the model has two main functions:

- Loading the model to compile and classify new images without training the model
- Displaying the parameters used layer by layer and displaying the weights reached during the learning and training phase

In the following section, we will load and display the model without training it.

Loading and displaying the model

A limited number of headers suffice to read a saved model with `READ_MODEL.py`, as implemented in the following lines:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
import numpy as np
from PIL import Image
#Directory
directory='dataset/'
print("directory",directory)
```

The `model3.h5` model saved is now loaded from its file, as shown here:

```
#_____________________LOAD MODEL_____________________________
loaded_model = keras.models.load_model(directory+"model/mod
el3.h5")
print(loaded_model.summary())
```

The loaded model needs to be compiled:

```
# ____________________compile loaded model
```

```
loaded_model.compile(loss='binary_crossentropy', optimizer=
'rmsprop', metrics=['accuracy'])
```

Reading and displaying the model is not a formality.

Printing the structure provides useful information:

```
print("GLOBAL MODEL STRUCTURE")
print(loaded_model.summary())
```

The trained model might or might not work on all datasets. In that case, the following output would point to problems that can be fixed through its structure, for example, as follows:

```
MODEL STRUCTURE
Model: "sequential"
_________________________________________________________________
_____
Layer (type)                 Output Shape              Para
m #
=================================================================
======
conv2d (Conv2D)              (None, 62, 62, 32)        896

_________________________________________________________________
_____
max_pooling2d (MaxPooling2D) (None, 31, 31, 32)        0

_________________________________________________________________
_____
conv2d_1 (Conv2D)            (None, 29, 29, 32)        9248

_________________________________________________________________
_____
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 32)        0

_________________________________________________________________
_____
flatten (Flatten)            (None, 6272)              0

_________________________________________________________________
_____
dense (Dense)                (None, 128)               8029
44
```

```
_________________________________________________________________
_____
dense_1 (Dense)              (None, 1)                 129

=================================================================
======
Total params: 813,217
Trainable params: 813,217
Non-trainable params: 0
```

Once the global structure has been displayed, it is possible to look into the structure of each layer. For example, we can peek into the `conv2d` layer:

```
DETAILED MODEL STRUCTURE
{'name': 'conv2d', 'trainable': True, 'batch_input_shape': (
None, 64, 64, 3), 'dtype': 'float32', 'filters': 32, 'kerne
l_size': (3, 3), 'strides': (1, 1), 'padding': 'valid', 'da
ta_format': 'channels_last', 'dilation_rate': (1, 1), 'acti
vation': 'relu', 'use_bias': True, 'kernel_initializer': {'
class_name': 'GlorotUniform', 'config': {'seed': None}}, 'b
ias_initializer': {'class_name': 'Zeros', 'config': {}}, 'k
ernel_regularizer': None, 'bias_regularizer': None, 'activi
ty_regularizer': None, 'kernel_constraint': None, 'bias_con
straint': None}
{'name': 'max_pooling2d', 'trainable': True, 'dtype': 'floa
t32', 'pool_size': (2, 2), 'padding': 'valid', 'strides': (2
, 2), 'data_format': 'channels_last'}
```

Each parameter contains very useful information. For example, `'padding':'valid'` means that padding has not been applied. In this model, the number and size of the kernels provide satisfactory results without padding, and the shape decreases to the final status layer (classification), as shown here:

```
initial shape (570, 597, 4)
lay: 1 filters shape (568, 595, 3)
lay: 2 Pooling shape (113, 119, 3)
lay: 3 filters shape (111, 117, 3)
lay: 4 pooling shape (22, 23, 3)
lay: 5 flatten shape (1518,)
lay: 6 dense shape (128,)
lay: 7 dense shape (1,)
```

However, suppose you want to control the output shape of a layer so that the spatial dimensions do not decrease faster than necessary. One reason could be that the next layer will explore the edges of the image and that we need to explore them with kernels that fit the shape.

In that case, padding of size 1 can be added with `0` values, as shown in the following matrix:

0	0	0	0	0	0
0	**1**	**3**	**24**	**4**	0
0	**3**	**7**	**8**	**5**	0
0	**6**	**4**	**5**	**4**	0
0	**5**	**4**	**3**	**1**	0
0	0	0	0	0	0

A padding of size 2 would add two rows and columns around the initial shape.

With that in mind, fine-tuning your training model by adding as many options as necessary will improve the quality of the results. The weights can be viewed by extracting them from the saved model file layer by layer, as shown in the following code snippet:

```
print("WEIGHTS")
for layer in loaded_model.layers:
    weights = layer.get_weights() # list of numpy arrays
    print(weights)
```

Analyzing the weights used by the program will provide useful information about the way the optimization process was carried out by the program. Sometimes, a program will get stuck, and the weights might seem off track. After all, a CNN can contain imperfections like any other program.

A look at the following output, for example, can help understand where the system went wrong:

```
WEIGHTS
[array([[ 6.25981949e-03,  2.35006157e-02, -1.28920656e-02,
 ...,
        -8.34930502e-03,  2.00010985e-02, -1.84428487e-02],
       [-1.01672988e-02,  1.87084991e-02,  2.49958578e-02,
...,
        -2.92361379e-02, -2.33592112e-02, -1.64737436e-03],
       [-2.71108598e-02,  2.53492035e-03, -2.90711448e-02,
...,
```

We can now use the loaded and checked model.

Loading the model to use it

Loading the model with `CNN_CONCEPT_STRATEGY.py` requires a limited number of headers, as follows:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
import numpy as np
from PIL import Image
```

Loading the model is done by using the same code as in `READ_MODEL.py`, described previously. Once you load it, compile the model with the `model.compile` function, as follows:

```
# ____________________compile loaded model
loaded_model.compile(loss='binary_crossentropy', optimizer=
'rmsprop', metrics=['accuracy'])
```

The model used for this example and the image identification function has been implemented in two parts. First, we're loading and resizing the image with the following function, for example:

```
def identify(target_image):
    filename = target_image
    original = load_img(filename, target_size=(64, 64))
    #print('PIL image size',original.size)
    if(display==1):
        plt.imshow(original)
        plt.show()
    numpy_image = img_to_array(original)
    inputarray = numpy_image[np.newaxis,...] # extra dimens
ion to fit model
    arrayresized=np.resize(inputarray,(64,64))
    #print('Resized',arrayresized)
```

The model expects another dimension in the input array to predict, so one is added to fit the model. In this example, one image at a time needs to be identified.

I added the following two prediction methods and returned one:

```
#___PREDICTION___
    prediction = loaded_model.predict_proba(inputarray)
    return prediction
```

There are two prediction methods because, basically, every component needs to be checked in a CNN during a project's implementation phase, to choose the best and fastest ones. To test `prediction2`, just change the `return` instruction.

> Once a CNN is running, it can prove difficult to find out what went wrong. Checking the output of each layer and component while building the network saves fine-tuning time once the full-blown model produces thousands of results.

The following example detects product **Γ gaps** on a conveyor belt in a food processing factory. The program loads the first image stored in the `classify` directory to predict its value. The program describes the prediction:

```
MS1='productive'
MS2='gap'
s=identify(directory+'classify/img1.jpg')
if (int(s)==0):
    print('Classified in class A')
```

```
    print(MS1)
```

The program displays (optional) the shaped image, as follows, which shows that the conveyor belt has a sufficient number of products at that point:

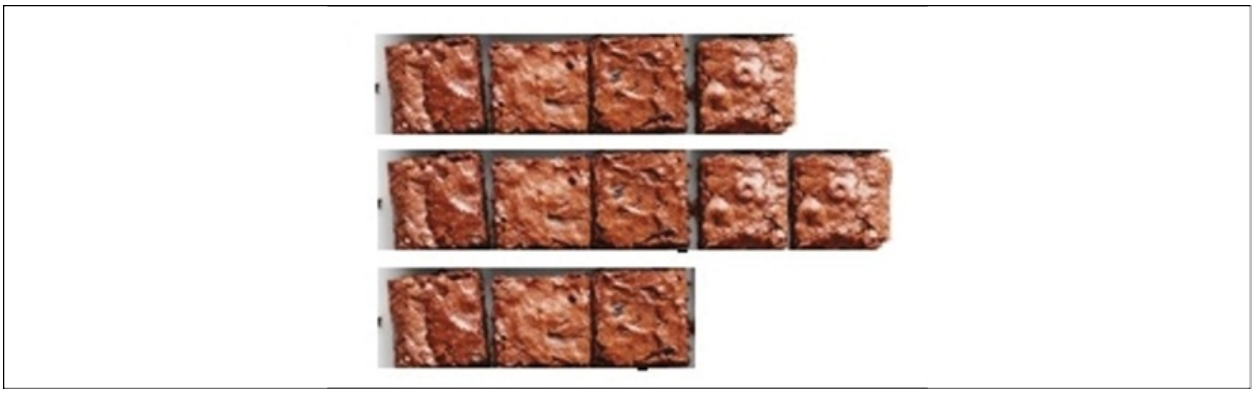


Figure 10.1: Output (shaped image)

The program then makes and displays its prediction `0`, meaning no real gap has been found on the conveyor belt on this production line:

```
directory dataset/
Strategy model loaded from training repository.
image dataset/classify/img1.jpg predict_proba: [[ 0.]] pred
ict: [[ 0.]]
Classified in class A
Productive
Seeking...
```

`Seeking...` means it is going to analyze the second image in the classify direction. It loads, displays, and predicts its value as shown in the following frame:

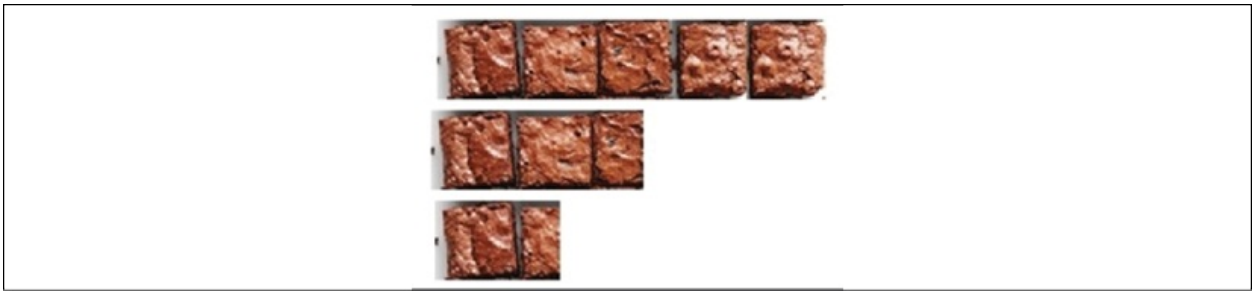


Figure 10.2: Output (shaped image)

The prediction (*value* = 1) correctly detected gaps on the conveyor belt, as shown in the following output:

```
image dataset/classify/img2.jpg predict_proba: [[ 1.]] pred
ict: [[ 1.]]
Classified in class B
gap
```

Now that the predictions of the CNN have been verified, the implementation strategy needs approval. A CNN contains marvels of applied mathematics. CNNs epitomize deep learning themselves. A researcher could easily spend hundreds of hours studying them.

However, applied mathematics in the business world requires profitability. As such, the components of CNNs appear to be ever-evolving concepts. Added kernels, activation functions, pooling, flattening, dense layers, and compiling and training methods act as a starting point for architectures, not as a finality.

Using transfer learning to be profitable or see a project stopped

At some point, a company will demand results and may shelve a project if those results are not delivered. If a spreadsheet represents a faster sufficient solution, a deep learning project will face potential competition and rejection. Many engineers learning artificial intelligence have to assume the role of standard SQL reporting experts before accessing real AI projects. Transfer learning is a profitable solution that can boost the credibility of an IT department.

Transfer learning appears to be a solution to the present cost of building and training a CNN program. Your model might just pay off that way. The idea is to get a basic AI model rolling profits in fast for your customer and management. Then, you will have everybody's attention. To do that, you must define a strategy.

Defining a strategy

If a deep learning CNN expert comes to a top manager saying that this CNN model can classify CIFAR-10 images of dogs, cats, cars, plants, and more, the answer will be, *so what? My 3-year-old child can too. In fact, so can my dog!*

The IT manager in that meeting might even blurt out something like, "We have all the decision tools we need right now, and our profits are increasing. Why would we invest in a CNN?"

The core problem of marketing AI to real-world companies is that it relies upon a belief in the necessity of a CNN in the first place. Spreadsheets, SQL queries, standard automation, and software do 99% of the job. Most of the time, it does not take a CNN to replace many jobs; just an automated spreadsheet, a query, or standard, straightforward software is enough. Jobs have been sliced into simple-enough parts to replace humans with basic software for decades.

Before presenting a CNN, a data scientist has to find out how much the company can earn using it.

> Understanding, designing, building, and running a CNN does not mean much regarding business. All the hard work we put into understanding and running these complex programs will add up to nothing if we cannot prove that a solution will generate profit. Without profit, the implementation costs cannot be recovered, and nobody will listen to a presentation about even a fantastic program.

Applying a model efficiently means implementing it in one area of a company and then other domains for a good return on investment.

Applying the model

In a food processing company, for example, one of the packaging lines has a performance problem. Sometimes, randomly, some of the cakes are missing on the conveyor belt, as shown in the following frame:

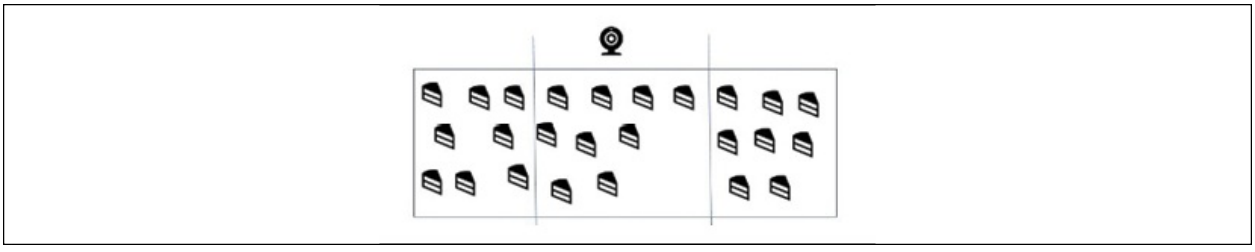


Figure 10.3: Food processing company example

To start a cost-effective project, a cheap webcam could be installed over the

conveyor belt. It'll take a random sample picture every 10 seconds and process it to find the holes shown in the interval in the center of the image. We can clearly see an empty space, a gap, a hole. If a hole is detected, it means some cakes have not made it to the conveyor belt (production errors).

A 2% to 5% productivity rate increase could be obtained by automatically sending a signal to the production robot when some cakes are missing. The production robot will then send a signal to the production line to increase production to compensate for the missing units in real-time. This type of automatic control already exists in various forms of automated food production lines. However, this provides a low-cost way to start implementing this on a production line.

Making the model profitable by using it for another problem

Let's say that the food processing experiment on the conveyor belt turns out to work well enough with dataset type d_1 and the CNN model M to encourage generalization to another dataset, d_2, in the same company.

Transfer learning consists of going from $M(d_1)$ to $M(d_2)$ using the same CNN model M, with some limited, cost-effective additional training. Variations will appear, but they can be solved by shifting a few parameters and working on the input data following some basic dataset preparation rules:

- **Overfitting**: When the model fits the training data quickly with 100% accuracy, this may or may not be a problem. In the case of classifying holes on the conveyor belt, overfitting might not prove critical. The shapes are always the same, and the environment remains stable. However, in an unstable situation with all sorts of different images or products, then overfitting will limit the effectiveness of a system.
- **Underfitting**: If the accuracy drops down to low levels, such as 20%, then the CNN will not work. The datasets and parameters need optimizing. Maybe the number of samples needs to be increased for $M(d_2)$, or reduced, or split into different groups.
- **Regularization**: Regularization, in general, involves the process of finding how to fix the generalization problem of $M(d_2)$, not the training errors of

$M(d_2)$. Maybe an activation function needs some improvements, or the way the weights have been implemented requires attention.

There is no limit to the number of methods you can apply to find a solution, just like standard software program improvements.

Where transfer learning ends and domain learning begins

Transfer learning can be used for similar types of objects or images in this example, as explained. The more similar images you can train within a company with the same model, the more return on investment (ROI) it will produce, and the more this company will ask you for more AI innovations.

Domain learning takes a model such as the one described in *Chapter 9, Abstract Image Classification with Convolutional Neural Networks (CNNs)*, and can generalize it. The generalization process will lead us to domain learning.

Domain learning

This section on domain learning builds a bridge between classic transfer learning, as described previously, and another use of domain learning I have found profitable on corporate projects: teaching a machine a concept (CRLMM). In this chapter, we are focusing on teaching a machine to learn how to recognize a gap in situations other than at the food processing company.

How to use the programs

You can read this whole chapter first to grasp the concepts or play with the programs first. Do as you feel is best for you. In any case, `CNN_TDC_STRATEGY.py` loads trained models (you do not have to train them again for this chapter) and `CNN_CONCEPT_STRATEGY.py` trains the models.

The trained models used in this section

This section uses `CNN_TDC_STRATEGY.py` to apply the trained models to the target

concept images. `READ_MODEL.py` (as shown previously) was converted into `CNN_TDC_STRATEGY.py` by adding variable directory paths (for the `model3.h5` files and images) and classification messages, as shown in the following code:

```
#loads,traffic,food processing
A=['dataset_0/','dataset_traffic/','dataset/']
MS1=['loaded','jammed','productive']
MS2=['unloaded','change','gap']
#_____________________LOAD MODEL_____________________________
loaded_model = keras.models.load_model(directory+"model/mod
el3.h5")....")
```

The loaded model now targets the images to classify:

```
s=identify(directory+'classify/img1.jpg')
```

Each subdirectory of the model contains four subdirectories:

- `classify`: Contains the images to classify
- `model`: The trained `model3.h5` used to classify the images
- `test_set`: The test set of conceptual images
- `training_set`: The training set of conceptual images

Now that we have explored the directory structure of our model, let's see how to use it in different situations.

The trained model program

For this chapter, you do not need to train a model. It was already trained in *Chapter 9, Abstract Image Classification with Convolutional Neural Networks (CNNs)*. The directory paths have become variables to access the subdirectories described previously. The paths can be called, as shown in the following code:

```
A=['dataset_0/','dataset_traffic/','dataset/']
scenario=3 #reference to A
directory=A[scenario] #transfer learning parameter (choice
of images)
print("directory",directory)
```

You do not need to run training for this chapter. The models were trained and automatically stored in their respective subdirectories on the virtual machine delivered with the book. This means that when you need to detect gaps for various types of images, you can simply change the scenario to fit the type of images you will be receiving from the frames of a webcam: cakes, cars, fabric, or abstract symbols.

> For this chapter, focus on understanding the concepts. You can read the chapter without running the programs, open them without running them, or run them—whatever makes you comfortable. The main goal is to grasp the concepts to prepare for the subsequent chapters.

We have loaded the model and a scenario. Now, we are going to use our trained model to detect if a production line is loaded or underloaded.

Gap – loaded or underloaded

The gap concept has just become a polysemy image-concept (polysemy means different meanings, as explained in *Chapter 6, Innovating AI with Google Translate*).

In the cake situation, the Γ gap was negative in its g_1 subset of meaning and concepts applied to a CNN, relating it to negative images $n + g_1$:

$$ng_1 = \{\text{missing, not enough, slowing production down … bad}\}$$

The full-of-products image was positive, $p + g_2$:

$$pg_2 = \{\text{good production flow, no gap}\}$$

In this example, the CNN is learning how to distinguish an abstract representation, not simply an image, like for the cakes. Another subset of Γ (the conceptual gap dataset) is loaded/underloaded. A "gap" is not a specific object but a general concept that can be applied to hundreds of different cases. This is why I use the term "conceptual gap dataset."

The following abstract image is loaded. The squares represent production

machines, and the arrows represent the load-in time.

This means that the *x* axis represents time and the *y* axis represents machine production resources:

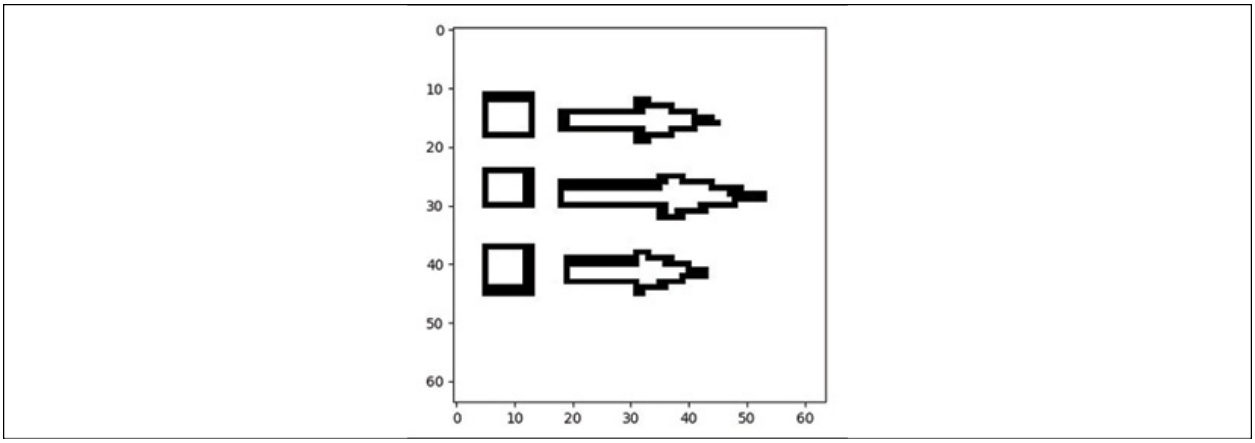


Figure 10.4: Abstract image 1

The CNN model runs and produces the following result:

```
directory dataset_O/
Strategy model loaded from training repository.
image dataset_O/classify/img1.jpg predict_proba: [[ 0.]] pr
edict: [[ 0.]]
Classified in class A
loaded
Seeking...
```

The CNN recognizes this as a correctly loaded model. The task goes beyond classifying. The system needs to recognize this to make a decision.

Another image produces a different result. In this case, an underloaded gap appears in the following screenshot:

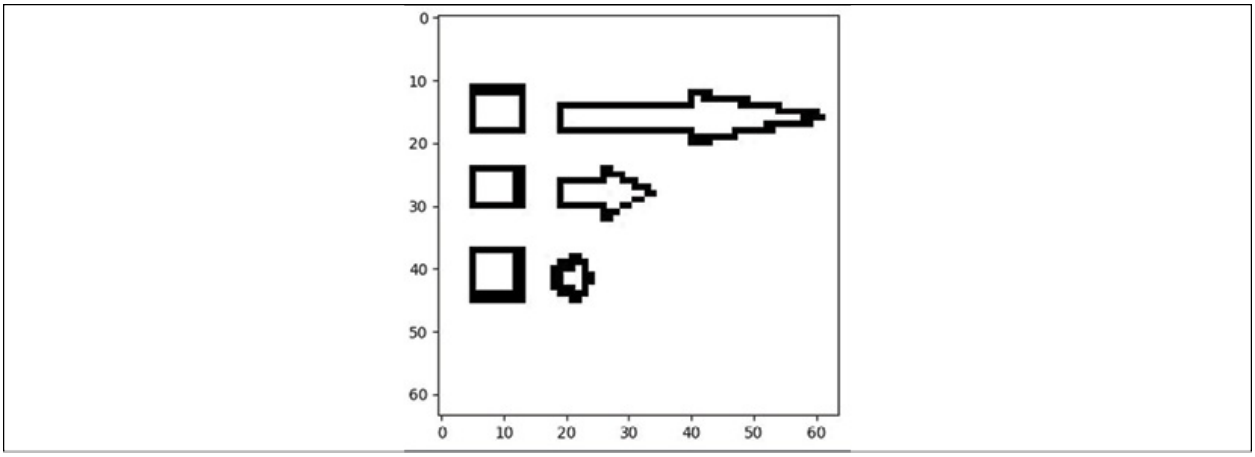


Figure 10.5: Abstract image 2

And the CNN has a different output, as shown here:

```
Seeking...
image dataset_0/classify/img2.jpg predict_proba: [[ 1.]] pr
edict: [[ 1.]]
Classified in class
unloaded
```

Read "unloaded" as "underloaded." Unloaded or underloaded represents empty spaces in any case. The gap concept Γ has added two other subsets, g_3 and g_4, to its dataset. We now have:

$$\Gamma = \{ng_1, g_2, g_3, g_4 \dots g_n\}$$

The four g_1 to g_4 subsets of Γ are:

ng_1 = {missing, not enough, slowing production down … bad}

$pg_2 = pg_2$ = {good production flow, no gap}

g_3 = {loaded}

g_4 = {unloaded}

The remaining problem will take some time to solve. g_4 (gap) can sometimes represent an opportunity for a machine that does not have a good workload to

be open to more production. In some cases, g_4 becomes pg_4 (p = positive). In other cases, it will become ng_4 (n = negative) if production rates go down.

In this section, we saw how to identify a "gap" in production lines. As explained, a "gap" is a generic concept for spaces everywhere. We will now explore jammed or "open" traffic lanes.

Gap – jammed or open lanes

The model in this chapter can be extended to other domains. A self-driving car needs to recognize whether it is in a traffic jam or not. Also, a self-driving car has to know how to change lanes when it detects enough space (a gap) to do that.

This produces two new subsets:

$$g_5 = \{\text{traffic jam, heavy traffic … too much traffic}\}$$

$$g_6 = \{\text{open lane, light traffic … normal traffic}\}$$

The model now detects g_5 (a traffic jam), as shown in the following screenshot:

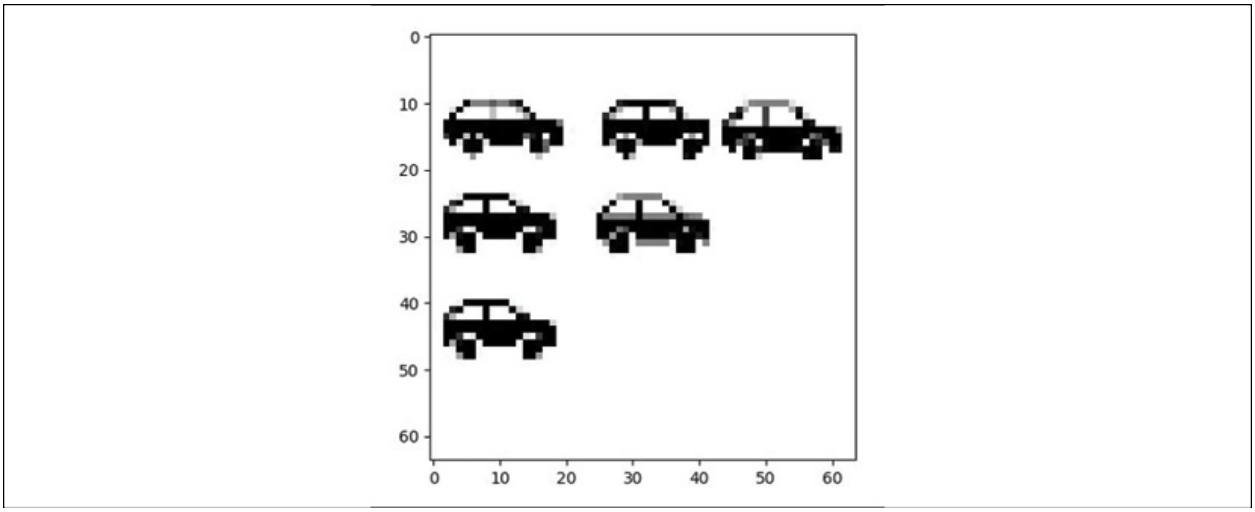


Figure 10.6: Traffic jam example

The following output appears correctly:

```
directory dataset_traffic/
Strategy model loaded from training repository.
image dataset_traffic/classify/img1.jpg predict_proba: [[ 0
.]] predict: [[ 0.]]
Classified in class A
jammed
```

g_6 comes out right as well, as shown in this screenshot:

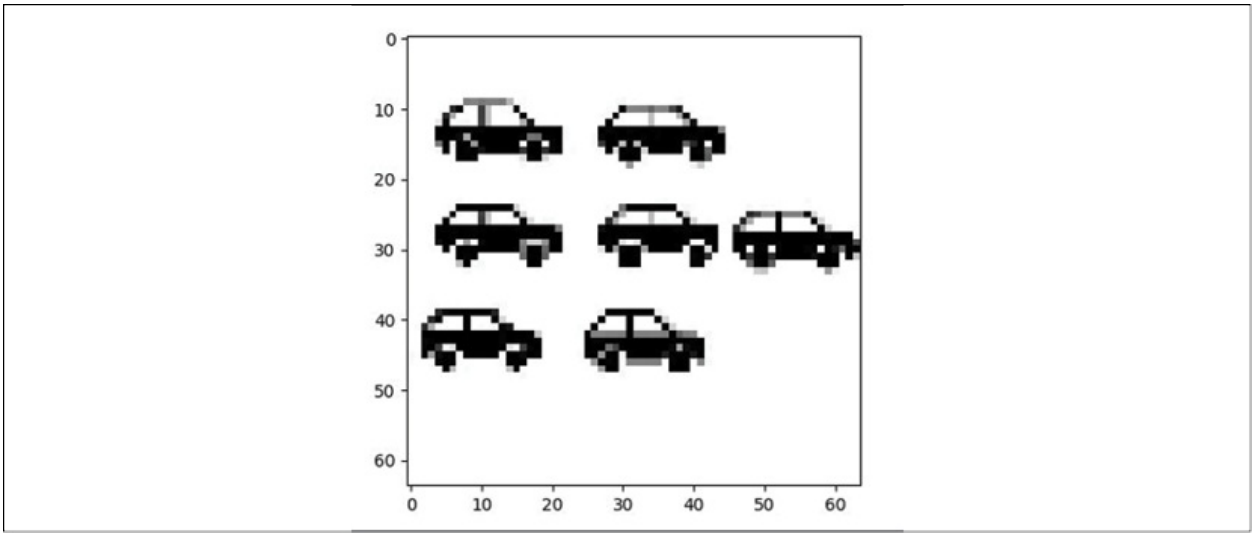


Figure 10.7: Traffic jam example

A potential lane change has become possible, as detected by the following code:

```
Seeking...
image dataset_traffic/classify/img2.jpg predict_proba: [[ 1
.]] predict: [[ 1.]]
Classified in class B
change
```

We have applied our CNN "gap" detection model to several types of images. We can now go deeper into the theory of conceptual datasets using "gaps" as an example.

Gap datasets and subsets

At this point, the Γ (the gap conceptual dataset) has begun to learn several

subsets:

$$\Gamma = \{g_1, g_2, g_3, g_4, g_5, g_6\}$$

In which:

ng_1 = {missing, not enough, slowing production down … bad}

$pg_2 = pg_2$ = {good production flow, no gap}

g_2 = {loaded}

g_3 = {unloaded}

pg_4 = {traffic jam, heavy traffic … too much traffic}

ng_5 = {open lane, light traffic … normal traffic}

Notice that g_2 and g_3 do not have labels yet. The food processing context provided the labels. Concept detection requires a context, which CRLMMs will provide.

Generalizing the Γ (the gap conceptual dataset)

The generalization of Γ (the gap conceptual dataset) will provide a conceptual tool for metamodels.

> Γ (the gap conceptual dataset) refers to negative, positive, or undetermined space between two elements (objects, locations, or products on a production line).

Γ (gamma) also refers to a gap in time: too long, not long enough, too short, or not short enough.

Γ represents the distance between two locations: too far or too close.

Γ can represent a misunderstanding or an understanding between two parties: a divergence of opinions or a convergence.

All of these examples refer to gaps in space and time viewed as space.

The motivation of conceptual representation learning metamodels applied to dimensionality

A CRLMM converts images into concepts. These abstract concepts will then be embedded in vectors that become logits for a softmax function, and in turn, will be converted into parameters for complex artificial intelligence programs' automatic scheduling, cognitive chatbots, and more.

The advantage of a concept is that it can apply to many different areas. With just one concept, "gap" (a hole, empty space, and so on), you can describe hundreds if not thousands of cases.

In some artificial intelligence projects, dimensionality reduction does not produce good results at all. When scheduling maintenance of airplanes, rockets, and satellite launchers, for example, thousands of features enter the system without leaving any out. A single missing screw in a rocket in the wrong place could cause a disaster. A single mistake in the engine of an airplane can cause an accident, a single component of a satellite can impair its precision.

Dimensionality must be taken into account. Some use the expression "curse of dimensionality" and I often prefer the "blessing" of dimensionality. Let's have a look at both approaches.

The curse of dimensionality

The number of features for a given project can reach large numbers. The following example contains 1,024 dimensions:

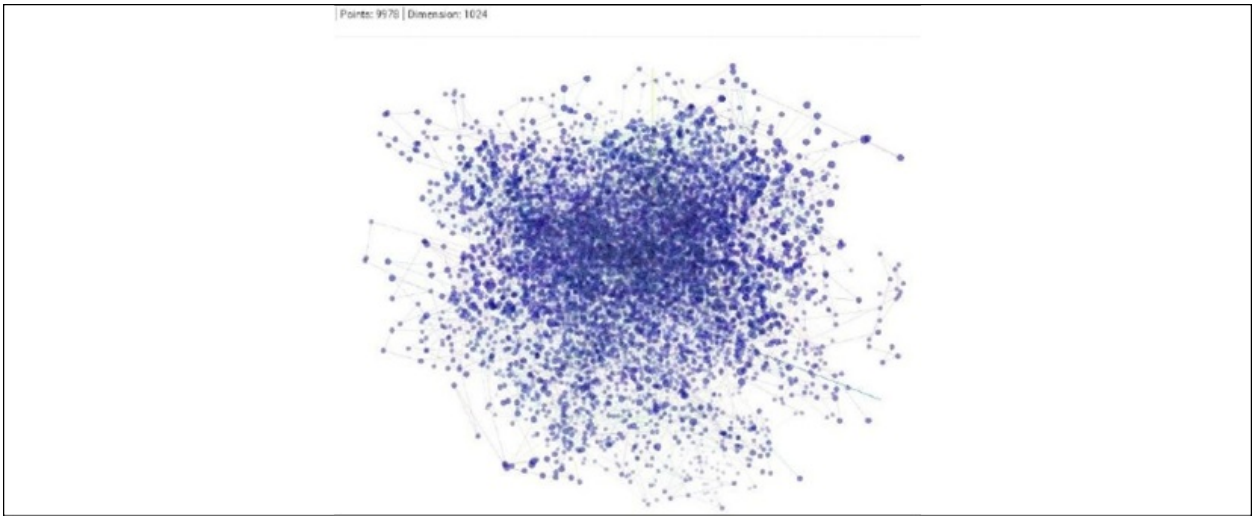


Figure 10.8: The curse of dimensionality

Each dot in the preceding representation represents a dimension that can be the features in an image for example.

Asking a CNN to analyze thousands of features in an image, for example, might make it impossible to reach an accurate result. Since each layer is supposed to reduce the size of the data analyzed to extract important features, too many dimensions might make the training of the model impossible. Remember, each dimension can contain a feature that requires weights to be trained. If there are too many, the training becomes either too long or too difficult to calculate.

Following standard CNN designs provides a good starting point. The limit of this approach occurs when the result does not meet expectations, such as in some of the cases we will look at in the upcoming chapters. In those cases, CRLMMs will increase the productivity of the solution, providing a useful abstract model.

When a solution requires a large number of unreduced dimensions, kernels, pooling, and other dimension reduction methods cannot be applied, CRLMMs will provide a *pair of glasses* for the system. That's when the "blessing" of dimensionality comes in handy.

The blessing of dimensionality

In some projects, when the model reaches limits that comprise a project, dimensionality is a blessing.

Let's take an example of rocket manufacturing using our CNN model. We want to identify gaps on a surface. This surface contains tiles to protect the rocket from overheating when it goes through the atmosphere. A gap in those tiles could cause a fatal accident.

If we take a few tiles out, take a picture and run it through our CNN model, it will *probably* detect a gap. The difference between that probability and an error could mean a critical failure of the rocket.

This means that we might not want to reduce the number of features, which in turn need weights and add up to high numbers of dimensions.

We could decide not to use pooling, which groups several dimensions into one as we saw. That could create calculations problems as we saw in the previous paragraph. Either there would be too many weights to calculate or the calculation could take too long.

In that case, we could reduce the size of the frame to the smallest portion of the rocket component we are examining. We could decide that our camera will only scan the smallest surface possible at a time sending minimal-sized frames to our CNN.

In that case, even with no pooling, the layers would contain more data, but the calculation would remain reasonable.

The "blessing" of dimensionality, in this case, resides in the fact that by avoiding pooling (grouping), we are examining more details that can make our model much more reliable to detect small cracks since we would train it to see very small gaps.

The curse of dimensionality usually leads to dimensionality reduction. But as we have just seen, it doesn't have to be so.

Summary

In this chapter, the CNN architecture built in *Chapter 9, Abstract Image Classification with Convolutional Neural Networks (CNNs),* was loaded to

classify physical gaps in a food processing company. The model uses image concepts, taking CNNs to another level. Neural networks can tap into their huge cognitive potential, opening the doors to the future of AI.

Then, the trained models were applied to transfer learning by identifying similar types of images. Some of those images represented concepts that led the trained CNN to identify ∏ concept gaps. Image concepts represent an avenue of innovative potential adding cognition to neural networks.

∏ concept gaps were applied to different fields using the CNN as a training and classification tool in domain learning.

∏ concept gaps have two main properties: negative n-gaps and positive p-gaps. To distinguish one from the other, a CRLMM provides a useful add-on. In the food processing company, installing a webcam on the right food processing conveyor belt provided a context for the system to decide whether a gap was positive or negative.

With these concepts in mind, let's build a solution for **advanced planning and scheduling** (**APS**) in the next chapter.

Questions

1. The curse of dimensionality leads to reducing dimensions and features in machine learning algorithms. (Yes | No)
2. Transfer learning determines the profitability of a project. (Yes | No)
3. Reading `model.h5` does not provide much information. (Yes | No)
4. Numbers without meaning are enough to replace humans. (Yes | No)
5. Chatbots prove that body language doesn't mean that much. (Yes | No)
6. Present-day ANNs provide enough theory to solve all AI requests. (Yes | No)
7. Chatbots can now replace humans in all situations. (Yes | No)
8. Self-driving cars have been approved and do not need conceptual training. (Yes | No)

9. Industries can implement AI algorithms for all of their needs. (Yes | No)

Further reading

- More on Keras layers: https://keras.io/layers/about-keras-layers/
- More on concept learning: https://www.sciencedirect.com/topics/psychology/concept-learning
- More on cognitive sciences, the brain, and the mind for conceptual models: http://catalog.mit.edu/schools/science/brain-cognitive-sciences/, https://symsys.stanford.edu/undergraduatesconcentrations/cognitive-science-cogsci-concentration

11

Combining Reinforcement Learning and Deep Learning

Amazon is one of the world's leading e-retailers, with sales over US$ 250 billion. Amazon's e-store sales exceed all of its other activities, such as AWS subscription services (premium, for example), retail third-party seller services, and physical stores.

This chapter focuses on apparel production activity, one of Amazon's markets for which the company recently registered a patent. Prime Wardrobe even offers a try-and-easy-return service. This new activity requires planning and scheduling. Amazon took the matter seriously and registered a patent for an apparel manufacturing system to control the production process of its apparel products.

> Artificial intelligence already plays a role in automatic planning and scheduling in the apparel business, from customer orders through to delivery.

Google has successfully combined deep learning and reinforcement learning (Q-learning) in a **deep Q-network** (**DQN**), a system that can beat humans at video games and other tasks. Google's AlphaGo DQN has obtained impressive results.

In this chapter, we go beyond merely describing Amazon's process or Google's process in particular. I added on my real-life implementations of what we will explore and build in Python from scratch. As such, we will combine the ideas of several systems (Amazon, Google, and my implementations).

We will add an innovation for the apparel manufacturing industry by adding a **conceptual representation learning metamodel** (**CRLMM**) to reinforcement learning.

We will go from scratch to a prototype that could be implemented on-site, the foundations are being established for further applications in the coming chapters.

The following topics will be covered in this chapter:

- Planning and scheduling today and tomorrow
- Further generalization of the CRLMM described in *Chapter 10, Conceptual Representation Learning*, applied to an apparel production process
- Feeding the CRLMM **convolutional neural network (CNN)** with a simulation of frames coming from a webcam on a production line
- Introducing an optimizer that will use weights applied to production stations to input a reward matrix to a **Markov decision process (MDP)**, which will then update the weights
- Building a program that will run continuously (no beginning, no end) on a production line using all the three components mentioned previously

We'll begin by talking about planning and scheduling today and tomorrow. The market is slowing moving from preplanned processes to real-time processes. Let's see how.

Planning and scheduling today and tomorrow

When Amazon decided to launch Prime Wardrobe, it brought a new service to its customers, enabling them to order, try out, and purchase clothing, shoes, and other accessories. The customer can establish a purchase plan. A purchase plan is a list of tasks to be carried out in a given time. An example of a purchase plan could be:

- Filling a box with clothing
- Trying on the clothing at home
- Returning the clothing if it does not fit
- Purchasing the items that are kept

Once the customer agrees to follow this plan, the time sequence becomes crucial:

- First, a box must be chosen.
- Then, there is a delivery period.
- Then, there is a trial period (you cannot try the products forever). During this period, the customer can choose not to purchase anything.
- Finally, the customer confirms the purchase.

Irrespective of whether Amazon Prime Wardrobe will remain a service in years to come, the precedent is set; just as physical book stores disappear every year, shopping for clothing online will continue to expand and occupy more of the market.

On top of that distribution, corporations will continue to expand their production sites to become manufacturing-distributing giants. Warehouses will progressively replace many stores as per the warehouse examples provided in some of the chapters in this book.

Supply chain management (SCM), combined with APS, has become a necessity. SCM-APS constraints vary constantly on a global market (depending on the manufacturer). APS stands for advanced planning and scheduling or automated planning and scheduling. We will explore the difference between these two concepts in a following section, *A real-time manufacturing revolution*. We will go beyond Amazon's approach to the subject since delivering in real-time has become a constraint for all of the actors on the market.

The pressure of the market has encouraged Amazon to produce its own clothing. Amazon has produced its in-house fashion labels with apparel and accessories sold worldwide.

To prove that it means business, Amazon registered several patents, including one for a *blended reality mirror*, an apparel manufacturing system, and more. With a blended reality mirror, a person can visualize how the clothing will fit.

In the following sections, we will explore the planning and scheduling side of Amazon's apparel in-house manufacturing plans, which will have the same effect on apparel factories as it did on physical book stores and all types of shops. Many new jobs will emerge, such as hundreds of thousands of jobs in the

artificial intelligence business, on websites, and in marketing and SCM. Many jobs will disappear as well. The improvements made will boost medical progress and also the defense industry. The social and economic implications are beyond the scope of this book, and possibly beyond our comprehension, as in all disruptive eras in the past.

We will focus in detail on the main aspects of Amazon's patented custom clothing process. As mentioned previously, irrespective of whether it proves successful, the precedent is set.

A real-time manufacturing process

Today, a customer wants to obtain a purchased product as soon as possible. If the waiting time is too long, the customer will go somewhere else. Delivering in nearly real-time has become a key concept in marketing for any company. Amazon's approach has always been real-time. Pushing the physical limits of centuries of commerce, Amazon's brand takes the manufacturing process right beyond the cutting edge.

Amazon must expand its services to face competition

Amazon could have avoided a great deal of upheaval if it continued to purchase products from its suppliers rather than manufacture them. Instead of researchers spending a great deal of time developing artificial intelligence capable of solving the conveyor belt problem, several sensors could be installed with traditional software solutions.

Humanity could also have continued to use horses instead of cars and paper instead of computers. In the short term, we would have avoided a great deal of work and change associated with the requirements and impacts of these technologies. However, in the long term, it is progress and innovation that wins out.

Once a disruptive innovation has been successfully launched by a company, the competition must either follow suit or disappear. Amazon needs to continue to get involved in the process that comes before simply storing goods in its warehouses. By getting involved in manufacturing, for example, they can reduce the time it takes to deliver a product to a customer. Amazon thus came up with

this apparel manufacturing patent along with 3D printers, and other innovations to increase its productivity.

In the same sense, the field of artificial intelligence needs to be pushed beyond its ever-expanding comfort zone constantly. Researchers must be confronted with difficult industrial problems to gain experience and produce new algorithms capable of achieving higher levels of machine learning and meeting greater challenges.

A real-time manufacturing revolution

Artificial intelligence software, though spectacular, is not the only revolution that is changing our lives forever. In the 1950s and 1960s, the consumers were discovering the joy of purchasing new products such as dishwashers, color televisions, and cool radios. They were so happy to be able to obtain these products that they were willing to wait days if not weeks to obtain the exact model they dreamt of. The supplier could sit on an order for a few days and then get slowly to work.

Today the consumer has lost that patience. If somebody wants to buy a product online, it has to be delivered within days. If not, the consumer will turn to another supplier. This puts pressure on the supplier, who must immediately start the process of delivering. Real-time is that process of starting to get things done in the seconds you are notified of a request.

Real-time is a strong force that is changing every process in the world.

Today, apparel manufacturing, and manufacturing in general, follow an advanced planning and scheduling process. "Advanced" means both a sophisticated algorithm and also an anticipating (planning in advance) process. The planning in advance aspect of AI algorithms is going through a revolution.

Amazon, as all manufacturing processes, requires automated planning and scheduling to meet shortening delivery times.

The fundamental difference between the two systems is the time factor, as shown in the following comparison table. The difference between an advanced and an automated system seems small, but Amazon, like others, will change the course of history through those differences. Both systems, in theory, can do both types

of tasks. In practice, these methods will specialize in their niches in the years to come. To manufacture an airplane, a sophisticated *advanced* algorithm still needs to plan in *advance* (days to weeks) with a lot of manual decision making. To manufacture basic T-shirts, *automated* planning can be done quickly and automatically in real-time (seconds to hours)

The trend of automated planning and scheduling is becoming a time-squashed version of advanced planning and scheduling.

The figures in the following table do not reflect the exact numbers but the trends:

Function	**Advanced planning and scheduling**	**Automated planning and scheduling**
Long-term plan	1 month to 5 years	A few days to less than a month
Short-term plan	1 day	1 minute
Production or event measurement	Taken into account on a daily basis in general	Real-time
Scheduling	1 hour to 1 week	Real-time
Re-planning when there is a problem	1 hour to 1 month	Real-time
Re-scheduling	1 hour to 1 week	Real-time
Resource adjustment	1 day to 1 month	Real-time
Load balancing	1 hour to 1 week	Real-time
Automatic functions of planning and scheduling	80%	99%

Although this table contains approximate information, the underlying trend is

extremely strong. We've seen the differences between the concepts of advanced and automated; let's also clarify the differences between planning and scheduling:

- A **plan** consists of processes preparing production for the future: purchasing material components and adapting human resources and physical resources. Amazon, for example, has a yearly business plan: putting together the necessary resources in advance to be ready for production. This implies purchasing or building warehouses, hiring employees, and purchasing basic material resources (boxes, labels, and other components).
- **Scheduling** confronts the plan with the time factor on a shorter horizon. Scheduling determines when each part of the plan will come first and be produced. For example, now that the warehouse is built or purchased (plan), at what time (schedule) should packaging start working next Monday and for the weeks to come?

A schedule can be regarded as a zoomed-in version of a plan.

If somebody walks into Jeff Bezos' office with a plan to build a warehouse at a given location with the cost and general timing of the implementation, that is fine. That person is presenting a two-year project. The project will start in 10 months and go on for two years.

Then, that person might (I suggest not!) say: *The plan is great since it will be ready in 10 months. But I'm worried about the daily schedule in 1 year of shift #2 at 4 p.m. Should they start at 7:30 a.m. or 7:45?* Jeff Bezos will no longer be listening. That zoom level is not part of his job description. He has to focus on a higher level. It is useless for a top manager to know what's going to happen a year from now at 4 p.m.!

An advanced planning and scheduling system mostly imports data from ERPs to make plans in the future. An automated planning and scheduling program mostly detects data with sensors to react and optimize in real-time. The present chapter deals with automated planning programs, not advanced planning and scheduling systems.

Think of the evolution between an advanced APS and an automated APS as a

logistic sigmoid function applied to planning-scheduling time squashing. Here are some examples:

- The advanced plan to manufacture a car spans a month to a year.
- The Google Maps itinerary automated plan for a self-driving car at its starting point: a few seconds to a minute, depending on the connection status of that location.
- The schedule to manufacture a car: 1 day to 1 month.
- The schedule to pilot a self-driving car following the Google Maps itinerary = *real-time*

To sum it up, the present trend represents a revolution in the history of human processes.

> Amazon's manufacturing patent reflects the revolution of real-time applied to every field.

Planning has moved up to planning in real-time, as shown in the following equation:

$$Z = \sum_{t_{xn}}^{t_{xn}} \frac{1}{1 + e^{-t_x}} \lambda$$

In which:

- x is a quantity to produce or any unit event.
- t_x is the time (t) it takes for x to start and end.
- A logistic function squashes t_x.
- λ (lambda) is the learning factor; the more a task is carried out, the more it is optimized.

X_t is the total weight of a group of n tasks at a given time t:

$$X_t = \{x_1, x_2, x_3 \dots x_n\}$$

The difference between X_t and $Z(X_t)$ is:

- X_t is the actual time it takes to manufacture products.
- $Z(X_t)$ is not the actual time it takes. $Z(X_t)$ is the result of an activation function that squashes time as an output of X_t in an RL-DL-CRLMM network. $Z(X_t)$ is a weighting factor.

The key to reducing the weighting factor further is a physical production process and other physical events (outputs, for example), that I call lambda: λ signifies all the improvements in real-life production that can reduce the production cycle as well as the natural events, such as production output.

In *Chapter 10, Conceptual Representation Learning*, Γ was introduced to reduce gaps. In this chapter, Γ will be generalized a step further to optimize λ.

This means that an RL-DL-CRLMM system will optimize the manufacturing process. We will now explore a game-changing automated apparel manufacturing process and build it in Python.

CRLMM applied to an automated apparel manufacturing process

With an automated planning and scheduling system, not an advanced planning and scheduling system, Amazon has brought apparel manufacturing closer to the consumer.

Artificial intelligence will boost existing processes. In this section, an RL-DL-CRLMM system will optimize an apparel manufacturing process.

An apparel manufacturing process

Amazon's apparel manufacturing patent can be summarized as follows:

- **P1**: Grouping apparel customer orders by products and sizes. This process has been around since the origins of industrial apparel manufacturing centuries ago.
- **P2**: Automatically cutting lays. A lay is a stack of pieces of cloth. It is like cutting a circle in a stack (lay) of several pieces of paper at the same time.
- **P3**: Moving the packs of the parts of clothing to the assembly lines on conveyor belts (see *Chapter 10, Conceptual Representation Learning*).
- **P4**: Other operations depending on the products (packaging or printing or other).
- **P5**: Storing and optimizing the distribution process through warehouses and deliveries and many more processes (tracking and data analysis, for example, finding late deliveries and optimizing their routes).

The following diagram represents the production flow at an apparel manufacturing company. First, the fabric is cut, and then it is stacked in piles and sent by a conveyor belt to sewing stations to assemble the clothing:

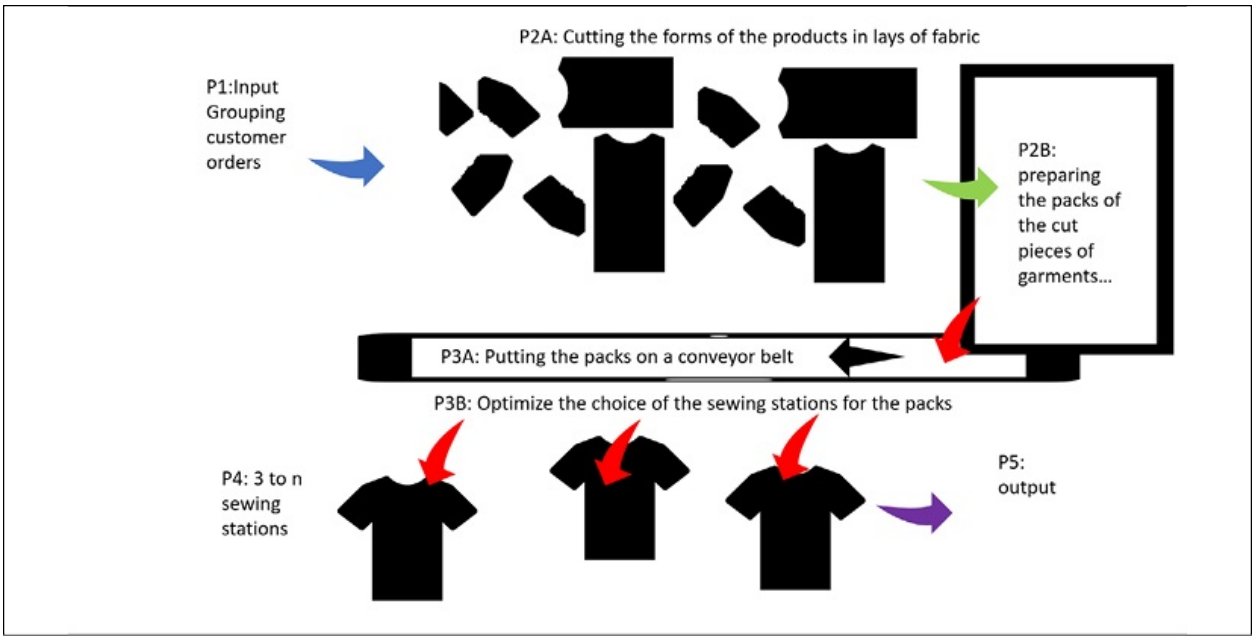


Figure 11.1: Apparel production flow

A webcam is set up right over P3, the conveyor belt. The following image represents the webcam above the conveyor belt:

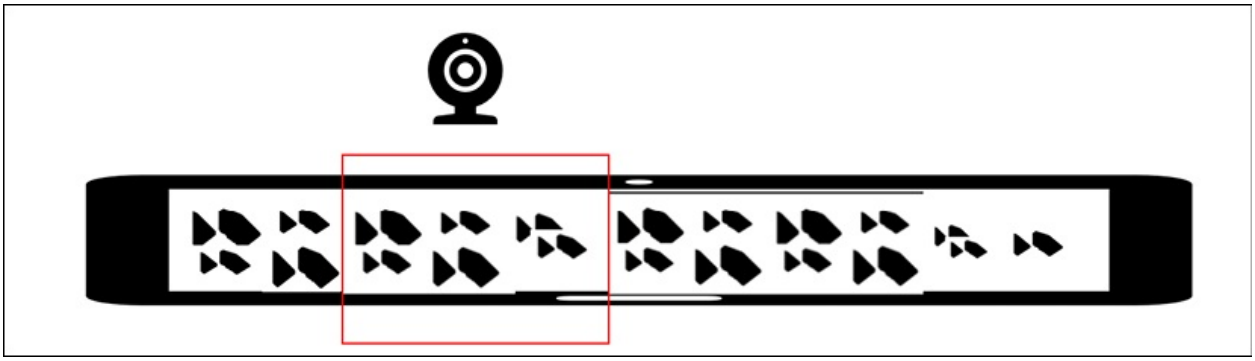


Figure 11.2: Webcam freezing frames on a conveyor belt

The webcam freezes a frame (within the red rectangle) every *n* seconds. The frame is the input of the CRL-CNN described later.

This image is a representation of the concepts. In reality, the webcam may be located at the beginning of the conveyor belt, or even over the output of the cutting process. For the prototype in this chapter, keep in mind that every *n* seconds, a frozen frame is sent to the trained CNN.

This P1 to P5 flowchart provides a general idea of an apparel manufacturing process. In an actual company, many more processes are required: market studies, designing products, testing prototypes, adding manufacturing processes for jeans (making holes with a laser, for example), and much more.

Every process described so far was invented individually 30+ years ago, including automatic cutting and conveyor belts applied to the apparel business. *What's new here?* springs to mind in the case of many apparel experts reading about Amazon. And that is the mind trap! Thinking that Amazon's apparel patent does not contain new components is a mistake.

Time squashing is the core of the Amazon innovation process. Bringing manufacturing closer to the consumer in near-real time is a revolution that will become disruptive. Add 3D printers to the time squashing equation, and you will easily picture the future of our consumer markets. However, artificial intelligence has the right to enter the optimizing competition as well.

To illustrate this, let's build an AI model that optimizes P3, the conveyor belt. Many solutions already exist as well, but an RL+DL can most probably beat them just as it will in many fields.

The first step is to generalize the Γ model described in *Chapter 10, Conceptual Representation Learning*, a bit further through training. Then, the RL-DL-CRLMM can be built. We will first explore how the CRLMM is trained to analyze web frames in production.

Training the CRLMM

Chapter 10, Conceptual Representation Learning, introduced CRLMMs using Γ (gap concepts) to illustrate one.

In the previous chapters, conceptual subsets around Γ (gap concepts) were designed as follows:

- $\Gamma = \{g_1, g_2, g_3, g_4, g_5, g_6\}$, which contains pg_i (p = positive) and ng_i (n = negative subsets)
- ng_1 is a subset of Γ; ng_1 = {missing, not enough, slowing production down … bad},
- pg_2 is a subset of Γ; pg_2 = pg_2 = {good production flow, no gap}
- g_2 = {loaded}
- g_3 = {unloaded},
- pg_4 = {traffic jam, heavy traffic … too much traffic}
- ng_5 = {open lane, light traffic … normal traffic}

`CNN_STRATEGY_MODEL.py` needs to be trained to recognize Γ in an apparel production environment as well as remember how to recognize former Γ concepts:

- The cutting section P2 (A and B) output of apparel packs flowing on a conveyor belt (P3).
- Remember how to classify the cakes of the food processing company to teach the model to recognize more situations.
- Remember how to perform a traffic analysis (see *Chapter 10, Conceptual Representation Learning*).
- Learn how to be able to classify an abstract representation of a gap.

Generalizing the unit training dataset

To generalize the unit training dataset, six types of images were created. Each image represents a webcam frame taken over a conveyor belt every *n* seconds. Four images are figurative in the sense of figurative painting. Two images bring the CRLMM program to a higher level of abstraction.

Food conveyor belt processing – positive pγ and negative nγ gaps

In the food processing industry example (see *Chapter 9, Abstract Image Classification with Convolutional Neural Networks (CNNs)*, a gap on the conveyor belt was most often negative, a *negative gamma* = n^{γ}.

The following frame shows that the first line of production is complete, but not lines two and three:

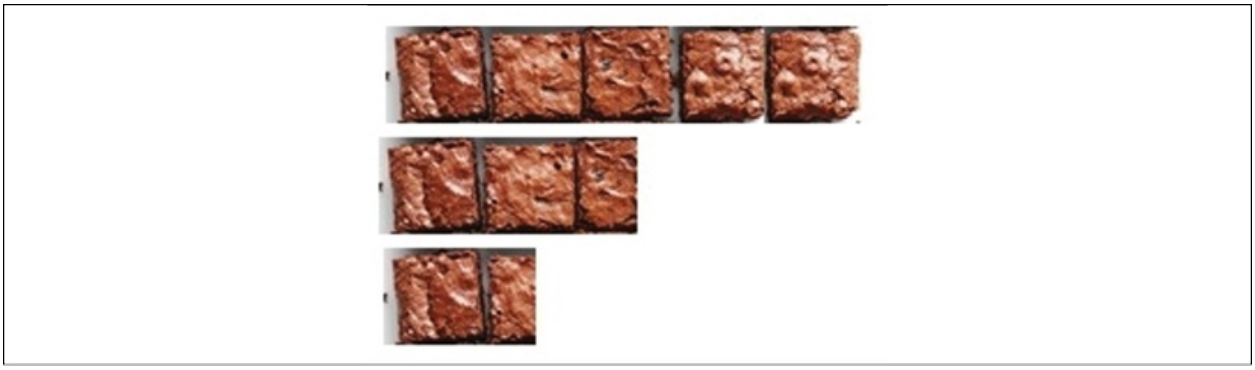


Figure 11.3: Food conveyor belt processing frame

On the contrary, with approximately no gaps, the load on the conveyor belt was viewed as positive, a *positive gamma* = p^{γ}.

The following frame shows that there is an acceptable number of products per line:

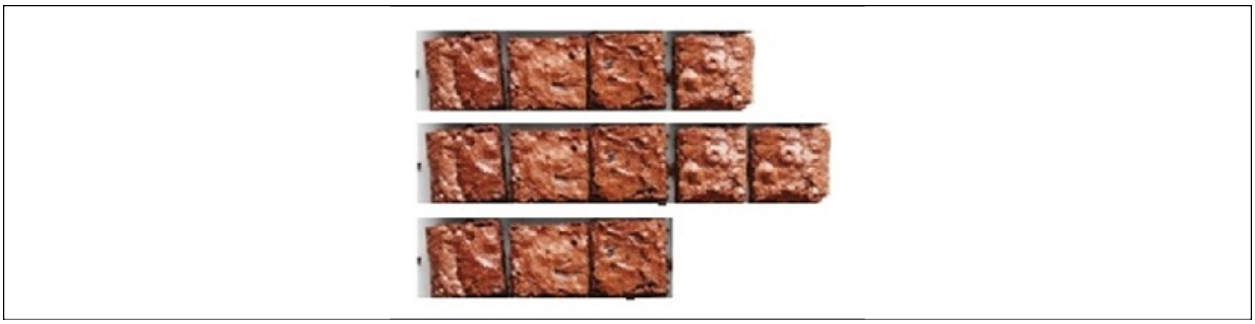


Figure 11.4: Food conveyor belt processing frame

As the preceding images show, whatever the product is on a conveyor belt, a gap remains a gap, an empty space. We will now apply our model to other gaps whatever the object may be. In this case, we will detect gaps on an apparel conveyor belt.

Apparel conveyor belt processing – undetermined gaps

A gap in an apparel conveyor belt process is most often undetermined. This constitutes a major optimization problem in itself. Naturally, if the conveyor belt is empty or saturated, the variance will attract the attention of a human operator. However, most of the time, optimization solves the problem.

The following frame shows a relatively well-loaded production flow of apparel packs of pieces of clothing to be assembled (by sewing them together) further down the line:

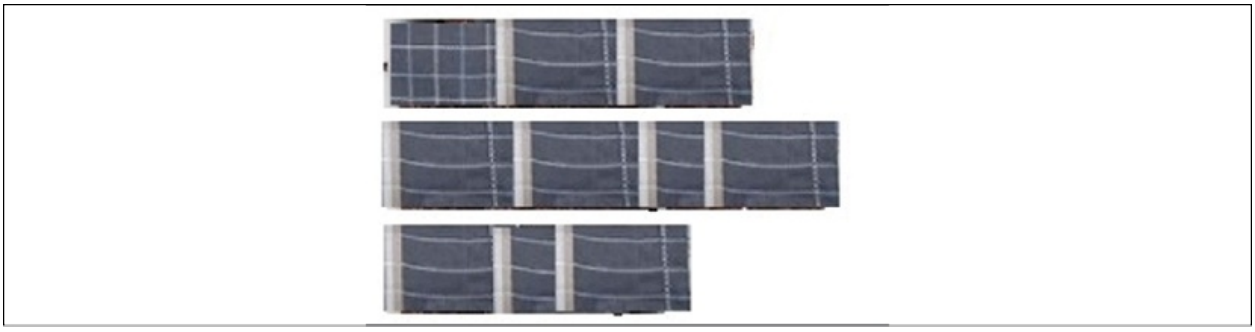


Figure 11.5: Well-loaded production flow

The following frame clearly shows a gap, meaning that the quantity that will be sent to a sewing station will not be high:

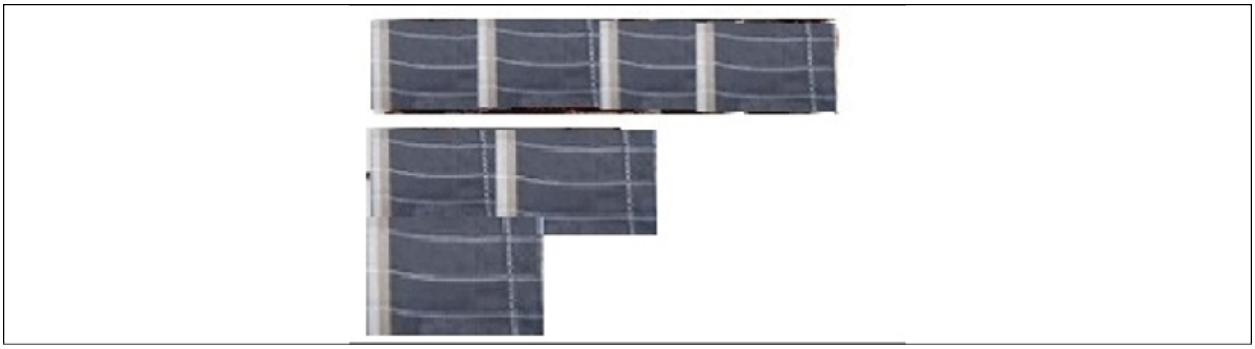


Figure 11.6: Production flow (gap)

Several observations are necessary to optimize this problem:

- The conveyor belt real-time problem excludes running advanced planning and scheduling.
- This problem requires real-time automated planning and scheduling.
- The automated planning and scheduling solution will have to both plan and schedule in real-time.
- It will take planning constraints into account (as explained in the following sections) to predict outputs.
- It will take scheduling constraints into account to optimize the sewing sections.

Amazon and others have slowly, but surely, brought many planning horizon problems (longer) down to shorter scheduling horizons, pushing the limits of SCM further and further.

The beginning of an abstract notion of gaps

The gaps shown have negative or positive properties depending on the context. The CRLMM model now has to learn a meta conceptual abstract representation of all the gaps mentioned hitherto. These gaps are all flow gaps of one sort or another. Something is moving from one point to another in small packs. The packs, therefore, are often not the same size, and this causes gaps to form.

These concepts can be applied to cattle herds, horse races, team sport attacks (football, soccer, rugby, basketball, handball, and other fields), Olympic races, marathons, conveyor belts, and much more.

When the packs in the flow are close to each other, an individual mental image crops up. Each person has a customized version. The following image is a generalized representation of a no-gap concept:

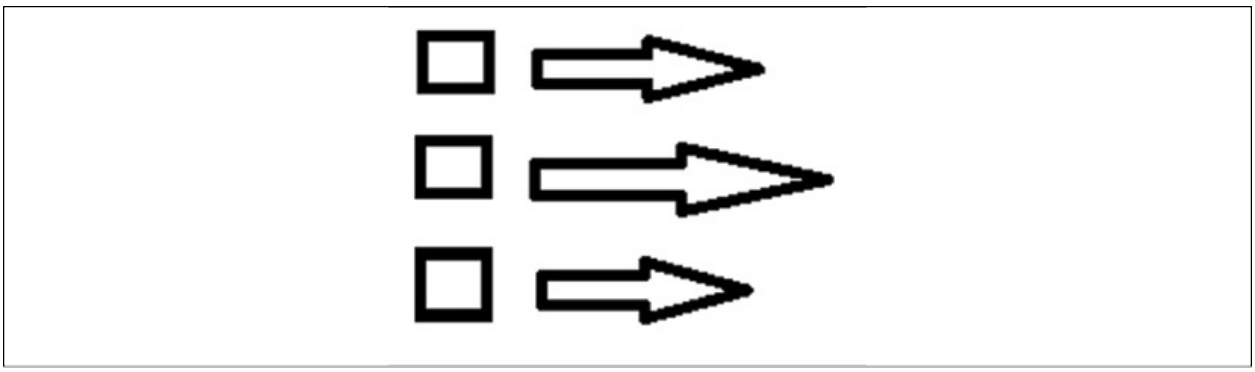


Figure 11.7: A generalized representation of a no-gap concept

Then, the packs have leaders and followers. Then, an abstract representation comes up as well. The following image shows a generalized representation of a gap concept:

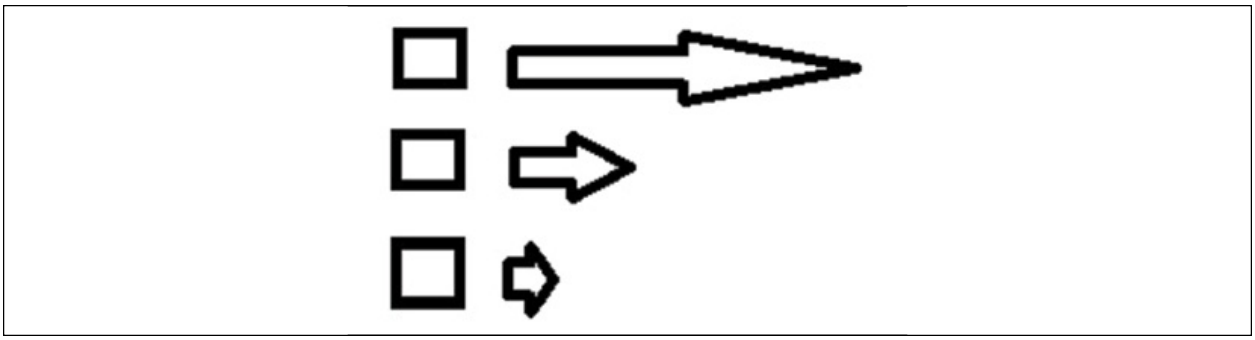


Figure 11.8: A generalized representation of a gap concept

Humans do not have a flow detection gap function for each thing they are observing. Humans have one brain that contains physical memories or other datasets, but, more importantly, they have abstract datasets.

Every single one of us billions of thinking bipeds has extremely efficient abstract datasets. The meta-concept shown previously means that through inferences, a human has a central meta-concept with a dataset of memories that fit with it through experience.

A meta-model uses these datasets. The dataset in the `dataset` directory contains the beginning of a CRLMM system. The program will learn what a *flow gap* is

and then apply it to what it sees by analyzing the context.

This goal of this dataset leads to a CRLMM, as explained in the following sections in which:

- The abstract learned meta-concept is applied to a situation; in this case, a frame.
- The CRLMM then determines whether it is a gap or no-gap situation.
- The CRLMM then makes a decision, using a thinking optimizer based on a decision weight-orientated activation function. This means that it is more than a mathematical *squash*. It's *weighing* the pros and cons of a decision-making process.

In this section, our CRLMM has learned to identify gaps on an apparel conveyor belt. If we were to implement this, we would add many web frames of a conveyor belt of our project to our datasets to do the training. For now, let us continue with this example and run a prediction program to classify the gaps.

Running a prediction program

The training was done with the same `CNN_STRATEGY_MODEL.py` program described in *Chapter 9*, *Abstract Image Classification with Convolutional Neural Networks (CNNs)*, which was designed to be generalized for subsequent chapters.

Once the dataset of the previous section has been installed, the model can make predictions with no further training in various domains.

The same `CNN_CONCEPT_STRATEGY.py` functions described in *Chapter 10*, *Conceptual Representation Learning*, were implemented.

Only the path to the dataset was changed, along with the prediction messages to display:

```
#II. Convolutional Neural Network (CNN)
#loads,traffic,food processing
A=['dataset_O/','dataset_traffic/','dataset/']
MS1=['loaded','jammed','productive']
MS2=['unloaded','change','gap']
```

```
display=1     #display images
scenario=2    #reference to A,MS1,MS2
directory=A[scenario] #transfer learning parameter (choice
of images)
CRLMN=1       # concept learning
print("Classifier frame directory",directory)
```

The CRLMM has now learned to represent real-life memories and inference of the associated memories in a meta-concept. Our CRLMM is ready to be assembled as one of the three components of RL-DL-CRLMM as we will see in the next section.

Building the RL-DL-CRLMM

The full code of the RL-DL-CRLMM program is `RL_DL.py`. It is built on the knowledge and programs of the previous chapters and previous sections of this chapter.

The RL-DL-CRLMM contains three components:

- A CRLMM convolutional network that will analyze each frame it receives from the webcam that is located right over the pieces of garment packs on the conveyor belt coming from the cutting section.
- An optimizer using a modified version of the $Z(X)$ described previously that plans how the assembly stations will be loaded in real-time.
- An MDP that will receive the input of the optimizer function and schedule the work of the assembly stations. It also produces the modified $Z(X)$ updated value of the weights of each assembly station for the next frame.

In the physical world, the conveyor belt transports the garment packs, a picture (frame) is taken every *n* seconds, and the RL-DL-CRLMM runs. The output of the RL-DL-CRLMM sends instructions to the conveyor belt and directs the garment packs to the optimized load of the assembly stations, as explained before.

Since `RL_DL.py` describes a continuous RL-DL-CRLMM process, there is no

> beginning or end to the program. The components are independent and are triggered by the input sent to them, and they trigger others with their output.

Thus, components will be described in the processing order but not in the code-in-line order. This is because the functions defined by `def` + `function()` precede the function calls to them. Code line numbers will thus be inserted in the comments following the code line like this: `# [Line 38]`.

A circular process

Once the three main components of the system are in place (CNN, MDP, optimizer), the circular property of this RL-DL-CRLMM is a stream-like system that never starts nor ends.

To define a circular process, let's take an everyday example. Customer C goes to the supermarket to purchase a product. The product was previously stored in a warehouse that we will name B. A factory named A manufactures this product. If you look at the whole supply chain at a given time, you will see there is no starting or ending point:

- If you are A, you are both monitoring the needs of C to send products to B
- If you are B, you are both monitoring what is coming from A to satisfy C
- If you are C, you are creating a demand for A to deliver B

In a real-life supply chain, there is no beginning and no end. The same applies to an automated production site to the flow of functions in this virtually memoryless system. The conveyor belt's webcam provides a stream of frames, forcing the system into circular stream-like behavior.

Implementing a CNN-CRLMM to detect gaps and optimize

The CNN-CRLMM function was described in the *Running a prediction program* section and in *Chapter 10, Conceptual Representation Learning*. The prediction function is part of the MDP output analysis described later. The `CRLMM` function

is called as shown in the following code:

```
def CRLMM(Q,lr,e):    # [Line 180]
```

The first part of this function squashes the *W* vector described in the following section. The second part of this function analyzes the input frame.

Since no webcam is connected to the system at this point (that must be done during project implementation), a random image (frame) choice is made. The following random code simulates the random occurrences of real-life production:

```
    status=random.randint(0,1)
```

In a real-life situation, random quantities will be going through the conveyor belt. With the status having being determined, the CNN simulates the addition of a frozen frame from the video stream coming from the webcam located directly on the conveyor belt. It runs the identity (image) function described before and returns a `gap` or `no gap` scenario for the optimizer (refer to the optimizer section). The following code describes a gap identification process:

```
    if(status==0):
        #Add frame from video stream (connect to webcam)
        s=identify(directory+'classify/img1.jpg',e)
    if(status==1):
        #Add frame from video stream (connect to webcam)
        s=identify(directory+'classify/img2.jpg',e)
    s1=int(s[0])
    if (int(s1)==0):
        print('Classified in class A')
        print(MS1[scenario])
        print('Seeking...')
    if (int(s1)==1):
        print('Classified in class B')
        print(MS2[scenario])
    return s1
```

Once `status` has been detected, irrespective of whether the load output of the cutting system is high or low, a decision must be made. This will be done with the MDP.

Q-learning – MDP

By adding an MDP Q-learning decision function (see *Chapter 1*, *Getting Started with Next-Generation Artifcial Intelligence through Reinforcement Learning*) in this CNN-CRLMM program, we are now entering the world of cognitive RL-DL programs.

> In this real-time system, there is no beginning, no end, and no real memory beyond a few frames.

The `mdp01.py` MDP program has been incorporated into `RL_DL.py`. Thus, only the changes made are described in this chapter.

MDP parameters come right after the import packages. Each vertex of the graph has its letter and location, as shown in the following code snippet:

```
L=['A','B','C','D','E','F']    # [Line 37]
```

When the program runs, the following graph will be displayed with red (target vertices) for this frame. It is then up to the MDP to choose, described as follows:

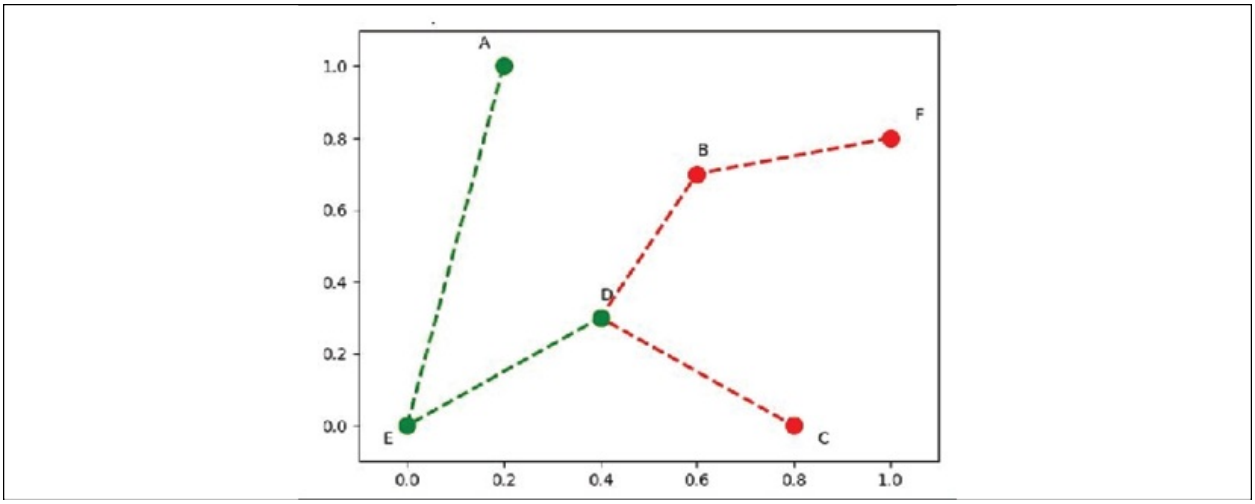


Figure 11.9: Output (target vertices)

Notice that this is an undirected graph with vertices (the colored dots) and no edge (lines) directions. The MDP will provide directions for each frame in this

model.

Each vertex represents an assembly station in this model. Each assembly station has a high workload or a low workload. The workload is rarely stabilized because products keep flowing in and out as they are produced.

MDP inputs and outputs

The MDP process in this model uses a reward matrix as an input and produces a weight vector.

The input is a neutral reward matrix

The MDP reward matrix is set to `0`, except for the values representing the edges of the graph that can be physically accessed; these are set to `1`, a small neutral value.

As it is, the MDP cannot provide a satisfactory result beyond reproducing the structure of this undirected graph. The reward matrix is now initialized and duplicated as shown in the following code, starting from line 41; at line 43, `R`, the reward matrix is built, and at `Ri`, a duplicate of `R` is created:

```
# R is The Reward Matrix for each state built on the physic
al graph    [Line 41]
# Ri is a memory of this initial state: no rewards and undi
rected
R = ql.matrix([[0,0,0,0,1,0],
               [0,0,0,1,0,1],
               [0,0,0,1,0,0],
               [0,1,1,0,1,0],
               [1,0,0,1,0,0],
               [0,1,0,0,0,0]])
Ri = ql.matrix([[0,0,0,0,1,0],
                [0,0,0,1,0,1],
                [0,0,0,1,0,0],
                [0,1,1,0,1,0],
                [1,0,0,1,0,0],
                [0,1,0,0,0,0]])
```

`Ri` is a copy of the initial-state zero-reward matrix. `R` is reset to `Ri` at every new frame. MDP trains the assembly location plan for each new frame in a

memoryless, undirected graph and unsupervised approach.

`Q` is the learning matrix in which rewards will be learned/stored, as shown in the code on line 58:

```
Q = ql.matrix(ql.zeros([6,6]))    # [Line 58]
```

The standard output of the MDP function

The load of the assembly stations is not the exact quantity produced. They are weights that are updated continuously during this continuous process, as explained further in the chapter.

At an initial state of the program, the initial weight of the vertices is set to `0` in the following line of code (line 40):

```
W=[0,0,0,0,0,0]    # [Line 40]
```

The weight vector (represented as an array) contains one value per assembly station location or vertex.

An initial state is not a real beginning. It is like a pause button on a video. The initial state can come from a holiday (no production), a maintenance day (no production), or a lack of input orders (no production). The initial state is just a case when the weights are equal to `0` because there is nothing on the conveyor or assembly stations.

The weight vector is part 1 of the optimizer (see the following section). The MDP produces an output matrix after having optimized the undirected graph. The optimizer will have provided a target.

The program contains the same source code as in *Chapter 1*, *Getting Started with Next-Generation Artifcial Intelligence through Reinforcement Learning*, with Bellman's equation starting from line 1166. The MDP produces its result, as shown in the following output:

```
[[ 0. 0. 0. 0. 105.352 0. ]
[ 0. 0. 0. 130.44 0. 201. ]
[ 0. 0. 0. 130.44 0. 0. ]
```

```
[ 0. 161.8 105.352 0. 105.352 0. ]
[ 85.2816 0. 0. 130.44 0. 0. ]
[ 0. 0. 0. 0. 0. 250. ]]
Normed Q :
[[ 0. 0. 0. 0. 42.1408 0. ]
[ 0. 0. 0. 52.176 0. 80.4 ]
[ 0. 0. 0. 52.176 0. 0. ]
[ 0. 64.72 42.1408 0. 42.1408 0. ]
[ 34.11264 0. 0. 52.176 0. 0. ]
[ 0. 0. 0. 0. 0. 100. ]]
```

Bear in mind that this entire DQN-CRLMM is not only based on the undirected memoryless MDP function but also has no real beginning nor end since it's a continuous and virtually memoryless process.

A graph interpretation of the MDP output matrix

After each run, the MDP matrix also produces a graph interpretation of the values of paths between the vertices from point to point (letter to letter), and displays it in the following output:

```
State of frame : 3 D
0 A E 161.8
1 B D 201.0
2 C D 201.0
3 D D 250.0
4 E A 130.44
4 E D 201.0
5 F B 161.8
```

This way, if the values are taken from the highest to the lowest edges (lines and, hence, values between two letters), it gives a visual idea of how the MDP function calculated its way through the graph.

`RL` is the letter vector. It is empty after each frame. It will be filled by finding the values of each edge. It will contain the letters of the vertices (nodes and dots) connected by the edges (lines represented by the values).

`RN` is the value of the edges. The following code shows how to implement `RL` and `RN` and update the weights of the locations in the weight vector (`W`):

```
    #Graph structure    [Line 187]
    RL=['','','','','','']
    RN=[0,0,0,0,0,0]
    print("State of frame :",lr,L[lr])
    for i in range(6):
        maxw=0
        for j in range(6):
            W[j]+=logistic_sigmoid(Q[i,j])
            if(Q[i,j]>maxw):
                RL[i]=L[j]
                RN[i]=Q[i,j]
                maxw=Q[i,j]
                print(i,L[i],RL[i],RN[i])
```

The logistic function in the preceding code is being updated while `RL` and `RN` are being updated.

The optimizer

I have written several optimizers for fabric optimization in the apparel industry. In this case, the optimizer will be used to regulate the flow of production.

The term "optimizer" is not the CNN `rmsprop` optimizer used in the previous chapters represented with the following code:

```
loaded_model.compile(loss='binary_crossentropy', optimizer=
'rmsprop', metrics=['accuracy'])    # [Line 92]
```

The term "optimizer" refers to a function that optimizes the production of this manufacturing site. It is not an optimizer used to train a CNN, for example. This optimizer is both an activation function and a regulator that was built from scratch to optimize production. This shows that you must sometimes invent the optimizer you need to generate a profitable solution for your customer or company.

The optimizer as a regulator

The whole concept of this RL-DL-CRLMM of real-time production, applied to P3, is to optimize *Z* over the load distribution of the assembly stations, as

explained before. This means reducing *Z* as much as possible through the following equation:

$$Z = \sum_{t_{xn}}^{t_{xn}} \frac{1}{1 + e^{-t_x}} \lambda$$

To achieve this optimization goal, *Z* needs to be taken apart and applied to strategic parts of the code.

Implementing Z – squashing the MDP result matrix

The output of the MDP functions provides the following `Q` matrix:

```
Q :
[[ 0. 0. 0. 0. 321.8 0. ]
 [ 0. 0. 0. 401. 0. 258.44]
 [ 0. 0. 0. 401. 0. 0. ]
 [ 0. 0. 0. 500. 0. 0. ]
 [ 258.44 0. 0. 401. 0. 0. ]
 [ 0. 321.8 0. 0. 0. 0. ]]
```

Each line represents a vertex in the graph: A, B, C, D, E, and F. Each value obtained needs to be squashed in the following *z*(*x*) function:

$$z(x) = \frac{1}{1 + e^{-t_x}}$$

The first step in the code that follows is to squash the weights provided by the MDP process for each line (vertex) *x* with a `logistic_sigmoid` function:

```
#Logistic Sigmoid function to squash the weights    [Line 1
18]
def logistic_sigmoid(w):
    return 1 / (1 + math.exp(-w))
```

The function is called by the transformation of the MDP `Q` output matrix into the weight vector for each column of each line, as shown in the following code:

```
    for i in range(6):    # [Line 191]
        maxw=0
        for j in range(6):
            W[j]+=logistic_sigmoid(Q[i,j])
```

At this point, each value of the MDP has lost any real-life value. It is a weight, just as in any other network. The difference is that the whole system is controlled. In a real-life project, keeping an eye on the calculations through reports is necessary for maintenance purposes. Even an automatic system requires quality control.

The MDP matrix has now been flattened into a weight matrix, as shown in the following output:

```
Vertex Weights [3.5, 3.5, 3.0, 5.0, 3.5, 3.5]
```

Each vertex (letter in the graph) now has a weight.

Implementing Z – squashing the vertex weights vector

Squashed `W` (vertex weights) grows after each frame analysis, and each MDP run since `W[j]+` is applied continuously, and `W` is never set to zero.

The main reasons:

- Once launched, the RL-DL-CRLMM is a continuous process, with no beginning and no end as long as the conveyor belt is running.
- The conveyor belt is sending assembly (mostly sewing) work to the assembly stations that take some time (*t*) to get the job (*x*) represented by t_x in the *Z* equation and the *W* vector in the program.
- Hence, work is piling up on each vertex (A to F), which represents an assembly station.

This is why the λ variable in the *Z* equation is implemented, as shown in the initial equation early in the chapter, as follows:

$$Z = \sum_{t_{xn}}^{t_{xn}} \frac{1}{1 + e^{-t_x}} \lambda$$

λ is implemented for two reasons:

- The sewing or assembly stations send their finished work to the next operation on the production line, packaging, for example. So every *m* minutes, their workload goes down, and so does the load feature weight.
- Production managers are constantly working on learning curves on assembly lines. When a new product arrives, it takes some time for the teams to adapt. Their output is a bit slower than usual. However, well-trained teams bring the learning period down regularly.

λ combines both concepts in a single variable. This might be enough for some projects. If not, more work and variables need to be added.

In this model, λ is activated by the following:

- `oif` represents the frequency of a `W` vector λ update. In this example, `oif` is set to `10`. This means that every 10 frames, `oir` will be applied.
- `oir` represents the output rate for the two reasons described before. This variable will squash the `W` vector by the % given. In this example, `oir=0.2`. That means that only 20% of the weights will be retained. The rest has been finished.

The following code shows how to implement `oif` and `oir`:

```
# input_output_frequency : output every n frames/ retained
memory    [Line 315]
oif=10
#input_output_rate p% (memory retained)
oir=0.2
fc=0 #frequency counter : memory output
for e in range(episodes):
    print("episode:frame #",e)
    fc=fc+1
    #memory management : lambda output
    if(fc>=10):
        for fci in range(6):
```

```
            W[fci]=W[fci]*oir
            fc=0
            print("OUTPUT OPERATION - MEMORY UPDATED FOR ",
L[fci],
                  " ",oir,"% retained")
```

The `W` vector has been squashed again, as shown in this output:

```
OUTPUT OPERATION - MEMORY UPDATED FOR A 0.2 % retained
OUTPUT OPERATION - MEMORY UPDATED FOR B 0.2 % retained
OUTPUT OPERATION - MEMORY UPDATED FOR C 0.2 % retained
OUTPUT OPERATION - MEMORY UPDATED FOR D 0.2 % retained
OUTPUT OPERATION - MEMORY UPDATED FOR E 0.2 % retained
OUTPUT OPERATION - MEMORY UPDATED FOR F 0.2 % retained
```

The optimizer has provided updated information on the status of each sewing station. Each sewing station is a location for the MDP function. We will now see how the MDP function uses the information provided by the optimizer.

Finding the main target for the MDP function

W, the weight vector, is updated after each frame with a short-term memory of *n* frames. This means that in every *n* frames, its memory is emptied of useless information.

The goal of the optimizer is to provide a target for the MDP function. On the first episode, since it does not have any information, the optimizer selects a random state, as shown in the following code excerpt:

```
    #first episode is random
    if(e==0):
        lr=random.randint(0,5)
```

This means that a random assembly station will be chosen on the MDP graph, which represents the six assembly sewing stations. Once this episode has been completed, the system enters a circular real-time cycle (refer to the next section).

The second episode has the `W` vector to rely upon.

This runs the `crlmm` (described before) CNN-CRLMM network to determine

whether the frame has a gap or no-gap feature, as shown in the following code:

```
    crlmm=CRLMM(Q,lr,e)    # [Line 388]
```

The optimizer will use `W` to:

- Choose the vertex (sewing station) with *somewhat* the smallest weight if the CNN on the frame produces a result with `no gap` (probability is zero). Since there is no gap, this means that there are many pieces to sew. Thus, it is much better to give the work to an assembly station that has a lower load than others.
- Choose the vertex (sewing station) with *somewhat* the highest weight if the CNN on the frame produces a result with a `gap` (probability is one). Since there is a gap, it means there are not that many pieces to sew. Thus, it is much better to give the work to an assembly station that already has a higher workload. It will balance the loads and optimize the load distribution over all the stations.
- Introduce a choice to find one target assembly station for the MDP function in each of these cases. It will look for stations (vertices, letters, dots in the graph) with the highest weight in one case and the lowest in another.
- Add the *somewhat* concept referred to before. The system must remain relatively free, or it will keep choosing the same best sewing stations, depriving others of work. Thus, each possibility (`gap` or `no gap`) is limited to a random choice of only three among the six locations for each weight class (higher or lower).

The optimizing function, shown in the following snippet, can prove extremely profitable on real-life industrial production lines:

```
    if(e>0):
        lr=0
        minw=10000000
        maxw=-1
        #no G => Finding the largest gap (most loaded resou
rce or a distance)
        if(crlmm==0):
            for wi in range(3):
                op=random.randint(0,5)
                if(W[op]<minw):
```

```
                    lr=op;minw=W[op]
        #G => Finding the smallest gap (a least loaded reso
urce or a distance)
        if(crlmm==1):
            for wi in range(3):
                op=random.randint(0,5)
                if(W[op]>maxw):
                    lr=op;maxw=W[op]
```

`lr` is the main location chosen for the MDP reinforcement learning function, as shown in the following code:

```
    print("LR TARGET STATE MDP number and letter:",lr,L[lr]
)
```

Reinforcement learning has to run every time since it faces new frames in the continuous process of the conveyor belt in the automated apparel system.

Now, the MDP reward matrix is reset to its initial state, as implemented in these lines of the program:

```
    #initial reward matrix set again
    for ei in range(6):
        for ej in range(6):
            Q[ei,ej]=0
            if(ei !=lr):
                R[ei,ej]=Ri[ei,ej]
            if(ei ==lr):
                R[ei,ej]=0 #to target, not from
```

The target location is initialized with a value that fits the gap or no-gap philosophy of the program. If there is no gap, the value is higher, and so will be the weight (represented load of the station) in the end. If there is a gap, the value is lower, and so will be the weight for that station, as shown in the following lines of code:

```
    #no G
    rew=100
    #G
    if(crlmm==1):
        rew=50
    R[lr,lr]=rew
```

```
print("Initial Reward matrix with vertex locations:",R)
```

We just explored the MDP process. However, the overall program we are looking at is a circular process that runs 24/7. At a given time, all of the components of our AI program are running at the same time. Let's see how.

A circular model – a stream-like system that never starts nor ends

We have successfully converted the linear stream of production (conveyor belt, optimizer, and sewing stations) into a circular process.

Since the RL-DL-CRLMM model runs nonstop, it can run at all of the following points at the same time:

- A = Captures a frame of the output of the cutting section with a webcam and a CNN-CRLMM
- B = Analyzes the load (histograms of each sewing section) with the CRLMM
- C = Optimizes the sewing stations with an MDP

ABC is one flow. But since each process is running nonstop, we can see all of the following flows at the same time: A, B, C or B, C, A or C, A, B.

The following graph represents the circular process:

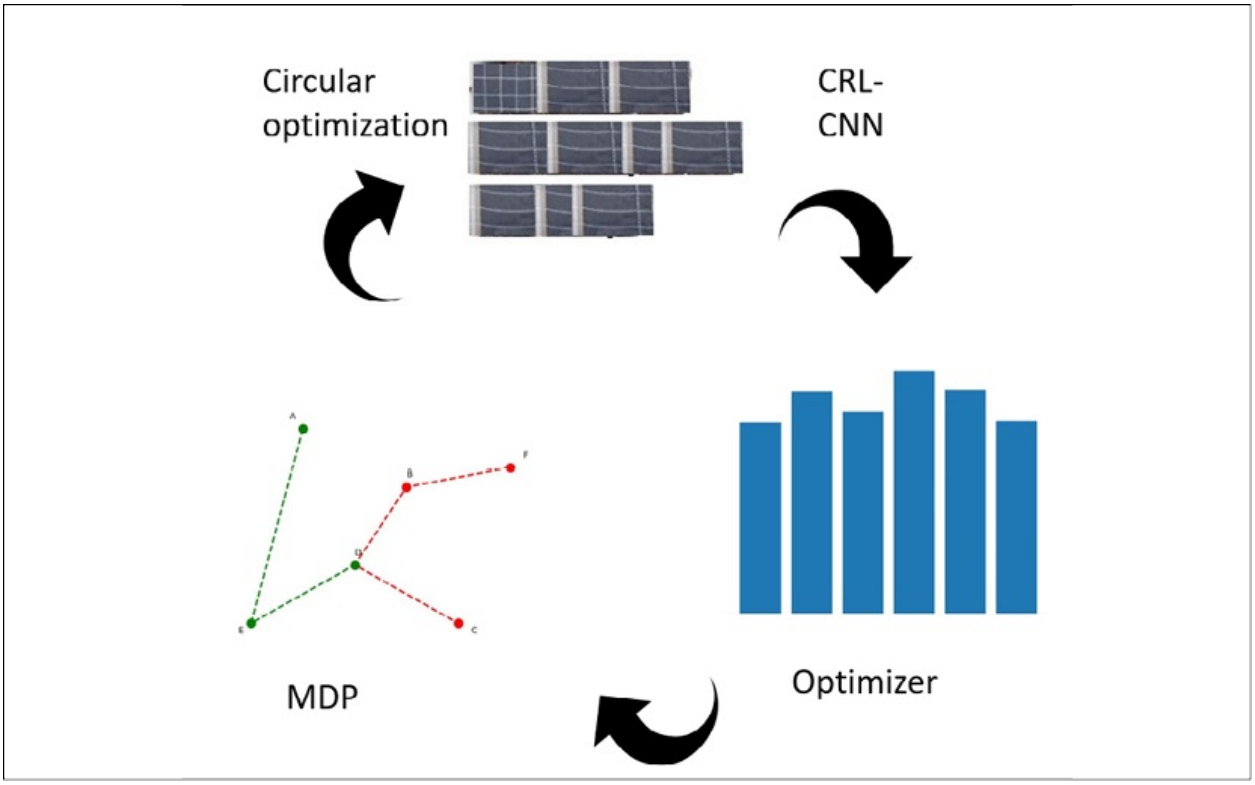


Figure 11.10: Circular RL-DL-CRLMM

The diagram describes an automatic process. However, let's go back a step and imagine we are in a factory in which humans still make the decisions. The process is circular, which means it never stops or starts. It runs 24/7. We will suppose a floor manager is an optimizer represented by the blue bar charts. The floor manager will look at the sewing stations to see who has a lot of work to do and who has little work to do. Then the floor manager will observe the conveyor belt (the web frame with the image of the packs) that is bringing packs of cut cloth to sew into garments. The floor manager will select several packs where there is a gap on the conveyor belt for the least loaded sewing stations, for example. Once the choice has been made, the floor manager will direct the packs to the right sewing station following calculating the graph (MDP) manually.

Let's explore these steps in further detail.

Step X: When the following frame arrives, the CRL-CNN is activated:

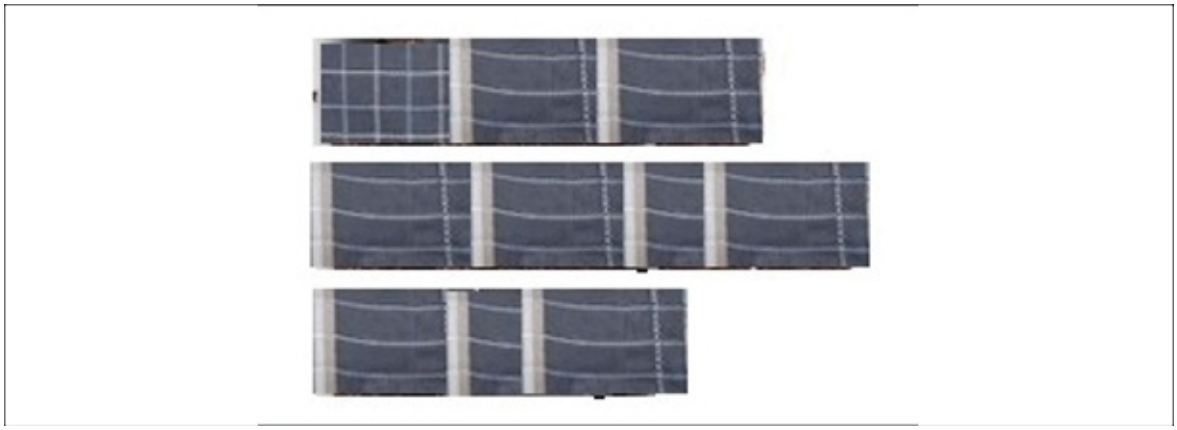


Figure 11.11: The CRL-CNN gets activated

The solution calls the `CRLMM` function, as shown in the following code:

```
    crlmm=CRLMM(Q,lr,e)    # [Line 388]
    print("GAP =0 or GAP =1 status: ",crlmm)
```

The following output displays the status of the image: `gap`, `no gap`:

```
image dataset/classify/img1.jpg predict_probability: [[ 0.]]
 prediction: [[ 0.]]
Classified in class A
Productive
Seeking...
GAP =0 or GAP =1 status: 0
```

The following weights of each sewing station (vertices A to F) are taken into account:

```
    MDP_GRAPH(lr,e)    # [Line 390]
    print("Vertex Weights",W)
```

The program displays them in text form:

```
Vertex Weights [9.4, 11.100000000000001, 10.2, 12.0, 11.7,
10.0]
```

The program also displays the following bar chart of the weights of each sewing

station:

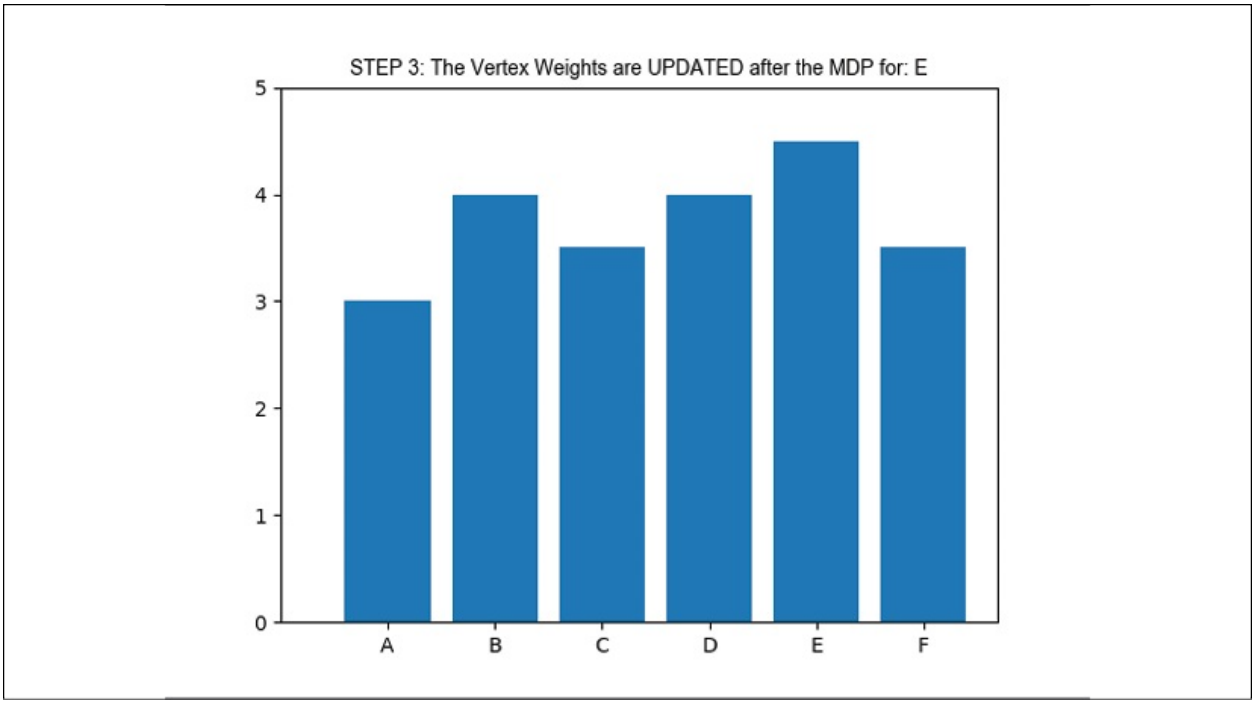


Figure 11.12: Bar chart of the weighs of each sewing station

The **STEP X+1** optimizer analyzes the weights to make a decision: send small quantities to the sewing stations that have a lot of work and large quantities to sewing stations that have less work to do.

By doing this, the optimizer makes sure that each sewing station has an optimal workload. If a sewing station has a lot of work piling up, it makes sense to give a large amount of incoming work to a sewing station with little work to do. That way, all of the sewing stations will be working at full capacity.

Γ (gamma) has now reached a point at which it understands that a gap is a comparison between two states and inference of a conceptual distance:

- **Overloading**: Too much
- **Underloading**: Not enough

Γ now contains two abstract concepts:

- Enough to too much
- Not enough to lacking

The goal of the circular process is to keep the bars at an approximately similar height—not an exact height, but also not a situation in which A would have no work to do, and E would be overloaded.

STEP X+2: The MDP, as described in this chapter, receives instructions to optimize a given sewing station and spreads work out to its neighbors. This is often a production constraint: one station sews the sleeves of a T-shirt. For example, the station nearby sews a pocket on the T-shirt. The MDP spreads out the work, starting with the target location. In this case, the target location is one of the sewing stations in the A-E-D area (the area is displayed in color when you run the Python program) as shown in the following graph:

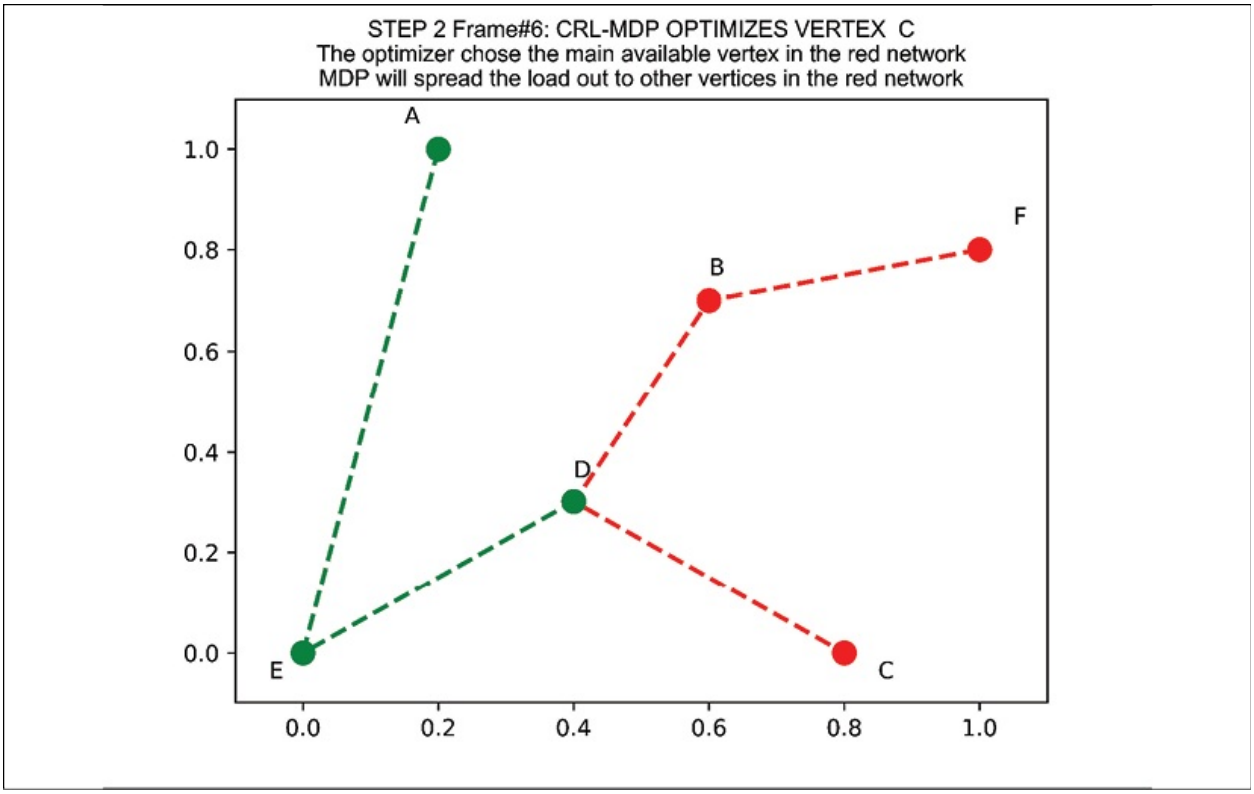


Figure 11.13: MDP spreads out

The MDP sends its results directly to the conveyor belt, which automatically follows the path instructions sent to it.

Then, the MDP updates the weights, empties its reward matrix, and waits for a new one. The system goes back to step X.

As you might have noticed, no human operator is involved in this entire process at all.

Summary

Applying artificial intelligence to Amazon's real-time sales, production, and delivery forces projects into reality.

Learning machine learning and deep learning with MNIST, CIFAR, and other ready-to-use datasets with ready-to-use programs is a prerequisite to mastering artificial intelligence. Learning mathematics is a must.

Building a few programs that can do various theoretical things cannot be avoided. However, managing a real project under corporate pressure will bring an AI specialist up to another level. The AI specialist will have to put AI theory into practice. The constraints of corporate specifications make machine learning projects exciting. During those projects, experts learn valuable information on how AI solutions work and can be improved.

This chapter described an RL-DL-CRLMM model with an optimizer. We learned how the market is evolving from planning manufacturing in advance to real-time planning challenging classical processes. We saw that a consumer wants to receive a purchased product as soon as possible. If that product is not available to demand, it must be produced in nearly real-time and delivered to the customer. To automate this process, we built a Python program that can scan an apparel conveyor built to detect gaps in a cutting process with a CNN, use an optimizing function to choose the best sewing workstation and then build a graph with MDP to represent the optimized path taken.

The next chapter explores an application of this solution with an SVM applied to a self-driving car situation in an IoT context, Swarm AI, and elementary AGI.

Questions

1. A CNN can be trained to understand an abstract concept? (Yes | No)
2. Is it better to avoid concepts and only use real-life images? (Yes | No)
3. Do planning and scheduling mean the same thing? (Yes | No)
4. Is Amazon's manufacturing patent a revolution? (Yes | No)
5. Learning how warehouses function is not useful. (Yes | No)
6. Online marketing does not require artificial intelligence. (Yes | No)

Further reading

- More information on Amazon's apparel manufacturing innovation and apparel market can be found here: https://www.nytimes.com/2017/04/30/technology/detailing-amazons-custom-clothing-patent.html, https://www.amazon.com/learn-more-prime-wardrobe/b?ie=UTF8&node=16122413011

12

AI and the Internet of Things (IoT)

Some people say the **Internet of things** (**IoT**) will turn out to become the fourth Industrial Revolution. Let's wait a few years until the smoke clears and then let historians figure out what sort of revolution we went through.

In any case, **connected objects** have been changing our lives for at least the past two decades. Given all that we have seen in recent years, we can safely say that IoT has become disruptive.

Artificial intelligence has *just* begun its long journey through human intellect. New, incredible innovations await us. Understand cutting-edge machine learning and deep learning theory is only the beginning of your adventure. Take everything you see seriously and see how it can be incorporated into your projects.

Your mind must remain open to accept the many innovations that are yet to come. For example, conceptual representation learning (see previous chapters) adds the power of human concepts to neural networks.

This chapter takes the technology of the previous chapter and applies it to the example of a self-driving car. The previous chapter used a webcam and a program and sent instructions to the conveyor belt. It was in the family of IoT. Let's add a **support vector machine** (**SVM**) to the program and take it out on the streets of a city to see what happens.

The chapter is divided into three main sections: a public service project, the configuration of the model, and running the model.

The following topics will be covered in this chapter:

- A self-driving solution
- Introducing a safe route parameter to trip planners

- Applying CNNs to parking lots
- Applying SVMs to safety on trip planning
- Teaching an MDP to find the safest route (not necessarily the shortest way)

Let's get started by outlining the problem and the goal of the project we'll be undertaking.

The public service project

The overall project in this example is to implement a self-driving, home-to-homeless-shelter delivery service:

- Families at homes have clothing and food they would like to give to others that need them.
- The self-driving car can be started at a distance and goes to homes and takes the goods to the shelters.
- The self-driving car does not need to have a base. It can park anywhere, go anywhere, and refuel at service stations with automatic electric recharging.

In this chapter, we will focus on the self-driving car when it has finished a delivery and is looking for a parking lot with a parking space. We will need the information to make decisions.

Some IoT projects plan to put sensors on every parking space and send the information to control centers. The city council finds that too expensive. Instead, the city council has decided to use a more cost-effective solution. A webcam will be installed on all the possible parking lots in the project's grid. This smart grid is for transporting products from homes to shelters.

We'll address this by first setting up an RL-DL-CRLMM model.

Setting up the RL-DL-CRLMM

model

This section describes how to set up the previous chapter's model for this project and add a few functions.

In *Chapter 11*, *Combining Reinforcement Learning and Deep Learning*, the RL-DL-CRLMM model analyzed webcam images of pieces of cut cloth to be sewed in real-time on a conveyor belt. The goal was to determine if they contained a gap (not too many pieces to sew) or not (a lot of pieces to sew). Then the model selected the best sewing station. A sewing station with a lot of work to do is best optimized with a small number of pieces to sew. A sewing station with little work to do will be best optimized with a large number of pieces to sew. By doing this, the RL-DL-CRLMM optimized the load on each sewing station, as shown in the following diagram:

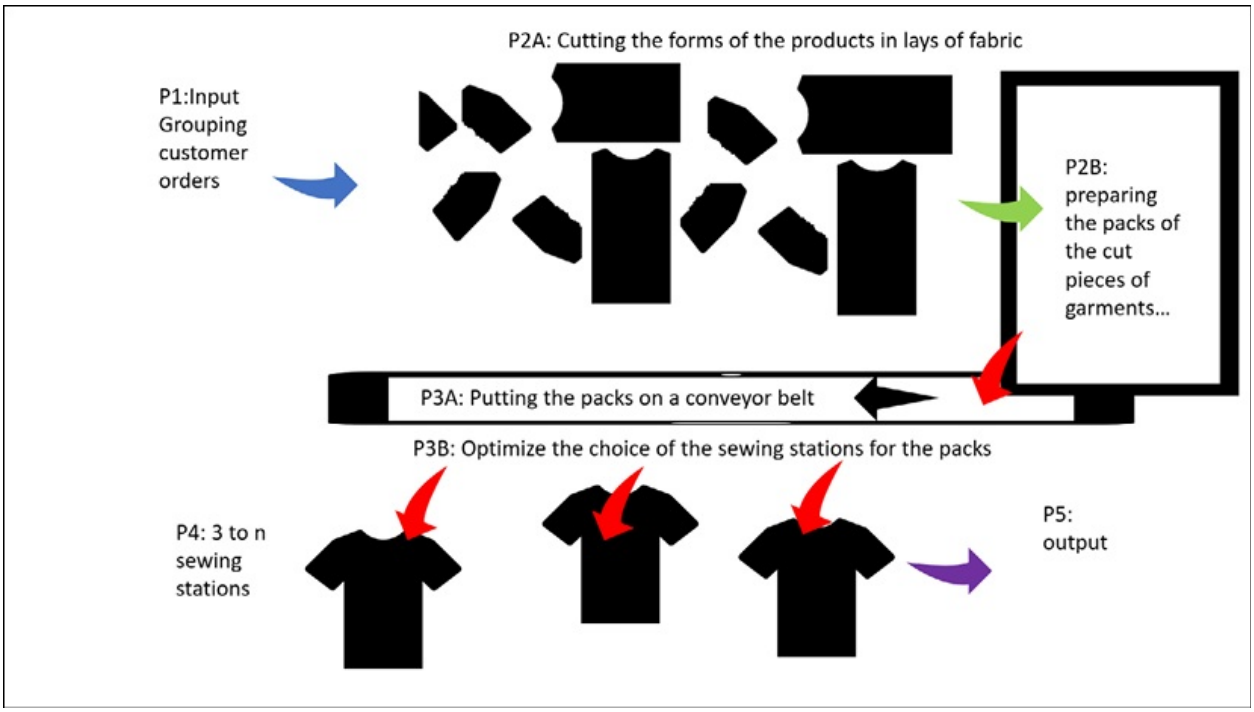


Figure 12.1: Apparel production flow

This leads to the following circular optimizing model:

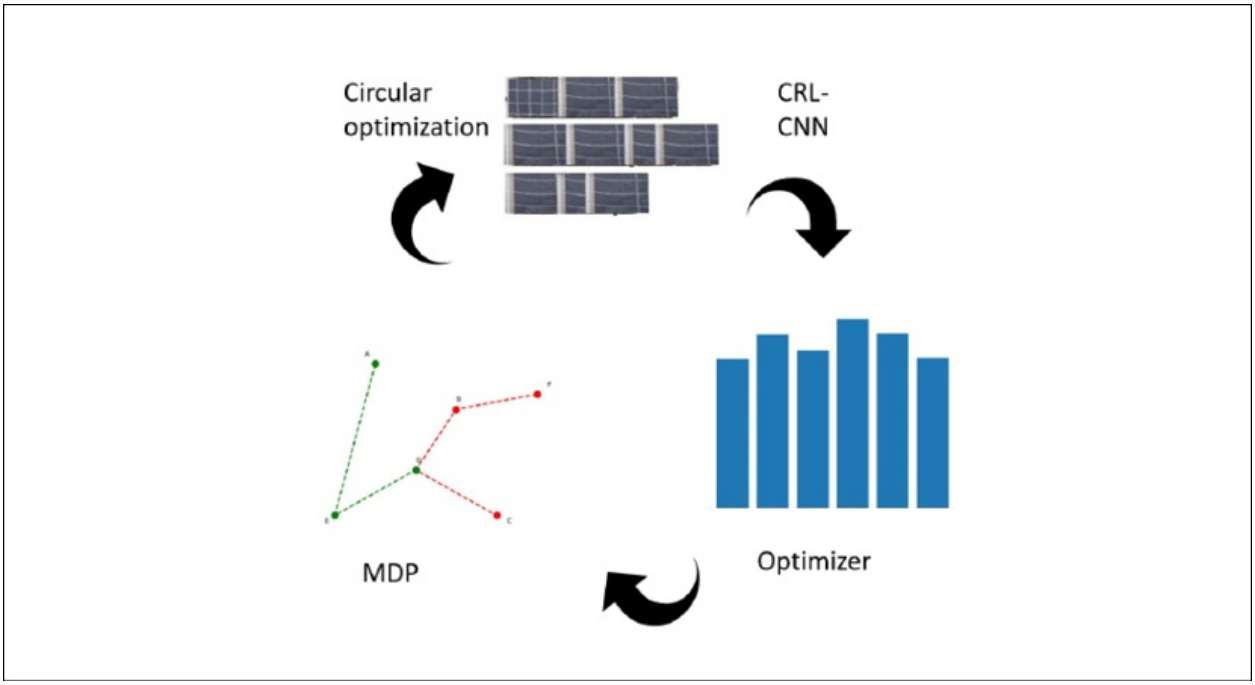


Figure 12.2: Circular RL-DL-CRLMM

This RL-DL-CRLMM model that we explored in *Chapter 11*, *Combining Reinforcement Learning and Deep Learning*, contains the following components:

- A CRL-CNN to see if there is a gap in the image. In this chapter, we will use the same model to see if there is a gap in a parking lot that represents an available parking space.
- An optimizer will rely on an SVM to add the concept of safety to the choice of an itinerary. It will then use optimization rules to make decisions, as in *Chapter 11, Combining Reinforcement Learning and Deep Learning*.
- An MDP for the itinerary as described in *Chapter 1*, *Getting Started with Next-Generation Artificial Intelligence through Reinforcement Learning*.

The RL-DL-CRLMM model of this chapter, which focuses on finding a parking lot with available parking space and go there will thus become:

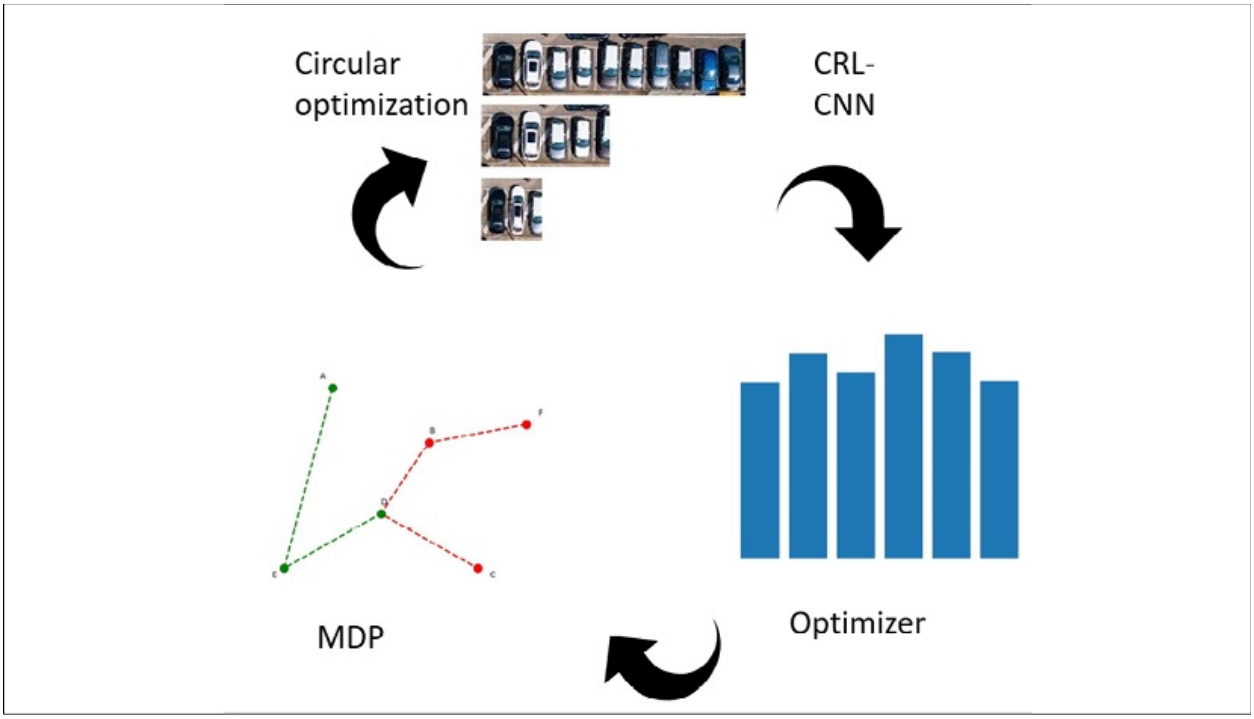


Figure 12.3: Circular RL-DL-CRLMM

The CRL-CNN in this model looks for spaces in a parking lot instead of gaps on a conveyor belt.

The RL-DL-CRLMM model contains a **convolutional neural network** (**CNN**) and a **Markov decision process** (**MDP**) linked together by an optimizer. The optimizer contains an SVM (safety evaluations) and a set of rules.

This system will now be referred to as a CRLMM.

Applying the model of the CRLMM

In *Chapter 9, Abstract Image Classification with Convolutional Neural Networks (CNNs)*, the CRLMM program, `CNN_STRATEGY_MODEL.py`, was trained to identify Γ (gamma concept) in outputs on the conveyor belt of a food processing factory. The end of the previous chapter brought Γ up to a higher abstraction level.

As long as a member of γ (gamma) of the Γ dataset is in an undetermined state, its generalization encompasses ambivalent but similar concepts. Up to this

chapter, these are the concepts Γ (uppercase gamma) has learned (conceptual representation learning).

$$\Gamma = \{\text{a gap, no gap, a load, no-load, not enough load, enough load, too much load, a space on a lane for a car, a distance between a high load of products and missing products on a conveyor belt, weights on sewing stations } \ldots n\}$$

The next step in the chapter is to use the CRLMM built in the previous chapters to recognize a parking space in a parking lot and send a signal to the self-driving vehicle:

- Γ will now perceive gaps as space.
- Γ will now perceive space as a distance (gap) between two objects.
- Γ will need a context to establish whether this space between two objects is a positive or negative distance.

Let's take a look at the dataset that we'll use to accomplish our goals.

The dataset

The datasets used in *Chapter 9, Abstract Image Classification with Convolutional Neural Networks (CNNs)*, to this chapter are sample datasets. In real-life projects, it will take some work to obtain the right real-time frames from a webcam. There will be lighting constraints, prerequisites, and more. However, the philosophy remains the same.

As in the previous chapters, the `dataset` directory of this chapter on GitHub contains the following:

- The training set
- The test set
- The model trained by `CNN_CONCEPT_STRATEGY.py`
- The `classify` directory used by `CNN_CONCEPT_STRATEGY.py`

As explained in the previous chapters, a full stream of frames from a well-configured webcam would take weeks, if not months, to prepare. They are projects in themselves.

The project would first start by deciding to use a camera webcam or an IP camera that can send images to the CNN that is embedded in the RL-DL-CRLMM program. The CNN will classify each image sent by the video feed either as containing a gap or an available parking space or not.

A webcam is usually connected to a computer, which will, in turn, send information to a distant server. An IP camera can send information directly to distant machines. Both solutions are connected IoT objects. An IP camera can be more expensive. To prove that the solution is a good one, an implementation team might start with a webcam first. Once the project has been accepted, then an IP camera might prove to be better in the long run. In this chapter, we will refer to webcams as in a research project with limited funds to begin with. We will thus consider that the images come from a webcam.

In this example, Γ has evolved into space (gap detection between (distance)) cars to find a parking space.

The following is a simulated frozen frame with no Γ-space taken by a webcam located on a building, pointing down to a parking lot:



Figure 12.4: Simulated frozen frame

I transformed the image to simulate some computer vision techniques that could have been used to simplify the image:



Figure 12.5: Parking lot with a little to no gaps (not enough available parking spaces)

The following frame represents a small but sufficient parking space on the right of the screen. The images in these examples were designed to explain how the system is built. I created the image for this example to show whether Γ-space is available or not. Once again, I simulated an image after a computer vision process, which is easy to do but beyond the scope of this book. It shows something like several available parking spaces in a higher-level representation of the parking lot:



Figure 12.6: Parking lot with a little to no gaps (not enough available parking spaces)

Using the trained model

The model was trained by using the same `CNN_STRATEGY_MODEL.py` program as in *Chapter 9, Abstract Image Classification with Convolutional Neural Networks (CNNs).*

Just make sure that the directory in the header of the following program is pointing to `dataset/`, which is scenario number 2:

```
A=['dataset_O/','dataset_traffic/','dataset/']
scenario=2 #reference to A
directory=A[scenario] #transfer learning parameter (choice
```

```
of images)
print("directory",directory)
```

The model is stored in `/dataset/model`. To test the model, `CNN_CONCEPT_STRATEGY.py`, improved in the previous chapter, was used. Just change the messages and limit the frame classification loop to `2`, as shown in the following code snippet:

```
MS1='available'
MS2='space'
I=['1','2','3','4','5','6']
```

The following loaded image was already resized before applying the CNN model:



Figure 12.7: Resized image

Classifying the parking lots

We showed that the model works. However, in the following sections, we will simulate the images that are sent with a random function, not with the actual images.

Now that the CRLMM has been trained to distinguish a full parking lot from a parking lot with available space, once an available parking lot has been found, an SVM takes over as an intermediate step, as we'll see in the next section.

Adding an SVM function

The self-driving car has delivered its packages to the shelters. Now it has to find a parking lot and park there. Instead of having a base like many other systems, this saves the city the cost of many useless trips.

Motivation – using an SVM to increase safety levels

The support vector system adds a new function to itinerary calculations—**safety**.

Most systems, such as Google Maps, focus on:

- The shortest trip
- The fastest trip
- Traffic

However, self-driving cars have to take extra precautions. Many humans do not feel secure on some roads. Safety comes first, no matter what. Once a suitable parking lot has been found, the SVM has to avoid traffic.

The goal is to find a path through traffic, even if the distance is longer. A *p* parameter allows for a *p*% variance in the distance. For example, 10% allows a 10% longer distance and will provide safe passage, as shown in the following SVM result:

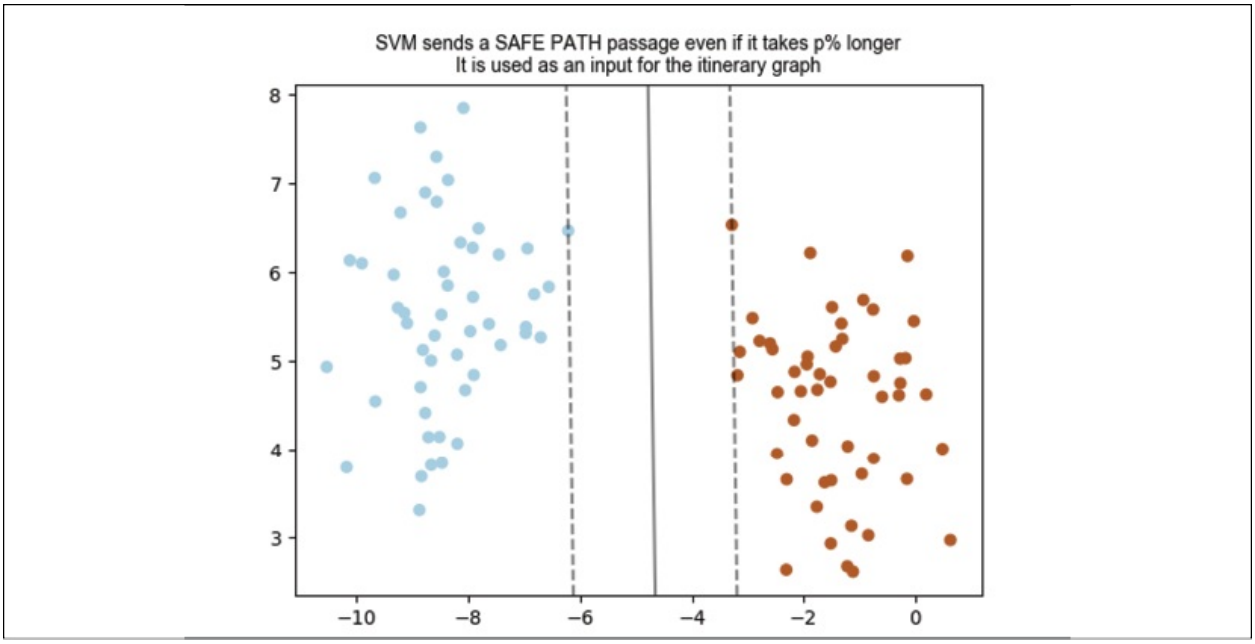


Figure 12.8: Traffic path

It is important to note that the datapoints are **not** the actual coordinates but a representation in a higher dimension, as explained in the following section.

Definition of a support vector machine

An SVM classifies data by transforming it into higher dimensions. It will then classify data into two classes, for example.

In this section, the SVM will be used to separate risky driving locations from safer driving locations:

The example in the code is random (as in real-life traffic), but the model can be developed much further.

Safety is the key to this model, so each driving location (road, crossing) possesses features related to this goal:

- Number of accidents at that location
- Traffic at that location
- Experience of driving through that location (near misses and no problems)

All of this data can be fed into an SVM program. The program will transform the data to make it linearly separable (see *Chapter 8, Solving the XOR Problem with a Feedforward Neural Network*).

The blue dots on the left will be the good locations, and the brown ones on the right will be the risky ones. A function will read the latitude and longitude features of the datapoint in another table to convert them back into GPS format.

For example, a blue dot on the left might be:

- Location A
- One accident in the past ten years
- Zero problems driving through that point in 1 year

A brown dot might be:

- Location D (a few blocks from A)
- Seventy-four accidents in 10 years
- Fifteen problems driving through that point in one year

The blue dots and brown dots thus have nothing to do with the real location on the preceding graph. The locations are **labels**. Their features have been separated as expected.

> To send the data to a GPS guiding system, all that needs to be done is to find the GPS coordinates of the locations that are part of the initial dataset.

Location A will be chosen, for example, instead of location D. Hence, the program looks into the dataset and finds its GPS location.

Let's put some words to the following SVM graph:

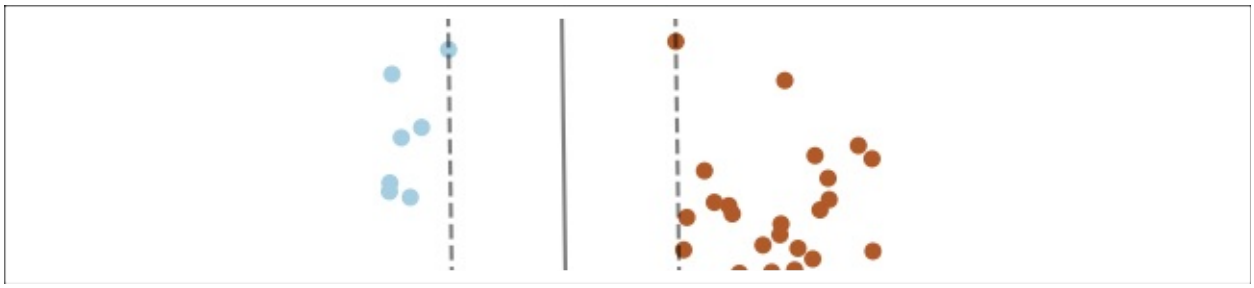


Figure 12.10: SVM graph

- The space between the dotted vertical lines is the **margin**. It's somewhat like the margin between two lines of a rugby or football team. When the players (datapoints) are lined up, an invisible space or margin separates them.
- The dots that touch those margins are critical because they are the ones that decide where the margin will be in the first place. As the rugby or football players line up in clusters, the SVM will calculate this (refer to the following Python function). These special datapoints are called **support points**. They are also referred to as **support vectors**.
- The vertical line running in the middle of the margin is the **decision line**.
- Since a line separates the dots, the dataset is linearly separable. This means that a line can be drawn between the datapoints and can separate them into classes. In this case, the system wants to obtain safe locations (blue dots) and avoid unsafe locations (brown dots).

Now that we defined what an SVM is, let's implement it in Python.

Python function

The `sklearn` packages provide the following `svm` function:

```
from sklearn import svm
from sklearn.datasets import make_blobs
```

The `make_blobs` function generates uniform data for this example in all directions. It is thus an **isotropic** distribution (*iso* = equal, *tropy* = way) of random data. A **blob** contains points of data. The points of data represent the concentration of cars in given areas, which were calculated using their longitude

and latitude.

scikit-learn contains a Gaussian factor for the generation function. A Gaussian kernel applies standard deviations from a mean. Imagine you are playing in a sandbox, and you make a little hill. Then, with your hand, you cut the pile in two. The mean is where you cut the sand pile; the standard deviation is shown by the slopes going down on both sides.

It might take days, and sometimes weeks, to put a good dataset together. But with scikit-learn, you can do it in one line, as shown in the following code snippet:

```
    #100 cars clusters(concentration of cars) represented
 [Line 323]
    X, y = make_blobs(n_samples=100, centers=2, random_state
=7)
```

This function offers many parameters. The ones used are as follows:

- `n_samples`, which represents the number of points spread out between the clusters. In this example, `100` already represents sub-clusters of car concentrations in an area.
- `Centers` is the number of centers from which to generate data. In this example, `2` represents areas close to the present location of the self-driving car and its future destination.
- `random_state` is the seed by the random number generator. This is where the sequence of random numbers will start. This is because what we think is random is pseudo-random, so it has a deterministic basis.

In this example, a linear kernel is used to fit the model, as shown in the following code:

```
    # the model is directly fitted. The goal is a global es
timate    [Line 326]
    clf = svm.SVC(kernel='linear', C=1000)
    clf.fit(X, y)
```

scikit-learn's SVM contains a parameter penalty, which explains the `C` in

`svm.SVC`. There are many more options, but the key option is the kernel. A linear kernel will produce a linear separation, as shown in the preceding screenshot.

An RBF kernel would produce a different result. The structure looks more regularized. As shown in the following screenshot, an RBF kernel acts as an efficient structural regularizing function:

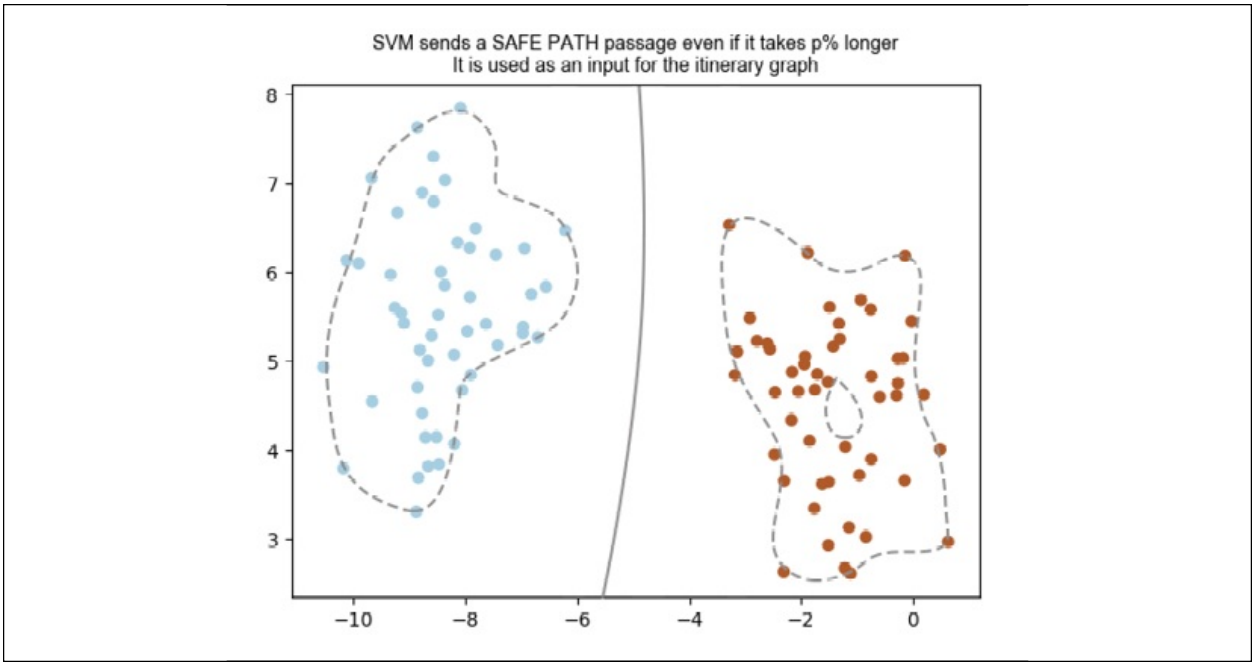


Figure 12.11: Regularized structure path

Bear in mind that the SVM can be used for image recognition on the MNIST and CIFAR datasets, for example. Artificial intelligence provides more than one way to solve a given problem. It is up to you to choose the right tools by having a flexible trial-and-error approach where necessary.

The plotting line instructions start at line 300. The main line of code to take into account is the function that will find and use the decision line (refer to the preceding definition) and scatter the datapoints on both sides of the margin. This is achieved by using the following `decision_function`:

```
Z = clf.decision_function(xy).reshape(XX.shape)
```

The result will be displayed, as shown previously. The SVM is now a component of the CRLMM in this self-driving car (SDC) model. We are ready the run the

CRLMM to find an available parking space for the SDC.

Running the CRLMM

The self-driving car's mission is a circular (no beginning, no end) one like the CRLMM described in the previous chapter:

- If it is in a parking lot, it can be activated by a home or a shelter.
- If it is at a given home, it will go to a shelter.
- If it is at a shelter, it can go to a home or a parking space.
- If it needs recharging, this can be done at a recharging space (or more probably at a parking space), which is already the case in some cities.

At one point, the self-driving car has to go from a specific home to a parking space. This part of its itinerary is the subject of the following sections.

Finding a parking space

`CRL-MM-IoT_SVM.py` uses a fine-tuned version of `RL_DL.py` described in the previous chapter.

A no-Γ (no gamma, no gap, no space) result is not acceptable. The result we are looking for is an image with a gap.

If the `crlmm` function, which classifies parking lot images into full or available space, returns a `0`, the program detects it and displays a message. The code samples contain the line number of the following code excerpt:

```
            if(crlmm==0):    # [Line 392]
                full = load_img("FULL.JPG")
                plt.subplot(111)
                plt.imshow(full)
                plt.title('PARKING LOT STATUS : This parkin
g lot is full.' + '\n' + 'Another webcam is consulted', fon
tname='Arial', fontsize=10)
                #plt.text(0.1,2, "The frame is the input of
```

```
 a trained CNN")
                plt.show()
                '''
                plt.show(block=False)
                time.sleep(5)
                plt.close()
                '''
                print("This parking lot is full, searching.
..")
```

The program displays the following full sign and closes it after a few seconds:

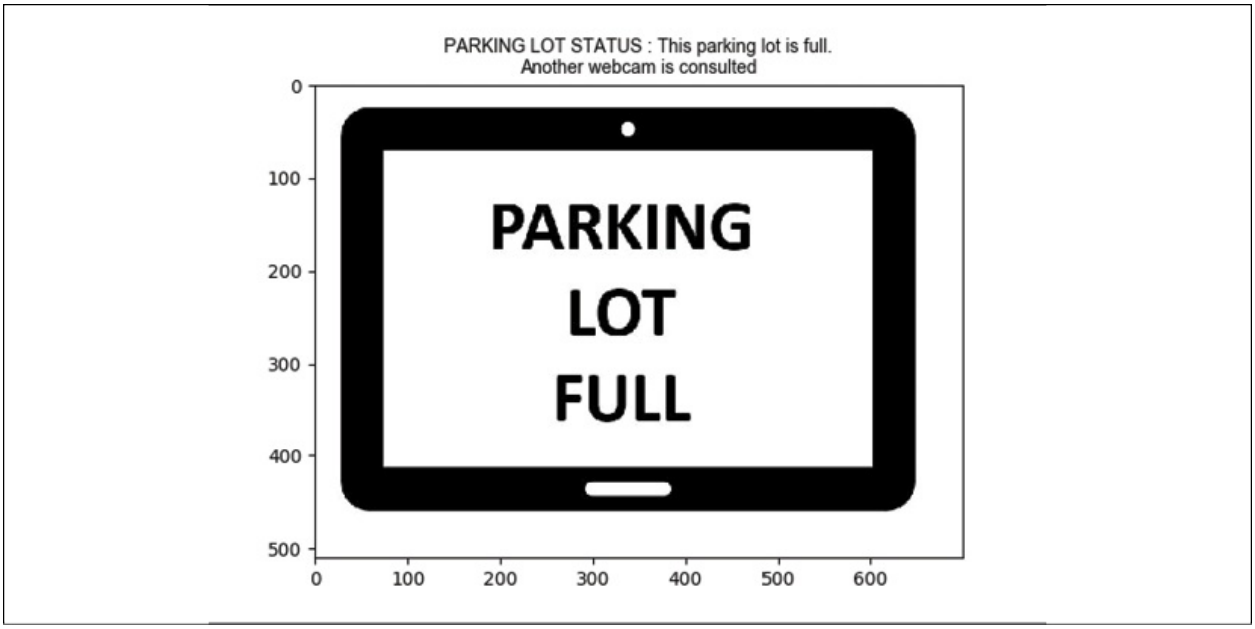


Figure 12.12: Parking lot status

The program must find a parking space. It will thus try searches of good parking lots as shown in this code snippet:

```
        for search in range(1000):    # [Line 391]
            if(crlmm==0):
```

A thousand searches looks like a lot, but it isn't difficult for a machine learning program.

Furthermore, looking for available parking spaces in a large city can be excruciating. More often than not, it will not be suitable: there will not be sufficient available parking spaces to be certain of finding one in the time it will

take to get there.

For this prototype, the number of optimal searches is limited to `2`. Beyond that value, the following `CRLMM` function is activated:

```
                if(search>2):    # [Line 405]
                    a=1
                crlmm=CRLMM(Q,lr,e,a)
```

After two fruitless searches, the program activates `a`, a flag for the `CRLMM` function.

The `CRLMM` function now contains a random search function, which simulates the choice of a parking lot and provides a first-level status:

```
    status=random.randint(0,10)    # [Line 199]
    if(status>5):
        status=1
    if(status<=5):
        status=0
```

The status represents a probabilistic estimate of the availability of the parking lot. The `a` flag simulates a program yet to be added that will scan all the parking lots and run this function to find an available space.

> To present a prototype at an initial meeting, you will always need enough to convince, but if you go too far, the cost of doing this becomes a risk if your idea is rejected.

So, if `a` is activated, the system simulates a scan (to be developed) and forces the status to `1`, as shown in the following code snippet:

```
    if(a>0 and status==0):    # [Line 204]
        #add an available search function here that scans a
ll the
        #webcams of then network until it finds one that su
its the model (not too far parameter and available)
        status=1
```

The program then continues and runs the CNN trained to identify an available parking lot (refer to the screenshot that follows), as explained in the preceding configuration section:

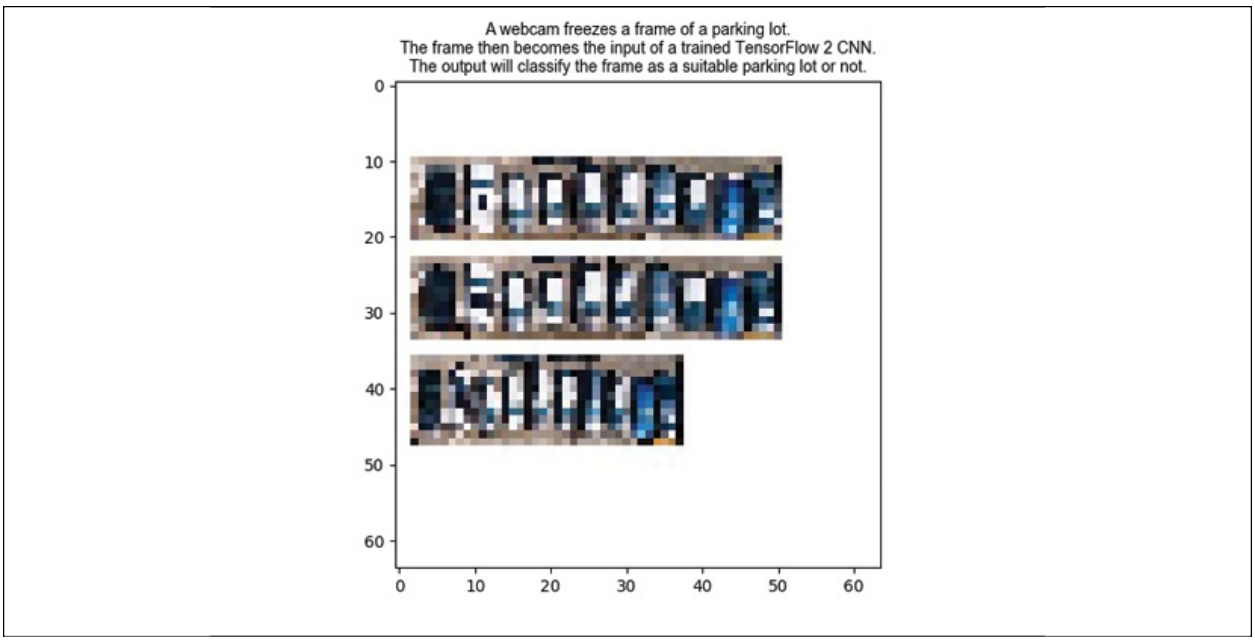


Figure 12.13: Webcam freezes a frame of a parking lot

Now that a parking lot with available space has been found (the empty spaces on the top left of the frame), the search function stops, and the following `break` instruction is activated:

```
                if(crlmm==1):    # [Line 408]
                    a=0
                    break
```

The `break` instruction is reached once a parking space has been found. Once an available parking space has been detected, we can decide how to get to the parking lot.

Deciding how to get to the parking lot

The CRLMM program has found a suitable parking lot, as shown in the following code, when `crlmm==1`:

```
        if(crlmm==1):    # [Line 412]
            available = load_img("AVAILABLE.JPG")
            plt.subplot(111)
            plt.imshow(available)
            plt.title('PARKING LOT STATUS : This parking lo
t has available space.' + '\n' + 'Now an SVM will suggest a
 safe route ', fontname='Arial', fontsize=10)
            #plt.text(0.1,2, "The frame is the input of a t
rained CNN"
            plt.show()
            '''
            plt.show(block=False)
            time.sleep(5)
            plt.close()
            '''
            print("This parking lot has available space..."
)
```

It displays the following message and a sign:



Figure 12.14: Parking lot status

Now that we found an available parking space, we need to find a safe route for the SDC. This means the SDC will avoid traffic to make it easier for the autonomous machine learning program. Now it is time to activate the function.

Support vector machine

The CRLMM now demands a safe route, even if it means taking longer (time or

distance). Self-driving vehicles require a strict safety-comes-first policy.

The program reaches the following SVM function:

```
            print("This parking lot has available space..."
)    # [Line 424]
            SAFE_SVM()
```

The SVM described in the configuration section provides a safe path through traffic, as shown in this screenshot of the result:

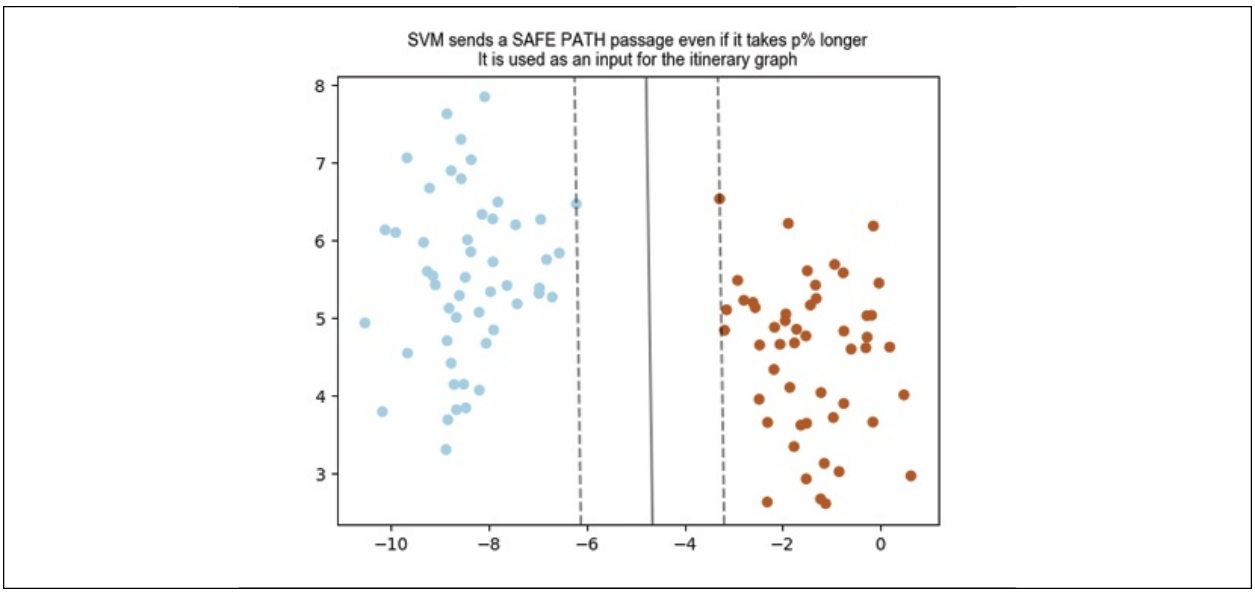


Figure 12.15: Traffic graph

For example, in this random case (traffic will be random most of the time), the blue dots on the left represent sparser traffic, and the brown dots represent areas with denser traffic. The goal, in real-life implementation, is to provide information to the MDP in the program so that it will find paths through sparser traffic for safety reasons to limit the errors an SDC can make in denser traffic. The weights of the statistics of past accidents and the car's experience can also be added to create a deeper vision of this safety model trip planner.

Suppose the self-driving car has to go to a point in the brown area. The SVM will:

- Suggest an itinerary that goes through blue dot areas as much as possible and only then in brown areas.

- Send the information to Google Maps. This will require an additional script that will read a dataset that contains GPS coordinates for each datapoint in the SVM.

Many drivers feel unsafe on loaded freeways or in dense cities. Adding the safest route function to mapping software would help.

The SVM brought the datapoints to a higher level (refer to the explanation in the configuration section of the chapter).

> The points represented in an SVM function are **not** the actual locations but an abstract representation. The dividing line needs to go through a function that will transform that information into a real location datapoint.

Once the SVM boundaries have been converted into location datapoints, the itinerary or trip graph is activated.

The itinerary graph

The prototype shows a simulation of an itinerary graph based on the SVM recommendations and its weight vector through the following function call:

```
            print("SAFE PASSAGE SUGGESTED")    # [Line 426]
            MDP_GRAPH(lr,e)
```

The following graph displays the safest itinerary in red, even if it takes longer (time or distance):

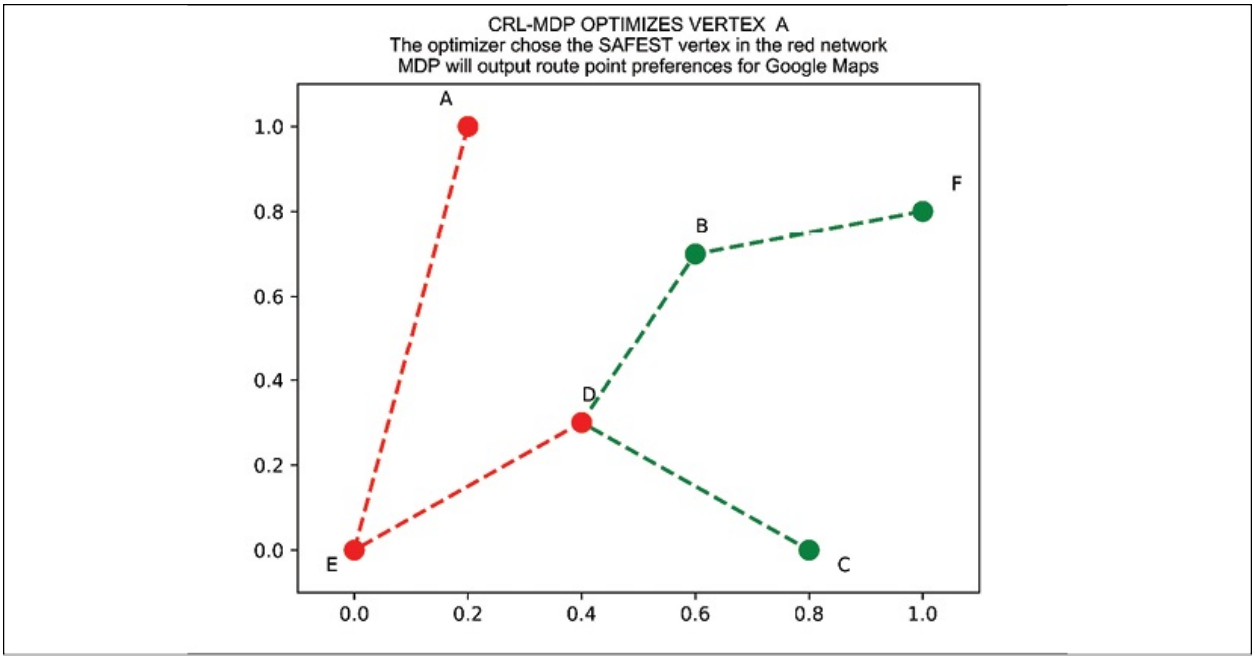


Figure 12.16: The optimizer chose the safest itinerary

For the purpose of the prototype, the SVM was not directly connected to the graph, which would require costly hours.

Instead, the following `random.randint` function was inserted, which simulates the random availability of parking space in any case:

```
for wi in range(6):    [Line 430]
    op=random.randint(0,5)
    if(W[op]>maxw):
        lr=op;maxw=W[op]
```

Bear in mind that it would be useless to execute further development for a prototype as regards the initial presentation to a prospect or manager.

This type of prototype is more powerful than a slideshow because it proves your legitimacy. Slideshows are static and don't prove your abilities on this particular subject. A prototype will show your expertise. Once we have the safest locations, we can update the weight vector for the MDP, as in the previous chapter.

The weight vector

The weight vector is displayed. In this model, the weights represent the

locations, just like in the previous chapter. However, in this chapter, the weights are a rating:

- Each weight has a high safety rank when a few accidents have occurred around that area in the past *n* years. This is a **safety rank**. This ranking should be part of our itinerary software. We should be informed.
- The self-driving car's experience will customize each weight. It is its own driving record. Near misses because of its faulty software will bring the weight down. Good track records will take the weights up. What seems easy for a human might be difficult for software, and vice versa.

The system now displays the weights used with an internal update program to be developed if the prototype is accepted. The following code calls the function that manages the weights of the safest routes for the self-driving vehicle and displays a histogram:

```
print("Vertex Weights",W)    # [Line 428]
MDP_CRL_graph(W,lr)
```

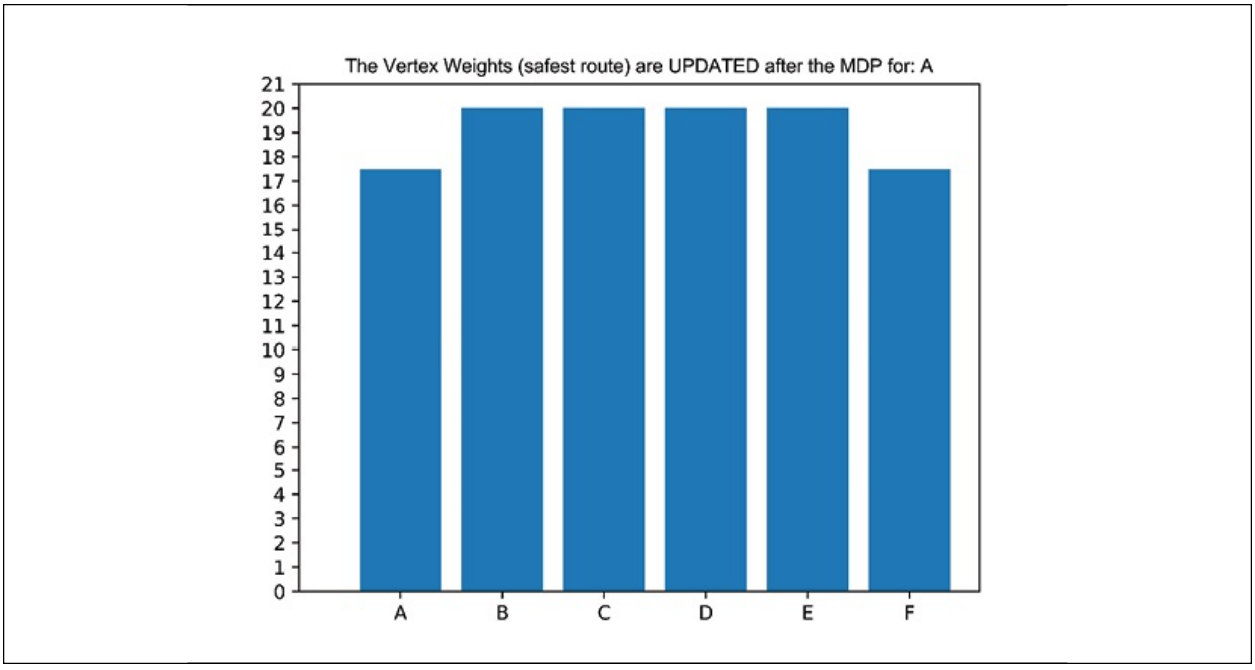


Figure 12.17: Histogram of the updated weights vertex (safest routes)

We detected an available parking space, asked an SVM to provide the safest areas to go through to get to that parking space, updated the weight of each area,

and sent the information to the MDP to calculate a safe route.

Summary

This chapter, like the previous chapter, described a connected IoT process with no humans involved. This trend will expand into every field in the years to come.

This also shows that knowing how to use a tool requires hard work, especially when learning artificial intelligence. Imagining a solution for a given market requires more than hard work. Creativity does not come with work. It develops by freeing your mind from any form of constraint.

Once the solution has been imagined, then comes the fine line between developing too much for a presentation and not showing enough. A CRLMM provides the kind of framework that helps build a technical solution (CNN, MDP, SVM, and optimizers) while keeping everyday concepts that others understand in mind.

The chapter also shows that an artificial intelligence model can contain an ensemble of algorithms, RL, DL, SVM, and CRL cognitive approaches, and more.

The next chapter will take us deeper under the hood of neural networks and TensorFlow 2 by peeking inside the ANN's processes.

Questions

1. Driving quickly to a location is better than safety in any situation. (Yes | No)
2. Self-driving cars will never really replace human drivers. (Yes | No)
3. Will a self-driving fire truck with robots be able to put out a fire one day? (Yes | No)

4. Do major cities need to invest in self-driving cars or avoid them? (Invest | Avoid)
5. Would you trust a self-driving bus to take children to school and back? (Yes | No)
6. Would you be able to sleep in a self-driving car on a highway? (Yes | No)
7. Would you like to develop a self-driving program for a project for a city? (Yes | No)

Further reading

- For more information on SVMs, refer these links: http://scikit-learn.org/stable/modules/svm.html, http://scikit-learn.org/stable/auto_examples/svm/plot_separating_hyperplane.html#sphx-glr-auto-examples-svm-plot-separating-hyperplane-py

13

Visualizing Networks with TensorFlow 2.x and TensorBoard

In this chapter, we are going to take a peek inside a machine's "mind" while it's "thinking" through the layers of a deep learning neural network. The number of lines of code required to build a sequential classifier for a **convolutional neural network** (**CNN**) has been drastically reduced with TensorFlow 2. Running the classifier only takes a click. However, to understand the program when something goes wrong is a more difficult task, and visualizing the outputs of the layers can be very productive.

Visualizing the output of the layers of a CNN can provide an in-depth knowledge of each individual step comprising the whole process.

In this chapter, as in several of the preceding chapters, we will define the layers of a CNN. This time, we will add more layers and extract the output of each layer to create output images. We will build this process from scratch in a bottom-to-top approach in TensorFlow 2 in Python.

Once the outputs have been defined, we will display the output of the convolutional, pooling, dropout, flattening, and dense layers.

Viewing the output of the layers provides an intuitive sense of what the layers are doing. Being able to visualize the global graph of the model makes the architecture of the CNN visible.

We will use TensorBoard to explore the conceptual model, the epochs versus the accuracy, and the detail of the operations of mathematical functions. These graphs and measurements will be built using a top-to-bottom approach using Google Colaboratory.

Google Colaboratory provides a free server with ready-to-use libraries

and modules. We will use a Google Colaboratory notebook to explore TensorBoard functions. The chapter is divided into three main sections. The first two sections describe how to build a sequential classifier with TensorFlow 2.2 and display the outputs of the layers with TensorFlow 2.2. The third section describes how to display the graph information and accuracy measurement with the TensorFlow 2 version of TensorBoard.

The topics covered in this chapter will provide visual insights into CNNs:

- Building a CNN layer by layer
- Displaying the dataset
- Displaying the output of the layers of the CNN
- Using Google Colaboratory
- Visualizing the architecture of a neural network with TensorBoard
- Visualizing accuracy measurements with TensorBoard

Let's start off this chapter by discussing how we can explore the output of the layers within a CNN.

Exploring the output of the layers of a CNN in two steps with TensorFlow

Many corporate contracts in the field of business intelligence require an explanation process for any algorithm that makes automatic and critical decisions. It is often mandatory for the editor of algorithms, artificial intelligence or not, to provide an explanation. We need to be prepared for that.

Also, maintenance becomes critical once artificial intelligence runs in production. Developers often move from one department to another, from one company to another. The person that has to maintain the program needs to understand it in detail.

Exploring and visualizing a CNN is a good way to get our hands dirty, open the hood of our roadster and see how the engine works!

- First, we will first build the CNN layer by layer. We will be building the sequential classifier with TensorFlow 2 from the bottom to the top.

  We will not be using a Keras model directly; we'll use TensorFlow's integrated Keras module, which brings the number of lines of header down to only two lines:

  ```
  import tensorflow as tf
  from tensorflow.keras import datasets, layers, models
  ```

- Then we will explore the visual output of the layers to gain insights into the way it "thinks."

With that, let's get on with building!

Building the layers of a CNN

A CNN was described in *Chapter 9, Abstract Image Classification with Convolutional Neural Networks (CNNs)*. In the example to follow, the CNN will contain more layers to visualize the way a neural network extracts features step by step.

We will be using a dataset that uses a single image to explore the layers of a CNN. This image is repeated several times for the training dataset and the test dataset is enough to build and run the model to visualize the layers of the neural network.

The dataset contains an image of a flower repeated several times—an iris.



Figure 13.1: The image of the image we are exploring in this chapter

The goal is not to have many variations of the image, but to simply see how a CNN represents an iris layer by layer. The dataset contains a repeated image. However, you can change these images and use a dataset of your own then display the images as follows using the same code:

```
cv_img=[]
images = []
for img_path in glob.glob('dataset/training_set/img/*.png')
:
    images.append(mpimg.imread(img_path))

plt.figure(figsize=(20,20)) #20,10
columns = 5
for i, image in enumerate(images):
    plt.subplot(len(images) / columns + 1, columns, i + 1)
    plt.imshow(image)
```

The result will be a figure with lines of images from the dataset:

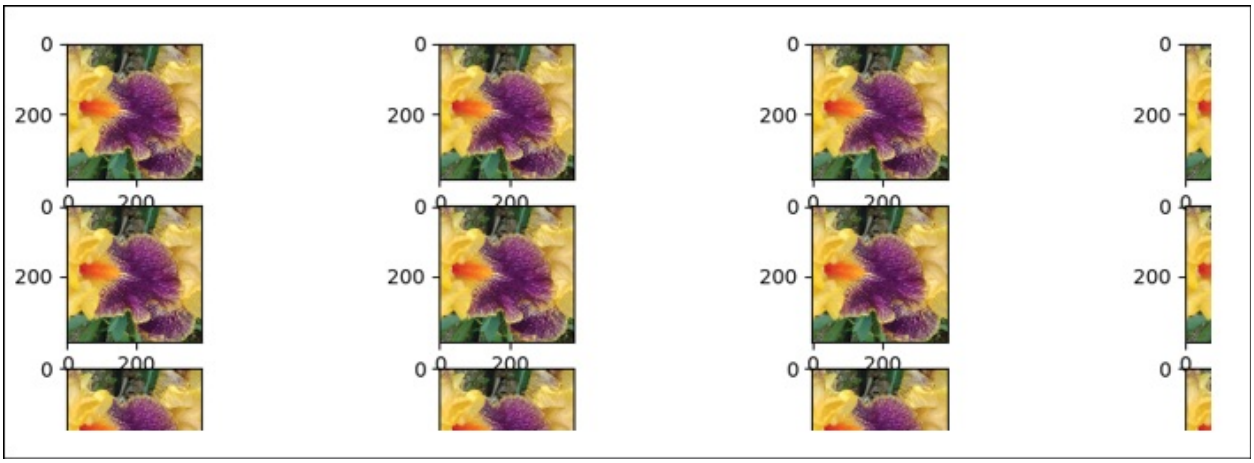


Figure 13.2: Displaying the dataset

We will first import the neural network modules:

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
```

Building the CNN only takes a few lines. This makes it deceptively simple because it appears to be a black box. In our example, the structure described in *Chapter 9, Abstract Image Classification with Convolutional Neural Networks (CNNs)*, in its enhanced version here, only requires a few minutes to create from lines 30 to 68:

```
#initializing the Tensorflow 2 classifier
classifier = models.Sequential()
#adding the convolution layers to the layers
classifier.add(layers.Conv2D(32, (3, 3), padding='same', in
put_shape = (28, 28, 3), activation = 'relu'))
classifier.add(layers.Conv2D(32, (3, 3), activation='relu')
)
...
#adding dense-dropout-dense layers
classifier.add(layers.Dense(units = 512, activation = 'relu'
))
```

We will go back to these layers in the next section when we explore their output. The main point to focus on here remains the simplicity of the code. Building a CNN as a black box in a few minutes might work. However, understanding each layer when a problem comes up requires a deeper understanding of the

representations of the layers.

Before exploring those layers, the program prints the structure of the classifier (CNN):

```
#Printing the model summary
print("Model Summary",classifier.summary())
```

The model contains a fair number of layers to explore:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        896
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 26, 26, 32)        9248
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 13, 13, 64)        18496
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 11, 11, 64)        36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
```

```
dropout_1 (Dropout)          (None, 5, 5, 64)          0

_________________________________________________________________
_____
conv2d_4 (Conv2D)            (None, 5, 5, 64)          3692
8
_________________________________________________________________
_____
conv2d_5 (Conv2D)            (None, 3, 3, 64)          3692
8
_________________________________________________________________
_____
max_pooling2d_2 (MaxPooling2 (None, 1, 1, 64)          0

_________________________________________________________________
_____
dropout_2 (Dropout)          (None, 1, 1, 64)          0

_________________________________________________________________
_____
flatten (Flatten)            (None, 64)                0

_________________________________________________________________
_____
dense (Dense)                (None, 512)               3328
0
_________________________________________________________________
_____
dropout_3 (Dropout)          (None, 512)               0

_________________________________________________________________
_____
dense_1 (Dense)              (None, 3)                 1539

=================================================================
====
```

Keep an eye on this summary. It will prove useful when choosing the number of layers you wish to explore to visualize the outputs.

The model is then compiled:

```
# Compiling the convolutional neural network (CNN)
classifier.compile(optimizer = 'rmsprop',
```

```
    loss = 'categorical_crossentropy',metrics = ['accuracy'
])
```

Then training and test datasets are processed (rescaled) and defined:

```
train_datagen = ImageDataGenerator(rescale = 1./255)
test_datagen = ImageDataGenerator(rescale = 1./255)
training_set = train_datagen.flow_from_directory(
                                    'dataset/training_set',
                                    target_size = (28, 28),
                                    batch_size = 16,
                                    class_mode =
                                    'categorical')
test_set = test_datagen.flow_from_directory('dataset/test_s
et',
                                            target_size = (28
, 28),
                                            batch_size = 16,
                                            class_mode =
                                            'categorical')
```

If we stop here, the CNN will work. But will we have really understood the model? I don't think so. Of course, it only takes a simple click to get the CNN to run after installing a ready-to-use dataset. This black box approach can work, but exploring the visual output of a layer provides a better representation of the network. Let's take a look at that next.

Processing the visual output of the layers of a CNN

The idea is to focus on an image and actually *see* the "mental," visual, representation a CNN calculates, layer by layer.

To process the layers, the program first selects an image for the activation model to work on:

```
#Selecting an image for the activation model
img_path = 'dataset/test_set/img/img1.png'
img1 = image.load_img('dataset/test_set/img/img1.png', targ
```

```
et_size=(28, 28))
img = image.img_to_array(img1)
img = np.expand_dims(img, axis=0)
img /= 255.
plt.imshow(img[0])
plt.show()
print("img tensor shape",img.shape)
```

Then the visualization process runs in a few steps, which will take us inside the CNN:

- **Selecting the number of layers to visualize using the `e` variable**: Going back to the model summary displayed previously, you can choose the layer you want to stop at. In this example, we're stopping at `e=12`. You can choose to start with `e=4` to visualize the first convolutional and pooling layers:

  ```
  #Selecting the number of layers to display
  e=12 #last layer displayed
  layer_outputs = [layer.output for layer in classifier.l
  ayers[0:e]]
  ```

  If `e=3`, the program will stop at `max_pooling2d`:

  ```
  Displaying layer: conv2d
  Displaying layer: conv2d_1
  Displaying layer: max_pooling2d
  ```

- **Selecting the top n layers that will be explored**: The program refers to `layer_outputs` to extract the information it needs to visualize the target layers:

  ```
  # Extracting the information of the top n layers
  activation_model = models.Model(inputs=classifier.input
  ,
                                  outputs=layer_outputs)
  ```

- **Applying the activation model to extract the requested layers**: Activating the model forces the classifier to get to work and run through the layers. That way, we can peek inside its "thought" process and see how it represents inputs:

  ```
  # Activating the model
  activations = activation_model.predict(img)
  ```

- **Retrieving the layer names to display, along with the visual representation of the layer**: Layer names help us understand what we are looking at. Use the model summary we printed earlier as a map to see where you are when the layer name is displayed, along with the representation of the output of that same layer:

```
#layer names
layer_names = []
for layer in classifier.layers[:12]:
    layer_names.append(layer.name)
```

- **Processing layer outputs and organizing them into grids**: To avoid having to watch a sequential display of the variations of the representations in a given layer, we are going to organize them in one grid image:

```
# Processing the layer outputs
for layer_name, layer_activation in zip(layer_names,
        activations):
    #getting the layer_names and their activations
    n_features = layer_activation.shape[-1] #features in
 the layer
    size = layer_activation.shape[1] #shape of the feat
ure map
    n_cols = n_features // images_per_row #number of im
ages per row
    display_grid = np.zeros((size * n_cols,
                             images_per_row * size)) #s
ize of the display grid
    for col in range(n_cols): #organizing the columns
        for row in range(images_per_row): #...and rows
to display
            image = layer_activation[0,:, :,
                col * images_per_row + row] #retrieving
 the image...
            image -= image.mean() #...and processing it
in the...
            if(image.std()>0): # ...following lines to
display it
                image /= image.std()
                image *= 64
                image += 128
                image = np.clip(image, 0,
                    255).astype('uint8')
                display_grid[col * size : (col + 1) * s
ize,
                    row * size : (row + 1) * size] = im
```

```
age
```

- **Displaying the processed layer outputs**: Now that the work is done, we just have to display the layer names along with the corresponding grids:

```
    #displaying the layer names and processed grids
    print("Displaying layer:",layer_name)
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                        scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='virid
is')
    plt.savefig("dataset/output/"+layer_name)
    plt.show()
```

Note that the figures are saved by `plt.savefig` in an output directory for later use.

You will obtain a list of figures with the name of the layer for the layers you chose to visualize. For example, you can view the images of the first seven layers. You can look at them in the following figures or by running the program. In any case, the best way to analyze the layers is to look closely at the first layer and then at the last layer. You will see that the CNN is carefully extracting an abstract representation of the image and displaying higher dimensions. The reason the differences between the layers are difficult to perceive with the human eye comes from two factors:

- The number of elements to analyze is extremely difficult for us to observe. Usually our brain does this without us having to think about it!
- There are several convolutional layers versus one layer that would rush through the process of obtaining an abstract representation of the image. It's going layer by layer, just like a human brain processes an image step-by-step.

Look at the following first layer then the last one, then go back and observe the differences between the layers.

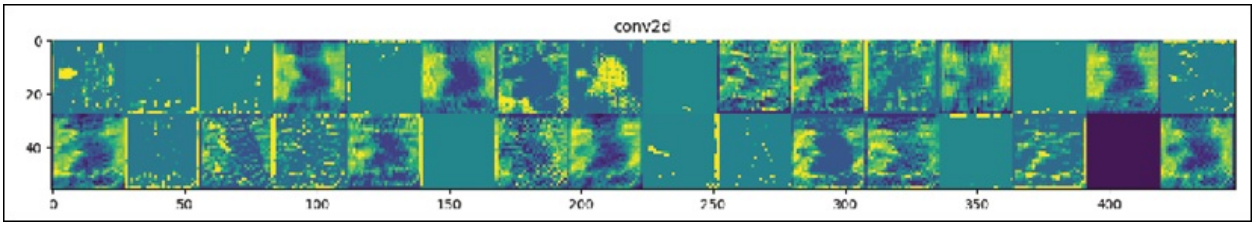


Figure 13.3: Convolutional layer

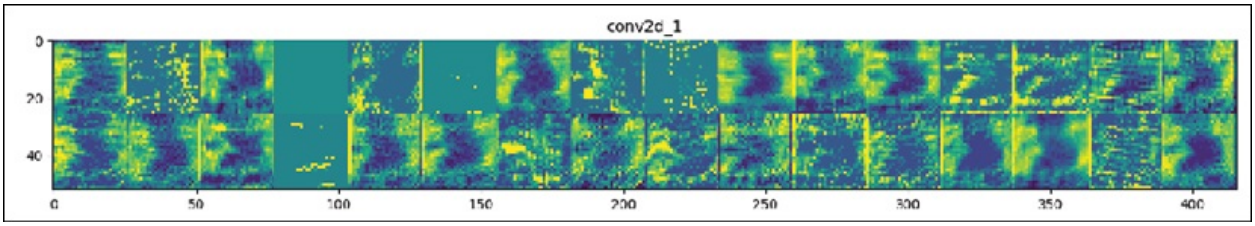


Figure 13.4: Convolutional layer

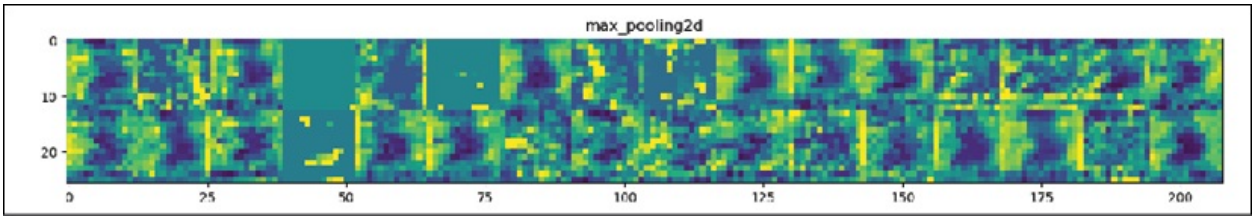


Figure 13.5: Pooling layer

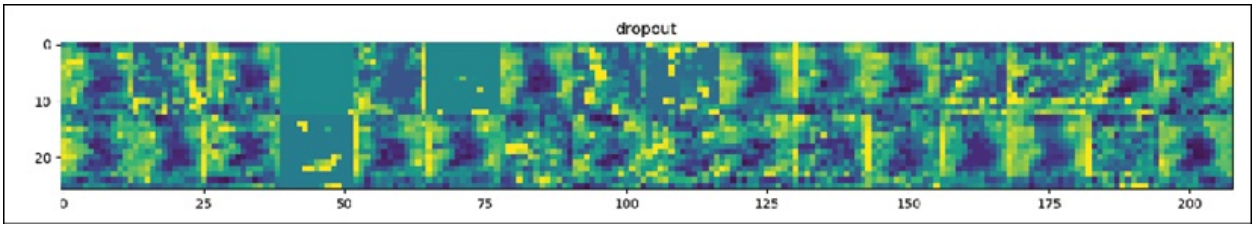


Figure 13.6: Dropout layer

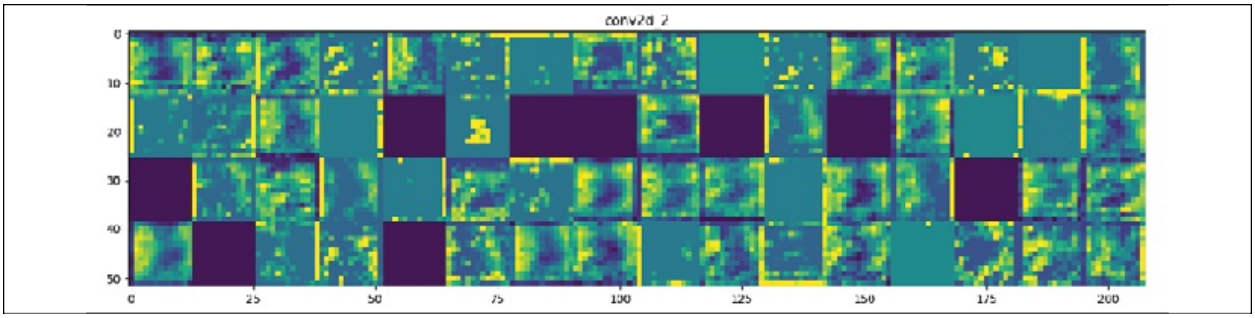


Figure 13.7: Convolutional layer

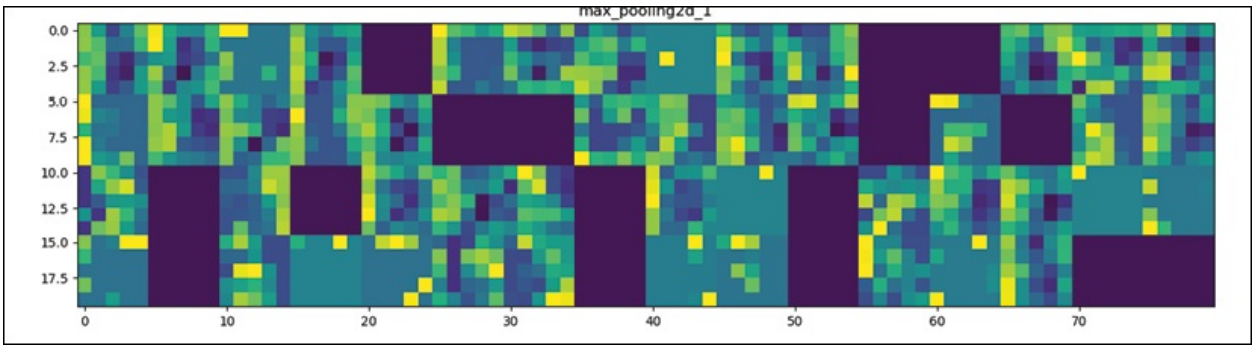


Figure 13.8: Convolutional layer

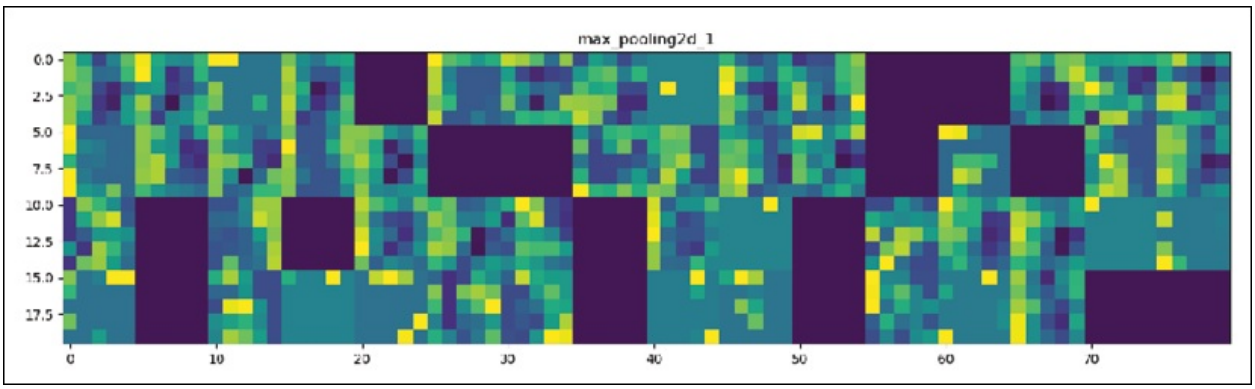


Figure 13.9: Pooling layer

Visualizing the output of the layers of a CNN provides a fantastic way to understand and analyze a neural network. Let's go a step further and analyze the layers.

Analyzing the visual output of the layers of a CNN

An input image is chaos until some form of intelligence makes sense of it. Any form of intelligence will detect patterns and structures. Machine intelligence works on the same grounds by increasing the level of abstraction through dimensionality reduction. The process of going from chaos to an organized representation is at the heart of the fantastic invention of present-day neural networks.

When running `cnn_layers.py`, the layer outputs will be displayed. Let's explore some of the layers. You can explore some or all of them by simply changing the value of the `e = <number-of-layers>` variable on line 107.

Convolutional layer activation functions

One of the key options of a convolutional layer is the activation function. `relu` is used in the following `cnn_layers.py`:

```
#adding more convolution layers to the layers
classifier.add(layers.Conv2D(64, (3, 3), padding='same', ac
tivation = 'relu'))
classifier.add(layers.Conv2D(64, (3, 3), activation='relu')
)
```

For more on ReLU, please go the explanation in *Chapter 9, Abstract Image Classification with Convolutional Neural Networks (CNNs).*

`relu` produces the following output for the `Conv2d` layer:

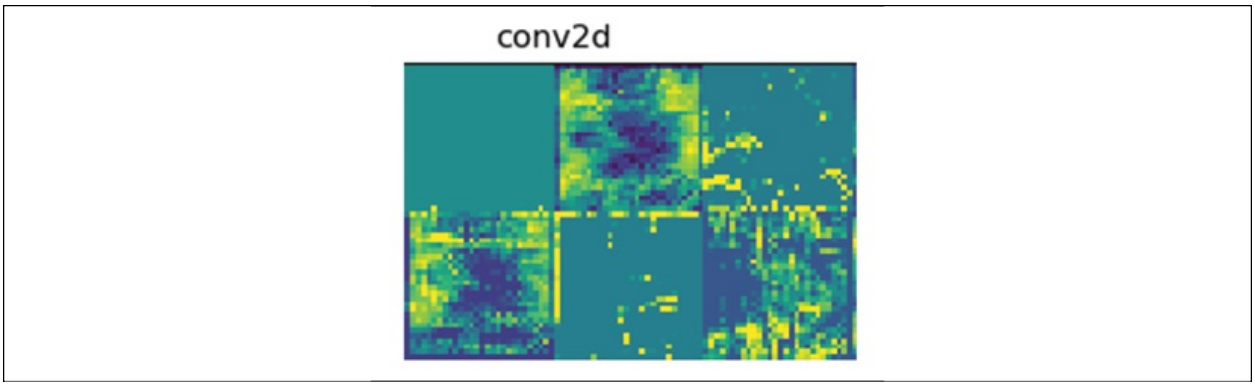


Figure 13.10: conv2d output 1

Now go to line 33 and replace `relu` with `softmax` as follows:

```
classifier.add(layers.Conv2D(32, (3, 3), padding='same',
    input_shape = (28, 28, 3), activation = 'softmax'))
```

The output is quite different, as we can see:

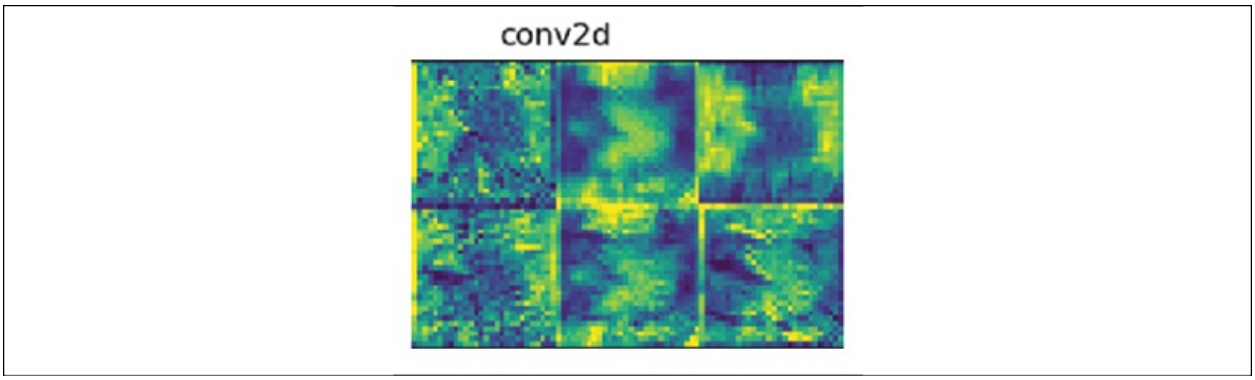


Figure 13.11: conv2d output 2

It takes some time for the human eye to adjust to the change. Look at each version. Try to memorize it for a few seconds by closing your eyes and then looking at the other one. Our brain does this implicitly which is why it takes an effort to do this explicitly.

Which one should you use? Welcome to deep learning! There is no certain answer to that question. It is a trial-and-error process. An activation function might fit one model and not another. Even if the accuracy of the network is acceptable during the training process, you might have to change the activation function over time when new data produces bad results.

For more on softmax, please go back to the explanation in *Chapter 2*, *Building a Reward Matrix – Designing Your Datasets*.

Let's try the logistic sigmoid activation function, `sigmoid`, also described in *Chapter 2*:

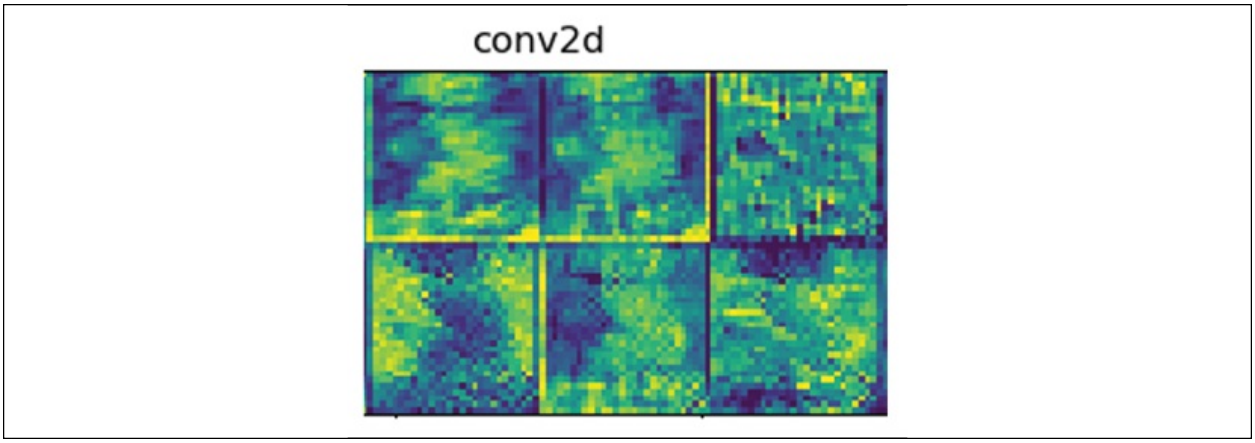


Figure 13.12: conv2d output 3

Notice the differences again. Observe the last image on row 1 for each activation function. The differences are fascinating because they provide a variety of possible representations.

Try other activation functions to get the feel of the way an artificial neural network transforms what it perceives into a higher level of abstraction through a reduction of the dimensions that it has to process.

Convolutional layers higher-level representations through the layers

Notice the incredible level of abstraction the sequential classifier reaches through the following outputs going from `conv2d` to `conv2d_5`, which is, in fact, the sixth (0 to 5) convolutional layer of `cnn_layers.py`.

The network starts at a relatively figurative representation to reach a highly abstract level at `conv2d_5`. We are literally inside the machine's "mind," watching it think and learn!

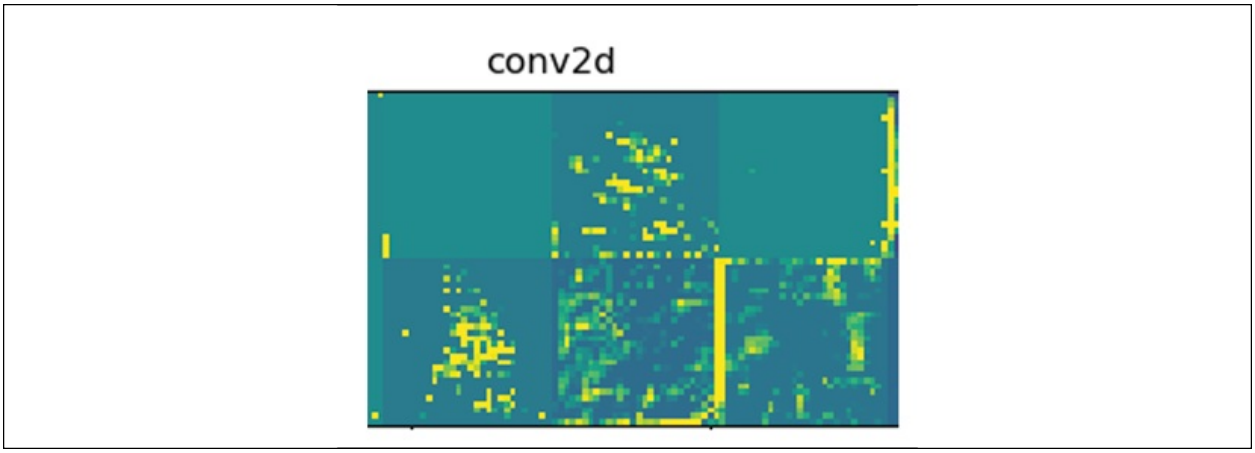


Figure 13.13: Initial conv2d output

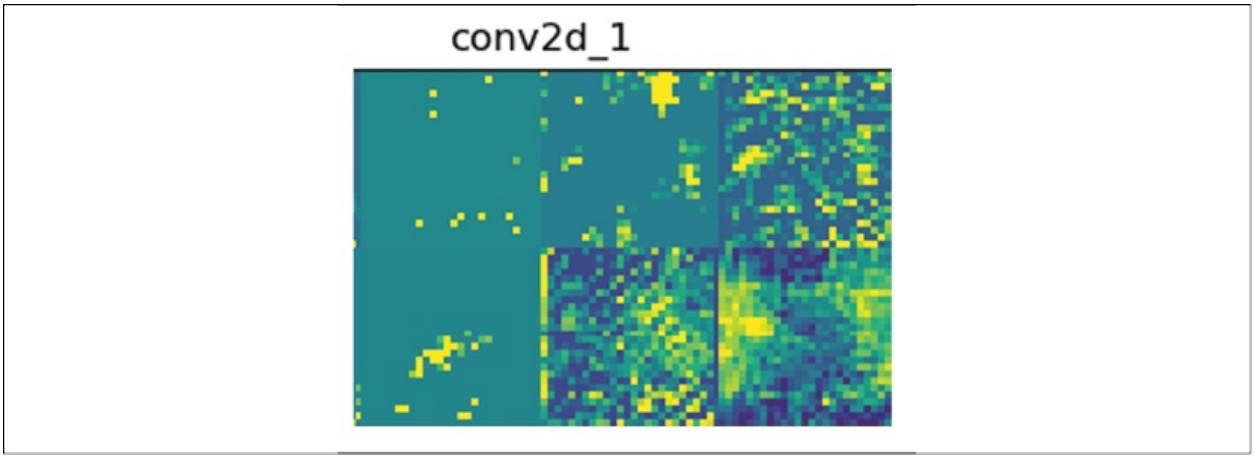


Figure 13.14: conv2d_1 output

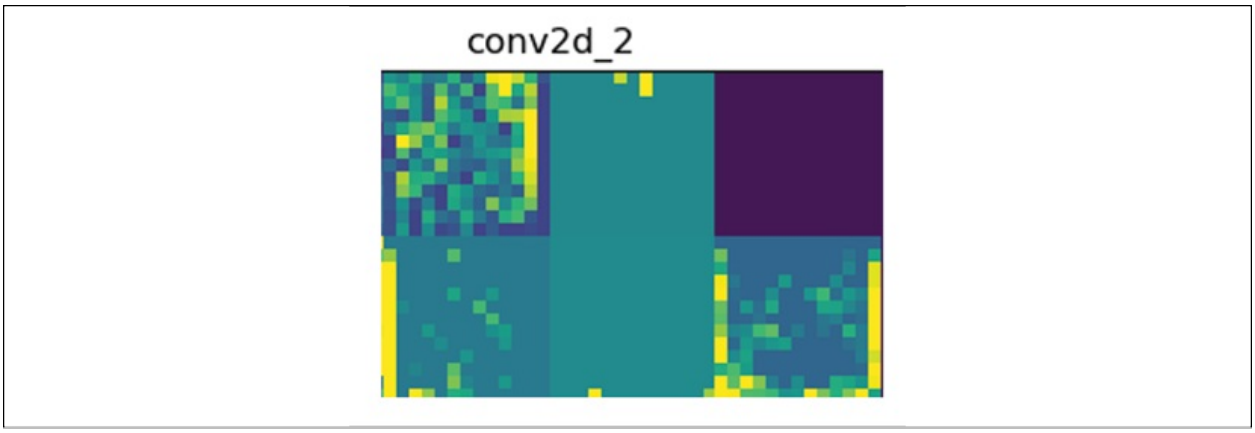


Figure 13.15: conv2d_2 output

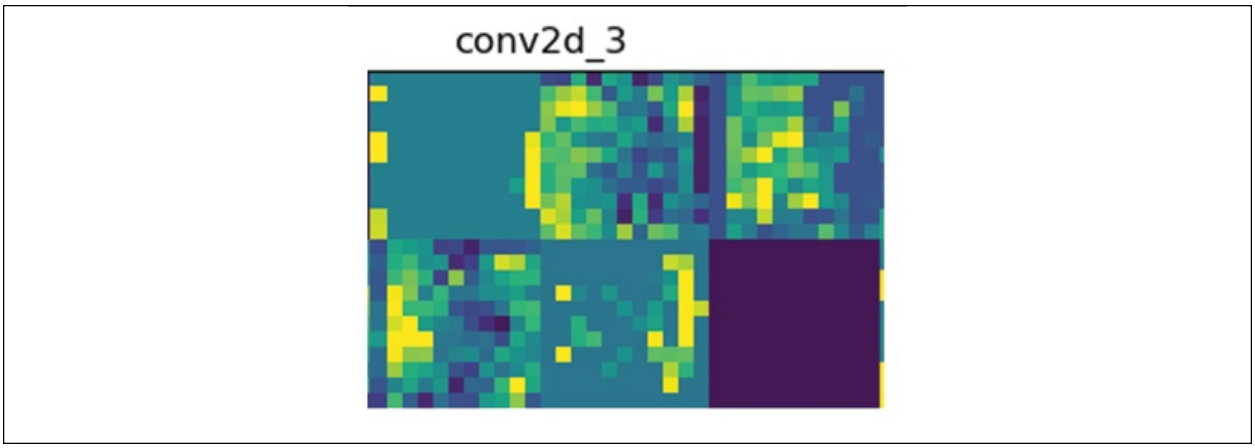


Figure 13.16: conv2d_3 output

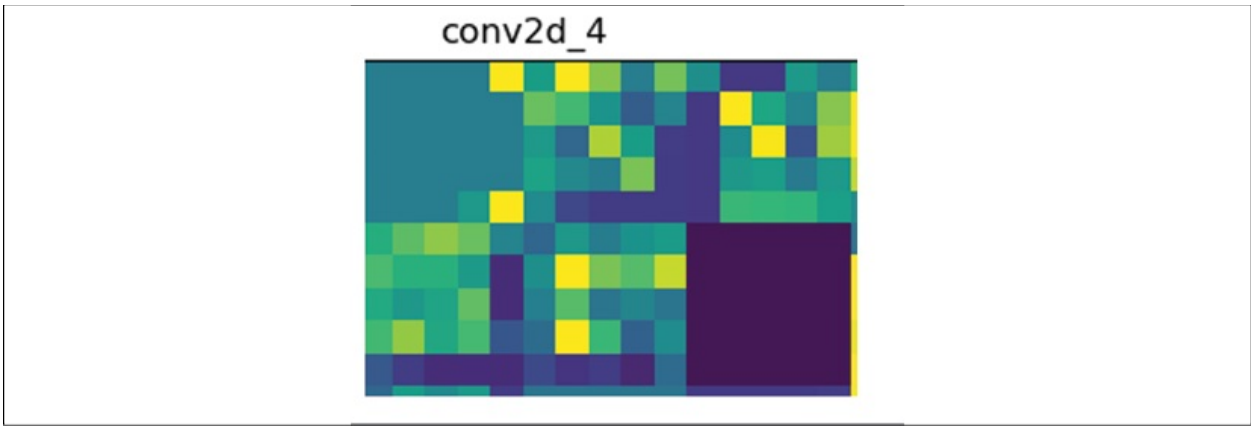


Figure 13.17: conv2d_4 output

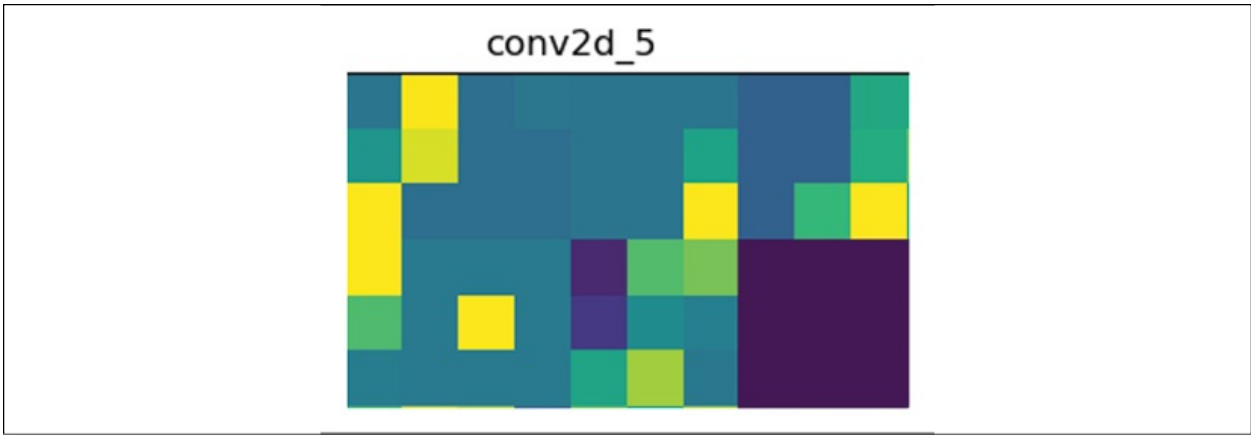


Figure 13.18: conv2d_5 output

This abstraction process owes a lot to the other layers, such as the pooling layer.

Pooling layers to obtain higher-level representations

The pooling layer is going to reduce the number of dimensions of its input and choose the most representative features it finds:

```
#adding a max pooling layer to the layers
classifier.add(layers.MaxPooling2D(pool_size=(2, 2)))
```

Let's explore the evolution of the first two pooling layers in this example:

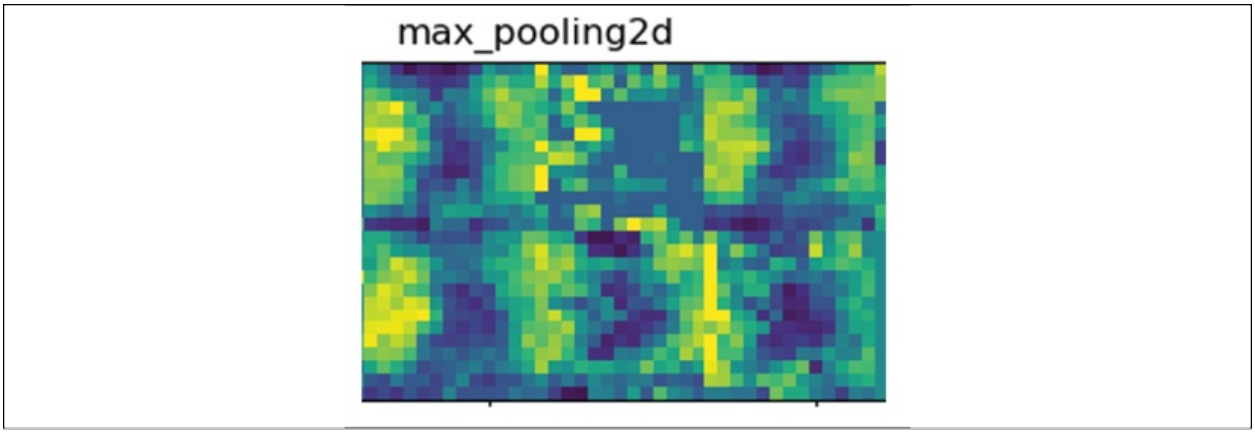


Figure 13.19: max_pooling2d output

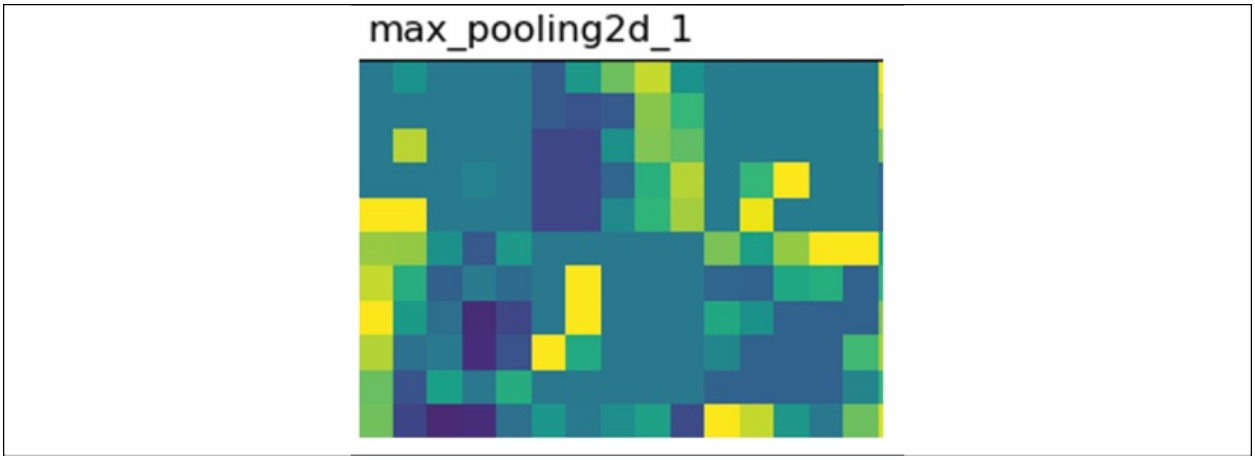


Figure 13.20: max_pooling2d_1 output

Once again, we can see the powerful level of abstraction a CNN can attain.

For more information on the pooling layer, please read the explanations in *Chapter 9, Abstract Image Classification with Convolutional Neural Networks (CNNs).*

Dropout layer higher-level representations through the layers

The dropout layers provide a way to abandon many features in order to reach a simplified higher level of representation:

```
classifier.add(layers.Dropout(0.5)) # antes era 0.25
```

It is not always necessary to add a dropout layer because it depends on the

productivity and architecture of the model you are exploring. For this example, the first two dropout layers are quite instructive. Dropouts are also a way to avoid overfitting. The model learns how to extract key features to obtain an abstract representation, not a literal one.

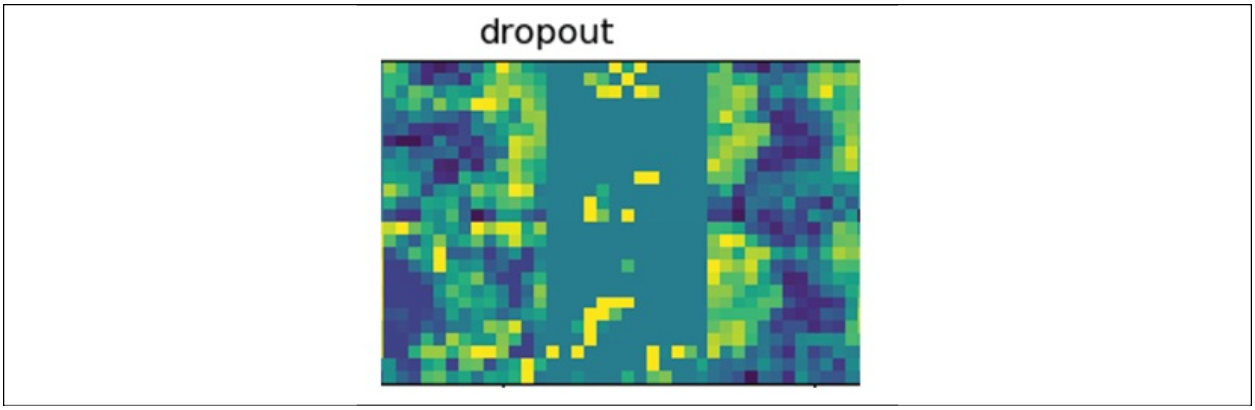


Figure 13.21: dropout output

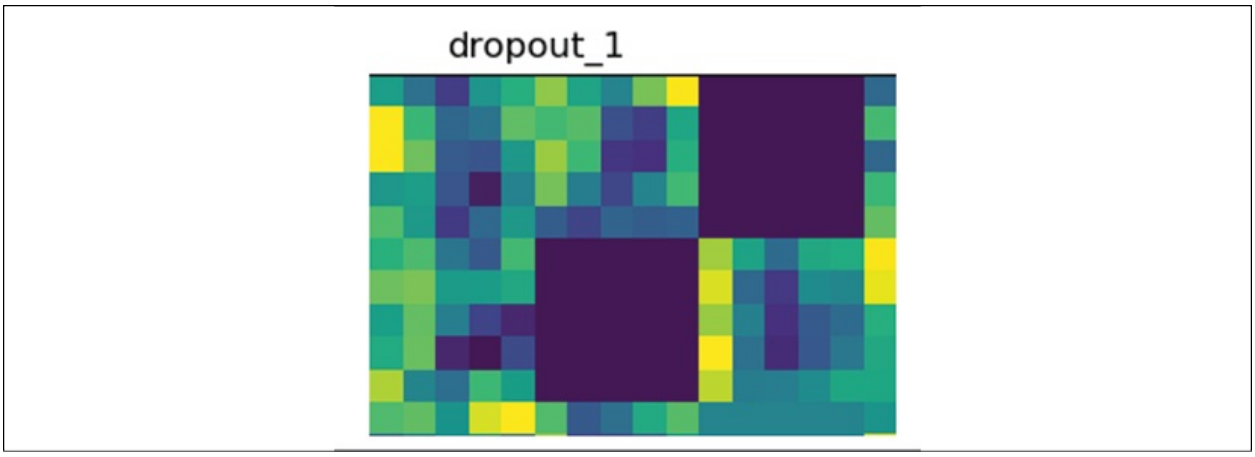


Figure 13.22: dropout_1 output

Observe again how the dropout layers accelerate the abstraction process. We can see the CNN's "mind" working.

Recommendation

I recommend you try different activation functions and various options for the layers of this example. Then run the program and get a feel of what is going inside a CNN's "machine thinking" process. Even if it is a purely mathematical architecture, it provides a good idea of how a CNN, while not human at all, has its own "machine thinking" approach.

Now that we have explored a CNN from bottom to top, let's see how to observe the accuracy of a CNN from top to bottom using TensorBoard.

Analyzing the accuracy of a CNN using TensorBoard

In this section, we will first get started with a free Google Colaboratory server and then explore some of the TensorBoard ANN measurement functions.

Getting started with Google Colaboratory

You can get access to your free instance of Google Colaboratory server in just a few steps:

1. Make sure you have a Google account and log into it.
2. Click on the following link, which takes leads you to Google Colaboratory: https://colab.research.google.com/notebooks/welcome.ipynb#recent=true

   You will be taken to the following page:

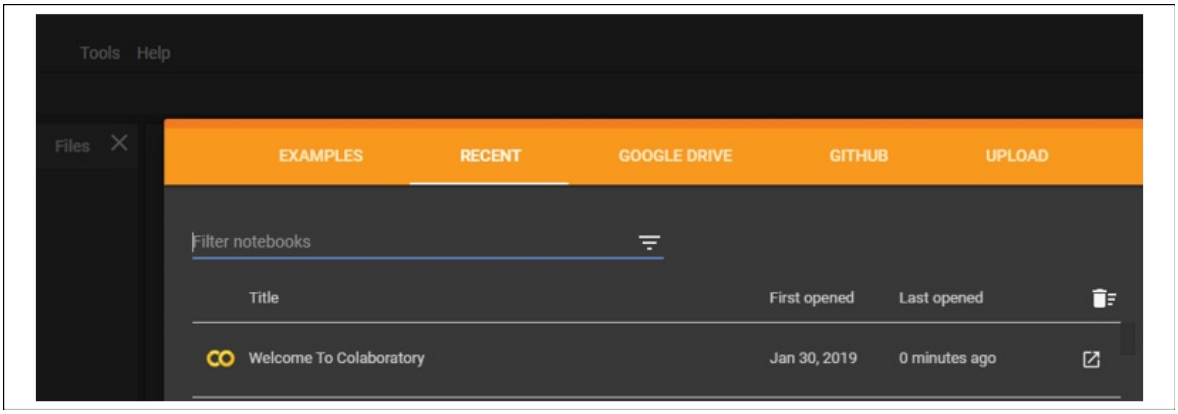


Figure 13.23: Colaboratory initial landing page

3. Click on the **UPLOAD** option in the top right:

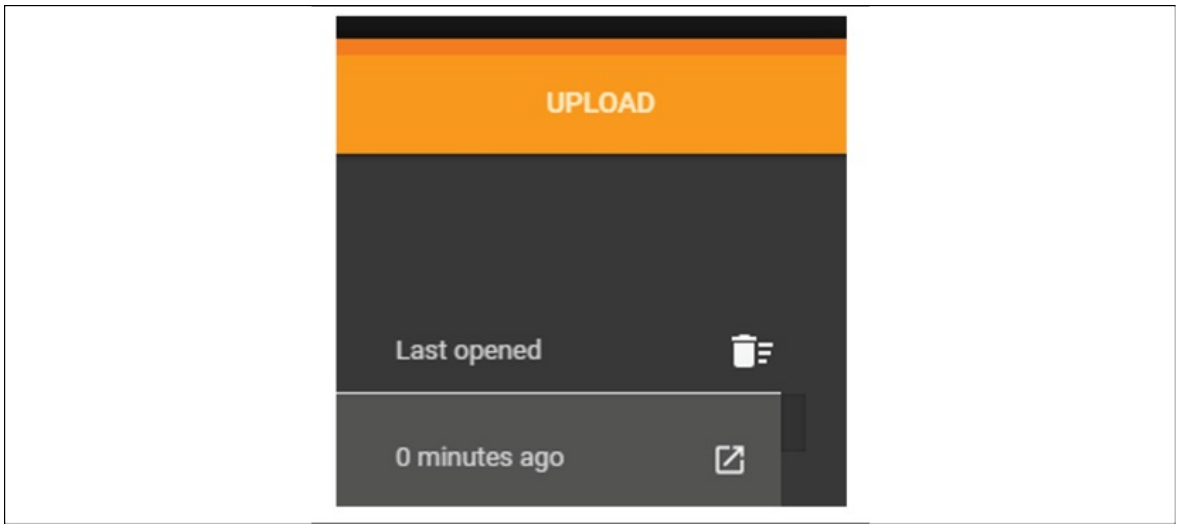


Figure 13.24: Upload option

You can choose or drag and drop a file, then upload it.

4. Upload `TF_2_graphs.ipynb`.

   You can download the program from this link and then upload it: https://github.com/PacktPublishing/Artificial-Intelligence-By-Example-Second-Edition/blob/master/CH13/TF_2_graphs.ipynb

5. Once the program is open, you will see the following page:

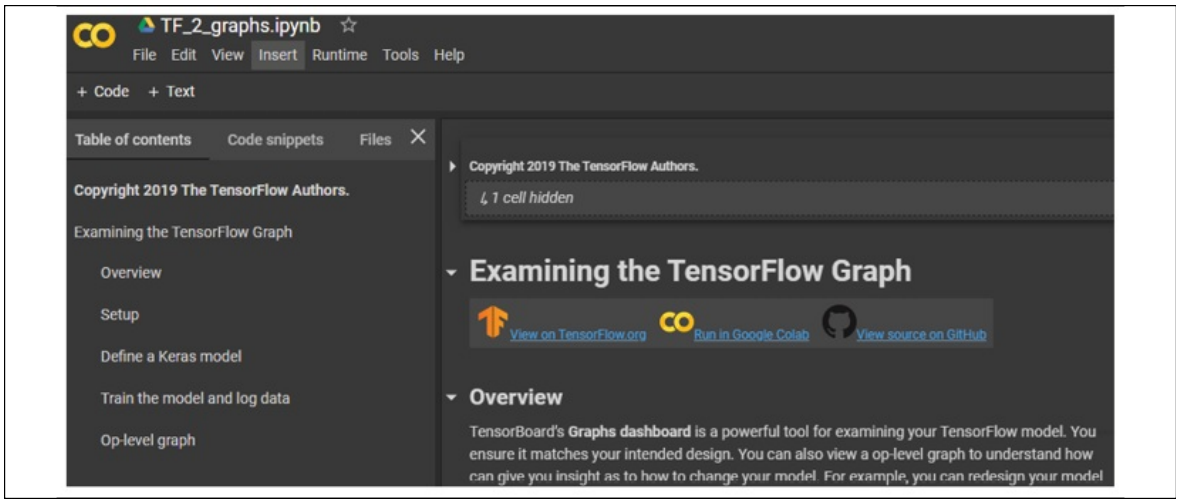


Figure 13.25: A Colaboratory notebook

6. Go to **File** in the menu and save the notebook to your Google Drive:

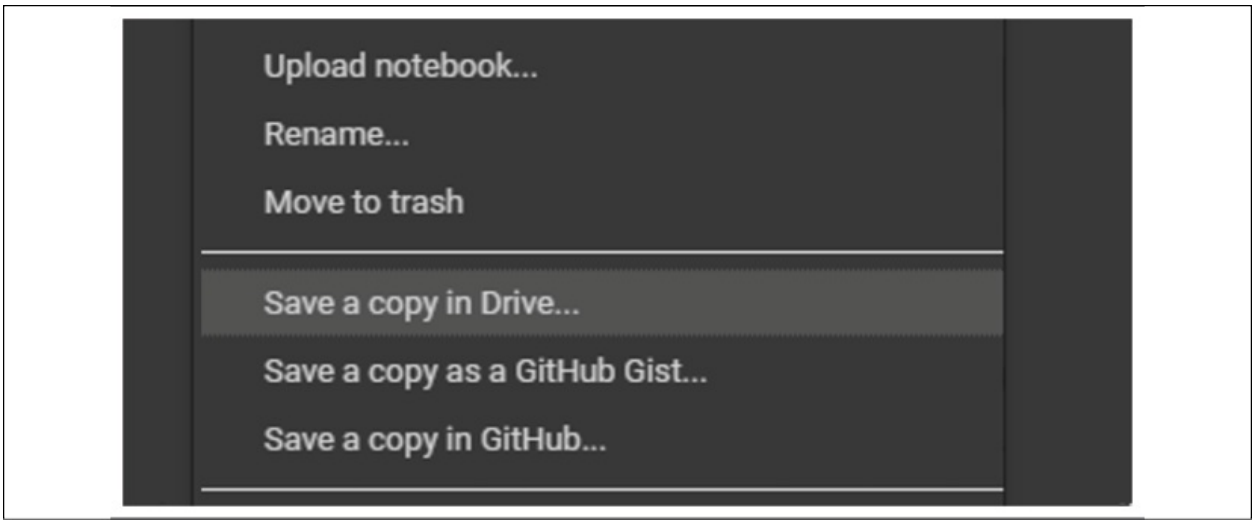


Figure 13.26: The File menu

Once the file is saved, you are ready to go!

You have many runtime options, such as making a choice between using a CPU or a GPU, display options (background, for example), and many more ways to use Google Colaboratory. I recommend reading the document to find the many options available.

You are on your free Colaboratory server, and you're ready to explore your notebook.

Defining and training the model

We will run the notebook and then analyze the results. This should provide an introduction to both Google Colaboratory and some TensorBoard functions.

First, run the program by clicking on the **Run all** option in the **Runtime** menu:

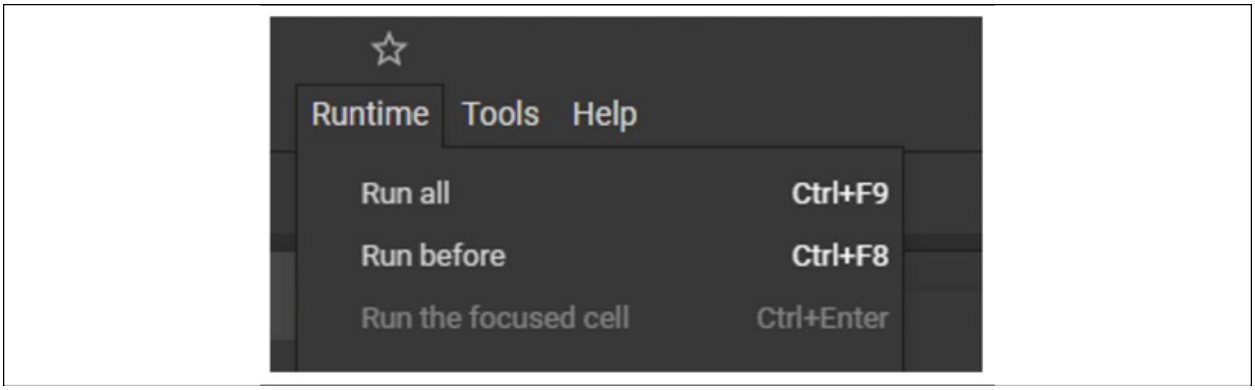


Figure 13.27: Runtime options

The program will then go through its cells and provide information on the training process. To obtain this information, we will explore some of TensorBoard's key functions.

We will first install TensorFlow and then run TensorBoard.

- **Installing TensorFlow and getting TensorBoard running**: As you saw in the previous section, you do not need to run the program cell by cell unless a cell contains an error. In that case, click on the run button in the cell:

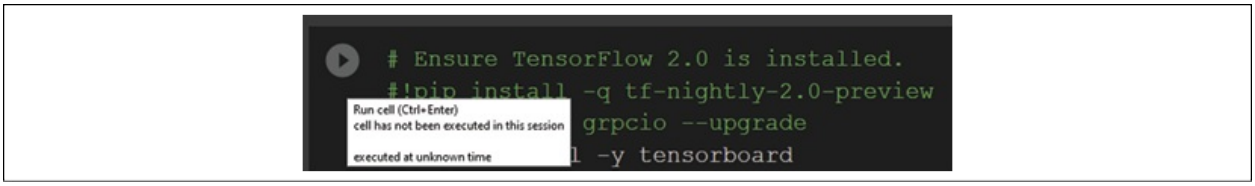


Figure 13.28: Running a cell

The cell will execute the code. In this case, it will install TensorFlow 2.x:

```
# Ensure TensorFlow 2.0 is installed.
!pip install -q tf-nightly-2.0-preview
# Load the TensorBoard notebook extension.
%load_ext tensorboard
```

Once the installation is finished and TensorBoard is loaded, the program runs through the headers to import the necessary modules.

Be careful with TensorBoard versions! You might have a previous or different version installed that you are using for another project. Before running this

program, check any application that is using TensorBoard in your environment. Check your configuration carefully before doing anything. If there is a risk, use another environment or just read the notebook without running it.

- **The program now defines a simplified model you are now familiar with**: The following model has been simplified. The goal here is to show how TensorBoard works. You can, of course, add more layers once you have explored the notebook:

```
# Define the model.
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(32, activation='relu'),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(10, activation='softmax')
])
```

- **The model is then compiled with an optimizer and accuracy metrics**: The model now needs to be compiled and run to provide measurement output:

```
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])
```

  For more on the Adam optimizer and cross-entropy, see *Chapter 9, Abstract Image Classification with Convolutional Neural Networks (CNNs)*.

The model is now trained and ready for metric callbacks for TensorBoard.

While the program was running the training, it was saving a log of the main functionalities to display. The graph of the model can be displayed in TensorBoard with one line along with many other functions.

```
%tensorboard --logdir logs
```

The following graph of the model contains many details:

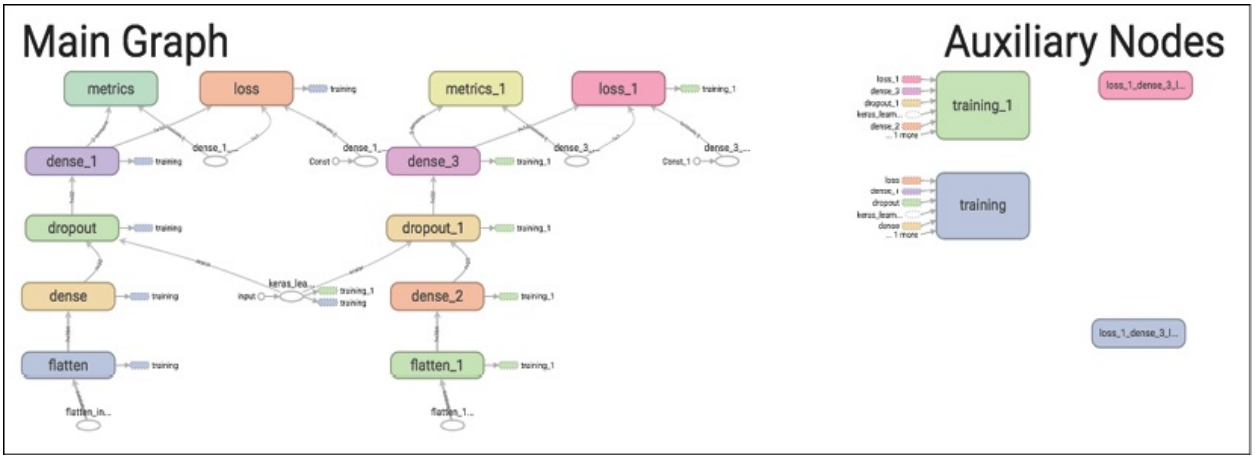


Figure 13.29: TensorFlow graph

If you want to have a simplified view, a conceptual view of the model is also displayed:



Figure 13.30: A partial view of the TensorFlow graph

We have explored the architecture of a neural network model using TensorBoard graphs. Let's see how to visualize the measurements of the training process of our model.

Introducing some of the measurements

While training, the program saved key information in its log directory that can now be displayed.

- **Epoch accuracy**: If the accuracy increases with the epochs, the classifier is progressing, and it is learning correctly. If it decreases, we are in trouble! We will have to go back and check the dataset, the activation functions, and how the layers were designed.

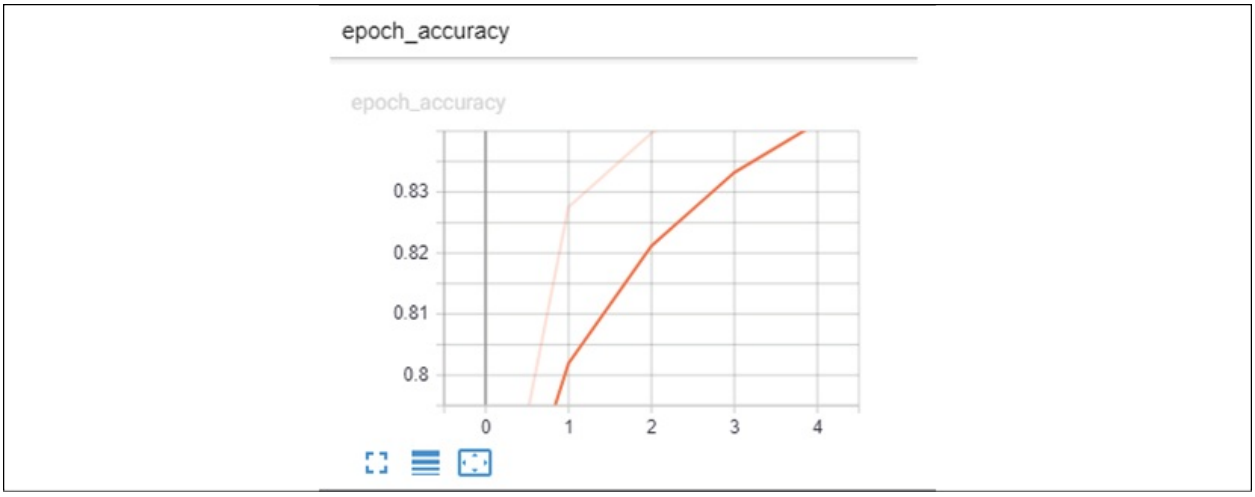


Figure 13.31: Accuracy of the model

- **Basic flow, including an activation function**: TensorBoard has drill-down functionality. You can drill down into the actual operations TensorFlow 2.x is calculating:

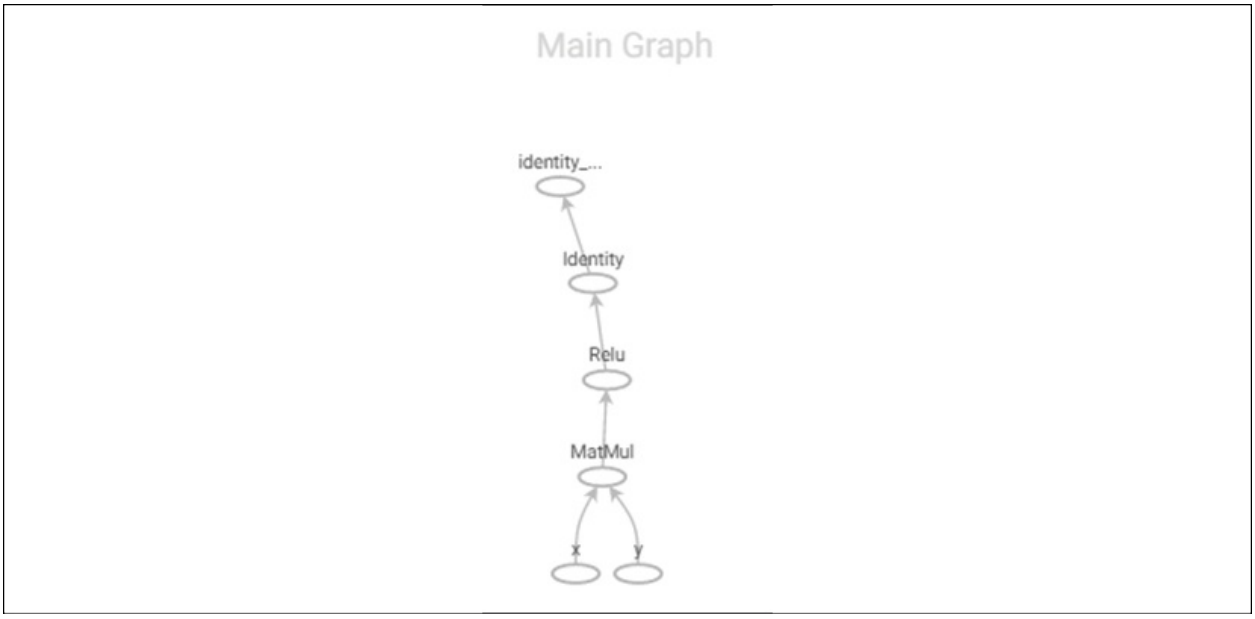


Figure 13.32: The activation function

- **Exploring the details of an activation function**: Once you have seen the flow of an operation, you can even peek inside it to see how it is built:

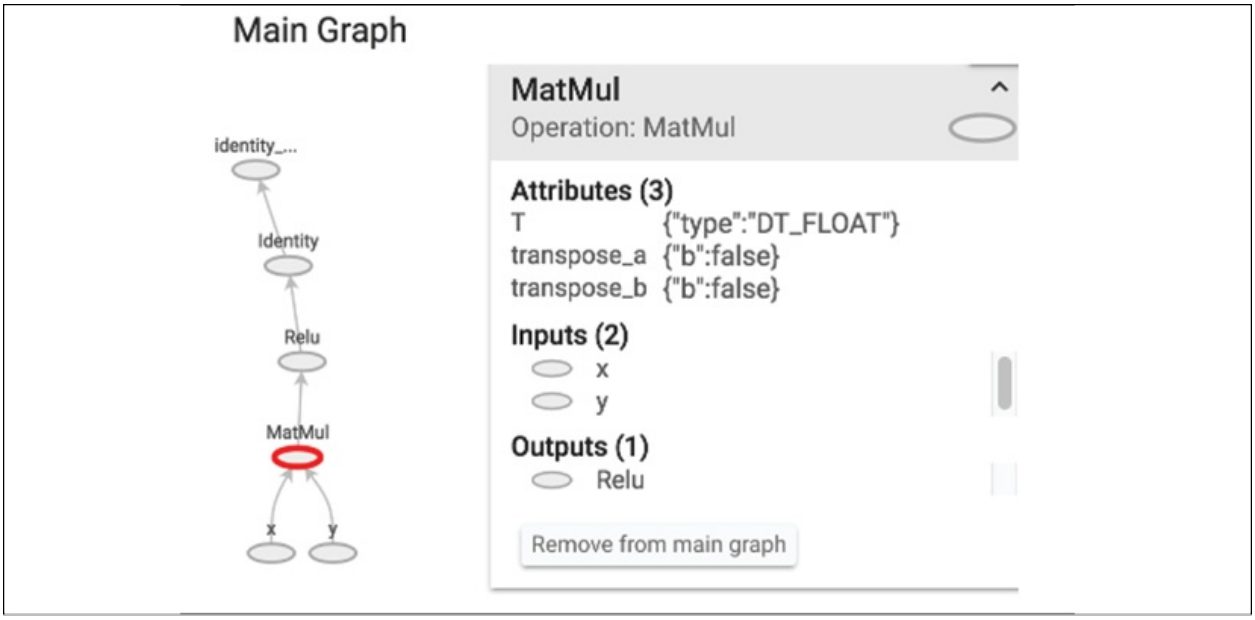


Figure 13.33: The activation function

These TensorBoard graphs and measurements help you dig into the mechanics of your model. They provide insights that will add to the ones you acquired when exploring the outputs of layers in the first section of this chapter.

In this section, we have explored the architecture functions of TensorBoard through the many graphs available as well as tools to measure the training performance of our model.

Summary

In this chapter, we explored deep learning from the inside. We saw that building a CNN is now easy with TensorFlow 2.x, but peeking inside the way it "thinks" gives critical insight.

We first built a CNN with many layers. The level of abstraction of a CNN increases through each layer. Reducing the number of dimensions per layer makes patterns appear. A neural network can be described as a process that goes

from chaos to meaning.

After building the CNN, we wrote a program that can read the "mental" images of the layers. The output of each layer shows how the network is creating patterns and structures. Since we humans often think using mental images, the output images of the CNN help us understand how a machine learns.

Finally, we used a Google Colaboratory server to visualize the measurements of the CNN's learning process with TensorBoard running on top of TensorFlow 2.x. Measuring the accuracy of the training process of a CNN is critical. Visualizing these measurements makes it easier to see what is going wrong. TensorBoard provides a graph of a model to help us go from source code to a mental representation of an ANN.

To sum this chapter up in a nutshell, we can say that artificial intelligence, through mathematics, transforms the chaos surrounding us into intelligible structures and patterns.

In the next chapter, we are going to go further and learn how to visualize another aspect of a neural network: weights. We are going to use the weights of a restricted Boltzmann machine (RBM) to create a visual representation in TensorBoard using principal component analysis.

Questions

1. A CNN always has the same number of layers. (Yes | No)
2. ReLU is the best activation function. (Yes | No)
3. It is not necessary to compile a sequential classifier. (Yes | No)
4. The output of a layer is best viewed without running a prediction. (Yes | No)
5. The names of the layers mean nothing when viewing their outputs. (Yes | No)
6. TensorFlow 2.x does not include Keras. (Yes | No)
7. Google Colaboratory is just a repository, like GitHub. (Yes | No)
8. Google Colaboratory cannot run notebooks. (Yes | No)

9. It is possible to run TensorBoard in Google Colaboratory notebooks. (Yes | No)
10. Accuracy is displayed in TensorBoard. (Yes | No)

Further reading

- For more information on activation functions, visit https://keras.io/activations/.
- Click on this link for more information on Google Colaboratory: https://colab.research.google.com/notebooks/welcome.ipynb#recent=true.

14

Preparing the Input of Chatbots with Restricted Boltzmann Machines (RBMs) and Principal Component Analysis (PCA)

In the following chapters, we will explore chatbot frameworks and build chatbots. You will find that creating a chatbot structure only takes a few clicks. However, no chatbot can be built without designing the input to prepare the desired dialog flow. The goal of this chapter is to demonstrate how to extract features from a dataset and then use them to gather the basic information to build a chatbot in *Chapter 15, Setting up a Cognitive NLP UI/CUI Chatbot*.

The input of a dialog requires in-depth research and designing. In this chapter, we will build a **restricted Boltzmann machine** (**RBM**) that will analyze a dataset. In *Chapter 13, Visualizing Networks with TensorFlow 2.x and TensorBoard*, we examined the layers of a convolutional neural network (CNN) and displayed their outputs. This time, we will explore the weights of the RBM. We will go further and use the weights of the RBM as features. The weights of an RBM can be transformed into feature vectors for a **principal component analysis** (**PCA**) algorithm.

We will use the feature vectors generated by the RBM to build a PCA display using TensorBoard Embedding Projector's functionality. We will then use the statistics obtained to lay the grounds for the inputs of a chatbot.

To illustrate the whole process, we will use streaming platform data as an example of how this is done. Streaming has become a central activity of almost all smartphone owners. The problem facing Netflix, YouTube, Amazon, or any platform offering streaming services is to offer us the right video to watch. If a

viewer watches a video, and the platform does not display a pertinent similar one to watch next, the viewer might choose to use another platform.

This chapter is divided into two parts:

- Building an RBM and then extending it to an automatic feature vector generator
- Using PCA to represent the weights of an RBM as features. TensorFlow's Embedding Projector possesses an inbuilt PCA function. The statistics produced will provide the basis of the dialog structure for *Chapter 15, Setting Up a Cognitive NLP UI/CUI Chatbot*.

Let's first define the basic terms we are using and our goals.

Defining basic terms and goals

The goal of this chapter is to prepare data to create the input of a chatbot we will build in *Chapter 15, Setting Up a Cognitive NLP UI/CUI Chatbot*.

Creating a chatbot requires preparation. We cannot just step into a project without a minimum amount of information. In our case, we will examine a dataset I created based on movie preferences. I did not choose to download huge datasets because we need to first focus on understanding the process and building a model using basic data.

The size of the datasets increase daily on an online platform. When we watch a movie on a streaming platform, on Netflix for example, we can like the movie or click on the thumbs-down button.

When we approve or disapprove of a movie on an online platform, our preferences are recorded. The features of these movies provide valuable information for the platform, which can then display choices we prefer: action, adventure, romantic, comedy, and more.

In this chapter, we will first use an RBM to extract a description (such as action, adventure, or comedy, for example) of the movies watched by a user or a group of users. We will take the output weights produced by the RBM to create a file of

features reflecting the user's preferences.

This file of features of a user's preferences can be considered as a "mental dataset" of a person. The name might seem strange at first. However, a "mental" representation of a person goes beyond the standard age, income, and other impersonal data. Features such as "love," "violence," and "horizons" (wider views, adventure) give us a deeper understanding of a person than information we can find on a driving license.

In the second part of the chapter, we will use the RBM's output of features of a person's "mind" as the input of a PCA. The PCA will calculate how the features relate to each other and how they vary, and we will display them in TensorBoard.

We will then actually *see* a representation of a person's mind through the key features drawn from the RBM. This information will then be used to help us create a customized chatbot in *Chapter 15*.

Let's move on to the first phase and build an RBM.

Introducing and building an RBM

RBMs are random and undirected graph models generally built with a visible and a hidden layer. They were used in a Netflix competition to predict future user behavior. The goal here is not to predict what a viewer will do but establish who the viewer is and store the data in a viewer's profile-structured mind dataset. The input data represents the features to be trained to learn about viewer X. Each column represents a feature of X's potential personality and tastes. Each line represents the features of a movie that X has watched. The following code (and this section) is in `RBM_01.py`:

```
np.array([[1,1,0,0,1,1],
          [1,1,0,1,1,0],
          [1,1,1,0,0,1],
          [1,1,0,1,1,0],
          [1,1,0,0,1,0],
          [1,1,1,0,1,0]])
```

The goal of this RBM is to define a profile of X by computing the features of the movies watched. The input data could also be images, words, and other forms of data, as in any neural network.

First, we will explore the architecture and define what an energy-driven neural network is. Then, we will build an RBM from scratch in Python.

The architecture of an RBM

The RBM model used contains two layers: a visible layer and a hidden layer. Many types of RBMs exist, but generally, they contain the following properties:

- There is no connection between the visible units, which is why it is *restricted*.
- There is no connection between the hidden units enforcing the restricted property of the network.
- There is no direction as in a feedforward neural network (FNN), as explored in *Chapter 8, Solving the XOR Problem with a Feedforward Neural Network*. An RBM's model is thus an *undirected* graph.
- The visible and hidden layers are connected by a weight matrix and a bias vector, which are the lines in the following diagram:

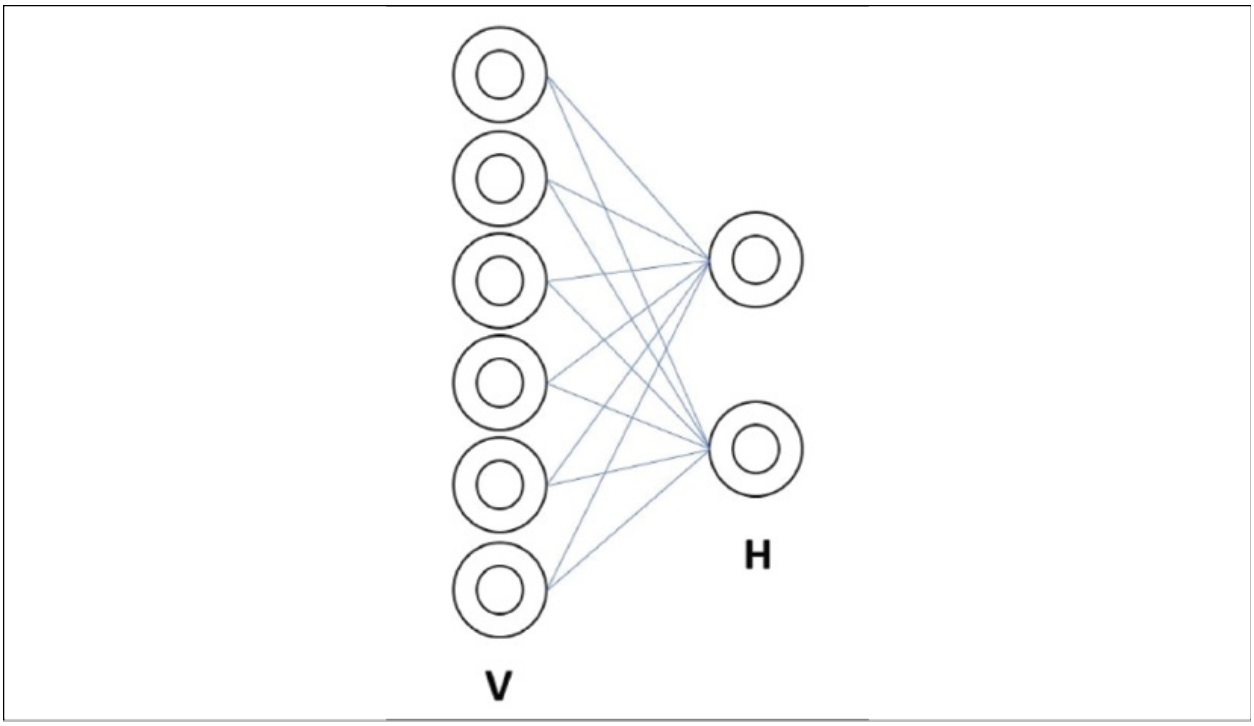


Figure 14.1: The connection between visible and hidden units

The network contains six visible and two hidden units, producing a weight matrix of 2×6 values to which we will add bias values.

You will note that there is no output. The system runs from the visible units to the hidden units and back. We are operating feature extraction with this type of network. In this chapter, for example, we will use the weights as features.

By forcing the network to represent its data contained in 6 units in 2 units through a weight matrix, the RBM creates feature representations. The hidden units, weights, and biases can be used for feature extraction.

An energy-based model

An RBM is an energy-based model. The higher the energy, the lower the probability of obtaining the correct information; the lower the energy, the higher the probability – in other words, the higher the accuracy.

To understand this, let's go back to the cup of tea we observed in *Chapter 1, Getting Started with Next-Generation Artificial Intelligence through*

Reinforcement Learning:



Figure 14.2: The complexity of a cup of tea

In *Chapter 1*, we observed a microstate of the cup through its global content and temperature. Then, we went on to use the Markov decision process (MDP) to run microstate calculations.

This time, we will focus on the temperature of the cup of tea. *x* will be the global temperature of all the molecules in the cup of tea:

- If $x = 1$, this means the temperature is very hot. The tea has just boiled.
- If $x = 0.5$, this means the temperature has gone down.
- If $x = 0.1$, this means the temperature is still a bit warm, but the tea is cooling.

The higher the temperature, the more the molecules will be bouncing around in the cup with a high level of energy, making it feel hot.

However, the hotter it is, the closer to very hot, the lower the probability we can drink it.

This leads to a probability *p* for a temperature *x*:

- $x \rightarrow 1$, $p \rightarrow 0$
- $x \rightarrow 0$, $p \rightarrow 1$

As you can see, in an energy-driven system, we will strive to lower the energy

level. Let's say we have a person with an unknown tolerance for hot drinks, and we want to wager whether they can drink our cup of tea. Nobody wants to drink cold (low-energy) tea, sure, but if our focus is on the likelihood of a person being able to drink the tea without finding it too hot (high-energy), then we want that tea to be as low-energy (that is, cool) as possible!

To illustrate the $p(x)$ system of our cup of tea, we will use Euler's number e, which is equal to 2.718281. $p(x)$ is the probability that we can drink our cup of tea, with p being the probability, and x the temperature or energy.

We will begin to introduce a simple energy function in which $p(x) = e^{(-x)}$:

- $p(e^{(-1)}) = 0.36$
- $p(e^{(-0.5)}) = 0.60$
- $p(e^{(-0.1)}) = 0.90$

You can see that as $-x$ (energy) decreases, the probability $p(x)$ increases.

The goal of the learning function of an RBM is to decrease the energy level by optimizing the weights and biases. By doing this, the RBM increases the probability that the hidden units, the weights, and biases are optimized.

To calculate the energy of an RBM, we will take the complete architecture of the network into account. Let's display our model again as follows:

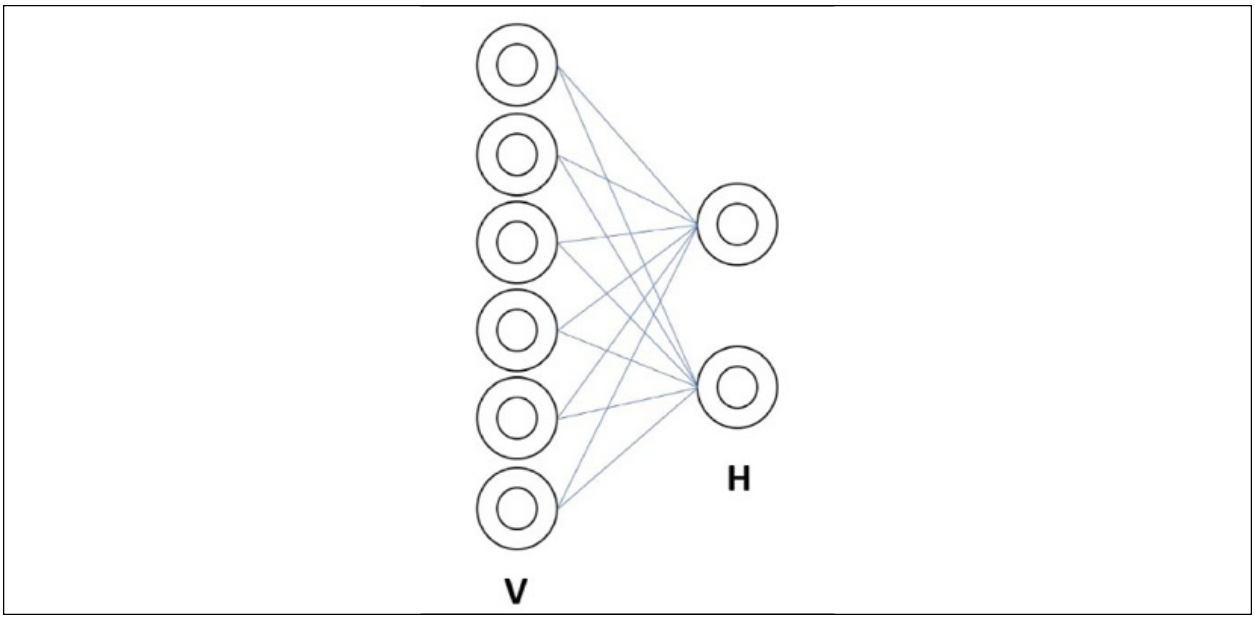


Figure 14.3: The connection between visible and hidden units

This RBM model contains the following values:

- $E(v, h)$, which is the energy function that takes the visible units (input data) and hidden units into account.
- v_i = the states of the visible units (input).
- a_i = the biases of the visible units.
- h_j = the states of the hidden units.
- b_j = the biases of the hidden units.
- w_{ij} = the weight matrix.

With these variables in mind, we can define the energy function of an RBM for $i \in \text{visible}$, $j \in \text{hidden}$, and *ij* as the lines and columns of the weight matrix as follows:

$$E(v,h) = -\sum a_i v_i - \sum b_j h_j - \sum v_i h_j \ w_{ij}$$

Now that we've got a better idea of what an RBM is and the principles behind it, let's start to consider how to build an RBM from scratch using Python.

Building the RBM in Python

We will build an RBM using `RBM_01.py` from scratch using our bare hands with no pre-built library. The idea is to understand an RBM from top to bottom to see how it ticks. We will explore more RBM theory as we build the machine.

Creating a class and the structure of the RBM

First, the RBM class is created:

```
class RBM:
    def __init__(self, num_visible, num_hidden):
        self.num_hidden = num_hidden
        self.num_visible = num_visible
```

The first function of the class will receive the number of hidden units (`2`) and the number of visible units (`6`).

The weight matrix is initialized with random weight values at line 20:

```
        np_rng = np.random.RandomState(1234)
        self.weights = np.asarray(np_rng.uniform(
            low=-0.1 * np.sqrt(6. / (num_hidden + num_visib
le)),
            high=0.1 * np.sqrt(6. / (num_hidden + num_visib
le)),
            size=(num_visible, num_hidden)))
```

The bias units will now be inserted in the first row and the first column at line 27:

```
        self.weights = np.insert(self.weights, 0, 0, axis =
0)
        self.weights = np.insert(self.weights, 0, 0, axis =
1)
```

The goal of this model will be to observe the behavior of the weights. Observing the weights will determine how to interpret the result in this model based on calculations between the visible and hidden units.

The first row and column are the biases, as shown in the preceding code snippets. Only the weights will be analyzed for the profiling functions. The weights and biases are now in place.

Creating a training function in the RBM class

On line 30, the training function is created:

```
    def train(self, data, max_epochs, learning_rate):
```

In this function:

- `self` is the class

- `data` is the 6×6 input array, containing 6 lines of movies and 6 columns of features of the movies:

```
np.array([[1,1,0,0,1,1],
          [1,1,0,1,1,0],
          [1,1,1,0,0,1],
          [1,1,0,1,1,0],
          [1,1,0,0,1,0],
          [1,1,1,0,1,0]])
```

  The RBM model in this chapter is using **visible binary units**, as shown in the input, which is the training data of this model. The RBM will use the input as training data.

  An RBM can contain other types of units: softmax units, Gaussian visible units, binomial units, rectified linear units, and more. Our model focuses on binary units.

- `max_epochs` is the number of epochs that the RBM will run to train.
- `learning_rate` is the learning rate that will be applied to the weight matrix containing the weights and the biases.

We will now insert bias units of `1` in the first column on line 35:

```
        data = np.insert(data, 0, 1, axis = 1)
```

There are other strategies to initialize biases. This is a trial-and-error process, depending on your project. In this case, bias units of `1` are sufficient to do the job.

Computing the hidden units in the training function

On line 37, we start training the RBM during `max_epochs` by computing the value of the hidden units:

```
        for epoch in range(max_epochs):
```

The first phase is to focus on the hidden units. We activate the probabilities of the hidden units with our weight matrix using dot matrix multiplication:

```
            pos_hidden_activations = np.dot(data, self.weig
hts)
```

Then, we apply the logistic function as we saw in *Chapter 2*, *Building a Reward Matrix – Designing Your Datasets*:

```
            pos_hidden_probs = self._logistic(
                pos_hidden_activations)
```

The logistic function called is on line 63:

```
    def _logistic(self, x):
        return 1.0 / (1 + np.exp(-x))
```

We set the biases to `1`:

```
            pos_hidden_probs[:,0] = 1 # Fix the bias unit.
```

We now have computed the first epoch of the probabilities of the hidden states with random weights.

Random sampling of the hidden units for the reconstruction and contractive divergence

There are many sampling methods, such as Gibbs sampling, for example, which has a randomized approach to avoid deterministic samples.

In this model, we will choose a random sample that chooses the values of the hidden probabilities that exceed the values of a random sample of values. The `random.rand` function creates a random matrix with values between `0` and `1`, with a size of `num_examples` × `self.num_hidden+1`:

```
            pos_hidden_states = pos_hidden_probs >
                np.random.rand(num_examples, self.num_hidde
n + 1)
```

This sample will be used for the **reconstruction** phase we will explore in the next section.

We also need to compute an association for the **contrastive divergence** (the function used to update the weight matrix) phase, which is explained hereinunder:

```
            pos_associations = np.dot(data.T, pos_hidden_pr
obs)
```

This dot product of the visible data units *v* × the hidden units *h* can be represented as follows:

$$v_i h_j \text{ data}$$

Now that the dot product has been implemented, we will build the reconstruction phase.

Reconstruction

An RBM uses its input data as its training data, computes the hidden weights using a random weight matrix, and then *reconstructs* the visible units. Instead of an output layer as in other neural networks, an RBM reconstructs the visible units and compares them to the original data.

The following code applies the same approach as for the hidden units described previously to generate visible units:

```
            neg_visible_activations = np.dot(pos_hidden_sta
tes,
                self.weights.T)
            neg_visible_probs = self._logistic(
                neg_visible_activations)
            neg_visible_probs[:,0] = 1 # Fix the bias unit
```

These negative visible units will be used to evaluate the error level of the RBM, as explained here.

Now that we have generated visible units with our sample of hidden unit states, we move on and generate the corresponding hidden states:

```
            neg_hidden_activations = np.dot(neg_visible_pro
```

```
bs,
                self.weights)
            neg_hidden_probs = self._logistic(
                neg_hidden_activations)
            neg_associations = np.dot(neg_visible_probs.T,
                neg_hidden_probs)
```

Note that `neg_associations` can be represented in the following form:

$$v_i h_j \text{ reconst}$$

Here, we have done the following:

- Computed positive hidden states using the visible units containing the training data
- Selected a random sample of those positive hidden states
- Reconstructed negative (generated from the hidden states, not the data) visible states
- And, in turn, generated hidden states from the visible states produced

We have *reconstructed* visible states through this process. However, we need to evaluate the result and update the weights.

Contrastive divergence

To update the weights, we do not use gradient descent. In this energy model, we use contrastive divergence, which can be expressed as follows:

$$\varepsilon\left(v_i h_j \text{ data} - v_i h_j \text{ reconst}\right)$$

The letter ε is the learning rate. The learning rate should be a small value and can be optimized throughout the training process. I applied a small value, 0.001 overall.

The source code for updating the weights is as follows:

```
            self.weights += learning_rate * ((pos_associati
ons -
                neg_associations))
```

Over the epochs, the weights will adjust, bringing the energy and error level down and, hence, bringing the accuracy of the probabilities up.

At this point, we will display the error level and the energy value of the RBM throughout the epochs.

Error and energy function

On line 56, the error function calculates the squared sum of the difference between the visible units provided by the data and the reconstructed visible units:

```
error = np.sum((data - neg_visible_probs) ** 2)
```

For the energy function, we can use our original energy equation:

$$E(v,h) = -\sum a_i v_i - \sum b_j h_j - \sum v_i h_j \ w_{ij}$$

In our code, we will not use the biases since we often set them to `1`.

We will also need a function to measure the evolution of the energy of the RBM.

The energy will be measured with a probabilistic function *p*:

$$p(x) = \frac{e^{-E(x)}}{Z}$$

Z is a **partition function** for making sure that the sum of the probabilities of each *x* input does not exceed 1:

$$\sum_x p(x) = 1$$

The partition function is the sum of all the individual probabilities of each *x*:

$$Z = \sum_x e^{-E(x)}$$

The corresponding code will calculate the energy of the RBM, which will decrease over time as the RBM goes through the epochs:

```
            energy=-np.sum(data) - np.sum(neg_hidden_probs)
-
                np.sum(pos_associations * self.weights)
            z=np.sum(data)+np.sum(neg_hidden_probs)
            if z>0: energy=np.exp(-energy)/z;
```

You will note that neither the error function nor the energy function influences the training process. The training process is based on contrastive divergence.

The error and energy values will measure the efficiency of the model by providing some insight into the behavior of the RBM as it trains.

Here is an example of these measurement values at the beginning of the process and at the end:

```
Epoch 0: error is 8.936507744240409  Energy: 15861064300520
73.0
...
Epoch 4999: error is 4.498343290467705  Energy: 2.426792619
597097e+46
```

At epoch 0, the error is high, and the energy is high, too.

At epoch 4999, the error is sufficiently low for the model to produce correct feature extraction values. The energy has significantly diminished.

Running the epochs and analyzing the results

Once the RBM has optimized the weight-bias matrix for *n* epochs, the matrix will provide the following information for the profiler system of person X, for example:

```
[[ 0.91393138 -0.06594172 -1.1465728 ]
[ 3.01088157 1.71400554 0.57620638]
[ 2.9878015 1.73764972 0.58420333]
[ 0.96733669 0.09742497 -3.26198615]
```

```
[-1.09339128 -1.21252634 2.19432393]
 0.19740106 0.30175338 2.59991769]
 0.99232358 -0.04781768 -3.00195143]]
```

An RBM model uses random values and will produce slightly different results each time it is trained.

The RBM will train the input and display the features added to X's profile.

The weights of the features have been trained for person X. The first line is the bias and examines columns 2 and 3. The following six lines are the weights of X's features:

```
[[ 0.913269 -0.06843517 -1.13654324]
 3.00969897 1.70999493 0.58441134]
 2.98644016 1.73355337 0.59234319]
 0.953465 0.08329804 -3.26016158]
[-1.10051951 -1.2227973 2.21361701]
 0.20618461 0.30940653 2.59980058]
 0.98040128 -0.06023325 -3.00127746]]
```

The weights (in bold) are lines 2 to 6 and columns 2 to 3. The first line and first column are the biases.

The way to interpret the weights of an RBM remains a careful strategy to build. In this case, a creative approach is experimented with to determine marketing behavior. There are many other uses of an RBM, such as image processing, for example. In this case, the weight matrix will provide a profile of X by summing the weight lines of the feature, as shown in the following code:

```
    for w in range(7):
        if(w>0):
            W=print(F[w-1],":",r.weights[w,1]+r.weights[w,2
])
```

The features are now labeled, as displayed in this output:

```
love : 2.25265339223
happiness : 2.28398311347
```

```
family : -3.16621250031
horizons : 0.946830830963
action : 2.88757989766
violence : -3.05188501936
A value>0 is positive, close to 0 slightly positive
A value<0 is negative, close to 0 slightly negative
```

We can see that beyond standard movie classifications, X likes horizons somewhat, does not like violence, and likes action. X finds happiness and love important, but not family at this point.

The RBM has provided a personal profile of X—not a prediction, but getting ready for a suggestion through a chatbot or just building X's machine mind-dataset.

We have taken a dataset and extracted the main features from it using an RBM. The next step will be to use the weights as feature vectors for PCA.

Using the weights of an RBM as feature vectors for PCA

In this section, we will be writing an enhanced version of `RBM_01.py`. `RBM_01.py` produces the feature vector of one viewer named X. The goal now is to extract the features of 12,000 viewers, for example, to have a sufficient number of feature vectors for PCA.

In `RBM_01.py`, viewer X's favorite movies were first provided in a matrix. The goal now is to produce a random sample of 12,000 viewer vectors.

The first task at hand is to create an RBM launcher to run the RBM 12,000 times to simulate a random choice of viewers and their favorite movies, which are the ones the viewer liked. Then, the feature vector of each viewer will be stored.

`RBM_launcher.py` first imports RBM as `rp`:

```
import RBM as rp
```

The primary goal of `RBM_launcher.py` is to carry out the basic functions to run RBM. Once `RBM` is imported, the feature vector's `.tsv` file is created:

```
#Create feature files
f=open("features.tsv","w+")
f.close
```

When `rp`, the `RBM` function imported as `rp`, is called, it will append the feature file.

The next step is to create the label file containing the metadata:

```
g=("viewer_name"+"\t"+"primary_emotion"+"\t"+"secondary_emo
tion"+
    "\n")
with open("labels.tsv", "w") as f:
    f.write(g)
```

You will notice the use of the word "emotion." In this context, "emotion" refers to features for sentiment analysis in general, not human emotions in particular. Please read "emotions" in this context as sentiment analysis features.

Now, we are ready to run RBM 12,000+ times, for example:

```
#Run the RBM feature detection program over v viewers
print("RBM start")
vn=12001
c=0
for v in range (0,vn):
    rp.main()
    c+=1
    if(c==1000):print(v+1);c=0;
print("RBM over")
```

`rp.main()` calls the `main()` function in `RBM.py` that we will now enhance for this process.

We will enhance `RBM_01.py` step-by-step in another file named `RBM.py`. We will adapt the code starting line 65 to create an RBM launcher option:

A variable name `pt` is set to `0` or `1`, depending on whether we wish to display

intermediate information:

```
    # RBM_launcher option
    pt=0  #restricted printing(0), printing(1)
```

Since this is an automatic process, `pt` is set to `0`.

The metadata of 10 movies is stored in `titles`:

```
    # Part I Feature extractions from data sources
    # The titles of 10 movies
    titles=["24H in Kamba","Lost","Cube Adventures",
            "A Holiday","Jonathan Brooks",
             "The Melbourne File", "WNC Detectives",
             "Stars","Space L","Zone 77"]
```

A feature matrix of movies with six features per movie is created starting at line 71, with the same features as in `RBM_01.py`:

```
    # The feature map of each of the 10 movies. Each line is
 a movie.
    # Each column is a feature. There are 6 features: ['lov
e', 'happiness', 'family', 'horizons', 'action', 'violence'
]
    # 1= the feature is activated, 0= the feature is not ac
tivated
    movies_feature_map = np.array([[1,1,0,0,1,1],
                                   [1,1,0,1,1,1],
                                   [1,0,0,0,0,1],
                                   [1,1,0,1,1,1],
                                   [1,0,0,0,1,1],
                                   [1,1,0,1,1,0],
                                   [1,0,0,0,0,0],
                                   [1,1,0,1,1,0],
                                   [1,1,0,0,0,1],
                                   [1,0,0,1,1,1],
                                   [1,1,0,0,1,0],
                                   [1,1,0,1,1,1],
                                   [1,1,0,0,1,1]])
```

Each line of the matrix contains a movie, and each column one of the six features of that movie. If the value is `0`, the feature is not present; if the value is

`1`, the feature is present.

In the years to come, the number of features per movie will be extended to an indefinite number of features per movie to fine-tune our preferences.

An empty output matrix is created. In `RBM_01.py`, the result was provided. In this example, it will be filled with random choices:

```
    #The output matrix is empty before the beginning of the
 analysis
    #The program will take the user "likes" of 6 out of the
10 movies

    dialog_output = np.array([[0,0,0,0,0,0],
                              [0,0,0,0,0,0],
                              [0,0,0,0,0,0],
                              [0,0,0,0,0,0],
                              [0,0,0,0,0,0],
                              [0,0,0,0,0,0]])
```

Now, the random movie selector will generate likes or dislikes per movie and per viewer:

```
    #An extraction of viewer's first 6 liked 6 movies out n
 choices
    #Hundreds of movies can be added. No dialog is needed s
ince a cloud streaming services stores the movie-likes we c
lick on
    mc=0   #Number of choices limited to 6 in this example
    a="no" #default input value if rd==1
    #for m in range(0,10):
    if pt==1:print("Movie likes:");
    while mc<6:
        m=randint(0,9)# filter a chosen movie or allow (thi
s case) a person can watch and like a movie twice=an indica
tion
        b=randint(0,1)# a person can like(dislike) a movie
the first time and not the second(or more) time
        if mc<6 and (a=="yes" or b==1):
            if pt==1:print("title likes: ",titles[m]);
            for i in range(0,6):dialog_output[mc,i]=
                movies_feature_map[m,i];
            mc+=1
```

```
        if mc>=6:
            break
```

We can choose whether to display the input:

```
    #The dialog_input is now complete
    if pt==1:print("dialog output",dialog_output);
```

The dialog output is the data collected by the platform through its like/dislike interface. The RBM runs its training session:

```
    #dialog_output= the training data
    training_data=dialog_output
    r = RBM(num_visible = 6, num_hidden = 2)
    max_epochs=5000
    learning_rate=0.001
    r.train(training_data, max_epochs,learning_rate)
```

The results of the RBM training session are now processed from line 185 to line 239 to transform the weights obtained into feature vectors and the corresponding metadata:

```
###Processing the results
    # feature labels
    F=["love","happiness","family","horizons","action","vio
lence"]
    .../...
        control=[0,0,0,0,0,0]
        for j in range(0,6):
            for i in range(0,6):
                control[i]+=dialog_output[j][i]
    ###End of processing the results
```

The goal is now to select the primary feature of a given movie chosen by a given viewer. This feature could be "love" or "violence" for example:

```
    #Selection of the primary feature
    for w in range(1,7):
        if(w>0):
            if pt==1:print(F[w-1],":",r.weights[w,0]);
            tw=r.weights[w,0]+pos
```

```
            if(tw>best1):
                f1=w-1
                best1=tw
            f.write(str(r.weights[w,0]+pos)+"\t")
    f.write("\n")
    f.close()
```

The secondary feature is also very interesting. It often provides more information than the primary feature. A viewer will tend to view a certain type of movie. However, the secondary features vary from movie to movie. For example, suppose a young viewer likes action movies. "Violence" could be the primary feature, but the secondary feature could be "love" in one case or "family" in another. The secondary featured is stored in the feature vector of this viewer:

```
    #secondary feature
    best2=-1000
    for w in range(1,7):
        if(w>0):
            tw=r.weights[w,0]+pos
            if(tw>best2 and w-1!=f1):
                f2=w-1
                best2=tw
```

The metadata is saved in the label file:

```
    #saving the metadata with the labels
    u=randint(1,10000)
    vname="viewer_"+str(u)
    if(pt==1):
        print("Control",control)
        print("Principal Features: ",vname,f1,f2,"control")

    f= open("labels.tsv","a")
    f.write(vname +"\t"+F[f1]+"\t"+F[f2]+"\t")
    f.write("\n")
    f.close()
```

This process will be repeated 12,000 times in this example.

The feature vector `features.tsv` file has been created:

```
13.426689288298416  11.753851952385375  7.313058537018188  9.270388600432321  10.317358081077947  12.518715620254277
14.16731308854518   10.76153264513799   6.720490965975493  11.094379311004566 11.073333294741081  11.50275895033197
12.484962258275809  10.716284246062651  7.768839107806554  12.46876514426894  12.422882161135337  10.408838835762321
12.832638922337567  9.964786691562942   7.692608889397371  10.563485187444483 10.516313781619747  12.83009727063435
14.003492968242266  10.968255799857026  7.078203153401292  11.172251380009826 11.090128581637876  7.755757145722892
14.908089834636389  11.69650900030709   6.580785728971158  9.42014852736986   9.384297565277508   10.092983647162638
14.261175650618203  10.163419999927958  6.813083646335597  11.408851103312355 11.332996671839853  9.86964395211538
```

Figure 14.4: The feature vector file

The feature vector `labels.tsv` metadata file matches the feature vector file:

```
viewer_name    primary_emotion    secondary_emotion
viewer_8481    love               violence
viewer_3568    love               violence
viewer_8667    love               horizons
viewer_2730    love               violence
viewer_3970    love               horizons
viewer_1140    love               happiness
```

You will note that "love" and "violence" appear often. This comes from the way I built the dataset based mostly on movies that contain action and some form of warm relationship between the characters, which is typical in movies for the younger generations.

Now that the feature vectors and the metadata file have been created, we can use PCA to represent the points.

Understanding PCA

PCA is applied very efficiently to marketing by Facebook, Amazon, Google, Microsoft, IBM, and many other corporations, among other feature processing algorithms.

Probabilistic machine learning training remains efficient when targeting apparel, food, books, music, travel, cars, and other market consumer segments.

However, humans are not just consumers; they are human beings. When they contact websites or call centers, standard answers or stereotyped emotional tone analysis approaches can depend on one's nerves. When humans are in contact with doctors, lawyers, and other professional services, a touch of humanity is necessary if major personal crises occur.

The goal of the PCA, in this context, is to extract key features to describe an individual or a population. The PCA phase will help us build a mental representation of X's profile, either to communicate with X or use X's mind as a powerful, *mindful* chatbot or decision-maker.

PCA isn't a simple concept, so let's take some time to understand it properly. We'll start with an intuitive explanation, and after that, we'll get into the mathematics behind it.

PCA takes data and represents it at a higher level.

For example, imagine you are in your bedroom. You have some books, magazines, and music (maybe on your smartphone) around the room. If you consider your room as a 3D Cartesian coordinate system, the objects in your room are all in specific *x*, *y*, *z* coordinates.

For experimentation purposes, take your favorite objects and put them on your bed. Put the objects you like the most near one another, and your second choices a bit further away. If you imagine your bed as a 2D Cartesian space, you have just made your objects change dimensions. You have brought the objects that you value the most into a higher dimension. They are now more visible than the ones that have less value for you.

They are not in their usual place anymore; they are on your bed and at specific coordinates depending on your taste.

That is the philosophy of PCA. If the number of data points in the dataset is very large, the PCA of a "mental dataset" of one person will always be different from the PCA representation of another person, like DNA. A "mental dataset" is a collection of thoughts, images, words, and feelings of a given person. It is more than the classic age, gender, income, and other neutral features. A "mental dataset" will take us inside somebody's mind.

> That is what a conceptual representation learning metamodel (CRLMM) is about as applied to a person's mental representation. Each person is different, and each person deserves a customized chatbot or bot treatment.

Mathematical explanation

The main steps in calculating PCA are important for understanding how to go from the intuitive approach to how TensorBoard Embedding Projector represents datasets using PCA.

Variance

Variance is when a value changes. For example, as the sun rises in summer, the temperature gets warmer and warmer. The variance is represented by the difference between the temperature at a given hour and then the temperature a few hours later. Covariance is when two variables change together. For example, the hotter it gets when we are outside, the more we will sweat to cool our bodies down.

- **Step 1**: Calculate the mean of the array `data1`. You can check this with `mathfunction.py`, as shown in the following function:

```
data1 = [1, 2, 3, 4]
M1=statistics.mean(data1)
print("Mean data1",M1)
```

  The answer is 2.5. The mean is not the median (the middle value of an array).

- **Step 2**: Calculate the mean of array `data2`. The mean calculation is executed with the following standard function:

```
data2 = [1, 2, 3, 5]
M2=statistics.mean(data2)
print("Mean data2",M2)
```

  The answer is $\bar{X} = 2.75$. The bar above the *X* signifies that it is a mean.

- **Step 3**: Calculate the variance using the following equation:

$$\text{var} = \frac{\sum_{x=1}^{x=n}(X - \bar{X})^2}{n}$$

Now, NumPy will calculate the variance with the absolute value of each *x* minus the mean, sum them up, and divide the sum by *n*, as shown in the following code snippet:

```
#var = mean(abs(x - x.mean())**2).
print("Variance 1", np.var(data1))
print("Variance 2", np.var(data2))
```

Some variances are calculated with $n - 1$ depending on the population of the dataset.

The result of the program for variances is as displayed in the following output:

```
Mean data1 2.5
Mean data2 2.75
Variance 1 1.25
Variance 2 2.1875
```

We can already see that `data2` varies a lot more than `data1`. Do they fit together? Are their variances close or not? Do they vary in the same way? Our goal in the following section is to find out whether two words, for example, will often be found together or close to one another, taking the output of the embedding program into account.

Covariance

The covariance will tell us whether these datasets vary together or not. The equation follows the same philosophy as variance, but now both variances are joined to see whether they belong together:

$$\operatorname{cov}(x, y) = \frac{\sum_{x=1}^{x=n}(X - \bar{X})(Y - \bar{Y})}{n}$$

As with the variance, the denominator can be $n - 1$ depending on your model. Also, in this equation, the numerator is expanded to visualize the co-part of covariance, as implemented in the following array in `mathfunction.py`:

```
x=np.array([[1, 2, 3, 4],
            [1, 2, 3, 5]])
a=np.cov(x)
print(a)
```

NumPy's output is a covariance matrix, `a`:

```
[[1.66666667 2.16666667]
 [2.16666667 2.91666667]]
```

If you increase some of the values of the dataset, it will increase the value of the parts of the matrix. If you decrease some of the values of the dataset, the elements of the covariance matrix will decrease.

Looking at some of the elements of the matrix increase or decrease that way takes time and observation. What if we could find one or two values that would give us that information?

Eigenvalues and eigenvectors

To make sense of the covariance matrix, the eigenvector will point to the direction in which the covariances are going. The eigenvalues will express the magnitude or importance of a given feature.

To sum it up, an eigenvector will provide the direction and the eigenvalue of the importance for the covariance matrix, `a`. With those results, we will be able to represent the PCA with TensorBoard Embedding Projector in a multidimensional space.

Let `w` be an eigenvalue(s) of `a`. An eigenvalue(s) must satisfy the following equation:

```
dot(a,v)=w * v
```

There must exist a vector, `v`, for which `dot(a,v)` is the same as `w*v`.

NumPy will do the math through the following function:

```
from numpy import linalg as LA
w, v = LA.eigh(a)
print("eigenvalue(s)",w)
```

The eigenvalues are displayed (in ascending order) in the following output:

```
eigenvalue(s) [0.03665681 4.54667652]
```

Now, we need the eigenvectors to see in which direction these values should be applied. NumPy provides a function to calculate both the eigenvalues and eigenvectors together. That is because eigenvectors are calculated using the eigenvalues of a matrix, as shown in this code snippet:

```
from numpy import linalg as LA
w, v = LA.eigh(a)
print("eigenvalue(s)",w)
print("eigenvector(s)",v)
```

The output of the program is as follows:

```
eigenvector(s) [[-0.79911221  0.6011819 ]
 [ 0.6011819   0.79911221]]
```

Eigenvalues come in a 1D array with the eigenvalues of `a`.

Eigenvectors come in a 2D square array with the corresponding value (for each eigenvalue) in columns.

Creating the feature vector

The remaining step is to sort the eigenvalues from the highest to the lowest value. The highest eigenvalue will provide the principal component (most important). The eigenvector that goes with it will be its feature vector. You can choose to ignore the lowest values or features. In the dataset, there will be hundreds, and often thousands, of features to represent. Now we have the feature vector:

$$\text{feature vector} = \text{FV} = \{\text{eigenvector}_1, \text{eigenvector}_2 \ldots n\}$$

n means that there could be many more features to transform into a PCA feature vector.

Deriving the dataset

The final step is to transpose the feature vector and original dataset and multiply the row feature vector by row data:

Data that will be displayed = row of feature vector * row of data

Summing it up

The highest value of eigenvalues is the principal component. The eigenvector will determine in which direction the data points will be oriented when multiplied by that vector.

Using TensorFlow's Embedding Projector to represent PCA

TensorBoard Embedding Projector offers an in-built PCA function that can be rapidly configured to fit our needs. TensorBoard can be called as a separate program or embedded in a program as we saw in *Chapter 13, Visualizing Networks with TensorFlow 2.x and TensorBoard*.

We will then extract key information on the viewer marketing segment that will be used to start building a chatbot in *Chapter 15, Setting Up a Cognitive NLP UI/CUI Chatbot*.

First, go to this link: `https://projector.tensorflow.org/`

For the following functions, bear in mind that TensorBoard Embedding Projector is working at each step and that it might take some time depending on your machine. We load the data produced by `RBM_launcher.py` and `RBM.py` by clicking on **Load**:

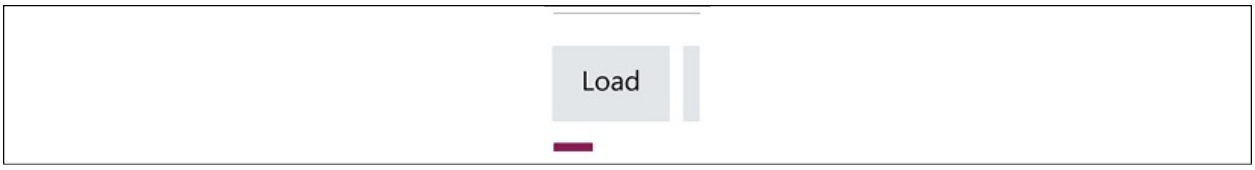


Figure 14.5: The Load button

Once the **Load data from your computer** windows appear, we load the feature vector `features.tsv` file by clicking on **Choose file**:

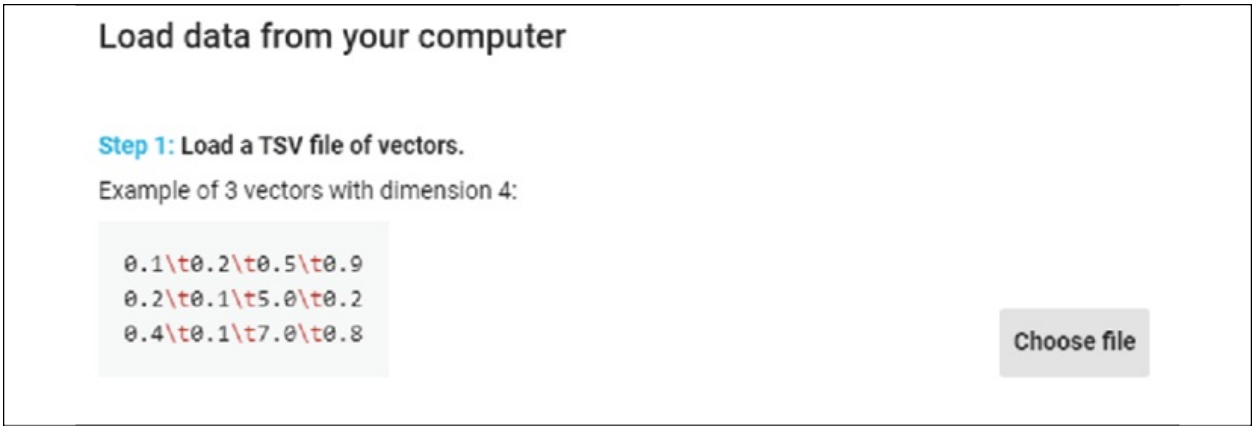


Figure 14.6: Loading the feature vector file

We load the `labels.tsv` metadata file by clicking on **Choose file** in **Step 2 (optional)**:

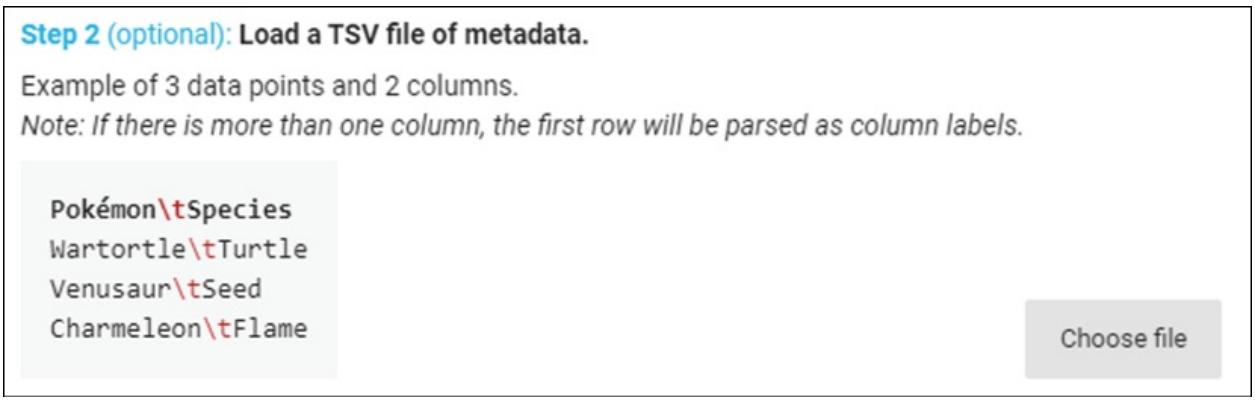


Figure 14.7: Loading a TSV file

To obtain a good representation of our 12,000+ features, click on **Sphereize data,** which is not checked in default mode:

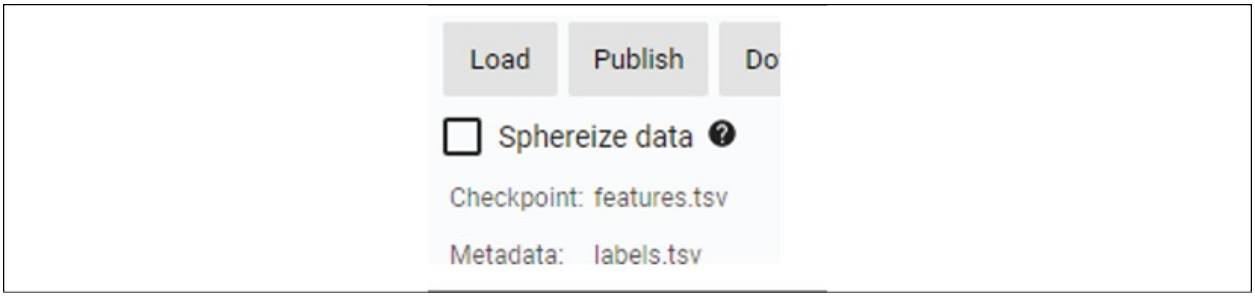


Figure 14.8: Sphereizing data

We now choose label by **secondary_emotion**, color by **secondary_emotion**,

along with edit by **secondary_emotion**:

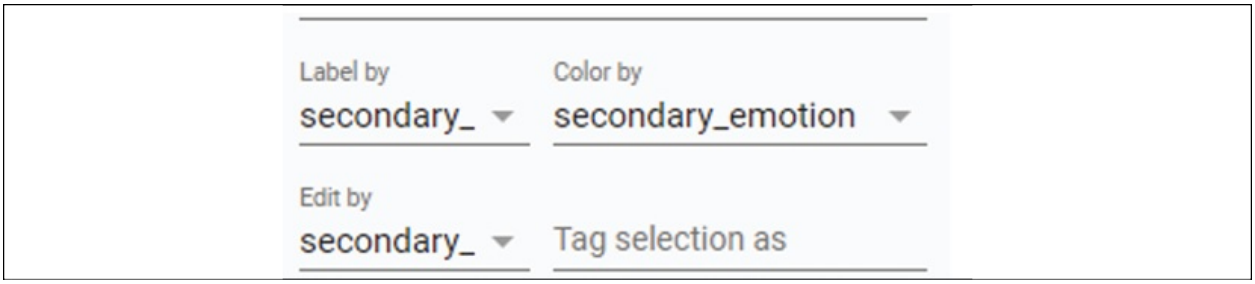


Figure 14.9: Managing labels

To get a nice view of the data, we activate night mode so that the moon should be active:



Figure 14.10: Activating the night mode

At this point, we have a nice PCA representation that turns like Earth, the ideas in our mind, or the minds of all of the viewers we are analyzing depending on how we use the features. The dots on the image are datapoints representing the features we calculated with an RBM and then represented with an image using PCA. It is like peeking inside the mind:

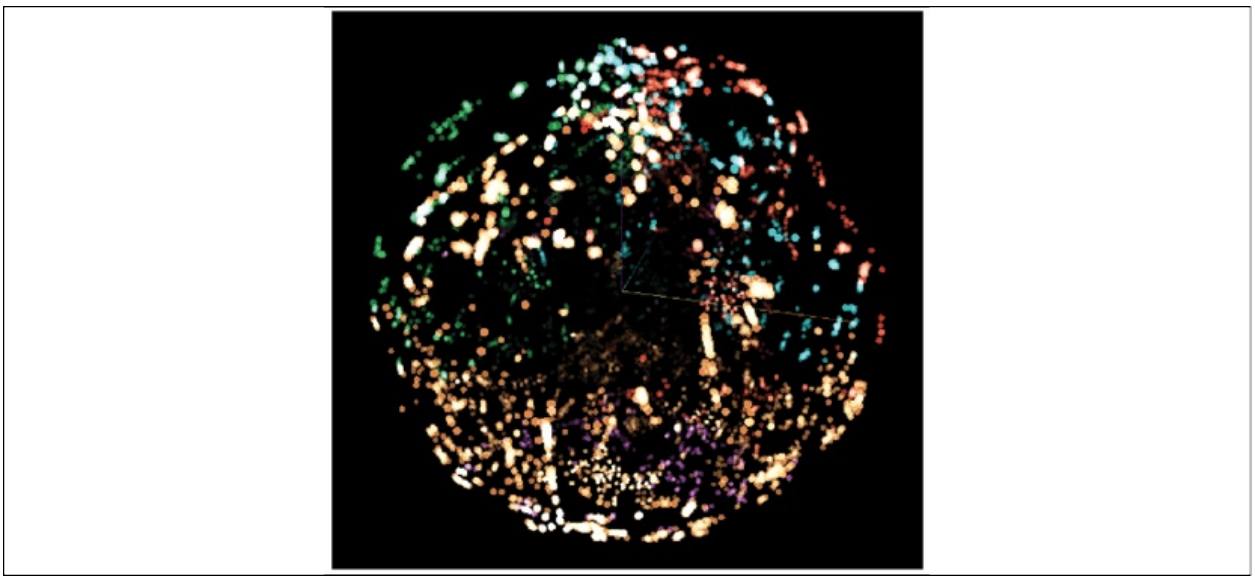


Figure 14.11: A PCA representation of features

The PCA representation of the features of a given person or a group of people provides vital information to create dialogs in a chatbot. Let's analyze the PCA to prepare data for a chatbot.

Analyzing the PCA to obtain input entry points for a chatbot

The goal is to gather some information to get started with our cognitive chatbot. We will use the filters provided by TensorBoard.

Choose **secondary_emotion** as the basis of our filters:

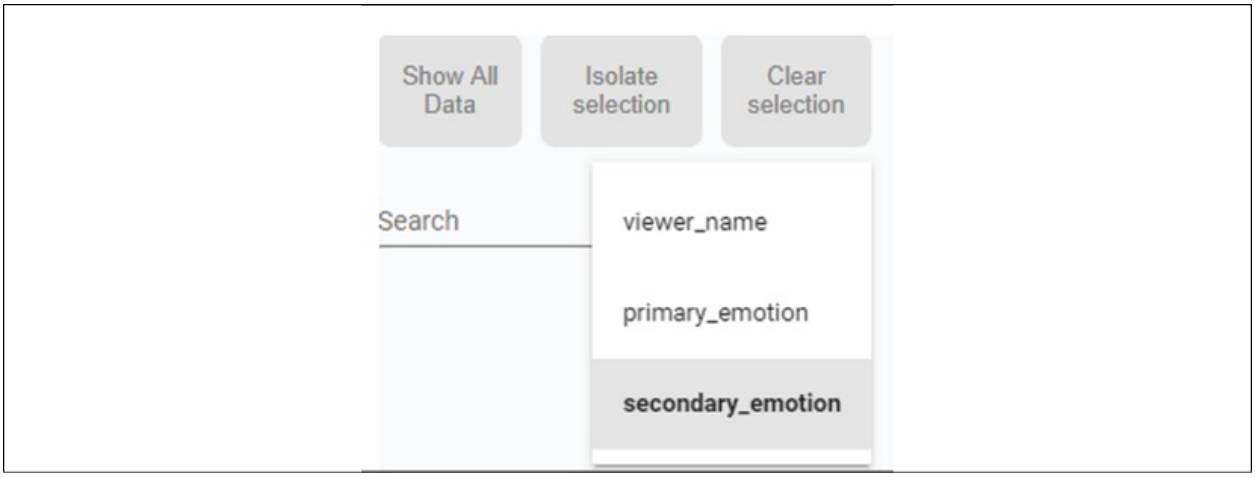


Figure 14.12: Filtering data

The features we are analyzing are as follows:

```
F=["love","happiness","family","horizons","action","violenc
e"]
```

We need to see the statistics per feature.

We type the feature in TensorBoard's search option, such as "love," for example, and then we click the down arrow:

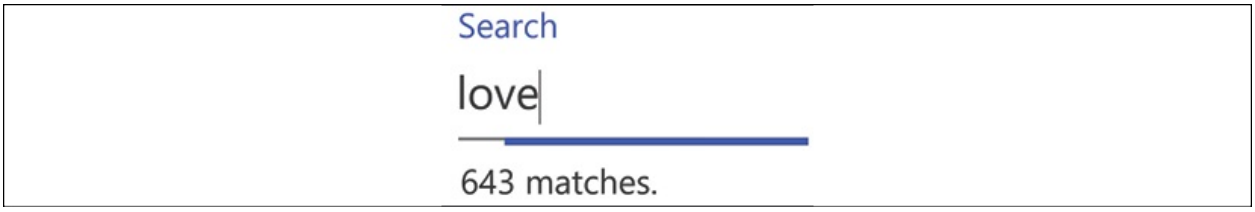


Figure 14.13: TensorBoard's search option

The PCA representation changes its view in realtime:

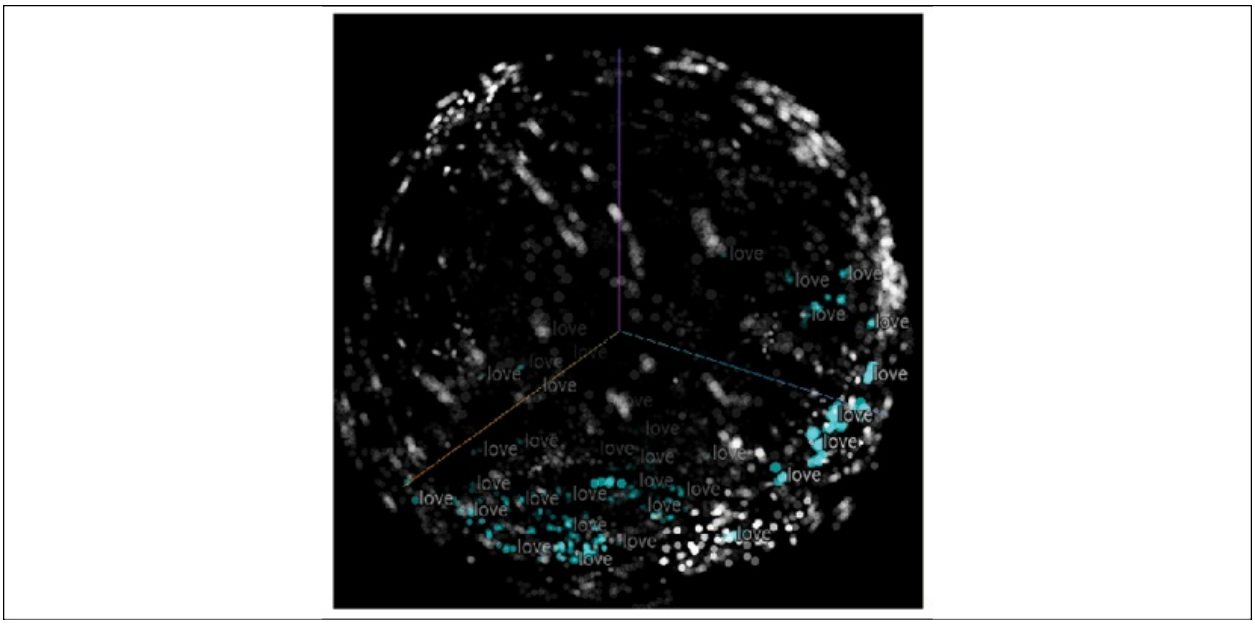


Figure 14.14: PCA representation of the RBM features

There are 643 points for "love." Notice that the "love" points are grouped in a relatively satisfactory way. They are mostly in the same area of the image and not spread out all over the image. This grouping shows that the weights of the RBM provided features that turned out to be sufficiently correct in the PCA for this experiment.

We repeat the process for each feature, to obtain the number of points per feature and visualize them. For the dataset supplied on GitHub for this chapter, we obtain:

- Love: 643
- Happiness: 2267
- Family: 0

- Horizons: 1521
- Action: 2976
- Violence: 4594

> **Important**: This result will naturally change if `RBM_launcher.py` runs again since it's a random viewer-movie choice process.

The results provide interesting information on the marketing segment we are targeting for the chatbot:

- Violence and action point to action movies.
- Family=0 points to younger viewers; teenagers, for example, more interested in action than creating a family.
- Discovering happiness and love are part of the horizons they are looking for.

This is typical of superhero series and movies. Superheroes are often solitary individuals.

We will see how this works out when we build our chatbot in *Chapter 15, Setting Up a Cognitive NLP UI/CUI Chatbot*.

Summary

In this chapter, we prepared key information to create the input dialog of a chatbot. Using the weights of an RBM as features constituted the first step. We saw that we could use neural networks to extract features from datasets and represent them using the optimized weights.

Processing the likes/dislikes of a movie viewer reveals the features of the movies that, in turn, provide a mental representation of a marketing segment.

PCA chained to an RBM will generate a vector space that can be viewed in TensorBoard Embedding Projector in a few clicks.

Once TensorBoard was set up, we analyzed the statistics to understand the marketing segment the dataset originated from. By listing the points per feature, we found the main features that drove this marketing segment.

Having discovered some of the key features of the marketing segment we were analyzing, we can now move on to the next chapter and start building a chatbot for the viewers. At the same time, we will keep backdoors available in case the dialogs show that we need to fine-tune our feature vector statistics.

Questions

1. RBMs are based on directed graphs. (Yes | No)
2. The hidden units of an RBM are generally connected to one another. (Yes | No)
3. Random sampling is not used in an RBM. (Yes | No)
4. PCA transforms data into higher dimensions. (Yes | No)
5. In a covariance matrix, the eigenvector shows the direction of the vector representing that matrix, and the eigenvalue shows the size of that vector. (Yes | No)
6. It is impossible to represent a human mind in a machine. (Yes | No)
7. A machine cannot learn concepts, which is why classical applied mathematics is enough to make efficient artificial intelligence programs for every field. (Yes | No)

Further reading

- For more on RBMs, refer to: https://skymind.ai/wiki/restricted-boltzmann-machine
- The original reference site for the source code in this chapter can be found here: https://github.com/echen/restricted-boltzmann-

machines/blob/master/README.md

- The original Geoffrey Hinton paper can be found here: http://www.cs.toronto.edu/~hinton/absps/guideTR.pdf
- For more on PCA, refer to this link: https://www.sciencedirect.com/topics/engineering/principal-component-analysis
- Ready-to-use RBM resources are located here: https://pypi.org/project/pydbm/

15

Setting Up a Cognitive NLP UI/CUI Chatbot

There are 300,000+ chatbots on Facebook alone. Adding another brick in that wall means next to nothing unless you give your chatbot a purpose and provide it with real content. Cognitive content represents the core goal of attracting more attention than your hundreds of thousands of competitors and SEO experts. We will put RBM-PCA chained algorithms to work in this chapter to bring a chatbot to another level. We will use the information provided by the RBM-PCA to design our dialog.

As you will discover in this first section, creating an agent with Dialogflow and beginning a dialog represents no effort at all. Google Dialogflow provides the intuitive features to get a chatbot running in no time. The Dialogflow tutorial can guide you to reach this simple goal in a few minutes. Understanding what an agent is, teaching it to ask a question, and providing an answer can be done by a 10-year-old child. I experimented with this by letting a 5-year-old and a 9-year-old child loose on this software. They both did not even realize it was work. They were having fun!

On Dialogflow, you don't need to know how to program, and you don't need to be a linguist or any other kind of expert. So, what will your market differentiation be? *Content*. Your chatbot needs to have a purpose, with well-prepared content beyond asking and answering simple questions.

Beyond creating your first dialog, the goal of this chapter will provide you with a sense of purpose and content that will help you produce meaningful chatbots.

That being said, let's create an agent together and a short dialog to illustrate both how to create a chatbot and also to provide meaningful content.

The following topics will be covered in this chapter:

- Creating a cognitive agent based on the preparation of *Chapter 14, Preparing the Input of Chatbots with Restricted Boltzmann Machines (RBMs) and Principal Component Analysis (PCA)*
- Learning the basic concepts of Dialogflow and chatbots in general
- Deploying the chatbot on your website

We will start with basic concepts and then create an agent with entities, intents, dialogs, and a fulfillment function. We will be using the preparation established in the previous chapter. We will test its UI with spelling correction and dialogs. Then, we will test the chatbot's **conversational user interface** (**CUI**) capability by setting up machine learning speech recognition and speech functions.

Basic concepts

Before creating an agent, we'll want to have an understanding of the basic concepts.

This is not a Dialogflow course, but rather an introductory chapter to get us started on making our own NLP CUI chatbot. We'll begin by defining some key terms.

Defining NLU

NLU means **natural language understanding**. NLU is a subset of **natural language processing** (**NLP**). Natural language refers to the everyday language we use without having to force ourselves to learn precise words in order to obtain information from a machine.

If we had to learn a dictionary of the only words that would work with a system, it would be easier just to read a text. NLP encompasses all forms of natural language processing including NLU. Through AI, NLU has become more involved in trying to understand what a given sentence means.

Why do we call chatbots "agents"?

A chatbot entails a chat between at least two parties. In our case, the bot is an NLU module. That's not a very nice marketing way to put it. It sounds like: "you are now talking to an NLU module." You cannot pretend a bot is a person. The word *agent* conveys the impression of a business agent, a sports agent, or a secret agent, which is mysterious! It came to mean a computer system that gathers information. Now it's an NLP agent with NLU capability.

Creating an agent to understand Dialogflow

The fastest way to learn Dialogflow is to create a dialog from scratch. Log into Dialogflow and go to the console. The following is part of a screenshot of Dialogflow's dashboard. You can see the link to the console on the top right:

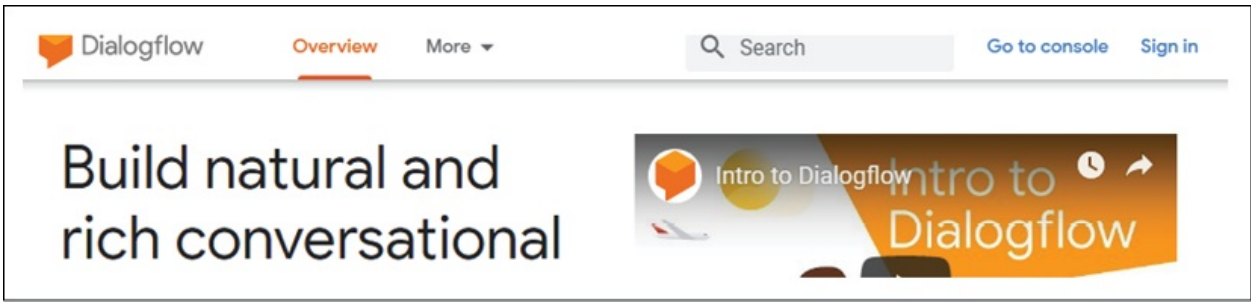


Figure 15.1: Accessing Dialogflow's console

Once you click on **Go to console**, you will be asked to sign in if you haven't signed in yet. A Google account is a prerequisite:

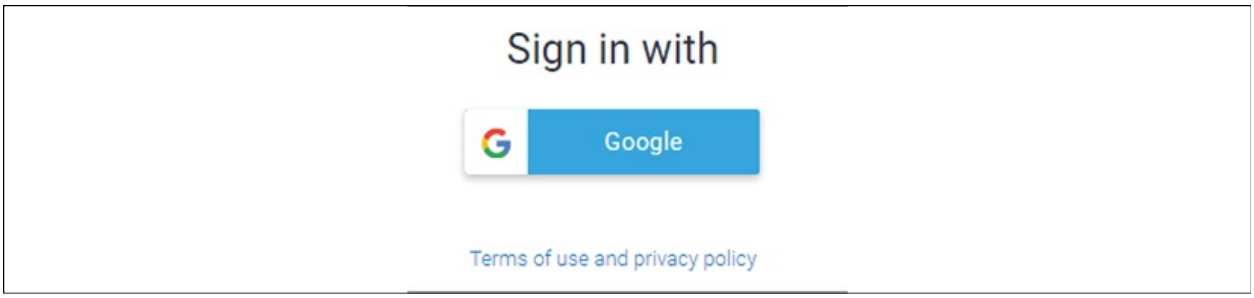


Figure 15.2: Signing in with a Google account

Once you have followed the sign in instructions and are signed in, you will reach

the Dialogflow console.

Click on the drop-down list in the top-right corner, irrespective of which default agent is displayed. A list of existing or default agents will be displayed:

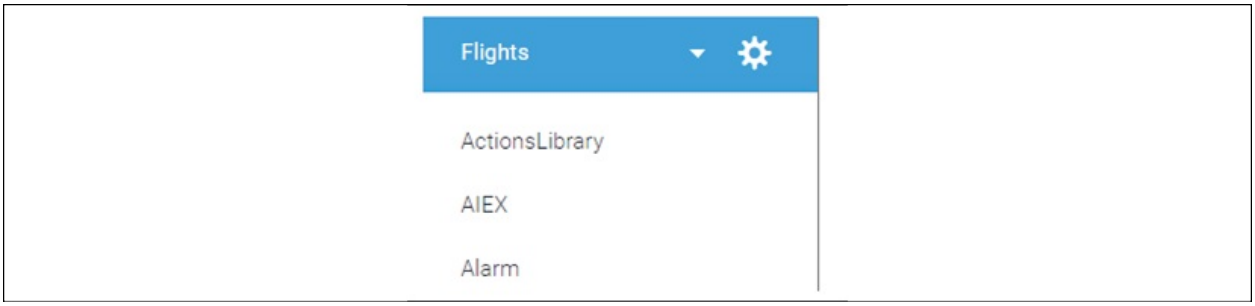


Figure 15.3: The list of agents

Scroll down the list until you reach **Create new agent** (if none, click on the **Create agent** option). Click on **Create new agent**, and you will reach the following window:

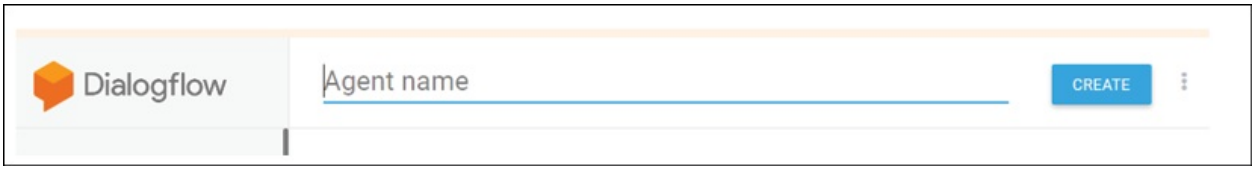


Figure 15.4: Entering the agent's name

Call the agent `Agent + <your name or initials>` to make sure you will have a unique name. I will call the one for this chapter `cogfilmdr`. The agent in the chapter will thus be referred to as `cogfilmdr`. Let Google create a default agent structure with English as the main language.

Once that is done, click on the settings button in the top left:



Figure 15.5: The settings button

You will reach the configuration window of your agent.

For the moment, we just have one important option to check. The version of the

API must be V2 API. V1 API will shut down on March 2020:

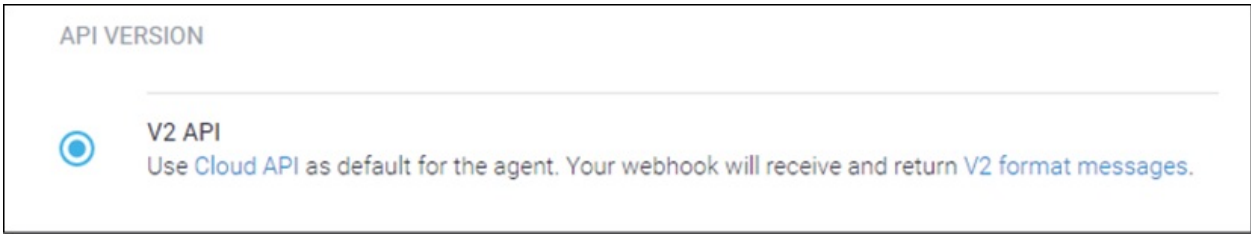


Figure 15.6: Using the V2 API

The agent is now created, and we can create entities.

Entities

Most chatbot tutorials explain intents first. I do not agree. Once you know where you're going, in this case, choosing a movie based on *Chapter 14*, *Preparing the Input of Chatbots with Restricted Boltzmann Machines (RBMs) and Principal Component Analysis (PCA)*, it makes sense to build some bricks before building your structure.

Dialogflow (or any chatbot) uses entities to extract useful information in a user's utterance (not necessarily a sentence) to understand their motivation.

We will use the entities created in *Chapter 14*. We will first create an entity named `movies` that will contain the 10 target movies used in *Chapter 14*.

```
titles=["24H in Kamba","Lost","Cube Adventures",
        "A Holiday","Jonathan Brooks",
        "The Melbourne File", "WNC Detectives",
        "Stars","Space L","Zone 77"]
```

Click on **Entities** on the left-hand side of the window:



Figure 15.7: Dialogflow menu

Then, click on **CREATE ENTITY**:

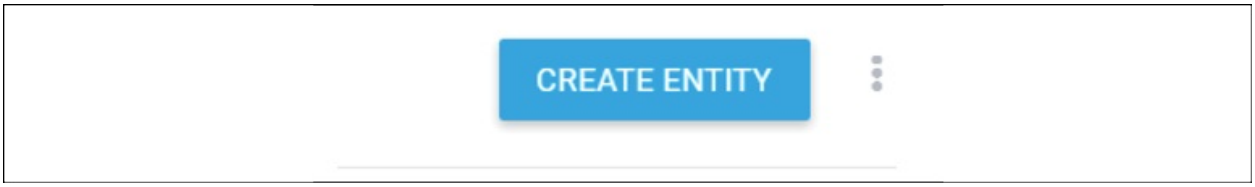


Figure 15.8: Creating an entity

You will be asked to provide an entity name:

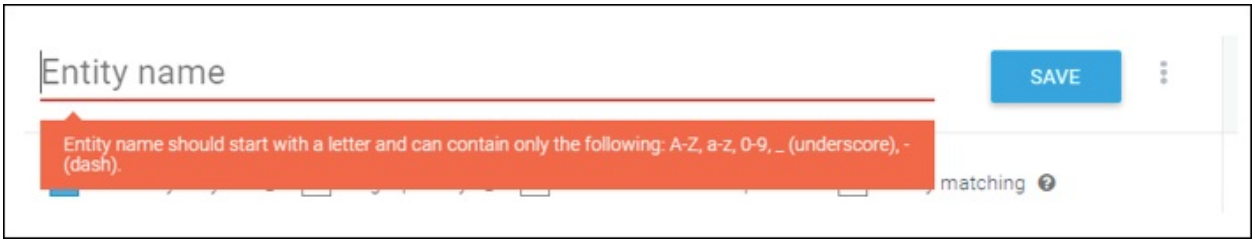


Figure 15.9: Entering the name of entity

Enter `movies`. Before saving the entity, we must enter the movies we have chosen:

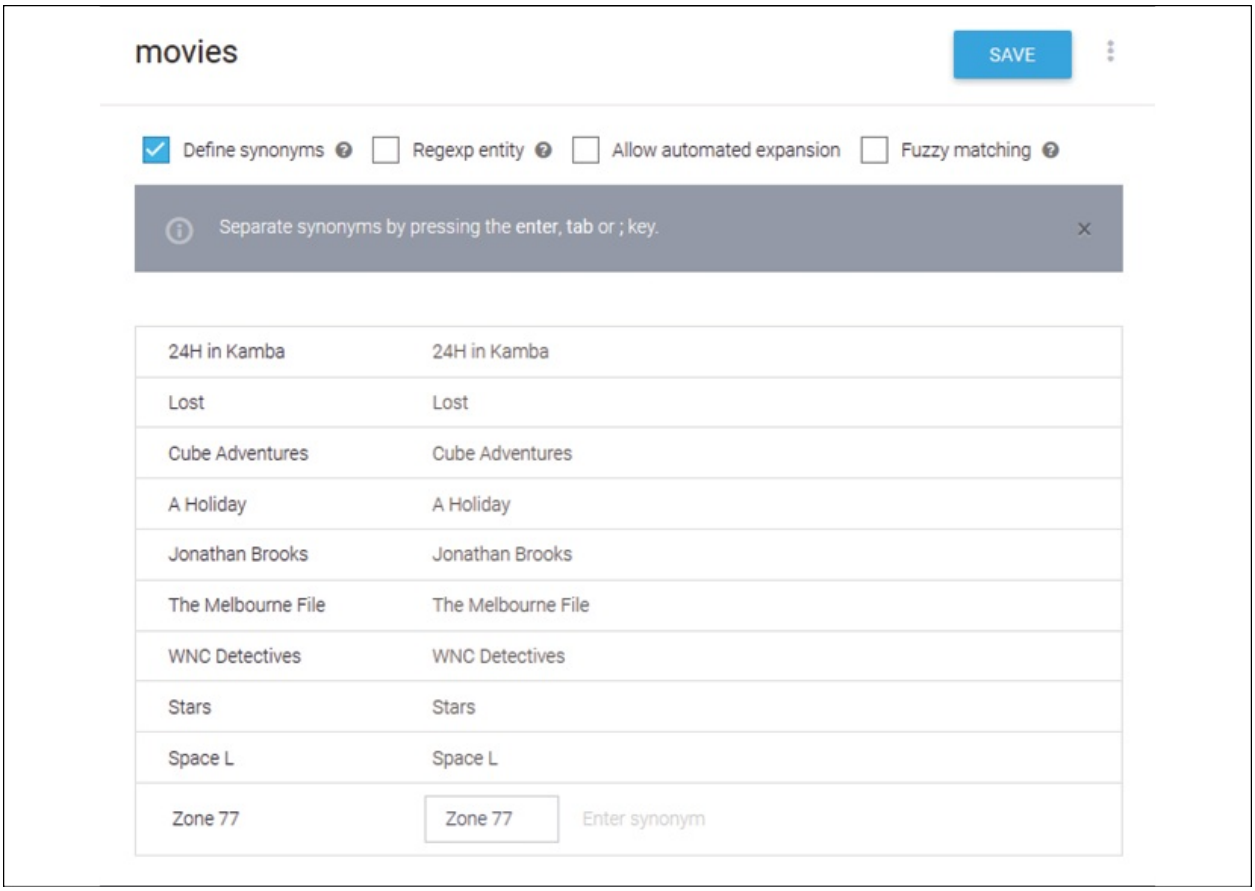


Figure 15.10: Entity list

You will notice that once you add a movie, a default synonym is filled in automatically. You can add other synonyms if you wish.

Once the titles are entered, click on the **SAVE** button, which is mandatory (it is not an auto-save interface):



Figure 15.11: Saving an entity

We will now create the feature entity. The features in the `RBM.py` program in *Chapter 14* were as follows:

```
# Each column is a feature. There are 6 features: ['love',
'happiness', 'family', 'horizons', 'action', 'violence']
```

We will name it `features`, follow the same process as for the `movies` entity, and then click on **SAVE**:

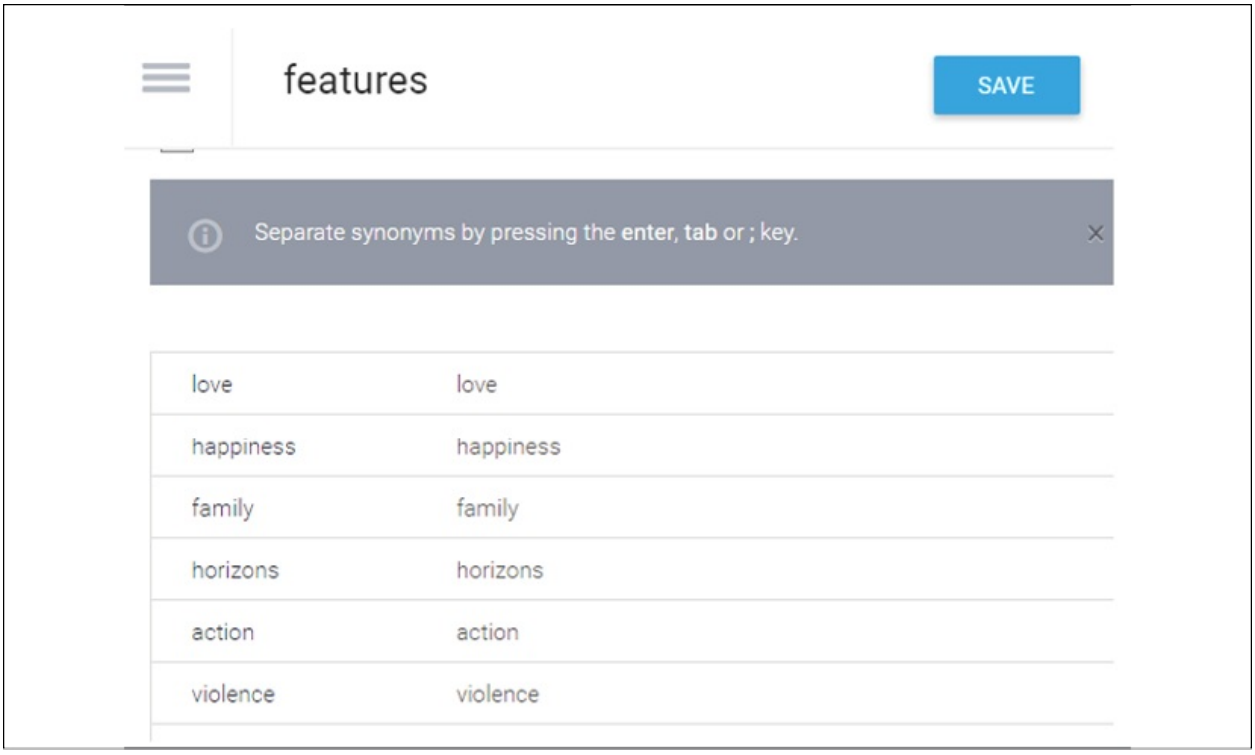


Figure 15.12: Creating a feature entity

Click on **Entities** under your agent, and you will see a list of entities for the agent, as shown in the following diagram:

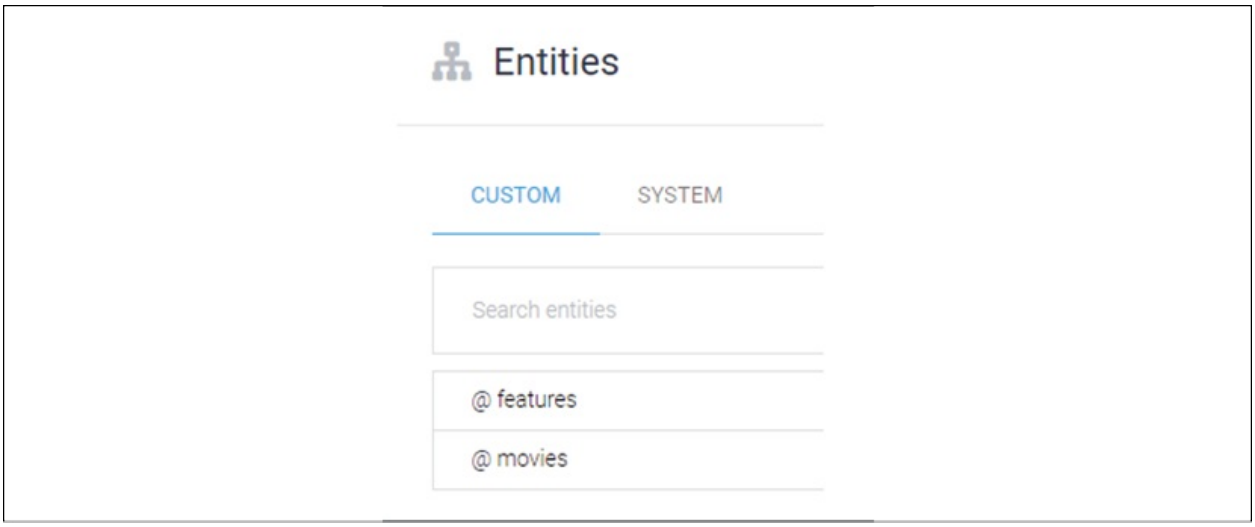


Figure 15.13: List of entities

If you click an entity, a list of possible choices will appear.

Now that your agent knows the movie and feature entities, creating the intents makes sense.

Intents

An intent is a clear formulation intention to do something. I named the agent `cogfilmdr`. For the agent, the user's intention may be to ask for a movie to watch.

To trigger a response, we must enter training phrases.

Training phrases are groups of words that the user will enter through text or speech. The more sentences you enter, the better your chatbot will become. This is why starting with a ready-to-use Dialogflow makes sense if an existing agent satisfies your needs.

To create our sample dialog, we will use the dataset results supplied on GitHub for *Chapter 14, Preparing the Input of Chatbots with Restricted Boltzmann Machines (RBMs) and Principal Component Analysis (PCA)*. The main terms have been extracted with their features that we displayed with TensorBoard. When we extracted the data from the RBM, we sorted the features as follows:

```
Love: 643
Happiness: 2267
Family: 0
Horizons: 1521
Action: 2976
Violence: 4594
```

We displayed the feature space in a PCA:

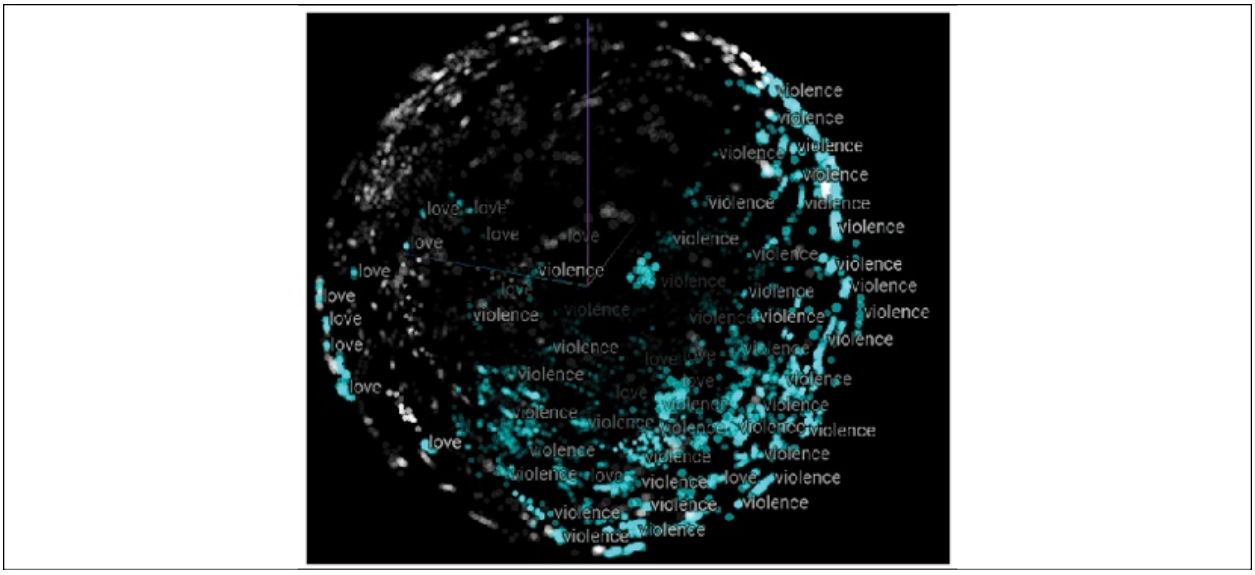


Figure 15.14: TensorBoard representation of features

> **Reminder**: This result will naturally change if `RBM_launcher.py` runs again since it's a random viewer-movie choice process.

When starting a chatbot project, it is best to be very careful with going straight to generating dialogs automatically. It is much better to start with a simple, well-structured chatbot that works on a limited amount of tasks. I call this a "closed chatbot" meaning that we control every aspect of dialog. An "open chatbot" means that information flows in automatically to create automatic dialogs. That can be a goal after getting the chatbot to run as a "closed chatbot" for some time using the information prepared with AI algorithms.

The results of the work we did in *Chapter 14* provide interesting information on the marketing segment we are targeting for the chatbot.

Violence and action point to action movies. Family=0 points to younger viewers, teenagers, for example, more interested in action than creating a family. Discovering happiness and love is part of the horizons they are looking for. This is typical of superhero series and movies. Superheroes are often solitary individuals.

We will now create an intent by entering the **Intents** window:

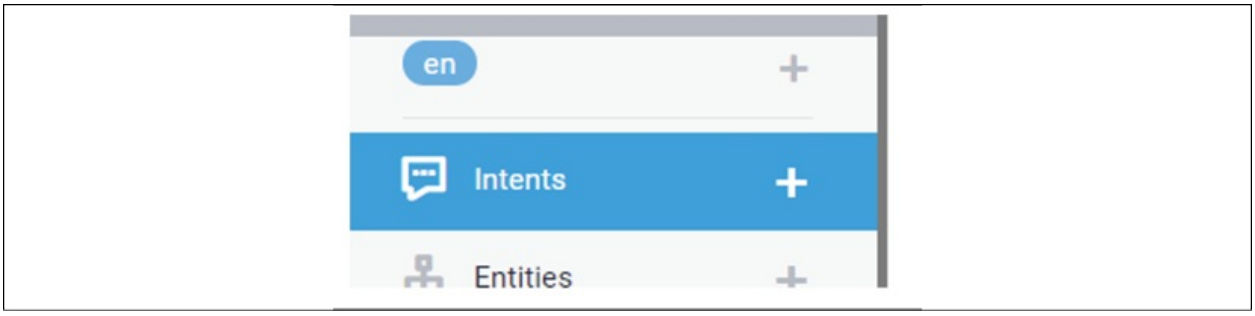


Figure 15.15: Choosing the Intents option

The **Intents** window appears. Click on **CREATE INTENT**:



Figure 15.16: Creating an intent

The intent window will appear to create a question-and-answer dialog in a few steps:

- First, enter `choose_movie` as the name of the intent.
- Then, in the training phrases section, enter: "I would like to watch one of your movies."

At this point, we have an intent name and a possible user question:

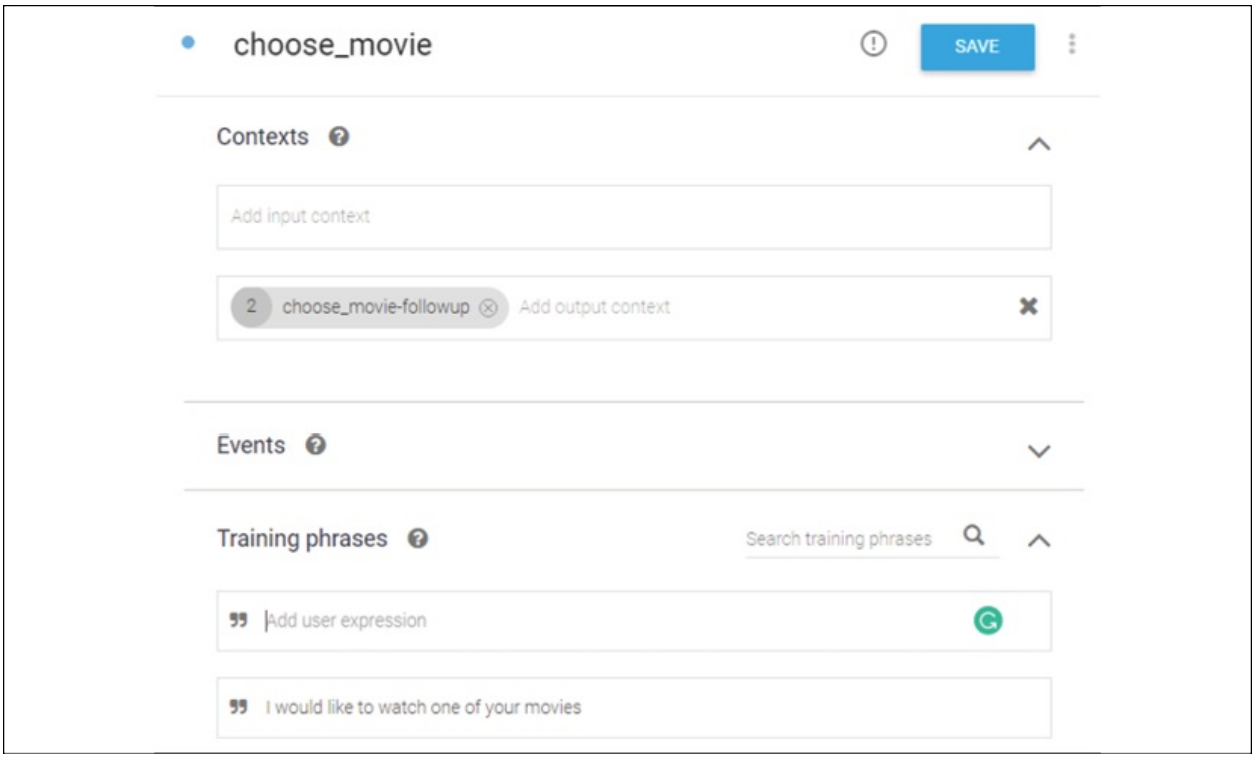


Figure 15.17: Entering intent information

Now, we need to provide a response based on the statistics for this market segment we drew from *Chapter 14*. We will use the word *action* to encompass a movie that contains violence and a happy ending as in the typical superhero movies. To do that, scroll down to the **Text Response** section to add a response and enter "Would you like to watch an action movie?", as shown in the following screenshot:

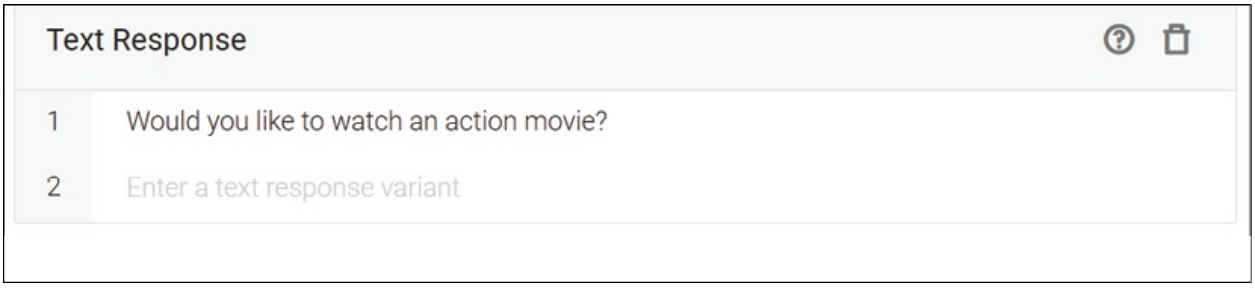


Figure 15.18: Entering the text response

Now that we have a basic dialog, let's save and test it. To do that, go to the test console in the top right of the console window:

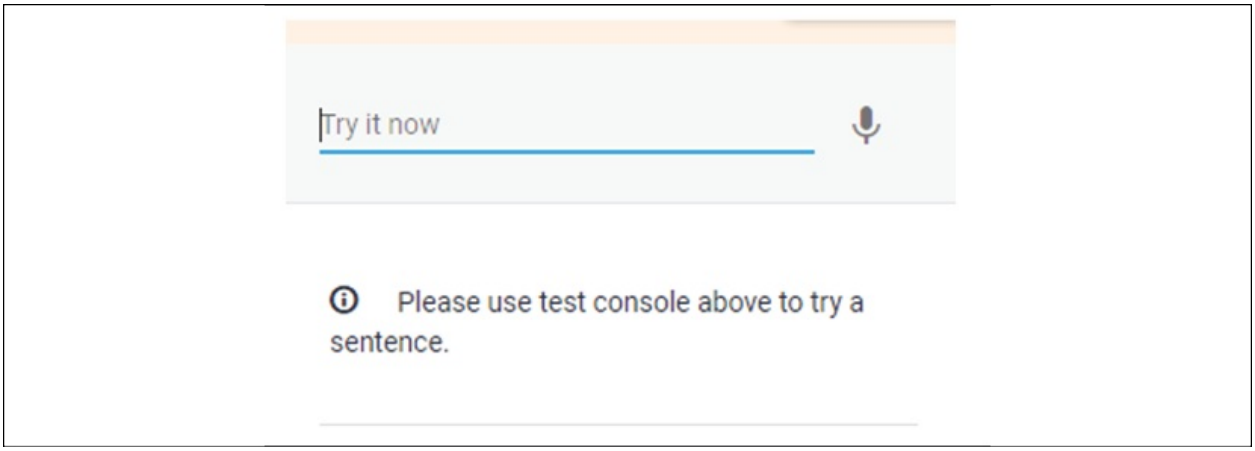


Figure 15.19: Test console

We can use a CUI or text:

- **Text dialog**: Enter the user phrase: "I would like to watch one of your movies."

  The user will be surprised to see the agent's answer, which is, "Would you like to watch an action movie?"

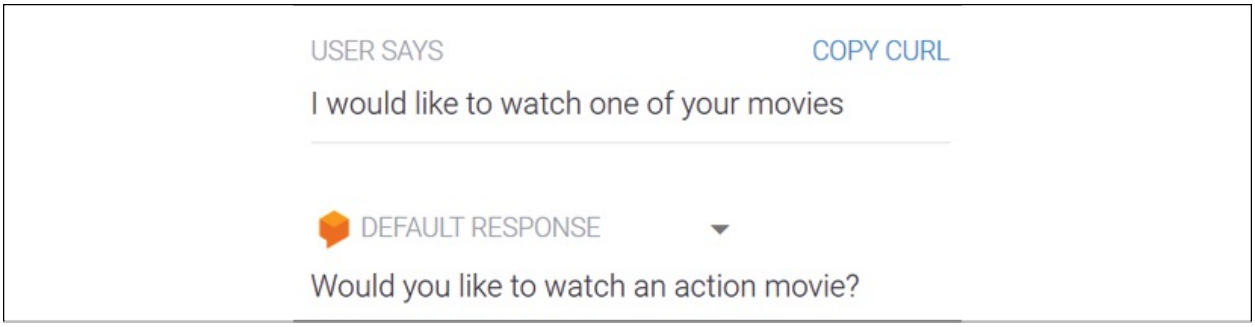


Figure 15.20: Responses

This suggestion comes as a surprise for the user and might seem strange. This is because the RBM-PCA approach we used to prepare the dialog targets a market segment.

Advanced machine learning shortens the path of a user request to a satisfactory response. It constitutes both a time saving and an energy saving process for the user.

- **CUI**: Click on the microphone icon in the test console. Make sure that this

microphone is authorized, or it will not work:

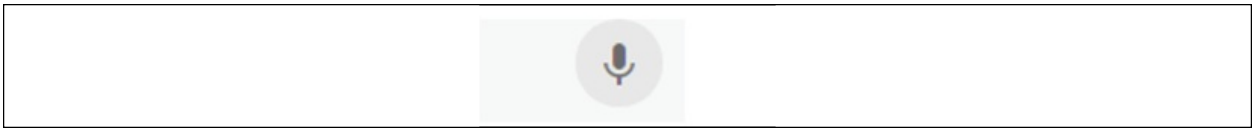


Figure 15.21: Microphone

When you click on the microphone, this will trigger a recording of your request. Say, "I would like to watch one of your movies." Then, click on the stop button to stop the recording. The response to the request will appear:

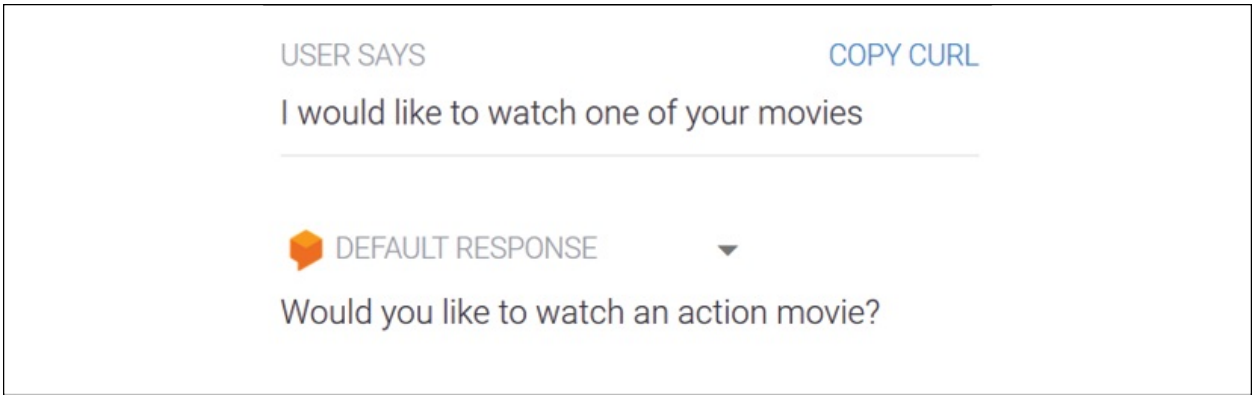


Figure 15.22: Text responses

To answer the question, we will need to use the context functionality of Dialogflow.

Context

Context means that Dialogflow is going to remember a dialog and use follow-up exchanges without starting from scratch each time.

The user has asked to watch a movie, and the bot suggested an action movie. The bot will remember this through context as it continues the dialog.

Click on the agent's **Intent** in the menu and hover over **choose_movie**. You will see **Add follow-up intent** appear. This means that all of the variables of the main intent can be stored, and a follow-up intent added that would remember what was said previously, just like us in a conversation.

Click on **Add follow-up intent**:



Figure 15.23: Add follow-up intent button

In this case, the agent has planned two cases, *yes* or *no*. We will explore the *yes* answers in this chapter, and the more complex *no* answers in *Chapter 16*, *Improving the Emotional Intelligence Deficiencies of Chatbots*.

In *Chapter 14*, we created a movie feature matrix with the movie titles and features:

```
    # Part I Feature extractions from data sources
    # The titles of 10 movies
    titles=["24H in Kamba","Lost","Cube Adventures",
            "A Holiday","Jonathan Brooks",
            "The Melbourne File", "WNC Detectives",
            "Stars","Space L","Zone 77"]
    # The feature map of each of the 10 movies. Each line is
 a movie.
    # Each column is a feature. There are 6 features: ['lov
e', 'happiness', 'family', 'horizons', 'action', 'violence'
]
    # 1= the feature is activated, 0= the feature is not ac
tivated
    movies_feature_map = np.array([[1,1,0,0,1,1],
                                   [1,1,0,1,1,1],
                                   [1,0,0,0,0,1],
                                   [1,1,0,1,1,1],
                                   [1,0,0,0,1,1],
                                   [1,1,0,1,1,0],
                                   [1,0,0,0,0,0],
                                   [1,1,0,1,1,0],
                                   [1,1,0,0,0,1],
                                   [1,0,0,1,1,1],
                                   [1,1,0,0,1,0],
                                   [1,1,0,1,1,1],
                                   [1,1,0,0,1,1]])
```

We now need to transpose this information in a chart we can use to add content

depth to the dialog:

MOVIE/FEATURE	LOVE	HAPPINESS	FAMILY	HORIZONS	ACTION	VIOLENCE
24H in Kamba	1	1	0	0	1	1
Lost	1	1	0	1	1	1
Cube Adventures	1	0	0	0	0	1
A Holiday	1	1	0	1	1	1
Jonathan Brooks	1	0	0	0	1	1
The Melbourne File	1	1	0	1	1	0
WNC Detectives	1	0	0	0	0	0
Stars	1	1	0	1	1	0
Space L	1	1	0	0	0	1
Zone 77	1	0	0	1	1	1

We have already surprised the user a bit by proposing an action movie directly without going through tedious lists. We are using all of the information we obtained through inputs, intermediate AI outputs, and final outputs.

Now, we are going even further by filtering the movies that fit the action-violence-happiness features extracted with the RBM-PCA chained algorithms.

Only the following movies in the chart match action-violence-happiness:

- 24H in Kamba

- Lost
- A Holiday
- Zone 77

At random, we will choose "Zone 77." Once we have entered many possibilities, a random choice can be suggested either in the response area or with scripts. This development is beyond the scope of this chapter. For this example, we suppose it is probable that the viewer will be satisfied with this suggestion we make. We are in a *yes* scenario of the dialog. In *Chapter 16*, *Improving the Emotional Intelligence Deficiencies of Chatbots*, we will explore the *no* scenarios of this dialog, which requires more cognitive designing to keep the satisfaction path short.

For the moment, let's suggest "Zone 77." To do this:

1. Click on **Add follow-up intent**.
2. Select **yes**.

You now have a follow-up intent linked to the dialog:

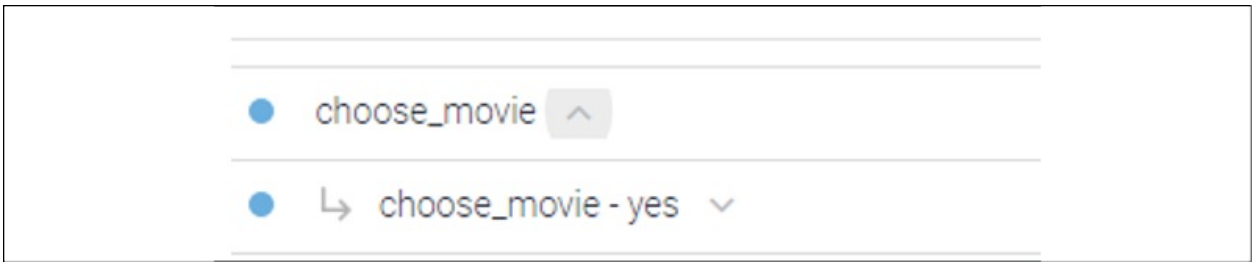


Figure 15.24: Follow-up intents

Click on **choose_movie - yes**. The intent will appear. You will notice that Dialogflow has already filled in several forms of *yes* in the **Training phrases** section, as shown in the following screenshot:

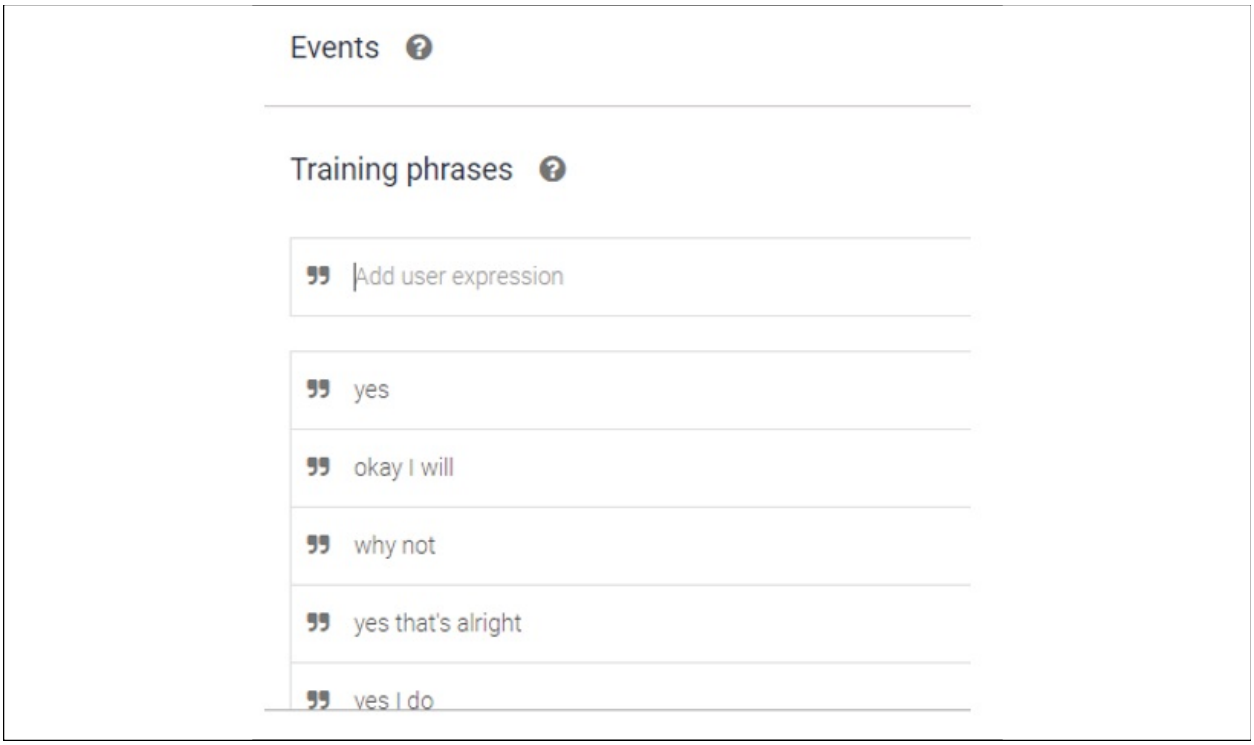


Figure 15.25: Training phrases

All that is left to do in this scenario is to scroll down to the **Responses** section and add our answer:



Figure 15.26: Text response

Now, we go back to the intent and add a *yes* follow-up to this follow-up to process the viewer's *yes* answer, just as we did previously:

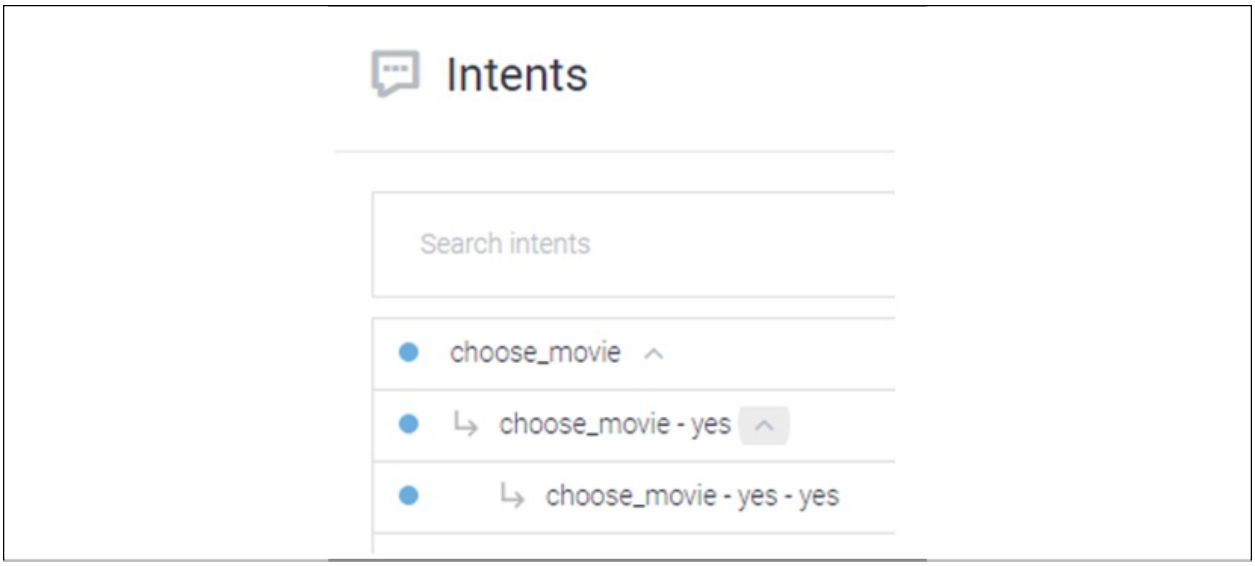


Figure 15.27: Follow-up intents

Now, we click on **choose_movie - yes - yes**, and we will see the *yes* answers that Dialogflow prepared for us:

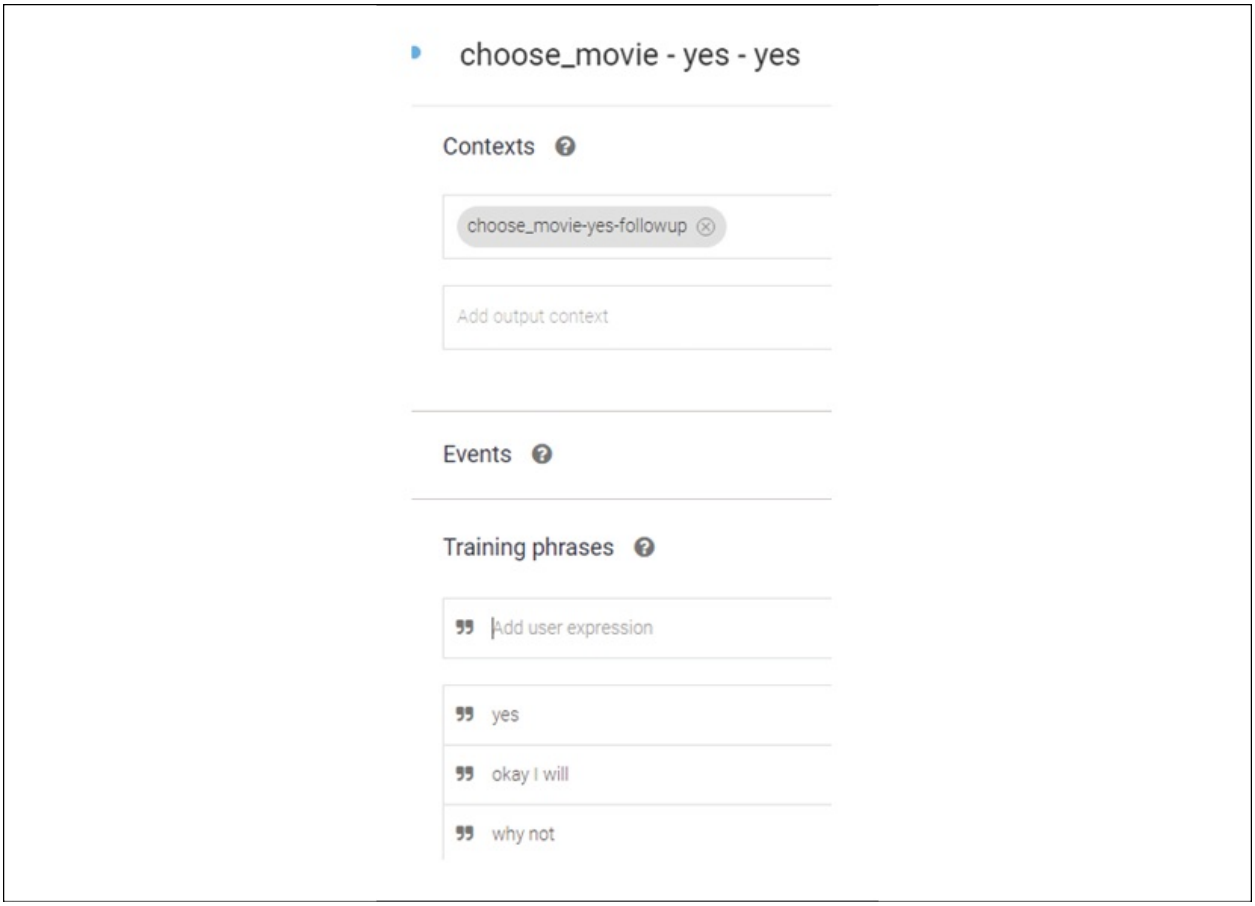


Figure 15.28: Training phrases for a follow-up intent

However, this time, we would like to answer with a script and not an answer we type in.

To do that, we can use fulfillment.

Adding fulfillment functionality to an agent

A dialog can quickly become boring in everyday life and even more so in a chatbot. When we begin to guess everything that an interlocutor has to say, our mind slowly drifts away. We cannot help it. Humans are a curious species. **Fulfillment** will change the perspective of dialog. That is what I call *purpose* beyond the pragmatic approach that says fulfillment adds business logic to a dialog.

To make the dialog sustainable, even from a practical point of view, it has to excite the user enough to want it to come back and discover more about your chatbot beyond obtaining business information from it.

If you look **fulfilling** up in a dictionary, you will find that it means providing happiness or satisfaction, which is exactly the feeling of purpose you want your chatbot to convey.

That being said, there is work to do in order to reach that goal. Dialogflow provides a wide array of tools to reach fulfillment for the user, the designer, and the developers.

To start with, Dialogflow uses an inbuilt, seamless version of Node.js for fulfilling functions.

Defining fulfillment

Various fulfillment or additional dialog functions are available:

- **Webhook**: A webhook is an event transmitted via HTTP. It is sent as a `POST`, which contains data posted to a predetermined URL. It works as an HTTP callback. The data sent to the URL will be parsed by a script on the server side. Once the service has processed the information, it will perform an action and send data back as a response.

  We will not use the webhook for this example. However, it is important to note that you can use a webhook to create dialogs of your own in another environment. You can even generate automatic dialogs and call them from Dialogflow.

  If you are interested in preparing dialogs and uploading them, you can go to the **Training** page of the agent on the left-hand side of the screen and upload phrases:



Figure 15.29: Training window

You can also upload an agent or even a prebuilt agent designed by Google Dialogflow. For our example, we will use the inline editor.

- **Fulfillment with the inline Node.js editor**: Defining a webhook URL can be the simplest approach. However, using the inline editor provides Node.js functionality for even more potential.
- **Fulfillment with the inline Node.js editor and Cloud Functions for Firebase**: The inline Node.js can call a large variety of Cloud Functions for Firebase in a few time-saving lines of code.

Enhancing the cogfilmdr agent with a fulfillment webhook

When the user answers *yes* to watch "Zone 77," we can answer with a response

or a link to a website. To use a response, go to the yes - yes follow-up of our dialog in the **Intents** window:

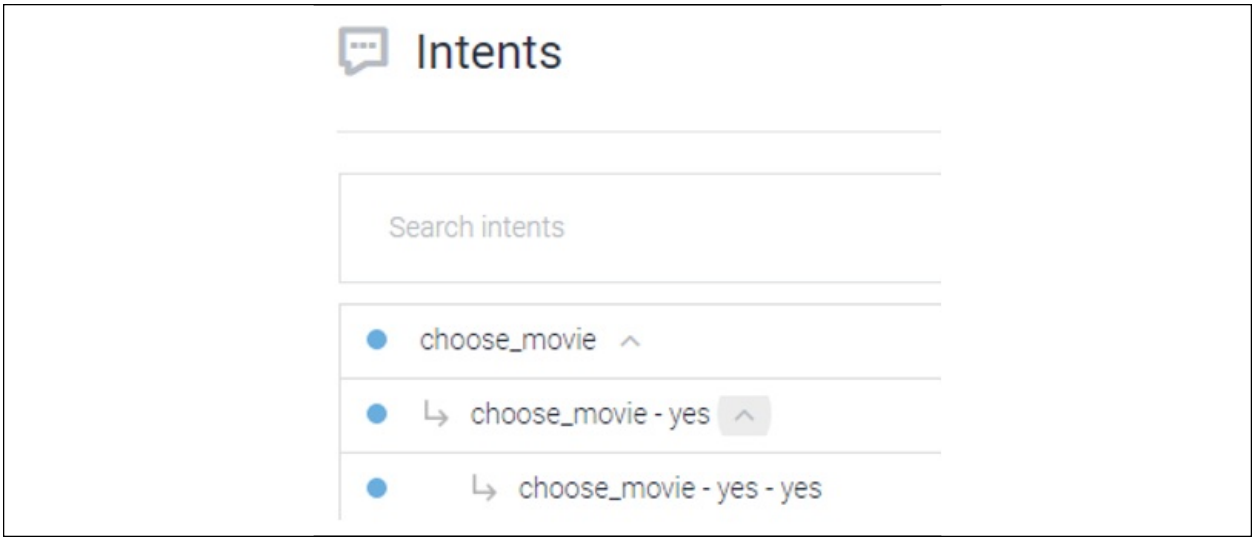


Figure 15.30: Intents and follow-up intents

Click on **choose_movie - yes - yes** and scroll down to **Text Response** and add a response such as "Sure, click on the movie and watch it," as shown in the following screenshot:

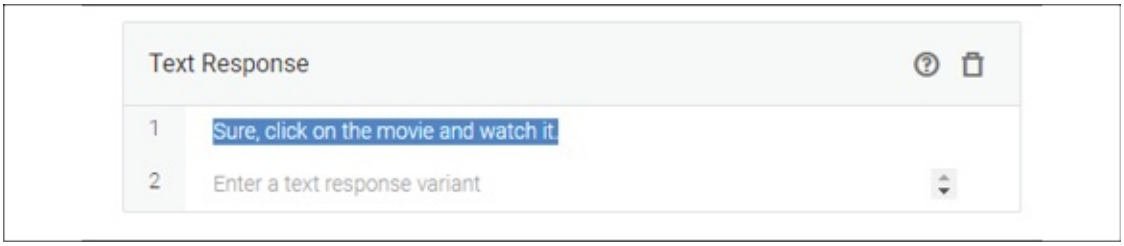


Figure 15.31: Text response

We would also decide that this is final and that it is the end of the conversation by activating the **Set this intent as end of conversation** option:

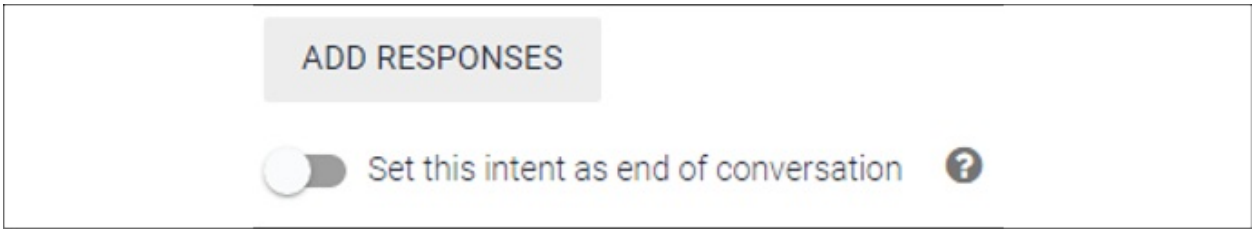


Figure 15.32: End of conversation option

But we will not do this for the moment; let's scroll down further to activate the inline webhook functionality:

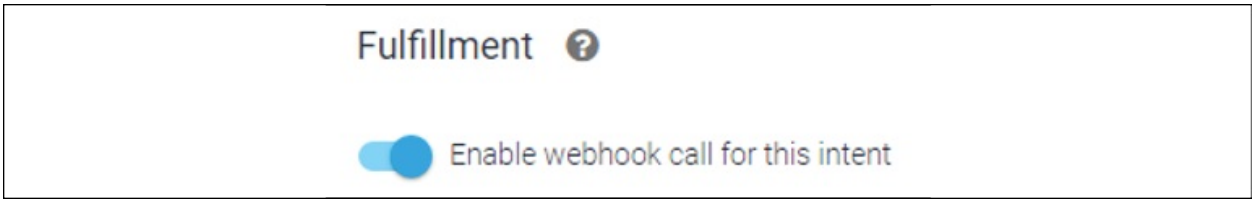


Figure 15.33: Enabling the webhook functionality

Now we will go to the **Fulfillment** window:

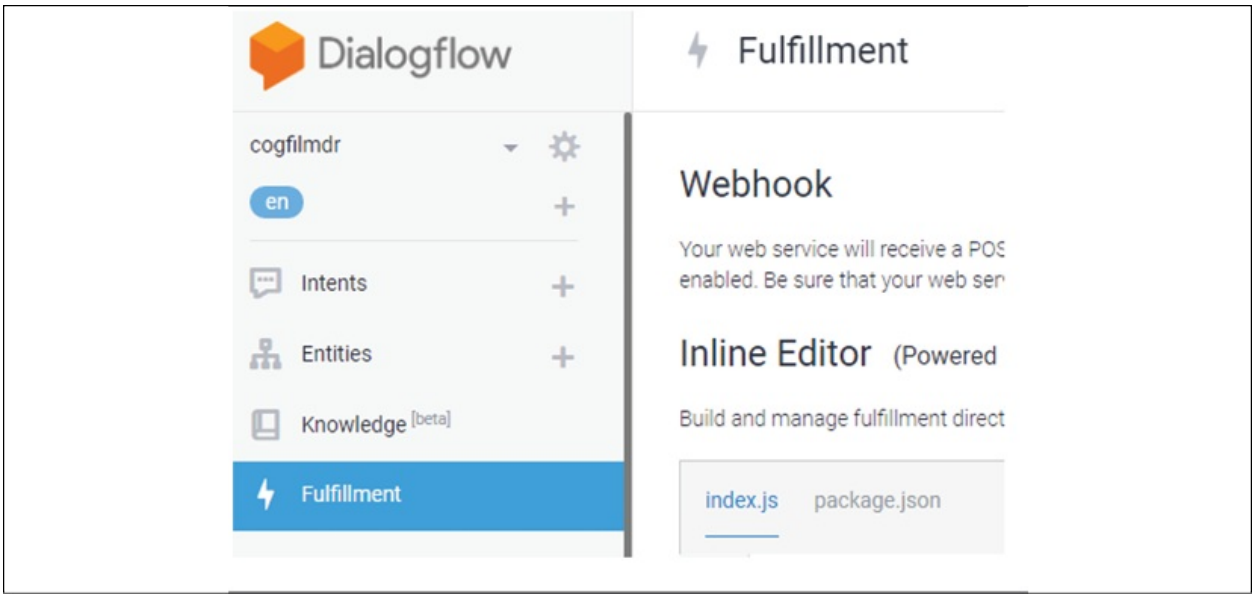


Figure 15.34: Access to the Fulfillment interface

First, enable the inline editor:

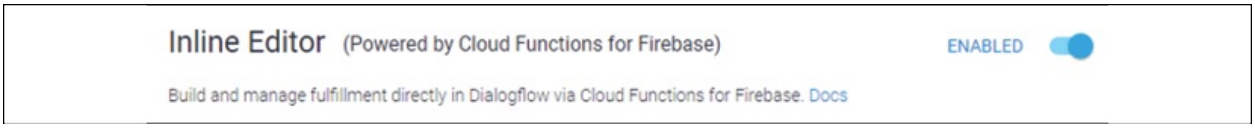


Figure 15.35: Inline editor

To add some fun to the dialog, let's suppose that chatbot is in a cool start-up coffee shop and that watching movies on individual screens is a service to attract customers. You can watch the movie with headsets (you, friends, family), for example. We add this service to our dialog.

We go to the `intentMap` of the script and add a `gotomovie` function:

```
let intentMap = new Map();
intentMap.set('Default Welcome Intent', welcome);
intentMap.set('Default Fallback Intent', fallback);
// intentMap.set('your intent name here', yourFunctionHandl
er);
// intentMap.set('your intent name here', googleAssistantHa
ndler);
intentMap.set('choose_movie-yes-yes', gotomovie);
```

The recommended format is `intentMap.set(<Intent>,<function>)`.

That done, we now write the function with our own text and website link:

```
function gotomovie(agent) {
    agent.add('This message is from Coffee Shop movie fans!'
);
    agent.add(new Card({
    title: 'A blog for Coffee Shops with movies to watch wi
th friends and family',
    imageUrl: 'https://www.eco-ai-horizons.com/coffeeshop.j
pg',
    text: 'The button takes you to the movie selection we h
ave\n  Great to have you here! :-)',
    buttonText: 'Click here',
    buttonUrl: 'https://www.primevideo.com/'
    })
    );
}
```

All we have to do now is click on **DEPLOY**, and that's it!

I added a link to Amazon Prime Video to show that you can use IBM Watson, Google, Microsoft, Amazon services, and more to enhance your chatbots!

> **Important**: You can customize all the dialogs you wish in this editor and use a range of Google Cloud functions. The sky is not even the limit. You can go to Mars!

For our example, once the script has been deployed, we go back to the beginning of our dialog until we reach this point, which will take us to the movie we wish to watch. In this case, I just displayed a streaming site.

You can obtain the full script of `index.js` in the `dialogflowFulfillment.zip` on GitHub in `CH15` and copy it into the editor without importing the whole package.

Getting the bot to work on your website

Let's get a bot running on your website in a few clicks.

1. Scroll down to **Integrations** for the agent you wish to deploy: either your own agent or the coffee shop agent.
2. Activate the **Web Demo** option.
3. An embedded code will appear along the lines of the following:

```
<iframe height="430" width="350" src="https://bot.dialo
gflow.com/81298e0c-1acb-44bb-9c68-89666805342a"></ifram
e>
```

Copy the code on the page of the website you wish to implement it on.

To test it first, you can copy the URL in your browser and test it.

> **Note**: You can use a speech dialog if you activate your microphone for this site on your browser. Don't forget to access the page using `https`, otherwise the microphone might be blocked. Also, fulfillment cannot be activated in this HTML page without some additional development.
>
> However, you can also click on Google Assistant in the console and create an application in a few clicks, and then deploy it on smartphones and Google Home, for example. If you create a nice chatbot, you can have the whole world use it in a few clicks!

Machine learning agents

An NLP chatbot cannot function without machine learning for text recognition, utterances, sentences, speech, entities, intents, and many other aspects of a dialog.

In this section, we will explore the following:

- Speech-to-text
- Text-to-speech
- Spelling correction

Let's see how we can apply machine learning in each of these contexts.

Using machine learning in a chatbot

Generally, when we hear of machine learning in a chatbot, we think of a machine learning program running during a dialog as a response.

In this section, we will focus on how machine learning is used to improve a chatbot and to make it work.

Speech-to-text

Without a speech-to-text function, there is no way you can implement a chatbot or any speech application on a smart speaker such as Google Home or Amazon Echo. Smart speakers are going to play an increasing part in our lives in the years to come.

Click on the settings button next to the name of the agent and then on **Speech**:

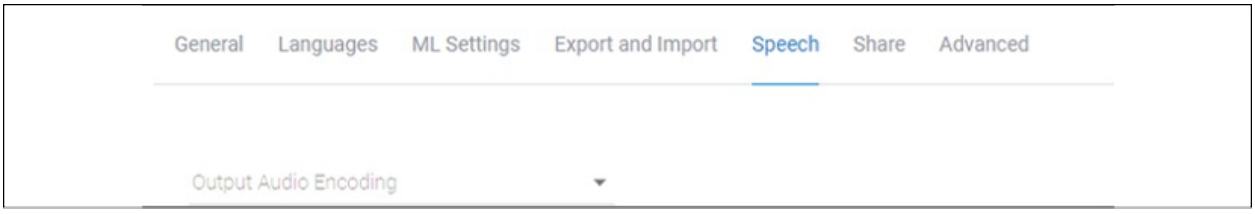


Figure 15.36: Speech options

We will focus on the main settings of the speech recognition functions:

- **Enhanced Speech Models**: This is an advanced machine learning option that comes with the Enterprise Edition. It shows how far speech recognition has come. In the standard version, the system already works fairly well. In the advanced version, it uses data logging functionality to enhance speech recognition.
- **Auto Speech Adaptation**: This is interesting because this function uses the intents and entities created to train and adapt to speech recognition of the agent's dialog. It can be activated in the free version as follows:

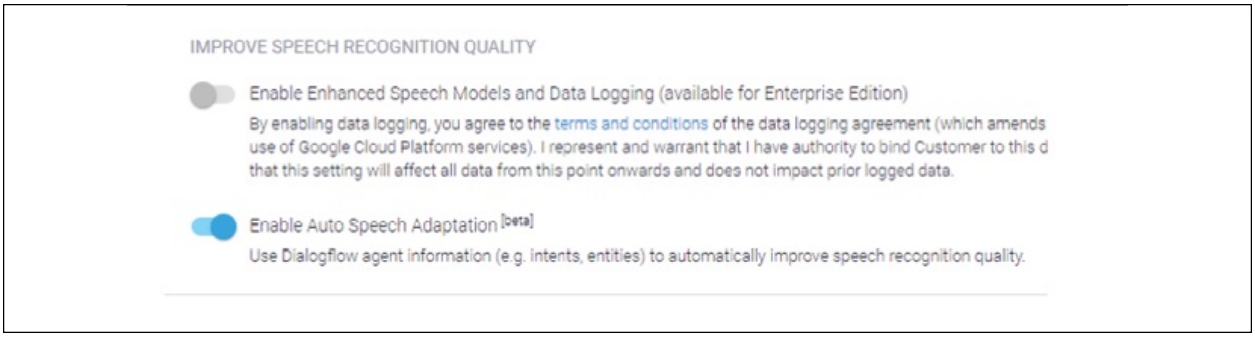


Figure 15.37: Enabling Auto Speech Adaptation

Save the settings before leaving this interface.

Text-to-speech

Now, we can go to the **Speech** tab and enable the **Automatic Text to Speech** function. I have a cloud account. If you cannot activate this in the lab, we will use the free online site to test the possibilities and limits of the machine learning algorithm.

> **Note**: There is an enhanced speech recognition model option, but you have to upgrade to the enterprise version.

Click on the settings button next to the name of the agent and then on **Speech**:

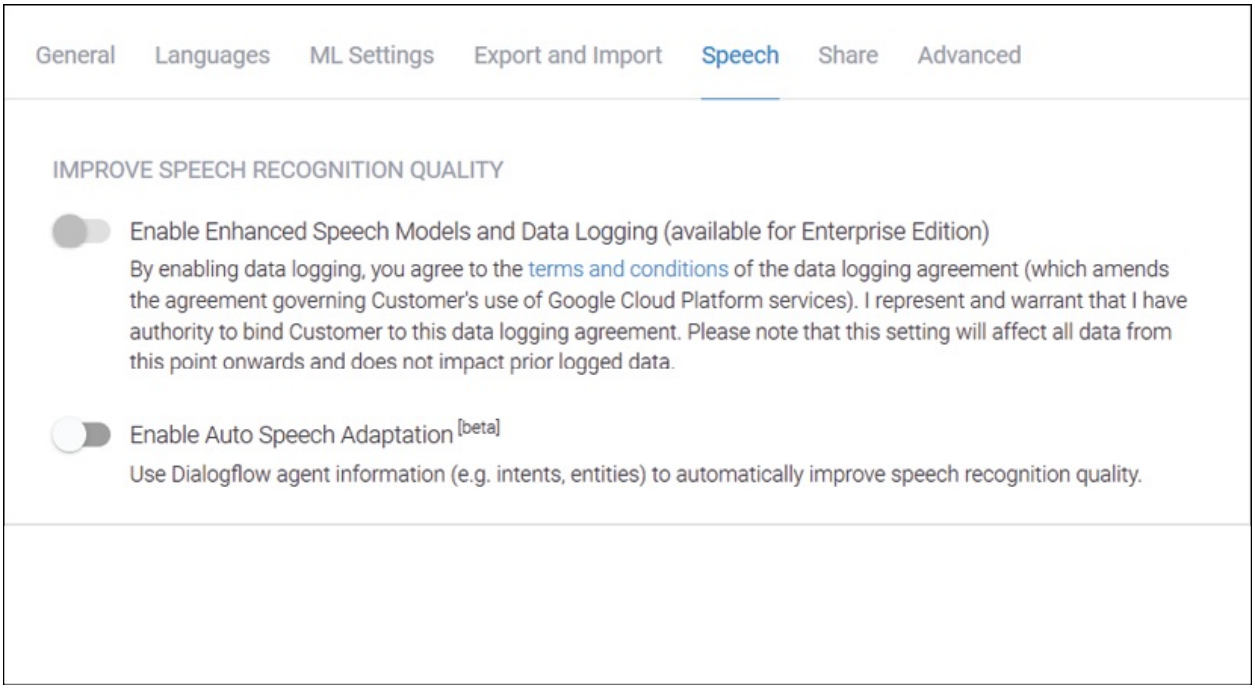


Figure 15.38: Speech options

We will first configure the main settings of text-to-speech:

- **Agent language**: Start with `en(English)` to reach the largest audience. However, bear in mind that Dialogflow produces good voice results in several languages.
- **Voice**: Start with `Automatic` before trying the different WaveNet model variations. WaveNet models build voices from scratch with neural networks.
- **Speaking rate**: You can leave it at `1`, or accelerate the rate or slow it down. For sports commentaries, for example, it could be faster.
- **Pitch**: You can make the voice higher or lower in semitones.
- **Volume gain**: You can reduce or increase the volume. The best is to leave it at `0` to start. Then, while testing the agent, see if it needs to be changed.

Once these parameters are set, save the model:

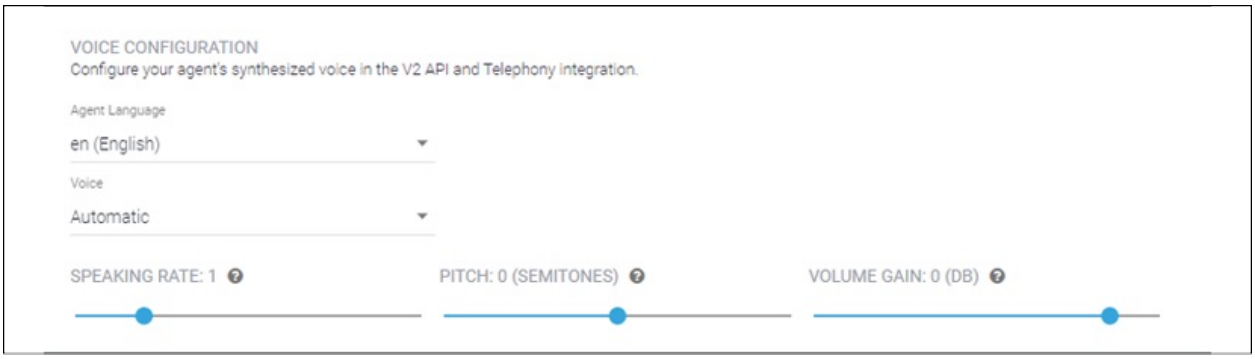


Figure 15.39: Voice configuration

Now, test your configuration or experiment with different settings by entering a sentence and clicking on the **PLAY** button:

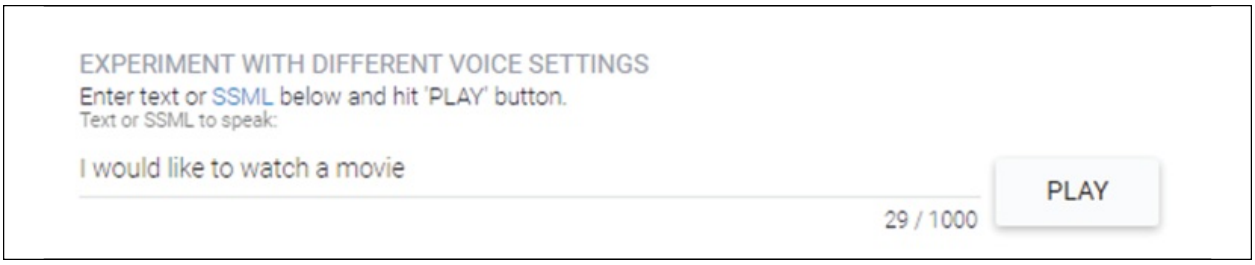


Figure 15.40: Experimenting with different voice settings

Once we have finished setting up the voice parameters, we need to set up the spelling machine learning features.

Spelling

We have one step left before we explore the possibilities and limits of the machine learning algorithms provided by Google. The machine learning spelling feature needs to be activated.

For that, we are going to click on the **ML Settings** tab, activate **AUTOMATIC SPELL CORRECTION**, and define an acceptable threshold below which our agent will refuse to recognize errors.

Click on the settings button next to the name of the agent and then on **ML Settings**:

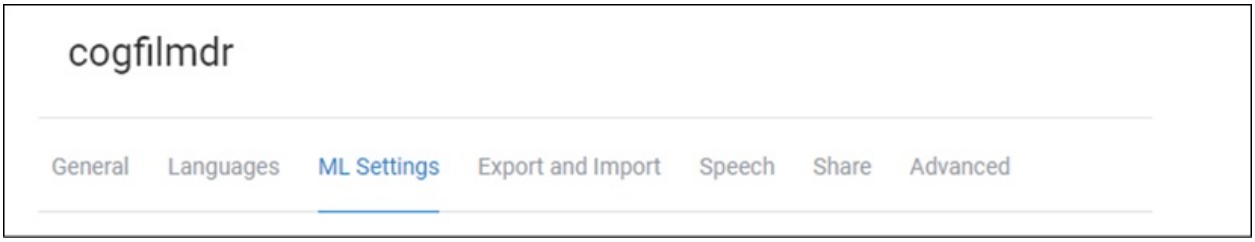


Figure 15.41: ML Settings tab

You will have access to a variety of options to work around a user's spelling mistakes. It works like the spelling corrector of a search engine when we mistype our request and Google, for example, suggests the correct spelling.

The options are as follows:

- **ML CLASSIFICATION THRESHOLD** determines a confidence score below which an intent will not be triggered unless there is a fallback intent (a general response).
- **AUTOMATIC SPELL CORRECTION** uses machine learning to correct user spelling mistakes. It should be activated.
- **AUTOMATIC TRAINING** may slow the dialog down, so careful use of this function is recommended.
- **AGENT VALIDATION** automatically validates an agent during the training process. Notice that training is triggered every time you save an intent, for example.

The following screenshot shows default values you might want to start with:

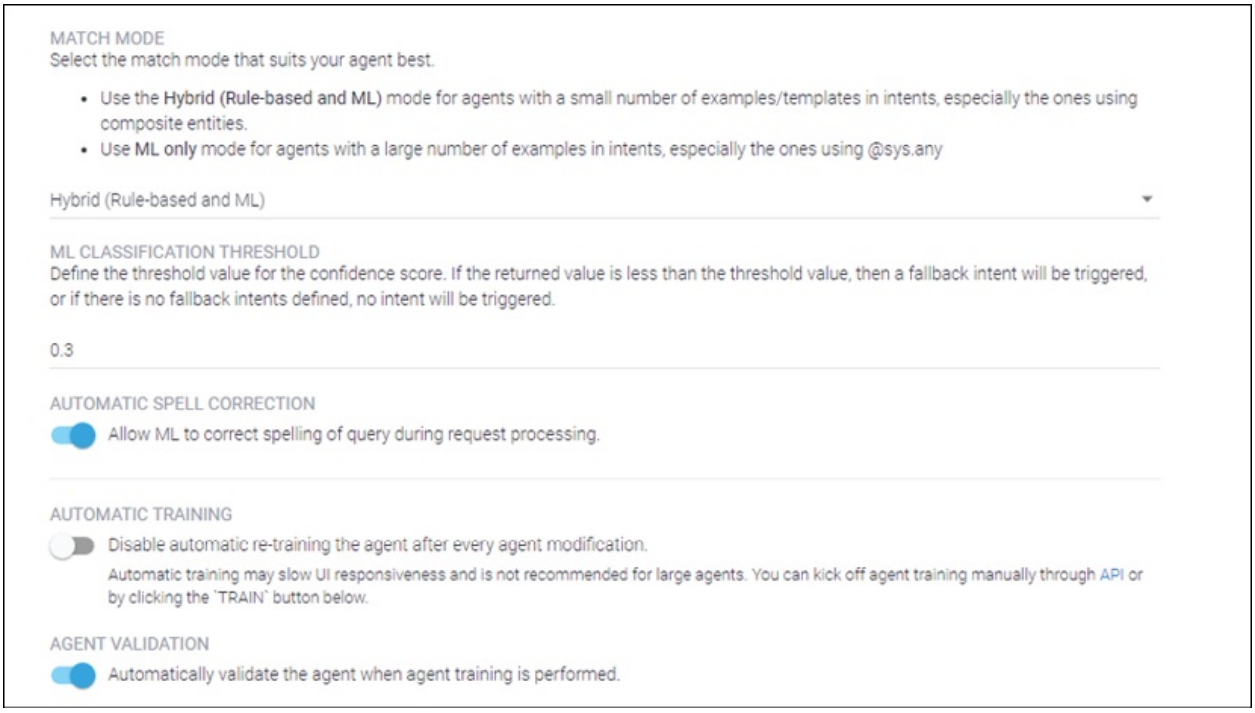


Figure 15.42: ML Settings options

Click on the **TRAIN** button every time you change an option.

Why are these machine learning algorithms important?

If it's just an educational chatbot like the `cogfilmdr` agent example, mistakes are acceptable. It's unpleasant, but acceptable. As we are going to see in *Chapter 16, Improving the Emotional Intelligence Deficiencies of Chatbots*, if a chatbot is going to be used by a large number of people, this means many days of training, tests, and creating workarounds to the machine learning limits of these functions. And that's just for one language!

If we are deploying in several languages, this means many days times the number of languages! Even with machine learning, it's tough work. Without machine learning, it's impossible.

Machine learning is not merely important; it is vital:

- If the chatbot cannot recognize a written utterance because of a simple

spelling mistake, we will get complaints, bad comments, and the SEO ship will sink.

- If the chatbot cannot recognize what you are saying on Google Home or any smart speaker, then that means a lot of trouble, maybe even a refund.
- If the chatbot's answer comes out in a phony voice that sounds like a 20th century robot, then nobody will want to use it.

Machine learning in chatbots is here to stay, but there are improvements to make in terms of emotional intelligence, like we must now explore in the next chapter.

Summary

Google Dialogflow provides a complete set of tools on Google Cloud to build a chatbot, add services to it, and customize it with your cognitive programs.

A good chatbot fits the requirements. Thinking the architecture through before starting development avoids underfitting (not enough functionality) or overfitting (functions that will not be used) of the model.

Careful AI preparation, as accomplished in *Chapter 14*, *Preparing the Input of Chatbots with Restricted Boltzmann Machines (RBMs) and Principal Component Analysis (PCA)*, provides a solid basis for a chatbot by making the path from the start of the dialog to the goal of the dialog much shorter and efficient.

Determining the right intent (what is expected of the chatbot), determining the entities to describe the intent (the subsets of phrases and words), and creating a proper dialog will take quite some time.

If necessary, adding services and specially customized machine learning functions will enhance the quality of the system. CUI with speech recognition, voice dialogs, and spelling correction features makes a chatbot frictionless to use.

The dialog we built in this chapter was based on *yes* answers. We supposed that the probabilities generated with the RBM and PCA in *Chapter 14* were correct.

However, humans are not easily confined to stereotypes.

The *Chapter 16, Improving the Emotional Intelligence Deficiencies of Chatbots*, explores emotional intelligence through basic concepts, Dialogflow functions, and a cognitive approach to improve the content of a chatbot.

Questions

1. Can a chatbot communicate like a human? (Yes | No)
2. Are chatbots necessarily AI programs? (Yes | No)
3. Chatbots only need words to communicate. (Yes | No)
4. Do humans only chat with words? (Yes | No)
5. Humans only think in words and numbers. (Yes | No)
6. Careful machine learning preparation is necessary to build a cognitive chatbot. (Yes | No)
7. For a chatbot to function, a dialog flow needs to be planned. (Yes | No)
8. A chatbot possesses general AI, so no prior development is required. (Yes | No)
9. A chatbot translates fine without any function other than a translation API. (Yes | No)
10. Chatbots can already chat like humans in most cases. (Yes | No)

Further reading

- For more on Google Dialogflow, refer to this link: https://dialogflow.com/
- For more on chatbots and UI development, refer to this link: https://www.packtpub.com/application-development/hands-chatbots-and-conversational-ui-development

16

Improving the Emotional Intelligence Deficiencies of Chatbots

Emotions remain irrational and subjective. AI algorithms never forsake rationality and objectivity. Cognitive dissonance ensues, which complicates the task of producing an efficient chatbot.

In *Chapter 14*, *Preparing the Input of Chatbots with Restricted Boltzmann Machines (RBMs) and Principal Component Analysis (PCA)*, we built a rational chained algorithm process with an RBM and a PCA approach. From there, we extracted critical objective data on a market segment. From that market segment and its features, we then designed a dialog in *Chapter 15*, *Setting Up a Cognitive NLP UI/CUI Chatbot*. The dialog was rational, and we produced a probable choice of services for the user. We did this out of good faith, to make the path from a request to its outcome as short as possible. It was a *yes* path in which everything went smoothly.

In this chapter, we will confront human nature with unexpected reactions. A *no* path will challenge our dialog. One of the problems we face resides in emotional polysemy, confusing emotional signals from a user.

We will address the *no* unexpected path with information drawn from *Chapter 14*, *Preparing the Input of Chatbots with Restricted Boltzmann Machines (RBM) and Principal Component Analysis (PCA)* and *Chapter 15*, *Setting up a Cognitive NLP UI/CUI Chatbot*, and go into the world of data logging.

Data logging will provide critical contextual data to satisfy the user. The goal will be to create emotions, not just react randomly to a user's emotional state.

Finally, we will open the door to researching ways to generate text automatically

through RNN-LSTM approaches. The idea will be to create automatic dialogs in the future based on data logging.

The following topics will be covered in this chapter:

- Emotional polysemy
- Small talk
- Data logging
- Creating emotions
- Exploring RNN-LSTM approaches

We will first explore the difference between simply reacting to emotions and creating emotions.

From reacting to emotions, to creating emotions

Designing a chatbot that reacts to what a user expresses is one thing. But creating emotions during a dialog like a human does requires deeper understanding of how a chatbot manages emotions. Let's start with emotional polysemy.

Solving the problems of emotional polysemy

We will be enhancing the emotional intelligence of a chatbot starting by addressing the issue of emotional polysemy. We are used to defining polysemy with words, not emotions, in the sense that polysemy is the capacity of a word to have multiple meanings. In *Chapter 6, How to Use Decision Trees to Enhance K-Means Clustering*, we explored the confusion that arose with the word "coach." "Coach" can mean a bus or a sports trainer, which leads to English to French translation issues.

Polysemy also applies to the interpretation of emotions by artificial intelligence.

We will explore this domain with two examples: greetings and affirmations.

Then we will go through the speech recognition and facial analysis as silver bullet solutions fallacies that mislead us into thinking it's easy to read emotions on faces.

The greetings problem example

To implement this chapter, open Dialogflow and go to the agent named `cogfilm+<your unique ID>` created in *Chapter 15, Setting Up a Cognitive NLP UI/CUI Chatbot*.

Suppose somebody types "Hi" to a chatbot agent. Almost everybody will think that this is a good beginning. But is it really?

Let's explore some of the many possible interpretations:

- **"Hi" meaning the person is very tense and irritated**: This could be the case of a top manager who never uses "Hi" to say hello, does not like chatbots, or doubts that the one that is being tested is worth anything at all.

  This manager usually says, "Good morning," "Good afternoon," and "Good evening."

- **"Hi" meaning "So what?"**: It is more like, "Yeah, hi." This could be a person, P, who dislikes person Q who just said good morning to P.
- **"Hi," meaning "I'm in trouble."**: This could be a usually chirpy, happy person who says, "Hello, everyone. How are things going today?" But today, it's just a brief "Hi." This will trigger alert reactions from others, such as "Are you okay?," "Is something wrong?"
- **"Hi," meaning "I'm trying to be nice."**: This could be a person that is usually grumpy in the morning and just sits down and stares down a laptop until the caffeine in their coffee kicks in. But today, this person comes in totally in shape, wide awake, and says, "Hi." This might trigger alert reactions from others such as, "Somebody had a great evening or night! Am I wrong?", with some laughter from the others.

I could go on with literally hundreds of other situations and uses of "Hi." Why? Because humans have an indefinite number of behaviors that can be reflected in

that first "Hi" in an encounter.

This could apply to ending a conversation without saying "bye" or saying it in many ways. The way a person says goodbye to another person in the morning can have an incredible number of significations.

This is therefore one of our challenges. Before we go further with this, let's look at one more challenge by considering the affirmation example.

The affirmation example

Suppose somebody types or says "Yes" in a chatbot. Does that really mean "Yes"?

- **"Yes", as in "Yeah, whatever."**: The user hates the chatbot. The dialog is boring. The user is thinking that if they do not say "Yes" and get it over with, this dialog will go on forever.
- **"Yes", as in "I'm afraid to say no."**: The user does not want to say "Yes." The question could be, "Are you satisfied with your job?" The user could fear the answers are logged and monitored. The user fears sanctions. Although this person hates their job, the answer will be "Yes" or might even be "I sure do!"
- **"Yes" as a good faith "yes" that a person regrets right after**: A person says "Yes" to a purchase, stimulated by the ad pressure at that moment in the chatbot. But minutes later, the same person thinks, "Why did I say yes and buy that?" Therefore, some platforms allow refunds even before the product or service is delivered.

Just as for "Hi," I could list hundreds of situations of emotional polysemy with "Yes."

Now, that we have understood the challenge at hand, let's explore the silver bullet fallacies mentioned previously.

The speech recognition fallacy

Many editors and developers believe that speech recognition will solve the problem of emotional intelligence by detecting the tone of a voice.

However, emotional polysemy applies to the tone of voice, as well. Human beings tend to hide their emotions when they feel threatened, and open up when they trust their environment.

Let's go back to the "Hi" and "Yes" examples.

"Hi" in a chirpy tone: A person, X, comes into an office, for example. Somebody says, "Oh, hi there! Great to see you!" Person Y answers "Hi" in a very happy tone. Google Home or Amazon Alexa, in their research lab, produces 0.9 probability that the conversation is going well.

This could be true. Or it could be false.

For example, person Y hates person X. Person X knows it and says, "Great to see you!" on purpose. Person Y knows that person X knows that they hate each other but won't give in to bursting out first. So person "Y" answers "Hi" in a super-happy tone.

At that point, many turn to facial analysis.

The facial analysis fallacy

Emotional polysemy also applies to facial analysis. Many think that deep learning facial analysis will solve the polysemy problem.

I saw a post recently by a developer with a picture of an obviously forced smile with the text stating that happiness could be detected with DL facial analysis!

Let's take two basic facial expressions and explore them: a smile and a frown. By now, you know that emotional polysemy will apply to both cases.

A smile

If somebody smiles and a DL facial analysis algorithm detects that smile, it means the person is happy. Is this true? Maybe. Maybe not.

Maybe the person is happy. Maybe the smile is ironic, meaning "Yeah, sure, dream if you want, but I don't agree!" It could mean "Get out of my way," or, "I'm happy because I'm going to hurt you," or, "I'm happy to see you." Who

knows?

The truth is that nobody knows, and sometimes even the person that smiles doesn't know. Sometimes a person will think, "Why did I smile at her/him? I hate her/him!"

A frown

If somebody frowns and a DL facial analysis algorithm detects that frown, it means the person is sad or unhappy. Is that true? Maybe. Maybe not.

Maybe the person is happy that day. Things are going smoothly, and the person just forgot a book, for example, at home before coming to this location. Maybe the second after the person will smile, thinking, "So what? It's a great day and I don't care!"

Maybe the person is unhappy. Maybe the person is having a great time watching some kind of ball game, and their favorite player missed something. The second after, the person thinks "Oh, so what? My team is winning anyway," and smiles. Some people just frown if they're thinking hard, but it doesn't mean they're unhappy.

We can now see that there are thousands of cases of emotional polysemy that occur with words, tone of voice, and facial expressions, and therefore there is no magical solution that is going to suddenly overcome the inherent difficulty that AI have when it comes to interpreting people's emotions.

We will now explore some realistic solutions to this problem.

Small talk

Small talk is not a silver bullet to solve the emotional intelligence problem of chatbots at all. In fact, even without speaking about chatbots, we all suffer from emotional distress in one situation or another. Small talk adds little unimportant phrases to a dialog such as "wow," "cool," "oops," "great," and more.

We do not need to seek perfection, but show goodwill. Every human knows the difficulty of emotional intelligence and polysemy. A human can accept an error

in a dialog if goodwill is shown by the other party to make up for that error.

Small talk is a little step in making amends to show goodwill.

To achieve the "making customers happy" purpose, scroll down the main menu to **Small Talk**, click on that option, and enable it, as shown in the following screenshot. We will be focusing on **Courtesy** and **Emotions**:

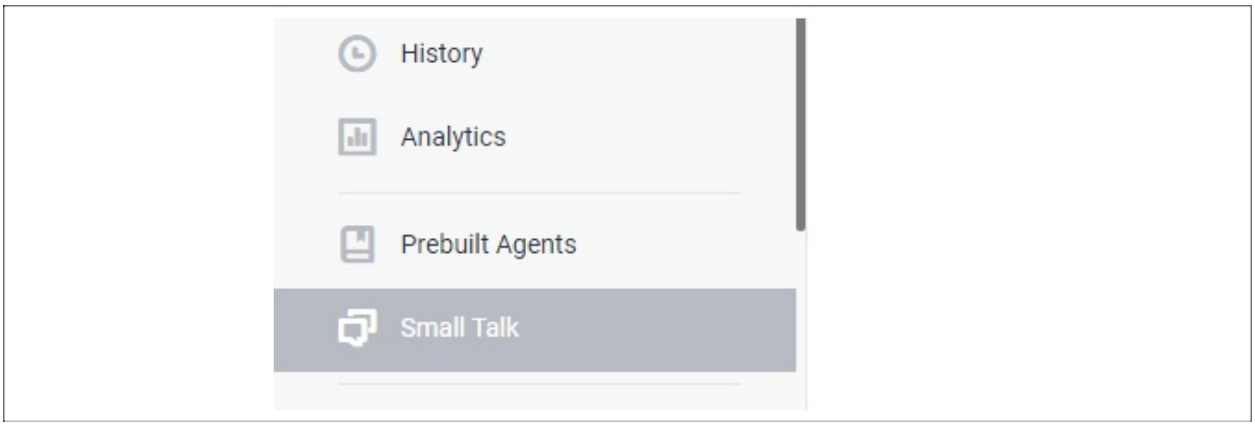


Figure 16.1: Small Talk in menu

Courtesy

Courtesy will help make a conversation smoother when things go wrong. Emotional intelligence is not answering 100% correctly every time.

Emotional intelligence (EI) is adjusting to an environment, correcting a mistake made, and trying to ease the tension at all times.

First, click on **Enable**, which will trigger small talk responses during a dialog:



Figure 16.2: Enabling Small Talk

You will notice that the **Courtesy** progress bar is at 0%. We need to increase EI:



Figure 16.3: Small talk themes

We will carefully answer the first possible "question" a user can ask or a phrase they might express: **That's bad.**

We are in trouble here! This is the worst-case scenario. We are going to have to work hard to make up for this.

Emotional polysemy makes the situation extremely difficult to deal with. The one thing we do not want to do is to pretend our bot is intelligent.

I would recommend two courses of action:

First, answer carefully, saying that we need to investigate this with something like:

I am very sorry about this. Could you please describe why it's bad? We will regularly check our history log and try to improve all the time. You can also send us an email at <your customer service email address>. We will answer as soon as possible.

You can enter this answer as follows and click on the **SAVE** button:

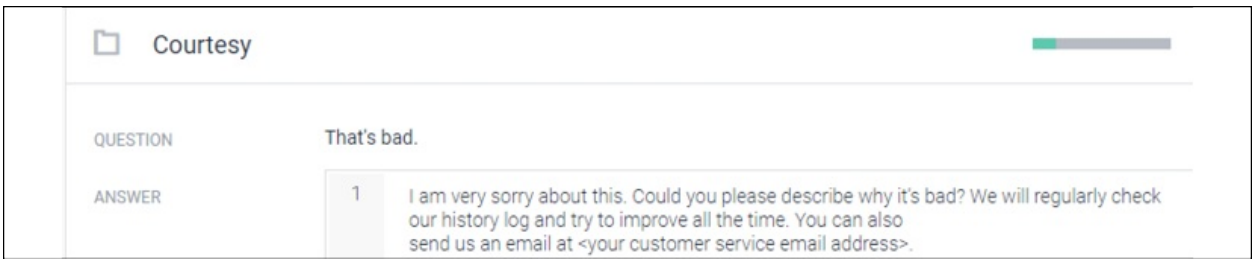


Figure 16.4: Courtesy

You will notice the **Courtesy** progress bar has jumped up to 17%. We have covered a critical area of a dialog. Default answers are provided when we don't fill everything in, but they are random, which makes it better to enter your own phrases if you activate this function.

Now test the dialog by entering "That's bad" in the test console at the top right. You will see the response appear:

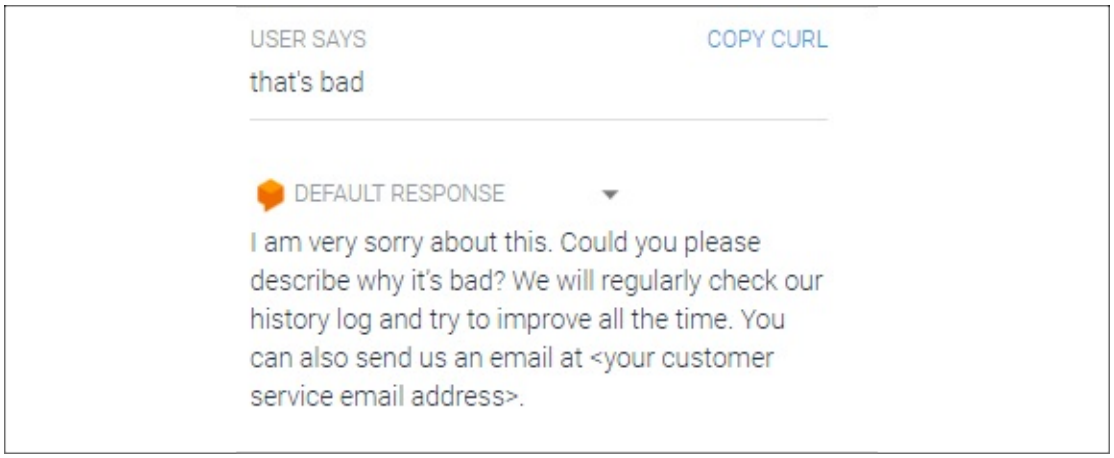


Figure 16.5: Default response

If you type "bad" instead of "That's bad," it will work too, thanks to the ML functionality of Dialogflow:

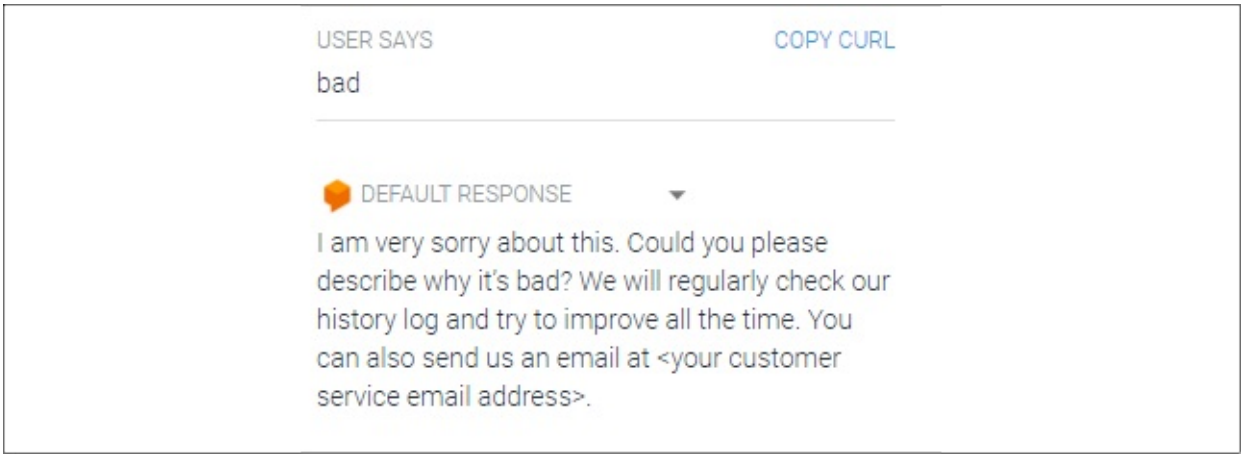


Figure 16.6: Default response

Data logging will tremendously help to boost the quality of a chatbot.

We will explore data logging in the next section. But let's check our emotions

first.

Emotions

We will deal with the first reaction: **Ha ha ha!** If we go back to emotional polysemy issues, knowing the user can say this at any time, we are in trouble again!

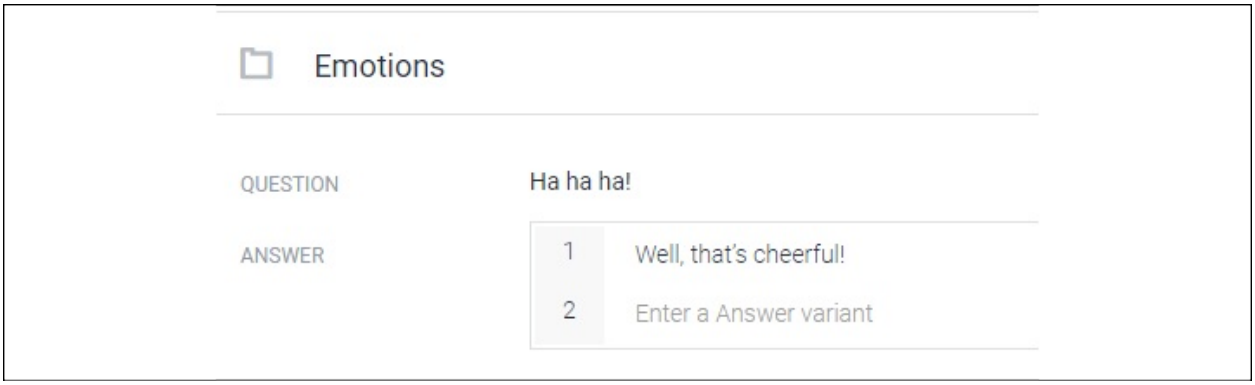


Figure 16.7: Managing Emotions

Is the user happy, or are they making fun of the chatbot? Who knows? Even with facial analysis and tone analysis, a quick "Ha ha ha!" is very difficult to interpret.

I would suggest a careful low-profile answer such as "Well, that's cheerful!", for example.

This will get the user to think that the chatbot has a sense of humor. When you click on **SAVE**, the **Emotions** progress bar will jump up.

You will notice that beyond the variants Dialogflow detects, you can also enter variants directly in your responses. Also, if the user enters a phrase that is not in the dialog, there is a fallback intent in the intents list.

Small talk might make a dialog smoother, but it is only one of the components of emotional intelligence, in a chatbot or in everyday life.

Data logging will take us a step further.

Data logging

In *Chapter 15, Setting Up a Cognitive NLP UI/CUI Chatbot*, we took the context of a dialog into account using follow-up intents. However, even follow-up intents will not provide solutions to unexpected answers on the part of a user.

To enhance a dialog, data logging will create a long-term memory for the chatbot by remembering the key aspects of a dialog.

A user and a Dialogflow designer have to agree to the terms of the Google Dialogflow data logging features, as described on this page: https://cloud.google.com/dialogflow/docs/data-logging.

Privacy is a serious matter. However, you will notice that when you use a search engine for a given product, you end up viewing or receiving ads related to the search. This is data logging.

Making this decision depends on your goal and target audience. Suppose the user accepts the terms of the agreement. Now, data logging is activated. Then, data logging will provide the chatbot with long-term memory.

The rest of this chapter explores data logging, with the assumption of it having been clearly accepted by the user.

Google Cloud, like all chatbot platforms (Amazon, Microsoft, and others), offers logs to improve chatbots. Many functions, interfaces, and services provide great support to boost the quality of dialogs.

Data logging can drive cognitive-adaptive dialogs beyond speech recognition tasks.

We will explore one way of doing this through the history of a dialog. Go to **History**:

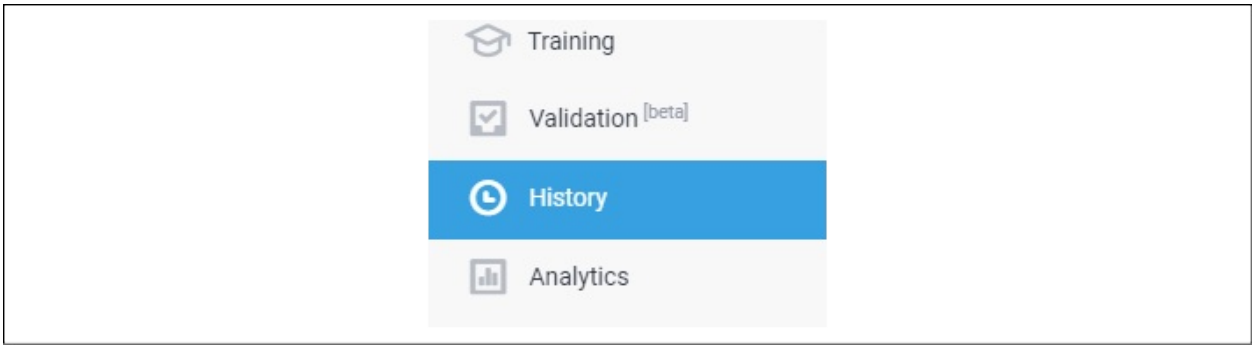


Figure 16.8: Dialog history option in the menu

You will see a list of past conversations:

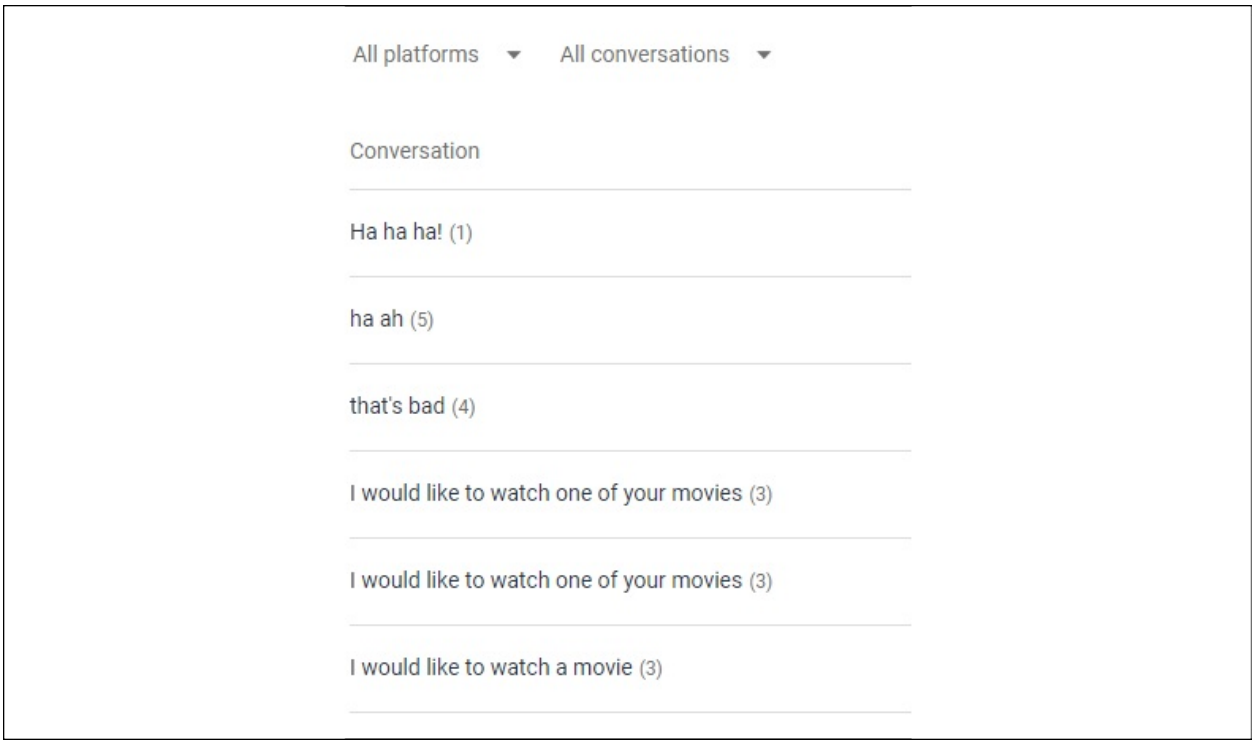


Figure 16.9: Dialog history

Notice the **All platforms** list, which contains information for Google Assistant and other platforms. You can deploy your chatbot by clicking on **See how it works on Google Assistant** on the right-hand side of the screen. From there, you can follow the instructions and have it running on smartphones, Google Home, and elsewhere. Also, you will have advanced log data to improve the chatbot.

If you tested "That's bad" in the *Courtesy* section, the history of the interactions

will be present:

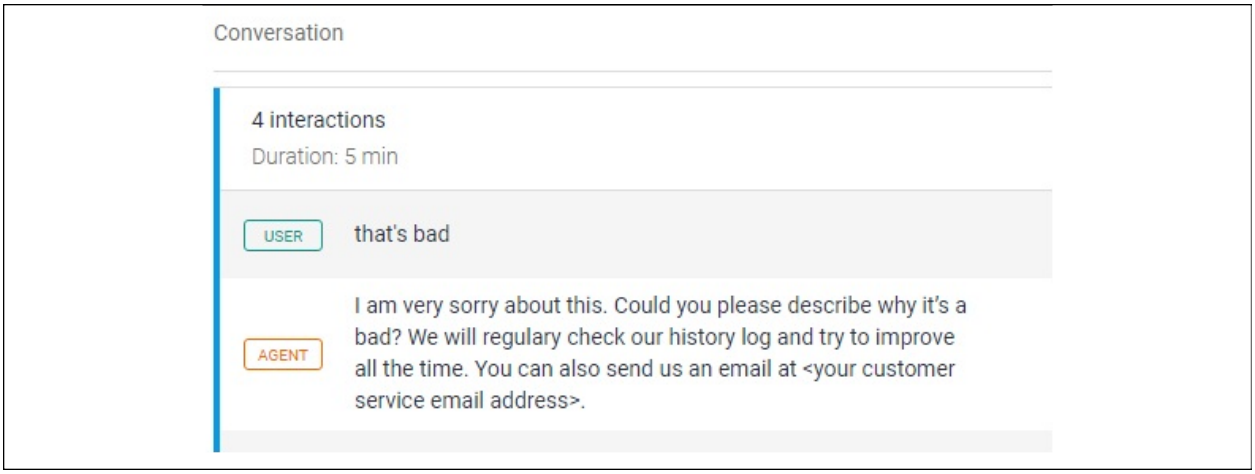


Figure 16.10: Chatbot interactions

One way to know the username is to ask the user their name when an issue comes up. This can come in handy to customize a dialog. We can thus have a special dialog for this person or this category of persons. We can thus ask the person to state their name in their response with an email address, for example. When we analyze the data logs manually or with scripts in the **Fulfillment** section, we can track the problem down and improve the chatbot on a personal level.

Having completed the **Small Talk** sections and then activated the data log authorization for your use of data logging, we can proceed to create emotions. Google will continue to improve our chatbot with our data logging features.

If we know which user said what, we can improve the dialog, as we will see in the next section.

Creating emotions

When the user enters ambiguous responses involving emotional polysemy, it is difficult for a chatbot to consider the hundreds of possibilities described in the previous sections.

In this section, we will focus on a user trying to obtain a service such as access

to a movie on a streaming platform.

An efficient chatbot should *create emotions in the user*. The most effective method is to:

- Generate *customer satisfaction*. Customer satisfaction is the ultimate emotion a chatbot should try to produce in a frictionless and expected dialog. If the customer is not satisfied with an answer, tensions and frustration will build up.
- Use functions such as the RBM-PCA approach of *Chapter 14*, *Preparing the Input of Chatbots with Restricted Boltzmann Machines (RBMs) and Principal Component Analysis (PCA)*, to suggest options that shorten the dialog path, thus its duration making the user "happy."

We will now explore the *no* path of the dialog encountered in *Chapter 15*, *Setting Up a Cognitive NLP UI/CUI Chatbot*.

To access the *no* path of the dialog, go to **Intents**, click on the **choose_movie** intent and click on **Add follow-up intent**, and click on **no** in the drop-down menu:



Figure 16.11: Adding a follow-up intent

A **choose_movie - no** option should now appear:

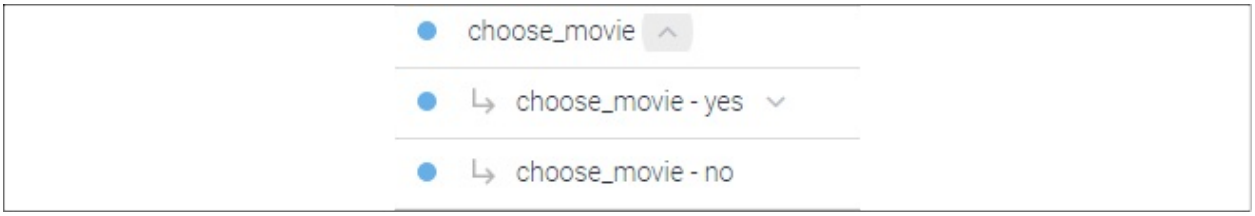


Figure 16.12: Follow-up options

Click on **choose_movie - no**.

Google has entered several default "no" variants, as shown in the following screenshot:



Figure 16.13: Dialogflow training phrases

This "no" response comes as a surprise to the chatbot. In *Chapter 14*, this market segment was explored. Something has gone wrong!

The chatbot was working on a specific market segment, the "action" superhero fan type viewer. The answer being "no" means that we need to examine the other features available.

The features in *Chapter 14*, *Preparing the Input of Chatbots with Restricted Boltzmann Machines (RBMs) and Principal Component Analysis (PCA)*, in the `RBM.py` program were:

```
    # Each column is a feature. There are 6 features:
    # ['love','happiness','family','horizons','action','vio
lence']
```

The "action" feature predicted so far groups several features:

Action = {happiness, action, violence}

The following features were not taken into account:

{love, family, horizons}

Since we want to keep the path short, we must find a way to ask a question that:

- Covers these three features
- Can use an existing feature matrix for another marketing segment

The viewer also may have:

- Recently seen enough action movies
- Progressively grown out of the superhero period of their life and be looking for other types of movies

In both cases, the viewer's market segment might overlap with another segment that contains family-love values.

As we saw in the *Adding fulfillment functionality to an agent* section in *Chapter 15, Setting up a Cognitive NLP UI/CUI Chatbot*, we can use a script to:

- Cover these three features
- Use an existing feature matrix for another marketing segment

Classical marketing segments take age into account. Let's continue in this direction and prepare for the possibility that the viewer, a young superhero fan, is growing a bit older and entering another age-movie-type segment that overlaps with the one used in `RBM.py` in *Chapter 14*:

```
    movies_feature_map = np.array([[1,1,0,0,1,1],
                                   [1,1,0,1,1,1],
                                   [1,0,0,0,0,1],
                                   [1,1,0,1,1,1],
                                   [1,0,0,0,1,1],
    .../...
```

We should add some love-family features in the matrix with the corresponding movies. We will then obtain another marketing segment. In the end, the chatbot will manage many marketing segments, which is the standard practice on many streaming platforms.

A variant of the chart in *Chapter 15, Setting Up a Cognitive NLP UI/CUI*

Chatbot, could be as follows:

MOVIE/FEATURE	LOVE	HAPPINESS	FAMILY	HORIZONS	ACTION	VIOLENCE
24H in Kamba	1	1	0	0	1	1
Lost	1	1	0	1	1	1
Cube Adventures	1	0	0	0	0	1
A Holiday	1	1	0	1	1	1
Jonathan Brooks	1	0	0	0	1	1
The Melbourne File	1	1	0	1	1	0
WNC Detectives	1	0	0	0	0	0
Stars	1	1	0	1	1	0
Space II	1	1	1	0	1	0
Zone 77	1	0	0	1	1	1

This feature matrix contains a movie with the missing features from the previous matrix: Space II.

A streaming platform contains many marketing segments:

$$M = \{s_1, s_2, \ldots s_n\}$$

Many of these marketing segments contain variants, merged features, combinations, and more.

Since data logging has been activated, from this point on we now have the following information:

- Whether this viewer has seen one of the several movies available in this marketing segment. This constitutes another tricky issue since some viewers may want to watch a movie again.
- The viewer's new marketing segment.

Building a chatbot for a streaming platform will take months of designing with many build possibilities. For this example, we will focus on the age progression scenario, keep the dialog path as short as possible, and provide the following response:

"Would you like to watch SPACE II? It's a blockbuster with a family that has an adventure in space. There is some action but it's mostly the story of a family that tries to survive in space."

Scroll down to the **Text Response** section and enter the response as follows, then click on **SAVE** to trigger the training process:



Figure 16.14: A look at the training process

If the viewer answers "yes," then the dialog will lead to the movie's page. To continue in this direction, go back to *Chapter 15*, *Setting Up a Cognitive NLP UI/CUI Chatbot*, and a "yes" follow-up exchange to this part of the dialog as you wish.

We have added some emotional intelligence to the agent. We will now explore the future of chatbot architecture through text augmentation with **recurrent**

neural networks (**RNNs**).

An RNN can process sequential data such as sequences of words, events, and more.

RNN research for future automatic dialog generation

The future of chatbots lies in producing dialogs automatically, based on data logging dialogs, their cognitive meanings, the personal profile of a user, and more. As RNNs progress, we will get closer to this approach. There are many generative approaches that can produce automatic sequences of sounds and texts. Understanding an RNN is a good place to start.

An RNN model is based on sequences, in this case, words. It analyzes anything in a sequence, including images. To speed the mind-dataset process up, data augmentation can be applied here, exactly as it is to images in other models.

A first look at its graph data flow structure shows that an RNN is a neural network like the others previously explored. The following diagram shows a conceptual view of an RNN:

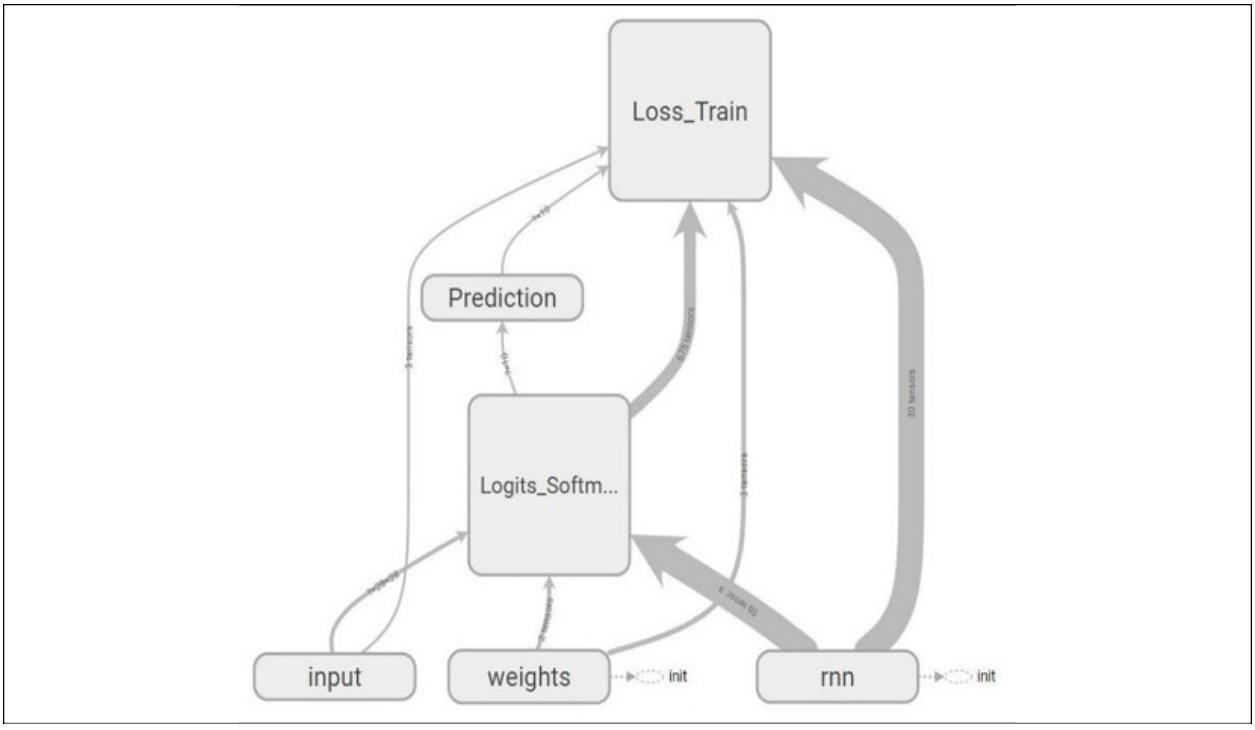


Figure 16.15: Data flow structure

The *y* inputs (test data) go to the loss function (**Loss_Train**). The *x* inputs (training data) will be transformed through weights and biases into logits with a softmax function.

Looking at the RNN area of the graph shows the following **basic_lstm_cell**:

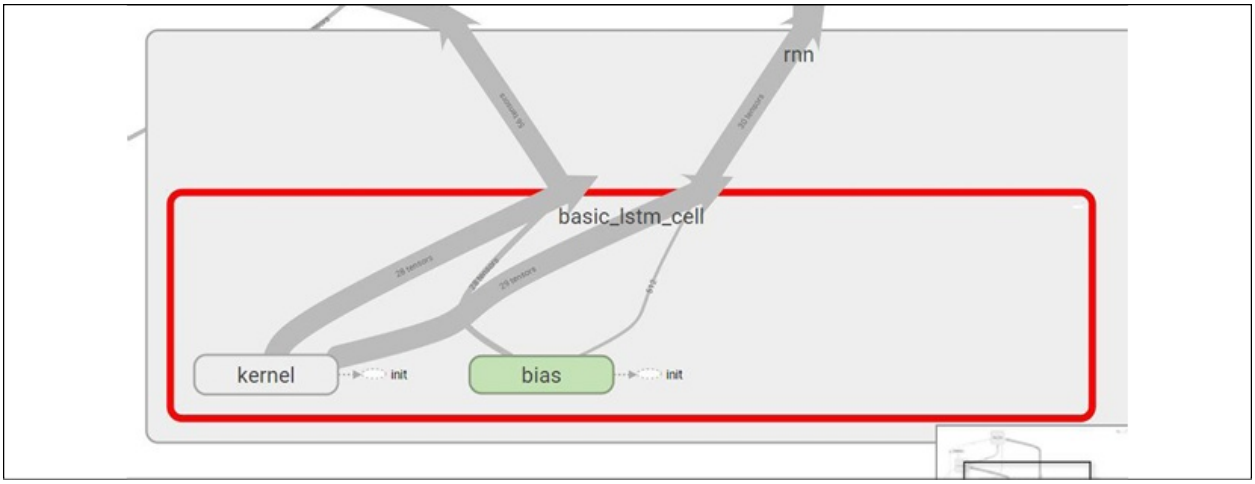


Figure 16.16: The basic_lstm_cell—the RNN area of the graph

The LSTM cell of an RNN contains "forget" gates that will prevent vanishing

gradients when the sequences become too long for an RNN unit.

RNNs at work

An RNN contains functions that take the output of a layer and feed it back to the input in sequences simulating time. This feedback process takes information in a sequence, for example:

The -> movie -> was -> **interesting** -> but -> I -> didn't -> like -> *it*

An RNN will unroll a stack of words into a sequence and parse a window of words to the right and the left. For example, in this sentence, an RNN can start with interesting (bold) and then read the words on the right and left (in italic). These are some of the hyperparameters of the RNN.

This sequence aspect opens the door to sequence prediction. Instead of recognizing a whole pattern of data at the same time, it is recognizing the sequence of data, as in this example.

A network with no RNN will recognize the following vector as a week, a pattern just like any other:

Monday

Tuesday

Wednesday

Thursday

Friday

Saturday

Sunday

An RNN will explore the same data in a sequence by unrolling streams of data:

Monday -> Tuesday -> Wednesday -> Thursday -> Friday -> Saturday ->

The main difference lies in the fact that once trained, the network will predict the word that follows; if Wednesday is the input, Thursday could be one of the outputs. This is shown in the next section.

RNN, LSTM, and vanishing gradients

To simulate sequences and memory, an RNN and an LSTM will use backpropagation algorithms. An LSTM is an improved version of RNN in some cases.

An RNN often has problems with gradients when calculating them over deeper and deeper layers in the network. Sometimes, it vanishes (too close to 0) due to the sequence property, just like us when a memory sequence becomes too long.

The backpropagation (just like us with a sequence) becomes less efficient. There are many backpropagation algorithms, such as vanilla backpropagation, which is commonly used. This algorithm performs efficient backpropagation because it updates the weights after every training pattern.

One way to force the gradient not to vanish is to use a ReLU activation function, $f(x) = \max(0, x)$, forcing values on the model so that it will not get stuck.

Another way is to use an LSTM cell containing a forget gate between the input and the output cells, a bit like us when we get stuck in a memory sequence, and we say "whatever" and move on.

The LSTM cell will act as a memory gate with 0 and 1 values, for example. This cell will forget some information to have a fresh view of the information it has unrolled into a sequence. In recent TensorFlow versions (2.0 and above), you can choose to use RNN or LSTM units in a layer. Your choice will depend on several factors. The key factor is the behavior of the gradient. If it vanishes in the RNN units, you might want to improve your model or move to LSTM units.

The key idea of an RNN to bear in mind is that it unrolls information into sequences, remembering the past to predict the future. The main idea of an LSTM relies upon its "forget" gate, avoiding the vanishing gradient. In TensorFlow 2.x, the choice of RNN or LSTM units can be made in a few lines.

Let's run an example on Google Colaboratory.

Text generation with an RNN

To view the program, log into your Dialogflow account, upload `text_generation_tf2.ipynb` (located in the `CH16` directory in the GitHub repository of this book) to your Google Colaboratory environment, and save it in your drive, as explained in the *Getting started with Google Colaboratory* section in *Chapter 13*, *Visualizing Networks with TensorFlow 2.x and TensorBoard*.

This TensorFlow authors' program has been well designed for educational purposes. The program starts by setting up TensorFlow 2.x and the necessary libraries.

In this section, we will thus focus on the main points of the program that you can then explore, run, and modify.

Vectorizing the text

The main entry step to an RNN consists of taking the sequence of words, the strings, and converting them into a **numerical representation**:

```
# Creating a mapping from unique characters to indices
char2idx = {u:i for i, u in enumerate(vocab)}
idx2char = np.array(vocab)
text_as_int = np.array([char2idx[c] for c in text])
```

We obtain a numerical value for each character:

```
{
  '\n':   0,
  ' ' :   1,
  '!' :   2,
  '$' :   3,
  '&' :   4,
  "'" :   5,
  ',' :   6,
  '-' :   7,
```

```
  '.' :   8,
  '3' :   9,
  ':' :  10,
  ';' :  11,
  '?' :  12,
  'A' :  13,
  'B' :  14,
  'C' :  15,
.../...
```

You will notice that this "dictionary" can be interpreted in two ways:

- character2number
- integer2character

The RNN will run its calculations but the predictions will come out in characters.

For example, the program can take the first sequence of the loaded text and produce the mapped integers of the text as follows:

```
# Show how the first 13 characters from the text are mapped
 to integers
print ('{} ---- characters mapped to int ---- > {}'.format(
    repr(text[:13]), text_as_int[:13]))
```

In this example, the result is:

```
'First Citizen' ---- characters mapped to int ---- > [18 47
56 57 58  1 15 47 58 47 64 43 52]
```

The RNN will run through numerical sequences, integer segments, or windows of the text to train and then make predictions. To do this, the program creates examples and targets as for all neural networks that have training batches.

Building the model

Building neural networks with TensorFlow 2 has become so simple to write in a

few lines that you can even miss seeing them in the example programs!

Let's clarify some basic concepts before getting to those few lines:

- A **sequential** model contains a pile or stack of layers.
- **Embedding** takes the number of each character and stores it in a vector.
- **GRU** stands for gated recurrent unit. A GRU contains gates that manage hidden units, keeping some information and forgetting other information. An RNN GRU can sometimes get confused when the sequences become long and thus mismanage the gradient, which then disappears. The more efficient LSTM units are part of a recurrent network unit as well with feedback connections with a cell, an input gate, an output gate, and a forget gate. But in the end the choice of the types units will always be yours depending on the context of your project. In any case, the key concept to keep in mind is that recurrent networks manage sequences of data, keeping the past in mind while forgetting some information.
- A **dense** layer, in this case, is the output layer.
- A **timestep** is a predefined sequence length. In another model, it could be actual time if we are working on time-dependent data.

A sequential model is built in three layers only:

```
def build_model(vocab_size, embedding_dim, rnn_units, batch
_size):
    model = tf.keras.Sequential([
        tf.keras.layers.Embedding(vocab_size, embedding_dim
,
                                  batch_input_shape=[batch_
size,
                                                     None])
,
    tf.keras.layers.GRU(rnn_units,
                        return_sequences=True,
                        stateful=True,
                        recurrent_initializer='glorot_unifo
rm'),
    tf.keras.layers.Dense(vocab_size)
    ])
    return model
```

And that's it! You can replace the basic `rnn_units` with an LSTM layer if the model requires it during the training phase. Once the model is built, the model:

- Looks an embedding up, as in a "dictionary."
- Runs the GRU for a timestep.
- The dense layer will then generate **logits** (see *Chapter 2, Building a Reward Matrix – Designing Your Datasets*) to produce a prediction using a likelihood function, a probability distribution.

The following figure of the TensorFlow author's program sums the process up:

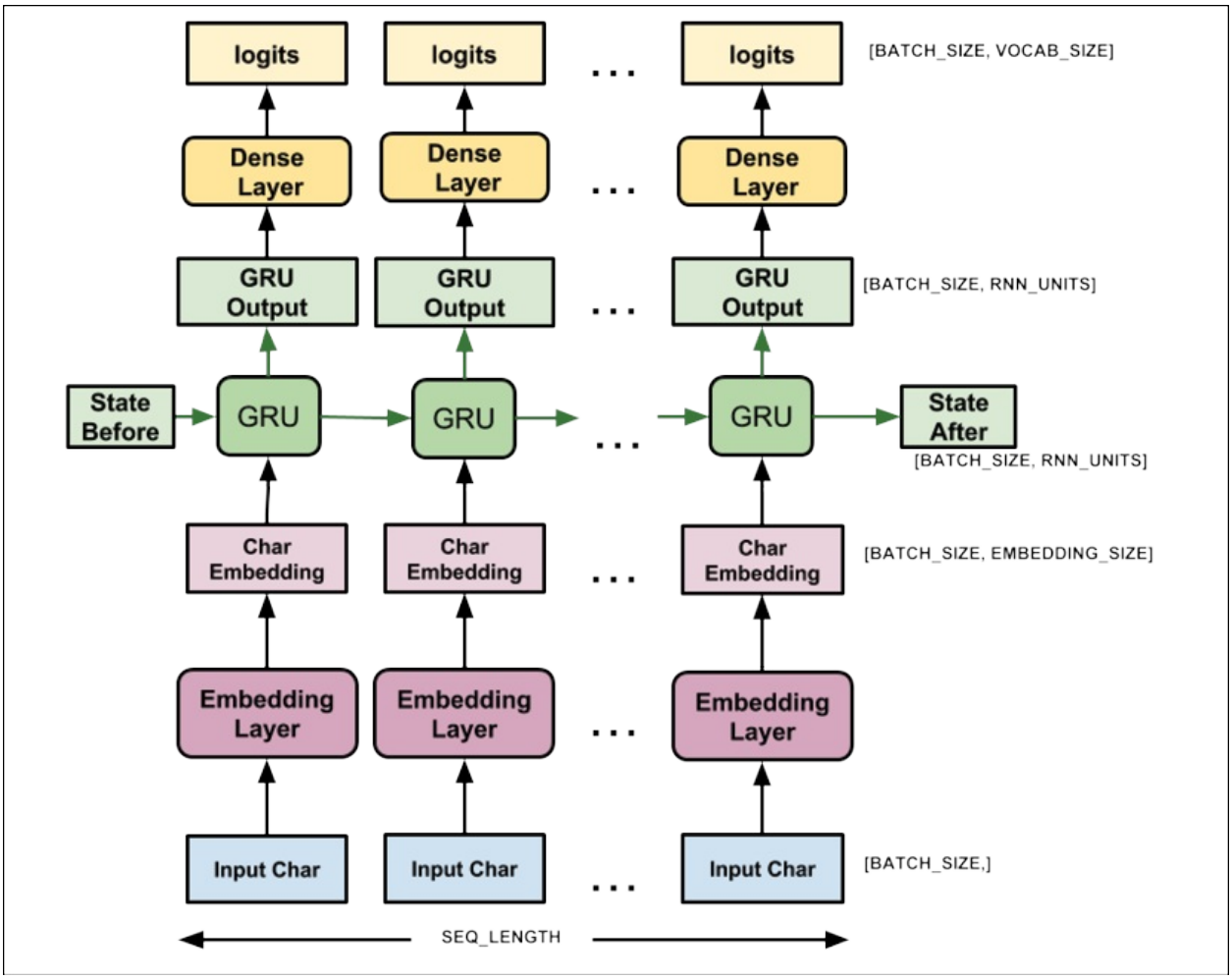


Figure 16.17: TensorFlow model

Generating text

After trying and training the model, the program will generate text automatically, for example:

```
print(generate_text(model, start_string=u"ROMEO: "))
```

As you'll notice, `ROMEO:` has been set up as the starting string. It then shows that the following predictions come from the initial text written by Shakespeare and are loaded at the beginning of the program:

```
ROMEO: Isick a tranch
It wast points for a sisten of resold thee, testement.
Petch doth my sweety beits are so of my sister.
KING RICHARD III:
Thou forget,
How did you burzenty day, 'tis oatly; heaven, for a womanou
s dear!
This is thy for mercy to the Kanging;
He that from the brothers of Gloucestersherding blame,
Thisble York, se me?
```

You can go back to the beginning of the program and change the URL. Instead of loading Shakespeare, change it to your own text:

```
path_to_file = tf.keras.utils.get_file('<YOUR FILE NAME>',
    '<YOUR URL>')
```

Before running the program, go to **Runtime** -> **Change runtime type**:

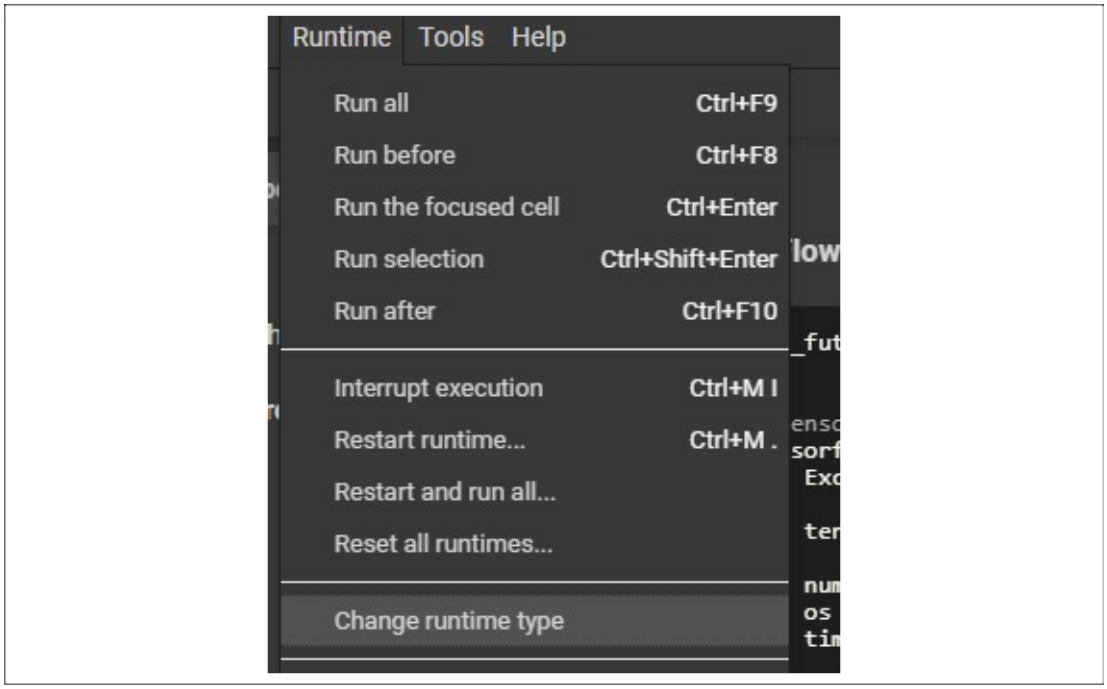


Figure 16.18: Runtime type

Click on **Change runtime type**:

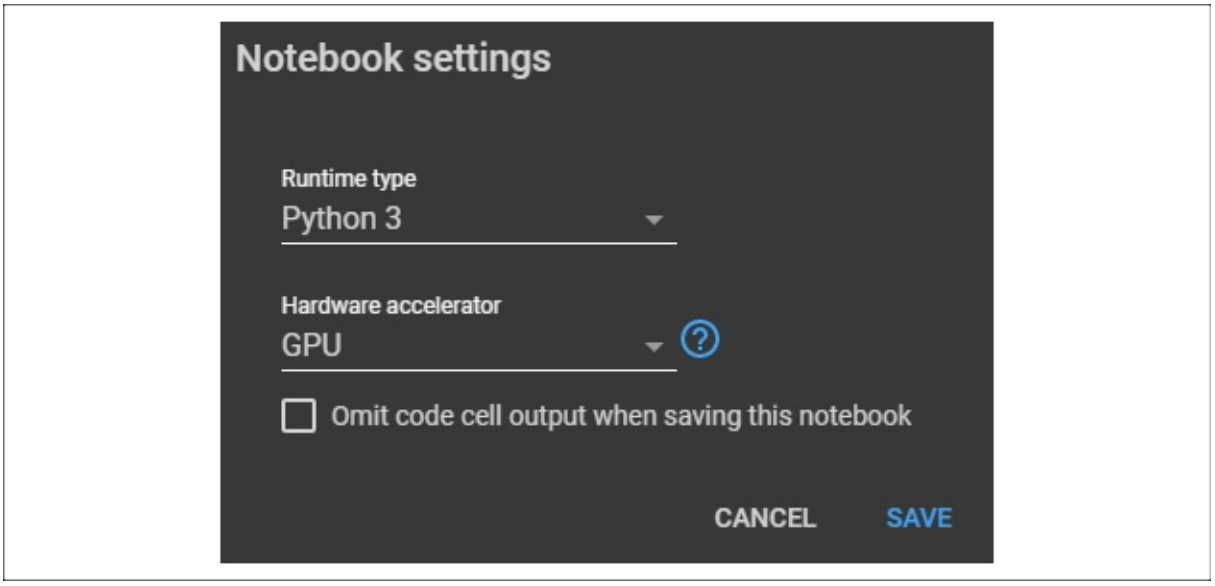


Figure 16.19: Notebook settings

I recommend using the GPU. Also, verify that **Omit code cell output when saving this notebook** is not checked if you want to save your notebook with the results produced when you run the program.

You are now ready to explore and do your own research to contribute to the future of automatic text generation!

Summary

Emotional polysemy makes human relationships rich and excitingly unpredictable. However, chatbots remain machines and do not have the ability to manage wide ranges of possible interpretations of a user's phrases.

Present-day technology requires hard work to get a cognitive NPL CUI chatbot up and running. Small talk will make the conversation smoother. It goes beyond being a minor feature; courtesy and pleasant emotional reactions are what make a conversation go well.

We can reduce the limits of present-day technology by creating emotions in the users through a meaningful dialog that creates a warmer experience. Customer satisfaction constitutes the core of an efficient chatbot. One way to achieve this goal is to implement cognitive functions based on data logging. We saw that when a user answers "no" when we expect "yes," the chatbot needs to adapt, exactly the way we humans do.

Cognitive data logging can be achieved through the preparation we explored in *Chapter 14*, *Preparing the Input of Chatbots with Restricted Boltzmann Machines (RBMs) and Principal Component Analysis (PCA)*, the cognitive dialog of *Chapter 15*, *Setting Up a Cognitive NLP UI/CUI Chatbot*, and the adaptive dialog built in this chapter. In our example, the viewer changed market segments, and the chatbot logged the new profile. Dialogflow-fulfillment scripts can manage the whole adaptive process, though that is beyond the scope of this book.

We looked at the study of sequences of data through RNNs eventually leading to automatic dialogs. Chatbots, using cognitive approaches such as the RBM-PCA and the adaptive data logging inferences of this chapter, will one day build their own dialogs.

The following chapters will explore ways to achieve higher levels of artificial intelligence through genes, biological neurons, and qubits. The next chapter

explores genetic algorithms and then implements them into a hybrid neural network.

Questions

1. When a chatbot fails to provide a correct response, a hotline with actual humans needs to take over the conversation. (Yes | No)
2. Small talk serves no purpose in everyday life or with chatbots. It is best to just get to the point. (Yes | No)
3. Data logging can be used to improve speech recognition. (Yes | No)
4. The history of a chatbot agent's conversations will contain valuable information. (Yes | No)
5. Present-day technology cannot make use of the data logging of a user's dialogs. (Yes | No)
6. An RNN uses sequences of data to make predictions. (Yes | No)
7. An RNN can generate the dialog flow of a chatbot automatically for all applications. (Yes | No)

Further reading

- Information on RNNs: https://www.tensorflow.org/tutorials/recurrent
- More on text generation: https://www.tensorflow.org/tutorials/text/text_generation

17

Genetic Algorithms in Hybrid Neural Networks

In this chapter and the following two chapters, we will explore the world inside us. First, in this chapter, we will use the model of our genes as an optimizing tool. In *Chapter 18*, *Neuromorphic Computing*, we will enter our biological brain activity and create neuromorphic networks. Finally, in *Chapter 19*, *Quantum Computing*, we will go even deeper and use the quantum material in us to build quantum mechanic models for quantum computing.

A slight change in any of these tiny entities (genes, neurons, qubits) within us can modify our whole existence.

In this chapter, we will discover how to enter our chromosome, find our genes, and understand how our reproduction process works. From there, we will begin to implement an evolutionary algorithm in Python, a **genetic algorithm** (**GA**).

Charles Darwin offered "survival of the fittest" as a model to represent evolution. In some ways, the model is controversial. In 21st century societies, we tend to provide support to those who are not the fittest, as best as possible. However, in mathematics, we do not have this ethical problem.

In AI, we need to provide an accurate solution. If we generate several solutions, we can apply the "survival of the fittest" to abstract numbers.

In some cases, GAs dramatically reduce the number of combinations required to find the optimal solution to a problem. By generating mathematical offspring, choosing the fittest, and producing new stronger abstract generations, the system often reaches a more optimal solution than propagating permutations.

A well-designed GA can optimize the architecture of a neural network, thereby producing a hybrid neural network.

The following topics will be covered in this chapter:

- Evolutionary algorithms; genetic algorithms
- Extending the genes of genetic algorithms to optimizing tools
- Hybrid neural networks
- Using a genetic algorithm to optimize an LSTM

Let's begin by first working to understand what evolutionary algorithms are.

Understanding evolutionary algorithms

In this section, we will drill down from our heredity down to our genes to understand the process that we will then represent while building our Python program.

Successive generations of humans activate some genes and not others, producing the wonderful diversity of humanity. A human lifetime is an episode in a long line of thousands of generations of humans. We all have two parents, four grandparents, and eight great-grandparents, which amounts to 2^3 ascendants. Suppose that we extend this line of reasoning to four generations per century and then over about 12,000 years when the last glacial period ended and the planet started warming up. We obtain:

- 4 * 1 century * 10 centuries = 1,000 years and 40 generations
- 40 generations * 12= 480
- Adding up to 2^{480} mathematical ascendants to anybody living today on the planet!

Even if we limit ourselves to 1,000 years, 2^{40}, that adds up to 1,099,511,627,776 ascendants a thousand years ago. But there is a problem. This figure is impossible! Today, we have reached the height of the human population, which is only 7,500,000,000. So, this means that our ascendants had many children who married their cousins of all degrees, making humanity one large extended

family, no matter what our skin color or hair color is!

Heredity in humans

First, men fertilize women when a male cell unites with a female cell. The fertilized egg grows and, after quite an adventure, is born and becomes one of us writing or reading this book.

To grow and live from being a fertilized egg to human adults, we must gather much of the outside world, absorb, and transform it.

This transformation of food and materials by us until we grow into something that is more or less like our ancestors is called heredity.

If life were calm and quiet, nothing noticeable would occur. However, our environment has exerted relentless pressure on us for millions of years, back to when we were just some kind of bacteria floating around in an ocean somewhere. That pressure brought about continuous natural selection; what worked would live on, whilst what didn't work would die out.

That pressure continues up to present day, forcing us, humans, to adapt genetically or disappear. Those humans who failed to adapt to face the pressure of their environment died out. We who live today have survived.

Evolution can be defined as a state of constant conflict. On one hand we have our relentless, often hostile environment. On the other, our equally relentless genes; many of which die out, but others morph, adapt, and continue on through heredity—indeed, they're doing that even now, and who knows what is going to happen next?

Our cells

In our cells, the nucleus contains personal biological data in the form of chromosomes. Our cells contain 46 chromosomes per cell, which, in turn, are formed by 23 pairs of chromosomes. One of these pairs is a sex cell to determine our sex.

Inside the chromosomes, we have genes, especially the mitochondrial genome,

which is the DNA in tiny cells that take our food and transform it into fuel for our cells. Each cell is a microscopic busy factory containing thousands of genes!

The human genome describes an incredible set of sequences for our building blocks contained in the 23 chromosomes of our cells.

How heredity works

Except for the sex cells, we inherit twenty-three of our mother's forty-six chromosomes and twenty-three of our father's chromosomes. In turn, our parents' cells contain the chromosomes of their parents—our grandparents—and so on in various proportions.

Let's take an example to view the complexity of what we are facing. We will take one of our parents, either our mother or father. *P* represents the set of chromosomes of that parent.

Letters with primes will represent their fathers'—our grandfathers'—chromosomes, and letters with double primes will represent their mothers'—our grandmothers'—chromosomes.

We can represent this as follows for your father in his sex cell:

$$P = \{A',\ B',\ C',\ D',\ E',\ F',\ G',\ H',\ I',\ J',\ K',\ L',\ M',\ N',\ O',\ P',\ Q',\ R',\ S',\ T',\ U',\ V'Y'$$
$$A'', B'', C'', D'', E'', F'', G'', H'', I'', J'', K'', L'', M'', N'', O'', P'', Q'', R'', S'', T'', U'', V''X''\}$$

For your mother, the last chromosome of the first set would be an *X* in her sex cell:

$$P = \{A',\ B',\ C',\ D',\ E',\ F',\ G',\ H',\ I',\ J',\ K',\ L',\ M',\ N',\ O',\ P',\ Q',\ R',\ S',\ T',\ U',\ V'X'$$
$$A'', B'', C'', D'', E'', F'', G'', H'', I'', J'', K'', L'', M'', N'', O'', P'', Q'', R'', S'', T'', U'', V''X''\}$$

Women are *X*-*X* and men *X*-*Y*.

Imagine the possibilities!

If we only take *A*, *B*, and *C* in only one of our parent's cells, we already obtain the following set, *C*, of the eight combinations we would inherit:

$$C = \{A'B'C', A''B''C'', A'B''C'', A''B'C', A'B'C'', A''B''C', A'B''C', A''B'C''\}$$

If we extend this to the twenty-three chromosomes, the distribution climbs up to 2^{23}, or 8,388,608 possibilities.

Our evolutionary process contains the right potential for evolutionary algorithms.

Evolutionary algorithms

In this section, we will drill down further into evolutionary algorithms, getting closer to our Python programs. Evolutionary algorithms can be used in any field in which combinations are useful: scheduling, medical research on DNA, weather forecasting, neural network architecture optimizing and a limitless number of domains.

Evolutionary computation is a set of algorithms that apply trial-and-error techniques to reproduce an abstract mathematical version of biological evolution. This mathematical model does not contend with having solved the difficult task of explaining evolution, which naturally cannot be reduced to a few equations.

However, our biological environment produces frameworks that, though the fruits of our imagination, enable us to create efficient abstract algorithms.

Evolutionary algorithms enter the category of *evolutionary computation*. An evolutionary algorithm contains processes such as mutation, crossover, and selection. Many models can achieve the goals set for an evolutionary process.

A GA introduces the category of an evolutionary algorithm.

We will first define the concepts involved in a GA, which are the building blocks of a Python program.

Going from a biological model to an algorithm

There are many ways of creating a GA model. You can reproduce the exact description of the human model described in the preceding section, you can simplify it, or you can create another view.

Our model will contain a set of genes in a chromosome and a population to interact with:

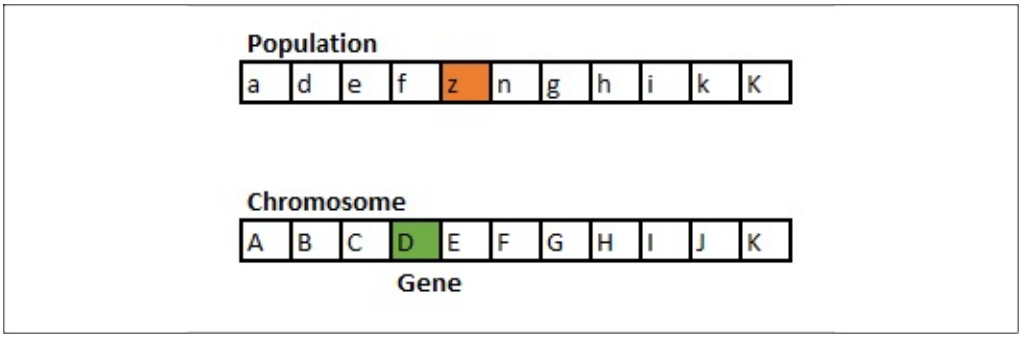


Figure 17.1: Chromosomes in genes

Our model is an abstract model for algorithms, not the actual representation of human chromosomes that come in pairs, for example. A gene in an individual's chromosome will interact with the gene set of the population. This process will be defined in the following sections.

Basic concepts

Let's first describe the concepts of the building blocks of our Python program:

- **Target**: Target defines the length and properties of the child we wish to obtain after *n* generations.

  In our model, the target can be specified or unspecified.

  A specified target contains a length and its value:

```
    target = "Algorithm" # No space unless specified as
 a character in the gene set
```

  In this case, our environment requires an exact gene set to remain fit in that environment. See the following fitness function.

  An unspecified target contains a length, but not its actual value, which must be found by the GA after *n* generations:

```
    target="AAAA" #unspecified target
```

In this case, our environment does not require an exact gene, but rather a gene with the features required to remain fit in that environment. See the following fitness function.

- **Population**: Population first defines the selection of an individual at random we will call **parent** that will contain a certain length of its string of genes:

```
def gen_parent(length)
```

Population also defines the potential individuals the parent can interact with to produce a child. In our genetic simulation, the size of the population is represented by the gene set of the population (see the following point).

- **Gene set of the parent**: The gene set of the parent will first be a random set of genes.
- **Gene set of the population**: This parent will then randomly encounter another person in the population with random gene choices. This population is represented by a gene set that we will draw genes from randomly:

```
geneSet = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQR
STUVWXYZ!.-"
```

Since a given population can possess any or several of these genes, our population is very large from the start.

- **Selection**: We will select genes randomly from our gene set to simulate the meeting of our parent with another parent:

```
    index=random.randrange(0,len(parent))
```

- **Crossover**: We will then form a child from the parent and the random gene taken from our population represented by the gene set.

In our model, once the random selection of the gene has been made, the gene will be replaced by a random gene taken from the population represented by the gene set of the population:

```
    if(newGene!=oldGene):childGenes[index]=newGene;
```

Notice that the new gene must be different from the old gene during the

crossover in our algorithm. Our algorithm thus avoids getting stuck in some local combination during the reproduction phase. Child after child, we will produce new generations.

- **Mutation**: We will not accept the same gene from a parent to form a child. If we detect this, we will randomly change a gene to make sure each generation is different.

  As described in the *Crossover* paragraph, we will not accept a child with the same genes as a parent. As seen in the heredity section of this chapter, it is unlikely that a child would inherit exactly all of the genes of a given parent. *Diversity* is the key to producing generation after generation of children that will adapt to their environment.

  In our model, a diversity rule forcing mutation has been introduced:

```
if(newGene==oldGene):childGenes[index]=alternate;
```

  We thus introduce an alternate gene, as we will see while building the Python program.

- **Child**: Child defines a set of genes that contains the same number of genes as the parent but with new genes. Generations of children of child gene strings will be produced and then complete the selection process with a fitness function.
- **Fitness**: Fitness defines the value of the child as defined in a given model. The fittest will then be selected to be the parent for the next generation.

  In our model, we have two fitness functions that we define in a scenario variable.

  If `scenario=1`, then a specified target scenario will be activated. The target will be specified to fit the surrounding environment.

  Polar bears became white, for example, to blend in with the surrounding snow and ice of their environment. In other areas, bears are often brown, for example, to blend in with the surrounding vegetation.

  The fitness function thus has a target to reach. For example, in one instance

in the program:

```
    target="FBDC"
```

This target could mean many things, as we will see. In the case of the bears, maybe these genes in one of their chromosomes trigger off their color: white or brown.

If our program simulating nature does not produce the right gene in a given generation, this means that the bear is not mutating correctly and will not survive. The fitness function of the Python program simulates nature by only keeping the strings of genes that make the child evolve in the right direction.

The specified target, in this case, is an identified string of genes that will make a life-and-death difference.

```
def get_fitness(this_choice,scenario):
    if(scenario==1):
```

As we will see later, `scenario==1` will make sure that the exact gene set required to survive is reached.

If `scenario=0`, then an unspecified target will be activated. The length of the target is not specified. However, a feature set will define the value of the gene set of the population. This feature contains numerical values that open the door to any optimization you wish to solve, as we will see in the Python program example. This numerical set is called the KPI set because the values are key performance indicators of the system we will explore in this model:

```
KPIset ="012345677201234567470123456998092345676701234
5671001234"   #KPI set
```

The `KPIset` feature set matches the size of the gene set of the population.

Building a genetic algorithm in Python

We will now build a GA from scratch using `GA.ipynb`.

You can use the `.py` version. The code is the same, although the line numbers in this chapter refer to the Jupyter notebook cells of the `.ipynb` version.

> At all times, you can go back to the previous section, *Basic concepts*, to consult the definitions used to describe the Python program in the following sections.

Importing the libraries

This program is built from scratch with no higher-level library to get the feel of a GA. Three lines are enough to get everything working:

```
import math
import random
import datetime
```

Calling the algorithm

In this program, we will be exploring three scenarios. Two scenarios generate specified targets, and one generates an unspecified target.

We will begin with a specified target and then move to the more advanced unspecified target that will prepare us for hybrid networks using a GA.

We will first go to the *Calling the Algorithm* cell of the program. The first task is to define which scenario and type of fitness function we will use on line 3:

```
scenario=0   # 1=target provided at start, 0=no target, gen
etic optimizer
```

If `scenario=1`, the program generates the correct exact genes for a child from several generations, starting with a random seed parent.

If `scenario=0`, the program generates the best features of a type of genes of a child from several generations, starting with a random seed parent.

In line 4, `GA=2` defines which target we are dealing with.

If `GA=1`, the gene set of the population and the target are defined. The main function is called:

```
if(GA==1):
    geneSet =
        "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWX
YZ!.-"
    # target with no space unless specified as a character
in the geneSet
    target = "Algorithm"  # No space unless specified as a
character in the geneSet
    print("geneSet:",geneSet,"\n","target:",target)
    ga_main()
```

At this point, the gene set of the population is printed along with the target:

```
geneSet: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWX
YZ!.-
 target: Algorithm
```

The last line calls `ga_main()`, the main function.

The main function

The code is in the `ga_main()` cell of the program.

The `ga_main()` function is divided into three parts: parent generation, child generation, and a summary.

The parent generation process

The parent generation runs from line 2 to line 7:

```
    startTime=datetime.datetime.now()
    print("starttime",startTime)
    alphaParent=gen_parent(len(target))
    bestFitness=get_fitness(alphaParent,scenario)
    display(alphaParent,bestFitness,bestFitness,startTime) #
no childFitness for generation 1 so default value sent
```

- `startTime` indicates the start time, which is printed:

```
starttime 2019-10-12 10:32:28.294943
```

- `alphaParent` is the first parent who will be created by the `gen_parent` function, which will be described in the following Jupyter notebook cell.
- `bestFitness` is the fitness estimated by the `get_fitness` function, which will be described in this chapter.
- `display` is the function that describes the result of this process.

We now have a process that creates a parent: `gen_parent`, `get_fitness`, and `display`.

We will now explore the parent generation function before resuming the main function.

Generating a parent

The parent generation function cell starts with zero genes and the length of the target:

```
def gen_parent(length):
```

`length` = the length of the target. The target is `"Algorithm"`, so `length=9`.

At the beginning of the process, the parent has no genes since the goal of this function is to produce a parent at random that contains a string of genes that is equal to the length of the target.

The string of genes to produce a parent is as follows:

```
    genes=[]                              #genes array
```

Now, a `while` loop starts on line 3 of the cell to fill `genes[]` until it reaches the target's length:

```
    while len(genes)<length:  #genes is constrained to the
```

```
length
        #sampleSize: length of target constraint
        sampleSize=min(length-len(genes),len(geneSet))
        #extend genes with a random sample the size of samp
leSize extracted from geneSet
        genes.extend(random.sample(geneSet,sampleSize))
```

- `sampleSize` is the sample of genes required from the gene set, `geneSet`, from which to choose a random gene for the parent.
- `genes.extend` adds a random gene to the `genes` array from `geneSet`.

Once the parent's gene set, `genes[]`, has reached the target's length, the `return` function sends the parent back to the main function, `ga_main()`, where it is displayed with the `display` function. The output of the parent in this random run was:

```
aFJPKzYBD
```

Naturally, the parent will be different at each run since this is a random process.

The string of genes is now returned to the `ga_main()` function:

```
    return ''.join(genes)
```

Now, let's explore the fitness function and the `display` function.

Fitness

At this point in the `ga_main()` function, the start time was printed out, and the parent created:

```
#I PARENT GENERATION
    startTime=datetime.datetime.now()
    print("starttime",startTime)
    alphaParent=gen_parent(len(target))
```

We need to evaluate the fitness of the parent before creating generations of children:

```
    bestFitness = get_fitness(alphaParent,scenario)
```

In this paragraph, we will only describe the specified target case, which is part of `scenario=1`. We will create a fitness function with the target of a given choice. In this case, only the sum of correct genes is calculated:

```
def get_fitness(this_choice,scenario):
    if(scenario==1):
        fitness = sum(1 for expected,
            actual in zip(target,this_choice) if expected==
actual)
```

- `this_choice` is the parent string of genes produced by the `gen_parent` function described in the preceding code snippet.
- `scenario` indicates whether the function calculates the sum of fit genes or evaluates the features of the genes. In this case, the sum of the correct genes is calculated.
- Fitness is the number of correct genes found when comparing the target, the expected value, to actual, the `this_choice` variable.
- If `expected==actual`, the sum is incremented.
- `zip`, in Python, is an efficient feature that iterates over two lists at the same time.

Once fitness, the sum of fit genes, is calculated, the function returns the value to the `ga_main()` function:

```
    return fitness
```

The parent generation will now be displayed by a function called in `main_ga()`.

Display parent

At this point, `ga_main()` has printed the start time, created a parent, and evaluated its fitness:

```
def ga_main():
    #I PARENT GENERATION
```

```
    startTime=datetime.datetime.now()
    print("starttime",startTime)
    alphaParent=gen_parent(len(target))
    bestFitness=get_fitness(alphaParent,scenario)
```

The program will now display the basic information concerning the first generation: the parent generation called the `display` function from `main_ga()`, line 7:

```
    display(alphaParent,bestFitness,bestFitness,startTime)
```

- `alphaParent` is the gene string of the parent
- `bestFitness` is its fitness
- Since there is no child yet, `bestFitness` is sent as the default value of a child's fitness
- `startTime`

In the display parent cell, line 2, the `display` function receives the data sent by `main_ga()`:

```
def display(selection,bestFitness,childFitness,startTime):
```

The `display` function calculates the time it has taken and prints the information in a few lines:

```
    timeDiff=datetime.datetime.now()-startTime
    print("Selection:",selection,"Fittest:",bestFitness,
          "This generation Fitness:",childFitness,
          "Time Difference:",timeDiff)
```

- `selection` is the string of genes of this generation.
- `bestFitness` is the value of the best string of genes created up to now.
- `childFitness` is this generation's fitness value. The first generation is the value of the parent who has, for the moment, the fittest genes. The parent is the child of another parent, although this parent is the first generation we are taking into consideration.
- `timeDiff` is an important value when dealing with larger gene sets. It will

help detect whether the algorithm is running well or reaching its limit.

The output will be displayed for the parent generation and each generation that gets closer to the fittest generation defined by the target:

```
Selection: BnVYkFcRK Fittest: 0 This generation Fitness: 0
Time Difference: 0:00:00.000198
```

This output will vary during each run of the program since this is a stochastic algorithm simulating the random events that occur in our natural and artificial environments.

Before exploring the loop that creates an unlimited number of generations, let's build the crossover function.

Crossover and mutation

Our model contains a `crossover` function with a mutation rule to ensure diversity.

The `crossover` function starts with the parent.

```
def crossover(parent):
```

Each child of each generation will become the parent of another child.

As in nature, a random gene in a parent's gene will be selected to be replaced:

```
    index=random.randrange(0,len(parent))#producing a rando
m position of the parent gene
```

The `index` designates the exact location of the gene that will be replaced:

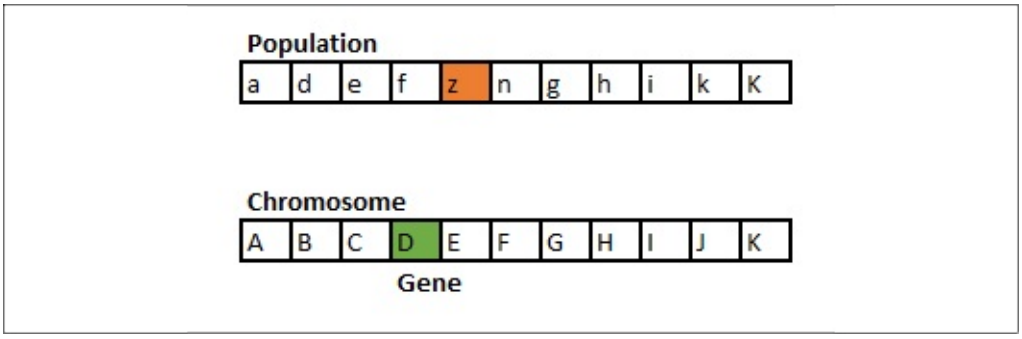


Figure 17.2: Chromosomes in genes

We can see that gene **D** in the chromosome will be replaced by the gene **z** of an individual of the population.

Now, we simulate the reproduction phase. The child inherits the genes of its parent:

```
    childGenes=list(parent)
```

The parent's string of genes is converted into a list with the `list` function.

The algorithm stores the parent's gene to replace in a variable:

```
    oldGene=childGenes[index]        # for diversity check
```

`oldGene` will be compared to the new gene generated to make sure that diversity is respected so as to avoid getting stuck in a local loop.

A new gene is chosen at random in the gene set of the population to simulate the interaction of the child with a given person among an indefinite number of persons:

```
    newGene,alternate=random.sample(geneSet,2)
```

Notice that at the same time, the new gene, `newGene`, is selected at random, while an alternate gene, `alternate`, is chosen as well. `alternate` is chosen to replace `newGene` to avoid making the wrong choice.

If the new gene, `newGene`, is not equal to the old gene, `oldGene`, that the child

can inherit:

```
    if(newGene!=oldGene):childGenes[index]=newGene;  #natur
al crossover
```

The new gene becomes part of the string of genes of the child.

However, if `newGene` is the same as `oldGene`, this could compromise the whole genetic process, with generations of children that do not evolve. Also, the algorithm might get stuck or waste quite some time making the right selections.

This is where the alternate gene comes in and becomes part of the string of genes of the child. This crossover rule and this alternate rule simulates the mutation process of this model.

```
    if(newGene==oldGene):childGenes[index]=alternate;  #mut
ation introduced to ensure diversity to avoid to get stuck
in a local minima
```

Diversity has been verified!

The function now returns the new string of genes of the child so that its fitness value can be calculated:

```
    return ''.join(childGenes)
```

Producing generations of children

At this point, we have generated the parent and explored the basic concepts and functions.

We are ready for the generation loop. First, we will view the code, then represent it in a flowchart, and then describe the lines of code.

Once the parent has been created, we enter the loop that simulates the evolutionary process for many generations:

```
    while True:
        g+=1
```

```
        child=crossover(bestParent)        #mutation
        childFitness=get_fitness(child,scenario) #number of
 correct genes
        if bestFitness>=childFitness:#
            continue
        display(child,bestFitness,childFitness,startTime)
        bestFitness=childFitness
        bestParent=child
        if scenario==1: goal=len(alphaParent);#number of go
od genes=parent length
        if scenario==0: goal=threshold;
        if childFitness>=goal:
            break
```

The loop is best represented in a flowchart:

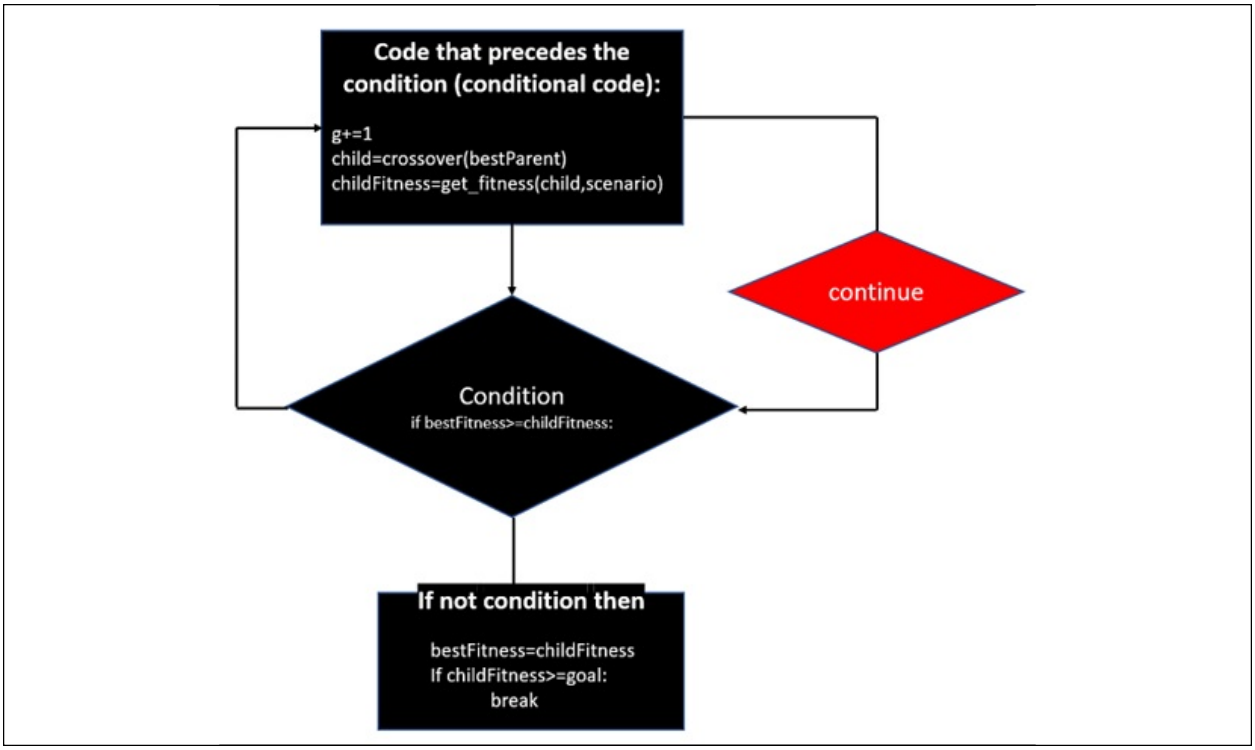


Figure 17.3: Genetic algorithm flow chart

The process of the flowchart is based on Python's `continue` method:

1. The code that precedes the condition:
    - Increments the generation counter:

```
        g+=1
```

- Calls the `crossover` function to produce a child:

```
        child=crossover(bestParent)
```

- Calls the fitness function to obtain the fitness value:

```
        childFitness=get_fitness(child,scenario) #numbe
r of correct genes
```

2. The condition to see if the `childFitness` is higher than the `bestFitness` obtained:

```
        if bestFitness>=childFitness:
```

   - If the condition is `True`, then the evolution must continue until a child is fitter than its parent. This sends the process back to the top of the `while` loop.
   - If the condition is `False`, this means that the child is fitter than the parent, and then the code will go beyond the condition.

3. The code beyond the condition and `continue` method:
   - The code displays the child, the `bestFitness` becomes the `childFitness`, and the `bestParent` is now the child:

```
        display(child,bestFitness,childFitness,startTim
e)
        bestFitness=childFitness
        bestParent=child
```

   - The goal of the two scenarios of our model is defined. The goal of `scenario==1` is to reach the length of the target with the right genes. The goal of `scenario==0` will be to reach a threshold we will define in the next section:

```
        if scenario==1: goal=len(alphaParent);
        if scenario==0: goal=threshold;
```

4. The `break` condition of the loop:

   The evolutionary process will stop when the fittest child has been created after several generations containing the genes that meet the target:

```
        if childFitness>=goal:
            break
```

The output of the *n* generations will appear as follows:

```
Genetic Algorithm
geneSet: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWX
YZ!.-
 target: Algorithm
starttime 2019-10-12 20:47:03.232931
Selection: Xe!bMSRzV Fittest: 0 This generation Fitness: 0
Time Difference: 0:00:00.000953
Selection: Xe!bMSRhV Fittest: 0 This generation Fitness: 1
Time Difference: 0:00:00.002404
Selection: Xl!bMSRhV Fittest: 1 This generation Fitness: 2
Time Difference: 0:00:00.004391
Selection: XlgbMSRhV Fittest: 2 This generation Fitness: 3
Time Difference: 0:00:00.006860
Selection: XlgoMSRhV Fittest: 3 This generation Fitness: 4
Time Difference: 0:00:00.009525
Selection: AlgoMSRhV Fittest: 4 This generation Fitness: 5
Time Difference: 0:00:00.011954
Selection: AlgorSRhV Fittest: 5 This generation Fitness: 6
Time Difference: 0:00:00.013745
Selection: AlgorSthV Fittest: 6 This generation Fitness: 7
Time Difference: 0:00:00.016339
Selection: AlgorithV Fittest: 7 This generation Fitness: 8
Time Difference: 0:00:00.019031
Selection: Algorithm Fittest: 8 This generation Fitness: 9
Time Difference: 0:00:00.022239
```

We can see the display of all of the generations as described in the `display` function in the *Display parent* section of this chapter.

Summary code

Once the evolutionary process is over, a summary code takes over:

```
#III. SUMMARY
    print("Summary-----------------------------------------
----------")
    endTime=datetime.datetime.now()
    print("endtime",endTime)
    print("geneSet:",geneSet);print("target:",target)
    print("geneSet length:",len(geneSet))
    print("target length:",len(target))
    print("generations:",g)
```

```
    print("Note: the process is stochastic so the number of
 generations will vary")
```

The output for the example displayed for the evolutionary loop is:

```
Summary-----------------------------------------------------
endtime 2019-10-12 20:47:03.257112
geneSet: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWX
YZ!.-
target: Algorithm
geneSet length: 55
target length: 9
generations: 782
Note: the process is stochastic so the number of generations
 will vary
```

Before we move on, it is important to note that using permutations for 9 out of 55 elements, it would take 10 ** 15.36 calculations to reach the goal instead of the 782 generations in this example. GAs are thus a productive way of generating permutations.

We have now explored the core of the evolutionary process and the Python code. We will now build the unspecified target and optimizing code that will lead us to hybrid neural networks.

Unspecified target to optimize the architecture of a neural network with a genetic algorithm

In this section, we are laying the grounds and motivation to optimize the architecture of neural networks through a hybrid neural network. The architecture of a physical neural network will be optimized by a GA.

We will study a physical neural network and then see how to optimize its architecture with our GA.

A physical neural network

We will begin with a physical network named S-FNN, a **feedforward neural network** (**FNN**). Please look very closely at the following figure and take as much time as necessary to understand its architecture:

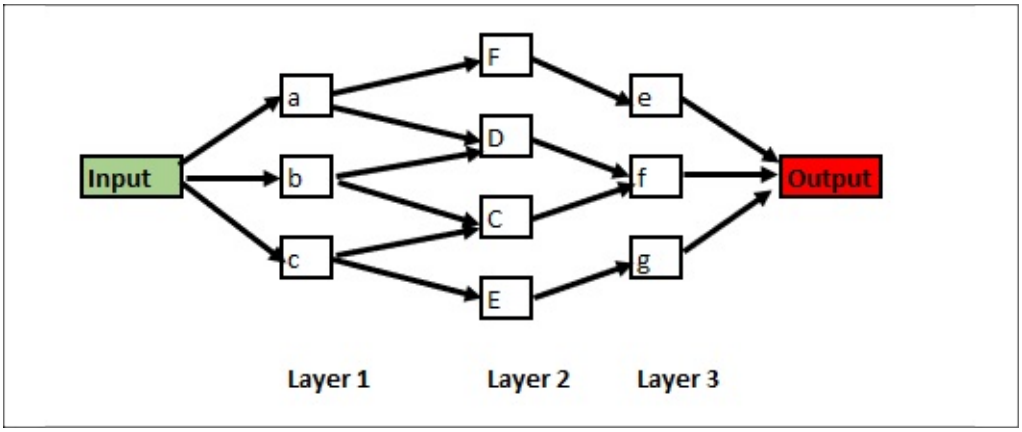


Figure 17.4: Architecture of a neural network

This physical network has some important specific features:

- This is an FNN
- There are three layers.
- The neurons in each layer are not fully connected to the neurons of the next layer. The connections between the neurons of the three layers are not based on all of the possibilities. The connections between the layers are based upon statistics of the best connections in the past when running this network.

  The connections are the result of many runs of this network to determine which ones are the best.

- The connections represented are the most productive ones, leading to an output of `1` and not `0`. An output of `1` is a success, while an output of `0` is a failure.
- The input is a random value of the dataset that must be classified as `1` if it is a success and `0` if it is a failure.
- After a careful study of past runs of this network, it has been found that the productivity of the network fully relies, in this example, on the number and quality of the neurons in layer 2.

Let's use what we have learned and apply it to the architecture of a soccer team.

What is the nature of this mysterious S-FNN?

S-FNN is a soccer team's architecture! It is the representation of a 3-4-3 disposition of a soccer team before a given game.

Like abstract neural networks, the number and quality of neurons we choose per layer are critical. In our abstract world of **artificial neural networks** (**ANNs**), we have many problems to solve. How many should we keep? How many should we eliminate through pooling layers? How many should we abandon through dropout layers? How do we really know? How long will trial and error take with large datasets and networks?

Now, let's go back to the figure, read the following explanation, and start finding a method to solve the architectural complexity of a neural network:

- **Input** represents the goal of best possible choices to input the ball in the game based on past game statistics.
- **Layer 1** is the defense layer of three players represented by three neurons.
- **Layer 2** is the middle field, the result of the transformation of layer 1's activity, the statistics of past games showing the fittest, the connections. The initial ball of the input has now been flowing through these layers game after game.
- **Layer 3** is the attack layer, the one that will lead to the output classification; 1 for a success meaning a goal has been scored, or 0 for failure. The connections between layer 2 and layer 3 show the fittest results found game after game.

The problem to solve: Layer 2, in our example, has been identified as the critical layer in this team. For years now, it has been a weak point. Let's see how our GA will help find the fittest players for this layer.

We could run `scenario==1` (specified target) with target `ga==3` as follows:

```
if(scenario==1 and GA==3):
    target="FBDC"  # No space unless specified as a charact
er in the geneSet
    print("geneSet:",geneSet,"\n","target:",target)
```

```
    ga_main()
```

The `geneSet` is the population of available players on the soccer market and `target` is the string of genes we need for layer 2 of our physical network:

```
Genetic Algorithm
geneSet: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWX
YZ!.-
target: FBDC
```

The target is found after 851 generations in this run.

However, we would like the GA to find the players by itself based on the features with an unspecified target. Let's explore this step by step, cell by cell.

Calling the algorithm cell

From lines 7 to 12, we defined the parameters of the target architecture we want for layer 2 of our network:

- `geneSet` is the set of all the available players for layer 2 of our network, whether they are in the team or on the market:

  ```
  geneSet="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRST
  UVWXYZ!.-"   #gene set
  ```

- `KPIset`, or the key performance indicator set, is a performance score per player on the market for middle field. Each integer of the set contains a score between 0 and 9 based on the statistics of a player when playing as a midfielder:

  ```
  KPIset ="01234567720123456747012345699809234567670123456
  71001234"   #KPI set
  ```

  The KPI set matches each member of the gene set.

- `threshold` is the sum of the midfielders' performance we need to attain in order to have a strong midfield in the team.

  ```
  threshold=35
  ```

Our goal is to find a string of genes based on their features, their KPI properties.

The goal is to reach the threshold.

We call our evolutionary function on line 23:

```
if(scenario==0 and GA==2):
    target="AAAA"                          #unspecified targ
et
    print("geneSet:",geneSet,"\n","target:",target)
    ga_main()
```

As you can see, `target` is set to a meaningless value that will rapidly evolve. The GA will have to find a fit sequence.

We can skip the intermediate fitness functions we have described in the previous section and focus on the scenarios within the fitness cell.

Fitness cell

We have already described the fitness function of the fitness cell in the previous sections. We will focus on the code of the unspecified target, `scenario==0`.

The first part of the scenario calculates the sum of the performance of each gene (potential midfielder):

```
    if(scenario==0):
        cc=list(this_choice) # cc= this choice
        gs=list(geneSet)     # gene set
        cv=list(KPIset)      # value of each KPI in the set
        fitness=0
        for op1 in range(0,len(geneSet)): #2.first find par
ent gene in gene set
            for op in range(0,len(target)):
                if cc[op]==gs[op1]:       #3.gene identifie
d in gene set
                    vc=int(cv[op1])       #4.value of criti
cal path constraint
                    fitness+=vc
```

The collective fitness of the sequence of genes (midfielders) is contained in the

`fitness` variable.

However, we cannot accept the same gene twice, which would mean we have a clone of the midfielder on the field! So, we add some safety code to set `fitness` to `0` in that case:

```
            for op in range(0,len(target)):
                for op1 in range(0,len(target)):
                    if op!=op1 and cc[op]==cc[op1]:
                        fitness=0    # no repetitions allow
ed, mutation enforcement
```

Now, we can go back to `ga_main()` and complete our process.

ga_main() cell

We have already described the fitness cell in the previous sections. Now, we will focus on the code of the unspecified target, `scenario==0`.

In the `ga_main()` cell, we simply need to examine lines 22 to 24:

```
        if scenario==0: goal=threshold;
        if childFitness>=goal:
            break
```

If the `scenario==0`, `childFitness` must be `>=goal` (sum of KPIs).

We have found our midfielders!

We will now display the result:

```
Genetic Algorithm
geneSet: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWX
YZ!.-
 target: AAAA
starttime 2019-10-13 09:08:21.043754
Selection: PNVx Fittest: 18 This generation Fitness: 18 Tim
e Difference: 0:00:00.001252
Selection: LNVx Fittest: 18 This generation Fitness: 24 Tim
e Difference: 0:00:00.004130
Selection: LNVq Fittest: 24 This generation Fitness: 27 Tim
```

```
e Difference: 0:00:00.004708
Selection: LNFq Fittest: 27 This generation Fitness: 29 Tim
e Difference: 0:00:00.010362
Selection: LBFq Fittest: 29 This generation Fitness: 31 Tim
e Difference: 0:00:00.015797
Selection: CBFq Fittest: 31 This generation Fitness: 33 Tim
e Difference: 0:00:00.024712
Selection: CBFt Fittest: 33 This generation Fitness: 34 Tim
e Difference: 0:00:00.032680
Selection: CBFD Fittest: 34 This generation Fitness: 35 Tim
e Difference: 0:00:00.048524
Summary----------------------------------------------------
endtime 2019-10-13 09:08:21.094005
geneSet: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWX
YZ!.-
target: AAAA
geneSet length: 55
target length: 4
generations: 154
Note: the process is stochastic so the number of generation
s will vary
```

In this case, the GA chose midfielders C, B, F, and D.

We now have all of the ingredients, concepts, and knowledge required to understand an artificial hybrid neural network.

Artificial hybrid neural networks

In the previous section, we used a GA to optimize a physical neural network.

In this section, we will extend the concept of *hybrid* we have just explored to ANNs. The principle is the same, so it will be relatively easy, with the concepts you now have in mind, to intuitively grasp the RNN we will optimize in this section.

The future of AI in society lies in the collective intelligence of humans (diversity), machines (AI and IoT), and nature (sustainable projects for our survival).

In AI, this diversity lies in ensemble algorithms, meta-algorithms and hybrid systems. Deep learning has proven its point. We can create a neural network with TensorFlow 2.x in a few lines. However, more often than not, it takes days, weeks, and often months to fine-tune ANN models that rely on large amounts of data to provide a reliable model. And that's where hybrid neural networks are necessary.

A deep learning network can use any form of another type of algorithm to become a more efficient system. In our case, we have chosen evolutionary algorithms that could be used in deep learning:

- To improve the input by extracting sequences of data that fit the patterns we are looking for using features of the data to find the fittest data points
- To optimize the weights of the layers of a network to boost the speed and power of traditional optimizers
- To improve the classification phase of the output of a network by selecting the fittest solutions processed as a sequence of genes
- To improve the tedious task of defining the architecture of a network

In this section, we will focus on an example of optimizing the architecture of a network with a GA by:

- Creating the RNN
- Introducing a GA that will optimize the architecture of the network

> **Note**: For information on genetic algorithm representations, please go back to the previous sections at all times to refresh the process of an evolutionary algorithm. Please also refer back to the previous chapters to consult the structure of a neural network, in particular, *Chapter 16*, *Improving the Emotional Intelligence Deficiencies of Chatbots*, which describes an RNN.

Building the LSTM

Open `Genetic_Algorithm_LSTM.ipynb` in Google Colaboratory or Jupyter on your machine or `genetic_algorithm_lstm.py`.

The model of this education example is an LSTM.

The goal will be to use the window size generated by the GA and run the LSTM with this window size. **Root-mean-square error** (**RMSE**) will be used to measure the fitness of the configuration.

In short, RMSE first calculates the square of the difference between the training data and the testing data, like many similar formulas. Then, the root of the result is calculated. Just keep in mind that the RMSE will compare what we expect to get with what we actually get and produce a value.

Let's now explore the main cells of the program:

- **Importing required packages cell**: `Genetic_Algorithm_LSTM.ipynb` starts by installing DEAP, an evolutionary computation framework:

  ```
  !pip install deap
  ```

  We built a GA from scratch in the previous sections. We won't need to begin from nothing this time, since this program uses a framework.

  Then, the program installs `bitstring`, which helps to process binary data.

- **Loading the data cell**: The data wind power forecast data in `train.csv` comes from https://www.kaggle.com/c/GEF2012-wind-forecasting/data.

  The `wp1` to `wp7` columns provide normalized data collected from measurements of the wind power of seven wind farms. The goal of the LSTM will be to take the sequence of data and make wind power forecasts.

- **Defining the basic functions cell**: This cell prepares the dataset and trains the model in a standard process. We will focus on line 14:

  ```
      window_size = window_size_bits.uint
  ```

- **Evolutionary model cell**: The model uses the DEAP framework function, but we easily recognize the concepts we explored in the previous sections and that are initialized as follows:

  ```
  population_size = 4
  num_generations = 2
  ```

```
gene_length = 10
```

The code is all set with a ready-to-use GA to optimize the window size for our network.

The goal of the model

As in our previous section, the goal is to find the best window size for this network, just as we were looking for the best layer 2 in an earlier section.

The model has done the following tasks:

- Installed the packages, loaded the data, and found the window size of the LSTM
- Then, it ran the GA model to test the possible window size of the LSTM
- An RMSE measurement is provided with each generation and production of epochs

The program takes some time to run, but the results I ran are saved in the notebook so that you can view them. Here is one result to view how the system works:

```
Epoch 1/5
17200/17200 [==============================] - 207s 12ms/sa
mple - loss: 0.0096
Epoch 2/5
17200/17200 [==============================] - 202s 12ms/sa
mple - loss: 0.0059
Epoch 3/5
17200/17200 [==============================] - 202s 12ms/sa
mple - loss: 0.0058
Epoch 4/5
17200/17200 [==============================] - 200s 12ms/sa
mple - loss: 0.0057
Epoch 5/5
17200/17200 [==============================] - 200s 12ms/sa
mple - loss: 0.0057
Test RMSE: 0.0926447226146452
```

As in the previous section, a GA optimized a section of an ANN. You can use a

GA for other components of an ANN. The sky is the limit! In fact, there is no limit. The potential of hybrid neural networks using GA or other algorithms to optimize their architecture or process takes your projects to another level!

Summary

Evolutionary algorithms bring new light to AI's optimizing potential. In this chapter, we studied how heredity deeply affects population distribution. The impact of our environment can be measured through genetic mutations.

Drilling down further, we focused on a class of GAs implementing a simulation of genetic transformations through many generations. We explored how a parent will transmit some genes, but how the selection of diverse genes from the generation population of genes will produce variations. A chromosome will inherit some genes but not others.

The pressure of nature and our environment will take over. A fitness function evaluates a string of genes. Only the fittest will survive. The fittest genetic material will produce a crossover and mutation of the child, making it fitter for its environment.

GAs can be used to represent strings of any type of data and also features of that data. The optimizing process can be applied to warehouses, transportation, and neural networks.

Hybrid networks will no doubt expand in the years to come, taking DL to the next level. Using a genetic algorithm to optimize the architecture of an RNN paves the way to optimize the architecture of any DL, ML or AutoML architecture. For example, a hybrid neural network can use a genetic algorithm to optimize inputs with feature reduction or as the weight optimizing function of the network.

Nature has provided us with invaluable tools to apply to our artificial network models. This chapter dealt with the invisible building blocks inside us. In the next chapter, *Neuromorphic Computing*, we will explore other components that enable us to adapt to our environment: neurons. We will explore how neural networks using biological models can solve complex problems.

Questions

1. A cell contains 42 chromosomes. (Yes | No)
2. A genetic algorithm is deterministic, not random. (Yes | No)
3. An evolutionary algorithm means that the program code evolves. (Yes | No)
4. It is best for a child to have the same genes as one of the parents even after many generations. (Yes | No)
5. Diversity makes the gene sets weaker. (Yes | No)
6. Building a neural network only takes a few lines, and the architecture always works. (Yes | No)
7. Building a neural network with a genetic algorithm can help optimize the architecture of the layers. (Yes | No)
8. Hybrid neural networks are useless since deep learning will constantly progress. (Yes | No)
9. Would you trust a genetic algorithm to make decisions for you? (Yes | No)
10. Would you trust a hybrid neural network to optimize the architecture of your network? (Yes | No)

Further reading

- https://github.com/DEAP/deap
- https://pypi.org/project/bitstring/

18

Neuromorphic Computing

Our brain activates thousands or even billions of neurons when necessary, getting our body battle-ready to face any situation. As we saw in *Chapter 17, Genetic Algorithms in Hybrid Neural Networks*, evolution has fined-tuned biological capacities over thousands of generations and millions of years.

In this chapter, we will take a deeper look into the cognitive power inside our bodies. We will go from the chromosomes of the previous chapter to biological neurons that make us intelligent creatures. The neurons interact in billions of ways producing cognitive patterns leading to mind structures.

Neuromorphic computing taps into the tremendous optimized power of our brain, which surprisingly consumes very little energy. On average, we consume a few watts, less than a lightbulb, to solve very complex problems. In itself, this shows that the neuronal structure of our brain has a unique architecture that we have yet to reproduce physically.

To bring neuromorphic computing into the real world requires hardware and software, as in all computer science models. In this chapter, we will focus on the software, though it is important to mention the hardware associated with the neuromorphic research Intel is conducting. That hardware takes the form of a chip named Loihi, after the emerging Hawaiian underwater volcano that will hit the surface one day. Loihi contains thousands upon thousands of neurons with their synapses, dendrites, and axons reproducing our brain activity. IBM and other corporations have been conducting research in this area.

We have around 100 billion neurons. These chips are only reaching hundreds of thousands of neurons. However, by connecting thousands of those chips in physical networks, neuromorphic computing will be an area we all will have to take into account in the near future.

We will first define what neuromorphic computing is and then explore Nengo,

a unique neuromorphic framework with solid tutorials and documentation. Nengo is one among many other approaches that go beyond the scope of this book. This chapter is not a neuromorphic course but rather an overview, inviting you to tap into the wonderful power of our brain structures to solve complex problems. The problems we will explore will bring us closer to understanding how our brain works.

The following topics will be covered in this chapter:

- What neuromorphic computing is
- Getting started with Nengo
- Basic Nengo concepts
- Exploring the Nengo tutorial and interface
- The difference between Nengo and classical AI
- Applying Nengo's unique **Semantic Pointer Architecture** (**SPA**) model to critical research areas

Let's start with the basics—what is neuromorphic computing?

Neuromorphic computing

Let's go directly to the core of our thought process to understand neuromorphic computing. For AI experts, I would like to summarize the voyage from our classical models to cutting-edge neuromorphic models in a single phrase:

from mind to brain

If we take this further, M is the set of all of our mental representations and B is the world of physical reactions that lead to thinking patterns.

In this sense, M is a set of everything we have explored up to this point in this book:

M = {rule based systems, machine learning, deep learning, evolutionary algorithms … m}

m is any mathematical *mental representation* of the world surrounding us. In deep learning, for example, an artificial neural network will try to make sense of the chaos of an image by searching the patterns it can find in an image through lower dimensions and higher levels of abstraction.

However, a mental construction, no matter how efficient it seems, remains a *representation*, not a physical reality.

Now, let's observe *B = brain constructions*:

B = phenomena/events (inside us or in the outer world) -> physical stimuli -> physical neural activity in the brain -> higher activity in the target zones -> physical electric learning reactions -> a human action

The architecture of *B* takes us much closer to reality! Mental representations are minimized, thus reducing the distortion of artificial constructions regardless of their efficiency.

Classical AI is about building mental representations of our cognitive activity.

Neuromorphic computing is about building a brain that can encode reality, process it like a human brain, and decode the result.

Bear in mind that corporations such as Intel are providing the chips to accomplish wonderful things with neuromorphic computing, as we will discover in this chapter.

Now that we have some idea of what neuromorphic computing is, let's take a look at the neuromorphic framework, Nengo.

Getting started with Nengo

In a nutshell, *Nengo builds brains, not mental representations*, as in classical machine learning and deep learning.

Nengo stands for *Neural Engineering Object*. It has both scripting capability with Nengo and a graphical capacity with Nengo GUI. We will be using NEF,

which is Nengo's **Neural Engineering Framework** (**NEF**).

Nengo was created by the Centre for Theoretical Neuroscience at the University of Waterloo (Ontario, Canada). Chris Eliasmith has played an important role in this project.

We have explored many ways to approach cognitive modeling in the previous chapters. Nengo uses an NEF to implement an SPA.

A *semantic pointer* is a neural representation in a biological system that carries structures that will lead to higher-level cognitive representations.

The term *pointer* refers to pointers as we know in C++, for example, because they can access data they do not contain.

The term *semantic* refers to the fact that they are not just mathematical tools as in C++ because they contain virtual representations through the distances between them.

If we put the two concepts together, this leads to the mind-blowing concept of meaning being generated through biological pointer activity located at various distances and states from each other in our brain. Let's dive into neuromorphic computing by first installing Nengo.

Installing Nengo and Nengo GUI

For this chapter, I used the Python interfaces with NumPy and Matplotlib libraries, as we have since the beginning of this book. All that is then required is to install Nengo with `pip`, for example:

```
pip install nengo
```

You can install it using other approaches. For more information, go to https://github.com/nengo/nengo.

Let's install the nice HTML 5 visualizer and interact with Nengo through this GUI I installed with `pip`:

```
pip install nengo-gui
```

You can install it using other approaches, too. For more information, go to: https://github.com/nengo/nengo-gui/.

Once both programs are installed, if you encounter any problems then consult the links, which contain a lot of information, or Nengo's support team, who provide excellent feedback.

There are many ways to use Nengo. This chapter describes a quick start method with Python.

Once you are all set, open your browser, Chrome, for example, and then type `nengo` in a command-line console and it should open the Nengo GUI interface in your browser, opening a `default.py` Python program:

```
1  import nengo
2
3  model = nengo.Network()
4  with model:
5      stim = nengo.Node([0])
6      a = nengo.Ensemble(n_neurons=50, dimensions=1)
7      nengo.Connection(stim, a)
8
```

Figure 18.1: Nengo Python code

Click on the folder icon in the top left and click on `built-in examples`:

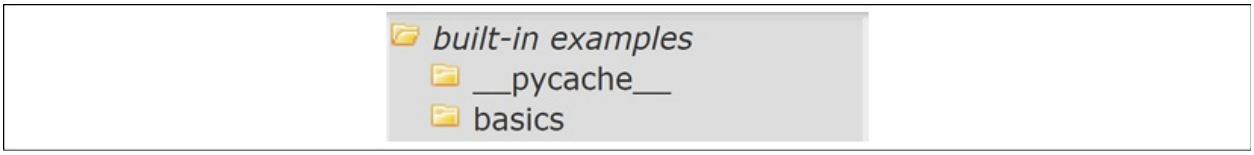


Figure 18.2: Nengo examples

A list will appear. Click on `tutorial`:

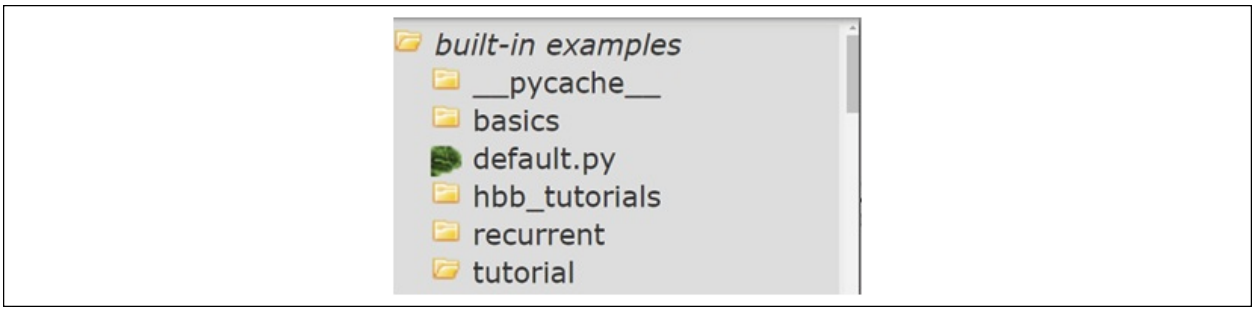


Figure 18.3: List of Nengo examples

A list of fascinating educational examples will appear:

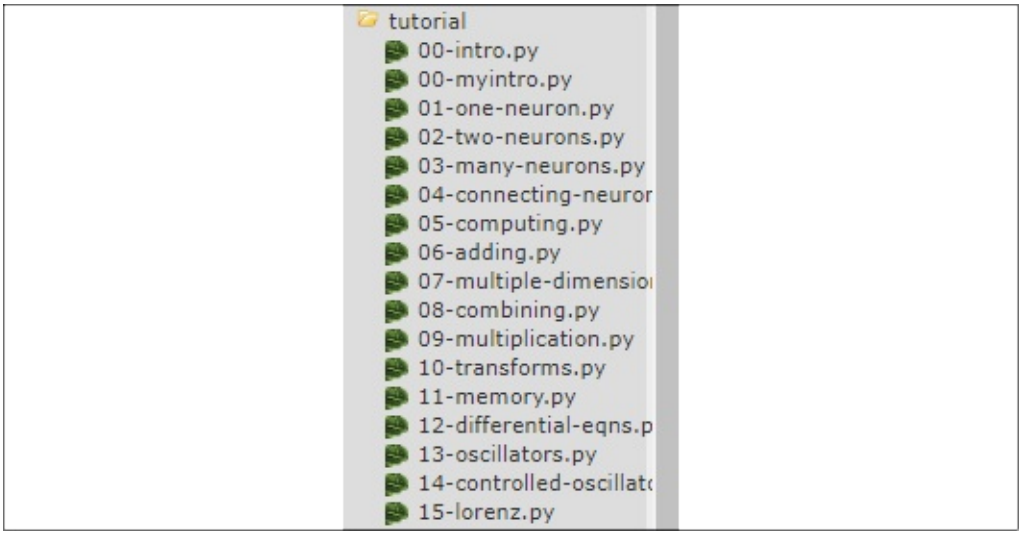


Figure 18.4: Examples in the tutorial section

The tutorial programs are in the directory the installer wrote them in.

You can see them in the URL of a Python example of the tutorial in the address bar of your browser.

Creating a Python program

Now, let's create a Python program and save the file through a few steps:

- Open an empty Python file but do not save it yet
- Write the following code to import the `nengo` library and create a model:

```
import nengo
```

```
model = nengo.Network()
with model:
    #<your code here>
```

Now, save the Python file in the path of the other programs of the tutorial. You can see this path when you open a Nengo example. Just navigate to that path to save your program. We will name it `00-myintro.py` to fit in nicely with the list of programs in the tutorial:

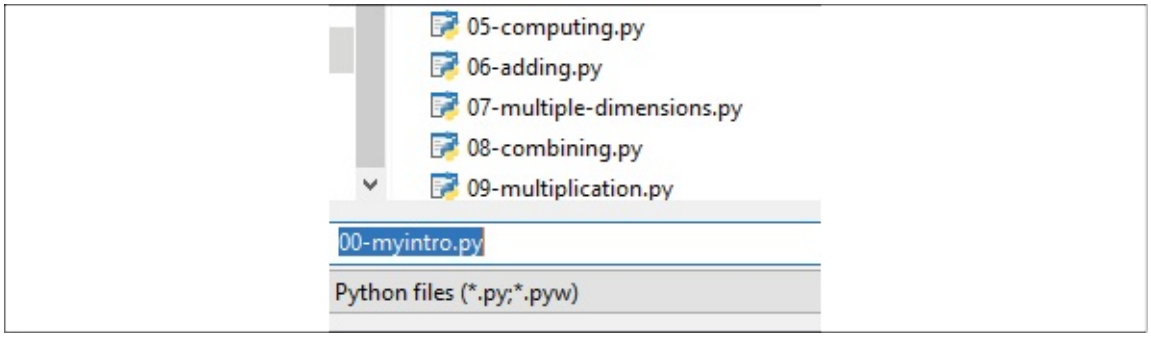


Figure 18.5: Saving a Python program

We will go back to the `tutorial` directory as we did previously, and we will open it to add basic Nengo objects.

Nengo objects are the building blocks of a Nengo model to create populations of neurons, connecting them to stimulation functions, and managing the outputs.

For our Python program, we will use some key Nengo objects, which are detailed in the following sections.

A Nengo ensemble

A Nengo ensemble is a group of neurons. It can be considered as a population of neurons that contain real numbers.

An ensemble is an object created with:

```
nengo.Ensemble
```

An ensemble can be created in one line:

```
ensemblex = nengo.Ensemble(n_neurons=5, dimensions=1)
```

As soon as we add a line to `00-myintro.py`, a representation of the ensemble appears on the left-hand pane of the Nengo interface:

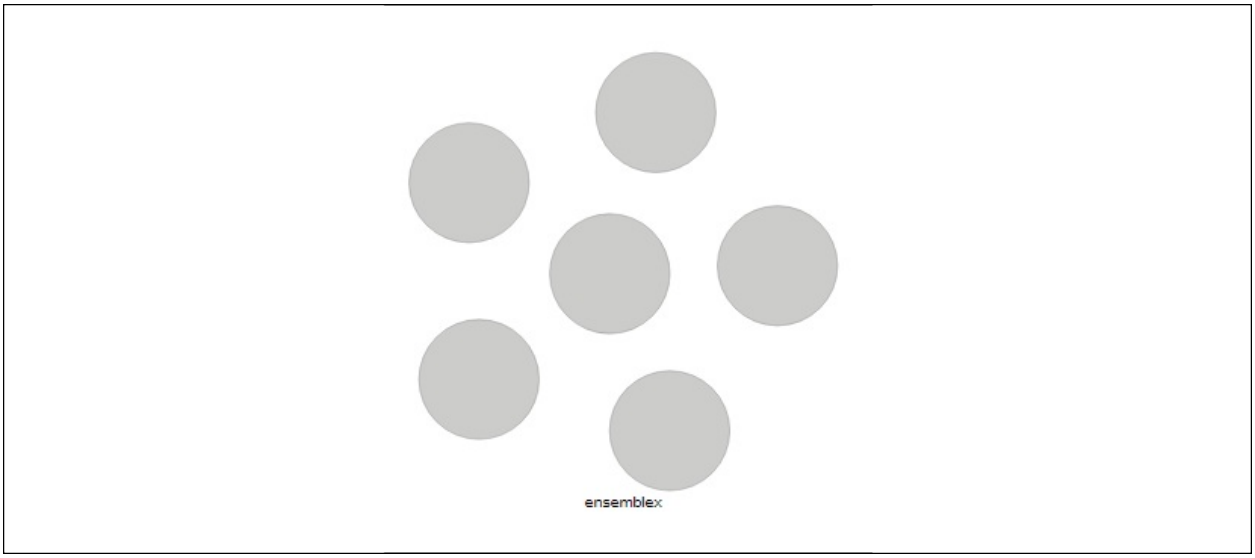


Figure 18.6: Neuron ensemble

While writing your Python code in the code editor on the right, you can visualize the visual flow of the model in the visual interface pane.

An ensemble can contain a population of one individual, a single neuron. This neuron is a representation of our biological neurons:

- **Postsynaptic currents** (**PSCs**) flow through our dendrites
- The *current* then reaches the core of the neuron (cell body)
- If the current exceeds a threshold at the axon's initial segment (axon hillock), then a *spike* is generated
- With ion channels open, the PSCs are produced in the receiving cell

Let's take a more detailed look at Nengo's neuron types.

Nengo neuron types

We created 50 neurons in our ensemble:

```
ens = nengo.Ensemble(n_neurons=50...
```

There are various neuron types. However, in our example, we will be using the default neuron, a **leaky integrate-and-fire** (**LIF**) neuron.

Neuronal dynamics are based on a summation process called **integration**. This integration is associated with a mechanism that will fire (trigger) up above a critical voltage.

A linear differential equation combined with a threshold that will trigger file spiking are the final components that make up the default LIF neuron we will be using, unless specified otherwise.

For more on Nengo neuron types, see https://www.nengo.ai/nengo-extras/neurons.html.

Nengo neuron dimensions

In our example, `dimensions` is set to `1`; this means that the ensemble is represented by one number (or dimension):

```
ens = nengo.Ensemble(..., dimensions=1)
```

A Nengo node

Now that we have defined our ensemble of neurons and their output dimension, we will define the output:

```
node_number = nengo.Node(output=0.5)
```

The stimulation will be a constant and will be displayed on the slider as such:

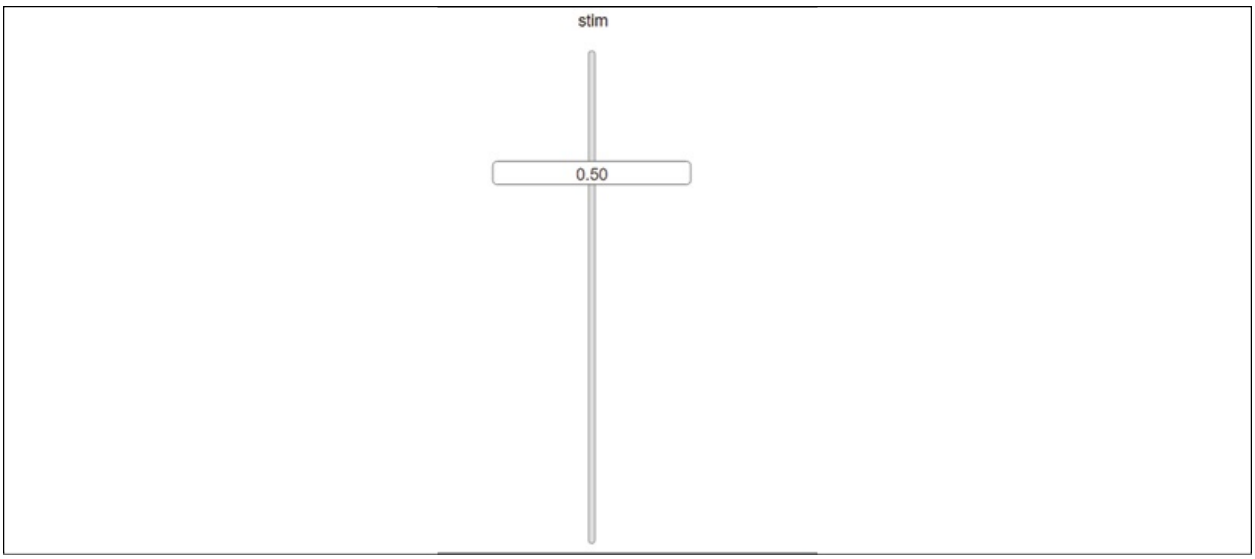


Figure 18.7: Nengo slider

An output with a number might not suffice in some cases. This number can be replaced by a function importing NumPy. A sine wave function can be used, for example:

```
node_function = nengo.Node(output=np.sin)
```

As soon as we enter our node function, it appears on the interface in addition to the previous information displayed:

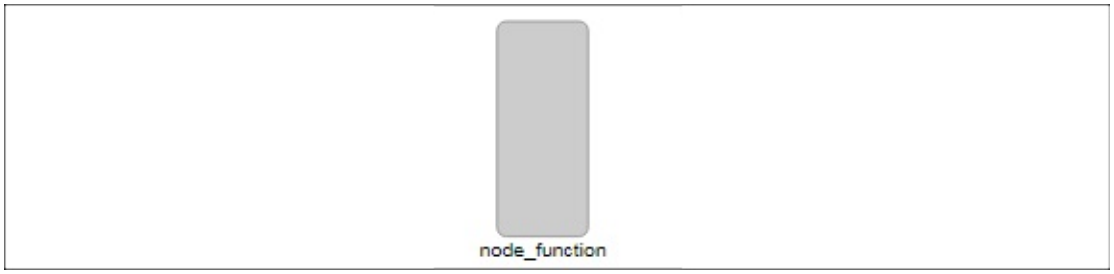


Figure 18.8: The node function on the interface

`node_function` provides a way to send non-neural inputs to Nengo objects.

We will explore such an implementation in the `15-lorenz.py` example in this chapter that is in the tutorial section Nengo's examples.

If you right-click on the `node_function` image and choose a value, you will see

a curve representing the real-time value of the sine wave stimulation:

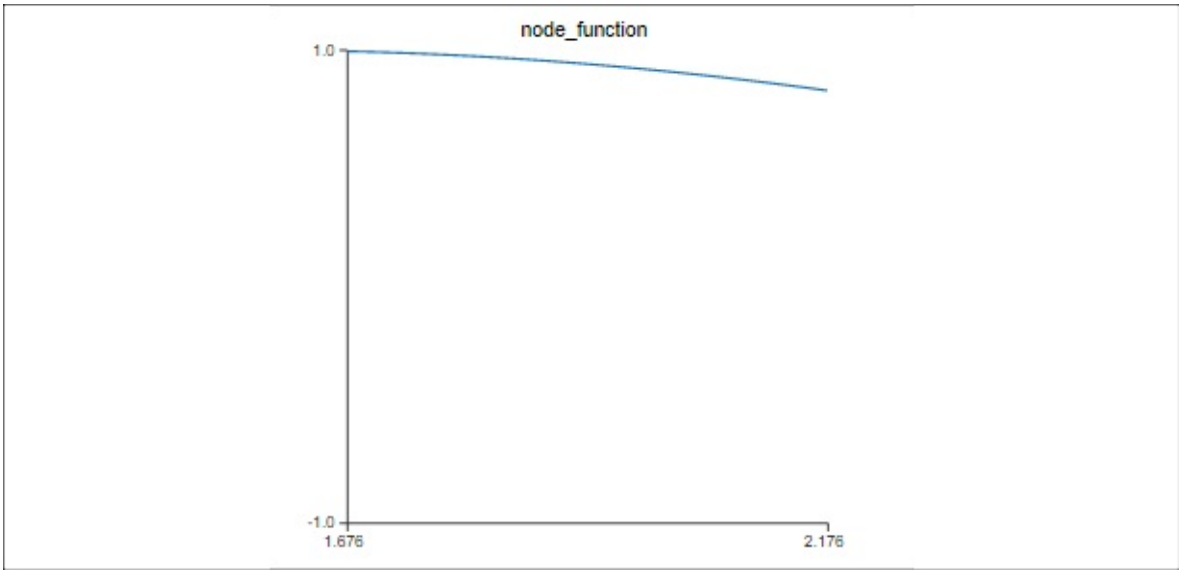


Figure 18.9: Node function

For more on Nengo objects, go to https://www.nengo.ai/nengo/getting_started.html#creating-nengo-objects.

Connecting Nengo objects

We now need to connect the ensemble and the node to make our system work. In this manner, the ensemble will have a function.

At this point, we have an ensemble and a node function, as shown in the following figure:

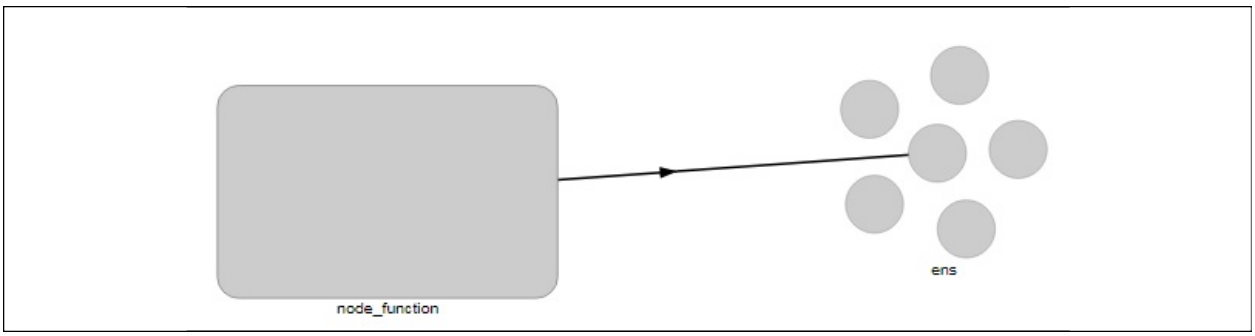


Figure 18.10: Ensemble and node function

To connect them, we will add a Nengo connection:

```
nengo.Connection(node_function, ens)
```

For more Nengo frontend API definitions, see https://www.nengo.ai/nengo/frontend_api.html.

Now, let's explore the exciting visual interface.

Visualizing data

The first step is to click on the play button in the bottom-right corner of the screen:



Figure 18.11: Play button

This will set time in motion, simulating our brain activity. Time is one of the unique features of neuromorphic computing. We do not pile layer upon layer of static mathematics into a mental representation. With Nengo, we simulate brain activity step by step, second by second!

Once you click on the play button, it feels like we are looking inside our brain!

We saw how to visualize the slider's activity in the previous section, which produces the stimulations. We will focus on our ensemble in this section.

If we right-click on the ensemble visualization, several options appear: **Value**, **Spikes**, **Voltages**, **Firing pattern**, and **Details...**:

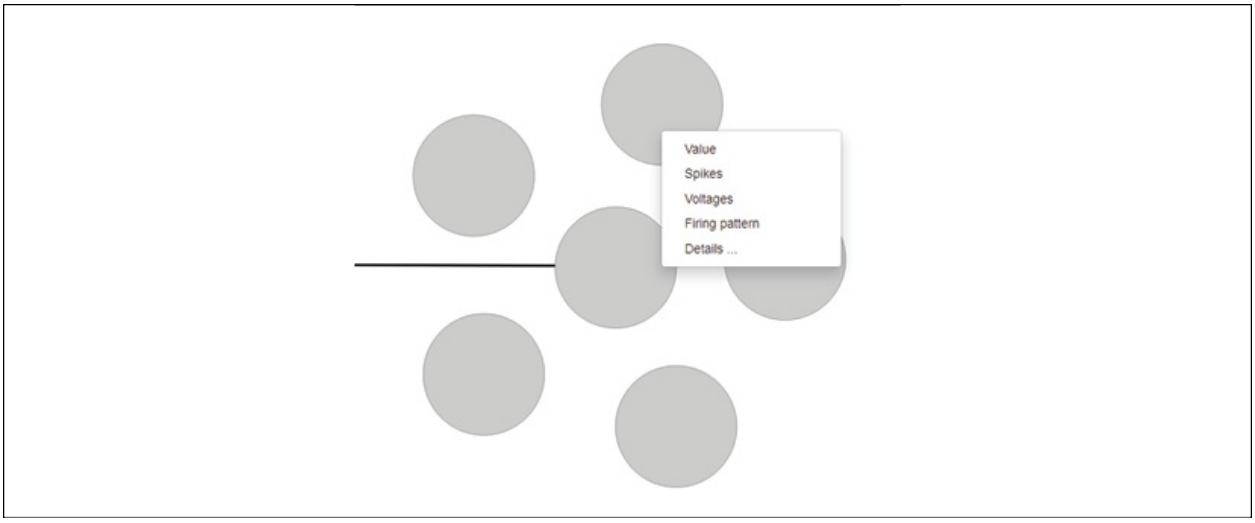


Figure 18.12: Ensemble options

- **Value**: The value of our ensemble will be displayed, from –1 to 1, for example:

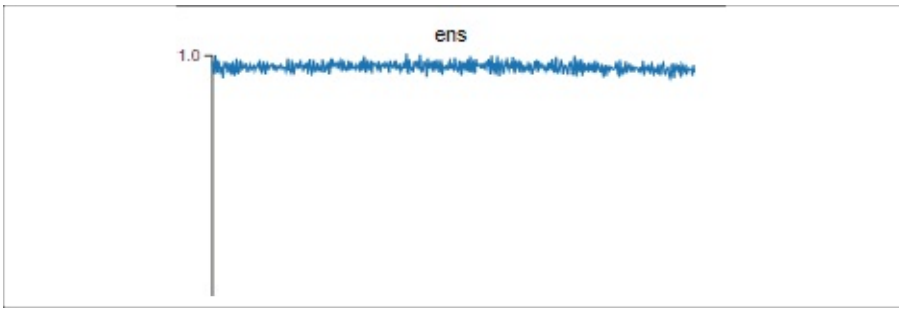


Figure 18.13: Values of an ensemble

- **Spikes**: The spiking activity produces nice colors that show how our neurons are reacting to sine wave stimulation:

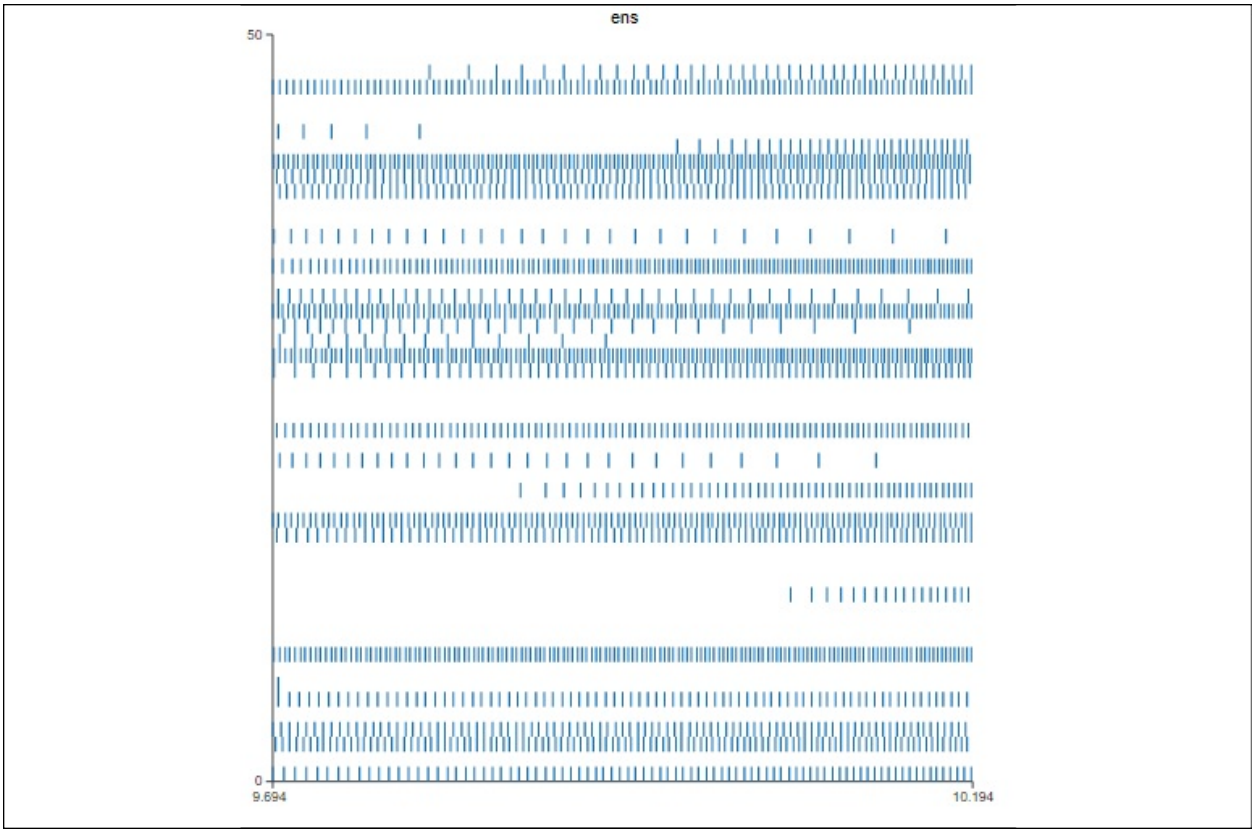


Figure 18.14: Spiking patterns

You will notice that each neuron has its own spiking channel. This property of ensembles produces a wide variety of responses.

- **Voltages**: The voltages provide interesting information on the current that is flowing through our neurons under stimulation from other neurons and, in turn, from yet more neurons that are gathering information from the world outside us.

  In the following screenshot, the ensemble contained a population of five neurons with color channels providing yet more information:

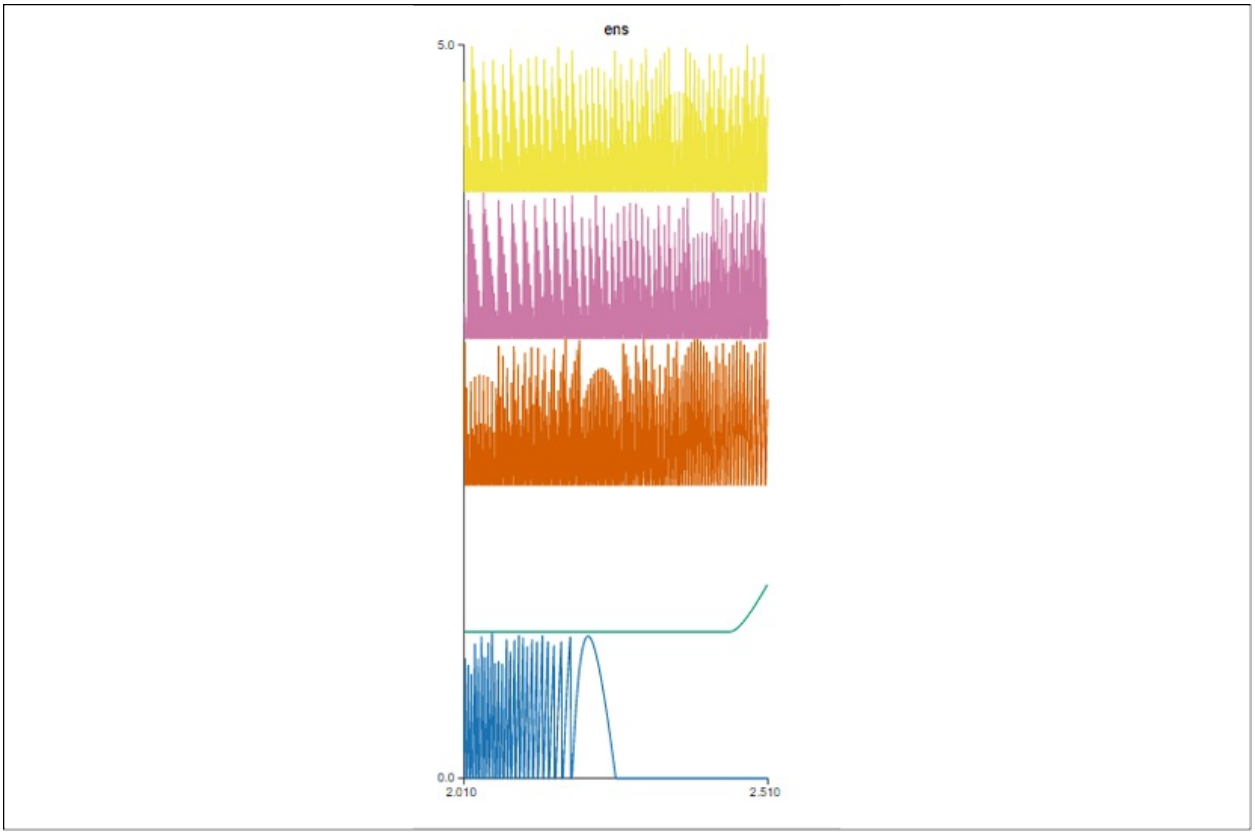


Figure 18.15: Neuron activity with color patterns

- **Firing pattern**: The firing pattern of the ensemble in the following image was generated with 50 neurons. The firing pattern is necessarily directly linked to the stimulations and connections:

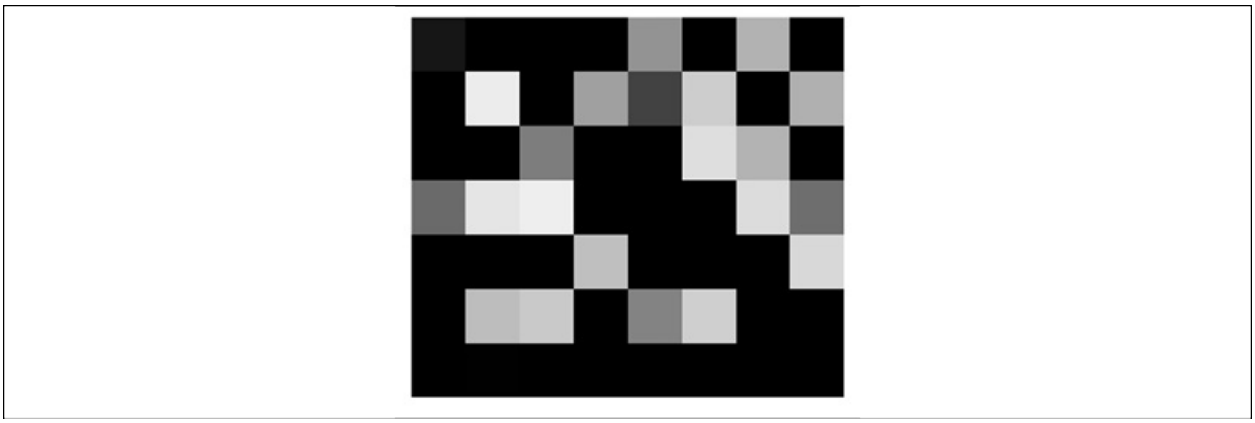


Figure 18.16: Firing pattern

I have been observing many patterns through the experiments I carried out. One of the areas of research I'm doing is to feed the thousands of frames of firing

patterns of a given function to an ANN and a stochastic Church-Turing algorithm and generate "thought" patterns. For example, running the channels of the frames could produce sequences of new data. It's worth experimenting with.

The following image represents the pattern of 500 neurons bringing complexity to the potential of running deep learning on thousands of frames of these patterns:

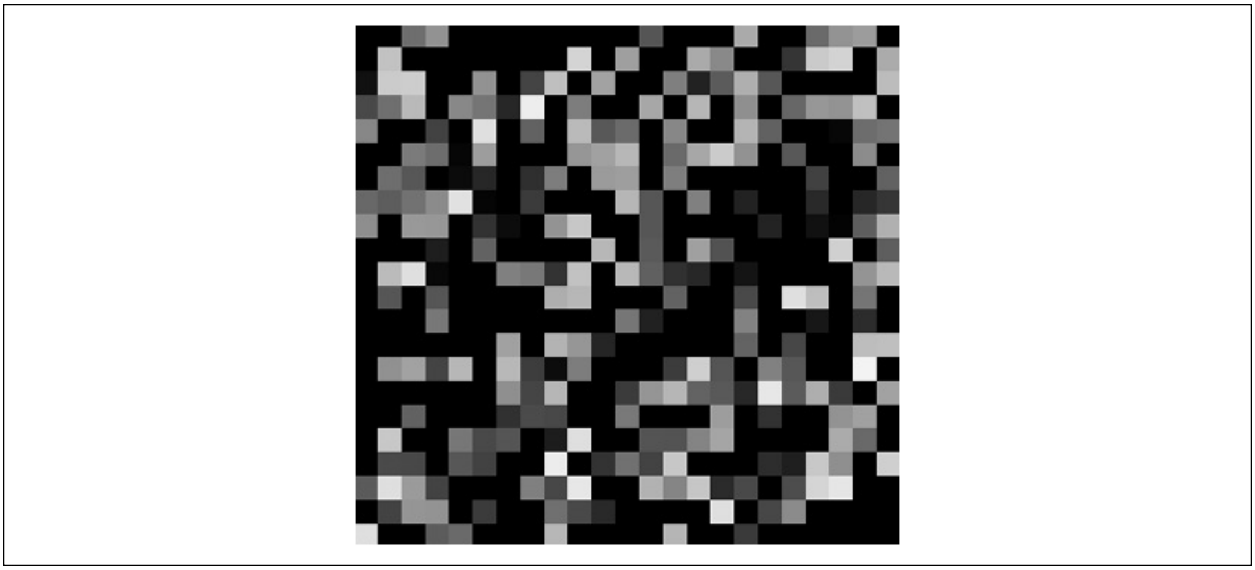


Figure 18.17: Firing patterns

- **Details...**: In the plots section, we can visualize input current and tuning curves. The tuning curve shows how the neurons are reacting to the input current. If there is no pattern, then there must be an explanation or a configuration problem.

  The visual control of tuning curves is a time saver by showing how they converge!

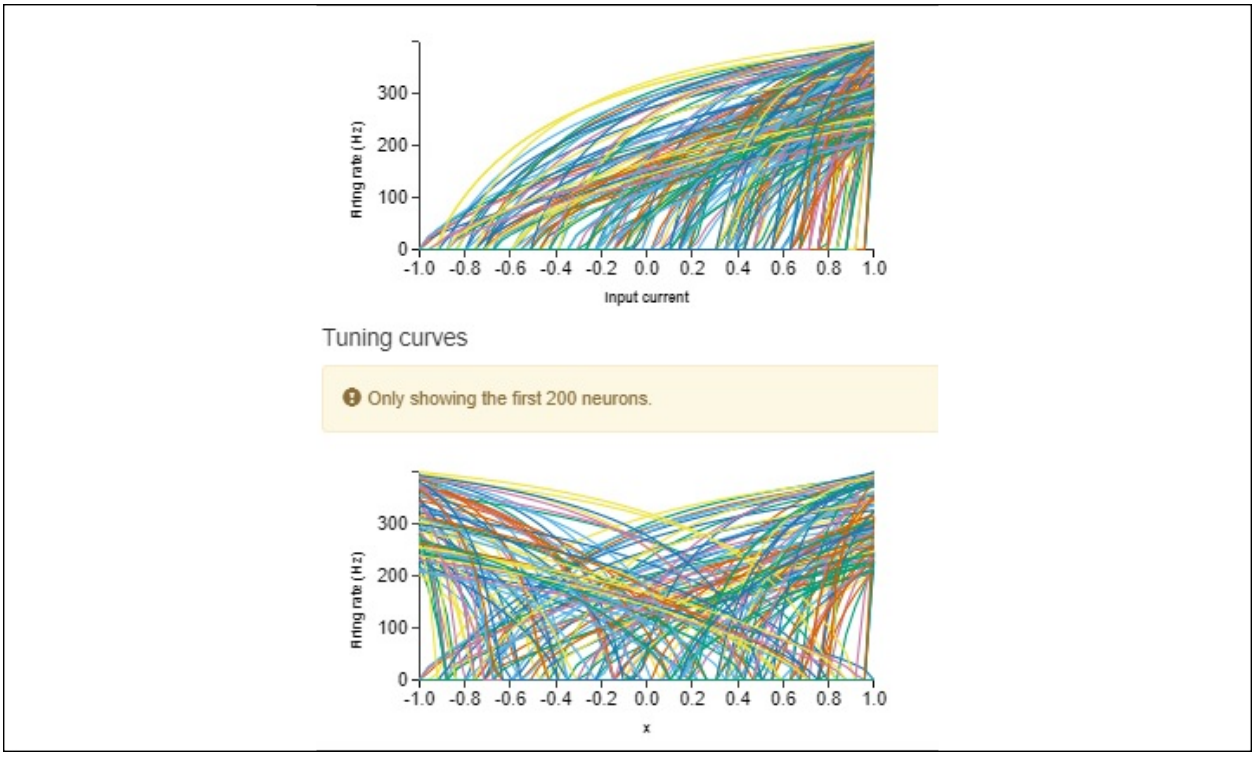


Figure 18.18: Tuning curves

We have covered some of the main visual tools Nengo provides.

Now, we will see how to retrieve data from our system with probes.

Probes

We can retrieve information with probes, either to visualize them or to process output data.

In this section, I enhanced `00-myintro.py` to produce numerical output as well as visual output using the information on the following page that you can get ideas from for your projects: https://www.nengo.ai/nengo/examples/basic/single_neuron.html

The program I created is `nengo_probing.py` and is available in the GitHub repository of this book. The program is a standalone program that does not run in Nengo's GUI. You cannot use simulation commands as shown in the

following in Nengo's GUI. Run this program in your Python interface. It shows yet another way to run the rich Nengo software.

The program contains additional headers for Matplotlib and distributions for data displaying and processing purposes:

```
import matplotlib.pyplot as plt
from nengo.utils.matplotlib import rasterplot
from nengo.dists import Uniform
```

The program contains the same architecture as `00-myintro.py`. It creates an ensemble, adds a function, and then connects the objects:

```
model = nengo.Network("Probing")
with model:
    ens = nengo.Ensemble(n_neurons=50, dimensions=1)
    #node_number = nengo.Node(output=0.5)
    node_function=nengo.Node(output=np.sin)
    nengo.Connection(node_function, ens)
    print(ens.probeable)
with model:
    # Connect the input signal to the neuron
    nengo.Connection(node_function, ens)
```

We will now add a probing function using `nengo.Probe`:

```
    # The original input
    function_probe = nengo.Probe(node_function)
    # The raw spikes from the neuron
    spikes = nengo.Probe(ens.neurons)
    # Subthreshold soma voltage of the neuron
    voltage = nengo.Probe(ens.neurons, 'voltage')
    # Spikes filtered by a 10ms post-synaptic filter
    filtered = nengo.Probe(ens, synapse=0.01)
```

To obtain some data, let's run the simulator for 5 seconds:

```
with nengo.Simulator(model) as sim:  # Create the simulator
    sim.run(5)
```

The simulator runs the calculation *before* displaying the outputs.

Then, we can probe the data and display it in numerical format. That way, we can retrieve output data from our system for further use, visualizing or chaining neuromorphic models to other algorithms in a few lines of code:

- **Decoded output**: The decoded output can be filtered (see the preceding filter):

```
print("Decoded output of the ensemble")
print(sim.trange(), sim.data[filtered])
```

The output data is then displayed or can be processed:

```
[1.000e-03 2.000e-03 3.000e-03 ... 4.998e+00 4.999e+00
5.000e+00] [[ 0.        ]
 [ 0.        ]
 [-0.03324582]
 ...
 [-1.26366121]
 [-1.22083471]
 [-1.18750863]]
```

Nengo can produce a chart with Matplotlib:

```
# Plot the decoded output of the ensemble
plt.figure()
plt.plot(sim.trange(), sim.data[filtered])
#plt.plot(sim.trange(), sim.data[node_function])
plt.xlim(0, 1)
plt.suptitle('Filter decoded output', fontsize=16)
```

The output of the preceding code is plotted as follows:

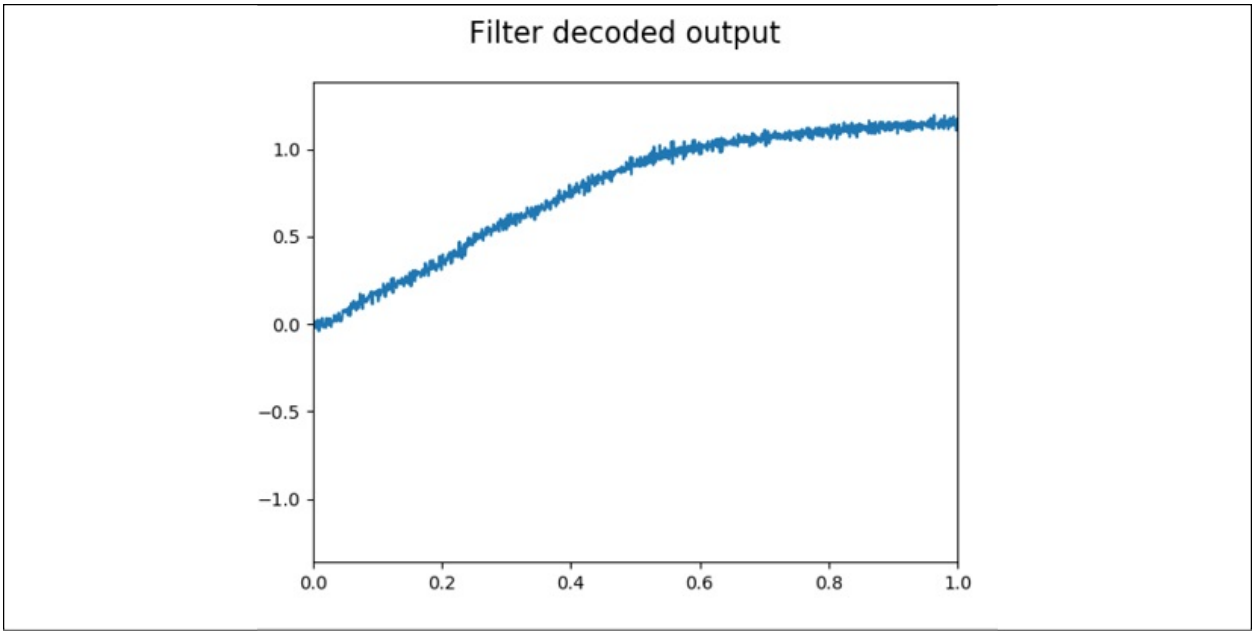


Figure 18.19: Decoded input

- **Spikes**: Spikes are retrieved in a single line of code:

```
print("Spikes")
print(sim.trange(),sim.data[spikes])
```

The output produces sequences of spikes:

```
[1.000e-03 2.000e-03 3.000e-03 ... 4.998e+00 4.999e+00
5.000e+00] [[   0.    0.    0. ...    0.    0.    0.]
 [   0.    0.    0. ...    0.    0.    0.]
 [   0.    0.    0. ... 1000.    0.    0.]
 ...
 [   0.    0. 1000. ...    0.    0.    0.]
 [   0.    0.    0. ...    0.    0.    0.]
```

The program produces a figure for spikes that matches the data:

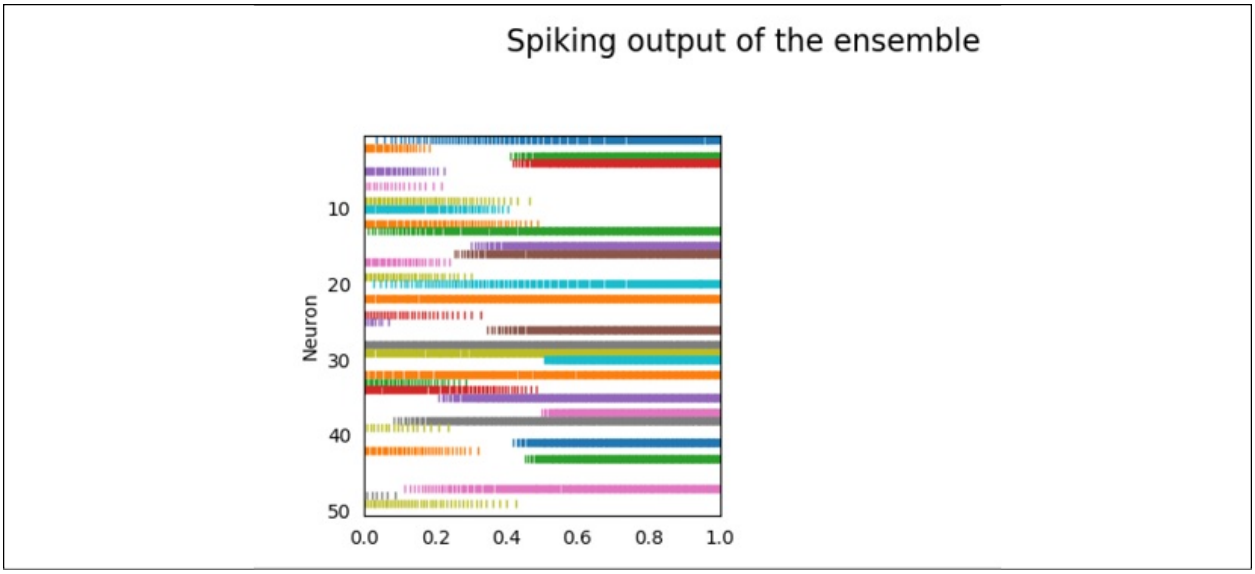


Figure 18.20: Spiking output

We can check the visual display with valuable raw data.

- **Voltage**: The simulation provides voltage data:

```
print("Voltage")
print((sim.trange(), sim.data[voltage][:, 0]))
```

The data is stored in a ready-to-use array:

```
(array([1.000e-03, 2.000e-03, 3.000e-03, ..., 4.998e+00
, 4.999e+00,
       5.000e+00]), array([0., 0., 0., ..., 0., 0., 0.]
))
```

The program produces a figure for the voltage of the ensemble as well:

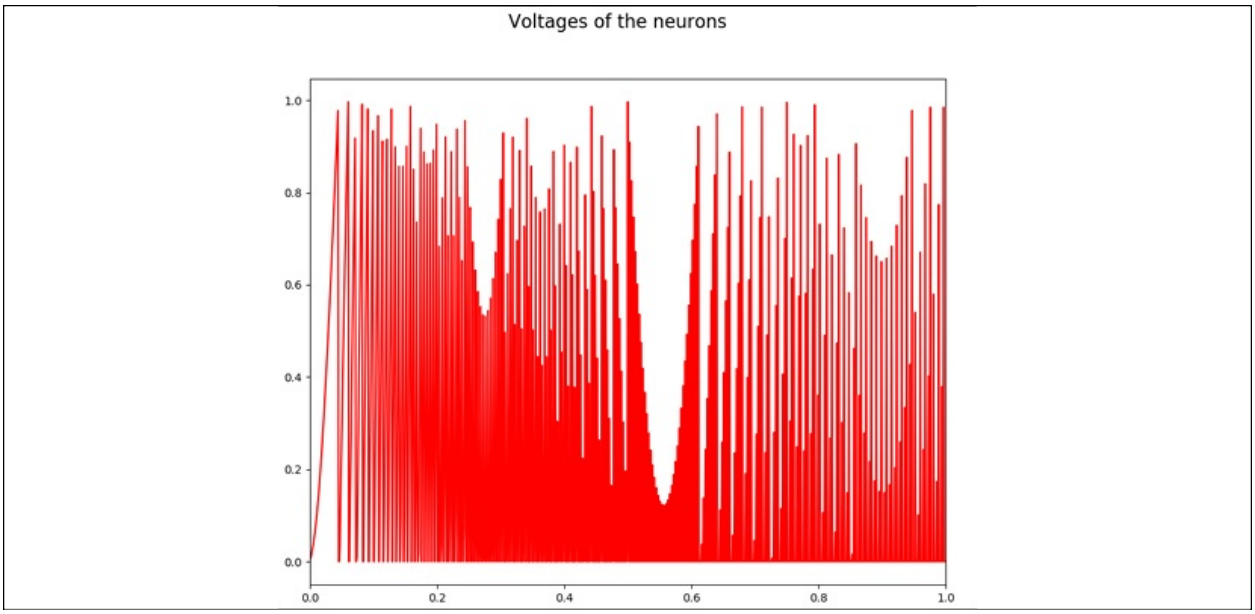


Figure 18.21: Neuron voltage

At this point, we have covered the main features of Nengo. We will now see how Nengo can boost research in critical AI research areas.

Applying Nengo's unique approach to critical AI research areas

It is useless to apply the power of brain neuromorphic models to simple arithmetic or classical neural networks that do not require any more than TensorFlow 2.x, for example.

But it is also a waste of time to try to solve problems with classical networks that neuromorphic computing can solve better with organic brain models. For example:

- Deep learning, TensorFlow 2. Convolutional models use a unique activation function such as ReLU (see *Chapter 9, Abstract Image Classification with Convolutional Neural Networks (CNNs)*). Neuromorphic neurons have a

variety of reactions when stimulated.

- Neuromorphic models integrate time versus more static DL algorithms. When we run neuromorphic models, we are closer to the reality of our time-driven biological models.
- The Human Brain Project, https://www.humanbrainproject.eu/en/, provides wide research and examples of how neuromorphic computing provides additional insights to classical computing.

I recommend testing a given problem with several AI tools and choosing the most efficient one.

The SPA examples demonstrate the efficiency of Nengo in several areas. All of the examples in the tutorial section are well documented and run well. You can apply the visualizing functions we explored in this chapter to them and also modify the code, experiment with them in your Python environment, and more.

I wish to highlight two domains: linguistics and weather representations:

- **Linguistics**: `25-spa-parse.py` processes neuronal signals and produces words. When you run it, the performances look magical, thanks to the SPA. The code is well documented.

  The program contains the mind-blowing `thalamus` module that can simulate the subcortical nuclei in our brains (forebrain and midbrain). This part of our brain, the basal ganglia, has high-density connections.

  This class can reduce or even eliminate low responses and intensify high responses to the stimulations. The program is worth running and exploring! The Nengo GUI makes it intuitive to understand:

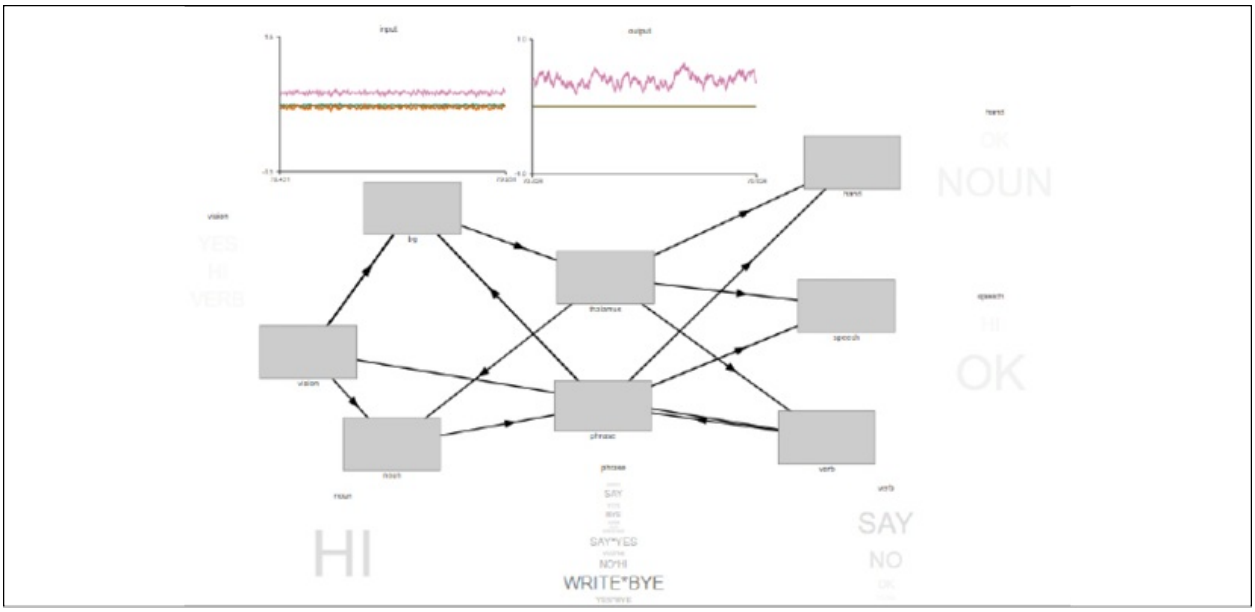


Figure 18.22: Nengo GUI options

- **Weather representations**: `15-lorenz.py` is not an SPA program. It is only a few lines long. It displays the graphs of the three basic Lorenz equations that represent temperatures and variations in the atmosphere. Nengo modified the code for educational purposes as explained in a publication at http://compneuro.uwaterloo.ca/publications/eliasmith2005b.html. The Nengo GUI displays an exciting representation:

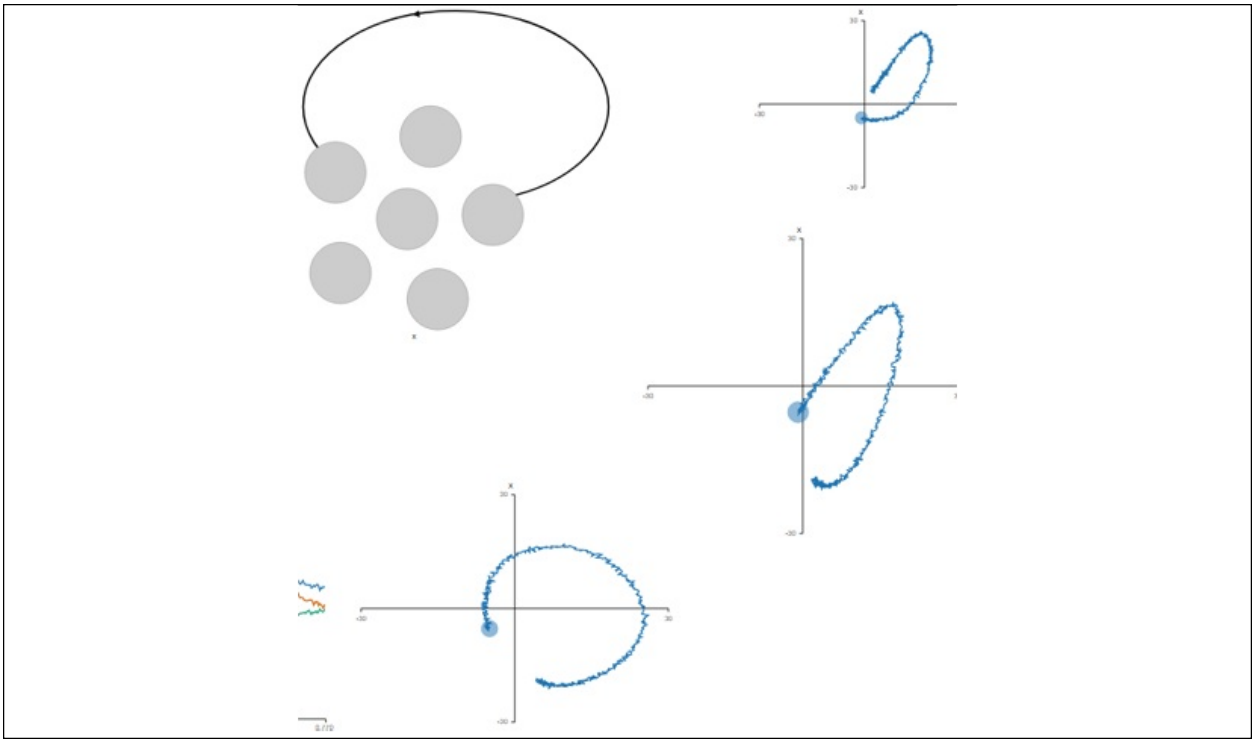


Figure 18.23: Lorenz equations

The code of `15-lorenz.py` is short, and the program looks simple. But weather forecasting is one of the toughest fields to represent events with AI models!

The potential of neuromorphic computing can be a real game-changer. Let's conduct a mind experiment. Imagine that:

- A hurricane is like a living organism
- That its center is connected to everything around it
- That it "feeds" on the heat and waters of our oceans
- That everything that is in it can be considered as small hurricane particles

Let's now continue the experiment by:

- Feeding the billions of particles in the neuromorphic model in a stream
- Using the power of a network of neuromorphic chips
- Using the calculation power of quantum computing (see *Chapter 19, Quantum Computing*) to perform computations with the input/output of the chip

- Applying SPA to the hurricane particle representations as if they were neurons and running predictions

I think the future of weather forecasting is in physical, neuromorphic models, that will take billions of parameters into account.

The result: we will be able to predict the course and level of a hurricane a few hours to a few days more in advance. This could save lives.

In a nutshell, neuromorphic computing has only just begun to demonstrate its worth. When neuromorphic chips hit the market, neuromorphic computing will grow exponentially.

Summary

In this chapter, we built neuromorphic Python programs from scratch. Populations of neurons, in Nengo ensembles, are made up of neurons. The system then has stimulation functions, connections, and probing objects. Nengo offers many other examples you can explore.

The NEF was designed to implement neuromorphic computing models. The novel concept of SPA shows that our brains have enhanced pointers that have a meaning and are linked to our physical data.

Neuromorphic computing opens tremendous horizons for a complex program that classical machine learning and deep learning cannot solve. Weather forecasting, with the power of the neuromorphic chips that are reaching the market, can tap into the complexity and variety of a machine brain. A machine brain can produce unique calculations by firing hundreds of thousands of neurons with both individual and collective behavior.

We have covered many algorithms and frameworks in this book. We have access to the most powerful intelligent algorithms in the history of humanity. From MDP to GA algorithms, and from KMC, KNN, PCA, NLP, and CUI algorithms to CNN, RBM, RNN, and LSTM networks, we have explored many AI tools. But there may be ways to build unified models. We will see in the years to come. In the meantime, neuromorphic computing makes our toolbox incredibly

intelligent. Google has TPU hardware to optimize TensorFlow, neuromorphic computing can rely on Intel chips, and many corporations are working to produce more innovative hardware.

The future will no doubt rely on hybrid architectures in which some or all of the AI tools will be built into meta-AI systems.

In *Chapter 19*, *Quantum Computing*, we will explore the exponential hardware available for quantum mechanics. Quantum computers, having no memory, rely on other systems to provide inputs and process outputs. Imagine a neuromorphic system chained to a quantum computer!

Questions

1. Neuromorphic computing reproduces our mental activity. (Yes | No)
2. Neuromorphic computing reproduces our brain activity. (Yes | No)
3. Semantic Pointer Architecture (SPA) is a hardware architecture. (Yes | No)
4. NEF stands for Neural Engineering Framework. (Yes | No)
5. Loihi is a classical chip. (Yes | No)
6. Reproducing our brain's neural activity cannot solve an equation. (Yes | No)
7. An ensemble in Nengo contains algorithms. (Yes | No)
8. Spiking blocks neuronal activity. (Yes | No)
9. Firing patterns can be used to analyze brain activity. (Yes | No)
10. Machine learning and deep learning are only metaphors of our brain's activity. (Yes | No)

References

Reference programs used for this chapter can be found at `https://www.nengo.ai`, `https://www.nengo.ai/examples/`.

Further reading

- **Research** – *How to Build a Brain*, *Chris Eliasmith*: This book provides the theoretical background for neuromorphic computing.

  Chris Eliasmith is also one of the designers of Nengo.

- **Software** – Nengo (https://www.nengo.ai/): Nengo is based on solid research, documentation, and an excellent community.
- **Hardware** – Intel: Intel is working hard to produce a neuromorphic chip (https://www.intel.fr/content/www/fr/fr/research/neuromorphic-computing.html).

19

Quantum Computing

IBM has begun to build quantum computers for research and business purposes. In a few years, quantum computing will become disruptive and will provide exponential computing power. Google, Xanadu, D-Wave, Rigetti, and others devote research budgets to quantum computing.

This unique computer capacity in the history of humanity opens the doors to obtaining results that would be impossible to obtain with classical computers. In 1994, Peter Shor showed that a quantum algorithm could perform better than a classical one for prime factors of an integer and the discrete logarithm problem. Then in 1995, Lov Grover added the unstructured search problem. With the rise of quantum computers, research can go much further.

Quantum computers in themselves will not provide revolutionary algorithms. Any algorithm can be broken down into components that run basic classical machines. Supercomputers can still run artificial intelligence algorithms much easier than implementing them with quantum computers. Quantum computers have no memory, for example, so they rely heavily on classical computers for the input and interpreting the output.

Though quantum computers have limits, in time, quantum computing power will take present-day algorithms beyond the limits of our imagination. Many corporations have invested in quantum computing research in the fields of banking, healthcare, cybersecurity and more.

This chapter explains why quantum computers are superior to classical computers, what a quantum bit is, how to use it, and how the quantum mind experiment could lead to a quantum thinking machine. The example provided is simply a recreative way to approach quantum computing.

We will be taking results from a classical machine, feed the results to quantum computer and then interpret the results provided by the quantum computing

algorithm. Some go further with hybrid quantum-classical algorithms, which is beyond the scope of this chapter.

In *Chapter 18, Neuromorphic Computing*, we explored how to use our brain to create neuromorphic models. In this chapter, we will create a higher level of representation with a research project to create a mind. Quantum mind is based on CRLMM, which I have been successfully applying to many corporate sites. In this research project, a quantum mind named MindX represents the mind of a random person. We will explore MindX in an exciting experiment.

The following topics will be covered in this chapter:

- Why quantum computers are more powerful than other classical computers
- What a quantum bit is
- The Bloch sphere
- Quantum computing
- How to build MindX, a thinking quantum computer research project

First, let's discuss some basics behind quantum computers, and what makes them so powerful.

> **Note**: This chapter is self-contained with screenshots of quantum circuits for those who do not wish to install anything before reading the chapter. IBM Q and Quirk can be used online without installing anything locally.

The rising power of quantum computers

Before we begin discussing MindX, an exciting research project to create a thinking quantum computer, we should start with the fundamentals. This section describes:

- Why quantum computers are faster

- What a qubit is
- How a qubit is measured (its position)
- How to create a quantum score (program) with quantum gates
- How to run a quantum score and use its results for a cognitive NLP chatbot

The goal of this chapter is not to go too deeply into the details of quantum computing, but rather to teach you enough to know how to build a thinking quantum computer. This chapter is simply meant to show how a quantum computer works and open ourselves to new ways to use computers.

Quantum computer speed

A standard computer bit has a 0 or a 1 state. A classical computer will manage 0 or 1, as the system chooses, but it remains limited to choosing 1 or (XOR) 0. It cannot manage both states at the same time.

A quantum computer is not constrained by an XOR state. It is an AND state. It can manage values between 0 and 1 at the same time until it is measured! A quantum state is unknown until it's observed, which means that a quantum program can use values between 0 and 1 at the same time. Once observed, the qubit will take a value of 0 or 1 because of the physical instability of a quantum state.

> Just observing a quantum state makes the state break down. In quantum computing, this is called **decoherence**. It is not magic. Qubits are unstable. When observed, the quantum state breaks down.

This means quantum computing is memoryless once it is measured. Storage does not exist in a quantum computer. The input is made by a classical computer, and the output goes back to a classical computer, to be stored through the following process:

1. A classical computer provides an input
2. A quantum computer processes the input and produces an output
3. A classical computer interprets the output

That being said, the computing power of a quantum computer fully justifies this architectural constraint.

Until a qubit is observed, it can have a 0 or 1 state or a probability in between, such as 0.1, 0.7, or 0.9.

Observing the situation is called measuring the state. When measured, only 0 or (XOR) 1 will become the result.

Until the state is measured, a large number of probabilities are possible. If a qubit is added to the system, we now have two qubits and four elementary combinations all at the same time.

Unlike standard computer logic, all four of the states can be used to compute algorithms at the same time in a parallel process. The volume of the possible states of a given algorithm will thus expand with the number of qubits involved. An estimation of the volume of the states can be made with the following number, in which *q* is the number of qubits:

$$2^q$$

Looking at this tiny equation does not seem awesome at all. Now, let's see what it looks like in a loop that runs up to 100 qubits with the number, `nb`, of possible states:

```
import numpy as np
for q in range(101):
    v=(2**q)
    print("Size of nb to describe",q," qubits:","{:,}".form
at(v))
```

The program does not appear fearsome either. However, the following output is awesome:

```
Size of nb to describe 0 qubits: 1
Size of nb to describe 1 qubits: 2
Size of nb to describe 2 qubits: 4
Size of nb to describe 3 qubits: 8
Size of nb to describe 4 qubits: 16
Size of nb to describe 5 qubits: 32
Size of nb to describe 6 qubits: 64
```

```
...
Size of nb to describe 10 qubits: 1,024
...
Size of nb to describe 50 qubits: 1,125,899,906,842,624
...
Size of nb to describe 97 qubits:
158,456,325,028,528,675,187,087,900,672
Size of nb to describe 98 qubits:
316,912,650,057,057,350,374,175,801,344
Size of nb to describe 99 qubits:
633,825,300,114,114,700,748,351,602,688
Size of nb to describe 100 qubits:
1,267,650,600,228,229,401,496,703,205,376
```

Presently, big data is often calculated in petabytes. A petabyte=10^{15} or about 2^{50} bytes.

Facebook stores data for over 2 billion accounts. Imagine Facebook managing 500 petabytes on a given day. Let's see what 500 petabytes approximately add up to in the following code:

```
print("Facebook in the near future:")
s=(2**50)*500
print("{:,}".format(v))
```

The output is quite surprising because it is about the size of data a 100-qubit quantum computer can compute in one run:

```
A segment of Facebook data :
1,267,650,600,228,229,401,496,703,205,376
```

This means that a single quantum computer with 100 qubits can run a calculation of the size of all the data that over 2,000,000,000 Facebook accounts might represent in the near future.

A quantum computer would not actually contain that volume of data at all, but it shows that one could produce a calculation with that volume of computation information.

More importantly, this also means that a single quantum computer can run a mind-dataset of a single mind (see the next section) and calculate associations.

This thinking process can generate an exponential volume of connections.

A classical *n*-bit computer manages *n* bits whereas a quantum computer will manage 2^n bits or 2^q bits.

Compared to quantum computers' 2^q exponential power, soon, classical computers will seem like relics of the past for scientific calculations. Classical computers will still be in use, but quantum computers will be the tools with which to explore the world beyond the present limits of artificial intelligence.

> Visualize all the AI solutions you have seen in this book. They will already seem to have some dust of the past on them once you get your hands on quantum computing.

Quantum computers will beat any other computer in many fields in the years to come. In one parallel computation, a quantum computer will do in one run something that it would take a classical computer years to calculate.

Now think about what a network of many quantum computers could do!

Often, we try to compare large volumes with the number of stars in the universe and we say, "That's more than the number of stars in our universe." We must now look in the opposite direction, at very small numbers.

The lesson is clear: the future lies in nano models. Quantum computing represents both a challenge and an opportunity.

Defining a qubit

A qubit, a quantum bit, has a physical counterpart. For example, a quantum state can be encoded in oscillating currents with superconductor loops. Google and IBM have experimented with this approach. Another way is to encode a qubit in an ion trapped in an electromagnetic field in a vacuum trap.

Photons, electrons, the state of light, and other technologies have emerged. Whatever the approach, calculations done by over 50-qubit quantum computers will outrun classical supercomputers.

The competition is fierce because the market will rapidly become huge. Get ready now to face the disruption that's coming!

Representing a qubit

The mathematical representation of a qubit is:

- $|0\rangle$ for a 0 value
- $|1\rangle$ for a 1 value
- $\alpha|0\rangle$ where alpha is a probability parameter
- $\beta|1\rangle$ where beta is a probability parameter

These brackets are called bracket or bra-ket notation.

This linear representation is called superposition. In itself, it explains most of the power of quantum computing.

The superposition of 0 and 1 in a quantum state can thus be expressed as follows in kets such as $|1\rangle$ and not bras such as in $\langle 1|$:

$$\text{quantum state} = \alpha|1\rangle + \beta|0\rangle$$

The alpha and beta probabilities look like weights, and the total probabilities of those probable states of qubit must add up to 1. We use partition functions, softmax, and other techniques to make sure to keep the sum of probabilities equal to 1. This is no surprise since computer geeks like us designed the way to program quantum computers.

Translated into mathematics, this means that the probabilities of α and β must add up to 1. In the case of qubit probabilities, the values are squared, leading to the following constraint:

$$|\alpha^2| + |\beta^2| = 1$$

> To describe the probable state of a qubit, we thus need three numbers: the 0 and 1 possible states and a number to determine the value of the probabilities (the other is implicit since the total must add up to 1).

Since qubits interact in their respective states, an interaction is described as an entanglement. An entanglement designates at least two interacting qubits. They cannot be described without taking all the states of all the qubits into account.

This has been reproduced physically, which means that this entanglement seems strange because their quantum entanglement (relation) can occur at a distance. One qubit can influence a qubit that is physically far away.

It's an odd way to think, and one that was not readily accepted at first. Albert Einstein is oft-quoted as referring to entanglement derisively as "spooky action at a distance."

The position of a qubit

One of the main ways to represent the state of a qubit is by using the Bloch sphere. It shows how a qubit spins and can be used to describe qubit states. To properly grasp this, the following section will first provide a refresher on some properties of a circle.

Radians, degrees, and rotations

The radius is the distance between the center of a circle and its circumference, as shown in the following diagram:

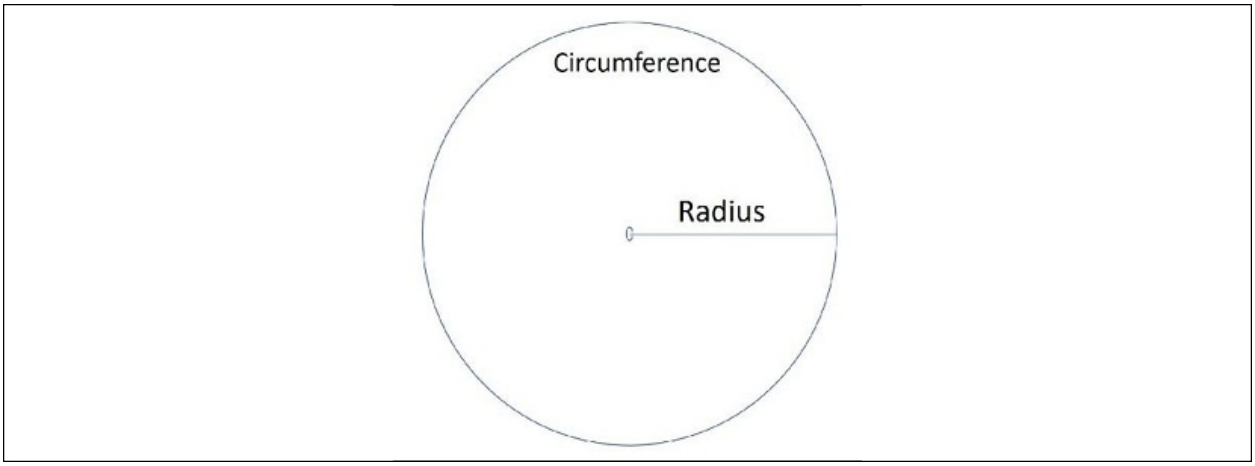


Figure 19.1: Radius of a circle

The radius of a circle is half of the circle's diameter. The relation between the radius *r* and the circumference *C* is (where $\pi = 3.14$):

$$C = 2\pi r$$

If the length of the radius is wrapped around the circumference of a circle, that arc forms a radian, as shown in the following diagram:

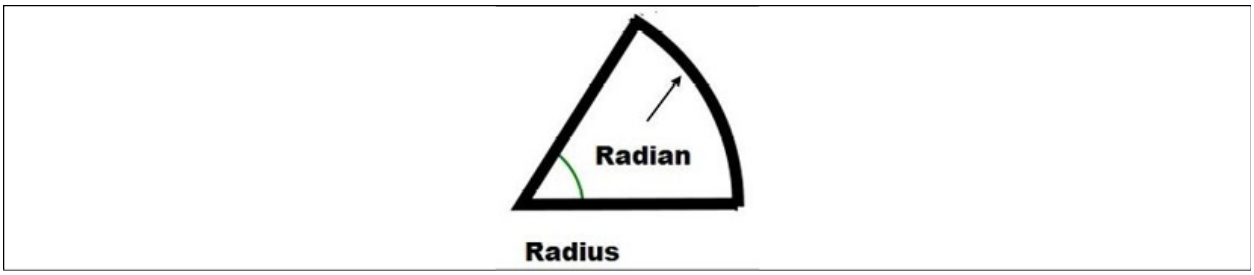


Figure 19.2: A radian

The angle formed by a radian is equal to about 57.29° (degrees).

The properties of the radian can be used for rotations:

- 3.14 × 57.29° = about 180°
- Thus π radians = 180°

Rotations are often described by radians expressed in π, as displayed in the following table:

Degrees	Radians
30°	$\pi/6$
45°	$\pi/4$
60°	$\pi/3$
90°	$\pi/2$
180°	π

270°	$3\pi/2$
360°	2π

Now that we've had that recap, let's explore the Bloch sphere.

The Bloch sphere

The radian table shown just now is a practical way to describe rotations. The Bloch sphere, shown in the following figure, provides a visual representation of the position and rotation of a qubit:

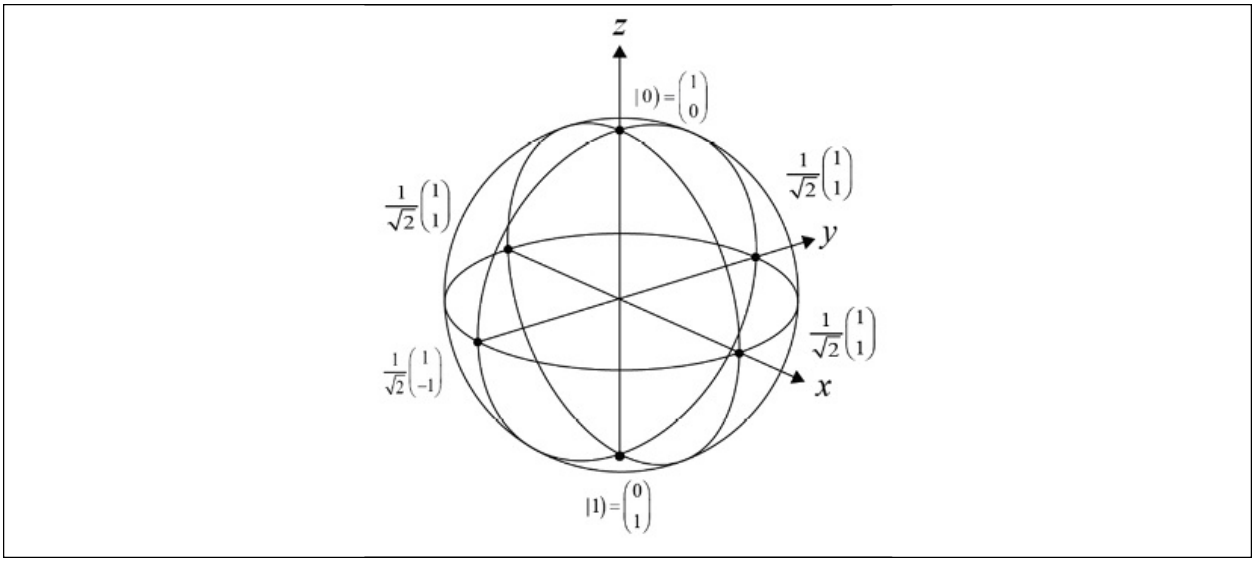


Figure 19.3: Bloch sphere

The North and South Pole (polar coordinates) represent the basic states of a qubit:

$$\text{"North" pole} = |1\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$\text{"South" pole} = |0\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

A qubit can take any value on the sphere.

Composing a quantum score

Composing a quantum score consists of positioning gates on a stave (or circuit) and adding a measurement. The input comes from a classical computer. After the measurement phase, the output goes back to a classical computer. The reason is that quantum computers have no memory and thus cannot store their intermediate states because of their instability.

This section uses Quirk, a very educative quantum circuit simulator, to present quantum gates and a quantum composer.

You can access Quirk online at this link:
https://algassert.com/quirk.

Quantum gates with Quirk

Qubits are represented by lines, and they start on the left, as shown in the following quantum gate programming interface:

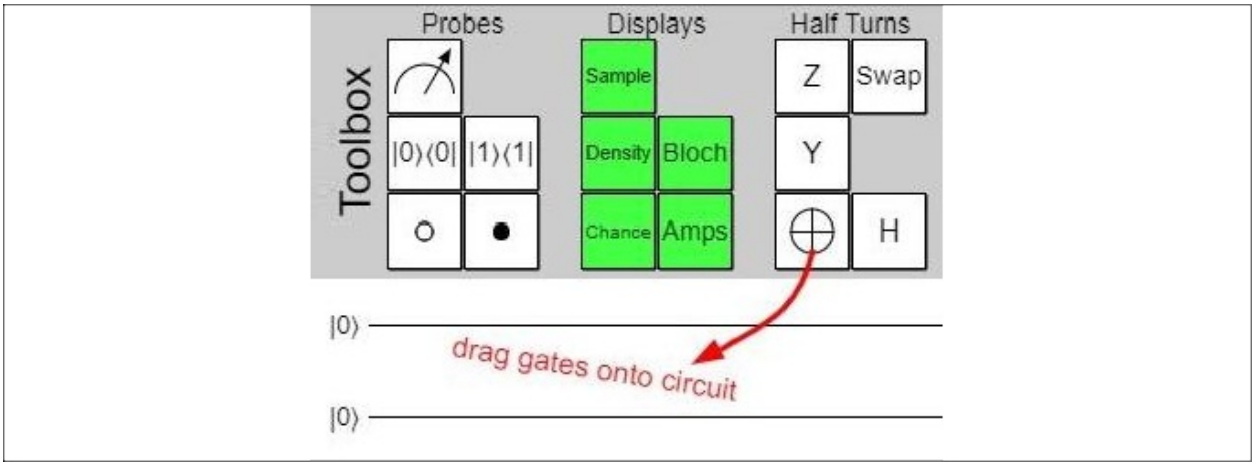


Figure 19.4: Quantum gate programming interface

The gates are logic gates that will transform the state of the qubits.

NOT gate

A NOT gate will transform a ket-zero $|0\rangle$ into a ket-one $|1\rangle$. It will transform a ket-one $|1\rangle$ into a ket-zero $|0\rangle$.

In the circuit description, **On** is the ket-one state and **Off** is the ket-zero state, as shown in the following quantum score:

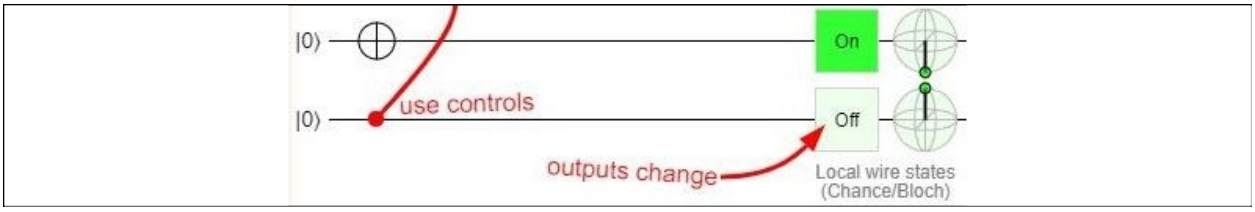


Figure 19.5: Circuit description of a quantum score

You can see that:

- A NOT gate symbol is a circle with a vertical and horizontal line inside it
- The **On** status means that the state is $|0\rangle$
- The Bloch sphere representation is at π (starting from the top of the Bloch sphere, as it should)

H gate

An H gate or Hadamard gate will perform the following transformation:

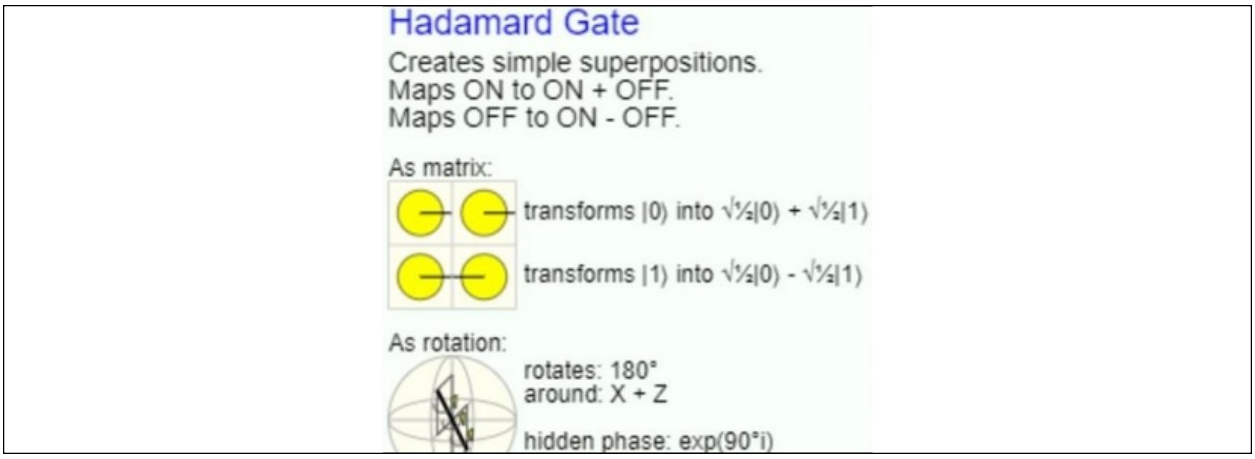


Figure 19.6: Hadamard gate transformation

The following 50% chance will be displayed in a rectangle and the position on the Bloch sphere will be displayed as well:



Figure 19.7: 50% chance in a rectangle and on the Bloch sphere

The fundamental role of a gate is to rotate the qubits on a Bloch sphere and produce a probable condition if measured. There are many possible gates to be explored and used, as shown in this design menu diagram:

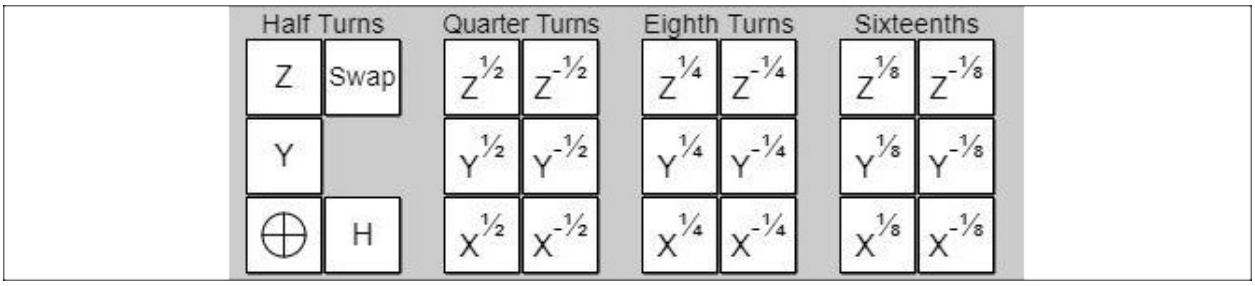


Figure 19.8: Gates design menu

These gates are more than enough to build many algorithms.

A quantum computer score with Quirk

The goal here is to play around with the interface to intuitively see how circuits run.

Building a quantum score (or circuit) with Quirk means the following:

- Dragging and dropping gates that will make a qubit turn in a specific direction and produce probable outcomes
- Adding another qubit, doing the same, and so on
- Being able to perform an intermediate measurement, although this is impossible for a real physical quantum computer (the observations make the system collapse)

A score is represented as follows, for example:

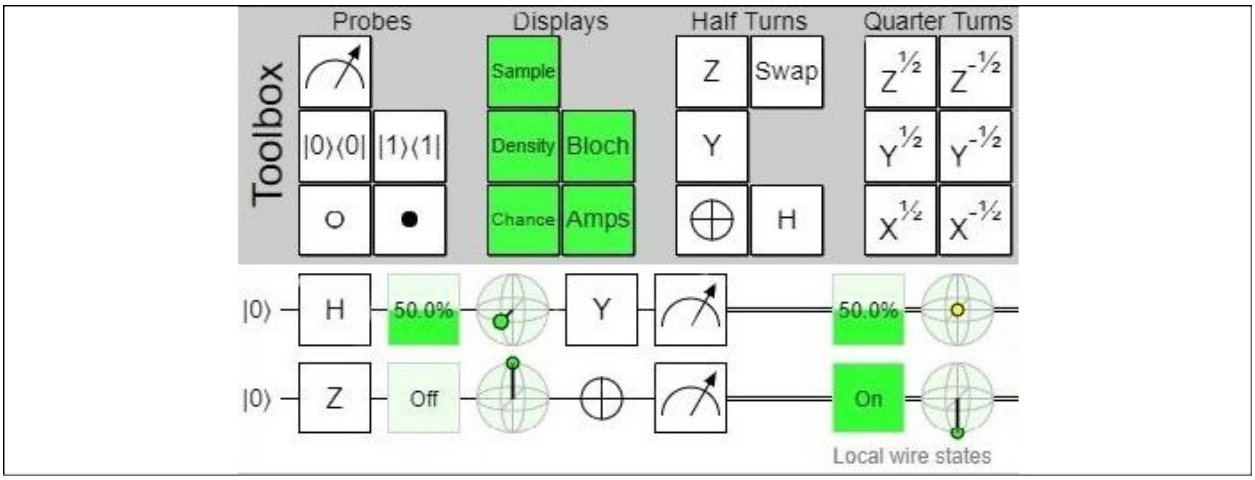


Figure 19.9: Quantum computer score representation

There are two qubits with a gate on each line to start. Then the intermediate result is shown, making the simulator very educational. Then two more gates are added. Finally, the following measurement probe is added at the end:



Figure 19.10: Measurement probe

Once the measurement is made, the final result is displayed on the right of the measurement symbols.

A quantum computer score with IBM Q

IBM Q provides a cloud platform to run a real physical quantum computer.

Create a free account and access the IBM quantum computing composer. Just as with Quirk, quantum gates are dragged on the following score, as shown in this diagram:

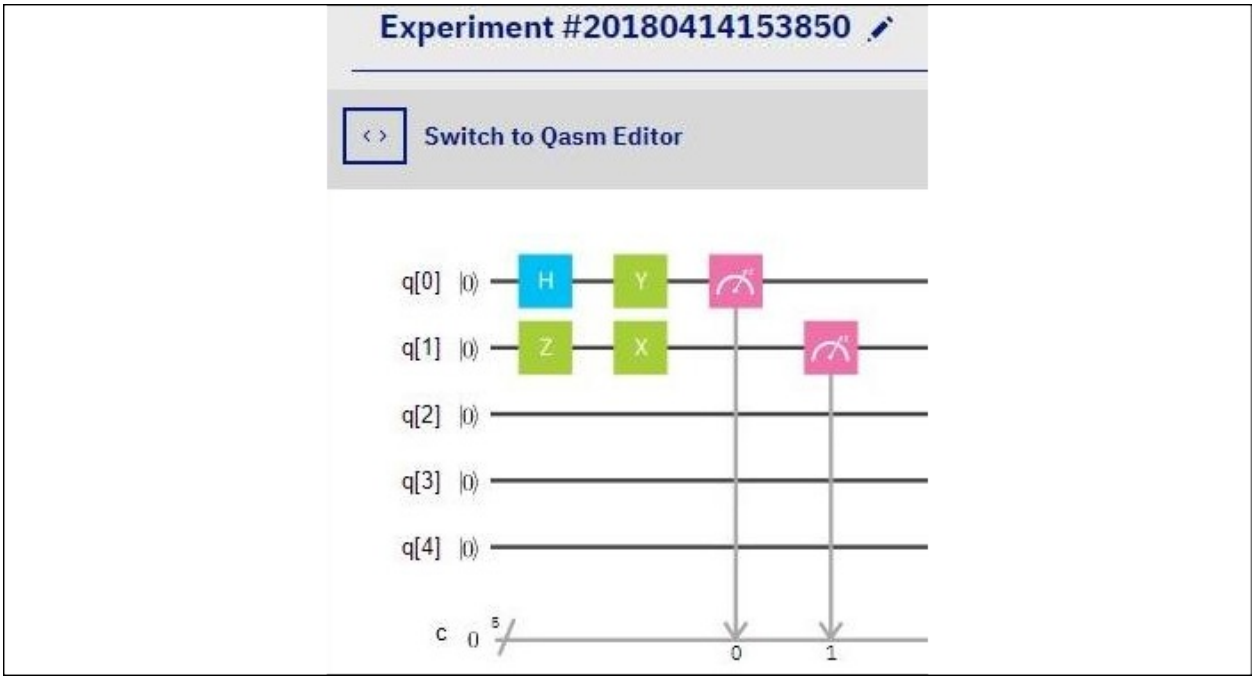


Figure 19.11: Quantum gates dragged

The score can be run on a simulator like Quirk or a real quantum computer, as shown in this interface diagram:



Figure 19.12: Score interface

Click on **Simulate**, which will run a simulator.

> **Run** launches a calculation on IBM's physical quantum computer. It is an exhilarating experience! The future is at the tip of your fingers.

The following output is interesting. It is a bit different from Quirk for the same score, but the probabilities add up to 1 as expected:

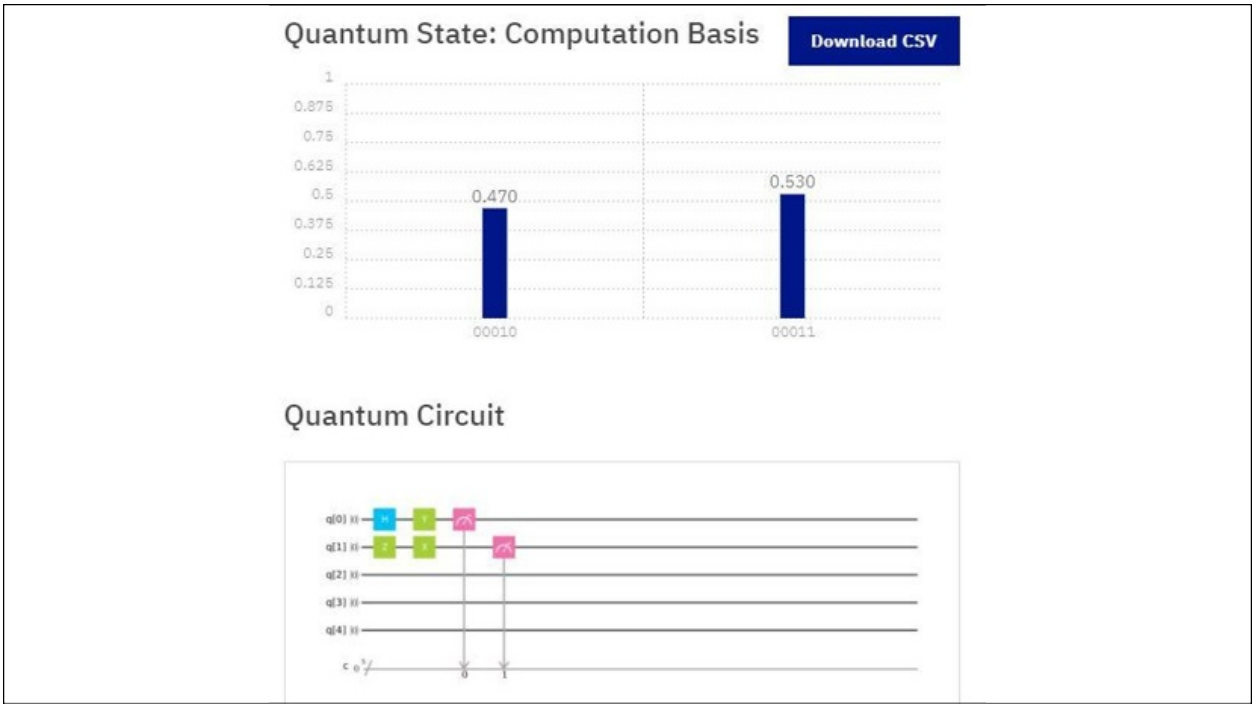


Figure 19.13: Quantum computer score output

IBM also possesses a source code version (QASM) of the score, as shown in the following code:

```
include "qelib1.inc";
qreg q[5];
creg c[5];
h q[0];
z q[1];
y q[0];
x q[1];
measure q[0] -> c[0];
measure q[1] -> c[1];
```

This language is an Open Quantum Assembly Language. It can be written in an editor like any other language on IBM's Q platform. A development kit can also be downloaded, and the APIs are functional. IBM provides extensive and detailed documentation on all the aspects of this approach.

All of this being said, let's find out how quantum computing can boost an AI project. It is now time to define the thinking quantum computer project.

A thinking quantum computer

A thinking quantum computer is not a reproduction of brain function, but rather a representation of the mind of a person. Neuromorphic computing is one approach that represents our neurons and how brain uses spiking neurons to think. Quantum computing can provide an exciting way to imitate our mind's capacity with mathematical algorithms that work with qubits, not neurons. We've just begun to explore the potential of these approaches, which in fact will most probably merge into an ensemble of hybrid software and hardware solutions.

The endeavor of the quantum MindX experiment is to build a personal mind named MindX, with memories of past events, conversations, chats, and photographs stored on a classical computer. The program will then transform subsets of the data into quantum circuits to see what happens, how our quantum circuit behaves when left to produce millions of possibilities.

This section describes how to build MindX, a thinking computer that is just a research project. It must be noted that this approach is experimental. It could be viewed as an advanced mind experiment. I've been doing research on this subject for many years. The power of quantum computation will no doubt boost research in this field.

Representing our mind's concepts

The input consists of encoding a state of mind of a PCA CRLMM representation, as we built in *Chapter 14, Preparing the Input of Chatbots with Restricted Boltzmann Machines (RBMs) and Principal Component Analysis (PCA)*. The CRLMM representation is not a general dictionary or encyclopedia dataset but a mind-dataset of actual personal data that is collected, classified, and transformed into a PCA.

Expanding MindX's conceptual representations

MindX's mind dataset will grow continuously if implemented beyond this research experiment. It will have sensors to process body temperature for emotion classification, facial recognition for expression detection, body

language detectors, and more. All of this technology is already available. The New York Stock Exchange already has artificial intelligence IoT agents that gather information from outer sources to make decisions. These AI agents have replaced a large amount of human decision-making. Imagine what is going to happen when quantum computing becomes disruptive!

The MindX experiment

The aim of the quantum MindX experiment is to build a mind and let it think with the power of a quantum computer. This section will show how to run a 16-qubit mind.

The size of the numbers needed to describe a 16-qubit quantum simulation is 65,536.

This section first describes how to do the following:

- Prepare the data
- Create and run a quantum score
- Use the output

Let's get on with preparing the data.

Preparing the data

To prepare data for higher-dimension calculations, I have been using a concept encoding method for corporate projects for over 30 years to provide embedded data to the algorithms I have developed. This guarantees high-level abstraction to solve problems. It is a very profitable way to implement transfer learning and domain learning. This way, you can apply the same model to many different fields.

The method consists of embedding data with the methods described in this book. The goal always remains the same—to transform the data points into higher dimensions to visualize features.

For quantum computing, the method remains the same for the MindX

experiment.

Transformation functions – the situation function

Two functions need to be applied before creating and running a score: a state of mind function and a quantum transformation function.

The situation function consists of building a vector of features in PCA dimensions. We accomplished this in *Chapter 14*, *Preparing the Input of Chatbots with Restricted Boltzmann Machines (RBMs) and Principal Component Analysis (PCA)*, and applied it to a chatbot in *Chapter 15*, *Setting Up a Cognitive NLP UI/CUI Chatbot*. The chatbot was able to use person X's mind-dataset to build a meaningful dialog.

In this chapter, the quantum dimensions will provide the chatbot with a nascent personal mind, not a general mind that can recognize, classify, and predict like a machine.

MindX has a mind that is biased by its way of thinking like humans. MindX has an exceptionally open mind to adapt to everyone, which gives it empathy.

MindX can doubt. Thus it can learn better than dogmatic machines.

A situation function will create a situation matrix for a movie suggestion by the MindX bot, which will communicate with consumers. In a situation where a movie suggestion is made by the MindX bot, for example, it could be something as follows for 16 qubits:

Qubit	Concept	Image	Initial polarity
1	cities	parking	0.146
2	darkness	dark forest	0.5
3	nostalgia	autumn leaves	0.5
4	worrying	dark background	0.146

5	job	sad face	0.5
	…	..	..
15	consider movie "Lost"		0.38
16	decision to suggest "Lost"		0.0

MindX is not analyzing person X anymore. It has now loaded an empathy matrix from its mind-dataset, which contains data and sentiment analysis *polarity*. Values close to 1 are positive. Values close to 0 or negative are negative. MindX loads its mind plus the mind of another person.

> Empathy colors your thoughts and feelings with the thoughts and feelings of another person.

The 16-qubit matrix shown just now contains four columns, as shown in the preceding table:

- **Qubit**: The line for this qubit on the quantum composer
- **Concepts and mental representation**: The concept loaded in MindX's situation dataset derived through the process described in *Chapter 14*, *Preparing the Input Chatbots with Restricted Boltzmann Machines (RBM) and Principal Component Analysis (PCA)*, in which we ran an RBM to generate features we represented through a PCA.

The mental representation appears as shown in the following PCA:

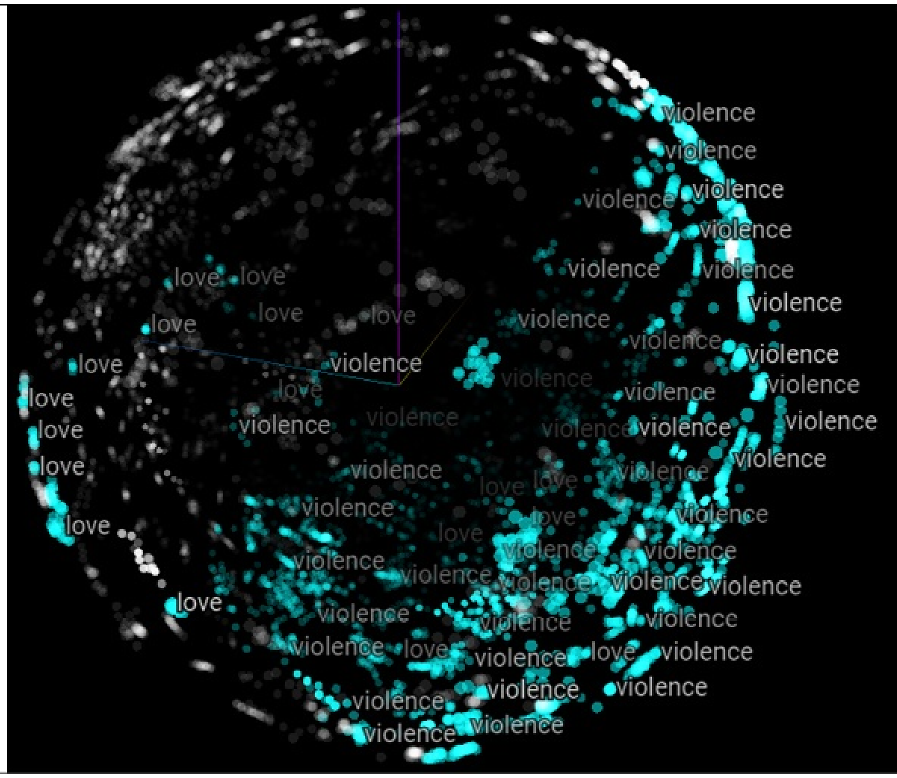


Figure 19.14: PCA representation of a "mind"

These features, drawn from the RBM on the movie preferences of a particular age segment, can now be used as input in a quantum circuit to generate random mental activity with information and noise.

> This movie example is for explaining how to create a thinking, empathetic chatbot. This approach can be applied to other commercial market segments or any situation in which a chatbot is required to think beyond preset answers. Many fields, such as healthcare, pharmaceutical research, and security, will benefit from mind-opening quantum algorithms.

Transformation functions – the quantum function

The algorithms have produced a qubit line number with labels (concepts and images). Each line also possesses a sentiment analysis polarity expressed in normalized values from 0 to 1 in probabilistic format. We will consider negative and positive views of the mind reacting to an object. Keep in mind that "negative" is close to 0, "positive" is close to 1, and the intermediate values give a more detailed approximation. 0.4 to 0.6 is a turning point.

This first transformation into another dimension is the first step to initialize the quantum transformation function. The quantum transformation function consists of initializing the first column of all 16 qubits with a quantum gate.

Just like datasets were transformed into principal component features, the polarity transformation function enables the quantum transformation function to bring the dataset into a quantum universe.

The function will automatically find the quantum gates that represent the normalized sentiment analysis polarity of the data points of the following situation matrix:

Initial polarity	**Quantum gate**
0.146	$X^{1/4}$
0.5	$X^{1/2}$
0.5	$X^{1/2}$
0.146	$X^{1/4}$
0.5	$X^{1/2}$
..	
0.38	$X^{1/8}$
0.0	$Z^{1/8}$

The state of mind of MindX, a random person, is now in a matrix, and its concepts have now been transformed into a quantum dimension, in which the data status can be traced throughout the creation of the quantum score.

Creating and running the score

A thought process of MindX is described in a quantum score. There are two ways to build this score:

- Manually, just like a musician writing a music score. This requires thinking about the previous logic gate and the effect of the next one, taking a decision process into account. It is like writing any artificial intelligence program.
- Automatically, by building a function that reads a rule base of MindX's way of thinking and applies the quantum gates through that rule base. You can apply this to machine learning and deep learning as well to test the performance of a quantum computer versus a classical one.

In any case, it requires writing an algorithm. This algorithm is a decision-making algorithm that takes emotions into account. Doubt, for example, is what keeps many humans from making bad decisions. Too much doubt, for example, will make a person back out of a situation.

Here is the quantum transcription of an algorithm that takes MindX's concepts into account and represents how they interact. This requires very precise thinking and cognitive science programming.

The following is an experimental quantum score I built with Quirk:

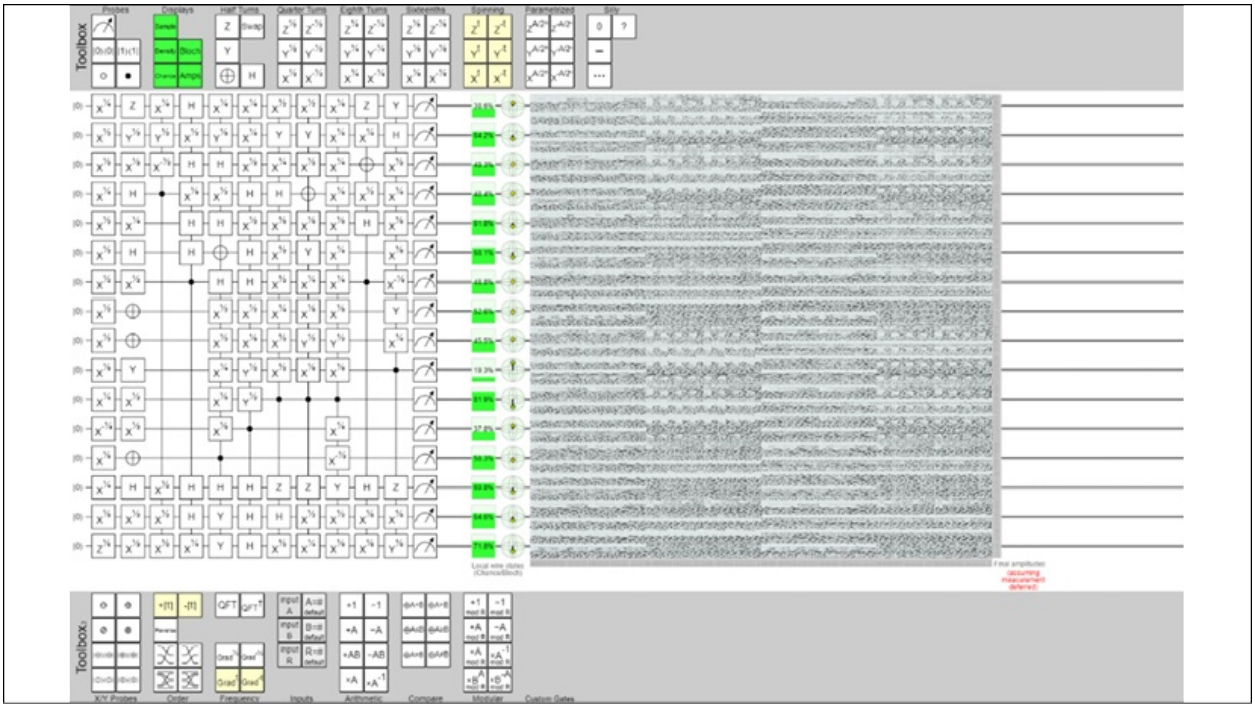


Figure 19.15: Quantum score built with Quirk

Once the quantum score has been run and the measurement has been done, the green rectangles on the right provide the output.

Using the output

The output of the quantum score is now added as a column to the situation matrix. If you want to implement such solutions, just bear in mind that it will take some time to write the functions with some sweat and tea to make it through the nights. MindX could be used to enhance a chatbot with unplanned responses. MindX has proven it has imagination potential.

Qubit	Concept	Image	Initial normalized polarity expressed in quantum gate form	Quantum output directly interpreted as sentiment analysis polarity
1	cities	parking	0.146	0.677
2	darkness	dark forest	0.5	0.691
		autumn		

3	nostalgia	leaves	0.5	0.5
4	worrying	dark background	0.146	0.48
5	job	sad face	0.5	0.36
	…	..	..	
15	consider movie "Lost"		0.38	0.82
16	decision to suggest "Lost"		0.0	0.75

MindX has given a 65,536 quantum state description of its thoughts about suggesting a given movie to person X. Lines 15 and 16 show that the normalized polarity value has risen over 0.5 to a positive feeling about the movie.

The reasoning is that the first lines show that MindX feels person X's doubts about life at that moment:

- That person X will identify with the "Lost" movie
- The movie has a happy ending (MindX knows that through the features of the movie)
- That person X's spirits will be most probably lifted after watching the movie

You can experiment with building quantum scores. You can use Quirk without installing anything and exploring the list of circuits available in many areas: Grover's search, Shor's period-finding, quantum Fourier transform, and more.

Summary

Quantum computers have opened the door to scientific experiments that could never have been carried out with classical computers. Within a few years, quantum computers will have become mainstream, unavoidable, and a key asset for businesses and research labs. The race has begun to conquer the market.

CRLMM applied to quantum computers could make MindX one of the most powerful thinking minds on earth—human or machine.

With an unlimited mind-dataset and a 2^q quantum computer starting at 250, a 50-qubit machine, MindX could gain the thinking power and experience of a human who has lived for 1,000 years. MindX's thinking power and an exponential amount of real-time memory of past experiences, loaded through transformation functions, could help solve many medical, logistic, and other decision-making problems.

Quantum thinking has just begun to change the perception of the world. Conceptual AI models such as CRLMM will no doubt be the starting point for the next generation of AI solutions. These CRLMM models will be much more powerful because they will be gifted with empathy and complex minds.

Hopefully, this overview of quantum computing will open the doors of your imagination to the new world awaiting you!

> Artificial intelligence has only just begun its long journey into our lives. Always trust innovations. Never trust a solution that solves a problem without opening the door to a universe of questions and ideas!

Questions

1. Beyond the hype, no quantum computer exists. (Yes | No)
2. A quantum computer can store data. (Yes | No)
3. The effect of quantum gates on qubits can be viewed with the Bloch sphere.

(Yes | No)

4. A mind that thinks with past experiences, images, words, and other bits of everyday information, like stored memories, will find deeper solutions to problems that mathematics alone cannot solve. (Yes | No)
5. A quantum computer will solve medical research problems that cannot be solved today. (Yes | No)
6. A quantum computer can solve mathematical problems exponentially faster than classical computers. (Yes | No)
7. Classical computers will soon disappear and smartphone processors too. (Yes | No)
8. A quantum score cannot be written in source code format but only with a visual interface. (Yes | No)
9. Quantum simulators can run as fast as quantum computers. (Yes | No)
10. Quantum computers produce intermediate results when they are running calculations. (Yes | No)

Further reading

- **Theory**: *Quantum Computation and Quantum Information: 10th Anniversary Edition, Michael Nielson, Isaac L. Chuang*
- Explore IBM Q and discover how you can implement quantum scores: https://www.ibm.com/quantum-computing/
- Use Quirk, an intuitive quantum score designing tool: http://algassert.com/2016/05/22/quirk.html

Answers to the Questions

Chapter 1 – Getting Started with Next-Generation Artificial Intelligence through Reinforcement Learning

1. **Is reinforcement learning memoryless? (Yes | No)**

   The answer is **yes**. Reinforcement learning is memoryless. The agent calculates the next state without looking into the past. This is significantly different from humans. Humans rely heavily on memory. A CPU-based reinforcement learning system finds solutions without experience. Human intelligence merely proves that intelligence can solve a problem. No more, no less. An adaptive thinker can then imagine new forms of machine intelligence.

   It must be noted that exceptions exist in some cases, but the general rule is a memoryless system.

2. **Does reinforcement learning use stochastic (random) functions? (Yes | No)**

   The answer is **yes**. In the particular Markov decision process model, the choices are random. In just two questions, you can see that the Bellman equation is memoryless and makes random decisions. No human reasons like that. Being an adaptive thinker is a leap of faith. You have to leave who you were behind and begin to think in terms of equations.

3. **Is the MDP based on a rule base? (Yes | No)**

   The answer is **no**. Human rule-based experience is useless in this process. Human thinking is often based on rules of cause and effect, for example. Furthermore, the MDP provides efficient alternatives to long consulting times with future users who cannot clearly express their problem.

4. **Is the Q function based on the MDP? (Yes | No)**

   The answer is **yes**. The use of the expression *Q* appeared around the time the Bellman equation, based on the MDP, came into fashion. It is more trendy to say you are using a Q function than to speak about Bellman, who put all of this together in 1957. The truth is that Andrey Markov was Russian and applied this method in 1913 using a dataset of 20,000 letters to predict the future use of letters in a novel. He then extended that to a dataset of 100,000 letters. This means that the theory was there 100 years ago. *Q* fits our new world of impersonal and powerful CPUs.

5. **Is mathematics essential to AI? (Yes | No)**

   The answer is **yes**. If you master the basics of linear algebra and probability, you will be on top of all the technology that is coming. It is worth spending a few months' worth of evenings on the subject or taking a MOOC. Otherwise, you will depend on others to explain things to you.

6. **Can the Bellman-MDP process in this chapter apply to many problems? (Yes | No)**

   The answer is **yes**. You can use this for robotics, market analysis, IoT, linguistics, and scores of other problems.

7. **Is it impossible for a machine learning program to create another program by itself? (Yes| No)**

   The answer is **no**. It is not impossible. It has already been done by DeepCode: https://www.deepcode.ai/.

   Do not be surprised. Now that you have become an adaptive thinker and know that these systems rely on equations, not humans, you can easily understand the fact that mathematical systems are not that difficult to reproduce.

8. **Is a consultant required to enter business rules in a reinforcement learning program? (Yes| No)**

   The answer is **no**. It is only an option. Reinforcement learning in the MDP process is memoryless and random. Consultants are there to manage,

explain, and train in these projects.

9. **Is reinforcement learning supervised or unsupervised? (Supervised | Unsupervised)**

   The answer is **unsupervised**. The whole point is to learn from unlabeled data. If the data is labeled, then we enter the world of supervised learning; that will be searching for patterns and learning them. At this point, you can easily see you are at sea in an adventure—a memoryless, random, and unlabeled world for you to discover.

10. **Can Q-learning run without a reward matrix? (Yes | No)**

    The answer is **no**. A smart developer could always find a way around this, of course. The system requires a starting point. You will see in the second chapter that it is quite a task to find the right reward matrix in real-life projects.

Chapter 2 – Building a Reward Matrix – Designing Your Datasets

1. **Raw data can be the input to a neuron and transformed with weights. (Yes | No)**

   The answer is **yes** if the data is in numerical format. If it is in a proper numerical format, the input can be multiplied by the weights and biases.

   If the data is not in a numerical format, then it requires a numerical encoding phase.

2. **Does a McCulloch-Pitts neuron require a threshold? (Yes | No)**

   The answer is **yes**. Adding up weights does not mean much if you do not have something to measure the value. It took months of work for McCulloch and Pitt to put this together. At first, time was in the equation, just like it is in our brain. But then, like Joseph Fourier (1768-1830), they found cycles that repeated themselves—periods that did not require much more than that neuron.

   Warren McCulloch and Walter Pitts invented the first neuron and published a paper in 1943. Legend has it that at age 12 years in 1935, Walter Pitts, a poor child living in a bad neighborhood, was chased by bullies and sought refuge in a library. There, he discovered *Principia Mathematica*, by *Bertrand Russell* and *Alfred Whitehead*. Anyway, not only did he find mistakes in the reasoning, but he also sent a letter to Bertrand Russell! From then on, Walter was noted for his genius in mathematics. With Warren McCulloch, another genius, they invented the first neuron. It seems simple. But it's the result of a number of sleepless nights. Just as the invention of the wheel appears simple, nothing better has been found to this day. This concept of a neuron is the wheel of AI.

3. **A logistic sigmoid activation function makes the sum of the weights larger. (Yes | No)**

The answer is **no**. The whole point of a sigmoid function is to reduce the sums when necessary to have comparable numbers to work with.

4. **A McCulloch-Pitts neuron sums the weights of its inputs. (Yes | No)**

   The answer is **yes**. It's only when you sum the weights that they make sense.

5. **A logistic sigmoid function is a log10 operation. (Yes | No)**

   The answer is **no**. The sigmoid function is based on Euler's number, *e*, a constant that is equal to 2.71828. This number produces a natural logarithm. Leonhard Euler (1707-1783) discovered this in the 18th century with a quill —no scientific calculator or computer! Did you notice that the main mathematical functions used in AI run far back in history? This aspect of the hype around what we think we have found now but has existed for decades, and sometimes centuries, will be dealt with in the following chapters.

6. **A logistic softmax is not necessary if a logistic sigmoid function is applied to a vector. (Yes | No)**

   The answer is **no**. Calculating the sum of several numbers of a vector and then dividing each number by that sum gives a view of the proportions involved. It is a precious tool to keep in mind.

7. **A probability is a value between –1 and 1. (Yes | No)**

   The answer is **no**. Probabilities lie between 0 and 1.

Chapter 3 – Machine Intelligence – Evaluation Functions and Numerical Convergence

1. **Can a human beat a chess engine? (Yes | No)**

   The answer is **no**. Today, the highest-level chess tournaments are not between humans but between chess engines. Each chess engine software editor prepares for these competitions by making their algorithms faster and requiring less CPU. Today, a top chess engine running on a smartphone can beat humans. In human-to-human chess competitions, the level of chess has reached very high limits of complexity. Humans now mostly train against machines.

2. **Humans can estimate decisions better than machines with intuition when it comes to large volumes of data. (Yes | No)**

   The answer is **no**. The sheer CPU power of an average machine or even a smartphone can generate better results than humans with the proper algorithms.

3. **Building a reinforcement learning program with a Q function is a feat in itself. Using the results afterward is useless. (Yes | No)**

   The answer is **no**. While learning AI, just verifying that the results are correct is enough. In real-life applications, the results are used in databases or as input to other systems.

4. **Supervised learning decision tree functions can be used to verify that the result of the unsupervised learning process will produce reliable, predictable results. (Yes | No)**

   The answer is **yes**. Decision tree functions are very efficient in many cases. When large volumes are involved, decision tree functions can be used to

analyze the results of the machine learning process and contribute to a prediction process.

5. **The results of a reinforcement learning program can be used as input to a scheduling system by providing priorities. (Yes | No)**

   The answer is **yes**. The output of reinforcement learning Q functions can be injected as input into another Q function. Several results can be consolidated in phase 1 and become the reward matrix of a phase 2 reinforcement learning session.

6. **Can artificial intelligence software think like humans? (Yes | No)**

   The answer is **yes**, and **no**. In the early days, this was attempted with neuroscience-based models. However, applying mathematical models is presently far more efficient.

   Who knows what will happen in future research with neuromorphic computing, for example? But for the time being, deep learning, the main trend, is based on mathematical functions.

Chapter 4 – Optimizing Your Solutions with K-Means Clustering

1. **Can a prototype be built with random data in corporate environments? (Yes | No)**

   The answer is **yes**, and **no**. To start developing a prototype, using random data can help make sure that the basic algorithm works as planned.

   However, once the prototype is advanced, it will be more reliable to use a well-designed dataset. Then, once the training has been accomplished, random data can again help to see how your system behaves in all situations.

2. **Do design matrices contain one example per matrix? (Yes | No)**

   The answer is **no**. A good design matrix contains one example in each row or each column depending on the shape you want it to have. But be careful; a design matrix that contains data that is too efficient might *overfit*. That means the learning algorithm will be efficient with that data but not adapt to new data. On the other hand, if the dataset contains too many errors, then the algorithm might *underfit*, meaning it won't learn correctly. A good design matrix should contain reliable data, some imprecise data, and some *noise* (some data that can influence the algorithm in unreliable ways).

3. **Automated guided vehicles (AGVs) can never be widespread. (Yes | No)**

   The answer is **no**. The sentence is not a correct assertion. AGVs will expand endlessly from now on: drones, cars, planes, warehouse vehicles, industrial vehicles, and more. AGVs, added to AI and IoT, constitute the fourth industrial revolution.

4. **Can k-means clustering be applied to drone traffic? (Yes | No)**

The answer is **yes**. Seeing where traffic builds up will prevent drone jams (drones circling and waiting).

5. **Can k-means clustering be applied to forecasting? (Yes | No)**

   The answer is **yes**. K-means clustering can be used for predictions.

6. **Lloyd's algorithm is a two-step approach. (Yes | No)**

   Yes, Lloyd's algorithm first classifies each data point in the best cluster. Then, once that is done, it calculates the geometric center or centroid of that center. When no data point changes the cluster anymore, the algorithm has been trained.

7. **Do hyperparameters control the behavior of the algorithm? (Yes | No)**

   The answer is **yes**. Hyperparameters determine the course of the computation: the number of clusters, features, batch sizes, and more.

8. **Once a program works, the way it is presented does not matter. (Yes | No)**

   The answer is **no**. Without a clear presentation of the results, the whole training process is confusing at best and useless at worst.

9. **K-means clustering is only a classification algorithm. It's not a prediction algorithm. (Yes | No)**

   The answer is **no**. K-means clustering can be used as a prediction algorithm as well.

Chapter 5 – How to Use Decision Trees to Enhance K-Means Clustering

The questions will focus on the hyperparameters.

1. **The number of k clusters is not that important. (Yes | No)**

   The answer is **no**. The number of clusters requires careful selection, possibly a trial-and-error approach. Each project will lead to different clusters.

2. **Mini-batches and batches contain the same amount of data. (Yes | No)**

   The answer is **no**. "Batch" generally refers to the dataset, and "mini-batch" represents a "subset" of data.

3. **K-means can run without mini-batches. (Yes | No)**

   The answer is **yes**, and **no**. If the volume of data remains small, then the training epochs can run on the whole dataset. If the data volume exceeds a reasonable amount of computer power (CPU or GPU), mini-batches must be created to optimize training computation.

4. **Must centroids be optimized for result acceptance? (Yes | No)**

   The answer is **yes**, and **no**. Suppose you want to put a key in a keyhole. The keyhole represents the centroid of your visual cluster. You must be precise. If you are simply throwing a piece of paper in your garbage can, you do not need to aim at the perfect center (centroid) of the cluster (marked by the rim of the garbage can) to attain that goal. Centroid precision depends on what is asked of the algorithm.

5. **It does not take long to optimize hyperparameters. (Yes | No)**

The answer is **yes**, and **no**. If it's a simple project, it will not take long. If you are facing a large dataset, it will take some time to find the optimal hyperparameters.

6. **It sometimes takes weeks to train a large dataset. (Yes | No)**

   The answer is **yes**. Media hype and hard work are two different worlds. Machine learning and deep learning are still tough projects to implement.

7. **Decision trees and random forests are unsupervised algorithms. (Yes | No)**

   The answer is **yes**, and **no**. Decision trees can both be used for supervised or unsupervised learning. Decision trees can start with a target value, which makes them supervised. Random forests can be used in the same way.

Chapter 6 – Innovating AI with Google Translate

1. **Is it better to wait until you have a top-quality product before putting it on the market? (Yes | No)**

   The answer is **yes**, and **no**. Context is everything. In the early 21st century, Airbus was struggling to complete the A380, the largest ever passenger airplane. Their engineers worked on hundreds of improvements before transporting commercial passengers. We would expect no less!

   In the case of Google Translate, it is a massive no. By putting Google Translate online and providing an API, Google encouraged thousands of AI developers, linguists, consultants, and users to provide feedback and improvements. Furthermore, Google, once again, occupies a large share of the web market.

2. **Considering the investment made, a new product should always be priced high to reach the top segment of the market. (Yes | No)**

   The answer is **yes** and **no**. Again, context determines the answer. When Ferrari puts a new car on the market, the price has to be high for two reasons; the quality of the car and the cost of production make it necessary to do so to make the innovation profitable. Also, Ferrari avoids mass production to keep its quality at high levels.

   When Amazon Web Services put machine learning on the market with SageMaker, it put a "pay-as-you-go" policy in place, starting at a very low end of the market. The product had, and has, its limits, but Amazon now receives tremendous volumes of feedback and is continuously improving the product.

3. **Inventing a new solution will make it known in itself. (Yes | No)**

   Yes. An invention society is ready to accept will make its way on its own no

matter what. You might be surprised to know that saving a camera's projected image by drawing it dates so far back in history that nobody knows for sure when it was first used. Nobody knows if it was invented or discovered. In any case, the first *camera obscura* (the first "cameras") was revolutionary. It is now proven that famous painters used the technique. The picture was projected on a paper or canvas. The "printer" was manual. The painter was the "printer." However, cameras, as we know, only became disruptive in the 20th century.

4. **AI can solve most problems without using standard non-learning algorithms. (Yes | No)**

   The answer is **no**. Non-learning classical algorithms are necessary to create datasets, for example. Furthermore, AI relies on cloud servers, architectures, standard algorithms in all languages (C++, Java, Python, and others), and Apache servers. Even on a self-driving car, the sensors installed require standard hard work to get them working and interpreting information before AI comes in to solve some of the problems.

   AI is like our brain. Without a body, it cannot function.

5. **Google Translate can satisfactorily translate all languages. (Yes | No)**

   After reading this chapter, you might be surprised to have a **yes** and **no** answer. If you are using Google Translate to say "hello," "how are you?", "thanks for the message," and similar friendly phrases on your favorite social network or in an email, it is good enough.

   But when dealing with more detailed phrases and sentences, Google Translate provides random satisfactory results. From a user's perspective, this is bad news. For a developer, it is a goldmine!

6. **If you are not creative, it is no use trying to innovate. (Yes | No)**

   The answer is a massive **no**. You certainly do not need to be either imaginative or creative to innovate. Do not let anybody convince you of such nonsense. If you are designing a solution and find a missing component, look for some alternative components on the web, talk about it, and find people that can help. Then get it done through teamwork. This works every time!

Even the great Bill Gates was smart enough to ask Tim Patterson for help to develop MS-DOS, and he went on to become a billionaire.

7. **If you are not a linguist, it is no use bothering with trying to improve Google Translate. (Yes | No)**

   The answer is **no**! Once again, never let somebody convince you of such nonsense. Innovating is teamwork. If you like Google Translate and you understand this chapter and have ideas, team up with a linguist around you or through a social network. The world is yours to improve!

8. **Translating is too complicated to understand. (Yes | No)**

   No. The way some explain it is too complicated. If you speak a language, you are an expert in translating your thoughts into words. With work, you can get into the translation business.

9. **AI has already reached its limits. (Yes | No)**

   Certainly **not**! We have just scratched the surface of both theory and applications.

Chapter 7 – Optimizing Blockchains with Naive Bayes

1. **Cryptocurrency is the only use of blockchains today. (Yes | No)**

   No. IBM HyperLedger, for example, uses blockchains to organize secure transactions in a supply chain environment.

2. **Mining blockchains can be lucrative. (Yes | No)**

   Yes. But it is a risk, like any other mining operation or any speculative endeavor. Some companies have huge resources to mine cryptocurrency, meaning that they can beat smaller competitors in creating a block.

3. **Blockchains for companies cannot be applied to sales. (Yes | No)**

   No. Blockchain cloud platforms provide smart contracts and a secure way of managing transactions during a sales process.

4. **Smart contracts for blockchains are more accessible to write than standard offline contracts. (Yes | No)**

   Yes, if they are standard contracts, this speeds the transaction up.

   On the other hand, **no**. If the transaction is complex and requires customization, a lawyer will have to write the contract, which can then only be used on a blockchain cloud platform.

5. **Once a block is in a blockchain network, everyone in the network can read the content. (Yes | No)**

   Yes, if no privacy rule has been enforced.

   No, if a privacy rule has been enforced. IBM Hyperledger, for example, provides privacy functions.

6. **A block in a blockchain guarantees that absolutely no fraud is possible. (Yes | No)**

   Yes and **no**. A block in a blockchain can never be changed again, thereby avoiding fraud. Nobody can tamper with the data. However, if the transaction is illegal in the first place, then the block will be fraudulent as well.

7. **There is only one way of applying Bayes' theorem. (Yes | No)**

   No. There are many variations of Bayes' theorem. Using naive Bayes, for example, avoids the conditional probability constraint. But another approach could use conditional probability.

8. **Training a naive Bayes dataset requires a standard function. (Yes | No)**

   No. Gaussian functions, for example, can be used to calculate naive Bayes algorithms, among others.

9. **Machine learning algorithms will not change the intrinsic nature of the corporate business. (Yes | No)**

   No. Well-designed machine learning will disrupt every area of business as algorithms spread through the company, optimizing processes.

Chapter 8 – Solving the XOR Problem with a Feedforward Neural Network

1. **Can the perceptron alone solve the XOR problem? (Yes | No)**

   Yes. The answer would have been no in 1969. A neural network, or some other mathematical process, is necessary to solve this problem. For the record, this is a common problem for electric circuits that function with "feedforward" electricity, and was solved long ago.

2. **Is the XOR function linearly non-separable? (Yes | No)**

   The answer is **no** if you use a single neuron, and **yes** if you use a hidden layer with at least two neurons. That is a major problem to address in deep learning. If you cannot separate the features of a face, for example, in a picture, recognizing that face will prove difficult. Imagine a picture with one half of the face in shadow and the other half in bright sunlight. Since the eye and features of one half are in shadow, a poor deep learning program might only capture half of the face, separating the face in the wrong place with a poor edge detection function. Linear separability is thus a key aspect of machine learning.

3. **One of the main goals of layers in a neural network is classification. (Yes | No)**

   The answer is **yes**. Once the data is identifiable with a given neural network architecture, predictions and many other functions become possible.
   The key to deep learning is to be able to transform data into pieces of information that will make sense through the abstraction obtained over the layers.

4. **Is deep learning the only way to classify data? (Yes | No)**

The answer is **no**. You can classify data with a SQL query, spreadsheets, AI, machine learning, and standard source code. Deep learning becomes vital when many dimensions of classification are involved: first finding the edges of objects in a picture, then forms, and then determining what the object represents. To do this with millions of pictures is beyond the scope of standard programming or early AI and machine learning programs.

5. **A cost function shows the increase in the cost of a neural network. (Yes | No)**

   The answer is **no**. A cost function determines how much the training costs you. Running 100,000 epochs is more expensive than running 50,000 epochs. So, at each epoch, the cost of training (how far the system is from its goal) must be estimated. Thus, a good cost function will decrease the cost of running a neural network.

6. **Can simple arithmetic be enough to optimize a cost function? (Yes | No)**

   The answer is **yes**. As long as you know to what extent your cost function is increasing or decreasing, anything that works is fine.

7. **A feedforward network requires inputs, layers, and an output. (Yes | No)**

   The answer is **yes**. Without layers, there is no network.

8. **A feedforward network always requires training with backpropagation. (Yes | No)**

   The answer is often **yes** in changing environments. Since the field is new, we tend to think that once the training is done, the work is done. If the datasets are very stable in a repetitive environment, such as recognizing the difference between various constant products in a shop, warehouse, or factory, then the neural network will do the classification it is designed for. If new products are introduced, then training can be initiated again.

9. **In real-life applications, solutions are only found by following existing theories. (Yes | No)**

   The answer is **no**. Without academic research, deep learning would not

even exist. Without universities, the ideas used would be so simple that they would never work well.

On the other hand, researchers need real-life feedback. If we find new ways of doing things they recommend, we should publish them to help global research. It's a two-way street.

Chapter 9 – Abstract Image Classification with Convolutional Neural Networks (CNNs)

1. **A convolutional neural network (CNN) can only process images. (Yes | No)**

   The answer is **no**. CNNs can process words, sounds, or video sequences, to classify and predict.

2. **A kernel is a preset matrix used for convolutions. (Yes | No)**

   The answer is **yes**, and **no**. There are many preset matrices used to process images, such as the one used in `Edge_detection_Kernel.py` in this chapter. However, in this chapter, kernels were created randomly, and then the network trained their weights to fit the target images.

3. **Does pooling have a pooling matrix, or is it random?**

   In some cases, a pooling matrix has a size that is an option when the pooling layer is added to the model, such as a 2×2 pooling window. However, in AutoML neural networks, for example, we can try to run optimizing algorithms that will test various sizes to see which one produces the best performance.

4. **The size of the dataset always has to be large. (Yes | No)**

   No. A dataset does not have a standard size. It depends on the training model. If the target images, for example, do not contain complex features, the dataset will be smaller than a complex feature dataset. Furthermore, the `ImageDataGenerator` function will expand the data by distorting it with the options provided.

5. **Finding a dataset is not a problem with all the available image banks**

on the web. (Yes | No)

The answer is **yes**, and **no**. Yes, because if the model remains a standard academic one, then the available images (CIFAR, MNIST, or others) will suffice.

No, because in real-life corporate situations, you will have to build your dataset and add images containing *noise*. Noise requires more fine-tuning of the model to become reliable and generalized.

6. **Once a CNN is built, training it does not take much time. (Yes | No)**

   The answer is **no**. Whatever the model is, training will remain time-consuming if you want it to be reliable. As seen in this chapter, a model requires a lot of options and mathematical thinking.

7. **A trained CNN model applies to only one type of image. (Yes | No)**

   Yes and **no**. There are three main types of overfitting:

   - Overfitting a model for a certain type of image with absolutely no consequence of implementation. In this case, the model classifies and predicts enough to satisfy the goals set. Suppose you are using a type of image with a very high definition. The CNN will detect the small details we may be trying to detect. However, when the application is in production, lower-quality images might make the model fail to identify what it had accurately detected with high-quality images.
   - Overfitting a model that creates implementation problems because it cannot adapt to different images at the same time. The model will then go through more training.
   - Overfitting a model that trains a certain type of image quite well but does not fit similar types of images when needed.

   Each situation has its constraints. As long as the model works, no general rules apply. It is up to you to decide.

8. **A quadratic loss function is not very efficient compared to a cross-entropy function. (Yes | No)**

The answer is **no**. Each model has its constraints. Quadratic loss functions work fine on some models and do not provide good results on others. This represents the main problems of training a model. No general rules will help you. You have to use your neurons or write a program that modifies the model automatically.

9. **The performance of a deep learning CNN does not present a real issue with modern CPUs and GPUs. (Yes | No)**

   The answer is **yes**, and **no**. If the model runs quickly enough for your needs, then performance will not limit the outcome of your project. However, in many cases, it remains a problem. Reducing features to focus on the best ones is one of the reasons that the layers bring the size to analyze down, layer by layer.

Chapter 10 – Conceptual Representation Learning

1. **The curse of dimensionality leads to reducing dimensions and features in machine learning algorithms. (Yes | No)**

   Yes. The volume of data and features makes it necessary to extract the main features of an observed event (an image, sound, and words) to make sense of it.

   Overfitting and underfitting apply to dimensionality reduction as well. Reducing the features until the system works in a lab (overfitting) might lead to nowhere once the application faces real-life data. Trying to use all the features might lead to underfitting because the application solves no problem at all.

   Regularization applies not just to data but to every aspect of a project.

2. **Transfer learning determines the profitability of a project. (Yes | No)**

   Yes, if an application of an AI model in itself was unprofitable the first time, but could generate profit if used for a similar type of learning. Reusing some functions would generate profit, no doubt.

   No, if the first application was extremely profitable but "overfitted" to meet the specifications of a given project.

3. **Reading model.h5 does not provide much information. (Yes | No)**

   No. No in this case means that the assertion of the sentence is wrong. The saved model does provide useful information. Saving the weights of a TensorFlow model, for example, is vital during the training process to control the values. Furthermore, trained models often use HDF files (`.h5`) to load the trained weights. A Hierarchical Data Format (HDF) contains multidimensional arrays of scientific data.

4. **Numbers without meaning are enough to replace humans. (Yes | No)**

   Yes, in the sense that in many cases, mathematics provides enough tools to replace humans for many tasks (games, optimization algorithms, and image recognition).

   No, in cases where mathematics cannot solve problems that require concepts, such as many aspects of NLP.

5. **Chatbots prove that body language doesn't mean that much. (Yes | No)**

   Yes, in the sense that in many applications, body language does not provide additional information. If only a yes or no answer is required, body language will not add much to the conversation.

   No, in the sense that if emotional intelligence is required to understand the tone of the user, a webcam detecting body language could provide useful information.

6. **Present-day artificial neural networks (ANNs) provide enough theory to solve all AI requests. (Yes | No)**

   No. Artificial neural networks (ANNs) cannot solve thousands of problems, for example, translating poetry novels or recognizing images with forms that constantly vary.

7. **Chatbots can now replace humans in all situations. (Yes | No)**

   No. Concepts need to be added. The market provides all the necessary tools. It will take some years to be able to speak effectively with chatbots.

8. **Self-driving cars have been approved and do not need conceptual training. (Yes | No)**

   Yes and **no**; there is some debate in this area. On the one hand, sensors and mathematics (linear algebra, probabilities) might succeed in effectively navigating roads in most driving conditions within a few years.

   However, certain problems will require concepts (and more robotics) in critical situations. If a self-driving car encounters a wounded person lying

on the road, what is the best approach? The choices are to call for help, find another person if the help arrives too late, pick up the victim, drive them to a hospital (robotics), and much more.

9. **Industries can implement AI algorithms for all of their needs. (Yes | No)**

   Yes and **no**. In many cases, all of the necessary tools are there to be used. If the right team decides to solve a problem with AI and robotics, it can be done.

   However, some tools are missing when it comes to addressing particular circumstances; for example, real-time management decision tools when faced with unplanned events. If a system breaks down, humans can still adapt faster to find alternative solutions to continue production.

Chapter 11 – Combining Reinforcement Learning and Deep Learning

1. **A CNN can be trained to understand an abstract concept? (Yes | No)**

   Yes. A CNN can classify images and make predictions. But CNNs can analyze any type of object or representation. An image, for example, can be linked to a word or phrase. The image thus becomes a message in itself.

2. **Is it better to avoid concepts and only use real-life images? (Yes | No)**

   No. Images provide many practical applications, but at some point, more is required to solve planning problems, for example.

   Planning requires much more than this type of dataset.

3. **Do planning and scheduling mean the same thing? (Yes | No)**

   No. Planning describes the tasks that must be carried out. Scheduling adds a time factor. Planning tells us what to do, and scheduling tells us when.

4. **Is Amazon's manufacturing patent a revolution? (Yes | No)**

   No. Manufacturing clothing has been mastered by factories around the world. It would be a revolution if something in the process was entirely new. However, automation in the apparel industry has been around for 20+ years. The patent is a specialized case of elements that exist, like a new brand of car.

   Yes. With such a worldwide distribution, Amazon has come very close to the end user. The end user can choose a new garment, and it will be manufactured directly on demand. This connectivity will change the apparel manufacturing processes and force its competitors to find new

ways of making and selling garments.

5. **Learning how warehouses function is not useful. (Yes | No)**

   No. Online shopping requires more and more warehouse space and processes. The number of warehouses will now increase faster than shops. There are many opportunities for AI applications in warehouses.

6. **Online marketing does not require AI. (Yes | No)**

   No. On the contrary, AI is used by applications for online marketing every day, and this will continue for decades.

Chapter 12 – AI and the Internet of Things

1. **Driving quickly to a location is better than safety in any situation. (Yes | No)**

   Yes and **no**.

   Self-driving cars face the same difficulties as human-driven cars: getting to a location on time, respecting speed limits, or driving as safely as possible. Self-driving cars, like humans, are constantly improving their driving abilities through experience.

   Yes. Sometimes, a self-driving car will perform better on a highway with little traffic.

   No. Sometimes, if the highways are dangerous (owing to weather conditions and heavy traffic), a self-driving car should take a safer road defined by slow speed and little to no traffic. This way, if difficulties occur, the self-driving car can slow down and even stop more easily than on a highway.

2. **Self-driving cars will never really replace human drivers. (Yes | No)**

   Nobody can answer that question. As self-driving cars build their abilities and experience, they might well end up driving better than humans.

   In very unpredictable situations, humans can go off the road to avoid another car and back off a bit, for example. It will take more work to get a self-driving car to do that.

   One thing is certain, though. If a human is driving all night and falls asleep, the self-driving car will detect the head slumping movement, take over, and save lives. The self-driving car can also save lives if the human has a medical problem while driving.

3. **Will a self-driving fire truck with robots be able to put out a fire one day? (Yes | No)**

   Yes. Combining self-driving fire trucks with robots will certainly save many lives when a fire department faces difficult fires to extinguish. Those saved lives include firefighters who risk their own lives. It might help firefighters focus on helping people while robots do tougher jobs. This robot-human team will no doubt save thousands of lives in the future.

4. **Do major cities need to invest in self-driving cars or avoid them? (Invest | Avoid)**

   Invest. With slow but safe self-driving cars, commuters could share public, free, or very cheap electric self-driving cars instead of having to drive. It would be like having a personal chauffeur.

5. **Would you trust a self-driving bus to take children to school and back? (Yes | No)**

   This answer will change with time, as technology continues to evolve.

   No. Not in the present state of self-driving vehicles. You should not fully trust an autonomous vehicle 100%!

   Yes, when self-driving cars, buses, and trucks prove that they can outperform humans. Self-driving vehicles will not make the same mistakes as humans: using smartphones while driving, talking to passengers without looking at the road, and many others.

6. **Would you be able to sleep in a self-driving car on a highway? (Yes | No)**

   No, not in the present state of self-driving vehicle technology.

   Yes, when reliability replaces doubts.

7. **Would you like to develop a self-driving program for a project for a city? (Yes | No)**

   That one is for you to think about! You can also apply the technology to

warehouses for AGVs by contacting the companies or AGV manufacturers directly.

Chapter 13 – Visualizing Networks with TensorFlow 2.x and TensorBoard

1. **A CNN always has the same number of layers. (Yes | No)**

   No. A CNN does not have the same number of layers or even the same type of layers. The number of layers is part of the work to optimize an artificial neural network.

2. **ReLU is the best activation function. (Yes | No)**

   No. ReLU is an efficient activation function, but there are others such as leaky ReLU, softmax, sigmoid, and tanh.

3. **It is not necessary to compile a sequential classifier. (Yes | No)**

   No. The assertion should be yes – it is necessary.

4. **The output of a layer is best viewed without running a prediction. (Yes | No)**

   No. The output of a layer and a prediction are unrelated. The output of the layer can be the transformation of a layer (convolutional, pooling, dropout, flattening, other) or a prediction.

5. **The names of the layers mean nothing when viewing their outputs. (Yes | No)**

   No. The layers mean everything! A pooling layer and a dropout layer are quite different.

6. **TensorFlow 2.x does not include Keras. (Yes | No)**

No. TensorFlow has now integrated Keras, which helps to build seamless neural networks.

7. **Google Colaboratory is just a repository, like GitHub. (Yes | No)**

   No. Google Colaboratory provides a small but free server to create and run programs online.

8. **Google Colaboratory cannot run notebooks. (Yes | No)**

   No. It can run notebooks.

9. **It is possible to run TensorBoard in Google Colaboratory notebooks (Yes | No)**

   Yes. This is a useful feature.

10. **Accuracy is displayed in TensorBoard (Yes | No)**

    Yes. It is an efficient way to evaluate the efficiency of ANNs, for example.

Chapter 14 – Preparing the Input of Chatbots with Restricted Boltzmann Machines (RBMs) and Principal Component Analysis (PCA)

1. **RBMs are based on directed graphs. (Yes | No)**

   No. RBM graphs are undirected, unsupervised, and memoryless, and the decision-making is based on random calculations.

2. **The hidden units of an RBM are generally connected to one another. (Yes | No)**

   No. The hidden units of an RBM are not generally connected to each other.

3. **Random sampling is not used in an RBM. (Yes | No)**

   No. False. Gibbs random sampling is frequently applied to RBMs.

4. **PCA transforms data into higher dimensions. (Yes | No)**

   Yes. The whole point of PCA is to transform data into a lower number of dimensions in higher abstraction dimensions (key dimensions isolated) to find the principal component (highest eigenvalue of a covariance matrix), then the second highest, down to the lowest values.

5. **In a covariance matrix, the eigenvector shows the direction of the vector representing that matrix, and the eigenvalue shows the size of that vector. (Yes | No)**

   Yes. Eigenvalues indicate how important a feature is, and eigenvectors

provide a direction.

6. **It is impossible to represent a human mind in a machine. (Yes | No)**

   No. It is possible to a certain extent with sensors and in a limited perimeter.

7. **A machine cannot learn concepts, which is why classical applied mathematics is enough to make efficient AI programs for every field. (Yes | No)**

   No. Never believe that. Progress is being made and will never stop until mind machines become mainstream.

Chapter 15 – Setting Up a Cognitive NLP UI/CUI Chatbot

1. **Can a chatbot communicate like a human? (Yes | No)**

   No. Communicating like a human means being human: having a body with body language, sensations, odors, fear hormones, and much more. Chatbots only emulate these behaviors.

2. **Are chatbots necessarily AI programs? (Yes | No)**

   No. Many call centers use the "press 1, press 2 … press *n*" method, which requires careful organization but no AI.

3. **Chatbots only need words to communicate. (Yes | No)**

   The answer is not a clear-cut one.

   Yes. Simple chatbots can communicate with words in a controlled situation.

   No. When polysemy (several meanings for the same word or situation) is involved, pictograms and more add more efficient dimensions.

4. **Do humans only chat with words? (Yes | No)**

   No. Humans express themselves through the tone of their voice, body language, or music, for example.

5. **Humans only think in words and numbers. (Yes | No)**

   No. Certainly not. Humans think in images, sounds, odors, and feelings.

6. **Careful machine learning preparation is necessary to build a cognitive chatbot. (Yes | No)**

   The answer depends on the context.

No. In limited "press 1 or press 2 " situations, chatbots can perform well with limited cognitive capacities.

Yes. To engage in a real conversation with a human, mental images are the key to providing an empathetic exchange.

7. **For a chatbot to function, a dialog flow needs to be planned. (Yes | No)**

   This depends upon what you want your chatbot to do.

   Yes. It will provide better results in a business environment.

   No. If you want the chatbot to talk freely, you need to free it a bit. This still requires planning of the dialog, but it is more flexible.

8. **A chatbot possesses general AI, so no prior development is required. (Yes | No)**

   No. This is presently impossible. Only narrow (specific to one or a few fields) AI exists in real life, contrary to science fiction movies and media hype.

9. **A chatbot translates fine without any function other than a translation API. (Yes | No)**

   No. At this point in the history of translation bots, the translations are not quite reliable without additional customization.

10. **Chatbots can already chat like humans in most cases. (Yes | No)**

    No. Interpreting a language will take quite some more challenging work and contributions.

Chapter 16 – Improving the Emotional Intelligence Deficiencies of Chatbots

1. **When a chatbot fails to provide a correct response, a hotline with actual humans needs to take over the conversation. (Yes | No)**

   This comes down to context and practicality.

   Yes. That is what the best solution would be. To have an interactive chat service kick in.

   No. In many cases, it would be too expensive. A nice support screen could do the job and send an email to the support team.

2. **Small talk serves no purpose in everyday life or with chatbots. It is best to just get to the point. (Yes | No)**

   Again, this is a matter of context.

   Yes. If it's an emergency bot, of course!

   No. If the chatbot is there to perform a tedious administrative function, some degree of small talk will make the system more bearable.

3. **Data logging can be used to improve speech recognition. (Yes | No)**

   Yes. Absolutely. More data means better machine learning training.

4. **The history of a chatbot agent's conversations will contain valuable information. (Yes | No)**

   Yes. Absolutely. More feedback means more machine learning progress.

5. **Present-day technology cannot make use of the data logging of a user's dialogs. (Yes | No)**

   No. We can, of course, parse data logging and extract valuable information.

6. **An RNN uses sequences of data to make predictions. (Yes | No)**

   Yes, it does.

7. **An RNN can generate the dialog flow of a chatbot automatically for all applications. (Yes | No)**

   Yes and **no**. It can, but the quality is sometimes still terrible at this point in the history of automatic dialog flows!

Chapter 17 – Genetic Algorithms in Hybrid Neural Networks

1. **A cell contains 42 chromosomes. (Yes | No)**

   No. There are 46 chromosomes in a cell.

2. **A genetic algorithm is deterministic, not random. (Yes | No)**

   No. A genetic algorithm is random, which makes it more efficient than many deterministic algorithms.

3. **An evolutionary algorithm means that program code evolves. (Yes | No)**

   There is not a single clear-cut answer.

   No. The program runs like any other program.

   Yes. In some ways, when it is used to optimize a neural network in a hybrid neural network, it changes the parameters of the system. Also, it is possible to use a genetic algorithm to add or take layers out of a CNN, for example.

4. **It is best for a child to have the same genes as one of the parents even after many generations. (Yes | No)**

   No. Certainly not! We need diverse genes to remain a fit genetic group.

5. **Diversity makes the gene sets weaker. (Yes | No)**

   No. The greater the diversity, the richer the genetic pool is to adapt and remain fit.

6. **Building a neural network only takes a few lines, and the architecture always works. (Yes | No)**

This depends on what you mean by "work."

Yes. Building a neural network only takes a few lines with TensorFlow 2.x, for example, and it will work.

No. The neural network will work, but it will most probably not be efficient until its architecture and hyperparameters are fine-tuned.

7. **Building a neural network with a genetic algorithm can help optimize the architecture of the layers. (Yes | No)**

   Yes. It is possible to extend a genetic algorithm to layer optimizing. Each layer can be a gene, and then the various alternatives can be run to check their accuracy.

8. **Hybrid neural networks are useless since deep learning will constantly progress. (Yes | No)**

   No. Deep learning will reach an asymptote, as do all systems. At that point, new dimensions can be added to deep learning architecture, such as genetic algorithms.

9. **Would you trust a genetic algorithm to make decisions for you? (Yes | No)**

   Yes, if the system is properly designed. No if you don't trust genetic algorithms!

10. **Would you trust a hybrid neural network to optimize the architecture of your network? (Yes | No)**

    Yes, if the system is properly designed. No, if it is unreliable or biased.

Chapter 18 – Neuromorphic Computing

1. **Neuromorphic computing reproduces our mental activity. (Yes | No)**

   No. That is the point. Neuromorphic begins with sub-symbolic low-level neuronal brain activity stimulations that do not form high-level mental activity yet. Our mental activity already uses symbols and contains representations in the form of words, images, numbers, and all kinds of constructions in general.

   However, we can say YES if we are referring to the output results translated into mental activity. My point is that yes and no answers limit our views of AI.

2. **Neuromorphic computing reproduces our brain activity. (Yes | No)**

   Yes. That is the point! The core concept is to go from brain activity to structures formed by neuron spikes.

3. **Semantic Pointer Architecture (SPA) is a hardware architecture. (Yes | No)**

   No. Semantic pointers are like computer program pointers such as C++ pointers. The difference is that they carry a partial meaning of representation to come, hence the word "semantic."

4. **NEF stands for Neural Engineering Framework. (Yes | No)**

   Yes, this is true.

5. **Loihi is a classical chip. (Yes | No)**

   No. Not at all! Loihi is an Intel neurocomputing chip containing a massive number of neurons. A human brain contains around 100 billion neurons.

Imagine that, soon, you'll have a network of neurocomputing chips (Intel or other) that attain that number. Then imagine what can be done with semantic pointers through neurocomputing.

6. **Reproducing our brain's neural activity cannot solve an equation. (Yes | No)**

   No. Of course, we can use neurocomputing to solve equations through the SPA approach.

7. **An ensemble in Nengo contains algorithms. (Yes | No)**

   No. But the question was tricky! Basically, Nengo uses a non-symbolic approach as discussed at length previously. However, it contains Python tutorials with many algorithms solved through neurocomputing, forming a complete problem-solving package.

8. **Spiking blocks neuronal activity. (Yes | No)**

   No. But again, this is a tricky question. Spiking is reflected in the level of activity in a neuron. So, in that respect, the answer is no. But a spiking neuron can inhibit another neuron, thereby blocking it indirectly.

9. **Firing patterns can be used to analyze brain activity. (Yes | No)**

   Yes. Firing patterns change in time in neurocomputing, providing useful information on how the neurons reach a given state.

10. **Machine learning and deep learning are only metaphors of our brain's activity. (Yes | No)**

    Yes. That is the core problem associated with deep learning. They are high-level representations of our brain's neuronal activity.

Chapter 19 – Quantum Computing

1. **Beyond the hype, no quantum computer exists. (Yes | No)**

   No. False. You can already run a quantum computer on IBM Q's cloud platform: https://www.research.ibm.com/ibm-q/.

   The following screenshot is the result of one of the real quantum computer (IBM) calculations I ran on a quantum score explained in the chapter:

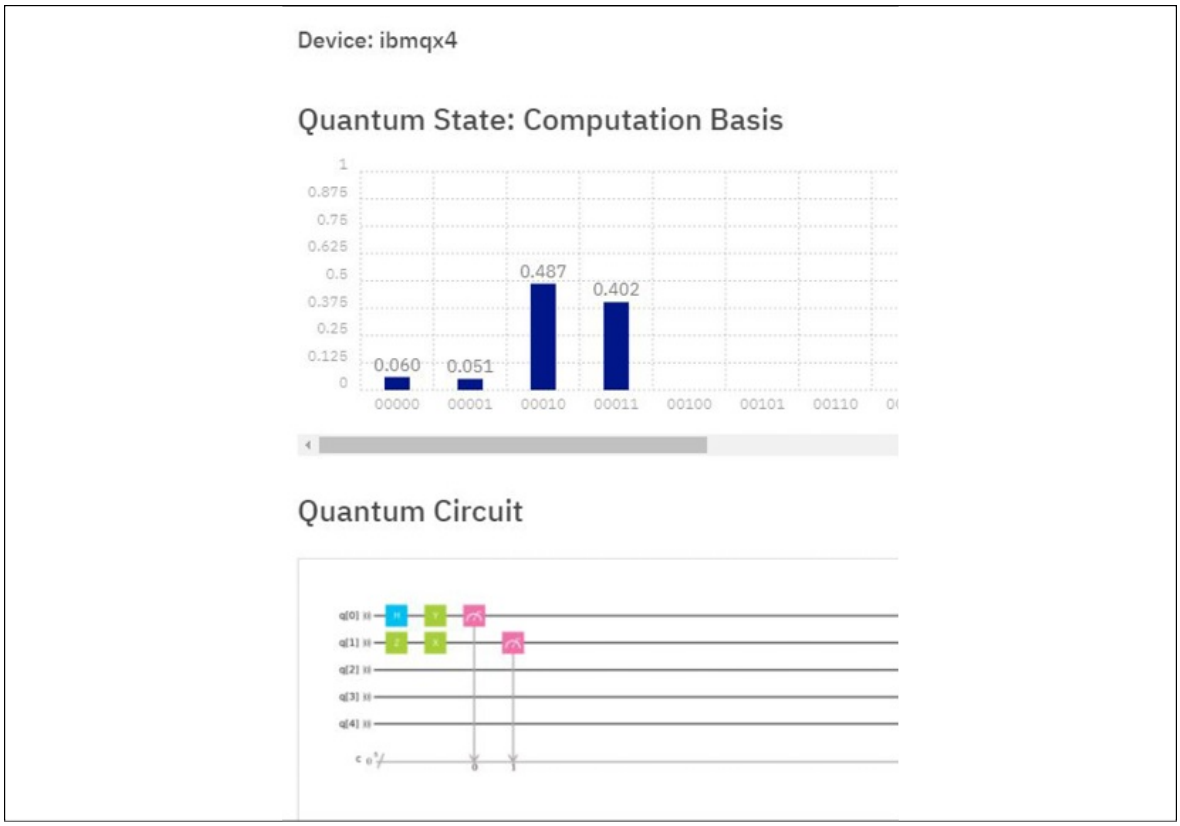


Figure A.1: IBM quantum computer calculation

2. **A quantum computer can store data. (Yes | No)**

   No. Instability prevents any form of storage at this point.

3. **The effect of quantum gates on qubits can be viewed with the Bloch**

sphere. (Yes | No)

Yes. A Bloch sphere will display the state of a qubit.

4. **A mind that thinks with past experiences, images, words, and other bits of everyday information, like stored memories, will find deeper solutions to problems that mathematics alone cannot solve. (Yes | No)**

   There is no single generally accepted answer for this. Just as qubits, the answer is somewhere between yes (1) and no (0)!

   No. False. Many researchers believe that mathematics alone can solve all human problems.

   Yes. True. Mathematics alone cannot replace deep thinking. Even if computers have incredible power and can beat human players at chess, for example, they still cannot adapt to new situations without going through a design and training process. Concepts need to be added and experienced (memory as well).

   I bet that machine mind concepts will become progressively more mainstream to solve deep thinking problems.

5. **A quantum computer will solve medical research problems that cannot be solved today. (Yes | No)**

   Yes. There is no doubt about that. The sheer computing power of a quantum computer can provide exponential DNA sequencing programs for epigenetic research.

6. **A quantum computer can solve mathematical problems exponentially faster than classical computers. (Yes | No)**

   Yes. Classical computers function at $2 \times n$ (number of bits), and quantum computers run at 2^n (n being the number of qubits)!

7. **Classical computers will soon disappear and smartphone processors too. (Yes | No)**

   No. Quantum computers require such a large amount of space and physical

stability that this will not happen soon. Furthermore, classical computers and smartphones can store data; quantum computers cannot.

8. **A quantum score cannot be written in source code format but only with a visual interface. (Yes | No)**

   No. False. IBM, for example, can swap the quantum from score to a QASM interface or display both, as shown here:

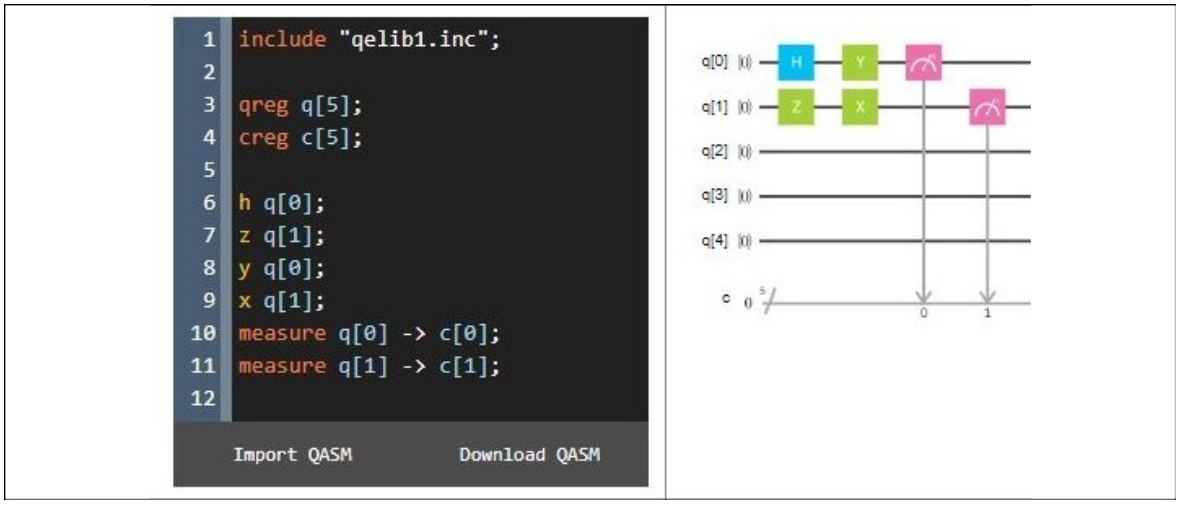


Figure A.2: QASM interface

9. **Quantum simulators can run as fast as quantum computers. (Yes | No)**

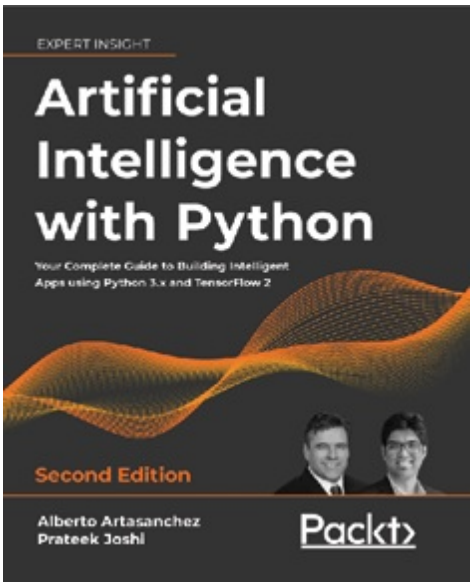
   No. Certainly not! A simulator shows how a quantum score would behave on a real quantum computer. Although the simulator can help build the score, a quantum computer will run exponentially faster than the simulator.

10. **Quantum computers produce intermediate results when they are running calculations. (Yes | No)**

    No. This is not possible. The qubits are too unstable. Observing them makes the system collapse. However, simulators such as Quirk come in handy. Since they are not real, intermediate results can be displayed to design a quantum score.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

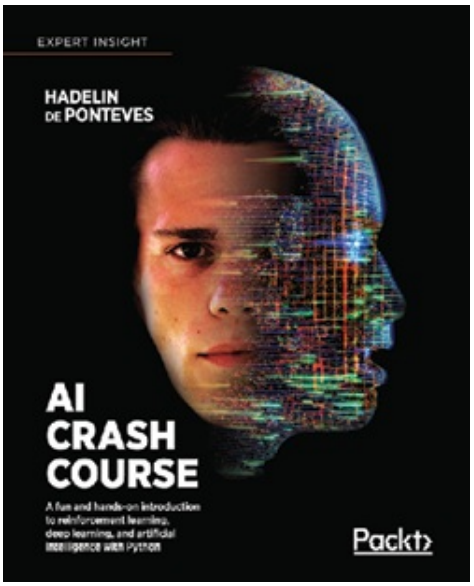


Artificial Intelligence with Python

Alberto Artasanchez, Prateek Joshi

ISBN: 978-1-83921-953-5

- Understand what artificial intelligence, machine learning, and data science are
- Explore the most common artificial intelligence use cases
- Learn how to build a machine learning pipeline
- Assimilate the basics of feature selection and feature engineering
- Identify the differences between supervised and unsupervised learning
- Discover the most recent advances and tools offered for AI development in the cloud
- Develop automatic speech recognition systems and chatbots
- Apply AI algorithms to time series data



AI Crash Course

Hadelin de Ponteves

ISBN: 978-1-83864-535-9

- Master the key skills of deep learning, reinforcement learning, and deep reinforcement learning
- Understand Q-learning and deep Q-learning
- Learn from friendly, plain English explanations and practical activities
- Build fun projects, including a virtual-self-driving car
- Use AI to solve real-world business problems and win classic video games
- Build an intelligent, virtual robot warehouse worker

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please

leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Symbols

2D convolution layer

adding 211

kernel 211, 212

shape 216

A

absolute error 118

activation function 341, 342

activation functions 216

Adam optimizer 226, 227

adaptive moment estimation (Adam) 226

adaptive thinker

becoming 5

affirmation

example 410

agent 5

creating, to learn Dialogflow 378, 379

fulfillment functionality, adding to 393, 394

AI

as frontier 136

disruption 123
emergency, in recent years 125
innovation 123
mathematical theories 124
public awareness 126

algorithm cell 457, 458

apparel conveyor belt processing 272

apparel manufacturing process 268, 269

artificial hybrid neural networks

about 460
LSTM, building
model, goal

artificial neural networks (ANNs)

about 206

automated apparel manufacturing process

CRLMM, applying to 266

Automatic Text to Speech function 400

Auto Speech Adaptation 400

B

backpropagation

used, for implementing vintage XOR solution in Python 190, 191, 192

backtranslation 147, 148, 149

Bellman equation

about 13, 15

mathematical representation, building of 12

binary cross-entropy 225, 226

bitcoin mining 159, 160

blob 306

Bloch sphere 493, 494

block

creating 166

exploring 167

blockchain anticipation novelty 171, 172

blockchain data

used, for optimizing storage levels 172

blockchains

used, for sharing information in supply chain 161, 162, 164, 165

using, in supply chain network 165

blockchain technology

background 158, 159

Boltzmann equation and constant 3

bot

running, on website 398

bot virtual clusters

as solutions

C

cells 436

centroid 87, 96

chatbot

agents 376

machine learning, using in 399

child 440

classifier 211

using, for training

CLT 96

clusters 87

CNN

layers, building of 319, 320, 321, 322

CNN-CRLMM

implementing, to detect gaps 277

CNN model

compiling 224

data, loading 229, 230

goal 223, 224

saving

training 222

cogfilmdr agent

enhancing, with fulfillment webhook 395, 396, 397

conceptual representation learning metamodels

blessing of dimensionality 256

curse of dimensionality 255

motivation 255

conditional probability 168

conditioning management 77, 79

context 139, 388, 389, 390, 391, 392

continuous cycle of k-means clustering

chaining, to decision tree 111, 113, 114, 115

contrastive divergence 354

convergence 11

about 48

implicit convergence 49

numerically controlled gradient descent convergence 49, 50, 52, 53, 54, 55

Conversational User Interfaces (CUI) 387

convolutional 208

convolutional layer

activation functions 328, 329

higher-level representations, through layers 329, 331

convolutional neural network (CNN) 297

convolutional neural networks (CNNs) 124

2D convolution layer, adding 211

about 205, 206

convolution layer 220

defining 208, 209, 210

dense layers 221

flattening layer 221

initializing 210

pooling 219, 220

pooling layer 220

cost function 192, 194, 195

covariance 365

CRLMM

applying, to automated apparel manufacturing process 266

model, applying of 297, 298

parking lots, classifying 302

running 308

trained model, using 301

training 270

crossover 439

cryptocurrency

using 160

curse of dimensionality 124

D

data augmentation

about 229

on testing dataset 230

data logging 416, 417, 419

data points 87

dataset

deriving 367

designing 69

dataset optimization 68

datasets 298, 300, 301

designing 28, 29

decision line 305

decision tree

training 106, 107, 108, 110

decoherence 487

deeper translation

with phrase-based translations 149, 151, 152, 153

degrees 492

dense activation functions 222

dense layer 430

derivative 50

design matrix

approval 71, 72

approval, obtaining on format 72, 74

Dialogflow

learning 378, 379

difficulty of problem

identifying 94

dimensionality reduction 74, 75, 76

disruption

in AI 123

disruptive solutions

versus revolutionary solutions 127

domain learning

about 247

gap concept 250, 252, 253

gap datasets 254

trained model program 249

trained models 249

double-checking function 117

dropout layers

for higher-level representations 333

E

eigenvalues 365, 366

eigenvectors 366

embedding 429

emotional polysemy

problems, solving of 408

emotions

creating 419, 420, 421, 424

energy function 354

Enhanced Speech Models 400

entities

about 379

creating 380

saving 382

episodes

evaluating, of learning sessions 46, 47, 48

epoch accuracy 340

epochs

running 355, 357

error function 354

evolutionary algorithms 434, 437, 438

evolutionary computation 437

F

facial analysis fallacy

about 412

frown 412

smile 412

feature entity

creating 383

features 29, 168

feature vector

creating 366

feedforward neural network (FNN)

about 179

building 185

defining 185

used, for implementing vintage XOR solution in Python 190, 191, 192

fitness 440, 441

fitness cell 459

FNN XOR function

applying, to optimize subsets of data 197, 199, 200, 201, 202, 203

food conveyor belt processing 271

frequentist error probability function

adding 154, 155

fulfillment

defining 394, 395

fulfillment functionality

adding, to agent 393, 394

fulfillment webhook

cogfilmdr agent, enhancing 395, 396, 397

full training dataset

training 98

G

ga_main() cell 459, 460

gamma 15

GA model

creating 438

gap concept 237, 238

gap conceptual dataset 254, 255

gaps

abstract notions 273, 274

Gaussian naive Bayes

implementing 176, 177

gene set of parent 439

gene set of population 439

genetic algorithm, building in Python

about 441
algorithm, calling 443
crossover function 448
display parent 447, 448
fitness 445, 447
generations, producing of children 450, 451, 452, 453
libraries, importing 441
main function 443
mutation function 449, 450
parent generation 444, 445
parent generation process 444
summary code 453

geometric center 87

Gibbs sampling 352

Gini impurity 64

Google Colaboratory

about 334, 335, 336
URL 334

Google's developers'API client library page

reference link 128

Google's translation service

implementing 129, 130

Google Translate 137

customizing, with Python program 138, 139

from linguist's perspective 130

header 128

linguistic assessment 131

program 128

using 128

working with 131

Google Translate customized experiment

conclusions 155

Google_Translate_Customized.py

KNN function, implementing in 145

gradient 50

gradient descent 192, 194, 195

graphics processing unit (GPU) 93

greetings problem

example 409

grouping 197

GRU 429

H

heredity

in humans 434, 436

working 436, 437

H gate 495, 496

hidden units

computing, in training function 351, 352

hub 171

human analytic capacity

evaluating 56, 57, 58, 59

hyperparameters 85

I

IBM Q

quantum computer score 497, 498, 499

implicit convergence 49

inductive abstraction 235

inductive thinking 235

innovation

in AI 123

integration 469

intent

about 383

creating 385, 386

inventions

versus innovations 126

isotropic distribution 306

J

Jargon 132

K

kernel

about 211

developers' approach 213, 214, 215

intuitive approach 212, 213

mathematical approach 215

k-means clustering

mathematical definition 81, 83

unsupervised learning 92, 93

k-means clustering algorithm

limits of implementation

k-means clustering program

about 80, 81

k-means clustering solution

data 77

implementing 76
model, loading 88, 89
model, saving 88, 89
results, analyzing 89
vision 76, 77

k-nearest neighbor algorithm 139

KNN algorithm

implementing 140, 141, 143

KNN function

implementing, in Google_Translate_Customized.py 145

knn_polysemy.py program 143, 144, 145

L

labels 305

layers

building, of CNN 319, 320, 321, 322

leaf 64

lexical field 132, 136

linearly separable model

about 181, 182
disadvantages 182, 183

linear separability 195, 196

linguistic assessment

of Google Translate 131

LLN

using 96

Lloyd's algorithm 84

load keyword 34

logistic activation functions 35

logistic classifier 36, 37

logistic function 37, 39

loss function

about 224

quadratic loss function 224, 225

LSTM 427

M

machine learning

using, in chatbot 399

versus traditional applications 25

machine learning agents

about 399

speech-to-text function 399

spelling correction 402, 403

text-to-speech function 400, 401

machine learning algorithms

significance 404, 405

machine thinking

adapting 5

macrostates 5

mapping environment 10

margin 305

Markov chain 14

Markov decision process (MDP) 1, 124, 297

in natural language 9, 10

mathematical representation, building of 12

Markov property 14

mathematical description 28

mathematical model

building 12

McCulloch-Pitts neuron

about 31, 32, 33, 34

using 29, 30, 31

MDP agent 10, 11

MDP function

standard output 279, 280
target, finding for 286, 287, 288

MDP input 279

MDP output 279, 280

MDP output matrix

graph interpretation 281

MDP result matrix 282, 283

metrics

about 227

microstates 5

MindX

conceptual representations 500

MindX experiment

about 501
data, preparing 501
quantum function 504
situation function 501, 502, 503

miner 160

mini-batches

random sampling, implementing 95

ML algorithms 103

ML/DL model

selecting 69, 71

ML projects

key standard corporate guidelines, avoiding 79, 80

model

applying, of CRLMM 297, 298

defining 336

training 336

Monte Carlo estimator

using 97

mutation 440

N

naive Bayes

example 168, 169, 171

implementing, in Python 176

limits 177

Python program 177

supply chain, optimizing with 168

natural language 28

natural language processing (NLP) 376

natural language understanding (NLU) 376

Nengo

about 463

data, visualizing 471, 472, 473, 475
examples 466
information, retrieving with probes 476, 477, 479
installing 464
neuron dimensions 469
node 469, 470
Python program, creating 467
reference link 464
unique approach, applying to critical AI research areas 480, 481, 482

Nengo ensemble 467, 468

Nengo frontend API definitions

reference link 471

Nengo GUI

installing 464
reference link 464

Nengo neuron types

about 468
reference link 469

Nengo objects

connecting 471
reference link 471

Neural Engineering Framework (NEF)

about 463

neural machine translation (NMT) 145

neural networks 124

neuromorphic computing

about 462, 463

nodes 160

NOT gate 495

NP-hard 94, 95

numerically controlled gradient descent 49, 50, 51, 52, 53, 54, 55

O

optimal production rate (OPR) 236

optimizer

about 282

as regulator 282

original perceptron

about 180

output of layers, CNN

exploring, with TensorFlow 318, 319

overfitting 247

P

parking lot, finding

about 311, 312
itenary graph 314, 315
support vector machine 312, 313
weight vector 315

parking space

finding 308, 309, 310, 311

partition function 354

PCA

about 362, 363
analyzing, to obtain entry point for chatbot 370, 371, 372
representing, with TensorBoard Projector 367, 368, 370
weights of RBM, using as feature vectors 357, 358, 359, 360, 361, 362

phrase-based machine translation (PBMT) 145

physical neural network

about 455
features 456

Pickle 88

plan 264

pointer 463

policy 10, 14

polysemy 137

pooling 219

pooling layer

for higher-level representations 332

poor conditioning 77

population 439

posterior probability 168

postsynaptic currents (PSCs) 468

prediction 88

prediction program

running 275

probabilistic distributions 3

problem

describing, to solve 9, 10

profit

generating, with transfer learning 234, 235

proof of concept (POC) 93

public service project 294

purchase plan

about 260

example 260

Python

naive Bayes, implementing 176

restricted Boltzmann machine (RBM), building in 350

vintage XOR solution, implementing with backpropagation 190, 191, 192

vintage XOR solution, implementing with FNN 190, 191, 192

Python program

about 84
Google Translate, customizing 138, 139
hyperparameters 85
k-means clustering algorithm 86
result labels, defining 86
results, displaying 87
training dataset 85

Python-TensorFlow architecture 35

Q

Q-learning 5

about 278

quadratic loss function 224, 225

quantum computers

about 486
speed 487, 489

quantum computer score

with IBM Q 497, 498, 499
with Quirk 496, 497

quantum gates, with Quirk

about 494

H gate 495, 496
NOT gate 495

quantum score

composing 494
creating 504, 505
output 506
running 505

qubit

defining 490
position 491
representing 490, 491

Quirk

quantum computer score 496, 497
URL 494

R

radians 492

radius 492

random forests 115, 116, 117

randomness 15

random sample

training, of training dataset 98, 100

random sampling

implementing, with mini-batches 95
of hidden units 352

raw data

preprocessing 103

RBM class

creating 350
training function, creating in 350, 351

real-life issues, overcoming with three-step approach

about 6
mathematical model, building 12
problem, describing 9, 10
source code, writing 16, 17, 18

real-time manufacturing process 262

real-time manufacturing revolution 263, 265, 266

reconstruction 353

rectified linear unit (ReLU) 216, 217, 218

recurrent neural network (RNN) 145

regularization 247

reinforcement learning model

use cases 22, 23, 24, 25

reinforcement learning (RL)

about 1, 14

Boltzmann equation and constant 3
Boltzmann, with optimal transport 3
concepts 2
lessons 18, 19
optimal transport 2
outputs, using 20, 21, 22
probabilistic distributions 3

restricted Boltzmann machine (RBM)

about 345
architecture 346, 347
building 345
building, in Python 350
energy-based model 347, 348, 349
structure 350

revolutionary solutions

versus disruptive solutions 127

reward 14

reward matrix 42

RL-DL-CRLMM

building 275, 276
circular model 288, 289, 291, 292
circular process 276
components 275, 276

RL-DL-CRLMM model

setting up 295, 296, 297

RNN

text generation 428

using 426, 427

RNN research

for future automatic dialog generation 425

root-mean-square deviation (RMSprop) 227

rotations 492, 493

S

safety rank 315

scheduling 264

scripts

pipeline 103

selection 439

semantic 464

semantic pointer 463

sequential model 429

services

expanding, to face competition 262

S-FNN

about 456, 457

shuffling 101, 102

small talk

about 413

courtesy 413, 414, 415

emotions 416

softmax function 39, 40, 41

software implementation 28

speech recognition fallacy 410

speech recognition functions, settings

Auto Speech Adaptation 400

Enhanced Speech Models 400

speech-to-text function 399

spelling

about 402, 403

standardization 29

state transition function 12

stochastic process 14

stock keep units (SKUs) 169

storage levels, optimizing with blockchain data

about 172

dataset, defining 172, 173

frequency, calculating 173, 174

likelihood, calculating 174

naive Bayes equation, applying 174, 176

stride 219

subject matter expert (SME) 7

subsets of data

optimizing, by applying FNN XOR function 197, 199, 200, 201, 202, 203

supervised learning

chaining, to verify unsupervised learning 102

used, for evaluating result that surpasses human analytic capacity 60, 62, 63, 64

supply chain

blockchains, using to share information 161, 162, 164, 165

optimizing, with naive Bayes 168

supply chain management (SCM) 171

supply chain network

blockchains, using in 165

support points 305

support vector machine

Python function 306, 307

used, for increasing safety levels 303

support vector machine (SVM)

about 304

example 304

support vectors 305

SVM function

adding 302

T

target 438

TensorBoard

accuracy, analyzing of CNN 334

representation of features 384

running 337

TensorBoard Projector

used, for representing PCA 367, 368, 369, 370

TensorFlow

installing 337

output of layers, exploring of CNN 318, 319

test dataset 88

testing data

loading 230

testing dataset 230

data augmentation 230

text

generating 431

vectorizing 428, 429

text dialog 387

text generation

with RNN 428

text-to-speech

about 400

settings, configuring 401

thinking quantum computer

about 500

time squashing 269

timestep 430

traditional applications

versus machine learning 25

trained TensorFlow 2.x model

applying 246

displaying 239, 240, 241

loading 238

loading, for usage 242, 243, 244

profitable, making 246

strategy, defining 245

training dataset

about 227

random sample, training of 98, 100

volume 76

training function

creating, in RBM class 350, 351

hidden units, computing in 351, 352

training phrases 383

transfer learning

motivation 235

profit, generating with 234, 235

using 245

transition 14

translating 132

translation

checking 135

U

underfitting 247

unit training dataset

generalizing 270

unsupervised learning 14

with k-means clustering 92, 93

unsupervised learning algorithm 85

unsupervised ML algorithm

data, exporting from 106
data, training from 105, 106

V

vanishing gradients 427

variance 364

vertex weights vector 283, 284, 286

vintage XOR solution

implementing, in Python with backpropagation 190, 192
implementing, in Python with FNN 190, 192

visible binary units 351

visual output of CNN layers

analyzing 327
processing 323, 324, 325, 327

W

webhook 394

X

X dataset

trabslating, line by line from English to French 147

XOR function

about 180, 181

XOR problem

solving, examples 186, 187, 189, 190

Index