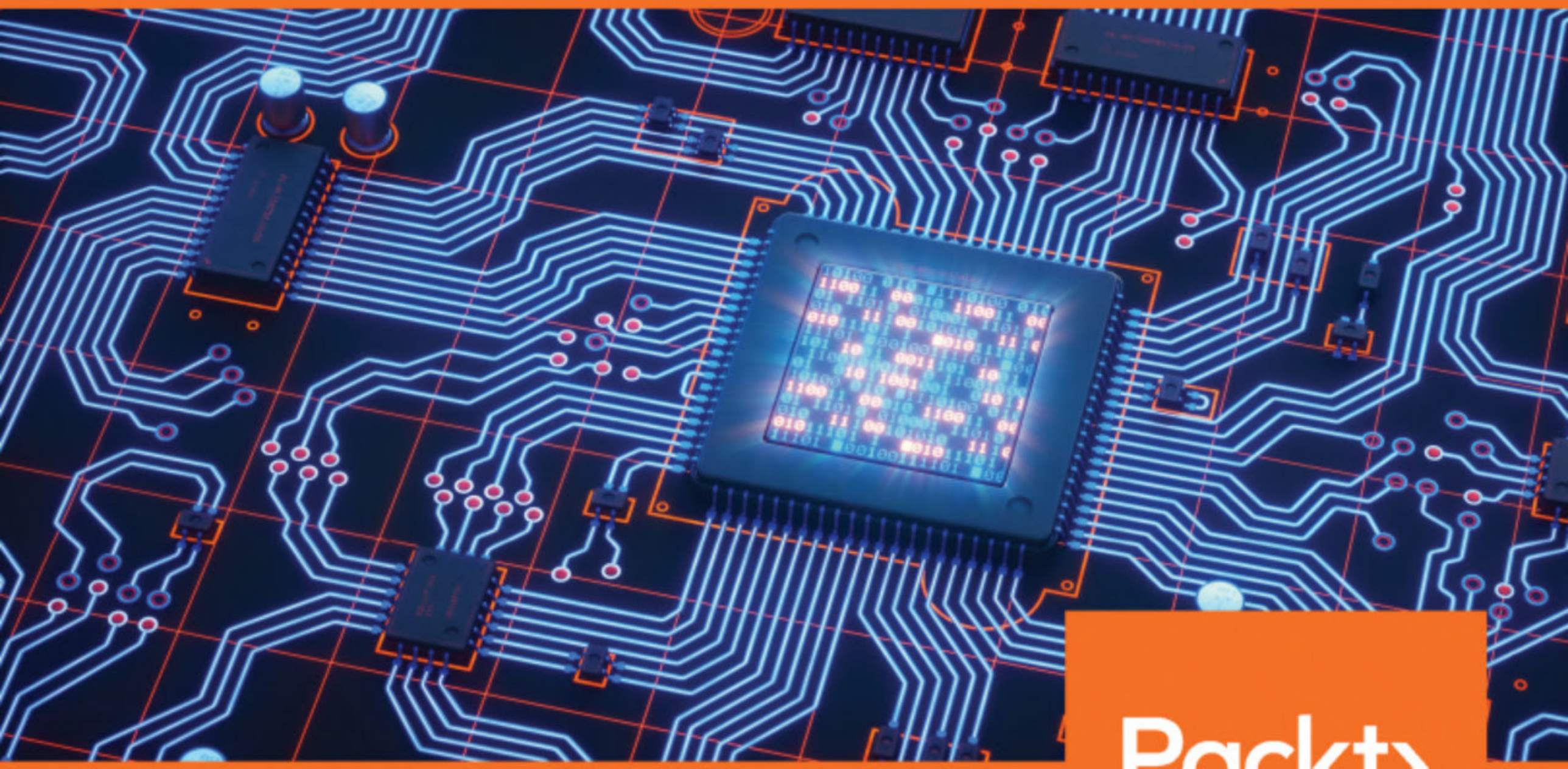
Artificial Intelligence By Example

Develop machine intelligence from scratch using real artificial intelligence use cases



By Denis Rothman

Packt
www.packt.com

Artificial Intelligence By Example

Develop machine intelligence from scratch using real artificial intelligence use cases

Denis Rothman

BIRMINGHAM - MUMBAI

Artificial Intelligence By Example

Commissioning Editor: Pravin Dhandre
Acquisition Editor: Tushar Gupta
Content Development Editor: Mayur Pawanikar
Technical Editor: Prasad Ramesh
Copy Editor: Vikrant Phadkay
Project Coordinator: Nidhi Joshi
Proofreader: Safis Editing
Indexer: Tejal Daruwale Soni
Graphics: Tania Dutta
Production Coordinator: Aparna Bhagat

First published: May 2018

Production reference: 2200618

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78899-054-7

www.packtpub.com

Table of Contents

Preface

This book will take you through all of the main aspects of artificial intelligence:

- The theory of machine learning and deep learning
- Mathematical representations of the main AI algorithms
- Real life case studies
- Tens of opensource Python programs using TensorFlow, TensorBoard, Keras and more
- Cloud AI Platforms: Google, Amazon Web Services, IBM Watson and IBM Q to introduce you to quantum computing
- An Ubuntu VM containing all the opensource programs that you can run in one-click
- Online videos

This book will take you to the cutting edge and beyond with innovations that show how to improve existing solutions to make you a key asset as a consultant, developer, professor or any person involved in artificial intelligence.

Who this book is for

This book contains the main artificial intelligence algorithms on the market today. Each machine learning and deep learning solution is illustrated by a case study and an open source program available on GitHub.

- Project managers and consultants: To understand how to manage AI input datasets, make a solution choice (cloud platform or development), and use the outputs of an AI system.
- Teachers, students, and developers: This book provides an overview of many key AI components, with tens of Python sample programs that run on Windows and Linux. A VM is available as well.
- Anybody who wants to understand how AI systems are built and what they are used for.

What this book covers

`Chapter 1`, *Become an Adaptive Thinker*, covers reinforcement learning through the Bellman equation based on the Markov Decision Process (MDP). A case study describes how to solve a delivery route problem with a human driver and a self-driving vehicle.

`Chapter 2`, *Think like a Machine*, demonstrates 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.

`Chapter 3`, *Apply Machine Thinking to a Human Problem*, 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.

`Chapter 4`, *Become an Unconventional Innovator*, 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 5`, *Manage the Power of Machine Learning and Deep Learning*, uses TensorFlow and TensorBoard to build an FNN and present it in meetings.

`Chapter 6`, *Don't Get Lost in Techniques – Focus on Optimizing Your Solutions*, covers a K-means clustering program with Lloyd's algorithm and how to apply to the optimization of automatic guided vehicles in a warehouse.

`Chapter 7`, *When and How to Use Artificial Intelligence*, shows cloud platform machine learning solutions. We use Amazon Web Services SageMaker to solve a K-means clustering problem. The business case describes how a corporation can analyze phone call durations worldwide.

`Chapter 8`, *Revolutions Designed for Some Corporations and Disruptive Innovations for Small to Large Companies*, explains the difference between a revolutionary innovation and a disruptive innovation. Google Translate will be described and enhanced with an innovative opensource add-on.

`Chapter 9`, *Getting Your Neurons to Work*, describes convolutional neural networks (CNN) in detail: kernels, shapes, activation functions, pooling, flattening, and dense layers. The case study illustrates the use of a CNN in a food processing company.

`Chapter 10`, *Applying Biomimicking to Artificial Intelligence*, describes the difference between neuroscience models and deep learning solutions when representing human thinking. A TensorFlow MNIST classifier is explained component by component and displayed in detail in TensorBoard. We cover images, accuracy, cross-entropy, weights, histograms, and graphs.

`Chapter 11`, *Conceptual Representation Learning*, explains Conceptual Representation Learning (CRL), an innovative way to solve production flows with a CNN transformed into a CRL Meta-model. The case study shows how to use a CRLMM for transfer and domain learning, extending the model to scheduling and self-driving cars.

`Chapter 12`, *Automated Planning and Scheduling*, combines CNNs with MDPs to build a DQN solution for automatic planning and scheduling with an optimizer. The case study is the optimization of the load of sewing stations in an apparel system, such as Amazon's production lines.

`Chapter 13`, *AI and the Internet of Things (IoT)*, covers Support Vector Machines (SVMs) assembled with a CNN. The case study shows how self-driving cars can find an available parking space automatically.

`Chapter 14`, *Optimizing Blockchains with AI*, is about mining blockchains and describes how blockchains function. We use Naive Bayes to optimize the blocks of a Supply Chain Management (SCM) blockchain by predicting transactions to anticipate storage levels.

`Chapter 15`, *Cognitive NLP Chatbots*, shows how to implement IBM Watson's chatbot with intents, entities, and a dialog flow. We add scripts to customize the dialogs, add sentiment analysis to give a human touch to the system, and use conceptual representation learning meta-models (CRLMMs) to enhance the dialogs.

`Chapter 16`, *Improve the Emotional Intelligence Deficiencies of Chatbots*, shows how to turn a chatbot into a machine that has empathy by using a variety of algorithms at the same time to build a complex dialog. We cover Restricted Boltzmann Machines (RBMs), CRLMM, RNN, word to vector (word2Vec) embedding, and principal component analysis (PCA). A Python program illustrates an empathetic dialog between a machine and a user.

`Chapter 17`, *Quantum Computers That Think*, describes how a quantum computer works, with qubits, superposition, and entanglement. We learn how to create a quantum program (score). A case study applies quantum computing to the building of MindX, a thinking machine. The chapter comes with programs and a video.

`Appendix`, *Answers to the Questions*, contains answers to the questions listed at the end of the chapters.

To get the most out of this book

Artificial intelligence projects rely on three factors:

- Subject Matter Experts (SMEs). This implies having a practical view of how solutions can be used, not just developed. Find real-life examples around you to extend the case studies presented in the book.
- Applied mathematics and algorithms. Do not skip the mathematical 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 the book.
- Development and production.

An artificial intelligence solution can be directly used on a cloud platform machine learning site (Google, Amazon, IBM, Microsoft, and others) online or with APIs. In the book, Amazon, IBM, and Google are described. Try to create an account of your own to explore cloud platforms.

Development still remains critical for artificial intelligence 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.

Download the example code files

You can download the example code files for this book from your account at `www.packtpub.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 `www.packtpub.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen 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`. 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: `http://www.packtpub.com/sites/default/files/downloads/ArtificialIntelligenceByExample_ColorImages.pdf`.

Code in Action

Visit the following link to check out videos of the code being run:
`https://goo.gl/M5ACiy`

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. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
MS1='full'
MS2='space'
I=['1','2','3','4','5','6']
for im in range(2):
```

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

```
Weights:
[[ 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]]
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "For this example, click on **Load data.**"

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

1
Become an Adaptive Thinker

In May 2017, Google revealed AutoML, an automated machine learning system that could create an artificial intelligence solution without the assistance of a human engineer. **IBM Cloud** and **Amazon Web Services** (**AWS**) offer machine learning solutions that do not require AI developers. GitHub and other cloud platforms already provide thousands of machine learning programs, reducing the need of having an AI expert at hand. These cloud platforms will slowly but surely reduce the need for artificial intelligence developers. Google Cloud's AI provides intuitive machine learning services. Microsoft Azure offers user-friendly machine learning interfaces.

At the same time, **Massive Open Online Courses** (**MOOC**) are flourishing everywhere. Anybody anywhere can pick up a machine learning solution on GitHub, follow a MOOC without even going to college, and beat any engineer to the job.

Today, artificial intelligence is mostly mathematics translated into source code which makes it difficult to learn for traditional developers. That is the main reason why Google, IBM, Amazon, Microsoft, and others have ready-made cloud solutions that will require fewer engineers in the future.

As you will see, starting with this chapter, you can occupy a central role in this new world as an adaptive thinker. There is no time to waste. In this chapter, we are going to dive quickly and directly into reinforcement learning, one of the pillars of Google Alphabet's DeepMind asset (the other being neural networks). Reinforcement learning often uses the **Markov Decision Process** (**MDP**). MDP contains a memoryless and unlabeled action-reward equation with a learning parameter. This equation, the Bellman equation (often coined as the Q function), was used to beat world-class Atari gamers.

The goal here is not to simply 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 start 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 Google's DeepMind projects.

The following topics will be covered in this chapter:

- A three-dimensional method to implement AI, ML, and DL
- Reinforcement learning
- MDP
- Unsupervised learning
- Stochastic learning
- Memoryless learning
- The Bellman equation
- Convergence
- A Python example of reinforcement learning with the Q action-value function
- Applying reinforcement learning to a delivery example

Technical requirements

- Python 3.6x 64-bit from `https://www.python.org/`
- NumPy for Python 3.6x
- Program on Github, `Chapter01 MDP.py`

Check out the following video to see the code in action:

`https://goo.gl/72tSxQ`

How to be 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, right? 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 a machine learning model, the agent is the function calculating through this trial-and-error process. This thought process in machine learning is the **MDP**. This form of action-value learning is sometimes called **Q**.

To master the outcomes of MDP in theory and practice, a three-dimensional method is a prerequisite.

The three-dimensional approach that will make you an artificial expert, in general terms, means:

- Starting by describing a problem to solve with real-life cases
- Then, building a mathematical model
- Then, write source code and/or using a cloud platform solution

It is a way for you to enter any project with an adaptive attitude from the outset.

Addressing real-life issues before coding a solution

In this chapter, we are going to tackle Markov's Decision Process (Q function) and apply it to reinforcement learning with the Bellman equation. You can find tons of source code and examples on the web. However, most of them are toy experiments that have nothing to do with real life. For example, reinforcement learning can be applied to an e-commerce business delivery person, self-driving vehicle, or a drone. You will find a program that calculates a drone delivery. However, it has many limits that need to be overcome. 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?
- 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?

In just a few minutes, you will be at the center of attention, among theoreticians who know more than you on one side and angry managers who want solutions they cannot get on the other side. Your real-life approach will solve these problems.

A foolproof method is the practical three-dimensional approach:

- **Be a subject matter expert** (**SME**): First, you have to be an SME. If a theoretician geek comes up with a hundred Google DeepMind TensorFlow functions to solve a drone trajectory problem, you now know it is going to be a tough ride if real-life parameters are taken into account.
  An SME knows the subject and thus can quickly identify the critical factors of a given field. Artificial intelligence often requires finding a solution to a hard 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 at your everyday life 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 solid 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 in 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.

Step 1 – MDP in natural language

Step 1 of any artificial intelligence problem is to 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. If you do not have a driver's license or never drive, you can easily replace moving around in a car by moving around on foot.

Let's say you are an e-commerce business driver delivering a package in an area you do not know. You are the operator of a self-driving vehicle. You have a GPS system with a beautiful color map on it. The areas 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 area **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 graph represents the locations and routes that you can possibly cover.

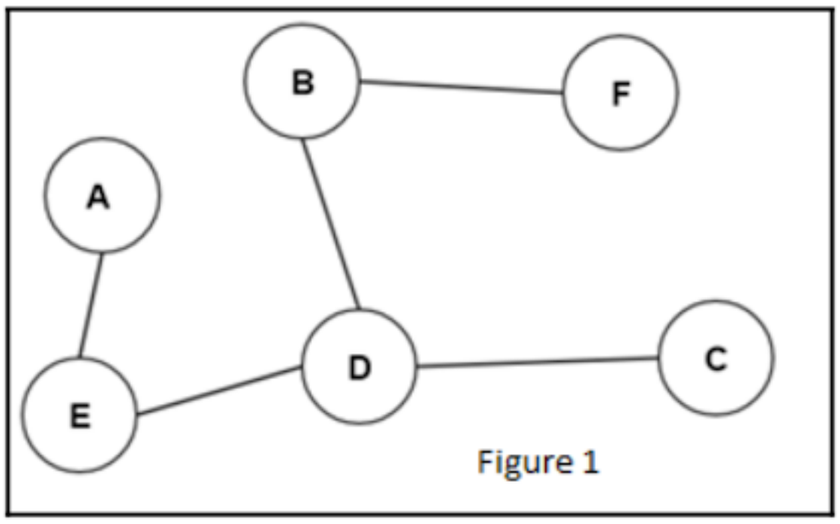


Figure 1

The guiding 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*.
- Your next **action** is the letter *a (action)*. This action a is not location A.
- The next action *a* (not location A) is to go to location **B**. You look at your guiding 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**.

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 area; 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 not driving anymore; 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.

The agent now sees what you have asked it to do and checks its mapping **environment**, which represents all the areas 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 if its strategy meets yours. You have your **policy** *P*—your way of thinking—which is to take the shortest paths possible. Will the agent agree? What's going on in its mind? You observe and begin to realize things you never noticed before. Since this is the first time you are using this car and guiding system, the agent is **memoryless**, which is an MDP feature. This means the agent just doesn't know anything about what went on before. It seems to be happy with just calculating from this **state** *s* at area **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 write 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 to go 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**. 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 your guiding system breaks down. All you can see on the screen is that static image of the areas of the last calculation. You look up and see that a traffic jam is building up. Area **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** to get a taxi that can take special lanes. You are going to need your agent!

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. This is the kind of problem that persons, or soon, self-driving vehicles (not to speak about drone air jams), delivering parcels encounter all day long to get the workload done. The number of parcels to delivery per hour is an example of the workload that needs to be taken into account when making a decision.

To represent the problem at this point, the best way is to express this whole process mathematically.

Step 2 – 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. Mathematics is viewed in the perspective of getting a job done. 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 only 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 of a problem through by finding something familiar around us, such as the delivery itinerary example covered before. 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 the parameters in a way you are used to, you can proceed further.

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

From MDP to the Bellman equation

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

To do that, there was a strategy—a policy represented by **P**. All of this can be shown in one mathematical expression, the MDP state transition function:

$$Pa(s, s')$$

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 called **state transition function:**

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

This is the basis of MDP. The reward (right or wrong) is represented in the same way:

$$Ra(s, s')$$

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. This means that potentially, all states can go to another state.

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 adapted to
# Reinforcement Learning
# 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] ])
```

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`, *Think like a Machine*).

There are several properties of this decision process. A few of them are mentioned here:

- **The Markov property**: The process is applied when the past is not taken into account. It is the memoryless property of this decision process, just as you do in a car with a guiding system. You move forward to reach your goal. This is called the Markov property.
- **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 trip. We would know exactly what A means and use that property to make a decision. We would be in the future looking at the past. MDP does not take these labels into account. This means that this is unsupervised learning**.** 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 **B** 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 thousands 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:

- $\mathbf{P_a(s,s')}$: A **policy**, **P**, or strategy to move from one state to another
- $\mathbf{T_a(s,s')}$: A **T**, or stochastic (random) **transition**, function to carry out that action
- $\mathbf{R_a(s,s')}$: 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's how reinforcement learning is often implemented.

Randomness is a property of MDP.

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

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

Once the code has been run, a new random action (state) has been chosen.

The Bellman equation is the road to programming reinforcement learning.

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

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

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 driver had to be happy. Maybe she/he had a crunch in a candy bar to feel good, which is the human counterpart of the reward matrix. The automatic driver maybe ate (reward matrix) some electricity, renewable energy of course! The reward is a number such as 50 or 100 to show the agent that it's on the right track. It's like when a student gets a good grade in an exam.
- `R(s)`: This is the sum of the values up to there. It's the total reward at that point.

- ϒ = **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 really know the outcome. You might have 80 out of 100 (0.8) chances of clearing it. That's painful, but that's life. This is what makes Bellman's equation and MDP 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).

Step 3 – 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 3.6. 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 state
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 only values > 0 are taken into account. They represent the 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):
    next_action = int(ql.random.choice(PossibleAction,1))
    return next_action

# Sample next action to be performed
action = ActionChoice(PossibleAction)
```

Now comes the core of the system containing Bellman's 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. It 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)
```

After the process is repeated and until the learning process is over, 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. The result comes out as a normed percentage.

View the Python program at `https://github.com/PacktPublishing/Artificial-Intelligence-By-Example/blob/master/Chapter01/MDP.py`.

The lessons of reinforcement learning

Unsupervised reinforcement machine learning, such as MDP and Bellman's equation, will topple traditional decision-making software in the next few years. Memoryless reinforcement learning requires few to no business rules and thus doesn't require human knowledge to run.

Being an adaptive AI thinker involves three requisites—the effort to be an SME, working on mathematical models, and understanding source code's potential and limits:

- **Lesson 1**: Machine learning through reinforcement learning can beat human intelligence in many cases. No use fighting! The technology and solutions are already here.
- **Lesson 2**: Machine learning 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. This job will never go away, even if you're setting up solutions such as Google's AutoML that don't require much development.

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. The days of neuroscience imitating humans are over. Cheap, powerful computers have all the leisure it takes to compute millions of possibilities and choose the best trajectories.

Humans need to be more intuitive, make a few decisions, and see what happens because humans cannot try 50,000 ways of doing something. Reinforcement learning marks a new era for human thinking by surpassing human reasoning power.

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 every-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 artificial intelligence.

How to use the outputs

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

```
# Markov Decision Process (MDP) - Bellman's equations adapted to
# Reinforcement Learning with the Q action-value(reward) matrix
```

```
# 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 Bellman's 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` (highest value divided by other values).

As Python geeks, we are overjoyed. We made something rather difficult to work, namely reinforcement learning. As mathematical amateurs, we are elated. We know what MDP and Bellman's 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 his/her 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 the dimensions of a subject. To solve this new problem, let's go back to step 1 with the result.

By formatting the result in Python, a graphics tool, or a spreadsheet, the result that is 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 to 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.

The most useful way of putting it remains the normalized version in percentages. This reflects the stochastic (random) property of the solution, which produces probabilities and not certainties, as shown in the following matrix:

	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 a road. But I made no mention of it in the result 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. This means we can use this program for many other subjects without writing thousands of lines of code.

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 going from point **F** to **C** in a warehouse, as shown in the following diagram:

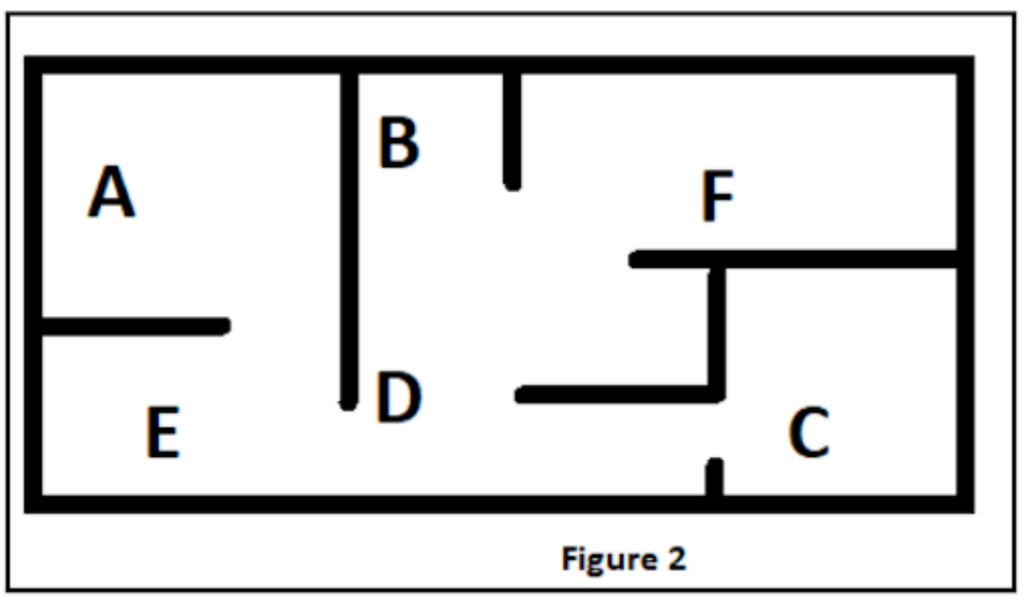


Figure 2

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 artificial intelligence begins in the abstract universe of human thinking.

Every single company, person, or system requires automatic planning and scheduling (see `Chapter 12`, *Automated Planning and Scheduling*). 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:*

$$x = \begin{bmatrix} x1 \\ x2 \\ x3 \\ x4 \\ x5 \\ x6 \end{bmatrix}$$

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.

Case 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.

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. Traditional software will soon be in the museum of computer science.

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; 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 its best. Change the reward matrix a little to see what happens. Design your own reward matrix trajectory. It can be an itinerary or a decision-making process.

Summary

Presently, artificial intelligence 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 with 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 (See `Chapter 16`, *Improve the Emotional Intelligence Deficiencies of Chatbots*, and `Chapter 17`, *Quantum Computers That Think*) in chatbots.

That being said, a multi-dimensional approach is a requisite 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 and/or writing code). In real-life, AI solutions do not just grow spontaneously in companies like trees. 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 Bellman's 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). This 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`, *Think Like a Machine*, 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 more 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 artificial intelligence? (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
Think like a Machine

The first chapter described a reinforcement learning algorithm through the Q action-value function used by DQN. The agent was a driver. You are at the heart of DeepMind's approach to AI.

DeepMind is no doubt one of the world leaders in applied artificial intelligence. Scientific, mathematical, and applications research drives its strategy.

DeepMind was founded in 2010, was acquired by Google in 2014, and is now part of Alphabet, a collection of companies that includes Google.

One of the focuses of DeepMind is on reinforcement learning. They came up with an innovate version of reinforcement learning called **DQN** and referring to deep neural networks using the Q function (Bellman's equation). A seminal article published in February 2015 in *Nature* (see the link at the end of the chapter) shows how DQN outperformed other artificial intelligence research by becoming a human game tester itself. DQN then went on to beat human game testers.

In this chapter, the agent will be an **automated guided vehicle** (**AGV**). An AGV takes over the transport tasks in a warehouse. This case study opens promising perspectives for jobs and businesses using DQN. Thousands upon thousands of warehouses require complex reinforcement learning and customized transport optimization.

This chapter focuses on creating the **reward matrix**, which was the entry point of the Python example in the first chapter. To do so, it describes how to add a primitive McCulloch-Pitts neuron in TensorFlow to create an intelligent adaptive network and add an N (network) to a Q model. It's a small N that will become a feedforward neural network in `Chapter 4`, *Become an Unconventional Innovator*, and more in `Chapter 12`, *Automated Planning and Scheduling*. The goal is not to copy DQN but to use the conceptual power of the model to build a variety of solutions.

The challenge in this chapter will be to think literally like a machine. The effort is not to imitate human thinking but to beat humans with machines. This chapter will take you very far from human reasoning into the depth of machine thinking.

The following topics will be covered in this chapter:

- AGV
- The McCulloch-Pitts neuron
- Creating a reward matrix
- Logistic classifiers
- The logistic sigmoid
- The softmax function
- The one-hot function
- How to apply machine learning tools to real-life problems such as warehouse management

Technical requirements

- Python 3.6x 64-bit from `https://www.python.org/`
- NumPy for Python 3.6x
- TensorFlow from `https://deepmind.com/` with TensorBoard

The following files:

- `https://github.com/PacktPublishing/Artificial-Intelligence-By-Example/blob/master/Chapter02/MCP.py`
- `https://github.com/PacktPublishing/Artificial-Intelligence-By-Example/blob/master/Chapter02/SOFTMAX.py`

Check out the following video to see the code in action:

`https://goo.gl/jMWLg8`

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 the other. First, it's important to just think and talk about the problem to solve 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, start the software implementation.

Designing datasets in natural language meetings

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 indifferently 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 design a matrix that everybody agrees with.

This means many meetings with the IT department to obtain data, the SME and reinforcement learning experts. An AGV requires information coming from different sources: daily forecasts and real-time warehouse flows.

At one point, the project will be at a standstill. It is simply too complicated to get the right data for the reinforcement program. This is a real-life case study that I modified a little for confidentiality reasons.

The warehouse manages thousands of locations and hundreds of thousands of inputs and outputs. The Q function does not satisfy the requirement in itself. A small neural network is required.

In the end, through tough negotiations with both the IT department and the users, a dataset format is designed that fits the needs of the reinforcement learning program and has enough properties to satisfy the AGV.

Using the McCulloch-Pitts neuron

The mathematical aspect relies on finding a model for inputs for huge volumes in a corporate warehouse.

In one mode, the inputs can be described as follows:

- Thousands of forecast product arrivals with a low priority weight: *w1 = 10*
- Thousands of confirmed arrivals with a high priority weight: *w2 = 70*
- Thousands of unplanned arrivals decided by the sales department: *w3 = 75*
- Thousands of forecasts with a high priority weight: *w4 = 60*
- Thousands of confirmed arrivals that have a low turnover and so have a low weight: *w5 = 20*

These weights represent vector w:

$$x = \begin{bmatrix} w1 \\ w2 \\ w3 \\ w4 \\ w5 \end{bmatrix} = \begin{bmatrix} 10 \\ 70 \\ 75 \\ 60 \\ 20 \end{bmatrix}$$

All of these products have to be stored in optimal locations, and the distance between nearly 100 docks and thousands of locations in the warehouse for the AGV has to be minimized.

Let's focus on our neuron. Only these weights will be used, though a system such as this one will add up to more than 50 weights and parameters per neuron.

In the first chapter, the reward matrix was size 6x6. Six locations were described (**A** to **F**), and now six locations (l1 to l6) will be represented in a warehouse.

A 6x6 reward matrix represents the target of the McCulloch-Pitts layer implemented for the six locations.

Also, this matrix was the input in the first chapter. In real life, and in real companies, you will have to find a way to build datasets from scratch. The reward matrix becomes the output of this part of the process. 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.

```
# 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 this warehouse problem, the McCulloch-Pitts neuron sums up the weights of the priority vector described previously to fill in the reward matrix.

Each location will require its neuron, with its weights.

$$INPUTS- > WEIGHTS - BIAS- > VALUES$$

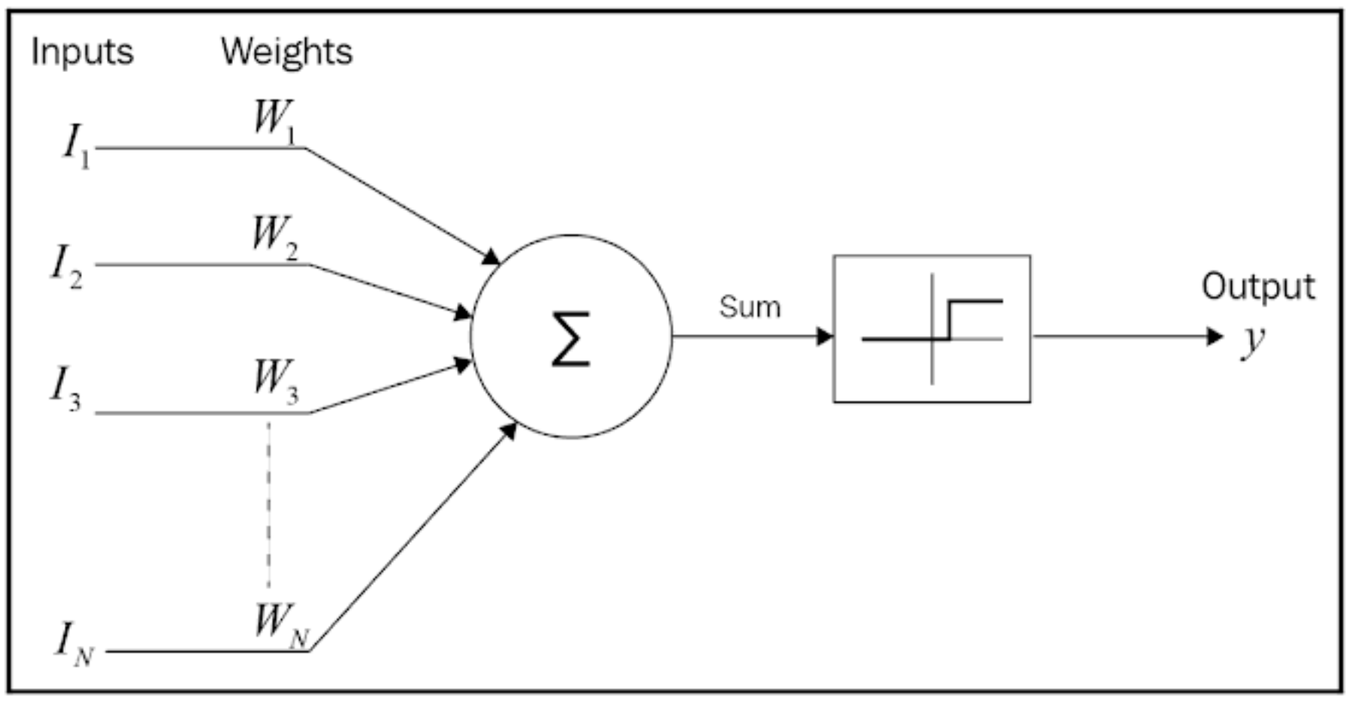
- Inputs are the flows in a warehouse or any form of data
- Weights will be defined in this model
- Bias is for stabilizing the weights
- Values will be the output

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. This is precisely where you need to think like a machine and forget about human neuroscience brain approaches for this type of problem. Starting from `Chapter 8`, *Revolutions Designed for Some Corporations and Disruptive Innovations Small to Large Companies*, human cognition will be built on top of these models, but the foundations need to remain mathematical.

The following diagram shows the McCulloch-Pitts, neuron model.



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

A Python-TensorFlow program, `MCP.py` will be used to illustrate the neuron.

When designing neurons, the computing performance needs to be taken into account. The following source code configures the threads. You can fine-tune your model according to your needs.

```
config = tf.ConfigProto(
    inter_op_parallelism_threads=4,
    intra_op_parallelism_threads=4
)
```

In the following source code, the placeholders that will contain the input values (`x`), the weights (`w`), and the bias (`b`) are initialized. A placeholder is not just a variable that you can declare and use later when necessary. It represents the structure of your graph:

```
x = tf.placeholder(tf.float32, shape=(1, 6), name='x')
w = tf.placeholder(tf.float32, shape=(6, 1), name='w')
b = tf.placeholder(tf.float32, shape=(1), name='b')
```

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

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

The mathematical function becomes a one-line code with 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.

```
y = tf.matmul(x, w) + b
s = tf.nn.sigmoid(y)
```

Before starting a session, the McCulloch-Pitts neuron (1943) needs an operator to directly set its weights. That is the main difference between the McCulloch-Pitts neuron and the perceptron (1957), which is the model of modern deep learning neurons. The perceptron optimizes its weights through optimizing processes. `Chapter 4`, *Become an Unconventional Innovator*, describes the modern perceptron.

The weights are now provided, and so are the quantities for each x stored at l_1, one of the locations of the warehouse:

$$x = \begin{bmatrix} w1 \\ w2 \\ w3 \\ w4 \\ w5 \end{bmatrix} = \begin{bmatrix} 10 \\ 70 \\ 75 \\ 60 \\ 20 \end{bmatrix}$$

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.

```
with tf.Session(config=config) as tfs:
    tfs.run(tf.global_variables_initializer())
    w_t = [[.1, .7, .75, .60, .20]]
    x_1 = [[10, 2, 1., 6., 2.]]
    b_1 = [1]
    w_1 = np.transpose(w_t)
    value = tfs.run(s,
        feed_dict={
            x: x_1,
            w: w_1,
            b: b_1
        }
    )
print ('value for threshold calculation',value)
```

The session starts; the weights (`w_t`) and the quantities (`x_1`) of the warehouse flow are entered. Bias is set to `1` in this model. `w_1` is transposed to fit `x_1`. The placeholders are solicited with `feed_dict`, and the value of the neuron is calculated using the sigmoid function.

The program returns the following value.

```
print ('value for threshold calculation',value)
value for threshold calculation [[ 0.99971133]]
```

This value represents the activity of location l_1 at a given date and a given time. The higher the value, the higher the probable saturation rate of this area. That means there is little space left for an AGV that would like to store products. 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**:

$$\mathbf{A} = \text{Availability} = 1 - \text{load}$$

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 lx',1-value)
Availability of lx [[ 0.00028867]]
```

For example, the load of l_1 has a probable load of 0.99 and its probable *availability* is 0.002. The goal of the AGV is to search and find the closest and most available location to optimize its trajectories. l_1 is obviously not a good candidate at that day and time. **Load** is a keyword in production activities as in the Amazon example in `Chapter 12`, *Automated Planning and Scheduling*.

When all of the six locations' availabilities has been calculated by the McCulloch-Pitts neuron—each with its respective *x* quantity inputs, weights, and bias—a location vector of the results of this system will be produced. This means that the program needs to be implemented to run all six locations and not just one location:

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

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

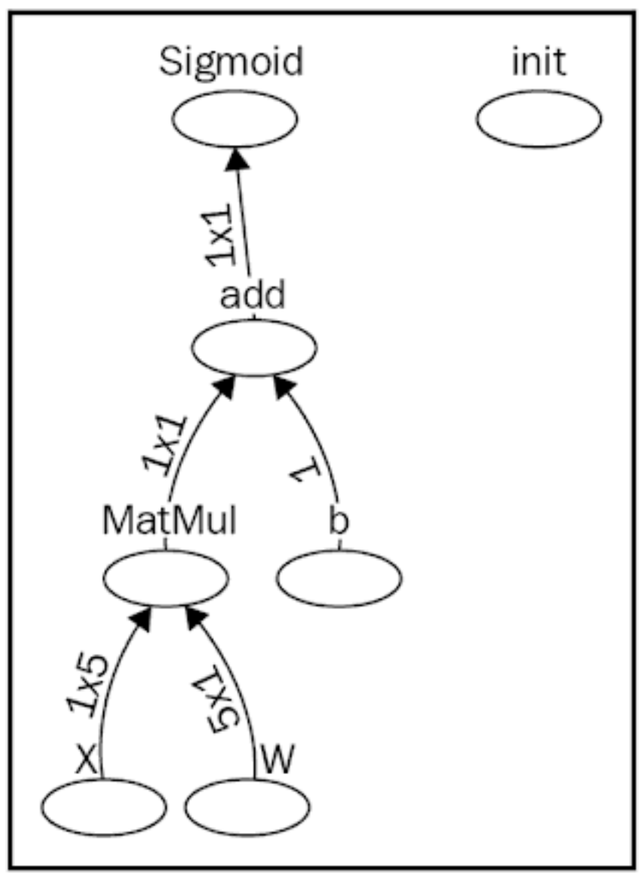
$$lv = \begin{bmatrix} 0.0002 \\ 0.2 \\ 0.9 \\ 0.0001 \\ 0.4 \\ 0.6 \end{bmatrix}$$

lv 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. Once the choice is made, the reinforcement learning program presented in the first chapter will optimize the AGV's trajectory to get to this specific warehouse location.

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

The architecture of Python TensorFlow

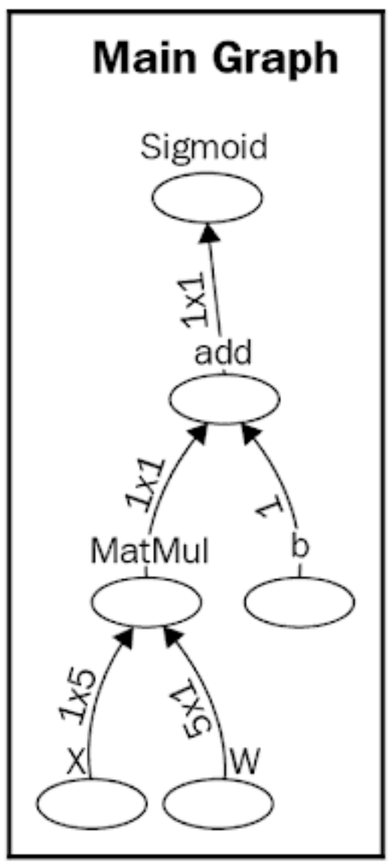
Implementation of the McCulloch-Pitts neuron can best be viewed with TensorBoard, as shown in the following graph:



This is obtained by adding the following TensorBoard code at the end of your session. This data flow graph will help optimize a program when things go wrong.

```
#_____________Tensorboard_____________________________

#with tf.Session() as sess:

Writer = tf.summary.FileWriter("directory on your machine", tfs.graph)
Writer.close()


def launchTensorBoard():
    import os
    #os.system('tensorboard --logdir=' + 'your directory')
    os.system('tensorboard --logdir=' + 'your directory')
    return

import threading
t = threading.Thread(target=launchTensorBoard, args=([]))
t.start()

tfs.close()
#Open your browser and go to http://localhost:6006
#Try the various options. It is a very useful tool.
#close the system window when your finished.
```

When you open the URL indicated in the code on your machine, you will see the following TensorBoard data flow graph:



Logistic activation functions and classifiers

Now that the value of each location of *L={l1,l2,l3,l4,l5,l6}* contains its availability in a vector, the locations can be sorted from the most available to least available location. From there, the reward matrix for the MDP process described in the first chapter can be built.

Overall architecture

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

- **Chapter 1**: *Become an Adaptive Thinker*: A reinforcement learning program based on the value-action Q function using a reward matrix that is yet to be calculated. The reward matrix was given in the first chapter, but in real life, you'll often have to build it from scratch. This could take weeks to obtain.
- **Chapter 2**: A set of six neurons that represent the flow of products at a given time at six locations. The output is the availability probability from 0 to 1. The highest value is the highest availability. The lowest value is the lowest availability.

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

- An AGV is moving in a warehouse and is waiting to receive its next location to use an MDP, in order to calculate an optimal trajectory of its mission, as shown in the first chapter.
- An AGV is using a reward matrix that was given in the first chapter, but it needs to be designed in a real-life project through meetings, reports, and acceptance of the process.
- A system of six neurons, one per location, weighing the real quantities and probable quantities to give an availability vector *lv* 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 *lv* and the reward matrix *R* is missing.

That bridge function is a logistic classifier based on the outputs of the *y* neurons.

At this point, the system:

- Took corporate data
- Used *y* 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 *lv* (the six location values) to find the best location for the AGV. It is based on the output of the six neurons (*input x 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, AI applications use applied mathematics with probable values, not raw outputs. In the warehouse model, 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 a number of 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. This will be used as the activation function of the neuron. The threshold is usually a value above which the neuron has a *y=1* value; or else it has *y=0*. In this case, the minimum value will be 0 because the activation function will be more complex.

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, x is the result of the logistic sigmoid function.

The code has been rearranged in the following example to show the reasoning process:

```
#For given variables:
x_1 = [[10, 2, 1., 6., 2.]]       # the x inputs
w_t = [[.1, .7, .75, .60, .20]]  # the corresponding weights
b_1 = [1]                         # the bias
# A given total weight y is calculated
y = tf.matmul(x, w) + b
```

```
# then the logistic sigmoid is applied to y which represents the "x" in the
formal definition of the Logistic Sigmoid
s = tf.nn.sigmoid(y)
```

Thanks to the logistic sigmoid function, the value for the first location in the model comes out as `0.99` (level of saturation of the location).

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

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

$$lv = \begin{bmatrix} 0.0002 \\ 0.2 \\ 0.9 \\ 0.0001 \\ 0.4 \\ 0.6 \end{bmatrix} \rightarrow [\,?\,]$$

When analyzing *lv*, 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 *lv* exceeds 1. That is not possible. Probability cannot exceed 1. The program needs to fix that. In the source code, *lv* will be named `y`.

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

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

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

Bear in mind that these *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. This is a fundamental function to understand and master. In the case of the warehouse model, an AGV needs to make a probable choice between six locations in the *lv* vector. However, the total of the *lv* values exceeds 1. *lv* requires normalization of the softmax function *S*. In this sense, the softmax function can be considered as a generalization of the logistic sigmoid function. In the code, *lv* vector will be named `y`.

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

The following code used is `SOFTMAX.py`; `y` represents the *lv* vector in the following source code.

```
# 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]
```

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

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

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

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

Now, each value of the vector can be normalized in this type of multinomial distribution stabilization by simply applying a division, as follows:

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

```
#Vector to be stabilized [2.0, 1.0, 0.1, 5.0, 6.0, 7.0]
#Stabilized vector [0.004, 0.002, 0.001, 0.089, 0.243, 0.661]
```

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

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

The following code details the process:

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

y1 = [0.0002, 0.2, 0.9,0.0001,0.4,0.6]
print("Stablized vector",softmax(y1))
print("sum of vector",sum(softmax(y1)))
# Stabilized vector [ 0.11119203 0.13578309 0.27343357 0.11118091
0.16584584 0.20256457]
# sum of vector 1.0
```

The softmax function can be used as the output of a classifier (pixels for example) or to make a decision. In this warehouse case, it transforms *lv* into a decision-making process.

The last part of the softmax function requires `softmax(lv)` to be rounded to 0 or 1. The higher the value in `softmax(lv)`, 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 found, as follows:

```
print("highest value in transformed y vector",max(softmax(y1)))
#highest value in normalized y vector 0.273433565194
```

Once line 3 (value `0.273`) has been chosen as the most probable location, it is 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 to encode 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.

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

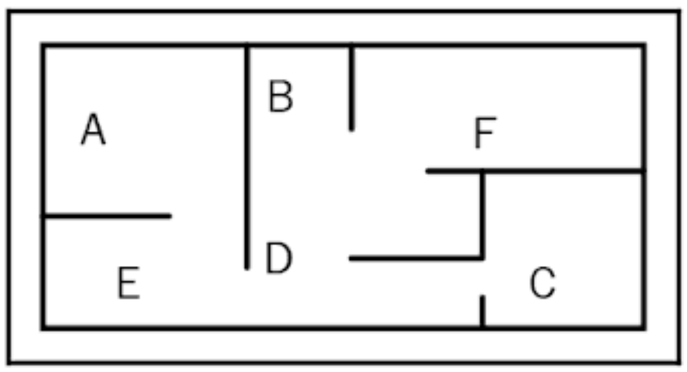
The softmax function is now complete. Location l_3 or **C** is the best solution for the AGV. The probability value is multiplied by 100 in the *R* function and the reward matrix described can now receive the input.

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

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

$$Locations=\{l_1\text{-}A, l_2\text{-}B, l_3\text{-}C, l_4\text{-}D, l_5\text{-}E, l_6\text{-}F\}$$

Value of locations in 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.



We obtain the following reward matrix `R` described in the first chapter.

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 in the first chapter. The output of this chapter is the input of the `R` matrix in the first chapter. The 0 values are there for the agent to avoid those values. This program is designed to stay close to probability standards with positive values, as shown in the following `R` matrix.

```
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 building blocks are in place to begin evaluating the results of the reinforcement learning program.

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 how to build real-life applications with AI technology.

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.

This shows that 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.

You can already use these first two chapters to present powerful trajectory models such as Amazon warehouses and deliveries to your team or customers. Furthermore, Amazon, Google, Facebook, Netflix, and many others are growing their data centers as we speak. Each data center has locations with data flows that need to be calibrated. You can use the ideas given in this chapter to represent the problems and real-time calculations required to calibrate product and data flows.

This neuronal approach is the parent of the multi-layer perceptron that will be introduced in `Chapter 5`, *Manage The Power of Machine Learning and Deep Learning*. There, a shift from machine learning to deep learning will be made.

However, before that, machine learning or deep learning requires evaluation functions. No result can be validated without evaluation, as explained in `Chapter 3`, *Apply Machine Thinking to a Human Problem*. In the next chapter, the evaluation process will be illustrated with chess and a real-life situation.

Questions

1. Was the concept of using an artificial neuron discovered in 1990? (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

- Exploring DeepMind `https://deepmind.com/`
- The TensorFlow site, and support `https://www.tensorflow.org/`
- The original DQN article `https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf`
- Automated solutions in logistics `https://www.logistics-systems.ie/automated-solutions-apm`

3
Apply Machine Thinking to a Human Problem

In the first chapter, the MDP reinforcement program produced a result as an output matrix. In `Chapter 2`, *Think Like a Machine*, the McCulloch-Pitts system of neurons produced an *input reward matrix*. However, the intermediate or final results of these two functions need to be constantly measured. Good measurement solves a substantial part of a given problem since decisions rely on them. Reliable decisions are made with reliable evaluations. The goal of this chapter is to introduce measurement methods.

The key 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.

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

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

This chapter focuses on the main concepts of evaluation and measurement; they set the path to deep learning gradient descent-driven models, which will be explained in the following chapter.

The following topics will be covered in this chapter:

- Evaluation of the episodes of a learning session
- Numerical convergence measurements
- An introduction to the idea of cross-entropy convergence
- Decision tree supervised learning as an evaluation method

- Decision tree supervised learning as a predictive model
- How to apply evaluation tools to a real-life problem you build on your own

Technical requirements

- Python version 3.6 is recommended
- NumPy compatible with Python 3.6
- TensorFlow with TensorBoard
- Graphviz 2.28 for use in Python

Programs are available on GitHub, `Chapter03`:

- `Q_learning_convergence.py`
- `Decision_Tree_Priority_classifier.py`

Check out the following video to see the code in action:

`https://goo.gl/Yrgb3i`

Determining what and how to measure

In `Chapter 2`, *Think Like a Machine*, the system of McCulloch-Pitts neurons generated a vector with a one-hot function in the following process.

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

`R`, the reward vector, represents the input of the reinforcement learning program and needs to be measured.

This chapter deals with an approach designed to build a reward matrix based on the company data. It relies on the data, weights, and biases provided. When deep learning forward feedback neural networks based on perception are introduced (`Chapter 4`, *Become an Unconventional Innovator*), a system cannot be content with a training set. Systems have a natural tendency to learn training sets through backpropagation. In this case, one set of company data is not enough.

In real-life company projects, a system will not be validated 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 consultants can work on with parameter scripts. The consultant introduces parameter scenarios that are tested by the system and measured. In systems with up to 200 parameters per neuron, a consultant will remain necessary for many years to come in an industrial environment. As of `Chapter 4`, *Become an Unconventional Innovator*, the system will be on its own without the help of a consultant. Even then, consultants often 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.

Beyond the reward matrix, the reinforcement program in the first chapter had a learning parameter $\lambda = 0.8$, as shown in the following code source.

```
# 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 something.

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. In the following code, the range is set to 50,000 to make sure the learning process reaches its goal.

```
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)
```

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

Convergence

Building the system was fun. Finding the factors that make the system go wrong is another story.

The model presented so far can be summed up as follows:

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

From *lv* to *R*, the process creates the reward matrix (`Chapter 2`, *Think Like a Machine*) required for the reinforcement learning program (`Chapter 1`, *Become an Adaptive Thinker*), which runs from reading R (reward matrix) to the results. Gamma is the learning parameter, *Q* is the Q learning function, and the results are the states of Q described in the first chapter.

The parameters to be measured are as follows:

- The company's input data. The training sets found on the Web such as MNIST are designed to be efficient. 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 the 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.

The best way to start relies on measuring the quality of convergence of the system, the last step of the whole process.

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

Implicit convergence

In the last part of `Reinforcement_Learning_Q_function.py` in the first chapter, a range of 50,000 is implemented.

The idea is to set the number of episodes at such a level that convergence is 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.

Numerical – controlled convergence

This approach can prove time-saving by using the target result, provided it exists beforehand. Training the reinforcement program in this manner validates the process.

In the following source code, an intuitive *cross-entropy* function is introduced (see `Chapter 9`, *Getting Your Neurons to Work*, for more on cross-entropy).

Cross-entropy refers to energy. The main concepts are as follows:

- Energy represents the difference between one distribution and another
- It is what makes the system continue to train
- When a lot of training needs to be done, there is a **high level of energy**
- When the training reaches the end of its cycles, **the level of energy is low**
- In the following code, **cross-entropy value** (**CEV**) measures the **difference between a target matrix and the episode matrix**
- Cross-entropy is often measured in more complex forms when necessary (see `Chapter 9`, *Getting Your Neurons to Work*, and `Chapter 10`, *Applying Biomimicking to Artificial Intelligence)*

In the following code, a basic function provides sufficient results.

```
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)
    if Q.sum()>0:
     #print("convergent episode:",i,"Q.Sum",Q.sum(),"numerical convergent
value e-1:",Q.sum()-sum)
     #print("convergent episode:",i,"numerical convergent value:",ceg-
Q.sum())
     CEV=-(math.log(Q.sum())-math.log(ceg))
     print("convergent episode:",i,"numerical convergent value:",CEV)
     sum=Q.sum()
     if(Q.sum()-3992==0):
      print("Final convergent episode:",i,"numerical convergent
value:",ceg-Q.sum())
      break; #break on average (the process is random) before 50000
```

The previous program stops before 50,000 epochs. This is because, in the model described in this chapter (see the previous code excerpt), the system stops when it reaches an acceptable `CEV` convergence value.

```
convergent episode: 1573 numerical convergent value: -0.0
convergent episode: 1574 numerical convergent value: -0.0
convergent episode: 1575 numerical convergent value: -0.0
convergent episode: 1576 numerical convergent value: -0.0
convergent episode: 1577 numerical convergent value: -0.0
Final convergent episode: 1577 numerical convergent value: 0.0
```

The program stopped at episode `1577`. Since the decision process is random, the same number will not be obtained twice in a row. Furthermore, the constant `3992` was known in advance. This is possible in closed environments where a pre-set goal has been set. This is not the case often but was used to illustrate the concept of convergence. The following chapters will explore better ways to reach convergence, such as gradient descent.

The Python program is available at:

`https://github.com/PacktPublishing/Artificial-Intelligence-By-Example/blob/master/Chapter03/Q_learning_convergence.py`

Applying machine thinking to a human problem

> *"An efficient manager has a high evaluation quotient. A machine has a better one, in chess and a number of increasing fields. The problem now is to keep up with what the machines are learning!"*
>
> *-Denis Rothman*

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. 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, cooperation between pieces, who owns each of the 64 squares, the king's safety, bishop pairs, knight positioning, and many other factors.

Evaluating a position in a chess game

Evaluating a position in a chess game shows why machines will surpass humans in quite some decision-making fields within the next few years.

The following position 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.

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 the 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 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 artificial intelligence 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	position	Value		Quality Value	TotalValue
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	2	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	TotalValue
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
Rook	a8	d8	5	semi-open d-file attacking 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 = {w1,w2....w30}*, 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 barely more intelligence than this type of pure calculation approach. They don't need more to beat humans.

No human, not even world champions, can calculate this position with this accuracy. The number of parameters to take into account overwhelms them each time they reach a position like this. They then play more or less randomly with some kind of idea in mind. It resembles a lottery sometimes. Chess expert annotators discover this when they run human-played games with powerful chess engines to check the game. The players themselves now tend to reveal their incapacity when questioned.

Now bear in mind that the position analyzed represents 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 also they are impossible to beat.

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

Applying the evaluation and convergence process to a business problem

What was once considered in chess as the ultimate proof of human intelligence has been battered by brute-force calculations with great CPU/RAM capacity. Almost any human problem requiring logic and reasoning can most probably be solved by a machine using relatively elementary processes expressed in mathematical terms.

Let's take the result matrix of the reinforcement learning example of the first chapter. It can also be viewed as a scheduling tool. Automated planning and scheduling have become a crucial artificial intelligence field, as explained in `Chapter 12`, *Automated Planning and Scheduling*. In this case, evaluating and measuring the result goes beyond convergence aspects.

In a scheduling process, the input of the reward matrix can represent the priorities of the packaging operation of some products in a warehouse. It would determine in which order customer products must be picked to be packaged and delivered. These priorities extend to the use of a machine that will automatically package the products in a FIFO mode (first in, first out). The systems provide good solutions, but, in real life, many unforeseen events change the order of flows in a warehouse and practically all schedules.

In this case, the result matrix can be transformed into a vector of a scheduled packaging sequence. The packaging department will follow the priorities produced by the system.

The reward matrix (see `Q_learning_convergence.py`) in this chapter is `R` (see the following code).

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

Its visual representation is the same as in `Chapter 1`, *Become an Adaptive Thinker*. But the values are a bit different for this application:

- **Negative values (-1)**: The agent cannot go there
- **0 values**: The agent can go there
- **100 values**: The agent should favor these locations

The result is produced in a Q function early in the first section of the chapter, in a matrix format, displayed as follows:

```
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. ]]
```

From that result, the following packaging priority order matrix can be deduced.

Priorities	**O1**	**O2**	**O3**	**O4**	**O5**	**O6**
O1	-	-	-	-	258.44	-
O2	-	-	-	321.8	-	207.75
O3	-	-	500	321.8	-	-
O4	-	258.44	401	-	258.44	-
O5	207.75	-	-	321.8	-	-
O6	-	258.44	-	-	-	-

The **non-prioritized vector** (**npv**) of packaging orders is *np*.

$$npv = \begin{bmatrix} O1 \\ O2 \\ O3 \\ O4 \\ O5 \\ O6 \end{bmatrix}$$

The *npv* contains the priority value of each cell in the matrix, which is not a location but an order priority. Combining this vector with the result matrix, the results become priorities of the packaging machine. They now need to be analyzed, and a final order must be decided to send to the packaging department.

Using supervised learning to evaluate result quality

Having now obtained the *npv*, a more business-like measurement must be implemented.

A warehouse manager, for example, will tell you the following:

- Your reinforcement learning program looks satisfactory (`Chapter 1`, *Become an Adaptive Thinker*)
- The reward matrix generated by the McCulloch-Pitts neurons works very well (`Chapter 2`, *Think Like a Machine*)
- The convergence values of the system look nice
- The results on this dataset look satisfactory

But then, the manager will always come up with a killer question, *How can you prove that this will work with other datasets in the future?*

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 relationships between the input and output data. It's not a random process like MDP. They are not trajectories anymore. They are priorities. One method is to used decision trees. In `Chapter 4`, *Become an Unconventional Innovator*, the problem will be solved with a feedforward backpropagation network.

In this model, the properties of the customer orders are analyzed so that we can classify them. This can be translated into decision trees depending on real-time data, to create a distribution representation to predict future outcomes.

1. The first step is to represent the properties of the orders O1 to O6.

```
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

2. The second step is to provide some priority parameters to the learning dataset:

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

3. Step 3 is 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. This simulates 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, 1,1],
      [320, 1,0],
      [256, 1,0]]
```

The features in step 1 apply to each column.

The values in step 2 apply to every line.

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

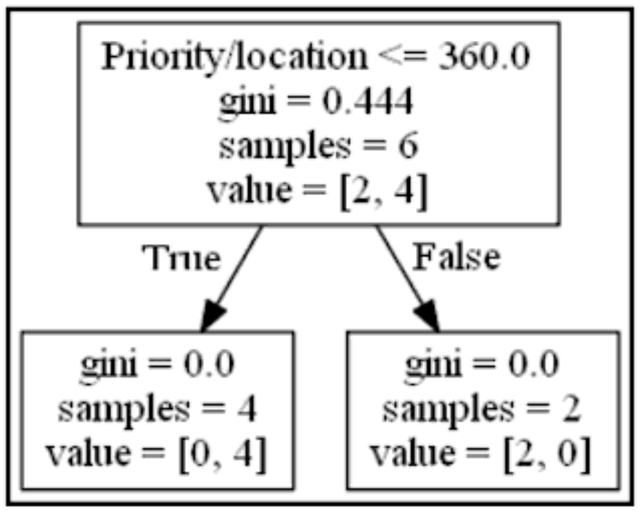
   - The properties of high-priority orders
   - The properties of low-priority orders

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)
```

Applied to thousands of orders on a given day, it will help adapt to real-time unplanned events that destabilize all scheduling systems: late trucks, bad quality products, robot breakdowns, and absent personnel. This means that the system must be able to constantly adapt to new situations and provide priorities to replan in real time.

The program will produce the following graph, which separates the orders into priority groups.



The goal now is to separate the best orders to replan among hundreds of thousands of simulating orders. In this case, the learning dataset has the six values you have been studying in the first two chapters from various angles.

- Priority/location<=360.0 is the division point between the most probable optimized orders (high) and less interesting ones (low).
- Gini impurity. This would be the measure of incorrect labeling if the choice were random. In this case, the dataset is stable.
- The false arrow points out the two values that are not <=360, meaning they are good choices, the optimal separation line of the representation. The ones that are not classified as `False` are considered as *don't eliminate* orders.
  The `True` elements mean: *eliminate orders as long as possible*.
- The value result reads as *[number of false elements, number of true elements]* of the dataset.

If you play around with the values in steps 1, 2, and 3, you'll obtain different separation points and values. This sandbox program will prepare you for the more complex scheduling problems of `Chapter 12`, Automated Planning and Scheduling.

You can use this part of the source code to generate images of this decision tree-supervised learning program:

```
# 5.Producing visualization if necessary
info =
tree.export_graphviz(classify,feature_names=features,out_file=None,filled=F
alse,rounded=False)
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_destination()))

    for edge in edges:
        edges[edge].sort()
        for i in range(2):
            dest = graph.get_node(str(edges[edge][i]))[0]

    graph.write_png('warehouse_example_decision_tree.png')
    print("Open the image to verify that the priority level prediction of the
    results fits the reality of the reward matrix inputs")
```

The preceding information represents a small part of what it takes to manage a real-life artificial intelligence program on premise.

A warehouse manager will want to run this supervised learning decision tree program on top of the system described in `Chapter 2`, *Think Like a Machine*, and `Chapter 3`, *Apply Machine Thinking to a Human Problem*. This is done to generalize these distributed representations directly to the warehouse data to improve the initial corporate data inputs. With better-proven priorities, the system will constantly improve, week by week.

This way of scheduling shows that human thinking was not used nor necessary.

Contrary to the hype surrounding artificial intelligence, most problems can be solved with no human intelligence involved and relatively little machine learning technology.

Human intelligence simply proves that intelligence can solve a problem.

Fortunately for the community of artificial intelligence experts, there are very difficult problems to solve that require more artificial intelligence thinking.

Such a problem will be presented and solved in `Chapter 4`, *Become an Unconventional Innovator*.

The Python program is available at `https://github.com/PacktPublishing/Artificial-Intelligence-By-Example/blob/master/Chapter03/Decision_Tree_Priority_classifier.py`.

Summary

This chapter led artificial intelligence exploration one more step away from neuroscience to reproduce human thinking. Solving a problem like a machine means using a chain of mathematical functions and properties.

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 alien intelligence.

The power of machine learning with *beyond-human* mathematical reasoning is that generalization to other fields is easier. A mathematical model, contrary to 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 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. This opens the door to the constant adaptation of the weights of the network to reach its objectives.

Machine evaluation for convergence through a chess example that has nothing to do with human thinking proves the limits of human intelligence. The decision tree example can beat most humans in classification situations where large amounts of data are involved.

Human intelligence is not being reproduced in many cases and has often been surpassed. In those cases, human intelligence just proves that intelligence can solve a problem, nothing more.

The next chapter goes a step further from human reasoning with self-weighting neural networks and introduces deep learning.

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: `https://chessbookreviews.wordpress.com/tag/zoran-petronijevic/`

4
Become an Unconventional Innovator

In corporate projects, there always comes the point when a problem that seems impossible to solve hits you. At that point, you try everything you 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 implemented successfully with an example of a **feedforward neural network** (**FNN**) with backpropagation.

Feedforward networks are the building blocks of deep learning. The battle around the XOR function perfectly illustrates how deep learning regained popularity in corporate environments. 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 network 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:

- How to hand build an FNN
- Solving XOR with an FNN
- Classification
- Backpropagation
- A cost function

- Cost function optimization
- Error loss
- Convergence

Technical requirements

- Python 3.6x 64-bit from `https://www.python.org/`
- NumPy for Python 3.6x

Programs from GitHub `Chapter04`:

- `FNN_XOR_vintage_tribute.py`
- `FFN_XOR_generalization.py`

Check out the following video to see the code in action:

`https://goo.gl/ASyLWz`

The XOR limit of the original perceptron

Once the feedforward network for solving the XOR problem is built, it will be applied to a material optimization business case. The material-optimizing solution will choose the best combinations of dimensions among billions to minimize the use of a material with the generalization of the XOR function. First, a solution to the XOR limitation of a perceptron must be fully clarified.

XOR and linearly separable models

In the academic world, like the private world, competition exists. Such a situation took place in 1969. Minsky and Papert published *Perceptrons*. They proved mathematically that a perceptron could *not* solve an XOR function. Fortunately, today the perceptron and its neocognitron version form the core model for neural networking.

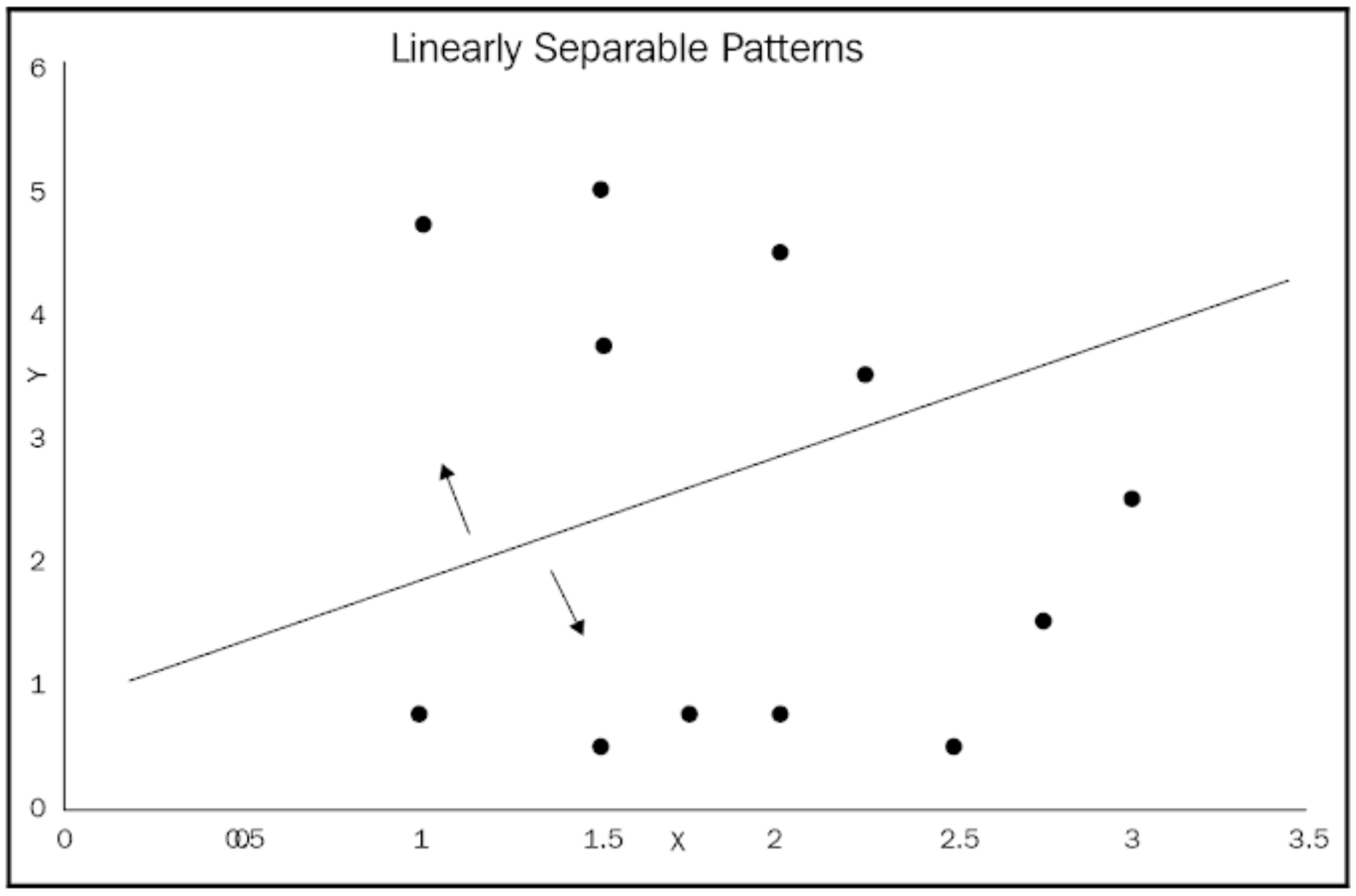
One might be tempted to think, *So what*? However, the entire field of neural networks relies on solving problems such as this to classify patterns. Without pattern classification, images, sounds, and words mean nothing to a machine.

Linearly separable models

The McCulloch-Pitts 1943 neuron (see `Chapter 2`, *Think Like a Machine*) lead to Rosenblatt's 1957-1962 perceptron and the 1960 Widrow-Hoff adaptive linear element (Adaline).

These models are linear models based on *f(x,w)*, requiring 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:



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.

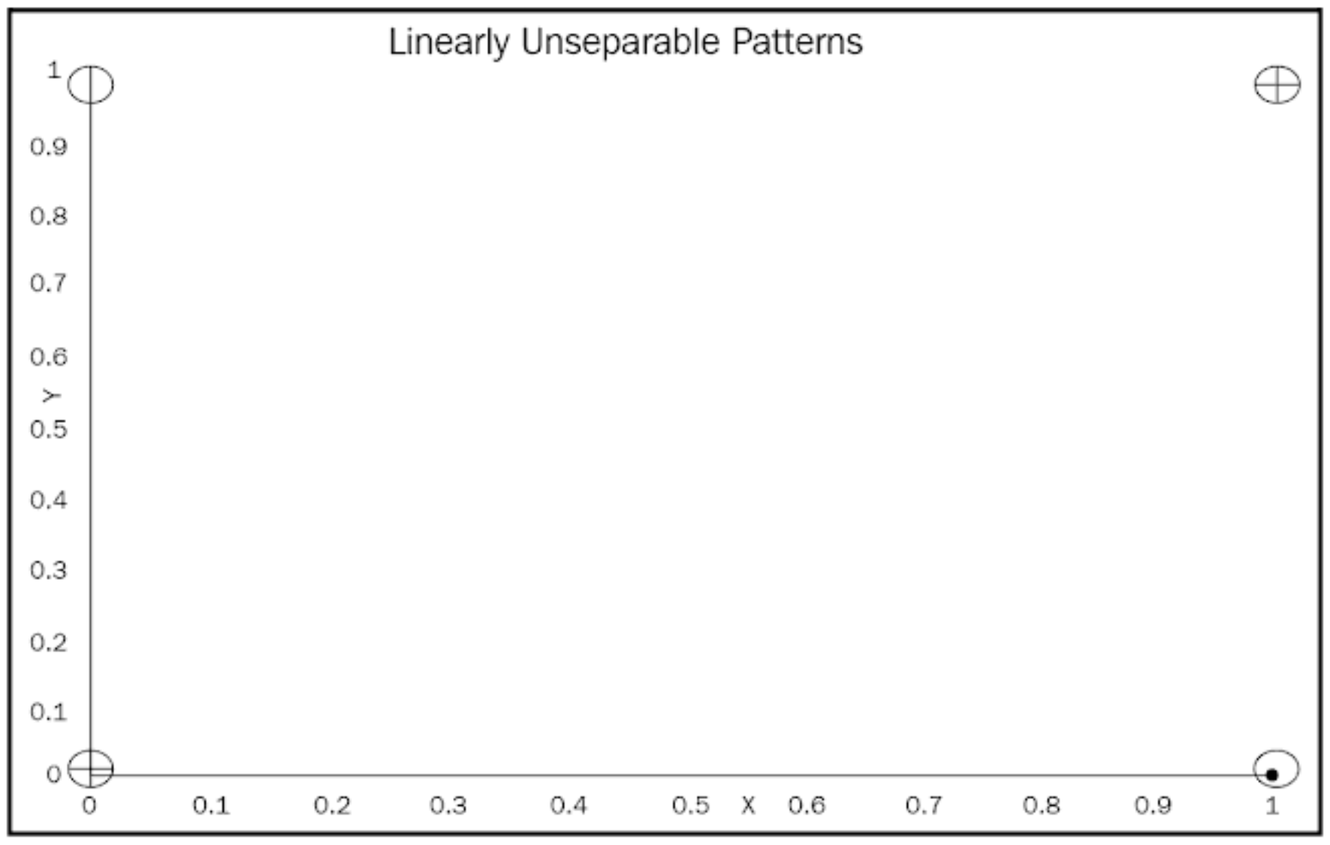
To be linearly separable, a function must be able to separate the *clouds* from the *trees* to classify them. The prerequisite to classification is **separability** of some sort, linear or nonlinear.

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 x_1	Value of x_2	Output
1	1	0
0	0	0
1	0	1
0	1	1

The following graph shows the linear inseparability of the XOR function represented by one perceptron:



The values of the table represent the Cartesian coordinates in this graph. The circle with a cross at *(1,1)* and *(0,0)* cannot be separated from the circles at (1,0) and (0,11). That's a huge problem. It means that Frank Rosenblatt's *f(x,w)* perceptron cannot separate, and thus not classify, these dots into *clouds* and *trees*; an object used to identify that requires linear separability.

Having invented the most powerful concept of the 20th century—**a neuron that can learn**—Frank Rosenblatt had to bear with this limitation through the 1960s.

Let's vindicate this injustice with a vintage solution.

Building a feedforward neural network from scratch

Let's get into a time machine. In nanoseconds, it takes us back to 1969. We have today's knowledge but nothing to prove it. Minsky and Papert have just published their book, *Perceptrons*. They've proven that a perceptron cannot implement the exclusive OR function XOR.

We are puzzled. We know that deep learning will be a great success in the 21st century. We want to try to change the course of history. Thanks to our time machine, we land in a small apartment. It's comfortable, with a vinyl record playing the music we like! There is a mahogany desk with a pad, a pencil, sharpener, and eraser waiting for us. We sit. A warm cup of coffee appears in a big mug. We're ready to solve the XOR problem from scratch. We have to find a way to classify those dots with a neural network.

Step 1 – Defining a feedforward neural network

We look at our piece of paper. We don't have a computer. We're going to have to write code; then we'll hopefully find a computer in a university or a corporation that has a 1960 state-of-the-art language to program in.

We have to be unconventional to solve this problem. First, we must ignore Minsky and Papert's publication and also forget complicated words and theory of the 21st century. In fact, we don't remember much anyway. Time travel made our future fuzzy!

A perceptron is usually represented by a graph. But that doesn't mean much right now. After all, I can't compute circles and lines. In fact, beyond seeing circles and lines, we type characters in computer languages, not circles. So, I decide to simply to write a layer in high-school format. A hidden layer will simply be:

$$h_1 = x * w$$

Ok, now I have one layer. In fact, I just realized that 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 kind of value to multiply x by. I also realized that hidden just means that it's the computation, just as x=2 and x+2 is the hidden layer that leads to 4.

At this point, I've defined a neural network in three lines:

- Input x.
- Some kind of 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.

Now that I know that basically any neural network is built with values transformed by an operation to become an output of something, I need the logic to solve the XOR problem.

Step 2 – how two children solve the XOR problem every day

Let's see how two children solve the XOR problem using a plain everyday example. I strongly recommend this method. I have taken very complex problems, broken them down into small parts to children'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 of 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/no or yes. That's possible. So, the answer is true.

Sipping my coffee in 1969, I imagine the two children. The eldest one is reasonable. The younger one doesn't know really how to count yet and wants to buy both packs of candy.

I decide to write that down on my piece of 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:

$$x_1 * w_1 \ \ or \ \ h_1 = x_1 * w_1$$

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 (w1).

- 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:

$$x_2 * w_3 \ \ or \ \ h_2 = x_2 * w_3$$

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.**

Now I 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 *w2 = 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

$$x_2 * w_4$$

I now have the first two equations expressed in high-school-level code. It's *what one thinks + 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 + other child's opinion.

`h2` sums up what is going on in the other child's mind and conversation: personal opinion + 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.

I don't have time to deal with complicated numbers in an activation function such as logistic sigmoid, so I decide to simply decide 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.

I decided that at this point, I liked both children. Why would I have to hurt one of them? So from now on, $w_3=w_2, w_4=w_1$. After all, I don't have a computer and my time travel window is consuming a lot of energy. I'm going to be pulled back soon.

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.

$$y = h_1 + h_2$$

Since I don't have a computer, I decide to start finding the weights in a practical manner, starting by setting the weights and biases to 0.5, as follows:

$$w_1 = 0.5; w_2 = 0.5; b_1 = 0.5$$
$$w_3 = w_2; w_4 = w_1; b_2 = b_1$$

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; I focus on only modifying w_2 and b_1 after a first try. An hour later, on paper, it works!

I can't believe that this is all there is to it. I copy the mathematical process on a clean sheet of 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 solution is found
w2=w2+0.5
b1=b1+0.5
```

I'm overexcited by the solution. I need to get this little sheet of paper to a newspaper to get it published and change the course of history. I rush to the door, open it but find myself back in the present! I jump and wake up in my bedroom sweating.

I rush to my laptop while this time-travel dream is fresh in my mind to get it into Python for this book.

Why wasn't this deceiving 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.

Implementing a vintage XOR solution in Python with an FNN and backpropagation

I'm still thinking that implementing XOR with so little mathematics might not be that simple. However, since the basic rule of innovating is to be unconventional, I write the code.

To stay in the spirit of a 1969 vintage solution, I decide not to use NumPy, TensorFlow, Theano, or any other high-level library. Writing a vintage FNN with backpropagation written in high-school mathematics is fun.

This also shows that 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, I went against running 20,000 or more episodes in modern CPU-rich solutions to solve the XOR problem. The logic used proves that, basically, 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 3 to 10 iterations (epochs or episodes) depending on how we think it through.

The top of the code simply contains a result matrix with four columns. Each represents the status (`1`=correct, `0`=false) of the four predicates to solve:

```
#FEEDFORWARD NEURAL NETWORK(FNN) WITH BACK PROPAGATION SOLUTION 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 the sheet of paper I 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 can be 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 epoch 0, 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 x_2 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 = 1;x2 = 0;pred=3
                #forward propagation on epoch 1
                hidden_layer_y(epoch,x1,x2,w1,w2,w3,w4,b1,b2,pred,result)
```

A simplified version of a cost function and gradient descent

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.

A 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 simply 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 has been trained correctly. `result[1,0,1,0]` means two out of 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.

In this case, I hijacked the concept and used it to set the learning rate with a one-line function. Why not? It helped to solve gradient descent optimization in one line:

```
if(convergence<0):w2+=0.05;b1=w2
```

By applying the vintage *children buying candy* logic to the whole XOR problem, I 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:

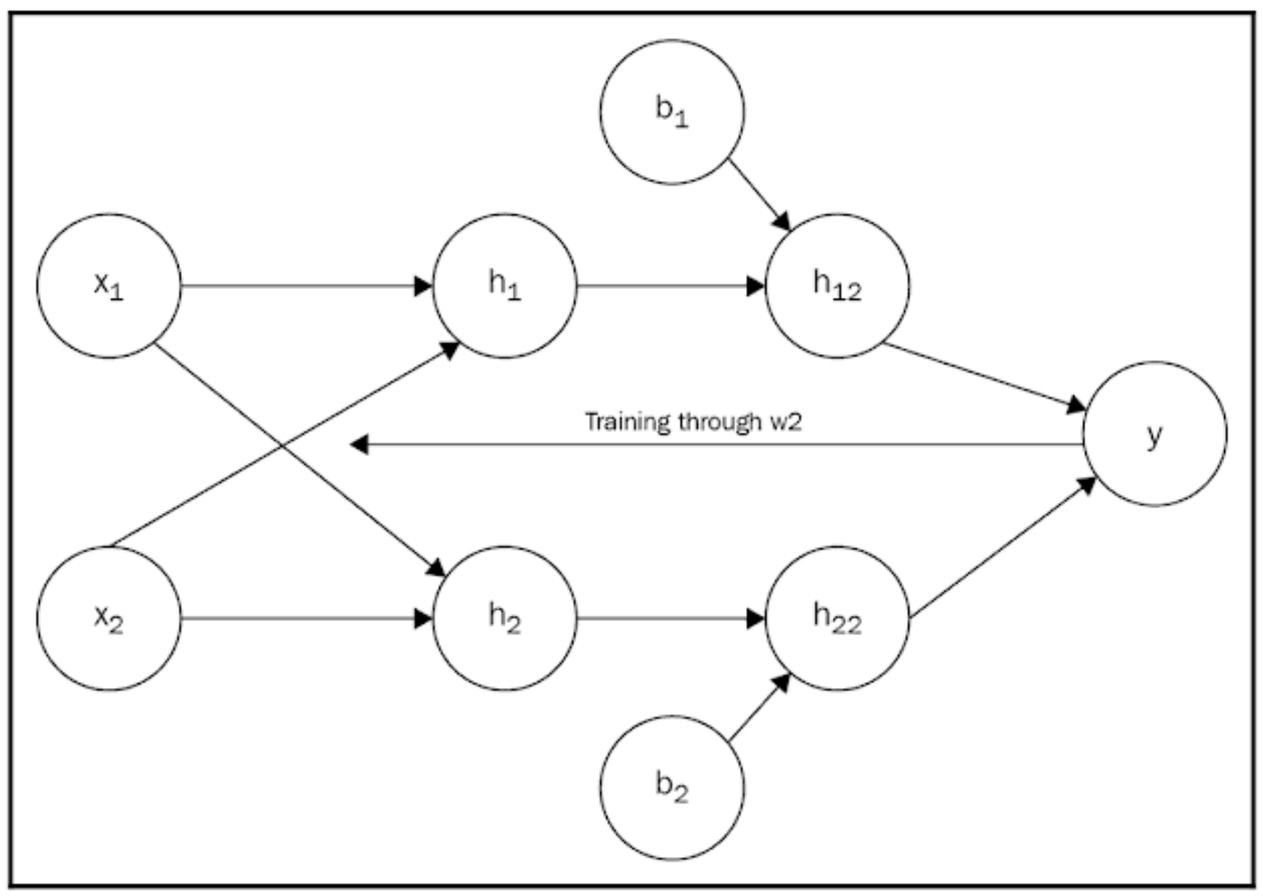
```
epoch: 10 optimization 0 w1: 0.5 w2: 1.0 w3: 1.0 w4: 0.5 b1: -1.0 b2: 1.0
```

This is not a mathematical calculation problem but a logical one, a *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 graph sums up the whole process:



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 counts.

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.

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 classification of features presented to the system. In this case, the features had `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.

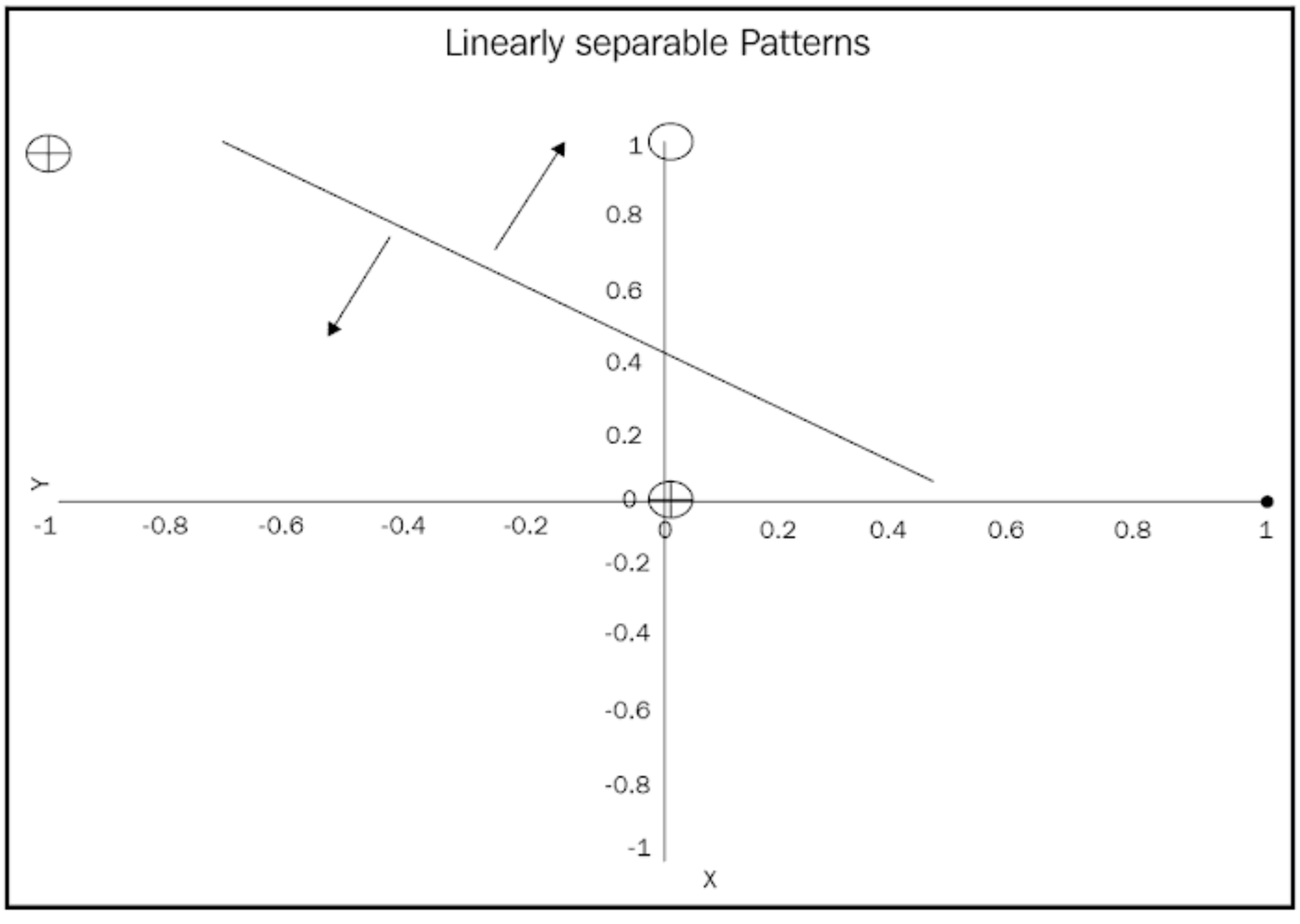
`h1` and `h2` will produce the Cartesian coordinate linear separability training axis, as implemented in the following code:

```
h1= h1 * -b1
h2= h2 * b2
print(h1,h2)
```

Running the program provides a view of the nonlinear input values once they have been trained by the hidden layers. The nonlinear values then become linear values in a linearly separable function:

```
linearly separability through cartesian training -1.0000000000000004
1.0000000000000004
linearly separability through cartesian training -0.0 0.0
linearly separability through cartesian training -0.0 1.0000000000000004
linearly separability through cartesian training -0.0 1.0000000000000004
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:



We have now obtained a separation between the top values (empty circle) representing the intermediate values of the 1,0 and 0,1 inputs, and the bottom values representing the 1,1 and 0,0 inputs. 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 linear 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.

Applying the FNN XOR solution to a case study to optimize subsets of data

The case study described here is a real-life project. The environment and functions have been modified to respect confidentiality. But, the philosophy is the same one as that used and worked on.

We are 7.5 billion people breathing air on this planet. In 2050, there will be about 2.5 billion more. 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 kind of 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 brand 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 x 50 units = 500,000 units* or **stock keeping unit** (**SKU**) 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, laser, packaging, and warehouse. See `Chapter 12`, *Automated Planning and Scheduling*, for Amazon's patented approach to apparel production.

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 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 fabric is wasted while manufacturing jeans, the company will survive the competition. At over 10%, there is a real problem to solve. Losing 20% on all the fabric consumed to manufacture 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 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 loss of fabric by choosing six sizes per day to build 25,000 unit subsets per day
- Generating cut plans of an average of three to six 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 six well-matched sizes among 500,000 units. This is calculated by 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 just find some easy way out of this even if it means wasting fabric. The problem was that in this project, I was paid on a percentage of the fabric I would manage to save. The contract stipulated that I must save 3% of all fabric consumption per month for the whole company to get paid a share of that. Or receive nothing at all. Believe me, once I solved that, I kept that contract as a trophy and a tribute to simplicity.

The first reaction we all have is that this is more than the number of stars in the universe and all that hype!

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. 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 analyze represent large quantities, it will come down to small units of information to analyze:

yottabyte (YB)	10^{24}	yobibyte (YiB)	2^{80}

Today, Google, Facebook, Amazon, and others have yottabytes of data to classify and make sense of. Using the word **big** data doesn't mean much. It's just a lot of data, and so what?

You do not need to analyze 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 six sizes from 1 to 6 were to be picked per day, the sizes could be classified as follows in a table:

Smaller sizes= S={1,2,3}

Larger sizes=L=[4,5,6}