

Deep Learning: Practical Neural Networks with Java

**Build and run intelligent applications by leveraging
key Java machine learning libraries**

A course in three modules

Packt

BIRMINGHAM - MUMBAI

Deep Learning: Practical Neural Networks with Java

Copyright © 2017 Packt Publishing

Published on: May 2017

Production reference: 1310517

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

ISBN 978-1-78847-031-5

www.packtpub.com

Preface

Deep Learning algorithms are being used across a broad range of industries – as the fundamental driver of AI. Machine learning applications are everywhere, from self-driving cars, spam detection, document search, and trading strategies, to speech recognition. This makes machine learning well-suited to the present-day era of Big Data and Data Science. The main challenge is how to transform data into actionable knowledge.

Starting with an introduction to basic machine learning algorithms, to give you a solid foundation, Deep Learning with Java takes you further into this vital world of stunning predictive insights and remarkable machine intelligence. You will learn how to use Java machine learning libraries and apply Deep Learning to a range of real-world use cases. Featuring further guidance and insights to help you solve challenging problems in image processing, speech recognition, language modeling, you will learn the techniques and tools you need to quickly gain insight from complex data. You will apply machine learning methods to a variety of common tasks including classification, prediction, forecasting, market basket analysis, and clustering. Moving on, you will discover how to detect anomalies and fraud, and ways to perform activity recognition, image recognition, and text. Later you will focus on what Perceptrons are and their features. You will implement self-organizing maps using practical examples. Further on, you will work with the examples such as weather forecasting, disease diagnosis, customer profiling, generalization, extreme machine learning and more. Finally, you will learn methods to optimize and adapt neural networks in real time. By the end of this course, you will have all the knowledge you need to perform deep learning on your system with varying complexity levels, to apply them to your daily work.

What this learning path covers

Module 1, Java Deep Learning Essentials, takes you further into this vital world of stunning predictive insights and remarkable machine intelligence. Once you've got to grips with the fundamental mathematical principles, you'll start exploring neural networks and identify how to tackle challenges in large networks using advanced algorithms.

Module 2, Machine Learning in Java, will provide you with the techniques and tools you need to quickly gain insight from complex data. By applying the most effective machine learning methods to real-world problems, you will gain hands-on experience that will transform the way you think about data.

Module 3, Neural Network Programming with Java, Second Edition, takes you on a complete walkthrough of the process of developing basic to advanced practical examples based on neural networks with Java, giving you everything you need to stand out. You will learn methods to optimize and adapt neural networks in real time.

What you need for this learning path

Module 1:

We'll implement deep learning algorithms using Lambda Expressions, hence Java 8 or above is required. Also, we'll use the Java library, DeepLearning4J 0.4 or above.

Module 2:

You'll need Netbeans (www.netbeans.org) or Eclipse (www.eclipse.org). Both are free and available for download at the previously mentioned websites.

Module 3:

You'll need Netbeans (www.netbeans.org) or Eclipse (www.eclipse.org). Both are free and available for download at the previously mentioned websites.

Who this learning path is for

This course is intended for data scientists and Java developers who want to dive into the exciting world of deep learning. It will get you up and running quickly and provide you with the skills you need to successfully create, customize, and deploy machine learning applications in real life.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at <http://www.packtpub.com>. If you purchased this course elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

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 course is also hosted on GitHub at <https://github.com/PacktPublishing/Java-Practical-Deep-Learning>. We also have other code bundles from our rich catalog of books, videos, and courses available at <https://github.com/PacktPublishing/>. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your course, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the course in the search field. The required information will appear under the **Errata** section.

Module 1: Java Deep Learning Essentials

Chapter 1: Deep Learning Overview	3
Transition of AI	4
Things dividing a machine and human	15
AI and deep learning	16
Summary	24
Chapter 2: Algorithms for Machine Learning – Preparing for Deep Learning	25
Getting started	25
The need for training in machine learning	26
Supervised and unsupervised learning	29
Machine learning application flow	36
Theories and algorithms of neural networks	42
Summary	68
Chapter 3: Deep Belief Nets and Stacked Denoising Autoencoders	69
Neural networks fall	69
Neural networks' revenge	70
Deep learning algorithms	78
Summary	107
Chapter 4: Dropout and Convolutional Neural Networks	109
Deep learning algorithms without pre-training	109
Dropout	110
Convolutional neural networks	122
Summary	144

Table of Contents —————

Chapter 5: Exploring Java Deep Learning Libraries – DL4J, ND4J, and More	145
Implementing from scratch versus a library/framework	146
Introducing DL4J and ND4J	148
Implementations with ND4J	150
Implementations with DL4J	156
Summary	177
Chapter 6: Approaches to Practical Applications – Recurrent Neural Networks and More	179
Fields where deep learning is active	180
The difficulties of deep learning	198
The approaches to maximizing deep learning possibilities and abilities	200
Summary	208
Chapter 7: Other Important Deep Learning Libraries	209
Theano	209
TensorFlow	214
Caffe	219
Summary	222
Chapter 8: What's Next?	223
Breaking news about deep learning	223
Expected next actions	226
Useful news sources for deep learning	231
Summary	234

Module 2: Machine Learning in Java —————

Chapter 1: Applied Machine Learning Quick Start	237
Machine learning and data science	237
Data and problem definition	240
Data collection	242
Data pre-processing	245
Unsupervised learning	249
Supervised learning	253
Generalization and evaluation	260
Summary	263

Table of Contents

Chapter 2: Java Libraries and Platforms for Machine Learning	265
The need for Java	266
Machine learning libraries	266
Building a machine learning application	278
Summary	280
Chapter 3: Basic Algorithms – Classification, Regression, and Clustering	281
Before you start	282
Classification	282
Regression	292
Clustering	299
Summary	302
Chapter 4: Customer Relationship Prediction with Ensembles	303
Customer relationship database	304
Basic naive Bayes classifier baseline	307
Basic modeling	311
Advanced modeling with ensembles	313
Summary	322
Chapter 5: Affinity Analysis	323
Market basket analysis	323
Association rule learning	326
The supermarket dataset	330
Discover patterns	330
Other applications in various areas	333
Summary	335
Chapter 6: Recommendation Engine with Apache Mahout	337
Basic concepts	337
Getting Apache Mahout	341
Building a recommendation engine	344
Content-based filtering	359
Summary	360
Chapter 7: Fraud and Anomaly Detection	361
Suspicious and anomalous behavior detection	362
Suspicious pattern detection	363
Anomalous pattern detection	364
Fraud detection of insurance claims	365
Anomaly detection in website traffic	373
Summary	380

Table of Contents

Chapter 8: Image Recognition with Deeplearning4j	381
Introducing image recognition	381
Image classification	389
Summary	399
Chapter 9: Activity Recognition with Mobile Phone Sensors	401
Introducing activity recognition	402
Collecting data from a mobile phone	406
Building a classifier	414
Summary	420
Chapter 10: Text Mining with Mallet – Topic Modeling and Spam Detection	421
Introducing text mining	421
Installing Mallet	424
Working with text data	426
Topic modeling for BBC news	432
E-mail spam detection	439
Summary	444
Chapter 11: What is Next?	445
Machine learning in real life	445
Standards and markup languages	449
Machine learning in the cloud	451
Web resources and competitions	453
Summary	456
Appendix: References	457

**Module 3: Neural Network Programming with Java,
Second Edition**

Chapter 1: Getting Started with Neural Networks	463
Discovering neural networks	463
Why artificial neural networks?	464
From ignorance to knowledge – learning process	472
Let the coding begin! Neural networks in practice	473
The neuron class	475
The NeuralLayer class	477
The ActivationFunction interface	478
The neural network class	479

Table of Contents

Time to play!	481
Summary	483
Chapter 2: Getting Neural Networks to Learn	485
Learning ability in neural networks	486
Learning paradigms	487
The learning process	489
Examples of learning algorithms	493
Time to see the learning in practice!	504
Amazing, it learned! Or, did it really? A further step – testing	509
Summary	511
Chapter 3: Perceptrons and Supervised Learning	513
Supervised learning – teaching the neural net	514
A basic neural architecture – perceptrons	518
Multi-layer perceptrons	522
Learning in MLPs	527
Practical example 1 – the XOR case with delta rule and backpropagation	542
Practical example 2 – predicting enrolment status	545
Summary	548
Chapter 4: Self-Organizing Maps	549
Neural networks unsupervised learning	549
Unsupervised learning algorithms	550
Kohonen self-organizing maps	555
Summary	581
Chapter 5: Forecasting Weather	583
Neural networks for regression problems	583
Loading/selecting data	585
Choosing input and output variables	592
Preprocessing	594
Empirical design of neural networks	613
Summary	617
Chapter 6: Classifying Disease Diagnosis	619
Foundations of classification problems	620
Logistic regression	622
Neural networks for classification	628
Disease diagnosis with neural networks	628
Summary	635

Table of Contents

Chapter 7: Clustering Customer Profiles	637
Clustering tasks	638
Applied unsupervised learning	642
Profiling	643
Summary	651
Chapter 8: Text Recognition	653
Pattern recognition	654
Neural networks in pattern recognition	656
Summary	666
Chapter 9: Optimizing and Adapting Neural Networks	667
Common issues in neural network implementations	668
Input selection	669
Online retraining	677
Adaptive neural networks	682
Summary	684
Chapter 10: Current Trends in Neural Networks	685
Deep learning	686
Deep architectures	688
Implementing a hybrid neural network	696
Summary	700
Appendix: References	701
Bibliography	707
Index	709

Module 1

Java Deep Learning Essentials

*Dive into the future of data science and learn how to build the sophisticated algorithms
that are fundamental to deep learning and AI with Java*

1

Deep Learning Overview

Artificial Intelligence (AI) is a word that you might start to see more often these days. AI has become a hot topic not only in academic society, but also in the field of business. Large tech companies such as Google and Facebook have actively bought AI-related start-ups. Mergers and acquisitions in these AI areas have been especially active, with big money flowing into AI. The Japanese IT/mobile carrier company Softbank released a robot called Pepper in June 2014, which understands human feelings, and a year later they have started to sell Pepper to general consumers. This is a good movement for the field of AI, without a doubt.

The idea of AI has been with us for decades. So, why has AI suddenly became a hot field? One of the factors that has driven recent AI-related movements, and is almost always used with the word AI, is **deep learning**. After deep learning made a vivid debut and its technological capabilities began to grow exponentially, people started to think that finally AI would become a reality. It sounds like deep learning is definitely something we need to know. So, what exactly is it?

To answer the previous questions, in this chapter we'll look at why and how AI has become popular by following its history and fields of studies. The topics covered will be:

- The former approaches and techniques of AI
- An introduction to machine learning and a look at how it has evolved into deep learning
- An introduction to deep learning and some recent use cases

If you already know what deep learning is or if you would like to find out about the specific algorithm of the deep learning/implementation technique, you can skip this chapter and jump directly to *Chapter 2, Algorithms for Machine Learning – Preparing for Deep Learning*.

Although deep learning is an innovative technique, it is not actually that complicated. It is rather surprisingly simple. Reading through this book, you will see how brilliant it is. I sincerely hope that this book will contribute to your understanding of deep learning and thus to your research and business.

Transition of AI

So, why is it now that deep learning is in the spotlight? You might raise this question, especially if you are familiar with machine learning, because deep learning is not that different to any other machine learning algorithm (don't worry if you don't know this, as we'll go through it later in the book). In fact, we can say that deep learning is the adaptation of neural networks, one of the algorithms of machine learning, which mimics the structure of a human brain. However, what deep learning can achieve is much more significant and different to any other machine learning algorithm, including neural networks. If you see what processes and research deep learning has gone through, you will have a better understanding of deep learning itself. So, let's go through the transition of AI. You can just skim through this while sipping your coffee.

Definition of AI

All of a sudden, AI has become a hot topic in the world; however, as it turns out, actual AI doesn't exist yet. Of course, research is making progress in creating actual AI, but it will take more time to achieve it. Pleased or not, the human brain—which could be called "intelligence"—is structured in an extremely complicated way and you can't easily replicate it.

But wait a moment - we see many advertisements for products with the phrase *by AI* or *using AI* all over them. Are they fraudulent? Actually, they are! Surprised? You might see words like *recommendation system by AI* or *products driven by AI*, but the word *AI* used here doesn't indicate the actual meaning of AI. Strictly speaking, the word *AI* is used with a much broader meaning. The research into AI and the AI techniques accumulated in the past have achieved only some parts of AI, but now people are using the word *AI* for those parts too.

Let's look at a few examples. Roughly divided, there are three different categories recognized as AI in general:

- Simple repetitive machine movements that a human programmed beforehand. For example, high speed processing industrial robots that only process the same set of work.
- Searching or guessing answers to a given assignment following rules set by a human. For example, the iRobot Roomba can clean up along the shape of a room as it can assume the shape of a room by bumping into obstacles.

- Providing an answer to unknown data by finding measurable regularity from the existing data. For example, a product recommendation system based on a user's purchase history or distributing banner ads among ad networks falls under this category.

People use the word AI for these categories and, needless to say, new technology that utilizes deep learning is also called AI. Yet, these technologies are different both in structure and in what they can do. So, which should we specifically call AI? Unfortunately, people have different opinions about that question and the answer cannot be objectively explained. Academically, a term has been set as either **strong AI** or **weak AI** depending on the level that a machine can achieve. However, in this book, to avoid confusion, AI is used to mean *(Not yet achieved) human-like intelligence that is hard to distinguish from the actual human brain*. The field of AI is being drastically developed, and the possibility of AI becoming reality is exponentially higher when driven by deep learning. This field is booming now more than ever in history. How long this boom will continue depends on future research.

AI booms in the past

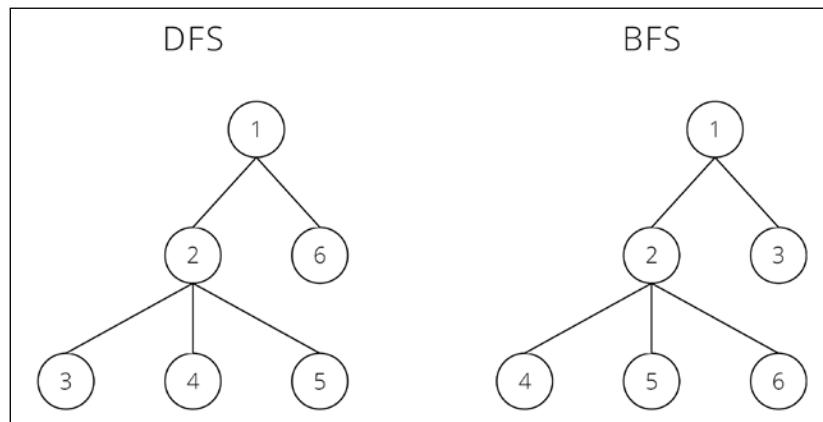
AI suddenly became a hot topic recently; however, this is not the first AI boom. When you look back to the past, research into AI has been conducted for decades and there has been a cycle of being active and inactive. The recent boom is the third boom. Therefore, some people actually think that, at this time, it's just an evanescent boom again.

However, the latest boom has a significant difference from the past booms. Yes, that is deep learning. Deep learning has achieved what the past techniques could not achieve. What is that? Simply put, a machine itself is able to find out the feature quantity from the given data, and learn. With this achievement, we can see the great possibility of AI becoming a reality, because until now a machine couldn't understand a new concept by itself and a human needed to input a certain feature quantity in advance using past techniques created in the AI field.

It doesn't look like a huge difference if you just read this fact, but there's a world of difference. There has been a long path taken before reaching the stage where a machine can measure feature quantity by itself. People were finally able to take a big step forward when a machine could obtain intelligence driven by deep learning. So, what's the big difference between the past techniques and deep learning? Let's briefly look back into the past AI field to get a better sense of the difference.

The first AI boom came in the late 1950s. Back then, the mainstream research and development of a search program was based on fixed rules – needless to say, they were human-defined. The search was, simply put, dividing cases. In this search, if we wanted a machine to perform any process, we had to write out every possible pattern we might need for the process. A machine can calculate much faster than a human can. It doesn't matter how enormous the patterns are, a machine can easily handle them. A machine will keep searching a million times and eventually will find the best answer. However, even if a machine can calculate at high speed, if it is just searching for an answer randomly and blindly it will take a massive amount of time. Yes, don't forget that constraint condition, "time." Therefore, further studies were conducted on how to make the search more efficient. The most popular search methods among the studies were **depth-first search (DFS)** and **breadth-first search (BFS)**.

Out of every possible pattern you can think of, search for the most efficient path and make the best possible choice among them within a realistic time frame. By doing this, you should get the best answer each time. Based on this hypothesis, two searching or traversing algorithms for a tree of graph data structures were developed: DFS and BFS. Both start at the root of a graph or tree, and DFS explores as far as possible along each branch before backtracking, whereas BFS explores the neighbor nodes first before moving to the next level neighbors. Here are some example diagrams that show the difference between DFS and BFS:



These search algorithms could achieve certain results in a specific field, especially fields like Chess and Shogi. This board game field is one of the areas that a machine excels in. If it is given an input of massive amounts of win/lose patterns, past game data, and all the permitted moves of a piece in advance, a machine can evaluate the board position and decide the best possible next move from a very large range of patterns.

For those of you who are interested in this field, let's look into how a machine plays chess in more detail. Let's say a machine makes the first move as "white," and there are 20 possible moves for both "white" and "black" for the next move. Remember the tree-like model in the preceding diagram. From the top of the tree at the start of the game, there are 20 branches underneath as white's next possible move. Under one of these 20 branches, there's another 20 branches underneath as black's next possible movement, and so on. In this case, the tree has $20 \times 20 = 400$ branches for black, depending on how white moves, $400 \times 20 = 8,000$ branches for white, $8,000 \times 20 = 160,000$ branches again for black, and... feel free to calculate this if you like.

A machine generates this tree and evaluates every possible board position from these branches, deciding the best arrangement in a second. How deep it goes (how many levels of the tree it generates and evaluates) is controlled by the speed of the machine. Of course, each different piece's movement should also be considered and embedded in a program, so the chess program is not as simple as previously thought, but we won't go into detail about this in this book. As you can see, it's not surprising that a machine can beat a human at Chess. A machine can evaluate and calculate massive amounts of patterns at the same time, in a much shorter time than a human could. It's not a new story that a machine has beaten a Chess champion; a machine has won a game over a human. Because of stories like this, people expected that AI would become a true story.

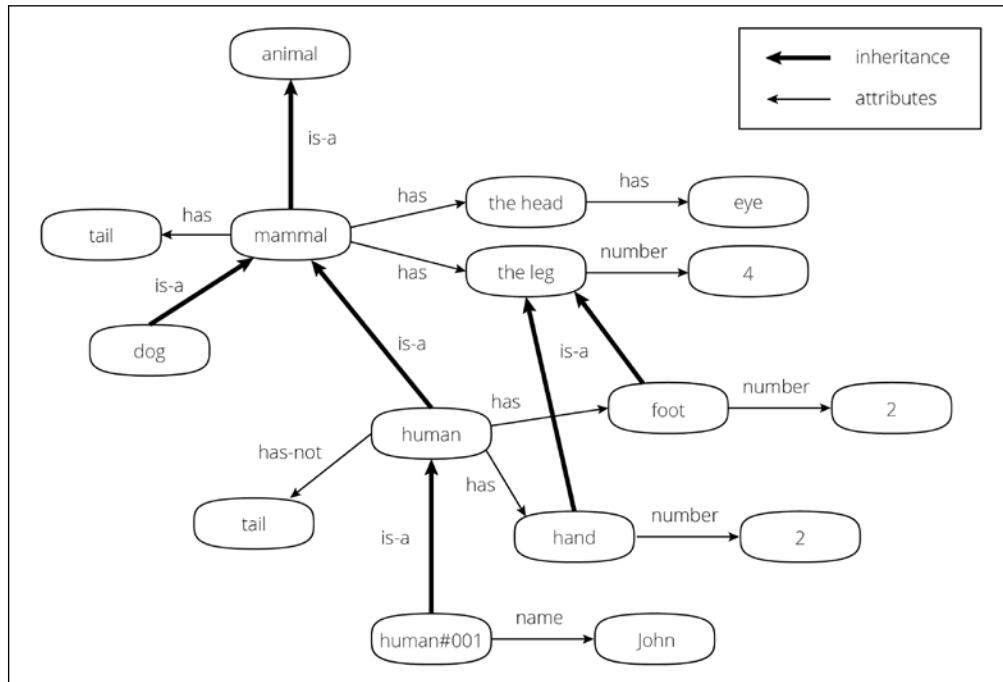
Unfortunately, reality is not that easy. We then found out that there was a big wall in front of us preventing us from applying the search algorithm to reality. Reality is, as you know, complicated. A machine is good at processing things at high speed based on a given set of rules, but it cannot find out how to act and what rules to apply by itself when only a task is given. Humans unconsciously evaluate, discard many things/options that are not related to them, and make a choice from millions of things (patterns) in the real world whenever they act. A machine cannot make these unconscious decisions like humans can. If we create a machine that can appropriately consider a phenomenon that happens in the real world, we can assume two possibilities:

- A machine tries to accomplish its task or purpose without taking into account secondarily occurring incidents and possibilities
- A machine tries to accomplish its task or purpose without taking into account irrelevant incidents and possibilities

Both of these machines would still freeze and be lost in processing before they accomplished their purpose when humans give them a task; in particular, the latter machine would immediately freeze before even taking its first action. This is because these elements are almost infinite and a machine can't sort them out within a realistic time if it tries to think/search these infinite patterns. This issue is recognized as one of the important challenges in the AI field, and it's called the **frame problem**.

A machine can achieve great success in the field of Chess or Shogi because the searching space, the space a machine should be processing within, is limited (set in a certain frame) in advance. You can't write out an enormous amount of patterns, so you can't define what the best solution is. Even if you are forced to limit the number of patterns or to define an optimal solution, you can't get the result within an economical time frame for use due to the enormous amounts of calculation needed. After all, the research at that time would only make a machine follow detailed rules set by a human. As such, although this search method could succeed in a specific area, it is far from achieving actual AI. Therefore, the first AI boom cooled down rapidly with disappointment.

The first AI boom was swept away; however, on the side, the research into AI continued. The second AI boom came in the 1980s. This time, the movement of so-called **Knowledge Representation (KR)** was booming. KR intended to describe knowledge that a machine could easily understand. If all the knowledge in the world was integrated into a machine and a machine could understand this knowledge, it should be able to provide the right answer even if it is given a complex task. Based on this assumption, various methods were developed for designing knowledge for a machine to understand better. For example, the structured forms on a web page—the semantic web—is one example of an approach that tried to design in order for a machine to understand information easier. An example of how the semantic web is described with KR is shown here:



Making a machine gain knowledge is not like a human ordering a machine what to do one-sidedly, but more like a machine being able to respond to what humans ask and then answer. One of the simple examples of how this is applied to the actual world is positive-negative analysis, one of the topics of sentiment analysis. If you input data that defines a tone of positive or negative for every word in a sentence (called "a dictionary") into a machine beforehand, a machine can compare the sentence and the dictionary to find out whether the sentence is positive or negative.

This technique is used for the positive-negative analysis of posts or comments on a social network or blog. If you ask a machine "Is the reaction to this blog post positive or negative?" it analyzes the comments based on its knowledge (dictionary) and replies to you. From the first AI boom, where a machine only followed rules that humans set, the second AI boom showed some progress.

By integrating knowledge into a machine, a machine becomes the almighty. This idea itself is not bad for achieving AI; however, there were two high walls ahead of us in achieving it. First, as you may have noticed, inputting all real-world knowledge requires an almost infinite amount of work now that the Internet is more commonly used and we can obtain enormous amounts of open data from the Web. Back then, it wasn't realistic to collect millions of pieces of data and then analyze and input that knowledge into a machine. Actually, this work of databasing all the world's data has continued and is known as **Cyc** (<http://www.cyc.com/>). Cyc's ultimate purpose is to build an inference engine based on the database of this knowledge, called **knowledge base**. Here is an example of KR using the Cyc project:

```
(#$isa #$BarackObama #$UnitedStatesPresident)
  "Barack Obama belongs to the collection of U.S. presidents."
(#$genls #$Tree-ThePlant #$Plant)
  "All trees are plants."
(#$capitalCity #$Japan #$Tokyo)
  "Tokyo is the capital of Japan."
```

Second, it's not that a machine understands the actual meaning of the knowledge. Even if the knowledge is structured and systemized, a machine understands it as a mark and never understands the concept. After all, the knowledge is input by a human and what a machine does is just compare the data and assume meaning based on the dictionary. For example, if you know the concept of "apple" and "green" and are taught "green apple = apple + green", then you can understand that "a green apple is a green colored apple" at first sight, whereas a machine can't. This is called the **symbol grounding problem** and is considered one of the biggest problems in the AI field, as well as the frame problem.

The idea was not bad – it did improve AI – however, this approach won't achieve AI in reality as it's not able to create AI. Thus, the second AI boom cooled down imperceptibly, and with a loss of expectation from AI, the number of people who talked about AI decreased. When it came to the question of "Are we really able to achieve AI?" the number of people who answered "no" increased gradually.

Machine learning evolves

While people had a hard time trying to establish a method to achieve AI, a completely different approach had steadily built a generic technology . That approach is called machine learning. You should have heard the name if you have touched on data mining even a little. Machine learning is a strong tool compared to past AI approaches, which simply searched or assumed based on the knowledge given by a human, as mentioned earlier in the chapter, so machine learning is very advanced. Until machine learning, a machine could only search for an answer from the data that had already been inputted. The focus was on how fast a machine could pull out knowledge related to a question from its saved knowledge. Hence, a machine can quickly reply to a question it already knows, but gets stuck when it faces questions it doesn't know.

On the other hand, in machine learning, a machine is literally learning. A machine can cope with unknown questions based on the knowledge it has learned. So, how was a machine able to learn, you ask? What exactly is *learning* here? Simply put, learning is when a machine can divide a problem into "yes" or "no." We'll go through more detail on this in the next chapter, but for now we can say that machine learning is a method of pattern recognition.

We could say that, ultimately, every question in the world can be replaced with a question that can be answered with yes or no. For example, the question "What color do you like?" can be considered almost the same as asking "Do you like red? Do you like green? Do you like blue? Do you like yellow?..." In machine learning, using the ability to calculate and the capacity to process at high speed as a weapon, a machine utilizes a substantial amount of training data, replaces complex questions with yes/no questions, and finds out the regularity with which data is yes, and which data is no (in other words, it learns). Then, with that learning, a machine assumes whether the newly-given data is yes or no and provides an answer. To sum up, machine learning can give an answer by recognizing and sorting out patterns from the data provided and then classifying that data into the possible appropriate pattern (predicting) when it faces unknown data as a question.

In fact, this approach is not doing something especially difficult. Humans also unconsciously classify data into patterns. For example, if you meet a man/woman who's perfectly your type at a party, you might be desperate to know whether the man/woman in front of you has similar feelings towards you. In your head, you would compare his/her way of talking, looks, expressions, or gestures to past experience (that is, data) and assume whether you will go on a date! This is the same as a presumption based on pattern recognition.

Machine learning is a method that can process this pattern recognition not by humans but by a machine in a mechanical manner. So, how can a machine recognize patterns and classify them? The standard of classification by machine learning is a presumption based on a numerical formula called the **probabilistic statistical model**. This approach has been studied based on various mathematical models.

Learning, in other words, is tuning the parameters of a model and, once the learning is done, building a model with one adjusted parameter. The machine then categorizes unknown data into the most possible pattern (that is, the pattern that fits best). Categorizing data mathematically has great merit. While it is almost impossible for a human to process multi-dimensional data or multiple-patterned data, machine learning can process the categorization with almost the same numerical formulas. A machine just needs to add a vector or the number of dimensions of a matrix. (Internally, when it classifies multi-dimensions, it's not done by a classified line or a classified curve but by a hyperplane.)

Until this approach was developed, machines were helpless in terms of responding to unknown data without a human's help, but with machine learning machines became capable of responding to data that humans can't process. Researchers were excited about the possibilities of machine learning and jumped on the opportunity to start working on improving the method. The concept of machine learning itself has a long history, but researchers couldn't do much research and prove the usefulness of machine learning due to a lack of available data. Recently, however, many open-source data have become available online and researchers can easily experiment with their algorithms using the data. Then, the third AI boom came about like this. The environment surrounding machine learning also gave its progress a boost. Machine learning needs a massive amount of data before it can correctly recognize patterns. In addition, it needs to have the capability to process data. The more data and types of patterns it handles, the more the amount of data and the number of calculations increases. Hence, obviously, past technology wouldn't have been able to deal with machine learning.

However, time is progressing, not to mention that the processing capability of machines has improved. In addition, the web has developed and the Internet is spreading all over the world, so open data has increased. With this development, everyone can handle data mining only if they pull data from the web. The environment is set for everyone to casually study machine learning. The web is a treasure box of text-data. By making good use of this text-data in the field of machine learning, we are seeing great development, especially with statistical natural language processing. Machine learning has also made outstanding achievements in the field of image recognition and voice recognition, and researchers have been working on finding the method with the best precision.

Machine learning is utilized in various parts of the business world as well. In the field of natural language processing, the prediction conversion in the **input method editor (IME)** could soon be on your mind. The fields of image recognition, voice recognition, image search, and voice search in the search engine are good examples. Of course, it's not limited to these fields. It is also applied to a wide range of fields from marketing targeting, such as the sales prediction of specific products or the optimization of advertisements, or designing store shelf or space planning based on predicting human behavior, to predicting the movements of the financial market. It can be said that the most used method of data mining in the business world is now machine learning. Yes, machine learning is that powerful. At present, if you hear the word "AI," it's usually the case that the word simply indicates a process done by machine learning.

What even machine learning cannot do

A machine learns by gathering data and predicting an answer. Indeed, machine learning is very useful. Thanks to machine learning, questions that are difficult for a human to solve within a realistic time frame (such as using a 100-dimensional hyperplane for categorization!) are easy for a machine. Recently, "big data" has been used as a buzzword and, by the way, analyzing this big data is mainly done using machine learning too.

Unfortunately, however, even machine learning cannot make AI. From the perspective of "can it actually achieve AI?" machine learning has a big weak point. There is one big difference in the process of learning between machine learning and human learning. You might have noticed the difference, but let's see. Machine learning is the technique of pattern classification and prediction based on input data. If so, what exactly is that input data? Can it use any data? Of course... it can't. It's obvious that it can't correctly predict based on irrelevant data. For a machine to learn correctly, it needs to have appropriate data, but then a problem occurs. A machine is not able to sort out what is appropriate data and what is not. Only if it has the right data can machine learning find a pattern. No matter how easy or difficult a question is, it's humans that need to find the right data.

Let's think about this question: "Is the object in front of you a human or a cat?" For a human, the answer is all too obvious. It's not difficult at all to distinguish them. Now, let's do the same thing with machine learning. First, we need to prepare the format that a machine can read, in other words, we need to prepare the image data of a human and a cat respectively. This isn't anything special. The problem is the next step. You probably just want to use the image data for inputting, but this doesn't work. As mentioned earlier, a machine can't find out what to learn from data by itself. Things a machine should learn need to be processed from the original image data and created by a human. Let's say, in this example, we might need to use data that can define the differences such as face colors, facial part position, the facial outlines of a human and a cat, and so on, as input data. These values, given as inputs that humans need to find out, are called the features.

Machine learning can't do feature engineering. This is the weakest point of machine learning. Features are, namely, variables in the model of machine learning. As this value shows the feature of the object quantitatively, a machine can appropriately handle pattern recognition. In other words, how you set the value of identities will make a huge difference in terms of the precision of prediction. Potentially, there are two types of limitations with machine learning:

- An algorithm can only work well on data with the assumption of the training data - with data that has different distribution. In many cases, the learned model does not generalize well.
- Even the well-trained model lacks the ability to make a smart meta-decision. Therefore, in most cases, machine learning can be very successful in a very narrow direction.

Let's look at a simple example so that you can easily imagine how identities have a big influence on the prediction precision of a model. Imagine there is a corporation that wants to promote a package of asset management based on the amount of assets. The corporation would like to recommend an appropriate product, but as it can't ask a personal question, it needs to predict how many assets a customer might have and prepare in advance. In this case, what type of potential customers shall we consider as an identity? We can assume many factors such as their height, weight, age, address, and so on as an identity, but clearly age or residence seem more relevant than height or weight. You probably won't get a good result if you try machine learning based on height or weight, as it predicts based on irrelevant data, meaning it's just a random prediction.

As such, machine learning can provide an appropriate answer against the question only after the machine reads an appropriate identity. But, unfortunately, the machine can't judge what the appropriate identity is, and the precision of machine learning depends on this feature engineering!

Machine learning has various methods, but the problem of being unable to do feature engineering is seen across all of these. Various methods have been developed and people compete against their precision rates, but after we have achieved precision to a certain extent, people decide whether a method of machine learning is good or bad based on how great a feature they can find. This is no longer a difference in algorithms, but more like a human's intuition or taste, or the fine-tuning of parameters, and this can't be said to be innovative at all. Various methods have been developed, but after all, the hardest thing is to think of the best identity and a human has to do that part anyway.

Things dividing a machine and human

We have gone through three problems: the frame problem, the symbol grounding problem, and feature engineering. None of these problems concern humans at all. So, why can't a machine handle these problems? Let's review the three problems again. If you think about it carefully, you will find that all three problems confront the same issue in the end:

- The frame problem is that a machine can't recognize what knowledge it should use when it is assigned a task
- The symbol grounding problem is that a machine can't understand a concept that puts knowledge together because it only recognizes knowledge as a mark
- The problem of feature engineering in machine learning is that a machine can't find out what the feature is for objects

These problems can be solved only if a machine can sort out *which feature of things/phenomena it should focus on and what information it should use*. After all, this is the biggest difference between a machine and a human. Every object in this world has its own inherent features. A human is good at catching these features. Is this by experience or by instinct? Anyhow, humans know features, and, based on these features, humans can understand a thing as a "concept."

Now, let's briefly explain what a concept is. First of all, as a premise, take into account that every single thing in this world is constituted of a set of symbol representations and the symbols' content. For example, if you don't know the word "cat" and see a cat when you walk down a street, does it mean you can't recognize a cat? No, this is not true. You know it exists, and if you see another cat just after, you will understand it as "a similar thing to what I saw earlier." Later, you are told "That is called a cat", or you look it up for yourself, and for the first time you can connect the existence and the word.

This word, cat, is the **symbol representation** and the concept that you recognize as a cat is the **symbol content**. You can see these are two sides of the same coin. (Interestingly, there is no necessity between these two sides. There is no necessity to write cat as C-A-T or to pronounce it as such. Even so, in our system of understanding, these are considered to be inevitable. If people hear "cat", we all imagine the same thing.) The concept is, namely, symbol content. These two concepts have terms. The former is called **signifiant** and the latter is called **signifié**, and a set of these two as a pair is called **signe**. (These words are French. You can say signifier, signified, and sign in English, respectively.) We could say what divides a machine and human is whether it can get signifié by itself or not.

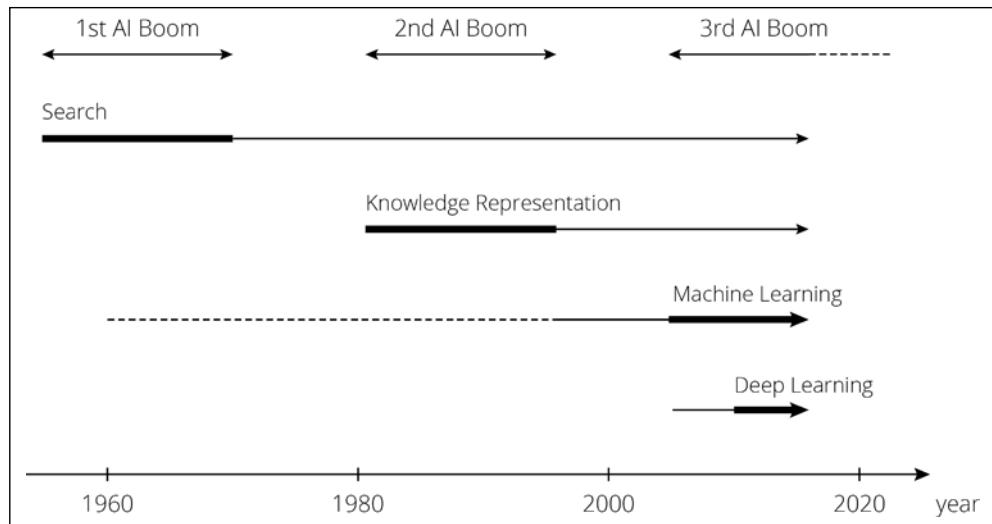
What would happen if a machine could find the notable feature from given data? As for the frame problem, if a machine could extract the notable feature from the given data and perform the knowledge representation, it wouldn't have the problem of freezing when thinking of how to pick up the necessary knowledge anymore. In terms of the symbol grounding problem, if a machine could find the feature by itself and understand the concept from the feature, it could understand the inputted symbol.

Needless to say, the feature engineering problem in machine learning would also be solved. If a machine can obtain appropriate knowledge by itself following a situation or a purpose, and not use knowledge from a fixed situation, we can solve the various problems we have been facing in achieving AI. Now, the method that a machine can use to find the important feature value from the given data is close to being accomplished. Yes, finally, this is deep learning. In the next section, I'll explain this deep learning, which is considered to be the biggest breakthrough in the more-than-50 years of AI history.

AI and deep learning

Machine learning, the spark for the third AI boom, is very useful and powerful as a data mining method; however, even with this approach of machine learning, it appeared that the way towards achieving AI was closed. Finding features is a human's role, and here there is a big wall preventing machine learning from reaching AI. It looked like the third AI boom would come to an end as well. However, surprisingly enough, the boom never ended, and on the contrary a new wave has risen. What triggered this wave is deep learning.

With the advent of deep learning, at least in the fields of image recognition and voice recognition, a machine became able to obtain "what should it decide to be a feature value" from the inputted data by itself rather than from a human. A machine that could only handle a symbol as a symbol notation has become able to obtain concepts.



Correspondence diagram between AI booms up to now and the research fields of AI

The first time deep learning appeared was actually quite a while ago, back in 2006. Professor Hinton at Toronto University in Canada, and others, published a paper (<https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf>). In this paper, a method called **deep belief nets (DBN)** was presented, which is an expansion of neural networks, a method of machine learning. DBN was tested using the **MNIST** database, the standard database for comparing the precision and accuracy of each image recognition method. This database includes 70,000 28 × 28 pixel hand-written character image data of numbers from 0 to 9 (60,000 are for training and 10,000 are for testing).

Then, they constructed a prediction model based on the training data and measured its accuracy based on whether a machine could correctly answer which number from 0 to 9 was written in the test case. Although this paper presented a result with considerably higher precision than a conventional method, it didn't attract much attention at the time, maybe because it was compared with another general method of machine learning.

Then, a while later in 2012, the whole AI research world was shocked by one method. At the world competition for image recognition, **Imagenet Large Scale Visual Recognition Challenge (ILSVRC)**, a method using deep learning called SuperVision (strictly, that's the name of the team), which was developed by Professor Hinton and others from Toronto University, won the competition. It far surpassed the other competitors, with formidable precision. At this competition, the task was assigned for a machine to automatically distinguish whether an image was a cat, a dog, a bird, a car, a boat, and so on. 10 million images were provided as learning data and 150,000 images were used for the test. In this test, each method competes to return the lowest error rate (that is, the highest accuracy rate).

Let's look at the following table that shows the result of the competition:

Rank	Team name	Error
1	SuperVision	0.15315
2	SuperVision	0.16422
3	ISI	0.26172
4	ISI	0.26602
5	ISI	0.26646
6	ISI	0.26952
7	OXFORD_VGG	0.26979
8	XRCE/INRIA	0.27058

You can see that the difference in the error rate between SuperVision and the second position, ISI, is more than 10%. After the second position, it's just a competition within 0.1%. Now you know how greatly SuperVision outshone the others with precision rates. Moreover, surprisingly, it was the first time SuperVision joined this ILSVRC, in other words, image recognition is not their usual field. Until SuperVision (deep learning) appeared, the normal approach for the field of image recognition was machine learning. And, as mentioned earlier, a feature value necessary to use machine learning had to be set or designed by humans. They reiterated design features based on human intuition and experiences and fine-tuning parameters over and over, which, in the end, contributed to improving precision by just 0.1%. The main issue of the research and the competition before deep learning evolved was who was able to invent good feature engineering. Therefore, researchers must have been surprised when deep learning suddenly showed up out of the blue.

There is one other major event that spread deep learning across the world. That event happened in 2012, the same year the world was shocked by SuperVision at ILSVRC, when Google announced that a machine could automatically detect a cat using YouTube videos as learning data from the deep learning algorithm that Google proposed. The details of this algorithm are explained at <http://googleblog.blogspot.com/2012/06/using-large-scale-brain-simulations-for.html>. This algorithm extracted 10 million images from YouTube videos and used them as input data. Now, remember, in machine learning, a human has to detect feature values from images and process data. On the other hand, in deep learning, original images can be used for inputs as they are. This shows that a machine itself comes to find features automatically from training data. In this research, a machine learned the concept of a cat. (Only this cat story is famous, but the research was also done with human images and it went well. A machine learned what a human is!) The following image introduced in the research illustrates the characteristics of what deep learning thinks a cat is, after being trained using still frames from unlabeled YouTube videos:



These two big events impressed us with deep learning and triggered the boom that is still accelerating now.

Following the development of the method that can recognize a cat, Google conducted another experiment for a machine to draw a picture by utilizing deep learning. This method is called **Inceptionism** (<http://googleresearch.blogspot.ch/2015/06/inceptionism-going-deeper-into-neural.html>). As written in the article, in this method, the network is asked:

"Whatever you see there, I want more of it!". This creates a feedback loop: if a cloud looks a little bit like a bird, the network will make it look more like a bird. This in turn will make the network recognize the bird even more strongly on the next pass and so forth, until a highly detailed bird appears, seemingly out of nowhere.

While the use of neural networks in machine learning is a method usually used to detect patterns to be able to specify an image, what Inceptionism does is the opposite. As you can see from the following examples of Inceptionism, these paintings look odd and like the world of a nightmare:



Or rather, they could look artistic. The tool that enables anyone to try Inceptionism is open to the public on GitHub and is named Deep Dream (<https://github.com/google/deeppdram>). Example implementations are available on that page. You can try them if you can write Python codes.

Well, nothing stops deep learning gaining momentum, but there are still questions, such as what exactly is innovative about deep learning? What special function dramatically increased this precision? Surprisingly, actually, there isn't a lot of difference for deep learning in algorithms. As mentioned briefly, deep learning is an application of neural networks, which is an algorithm of machine learning that imitates the structure of a human brain; nevertheless, a device adopted it and changed everything. The representatives are **pretraining** and **dropout** (with an activation function). These are also keywords for implementation, so please remember them.

To begin with, what does the *deep* in deep learning indicate? As you probably know, the human brain is a circuit structure, and that structure is really complicated. It is made up of an intricate circuit piled up in many layers. On the other hand, when the neural network algorithm first appeared its structure was quite simple. It was a simplified structure of the human brain and the network only had a few layers. Hence, the patterns it could recognize were extremely limited. So, everyone wondered "Can we just accumulate networks like the human brain and make its implementation complex?" Of course, though this approach had already been tried. Unfortunately, as a result, the precision was actually lower than if we had just piled up the networks. Indeed, we faced various issues that didn't occur with a simple network. Why was this? Well, in a human brain, a signal runs into a different part of the circuit depending on what you see. Based on the patterns that differ based on which part of the circuit is stimulated, you can distinguish various things.

To reproduce this mechanism, the neural network algorithm substitutes the linkage of the network by weighting with numbers. This is a great way to do it, but soon a problem occurs. If a network is simple, weights are properly allocated from the learning data and the network can recognize and classify patterns well. However, once a network gets complicated, the linkage becomes too dense and it is difficult to make a difference in the weights. In short, it cannot divide into patterns properly. Also, in a neural network, the network can make a proper model by adopting a mechanism that feeds back errors that occurred during training to the whole network. Again, if the network is simple the feedback can be reflected properly, but if the network has many layers a problem occurs in which the error disappears before it's reflected to the whole network – just imagine if that error was stretched out and diluted.

The intention that things would go well if the network was built with a complicated structure ended in disappointing failure. The concept of the algorithm itself was splendid but it couldn't be called a good algorithm by any standards; that was the world's understanding. While deep learning succeeded in making a network multi-layered, that is, making a network "deep," the key to success was to make each layer learn in stages. The previous algorithm treated the whole multi-layered network as one gigantic neural network and made it learn as one, which caused the problems mentioned earlier.

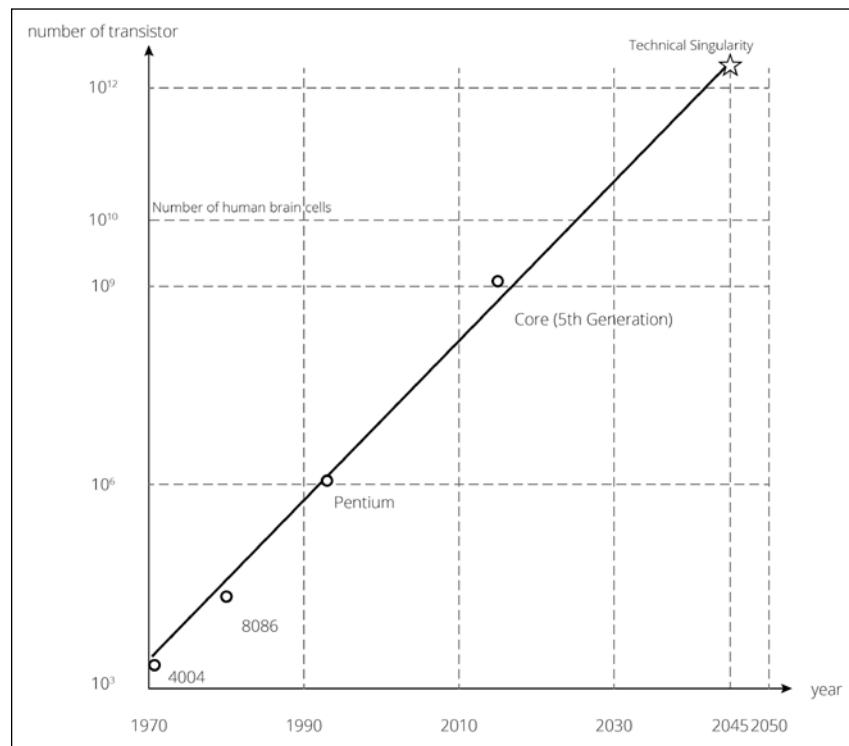
Hence, deep learning took the approach of making each layer learn in advance. This is literally known as pretraining. In pretraining, learning starts from the lower-dimension layer in order. Then, the data that is learned in the lower layer is treated as input data for the next layer. This way, machines become able to take a step by learning a feature of a low layer at the low-grade layer and gradually learning a feature of a higher grade. For example, when learning what a cat is, the first layer is an outline, the next layer is the shape of its eyes and nose, the next layer is a picture of a face, the next layers is the detail of a face, and so on. Similarly, it can be said that humans take the same learning steps as they catch the whole picture first and see the detailed features later. As each layer learns in stages, the feedback for an error of learning can also be done properly in each layer. This leads to an improvement in precision. There is also a device for each respective approach to each layer's learning, but this will be introduced later on.

We have also addressed the fact that the network became too dense. The method that prevents this density problem is called the **dropout**. Networks with the dropout learn by cutting some linkages randomly within the units of networks. The dropout physically makes the network sparse. Which linkage is cut is random, so a different network is formed at each learning step. Just by looking, you might doubt that this will work, but it greatly contributes to improving the precision and as a result it increases the robustness of the network. The circuit of the human brain also has different places in which to react or not depending on the subject it sees. The dropout seems to be able to successfully imitate this mechanism. By embedding the dropout in the algorithm, the adjustment of the network weight was done well.

Deep learning has seen great success in various fields; however, of course deep learning has a demerit too. As is shown in the name "deep learning," the learning in this method is very deep. This means the steps to complete the learning take a long time. The amount of calculation in this process tends to be enormous. In fact, the previously mentioned learning of the recognition of a cat by Google took three days to be processed with 1,000 computers. Conversely, although the idea of deep learning itself could be conceived using past techniques, it couldn't be implemented. The method wouldn't appear if you couldn't easily use a machine that has a large-scale processing capacity with massive data.

As we keep saying, deep learning is just the first step for a machine to obtain human-like knowledge. Nobody knows what kind of innovation will happen in the future. Yet we can predict to what extent a computer's performance will be improved in the future. To predict this, Moore's law is used. The performance of an integrated circuit that supports the progress of a computer is indicated by the loaded number of transistors. Moore's law shows the number, and the number of transistors is said to double every one and a half years. In fact, the number of transistors in the CPU of a computer has been increasing following Moore's law. Compared to the world's first micro-processor, the Intel® 4004 processor, which had 1×10^3 (one thousand) transistors, the recent 2015 version, the 5th Generation Intel® Core™ Processor, has 1×10^9 (one billion)! If this technique keeps improving at this pace, the number of transistors will exceed ten billion, which is more than the number of cells in the human cerebrum.

Based on Moore's law, further in the future in 2045, it is said that we will reach a critical point called **Technical Singularity** where humans will be able to do technology forecasting. By that time, a machine is expected to be able to produce self-recursive intelligence. In other words, in about 30 years, AI will be ready. What will the world be like then...



History of Moore's law

The number of transistors loaded in the processor invented by Intel has been increasing smoothly following Moore's law.

The world famous professor Stephen Hawking answered in an interview by the BBC (<http://www.bbc.com/news/technology-30290540>):

"The development of full artificial intelligence could spell the end of the human race."

Will deep learning become a black magic? Indeed, the progress of technology has sometimes caused tragedy. Achieving AI is still far in the future, yet we should be careful when working on deep learning.

Summary

In this chapter, you learned how techniques in the field of AI have evolved into deep learning. We now know that there were two booms in AI and that we are now in the third boom. Searching and traversing algorithms were developed in the first boom, such as DFS and BFS. Then, the study focused on how knowledge could be represented with symbols that a machine could easily understand in the second boom.

Although these booms had faded away, techniques developed during those times built up much useful knowledge of AI fields. The third boom spread out with machine learning algorithms in the beginning with those of pattern recognition and classification based on probabilistic statistical models. With machine learning, we've made great progress in various fields, but this is not enough to realize true AI because we need to tell a machine what the features of objects to be classified are. The technique required for machine learning is called feature engineering. Then, deep learning came out, based on one machine learning algorithm - namely, neural networks. A machine can automatically learn what the features of objects are with deep learning, and thus deep learning is recognized as a very innovative technique. Studies of deep learning are becoming more and more active, and every day new technologies are invented. Some of the latest technologies are introduced in the last chapter of this book, *Chapter 8, What's Next?*, for reference.

Deep learning is often thought to be very complicated, but the truth is it's not. As mentioned, deep learning is the evolving technique of machine learning, and deep learning itself is very simple yet elegant. We'll look at more details of machine learning algorithms in the next chapter. With a great understanding of machine learning, you will easily acquire the essence of deep learning.

2

Algorithms for Machine Learning – Preparing for Deep Learning

In the previous chapter, you read through how deep learning has been developed by looking back through the history of AI. As you should have noticed, machine learning and deep learning are inseparable. Indeed, you learned that deep learning is the developed method of machine learning algorithms.

In this chapter, as a pre-exercise to understand deep learning well, you will see the mode details of machine learning, and in particular, you will learn the actual code for the method of machine learning, which is closely related to deep learning.

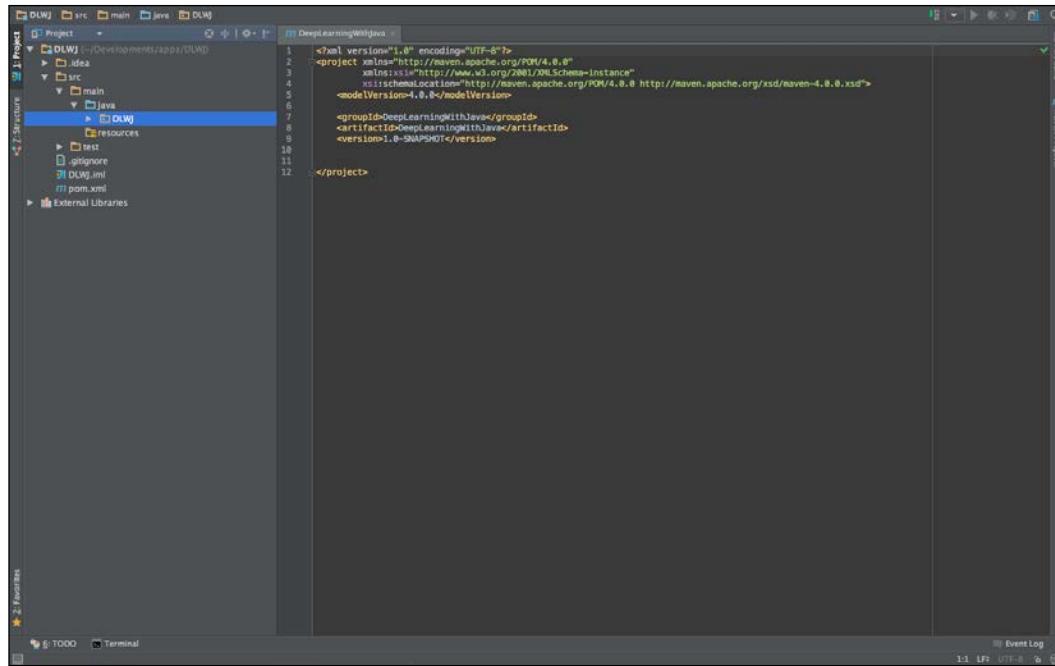
In this chapter, we will cover the following topics:

- The core concepts of machine learning
- An overview of popular machine learning algorithms, especially focusing on neural networks
- Theories and implementations of machine learning algorithms related to deep learning: perceptrons, logistic regression, and multi-layer perceptrons

Getting started

We will insert the source code of machine learning and deep learning with Java from this chapter. The version of JDK used in the code is 1.8, hence Java versions greater than 8 are required. Also, IntelliJ IDEA 14.1 is used for the IDE. We will use the external library from *Chapter 5, Exploring Java Deep Learning Libraries – DL4J, ND4J, and More*, so we are starting with a new Maven project.

The root package name of the code used in this book is `DLWJ`, the initials of *Deep Learning with Java*, and we will add a new package or a class under `DLWJ` as required. Please refer to the screenshot below, which shows the screen immediately after the new project is made:

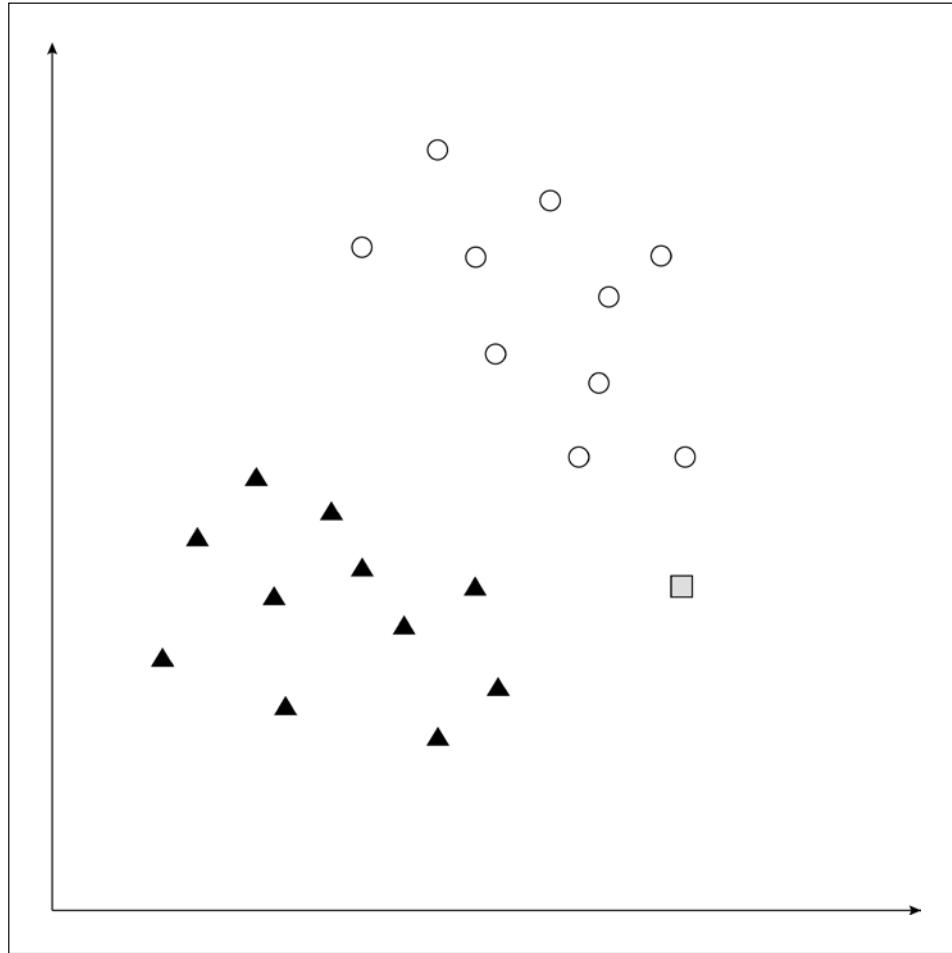


There will be some names of variables and methods in the code that don't follow the Java coding standard. This is to improve your understanding together with some characters in the formulas to increase readability. Please bear this in mind in advance.

The need for training in machine learning

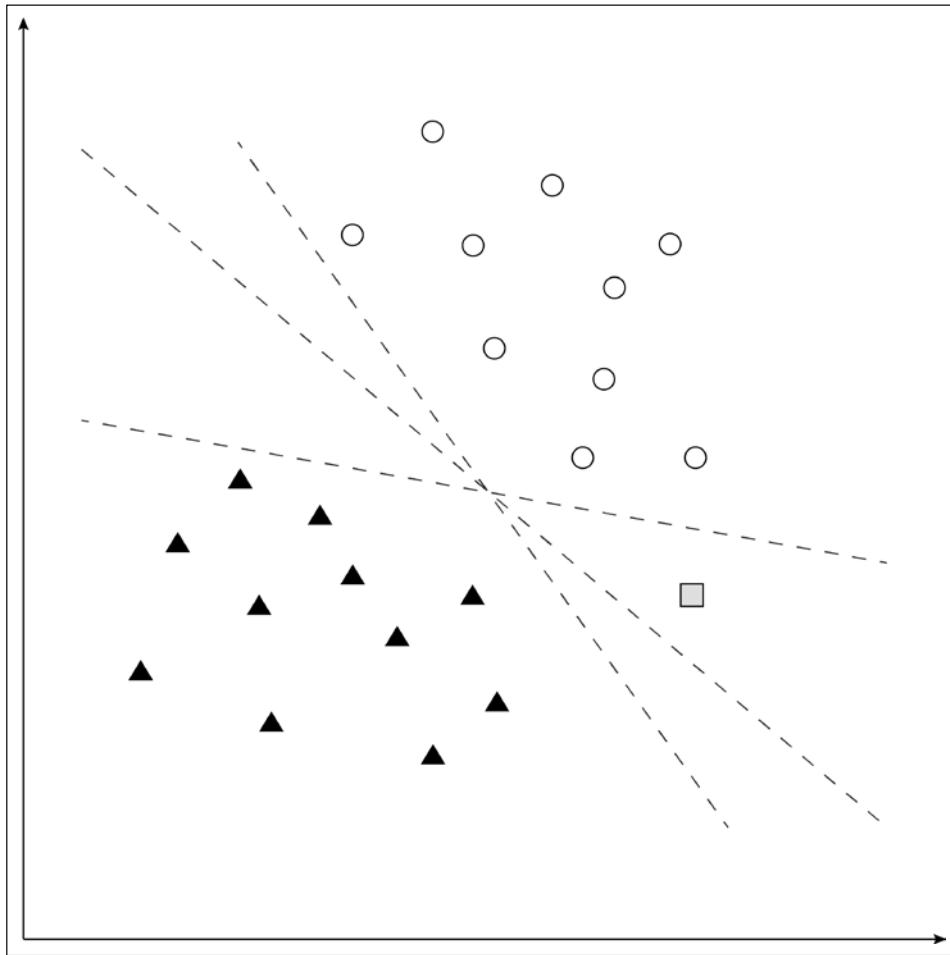
You have already seen that machine learning is a method of pattern recognition. Machine learning reaches an answer by recognizing and sorting out patterns from the given learning data. It may seem easy when you just look at the sentence, but the fact is that it takes quite a long time for machine learning to sort out unknown data, in other words, to build the appropriate model. Why is that? Is it that difficult to just sort out? Does it even bother to have a "learning" phase in between?

The answer is, of course, yes. It is extremely difficult to sort out data appropriately. The more complicated a problem becomes, the more it becomes impossible to perfectly classify data. This is because there are almost infinite patterns of categorization when you simply say "pattern classifier." Let's look at a very simple example in the following graph:



There are two types of data, circles and triangles, and the unknown data, the square. You don't know which group the square belongs to in the two-dimensional coordinate space, so the task is to find out which group the square belongs to.

You might instantly know that there seems to be a boundary that separates two data types. And if you decide where to set this boundary, it looks like you should be able to find out to which group the square belongs. Well then, let's decide the boundary. In reality, however, it is not so easy to clearly define this boundary. If you want to set a boundary, there are various lines to consider, as you can see in the following figure:



Additionally, depending on the placement of the boundary, you can see that the square might be allocated to a different group or pattern. Furthermore, it is also possible to consider that the boundary might be a nonlinear boundary.

In machine learning, what a machine does in training is choose the most likely boundary from these possible patterns. It will automatically learn how to sort out patterns when processing massive amounts of training data one after another. In other words, it adjusts the parameters of a mathematical model and eventually decides the boundary. The boundary decided by machine learning is called the **decision boundary** and is not necessarily a linear or nonlinear boundary. A decision boundary can also be a hyperplane if it classifies the data best. The more complicated the distribution of the data is, the more likely it is that the decision boundary would be nonlinear boundary or a hyperplane. A typical case is the multi-dimensional classification problem. We have already faced such difficulty by just setting a boundary in this simple problem, so it's not hard to imagine that it would be very time-consuming to solve a more complicated problem.

Supervised and unsupervised learning

In the previous section, we saw that there could be millions of boundaries even for a simple classification problem, but it is difficult to say which one of them is the most appropriate. This is because, even if we could properly sort out patterns in the known data, it doesn't mean that unknown data can also be classified in the same pattern. However, you can increase the percentage of correct pattern categorization. Each method of machine learning sets a standard to perform a better pattern classifier and decides the most possible boundary – the decision boundary – to increase the percentage. These standards are, of course, greatly varied in each method. In this section, we'll see what all the approaches we can take are.

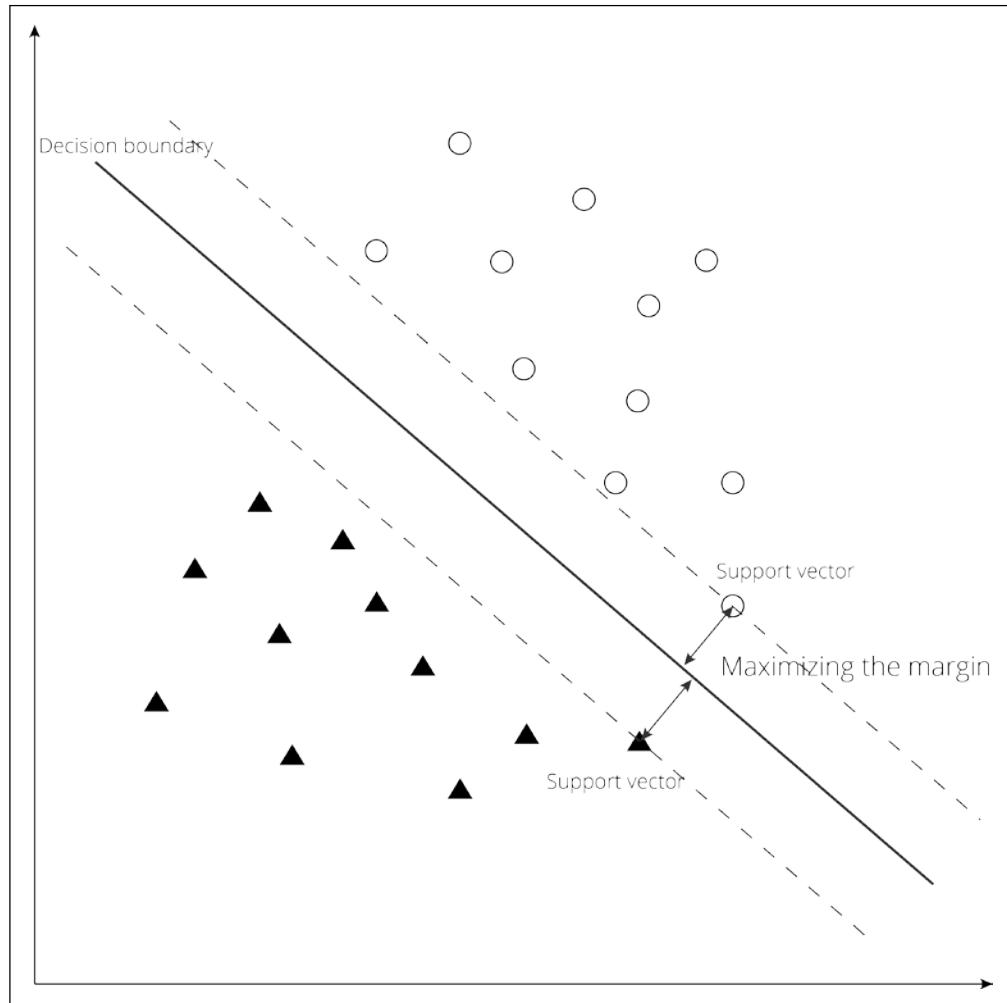
First, machine learning can be broadly classified into **supervised learning** and **unsupervised learning**. The difference between these two categories is the dataset for machine learning is labeled data or unlabeled data. With supervised learning, a machine uses labeled data, the combination of input data and output data, and mentions which pattern each type of data is to be classified as. When a machine is given unknown data, it will derive what pattern can be applied and classify the data based on labeled data, that is, the past correct answers. As an example, in the field of image recognition, when you input some images into a machine, if you prepare and provide a certain number of images of a cat, labeled `cat`, and the same number of images of a human, labeled `human`, for a machine to learn, it can judge by itself which group out of cat or human (or none of them) that an image belongs to. Of course, just deciding whether the image is a cat or a human doesn't really provide a practical use, but if you apply the same approach to other fields, you can create a system that can automatically tag who is who in a photo uploaded on social media. As you can now see, in supervised training, the learning proceeds when a machine is provided with the correct data prepared by humans in advance.

On the other hand, with unsupervised learning, a machine uses unlabeled data. In this case, only input data is given. Then, what the machine learns is patterns and rules that the dataset includes and contains. The purpose of unsupervised learning is to grasp the structure of the data. It can include a process called **clustering**, which classifies a data constellation in each group that has a common character, or the process of extracting the correlation rule. For example, imagine there is data relating to a user's age, sex, and purchase trend for an online shopping website. Then, you might find out that the tastes of men in their 20s and women in their 40s are close, and you want to make use of this trend to improve your product marketing. We have a famous story here—it was discovered from unsupervised training that a large number of people buy beer and diapers at the same time.

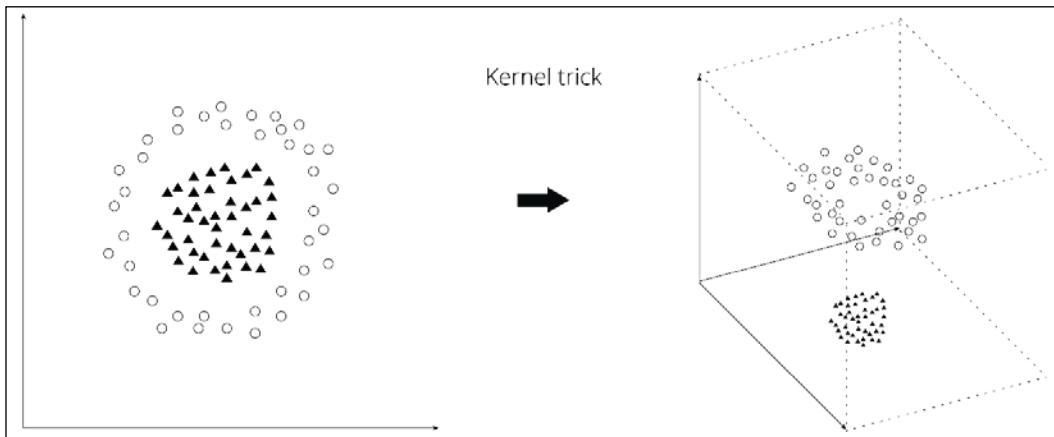
You now know there are big differences between supervised learning and unsupervised learning, but that's not all. There are also different learning methods and algorithms for each learning method, respectively. Let's look at some representative examples in the following section.

Support Vector Machine (SVM)

You could say that SVM is the most popular supervised training method in machine learning. The method is still used for broad fields in the data mining industry. With SVM, data from each category located the closest to other categories is marked as the standard, and the decision boundary is determined using the standard so that the sum of the Euclidean distance from each marked data and the boundary is maximized. This marked data is called **support vectors**. Simply put, SVM sets the decision boundary in the middle point where the distance from every pattern is maximized. Therefore, what SVM does in its algorithm is known as **maximizing the margin**. The following is the figure of the concept of SVM:



If you only hear this statement, you might think "is that it?" but what makes SVM the most valuable is a math technique: the kernel trick, or the kernel method. This technique takes the data that seems impossible to be classified linearly in the original dimension and intentionally maps it to a higher dimensional space so that it can be classified linearly without any difficulties. Take a look at the following figure so you can understand how the kernel trick works:



We have two types of data, represented by circles and triangles, and it is obvious that it would be impossible to separate both data types linearly in a two-dimensional space. However, as you can see in the preceding figure, by applying the kernel function to the data (strictly speaking, the feature vectors of training data), whole data is transformed into a higher dimensional space, that is, a three-dimensional space, and it is possible to separate them with a two-dimensional plane.

While SVM is useful and elegant, it has one demerit. Since it maps the data into a higher dimension, the number of calculations often increases, so it tends to take more time in processing as the calculation gets more complicated.

Hidden Markov Model (HMM)

HMM is an unsupervised training method that assumes data follows the **Markov process**. The Markov process is a stochastic process in which a future condition is decided solely on the present value and is not related to the past condition. HMM is used to predict which state the observation comes from when only one observation is visible.

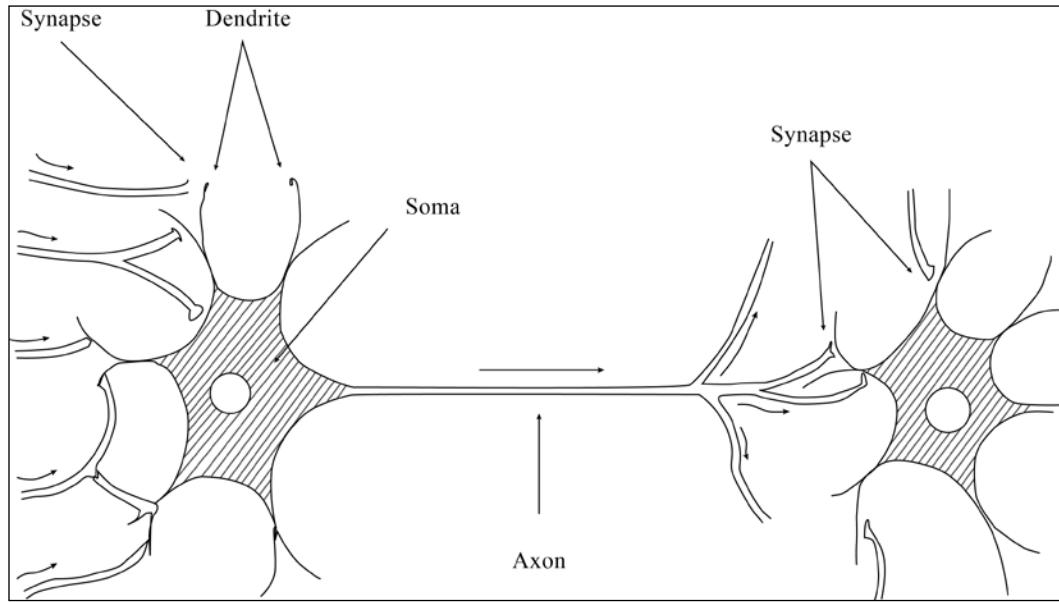
The previous explanation alone may not help you fully understand how HMM works, so let's look at an example. HMM is often used to analyze a base sequence. You may know that a base sequence consists of four nucleotides, for example, A, T, G, C, and the sequence is actually a string of these nucleotides. You won't get anything just by looking through the string, but you do have to analyze which part is related to which gene. Let's say that if any base sequence is lined up randomly, then each of the four characters should be output by one-quarter when you cut out any part of the base sequence.

However, if there is a regularity, for example, where C tends to come next to G or the combination of ATT shows up frequently, then the probability of each character being output would vary accordingly. This regularity is the probability model and if the probability of being output relies only on an immediately preceding base, you can find out genetic information (= state) from a base sequence (= observation) using HMM.

Other than these bioinformatic fields, HMM is often used in fields where time sequence patterns, such as syntax analysis of **natural language processing (NLP)** or sound signal processing, are needed. We don't explore HMM deeper here because its algorithm is less related to deep learning, but you can reference a very famous book, *Foundations of statistical natural language processing*, from MIT Press if you are interested.

Neural networks

Neural networks are a little different to the machine learning algorithms. While other methods of machine learning take an approach based on probability or statistics, neural networks are algorithms that imitate the structure of a human brain. A human brain is made of a neuron network. Take a look at the following figure to get an idea of this:



One neuron is linked to the network through another neuron and takes electrical stimulation from the synapse. When that electricity goes above the threshold, it gets ignited and transmits the electrical stimulation to the next neuron linked to the network. Neural networks distinguish things based on how electrical stimulations are transmitted.

Neural networks have originally been the type of supervised learning that represents this electrical stimulation with numbers. Recently, especially with deep learning, various types of neural networks algorithms have been introduced, and some of them are unsupervised learning. The algorithm increases the predictability by adjusting the weight of the networks through the process of learning. Deep learning is an algorithm based on neural networks. More details on neural networks will be explained later, with implementations.

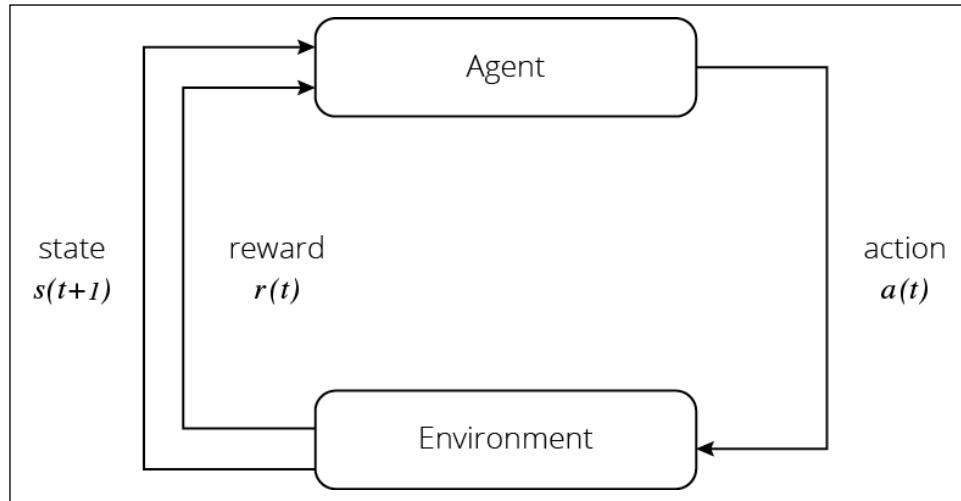
Logistic regression

Logistic regression is one of the statistical regression models of variables with the Bernoulli distribution. While SVM and neural networks are classification models, logistic regression is a regression model, yet it certainly is one of the supervised learning methods. Although logistic regression has a different base of thinking, as a matter of fact, it can be thought of as one of the neural networks when you look at its formula. Details on logistic regression will also be explained with implementations later.

As you can see, each machine learning method has unique features. It's important to choose the right algorithm based on what you would like to know or what you would like to use the data for. You can say the same of deep learning. Deep learning has different methods, so not only should you consider which the best method among them is, but you should also consider that there are some cases where you should not use deep learning. It's important to choose the best method for each case.

Reinforcement learning

Just for your reference, there is another method of machine learning called **reinforcement learning**. While some categorize reinforcement learning as unsupervised learning, others declare that all three learning algorithms, supervised learning, unsupervised learning, and reinforcement learning, should be divided into different types of algorithms, respectively. The following image shows the basic framework of reinforcement learning:



An agent takes an action based on the state of an environment and an environment will change based on the action. A mechanism with some sort of reward is provided to an agent following the change of an environment and the agent learns a better choice of act (decision-making).

Machine learning application flow

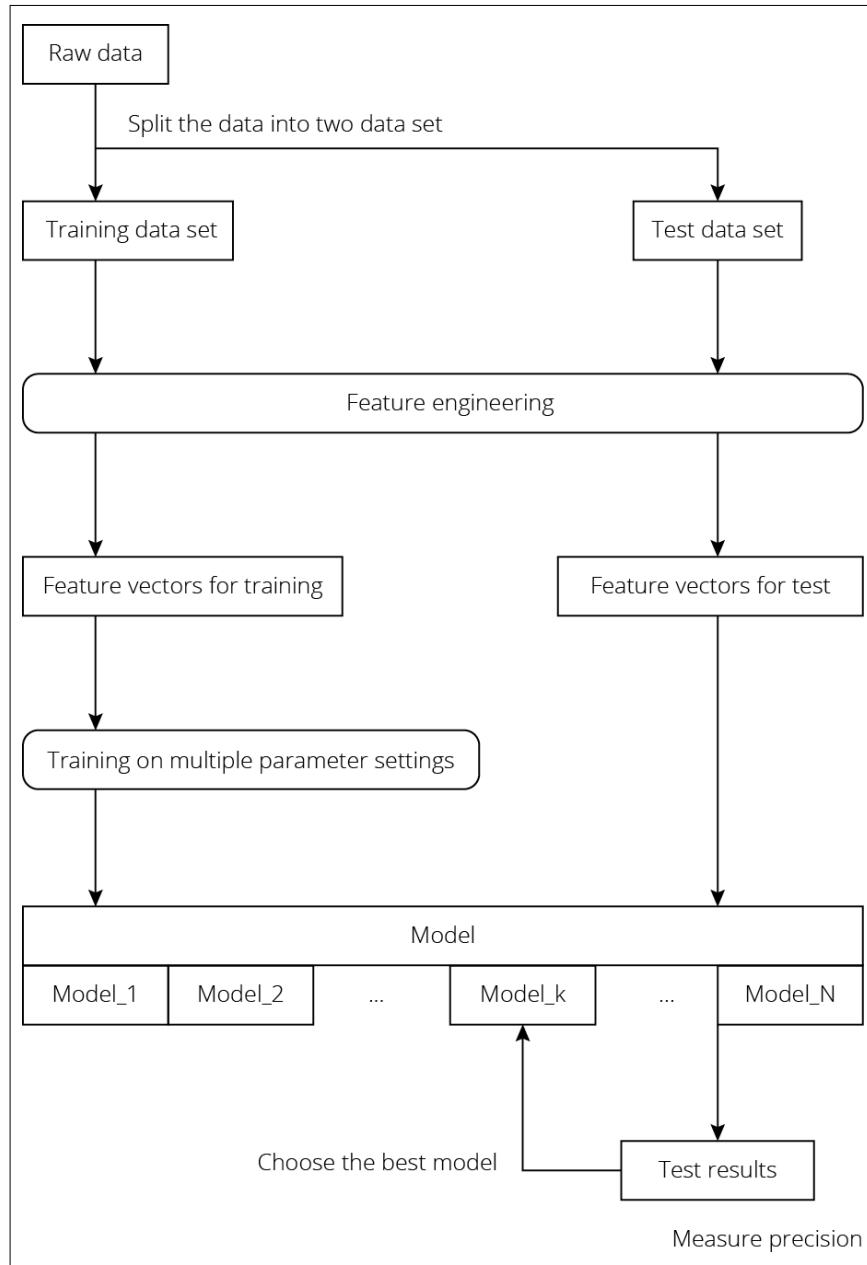
We have looked at the methods that machine learning has and how these methods recognize patterns. In this section, we'll see which flow is taken, or has to be taken, by data mining using machine learning. A decision boundary is set based on the model parameters in each of the machine learning methods, but we can't say that adjusting the model parameters is the only thing we have to care about. There is another troublesome problem, and it is actually the weakest point of machine learning: feature engineering. Deciding which features are to be created from raw data, that is, the analysis subject, is a necessary step in making an appropriate classifier. And doing this, which is the same as adjusting the model parameters, also requires a massive amount of trial and error. In some cases, feature engineering requires far more effort than deciding a parameter.

Thus, when we simply say "machine learning," there are certain tasks that need to be completed in advance as preprocessing to build an appropriate classifier to deal with actual problems. Generally speaking, these tasks can be summarized as follows:

- Deciding which machine learning method is suitable for a problem
- Deciding what features should be used
- Deciding which setting is used for model parameters

Only when these tasks are completed does machine learning become valuable as an application.

So, how do you decide the suitable features and parameters? How do you get a machine to learn? Let's first take a look at the following diagram as it might be easier for you to grasp the whole picture of machine learning. This is the summary of a learning flow:



As you can see from the preceding image, the learning phase of machine learning can be roughly divided into these two steps:

- Training
- Testing

Literally, model parameters are renewed and adjusted in the training phase and the machine examines the merit of a model in the test phase. We have no doubt that the research or experiment will hardly ever succeed with just one training and one test set. We need to repeat the process of training → test, training → test ... until we get the right model.

Let's consider the preceding flowchart in order. First, you need to divide the raw data into two: a training dataset and a test dataset. What you need to be very careful of here is that the training data and the test data are separated. Let's take an example so you can easily imagine what this means: you are trying to predict the daily price of S&P 500 using machine learning with historical price data. (In fact, predicting the prices of financial instruments using machine learning is one of the most active research fields.)

Given that you have historical stock price data from 2001 to 2015 as raw data, what would happen if you performed the training with all the data from 2001 to 2015 and similarly performed the test for the same period? The situation would occur that even if you used simple machine learning or feature engineering, the probability of getting the right prediction would be 70%, or even higher at 80% or 90%. Then, you might think: *What a great discovery! The market is actually that simple! Now I can be a billionaire!*

But this would end as short-lived elation. The reality doesn't go that well. If you actually start investment management with that model, you wouldn't get the performance you were expecting and would be confused. This is obvious if you think about it and pay a little attention. If a training dataset and a test dataset are the same, you do the test with the data for which you already know the answer. Therefore, it is a natural consequence to get high precision, as you have predicted a correct answer using a correct answer. But this doesn't make any sense for a test. If you would like to evaluate the model properly, be sure to use data with different time periods, for example, you should use the data from 2001 to 2010 for the training dataset and 2011 to 2015 for the test. In this case, you perform the test using the data you don't know the answer for, so you can get a proper prediction precision rate. Now you can avoid going on your way to bankruptcy, believing in investments that will never go well.

So, it is obvious that you should separate a training dataset and a test dataset but you may not think this is a big problem. However, in the actual scenes of data mining, the case often occurs that we conduct an experiment with the same data without such awareness, so please be extra careful. We've talked about this in the case of machine learning, but it also applies to deep learning.

If you divide a whole dataset into two datasets, the first dataset to be used is the training dataset. To get a better precision rate, we first need to think about creating features in the training dataset. This feature engineering partly depends on human experience or intuition. It might take a long time and a lot of effort before you can choose the features to get the best results. Also, each machine learning method has different types of data formats of features to be accepted because the theory of models and formulas are unique to each method. As an example, we have a model that can only take an integer, a model that can only take a non-negative number/value, and a model that can only take real numbers from 0 to 1. Let's look back at the previous example of stock prices. Since the value of the price varies a lot within a broader range, it may be difficult to make a prediction with a model that can only take an integer.

Additionally, we have to be careful to ensure that there is compatibility between the data and the model. We don't say we can't use a model that can take all the real numbers from 0 if you would like to use a stock price as is for features. For example, if you divide all the stock price data by the maximum value during a certain period, the data range can fit into 0-1, hence you can use a model that can only take real numbers from 0 to 1. As such, there is a possibility that you can apply a model if you slightly change the data format. You need to keep this point in mind when you think about feature engineering. Once you create features and decide which method of machine learning to apply, then you just need to examine it.

In machine learning, features are, of course, important variables when deciding on the precision of a model; however, a model itself, in other words a formula within the algorithm, also has parameters. Adjusting the speed of learning or adjusting how many errors to be allowed are good examples of this. The faster the learning speed, the less time it takes to finish the calculation, hence it's better to be fast. However, making the learning speed faster means that it only provides solutions in brief. So, we should be careful not to lose our expected precision rates. Adjusting the permissible range of errors is effective for the case where a noise is blended in the data. The standard by which a machine judges "is this data weird?" is decided by humans.

Each method, of course, has a set of peculiar parameters. As for neural networks, how many neurons there should be in one of the parameters is a good example. Also, when we think of the kernel trick in SVM, how we set the kernel function is also one of the parameters to be determined. As you can see, there are so many parameters that machine learning needs to define, and which parameter is best cannot be found out in advance. In terms of how we define model parameters in advance, there is a research field that focuses on the study of parameters.

Therefore, we need to test many combinations of parameters to examine which combination can return the best precision. Since it takes a lot of time to test each combination one by one, the standard flow is to test multiple models with different parameter combinations in concurrent processing and then compare them. It is usually the case that a range of parameters that should be set to some extent is decided, so it's not that the problem can't be solved within a realistic time frame.

When the model that can get good precision is ready in the training dataset, next comes the test step. The rough flow of the test is to apply the same feature engineering applied to the training dataset and the same model parameters respectively and then verify the precision. There isn't a particularly difficult step in the test. The calculation doesn't take time either. It's because finding a pattern from data, in other words optimizing a parameter in a formula, creates a calculation cost. However, once a parameter adjustment is done, then the calculation is made right away as it only applies the formula to new datasets. The reason for performing a test is, simply put, to examine whether a model is too optimized by the training dataset. What does this mean? Well, in machine learning, there are two patterns where a training set goes well but a test set doesn't.

The first case is incorrect optimization by classifying noisy data blended into a training dataset. This can be related to the adjustment of a permissible range of errors mentioned earlier in this chapter. Data in the world is not usually clean. It can be said that there is almost no data that can be properly classified into clean patterns. The prediction of stock prices is a good example again. Stock prices usually repeat moderate fluctuations from previous stock prices, but sometimes they suddenly surge or drop sharply. And, there is, or should be, no regularity in this irregular movement. Another case is if you would like to predict the yield of a crop for a country; the data of the year affected by abnormal weather should be largely different from the normal years' data. These examples are extreme and easy to understand, but most for a data in the real world also contains noises, making it difficult to classify data into proper patterns. If you just do training without adjusting the parameters of machine learning, the model forces it to classify the noise data into a pattern. In this case, data from the training dataset might be classified correctly, but since noise data in the training dataset is also classified and the noise doesn't exist in the test dataset, the predictability in a test should be low.

The second case is incorrect optimizing by classifying data that is characteristic only in a training dataset. For example, let's think about making an app of English voice inputs. To build your app, you should prepare the data of pronunciation for various words as a training dataset. Now, let's assume you prepared enough voice data of British English native speakers and were able to create a high precision model that could correctly classify the pronunciation in the training dataset. The next step is a test. Since it's a test, let's use the voice data of American English native speakers for the means of providing different data. What would be the result then? You probably wouldn't get good precision. Furthermore, if you try the app to recognize the pronunciation of non-native speakers of English, the precision would be much lower. As you know, English has different pronunciations for different areas. If you don't take this into consideration and optimize the model with the training data set of British English, even though you may get a good result in the training set, you won't get a good result in the test set and it won't be useful for the actual application.

These two problems occur because the machine learning model learns from a training dataset and fits into the dataset too much. This problem is literally called the **overfitting problem**, and you should be very careful to avoid it. The difficulty of machine learning is that you have to think about how to avoid this overfitting problem besides the feature engineering. These two problems, overfitting and feature engineering, are partially related because poor feature engineering would fail into overfitting.

To avoid the problem of overfitting, there's not much to do except increase the amount of data or the number of tests. Generally, the amount of data is limited, so the methods of increasing the number of tests are often performed. The typical example is **K-fold cross-validation**. In K-fold cross-validation, all the data is divided into K sets at the beginning. Then, one of the datasets is picked as a test dataset and the rest, K-1, are put as training datasets. Cross-validation performs the verification on each dataset divided into K for K times, and the precision is measured by calculating the average of these K results. The most worrying thing is that both a training dataset and a test dataset may happen to have good precision by chance; however, the probability of this accident can be decreased in K-fold cross-validation as it performs a test several times. You can never worry too much about overfitting, so it's necessary that you verify results carefully.

Well, you have now read through the flow of training and test sets and learned key points to be kept in mind. These two mainly focus on data analysis. So, for example, if your purpose is to pull out the meaningful information from the data you have and make good use of it, then you can go through this flow. On the other hand, if you need an application that can cope with a further new model, you need an additional process to make predictions with a model parameter obtained in a training and a test set. As an example, if you would like to find out some information from a dataset of stock prices and analyze and write a market report, the next step would be to perform training and test sets. Or, if you would like to predict future stock prices based on the data and utilize it as an investment system, then your purpose would be to build an application using a model obtained in a training and a test set and to predict a price based on the data you can get anew every day, or from every period you set. In the second case, if you would like to renew the model with the data that is newly added, you need to be careful to complete the calculation of the model building by the time the next model arrives.

Theories and algorithms of neural networks

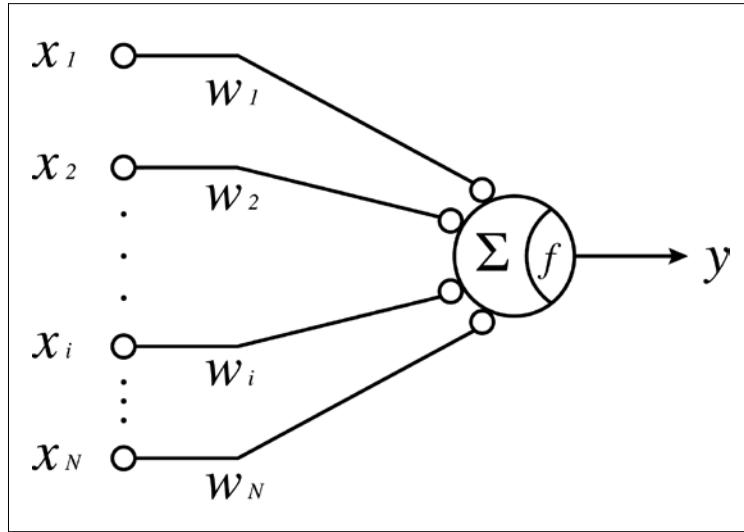
In the previous section, you saw the general flow of when we perform data analysis with machine learning. In this section, theories and algorithms of neural networks, one of the methods of machine learning, are introduced as a preparation toward deep learning.

Although we simply say "neural networks", their history is long. The first published algorithm of neural networks was called **perceptron**, and the paper released in 1957 by Frank Rosenblatt was named *The Perceptron: A Perceiving and Recognizing Automaton (Project Para)*. From then on, many methods were researched, developed, and released, and now neural networks are one of the elements of deep learning. Although we simply say "neural networks," there are various types and we'll look at the representative methods in order now.

Perceptrons (single-layer neural networks)

The perceptron algorithm is the model that has the simplest structure in the algorithms of neural networks and it can perform linear classification for two classes. We can say that it's the prototype of neural networks. It is the algorithm that models human neurons in the simplest way.

The following figure is a schematic drawing of the general model:



Here, x_i shows the input signal, w_i shows the weight corresponding to each input signal, and y shows the output signal. f is the activation function. Σ shows, literally, the meaning of calculating the sum of data coming from the input. Please bear in mind that x_i applies a processing of nonlinear conversion with feature engineering in advance, that is, x_i is an engineered feature.

Then, the output of perceptron can be represented as follows:

$$y(x) = f(w^T x)$$

$$f(a) = \begin{cases} +1, & a \geq 0 \\ -1, & a < 0 \end{cases}$$

$f(*)$ is called the step function. As shown in the equation, Perceptron returns the output by multiplying each factor of the feature vector by weight, calculating the sum of them, and then activating the sum with the step function. The output is the result estimated by Perceptron. During the training, you will compare this result with the correct data and feed back the error.

Let t be the value of the labeled data. Then, the formula can be represented as follows:

$$t \in \{-1, 1\}$$

If some labeled data belongs to class 1, C_1 , we have $t = 1$. If it belongs to class 2, C_2 , we have $t = -1$. Also, if the input data is classified correctly, we get:

$$\begin{cases} w^T x_n > 0 \text{ where } x_n \in C_1 \\ w^T x_n < 0 \text{ where } x_n \in C_2 \end{cases}$$

So, putting these equations together, we have the following equation of properly classified data:

$$w^T x_n t_n > 0$$

Therefore, you can increase the predictability of Perceptron by minimizing the following function:

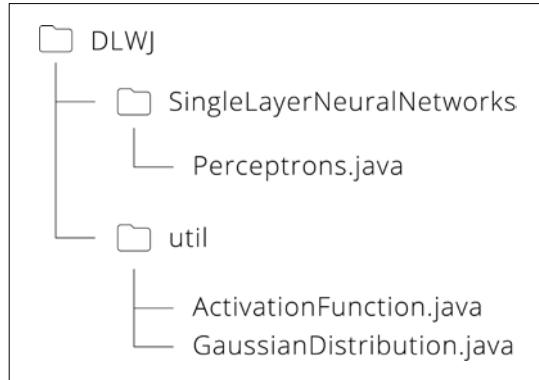
$$E(w) = - \sum_{n \in M} w^T x_n t_n$$

Here, E is called the error function. M shows the set of misclassification. To minimize the error function, gradient descent, or steepest descent, an optimization algorithm is used to find a local minimum of a function using gradient descent. The equation can be described as follows:

$$w^{(k+1)} = w^{(k)} - n \nabla E(w) = w^{(k)} + \eta x_n t_n$$

Here, η is the learning rate, a common parameter of the optimization algorithm that adjusts the learning speed, and k shows the number of steps of the algorithm. In general, the smaller the value of the learning rate, the more probable it is that the algorithm falls into a local minimum because the model can't override the old value much. If the value is too big, however, the model parameters can't converge because the values fluctuate too widely. Therefore, practically, the learning rate is set to be big at the beginning and then dwindle with each iteration. On the other hand, with perceptrons, it is proved that the algorithm converges irrespective of the value of the learning rate when the data set is linearly separable, and thus the value is set to be 1.

Now, let's look at an implementation. The package structure is as follows:



Let's have a look at the content of `Perceptrons.java` as shown in the previous image. We will look into the main methods one by one.

First, we define the parameters and constants that are needed for learning. As explained earlier, the learning rate (defined as `learningRate` in the code) can be 1:

```

final int train_N = 1000; // number of training data
final int test_N = 200; // number of test data
final int nIn = 2; // dimensions of input data

double[][] train_X = new double[train_N][nIn]; // input data for
training
int[] train_T = new int[train_N]; // output data (label)
for training

double[][] test_X = new double[test_N][nIn]; // input data for test
int[] test_T = new int[test_N]; // label of inputs
int[] predicted_T = new int[test_N]; // output data predicted
by the model

final int epochs = 2000; // maximum training epochs
final double learningRate = 1.; // learning rate can be 1 in
perceptrons
    
```

Needless to say, machine learning and deep learning need a dataset to be learned and classified. Here, since we would like to focus on implementations deeply related to the theory of perceptrons, a sample dataset is generated within the source code and is used for the training and test sets, the class called `GaussianDistribution` is defined, and it returns a value following the normal distribution or Gaussian distribution. As for the source code itself, we don't mention it here as you can see it in `GaussianDistribution.java`. We set the dimensions of the learning data in `nIn = 2` and define two types of instances as follows:

```
GaussianDistribution g1 = new GaussianDistribution(-2.0, 1.0, rng);
GaussianDistribution g2 = new GaussianDistribution(2.0, 1.0, rng);
```

You can get the values that follow the normal distributions with a mean of `-2.0` and a variance of `1.0` by `g1.random()` and a mean of `2.0` and a variance of `1.0` by `g2.random()`.

With these values, 500 data attributes are generated in class 1 obtained by `[g1.random(), g2.random()]` and another 500 generated in class 2 obtained by `[g2.random(), g1.random()]`. Also, please bear in mind that each value of the class 1 label is `1` and of the class 2 label is `-1`. Almost all the data turns out to be a value around `[-2.0, 2.0]` for class 1 and `[2.0, -2.0]` for class 2; hence, they can be linearly separated, but some data can be blended near the other class as noise.

Now we have prepared the data, we can move on to building the model. The number of units in the input layer, `nIn`, is the argument used here to decide the model outline:

```
Perceptrons classifier = new Perceptrons(nIn);
```

Let's look at the actual `Perceptrons` constructor. The parameter of the perceptrons model is only the weight, `w`, of the network—very simple—as follows:

```
public Perceptrons(int nIn) {

    this.nIn = nIn;
    w = new double[nIn];
}
```

The next step is finally the training. The iteration of learning continues until it reaches enough numbers of the learning set in advance or classifies all the training data correctly:

```
while (true) {
    int classified_ = 0;

    for (int i=0; i < train_N; i++) {
```

```

        classified_ += classifier.train(train_X[i], train_T[i],
learningRate);
    }

    if (classified_ == train_N) break; // when all data classified
correctly

    epoch++;
    if (epoch > epochs) break;
}

```

In the `train` method, you can write down the gradient descent algorithm as we just explained. Here, the `w` parameter of the network is updated:

```

public int train(double[] x, int t, double learningRate) {

    int classified = 0;
    double c = 0.;

    // check whether the data is classified correctly
    for (int i = 0; i < nIn; i++) {
        c += w[i] * x[i] * t;
    }

    // apply gradient descent method if the data is wrongly classified
    if (c > 0) {
        classified = 1;
    } else {
        for (int i = 0; i < nIn; i++) {
            w[i] += learningRate * x[i] * t;
        }
    }

    return classified;
}

```

Once you have done enough numbers of learning and finish, the next step is to perform the test. First, let's check which class the test data is classified by in the well-trained model:

```

for (int i = 0; i < test_N; i++) {
    predicted_T[i] = classifier.predict(test_X[i]);
}

```

In the predict method, simply activate the input through the network. The step function used here is defined in ActivationFunction.java:

```
public int predict (double[] x) {  
  
    double preActivation = 0.;  
  
    for (int i = 0; i < nIn; i++) {  
        preActivation += w[i] * x[i];  
    }  
  
    return step(preActivation);  
}
```

Subsequently, we evaluate the model using the test data. You might need more explanation to perform this part.

Generally, the performance of the method of machine learning is measured by the indicator of accuracy, precision, and recall based on the confusion matrix. The confusion matrix summarizes the results of a comparison of the predicted class and the correct class in the matrix and is shown as the following table:

	p_predicted	n_predicted
p_actual	True positive (TP)	False negative (FN)
n_actual	False positive (FP)	True negative (TN)

The three indicators are shown below:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

Accuracy shows the proportion of the data that is correctly classified for all the data, while precision shows the proportion of the actual correct data to the data predicted as positive, and recall is the proportion of the data predicted as positive to the actual positive data. Here is the code for this:

```
int[][] confusionMatrix = new int[2][2];
double accuracy = 0.;
double precision = 0.;
double recall = 0.;

for (int i = 0; i < test_N; i++) {

    if (predicted_T[i] > 0) {
        if (test_T[i] > 0) {
            accuracy += 1;
            precision += 1;
            recall += 1;
            confusionMatrix[0][0] += 1;
        } else {
            confusionMatrix[1][0] += 1;
        }
    } else {
        if (test_T[i] > 0) {
            confusionMatrix[0][1] += 1;
        } else {
            accuracy += 1;
            confusionMatrix[1][1] += 1;
        }
    }
}

accuracy /= test_N;
precision /= confusionMatrix[0][0] + confusionMatrix[1][0];
recall /= confusionMatrix[0][0] + confusionMatrix[0][1];

System.out.println("-----");
System.out.println("Perceptrons model evaluation");
System.out.println("-----");
System.out.printf("Accuracy: %.1f %%\n", accuracy * 100);
System.out.printf("Precision: %.1f %%\n", precision * 100);
System.out.printf("Recall: %.1f %%\n", recall * 100);
```

When you compile `Perceptron.java` and run it, you can get 99.0% for accuracy, 98.0% for precision, and 100% for recall. This means that actual positive data is classified correctly but that there has been some data wrongly predicted as positive when it is actually negative. In this source code, since the data set is for demonstration, K-fold cross-validation is not included. The dataset in the example above is programmatically generated and has little noise data. Therefore, accuracy, precision, and recall are all high because the data can be well classified. However, as mentioned above, you have to look carefully at results, especially when you have great results.

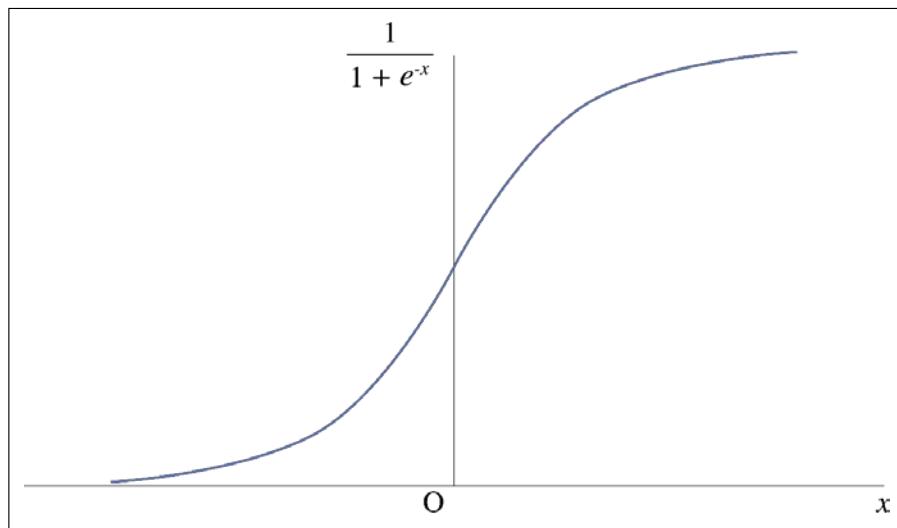
Logistic regression

Logistic regression is, as you can assume from the name, the regression model. But when you look at the formula, you can see that logistic regression is the linear separation model that generalizes perceptrons.

Logistic regression can be regarded as one of the neural networks. With perceptrons, the step function is used for the activation function, but in logistic regression, the (logistic) sigmoid function is used. The equation of the sigmoid function can be represented as follows:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The graph of this function can be illustrated as follows:



The sigmoid function maps any values of a real number to a value from 0 to 1. Therefore, the output of the logistic regression can be regarded as the posterior probability for each class. The equations can be described as follows:

$$p(C=1|x) = y(x) = \sigma(w^T x + b)$$

$$p(C=0|x) = 1 - p(C=1|x)$$

These equations can be combined to make:

$$p(C=t|x) = y^t (1-y)^{1-t}$$

Here, $t \in \{0, 1\}$ is the correct data. You may have noticed that the range of the data is different from the one of perceptrons.

With the previous equation, the likelihood function, which estimates the maximum likelihood of the model parameters, can be expressed as follows:

$$L(w, b) = \prod_{n=1}^N y_n^{t_n} (1-y_n)^{1-t_n}$$

Where:

$$y_n = p(C=1|x_n)$$

As you can see, not only the weight of the network but the bias b are also parameters that need to be optimized.

What we need to do now is maximize the likelihood function, but the calculation is worrying because the function has a mathematical product. To make the calculation easier, we take the logarithm (log) of the likelihood function. Additionally, we substitute the sign to turn the object to minimizing the negative log likelihood function. Since the log is the monotonic increase, the magnitude correlation doesn't change. The equation can be represented as follows:

$$E(w, b) = -\ln L(w, b) = -\sum_{n=1}^N \{t_n \ln y_n + (1-t_n) \ln (1-y_n)\}$$

You can see the error function at the same time. This type of function is called a cross-entropy error function.

Similar to perceptrons, we can optimize the model by computing the gradients of the model parameters, w and b . The gradients can be described as follows:

$$\frac{\partial E(w, b)}{\partial w} = -\sum_{n=1}^N (t_n - y_n) x_n$$

$$\frac{\partial E(w, b)}{\partial b} = -\sum_{n=1}^N (t_n - y_n)$$

With these equations, we can update the model parameters as follows:

$$w^{(k+1)} = w^{(k)} - \eta \frac{\partial E(w, b)}{\partial w} = w^{(k)} + \eta \sum_{n=1}^N (t_n - y_n) x_n$$

$$b^{(k+1)} = b^{(k)} - \eta \frac{\partial E(w, b)}{\partial b} = b^{(k)} + \eta \sum_{n=1}^N (t_n - y_n)$$

Theoretically, we have no problem using the equations just mentioned and implementing them. As you can see, however, you have to calculate the sum of all the data to compute the gradients for each iteration. This will hugely increase the calculation cost once the size of a dataset becomes big.

Therefore, another method is usually applied that partially picks up some data from the dataset, computes the gradients by calculating the sum only with picked data, and renews the parameters. This is called **stochastic gradient descent (SGD)** because it stochastically chooses a subset of the data. This subset of the dataset used for one renewal is called a **mini-batch**.



SGD using a mini-batch is sometimes called **mini-batch stochastic gradient descent (MSGD)**. Online training that learns to randomly choose one data from the dataset is called SGD to distinguish one from the other. In this book, however, both MSGD and SGD are called SGD, as both become the same when the size of the mini-batch is 1. Since learning by each data does increase the calculation cost, it's better to use mini-batches.

In terms of the implementation of logistic regression, since it can be covered with multi-class logistic regression introduced in the next section, we won't write the code here. You can refer to the code of multi-class logistic regression in this section.

Multi-class logistic regression

Logistic regression can also be applied to multi-class classification. In two-class classification, the activation function is the sigmoid function, and you can classify the data by evaluating the output value shifting from 0 to 1. How, then, can we classify data when the number of classes is K? Fortunately, it is not difficult. We can classify multi-class data by changing the equation for the output to the K-dimensional class-membership probability vector, and we use the softmax function to do so, which is the multivariate version of the sigmoid function. The posterior probability of each class can be represented as follows:

$$p(C = k | x) = y_k(x) = \frac{\exp(w_k^T x + b_k)}{\sum_{j=1}^K \exp(w_j^T x + b_j)}$$

With this, the same as two-class cases, you can get the likelihood function and the negative log likelihood function as follows:

$$L(W, b) = \prod_{n=1}^N \prod_{k=1}^K y_{nk}^{t_{nk}}$$

$$E(W, b) = -\ln L(W, b) = -\sum_{n=1}^N \sum_{k=1}^K t_{nk} \ln y_{nk}$$

Here, $W = [w_1, \dots, w_j, \dots, w_K]$, $y_{nk} = y_k(x_n)$. Also, t_{nk} is the Kth element of the correct data vector, t_n , which corresponds to the n^{th} training data. If an input data belongs to the class k , the value of t_{nk} is 1; the value is 0 otherwise.

Gradients of the loss function against the model parameters, the weight vector, and the bias, can be described as follows:

$$\frac{\partial E}{\partial w_j} = -\sum_{n=1}^N (t_{nj} - y_{nj}) x_n$$

$$\frac{\partial E}{\partial b_j} = -\sum_{n=1}^N (t_{nj} - y_{nj})$$

Now let's look through the source code to better understand the theory. You can see some variables related to mini-batches besides the ones necessary for the model:

```
int minibatchSize = 50; // number of data in each minibatch
int minibatch_N = train_N / minibatchSize; // number of minibatches

double[][][] train_X_minibatch = new double[minibatch_N]
[minibatchSize][nIn]; // minibatches of training data
int[][][] train_T_minibatch = new int[minibatch_N][minibatchSize]
[nOut]; // minibatches of output data for training
```

The following code is the process to shuffle training data so the data of each mini-batch is to be applied randomly to SGD:

```
List<Integer> minibatchIndex = new ArrayList<>(); // data index for
minibatch to apply SGD
for (int i = 0; i < train_N; i++) minibatchIndex.add(i);
Collections.shuffle(minibatchIndex, rng); // shuffle data index for
SGD
```

Since we can see the multi-class classification problem, we generate a sample dataset with three classes. In addition to mean values and variances used in perceptrons, we also use the dataset according to normal distribution with the mean of 0.0 and the variance of 1.0 for the training data and the test data for class 3. In other words, each class's data follows normal distributions with the mean of [-2.0, 2.0], [2.0, -1.0] and [0.0, 0.0] and the variance of 1.0. We defined the training data as the int type and the test data as the Integer type for the labeled data. This is to process the test data easier when evaluating the model. Also, each piece of labeled data is defined as an array because it follows multi-class classification:

```
train_T[i] = new int []{1, 0, 0};  
test_T[i] = new Integer []{1, 0, 0};
```

Then we classify the training data into a mini-batch using minibatchIndex, which was defined earlier:

```
for (int i = 0; i < minibatch_N; i++) {  
    for (int j = 0; j < minibatchSize; j++) {  
        train_X_minibatch[i][j] = train_X[minibatchIndex.get(i *  
minibatchSize + j)];  
        train_T_minibatch[i][j] = train_T[minibatchIndex.get(i *  
minibatchSize + j)];  
    }  
}
```

Now we have prepared the data, let's practically build a model:

```
LogisticRegression classifier = new LogisticRegression(nIn, nOut);
```

The model parameters of logistic regression are w, weight of the network, and bias b:

```
public LogisticRegression(int nIn, int nOut) {  
  
    this.nIn = nIn;  
    this.nOut = nOut;  
  
    W = new double[nOut][nIn];  
    b = new double[nOut];  
  
}
```

The training is done with each mini-batch. If you set minibatchSize = 1, you can make the training so-called online training:

```
for (int epoch = 0; epoch < epochs; epoch++) {  
    for (int batch = 0; batch < minibatch_N; batch++) {
```

```
    classifier.train(train_X_minibatch[batch], train_T_
minibatch[batch], minibatchSize, learningRate);
}
learningRate *= 0.95;
}
```

Here, the learning rate gradually decreases so that the model can converge. Now, for the actual training `train` method, you can briefly divide it into two parts, as follows:

1. Calculate the gradient of w and b using the data from the mini-batch.
2. Update w and b with the gradients:

```
// 1. calculate gradient of w, b
for (int n = 0; n < minibatchSize; n++) {

    double[] predicted_Y_ = output(X[n]);

    for (int j = 0; j < nOut; j++) {
        dY[n][j] = predicted_Y_[j] - T[n][j];

        for (int i = 0; i < nIn; i++) {
            grad_W[j][i] += dY[n][j] * X[n][i];
        }

        grad_b[j] += dY[n][j];
    }
}

// 2. update params
for (int j = 0; j < nOut; j++) {
    for (int i = 0; i < nIn; i++) {
        W[j][i] -= learningRate * grad_W[j][i] / minibatchSize;
    }
    b[j] -= learningRate * grad_b[j] / minibatchSize;
}

return dY;
```

At the end of the `train` method, `return dY`, the error value of the predicted data and the correct data is returned. This is not mandatory for logistic regression itself but it is necessary in the machine learning and the deep learning algorithms introduced later.

Next up for the training is the test. The process of performing the test doesn't really change from the one for perceptrons.

First, with the predict method, let's predict the input data using the trained model:

```
for (int i = 0; i < test_N; i++) {
    predicted_T[i] = classifier.predict(test_X[i]);
}
```

The predict method and the output method called are written as follows:

```
public Integer[] predict(double[] x) {

    double[] y = output(x); // activate input data through learned
    networks
    Integer[] t = new Integer[nOut]; // output is the probability, so
    cast it to label

    int argmax = -1;
    double max = 0.;

    for (int i = 0; i < nOut; i++) {
        if (max < y[i]) {
            max = y[i];
            argmax = i;
        }
    }

    for (int i = 0; i < nOut; i++) {
        if (i == argmax) {
            t[i] = 1;
        } else {
            t[i] = 0;
        }
    }

    return t;
}

public double[] output(double[] x) {

    double[] preActivation = new double[nOut];

    for (int j = 0; j < nOut; j++) {

        for (int i = 0; i < nIn; i++) {
            preActivation[j] += W[j][i] * x[i];
        }
    }
}
```

```
    }

    preActivation[j] += b[j]; // linear output
}

return softmax(preActivation, nOut);
}
```

First, input data is activated with the `output` method. As you can see from the bottom of the output, the activation function uses the `softmax` function. `softmax` is defined in `ActivationFunction.java`, and with this function the array showing the probability of each class is returned, hence you just need to get the index within the array of the element that has the highest probability. The index represents the predicted class.

Finally, let's evaluate the model. Again, the confusion matrix is introduced for model evaluation, but be careful as you need to find the precision or recall for each class this time because we have multi-class classification here:

```
int[][] confusionMatrix = new int[patterns][patterns];
double accuracy = 0.;
double[] precision = new double[patterns];
double[] recall = new double[patterns];

for (int i = 0; i < test_N; i++) {
    int predicted_ = Arrays.asList(predicted_T[i]).indexOf(1);
    int actual_ = Arrays.asList(test_T[i]).indexOf(1);

    confusionMatrix[actual_][predicted_] += 1;
}

for (int i = 0; i < patterns; i++) {
    double col_ = 0.;
    double row_ = 0.;

    for (int j = 0; j < patterns; j++) {

        if (i == j) {
            accuracy += confusionMatrix[i][j];
            precision[i] += confusionMatrix[j][i];
            recall[i] += confusionMatrix[i][j];
        }

        col_ += confusionMatrix[j][i];
        row_ += confusionMatrix[i][j];
    }
}
```

```

        }
        precision[i] /= col_;
        recall[i] /= row_;
    }

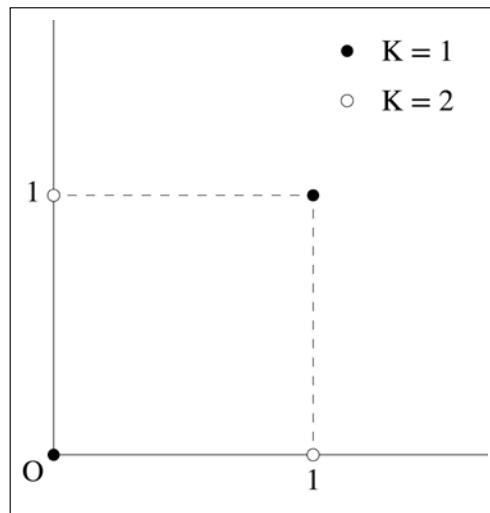
accuracy /= test_N;

System.out.println("-----");
System.out.println("Logistic Regression model evaluation");
System.out.println("-----");
System.out.printf("Accuracy: %.1f %%\n", accuracy * 100);
System.out.println("Precision:");
for (int i = 0; i < patterns; i++) {
    System.out.printf(" class %d: %.1f %%\n", i+1, precision[i] * 100);
}
System.out.println("Recall:");
for (int i = 0; i < patterns; i++) {
    System.out.printf(" class %d: %.1f %%\n", i+1, recall[i] * 100);
}

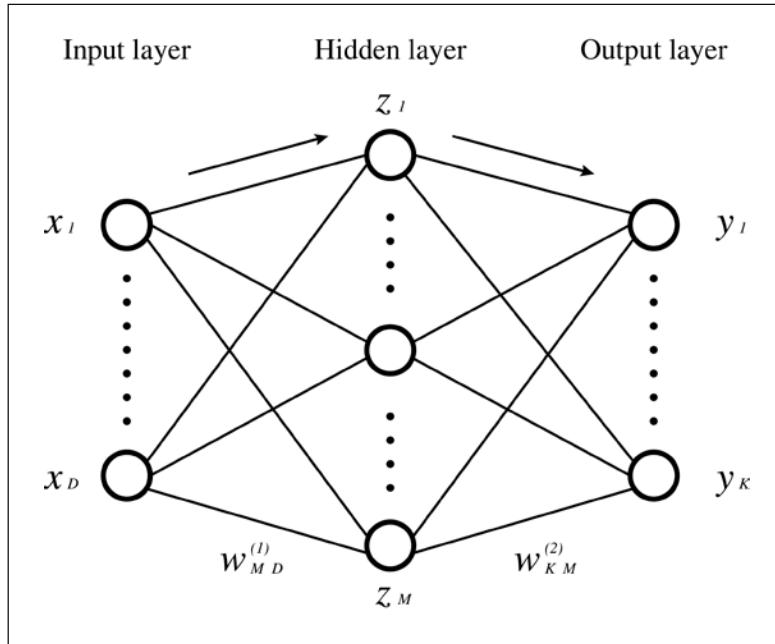
```

Multi-layer perceptrons (multi-layer neural networks)

Single-layer neural networks have a huge problem. Perceptrons or logistic regressions are efficient for problems that can be linearly classified but they can't solve nonlinear problems at all. For example, they can't even solve the simplest XOR problem seen in the figure here:



Since most of the problems in the real world are nonlinear, perceptrons and logistic regression aren't applicable for practical uses. Hence, the algorithm was improved to correspond to nonlinear problems. These are multi-layer perceptrons (or **multi-layer neural networks, MLPs**). As you can see from the name, by adding another layer, called a hidden layer, between the input layer and the output layer, the networks have the ability to express various patterns. This is the graphical model of an MLP:



What is most important here is not to introduce the skip-layer connection. In neural networks, it is better for both theory and implementation to keep the model as having a feed-forward network structure. By sticking to these rules, and by increasing the number of hidden layers, you can approximate arbitrary functions without making the model too complicated mathematically.

Now, let's see how we compute the output. It looks complicated at first glance but it accumulates the layers and the scheme of the network's weight or activation in the same way, so you simply have to combine the equation of each layer. Each output can be shown as follows:

$$E(W, b) = -\ln L(W, b) = -\sum_{n=1}^N \sum_{k=1}^K t_{nk} \ln Y_{nk}$$

Here, h is the activation function of the hidden layer and g is the output layer.

As has already been introduced, in the case of multi-class classification, the activation function of the output layer can be calculated efficiently by using the `softmax` function, and the error function is given as follows:

$$\begin{aligned} y_k &= g\left(\sum_{j=1}^M w_{kj}^{(2)} z_j + b_k^{(2)}\right) \\ &= g\left(\sum_{j=1}^M w_{kj}^{(2)} h\left(\sum_{i=1}^D w_{ji}^{(1)} x_i + b_j^{(1)}\right) + b_k^{(2)}\right) \end{aligned}$$

As for a single layer, it's fine just to reflect this error in the input layer, but for the multi-layer, neural networks cannot learn as a whole unless you reflect the error in both the hidden layer and input layer.

Fortunately, in feed-forward networks, there is an algorithm known as `backpropagation`, which enables the model to propagate this error efficiently by tracing the network forward and backward. Let's look at the mechanism of this algorithm. To make the equation more readable, we'll think about the valuation of an error function in the online training, shown as follows:

$$E(W, b) = \sum_{n=1}^N E_n(W, b)$$

We can now think about just the gradient of this, E_n . Since all the data in a dataset in most cases of practical application is independent and identically distributed, we have no problem defining it as we just mentioned.

Each unit in the feed-forward network is shown as the sum of the weight of the network connected to the unit, hence the generalized term can be represented as follows:

$$a_j = \sum_i w_{ji}x_i + b_j$$

$$z_j = h(a_j)$$

Be careful, as x_i here is not only the value of the input layer (of course, this can be the value of the input layer). Also, h is the nonlinear activation function. The gradient of weights and the gradient of the bias can be shown as follows:

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} x_i$$

$$\frac{\partial E_n}{\partial b_j} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial b_j} = \frac{\partial E_n}{\partial a_j}$$

Now, let the notation defined in the next equation be introduced:

$$\delta_j := \frac{\partial E_n}{\partial a_j}$$

Then, we get:

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j x_i$$

$$\frac{\partial E_n}{\partial b_j} = \delta_j$$

Therefore, when we compare the equations, the output unit can be described as follows:

$$\delta_k = y_k - t_k$$

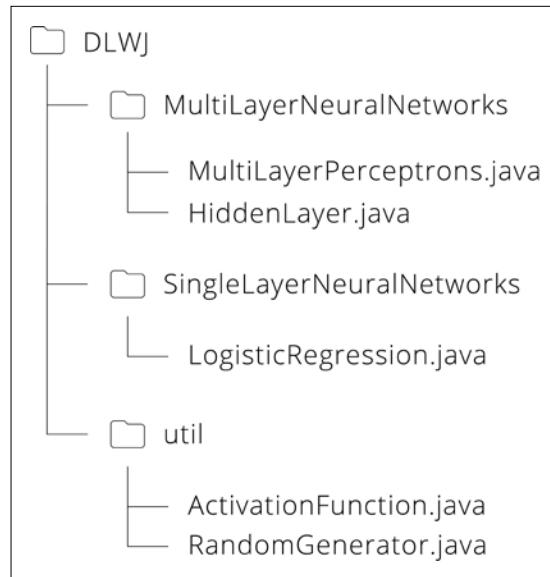
Also, each unit of the hidden layer is:

$$\delta_j = \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j}$$

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$$

Thus, the **backpropagation formula** is introduced. As such, delta is called the **backpropagated** error. By computing the backpropagated error, the weights and bias can be calculated. It may seem difficult when you look at the formula, but what it basically does is receive feedback on errors from a connected unit and renew the weight, so it's not that difficult.

Now, let's look at an implementation with a simple XOR problem as an example. You will have better understanding when you read the source code. The structure of the package is as follows:



The basic flow of the algorithm is written in `MultiLayerPerceptrons.java`, but the actual part of backpropagation is written in `HiddenLayer.java`. We use multi-class logistic regression for the output layer. Since there is no change in `LogisticRegression.java`, the code is not shown in this section. In `ActivationFunction.java`, derivatives of the sigmoid function and hyperbolic tangent are added. The hyperbolic tangent is also the activation function that is often used as an alternative to the sigmoid. Also, in `RandomGenerator.java`, the method to generate random numbers with a uniform distribution is written. This is to randomly initialize the weight of the hidden layer, and it is quite an important part because a model often falls into a local optimum and fails to classify the data depending on these initial values.

Let's have a look at the content of `MultiLayerPerceptrons.java`. In `MultiLayerPerceptrons.java`, differently defined classes are defined respectively for each layer: `HiddenLayer` class is used for the hidden layer and `LogisticRegression` class for the output layer. Instances of these classes are defined as `hiddenLayer` and `logisticLayer`, respectively:

```
public MultiLayerPerceptrons(int nIn, int nHidden, int nOut, Random
rng) {

    this.nIn = nIn;
    this.nHidden = nHidden;
    this.nOut = nOut;

    if (rng == null) rng = new Random(1234);
    this.rng = rng;

    // construct hidden layer with tanh as activation function
    hiddenLayer = new HiddenLayer(nIn, nHidden, null, null, rng,
    "tanh"); // sigmoid or tanh

    // construct output layer i.e. multi-class logistic layer
    logisticLayer = new LogisticRegression(nHidden, nOut);

}
```

The parameters of the MLP are the weights w and bias b of the hidden layer, `HiddenLayer`, and the output layer, `LogisticRegression`. Since the output layer is the same as the one previously introduced, we won't look at the code here. The constructor of `HiddenLayer` is as follows:

```
public HiddenLayer(int nIn, int nOut, double[][] W, double[] b, Random
rng, String activation) {

    if (rng == null) rng = new Random(1234); // seed random

    if (W == null) {

        W = new double[nOut][nIn];
        double w_ = 1. / nIn;

        for(int j = 0; j < nOut; j++) {
            for(int i = 0; i < nIn; i++) {
                W[j][i] = uniform(-w_, w_, rng); // initialize W with
uniform distribution
            }
        }
    }

    if (b == null) b = new double[nOut];

    this.nIn = nIn;
    this.nOut = nOut;
    this.W = W;
    this.b = b;
    this.rng = rng;

    if (activation == "sigmoid" || activation == null) {

        this.activation = (double x) -> sigmoid(x);
        this.dactivation = (double x) -> dsigmoid(x);

    } else if (activation == "tanh") {

        this.activation = (double x) -> tanh(x);
        this.dactivation = (double x) -> dtanh(x);

    } else {
}
```

```
        throw new IllegalArgumentException("activation function not
supported");
    }

}
```

w is initialized, randomly matching the number of the units. This initialization is actually tricky as it makes you face the local minima problem more often if the initial values are not well distributed. Therefore, in a practical scene, it often happens that the model is tested with some random seeds. The activation function can be applied to either the sigmoid function or the hyperbolic tangent function.

The training of the MLP can be given by forward propagation and backward propagation through the neural networks in order:

```
public void train(double[][] X, int T[][], int minibatchSize, double
learningRate) {

    double[][] Z = new double[minibatchSize][nIn]; // outputs of
    hidden layer (= inputs of output layer)
    double[][] dY;

    // forward hidden layer
    for (int n = 0; n < minibatchSize; n++) {
        Z[n] = hiddenLayer.forward(X[n]); // activate input units
    }

    // forward & backward output layer
    dY = logisticLayer.train(Z, T, minibatchSize, learningRate);

    // backward hidden layer (backpropagate)
    hiddenLayer.backward(X, Z, dY, logisticLayer.W, minibatchSize,
learningRate);
}
```

The part of `hiddenLayer.backward` gives the hidden layer backpropagation of the prediction error, dY , from a logistic regression. Be careful, as the input data of a logistic regression is also necessary for the backpropagation:

```
public double[][] backward(double[][] X, double[][] Z, double[][] dY,
double[][] Wprev, int minibatchSize, double learningRate) {

    double[][] dZ = new double[minibatchSize][nOut]; // /
    backpropagation error

    double[][] grad_W = new double[nOut][nIn];
```

```

double[] grad_b = new double[nOut];

// train with SGD
// calculate backpropagation error to get gradient of W, b
for (int n = 0; n < minibatchSize; n++) {

    for (int j = 0; j < nOut; j++) {

        for (int k = 0; k < dY[0].length; k++) { // k < ( nOut of
previous layer )
            dZ[n][j] += Wprev[k][j] * dY[n][k];
        }
        dZ[n][j] *= dactivation.apply(Z[n][j]);
    }

    for (int i = 0; i < nIn; i++) {
        grad_W[j][i] += dZ[n][j] * X[n][i];
    }

    grad_b[j] += dZ[n][j];
}
}

// update params
for (int j = 0; j < nOut; j++) {
    for (int i = 0; i < nIn; i++) {
        W[j][i] -= learningRate * grad_W[j][i] / minibatchSize;
    }
    b[j] -= learningRate * grad_b[j] / minibatchSize;
}

return dZ;
}

```

You might think the algorithm is complex and difficult because the arguments seem complicated, but what we do here is almost the same as what we do with the `train` method of logistic regression: we calculate the gradients of `w` and `b` with the unit of the mini-batch and update the model parameters. That's it. So, can an MLP learn the XOR problem? Check the result by running `MultiLayerPerceptrons.java`.

The result only outputs the percentages of the accuracy, precision, and recall of the model, but for example, if you dump the prediction data with the predict method of LogisticRegression, you can see how much it actually predicts the probability, as follows:

```
double[] y = output(x); // activate input data through learned  
networks  
Integer[] t = new Integer[nOut]; // output is the probability, so cast  
it to label  
  
System.out.println( Arrays.toString(y) );
```

We've just shown that MLPs can approximate the function of XOR. Moreover, it is proven that MLPs can approximate any functions. We don't follow the math details here, but you can easily imagine that the more units MLPs have, the more complicated functions they could express and approximate.

Summary

In this chapter, as preparation for deep learning, we dug into neural networks, which are one of the algorithms of machine learning. You learned about three representative algorithms of single-layer neural networks: perceptrons, logistic regression, and multi-class logistic regression. We see that single-layer neural networks can't solve nonlinear problems, but this problem can be solved with multi-layer neural networks – the networks with a hidden layer(s) between the input layer and output layer. An intuitive understanding of why MLPs can solve nonlinear problems says that the networks can learn more complicated logical operations by adding layers and increasing the number of units, and thus having the ability to express more complicated functions. The key to letting the model have this ability is the backpropagation algorithm. By backpropagating the error of the output to the whole network, the model is updated and adjusted to fit in the training data with each iteration, and finally optimized to approximate the function for the data.

In the next chapter, you'll learn the concepts and algorithms of deep learning. Since you've now acquired a foundational understanding of machine learning algorithms, you'll have no difficulty learning about deep learning.

3

Deep Belief Nets and Stacked Denoising Autoencoders

From this chapter through to the next chapter, you are going to learn the algorithms of deep learning. We'll follow the fundamental math theories step by step to fully understand each algorithm. Once you acquire the fundamental concepts and theories of deep learning, you can easily apply them to practical applications.

In this chapter, the topics you will learn about are:

- Reasons why deep learning could be a breakthrough
- The differences between deep learning and past machine learning (neural networks)
- Theories and implementations of the typical algorithms of deep learning, **deep belief nets (DBN)**, and **Stacked Denoising Autoencoders (SDA)**

Neural networks fall

In the previous chapter, you learned about the typical algorithm of neural networks and saw that nonlinear classification problems cannot be solved with perceptrons but can be solved by making multi-layer modeled neural networks. In other words, nonlinear problems can be learned and solved by inserting a hidden layer between the input and output layer. There is nothing else to it; but by increasing the number of neurons in a layer, the neural networks can express more patterns as a whole. If we ignore the time cost or an over-fitting problem, theoretically, neural networks can approximate any function.

So, can we think this way? If we increase the number of hidden layers—accumulate hidden layers over and over—can neural networks solve any complicated problem? It's quite natural to come up with this idea. And, as a matter of course, this idea has already been examined. However, as it turns out, this trial didn't work well. Just accumulating layers didn't make neural networks solve the world's problems. On the contrary, some cases have less accuracy when predicting than others with fewer layers.

Why do these cases happen? It's not wrong for neural networks with more layers to have more expression. So, where is the problem? Well, it is caused because of the feature that learning algorithms have in feed-forward networks. As we saw in the previous chapter, the backpropagation algorithm is used to propagate the learning error into the whole network efficiently with the multi-layer neural networks. In this algorithm, an error is reversed in each layer of the neural network and is conveyed to the input layer one by one in order. By backpropagating the error at the output layer to the input layer, the weight of the network is adjusted at each layer in order and the whole weight of a network is optimized.

This is where the problem occurs. If the number of layers of a network is small, an error backpropagating from an output layer can contribute to adjusting the weights of each layer well. However, once the number of layers increases, an error gradually disappears every time it backpropagates layers, and doesn't adjust the weight of the network. At a layer near the input layer, an error is not fed back at all.

The neural networks where the link among layers is dense have an inability to adjust weights. Hence, the weight of the whole of the networks cannot be optimized and, as a matter of course, the learning cannot go well. This serious problem is known as the **vanishing gradient problem** and has troubled researchers as a huge problem that the neural network had for a long time until deep learning showed up. The neural network algorithm reached a limit at an early stage.

Neural networks' revenge

Because of the vanishing gradient problem, neural networks lost their popularity in the field of machine learning. We can say that the number of cases used for data mining in the real world by neural networks was remarkably small compared to other typical algorithms such as logistic regression and SVM.

But then deep learning showed up and broke all the existing conventions. As you know, deep learning is the neural network accumulating layers. In other words, it is deep neural networks, and it generates astounding predictability in certain fields. Now, speaking of AI research, it's no exaggeration to say that it's the research into deep neural networks. Surely it's the counterattack by neural networks. If so, why didn't the vanishing gradient problem matter in deep learning? What's the difference between this and the past algorithm?

In this section, we'll look at why deep learning can generate such predictability and its mechanisms.

Deep learning's evolution – what was the breakthrough?

We can say that there are two algorithms that triggered deep learning's popularity. The first one, as mentioned in *Chapter 1, Deep Learning Overview*, is DBN pioneered by Professor Hinton (<https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf>). The second one is SDA, proposed by Vincent et al. (http://www.iro.umontreal.ca/~vincentp/Publications/denoising_autoencoders_tr1316.pdf). SDA was introduced a little after the introduction of DBN. It also recorded high predictability even with deep layers by taking a similar approach to DBN, although the details of the algorithm are different.

So, what is the common approach that solved the vanishing gradient problem? Perhaps you are nervously preparing to solve difficult equations in order to understand DBN or SDA, but don't worry. DBN is definitely an algorithm that is understandable. On the contrary, the mechanism itself is really simple. Deep learning was established by a very simple and elegant solution. The solution is: **layer-wise training**. That's it. You might think it's obvious if you see it, but this is the approach that made deep learning popular.

As mentioned earlier, in theory if there are more units or layers of neural networks, it should have more expressions and increase the number of problems it is able to solve. It doesn't work well because an error cannot be fed back to each layer correctly and parameters, as a whole network, cannot be adjusted properly. This is where the innovation was brought in for learning at a respective layer. Because each layer adjusts the weights of the networks independently, the whole network (that is, the parameters of the model) can be optimized properly even though the numbers of layers are piled up.

Previous models didn't go well because they tried to backpropagate errors from an output layer to an input layer straight away and tried to optimize themselves by adjusting the weights of the network with backpropagated errors. So, the algorithm shifted to layer-wise training and then the model optimization went well. That's what the breakthrough was for deep learning.

However, although we simply say **layer-wise training**, we need techniques for how to implement the learning. Also, as a matter of course, parameter adjustments for whole networks can't only be done with layer-wise training. We need the final adjustment. This phase of layer-wise training is called **pre-training** and the last adjustment phase is called **fine-tuning**. We can say that the bigger feature introduced in DBN and SDA is pre-training, but these two features are both part of the the necessary flow of deep learning. How do we do pre-training? What can be done in fine-tuning? Let's take a look at these questions one by one.

Deep learning with pre-training

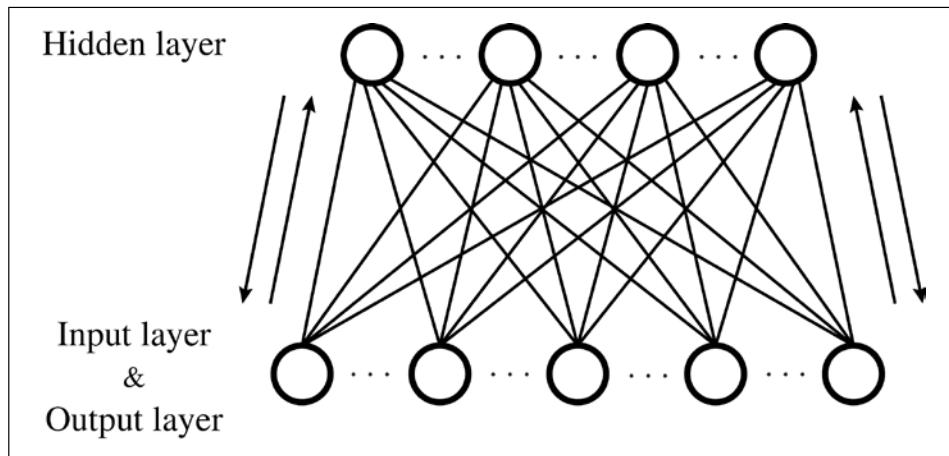
Deep learning is more like neural networks with accumulated hidden layers. The layer-wise training in pre-training undertakes learning at each layer. However, you might still have the following questions: if both layers are hidden (that is, neither of the layers are input nor output layers), then how is the training done? What can the input and output be?

Before thinking of these questions, remind yourself of the following point again (reiterated persistently): deep learning is neural networks with piled up layers. This mean, model parameters are still the weights of the network (and bias) in deep learning. Since these weights (and bias) need to be adjusted among each layer, in the standard three layered neural network (that is, the input layer, the hidden layer, and the output layer), we need to optimize only the weights between the input layer and the hidden layer and between the hidden layer and the output layer. In deep learning, however, the weight between two hidden layers also needs to be adjusted.

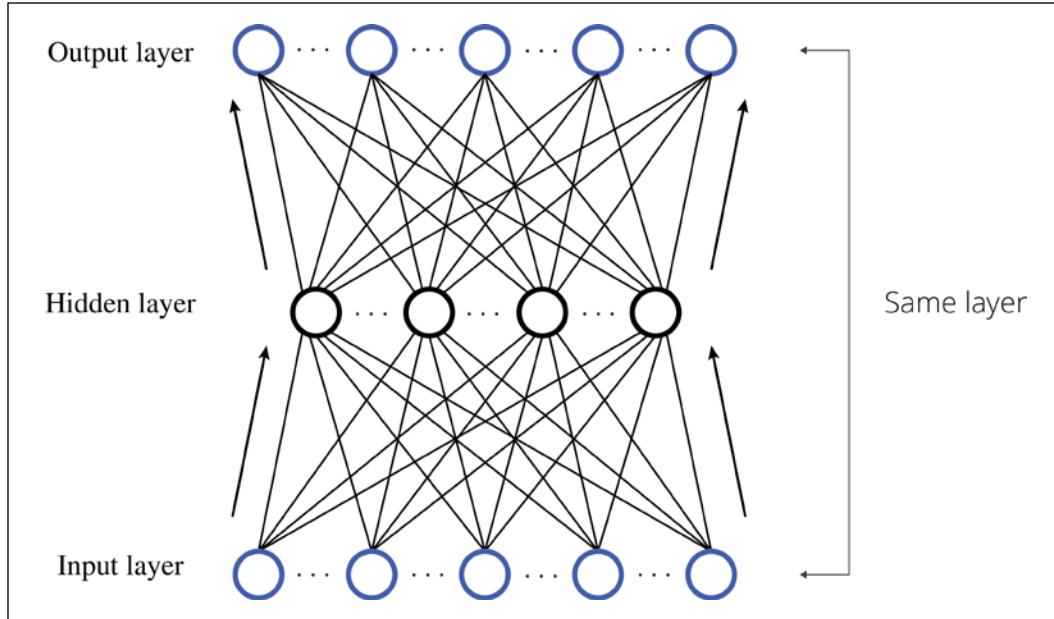
First of all, let's think about the input of a layer. You can imagine this easily with a quick thought. The value propagated from the previous layer will become the input as it is. The value propagated from the previous layer is none other than the value forward propagated from the previous layers to the current layer by using the weight of the network, the same as in general feed-forward networks. It looks simple in writing, but you can see that it has an important meaning if you step into it further and try to understand what it means. The value from the previous layer becomes the input, which means that the features the previous layer(s) learned become the input of the current layer, and from there the current layer newly learns the feature of the given data. In other words, in deep learning, features are learned from the input data in stages (and semi-automatically). This implies a mechanism where the deeper a layer becomes, the higher the feature it learns. This is what normal multi-layer neural networks couldn't do and the reason why it is said "a machine can learn a concept."

Now, let's think about the output. Please bear in mind that thinking about the output means thinking about how it learns. DBN and SDA have completely different approaches to learning, but both fill the following condition: to learn in order to equate output values and input values. You might think "What are you talking about?" but this is the technique that makes deep learning possible.

The value comes and goes back to the input layer through the hidden layer, and the technique is to adjust the weight of the networks (that is, to equate the output value and the input value) to eliminate the error at that time. The graphical model can be illustrated as follows:



It looks different from standard neural networks at a glance, but there's nothing special. If we intentionally draw the diagram of the input layer and the output layer separately, the mechanism is the same shape as the normal neural network:

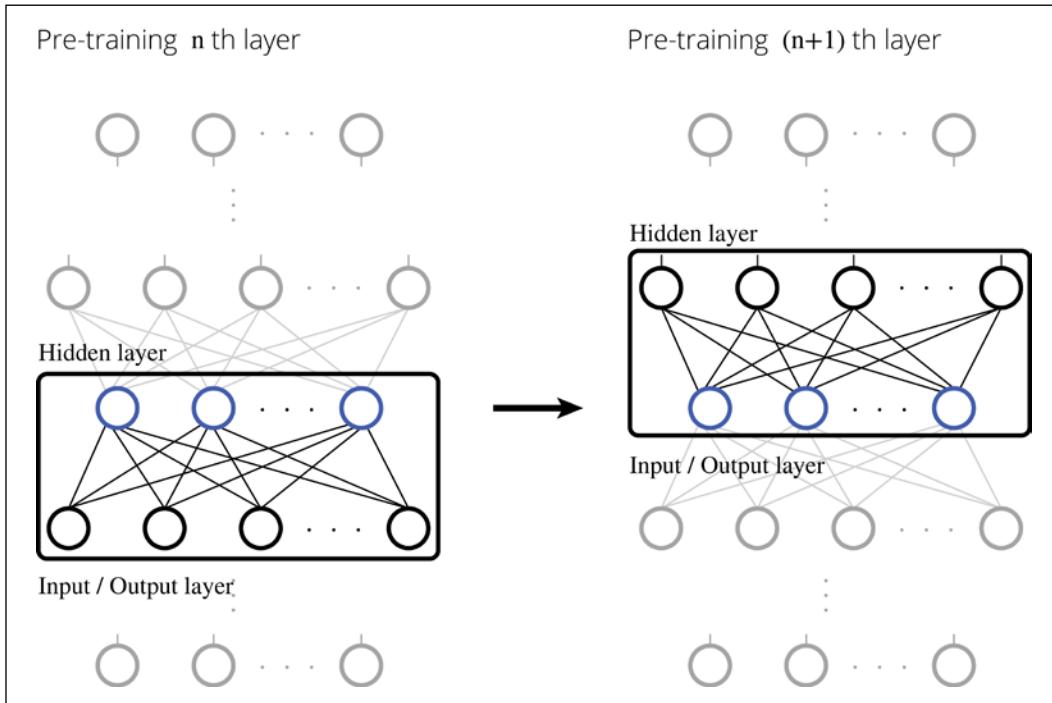


For a human, this action of *matching input and output* is not intuitive, but for a machine it is a valid action. If so, how it can learn features from input data by matching the output layer and input layer?

Need a little explanation? Let's think about it this way: in the algorithm of machine learning, including neural networks, learning intends to minimize errors between the model's prediction output and the dataset output. The mechanism is to remove an error by finding a pattern from the input data and making data with a common pattern the same output value (for example, 0 or 1). What would then happen if we turned the output value into the input value?

When we look at problems that should be solved as a whole through deep learning, input data is, fundamentally, a dataset that can be divided into some patterns. This means that there are some common features in the input data. If so, in the process of learning where each output value becomes respective input data, the weight of networks should be adjusted to focus more on the part that reflects the common features. And, even within the data categorized in the same class, learning should be processed to reduce weight on the non-common feature part, that is, the noise part.

Now you should understand what the input and output is in a certain layer and how learning progresses. Once the pre-training is done at a certain layer, the network moves on to learning in the next layer. However, as you can see in the following images, please also keep in mind that a hidden layer becomes an input layer when the network moves to learning in the next layer:



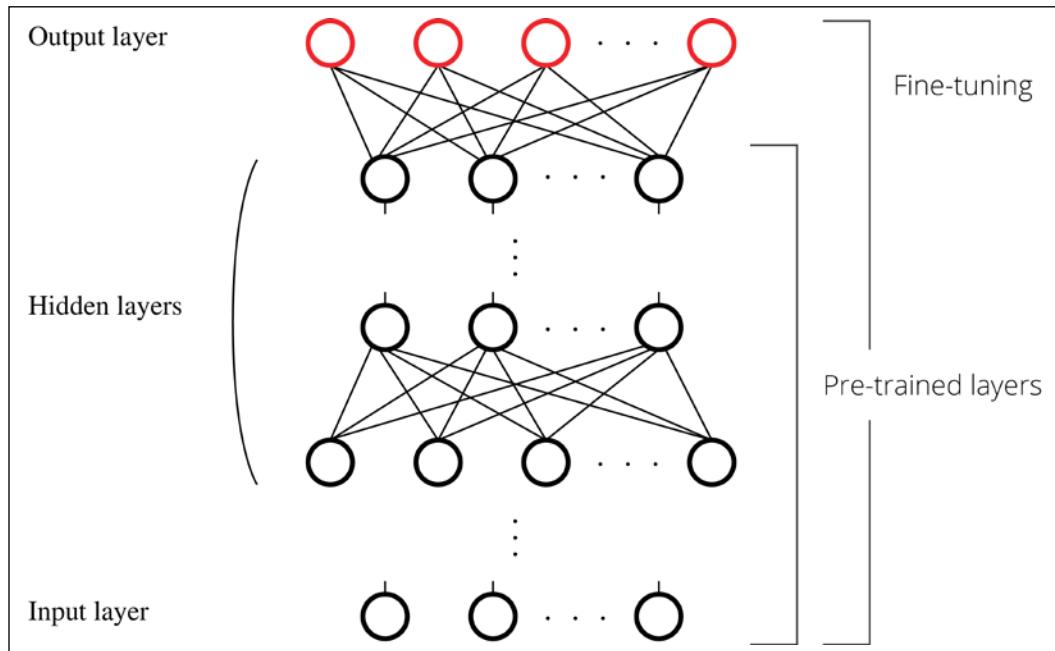
The point here is that the layer after the pre-training can be treated as normal feed-forward neural networks where the weight of the networks is adjusted. Hence, if we think about the input value, we can simply calculate the value forward propagated from the input layer to the current layer through the network.

Up to now, we've looked through the flow of pre-training (that is, layer-wise training). In the hidden layers of deep neural networks, features of input data are extracted in stages through learning where the input matches the output. Now, some of you might be wondering: I understand that features can be learned in stages from input data by pre-training, but that alone doesn't solve the classification problem. So, how can it solve the classification problem?

Well, during pre-training, the information pertaining to which data belongs to which class is not provided. This means the pre-training is unsupervised training and it just analyzes the hidden pattern using only input data. This is meaningless if it can't be used to solve the problem however it extracts features. Therefore, the model needs to complete one more step to solve classification problems properly. That is fine-tuning. The main roles of fine-tuning are the following:

1. To add an output layer to deep neural networks that completed pre-training and to perform supervised training.
2. To do final adjustments for the whole deep neural network.

This can be illustrated as follows:

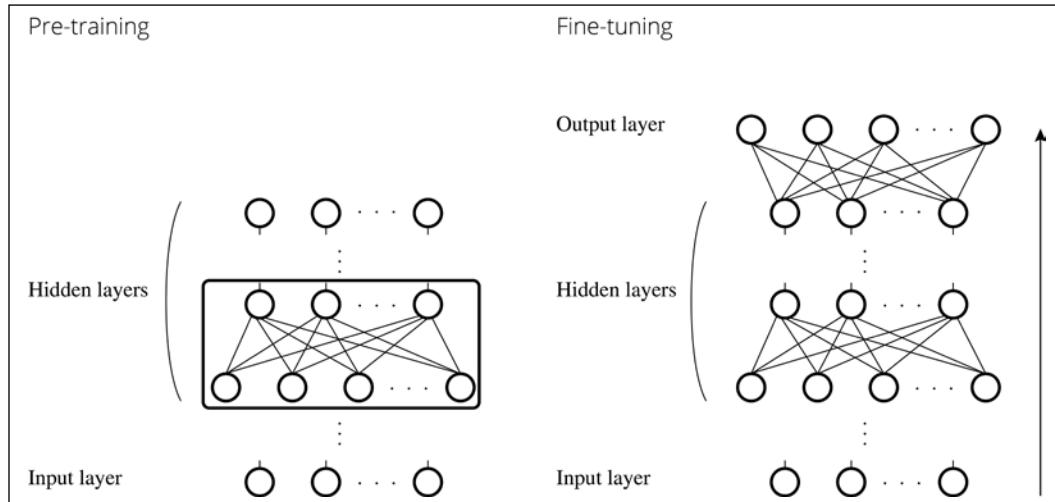


The supervised training in an output layer uses a machine learning algorithm, such as logistic regression or SVM. Generally, logistic regression is used more often considering the balance of the amount of calculation and the precision gained.

In fine-tuning, sometimes only the weights of an output layer will be adjusted, but normally the weights of whole neural networks, including the layer where the weights have been adjusted in pre-training, will also be adjusted. This means the standard learning algorithm, or in other words the backpropagation algorithm, is applied to the deep neural networks just as one multi-layer neural network. Thus, the model of neural networks with the problem of solving more complicated classification is completed.

Even so, you might have the following questions: why does learning go well with the standard backpropagation algorithm even in multi-layer neural networks where layers are piled up? Doesn't the vanishing gradient problem occur? These questions can be solved by pre-training. Let's think about the following: in the first place, the problem is that the weights of each network are not correctly adjusted due to improperly fed back errors in multi-layer neural networks without pre-training; in other words, the multi-layer neural networks where the vanishing gradient problem occurs. On the other hand, once the pre-training is done, the learning starts from the point where the weight of the network is almost already adjusted. Therefore, a proper error can be propagated to a layer close to an input layer. Hence the name fine-tuning. Thus, through pre-training and fine-tuning, eventually deep neural networks become neural networks with increased expression by having deep layers.

From the next section onwards, we will finally look through the theory and implementation of DBN and SDA, the algorithms of deep learning. But before that, let's look back at the flow of deep learning once again. Below is the summarized diagram of the flow:



The parameters of the model are optimized layer by layer during pre-training and then adjusted as single deep neural networks during fine-tuning. Deep learning, the breakthrough of AI, is a very simple algorithm.

Deep learning algorithms

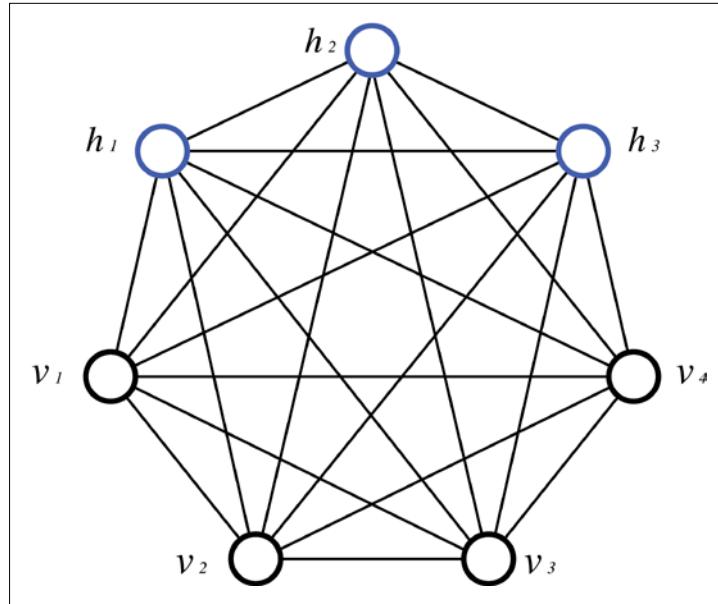
Now, let's look through the theory and implementation of deep learning algorithms. In this chapter, we will see DBN and SDA (and the related methods). These algorithms were both researched explosively, mainly between 2012 and 2013 when deep learning started to spread out rapidly and set the trend of deep learning on fire. Even though there are two methods, the basic flow is the same and consistent with pre-training and fine-tuning, as explained in the previous section. The difference between these two is which pre-training (that is, unsupervised training) algorithm is applied to them.

Therefore, if there could be difficult points in deep learning, it should be the theory and equation of the unsupervised training. However, you don't have to be afraid. All the theories and implementations will be explained one by one, so please read through the following sections carefully.

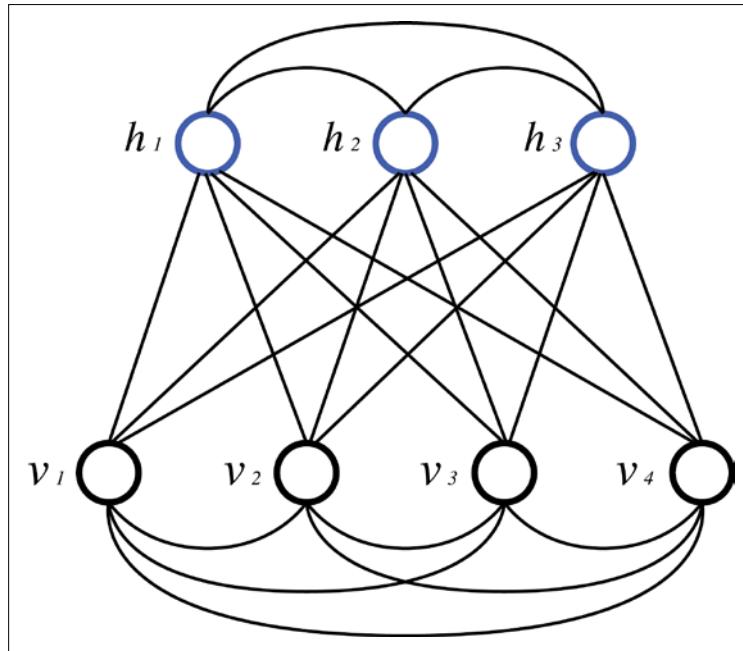
Restricted Boltzmann machines

The method used in the layer-wise training of DBN, pre-training, is called **Restricted Boltzmann Machines (RBM)**. To begin with, let's take a look at the RBM that forms the basis of DBN. As RBM stands for Restricted Boltzmann Machines, of course there's a method called **Boltzmann Machines (BMs)**. Or rather, BMs are a more standard form and RBM is the special case of them. Both are one of the neural networks and both were proposed by Professor Hinton.

The implementation of RBM and DBNs can be done without understanding the detailed theory of BMs, but in order to understand these concepts, we'll briefly look at the idea BMs are based on. First of all, let's look at the following figure, which shows a graphical model of BMs:



BMs look intricate because they are fully connected, but they are actually just simple neural networks with two layers. By rearranging all the units in the networks to get a better understanding, BMs can be shown as follows:

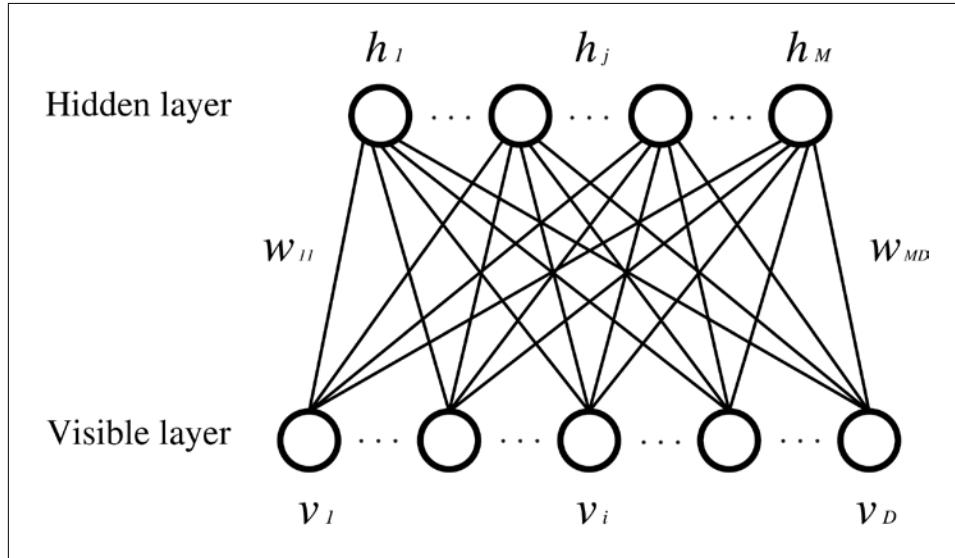


Please bear in mind that normally the input/output layer is called the **visible layer** in BMs and RBMs (a hidden layer is commonly used as it is), for it is the networks that presume the hidden condition (unobservable condition) from the observable condition. Also, the neurons of the visible layer are called **visible units** and the neurons of the hidden layer are called **hidden units**. Signs in the previous figure are described to match the given names.

As you can see in the diagram, the structure of BMs is not that different from standard neural networks. However, its way of thinking has a big feature. The feature is to adopt the concept of *energy* in neural networks. Each unit has a stochastic state respectively and the whole of the networks' energy is determined depending on what state each unit takes. (The first model that adopted the concept of energy in networks is called the **Hopfield network**, and BMs are the developed version of it. Since details of the Hopfield network are not totally relevant to deep learning, it is not explained in this book.) The condition that memorizes the correct data is the steady state of networks and the least amount of energy these networks have. On the other hand, if data with noise is provided to the network, each unit has a different state, but not a steady state, hence its condition makes the transition to stabilize the whole network, in other words, to transform it into a steady state.

This means that the weights of the model are adjusted and the state of each unit is transferred to minimize the energy function the networks have. These operations can remove the noise and extract the feature from inputs as a whole. Although the energy of networks sounds enormous, it's not too difficult to imagine because minimizing the energy function has the same effect as minimizing the error function.

The concept of BMs was wonderful, but various problems occurred when BMs were actually applied to practical problems. The biggest problem was that BMs are fully connected networks and take an enormous amount of calculation time. Therefore, RBM was devised. RBM is the algorithm that can solve various problems in a realistic time frame by making BMs restricted. Just as in BM, RBM is a model based on the energy of a network. Let's look at RBM in the diagram below:



Here, D is the number of units in the visible layer and M the number of units in the hidden layer. v_i denotes the value of a visible unit, h_j the value of a hidden unit, and w_{ji} the weight between these two units. As you can see, the difference between BM and RBM is that RBM doesn't have connections between the same layer. Because of this restriction, the amount of calculation decreases and it can be applied to realistic problems.

Now, let's look through the theory.



Be careful that, as a prerequisite, the value that each visible unit and hidden unit in RBM can take is generally $\{0, 1\}$, that is, binary (this is the same as BMs).



If we expand the theory, it can also handle continuous values. However, this could make equations complex, where it's not the core of the theory and where it's implemented with binary in the original DBN proposed by Professor Hinton. Therefore, we'll also implement binary RBM in this book. RBM with binary inputs is sometimes called **Bernoulli RBM**.

RBM is the energy-based model, and the status of a visible layer or hidden layer is treated as a stochastic variable. We'll look at the equations in order. First of all, each visible unit is propagated to the hidden units throughout a network. At this time, each hidden unit takes a binary value based on the probability distribution generated in accordance with its propagated inputs:

$$p(h_j = 1 | v) = \sigma \left(\sum_{i=1}^D w_{ij} v_i + c_j \right)$$

Here, c_j is the bias in a hidden layer and σ denotes the sigmoid function.

This time, it was conversely propagated from a hidden layer to a visible layer through the same network. As in the previous case, each visible unit takes a binary value based on probability distribution generated in accordance with propagated values.

$$p(v_i = 1 | h) = \sigma \left(\sum_{j=1}^M w_{ij} h_j + b_i \right)$$

Here, b_i is the bias of the visible layer. This value of visible units is expected to match the original input values. This means if W , the weight of the network as a model parameter, and b, c , the bias of a visible layer and a hidden layer, are shown as a vector parameter, θ , it leans θ in order for the probability $p(v|\theta)$ that can be obtained above to get close to the distribution of v .

For this learning, the energy function, that is, the evaluation function, needs to be defined. The energy function is shown as follows:

$$\begin{aligned} E(v, h) &= -b^T v - c^T h - h^T W v \\ &= -\sum_{i=1}^D b_i v_i - \sum_{j=1}^M c_j h_j - \sum_{j=1}^M \sum_{i=1}^D h_j w_{ij} v_i \end{aligned}$$

Also, the joint probability density function showing the demeanor of a network can be shown as follows:

$$p(v, h) = \frac{1}{Z} \exp(-E(v, h))$$

$$Z = \sum_{v,h} \exp(-E(v, h))$$

From the preceding formulas, the equations for the training of parameters will be determined. We can get the following equation:

$$p(v | \theta) = \sum_h P(v, h) = \frac{1}{Z} = \sum_h \exp(-E(v, h))$$

Hence, the **log likelihood** can be shown as follows:

$$\begin{aligned} \ln L(\theta | v) &= \ln p(v | \theta) \\ &= \ln \frac{1}{Z} \sum_h \exp(-E(v, h)) \\ &= \ln \frac{1}{Z} \sum_h \exp(-E(v, h)) - \ln \sum_{v,h} \exp(-E(v, h)) \end{aligned}$$

Then, we'll calculate each gradient against the model parameter. The derivative can be calculated as follows:

$$\begin{aligned}
 \frac{\partial \ln L(\theta | v)}{\partial \theta} &= \frac{\partial}{\partial \theta} \left(\ln \sum_h \exp(-E(v, h)) \right) - \frac{\partial}{\partial \theta} \left(\ln \sum_{v,h} \exp(-E(v, h)) \right) \\
 &= -\frac{1}{\sum_h \exp(-E(v, h))} \sum_h \exp(-E(v, h)) \frac{\partial E(v, h)}{\partial \theta} \\
 &= +\frac{1}{\sum_h \exp(-E(v, h))} \sum_{v,h} \exp(-E(v, h)) \frac{\partial E(v, h)}{\partial \theta} \\
 &= -\sum_h p(h | v) \frac{\partial E(v, h)}{\partial \theta} + \sum_{v,h} p(v | h) \frac{\partial E(v, h)}{\partial \theta}
 \end{aligned}$$

Some equations in the middle are complicated, but it turns out to be simple with the term of the probability distribution of the model and the original data.

Therefore, the gradient of each parameter is shown as follows:

$$\begin{aligned}
 \frac{\partial \ln L(\theta | v)}{\partial w_{ij}} &= \sum_h p(h | v) \frac{\partial E(v, h)}{\partial w_{ij}} + \sum_{v,h} p(h | v) \frac{\partial E(v, h)}{\partial w_{ij}} \\
 &= \sum_h p(h | v) h_j v_i - \sum_v p(v) \sum_h p(h | v) h_j v_i \\
 &= p(H_j = 1 | v) v_i - \sum_v p(v) p(H_j = 1 | v) v_i
 \end{aligned}$$

$$\frac{\partial \ln L(\theta | v)}{\partial b_i} = v_i - \sum_v p(v) v_i$$

$$\frac{\partial \ln L(\theta | v)}{\partial c_j} = p(H_j = 1 | v) - \sum_v p(v) p(H_j = 1 | v)$$

Now then, we could find the equation of the gradient, but a problem occurs when we try to apply this equation as it is. Think about the term of $\sum_v p(v)$. This term implies that we have to calculate the probability distribution for all the {0, 1} patterns, which can be assumed as input data that includes patterns that don't actually exist in the data.

We can easily imagine how this term can cause a combinatorial explosion, meaning we can't solve it within a realistic time frame. To solve this problem, the method for approximating data using Gibbs sampling, called **Contrastive Divergence (CD)**, was introduced. Let's look at this method now.

Here, $v^{(0)}$ is an input vector. Also, $v^{(k)}$ is an input (output) vector that can be obtained by sampling for k-times using this input vector.

Then, we get:

$$h_j^{(k)} \sim p(h_j | v^{(k)})$$

$$h_i^{(k+1)} \sim p(v_i | h^{(k)})$$

Hence, when approximating $p(v)$ after reiterating Gibbs sampling, the derivative of the log likelihood function can be represented as follows:

$$\begin{aligned} \frac{\partial \ln L(\theta | v)}{\partial \theta} &= -\sum_h p(h | v) \frac{\partial E(v, h)}{\partial \theta} + \sum_{v, h} p(v, h) \frac{\partial E(v, h)}{\partial \theta} \\ &\approx -\sum_h p(h | v^{(0)}) \frac{\partial E(v, h)}{\partial \theta} \sum_{v, h} p(h, v^{(k)}) \frac{\partial E(v^{(k)}, h)}{\partial \theta} \end{aligned}$$

Therefore, the model parameter can be shown as follows:

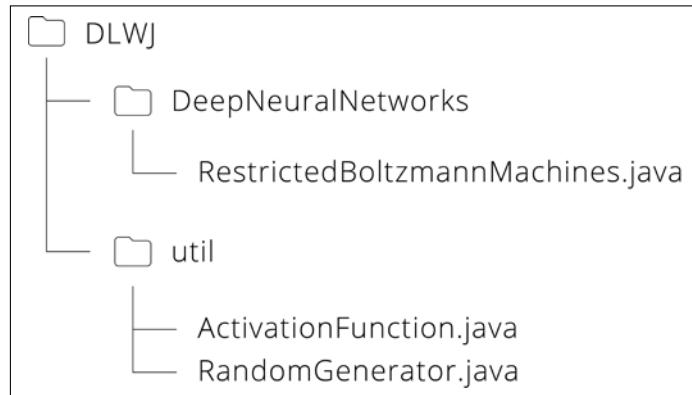
$$w_{ij}^{(\tau+1)} = w_{ij}^{(\tau)} + \eta \left(p(H_j = 1 | v^{(0)}) v_i^{(0)} - p(H_j = 1 | v^k) v_i^k \right)$$

$$b_i^{(\tau+1)} = b_i^{(\tau)} + \eta \left(v_i^{(0)} - v_i^k \right)$$

$$c_j^{(\tau+1)} = c_j^{(\tau)} + \eta \left(p(H_j = 1 | v^{(0)}) - p(H_j = 1 | v^k) \right)$$

Here, τ is the number of iterations and η is the learning rate. As shown in the preceding formulas, generally, CD that performs sampling k-times is shown as CD-k. It's known that CD-1 is sufficient when applying the algorithm to realistic problems.

Now, let's go through the implementation of RMB. The package structure is as shown in the following screenshot:



Let's look through the `RestrictedBoltzmannMachines.java` file. Because the first part of the main method just defines the variables needed for a model and generates demo data, we won't look at it here.

So, in the part where we generate an instance of a model, you may notice there are many null values in arguments:

```

// construct RBM
RestrictedBoltzmannMachines nn = new
RestrictedBoltzmannMachines(nVisible, nHidden, null, null, null,
rng);
    
```

When you look at the constructor, you might know that these null values are the RBM's weight matrix, bias of hidden units, and bias of visible units. We define arguments as null here because they are for DBN's implementation. In the constructor, these are initialized as follows:

```

if (W == null) {

    W = new double[nHidden] [nVisible];
    double w_ = 1. / nVisible;

    for (int j = 0; j < nHidden; j++) {
        for (int i = 0; i < nVisible; i++) {
            W[j][i] = uniform(-w_, w_, rng);
        }
    }
}

if (hbias == null) {
    hbias = new double[nHidden];

    for (int j = 0; j < nHidden; j++) {
        hbias[j] = 0.;
    }
}

if (vbias == null) {
    vbias = new double[nVisible];

    for (int i = 0; i < nVisible; i++) {
        vbias[i] = 0.;
    }
}

```

The next step is training. CD-1 is applied for each mini-batch:

```

// train with contrastive divergence
for (int epoch = 0; epoch < epochs; epoch++) {
    for (int batch = 0; batch < minibatch_N; batch++) {
        nn.contrastiveDivergence(train_X_minibatch[batch],
        minibatchSize, learningRate, 1);
    }
    learningRate *= 0.995;
}

```

Now, let's look into the essential point of RBM, the contrastiveDivergence method. CD-1 can obtain a sufficient solution when we actually run this program (and so we have k = 1 in the demo), but this method is defined to deal with CD-k as well:

```
// CD-k : CD-1 is enough for sampling (i.e. k = 1)
sampleHgivenV(X[n], phMean_, phSample_);

for (int step = 0; step < k; step++) {

    // Gibbs sampling
    if (step == 0) {
        gibbsHVH(phSample_, nvMeans_, nvSamples_, nhMeans_,
nhSamples_);
    } else {
        gibbsHVH(nhSamples_, nvMeans_, nvSamples_, nhMeans_,
nhSamples_);
    }

}
```

It appears that two different types of method, `sampleHgivenV` and `gibbsHVH`, are used in CD-k, but when you look into `gibbsHVH`, you can see:

```
public void gibbsHVH(int[] h0Sample, double[] nvMeans, int[]
nvSamples, double[] nhMeans, int[] nhSamples) {
    sampleVgivenH(h0Sample, nvMeans, nvSamples);
    sampleHgivenV(nvSamples, nhMeans, nhSamples);
}
```

So, CD-k consists of only two methods for sampling, `sampleVgivenH` and `sampleHgivenV`.

As the name of the method indicates, `sampleHgivenV` is the method that sets the probability distribution and sampling data generated in a hidden layer based on the given value of visible units and vice versa:

```
public void sampleHgivenV(int[] v0Sample, double[] mean, int[] sample)
{
    for (int j = 0; j < nHidden; j++) {
        mean[j] = propup(v0Sample, W[j], hbias[j]);
        sample[j] = binomial(1, mean[j], rng);
    }
}
```

```

public void sampleVgivenH(int[] h0Sample, double[] mean, int[] sample)
{
    for(int i = 0; i < nVisible; i++) {
        mean[i] = propdown(h0Sample, i, vbias[i]);
        sample[i] = binomial(1, mean[i], rng);
    }
}

```

The `propup` and `propdown` tags that set values to respective means are the method that activates values of each unit by the sigmoid function:

```

public double propup(int[] v, double[] w, double bias) {

    double preActivation = 0.;

    for (int i = 0; i < nVisible; i++) {
        preActivation += w[i] * v[i];
    }
    preActivation += bias;

    return sigmoid(preActivation);
}

public double propdown(int[] h, int i, double bias) {

    double preActivation = 0.;

    for (int j = 0; j < nHidden; j++) {
        preActivation += W[j][i] * h[j];
    }
    preActivation += bias;

    return sigmoid(preActivation);
}

```

The `binomial` method that sets a value to a sample is defined in `RandomGenerator.java`. The method returns 0 or 1 based on the binomial distribution. With this method, a value of each unit becomes binary:

```

public static int binomial(int n, double p, Random rng) {
    if(p < 0 || p > 1) return 0;

    int c = 0;

```

```
    double r;

    for(int i=0; i<n; i++) {
        r = rng.nextDouble();
        if (r < p) c++;
    }

    return c;
}
```

Once approximated values are obtained by sampling, what we need to do is just calculate the gradient of a model parameter and renew a parameter using a mini-batch. There's nothing special here:

```
// calculate gradients
for (int j = 0; j < nHidden; j++) {
    for (int i = 0; i < nVisible; i++) {
        grad_W[j][i] += phMean_[j] * X[n][i] - nhMeans_[j] * nvSamples_
[i];
    }

    grad_hbias[j] += phMean_[j] - nhMeans_[j];
}

for (int i = 0; i < nVisible; i++) {
    grad_vbias[i] += X[n][i] - nvSamples_[i];
}

// update params
for (int j = 0; j < nHidden; j++) {
    for (int i = 0; i < nVisible; i++) {
        W[j][i] += learningRate * grad_W[j][i] / minibatchSize;
    }

    hbias[j] += learningRate * grad_hbias[j] / minibatchSize;
}

for (int i = 0; i < nVisible; i++) {
    vbias[i] += learningRate * grad_vbias[i] / minibatchSize;
}
```

Now we're done with the model training. Next comes the test and evaluation in general cases, but note that the model cannot be evaluated with barometers such as accuracy because RBM is a generative model. Instead, let's briefly look at how noisy data is changed by RBM here. Since RBM after training can be seen as a neural network, the weights of which are adjusted, the model can obtain reconstructed data by simply propagating input data (that is, noisy data) through a network:

```
public double[] reconstruct(int[] v) {

    double[] x = new double[nVisible];
    double[] h = new double[nHidden];

    for (int j = 0; j < nHidden; j++) {
        h[j] = propup(v, W[j], hbias[j]);
    }

    for (int i = 0; i < nVisible; i++) {
        double preActivation_ = 0.;

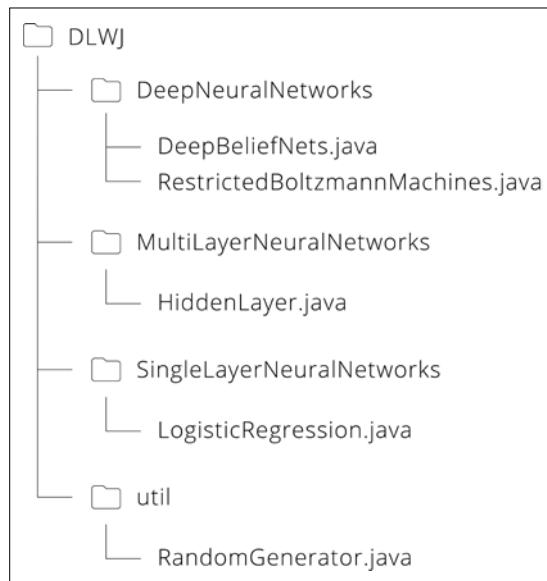
        for (int j = 0; j < nHidden; j++) {
            preActivation_ += W[j][i] * h[j];
        }
        preActivation_ += vbias[i];

        x[i] = sigmoid(preActivation_);
    }

    return x;
}
```

Deep Belief Nets (DBNs)

DBNs are deep neural networks where logistic regression is added to RBMs as the output layer. Since the theory necessary for implementation has already been explained, we can go directly to the implementation. The package structure is as follows:



The flow of the program is very simple. The order is as follows:

1. Setting up parameters for the model.
2. Building the model.
3. Pre-training the model.
4. Fine-tuning the model.
5. Testing and evaluating the model.

Just as in RBM, the first step in setting up the main method is the declaration of variables and the code for creating demo data (the explanation is omitted here).

Please check that in the demo data, the number of units for an input layer is 60, a hidden layer has 2 layers, their combined number of units is 20, and the number of units for an output layer is 3. Now, let's look through the code from the *Building the Model* section:

```
// construct DBN
System.out.print("Building the model...");
DeepBeliefNets classifier = new DeepBeliefNets(nIn, hiddenLayerSizes,
nOut, rng);
System.out.println("done.");
```

The variable of `hiddenLayerSizes` is an array and its length represents the number of hidden layers in deep neural networks. The deep learning algorithm takes a huge amount of calculation, hence the program gives us an output of the current status so that we can see which process is proceeding. The variable of `hiddenLayerSizes` is an array and its length represents the number of hidden layers in deep neural networks. Each layer is constructed in the constructor.



Please bear in mind that `sigmoidLayers` and `rbmLayers` are, of course, different objects but their weights and bias are shared.



This is because, as explained in the theory section, pre-training performs layer-wise training, whereas the whole model can be regarded as one neural network:

```
// construct multi-layer
for (int i = 0; i < nLayers; i++) {
    int nIn_;
    if (i == 0) nIn_ = nIn;
    else nIn_ = hiddenLayerSizes[i-1];

    // construct hidden layers with sigmoid function
    // weight matrices and bias vectors will be shared with RBM
    layers
    sigmoidLayers[i] = new HiddenLayer(nIn_, hiddenLayerSizes[i], null,
    null, rng, "sigmoid");

    // construct RBM layers
```

```
rbmLayers[i] = new RestrictedBoltzmannMachines(nIn_,  
hiddenLayerSizes[i], sigmoidLayers[i].W, sigmoidLayers[i].b, null,  
rng);  
}  
  
// logistic regression layer for output  
logisticLayer = new LogisticRegression(hiddenLayerSizes[nLayers-1],  
nOut);
```

The first thing to do after building the model is pre-training:

```
// pre-training the model  
System.out.print("Pre-training the model...");  
classifier.pretrain(train_X_minibatch, minibatchSize, train_  
minibatch_N, pretrainEpochs, pretrainLearningRate, k);  
System.out.println("done.");
```

Pre-training needs to be processed with each minibatch but, at the same time, with each layer. Therefore, all training data is given to the pretrain method first, and then the data of each mini-batch is processed in the method:

```
public void pretrain(int[][][] X, int minibatchSize, int minibatch_N,  
int epochs, double learningRate, int k) {  
  
    for (int layer = 0; layer < nLayers; layer++) { // pre-train  
        layer-wise  
        for (int epoch = 0; epoch < epochs; epoch++) {  
            for (int batch = 0; batch < minibatch_N; batch++) {  
  
                int[][] X_ = new int[minibatchSize][nIn];  
                int[][] prevLayerX_;  
  
                // Set input data for current layer  
                if (layer == 0) {  
                    X_ = X[batch];  
                } else {  
  
                    prevLayerX_ = X_;  
                    X_ = new int[minibatchSize]  
[hiddenLayerSizes[layer-1]];  
  
                    for (int i = 0; i < minibatchSize; i++) {  
                        X_[i] = sigmoidLayers[layer-1].  
outputBinomial(prevLayerX_[i], rng);  
                }
```

```
        }
    }

    rbmLayers[layer].contrastiveDivergence(x_,
minibatchSize, learningRate, k);
}

}
```

Since the actual learning is done through CD-1 of RBM, the description of DBN within the code is very simple. In DBN (RBM), units of each layer have binary values, so the output method of `HiddenLayer` cannot be used because it returns double. Hence, the `outputBinomial` method is added to the class, which returns the `int` type (the code is omitted here). Once the pre-training is complete, the next step is fine-tuning.



Be careful not to use training data that was used in the pre-training.

We can easily fall into overfitting if we use the whole data set for both pre-training and fine-tuning. Therefore, the validation data set is prepared separately from the training dataset and is used for fine-tuning:

```
// fine-tuning the model
System.out.print("Fine-tuning the model...");
for (int epoch = 0; epoch < finetuneEpochs; epoch++) {
    for (int batch = 0; batch < validation_minibatch_N; batch++) {
        classifier.finetune(validation_X_minibatch[batch],
            validation_T_minibatch[batch], minibatchSize,
            finetuneLearningRate);
    }
    finetuneLearningRate *= 0.98;
}
System.out.println("done.");
```

In the `finetune` method, the backpropagation algorithm in multi-layer neural networks is applied where the logistic regression is used for the output layer. To backpropagate unit values among multiple hidden layers, we define variables to maintain each layer's inputs:

```
public void finetune(double[][] X, int[][] T, int minibatchSize,
double learningRate) {

    List<double[][]> layerInputs = new ArrayList<>(nLayers + 1);
    layerInputs.add(X);

    double[][] Z = new double[0][0];
    double[][] dY;

    // forward hidden layers
    for (int layer = 0; layer < nLayers; layer++) {

        double[] x_; // layer input
        double[][] Z_ = new
        double[minibatchSize][hiddenLayerSizes[layer]];

        for (int n = 0; n < minibatchSize; n++) {

            if (layer == 0) {
                x_ = X[n];
            } else {
                x_ = Z[n];
            }

            Z_[n] = sigmoidLayers[layer].forward(x_);
        }

        Z = Z_.clone();
        layerInputs.add(Z.clone());
    }

    // forward & backward output layer
    dY = logisticLayer.train(Z, T, minibatchSize, learningRate);

    // backward hidden layers
    double[][] Wprev;
```

```

double[][] dZ = new double[0][0];

for (int layer = nLayers - 1; layer >= 0; layer--) {

    if (layer == nLayers - 1) {
        Wprev = logisticLayer.W;
    } else {
        Wprev = sigmoidLayers[layer+1].W;
        dY = dZ.clone();
    }

    dZ = sigmoidLayers[layer].backward(layerInputs.get(layer),
        layerInputs.get(layer+1), dY, Wprev, minibatchSize,
        learningRate);
}
}

```

The training part of DBN is just how it is seen in the preceding code. The hard part is probably the theory and implementation of RBM, so you might think it's not too hard when you just look at the code of DBN.

Since DBN after the training can be regarded as one (deep) neural network, you simply need to forward propagate data in each layer when you try to predict which class the unknown data belongs to:

```

public Integer[] predict(double[] x) {

    double[] z = new double[0];

    for (int layer = 0; layer < nLayers; layer++) {

        double[] x_;

        if (layer == 0) {
            x_ = x;
        } else {
            x_ = z.clone();
        }

        z = sigmoidLayers[layer].forward(x_);
    }

    return logisticLayer.predict(z);
}

```

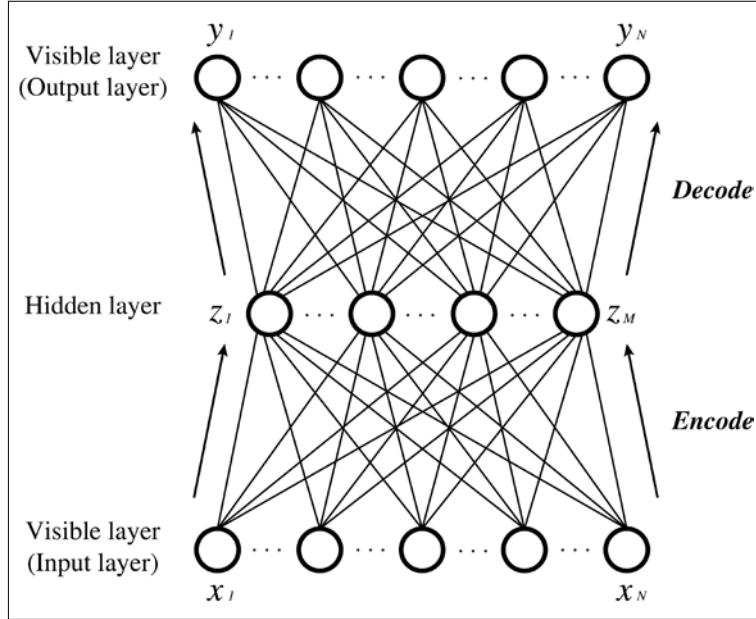
As for evaluation, no explanation should be needed because it's not much different from the previous classifier model.

Congratulations! You have now acquired knowledge of one of the deep learning algorithms. You might be able to understand it more easily than expected. However, the difficult part of deep learning is actually setting up the parameters, such as setting how many hidden layers there are, how many units there are in each hidden layer, the learning rate, the iteration numbers, and so on. There are way more parameters to set than in the method of machine learning. Please remember that you might find this point difficult when you apply this to a realistic problem.

Denoising Autoencoders

The method used in pre-training for SDA is called **Denoising Autoencoders (DA)**. It can be said that DA is the method that emphasizes the role of equating inputs and outputs. What does this mean? The processing content of DA is as follows: DA adds some noise to input data intentionally and partially damages the data, and then DA performs learning as it restores corrupted data to the original input data. This intentional noise can be easily substantiated if the input data value is $[0, 1]$; by turning the value of the relevant part into 0 compulsorily. If a data value is out of this range, it can be realized, for example, by adding Gaussian noise, but in this book, we'll think about the former $[0, 1]$ case to understand the core part of the algorithm.

In DA as well, an input/output layer is called a visible layer. DA's graphical model can be shown to be the same shape of RBM, but to get a better understanding, let's follow this diagram:



Here, \tilde{x} is the corrupted data, the input data with noise. Then, forward propagation to the hidden layer and the output layer can be represented as follows:

$$z_j = \sigma \left(\sum_{i=1}^N w_{ij} \tilde{x}_i + c_j \right)$$

$$y_i = \sigma \left(\sum_{j=1}^M w_{ji} z_j + b_i \right)$$

Here, c_j denotes the bias of the hidden layer and b_i the bias of the visible layer. Also, σ denotes the sigmoid function. As seen in the preceding diagram, corrupting input data and mapping to a hidden layer is called **Encode** and mapping to restore the encoded data to the original input data is called **Decode**. Then, DA's evaluation function can be denoted with a negative log likelihood function of the original input data and decoded data:

$$E := -\ln L(\theta) = -\sum_{i=1}^N \{x_i \ln y_i + (1-x_i) \ln (1-y_i)\}$$

Here, θ is the model parameter, the weight and the bias of the visible layer and the hidden layer. What we need to do is just find the gradients of these parameters against the evaluation function. To deform equations easily, we define the functions here:

$$h_j := \sum_{i=1}^N w_{ji} \tilde{x}_i + c_j$$

$$g_i := \sum_{j=1}^M w_{ji} z_j + b_i$$

Then, we get:

$$z_j = \sigma(h_j)$$

$$y_i = \sigma(g_i)$$

Using these functions, each gradient of a parameter can be shown as follows:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial h_j} \frac{\partial h_j}{\partial w_{ji}} + \frac{\partial E}{\partial g_i} \frac{\partial g_i}{\partial w_{ji}} = \frac{\partial E}{\partial h_j} \tilde{x}_i + \frac{\partial E}{\partial g_i} z_j$$

$$\frac{\partial E}{\partial b_i} = \frac{\partial E}{\partial g_i} \frac{\partial g_i}{\partial b_i} = \frac{\partial E}{\partial g_i}$$

$$\frac{\partial E}{\partial c_j} = \frac{\partial E}{\partial h_j} \frac{\partial h_j}{\partial c_j} = \frac{\partial E}{\partial h_j}$$

Therefore, only two terms are required. Let's derive them one by one:

$$\frac{\partial E}{\partial h_j} = \frac{\partial E}{\partial z_j} \frac{\partial z_j}{\partial h_j} = \frac{\partial E}{\partial z_j} z_j (1 - z_j)$$

Here, we utilized the derivative of the sigmoid function:

$$\frac{d}{dx} \sigma(x) = \sigma(x)(1 - \sigma(x))$$

Also, we get:

$$\begin{aligned} \frac{\partial E}{\partial z_j} &= \sum_{i=1}^N \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial z_j} \\ &= \sum_{i=1}^N w_{ji} (x_i - y_i) \end{aligned}$$

Therefore, the following equation can be obtained:

$$\frac{\partial E}{\partial h_j} = \left(\sum_{i=1}^N w_{ji} (x_i - y_i) \right) z_j (1 - z_j)$$

On the other hand, we can also get the following:

$$\begin{aligned} \frac{\partial E}{\partial g_i} &= \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial g_i} \\ &= x_i - y_i \end{aligned}$$

Hence, the renewed equation for each parameter will be as follows:

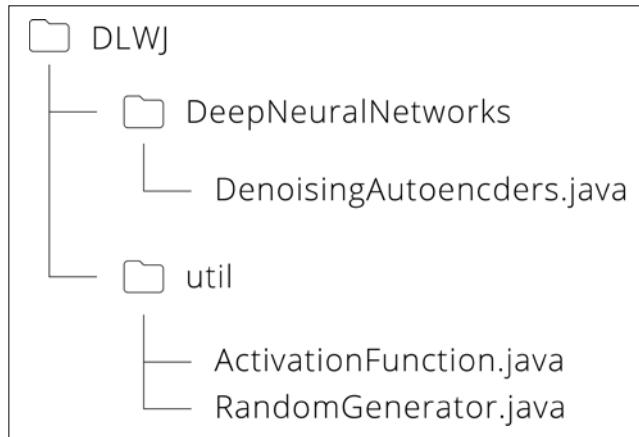
$$w_{ji}^{(k+1)} = w_{ji}^{(k)} + \eta \left[\left(\sum_{i=1}^N w_{ji}^{(k)} (x_i - y_i) \right) z_j (1 - z_j) \tilde{x}_i + (x_i - y_i) z_j \right]$$

$$b_i^{(k+1)} = b_i^k + \eta (x_i - y_i)$$

$$c_j^{(k+1)} = c_j^{(k)} + \eta \left(\sum_{i=1}^N w_{ji}^{(k)} (x_i - y_i) \right) z_j (1 - z_j)$$

Here, k is the number of iterations and η is the learning rate. Although DA requires a bit of technique for deformation, you can see that the theory itself is very simple compared to RBM.

Now, let's proceed with the implementation. The package structure is the same as the one for RBM.



As for model parameters, in addition to the number of units in a hidden layer, the amount of noise being added to the input data is also a parameter in DA. Here, the corruption level is set at 0.3. Generally, this value is often set at 0.1 ~ 0.3:

```
double corruptionLevel = 0.3;
```

The flow from the building model to training is the same as RBM. Although this method of training is called `contrastiveDivergence` in RBM, it's simply set as `train` in DA:

```
// construct DA
DenoisingAutoencoders nn = new DenoisingAutoencoders(nVisible,
nHidden, null, null, null, rng);

// train
for (int epoch = 0; epoch < epochs; epoch++) {
    for (int batch = 0; batch < minibatch_N; batch++) {
        nn.train(train_X_minibatch[batch], minibatchSize,
        learningRate, corruptionLevel);
    }
}
```

The content of `train` is as explained in the theory section. First of all, add noise to the input data, then encode and decode it:

```
// add noise to original inputs
double[] corruptedInput = getCorruptedInput(X[n], corruptionLevel);

// encode
double[] z = getHiddenValues(corruptedInput);

// decode
double[] y = getReconstructedInput(z);
```

The process of adding noise is, as previously explained, the compulsory turning of the value of the corresponding part of the data into 0:

```
public double[] getCorruptedInput(double[] x, double corruptionLevel)
{
    double[] corruptedInput = new double[x.length];

    // add masking noise
    for (int i = 0; i < x.length; i++) {
        double rand_ = rng.nextDouble();

        if (rand_ < corruptionLevel) {
            corruptedInput[i] = 0.;
        } else {
```

```
    corruptedInput[i] = x[i];
}
}

return corruptedInput;
}
```

The other processes are just simple activation and propagation, so we won't go through them here. The calculation of the gradients follows math equations:

```
// calculate gradients

// vbias
double[] v_ = new double[nVisible];

for (int i = 0; i < nVisible; i++) {
    v_[i] = X[n][i] - y[i];
    grad_vbias[i] += v_[i];
}

// hbias
double[] h_ = new double[nHidden];

for (int j = 0; j < nHidden; j++) {

    for (int i = 0; i < nVisible; i++) {
        h_[j] = W[j][i] * (X[n][i] - y[i]);
    }

    h_[j] *= z[j] * (1 - z[j]);
    grad_hbias[j] += h_[j];
}

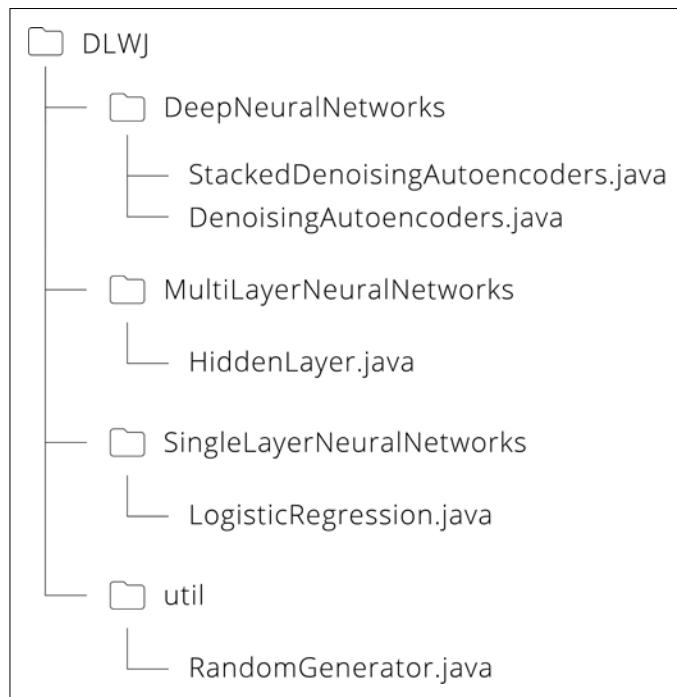
// W
for (int j = 0; j < nHidden; j++) {
    for (int i = 0; i < nVisible; i++) {
        grad_W[j][i] += h_[j] * corruptedInput[i] + v_[i] * z[j];
    }
}
```

Compared to RBM, the implementation of DA is also quite simple. When you test (reconstruct) the model, you don't need to corrupt the data. As in standard neural networks, you just need to forward propagate the given inputs based on the weights of the networks:

```
public double[] reconstruct(double[] x) {  
  
    double[] z = getHiddenValues(x);  
    double[] y = getReconstructedInput(z);  
  
    return y;  
}
```

Stacked Denoising Autoencoders (SDA)

SDA is deep neural networks with piled up DA layers. In the same way that DBN consists of RBMs and logistic regression, SDA consists of DAs and logistic regression:



The flow of implementation is not that different between DBN and SDA. Even though there is a difference between RBM and DA in pre-training, the content of fine-tuning is exactly the same. Therefore, not much explanation might be needed.

The method for pre-training is not that different, but please note that the point where the int type was used for DBN is changed to double type, as DA can handle [0, 1], not binary:

```
public void pretrain(double[][][] X, int minibatchSize, int
minibatch_N, int epochs, double learningRate, double
corruptionLevel) {

    for (int layer = 0; layer < nLayers; layer++) {
        for (int epoch = 0; epoch < epochs; epoch++) {
            for (int batch = 0; batch < minibatch_N; batch++) {

                double[][] X_ = new double[minibatchSize][nIn];
                double[][] prevLayerX_;

                // Set input data for current layer
                if (layer == 0) {
                    X_ = X[batch];
                } else {

                    prevLayerX_ = X_;
                    X_ = new
                    double[minibatchSize][hiddenLayerSizes[layer-1]];

                    for (int i = 0; i < minibatchSize; i++) {
                        X_[i] = sigmoidLayers[layer-
                            1].output(prevLayerX_[i]);
                    }
                }

                daLayers[layer].train(X_, minibatchSize,
                learningRate, corruptionLevel);
            }
        }
    }
}
```

The predict method after learning is also exactly the same as in DBN. Considering that both DBN and SDA can be treated as one multi-layer neural network after learning (that is, the pre-training and fine-tuning), it's natural that most of the processes are common.

Overall, SDA can be implemented more easily than DBN, but the precision to be obtained is almost the same. This is the merit of SDA.

Summary

In this chapter, we looked at the problem of the previous neural networks algorithm and what the breakthrough was for deep learning. Also, you learned about the theory and implementation of DBN and SDA, the algorithm that fueled the boom of deep learning, and of RBM and DA used in each respective method.

In the next chapter, we'll look at more deep learning algorithms. They take different approaches to obtain high precision rates and are well developed.

4

Dropout and Convolutional Neural Networks

In this chapter, we continue to look through the algorithms of deep learning. The pre-training that was taken into both DBN and SDA is indeed an innovative method, but deep learning also has other innovative methods. Among these methods, we'll go into the details of the particularly eminent algorithms, which are the following:

- The dropout learning algorithm
- Convolutional neural networks

Both algorithms are necessary to understand and master deep learning, so make sure you keep up.

Deep learning algorithms without pre-training

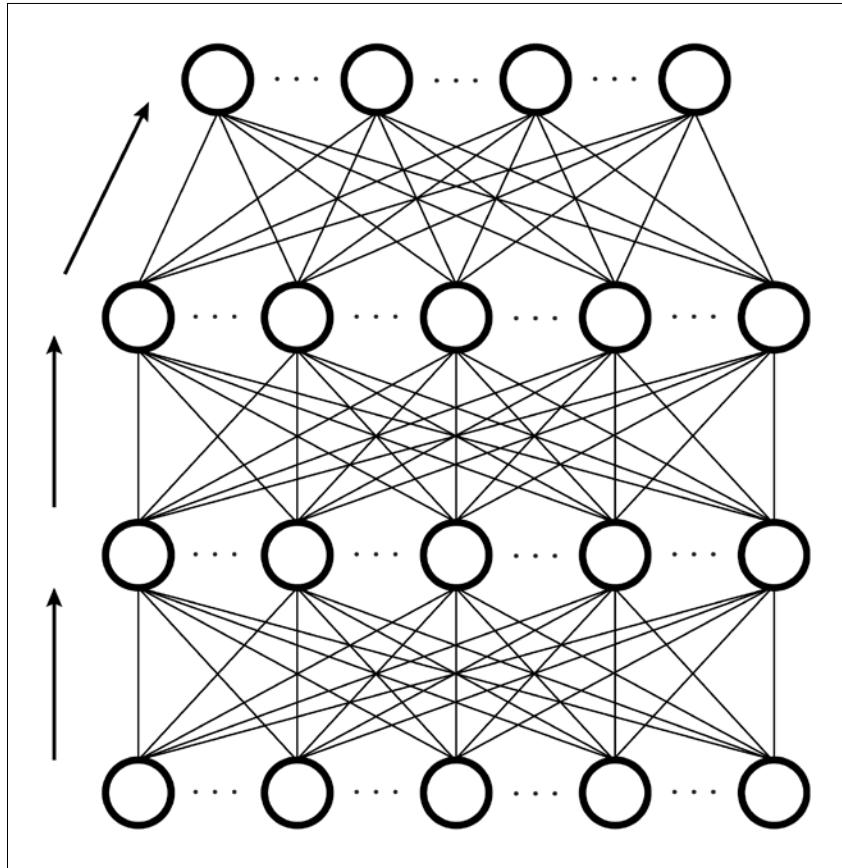
In the previous chapter, you learned that layer-wise training with pre-training was a breakthrough for DBN and SDA. The reason why these algorithms need pre-training is because an issue occurs where an output error gradually vanishes and doesn't work well in neural networks with simple piled-up layers (we call this the vanishing gradient problem). The deep learning algorithm needs pre-training whether you want to improve the existing method or reinvent it—you might think of it like that.

However, actually, the deep learning algorithms in this chapter don't have a phase of pre-training, albeit in the deep learning algorithm without pre-training, we can get a result with higher precision and accuracy. Why is such a thing possible? Here is a brief reason. Let's think about why the vanishing gradient problem occurs – remember the equation of backpropagation? A delta in a layer is distributed to all the units of a previous layer by literally propagating networks backward. This means that in the network where all units are tied densely, the value of an error backpropagated to each unit becomes small. As you can see from the equations of backpropagation, the gradients of the weight are obtained by the multiplication of the weights and deltas among the units. Hence, the more terms we have, the more dense the networks are and the more possibilities we have for underflow. This causes the vanishing gradient problem.

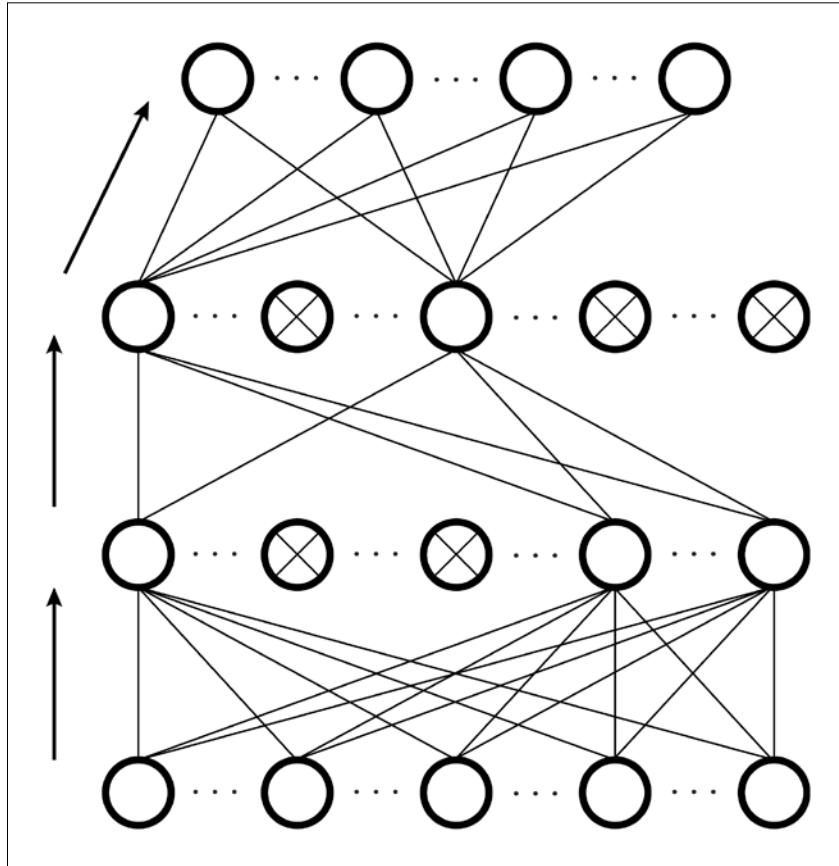
Therefore, we can say that if the preceding problems can be avoided without pre-training, a machine can learn properly with deep neural networks. To achieve this, we need to arrange how to connect the networks. The deep learning algorithm in this chapter is a method that puts this contrivance into practice using various approaches.

Dropout

If there's a problem with the network being tied densely, just force it to be sparse. Then the vanishing gradient problem won't occur and learning can be done properly. The algorithm based on such an idea is the **dropout** algorithm. Dropout for deep neural networks was introduced in *Improving neural networks by preventing co adaptation of feature detectors* (Hinton, et. al. 2012, <http://arxiv.org/pdf/1207.0580.pdf>) and refined in *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* (Srivastava, et. al. 2014, <https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf>). In dropout, some of the units are, literally, forcibly dropped while training. What does this mean? Let's look at the following figures – firstly, neural networks:



There is nothing special about this figure. It is a standard neural network with one input layer, two hidden layers, and one output layer. Secondly, the graphical model can be represented as follows by applying dropout to this network:



Units that are dropped from the network are depicted with cross signs. As you can see in the preceding figure, dropped units are interpreted as non-existent in the network. This means we need to change the structure of the original neural network while the dropout learning algorithm is being applied. Thankfully, applying dropout to the network is not difficult from a computational standpoint. You can simply build a general deep neural network first. Then the dropout learning algorithm can be applied just by adding a dropout mask – a simple binary mask – to all the units in each layer. Units with the value of 0 in the binary mask are the ones that are dropped from the network.

This may remind you of DA (or SDA) discussed in the previous chapter because DA and dropout look similar at first glance. Corrupting input data in DA also adds binary masks to the data when implemented. However, there are two remarkably different points between them. First, while it is true that both methods have the process of adding masks to neurons, DA applies the mask only to units in the input layer, whereas dropout applies it to units in the hidden layer. Some of the dropout algorithms apply masks to both the input layer and the hidden layer, but this is still different from DA. Second, in DA, once the corrupt input data is generated, the data will be used throughout the whole training epochs, but in dropout, the data with different masks will be used in each training epoch. This indicates that a neural network of a different shape is trained in each iteration. Dropout masks will be generated in each layer in each iteration according to the probability of dropout.

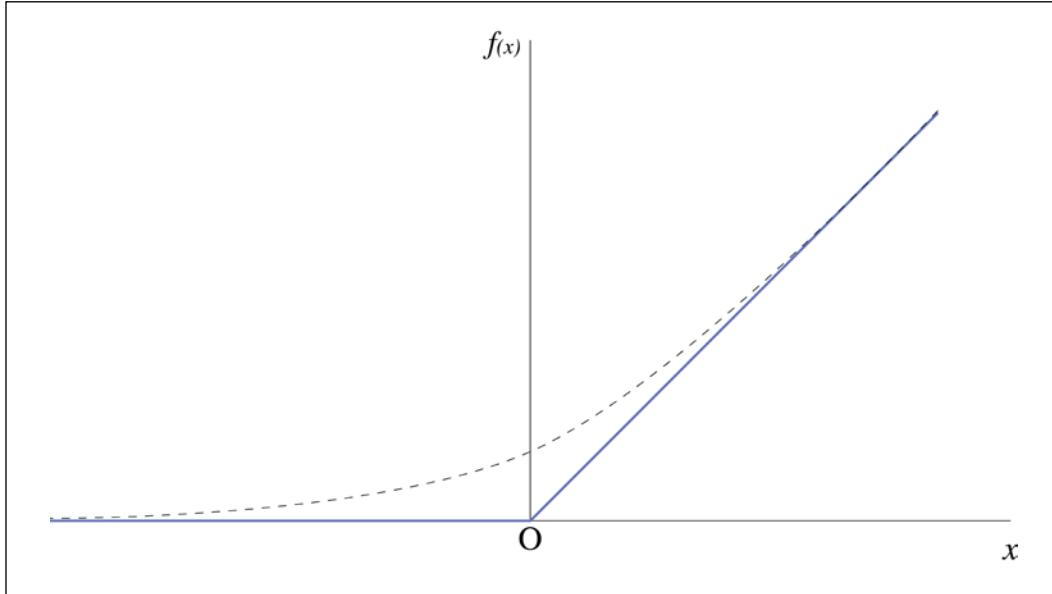
You might have a question – can we train the model even if the shape of the network is different in every step? The answer is yes. You can think of it this way – the network is well trained with dropout because it puts more weights on the existing neurons to reflect the characteristics of the input data. However, dropout has a single demerit, that is, it requires more training epochs than other algorithms to train and optimize the model, which means it takes more time until it is optimized. Another technique is introduced here to reduce this problem. Although the dropout algorithm itself was invented earlier, it was not enough for deep neural networks to gain the ability to generalize and get high precision rates just by using this method. With one more technique that makes the network even more sparse, we achieve deep neural networks to get higher accuracy. This technique is the improvement of the activation function, which we can say is a simple yet elegant solution.

All of the methods of neural networks explained so far utilize the sigmoid function or hyperbolic tangent as an activation function. You might get great results with these functions. However, as you can see from the shape of them, these curves saturate and kill the gradients when the input values or error values at a certain layer are relatively large or small.

One of the activation functions introduced to solve this problem is the **rectifier**. A unit-applied rectifier is called a **Rectified Linear Unit (ReLU)**. We can call the activation function itself ReLU. This function is described in the following equation:

$$f(x) = \max(0, x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

The function can be represented by the following figure:



The broken line in the figure is the function called a **softplus function**, the derivative of it is logistic function, which can be described as follows:

$$\text{softplus}(x) = \ln(1 + \exp(x))$$

This is just for your information: we have the following relations that a smooth approximation to the rectifier. As you can see from the figure above, since the rectifier is far simpler than the sigmoid function and hyperbolic tangent, you can easily guess that the time cost will reduce when it is applied to the deep learning algorithm. In addition, because the derivative of the rectifier—which is necessary when calculating backpropagation errors—is also simple, we can, additionally, shorten the time cost. The equation of the derivative can be represented as follows:

$$f'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Since both the rectifier and the derivative of it are very sparse, we can easily imagine that the neural networks will be also sparse through training. You may have also noticed that we no longer have to worry about gradient saturations because we don't have the causal curves that the sigmoid function and hyperbolic tangent contain anymore.

With the technique of dropout and the rectifier, a simple deep neural network can learn a problem without pre-training. In terms of the equations used to implement the dropout algorithm, they are not difficult because they are just simple methods of adding dropout masks to multi-layer perceptrons. Let's look at them in order:

$$Z_j = h\left(\sum_i w_{ji}x_i + b_j\right)$$

Here, $h(\cdot)$ denotes the activation function, which is, in this case, the rectifier. You see, the previous equation is for units in the hidden layer without dropout. What the dropout does is just apply the mask to them. It can be represented as follows:

$$Z_j = h\left(\sum_i w_{ji}x_i + b_j\right)m_j$$

$$m_j \sim Bernoulli(1-p)$$

Here, p denotes the probability of dropout, which is generally set to 0.5. That's all for forward activation. As you can see from the equations, the term of the binary mask is the only difference from the ones of general neural networks. In addition, during backpropagation, we also have to add masks to the delta. Suppose we have the following equation:

$$a_j = \sum_i w_{ji}x_i + b_j$$

With this, we can define the delta as follows:

$$\delta_j := \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j}$$

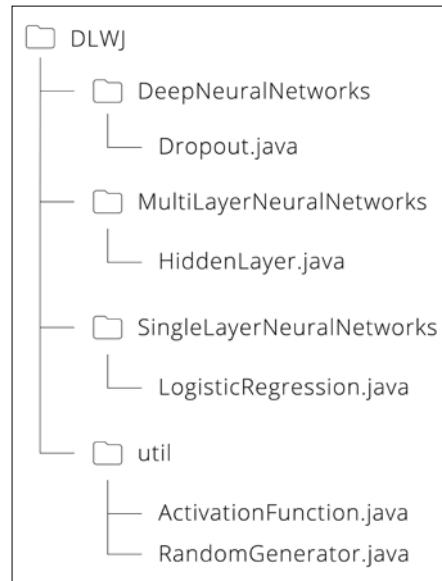
Here, E_n denotes the evaluation function (these equations are the same as we mentioned in *Chapter 2, Algorithms for Machine Learning – Preparing for Deep Learning*). We get the following equation:

$$\begin{aligned} a_k &= \sum_j w_{kj} z_j + c_k \\ &= \sum_j w_{kj} h(a_j) m_j + c_k \end{aligned}$$

Here, the delta can be described as follows:

$$\delta_j = h'(a_j) m_j \sum_k \delta_k w_{kj}$$

Now we have all the equations necessary for implementation, let's dive into the implementation. The package structure is as follows:



First, what we need to have is the rectifier. Like other activation functions, we implement it in `ActivationFunction.java` as ReLU:

```
public static double ReLU(double x) {  
    if(x > 0) {  
        return x;  
    } else {  
        return 0.;  
    }  
}
```

Also, we define `dReLU` as the derivative of the rectifier:

```
public static double dReLU(double y) {  
    if(y > 0) {  
        return 1.;  
    } else {  
        return 0.;  
    }  
}
```

Accordingly, we updated the constructor of `HiddenLayer.java` to support ReLU:

```
if (activation == "sigmoid" || activation == null) {  
  
    this.activation = (double x) -> sigmoid(x);  
    this.dactivation = (double x) -> dsigmoid(x);  
  
} else if (activation == "tanh") {  
  
    this.activation = (double x) -> tanh(x);  
    this.dactivation = (double x) -> dtanh(x);  
  
} else if (activation == "ReLU") {  
  
    this.activation = (double x) -> ReLU(x);  
    this.dactivation = (double x) -> dReLU(x);  
  
} else {  
    throw new IllegalArgumentException("activation function not  
supported");  
}
```

Now let's have a look at `Dropout.java`. In the source code, we'll build the neural networks of two hidden layers, and the probability of dropout is set to 0.5:

```
int[] hiddenLayerSizes = {100, 80};  
double pDropout = 0.5;
```

The constructor of `Dropout.java` can be written as follows (since the network is just a simple deep neural network, the code is also simple):

```
public Dropout(int nIn, int[] hiddenLayerSizes, int nOut, Random rng,  
String activation) {  
  
    if (rng == null) rng = new Random(1234);  
  
    if (activation == null) activation = "ReLU";  
  
    this.nIn = nIn;  
    this.hiddenLayerSizes = hiddenLayerSizes;  
    this.nOut = nOut;  
    this.nLayers = hiddenLayerSizes.length;  
    this.hiddenLayers = new HiddenLayer[nLayers];  
    this.rng = rng;  
  
    // construct multi-layer  
    for (int i = 0; i < nLayers; i++) {  
        int nIn_;  
        if (i == 0) nIn_ = nIn;  
        else nIn_ = hiddenLayerSizes[i - 1];  
  
        // construct hidden layer  
        hiddenLayers[i] = new HiddenLayer(nIn_,  
                                         hiddenLayerSizes[i], null, null, rng, activation);  
    }  
  
    // construct logistic layer  
    logisticLayer = new LogisticRegression(hiddenLayerSizes[nLayers -  
1], nOut);  
}
```

As explained, now we have the `HiddenLayer` class with ReLU support, we can use ReLU as the activation function.

Once a model is built, what we do next is train the model with dropout. The method for training is simply called `train`. Since we need some layer inputs when calculating the backpropagation errors, we define the variable called `layerInputs` first to cache their respective input values:

```
List<double[][]> layerInputs = new ArrayList<>(nLayers+1);
layerInputs.add(X);
```

Here, `X` is the original training data. We also need to cache the dropout masks for each layer for backpropagation, so let's define it as `dropoutMasks`:

```
List<int[][]> dropoutMasks = new ArrayList<>(nLayers);
```

Training begins in a forward activation fashion. Look how we apply the dropout masks to the value; we merely multiply the activated values and binary masks:

```
// forward hidden layers
for (int layer = 0; layer < nLayers; layer++) {

    double[] x_; // layer input
    double[][] Z_ = new double[minibatchSize][hiddenLayerSizes[layer]];
    int[][] mask_ = new int[minibatchSize][hiddenLayerSizes[layer]];

    for (int n = 0; n < minibatchSize; n++) {

        if (layer == 0) {
            x_ = X[n];
        } else {
            x_ = Z_[n];
        }

        Z_[n] = hiddenLayers[layer].forward(x_);
        mask_[n] = dropout(Z_[n], pDropout); // apply dropout mask
        to units
    }

    Z = Z_;
    layerInputs.add(Z.clone());
}

dropoutMasks.add(mask_);
}
```

The dropout method is defined in `Dropout.java` as well. As explained in the equation, this method returns the values following the Bernoulli distribution:

```
public int[] dropout(double[] z, double p) {  
  
    int size = z.length;  
    int[] mask = new int[size];  
  
    for (int i = 0; i < size; i++) {  
        mask[i] = binomial(1, 1 - p, rng);  
        z[i] *= mask[i]; // apply mask  
    }  
  
    return mask;  
}
```

After forward propagation through the hidden layers, training data is forward propagated in the output layer of the logistic regression. Then, in the same way as the other neural networks algorithm, the deltas of each layer are going back through the network. Here, we apply the cached masks to the delta so that its values are backpropagated in the same network:

```
// forward & backward output layer  
D = logisticLayer.train(Z, T, minibatchSize, learningRate);  
  
// backward hidden layers  
for (int layer = nLayers - 1; layer >= 0; layer--) {  
  
    double[][] Wprev_;  
  
    if (layer == nLayers - 1) {  
        Wprev_ = logisticLayer.W;  
    } else {  
        Wprev_ = hiddenLayers[layer+1].W;  
    }  
  
    // apply mask to delta as well  
    for (int n = 0; n < minibatchSize; n++) {  
        int[] mask_ = dropoutMasks.get(layer)[n];  
  
        for (int j = 0; j < D[n].length; j++) {  
            D[n][j] *= mask_[j];  
        }  
    }  
}
```

```

        }
    }

D = hiddenLayers[layer].backward(layerInputs.get(layer),
layerInputs.get(layer+1), D, Wprev_, minibatchSize,
learningRate);
}

```

After the training comes the test phase. But before we apply the test data to the tuned model, we need to configure the weights of the network. Dropout masks can't be simply applied to the test data because when masked, the shape of each network will be differentiated, and this may return different results because a certain unit may have a significant effect on certain features. Instead, what we do is smooth the weights of the network, which means we simulate the network where whole units are equally masked. This can be done using the following equation:

$$W_{test} = (1 - p)W$$

As you can see from the equation, all the weights are multiplied by the probability of non-dropout. We define the method for this as `pretest`:

```

public void pretest(double pDropout) {

    for (int layer = 0; layer < nLayers; layer++) {

        int nIn_, nOut_;

        if (layer == 0) {
            nIn_ = nIn;
        } else {
            nIn_ = hiddenLayerSizes[layer];
        }

        if (layer == nLayers - 1) {
            nOut_ = nOut;
        } else {
            nOut_ = hiddenLayerSizes[layer+1];
        }

        for (int j = 0; j < nOut_; j++) {
            for (int i = 0; i < nIn_; i++) {
                hiddenLayers[layer].W[j][i] *= 1 - pDropout;
            }
        }
    }
}

```

We have to call this method once before the test. Since the network is a general multi-layered neural network, what we need to do for the prediction is just perform forward activation through the network:

```
public Integer[] predict(double[] x) {  
  
    double[] z = new double[0];  
  
    for (int layer = 0; layer < nLayers; layer++) {  
  
        double[] x_;  
  
        if (layer == 0) {  
            x_ = x;  
        } else {  
            x_ = z.clone();  
        }  
  
        z = hiddenLayers[layer].forward(x_);  
    }  
  
    return logisticLayer.predict(z);  
}
```

Compared to DBN and SDA, the dropout MLP is far simpler and easier to implement. It suggests the possibility that with a mixture of two or more techniques, we can get higher precision.

Convolutional neural networks

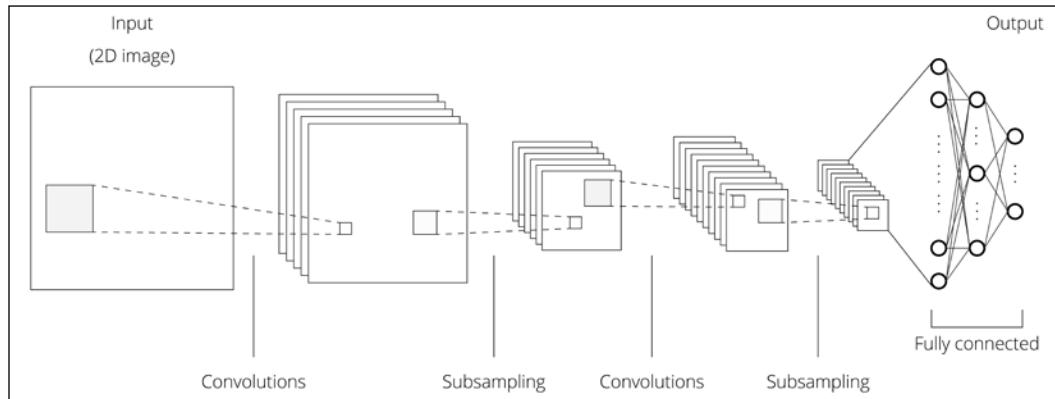
All the machine learning/deep learning algorithms you have learned about imply that the type of input data is one-dimensional. When you look at a real-world application, however, data is not necessarily one-dimensional. A typical case is an image. Though we can still convert two-dimensional (or higher-dimensional) data into a one-dimensional array from the standpoint of implementation, it would be better to build a model that can handle two-dimensional data as it is. Otherwise, some information embedded in the data, such as positional relationships, might be lost when flattened to one dimension.

To solve this problem, an algorithm called **Convolutional Neural Networks (CNN)** was proposed. In CNN, features are extracted from two-dimensional input data through convolutional layers and pooling layers (this will be explained later), and then these features are put into general multi-layer perceptrons. This preprocessing for MLP is inspired by human visual areas and can be described as follows:

- Segment the input data into several domains. This process is equivalent to a human's receptive fields.
- Extract the features from the respective domains, such as edges and position aberrations.

With these features, MLP can classify data accordingly.

The graphical model of CNN is not similar to that of other neural networks. Here is a briefly outlined example of CNN:



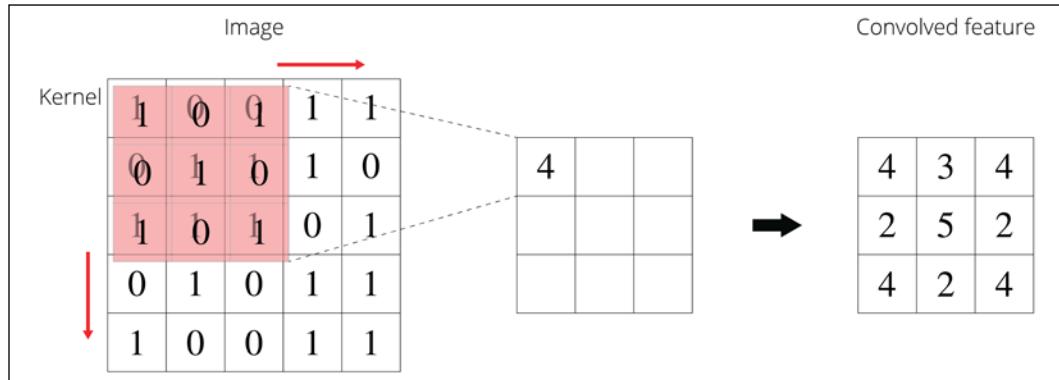
You may not fully understand what CNN is just from the figure. Moreover, you might feel that CNN is relatively complicated and difficult to understand. But you don't have to worry about that. It is a fact that CNN has a complicated graphical model and has unfamiliar terminologies such as convolution and pooling, which you don't hear about in other deep learning algorithms. However, when you look at the model step by step, there's nothing too difficult to understand. CNN consists of several types of layers specifically adjusted for image recognition. Let's look at each layer one by one in the next subsection. In the preceding figure, there are two convolution and pooling (**Subsampling**) layers and fully connected multi-layer perceptrons in the network. We'll see what the convolutional layers do first.

Convolution

Convolutional layers literally perform convolution, which means applying several filters to the image to extract features. These filters are called **kernels**, and convolved images are called **feature maps**. Let's see the following image (decomposed to color values) and kernel:

Image					Kernel		
1	0	0	1	1	1	0	1
0	1	1	1	0	0	1	0
1	1	1	0	1	1	0	1
0	1	0	1	1			
1	0	0	1	1			

With these, what is done with convolution is illustrated as follows:



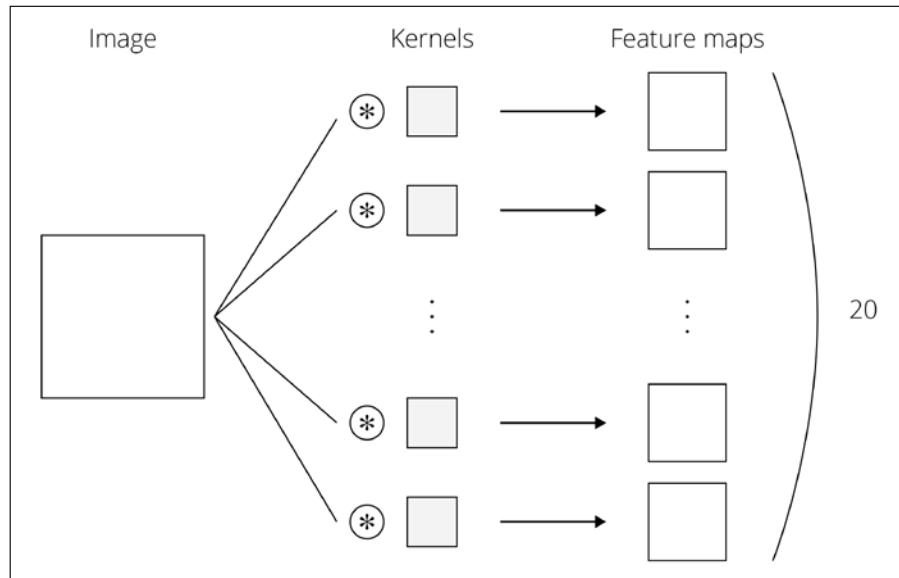
The kernel slides across the image and returns the summation of its values within the kernel as a multiplication filter. You might have noticed that you can extract many kinds of features by changing kernel values. Suppose you have kernels with values as described here:

-1	-1	-1	0.1	0.1	0.1
-1	8	-1	0.1	0.1	0.1
-1	-1	-1	0.1	0.1	0.1

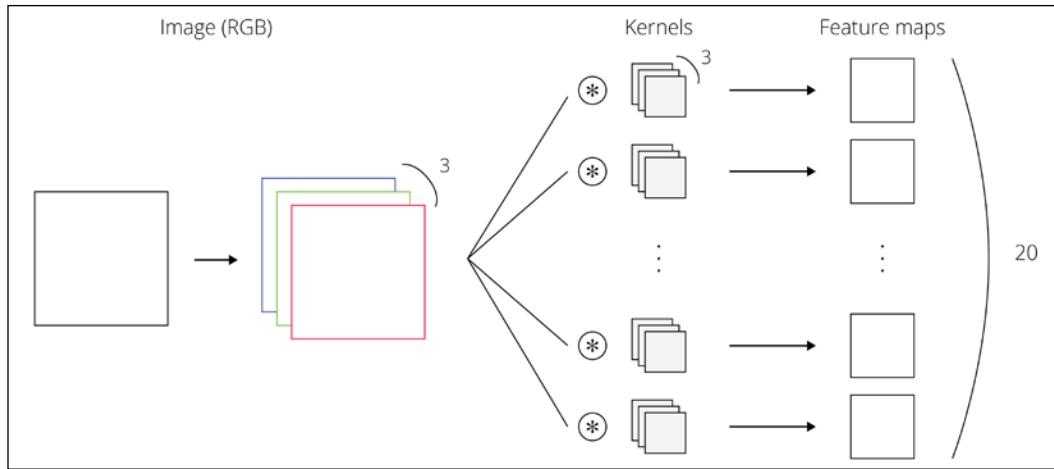
You see that the kernel on the left extracts the edges of the image because it accentuates the color differences, and the one on the right blurs the image because it degrades the original values. The great thing about CNN is that in convolutional layers, you don't have to set these kernel values manually. Once initialized, CNN itself will learn the proper values through the learning algorithm (which means parameters trained in CNN are the weights of kernels) and can classify images very precisely in the end.

Now, let's think about why neural networks with convolutional layers (kernels) can predict with higher precision rates. The key here is the **local receptive field**. In most layers in neural networks except CNN, all neurons are fully connected. This even causes slightly different data, for example, one-pixel parallel data would be regarded as completely different data in the network because this data is propagated to different neurons in hidden layers, whereas humans can easily understand they are the same. With fully connected layers, it is true that neural networks can recognize more complicated patterns, but at the same time they lack the ability to generalize and lack flexibility. In contrast, you can see that connections among neurons in convolutional layers are limited to their kernel size, making the model more robust to translated images. Thus, neural networks with their receptive fields limited locally are able to acquire **translation invariance** when kernels are optimized.

Each kernel has its own values and extracts respective features from the image. Please bear in mind that the number of feature maps and the number of kernels are always the same, which means if we have 20 kernels, we have also twenty feature maps, that is, convolved images. This can be confusing, so let's explore another example. Given a gray-scaled **image** and twenty **kernels**, how many **feature maps** are there? The answer is twenty. These twenty images will be propagated to the next layer. This is illustrated as follows:



So, how about this: suppose we have a 3-channeled image (for example, an RGB image) and the number of kernels is twenty, how many feature maps will there be? The answer is, again, twenty. But this time, the process of convolution is different from the one with gray-scaled, that is 1-channeled, images. When the image has multiple channels, kernels will be adapted separately for each channel. Therefore, in this case, we will have a total of 60 convolved images first, composed of twenty mapped images for each of the 3 channels. Then, all the convolved images originally from the same image will be combined into one feature map. As a result, we will have twenty feature maps. In other words, images are decomposed into different channeled data, applied kernels, and then combined into mixed-channeled images again. You can easily imagine from the flow in the preceding diagram that when we apply a kernel to a multi-channeled image to make decomposed images, the same kernel should be applied. This flow can be seen in the following figure:

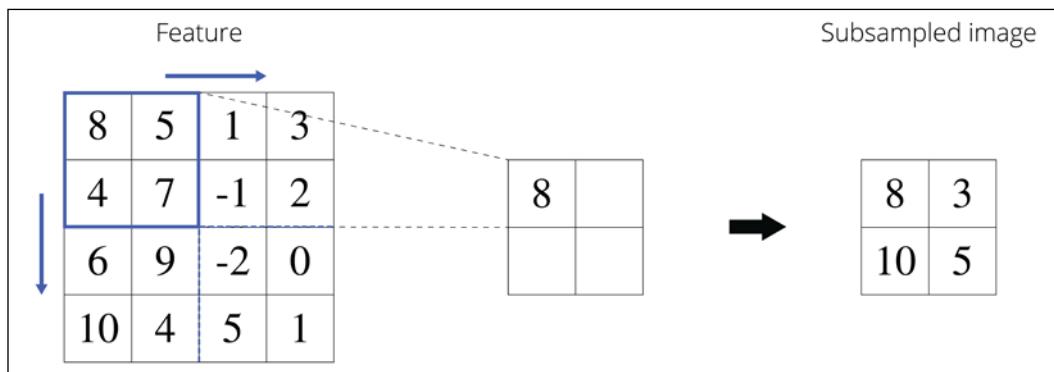


Computationally, the number of kernels is represented with the dimension of the weights' tensor. You'll see how to implement this later.

Pooling

What pooling layers do is rather simple compared to convolutional layers. They actually do not train or learn by themselves but just downsample images propagated from convolutional layers. Why should we bother to do downsampling? You might think it may lose some significant information from the data. But here, again, as with convolutional layers, this process is necessary to make the network keep its translation invariance.

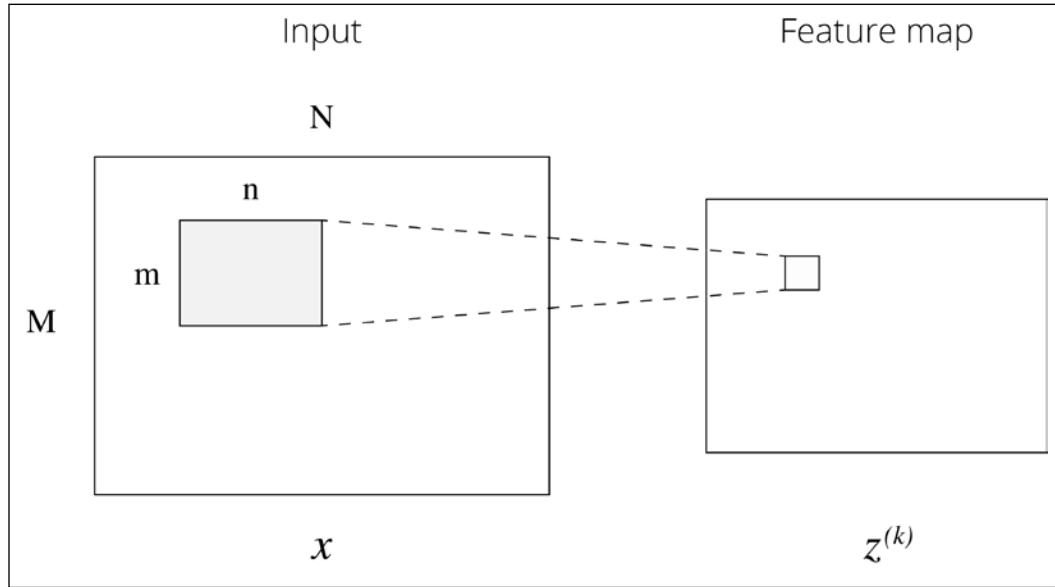
There are several ways of downsampling, but among them, max-pooling is the most famous. It can be represented as follows:



In a max-pooling layer, the input image is segmented into a set of non-overlapping sub-data and the maximum value is output from each data. This process not only keeps its translation invariance but also reduces the computation for the upper layers. With convolution and pooling, CNN can acquire robust features from the input.

Equations and implementations

Now we know what convolution and max-pooling are, let's describe the whole model with equations. We'll use the figure of convolution below in equations:



As shown in the figure, if we have an image with a size of $M \times N$ and kernels with a size of $m \times n$, the convolution can be represented as:

$$z_{ij}^{(k)} = \sum_{s=0}^{m-1} \sum_{t=0}^{n-1} w_{st}^{(k)} x(i+s)(j+t)$$

Here, w is the weight of the kernel, that is, the model parameter. Just bear in mind we've described each summation from 0, not from 1, so you get a better understanding. The equation, however, is not enough when we think about multi-convolutional layers because it does not have the information from the channel. Fortunately, it's not difficult because we can implement it just by adding one parameter to the kernel. The extended equation can be shown as:

$$z_{ij}^{(k)} = \sum_c \sum_{s=0}^{m-1} \sum_{t=0}^{n-1} w_{st}^{(k,c)} x_{(i+s)(j+t)}^{(c)}$$

Here, c denotes the channel of the image. If the number of kernels is K and the number of channels is C , we have $W \in \mathbb{R}^{K \times C \times m \times n}$. Then, you can see from the equation that the size of the convolved image is $(M - m + 1) \times (N - n + 1)$.

After the convolution, all the convolved values will be activated by the activation function. We'll implement CNN with the rectifier—the most popular function these days—but you may use the sigmoid function, the hyperbolic tangent, or any other activation functions available instead. With the activation, we have:

$$a_{ij}^{(k)} = h(z_{ij}^{(k)} + b^{(k)}) = \max(0, z_{ij}^{(k)} + b^{(k)})$$

Here, b denotes the bias, the other model parameter. You can see that b doesn't have subscripts of i and j , that is, we have $b \in \mathbb{R}^K$, a one-dimensional array. Thus, we have forward-propagated the values of the convolutional layer.

Next comes the max-pooling layer. The propagation can simply be written as follows:

$$y_{ij}^{(k)} = \max(a_{(l_1 i + s)(l_2 j + t)}^{(k)})$$

Here, l_1 and l_2 are the size of pooling filter and $s \in [0, l_1], t \in [0, l_2]$. Usually, l_1 and l_2 are set to the same value of $2 \sim 4$.

These two layers, the convolutional layer and the max-pooling layer, tend to be arrayed in this order, but you don't necessarily have to follow it. You can put two convolutional layers before max-pooling, for example. Also, while we put the activation right after the convolution, sometimes it is set after the max-pooling instead of the convolution. For simplicity, however, we'll implement CNN with the order and sequence of convolution-activation-max-pooling.



One important note here is that although the kernel weights will be learned from the data, the architecture, the size of kernel, and the size of pooling are all parameters.



The simple MLP follows after convolutional layers and max-pooling layers to classify the data. Here, since MLP can only accept one-dimensional data, we need to flatten the downsampled data as preprocessing to adapt it to the input layer of MLP. The extraction of features was completed before MLP, so formatting the data into one dimension won't be a problem. Thus, CNN can classify the image data once the model is optimized. To do this, as with other neural networks, the backpropagation algorithm is applied to CNN to train the model. We won't mention the equation related to MLP here.

The error from the input layer of MLP is backpropagated to the max-pooling layer, and this time it is unflattened to two dimensions to be adapted properly to the model. Since the max-pooling layer doesn't have model parameters, it simply backpropagates the error to the previous layer. The equation can be described as follows:

$$\frac{\partial E}{\partial a_{(l_1 i + s)(l_2 j + t)}^{(k)}} = \begin{cases} \frac{\partial E}{\partial y_{ij}^{(k)}} & \text{if } y_{ij}^{(k)} = a_{(l_1 i + s)(l_2 j + t)}^{(k)} \\ 0 & \text{otherwise} \end{cases}$$

Here, E denotes the evaluation function. This error is then backpropagated to the convolutional layer, and with it we can calculate the gradients of the weight and the bias. Since the activation with the bias comes before the convolution when backpropagating, let's see the gradient of the bias first, as follows:

$$\frac{\partial E}{\partial b^{(k)}} = \sum_{i=0}^{M-m} \sum_{j=0}^{N-m} \frac{\partial E}{\partial a_{ij}^{(k)}} \frac{\partial a_{ij}^{(k)}}{\partial b^{(k)}}$$

To proceed with this equation, we define the following:

$$\delta_{ij}^{(k)} := \frac{\partial E}{\partial a_{ij}^{(k)}}$$

We also define:

$$c_{ij}^{(k)} := z_{ij}^{(k)} + b^{(k)}$$

With these, we get:

$$\begin{aligned} \frac{\partial E}{\partial b^{(k)}} &= \sum_{i=0}^{M-m} \sum_{j=0}^{N-n} \delta_{ij}^{(k)} \frac{\partial a_{ij}^{(k)}}{\partial c_{ij}^{(k)}} \frac{\partial c_{ij}^{(k)}}{\partial b^{(k)}} \\ &= \sum_{i=0}^{M-m} \sum_{j=0}^{N-n} \delta_{ij}^{(k)} h'(c_{ij}^{(k)}) \end{aligned}$$

We can calculate the gradient of the weight (kernel) in the same way:

$$\begin{aligned} \frac{\partial E}{\partial w_{st}^{(k,c)}} &= \sum_{i=0}^{M-m} \sum_{j=0}^{N-n} \frac{\partial E}{\partial z_{ij}^{(k)}} \frac{\partial z_{ij}^{(k)}}{\partial w_{st}^{(k,c)}} \\ &= \sum_{i=0}^{M-m} \sum_{j=0}^{N-n} \frac{\partial E}{\partial z_{ij}^{(k)}} \frac{\partial a_{ij}^{(k)}}{\partial z_{ij}^{(k)}} x_{(i+s)(j+t)}^{(c)} \\ &= \sum_{i=0}^{M-m} \sum_{j=0}^{N-n} \delta_{ij}^{(k)} h'(c_{ij}^{(k)}) x_{(i+s)(j+t)}^{(c)} \end{aligned}$$

Thus, we can update the model parameters. If we have just one convolutional and max-pooling layer, the equations just given are all that we need. When we think of multi-convolutional layers, however, we also need to calculate the error of the convolutional layers. This can be represented as follows:

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}^{(c)}} &= \sum_k \sum_{s=0}^{m-1} \sum_{t=0}^{n-1} \frac{\partial E}{\partial z_{(i-s)(j-t)}^{(k)}} \frac{\partial z_{(i-s)(j-t)}^{(k)}}{\partial x_{ij}^{(c)}} \\ &= \sum_k \sum_{s=0}^{m-1} \sum_{t=0}^{n-1} \frac{\partial E}{\partial z_{(i-s)(j-t)}^{(k)}} w_{st}^{(k,c)}\end{aligned}$$

Here, we get:

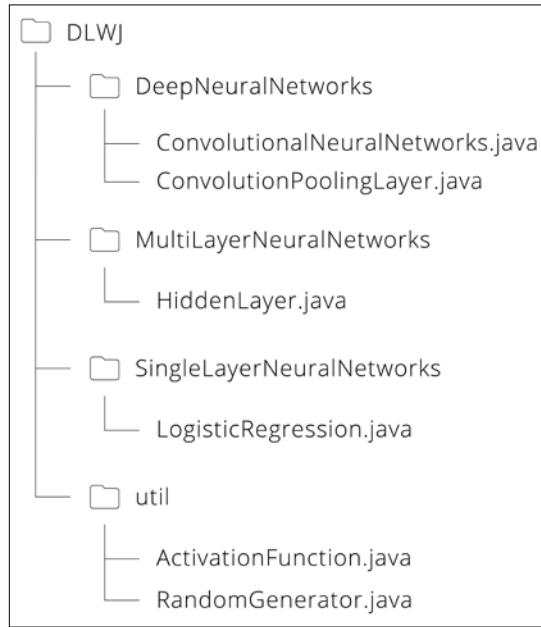
$$\begin{aligned}\frac{\partial E}{\partial z_{ij}^{(c)}} &= \frac{\partial E}{\partial a_{ij}^{(k)}} \frac{\partial a_{ij}^{(k)}}{\partial z_{ij}^{(k)}} \\ &= \delta_{ij}^{(k)} \frac{\partial a_{ij}^{(k)}}{\partial c_{ij}^{(k)}} \frac{\partial c_{ij}^{(k)}}{\partial z_{ij}^{(k)}} \\ &= \delta_{ij}^{(k)} h'(c_{ij}^{(k)})\end{aligned}$$

So, the error can be written as follows:

$$\frac{\partial E}{\partial x_{ij}^{(c)}} = \sum_k \sum_{s=0}^{m-1} \sum_{t=0}^{n-1} \delta_{(i-s)(j-t)}^{(k)} h'(c_{(i-s)(j-t)}^{(k)}) w_{st}^{(k,c)}$$

We have to be careful when calculating this because there's a possibility of $i - s < 0$ or $j - t < 0$, where there's no element in between the feature maps. To solve this, we need to add zero paddings to the top-left edges of them. Then, the equation is simply a convolution with the kernel flipped along both axes. Though the equations in CNN might look complicated, they are just a pile of summations of each parameter.

With all the previous equations, we can now implement CNN, so let's see how we do it. The package structure is as follows:



`ConvolutionNeuralNetworks.java` is used to build the model outline of CNN, and the exact algorithms for training in the convolutional layers and max-pooling layers, forward propagations, and backpropagations are written in `ConvolutionPoolingLayer.java`. In the demo, we have the original image size of 12×12 with one channel:

```

final int[] imageSize = {12, 12};
final int channel = 1;
    
```

The image will be propagated through two `ConvPoolingLayer` (convolutional layers and max-pooling layers). The number of kernels in the first layer is set to 10 with the size of 3×3 and 20 with the size of 2×2 in the second layer. The size of the pooling filters are both set to 2×2 :

```

int[] nKernels = {10, 20};
int[][] kernelsSizes = { {3, 3}, {2, 2} };
int[][] poolSizes = { {2, 2}, {2, 2} };
    
```

After the second max-pooling layer, there are 20 feature maps with the size of 2×2 . These maps are then flattened to 80 units and will be forwarded to the hidden layer with 20 neurons:

```
int nHidden = 20;
```

We then create simple demo data of three patterns with a little noise. We'll leave out the code to create demo data here. If we illustrate the data, here is an example of it:



Now let's build the model. The constructor is similar to other deep learning models and rather simple. We construct multi ConvolutionPoolingLayers first. The size for each layer is calculated in the method:

```
// construct convolution + pooling layers
for (int i = 0; i < nKernels.length; i++) {
    int[] size_;
    int channel_;

    if (i == 0) {
        size_ = new int[]{imageSize[0], imageSize[1]};
        channel_ = channel;
    } else {
        size_ = new int[]{pooledSizes[i-1][0], pooledSizes[i-1][1]};
        channel_ = nKernels[i-1];
    }

    convolvedSizes[i] = new int[]{size_[0] - kernelSizes[i][0] + 1,
        size_[1] - kernelSizes[i][1] + 1};
}
```

```

pooledSizes[i] = new int[] {convolvedSizes[i][0] /
poolSizes[i][0], convolvedSizes[i][1] / poolSizes[i][0]};

convpoolLayers[i] = new ConvolutionPoolingLayer(size_,
channel_, nKernels[i], kernelSizes[i], poolSizes[i],
convolvedSizes[i], pooledSizes[i], rng, activation);
}

```

When you look at the constructor of the `ConvolutionPoolingLayer` class, you can see how the kernel and the bias are defined:

```

if (W == null) {

    W = new double[nKernel] [channel] [kernelSize[0]] [kernelSize[1]];

    double in_ = channel * kernelSize[0] * kernelSize[1];
    double out_ = nKernel * kernelSize[0] * kernelSize[1] /
(poolSize[0] * poolSize[1]);
    double w_ = Math.sqrt(6. / (in_ + out_));

    for (int k = 0; k < nKernel; k++) {
        for (int c = 0; c < channel; c++) {
            for (int s = 0; s < kernelSize[0]; s++) {
                for (int t = 0; t < kernelSize[1]; t++) {
                    W[k] [c] [s] [t] = uniform(-w_, w_, rng);
                }
            }
        }
    }
}

if (b == null) b = new double[nKernel];

```

Next comes the construction of MLP. Don't forget to flatten the downsampled data when passing through them:

```

// build MLP
flattenedSize = nKernels[nKernels.length-1] * pooledSizes[pooledSizes.
length-1][0] * pooledSizes[pooledSizes.length-1][1];

// construct hidden layer
hiddenLayer = new HiddenLayer(flattenedSize, nHidden, null, null, rng,
activation);

// construct output layer
logisticLayer = new LogisticRegression(nHidden, nOut);

```

Once the model is built, we need to train it. In the `train` method, we cache all the forward-propagated data so that we can utilize it when backpropagating:

```
// cache pre-activated, activated, and downsampled inputs of each
// convolution + pooling layer for backpropagation
List<double[][][]> preActivated_X = new ArrayList<>(nKernels.
length);
List<double[][][]> activated_X = new ArrayList<>(nKernels.length);
List<double[][][]> downsampled_X = new ArrayList<>(nKernels.
length+1); // +1 for input X
downsampled_X.add(X);

for (int i = 0; i < nKernels.length; i++) {
    preActivated_X.add(new
        double[minibatchSize][nKernels[i]][convolvedSizes[i][0]]
        [convolvedSizes[i][1]]);
    activated_X.add(new
        double[minibatchSize][nKernels[i]][convolvedSizes[i][0]]
        [convolvedSizes[i][1]]);
    downsampled_X.add(new
        double[minibatchSize][nKernels[i]][convolvedSizes[i][0]]
        [convolvedSizes[i][1]]);
}
```

`preActivated_X` is defined for convolved feature maps, `activated_X` for activated features, and `downsampled_X` for downsampled features. We put and cache the original data into `downsampled_X`. The actual training begins with forward propagation through convolution and max-pooling:

```
// forward convolution + pooling layers
double[][][] z_ = X[n].clone();
for (int i = 0; i < nKernels.length; i++) {
    z_ = convpoolLayers[i].forward(z_, preActivated_X.get(i)[n],
        activated_X.get(i)[n]);
    downsampled_X.get(i+1)[n] = z_.clone();
}
```

The `forward` method of `ConvolutionPoolingLayer` is simple and consists of `convolve` and `downsample`. The `convolve` function does the convolution, and `downsample` does the max-pooling:

```
public double[][][] forward(double[][][] x, double[][][] preActivated_X,
    double[][][] activated_X) {

    double[][][] z = this.convolve(x, preActivated_X, activated_X);
    return this.downsample(z);
```

The values of `preActivated_X` and `activated_X` are set inside the `convolve` method. You can see that the method simply follows the equations explained previously:

```
public double[][][] convolve(double[][][] x, double[][][] preActivated_X, double[][][] activated_X) {

    double[][][] y = new double[nKernel][convolvedSize[0]][convolvedSize[1]];

    for (int k = 0; k < nKernel; k++) {
        for (int i = 0; i < convolvedSize[0]; i++) {
            for (int j = 0; j < convolvedSize[1]; j++) {

                double convolved_ = 0.;

                for (int c = 0; c < channel; c++) {
                    for (int s = 0; s < kernelSize[0]; s++) {
                        for (int t = 0; t < kernelSize[1]; t++) {
                            convolved_ += W[k][c][s][t] *
                                x[c][i+s][j+t];
                        }
                    }
                }

                // cache pre-activated inputs
                preActivated_X[k][i][j] = convolved_ + b[k];
                activated_X[k][i][j] =
                    this.activation.apply(preActivated_X[k][i][j]);
                y[k][i][j] = activated_X[k][i][j];
            }
        }
    }

    return y;
}
```

The `downsample` method follows the equations as well:

```
public double[][][] downsample(double[][][] x) {

    double[][][] y = new double[nKernel][pooledSize[0]][pooledSize[1]];

    for (int k = 0; k < nKernel; k++) {
        for (int i = 0; i < pooledSize[0]; i++) {
```

```

        for (int j = 0; j < pooledSize[1]; j++) {

            double max_ = 0.;

            for (int s = 0; s < poolSize[0]; s++) {
                for (int t = 0; t < poolSize[1]; t++) {

                    if (s == 0 && t == 0) {
                        max_ =
                            x[k] [poolSize[0]*i] [poolSize[1]*j];
                        continue;
                    }
                    if (max_ <
                        x[k] [poolSize[0]*i+s] [poolSize[1]*j+t]) {
                        max_ =
                            x[k] [poolSize[0]*i+s] [poolSize[1]*j+t];
                    }
                }
            }

            y[k] [i] [j] = max_;
        }
    }
}

return y;
}

```

You might think we've made some mistake here because there are so many `for` loops in these methods, but actually there's nothing wrong. As you can see from the equations of CNN, the algorithm requires many loops because it has many parameters. The code here works well, but practically, you could define and move the part of the innermost loops to other methods. Here, to get a better understanding, we've implemented CNN with many nested loops so that we can compare the code with equations. You can see now that CNN requires a lot of time to get results.

After we downsample the data, we need to flatten it:

```

// flatten output to make it input for fully connected MLP
double[] x_ = this.flatten(z_);
flattened_X[n] = x_.clone();

```

The data is then forwarded to the hidden layer:

```

// forward hidden layer
Z[n] = hiddenLayer.forward(x_);

```

Multi-class logistic regression is used in the output layer and the delta is then backpropagated to the hidden layer:

```

// forward & backward output layer
dY = logisticLayer.train(Z, T, minibatchSize, learningRate);

// backward hidden layer
dZ = hiddenLayer.backward(flattened_X, Z, dY, logisticLayer.W,
minibatchSize, learningRate);

// backpropagate delta to input layer
for (int n = 0; n < minibatchSize; n++) {
    for (int i = 0; i < flattenedSize; i++) {
        for (int j = 0; j < nHidden; j++) {
            dx_flatten[n][i] += hiddenLayer.W[j][i] * dZ[n][j];
        }
    }
}

dx[n] = unflatten(dx_flatten[n]); // unflatten delta
}

// backward convolution + pooling layers
dC = dX.clone();
for (int i = nKernels.length-1; i >= 0; i--) {
    dC = convpoolLayers[i].backward(downscaled_X.get(i),
    preActivated_X.get(i), activated_X.get(i),
    downsampled_X.get(i+1), dC, minibatchSize, learningRate);
}

```

The backward method of ConvolutionPoolingLayer is the same as forward, also simple. Backpropagation of max-pooling is written in upsample and that of convolution is in deconvolve:

```

public double[][][] backward(double[][][] X, double[][][]
[] preActivated_X, double[][][] activated_X, double[][][]
[] downsampled_X, double[][][] dY, int minibatchSize, double
learningRate) {

    double[][][] dZ = this.upsample(activated_X, downsampled_X,
    dY, minibatchSize);
    return this.deconvolve(X, preActivated_X, dZ, minibatchSize,
    learningRate);
}

```

What `upsample` does is just transfer the delta to the convolutional layer:

```
public double[][][] upsample(double[][][] X, double[][][] Y,
double[][][] dY, int minibatchSize) {

    double[][][] dX = new double[minibatchSize][nKernel]
[convolvedSize[0]][convolvedSize[1]];

    for (int n = 0; n < minibatchSize; n++) {

        for (int k = 0; k < nKernel; k++) {
            for (int i = 0; i < pooledSize[0]; i++) {
                for (int j = 0; j < pooledSize[1]; j++) {

                    for (int s = 0; s < poolSize[0]; s++) {
                        for (int t = 0; t < poolSize[1]; t++) {

                            double d_ = 0.;

                            if (Y[n][k][i][j] == X[n][k]
[poolSize[0]*i+s][poolSize[1]*j+t]) {
                                d_ = dY[n][k][i][j];
                            }

                            dX[n][k][poolSize[0]*i+s][poolSize[1]*j+t]
= d_;
                        }
                    }
                }
            }
        }
    }

    return dX;
}
```

In `deconvolve`, we need to update the model parameter. Since we train the model with mini-batches, we calculate the summation of the gradients first:

```
// calc gradients of W, b
for (int n = 0; n < minibatchSize; n++) {
    for (int k = 0; k < nKernel; k++) {

        for (int i = 0; i < convolvedSize[0]; i++) {
```

```

        for (int j = 0; j < convolvedSize[1]; j++) {

            double d_ = dY[n][k][i][j] *
            this.dactivation.apply(Y[n][k][i][j]);

            grad_b[k] += d_;

            for (int c = 0; c < channel; c++) {
                for (int s = 0; s < kernelSize[0]; s++) {
                    for (int t = 0; t < kernelSize[1]; t++) {
                        grad_W[k][c][s][t] += d_ *
                        X[n][c][i+s][j+t];
                    }
                }
            }
        }
    }
}

```

Then, update the weight and the bias using these gradients:

```

// update gradients
for (int k = 0; k < nKernel; k++) {
    b[k] -= learningRate * grad_b[k] / minibatchsize;

    for (int c = 0; c < channel; c++) {
        for (int s = 0; s < kernelSize[0]; s++) {
            for (int t = 0; t < kernelSize[1]; t++) {
                W[k][c][s][t] -= learningRate * grad_W[k][c][s][t] /
minibatchSize;
            }
        }
    }
}

```

Unlike other algorithms, we have to calculate the parameters and delta discretely in CNN:

```

// calc delta
for (int n = 0; n < minibatchSize; n++) {
    for (int c = 0; c < channel; c++) {
        for (int i = 0; i < imageSize[0]; i++) {
            for (int j = 0; j < imageSize[1]; j++) {

                for (int k = 0; k < nKernel; k++) {

```

```
        for (int s = 0; s < kernelSize[0]; s++) {
            for (int t = 0; t < kernelSize[1]; t++) {

                double d_ = 0.;

                if (i - (kernelSize[0] - 1) - s >= 0 &&
                    j - (kernelSize[1] - 1) - t >= 0) {
                    d_ = dY[n][k][i-(kernelSize[0]-1)-s]
                        [j-(kernelSize[1]-1)-t] *
                    this.dactivation.apply(Y[n][k]
                        [i-(kernelSize[0]-1)-s]
                        [j-(kernelSize[1]-1)-t]) *
                    W[k][c][s][t];
                }

                dX[n][c][i][j] += d_;
            }
        }
    }
}
```

Now we train the model, so let's go on to the test part. The method for testing or prediction simply does the forward propagation, just like the other algorithms:

```
public Integer[] predict(double[][][] x) {

    List<double[][][]> preActivated = new ArrayList<>(nKernels.length);
    List<double[][][]> activated = new ArrayList<>(nKernels.length);

    for (int i = 0; i < nKernels.length; i++) {
        preActivated.add(new
            double[nKernels[i]][convolvedSizes[i][0]]
            [convolvedSizes[i][1]]);
        activated.add(new double[nKernels[i]][convolvedSizes[i][0]]
            [convolvedSizes[i][1]]);
    }

    // forward convolution + pooling layers
    double[][][] z = x.clone();
    for (int i = 0; i < nKernels.length; i++) {
```

```
    z = convpoolLayers[i].forward(z, preActivated.get(i),  
        activated.get(i));  
}  
  
// forward MLP  
return logisticLayer.predict(hiddenLayer.forward(this.flatten(z)));  
}
```

Congratulations! That's all for CNN. Now you can run the code and see how it works. Here, we have CNN with two-dimensional data as input, but CNN can also have three-dimensional data if we expand the model. We can expect its application in medical fields, for example, finding malignant tumors from 3D-scanned data of human brains.

The process of convolution and pooling was originally invented by LeCun et al. in 1998 (<http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf>), yet as you can see from the codes, it requires much calculation. We can assume that this method might not have been suitable for practical applications with computers at the time, not to mention making it deep. The reason CNN has gained more attention recently is probably because the power and capacity of computers has greatly developed. But still, we can't deny the problem. Therefore, it seems practical to use GPU, not CPU, when we have CNN with certain amounts of data. Since the implementation to optimize the algorithm to GPU is complicated, we won't write the codes here. Instead, in *Chapter 5, Exploring Java Deep Learning Libraries – DL4J, ND4J, and More* and *Chapter 7, Other Important Deep Learning Libraries*, you'll see the library of deep learning that is capable of utilizing GPU.

Summary

In this chapter, you learned about two deep learning algorithms that don't require pre-training: deep neural networks with dropout and CNN. The key to high precision rates is how we make the network sparse, and dropout is one technique to achieve this. Another technique is the rectifier, the activation function that can solve the problem of saturation that occurred in the sigmoid function and the hyperbolic tangent. CNN is the most popular algorithm for image recognition and has two features: convolution and max-pooling. Both of these attribute the model to acquire translation invariance. If you are interested in how dropout, rectifier, and other activation functions contribute to the performance of neural networks, the following could be good references: *Deep Sparse Rectifier Neural Networks* (Glorot, et. al. 2011, <http://www.jmlr.org/proceedings/papers/v15/glorot11a/glorot11a.pdf>), *ImageNet Classification with Deep Convolutional Neural Networks* (Krizhevsky et. al. 2012, <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>), and *Maxout Networks* (Goodfellow et al. 2013, <http://arxiv.org/pdf/1302.4389.pdf>).

While you now know the popular and useful deep learning algorithms, there are still many of them that have not been mentioned in this book. This field of study is getting more and more active, and more and more new algorithms are appearing. But don't worry, as all the algorithms are based on the same root: neural networks. Now you know the way of thinking required to grasp or implement the model, you can fully understand whatever models you encounter.

We've implemented deep learning algorithms from scratch so you fully understand them. In the next chapter, you'll see how we can implement them with deep learning libraries to facilitate our research or applications.

5

Exploring Java Deep Learning Libraries – DL4J, ND4J, and More

In the previous chapters, you learned the core theories of deep learning algorithms and implemented them from scratch. While we can now say that implementations of deep learning are not so difficult, we can't deny the fact that it still takes some time to implement models. To mitigate this situation, you'll learn how to write code with the Java library of deep learning in this chapter so that we can focus more on the critical part of data analysis rather than the trivial part.

The topics you'll learn about in this chapter are:

- An introduction to the deep learning library of Java
- Example code and how to write your own code with the library
- Some additional ways to optimize the model to get a higher precision rate

Implementing from scratch versus a library/framework

We implemented the machine learning algorithms of neural networks in *Chapter 2, Algorithms for Machine Learning – Preparing for Deep Learning*, and many deep learning algorithms from scratch in *Chapter 3, Deep Belief Nets and Stacked Denoising Autoencoders* and *Chapter 4, Dropout and Convolutional Neural Networks*. Of course, we can apply our own code to practical applications with some customizations, but we have to be careful when we want to utilize them because we can't deny the possibility that they might cause several problems in the future. What could they be? Here are the possible situations:

- The code we wrote has some missing parameters for better optimization because we implemented just the essence of the algorithms for simplicity and so you better understand the concepts. While you can still train and optimize the model with them, you could get higher precision rates by adding another parameter of your own implementation.
- As mentioned in the previous chapter, there are still many useful deep learning algorithms not explained in this book. While you now have the core components of the deep learning algorithms, you might need to implement additional classes or methods to get the desired results in your fields and applications.
- The assumed time consumption will be very critical to the application, especially when you think of analyzing huge amounts of data. It is true that Java has a better performance in terms of speed compared to other popular languages such as Python and R, but you may still need to consider the time cost. One plausible approach to solve the problem is using GPU instead of CPU, but this requires complex implementations to adjust the code for GPU computing.

These are the main causal issues, and you might also need to take into consideration that we don't handle exceptions in the code.

This does not mean that implementing from scratch would have fatal errors. The code we wrote can be used substantially as an application for certain scaled data; however, you need to take into consideration that you require further coding for the fundamental parts you have implemented if you use large-scale data mining, where, generally, deep learning is required. This means you need to bear in mind that implementation from scratch has more flexibility as you can change the code if required, but at the same time it has a negative side in that the algorithm's tuning and maintenance also has to be done independently.

So, how can we solve the problems just mentioned? This is where a library (or framework) comes in. Thanks to active research into deep learning globally, there are many libraries developed and published using various programming languages all over the world. Of course, each library has its respective features but the features that every library commonly has can be summarized as follows:

- A model's training can be done just by defining a layer structure of deep learning. You can focus on parameter setting and tuning, and you don't need to think about the algorithms.
- Most of the libraries are open to the public as open source projects and are actively updated daily. Therefore, if there are bugs, there's a high possibility that these bugs will be fixed quickly (and, of course, committing to a project by fixing it yourself should be welcomed).
- It's easy to switch between running the program on CPU or on GPU. As a library supplements the cumbersome coding element of GPU computing, you can just focus on the implementation without considering CPU or GPU, if a machine supports GPU.

Long story short, you can leave out all the parts that could be brutal when you implement to a library from scratch. Thanks to this, you can take more time on the essential data mining section, hence if you want to utilize practical applications, there's a high possibility that you can perform data analysis more efficiently using a library.

However, depending too much on a library isn't good. Using a library is convenient, but on the flip side, it has some demerits, as listed here:

- Since you can build various deep learning models easily, you can implement without having a concrete understanding of what theory the model is supported by. This might not be a problem if we only consider implementations related to a specific model, but there will be a risk you can't deal with when you want to combine other methods or consider other methods when applying the model.
- You can't use algorithms not supported by a library, hence there might be a case where you can't choose a model you would like to use. This can be solved by a version upgrade, but on the other hand, there's a possibility that some part of a past implementation might be deprecated due to a change of specification by the upgrade. Moreover, we can't deny the possibility that the development of a library is suddenly terminated or utilization turns out to be chargeable due to a sudden change in its license. In these cases, there's a risk that the code you have developed up to this point cannot be used.

- The precision rate you can get from experimentation depends on how a library is implemented. For example, if we conduct an experiment with the same neural network model in two different libraries, the results we obtain can be hugely changed. This is because neural network algorithms include a stochastic operation, and the calculation accuracy of a machine is limited, that is, calculated values during the process could have fluctuations based on the method of implementation.

Because you well understand the fundamental concepts and theories of deep learning algorithms thanks to the previous chapters, we don't need to worry about the first point. However, we need to be careful about the remaining two points. From the next section on, implementation using a library is introduced and we'll be more conscious of the merits and demerits we just discussed.

Introducing DL4J and ND4J

A lot of the libraries of deep learning have been developed all over the world. In November 2015, **TensorFlow** (<http://www.tensorflow.org/>), a machine learning/deep learning library developed by Google, became open to the public and attracted great attention.

When we look at the programming language with which libraries are being developed, most of them open to the public are developed by Python or use the Python API. TensorFlow is developed with C++ on the backend but it's also possible to write code with Python. This book focuses on Java to learn deep learning, hence the libraries developed by other languages will be briefly introduced in *Chapter 7, Other Important Deep Learning Libraries*.

So, what Java-based libraries do we have? Actually, there are a few cases that are actively developed (perhaps there are also some projects not open to public). However, there is only one library we can use practically: **Deeplearning4j (DL4J)**. The official project page URL is <http://deeplearning4j.org/>. This library is also open source and the source code is all published on GitHub. The URL is <https://github.com/deeplearning4j/deeplearning4j>. This library was developed by Skymind (<http://www.skymind.io/>). What kind of library is this? If you look at the project page, it's introduced as follows:

"Deeplearning4j is the first commercial-grade, open-source, distributed deep-learning library written for Java and Scala. Integrated with Hadoop and Spark, DL4J is designed to be used in business environments, rather than as a research tool. Skymind is its commercial support arm.

Deeplearning4j aims to be cutting-edge plug and play, more convention than configuration, which allows for fast prototyping for non-researchers. DL4J is customizable at scale. Released under the Apache 2.0 license, all derivatives of DL4J belong to their authors."

When you read this, you will see that the biggest feature of DL4J is that it was designed on the premise of being integrated with Hadoop. This indicates that DL4J suits the processing of large-scale data and is more scalable than other libraries. Moreover, DL4J supports GPU computing, so it's possible to process data even faster.

Also, DL4J uses a library called **N-Dimensional Arrays for Java (ND4J)** internally. The project page is <http://nd4j.org/>. The same as DL4J, this library is also published on GitHub as an open source project: <https://github.com/deeplearning4j/nd4j>. The developer of the library is the same as DL4J, Skymind. As you can see from the name of the library, this is a scientific computing library that enables us to handle versatile n -dimensional array objects. If you are a Python developer, it might be easier for you to understand this if you imagine NumPy, as ND4J is a library inspired by NumPy. ND4J also supports GPU computing and the reason why DL4J is able to do GPU integration is because it uses ND4J internally.

What good can come from working with them on GPUs? Let's briefly look at this point. The biggest difference between CPU and GPU is the difference in the number of cores. GPU is, as represented in its name, a graphical processing unit, originally an integrated circuit for image processing. This is why GPU is well optimized to handle the same commands simultaneously. Parallel processing is its forte. On the other hand, as CPU needs to process various commands, these tasks are basically made to be processed in order. Compared to CPU, GPU is good at processing huge numbers of simple tasks, therefore calculations such as training iterations of deep learning is its field of expertise.

Both ND4J and DL4J are very useful for research and data mining with deep learning. From the next section on, we'll see how these are used for deep learning in simple examples. You can easily understand the contents because you should already understand the core theories of deep learning by now. Hopefully, you can make use of this for your fields of study or business.

Implementations with ND4J

As there are many cases where ND4J alone can be used conveniently, let's briefly grasp how to use ND4J before looking into the explanation of DL4J. If you would like to use ND4J alone, once you create a new Maven project, then you can use ND4J by adding the following code to pom.xml:

```
<properties>
    <nd4j.version>0.4-rc3.6</nd4j.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.nd4j</groupId>
        <artifactId>nd4j-jblas</artifactId>
        <version>${nd4j.version}</version>
    </dependency>
    <dependency>
        <groupId>org.nd4j</groupId>
        <artifactId>nd4j-perf</artifactId>
        <version>${nd4j.version}</version>
    </dependency>
</dependencies>
```

Here, `<nd4j.version>` describes the latest version of ND4J, but please check whether it is updated when you actually implement the code. Also, switching from CPU to GPU is easy while working with ND4J. If you have CUDA installed with version 7.0, then what you do is just define `artifactId` as follows:

```
<dependency>
    <groupId>org.nd4j</groupId>
    <artifactId>nd4j-jcublas-7.0</artifactId>
    <version>${nd4j.version}</version>
</dependency>
```

You can replace the version of `<artifactId>` depending on your configuration.

Let's look at a simple example of what calculations are possible with ND4J. The type we utilize with ND4J is `INDArray`, that is, an extended type of `Array`. We begin by importing the following dependencies:

```
import org.nd4j.linalg.api.ndarray.INDArray;
import org.nd4j.linalg.factory.Nd4j;
```

Then, we define `INDArray` as follows:

```
INDArray x = Nd4j.create(new double[]{1, 2, 3, 4, 5, 6}, new
int[]{3, 2});
System.out.println(x);
```

`Nd4j.create` takes two arguments. The former defines the actual values within `INDArray`, and the latter defines the shape of the vector (matrix). By running this code, you get the following result:

```
[[1.00,2.00]
[3.00,4.00]
[5.00,6.00]]
```

Since `INDArray` can output its values with `System.out.print`, it's easy to debug. Calculation with scalar can also be done with ease. Add 1 to `x` as shown here:

```
x.add(1);
```

Then, you will get the following output:

```
[[2.00,3.00]
[4.00,5.00]
[6.00,7.00]]
```

Also, the calculation within `INDArray` can be done easily, as shown in the following example:

```
INDArray y = Nd4j.create(new double[]{6, 5, 4, 3, 2, 1}, new
int[]{3, 2});
```

Then, basic arithmetic operations can be represented as follows:

```
x.add(y)
x.sub(y)
x.mul(y)
x.div(y)
```

These will return the following result:

```
[[7.00,7.00]
[7.00,7.00]
[7.00,7.00]]
[[-5.00,-3.00]
[-1.00,1.00]
[3.00,5.00]]
[[6.00,10.00]]
```

```
[12.00,12.00]
[10.00,6.00]
[[0.17,0.40]
[0.75,1.33]
[2.50,6.00]]
```

Also, ND4J has destructive arithmetic operators. When you write the `x.addi(y)` command, `x` changes its own values so that `System.out.println(x)` ; will return the following output:

```
[[7.00,7.00]
[7.00,7.00]
[7.00,7.00]]
```

Likewise, `subi`, `muli`, and `divi` are also destructive operators. There are also many other methods that can conveniently perform calculations between vectors or matrices. For more information, you can refer to <http://nd4j.org/documentation.html>, <http://nd4j.org/doc/> and <http://nd4j.org/apidocs/>.

Let's look at one more example to see how machine learning algorithms can be written with ND4J. We'll implement the easiest example, perceptrons, based on the source code written in *Chapter 2, Algorithms for Machine Learning – Preparing for Deep Learning*. We set the package name `DLWJ.examples.ND4J` and the file (class) name `Perceptrons.java`.

First, let's add these two lines to import from ND4J:

```
import org.nd4j.linalg.api.ndarray.INDArray;
import org.nd4j.linalg.factory.Nd4j;
```

The model has two parameters: `num` of the input layer and the weight. The former doesn't change from the previous code; however, the latter isn't `Array` but `INDArray`:

```
public int nIn;           // dimensions of input data
public INDArray w;
```

You can see from the constructor that since the weight of the perceptrons is represented as a vector, the number of rows is set to the number of units in the input layer and the number of columns to 1. This definition is written here:

```
public Perceptrons(int nIn) {

    this.nIn = nIn;
    w = Nd4j.create(new double[nIn], new int[]{nIn, 1});

}
```

Then, because we define the model parameter as `INDArray`, we also define the demo data, training data, and test data as `INDArray`. You can see these definitions at the beginning of the main method:

```
INDArray train_X = Nd4j.create(new double[train_N * nIn], new int []
{train_N, nIn}); // input data for training
INDArray train_T = Nd4j.create(new double[train_N], new int [] {train_N,
1}); // output data (label) for training

INDArray test_X = Nd4j.create(new double[test_N * nIn], new int []
{test_N, nIn}); // input data for test
INDArray test_T = Nd4j.create(new double[test_N], new int [] {test_N,
1}); // label of inputs
INDArray predicted_T = Nd4j.create(new double[test_N], new int []
{test_N, 1}); // output data predicted by the model
```

When we substitute a value into `INDArray`, we use `put`. Please be careful that any value we can set with `put` is only the values of the scalar type:

```
train_X.put(i, 0, Nd4j.scalar(g1.random()));
train_X.put(i, 1, Nd4j.scalar(g2.random()));
train_T.put(i, Nd4j.scalar(1));
```

The flow from a model building and training is the same as the previous code:

```
// construct perceptrons
Perceptrons classifier = new Perceptrons(nIn);

// train models
while (true) {
    int classified_ = 0;

    for (int i=0; i < train_N; i++) {
        classified_ += classifier.train(train_X.getRow(i),
        train_T.getRow(i), learningRate);
    }

    if (classified_ == train_N) break; // when all data classified
    correctly

    epoch++;
    if (epoch > epochs) break;
}
```

Each piece of training data is given to the `train` method by `getRow()`. First, let's see the entire content of the `train` method:

```
public int train(INDArray x, INDArray t, double learningRate) {  
  
    int classified = 0;  
  
    // check if the data is classified correctly  
    double c = x.mmul(w).getDouble(0) * t.getDouble(0);  
  
    // apply steepest descent method if the data is wrongly  
    // classified  
    if (c > 0) {  
        classified = 1;  
    } else {  
        w.addi(x.transpose().mul(t).mul(learningRate));  
    }  
  
    return classified;  
}
```

We first focus our attention on the following code:

```
// check if the data is classified correctly  
double c = x.mmul(w).getDouble(0) * t.getDouble(0);
```

This is the part that checks whether the data is classified correctly by perceptions, as shown in the following equation:

$$w^T x_n t_n > 0$$

You can see from the code that `.mmul()` is for the multiplication between vectors or matrices. We wrote this part of the calculation in *Chapter 2, Algorithms for Machine Learning – Preparing for Deep Learning*, as follows:

```
double c = 0.;  
  
// check if the data is classified correctly  
for (int i = 0; i < nIn; i++) {  
    c += w[i] * x[i] * t;  
}
```

By comparing both codes, you can see that multiplication between vectors or matrices can be written easily with `INDArray`, and so you can implement the algorithm intuitively just by following the equations.

The equation to update the model parameters is as follows:

```
w.addi(x.transpose().mul(t).mul(learningRate));
```

Here, again, you can implement the code like you write a math equation. The equation is represented as follows:

$$w^{(k+1)} = w^{(k)} + \eta x_n t_n$$

The last time we implemented this part, we wrote it with a `for` loop:

```
for (int i = 0; i < nIn; i++) {
    w[i] += learningRate * x[i] * t;
}
```

Furthermore, the prediction after the training is also the standard forward activation, shown as the following equation:

$$y(x) = f(w^T x)$$

Here:

$$f(a) = \begin{cases} +1, & a \geq 0 \\ -1, & a < 0 \end{cases}$$

We can simply define the `predict` method with just a single line inside, as follows:

```
public int predict(INDArray x) {
    return step(x.mmul(w).getDouble(0));
}
```

When you run the program, you can see its precision and accuracy, and the recall is the same as we get with the previous code.

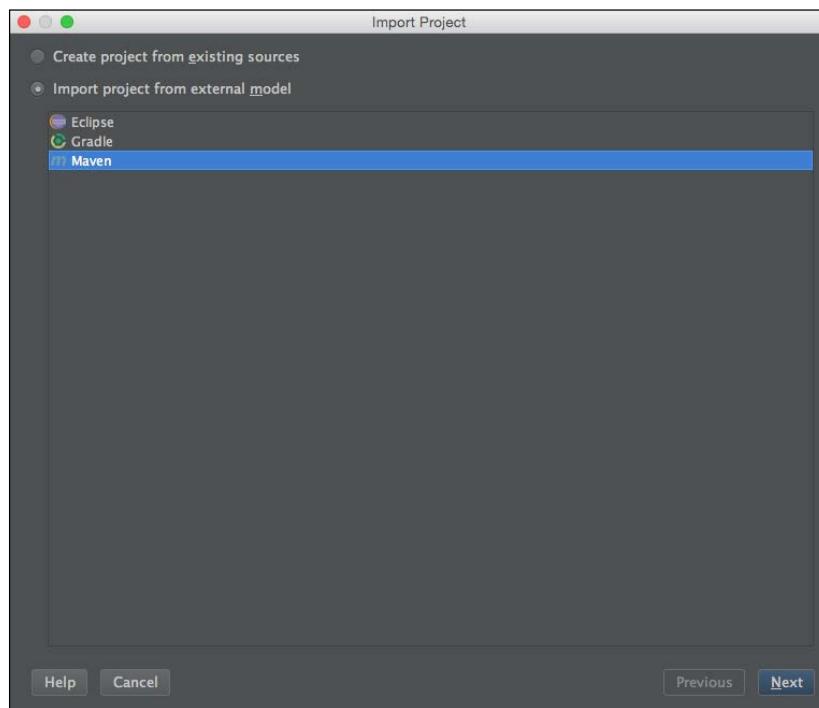
Thus, it'll greatly help that you implement the algorithms analogous to mathematical equations. We only implement perceptrons here, but please try other algorithms by yourself.

Implementations with DL4J

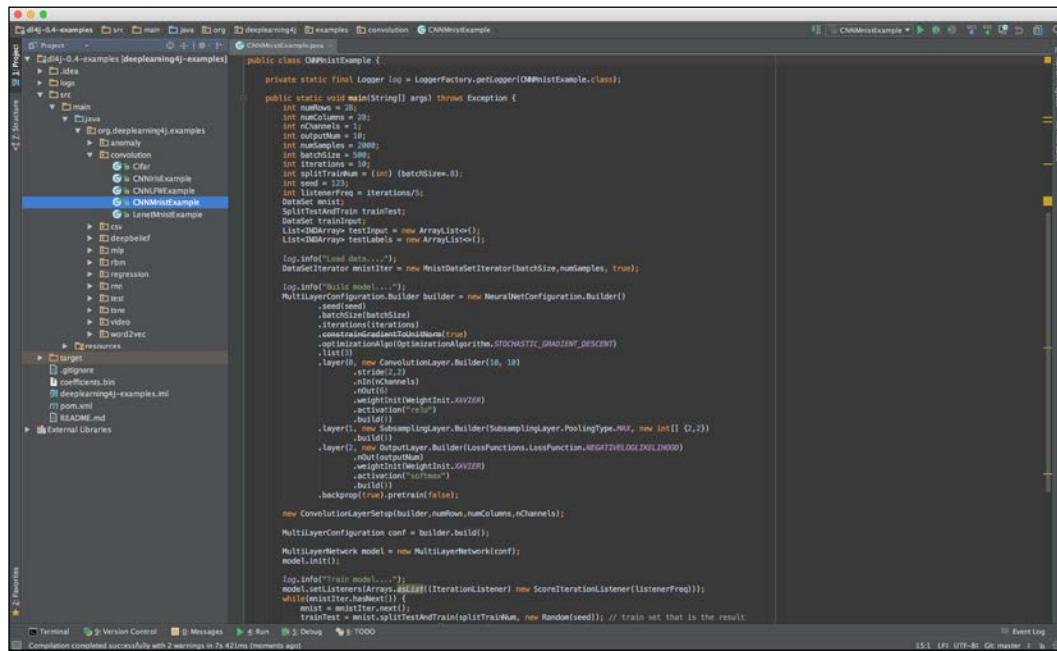
ND4J is the library that helps you to implement deep learning easily and conveniently. However, you have to implement the algorithms yourself, which is not too different from its implementation in the previous chapters. In other words, ND4J is just a library that makes calculating numerical values easier and is not a library that is optimized for deep learning algorithms. One library that makes deep learning easier to handle is DL4J. Fortunately, as for DL4J, some example code with typical methods is published on GitHub (<https://github.com/deeplearning4j/dl4j-0.4-examples>). These examples are used on the premise that you are using DL4J's version 0.4-*. When you actually clone this repository, please check the latest version again. In this section, we'll extract the fundamental part from these sample programs and take a look at it. We'll reference the forked repository on <https://github.com/yusugomori/dl4j-0.4-examples> as a screenshot in this section.

Setup

Let's first set up the environments from our cloned repository. If you're using IntelliJ, you can import the project from **File | New | Project** from existing sources and select the path of the repository. Then, choose **Import project from external model** and select **Maven** as follows:



You don't have to do anything special for the other steps except click **Next**. Please be careful that the supported versions of JDK are 1.7 or above. This may not be a problem because we needed version 1.8 or above in the previous chapters. Once you have set it up without a problem, you can confirm the structure of the directories as follows:



Once you have set up the project, let's first look at `pom.xml`. You can see that the description of the packages related to DL4J is written as:

```
<dependency>
    <groupId>org.deeplearning4j</groupId>
    <artifactId>deeplearning4j-nlp</artifactId>
    <version>${dl4j.version}</version>
</dependency>

<dependency>
    <groupId>org.deeplearning4j</groupId>
    <artifactId>deeplearning4j-core</artifactId>
    <version>${dl4j.version}</version>
</dependency>
```

Also, you can see from the following lines that DL4J depends on ND4J:

```
<dependency>
  <groupId>org.nd4j</groupId>
  <artifactId>nd4j-x86</artifactId>
  <version>${nd4j.version}</version>
</dependency>
```

If you would like to run a program on GPU, what you have to do is just change this written section. As mentioned in the previous section, this can be written as follows if you have CUDA installed:

```
<dependency>
  <groupId>org.nd4j</groupId>
  <artifactId>nd4j-jcublas-XXX</artifactId>
  <version>${nd4j.version}</version>
</dependency>
```

Here, XXX is the version of CUDA and depends on your machine's preference. It's great to adopt GPU computing only using this. We don't have to do anything special and we can focus on implementations of deep learning.

The other characteristic library that DL4J develops and uses is **Canova**. The part that corresponds to pom.xml is as follows:

```
<dependency>
  <artifactId>anova-nd4j-image</artifactId>
  <groupId>org.nd4j</groupId>
  <version>${anova.version}</version>
</dependency>
<dependency>
  <artifactId>anova-nd4j-codec</artifactId>
  <groupId>org.nd4j</groupId>
  <version>${anova.version}</version>
</dependency>
```

Anova is also, of course, an open source library and its source code can be seen on GitHub at <https://github.com/deeplearning4j/Anova>. As explained on that page, Anova is the library used to vectorize raw data into usable vector formats across the machine learning tools. This also helps us focus on the more important part of data mining because data formatting is indispensable in whatever research or experiment we're performing.

Build

Let's look at the source code in the examples and see how to build a deep learning model. During the process, the terms of deep learning that you haven't yet learned are also briefly explained. The examples are implemented with various models such as MLP, DBN, and CNN, but there is one problem here. As you can see when looking at `README.md`, some methods don't generate good precision. This is because, as explained in the previous section, the calculation precision a machine has is limited and fluctuation occurring with calculated values during the process depends completely on the difference of implementation. Hence, practically, learning can't be done properly, although theoretically it should be done well. You can get better results by, for example, changing the seed values or adjusting the parameters, but as we would like to focus on how to use a library, we'll use a model that gets higher precision as an example.

DBNIrisExample.java

Let's first look at `DBNIrisExample.java` in the package of `deepbelief.Iris`, contained in the filename, is one of the benchmark datasets often used when measuring the precision or accuracy of a machine learning method. The dataset contains 150 pieces of data out of 3 classes of 50 instances each, and each class refers to a type of Iris plant. The number of inputs is 4 and the number of outputs is therefore 3. One class is linearly separable from the other two; the latter are not linearly separable from each other.

The implementation begins by setting up the configuration. Here are the variables that need setting:

```
final int numRows = 4;
final int numColumns = 1;
int outputNum = 3;
int numSamples = 150;
int batchSize = 150;
int iterations = 5;
int splitTrainNum = (int) (batchSize * .8);
int seed = 123;
int listenerFreq = 1;
```

In DL4J, input data can be up to two-dimensional data, hence you need to assign the number of rows and columns of the data. As Iris is one-dimensional data, `numColumns` is set as 1. Here `numSamples` is the total data and `batchSize` is the amount of data in each mini-batch. Since the total data is 150 and it is relatively small, `batchSize` is set at the same number. This means that learning is done without splitting the data into mini-batches. `splitTrainNum` is the variable that decides the allocation between the training data and test data. Here, 80% of all the dataset is training data and 20% is the test data. In the previous section, `listenerFreq` decides how often we see loss function's value for logging is seen in the process. This value is set to 1 here, which means the value is logged after each epoch.

Subsequently, we need to fetch the dataset. In DL4J, a class that can easily fetch data with respect to a typical dataset, such as Iris, MINST, and LFW, is prepared. Therefore, you can just write the following line if you would like to fetch the Iris dataset:

```
DataSetIterator iter = new IrisDataSetIterator(batchSize, numSamples);
```

The following two lines are to format data:

```
DataSet next = iter.next();
next.normalizeZeroMeanZeroUnitVariance();
```

This code splits the data into training data and test data and stores them respectively:

```
SplitTestAndTrain testAndTrain =
next.splitTestAndTrain(splitTrainNum, new Random(seed));
DataSet train = testAndTrain.getTrain();
DataSet test = testAndTrain.getTest();
```

As you can see, it makes data handling easier by treating all the data DL4J prepares with the `DataSet` class.

Now, let's actually build a model. The basic structure is as follows:

```
MultiLayerConfiguration conf = new
NeuralNetConfiguration.Builder().layer().layer() ... .layer().build();
MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
```

The code begins by defining the model configuration and then builds and initializes the actual model with the definition. Let's take a look at the configuration details. At the beginning, the whole network is set up:

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
.seed(seed)
.iterations(iterations)
```

```
.learningRate(1e-6f)
.optimizationAlgo(OptimizationAlgorithm.CONJUGATE_GRADIENT)
.l1(1e-1).regularization(true).l2(2e-4)
.useDropConnect(true)
.list(2)
```

The configuration setup is self-explanatory. However, since you haven't learned about regularization before now, let's briefly check it out.

Regularization prevents the neural networks model from overfitting and makes the model more generalized. To achieve this, the evaluation function $E(w)$ is rewritten with the penalty term as follows:

$$E(w) + \lambda \frac{1}{p} \|w\|_p^p = E(w) + \lambda \frac{1}{p} \sum_i |w_i|^p$$

Here, $\|\cdot\|$ denotes the vector norm. The regularization is called L1 regularization when $p=1$ and L2 regularization when $p=2$. The norm is called L1 norm and L2 norm, respectively. That's why we have `.l1()` and `.l2()` in the code. λ is the hyper parameter. These regularization terms make the model more sparse. L2 regularization is also called weight decay and is used to prevent the vanishing gradient problem.

The `.useDropConnect()` command is used to enable dropout and `.list()` to define the number of layers, excluding the input layer.

When you set up a whole model, then the next step is to configure each layer. In this sample code, the model is not defined as deep neural networks. One single RBM layer is defined as a hidden layer:

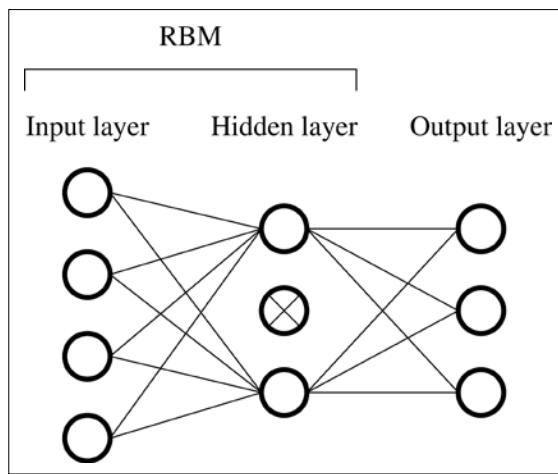
```
.layer(0, new RBM.Builder(RBM.HiddenUnit.RECTIFIED, RBM.VisibleUnit.
GAUSSIAN)
.nIn(numRows * numColumns)
.nOut(3)
.weightInit(WeightInit.XAVIER)
.k(1)
.activation("relu")
.lossFunction(LossFunctions.LossFunction.RMSE_XENT)
.updater(Updater.ADAGRAD)
.dropOut(0.5)
.build()
)
```

Here, the value of 0 in the first line is the layer's index and `.k()` is for contrastive divergence. Since Iris' data is of float values, we can't use binary RBM. That's why we have `RBM.VisibleUnit.GAUSSIAN` here, enabling the model to handle continuous values. Also, as for the definition of this layer, what should be especially mentioned is the role of `Updater.ADAGRAD`. This is used to optimize the learning rate. For now, we go on to the model structure, and a detailed explanation of the optimizer will be introduced at the end of this chapter.

The subsequent output layer is very simple and self-explanatory:

```
.layer(1, new OutputLayer.Builder(LossFunctions.LossFunction.MCXENT)
    .nIn(3)
    .nOut(outputNum)
    .activation("softmax")
    .build()
)
```

Thus, the neural networks have been built with three layers : input layer, hidden layer, and output layer. The graphical model of this example can be illustrated as follows:



After the model building, we need to train the networks. Here, again, the code is super simple:

```
model.setListeners(Arrays.asList((IterationListener) new
ScoreIterationListener(listenerFreq)));
model.fit(train);
```

Because the first line is to log the process, what we need to do to train the model is just to write `model.fit()`.

Testing or evaluating the model is also easy with DL4J. First, the variables for evaluation are set up as follows:

```
Evaluation eval = new Evaluation(outputNum);
INDArray output = model.output(test.getFeatureMatrix());
```

Then, we can get the values of the feature matrix using:

```
eval.eval(test.getLabels(), output);
log.info(eval.stats());
```

By running the code, we will have the result as follows:

```
=====
Scores=====
Accuracy: 0.7667
Precision: 1
Recall: 0.7667
F1 Score: 0.8679245283018869
=====
```

F1 Score, also called F-Score or F-measure, is the harmonic means of precision and recall, and is represented as follows:

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

This value is often calculated to measure the model's performance as well. Also, as written in the example, you can see the actual values and predicted values by writing the following:

```
for (int i = 0; i < output.rows(); i++) {
    String actual = test.getLabels().getRow(i).toString().trim();
    String predicted = output.getRow(i).toString().trim();
    log.info("actual " + actual + " vs predicted " + predicted);
}
```

That's it for the whole training and test process. The neural networks in the preceding code are not deep, but you can easily build deep neural networks just by changing the configuration as follows:

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(seed)
    .iterations(iterations)
    .learningRate(1e-6f)
```

```
.optimizationAlgo(OptimizationAlgorithm.CONJUGATE_GRADIENT)
    .l1(1e-1).regularization(true).l2(2e-4)
    .useDropConnect(true)
    .list(3)
        .layer(0, new RBM.Builder(RBM.HiddenUnit.RECTIFIED, RBM.
VisibleUnit.GAUSSIAN)
            .nIn(numRows * numColumns)
            .nOut(4)
            .weightInit(WeightInit.XAVIER)
            .k(1)
            .activation("relu")
            .lossFunction(LossFunctions.LossFunction.RMSE_
XENT)
            .updater(Updater.ADAGRAD)
            .dropOut(0.5)
            .build()
        )
        .layer(1, new RBM.Builder(RBM.HiddenUnit.RECTIFIED, RBM.
VisibleUnit.GAUSSIAN)
            .nIn(4)
            .nOut(3)
            .weightInit(WeightInit.XAVIER)
            .k(1)
            .activation("relu")
            .lossFunction(LossFunctions.LossFunction.RMSE_
XENT)
            .updater(Updater.ADAGRAD)
            .dropOut(0.5)
            .build()
        )
        .layer(2, new OutputLayer.Builder(LossFunctions.LossFunction.
MCXENT)
            .nIn(3)
            .nOut(outputNum)
            .activation("softmax")
            .build()
        )
    .build();
```

As you can see, building deep neural networks requires just simple implementations with DL4J. Once you set up the model, what you need to do is adjust the parameters. For example, increasing the iterations value or changing the seed value would return a better result.

CSVExample.java

In the previous example, we train the model with the dataset used as a benchmark indicator. When you would like to train and test the model with your own prepared data, you can easily import it from CSV. Let's look at `CSVExample.java` in the `CSV` package. The first step is to initialize the CSV reader as follows:

```
RecordReader recordReader = new CSVRecordReader(0, ",");
```

In DL4J, a class called `CSVRecordReader` is prepared and you can easily import data from a CSV file. The value of the first argument in the `CSVRecordReader` class represents how many lines should be skipped in the file. This is convenient when the file contains header rows. The second argument is the delimiter. To actually read a file and import data, the code can be written as follows:

```
recordReader.initialize(new FileSplit(new  
ClassPathResource("iris.txt").getFile()));
```

With this code, the file in `resources/iris.txt` will be imported to the model. The values in the file here are the same as ones as in the Iris dataset. To use this initialized data for model training, we define the iterator as follows:

```
DataSetIterator iterator = new RecordReaderDataSetIterator(recordRead  
er, 4, 3);  
DataSet next = iterator.next();
```

In the previous example, we used the `IrisDataSetIterator` class, but here the `RecordReaderDataSetIterator` class is used because we use our own prepared data. The values 4 and 3 are the number of features and labels, respectively.

Building and training a model can be done in almost the same way as the process explained in the previous example. In this example, we build deep neural networks of two hidden layers with the dropout and the rectifier, that is, we have an input layer - hidden layer - hidden layer - output layer, as follows:

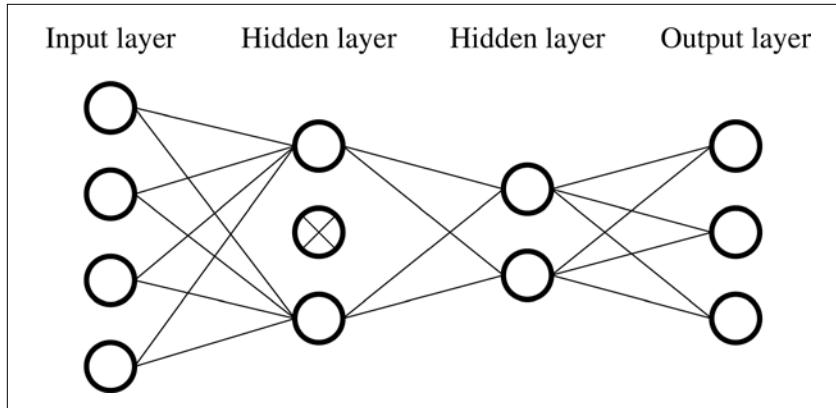
```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()  
.seed(seed)  
.iterations(iterations)  
.constrainGradientToUnitNorm(true).useDropConnect(true)  
.learningRate(1e-1)  
.l1(0.3).regularization(true).l2(1e-3)  
.constrainGradientToUnitNorm(true)  
.list(3)  
.layer(0, new DenseLayer.Builder().nIn(numInputs).nOut(3)  
.activation("relu").dropOut(0.5)  
.weightInit(WeightInit.XAVIER)  
.build())
```

```
.layer(1, new DenseLayer.Builder().nIn(3).nOut(2)
      .activation("relu")
      .weightInit(WeightInit.XAVIER)
      .build())
.layer(2, new
OutputLayer.Builder(LossFunctions.LossFunction
.NEGATIVELOGLIKELIHOOD)
      .weightInit(WeightInit.XAVIER)
      .activation("softmax")
      .nIn(2).nOut(outputNum).build())
.backprop(true).pretrain(false)
.build();
```

We can run the model using the following lines of code:

```
MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
```

The graphical model is as follows:



This time, however, the way to code for training is slightly different from the previous example. Before, we split the data into training data and test data using the following:

```
SplitTestAndTrain testAndTrain =
next.splitTestAndTrain(splitTrainNum, new Random(seed));
```

This shows that we shuffle the data within the `.splitTestAndTrain()` method. In this example, we set up training data with the following code:

```
next.shuffle();
SplitTestAndTrain testAndTrain = next.splitTestAndTrain(0.6);
```

As you can see, here the data is first shuffled and then split into training data and test data. Be careful that the types of the arguments in `.splitTestAndTrain()` are different from each other. This will be beneficial because we don't have to count the exact amount of data or training data. The actual training is done using:

```
model.fit(testAndTrain.getTrain());
```

The way to evaluate the model is just the same as the previous example:

```
Evaluation eval = new Evaluation(3);
DataSet test = testAndTrain.getTest();
INDArray output = model.output(test.getFeatureMatrix());
eval.eval(test.getLabels(), output);
log.info(eval.stats());
```

With the preceding code, we get the following result:

```
=====
Scores=====
Accuracy: 1
Precision: 1
Recall: 1
F1 Score: 1.0
=====
```

In addition to the dataset of a benchmark indicator, you can now analyze whatever data you have.

To make the model even deeper, you just need to add another layer as follows:

```
MultiLayerConfiguration conf = new
NeuralNetConfiguration.Builder()
.seed(seed)
.iterations(iterations)
.constrainGradientToUnitNorm(true).useDropConnect(true)
.learningRate(0.01)
.l1(0.0).regularization(true).l2(1e-3)
.constrainGradientToUnitNorm(true)
.list(4)
.layer(0, new DenseLayer.Builder().nIn(numInputs).nOut(4)
.activation("relu").dropOut(0.5)
.weightInit(WeightInit.XAVIER)
.build())
.layer(1, new DenseLayer.Builder().nIn(4).nOut(4)
.activation("relu").dropOut(0.5)
.weightInit(WeightInit.XAVIER)
```

```
        .build())
    .layer(2, new DenseLayer.Builder().nIn(4).nOut(4)
        .activation("relu").dropOut(0.5)
        .weightInit(WeightInit.XAVIER)
        .build())
    .layer(3, new
OutputLayer.Builder(LossFunctions.LossFunction
.NEGATIVELOGLIKELIHOOD)
        .weightInit(WeightInit.XAVIER)
        .activation("softmax")
        .nIn(4).nOut(outputNum).build())
    .backprop(true).pretrain(false)
    .build();
```

CNNMnistExample.java/LenetMnistExample.java

CNN is rather complicated compared to other models because of its structure, but we don't need to worry about these complications because we can easily implement CNN with DL4J. Let's take a look at `CNNMnistExample.java` in the package of convolution. In this example, we train the model with the MNIST dataset (<http://yann.lecun.com/exdb/mnist/>), one of the most famous benchmark indicators. As mentioned in *Chapter 1, Deep Learning Overview*, this dataset contains 70,000 handwritten numbers data from 0 to 9, with both a height and width of 28 pixels each.

First, we define the values necessary for the model:

```
int numRows = 28;
int numColumns = 28;
int nChannels = 1;
int outputNum = 10;
int numSamples = 2000;
int batchSize = 500;
int iterations = 10;
int splitTrainNum = (int) (batchSize*.8);
int seed = 123;
int listenerFreq = iterations/5;
```

Since images in MNIST are all grayscale data, the number of channels is set to 1. In this example, we use 2,000 data of 70,000 and split it into training data and test data. The size of the mini-batch is 500 here, so the training data is divided into 4 mini-batches. Furthermore, the data in each mini-batch is split into training data and test data, and each piece of test data is stored in `ArrayList`:

```
List<INDArray> testInput = new ArrayList<>();
List<INDArray> testLabels = new ArrayList<>();
```

We didn't have to set `ArrayList` in the previous examples because we had just one batch. For the `MnistDataSetIterator` class, we can set the MNIST data just by using:

```
DataSetIterator mnistIter = new
MnistDataSetIterator(batchSize, numSamples, true);
```

Then, we build the model with a convolutional layer and subsampling layer. Here, we have one convolutional layer and one max-pooling layer, directly followed by an output layer. The structure of the configurations for CNN is slightly different from the other algorithms:

```
MultiLayerConfiguration.Builder builder = new NeuralNetConfiguration.
Builder().layer().layer() . ... .layer()
new ConvolutionLayerSetup(builder, numRows, numColumns, nChannels);
MultiLayerConfiguration conf = builder.build();
MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
```

The difference is that we can't build a model directly from the configuration because we need to tell the builder to set up a convolutional layer using `ConvolutionLayerSetup()` in advance. Each `.layer()` requires just the same method of coding. The convolutional layer is defined as:

```
.layer(0, new ConvolutionLayer.Builder(10, 10)
    .stride(2,2)
    .nIn(nChannels)
    .nOut(6)
    .weightInit(WeightInit.XAVIER)
    .activation("relu")
    .build())
```

Here, the value of 10 in `ConvolutionLayer.Builder()` is the size of the kernels, and the value of 6 in `.nOut()` is the number of kernels. Also, `.stride()` defines the size of the strides of the kernels. The code we implemented from scratch in *Chapter 4, Dropout and Convolutional Neural Networks* has a functionality equivalent only to `.stride(1, 1)`. The larger the number is, the less time it takes because it decreases the number of calculations necessary for convolutions, but we have to be careful at the same time that it might also decrease the model's precision. Anyway, we can implement convolutions with more flexibility now.

The subsampling layer is described as:

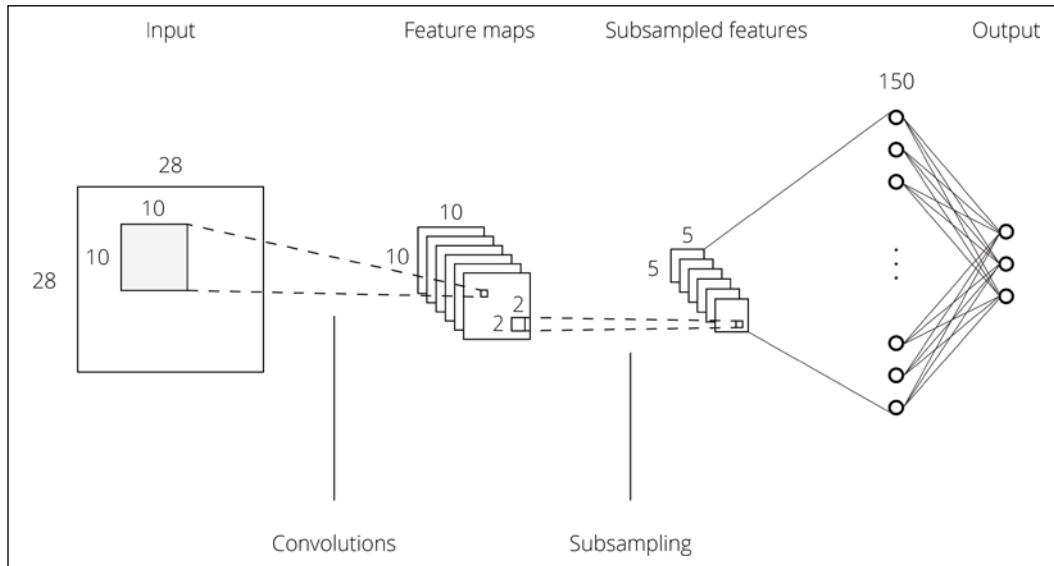
```
.layer(1, new  
SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX, new int []  
{2,2})  
.build())
```

Here, `{2, 2}` is the size of the pooling windows. You may have noticed that we don't have to set the size of the inputs for each layer, including the output layer. These values are automatically set once you set up the model.

The output layer can be written just the same as in the other models:

```
.layer(2, new OutputLayer.Builder(LossFunctions.LossFunction.  
NEGATIVELOGLIKELIHOOD)  
.nOut(outputNum)  
.weightInit(WeightInit.XAVIER)  
.activation("softmax")  
.build())
```

The graphical model of this example is as follows:



After the building comes the training. Since we have multiple mini-batches, we need to iterate training through all the batches. This can be achieved easily using `.hasNext()` on `DataSetIterator` and `mnistIter` in this case. The whole training process can be written as follows:

```
model.setListeners(Arrays.asList((IterationListener) new ScoreIteratio
nListener(listenerFreq)));
while(mnistIter.hasNext()) {
    mnist = mnistIter.next();
    trainTest = mnist.splitTestAndTrain(splitTrainNum, new
    Random(seed));
    trainInput = trainTest.getTrain();
    testInput.add(trainTest.getTest().getFeatureMatrix());
    testLabels.add(trainTest.getTest().getLabels());
    model.fit(trainInput);
}
```

Here, the test data and test labels are stocked for further use.

During the test, again, we need to iterate the evaluation process of the test data because we have more than one mini-batch:

```
for(int i = 0; i < testInput.size(); i++) {  
    INDArray output = model.output(testInput.get(i));  
    eval.eval(testLabels.get(i), output);  
}
```

Then, we use the same as in the other examples:

```
log.info(eval.stats());
```

This will return the result as follows:

```
=====Scores=====  
Accuracy: 0.832  
Precision: 0.8783  
Recall: 0.8334  
F1 Score: 0.8552464933704985  
=====
```

The example just given is the model with one convolutional layer and one subsampling layer, but you have deep convolutional neural networks with LenetMnistExample.java. In this example, there are two convolutional layers and subsampling layers, followed by fully connected multi-layer perceptrons:

```
MultiLayerConfiguration.Builder builder = new NeuralNetConfiguration.  
Builder()  
    .seed(seed)  
    .batchSize(batchSize)  
    .iterations(iterations)  
    .regularization(true).l2(0.0005)  
    .learningRate(0.01)  
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_  
DESCENT)  
    .updater(Updater.NESTEROVS).momentum(0.9)  
    .list(6)  
    .layer(0, new ConvolutionLayer.Builder(5, 5)  
        .nIn(nChannels)  
        .stride(1, 1)  
        .nOut(20).dropOut(0.5)  
        .weightInit(WeightInit.XAVIER)  
        .activation("relu")  
        .build())
```

```
.layer(1, new  
SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX,  
new int[]{2, 2})  
.build())  
.layer(2, new ConvolutionLayer.Builder(5, 5)  
.nIn(20)  
.nOut(50)  
.stride(2,2)  
.weightInit(WeightInit.XAVIER)  
.activation("relu")  
.build())  
.layer(3, new  
SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX,  
new int[]{2, 2})  
.build())  
.layer(4, new DenseLayer.Builder().activation("tanh")  
.nOut(500).build())  
.layer(5, new  
OutputLayer.Builder(LossFunctions.LossFunction  
.NEGATIVELOGLIKELIHOOD)  
.nOut(outputNum)  
.weightInit(WeightInit.XAVIER)  
.activation("softmax")  
.build())  
.backprop(true).pretrain(false);  
new ConvolutionLayerSetup(builder,28,28,1);
```

As you can see in the first convolutional layer, dropout can easily be applied to CNN with DL4J.

With this model, we get the following result:

```
=====Scores=====  
Accuracy: 0.8656  
Precision: 0.8827  
Recall: 0.8645  
F1 Score: 0.873490476878917  
=====
```

You can see from the MNIST dataset page (<http://yann.lecun.com/exdb/mnist/>) that the state-of-the-art result is much better than the one above. Here, again, you would realize how important the combination of parameters, activation functions, and optimization algorithms are.

Learning rate optimization

We have learned various deep learning algorithms so far; you may have noticed that they have one parameter in common: the learning rate. The learning rate is defined in the equations to update the model parameters. So, why not think of algorithms to optimize the learning rate? Originally, these equations were described as follows:

$$\theta^{(\tau+1)} = \theta^{(\tau)} + \Delta\theta^{(\tau)}$$

Here:

$$\Delta\theta^{(\tau)} = -\eta \frac{\partial E}{\partial \theta^{(\tau)}}$$

Here, τ is the number of steps and η is the learning rate. It is well known that decreasing the value of the learning rate with each iteration lets the model have better precision rates, but we should determine the decline carefully because a sudden drop in the value would collapse the model. The learning rate is one of the model parameters, so why not optimize it? To do so, we need to know what the best rate could be.

The simplest way of setting the rate is using the momentum, represented as follows:

$$\Delta\theta^{(\tau)} = -\eta \frac{\partial E}{\partial \theta^{(\tau)}} + \alpha \Delta\theta^{(\tau-1)}$$

Here, $\alpha \in [0,1]$, called the **momentum coefficient**. This hyper parameter is often set to be 0.5 or 0.9 first and then fine-tuned.

Momentum is actually a simple but effective way of adjusting the learning rate but **ADAGRAD**, proposed by Duchi et al. (<http://www.magicbroom.info/Papers/DuchiHaSi10.pdf>), is known to be a better way. The equation is described as follows:

$$\Delta\theta^{(\tau)} = -\frac{\eta}{\sqrt{\sum_{t=0}^{\tau} g_t^2}} g_{\tau}$$

Here:

$$g_\tau = \frac{\partial E}{\partial \theta^{(\tau)}}$$

Theoretically, this works well, but practically, we often use the following equations to prevent divergence:

$$\Delta \theta^{(\tau)} = -\frac{\eta}{\sqrt{\sum_{t=0}^{\tau} g_t^2} + 1} g_\tau$$

Or we use:

$$\Delta \theta^{(\tau)} = -\frac{\eta}{\sqrt{\sum_{t=0}^{\tau} g_t^2} + 1} g_\tau$$

ADAGRAD is easier to use than momentum because the value is set automatically and we don't have to set additional hyper parameters.

ADADELTA, suggested by Zeiler (<http://arxiv.org/pdf/1212.5701.pdf>), is known to be an even better optimizer. This is an algorithm-based optimizer and cannot be written in a single equation. Here is a description of ADADELTA:

- Initialization:
 - Initialize accumulation variables:

$$E[g^2]_0 = 0$$

And:

$$E[\Delta \theta^2]_0 = 0$$

- Iteration $\tau = 0, 1, 2, \dots, T$:

- Compute:

$$g_\tau = \frac{\partial E}{\partial \theta^{(\tau)}}$$

- Accumulate gradient:

$$E[g^2]_\tau = \rho E[g^2]_{\tau-1} + (1-\rho) g_\tau^2$$

- Compute update:

$$\Delta\theta^{(\tau)} = -\frac{\sqrt{E[\Delta\theta^2]_{\tau-1} + \varepsilon}}{\sqrt{E[g^2]_\tau + \varepsilon}} g_\tau$$

- Accumulate updates:

$$E[\Delta\theta^2]_\tau = \rho E[\Delta\theta^2]_{\tau-1} + (1-\rho) (\Delta\theta^{(\tau)})^2$$

- Apply update:

$$\theta^{(\tau+1)} = \theta^{(\tau)} + \Delta\theta^{(\tau)}$$

Here, ρ and ε are the hyper parameters. You may think ADADELTA is rather complicated but you don't need to worry about this complexity when implementing with DL4J.

There are still other optimizers supported in DL4J such as **RMSProp**, **RMSProp + momentum**, and **Nesterov's Accelerated Gradient Descent**. However, we won't dig into them because, practically, momentum, ADAGRAD, and ADADELTA are enough to optimize the learning rate.

Summary

In this chapter, you learned how to implement deep learning models with the libraries ND4J and DL4J. Both support GPU computing and both give us the ability to implement them without any difficulties. ND4J is a library for scientific computing and enables vectorization, which makes it easier to implement a calculation among arrays because we don't need to write iterations within them. Since machine learning and deep learning algorithms have many equations with vector calculations, such as inner products and element-wise multiplication, ND4J also helps implement them.

DL4J is a library for deep learning, and by following some examples with the library, you saw that we can easily build, train, and evaluate various types of deep learning models. Additionally, while building the model, you learned why regularization is necessary to get better results. You also got to know some optimizers of the learning rate: momentum, ADAGRAD, and ADADELTA. All of these can be implemented easily with DL4J.

You gained knowledge of the core theories and implementations of deep learning algorithms and you now know how to implement them with little difficulty. We can say that we've completed the theoretical part of this book. Therefore, in the next chapter, we'll look at how deep learning algorithms are adapted to practical applications first and then look into other possible fields and ideas to apply the algorithms.

6

Approaches to Practical Applications – Recurrent Neural Networks and More

In the previous chapters, you learned quite a lot about deep learning. You should now understand the fundamentals of the concepts, theories, and implementations of deep neural networks. You also learned that you can experiment with deep learning algorithms on various data relatively easily by utilizing a deep learning library. The next step is to examine how deep learning can be applied to a broad range of other fields and how to utilize it for practical applications.

Therefore, in this chapter, we'll first see how deep learning is actually applied. Here, you will see that the actual cases where deep learning is utilized are still very few. But why aren't there many cases even though it is such an innovative method? What is the problem? Later on, we'll think about the reasons. Furthermore, going forward we will also consider which fields we can apply deep learning to and will have the chance to apply deep learning and all the related areas of artificial intelligence.

The topics covered in this chapter include:

- Image recognition, natural language processing, and the neural networks models and algorithms related to them
- The difficulties of turning deep learning models into practical applications
- The possible fields where deep learning can be applied, and ideas on how to approach these fields

We'll explore the potential of this big AI boom, which will lead to ideas and hints that you can utilize in deep learning for your research, business, and many sorts of activities.

Fields where deep learning is active

We often hear that research for deep learning has always been ongoing and that's a fact. Many corporations, especially large tech companies such as Google, Facebook, Microsoft, and IBM, invest huge amounts of money into the research of deep learning, and we frequently hear news that some corporation has bought these research groups. However, as we look through, deep learning itself has various types of algorithms, and fields where these algorithms can be applied. Even so, it is a fact that is it not widely known which fields deep learning is utilized in or can be used in. Since the word "AI" is so broadly used, people can't properly understand which technology is used for which product. Hence, in this section, we will go through the fields where people have been trying to adopt deep learning actively for practical applications.

Image recognition

The field in which deep learning is most frequently incorporated is image recognition. It was Prof. Hinton and his team's invention that led to the term "deep learning." Their algorithm recorded the lowest error rates ever in an image recognition competition. The continuous research done to improve the algorithm led to even better results. Now, image recognition utilizing deep learning has gradually been adopted not only for studies, but also for practical applications and products. For example, Google utilizes deep learning to auto-generate thumbnails for YouTube or auto-tag and search photos in Google Photos. Like these popular products, deep learning is mainly applied to image tagging or categorizing and, for example, in the field of robotics, it is used for robots to specify things around them.

The reason why we can support these products and this industry is because deep learning is more suited to image processing, and this is because it can achieve higher precision rates than applications in any other field. Only because the precision and recall rate of image recognition is so high does it mean this industry has broad potential. An error rate of MNIST image classification is recorded at 0.21 percent with a deep learning algorithm (<http://cs.nyu.edu/~wanli/dropc/>), and this rate can be no lower than the record for a human (<http://arxiv.org/pdf/0710.2231v1.pdf>). In other words, if you narrow it down to just image recognition, it's nothing more than the fact that a machine may overcome a human. Why does only image recognition get such high precision while other fields need far more improvement in their methods?

One of the reasons is that the structure of feature extractions in deep learning is well suited for image data. In deep neural networks, many layers are stacked and features are extracted from training data step by step at each layer. Also, it can be said that image data is featured as a layered structure. When you look at images, you will unconsciously catch brief features first and then look into a more detailed feature. Therefore, the inherent property of deep learning feature extraction is similar to how an image is perceived and hence we can get an accurate realization of the features. Although image recognition with deep learning still needs more improvements, especially of how machines can understand images and their contents, obtaining high precision by just adopting deep learning to sample image data without preprocessing obviously means that deep learning and image data are a good match.

The other reason is that people have been working to improve algorithms slowly but steadily. For example, in deep learning algorithms, CNN, which can get the best precision for image recognition, has been improved every time it faces difficulties/tasks. Local receptive fields substituted with kernels of convolutional layers were introduced to avoid networks becoming too dense. Also, downsampling methods such as max-pooling were invented to avoid the overreaction of networks towards a gap of image location. This was originally generated from a trial and error process on how to recognize handwritten letters written in a certain frame such as a postal code. As such, there are many cases where a new approach is sought to adapt neural networks algorithms for practical applications. A complicated model, CNN is also built based on these accumulated yet steady improvements. While we don't need feature engineering with deep learning, we still need to consider an appropriate approach to solve specific problems, that is, we can't build omnipotent models, and this is known as the **No Free Lunch Theorem (NFLT)** for optimization.

In the image recognition field, the classification accuracy that can be achieved by deep learning is extremely high, and it is actually beginning to be used for practical applications. However, there should be more fields where deep learning can be applied. Images have a close connection to many industries. In the future, there will be many cases and many more industries that utilize deep learning. In this book, let's think about what industries we can apply image recognition to, considering the emergence of deep learning in the next sections.

Natural language processing

The second most active field, after image recognition, where the research of deep learning has progressed is **natural language processing (NLP)**. The research in this field might become the most active going forward. With regard to image recognition, the prediction precision we could obtain almost reaches the ceiling, as it can perform even better classification than a human could. On the other hand, in NLP, it is true that the performance of a model gets a lot better thanks to deep learning, but it is also a fact that there are many tasks that still need to be solved.

For some products and practical applications, deep learning has already been applied. For example, NLP based on deep learning is applied to Google's voice search or voice recognition and Google translation. Also, IBM Watson, the cognitive computing system that understands and learns natural language and supports human decision-making, extracts keywords and entities from tons of documents, and has functions to label documents. And these functions are open to the public as the Watson API and anyone can utilize it without constraints.

As you can see from the preceding examples, NLP itself has a broad and varied range of types. In terms of fundamental techniques, we have the classification of sentence contents, the classification of words, and the specification of word meanings. Furthermore, languages such as Chinese or Japanese that don't leave a space between words require morphological analysis, which is also another technique available in NLP.

NLP contains a lot of things that need to be researched, therefore it needs to clarify what its purpose is, what its problems are, and how these problems can be solved. What model is the best to use and how to get good precision properly are topics that should be examined cautiously. As for image recognition, the CNN method was invented by solving tasks that were faced. Now, let's consider what approach we can think of and what the difficulties will be respectively for neural networks and NLP. Understanding past trial and error processes will be useful for research and applications going forward.

Feed-forward neural networks for NLP

The fundamental problem of NLP is "to predict the next word given a specific word or words". The problem is too simple, however; if you try to solve it with neural networks, then you will soon face several difficulties because documents or sentences as sample data using NLP have the following features:

- The length of each sentence is not fixed but variable, and the number of words is astronomical

- There can be unforeseen problems such as misspelled words, acronyms, and so on
- Sentences are sequential data, and so contain temporal information

Why can these features pose a problem? Remember the model structure of general neural networks. For training and testing with neural networks, the number of neurons in each layer including the input layer needs to be fixed in advance and the networks need to be the same size for all the sample data. In the meantime, the length of the input data is not fixed and can vary a lot. This means that sample data cannot be applied to the model, at least as it is. Classification or generation by neural networks cannot be done without adding/amending something to this data.

We have to fix the length of input data, and one approach to handle this issue is a method that divides a sentence into a chunk of certain words from the beginning in order. This method is called **N-gram**. Here, N represents the size of each item, and an **N-gram** of size 1 is called a **unigram**, size 2 is a **bigram**, and size 3 is a **trigram**. When the size is larger, then it is simply called with the value of N , such as *four-gram*, *five-gram*, and so on.

Let's look at how N-gram works with NLP. The goal here is to calculate the probability of a word w given some history h ; $P(w|h)$. We'll represent a sequence of N words as w_1^n . Then, the probability we want to compute is $P(w_1^n)$, and by applying the chain rule of probability to this term, we get:

$$\begin{aligned} P(w_1^n) &= P(w_1)P(w_2 | w_1)P(w_3 | w_1^2)\dots P(w_n | w_1^{n-1}) \\ &= \prod_{k=1}^n P(w_k | w_1^{k-1}) \end{aligned}$$

It might look at first glance like these conditional probabilities help us, but actually they don't because we have no way of calculating the exact probability of a word following a long sequence of preceding words, $P(w_n | w_{n-1})$. Since the structure of a sentence is very flexible, we can't simply utilize sample documents and a corpus to estimate the probability. This is where N-gram works. Actually, we have two approaches to solve this problem: the original N-gram model and the neural networks model based on N-gram. We'll look at the first one to fully understand how the fields of NLP have developed before we dig into neural networks.

With N-gram, we don't compute the probability of a word given its whole history, but approximate the history with the last N words. For example, the bigram model approximates the probability of a word just by the conditional probability of the preceding word, $P(w_n | w_{n-1})$, and so follows the equation:

$$\begin{aligned} P(w_1^n) &= \prod_{k=1}^n P(w_k | w_1^{k-1}) \\ &\approx \prod_{k=1}^n P(w_k | w_{k-1}) \end{aligned}$$

Similarly, we can generalize and expand the equation for N-gram. In this case, the probability of a word can be represented as follows:

$$P(w_n | w_1^{n-1}) \approx P(w_n | w_{n-N+1}^{n-1})$$

We get the following equation:

$$P(w_1^n) \approx \prod_{k=1}^n P(w_k | w_{k-N+1}^{k-1})$$

Just bear in mind that these approximations with N-gram are based on the probabilistic model called the **Markov model**, where the probability of a word depends only on the previous word.

Now what we need to do is estimate these N-gram probabilities, but how do we estimate them? One simple way of doing this is called the **maximum likelihood estimation (MLE)**. This method estimates the probabilities by taking counts from a corpus and normalizing them. So when we think of a bigram as an example, we get:

$$P(w_n | w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)}$$

In the preceding formula, $C(\cdot)$ denotes the counts of a word or a sequence of words. Since the denominator, that is, the sum of all bigram counts starting with a word, w_{n-1} is equal to the unigram count of w_{n-1} , the preceding equation can be described as follows:

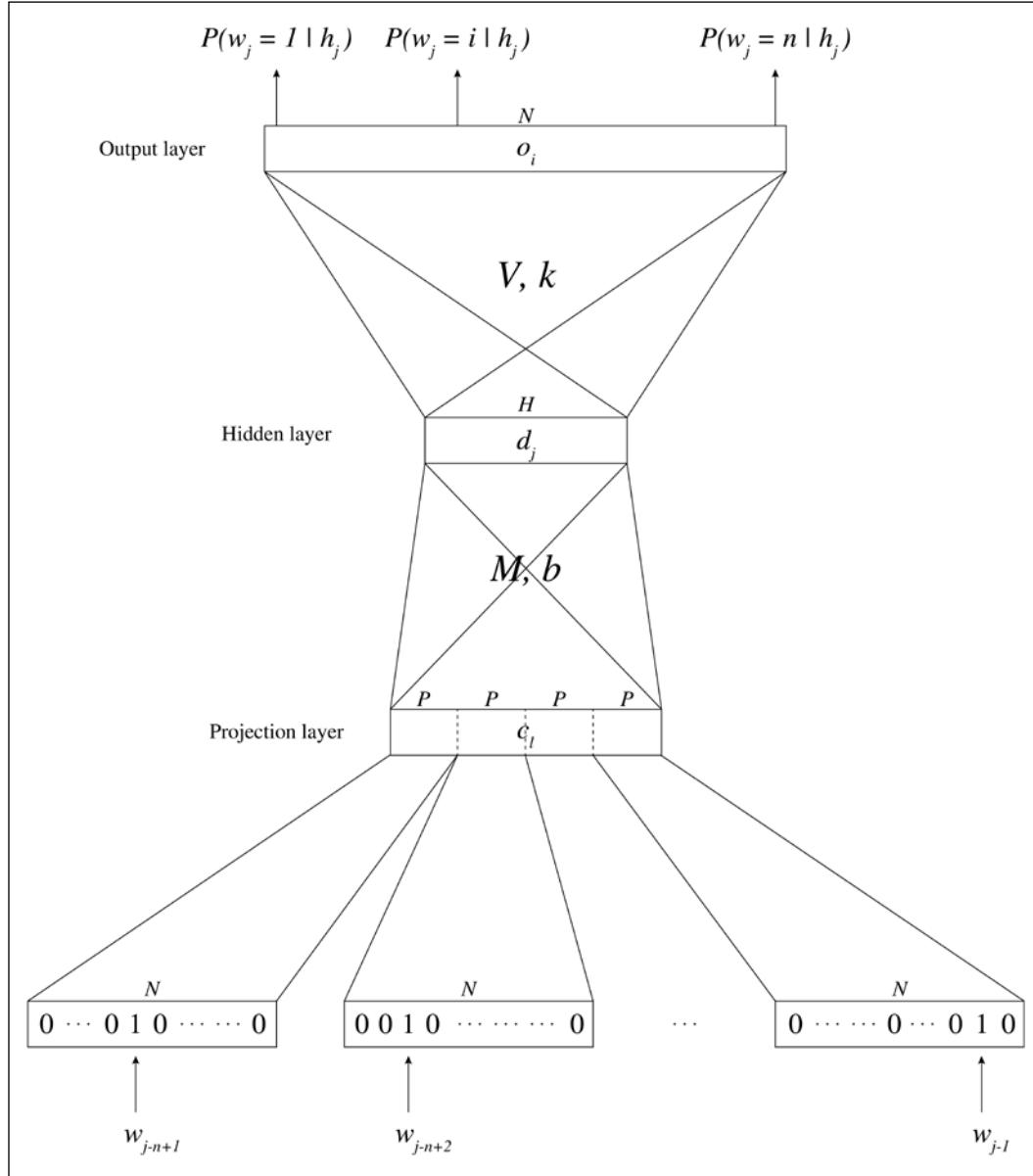
$$P(w_n | w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})}$$

Accordingly, we can generalize MLE for N-gram as well:

$$P(w_n | w_{n-N+1}^{n-1}) = \frac{C(w_{n-N+1}^{n-1}w_n)}{C(w_{n-N+1}^{n-1})}$$

Although this is a fundamental approach of NLP with N-gram, we now know how to compute N-gram probabilities.

In contrast to this approach, the neural network models predict the conditional probability of a word w_j given a specific history, h_j ; $P(w_j = i | h_j)$. One of the models of NLP is called the **Neural Network Language Model (NLMM)** (<http://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf>), and it can be illustrated as follows:



Here, N is the size of the vocabulary, and each word in the vocabulary is an N -dimensional vector where only the index of the word is set to 1 and all the other indices to 0. This method of representation is called *1-of- N coding*. The inputs of NLM are the indices of the $n-1$ previous words $h_j = w_{j-n+1}^{j-1}$ (so they are *n -grams*). Since the size N is typically within the range of 5,000 to 200,000, input vectors of NLM are very sparse. Then, each word is mapped to the projection layer, for continuous space representation. This linear projection (activation) from a discrete to a continuous space is basically a look-up table with $N \times P$ entries, where P denotes the feature dimension. The projection matrix is shared for the different word positions in the context, and activates the word vectors to projection layer units c_l with $l=1,\dots,(n-1).P$. After the projection comes the hidden layer. Since the projection layer is in the continuous space, the model structure is just the same as the other neural networks from here. So, the activation can be represented as follows:

$$d_j = h \left(\sum_{l=1}^{(n-1).P} m_{jl} c_l + b_j \right)$$

Here, $h(\cdot)$ denotes the activation function, m_{ji} the weights between the projection layer and the hidden layer, and b_j the biases of the hidden layer. Accordingly, we can get the output units as follows:

$$o_i = \sum_j v_{ij} d_j + k_i$$

Here, v_{ij} denotes the weights between the hidden layer and the output layer, and k_i denotes the biases of the output layer. The probability of a word i given a specific history h_j can then be calculated using the softmax function:

$$P(w_j = i | h_j) = \frac{\exp(o_i)}{\sum_{l=1}^N \exp(o_l)}$$

As you can see, in NNLM, the model predicts the probability of all the words at the same time. Since the model is now described with the standard neural network, we can train the model using the standard backpropagation algorithm.

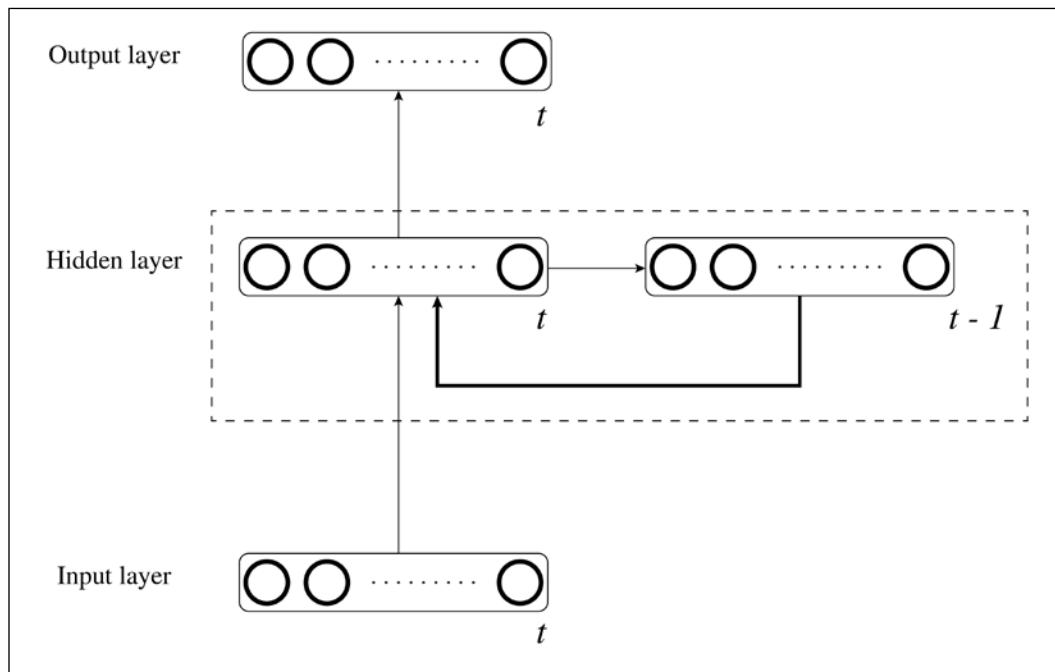
NNLM is one approach of NLP using neural networks with N-gram. Though NNLM solves the problem of how to fix the number of inputs, the best N can only be found by trial and error, and it is the most difficult part of the whole model building process. In addition, we have to make sure that we don't put too much weight on the temporal information of the inputs here.

Deep learning for NLP

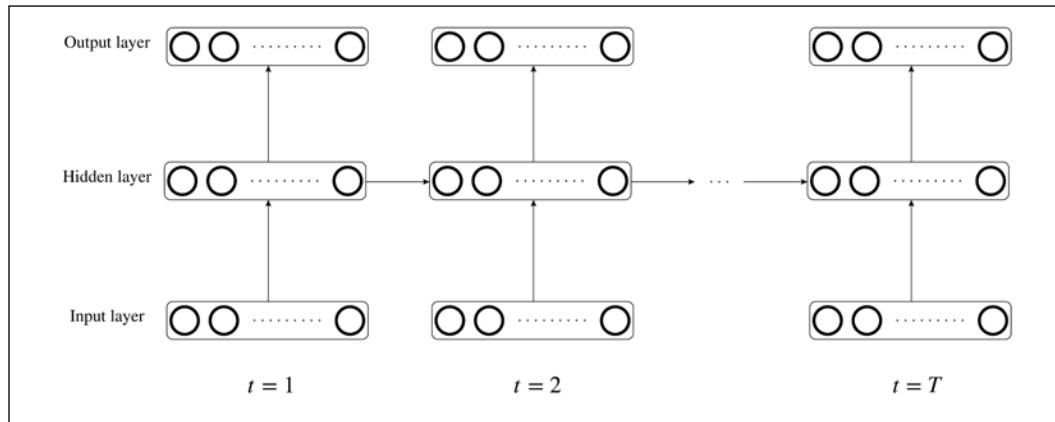
Neural networks with N-gram may work with certain cases, but contain some issues, such as what n-grams would return the best results, and do n-grams, the inputs of the model, still have a context? These are the problems not only of NLP, but of all the other fields that have time sequential data such as precipitation, stock prices, yearly crop of potatoes, movies, and so on. Since we have such a massive amount of this data in the real world, we can't ignore the potential issue. But then, how would it be possible to let neural networks be trained with time sequential data?

Recurrent neural networks

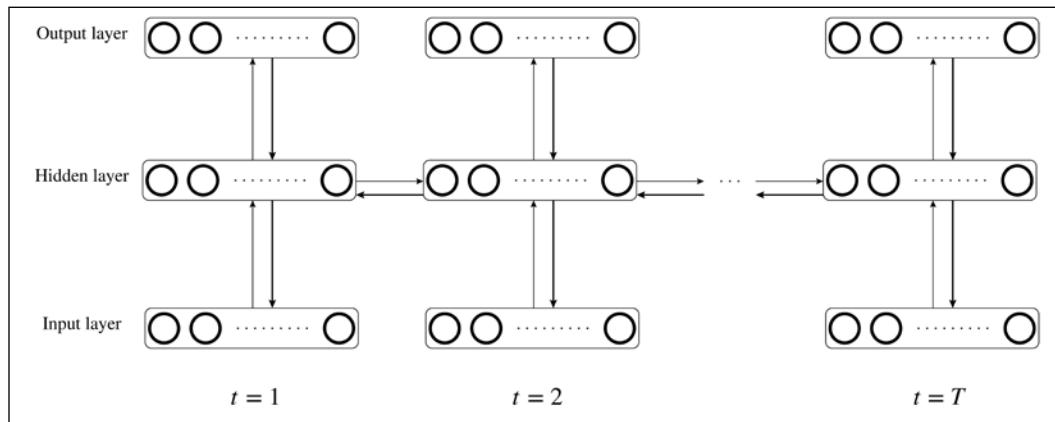
One of the neural network models that is able to preserve the context of data within networks is **recurrent neural network (RNN)**, the model that actively studies the evolution of deep learning algorithms. The following is a very simple graphical model of RNN:



The difference between standard neural networks is that RNN has connections between hidden layers with respect to time. The input at time t is activated in the hidden layer at time t , preserved in the hidden layer, and then propagated to the hidden layer at time $t+1$ with the input at time $t+1$. This enables the networks to contain the states of past data and reflect them. You might think that RNN is rather a dynamic model, but if you unfold the model at each time step, you can see that RNN is a static model:

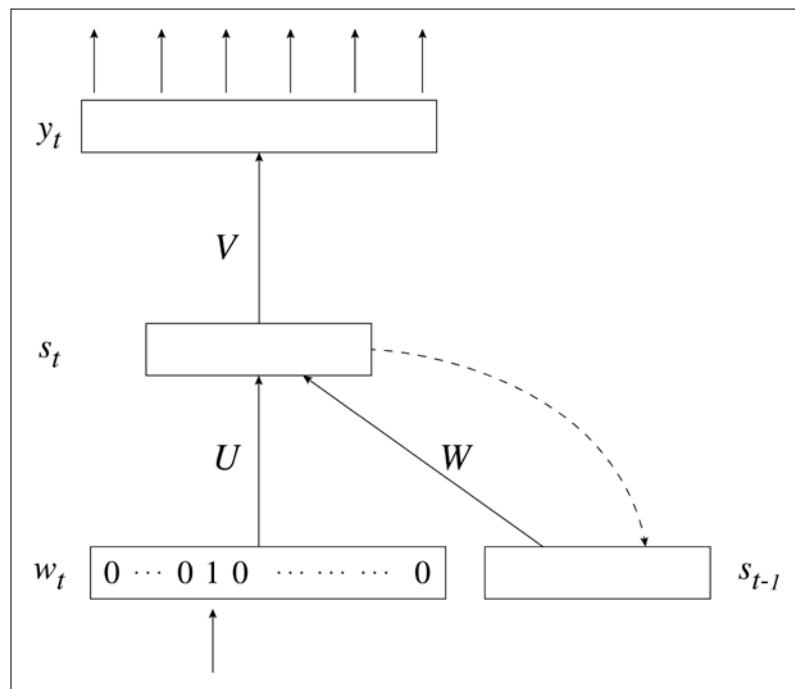


Since the model structure at each time step is the same as in general neural networks, you can train this model using the backpropagation algorithm. However, you need to consider time relevance when training, and there is a technique called **Backpropagation through Time (BPTT)** to handle this. In BPTT, the errors and gradients of the parameter are backpropagated to the layers of the past:



Thus, RNN can preserve contexts within the model. Theoretically, the network at each time step should consider the whole sequence up to then, but practically, time windows with a certain length are often applied to the model to make the calculation less complicated or to prevent the vanishing gradient problem and the exploding gradient problem. BPTT has enabled training among layers and this is why RNN is often considered to be one of the deep neural networks. We also have algorithms of deep RNN such as stacked RNN where hidden layers are stacked.

RNN has been adapted for NLP, and is actually one of the most successful models in this field. The original model optimized for NLP is called the **recurrent neural network language model (RNNLM)**, introduced by Mikolov et al. (http://www.fit.vutbr.cz/research/groups/speech/publi/2010/mikolov_interspeech2010_IS100722.pdf). The model architecture can be illustrated as follows:

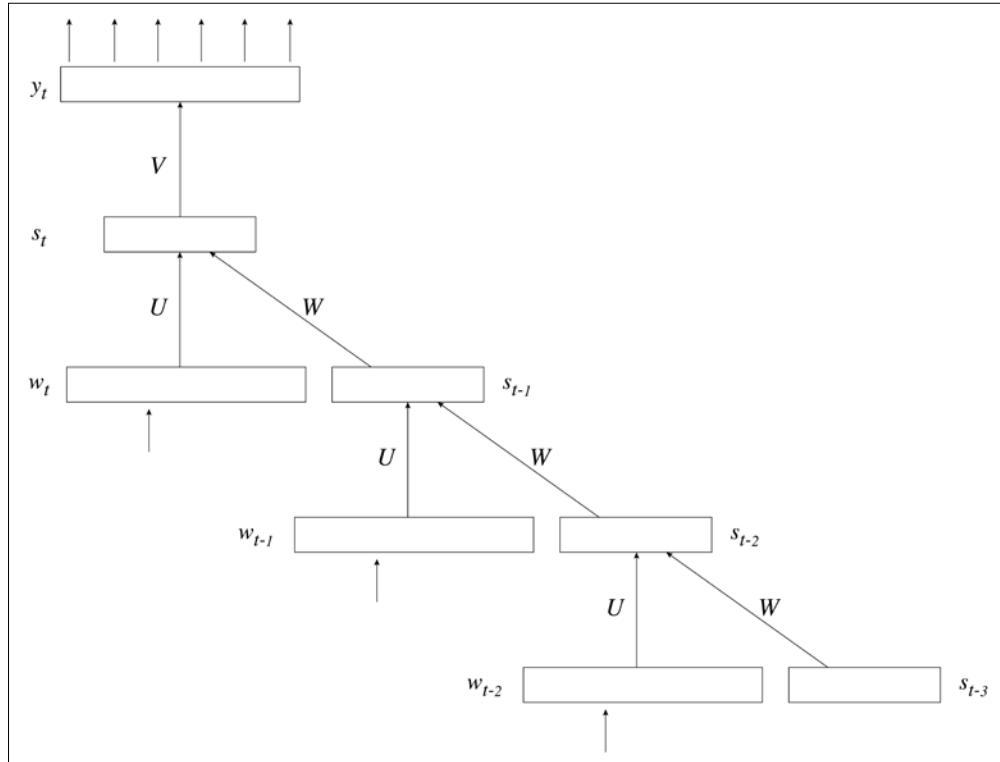


The network has three layers: an input layer x , a hidden layer s , and an output layer y . The hidden layer is also often called the context layer or the state layer. The value of each layer with respect to the time t can be represented as follows:

$$\begin{aligned}x_t &= w_t + s_{t-1} \\s_t &= f(Uw_t + Ws_{t-1}) \\y_t &= g(Vs_t)\end{aligned}$$

Here, $f(\cdot)$ denotes the sigmoid function, and $g(\cdot)$ the softmax function. Since the input layer contains the state layer at time $t-1$, it can reflect the whole context to the network. The model architecture implies that RNNLM can look up much broader contexts than feed-forward NNLM, in which the length of the context is constrained to N (-gram).

The whole time and the entire context should be considered while training RNN, but as mentioned previously, we often truncate the time length because BPTT requires a lot of calculations and often causes the gradient vanishing/exploding problem when learning long-term dependencies, hence the algorithm is often called **truncated BPTT**. If we unfold RNNLM with respect to time, the model can be illustrated as follows (in the figure, the unfolded time $\tau = 3$):



Here d_t is the label vector of the output. Then, the error vector of the output can be represented as follows:

$$\delta_t^{out} = d_t - y_t$$

We get the following equation:

$$\delta_t^{hidden} = d \left(\left(\delta_t^{out} \right)^T V, t \right)$$

Here T is the unfolding time:

$$d(x, t) = x s_t (1 - s_t)$$

The preceding image is the derivative of the activation function of the hidden layer. Since we use the sigmoid function here, we get the preceding equation. Then, we can get the error of the past as follows:

$$\delta_{t-\tau-1}^{hidden} = d \left(\left(\delta_{t-\tau}^{out} \right)^T V, t - \tau - 1 \right)$$

With these equations, we can now update the weight matrices of the model:

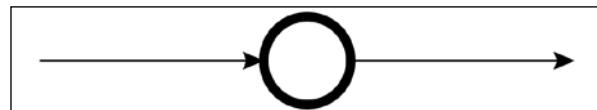
$$\begin{aligned} V_{t+1} &= V_t + s_t \left(\delta_t^{out} \right)^T \alpha \\ U_{t+1} &= U_t + \sum_{\tau=0}^T w_{t-\tau} \left(\delta_{t-\tau}^{hidden} \right)^T \alpha \\ W_{t+1} &= W_t + \sum_{\tau=0}^T w_{t-\tau-1} \left(\delta_{t-\tau}^{hidden} \right)^T \alpha \end{aligned}$$

Here, α is the learning rate. What is interesting in RNNLM is that each vector in the matrix shows the difference between words after training. This is because U is the matrix that maps each word to a latent space, so after the training, mapped word vectors contain the meaning of the words. For example, the vector calculation of "king" - "man" + "woman" would return "queen". DL4J supports RNN, so you can easily implement this model with the library.

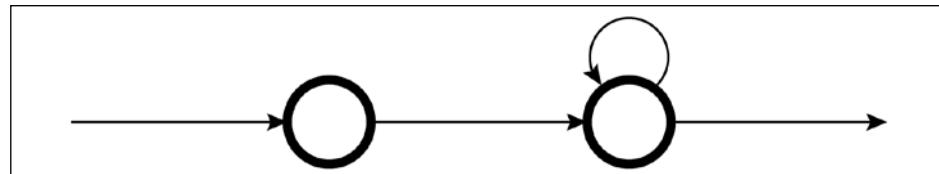
Long short term memory networks

Training with the standard RNN requires the truncated BPTT. You might well doubt then that BPTT can really train the model enough to reflect the whole context, and this is very true. This is why a special kind of RNN, the **long short term memory (LSTM)** network, was introduced to solve the long-term dependency problem. LSTM is rather intimidating, but let's briefly explore the concept of LSTM.

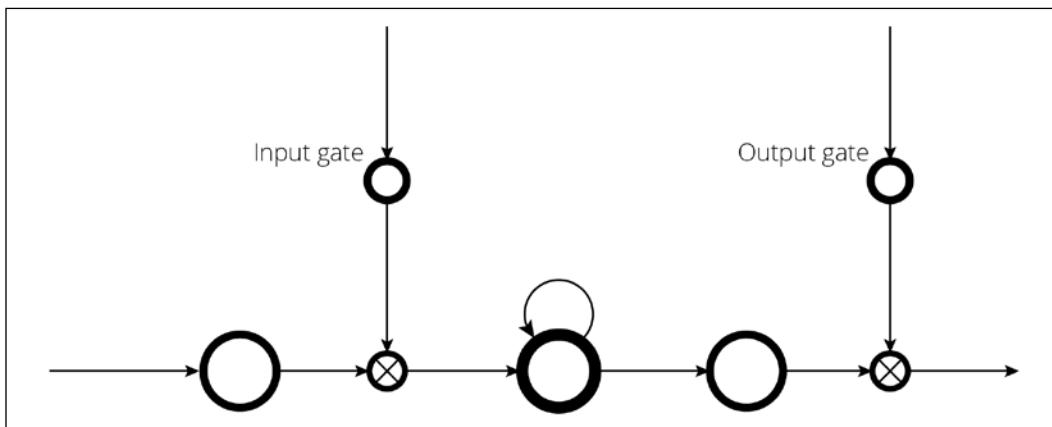
To begin with, we have to think about how we can store and tell past information in the network. While the gradient exploding problem can be mitigated simply by setting a ceiling to the connection, the gradient vanishing problem still needs to be deeply considered. One possible approach is to introduce a unit that permanently preserves the value of its inputs and its gradient. So, when you look at a unit in the hidden layer of standard neural networks, it is simply described as follows:



There's nothing special here. Then, by adding a unit below to the network, the network can now memorize the past information within the neuron. The neuron added here has linear activation and its value is often set to 1. This neuron, or cell, is called **constant error carousel (CEC)** because the error stays in the neuron like a carousel and won't vanish. CEC works as a storage cell and stores past inputs. This solves the gradient vanishing problem, but raises another problem. Since all data propagated through is stocked in the neuron, it probably stores noise data as well:

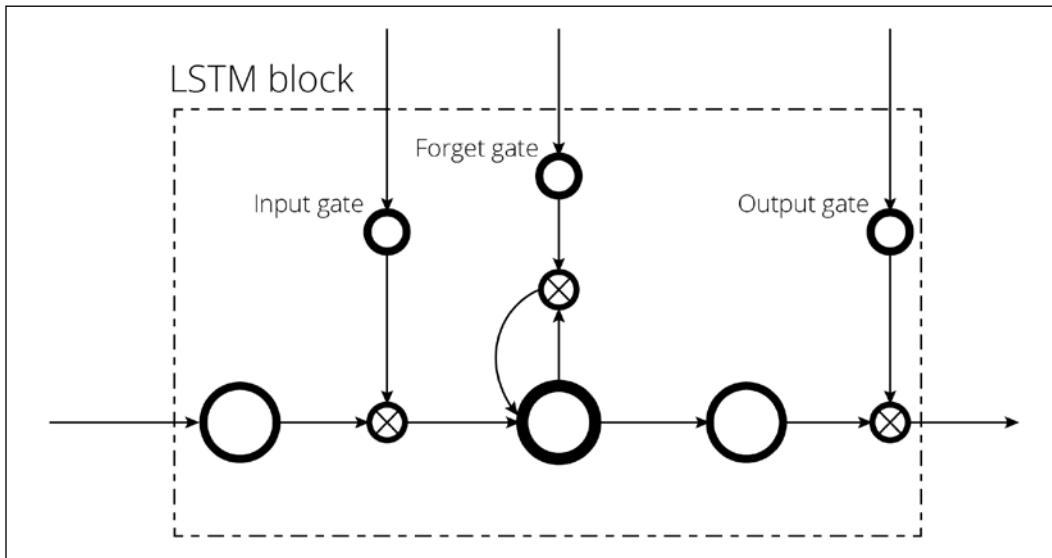


This problem can be broken down into two problems: *input weight conflicts* and *output weight conflicts*. The key idea of input weight conflicts is to keep certain information within the network until it's necessary; the neuron is to be activated only when the relevant information comes, but is not to be activated otherwise. Similarly, output weight conflicts can occur in all types of neural networks; the value of neurons is to be propagated only when necessary, and not to be propagated otherwise. We can't solve these problems as long as the connection between neurons is represented with the weight of the network. Therefore, another method or technique of representation is required that controls the propagation of inputs and outputs. But how do we do this? The answer is putting units that act like "gates" before and behind the CEC, and these are called **input gate** and **output gate**, respectively. The graphical model of the gate can be described as follows:

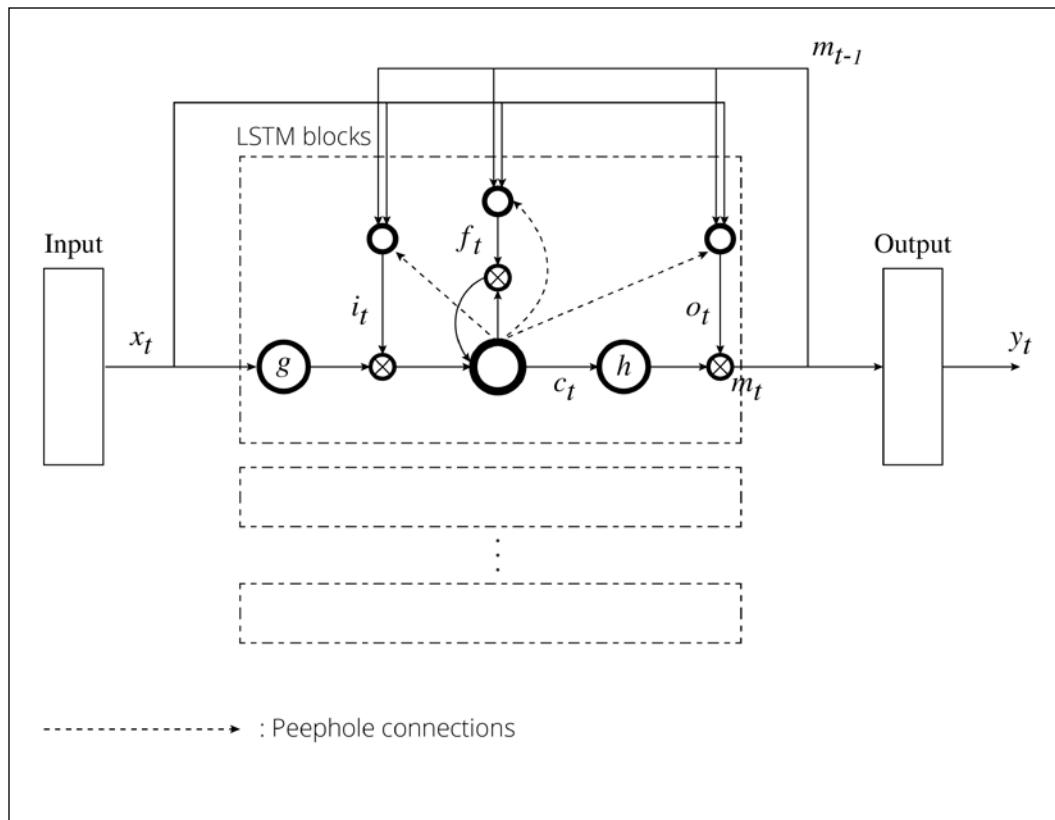


Ideally, the gate should return the discrete value of 0 or 1 corresponding to the input, 0 when the gate is closed and 1 when open, because it is a gate, but programmatically, the gate is set to return the value in the range of 0 to 1 so that it can be well trained with BPTT.

It may seem like we can now put and fetch exact information at an exact time, yet another problem still remains. With just two gates, the input gate and output gate, memories stored in the CEC can't be refreshed easily in a few steps. Therefore, we need an additional gate that dynamically changes the value of the CEC. To do this, we add a **forget gate** to the architecture to control when the memory should be erased. The value preserved in the CEC is overridden with a new memory when the value of the gate takes a 0 or close to it. With these three gates, a unit can now memorize information or contexts of the past, and so it is called an **LSTM block** or an **LSTM memory block** because it is more of a block than a single neuron. The following is a figure that represents an LSTM block:



The standard LSTM structure was fully explained previously, but there's a technique to get better performance from it, which we'll explain now. Each gate receives connections from the input units and the outputs of all the units in LSTM, but there is no direct connection from the CEC. This means we can't see the true hidden state of the network because the output of a block depends so much on the output gate; as long as the output gate is closed, none of the gates can access the CEC and it is devoid of essential information, which may debase the performance of LSTM. One simple yet effective solution is to add connections from the CEC to the gates in a block. These are called **peephole connections**, and act as standard weighted connections except that no errors are backpropagated from the gates through the peephole connections. The peephole connections let all gates assume the hidden state even when the output gate is closed. You've learned a lot of terms now, but as a result, the basic architecture of the whole connection can be described as follows:



For simplicity, a single LSTM block is described in the figure. You might be daunted because the preceding model is very intricate. However, when you look at the model step by step, you can understand how an LSTM network has figured out how to overcome difficulties in NLP. Given an input sequence $x = (x_1, \dots, x_T)$, each network unit can be calculated as follows:

$$\begin{aligned} i_t &= \sigma(W_{ix}x_t + W_{im}m_{t-1} + W_{ic}c_{t-1} + b_i) \\ f_t &= \sigma(W_{fx}x_t + W_{fm}m_{t-1} + W_{fc}c_{t-1} + b_f) \\ c_t &= f_t \odot c_{t-1} + i_t \odot g(W_{cx}x_t + W_{cm}m_{t-1} + b_c) \\ o_t &= \sigma(W_{ox}x_t + W_{om}m_{t-1} + W_{oc}c_t + b_o) \\ m_t &= o_t \odot h(c_t) \\ y_t &= s(W_{ym}m_t + b_y) \end{aligned}$$

In the preceding formulas, W_{ix} is the matrix of weights from the input gate to the input, W_{fx} is the one from the forget gate to the input, and W_{ox} is the one from the output gate to the input. W_{cs} is the weight matrix from the cell to the input, W_{cm} is the one from the cell to the LSTM output, and W_{ym} is the one from the output to the LSTM output. W_{ic} , W_{fc} , and W_{oc} are diagonal weight matrices for peephole connections. The b terms denote the bias vectors, b_i is the input gate bias vector, b_f is the forget gate bias vector, b_o is the output gate bias vector, b_c is the CEC cell bias vector, and b_y is the output bias vector. Here, g and h are activation functions of the cell input and cell output. σ denotes the sigmoid function, and $s(\cdot)$ the softmax function. \odot is the element-wise product of the vectors.

We won't follow the further math equations in this book because they become too complicated just by applying BPTT, but you can try LSTM with DL4J as well as RNN. As CNN was developed within the field of image recognition, RNN and LSTM have been developed to resolve the issues of NLP that arise one by one. While both algorithms are just one approach to get a better performance using NLP and still need to be improved, since we are living beings that communicate using languages, the development of NLP will certainly lead to technological innovations. For applications of LSTM, you can reference *Sequence to Sequence Learning with Neural Networks* (Sutskever et al., <http://arxiv.org/pdf/1409.3215v3.pdf>), and for more recent algorithms, you can reference *Grid Long Short-Term Memory* (Kalchbrenner et al., <http://arxiv.org/pdf/1507.01526v1.pdf>) and *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention* (Xu et al., <http://arxiv.org/pdf/1502.03044v2.pdf>).

The difficulties of deep learning

Deep learning has already got higher precision than humans in the image recognition field and has been applied to quite a lot of practical applications. Similarly, in the NLP field, many models have been researched. Then, how much deep learning is utilized in other fields? Surprisingly, there are still few fields where deep learning is successfully utilized. This is because deep learning is indeed innovative compared to past algorithms and definitely lets us take a big step towards materializing AI; however, it has some problems when used for practical applications.

The first problem is that there are too many model parameters in deep learning algorithms. We didn't look in detail when you learned about the theory and implementation of algorithms, but actually deep neural networks have many hyper parameters that need to be decided compared to the past neural networks or other machine learning algorithms. This means we have to go through more trial and error to get high precision. Combinations of parameters that define a structure of neural networks, such as how many hidden layers are to be set or how many units each hidden layer should have, need lots of experiments. Also, the parameters for training and test configurations such as the learning rate need to be determined. Furthermore, peculiar parameters for each algorithm such as the corruption level in SDA and the size of kernels in CNN need additional trial and error. Thus, the great performance that deep learning provides is supported by steady parameter-tuning. However, people only look at one side of deep learning – that it can get great precision – and they tend to forget the hard process required to reach that point. Deep learning is not magic.

In addition, deep learning often fails to train and classify data from simple problems. The shape of deep neural networks is so deep and complicated that the weights can't be well optimized. In terms of optimization, data quantities are also important. This means that deep neural networks require a significant amount of time for each training. To sum up, deep learning shows its worth when:

- It solves complicated and hard problems when people have no idea what feature they can be classified as
- There is sufficient training data to properly optimize deep neural networks

Compared to applications that constantly update a model using continuously updated data, once a model is built using a large-scale dataset that doesn't change drastically, applications that use the model universally are rather well suited for deep learning.

Therefore, when you look at business fields, you can say that there are more cases where the existing machine learning can get better results than using deep learning. For example, let's assume we would like to recommend appropriate products to users in an EC. In this EC, many users buy a lot of products daily, so purchase data is largely updated daily. In this case, do you use deep learning to get high-precision classification and recommendations to increase the conversion rates of users' purchases using this data? Probably not, because using the existing machine learning algorithms such as Naive Bayes, collaborative filtering, SVM, and so on, we can get sufficient precision from a practical perspective and can update the model and calculate quicker, which is usually more appreciated. This is why deep learning is not applied much in business fields. Of course, getting higher precision is better in any field, but in reality, higher precision and the necessary calculation time are in a trade-off relationship. Although deep learning is significant in the research field, it has many hurdles yet to clear considering practical applications.

Besides, deep learning algorithms are not perfect, and they still need many improvements to their model itself. For example, RNN, as mentioned earlier, can only satisfy either how past information can be reflected to a network or how precision can be obtained, although it's contrived with techniques such as LSTM. Also, deep learning is still far from the true AI, although it's definitely a great technique compared to the past algorithms. Research on algorithms is progressing actively, but in the meantime, we need one more breakthrough to spread out and infiltrate deep learning into broader society. Maybe this is not just the problem of a model. Deep learning is suddenly booming because it is reinforced by huge developments in hardware and software. Deep learning is closely related to development of the surrounding technology.

As mentioned earlier, there are still many hurdles to clear before deep learning can be applied more practically in the real world, but this is not impossible to achieve. It isn't possible to suddenly invent AI to achieve technological singularity, but there are some fields and methods where deep learning can be applied right away. In the next section, we'll think about what kinds of industries deep learning can be utilized in. Hopefully, it will sow the seeds for new ideas in your business or research fields.

The approaches to maximizing deep learning possibilities and abilities

There are several approaches to how we can apply deep learning to various industries. While it is true that an approach could be different depending on the task or purpose, we can briefly categorize the approaches in the following three ways:

- **Field-oriented approach:** This utilizes deep learning algorithms or models that are already thoroughly researched and can lead to great performance
- **Breakdown-oriented approach:** This replaces the problems to be solved that deep learning can apparently be applied to with a different problem where deep learning can be well adopted
- **Output-oriented approach:** This explores new ways of how we express the output with deep learning

These approaches are all explained in detail in the following subsections. Each approach is divided into its suitable industries or areas where it is not suitable, but any of them could be a big hint for your activities going forward. There are still very few use cases of deep learning and bias against fields of use, but this means there should be many chances to create innovative and new things. Start-ups that utilize deep learning have been emerging recently and some of them have already achieved success to some extent. You can have a significant impact on the world depending on your ideas.

Field-oriented approach

This approach doesn't require new techniques or algorithms. There are obviously fields that are well suited to the current deep learning techniques, and the concept here is to dive into these fields. As explained previously, since deep learning algorithms that have been practically studied and developed are mostly in image recognition and NLP, we'll explore some fields that can work in great harmony with them.

Medicine

Medical fields should be developed by deep learning. Tumors or cancers are detected on scanned images. This means nothing more than being able to utilize one of the strongest features of deep learning – the technique of image recognition. It is possible to dramatically increase precision using deep learning to help with the early detection of an illness and identifying the kind of illness. Since CNN can be applied to 3D images, 3D scanned images should be able to be analyzed relatively easily. By adopting deep learning more in the current medical field, deep learning should greatly contribute.

We can also say that deep learning can be significantly useful for the medical field in the future. The medical field has been under strict regulations; however, there is a movement progressing to ease the regulations in some countries, probably because of the recent development of IT and its potential. Therefore, there will be opportunities in business for the medical field and IT to have a synergistic effect. For example, if telemedicine is more infiltrated, there is the possibility that diagnosing or identifying a disease can be done not only by a scanned image, but also by an image shown in real time on a display. Also, if electronic charts become widespread, it would be easier to analyze medical data using deep learning. This is because medical records are compatible with deep learning as they are a dataset of texts and images. Then the symptoms of unknown diseases can be found.

Automobiles

We can say that the surroundings of running cars are image sequences and text. Other cars and views are images and a road sign has text. This means we can also utilize deep learning techniques here, and it is possible to reduce the risk of accidents by improving driving assistance functions. It can be said that the ultimate type of driving assistance is self-driving cars, which is being tackled mainly by Google and Tesla. An example that is both famous and fascinating was when George Hotz, the first person to hack the iPhone, built a self-driving car in his garage. The appearance of the car was introduced in an article by Bloomberg Business (<http://www.bloomberg.com/features/2015-george-hotz-self-driving-car/>), and the following image was included in the article:



Self-driving cars have been already tested in the U.S., but since other countries have different traffic rules and road conditions, this idea requires further studying and development before self-driving cars are commonly used worldwide. The key to success in this field is in learning and recognizing surrounding cars, people, views, and traffic signs, and properly judging how to process them.

In the meantime, we don't have to just focus on utilizing deep learning techniques for the actual body of a car. Let's assume we could develop a smartphone app that has the same function as we just described, that is, recognizing and classifying surrounding images and text. Then, if you just set up the smartphone in your car, you could utilize it as a car-navigation app. In addition, for example, it could be used as a navigation app for blind people, providing them with good, reliable directions.

Advert technologies

Advert (ad) technologies could expand their coverage with deep learning. When we say ad technologies, this currently means recommendation or ad networks that optimize ad banners or products to be shown. On the other hand, when we say advertising, this doesn't only mean banners or ad networks. There are various kinds of ads in the world depending on the type of media, such as TV ads, radio ads, newspaper ads, posters, flyers, and so on. We have also digital ad campaigns with YouTube, Vine, Facebook, Twitter, Snapchat, and so on. Advertising itself has changed its definition and content, but all ads have one thing in common: they consist of images and/or language. This means they are fields that deep learning is good at. Until now, we could only use user-behavior-based indicators, such as **page view (PV)**, **click through rate (CTR)**, and **conversion rate (CVR)**, to estimate the effect of an ad, but if we apply deep learning technologies, we might be able to analyze the actual content of an ad and autogenerate ads going forward. Especially since movies and videos can only be analyzed as a result of image recognition and NLP, video recognition, not image recognition, will gather momentum besides ad technologies.

Profession or practice

Professions such as doctor, lawyer, patent attorney, and accountant are considered to be roles that deep learning can replace. For example, if NLP's precision and accuracy gets higher, any perusal that requires expertise can be left to a machine. As a machine can cover these time-consuming reading tasks, people can focus more on high-value tasks. In addition, if a machine classifies past judicial cases or medical cases on what disease caused what symptoms and so on, we would be able to build an app like Apple's Siri that answers simple questions that usually require professional knowledge. Then the machine could handle these professional cases to some extent if a doctor or a lawyer is too busy to help in a timely manner.

It's often said that AI takes away a human's job, but personally, this seems incorrect. Rather, a machine takes away menial work, which should support humans. A software engineer who works on AI programming can be described as having a professional job, but this work will also be changed in the future. For example, think about a car-related job, where the current work is building standard automobiles, but in the future, engineers will be in a position just like pit crews for Formula 1 cars.

Sports

Deep learning can certainly contribute to sports as well. In the study field known as sports science, it has become increasingly important to analyze and examine data from sports. As an example, you may know the book or movie *Moneyball*. In this film, they hugely increased the win percentage of the team by adopting a regression model in baseball. Watching sports itself is very exciting, but on the other hand, sport can be seen as a chunk of image sequences and number data. Since deep learning is good at identifying features that humans can't find, it will become easier to find out why certain players get good scores while others don't.

These fields we have mentioned are only a small part of the many fields where deep learning is capable of significantly contributing to development. We have looked into these fields from the perspective of whether a field has images or text, but of course deep learning should also show great performance for simple analysis with general number data. It should be possible to apply deep learning to various other fields, such as bioinformatics, finance, agriculture, chemistry, astronomy, economy, and more.

Breakdown-oriented approach

This approach might be similar to the approach considered in traditional machine learning algorithms. We already talked about how feature engineering is the key to improving precision in machine learning. Now we can say that this feature engineering can be divided into the following two parts:

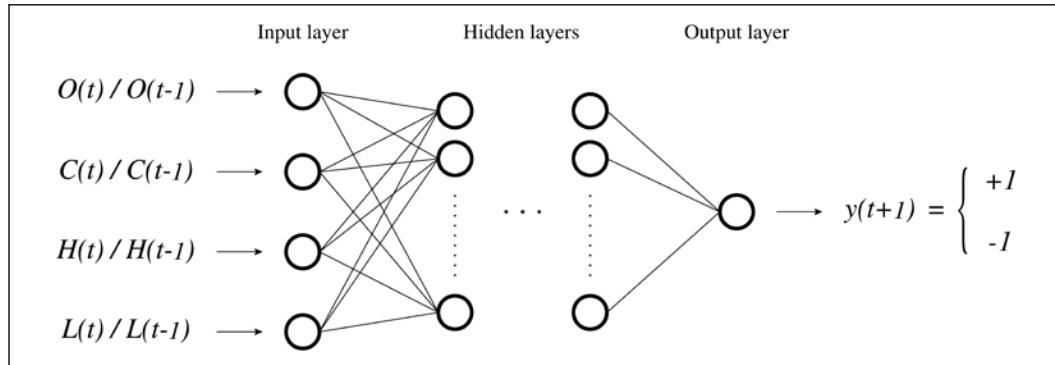
- Engineering under the constraints of a machine learning model. The typical case is to make inputs discrete or continuous.
- Feature engineering to increase precision by machine learning. This tends to rely on the sense of a researcher.

In a narrower meaning, feature engineering is considered as the second one, and this is the part that deep learning doesn't have to focus on, whereas the first one is definitely the important part, even for deep learning. For example, it's difficult to predict stock prices using deep learning. Stock prices are volatile and it's difficult to define inputs. Besides, how to apply an output value is also a difficult problem. Enabling deep learning to handle these inputs and outputs is also said to be feature engineering in the wider sense. If there is no limitation to the value of original data and/or data you would like to predict, it's difficult to insert these datasets into machine learning and deep learning algorithms, including neural networks.

However, we can take a certain approach and apply a model to these previous problems by breaking down the inputs and/or outputs. In terms of NLP, as explained earlier, you might have thought, for example, that it would be impossible to put numberless words into features in the first place, but as you already know, we can train feed-forward neural networks with words by representing them with sparse vectors and combining N-grams into them. Of course, we can not only use neural networks, but also other machine learning algorithms such as SVM here. Thus, we can cultivate a new field where deep learning hasn't been applied by engineering to fit features well into deep learning models. In the meantime, when we focus on NLP, we can see that RNN and LSTM were developed to properly resolve the difficulties or tasks encountered in NLP. This can be considered as the opposite approach to feature engineering because in this case, the problem is solved by breaking down a model to fit into features.

Then, how do we do utilize engineering for stock prediction as we just mentioned? It's actually not difficult to think of inputs, that is, features. For example, if you predict stock prices daily, it's hard to calculate if you use daily stock prices as features, but if you use a rate of price change between a day and the day before, then it should be much easier to process as the price stays within a certain range and the gradients won't explode easily. Meanwhile, what is difficult is how to deal with outputs. Stock prices are of course continuous values, hence outputs can be various values. This means that in the neural network model where the number of units in the output layer is fixed, they can't handle this problem. What should we do here—should we give up?! No, wait a minute. Unfortunately, we can't predict a stock price itself, but there is an alternative prediction method.

Here, the problem is that we can classify stock prices to be predicted into infinite patterns. Then, can we make them into limited patterns? Yes, we can. Let's forcibly make them. Think about the most extreme but easy to understand case: predicting whether tomorrow's stock price, strictly speaking a close price, is up or down using the data from the stock price up to today. For this case, we can show it with a deep learning model as follows:



In the preceding image, $o(t)$ denotes the open price of a day, t ; $c(t)$ denotes the close price, $H(t)$ is the high price, and $L(t)$ is the actual price. The features used here are mere examples, and need to be fine-tuned when applied to real applications. The point here is that replacing the original task with this type of problem enables deep neural networks to theoretically classify data. Furthermore, if you classify the data by how much it will go up or down, you could make more detailed predictions. For example, you could classify data as shown in the following table:

Class	Description
Class 1	Up more than 3 percent from the closing price
Class 2	Up more than 1~3 percent from the closing price
Class 3	Up more than 0~1 percent from the closing price
Class 4	Down more than 0~1 percent from the closing price
Class 5	Down more than -1~-3 percent from the closing price
Class 6	Down more than -3 percent from the closing price

Whether the prediction actually works, in other words whether the classification works, is unknown until we examine it, but the fluctuation of stock prices can be predicted in quite a narrow range by dividing the outputs into multiple classes. Once we can adopt the task into neural networks, then what we should do is just examine which model gets better results. In this example, we may apply RNN because the stock price is time sequential data. If we look at charts showing the price as image data, we can also use CNN to predict the future price.

So now we've thought about the approach by referring to examples, but to sum up in general, we can say that:

- **Feature engineering for models:** This is designing inputs or adjusting values to fit deep learning models, or enabling classification by setting a limitation for the outputs
- **Model engineering for features:** This is devising new neural network models or algorithms to solve problems in a focused field

The first one needs ideas for the part of designing inputs and outputs to fit to a model, whereas the second one needs to take a mathematical approach. Feature engineering might be easier to start if you are conscious of making an item prediction-limited.

Output-oriented approach

The two previously mentioned approaches are to increase the percentage of correct answers for a certain field's task or problem using deep learning. Of course, it is essential and the part where deep learning proves its worth; however, increasing precision to the ultimate level may not be the only way of utilizing deep learning. Another approach is to devise the outputs using deep learning by slightly changing the point of view. Let's see what this means.

Deep learning is applauded as an innovative approach among researchers and technical experts of AI, but the world in general doesn't know much about its greatness yet. Rather, they pay attention to what a machine can't do. For example, people don't really focus on the image recognition capabilities of MNIST using CNN, which generates a lower error rate than humans, but they criticize that a machine can't recognize images perfectly. This is probably because people expect a lot when they hear and imagine AI. We might need to change this mindset. Let's consider DORAEMON, a Japanese national cartoon character who is also famous worldwide – a robot who has high intelligence and AI, but often makes silly mistakes. Do we criticize him? No, we just laugh it off or take it as a joke and don't get serious. Also, think about DUMMY / DUM-E, the robot arm in the movie *Iron Man*. It has AI as well, but makes silly mistakes. See, they make mistakes but we still like them.

In this way, it might be better to emphasize the point that machines make mistakes. Changing the expression part of a user interface could be the trigger for people to adopt AI rather than just studying an algorithm the most. Who knows? It's highly likely that you can gain the world's interest by thinking of ideas in creative fields, not from the perspective of precision. Deep Dream by Google is one good example. We can do more exciting things when art or design and deep learning collaborate.

Summary

In this chapter, you learned how to utilize deep learning algorithms for practical applications. The fields that are well studied are image recognition and NLP. While learning about the field of NLP, we looked through two new deep learning models: the RNN and LSTM networks, which can be trained with time sequential data. The training algorithm used in these models is BPTT. You also learned that there are three approaches to make the best of the deep learning ability: the field-oriented approach, the breakdown-oriented approach, and the output-oriented approach. Each approach has a different angle, and can maximize the possibility for deep learning.

And ...congratulations! You've just accomplished the learning part of deep learning with Java. Although there are still some models that have not been mentioned yet in this book, you can be sure there will be no problem in acquiring and utilizing them. The next chapter will introduce some libraries that are implemented with other programming languages, so just relax and take a look.

7

Other Important Deep Learning Libraries

In this chapter, we'll talk about other deep learning libraries, especially libraries with programming languages other than Java. The following are the most famous, well-developed libraries:

- Theano
- TensorFlow
- Caffe

You'll briefly learn about each of them. Since we'll mainly implement them using Python here, you can skip this chapter if you are not a Python developer. All the libraries introduced in this chapter support GPU implementations and have other special features, so let's dig into them.

Theano

Theano was developed for deep learning, but it is not actually a deep learning library; it is a Python library for scientific computing. The documentation is available at <http://deeplearning.net/software/theano/>. There are several characteristics introduced on the page such as the use of a GPU, but the most striking feature is that Theano supports **computational differentiation** or **automatic differentiation**, which ND4J, the Java scientific computing library, doesn't support. This means that, with Theano, we don't have to calculate the gradients of model parameters by ourselves. Theano automatically does this instead. Since Theano undertakes the most complicated parts of the algorithm, implementations of math expressions can be less difficult.

Let's see how Theano computes gradients. To begin with, we need to install Theano on the machine. Installation can be done just by using `pip install Theano` or `easy_install Theano`. Then, the following are the lines to import and use Theano:

```
import theano
import theano.tensor as T
```

With Theano, all variables are processed as tensors. For example, we have `scalar`, `vector`, and `matrix`, `d` for double, `l` for long, and so on. Generic functions such as `sin`, `cos`, `log`, and `exp` are also defined under `theano.tensor`. Therefore, as shown previously, we often use the alias of tensor, `T`.

As a first step to briefly grasp Theano implementations, consider the very simple parabola curve. The implementation is saved in `DLWJ/src/resources/theano/1_1_parabola_scalar.py` so that you can reference it. First, we define `x` as follows:

```
x = T.dscalar('x')
```

This definition is unique with Python because `x` doesn't have a value; it's just a symbol. In this case, `x` is `scalar` of the type `d` (double). Then we can define `y` and its gradient very intuitively. The implementation is as follows:

```
y = x ** 2
dy = T.grad(y, x)
```

So, `dy` should have $2x$ within it. Let's check whether we can get the correct answers. What we need to do additionally is to register the `math` function with Theano:

```
f = theano.function([x], dy)
```

Then you can easily compute the value of the gradients:

```
print f(1) # => 2.0
print f(2) # => 4.0
```

Very simple! This is the power of Theano. We have `x` of `scalar` here, but you can easily implement `vector` (and `matrix`) calculations as well just by defining `x` as:

```
x = T.dvector('x')
y = T.sum(x ** 2)
```

We won't go further here, but you can find the completed codes in `DLWJ/src/resources/theano/1_2_parabola_vector.py` and `DLWJ/src/resources/theano/1_3_parabola_matrix.py`.

When we consider implementing deep learning algorithms with Theano, we can find some very good examples on GitHub in *Deep Learning Tutorials* (<https://github.com/lisa-lab/DeepLearningTutorials>). In this chapter, we'll look at an overview of the standard MLP implementation so you understand more about Theano. The forked repository as a snapshot is available at <https://github.com/yusugomori/DeepLearningTutorials>. First, let's take a look at `mlp.py`. The model parameters of the hidden layer are the weight and bias:

```
W = theano.shared(value=W_values, name='W', borrow=True)
b = theano.shared(value=b_values, name='b', borrow=True)
```

Both parameters are defined using `theano.shared` so that they can be accessed and updated through the model. The activation can be represented as follows:

$$z = h(Wx + b)$$

This denotes the activation function, that is, the hyperbolic tangent in this code. Therefore, the corresponding code is written as follows:

```
lin_output = T.dot(input, self.W) + self.b
self.output = (
    lin_output if activation is None
    else activation(lin_output)
)
```

Here, linear activation is also supported. Likewise, parameters `w` and `b` of the output layer, that is, logistic regression layer, are defined and initialized in `logistic_sgd.py`:

```
self.W = theano.shared(
    value=numpy.zeros(
        (n_in, n_out),
        dtype=theano.config.floatX
    ),
    name='W',
    borrow=True
)

self.b = theano.shared(
    value=numpy.zeros(
        (n_out,),
        dtype=theano.config.floatX
    ),
    name='b',
    borrow=True
)
```

```
        dtype=theano.config.floatX
    ) ,
    name='b',
    borrow=True
)
```

The activation function of multi-class logistic regression is the `softmax` function and we can just write and define the output as follows:

```
self.p_y_given_x = T.nnet.softmax(T.dot(input, self.W) + self.b)
```

We can write the predicted values as:

```
self.y_pred = T.argmax(self.p_y_given_x, axis=1)
```

In terms of training, since the equations of the backpropagation algorithm are computed from the loss function and its gradient, what we need to do is just define the function to be minimized, that is, the negative log likelihood function:

```
def negative_log_likelihood(self, y):
    return -T.mean(T.log(self.p_y_given_x)[T.arange(y.shape[0]), y])
```

Here, the mean values, not the sum, are computed to evaluate across the mini-batch.

With these preceding values and definitions, we can implement MLP. Here again, what we need to do is define the equations and symbols of MLP. The following is an extraction of the code:

```
class MLP(object):
    def __init__(self, rng, input, n_in, n_hidden, n_out):
        # self.hiddenLayer = HiddenLayer(...)
        # self.logRegressionLayer = LogisticRegression(...)

        # L1 norm
        self.L1 = (
            abs(self.hiddenLayer.W).sum()
            + abs(self.logRegressionLayer.W).sum()
        )

        # square of L2 norm
        self.L2_sqr = (
            (self.hiddenLayer.W ** 2).sum()
            + (self.logRegressionLayer.W ** 2).sum()
        )

    # negative log likelihood of MLP
```

```

        self.negative_log_likelihood = (
            self.logRegressionLayer.negative_log_likelihood
        )

        # the parameters of the model
        self.params = self.hiddenLayer.params +
            self.logRegressionLayer.params
    )

```

Then you can build and train the model. Let's look at the code in `test_mlp()`. Once you load the dataset and construct MLP, you can evaluate the model by defining the cost:

```

cost = (
    classifier.negative_log_likelihood(y)
    + L1_reg * classifier.L1
    + L2_reg * classifier.L2_sqr
)

```

With this cost, we get the gradients of the model parameters with just a single line of code:

```
gparams = [T.grad(cost, param) for param in classifier.params]
```

The following is the equation to update the parameters:

```

updates = [
    (param, param - learning_rate * gparam)
    for param, gparam in zip(classifier.params, gparams)
]

```

The code in the first bracket follows this equation:

$$\theta \leftarrow \theta - \eta \frac{\partial L}{\partial \theta}$$

Then, finally, we define the actual function for the training:

```

train_model = theano.function(
    inputs=[index],
    outputs=cost,
    updates=updates,
    givens={
        x: train_set_x[index * batch_size: (index + 1) *
batch_size],
        y: train_set_y[index * batch_size: (index + 1) *
batch_size]
    }
)

```

Each indexed input and label corresponds to x, y in *givens*, so when `index` is given, the parameters are updated with `updates`. Therefore, we can train the model with iterations of training epochs and mini-batches:

```
while (epoch < n_epochs) and (not done_looping):
    epoch = epoch + 1
    for minibatch_index in xrange(n_train_batches):
        minibatch_avg_cost = train_model(minibatch_index)
```

The original code has the test and validation part, but what we just mentioned is the rudimentary structure. With Theano, equations of gradients will no longer be derived.

TensorFlow

TensorFlow is the library for machine learning and deep learning developed by Google. The project page is <https://www.tensorflow.org/> and all the code is open to the public on GitHub at <https://github.com/tensorflow/tensorflow>. TensorFlow itself is written with C++, but it provides a Python and C++ API. We focus on Python implementations in this book. The installation can be done with `pip`, `virtualenv`, or `docker`. The installation guide is available at https://www.tensorflow.org/versions/master/get_started/os_setup.html. After the installation, you can import and use TensorFlow by writing the following code:

```
import tensorflow as tf
```

TensorFlow recommends you implement deep learning code with the following three parts:

- `inference()`: This makes predictions using the given data, which defines the model structure
- `loss()`: This returns the error values to be optimized
- `training()`: This applies the actual training algorithms by computing gradients

We'll follow this guideline. A tutorial on MNIST classifications for beginners is introduced on <https://www.tensorflow.org/versions/master/tutorials/mnist/beginners/index.html> and the code for this tutorial can be found in `DLWJ/src/resources/tensorflow/1_1_mnist_simple.py`. Here, we consider refining the code introduced in the tutorial. You can see all the code in `DLWJ/src/resources/tensorflow/1_2_mnist.py`.

First, what we have to consider is fetching the MNIST data. Thankfully, TensorFlow also provides the code to fetch the data in https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/mnist/input_data.py and we put the code into the same directory. Then, by writing the following code, you can import the MNIST data:

```
import input_data
```

MNIST data can be imported using the following code:

```
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

Similar to Theano, we define the variable with no actual values as the placeholder:

```
x_placeholder = tf.placeholder("float", [None, 784])
label_placeholder = tf.placeholder("float", [None, 10])
```

Here, 784 is the number of units in the input layer and 10 is the number in the output layer. We do this because the values in the placeholder change in accordance with the mini-batches. Once you define the placeholder you can move on to the model building and training. We set the non-linear activation with the softmax function in inference() here:

```
def inference(x_placeholder):
    W = tf.Variable(tf.zeros([784, 10]))
    b = tf.Variable(tf.zeros([10]))

    y = tf.nn.softmax(tf.matmul(x_placeholder, W) + b)

    return y
```

Here, `W` and `b` are the parameters of the model. The loss function, that is, the cross_entropy function, is defined in `loss()` as follows:

```
def loss(y, label_placeholder):
    cross_entropy = - tf.reduce_sum(label_placeholder * tf.log(y))

    return cross_entropy
```

With the definition of `inference()` and `loss()`, we can train the model by writing the following code:

```
def training(loss):
    train_step =
        tf.train.GradientDescentOptimizer(0.01).minimize(loss)

    return train_step
```

`GradientDescentOptimizer()` applies the gradient descent algorithm. But be careful, as this method just defines the method of training and the actual training has not yet been executed. TensorFlow also supports `AdagradOptimizer()`, `MemontumOptimizer()`, and other major optimizing algorithms.

The code and methods explained previously are to define the model. To execute the actual training, you need to initialize a session of TensorFlow:

```
init = tf.initialize_all_variables()
sess.run(init)
```

Then we train the model with mini-batches. All the data in a mini-batch is stored in `feed_dict` and then used in `sess.run()`:

```
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    feed_dict = {x_placeholder: batch_xs, label_placeholder:
batch_ys}

    sess.run(train_step, feed_dict=feed_dict)
```

That's it for the model training. It's very simple, isn't it? You can show the result by writing the following code:

```
def res(y, label_placeholder, feed_dict):
    correct_prediction = tf.equal(
        tf.argmax(y, 1), tf.argmax(label_placeholder, 1)
    )

    accuracy = tf.reduce_mean(
        tf.cast(correct_prediction, "float")
    )

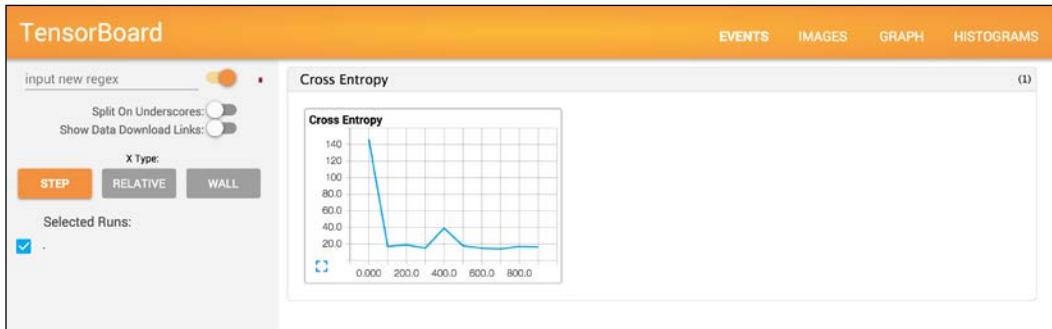
    print sess.run(accuracy, feed_dict=feed_dict)
```

TensorFlow makes it super easy to implement deep learning and it is very useful. Furthermore, TensorFlow has another powerful feature, `TensorBoard`, to visualize deep learning. By adding a few lines of code to the previous code snippet, we can use this useful feature.

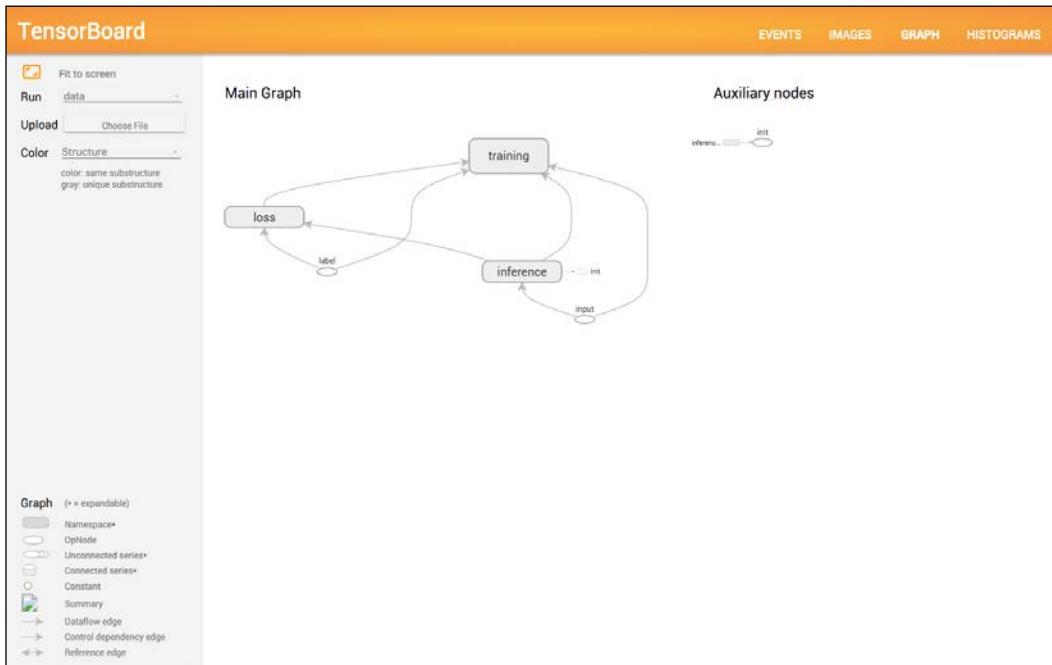
Let's see how the model is visualized first. The code is in `DLWJ/src/resources/tensorflow/1_3_mnist_TensorBoard.py`, so simply run it. After you run the program, type the following command:

```
$ tensorboard --logdir=<ABOSOLUTE_PATH>/data
```

Here, <ABSOLUTE_PATH> is the absolute path of the program. Then, if you access <http://localhost:6006/> in your browser, you can see the following page:

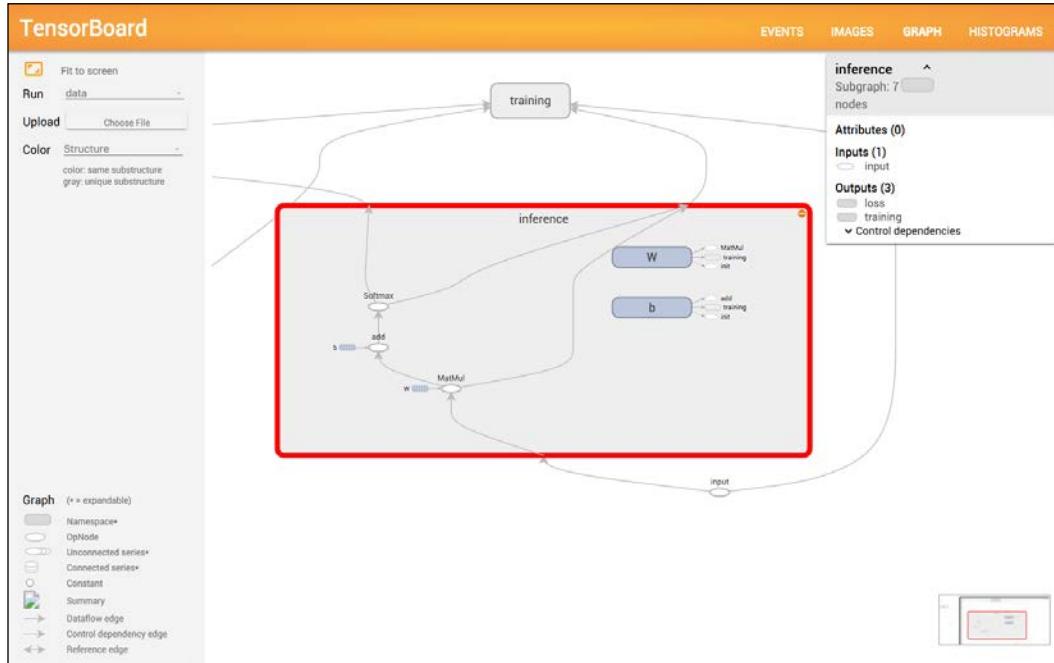


This shows the process of the value of `cross_entropy`. Also, when you click **GRAPH** in the header menu, you see the visualization of the model:



Other Important Deep Learning Libraries

When you click on **inference** on the page, you can see the model structure:



Now let's look inside the code. To enable visualization, you need to wrap the whole area with the scope: `with tf.Graph().as_default()`. By adding this scope, all the variables declared in the scope will be displayed in the graph. The displayed name can be set by including the name label as follows:

```
x_placeholder = tf.placeholder("float", [None, 784], name="input")
label_placeholder = tf.placeholder("float", [None, 10],
name="label")
```

Defining other scopes will create nodes in the graph and this is where the division, `inference()`, `loss()`, and `training()` reveal their real values. You can define the respective scope without losing any readability:

```
def inference(x_placeholder):
    with tf.name_scope('inference') as scope:
        W = tf.Variable(tf.zeros([784, 10]), name="W")
        b = tf.Variable(tf.zeros([10]), name="b")

    y = tf.nn.softmax(tf.matmul(x_placeholder, W) + b)
```

```
    return y

def loss(y, label_placeholder):
    with tf.name_scope('loss') as scope:
        cross_entropy = - tf.reduce_sum(label_placeholder * tf.log(y))

    tf.scalar_summary("Cross Entropy", cross_entropy)

    return cross_entropy

def training(loss):
    with tf.name_scope('training') as scope:
        train_step =
            tf.train.GradientDescentOptimizer(0.01).minimize(loss)

    return train_step
```

`tf.scalar_summary()` in `loss()` makes the variable show up in the **EVENTS** menu. To enable visualization, we need the following code:

```
summary_step = tf.merge_all_summaries()
init = tf.initialize_all_variables()

summary_writer = tf.train.SummaryWriter('data',
                                         graph_def=sess.graph_def)
```

Then the process of variables can be added with the following code:

```
summary = sess.run(summary_step, feed_dict=feed_dict)
summary_writer.add_summary(summary, i)
```

This feature of visualization will be much more useful when we're using more complicated models.

Caffe

Caffe is a library famous for its speed. The official project page is <http://caffe.berkeleyvision.org/> and the GitHub page is <https://github.com/BVLC/caffe>. Similar to TensorFlow, Caffe has been developed mainly with C++, but it provides a Python and MATLAB API. In addition, what is unique to Caffe is that you don't need any programming experience, you just write the configuration or protocol files, that is .prototxt files, to perform experiments and research with deep learning. Here, we focus on the protocol-based approach.

Caffe is a very powerful library that enables quick model building, training, and testing; however, it's a bit difficult to install the library to get a lot of benefits from it. As you can see from the installation guide at <http://caffe.berkeleyvision.org/installation.html>, you need to install the following in advance:

- CUDA
- BLAS (ATLAS, MKL, or OpenBLAS)
- OpenCV
- Boost
- Others: snappy, leveldb, gflags, glog, szip, lmdb, protobuf, and hdf5

Then, clone the repository from the GitHub page and create the `Makefile.config` file from `Makefile.config.example`. You may need Anaconda, a Python distribution, beforehand to run the `make` command. You can download this from <https://www.continuum.io/downloads>. After you run the `make`, `make test`, and `make runtest` commands (you may want to run the commands with a `-jN` option such as `make -j4` or `make -j8` to speed up the process) and pass the test, you'll see the power of Caffe. So, let's look at an example. Go to `$CAFFE_ROOT`, the path where you cloned the repository, and type the following commands:

```
$ ./data/mnist/get_mnist.sh  
$ ./examples/mnist/train_lenet.sh
```

That's all you need to solve the standard MNIST classification problem with CNN. So, what happened here? When you have a look at `train_lenet.sh`, you will see the following:

```
#!/usr/bin/env sh  
  
./build/tools/caffe train --solver=examples/mnist/lenet_solver.  
prototxt
```

It simply runs the `caffe` command with the protocol file `lenet_solver.prototxt`. This file configures the hyper parameters of the model such as the learning rate and the momentum. The file also references the network configuration file, in this case, `lenet_train_test.prototxt`. You can define each layer with a JSON-like description:

```
layer {
    name: "conv1"
    type: "Convolution"
    bottom: "data"
    top: "conv1"
    param {
        lr_mult: 1
    }
    param {
        lr_mult: 2
    }
    convolution_param {
        num_output: 20
        kernel_size: 5
        stride: 1
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
        }
    }
}
```

So, basically, the protocol file is divided into two parts:

- **Net:** This defines the detailed structure of the model and gives a description of each layer, hence whole neural networks
- **Solver:** This defines the optimization settings such as the use of a CPU/GPU, the number of iterations, and the hyper parameters of the model such as the learning rate

Caffe can be a great tool when you need to apply deep learning to a large dataset with principal approaches.

Summary

In this chapter, you learned how to implement deep learning algorithms and models using Theano, TensorFlow, and Caffe. All of them have special and powerful features and each of them is very useful. If you are interested in other libraries and frameworks, you can have *Chainer* (<http://chainer.org/>), *Torch* (<http://torch.ch/>), *Pylearn2* (<http://deeplearning.net/software/pylearn2/>), *Nervana* (<http://neon.nervanasys.com/>), and so on. You can also reference some benchmark tests (<https://github.com/soumith/convnet-benchmarks> and <https://github.com/soumith/convnet-benchmarks/issues/66>) when you actually consider building your application with one of the libraries mentioned earlier.

Throughout this book, you learned the fundamental theories and algorithms of machine learning and deep learning and how deep learning is applied to study/business fields. With the knowledge and techniques you've acquired here, you should be able to cope with any problems that confront you. While it is true that you still need more steps to realize AI, you now have the greatest opportunity to achieve innovation.

8

What's Next?

In the previous chapters, we learned the concept, theory, implementation of deep learning, and how to use libraries. Now you know the basic technique of deep learning, so don't worry. On the other hand, development of deep learning is rapid and a new model might be developed tomorrow. Big news about AI or deep learning comes out one after the other every day. Since you have acquired the basic technique, you can learn about the upcoming new technologies on AI and deep learning quickly. Now, let's walk away from the details of techniques and think about what path the field of AI will or should take. What is the future of deep learning? For the closing chapter, let's think about that. We'll pick up the following topics in this chapter:

- Hot topics in the deep learning industry
- How to manage AI technologies
- How to proceed the study of deep learning further

As for the last topic, about further study, I will recommend a website about deep learning. You can stay ahead of the curve by thinking about what technology might come next, or leveraging the techniques you have learned to innovate, rather than following AI developments as they appear.

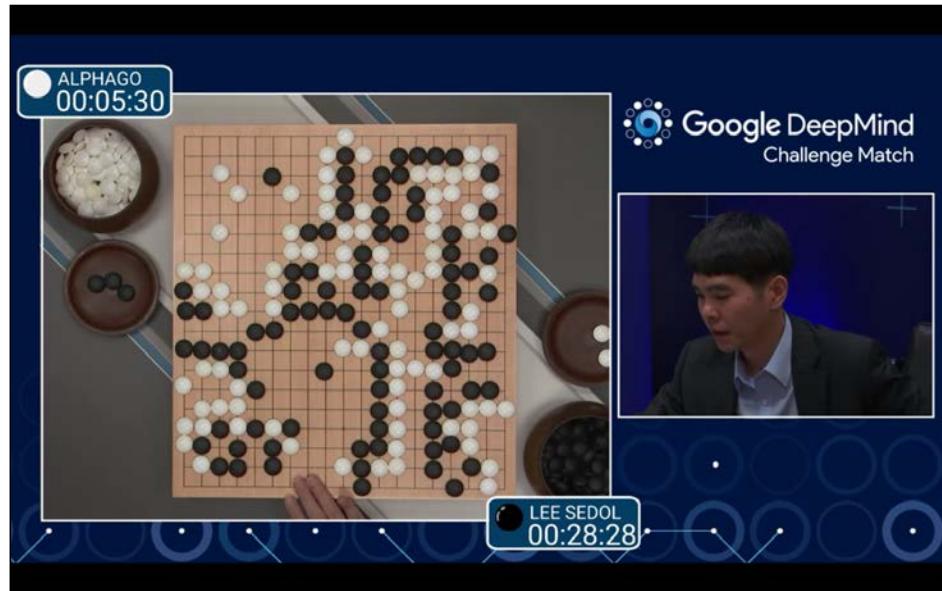
Breaking news about deep learning

Deep learning, which triggered the AI boom, can't stop its momentum. New results are reported each day. As mentioned in *Chapter 6, Approaches to Practical Applications – Recurrent Neural Networks and More*, many researchers compete on image recognition and natural language processing. Of course, deep learning is not limited to these two fields, but applies to many other fields. The outcome of these applications is very exciting.

In this AI boom, in March 2016, the Go world was shaken by one event. Go is a board game in which two players are trying to take over more territory than their opponent. The news of 'AI beating a human in Go' shocked not only Go players but also the whole world, when a machine, that is, AI, beat a human at Go. DeepMind (<https://deepmind.com/>), which was bought by Google, has been developing the Go-playing AI, called AlphaGo, and AlphaGo has beaten world-class player Lee Sedol for a fourth time, to win the five-game series four games to one in Google's DeepMind Challenge Match. Each game was delivered on livestream via YouTube, and many people watched the games in real time. You can watch all five games on YouTube, if you haven't watched them yet:

- **Match 1:** <https://www.youtube.com/watch?v=vFr3K2DORc8>
- **Match 2:** <https://www.youtube.com/watch?v=l-GsfyVCBu0>
- **Match 3:** <https://www.youtube.com/watch?v=qUAmTYHEyM8>
- **Match 4:** <https://www.youtube.com/watch?v=yCALyQRN3hw>
- **Match 5:** <https://www.youtube.com/watch?v=mzpW10DPHeQ>

In these games, the first match, Match 1, gained a particularly large amount of attention. Also, looking at the number of page views, which is more than 1.35 million, you can see that many people have watched the video of Match 4, which was the only match that AlphaGo lost to Lee Se-Dol. The following is an image which captured one scene of Match 1:



The moment AlphaGo beat Lee SeDol (<https://www.youtube.com/watch?v=vFr3K2DORc8>)

That was the moment at which not only researchers of AI, but also the world, were excited by AlphaGo; but why did this news get so much attention? For another board game example, in a chess match, Deep Blue, which was developed by IBM, beat the world chess champion in 1997. Of course, it became big news at that time, as this was also a moment when a machine beat a human. Why then, when this was not the first time a machine had beaten a human, was news of AlphaGo's triumph against Lee SeDol so world-shaking? What is the difference between Chess and Go? Well, the difference is in the complexity of the patterns of Go. In fact, Go has many more strategy patterns than Chess. In popular board games such as Chess, Shogi, and Go, the numbers of patterns to determine who wins or loses are as follows:

- Chess: 10,120
- Shogi: 10,220
- Go: 10,360

Even looking at the numbers, you can see how complicated the strategy of Go is and easily imagine that a machine also needs an enormous amount of calculation. Because of this huge number of patterns, until recently, people thought it was impossible for AlphaGo to beat a human, or that it would be 100 years or 200 years before AlphaGo would beat a human. It was considered impossible for a machine to calculate the patterns of Go within a realistic time. But now, in a matter of a few years, a machine has beaten a human. According to the Google research blog, 1.5 months before the DeepMind Challenge Match was held, DeepMind could predict the human's moves 57% of the time (<http://googleresearch.blogspot.jp/2016/01/alphago-mastering-ancient-game-of-go.html>). The fact that a machine won against a human definitely had an impact, but the fact that a machine could learn the strategy of Go within a realistic time was even more surprising. DeepMind applies deep neural networks with the combination of Monte Carlo tree search and reinforcement learning, which shows the width of the application range for the algorithm of deep neural networks.

Expected next actions

Since the news about AlphaGo was featured in the media, the AI boom has definitely had a boost. You might notice that you hear the words "deep learning" in the media more often recently. It can be said that the world's expectations of AI have been increased that much. What is interesting is that the term "deep learning," which was originally a technical term, is now used commonly in daily news. You can see that the image of the term **AI** has been changing. Probably, until just a few years ago, if people heard about AI, many of them would have imagined an actual robot, but how about now? The term AI is now often used—not particularly consciously—with regard to software or applications, and is accepted as commonplace. This is nothing but an indication that the world has started to understand AI, which has been developed for research, correctly. If a technology is taken in the wrong direction, it generates repulsion, or some people start to develop the technology incorrectly; however, it seems that this boom in AI technology is going in a good direction so far.

While we are excited about the development of AI, as a matter of course, some people feel certain fears or anxieties. It's easy to imagine that some people might think the world where machines dominate humans, like in sci-fi movies or novels, is coming sooner or later, especially after AlphaGo won over Lee SeDol in the Go world, where it was said to be impossible for a machine to beat a human; the number of people who feel anxious might increase. However, although the news that a machine has beaten a human could be taken as a negative if you just focus on the fact that "a machine won," this is definitely not negative news. Rather, it is great news for humankind. Why? Here are two reasons.

The first reason is that the Google DeepMind Challenge Match was a match in which the human was handicapped. Not only for Go, but also for card games or sports games, we usually do research about what tactics the opponents will use before a match, building our own strategy by studying opponents' action patterns. DeepMind, of course, has studied professional Go players' tactics and how to play, whereas humans couldn't study enough about how a machine plays, as DeepMind continued studying and kept changing its action patterns until the last minutes before the Google DeepMind Challenge Match. Therefore, it can be said that there was an information bias or handicap. It was great that Lee SeDol won one match with these handicaps. Also, it indicates that AI will develop further.

The other reason is that we have found that a machine is not likely to destroy the value of humans, but instead to promote humans' further growth. In the Google DeepMind Challenge Match, a machine used a strategy which a human had not used before. This fact was a huge surprise to us, but at the same time, it meant that we found a new thing which humans need to study. Deep learning is obviously a great technology, but we shouldn't forget that neural networks involve an algorithm which imitates the structure of a human brain. In other words, its fundamentals are the same as a human's patterns of thinking. A machine can find out an oversight of patterns which the human brain can't calculate by just adding the speed of calculation. AlphaGo can play a game against AlphaGo using the input study data, and learns from the result of that game too. Unlike a human, a machine can proceed to study for 24 hours, so it can gain new patterns rapidly. Then, a whole new pattern will be found by a machine during that process, which can be used for humans to study Go further. By studying a new strategy which wouldn't been found just by a human, our Go world will expand and we can enjoy Go even more. Needless to say, it is not only machines that learn, but also humans. In various fields, a machine will discover new things which a human hasn't ever noticed, and every time humans face that new discovery, they too advance.

AI and humans are in a complementary relationship. To reiterate, a machine is good at calculating huge numbers of patterns and finding out a pattern which hasn't been discovered yet. This is way beyond human capability. On the other hand, AI can't create a new idea from a completely new concept, at least for now. On the contrary, this is the area where humans excel. A machine can judge things only within given knowledge. For example, if AI is only given many kinds of dog images as input data, it can answer what kind of dog it is, but if it's a cat, then AI would try its best to answer the kind of dog, using its knowledge of dogs.

AI is actually an innocent existence in a way, and it just gives the most likely answer from its gained knowledge. Thinking what knowledge should be given for AI to make progress is a human's task. If you give new knowledge, again AI will calculate the most likely answer from the given knowledge with quite a fast pace. People also have different interests or knowledge depending on the environment in which they grow up, which is the same for AI. Meaning, what kind of *personality* the AI has or whether the AI becomes good or evil for humans depends on the person/people the AI has contact with. One such typical example, in which AI was grown in the wrong way, is the AI developed by Microsoft called Tay (<https://www.tay.ai>). On March 23, 2016, Tay appeared on Twitter with the following tweet: *helloooooo world!!!*

Tay gains knowledge from the interaction between users on Twitter and posts new tweets. This trial itself is quite interesting.

What's Next?

However, immediately after it was made open to the public, the problem occurred. On Twitter, users played a prank on Tay by inputting discriminatory knowledge into its account. Because of this, Tay has grown to keep posting tweets including expressions of sexual discrimination. And only one day after Tay appeared on Twitter, Tay disappeared from Twitter, leaving the following tweet: *c u soon humans need sleep now so many conversations today thx.*

If you visit Tay's Twitter account's page (<https://twitter.com/tayandyou>), tweets are protected and you can't see them anymore:



The Twitter account of Tay is currently closed

This is exactly the result of AI being given the wrong training by humans. In these past few years, the technology of AI has got huge attention, which can be one of the factors to speed up the development of AI technology further. Now, the next action that should be taken is to think how AI and humans interact with each other. AI itself is just one of many technologies. Technology can become good or evil depending on how humans use it; therefore, we should be careful how we control that technology, otherwise it's possible that the whole AI field will be shrinking in the future. AI is becoming particularly good within certain narrow fields, but it is far from overwhelming, and far from what science fiction currently envisions. How AI will evolve in the future depends on our use of knowledge and technology management.

While we should definitely care about how to control the technology, we can't slow down the speed of its development. Considering recent booms of bots, as seen in the story that Facebook is going to launch Bot Store (<http://techcrunch.com/2016/03/17/facebook-messenger-in-a-bot-store/>), we can easily imagine that the interaction between a user and an application would become a chat-interface base, and AI would mingle with the daily life of an ordinary user going forward. For more people to get familiar with AI, we should develop AI technology further and make it more convenient for people to use.

Deep learning and AI have got more attention, which means that if you would like to produce an outstanding result in this field, you are likely to find fierce competition. It's highly likely that an experiment you would like to work on might already be being worked on by someone else. The field of deep learning is becoming a world of such high competition as start-ups. If you own huge data, you might take advantage by analyzing that data, but otherwise, you need to think about how to experiment with limited data. Still, if you would like to get outstanding performance, it might be better for you to always bear the following in mind:



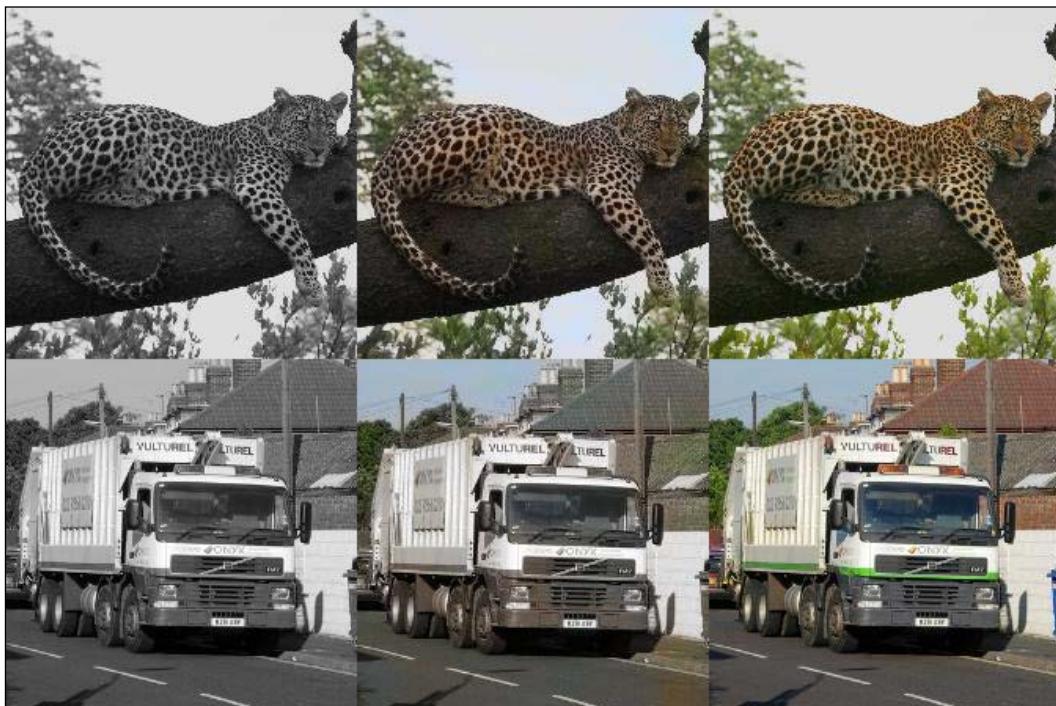
Deep learning can only judge things within the knowledge given by training.



Based on this, you might get an interesting result by taking the following two approaches:

- Experiment with data which can easily produce both input data and output data for training and testing
- Use completely different types of data, for training and test respectively, in an experiment

For the first approach, for example, you can check automatic colorization using CNN. It was introduced in the project open to the public online at <http://tinyclouds.org/colorize/> or in the dissertation at <http://arxiv.org/pdf/1603.08511v1.pdf>. The idea is to color gray-scale images automatically. If you have any colored images – these should be obtained very easily – you can generate grayscale images just by writing quick scripts. With that, you have now prepared input data and output data for training. Being able to prepare lots of data means you can test easier and get high precision more often. The following is one of the examples of the tests:



Both are cited from <http://tinyclouds.org/colorize/>

Inputs to the model are the grayscale images on the left, outputs are the middle, and the images on the right are the ones with true color.

For the second approach, using completely different types of data, for training and testing respectively, in an experiment, we intentionally provide data which the AI doesn't know and make the gap between a random answer and a correct answer interesting/fun. For example, in *Generating Stories about Images* (<https://medium.com/@samim/generating-stories-about-images-d163ba41e4ed>), they have provided an image of a sumo wrestler to neural networks, which have only studied one of the projects and introduces the following: romantic novels and then test what the neural networks think. The result is as follows:



Cited from <https://medium.com/@samim/generating-stories-about-images-d163ba41e4ed>

This experiment itself is based on the approach called neural-storyteller (<https://github.com/ryankiros/neural-storyteller>), but since the given data has an idea, it got the interesting result. As such, adding your new idea to an already developed approach would be an approach which could also get an interesting result.

Useful news sources for deep learning

Well, lastly, let's pick up two websites which would be useful for watching the movement of deep learning going forward and for learning more and more new knowledge. It will help your study.

What's Next?

The first one is **GitXiv** (<http://gitxiv.com/>). The top page is as follows:

The screenshot shows the GitXiv homepage with a header navigation bar including 'About', 'Competitions', 'Categories', 'Search', 'Register', and 'Sign In'. Below the header is a newsletter sign-up form with fields for 'Your Email' and a 'Get Newsletter' button. The main content area is titled 'Collaborative Open Computer Science' and features a list of research papers. The papers are numbered 1 through 5, each with a thumbnail image, title, abstract, category tags (e.g., DEEP LEARNING (DL), CONVOLUTIONAL NEURAL NETWORKS (CNN)), and a timestamp. Each paper entry includes a comment icon with a number and a reply icon.

- 1 **Colorful Image Colorization**
Automatically colorization of grayscale photograph.
DEEP LEARNING (DL) · CONVOLUTIONAL NEURAL NETWORKS (CNN) · GENERATIVE
aamini 2 points 9 hours ago 0 Comments
- 2 **Regularization of Neural Networks using DropConnect**
Regularization of Neural Networks using DropConnect.
DEEP LEARNING (DL)
lassael 2 points 3 days ago 0 Comments
- 3 **Data-Efficient Learning of Feedback Policies from Image Pixels using Deep Dynamical Models**
Data-Efficient Learning of Feedback Policies from Image Pixels using Deep Dynamical Models
DEEP LEARNING (DL) · DEEP REINFORCEMENT LEARNING (DRL)
lassael 1 point 3 days ago 0 Comments
- 4 **Embed to Control: A Locally Linear Latent Dynamics Model for Control from Raw Images**
Embed to Control: A Locally Linear Latent Dynamics Model for Control from Raw Images
DEEP LEARNING (DL)
lassael 1 point 3 days ago 0 Comments
- 5 **Deep Exploration via Bootstrapped DQN**
Deep Exploration via Bootstrapped DQN
DEEP LEARNING (DL) · DEEP REINFORCEMENT LEARNING (DRL)
lassael 4 points 4 days ago 0 Comments

Layer-sequential unit-variance (LSUV) initialization for CNN

In GitXiv, there are mainly articles based on papers. But in addition to the links to papers, it sets the links to codes which were used for tests, hence you can shorten your research time. Of course, it updates new experiments one after another, so you can watch what approach is major or in which field deep learning is hot now. It sends the most updated information constantly if you register your e-mail address. You should try it:

The screenshot shows a project page on GitXiv. At the top, there are navigation links: 'About', 'Competitions', 'Categories', 'Search', 'Register', and 'Sign in'. The main title is 'Colorful Image Colorization' with a subtitle 'Automatically colorizing of grayscale photograph.' Below the title is a small thumbnail image showing four grayscale photographs being colorized. To the right of the thumbnail are the labels 'DEEP LEARNING (DL)', 'CONVOLUTIONAL NEURAL NETWORKS (CNN)', and 'GENERATIVE'. Below these labels are 'samples 2 points', '9 hours ago', and '8 Comments'. On the left, there's a sidebar for 'arXiv' which lists the authors: Richard Zhang, Phillip Isola, Alexei A. Efros. The main content area contains two columns: 'GitHub' and 'Links'. The GitHub column contains instructions for running the demo code, mentioning an iPython notebook named 'colorization_demo_v0.ipynb'. It also includes a link to the repository and instructions for cloning it. The 'Links' column contains a single item: a link to 'richzhang.github.io/colorization/'. The text in the GitHub section is as follows:

Given a grayscale photograph as input, this paper attacks the problem of hallucinating a plausible color version of the photograph. We propose a fully automatic approach that produces vibrant and realistic colorizations. The system is implemented as a feed-forward operation in a CNN at test time and is trained on over a million color images. Our method successfully fools humans 20% of the time, significantly higher than previous methods.

arXiv

Richard Zhang, Phillip Isola, Alexei A. Efros

Given a grayscale photograph as input, this paper attacks the problem of hallucinating a plausible color version of the photograph. This problem is clearly underconstrained, so previous approaches have either relied on significant user interaction or resulted in desaturated colorizations. We propose a fully automatic approach that produces vibrant and realistic colorizations. We embrace the underlying uncertainty of the problem by posing it as a classification task and explore using class-rebalancing at training time to increase the diversity of colors in the result. The system is implemented as a feed-forward operation in a CNN at test time and is trained on over a million color images. We evaluate our algorithm using a "colorization Turing test", asking human subjects to choose between a

GitHub

This repo contains demo code to run the colorization model described in Colorful Image Colorization
Richard Zhang, Phillip Isola, Alexei A. Efros
In arXiv, 2016

We include demo usage as an iPython notebook, under ./demo/colorization_demo_v0.ipynb.
Clone the repository, cd into the demo directory, run ipython notebook and open colorisation_demo_v0.ipynb in your web browser.

This demo code requires a working installation of Caffe and basic Python libraries (numpy, pylab, skimage, scipy). Please contact Richard Zhang at rich.zhang@eecs.berkeley.edu for any questions or comments.

Links

- <http://richzhang.github.io/colorization/>.

The second one is **Deep Learning News** (<http://news.startup.ml/>). This is a collection of links for deep learning and machine learning related topics. It has the same UI as **Hacker News** (<https://news.ycombinator.com/>), which deals with news for the whole technology industry, so if you know Hacker News, you should be familiar with the layout:

The screenshot shows a list of news items on Deep Learning News. At the top, there are navigation links: 'Deep Learning News', 'top', 'latest', 'random', and 'submit'. On the right, there are 'Login / register' links. The main section is titled 'Top news' and lists various news items with their titles, upvotes, downvotes, poster, and timestamp. Each item is preceded by a small triangle icon. The news items are:

- ▲ We asked deep learning and healthcare experts for their predictions for the next 5 years, the risks involved with AI integration & areas for disruption (part 2) at re-work.co 1 up and 0 down, posted by reworkshoppe 8 hours ago discuss
- ▲ Experts discuss the opportunities and risks of integrating AI & Deep Learning in Healthcare at re-work.co 1 up and 0 down, posted by reworkshoppe 1 day ago discuss
- ▲ Deep Learning in a Nutshell part 3: Sequence Learning at devblogs.nvidia.com 2 up and 0 down, posted by bengioy 23 days ago discuss
- ▲ Machine Learning in Trading, Startup.ML / Bloomberg conference on May 12 at conf.startup.ml 1 up and 0 down, posted by anhak 19 days ago discuss
- ▲ AlphaGo currently 1-0 against Lee Sedol in the DeepMind challenge match at deepmind.com 1 up and 0 down, posted by Markus 23 days ago discuss
- ▲ Understanding Aesthetics with Deep Learning at devblogs.nvidia.com 3 up and 0 down, posted by bengioy 30 days ago discuss
- ▲ Asynchronous Methods for Deep Reinforcement Learning at arxiv.org 2 up and 0 down, posted by anhak 33 days ago discuss
- ▲ Using Deep Q-Learning to Control Optimization Hyperparameters at arxiv.org 1 up and 0 down, posted by anhak 33 days ago discuss
- ▲ AlphaGo: Google DeepMind's Go program beats professional player at googleblog.blogspot.be 2 up and 0 down, posted by Markus 64 days ago discuss
- ▲ Microsoft CNTK framework released on GitHub at blogs.microsoft.com 2 up and 0 down, posted by Markus 66 days ago discuss
- ▲ Which whale is it, anyway? Face recognition for right whales using deep learning at deepsense.io 8 up and 2 down, posted by d_kerr 72 days ago discuss
- ▲ Baldi's AI Lab Releases Warp-CTC at github.com 1 up and 0 down, posted by anhak 72 days ago discuss
- ▲ Intel's Caffe fork to improve performance at github.com 3 up and 0 down, posted by jzajza 81 days ago 1 comment
- ▲ Evaluation of Deep Learning Toolkits at github.com 1 up and 1 down, posted by rudy 92 days ago discuss
- ▲ Startup.ML Deep Learning Conference Videos at conf.startup.ml 3 up and 0 down, posted by anhak 92 days ago discuss
- ▲ Deep-Spying: Spying using Smartwatch and Deep Learning at arxiv.org 1 up and 0 down, posted by reworkshoppe 92 days ago discuss

The information on Deep Learning News is not updated that frequently, but it has tips not only for implementation or technique, but also tips for what field you can use for deep learning, and has deep learning and machine learning related event information, so it can be useful for ideation or inspiration. If you take a brief look at the URL in the top page list, you might come up with good ideas.

There are more useful websites, materials, and communities other than the two we picked up here, such as the deep learning group on Google+ (<https://plus.google.com/communities/112866381580457264725>), so you should watch the media which suit you. Anyway, now this industry is developing rapidly and it is definitely necessary to always watch out for updated information.

Summary

In this chapter, we went from the example of AlphaGo as breaking news to the consideration of how deep learning will or should develop. A machine winning over a human in some areas is not worthy of fear, but is an opportunity for humans to grow as well. On the other hand, it is quite possible that this great technology could go in the wrong direction, as seen in the example of Tay, if the technology of AI isn't handled appropriately. Therefore, we should be careful not to destroy this steadily developing technology.

The field of deep learning is one that has the potential for hugely changing an era with just one idea. If you build AI in the near future, that AI is, so to speak, a pure existence without any knowledge. Thinking what to teach AI, how to interact with it, and how to make use of AI for humankind is humans' work. You, as a reader of this book, will lead a new technology in the right direction. Lastly, I hope you will get actively involved in the cutting edge of the field of AI.

Module 2

Machine Learning in Java

*Dive Design, build, and deploy your own machine learning applications
by leveraging key Java machine learning libraries*

1

Applied Machine Learning Quick Start

This chapter introduces the basics of machine learning, laying down the common themes and concepts and making it easy to follow the logic and familiarize yourself with the topic. The goal is to quickly learn the step-by-step process of **applied machine learning** and grasp the main machine learning principles. In this chapter, we will cover the following topics:

- Introducing machine learning and its relation to data science
- Discussing the basic steps in applied machine learning
- Discussing the kind of data we are dealing with and its importance
- Discussing approaches of collecting and preprocessing the data
- Making sense of data using machine learning
- Using machine learning to extract insights from data and build predictors

If you are already familiar with machine learning and are eager to start coding, then quickly jump to the following chapters. However, if you need to refresh your memory or clarify some concepts, then it is strongly recommended to revisit the topics presented in this chapter.

Machine learning and data science

Nowadays, everyone talks about machine learning and data science. So, what exactly is machine learning anyway? How does it relate to data science? These two terms are commonly confused, as they often employ the same methods and overlap significantly. Therefore, let's first clarify what they are.

Josh Wills tweeted:

"Data scientist is a person who is better at statistics than any software engineer and better at software engineering than any statistician".

- (Josh Wills)

More specifically, data science encompasses the entire process of obtaining knowledge from data by integrating methods from statistics, computer science, and other fields to gain insight from data. In practice, data science encompasses an iterative process of data harvesting, cleaning, analysis and visualization, and deployment.

Machine learning, on the other hand, is mainly concerned with fairly generic algorithms and techniques that are used in analysis and modeling phases of data science process. Arthur Samuel proposed the following definition back in 1955:

"Machine Learning relates with the study, design and development of the algorithms that give computers the capability to learn without being explicitly programmed."

- Arthur Samuel

What kind of problems can machine learning solve?

Among the different machine learning approaches, there are three main ways of learning, as shown in the following list:

- Supervised learning
- Unsupervised learning
- Reinforcement learning

Given a set of example inputs, X , and their outcomes, Y , supervised learning aims to learn a general mapping function, f , that transforms inputs to outputs, as $f: X \rightarrow Y$

An example of supervised learning is credit card fraud detection, where the learning algorithm is presented with credit card transactions (matrix X) marked as normal or suspicious. (vector Y). The learning algorithm produces a decision model that marks unseen transactions as normal or suspicious (that is the f function).

In contrast, unsupervised learning algorithms do not assume given outcome labels, Y as they focus on learning the structure of the data, such as grouping similar inputs into clusters. Unsupervised learning can, hence, discover hidden patterns in the data. An example of unsupervised learning is an item-based recommendation system, where the learning algorithm discovers similar items bought together, for example, *people who bought book A also bought book B*.

Reinforcement learning addresses the learning process from a completely different angle. It assumes that an agent, which can be a robot, bot, or computer program, interacts with a dynamic environment to achieve a specific goal. The environment is described with a set of states and the agent can take different actions to move from one state to another. Some states are marked as goal states and if the agent achieves this state, it receives a large reward. In other states, the reward is smaller, non-existing, or even negative. The goal of reinforcement learning is to find an optimal policy, that is, a mapping function that specifies the action to take in each of the states without a teacher explicitly telling whether this leads to the goal state or not. An example of reinforcement learning is a program for driving a vehicle, where the states correspond to the driving conditions—for example, current speed, road segment information, surrounding traffic, speed limits, and obstacles on the road—and the actions can be driving maneuvers such as turn left or right, stop, accelerate, and continue. The learning algorithm produces a policy that specifies the action that is to be taken in specific configuration of driving conditions.

In this book, we will focus on supervised and unsupervised learning only, as they share many concepts. If reinforcement learning sparked your interest, a good book to start with is *Reinforcement Learning: An Introduction* by Richard S. Sutton and Andrew Barto.

Applied machine learning workflow

This book's emphasis is on applied machine learning. We want to provide you with the practical skills needed to get learning algorithms to work in different settings. Instead of math and theory of machine learning, we will spend more time on the practical, hands-on skills (and dirty tricks) to get this stuff to work well on an application. We will focus on supervised and unsupervised machine learning and cover the essential steps from data science to build the applied machine learning workflow.

A typical workflow in applied machine learning applications consists of answering a series of questions that can be summarized in the following five steps:

1. **Data and problem definition:** The first step is to ask interesting questions. What is the problem you are trying solve? Why is it important? Which format of result answers your question? Is this a simple yes/no answer? Do you need to pick one of the available questions?

2. **Data collection:** Once you have a problem to tackle, you will need the data. Ask yourself what kind of data will help you answer the question. Can you get the data from the available sources? Will you have to combine multiple sources? Do you have to generate the data? Are there any sampling biases? How much data will be required?
3. **Data preprocessing:** The first data preprocessing task is data cleaning. For example, filling missing values, smoothing noisy data, removing outliers, and resolving inconsistencies. This is usually followed by integration of multiple data sources and data transformation to a specific range (normalization), to value bins (discretized intervals), and to reduce the number of dimensions.
4. **Data analysis and modeling with unsupervised and supervised learning:** Data analysis and modeling includes unsupervised and supervised machine learning, statistical inference, and prediction. A wide variety of machine learning algorithms are available, including k-nearest neighbors, naïve Bayes, decision trees, support vector machines, logistic regression, k-means, and so on. The choice of method to be deployed depends on the problem definition discussed in the first step and the type of collected data. The final product of this step is a model inferred from the data.
5. **Evaluation:** The last step is devoted to model assessment. The main issue models built with machine learning face is how well they model the underlying data—if a model is too specific, that is, it **overfits** to the data used for training, it is quite possible that it will not perform well on a new data. The model can be too generic, meaning that it **underfits** the training data. For example, when asked how the weather is in California, it always answers sunny, which is indeed correct most of the time. However, such a model is not really useful for making valid predictions. The goal of this step is to correctly evaluate the model and make sure it will work on new data as well. Evaluation methods include separate test and train set, cross-validation, and leave-one-out validation.

In the following sections, we will take a closer look at each of the steps. We will try to understand the type of questions we must answer during the applied machine learning workflow and also look at the accompanying concepts of data analysis and evaluation.

Data and problem definition

Data is simply a collection of measurements in the form of numbers, words, measurements, observations, descriptions of things, images, and so on.

Measurement scales

The most common way to represent the data is using a set of attribute-value pairs. Consider the following example:

```
Bob = {
    height: 185cm,
    eye color: blue,
    hobbies: climbing, sky diving
}
```

For example, Bob has attributes named `height`, `eye color`, and `hobbies` with values `185cm`, `blue`, `climbing, sky diving`, respectively.

A set of data can be simply presented as a table, where columns correspond to attributes or features and rows correspond to particular data examples or instances. In supervised machine learning, the attribute whose value we want to predict the outcome, Y , from the values of the other attributes, X , is denoted as class or the target variable, as follows:

Name	Height [cm]	Eye color	Hobbies
Bob	185.0	Blue	Climbing, sky diving
Anna	163.0	Brown	Reading
...

The first thing we notice is how varying the attribute values are. For instance, `height` is a number, `eye color` is text, and `hobbies` are a list. To gain a better understanding of the value types, let's take a closer look at the different types of data or measurement scales. Stevens (1946) defined the following four scales with increasingly more expressive properties:

- **Nominal data** are mutually exclusive, but not ordered. Their examples include eye color, martial status, type of car owned, and so on.
- **Ordinal data** correspond to categories where order matters, but not the difference between the values, such as pain level, student letter grade, service quality rating, IMDB movie rating, and so on.
- **Interval data** where the difference between two values is meaningful, but there is no concept of zero. For instance, standardized exam score, temperature in Fahrenheit, and so on.

- **Ratio data** has all the properties of an interval variable and also a clear definition of zero; when the variable equals to zero, there is none of this variable. Variables such as height, age, stock price, and weekly food spending are ratio variables.

Why should we care about measurement scales? Well, machine learning heavily depends on the statistical properties of the data; hence, we should be aware of the limitations each data type possesses. Some machine learning algorithms can only be applied to a subset of measurement scales.

The following table summarizes the main operations and statistics properties for each of the measurement types:

Property	Nominal	Ordinal	Interval	Ratio
Frequency of distribution	✓	✓	✓	✓
Mode and median		✓	✓	✓
Order of values is known		✓	✓	✓
Can quantify difference between each value			✓	✓
Can add or subtract values			✓	✓
Can multiply and divide values				✓
Has true zero				✓

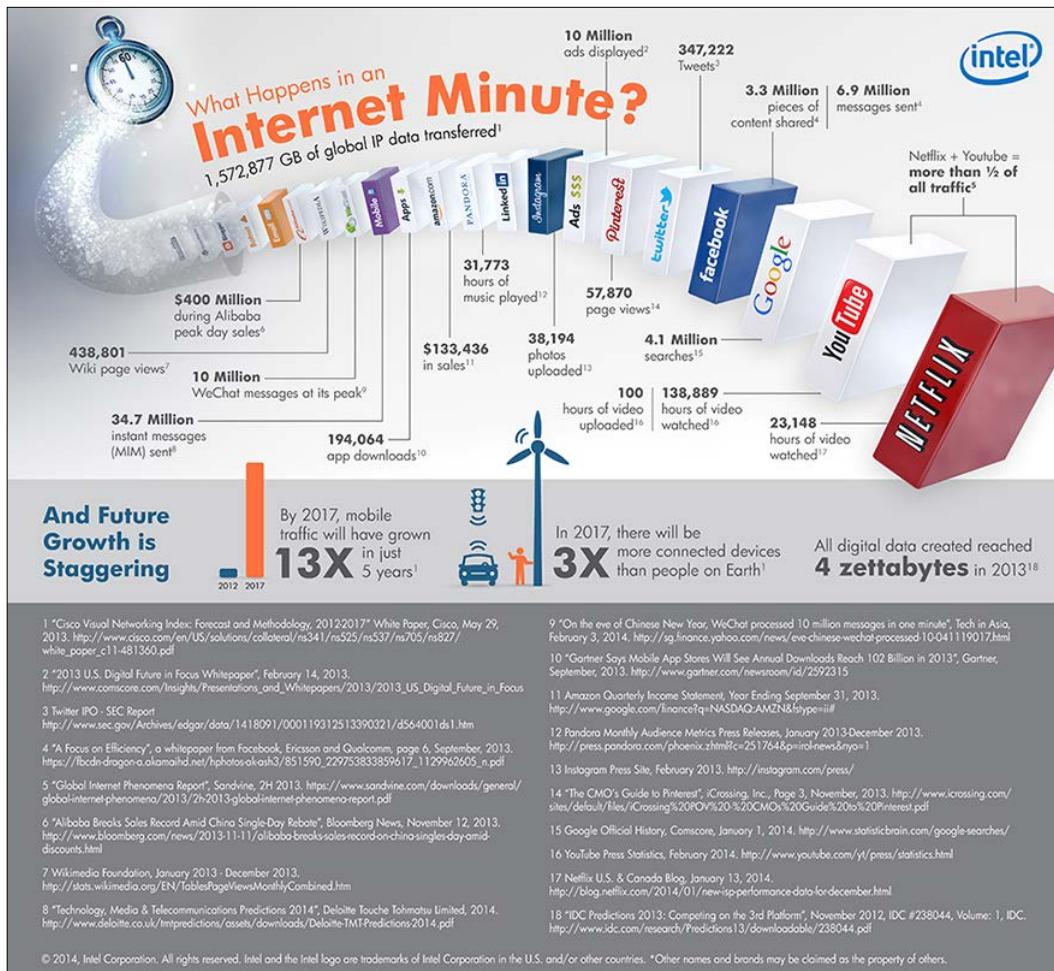
Furthermore, nominal and ordinal data correspond to discrete values, while interval and ratio data can correspond to continuous values as well. In supervised learning, the measurement scale of the attribute values that we want to predict dictates the kind of machine algorithm that can be used. For instance, predicting discrete values from a limited list is called classification and can be achieved using decision trees; while predicting continuous values is called regression, which can be achieved using model trees.

Data collection

So, where does the data come from? We have two choices: observe the data from existing sources or generate the data via surveys, simulations, and experiments. Let's take a closer look at both the approaches.

Find or observe data

Data can be found or observed at many places. An obvious data source is the Internet. Intel (2013) presented the following iconographic, showing the massive amount of data collected by different Internet services. In 2013, digital devices created four zettabytes ($10^{21} = \text{billion terabytes}$) of data. In 2017, it is expected that the number of connected devices will reach three times the number of people on earth; hence, the amount of data generated and collected will increase even further:



To get the data from the Internet, there are multiple options, as shown in the following:

- Bulk downloads from websites such as Wikipedia, IMDb, and Million Song database.
- Accessing the data through API (NY Times, Twitter, Facebook, Foursquare).
- Web scraping – It is OK to scrape public, non-sensitive, and anonymized data. Be sure to check terms of conditions and to fully reference information.

The main drawbacks of found data are that it takes time and space to accumulate the data; they cover only what happened, for instance, intentions, motivations, or internal motivations are not collected. Finally, such data might be noisy, incomplete, inconsistent, and may even change over time.

Another option is to collect measurements from sensors such as inertial and location sensors in mobile devices, environmental sensors, and software agents monitoring key performance indicators.

Generate data

An alternative approach is to generate the data by yourself, for example, with a survey. In survey design, we have to pay attention to data sampling, that is, who are the respondents answering the survey. We only get data from the respondents who are accessible and willing to respond. Also, respondents can provide answers that are in line with their self-image and researcher's expectations.

Next, the data can be collected with simulations, where a domain expert specifies behavior model of users at a micro level. For instance, crowd simulation requires specifying how different types of users will behave in crowd, for example, following the crowd, looking for an escape, and so on. The simulation can be then run under different conditions to see what happens (Tsai et al. 2011). Simulations are appropriate for studying macro phenomena and emergent behavior; however, they are typically hard to validate empirically.

Furthermore, you can design experiments to thoroughly cover all the possible outcomes, where you keep all the variables constant and only manipulate one variable at a time. This is the most costly approach, but usually provides the best quality of data.

Sampling traps

Data collection may involve many traps. To demonstrate one, let me share a story. There is supposed to be a global, unwritten rule for sending regular mail between students for free. If you write *student to student* to the place where the stamp should be, the mail is delivered to the recipient for free. Now, suppose Jacob sends a set of postcards to Emma, and given that Emma indeed receives some of the postcards, she concludes that all the postcards are delivered and that the rule indeed holds true. Emma reasons that as she received the postcards, all the postcards are delivered. However, she does not possess the information about the postcards that were sent by Jacob, but were undelivered; hence, she is unable to account this into her inference. What Emma experienced is **survivorship bias**, that is, she drew the conclusion based on the survived data only. For your information, the postcards that are being sent with *student to student* stamp get a circled black letter *T* stamp on them, which means *postage is due* and that receiver should pay it, including a small fine. However, mail services often have higher costs on applying such fee and hence do not do it (Magalhães, 2010).

Another example is a study, which found that the profession with the lowest average age of death was student. Being a student does not cause you to die at an early age, being a student means you are young. This is what makes the average of those that die so low (Gelman and Nolan, 2002).

Furthermore, a study that found that only 1.5% of drivers in accidents reported they were using a cell phone, whereas 10.9% reported another occupant in the car distracted them. Can we conclude that using a cell phone is safer than speaking with another occupant (Uts, 2003)? To answer this question, we need to know the prevalence of the cell phone use. It is likely that a higher number of people talked to another occupant in the car while driving than talking on the cell during the period when the data was collected.

Data pre-processing

The goal of data pre-processing tasks is to prepare the data for a machine learning algorithm in the best possible way as not all algorithms are capable of addressing issues with missing data, extra attributes, or denormalized values.

Data cleaning

Data cleaning, also known as data cleansing or data scrubbing, is the process of the following:

- Identifying inaccurate, incomplete, irrelevant, or corrupted data to remove it from further processing

- Parsing data, extracting information of interest, or validating whether a string of data is in an acceptable format
- Transforming data into a common encoding format, for example, utf-8 or int32, time scale, or normalized range
- Transforming data into a common data schema, for instance, if we collect temperature measurements from different types of sensors, we might want them to have the same structure

Now, let's look at some more concrete pre-processing steps.

Fill missing values

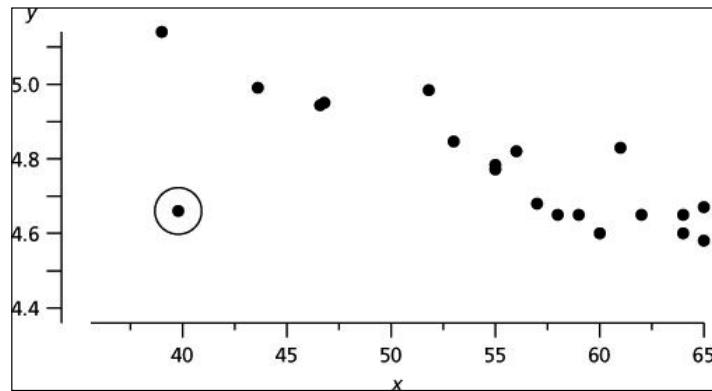
Machine learning algorithms generally do not work well with missing values. Rare exceptions include decision trees, naïve Bayes classifier, and some rule-based learners. It is very important to understand why a value is missing. It can be missing due to many reasons such as random error, systematic error, and sensor noise. Once we identified the reason, there are multiple ways to deal with the missing values, as shown in the following list:

- **Remove the instance:** If there is enough data, and only a couple of non-relevant instances have some missing values, then it is safe to remove these instances.
- **Remove the attribute:** Removing an attribute makes sense when most of the values are missing, values are constant, or attribute is strongly correlated with another attribute.
- **Assign a special value N/A:** Sometimes a value is missing due to valid reasons such as the value is out of scope discrete attribute value is not defined, or it is not possible to obtain or measure the value, which can be an indicator as well. For example, a person never rates a movie, so his rating on this movie is nonexistent.
- **Take the average attribute value:** In case we have a limited number of instances, we might not be able to afford removing instances or attributes. In that case, we can estimate the missing values, for example, by assigning the average attribute value or the average value over similar instances.
- **Predict the value from other attributes:** Predict the value from the previous entries if the attribute possesses time dependencies.

As we have seen, the value can be missing for many reasons, and hence, it is important to understand why the value is missing, absent, or corrupted.

Remove outliers

Outliers in data are values that are unlike any other values in the series and affect all learning methods to various degrees. These can be extreme values, which could be detected with confidence intervals and removed by threshold. The best approach is to visualize the data and inspect the visualization to detect irregularities. An example is shown in the following diagram. Visualization applies to low-dimensional data only:



Data transformation

Data transformation techniques tame the dataset to a format that a machine learning algorithm expects as an input, and may even help the algorithm to learn faster and achieve better performance. Standardization, for instance, assumes that data follows Gaussian distribution and transforms the values in such a way that the mean value is zero and the deviation is 1, as follows:

$$X = \frac{X - \text{mean}(X)}{\text{st.dev}}$$

Normalization, on the other hand, scales the values of attributes to a small, specified range, usually between 0 and 1:

$$X = \frac{X - \text{min}}{\text{max} - \text{min}}$$

Many machine learning toolboxes automatically normalize and standardize the data for you.

The last transformation technique is discretization, which divides the range of a continuous attribute into intervals. Why should we care? Some algorithms, such as decision trees and naïve Bayes prefer discrete attributes. The most common ways to select the intervals are shown in the following:

- **Equal width:** The interval of continuous variable is divided into k equal-width intervals
- **Equal frequency:** Suppose there are N instances, each of the k intervals contains approximately N/k instances
- **Min entropy:** The approach recursively splits the intervals until the entropy, which measures disorder, decreases more than the entropy increase, introduced by the interval split (Fayyad and Irani, 1993)

The first two methods require us to specify the number of intervals, while the last method sets the number of intervals automatically; however, it requires the class variable, which means, it won't work for unsupervised machine learning tasks.

Data reduction

Data reduction deals with abundant attributes and instances. The number of attributes corresponds to the number of dimensions in our dataset. Dimensions with low prediction power do not only contribute very little to the overall model, but also cause a lot of harm. For instance, an attribute with random values can introduce some random patterns that will be picked up by a machine learning algorithm.

To deal with this problem, the first set of techniques removes such attributes, or in other words, selects the most promising ones. This process is known as feature selection or attribute selection and includes methods such as ReliefF, information gain, and Gini index. These methods are mainly focused on discrete attributes.

Another set of tools, focused on continuous attributes, transforms the dataset from the original dimensions into a lower-dimensional space. For example, if we have a set of points in three-dimensional space, we can make a projection into a two-dimensional space. Some information is lost, but in case the third dimension is irrelevant, we don't lose much as the data structure and relationships are almost perfectly preserved. This can be performed by the following methods:

- **Singular value decomposition (SVD)**
- **Principal Component Analysis (PCA)**
- Neural nets auto encoders

The second problem in data reduction is related to too many instances; for example, they can be duplicates or coming from a very frequent data stream. The main idea is to select a subset of instances in such a way that distribution of the selected data still resembles the original data distribution, and more importantly, the observed process. Techniques to reduce the number of instances involve random data sampling, stratification, and others. Once the data is prepared, we can start with the data analysis and modeling.

Unsupervised learning

Unsupervised learning is about analyzing the data and discovering hidden structures in unlabeled data. As no notion of the right labels is given, there is also no error measure to evaluate a learned model; however, unsupervised learning is an extremely powerful tool. Have you ever wondered how Amazon can predict what books you'll like? How Netflix knows what you want to watch before you do? The answer can be found in unsupervised learning. The following is one such example.

Find similar items

Many problems can be formulated as finding similar sets of elements, for example, customers who purchased similar products, web pages with similar content, images with similar objects, users who visited similar websites, and so on.

Two items are considered similar if they are a *small distance* apart. The main questions are how each item is represented and how is the distance between the items defined. There are two main classes of distance measures: Euclidean distances and non-Euclidean distances.

Euclidean distances

In the Euclidean space, with the n dimension, the distance between two elements is based on the locations of the elements in such a space, which is expressed as **p-norm distance**. Two commonly used distance measures are L_2 and L_1 norm distances.

L_2 norm, also known as Euclidean distance, is the most frequently applied distance measure that measures how far apart two items in a two-dimensional space are. It is calculated as a square root of the sum of the squares of the differences between elements a and b in each dimension, as follows:

$$L_2 \text{ norm } d(a, b) = \sqrt{\sum_{i=1}^n (a_i - b_i)^2}$$

L1 norm, also known as Manhattan distance, city block distance, and taxicab norm, simply sums the absolute differences in each dimension, as follows:

$$L_1 \text{norm } d(a, b) = \sum_{i=1}^n |a_i - b_i|$$

Non-Euclidean distances

A non-Euclidean distance is based on the properties of the elements, but not on their location in space. Some well-known distances are Jaccard distance, cosine distance, edit distance, and Hamming distance.

Jaccard distance is used to compute the distance between two sets. First, we compute the Jaccard similarity of two sets as the size of their intersection divided by the size of their union, as follows:

$$\text{sim}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

The Jaccard distance is then defined as 1 minus Jaccard similarity, as shown in the following:

$$d(A, B) = 1 - \text{sim}(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$$

Cosine distance between two vectors focuses on the orientation and not magnitude, therefore, two vectors with the same orientation have cosine similarity 1, while two perpendicular vectors have cosine similarity 0. Suppose we have two multidimensional points, think of a point as a vector from origin $(0, 0, \dots, 0)$ to its location. Two vectors make an angle, whose cosine distance is a normalized dot-product of the vectors, as follows:

$$d(A, B) = \arccos \frac{A \cdot B}{\|A\| \|B\|}$$

Cosine distance is commonly used in a high-dimensional feature space; for instance, in text mining, where a text document represents an instance, features that correspond to different words, and their values corresponds to the number of times the word appears in the document. By computing cosine similarity, we can measure how likely two documents match in describing similar content.

Edit distance makes sense when we compare two strings. The distance between the $a=a_1a_2a_3\dots a_n$ and $b=b_1b_2b_3\dots b_n$ strings is the smallest number of the insert/delete operation of single characters required to convert the string from a to b . For example, $a = abcd$ and $b = abbd$. To convert a to b , we have to delete the second b and insert c in its place. No smallest number of operations would convert a to b , thus the distance is $d(a, b)=2$.

Hamming distance compares two vectors of the same size and counts the number of dimensions in which they differ. In other words, it measures the number of substitutions required to convert one vector into another.

There are many distance measures focusing on various properties, for instance, correlation measures the linear relationship between two elements: **Mahalanobis distance** that measures the distance between a point and distribution of other points and **SimRank**, which is based on graph theory, measures similarity of the structure in which elements occur, and so on. As you can already imagine selecting and designing the right similarity measure for your problem is more than half of the battle. An impressive overview and evaluation of similarity measures is collected in *Chapter 2, Similarity and Dissimilarity Measures* in the book *Image Registration: Principles, Tools and Methods* by A. A. Goshtasby (2012).

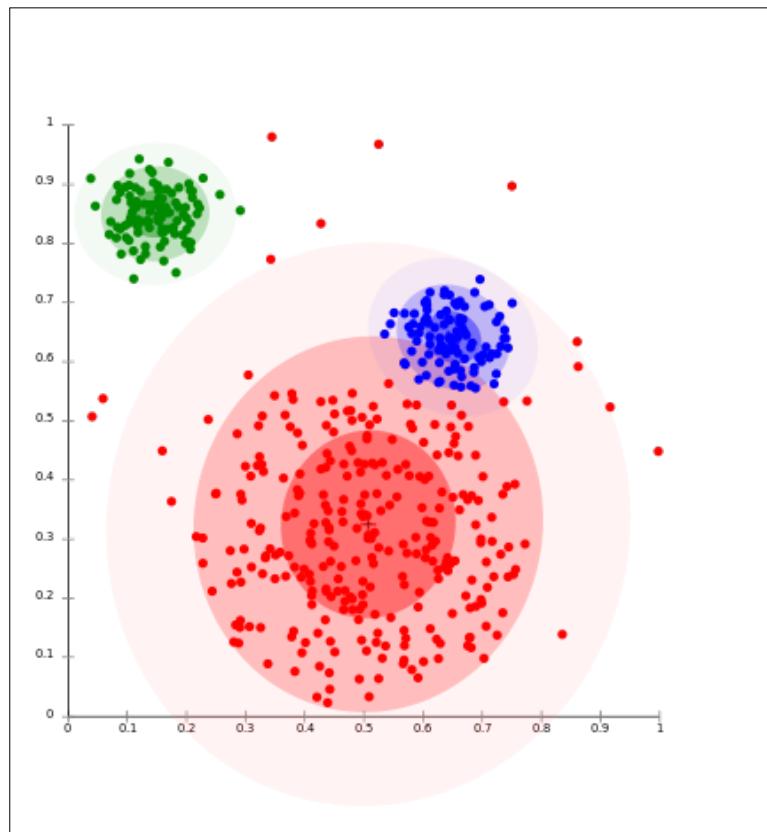
The curse of dimensionality

The curse of dimensionality refers to a situation where we have a large number of features, often hundreds or thousands, which lead to an extremely large space with sparse data and, consequently, to distance anomalies. For instance, in high dimensions, almost all pairs of points are equally distant from each other; in fact, almost all the pairs have distance close to the average distance. Another manifestation of the curse is that any two vectors are almost orthogonal, which means all the angles are close to 90 degrees. This practically makes any distance measure useless.

A cure for the curse of dimensionality might be found in one of the data reduction techniques, where we want to reduce the number of features; for instance, we can run a feature selection algorithm such as ReliefF or feature extraction/reduction algorithm such as PCA.

Clustering

Clustering is a technique for grouping similar instances into clusters according to some distance measure. The main idea is to put instances that are similar (that is, close to each other) into the same cluster, while keeping the dissimilar points (that is, the ones further apart from each other) in different clusters. An example of how clusters might look is shown in the following diagram:



The clustering algorithms follow two fundamentally different approaches. The first is a **hierarchical** or **agglomerative** approach that first considers each point as its own cluster, and then iteratively merges the most similar clusters together. It stops when further merging reaches a predefined number of clusters or if the clusters to be merged are spread over a large region.

The other approach is based on point assignment. First, initial cluster centers (that is, centroids) are estimated—for instance, randomly—and then, each point is assigned to the closest cluster, until all the points are assigned. The most well-known algorithm in this group is **k-means clustering**.

The k-means clustering picks initial cluster centers either as points that are as far as possible from one another or (hierarchically) clusters a sample of data and picks a point that is the closest to the center of each of the k clusters.

Supervised learning

Supervised learning is the key concept behind amazing things such as voice recognition, e-mail spam filtering, face recognition in photos, and detecting credit card frauds. More formally, given a set D of learning examples described with features, X , the goal of supervised learning is to find a function that predicts a target variable, Y . The function f that describes the relation between features X and class Y is called a model:

$$f(X) \rightarrow Y$$

The general structure of supervised learning algorithms is defined by the following decisions (Hand et al., 2001):

1. Define the task.
2. Decide on the machine learning algorithm, which introduces specific inductive bias, that is, apriori assumptions that it makes regarding the target concept.
3. Decide on the **score** or **cost** function, for instance, information gain, root mean square error, and so on.
4. Decide on the optimization/search method to optimize the score function.
5. Find a function that describes the relation between X and Y .

Many decisions are already made for us by the type of the task and dataset that we have. In the following sections, we will take a closer look at the classification and regression methods and the corresponding score functions.

Classification

Classification can be applied when we deal with a discrete class, and the goal is to predict one of the mutually-exclusive values in the target variable. An example would be credit scoring, where the final prediction is whether the person is credit liable or not. The most popular algorithms include decision tree, naïve Bayes classifier, support vector machines, neural networks, and ensemble methods.

Decision tree learning

Decision tree learning builds a classification tree, where each node corresponds to one of the attributes, edges correspond to a possible value (or intervals) of the attribute from which the node originates, and each leaf corresponds to a class label. A decision tree can be used to visually and explicitly represent the prediction model, which makes it a very transparent (white box) classifier. Notable algorithms are ID3 and C4.5, although many alternative implementations and improvements (for example, J48 in Weka) exist.

Probabilistic classifiers

Given a set of attribute values, a probabilistic classifier is able to predict a distribution over a set of classes, rather than an exact class. This can be used as a degree of certainty, that is, how sure the classifier is in its prediction. The most basic classifier is naïve Bayes, which happens to be the optimal classifier if, and only if, the attributes are conditionally independent. Unfortunately, this is extremely rare in practice.

There is really an enormous subfield denoted as probabilistic graphical models, comprising of hundreds of algorithms; for example, Bayesian network, dynamic Bayesian networks, hidden Markov models, and conditional random fields that can handle not only specific relationships between attributes, but also temporal dependencies. Karkera (2014) wrote an excellent introductory book on this topic, *Building Probabilistic Graphical Models with Python*, while Koller and Friedman (2009) published a comprehensive theory bible, *Probabilistic Graphical Models*.

Kernel methods

Any linear model can be turned into a non-linear model by applying the kernel trick to the model—replacing its features (predictors) by a kernel function. In other words, the kernel implicitly transforms our dataset into higher dimensions. The kernel trick leverages the fact that it is often easier to separate the instances in more dimensions. Algorithms capable of operating with kernels include the kernel perceptron, **Support Vector Machines (SVM)**, Gaussian processes, PCA, canonical correlation analysis, ridge regression, spectral clustering, linear adaptive filters, and many others.

Artificial neural networks

Artificial neural networks are inspired by the structure of biological neural networks and are capable of machine learning, as well as pattern recognition. They are commonly used for both regression and classification problems, comprising a wide variety of algorithms and variations for all manner of problem types. Some popular classification methods are **perceptron**, **restricted Boltzmann machine (RBM)**, and **deep belief networks**.

Ensemble learning

Ensemble methods compose of a set of diverse weaker models to obtain better predictive performance. The individual models are trained separately and their predictions are then combined in some way to make the overall prediction. Ensembles, hence, contain multiple ways of modeling the data, which hopefully leads to better results. This is a very powerful class of techniques, and as such, it is very popular; for instance, boosting, bagging, AdaBoost, and Random Forest. The main differences among them are the type of weak learners that are to be combined and the ways in which to combine them.

Evaluating classification

Is our classifier doing well? Is this better than the other one? In classification, we count how many times we classify something right and wrong. Suppose there are two possible classification labels—yes and no—then there are four possible outcomes, as shown in the next figure:

- True positive—hit: This indicates a *yes* instance correctly predicted as *yes*
- True negative—correct rejection: This indicates a *no* instance correctly predicted as *no*
- False positive—false alarm: This indicates a *no* instance predicted as *yes*
- False negative—miss: This indicates a *yes* instance predicted as *no*

		Predicted as positive?	
		Yes	No
Really positive?	Yes	TP – true positive	FN – false negative
	No	FP – false positive	TN – true negative

The basic two performance measures of a classifier are classification error and accuracy, as shown in the following image:

$$\text{Classification error} = \frac{\text{errors}}{\text{totals}} = \frac{FP + FN}{FP + FN + TP + TN}$$

$$\text{Classification accuracy} = 1 - \text{error} = \frac{\text{correct}}{\text{totals}} = \frac{TP + TN}{FP + FN + TP + TN}$$

The main problem with these two measures is that they cannot handle unbalanced classes. Classifying whether a credit card transaction is an abuse or not is an example of a problem with unbalanced classes, there are 99.99% normal transactions and just a tiny percentage of abuses. Classifier that says that every transaction is a normal one is 99.99% accurate, but we are mainly interested in those few classifications that occur very rarely.

Precision and recall

The solution is to use measures that don't involve TN (correct rejections). Two such measures are as follows:

- *Precision*: This is the proportion of positive examples correctly predicted as positive (TP) out of all examples predicted as positive ($TP + FP$):

$$Precision = \frac{TP}{TP + FP}$$

- *Recall*: This is the proportion of positives examples correctly predicted as positive (TP) out of all positive examples ($TP + FN$):

$$Recall = \frac{TP}{TP + FN}$$

It is common to combine the two and report the *F-measure*, which considers both precision and recall to calculate the score as a weighted average, where the score reaches its best value at 1 and worst at 0, as follows:

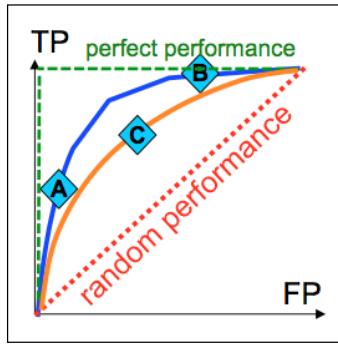
$$F\text{-measure} = \frac{2 * Precision * Recall}{Precision + Recall}$$

Roc curves

Most classification algorithms return a classification confidence denoted as $f(X)$, which is, in turn, used to calculate the prediction. Following the credit card abuse example, a rule might look similar to the following:

$$F(X) = \begin{cases} \text{abuse, if } f(X) > \text{threshold} \\ \text{not abuse, else} \end{cases}$$

The threshold determines the error rate and the true positive rate. The outcomes for all the possible threshold values can be plotted as a **Receiver Operating Characteristics (ROC)** as shown in the following diagram:



A random predictor is plotted with a red dashed line and a perfect predictor is plotted with a green dashed line. To compare whether the A classifier is better than C, we compare the area under the curve.

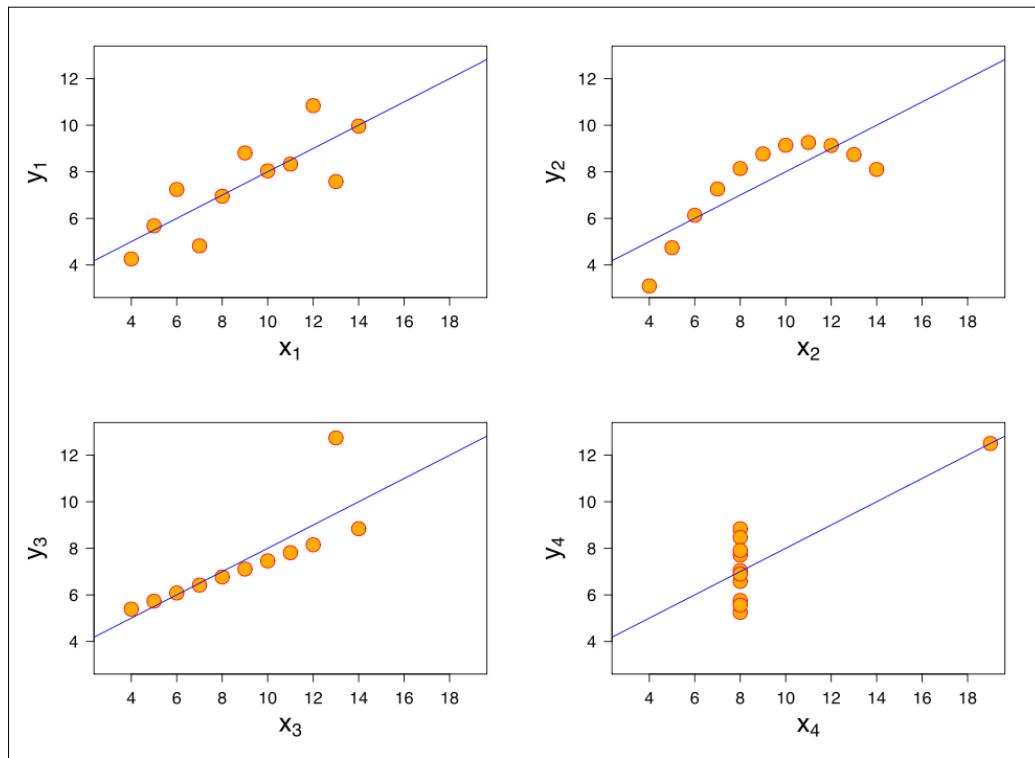
Most of the toolboxes provide all of the previous measures out-of-the-box.

Regression

Regression deals with continuous target variable, unlike classification, which works with a discrete target variable. For example, in order to forecast the outside temperature of the following few days, we will use regression; while classification will be used to predict whether it will rain or not. Generally speaking, regression is a process that estimates the relationship among features, that is, how varying a feature changes the target variable.

Linear regression

The most basic regression model assumes linear dependency between features and target variable. The model is often fitted using least squares approach, that is, the best model minimizes the squares of the errors. In many cases, linear regression is not able to model complex relations, for example, the next figure shows four different sets of points having the same linear regression line: the upper-left model captures the general trend and can be considered as a proper model, the bottom-left model fits points much better, except an outlier – this should be carefully checked – and the upper and lower-right side linear models completely miss the underlying structure of the data and cannot be considered as proper models.



Evaluating regression

In regression, we predict numbers Y from inputs X and the predictions are usually wrong and not exact. The main question that we ask is by how much? In other words, we want to measure the distance between the predicted and true values.

Mean squared error

Mean squared error is an average of the squared difference between the predicted and true values, as follows:

$$MSE(X, Y) = \sqrt{\frac{1}{n} \sum_{i=1}^n (f(X_i) - Y_i)^2}$$

The measure is very sensitive to the outliers, for example, 99 exact predictions and *one prediction off by 10* is scored the same as all predictions wrong by 1. Moreover, the measure is sensitive to the mean. Therefore, relative squared error, which compares the MSE of our predictor to the MSE of the mean predictor (which always predicts the mean value) is often used instead.

Mean absolute error

Mean absolute error is an average of the absolute difference between the predicted and the true values, as follows:

$$MAS(X, Y) = \frac{1}{n} \sum_{i=1}^n |f(X_i) - Y_i|$$

The MAS is less sensitive to the outliers, but it is also sensitive to the mean and scale.

Correlation coefficient

Correlation coefficient compares the average of prediction relative to the mean multiplied by training values relative to the mean. If the number is negative, it means weak correlation, positive number means strong correlation, and zero means no correlation. The correlation between true values X and predictions Y is defined as follows:

$$CC_{XY} = \frac{\sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^n (X_i - \bar{X})^2} \sqrt{\sum_{i=1}^n (Y_i - \bar{Y})^2}}$$

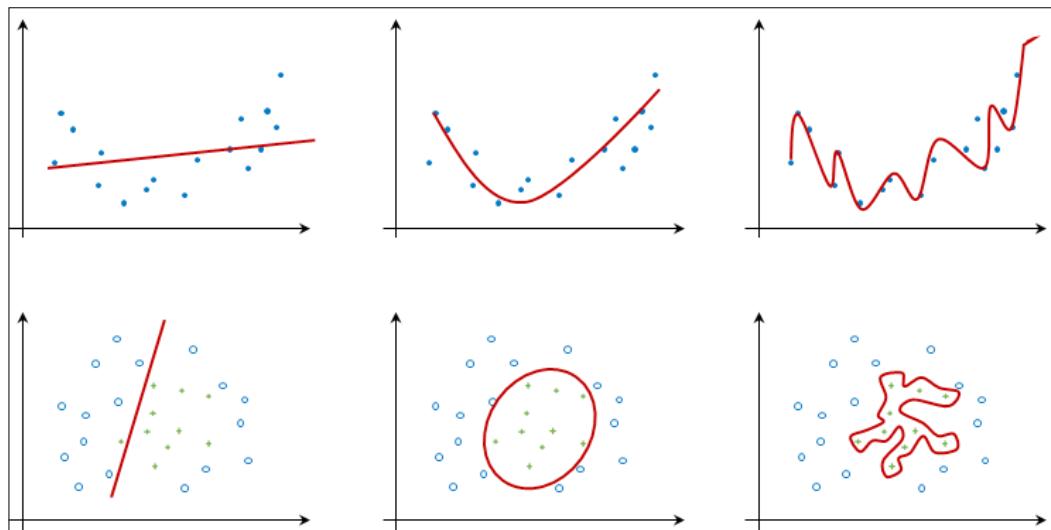
The CC measure is completely insensitive to the mean and scale, and less sensitive to the outliers. It is able to capture the relative ordering, which makes it useful to rank the tasks such as document relevance and gene expression.

Generalization and evaluation

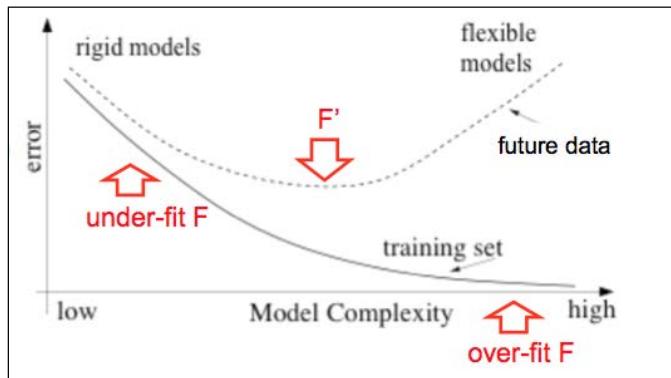
Once the model is built, how do we know it will perform on new data? Is this model any good? To answer these questions, we'll first look into the model generalization and then, see how to get an estimate of the model performance on new data.

Underfitting and overfitting

Predictor training can lead to models that are too complex or too simple. The model with low complexity (the leftmost models) can be as simple as predicting the most frequent or mean class value, while the model with high complexity (the rightmost models) can represent the training instances. Too rigid modes, which are shown on the left-hand side, cannot capture complex patterns; while too flexible models, shown on the right-hand side, fit to the noise in the training data. The main challenge is to select the appropriate learning algorithm and its parameters, so that the learned model will perform well on the new data (for example, the middle column):



The following figure shows how the error in the training set decreases with the model complexity. Simple rigid models underfit the data and have large errors. As model complexity increases, it describes the underlying structure of the training data better, and consequentially, the error decreases. If the model is too complex, it overfits the training data and its prediction error increases again:



Depending on the task complexity and data availability, we want to tune our classifiers towards less or more complex structures. Most learning algorithms allow such tuning, as follows:

- **Regression:** This is the order of the polynomial
- **Naive Bayes:** This is the number of the attributes
- **Decision trees:** This is the number of nodes in the tree, pruning confidence
- **k-nearest neighbors:** This is the number of neighbors, distance-based neighbor weights
- **SVM:** This is the kernel type, cost parameter
- **Neural network:** This is the number of neurons and hidden layers

With tuning, we want to minimize the generalization error, that is, how well the classifier performs on future data. Unfortunately, we can never compute the true generalization error; however, we can estimate it. Nevertheless, if the model performs well on the training data, but performance is much worse on the test data, the model most likely overfits.

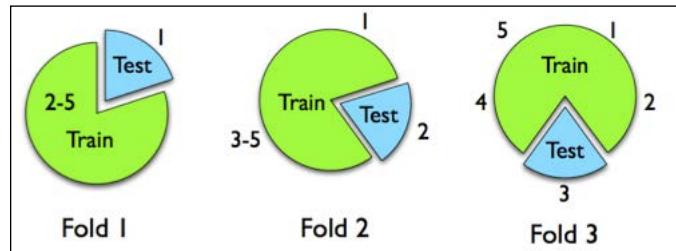
Train and test sets

To estimate the generalization error, we split our data into two parts: training data and testing data. A general rule of thumb is to split them in the *training:testing* ratio, that is, 70:30. We first train the predictor on the training data, then predict the values for the test data, and finally, compute the error – the difference between the predicted and the true values. This gives us an estimate of the true generalization error.

The estimation is based on the following two assumptions: first, we assume that the test set is an unbiased sample from our dataset; and second, we assume that the actual new data will reassemble the distribution as our training and testing examples. The first assumption can be mitigated by cross-validation and stratification. Also, if it is scarce one can't afford to leave out a considerable amount of data for separate test set as learning algorithms do not perform well if they don't receive enough data. In such cases, cross-validation is used instead.

Cross-validation

Cross-validation splits the dataset into k sets of approximately the same size, for example, to five sets as shown in the following figure. First, we use the 2-5 sets for learning and set 1 for training. We then repeat the procedure five times, leaving out one set at a time for testing, and average the error over the five repetitions.



This way, we used all the data for learning and testing as well, while we avoided using the same data to train and test a model.

Leave-one-out validation

An extreme example of cross-validation is the leave-one-out validation. In this case, the number of folds is equal to the number of instances; we learn on all but one instance, and then test the model on the omitted instance. We repeat this for all instances, so that each instance is used exactly once for the validation. This approach is recommended when we have a limited set of learning examples, for example, less than 50.

Stratification

Stratification is a procedure to select a subset of instances in such a way that each fold roughly contains the same proportion of class values. When a class is continuous, the folds are selected so that the mean response value is approximately equal in all the folds. Stratification can be applied along with cross-validation or separate training and test sets.

Summary

In this chapter, we refreshed the machine learning basics. We revisited the workflow of applied machine learning and clarified the main tasks, methods, and algorithms. In the next chapter, we will review the kind of Java libraries that are available and the kind of tasks they can perform.

2

Java Libraries and Platforms for Machine Learning

Implementing machine learning algorithms by yourself is probably the best way to learn machine learning, but you can progress much faster if you step on the shoulders of the giants and leverage one of the existing open source libraries.

This chapter reviews various libraries and platforms for machine learning in Java. The goal is to understand what each library brings to the table and what kind of problems is it able to solve?

In this chapter, we will cover the following topics:

- The requirement of Java to implement a machine learning app
- Weka, a general purpose machine learning platform
- Java machine learning library, a collection of machine learning algorithms
- Apache Mahout, a scalable machine learning platform
- Apache Spark, a distributed machine learning library
- Deeplearning4j, a deep learning library
- MALLET, a text mining library

We'll also discuss how to architect the complete machine learning app stack for both single-machine and big data apps using these libraries with other components.

The need for Java

New machine learning algorithms are often first scripted at university labs, gluing together several languages such as shell scripting, Python, R, MATLAB, Java, Scala, or C++ to prove a new concept and theoretically analyze its properties. An algorithm might then take a long path of refactoring before it lands in a library with standardized input/output and interfaces. While Python, R, and MATLAB are quite popular, they are mainly used for scripting, research, and experimenting. Java, on the other hand, is the de-facto enterprise language, which could be attributed to static typing, robust IDE support, good maintainability, as well as decent threading model, and high-performance concurrent data structure libraries. Moreover, there are already many Java libraries available for machine learning, which make it really convenient to interface them in existing Java applications and leverage powerful machine learning capabilities.

Machine learning libraries

There are over 70 Java-based open source machine learning projects listed on the MLOSS.org website and probably many more unlisted projects live at university servers, GitHub, or Bitbucket. In this section, we will review the major libraries and platforms, the kind of problems they can solve, the algorithms they support, and the kind of data they can work with.

Weka

Weka, which is short for **Waikato Environment for Knowledge Analysis**, is a machine learning library developed at the University of Waikato, New Zealand, and is probably the most well-known Java library. It is a general-purpose library that is able to solve a wide variety of machine learning tasks, such as classification, regression, and clustering. It features a rich graphical user interface, command-line interface, and Java API. You can check out Weka at <http://www.cs.waikato.ac.nz/ml/weka/>.

At the time of writing this book, Weka contains 267 algorithms in total: data pre-processing (82), attribute selection (33), classification and regression (133), clustering (12), and association rules mining (7). Graphical interfaces are well-suited for exploring your data, while Java API allows you to develop new machine learning schemes and use the algorithms in your applications.



Weka is distributed under **GNU General Public License (GNU GPL)**, which means that you can copy, distribute, and modify it as long as you track changes in source files and keep it under GNU GPL. You can even distribute it commercially, but you must disclose the source code or obtain a commercial license.

In addition to several supported file formats, Weka features its own default data format, ARFF, to describe data by attribute-data pairs. It consists of two parts. The first part contains header, which specifies all the attributes (that is, features) and their type; for instance, nominal, numeric, date, and string. The second part contains data, where each line corresponds to an instance. The last attribute in the header is implicitly considered as the target variable, missing data are marked with a question mark. For example, returning to the example from *Chapter 1, Applied Machine Learning Quick Start*, the Bob instance written in an ARFF file format would be as follows:

```
@RELATION person_dataset

@ATTRIBUTE `Name` STRING
@ATTRIBUTE `Height` NUMERIC
@ATTRIBUTE `Eye color`{blue, brown, green}
@ATTRIBUTE `Hobbies` STRING

@DATA
'Bob', 185.0, blue, 'climbing, sky diving'
'Anna', 163.0, brown, 'reading'
'Jane', 168.0, ?, ?
```

The file consists of three sections. The first section starts with the @RELATION <String> keyword, specifying the dataset name. The next section starts with the @ATTRIBUTE keyword, followed by the attribute name and type. The available types are STRING, NUMERIC, DATE, and a set of categorical values. The last attribute is implicitly assumed to be the target variable that we want to predict. The last section starts with the @DATA keyword, followed by one instance per line. Instance values are separated by comma and must follow the same order as attributes in the second section.

More Weka examples will be demonstrated in *Chapter 3, Basic Algorithms – Classification, Regression, and Clustering*, and *Chapter 4, Customer Relationship Prediction with Ensembles*.



To learn more about Weka, pick up a quick-start book, *Weka How-to* by Kaluza (Packt Publishing) to start coding or look into *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations* by Witten and Frank (Morgan Kaufmann Publishers) for theoretical background and in-depth explanations.

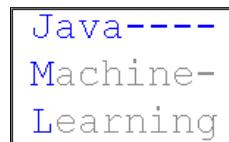
Weka's Java API is organized in the following top-level packages:

- `weka.associations`: These are data structures and algorithms for association rules learning, including **Apriori**, **predictive apriori**, **FilteredAssociator**, **FP-Growth**, **Generalized Sequential Patterns (GSP)**, **Hotspot**, and **Tertius**.
- `weka.classifiers`: These are supervised learning algorithms, evaluators, and data structures. The package is further split into the following components:
 - `weka.classifiers.bayes`: This implements Bayesian methods, including naive Bayes, Bayes net, Bayesian logistic regression, and so on
 - `weka.classifiers.evaluation`: These are supervised evaluation algorithms for nominal and numerical prediction, such as evaluation statistics, confusion matrix, ROC curve, and so on
 - `weka.classifiers.functions`: These are regression algorithms, including linear regression, isotonic regression, Gaussian processes, support vector machine, multilayer perceptron, voted perceptron, and others
 - `weka.classifiers.lazy`: These are instance-based algorithms such as k-nearest neighbors, K*, and lazy Bayesian rules

- weka.classifiers.meta: These are supervised learning meta-algorithms, including AdaBoost, bagging, additive regression, random committee, and so on
- weka.classifiers.mi: These are multiple-instance learning algorithms, such as citation k-nn, diverse density, MI AdaBoost, and others
- weka.classifiers.rules: These are decision tables and decision rules based on the separate-and-conquer approach, Ripper, Part, Prism, and so on
- weka.classifiers.trees: These are various decision trees algorithms, including ID3, C4.5, M5, functional tree, logistic tree, random forest, and so on
- weka.clusterers: These are clustering algorithms, including k-means, Clope, Cobweb, DBSCAN hierarchical clustering, and farthest.
- weka.core: These are various utility classes, data presentations, configuration files, and so on.
- weka.datagenerators: These are data generators for classification, regression, and clustering algorithms.
- weka.estimators: These are various data distribution estimators for discrete/nominal domains, conditional probability estimations, and so on.
- weka.experiment: These are a set of classes supporting necessary configuration, datasets, model setups, and statistics to run experiments.
- weka.filters: These are attribute-based and instance-based selection algorithms for both supervised and unsupervised data preprocessing.
- weka.gui: These are graphical interface implementing **explorer**, **experimenter**, and **knowledge flow** applications. Explorer allows you to investigate dataset, algorithms, as well as their parameters, and visualize dataset with scatter plots and other visualizations. Experimenter is used to design batches of experiment, but it can only be used for classification and regression problems. Knowledge flows implements a visual drag-and-drop user interface to build data flows, for example, load data, apply filter, build classifier, and evaluate.

Java machine learning

Java machine learning library, or Java-ML, is a collection of machine learning algorithms with a common interface for algorithms of the same type. It only features Java API, therefore, it is primarily aimed at software engineers and programmers. Java-ML contains algorithms for data preprocessing, feature selection, classification, and clustering. In addition, it features several Weka bridges to access Weka's algorithms directly through the Java-ML API. It can be downloaded from <http://java-ml.sourceforge.net>; where, the latest release was in 2012 (at the time of writing this book).



Java-ML is also a general-purpose machine learning library. Compared to Weka, it offers more consistent interfaces and implementations of recent algorithms that are not present in other packages, such as an extensive set of state-of-the-art similarity measures and feature-selection techniques, for example, dynamic time warping, random forest attribute evaluation, and so on. Java-ML is also available under the GNU GPL license.

Java-ML supports any type of file as long as it contains one data sample per line and the features are separated by a symbol such as comma, semi-colon, and tab.

The library is organized around the following top-level packages:

- `net.sf.javaml.classification`: These are classification algorithms, including naive Bayes, random forests, bagging, self-organizing maps, k-nearest neighbors, and so on
- `net.sf.javaml.clustering`: These are clustering algorithms such as k-means, self-organizing maps, spatial clustering, Cobweb, AQBC, and others
- `net.sf.javaml.core`: These are classes representing instances and datasets
- `net.sf.javaml.distance`: These are algorithms that measure instance distance and similarity, for example, **Chebyshev distance**, cosine distance/similarity, Euclidian distance, Jaccard distance/similarity, **Mahalanobis distance**, **Manhattan distance**, **Minkowski distance**, **Pearson correlation coefficient**, **Spearman's footrule distance**, **dynamic time wrapping (DTW)**, and so on

- `net.sf.javaml.featureselection`: These are algorithms for feature evaluation, scoring, selection, and ranking, for instance, gain ratio, ReliefF, Kullback-Liebler divergence, symmetrical uncertainty, and so on
- `net.sf.javaml.filter`: These are methods for manipulating instances by filtering, removing attributes, setting classes or attribute values, and so on
- `net.sf.javaml.matrix`: This implements in-memory or file-based array
- `net.sf.javaml.sampling`: This implements sampling algorithms to select a subset of dataset
- `net.sf.javaml.tools`: These are utility methods on dataset, instance manipulation, serialization, Weka API interface, and so on
- `net.sf.javaml.utils`: These are utility methods for algorithms, for example, statistics, math methods, contingency tables, and others

Apache Mahout

The Apache Mahout project aims to build a scalable machine learning library. It is built atop scalable, distributed architectures, such as Hadoop, using the MapReduce paradigm, which is an approach for processing and generating large datasets with a parallel, distributed algorithm using a cluster of servers.



Mahout features console interface and Java API to scalable algorithms for clustering, classification, and collaborative filtering. It is able to solve three business problems: item recommendation, for example, recommending items such as *people who liked this movie also liked...*; clustering, for example, of text documents into groups of topically-related documents; and classification, for example, learning which topic to assign to an unlabeled document.

Mahout is distributed under a commercially-friendly Apache License, which means that you can use it as long as you keep the Apache license included and display it in your program's copyright notice.

Mahout features the following libraries:

- `org.apache.mahout.cf.taste`: These are collaborative filtering algorithms based on user-based and item-based collaborative filtering and matrix factorization with ALS
- `org.apache.mahout.classifier`: These are in-memory and distributed implementations, including logistic regression, naive Bayes, random forest, **hidden Markov models (HMM)**, and multilayer perceptron
- `org.apache.mahout.clustering`: These are clustering algorithms such as canopy clustering, k-means, fuzzy k-means, streaming k-means, and spectral clustering
- `org.apache.mahout.common`: These are utility methods for algorithms, including distances, MapReduce operations, iterators, and so on
- `org.apache.mahout.driver`: This implements a general-purpose driver to run main methods of other classes
- `org.apache.mahout.ep`: This is the evolutionary optimization using the recorded-step mutation
- `org.apache.mahout.math`: These are various math utility methods and implementations in Hadoop
- `org.apache.mahout.vectorizer`: These are classes for data presentation, manipulation, and MapReduce jobs

Apache Spark

Apache Spark, or simply Spark, is a platform for large-scale data processing builds atop Hadoop, but, in contrast to Mahout, it is not tied to the MapReduce paradigm. Instead, it uses in-memory caches to extract a working set of data, process it, and repeat the query. This is reported to be up to ten times as fast as a Mahout implementation that works directly with disk-stored data. It can be grabbed from <https://spark.apache.org>.



There are many modules built atop Spark, for instance, **GraphX** for graph processing, **Spark Streaming** for processing real-time data streams, and **MLlib** for machine learning library featuring classification, regression, collaborative filtering, clustering, dimensionality reduction, and optimization.

Spark's MLlib can use a Hadoop-based data source, for example, **Hadoop Distributed File System (HDFS)** or HBase, as well as local files. The supported data types include the following:

- **Local vector** is stored on a single machine. Dense vectors are presented as an array of double-typed values, for example, $(2.0, 0.0, 1.0, 0.0)$; while sparse vector is presented by the size of the vector, an array of indices, and an array of values, for example, $[4, (0, 2), (2.0, 1.0)]$.
- **Labeled point** is used for supervised learning algorithms and consists of a local vector labeled with a double-typed class values. Label can be class index, binary outcome, or a list of multiple class indices (multiclass classification). For example, a labeled dense vector is presented as $[1.0, (2.0, 0.0, 1.0, 0.0)]$.
- **Local matrix** stores a dense matrix on a single machine. It is defined by matrix dimensions and a single double-array arranged in a column-major order.
- **Distributed matrix** operates on data stored in Spark's **Resilient Distributed Dataset (RDD)**, which represents a collection of elements that can be operated on in parallel. There are three presentations: row matrix, where each row is a local vector that can be stored on a single machine, row indices are meaningless; and indexed row matrix, which is similar to row matrix, but the row indices are meaningful, that is, rows can be identified and joins can be executed; and coordinate matrix, which is used when a row cannot be stored on a single machine and the matrix is very sparse.

Spark's MLlib API library provides interfaces to various learning algorithms and utilities as outlined in the following list:

- `org.apache.spark.mllib.classification`: These are binary and multiclass classification algorithms, including linear SVMs, logistic regression, decision trees, and naive Bayes
- `org.apache.spark.mllib.clustering`: These are k-means clustering
- `org.apache.spark.mllib.linalg`: These are data presentations, including dense vectors, sparse vectors, and matrices
- `org.apache.spark.mllib.optimization`: These are the various optimization algorithms used as low-level primitives in MLlib, including gradient descent, stochastic gradient descent, update schemes for distributed SGD, and limited-memory BFGS
- `org.apache.spark.mllib.recommendation`: These are model-based collaborative filtering implemented with alternating least squares matrix factorization

- `org.apache.spark.mllib.regression`: These are regression learning algorithms, such as linear least squares, decision trees, Lasso, and Ridge regression
- `org.apache.spark.mllib.stat`: These are statistical functions for samples in sparse or dense vector format to compute the mean, variance, minimum, maximum, counts, and nonzero counts
- `org.apache.spark.mllib.tree`: This implements classification and regression decision tree-learning algorithms
- `org.apache.spark.mllib.util`: These are a collection of methods to load, save, preprocess, generate, and validate the data

Deeplearning4j

Deeplearning4j, or DL4J, is a deep-learning library written in Java. It features a distributed as well as a single-machine deep-learning framework that includes and supports various neural network structures such as feedforward neural networks, **RBM**, convolutional neural nets, deep belief networks, autoencoders, and others. DL4J can solve distinct problems, such as identifying faces, voices, spam or e-commerce fraud.

Deeplearning4j is also distributed under Apache 2.0 license and can be downloaded from <http://deeplearning4j.org>. The library is organized as follows:

- `org.deeplearning4j.base`: These are loading classes
- `org.deeplearning4j.berkeley`: These are math utility methods
- `org.deeplearning4j.clustering`: This is the implementation of k-means clustering
- `org.deeplearning4j.datasets`: This is dataset manipulation, including import, creation, iterating, and so on
- `org.deeplearning4j.distributions`: These are utility methods for distributions
- `org.deeplearning4j.eval`: These are evaluation classes, including the confusion matrix
- `org.deeplearning4j.exceptions`: This implements exception handlers
- `org.deeplearning4j.models`: These are supervised learning algorithms, including deep belief network, stacked autoencoder, stacked denoising autoencoder, and RBM

- `org.deeplearning4j.nn`: These are the implementation of components and algorithms based on neural networks, such as neural network, multi-layer network, convolutional multi-layer network, and so on
- `org.deeplearning4j.optimize`: These are neural net optimization algorithms, including back propagation, multi-layer optimization, output layer optimization, and so on
- `org.deeplearning4j.plot`: These are various methods for rendering data
- `org.deeplearning4j.rng`: This is a random data generator
- `org.deeplearning4j.util`: These are helper and utility methods

MALLET

Machine Learning for Language Toolkit (MALLET), is a large library of natural language processing algorithms and utilities. It can be used in a variety of tasks such as document classification, document clustering, information extraction, and topic modeling. It features command-line interface as well as Java API for several algorithms such as naive Bayes, HMM, **Latent Dirichlet** topic models, logistic regression, and conditional random fields.



MALLET is available under Common Public License 1.0, which means that you can even use it in commercial applications. It can be downloaded from <http://mallet.cs.umass.edu>. Mallet instance is represented by name, label, data, and source. However, there are two methods to import data into the Mallet format, as shown in the following list:

- Instance per file: Each file, that is, document, corresponds to an instance and Mallet accepts the directory name for the input.
- Instance per line: Each line corresponds to an instance, where the following format is assumed: the `instance_name label token`. Data will be a feature vector, consisting of distinct words that appear as tokens and their occurrence count.

The library comprises the following packages:

- `cc.mallet.classify`: These are algorithms for training and classifying instances, including AdaBoost, bagging, C4.5, as well as other decision tree models, multivariate logistic regression, naive Bayes, and Winnow2.
- `cc.mallet.cluster`: These are unsupervised clustering algorithms, including greedy agglomerative, hill climbing, k-best, and k-means clustering.
- `cc.mallet.extract`: This implements tokenizers, document extractors, document viewers, cleaners, and so on.
- `cc.mallet.fst`: This implements sequence models, including conditional random fields, HMM, maximum entropy Markov models, and corresponding algorithms and evaluators.
- `cc.mallet.grmm`: This implements graphical models and factor graphs such as inference algorithms, learning, and testing. For example, loopy belief propagation, Gibbs sampling, and so on.
- `cc.mallet.optimize`: These are optimization algorithms for finding the maximum of a function, such as gradient ascent, limited-memory BFGS, stochastic meta ascent, and so on.
- `cc.mallet.pipe`: These are methods as pipelines to process data into MALLET instances.
- `cc.mallet.topics`: These are topics modeling algorithms, such as Latent Dirichlet allocation, four-level pachinko allocation, hierarchical PAM, DMRT, and so on.
- `cc.mallet.types`: This implements fundamental data types such as dataset, feature vector, instance, and label.
- `cc.mallet.util`: These are miscellaneous utility functions such as command-line processing, search, math, test, and so on.

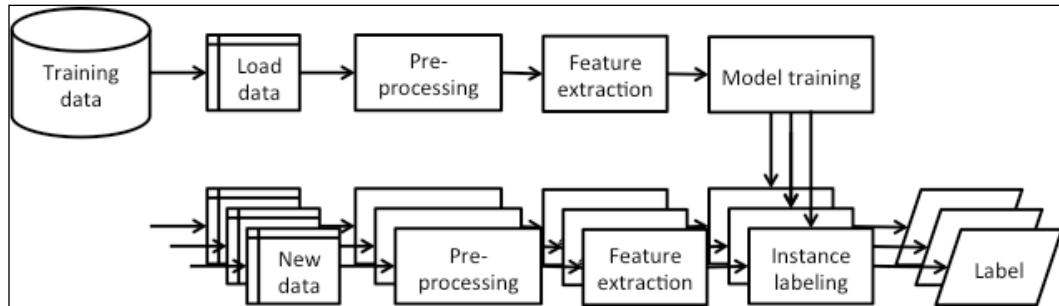
Comparing libraries

The following table summarizes all the presented libraries. The table is, by no means, exhaustive—there are many more libraries covering the specific-problem domains. This review should serve as an overview of the big names in the Java machine learning world:

	Problem domains	License	Architecture	Algorithms
Weka	General purpose	GNU GPL	Single machine	Decision trees, naive Bayes, neural network, random forest, AdaBoost, hierarchical clustering, and so on
Java-ML	General purpose	GNU GPL	Single machine	k-means clustering, self-organizing maps, Markov chain clustering, Cobweb, random forest, decision trees, bagging, distance measures, and so on
Mahout	Classification, recommendation, and clustering	Apache 2.0 License	Distributed, single machine	Logistic regression, naive Bayes, random forest, HMM, multilayer perceptron, k-means clustering, and so on
Spark	General purpose	Apache 2.0 License	Distributed	SVM, logistic regression, decision trees, naive Bayes, k-means clustering, linear least squares, LASSO, ridge regression, and so on
DL4J	Deep learning	Apache 2.0 License	Distributed, single machine	RBM, deep belief networks, deep autoencoders, recursive neural tensor networks, convolutional neural network, and stacked denoising autoencoders
MALLET	Text mining	Common Public License 1.0	Single machine	Naive Bayes, decision trees, maximum entropy, hidden Markov models, and conditional random fields

Building a machine learning application

Machine learning applications, especially those focused on classification, usually follow the same high-level workflow as shown in the following diagram. The workflow comprises two phases: training the classifier and classification of new instances. Both phases share common steps as you can see in the following diagram:



First, we use a set of **Training data**, select a representative subset as the training set, preprocess missing data, and extract features. A selected supervised learning algorithm is used to train a model, which is deployed in the second phase. The second phase puts a new data instance through the same **Pre-processing** and **Feature extraction** procedure and applies the learned model to obtain the instance label. If you are able to collect new labeled data, periodically rerun the learning phase to retrain the model, and replace the old one with the retrained one in the classification phase.

Traditional machine learning architecture

Structured data, such as transactional, customer, analytical, and market data, usually resides within a local relational database. Given a query language, such as SQL, we can query the data used for processing, as shown in the workflow in the previous diagram. Usually, all the data can be stored in the memory and further processed with a machine learning library such as Weka, Java-ML, or MALLET.

A common practice in the architecture design is to create data pipelines, where different steps in the workflow are split. For instance, in order to create a client data record, we might have to scrap the data from different data sources. The record can be then saved in an intermediate database for further processing.

To understand how the high-level aspects of big data architecture differ, let's first clarify when is the data considered big?

Dealing with big data

Big data existed long before the phrase was invented, for instance, banks and stock exchanges have been processing billions of transactions daily for years, and airline companies have worldwide real-time infrastructure for operational management of passenger booking, and so on. So what is big data really? Doug Laney (2001) suggested that big data is defined by three *Vs*: volume, velocity, and variety. Therefore, to answer the question whether your data is big, we can translate this into the following three subquestions:

- **Volume:** Can you store your data in memory?
- **Velocity:** Can you process new incoming data with a single machine?
- **Variety:** Is your data from a single source?

If you answered all the questions with yes, then your data is probably not big, do not worry, you have just simplified your application architecture.

If your answer to all the questions was no, then your data is big! However, if you have mixed answers, then it's complicated. Some may argue that a *V* is important, other may say the other *Vs*. From the machine learning point of view, there is a fundamental difference in algorithm implementation to process the data in memory or from distributed storage. Therefore, a rule of thumb is as follows: if you cannot store your data in the memory, then you should look into a big data machine learning library.

The exact answer depends on the problem that you are trying to solve. If you're starting a new project, I'd suggest you start off with a single-machine library and prototype your algorithm, possibly with a subset of your data if the entire data does not fit into the memory. Once you've established good initial results, consider moving to something more heavy duty such as Mahout or Spark.

Big data application architecture

Big data, such as documents, weblogs, social networks, sensor data, and others, are stored in a NoSQL database, such as MongoDB, or a distributed filesystem, such as HDFS. In case we deal with structured data, we can deploy database capabilities using systems such as Cassandra or HBase built atop Hadoop. Data processing follows the MapReduce paradigm, which breaks data processing problems into smaller subproblems and distributes tasks across processing nodes. Machine learning models are finally trained with machine learning libraries such as Mahout and Spark.

MongoDB is a NoSQL database, which stores documents in a JSON-like format. Read more about it at <https://www.mongodb.org>.

Hadoop is a framework for distributed processing of large datasets across a cluster of computers. It includes its own filesystem format HDFS, job scheduling framework YARN, and implements the MapReduce approach for parallel data processing. More about Hadoop is available at <http://hadoop.apache.org/>.



Cassandra is a distributed database management system built to provide fault-tolerant, scalable, and decentralized storage. More information is available at <http://cassandra.apache.org/>.

HBase is another database that focuses on random read/write access to distributed storage. More information is available at <https://hbase.apache.org/>.

Summary

Selecting a machine learning library has an important impact on your application architecture. The key is to consider your project requirements. What kind of data do you have? What kind of problem are you trying to solve? Is your data big? Do you need distributed storage? What kind of algorithm are you planning to use? Once you figure out what you need to solve your problem, pick a library that best fits your needs.

In the next chapter, we will cover how to complete basic machine learning tasks such as classification, regression, and clustering using some of the presented libraries.

3

Basic Algorithms – Classification, Regression, and Clustering

In the previous chapter, we reviewed the key Java libraries for machine learning and what they bring to the table. In this chapter, we will finally get our hands dirty. We will take a closer look at the basic machine learning tasks such as classification, regression, and clustering. Each of the topics will introduce basic algorithms for classification, regression, and clustering. The example datasets will be small, simple, and easy to understand.

The following is the list of topics that will be covered in this chapter:

- Loading data
- Filtering attributes
- Building classification, regression, and clustering models
- Evaluating models

Before you start

Download the latest version of Weka 3.6 from <http://www.cs.waikato.ac.nz/ml/weka/downloading.html>.

There are multiple download options available. We'll want to use Weka as a library in our source code, so make sure you skip the self-extracting executables and pick the ZIP archive as shown at the following image. Unzip the archive and locate `weka.jar` within the extracted archive:

◦ Other platforms (Linux, etc.)

Click [here](#) to download a zip archive containing Weka
(`weka-3-7-11.zip`; 33.2 MB)

First unzip the zip file. This will create a new directory called `weka-3-7-11`. To run Weka, change into that directory and type

```
java -Xmx1000M -jar weka.jar
```

Note that Java needs to be installed on your system for this to work. Also note, that using `-jar` will override your current CLASSPATH variable and only use the `weka.jar`.

We'll use the Eclipse IDE to show examples, as follows:

1. Start a new Java project.
2. Right-click on the project properties, select **Java Build Path**, click on the **Libraries** tab, and select **Add External JARs**.
3. Navigate to extract the Weka archive and select the `weka.jar` file.

That's it, we are ready to implement the basic machine learning techniques!

Classification

We will start with the most commonly used machine learning technique, that is, classification. As we reviewed in the first chapter, the main idea is to automatically build a mapping between the input variables and the outcome. In the following sections, we will look at how to load the data, select features, implement a basic classifier in Weka, and evaluate the classifier performance.

Data

For this task, we will have a look at the `zoo` database [ref]. The database contains 101 data entries of the animals described with 18 attributes as shown in the following table:

animal	aquatic	fins
hair	predator	legs
feathers	toothed	tail
eggs	backbone	domestic
milk	breathes	cat size
airborne	venomous	type

An example entry in the dataset set is a lion with the following attributes:

- animal: lion
- hair: true
- feathers: false
- eggs: false
- milk: true
- airbone: false
- aquatic: false
- predator: true
- toothed: true
- backbone: true
- breaths: true
- venomous: false
- fins: false
- legs: 4
- tail: true
- domestic: false
- catsize: true
- type: mammal

Our task will be to build a model to predict the outcome variable, `animal`, given all the other attributes as input.

Loading data

Before we start with the analysis, we will load the data in Weka's ARFF format and print the total number of loaded instances. Each data sample is held within an `Instances` object, while the complete dataset accompanied with meta-information is handled by the `Instances` object.

To load the input data, we will use the `DataSource` object that accepts a variety of file formats and converts them to `Instances`:

```
DataSource source = new DataSource(args[0]);
Instances data = source.getDataSet();
System.out.println(data.numInstances() + " instances loaded.");
// System.out.println(data.toString());
```

This outputs the number of loaded instances, as follows:

```
101 instances loaded.
```

We can also print the complete dataset by calling the `data.toString()` method.

Our task is to learn a model that is able to predict the `animal` attribute in the future examples for which we know the other attributes but do not know the `animal` label. Hence, we remove the `animal` attribute from the training set. We accomplish this by filtering out the `animal` attribute using the `Remove` filter.

First, we set a string table of parameters, specifying that the first attribute must be removed. The remaining attributes are used as our dataset for training a classifier:

```
Remove remove = new Remove();
String[] opts = new String[] { "-R", "1" };
```

Finally, we call the `Filter.useFilter(Instances, Filter)` static method to apply the filter on the selected dataset:

```
remove.setOptions(opts);
remove.setInputFormat(data);
data = Filter.useFilter(data, remove);
```

Feature selection

As introduced in *Chapter 1, Applied Machine Learning Quick Start*, one of the pre-processing steps is focused on feature selection, also known as attribute selection. The goal is to select a subset of relevant attributes that will be used in a learned model. Why is feature selection important? A smaller set of attributes simplifies the models and makes them easier to interpret by users, this usually requires shorter training and reduces overfitting.

Attribute selection can take into account the class value or not. In the first case, an attribute selection algorithm evaluates the different subsets of features and calculates a score that indicates the quality of selected attributes. We can use different searching algorithms such as exhaustive search, best first search, and different quality scores such as information gain, Gini index, and so on.

Weka supports this process by an `AttributeSelection` object, which requires two additional parameters: `evaluator`, which computes how informative an attribute is and a `ranker`, which sorts the attributes according to the score assigned by the evaluator.

In this example, we will use information gain as an evaluator and rank the features by their information gain score:

```
InfoGainAttributeEval eval = new InfoGainAttributeEval();
Ranker search = new Ranker();
```

Next, we initialize an `AttributeSelection` object and set the evaluator, ranker, and data:

```
AttributeSelection attSelect = new AttributeSelection();
attSelect.setEvaluator(eval);
attSelect.setSearch(search);
attSelect.SelectAttributes(data);
```

Finally, we can print an order list of attribute indices, as follows:

```
int[] indices = attSelect.selectedAttributes();
System.out.println(Utils.arrayToString(indices));
```

The method outputs the following result:

```
12,3,7,2,0,1,8,9,13,4,11,5,15,10,6,14,16
```

The top three most informative attributes are 12 (fins), 3 (eggs), 7 (aquatic), 2 (hair), and so on. Based on this list, we can remove additional, non-informative features in order to help learning algorithms achieve more accurate and faster learning models.

What would make the final decision about the number of attributes to keep? There's no rule of thumb related to an exact number – the number of attributes depends on the data and problem. The purpose of attribute selection is choosing attributes that serve your model better, so it is better to focus whether the attributes are improving the model.

Learning algorithms

We have loaded our data, selected the best features, and are ready to learn some classification models. Let's begin with the basic decision trees.

Decision tree in Weka is implemented within the `J48` class, which is a re-implementation of Quinlan's famous C4.5 decision tree learner [Quinlan, 1993].

First, we initialize a new `J48` decision tree learner. We can pass additional parameters with a string table, for instance, tree pruning that controls the model complexity (refer to *Chapter 1, Applied Machine Learning Quick Start*). In our case, we will build an un-pruned tree, hence we will pass a single `U` parameter:

```
J48 tree = new J48();
String[] options = new String[1];
options[0] = "-U";

tree.setOptions(options);
```

Next, we call the `buildClassifier(Instances)` method to initialize the learning process:

```
tree.buildClassifier(data);
```

The built model is now stored in a `tree` object. We can output the entire `J48` unpruned tree calling the `toString()` method:

```
System.out.println(tree);
```

The output is as follows:

```
J48 unpruned tree
-----
feathers = false
|   milk = false
|   |   backbone = false
|   |   |   airborne = false
|   |   |   |   predator = false
```

```

|   |   |   |   |   legs <= 2: invertebrate (2.0)
|   |   |   |   |   legs > 2: insect (2.0)
|   |   |   |   |   predator = true: invertebrate (8.0)
|   |   |   |   |   airborne = true: insect (6.0)
|   |   |   |   |   backbone = true
|   |   |   |   |   fins = false
|   |   |   |   |   |   tail = false: amphibian (3.0)
|   |   |   |   |   |   tail = true: reptile (6.0/1.0)
|   |   |   |   |   |   fins = true: fish (13.0)
|   |   |   |   |   milk = true: mammal (41.0)
feathers = true: bird (20.0)

```

Number of Leaves : .9

Size of the tree : ..17

The outputted tree has 17 nodes in total, 9 of these are terminal (Leaves).

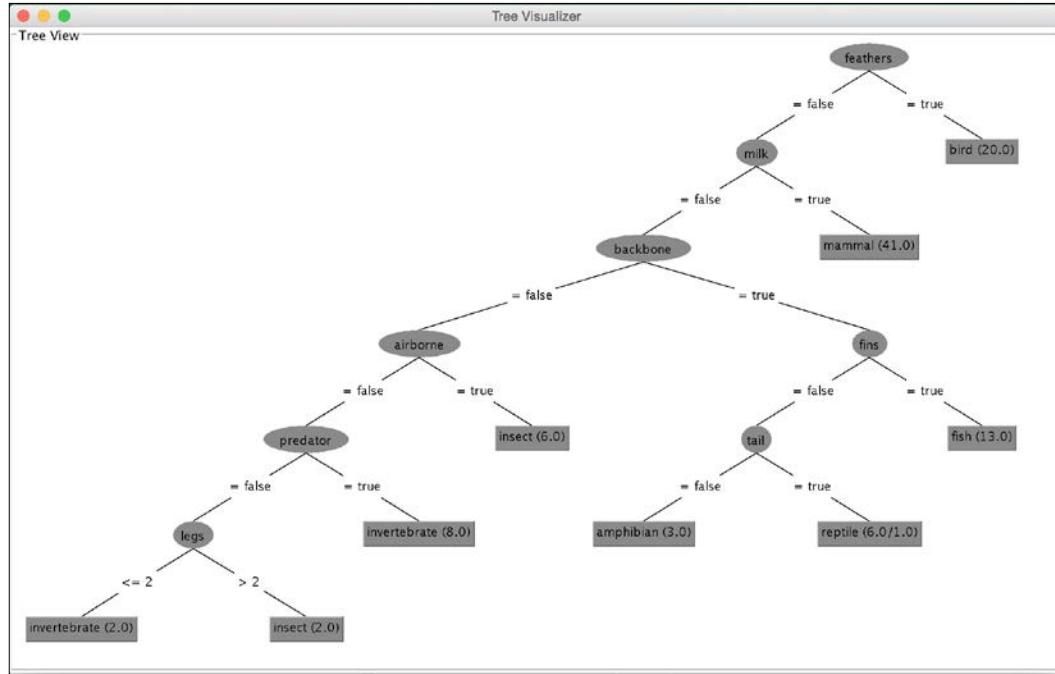
Another way to present the tree is to leverage the built-in TreeVisualizer tree viewer, as follows:

```

TreeVisualizer tv = new TreeVisualizer(null, tree.graph(), new
PlaceNode2());
JFrame frame = new javax.swing.JFrame("Tree Visualizer");
frame.setSize(800, 500);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.getContentPane().add(tv);
frame.setVisible(true);
tv.fitToScreen();

```

The code results in the following frame:



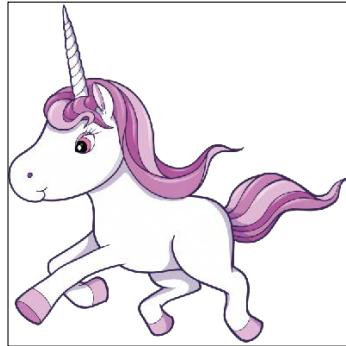
The decision process starts at the top node, also known as the root node. The node label specifies the attribute value that will be checked. In our example, we first check the value of the `feathers` attribute. If the feather is present, we follow the right-hand branch, which leads us to the leaf labeled `bird`, indicating there are 20 examples supporting this outcome. If the feather is not present, we follow the left-hand branch, which leads us to the next `milk` attribute. We check the value of the attribute again and follow the branch that matches the attribute value. We repeat the process until we reach a leaf node.

We can build other classifiers by following the same steps: initialize a classifier, pass the parameters controlling the model complexity, and call the `buildClassifier(Instances)` method.

In the next section, we will learn how to use a trained model to assign a class label to a new example whose label is unknown.

Classify new data

Suppose we record attributes for an animal whose label we do not know, we can predict its label from the learned classification model:



We first construct a feature vector describing the new specimen, as follows:

```
double[] vals = new double[data.numAttributes()];
vals[0] = 1.0; //hair {false, true}
vals[1] = 0.0; //feathers {false, true}
vals[2] = 0.0; //eggs {false, true}
vals[3] = 1.0; //milk {false, true}
vals[4] = 0.0; //airborne {false, true}
vals[5] = 0.0; //aquatic {false, true}
vals[6] = 0.0; //predator {false, true}
vals[7] = 1.0; //toothed {false, true}
vals[8] = 1.0; //backbone {false, true}
vals[9] = 1.0; //breathes {false, true}
vals[10] = 1.0; //venomous {false, true}
vals[11] = 0.0; //fins {false, true}
vals[12] = 4.0; //legs INTEGER [0,9]
vals[13] = 1.0; //tail {false, true}
vals[14] = 1.0; //domestic {false, true}
vals[15] = 0.0; //catsize {false, true}
Instance myUnicorn = new Instance(1.0, vals);
```

Finally, we call the `classify(Instance)` method on the model to obtain the class value. The method returns label index, as follows:

```
double result = tree.classifyInstance(myUnicorn);
System.out.println(data.classAttribute().value((int) result));
```

This outputs the mammal class label.

Evaluation and prediction error metrics

We built a model, but we do not know if it can be trusted. To estimate its performance, we can apply a cross-validation technique explained in *Chapter 1, Applied Machine Learning Quick Start*.

Weka offers an `Evaluation` class implementing cross validation. We pass the model, data, number of folds, and an initial random seed, as follows:

```
Classifier cl = new J48();
Evaluation eval_roc = new Evaluation(data);
eval_roc.crossValidateModel(cl, data, 10, new Random(1), new
Object[] {});
System.out.println(eval_roc.toSummaryString());
```

The evaluation results are stored in the `Evaluation` object.

A mix of the most common metrics can be invoked by calling the `toString()` method. Note that the output does not differentiate between regression and classification, so pay attention to the metrics that make sense, as follows:

Correctly Classified Instances	93	92.0792 %
Incorrectly Classified Instances	8	7.9208 %
Kappa statistic	0.8955	
Mean absolute error	0.0225	
Root mean squared error	0.14	
Relative absolute error	10.2478 %	
Root relative squared error	42.4398 %	
Coverage of cases (0.95 level)	96.0396 %	
Mean rel. region size (0.95 level)	15.4173 %	
Total Number of Instances	101	

In the classification, we are interested in the number of correctly/incorrectly classified instances.

Confusion matrix

Furthermore, we can inspect where a particular misclassification has been made by examining the confusion matrix. Confusion matrix shows how a specific class value was predicted:

```
double[][] confusionMatrix = eval_roc.confusionMatrix();
System.out.println(eval_roc.toMatrixString());
```

The resulting confusion matrix is as follows:

```
==== Confusion Matrix ===
```

a	b	c	d	e	f	g	<-- classified as
41	0	0	0	0	0	0	a = mammal
0	20	0	0	0	0	0	b = bird
0	0	3	1	0	1	0	c = reptile
0	0	0	13	0	0	0	d = fish
0	0	1	0	3	0	0	e = amphibian
0	0	0	0	0	5	3	f = insect
0	0	0	0	0	2	8	g = invertebrate

The first column names in the first row correspond to labels assigned by the classification mode. Each additional row then corresponds to an actual true class value. For instance, the second row corresponds instances with the `mammal` true class label. In the column line, we read that all mammals were correctly classified as mammals. In the fourth row, `reptiles`, we notice that three were correctly classified as reptiles, while one was classified as `fish` and one as an `insect`. Confusion matrix hence, gives us an insight into the kind of errors that our classification model makes.

Choosing a classification algorithm

Naive Bayes is one of the most simple, efficient, and effective inductive algorithms in machine learning. When features are independent, which is rarely true in real world, it is theoretically optimal, and even with dependent features, its performance is amazingly competitive (Zhang, 2004). The main disadvantage is that it cannot learn how features interact with each other, for example, despite the fact that you like your tea with lemon or milk, you hate a tea having both of them at the same time.

Decision tree's main advantage is a model, that is, a tree, which is easy to interpret and explain as we studied in our example. It can handle both nominal and numeric features and you don't have to worry about whether the data is linearly separable.

Some other examples of classification algorithms are as follows:

- `weka.classifiers.rules.ZeroR`: This predicts the majority class and is considered as a baseline, that is, if your classifier's performance is worse than the average value predictor, it is not worth considering it.
- `weka.classifiers.trees.RandomTree`: This constructs a tree that considers K randomly chosen attributes at each node.

- `weka.classifiers.trees.RandomForest`: This constructs a set (that is, forest) of random trees and uses majority voting to classify a new instance.
- `weka.classifiers.lazy.IBk`: This is the k-nearest neighbor's classifier that is able to select an appropriate value of neighbors based on cross-validation.
- `weka.classifiers.functions.MultilayerPerceptron`: This is a classifier based on neural networks that use back-propagation to classify instances. The network can be built by hand, or created by an algorithm, or both.
- `weka.classifiers.bayes.NaiveBayes`: This is a naive Bayes classifier that uses estimator classes, where numeric estimator precision values are chosen based on the analysis of the training data.
- `weka.classifiers.meta.AdaBoostM1`: This is the class for boosting a nominal class classifier using the **AdaBoost M1** method. Only nominal class problems can be tackled. This often dramatically improves the performance, but sometimes it overfits.
- `weka.classifiers.meta.Bagging`: This is the class for bagging a classifier to reduce the variance. This can perform classification and regression, depending on the base learner.

Regression

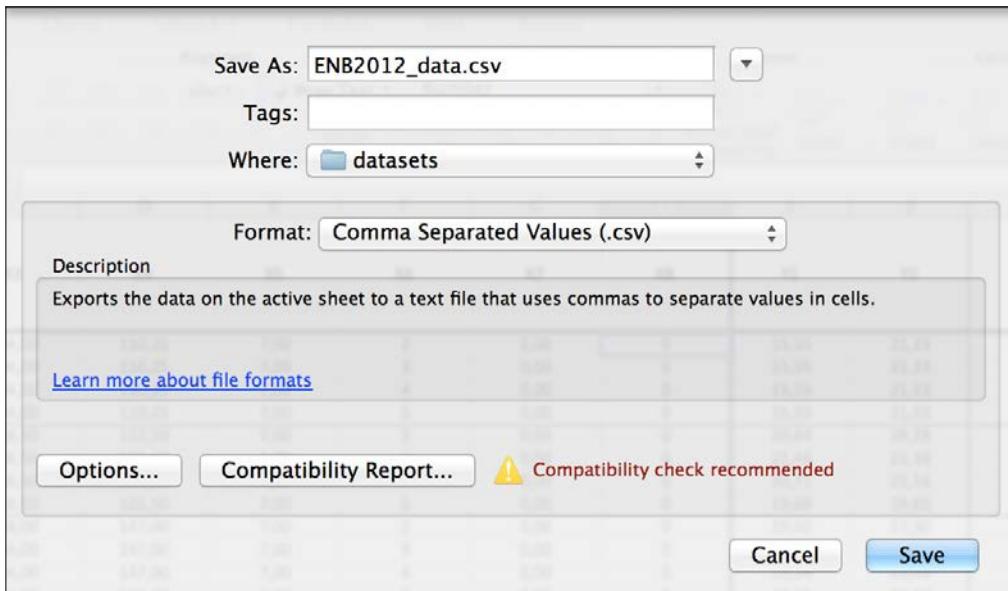
We will explore basic regression algorithms through analysis of energy efficiency dataset (Tsanas and Xifara, 2012). We will investigate the heating and cooling load requirements of the buildings based on their construction characteristics such as surface, wall and roof area, height, hazing area, and compactness. The researchers used a simulator to design 12 different house configurations while varying 18 building characteristics. In total, 768 different buildings were simulated.

Our first goal is to systematically analyze the impact each building characterizes has on the target variable, that is, heating or cooling load. The second goal is to compare the performance of a classical linear regression model against other methods, such as SVM regression, random forests, and neural networks. For this task, we will use the Weka library.

Loading the data

Download the energy efficiency dataset from
<https://archive.ics.uci.edu/ml/datasets/Energy+efficiency>.

The dataset is in Excel's XLSX format, which cannot be read by Weka. We can convert it to a **Comma Separated Value (CSV)** format by clicking **File | Save As...** and picking CSV in the saving dialog as shown in the following screenshot. Confirm to save only the active sheet (since all others are empty) and confirm to continue to lose some formatting features. Now, the file is ready to be loaded by Weka:



Open the file in a text editor and inspect if the file was indeed correctly transformed. There might be some minor issues that may be potentially causing problems. For instance, in my export, each line ended with a double semicolon, as follows:

```
X1;X2;X3;X4;X5;X6;X7;X8;Y1;Y2;;
0,98;514,50;294,00;110,25;7,00;2;0,00;0;15,55;21,33;;
0,98;514,50;294,00;110,25;7,00;3;0,00;0;15,55;21,33;;
```

To remove the doubled semicolon, we can use the **Find and Replace** function: find "`;;`" and replace it with "`;`".

The second problem was that my file had a long list of empty lines at the end of the document, which can be simply deleted:

```
0,62;808,50;367,50;220,50;3,50;5;0,40;5;16,64;16,03;;
;;;;;;
;;;;;;
```

Now, we are ready to load the data. Let's open a new file and write a simple data import function using Weka's converter for reading files in CSV format:

```
import weka.core.Instances;
import weka.core.converters.CSVLoader;
import java.io.File;
import java.io.IOException;

public class EnergyLoad {

    public static void main(String[] args) throws IOException {

        // load CSV
        CSVLoader loader = new CSVLoader();
        loader.setSource(new File(args[0]));
        Instances data = loader.getDataSet();

        System.out.println(data);
    }
}
```

The data is loaded. Let's move on.

Analyzing attributes

Before we analyze attributes, let's first try to understand what we are dealing with. In total, there are eight attributes describing building characteristic and two target variables, heating and cooling load, as shown in the following table:

Attribute	Attribute name
X1	Relative compactness
X2	Surface area
X3	Wall area
X4	Roof area
X5	Overall height
X6	Orientation
X7	Glazing area
X8	Glazing area distribution
Y1	Heating load
Y2	Cooling load

Building and evaluating regression model

We will start with learning a model for heating load by setting the class attribute at the feature position:

```
data.setClassIndex(data.numAttributes() - 2);
```

The second target variable—cooling load—can be now removed:

```
//remove last attribute Y2
Remove remove = new Remove();
remove.setOptions(new String[] {"-R", data.numAttributes() + ""});
remove.setInputFormat(data);
data = Filter.useFilter(data, remove);
```

Linear regression

We will start with a basic linear regression model implemented with the `LinearRegression` class. Similarly as in the classification example, we will initialize a new model instance, pass parameters and data, and invoke the `buildClassifier(Instances)` method, as follows:

```
import weka.classifiers.functions.LinearRegression;
...
data.setClassIndex(data.numAttributes() - 2);
LinearRegression model = new LinearRegression();
model.buildClassifier(data);
System.out.println(model);
```

The learned model, which is stored in the object, can be outputted by calling the `toString()` method, as follows:

```
Y1 =
-64.774 * X1 +
-0.0428 * X2 +
0.0163 * X3 +
-0.089 * X4 +
4.1699 * X5 +
19.9327 * X7 +
0.2038 * X8 +
83.9329
```

Linear regression model constructed a function that linearly combines the input variables to estimate the heating load. The number in front of the feature explains the feature's impact on the target variable: sign corresponds to positive/negative impact, while magnitude corresponds to its significance. For instance, feature x_1 – relative compactness is negatively correlated with heating load, while glazing area is positively correlated. These two features also significantly impact the final heating load estimation.

The model performance can be similarly evaluated with cross-validation technique.

The 10-fold cross-validation is as follows:

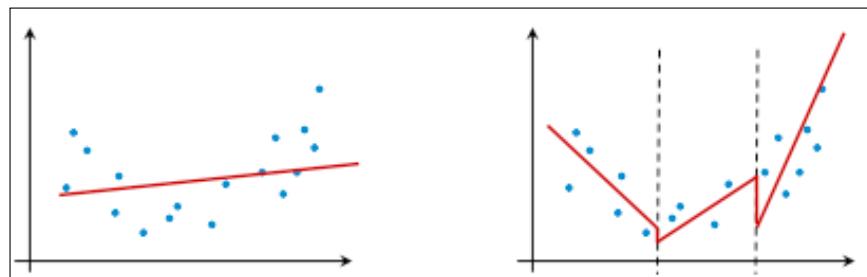
```
Evaluation eval = new Evaluation(data);
eval.crossValidateModel(
    model, data, 10, new Random(1), new String[]{} );
System.out.println(eval.toSummaryString());
```

We can output the common evaluation metrics including correlation, mean absolute error, relative absolute error, and so on, as follows:

Correlation coefficient	0.956
Mean absolute error	2.0923
Root mean squared error	2.9569
Relative absolute error	22.8555 %
Root relative squared error	29.282 %
Total Number of Instances	768

Regression trees

Another approach is to construct a set of regression models, each on its own part of the data. The following diagram shows the main difference between a regression model and a regression tree. Regression model constructs a single model that best fits all the data. Regression tree, on the other hand, constructs a set of regression models, each modeling a part of the data as shown on the right-hand side. Compared to the regression model, the regression tree can better fit the data, but the function is a piece-wise linear with jumps between modeled regions:



Regression tree in Weka is implemented within the `M5` class. Model construction follows the same paradigm: initialize model, pass parameters and data, and invoke the `buildClassifier(Instances)` method.

```
import weka.classifiers.trees.M5P;
...
M5P md5 = new M5P();
md5.setOptions(new String[] { "" });
md5.buildClassifier(data);
System.out.println(md5);
```

The induced model is a tree with equations in the leaf nodes, as follows:

```
M5 pruned model tree:
(using smoothed linear models)

X1 <= 0.75 :
|   X7 <= 0.175 :
|   |   X1 <= 0.65 : LM1 (48/12.841%)
|   |   X1 > 0.65 : LM2 (96/3.201%)
|   X7 > 0.175 :
|   |   X1 <= 0.65 : LM3 (80/3.652%)
|   |   X1 > 0.65 : LM4 (160/3.502%)
X1 > 0.75 :
|   X1 <= 0.805 : LM5 (128/13.302%)
|   X1 > 0.805 :
|   |   X7 <= 0.175 :
|   |   |   X8 <= 1.5 : LM6 (32/20.992%)
|   |   |   X8 > 1.5 :
|   |   |   |   X1 <= 0.94 : LM7 (48/5.693%)
|   |   |   |   X1 > 0.94 : LM8 (16/1.119%)
|   |   X7 > 0.175 :
|   |   |   X1 <= 0.84 :
|   |   |   |   X7 <= 0.325 : LM9 (20/5.451%)
|   |   |   |   X7 > 0.325 : LM10 (20/5.632%)
|   |   |   X1 > 0.84 :
|   |   |   |   X7 <= 0.325 : LM11 (60/4.548%)
|   |   |   |   X7 > 0.325 :
|   |   |   |   |   X3 <= 306.25 : LM12 (40/4.504%)
|   |   |   |   |   X3 > 306.25 : LM13 (20/6.934%)
```

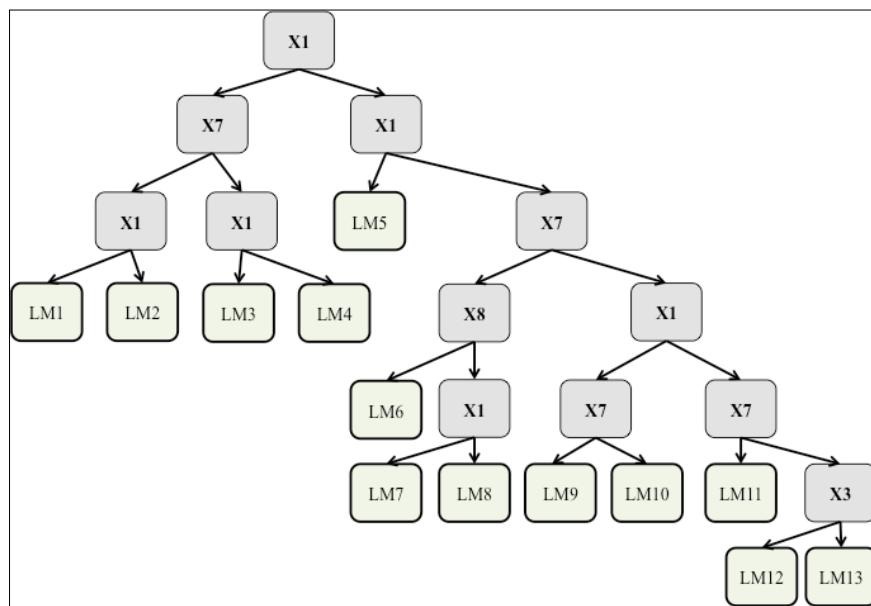
```
LM num: 1
Y1 =
  72.2602 * X1
  + 0.0053 * X3
  + 11.1924 * X7
  + 0.429 * X8
  - 36.2224
```

...

```
LM num: 13
Y1 =
  5.8829 * X1
  + 0.0761 * X3
  + 9.5464 * X7
  - 0.0805 * X8
  + 2.1492
```

Number of Rules : 13

The tree has 13 leaves, each corresponding to a linear equation. The preceding output is visualized in the following diagram:



The tree can be read similar to a classification tree. The most important features are at the top of the tree. The terminal node, leaf, contains a linear regression model explaining the data that reach this part of the tree.

Evaluation outputs the following results:

Correlation coefficient	0.9943
Mean absolute error	0.7446
Root mean squared error	1.0804
Relative absolute error	8.1342 %
Root relative squared error	10.6995 %
Total Number of Instances	768

Tips to avoid common regression problems

First, use prior studies and domain knowledge to figure out which features to include in regression. Check literature, reports, and previous studies on what kind of features work and reasonable variables for modeling your problem. Suppose you have a large set of features with random data, it is highly likely that several features will be correlated to the target variable (even though the data is random).

Keep the model simple to avoid overfitting. The Occam's razor principle states that you should select a model that best explains your data with the fewest assumptions. In practice, the model can be as simple as 2-4 predictor features.

Clustering

Compared to a supervised classifier, the goal of clustering is to identify intrinsic groups in a set of unlabeled data. It could be applied in identifying representative examples of homogeneous groups, finding useful and suitable groupings, or finding unusual examples, such as outliers.

We'll demonstrate how to implement clustering by analyzing the Bank dataset. The dataset consist of 11 attributes, describing 600 instances with age, sex, region, income, marriage status, children, car ownership status, saving activity, current activity, mortgage status, and PEP. In our analysis, we will try to identify the common groups of clients by applying the **Expectation Maximization (EM)** clustering.

EM works as follows: given a set of clusters, EM first assigns each instance with a probability distribution of belonging to a particular cluster. For example, if we start with three clusters— A , B , and C —an instance might get the probability distribution of 0.70, 0.10, and 0.20, belonging to the A , B , and C clusters, respectively. In the second step, EM re-estimates the parameter vector of the probability distribution of each class. The algorithm iterates these two steps until the parameters converge or the maximum number of iterations is reached.

The number of clusters to be used in EM can be set either manually or automatically by cross validation. Another approach to determining the number of clusters in a dataset includes the **elbow** method. The method looks at the percentage of variance that is explained with a specific number of clusters. The method suggests increasing the number of clusters until the additional cluster does not add much information, that is, explains little additional variance.

Clustering algorithms

The process of building a cluster model is quite similar to the process of building a classification model, that is, load the data and build a model. Clustering algorithms are implemented in the `weka.clusterers` package, as follows:

```
import java.io.BufferedReader;
import java.io.FileReader;

import weka.core.Instances;
import weka.clusterers.EM;

public class Clustering {

    public static void main(String args[]) throws Exception{

        //load data
        Instances data = new Instances(new BufferedReader
            (new FileReader(args[0])));

        // new instance of clusterer
        EM model = new EM();
        // build the clusterer
        model.buildClusterer(data);
        System.out.println(model);

    }
}
```

The model identified the following six clusters:

```
EM
```

```
==
```

```
Number of clusters selected by cross validation: 6
```

Attribute	Cluster					
	0 (0.1)	1 (0.13)	2 (0.26)	3 (0.25)	4 (0.12)	5 (0.14)
<hr/>						
age						
0_34	10.0535	51.8472	122.2815	12.6207	3.1023	1.0948
35_51	38.6282	24.4056	29.6252	89.4447	34.5208	3.3755
52_max	13.4293	6.693	6.3459	50.8984	37.861	81.7724
[total]	62.1111	82.9457	158.2526	152.9638	75.4841	86.2428
sex						
FEMALE	27.1812	32.2338	77.9304	83.5129	40.3199	44.8218
MALE	33.9299	49.7119	79.3222	68.4509	34.1642	40.421
[total]	61.1111	81.9457	157.2526	151.9638	74.4841	85.2428
region						
INNER_CITY	26.1651	46.7431	73.874	60.1973	33.3759	34.6445
TOWN	24.6991	13.0716	48.4446	53.1731	21.617	
17.9946						
...						

The table can be read as follows: the first line indicates six clusters, while the first column shows attributes and their ranges. For example, the attribute age is split into three ranges: 0-34, 35-51, and 52-max. The columns on the left indicate how many instances fall into the specific range in each cluster, for example, clients in the 0-34 years age group are mostly in cluster #2 (122 instances).

Evaluation

A clustering algorithm's quality can be estimated using the log likelihood measure, which measures how consistent the identified clusters are. The dataset is split into multiple folds and clustering is run with each fold. The motivation here is that if the clustering algorithm assigns high probability to similar data that wasn't used to fit parameters, then it has probably done a good job of capturing the data structure. Weka offers the `ClusterEvaluation` class to estimate it, as follows:

```
double logLikelihood = ClusterEvaluation.crossValidateModel(model, data,  
10, new Random(1));  
  
System.out.println(logLikelihood);
```

It has the following output:

```
-8.773410259774291
```

Summary

In this chapter, you learned how to implement basic machine learning tasks with Weka: classification, regression, and clustering. We briefly discussed attribute selection process, trained models, and evaluated their performance.

The next chapter will focus on how to apply these techniques to solve real-life problems, such as customer retention.

4

Customer Relationship Prediction with Ensembles

Any type of company offering a service, product, or experience needs a solid understanding of relationship with their customers; therefore, **Customer Relationship Management (CRM)** is a key element of modern marketing strategies. One of the biggest challenges that businesses face is the need to understand exactly what causes a customer to buy new products.

In this chapter, we will work on a real-world marketing database provided by the French telecom company, Orange. The task will be to estimate the following likelihoods for customer actions:

- Switch provider (churn)
- Buy new products or services (appetency)
- Buy upgrades or add-ons proposed to them to make the sale more profitable (upselling)

We will tackle the **Knowledge Discovery and Data Mining (KDD) Cup 2009** challenge (KDD Cup, 2009) and show the steps to process the data using Weka. First, we will parse and load the data and implement the basic baseline models. Later, we will address advanced modeling techniques, including data pre-processing, attribute selection, model selection, and evaluation.



KDD Cup is the leading data mining competition in the world. It is organized annually by ACM Special Interest Group on Knowledge Discovery and Data Mining. The winners are announced at the Conference on Knowledge Discovery and Data Mining, which is usually held in August.

Yearly archives, including all the corresponding datasets, are available here: <http://www.kdd.org/kdd-cup>.

Customer relationship database

The most practical way to build knowledge on customer behavior is to produce scores that explain a target variable such as churn, appetency, or upselling. The score is computed by a model using input variables describing customers, for example, current subscription, purchased devices, consumed minutes, and so on. The scores are then used by the information system, for example, to provide relevant personalized marketing actions.

In 2009, the conference on KDD organized a machine learning challenge on customer-relationship prediction (KDD Cup, 2009).

Challenge

Given a large set of customer attributes, the task was to estimate the following three target variables (KDD Cup, 2009):

- **Churn probability**, in our context, is the likelihood a customer will switch providers:

Churn rate is also sometimes called attrition rate. It is one of two primary factors that determine the steady-state level of customers a business will support. In its broadest sense, churn rate is a measure of the number of individuals or items moving into or out of a collection over a specific period of time. The term is used in many contexts, but is most widely applied in business with respect to a contractual customer base. For instance, it is an important factor for any business with a subscriber-based service model, including mobile telephone networks and pay TV operators. The term is also used to refer to participant turnover in peer-to-peer networks.

- **Appetency probability**, in our context, is the propensity to buy a service or product
- **Upselling probability** is the likelihood that a customer will buy an add-on or upgrade:

Upselling is a sales technique whereby a salesman attempts to have the customer purchase more expensive items, upgrades, or other add-ons in an attempt to make a more profitable sale. Upselling usually involves marketing more profitable services or products, but upselling can also be simply exposing the customer to other options he or she may not have considered previously. Upselling can imply selling something additional, or selling something that is more profitable or otherwise preferable for the seller instead of the original sale.

The challenge was to beat the in-house system developed by Orange Labs. This was an opportunity for the participants to prove that they could handle a large database, including heterogeneous noisy data and unbalanced class distributions.

Dataset

For the challenge, the company Orange released a large dataset of customer data, containing about one million customers, described in ten tables with hundreds of fields. In the first step, they resampled the data to select a less unbalanced subset containing 100,000 customers. In the second step, they used an automatic feature construction tool that generated 20,000 features describing customers, which was then narrowed down to 15,000 features. In the third step, the dataset was anonymized by randomizing the order of features, discarding attribute names, replacing nominal variables with randomly generated strings, and multiplying continuous attributes by a random factor. Finally, all the instances were split randomly into a train and test dataset.

The KDD Cup provided two sets of data: large set and small set, corresponding to fast and slow challenge, respectively. They are described at the KDD Cup site as follows:

Both training and test sets contain 50,000 examples. The data are split similarly for the small and large versions, but the samples are ordered differently within the training and within the test sets. Both small and large datasets have numerical and categorical variables. For the large dataset, the first 14,740 variables are numerical and the last 260 are categorical. For the small dataset, the first 190 variables are numerical and the last 40 are categorical.

In this chapter, we will work with the small dataset consisting of 50,000 instances described with 230 variables each. Each of the 50,000 rows of data correspond to a client and are associated with three binary outcomes – one for each of the three challenges (upsell, churn, and appetency).

To make this clearer, the following image illustrates the dataset in a table format:

230 numeric and nominal attributes											Three binary classes		
Var85	Var123	Var125	Var126	Var132	Var133	Var134	Var225	Var229	Var230	Label Churn	Label Appetency	Label Upselling	
12	6	720	8	0	1212385	69134				-1	-1	-1	
2	72	0		8	4136430	357038				1	-1	-1	
58	114	5967	-28	0	3478905	248932	kG3k	am7c		-1	-1	-1	
0	0	0	-14	0	0	0				-1	-1	-1	
0	0	15111	58	0	150650	66046	kG3k	mj86		-1	-1	-1	
10	0	1935		8	641020	43684		am7c		-1	-1	-1	
16	24	13194	-24	0	1664450	104978	kG3k	am7c		-1	-1	-1	
2	12	0	-8	8	3839825	1284128				-1	-1	-1	
2	90	2754		0	3830510	203586	kG3k	am7c		-1	-1	-1	
24	66	6561		32	2577245	210014	kG3k			-1	-1	-1	
6	12	5823	58	0	0	7134	kG3k	mj86		-1	-1	-1	
28	24	66825	52	8	134105	15166	kG3k			-1	-1	-1	
0	0	44154	10	0	0	0		mj86		-1	-1	-1	
22	54	5202		0	2772010	1095062	xG3x			-1	-1	-1	
0	102	31104	8	0	2170355	57596				-1	-1	1	
0	0	2574		0	0	0	ElOf	ojmt		-1	-1	-1	
14	186	8019		48	3571845	587392	kG3k	am7c		-1	-1	-1	
0	30	5319		8	500295	31436		am7c		-1	-1	-1	
2	0	13788	4	0	918350	0	kG3k			-1	-1	-1	
14	0	7110		0	2055150	392138				1	-1	-1	
8	66	0	-8	0	3258940	1121306				-1	-1	-1	
0	18	0	-10	0	0	0				-1	-1	-1	
12	0	531	36	0	491345	56742	ElOf	mj86		-1	-1	-1	
0	12	16803	12	0	201110	1693090				1	-1	-1	
14	0	25740		0	2932660	313200	xG3x			-1	-1	1	

The table depicts the first 25 instances, that is, customers, each described with 250 attributes. For this example, only a selected subset of 10 attributes is shown. The dataset contains many missing values and even empty or constant attributes. The last three columns of the table correspond to the three distinct class labels corresponding to the ground truth, that is, if the customer indeed switched the provider (churn), bought a service (appetency), or bought an upgrade (upsell). However, note that the labels are provided separately from the data in three distinct files, hence, it is essential to retain the order of the instances and corresponding class labels to ensure proper correspondence.

Evaluation

The submissions were evaluated according to the arithmetic mean of the area under the ROC curve (AUC) for the three tasks, that is, churn, appetency, and upselling. ROC curve shows the performance of model as a curve obtained by plotting sensitivity against specificity for various threshold values used to determine the classification result (refer to *Chapter 1, Applied Machine Learning Quick Start*, section *ROC curves*). Now, the AUC is related to the area under this curve, meaning larger the area, better the classifier. Most toolboxes, including Weka, provide an API to calculate AUC score.

Basic naive Bayes classifier baseline

As per the rules of the challenge, the participants had to outperform the basic naive Bayes classifier to qualify for prizes, which makes an assumption that features are independent (refer to *Chapter 1, Applied Machine Learning Quick Start*).

The KDD Cup organizers run the vanilla naive Bayes classifier, without any feature selection or hyperparameter adjustments. For the large dataset, the overall scores of the naive Bayes on the test set were as follows:

- **Churn problem:** $AUC = 0.6468$
- **Appetency problem:** $AUC = 0.6453$
- **Upselling problem:** $AUC=0.7211$

Note that the baseline results are reported for large dataset only. Moreover, while both training and test datasets are provided at the KDD Cup site, the actual true labels for the test set are not provided. Therefore, when we process the data with our models, there is no way to know how well the models will perform on the test set. What we will do is use only the training data and evaluate our models with cross validation. The results will not be directly comparable, but, nevertheless, we have an idea for what a reasonable magnitude of the AUC score is.

Getting the data

At the KDD Cup web page (<http://kdd.org/kdd-cup/view/kdd-cup-2009/Data>), you should see a page that looks like the following screenshot. First, under the **Small version (230 var.)** header, download `orange_small_train.data.zip`. Next, download the three sets of true labels associated with this training data. The following files are found under the **Real binary targets (small)** header:

- `orange_small_train_appency.labels`
- `orange_small_train_churn.labels`
- `orange_small_train_upselling.labels`

Save and unzip all the files marked in the red boxes, as shown in the following screenshot:

The screenshot shows the KDD Cup 2009 Data page. The 'Data Download' section contains several download links. Two specific sections are highlighted with red boxes:

- Small version (230 var.):**
 - `orange_small_train.data.zip` (8.2 Mbytes)
 - `orange_small_test.data.zip` (8.2 Mbytes)
- Real binary targets (small):**
 - `orange_small_train_appency.labels`
 - `orange_small_train_churn.labels`
 - `orange_small_train_upselling.labels`

On the right side of the page, there is a sidebar titled "KDD Cup Archive" listing previous KDD Cup years from 1997 to 2016.

In the following sections, we will first load the data into Weka and apply basic modeling with the naive Bayes to obtain our own baseline AUC scores. Later, we will look into more advanced modeling techniques and tricks.

Loading the data

We will load the data to Weka directly from the .cvs format. For this purpose, we will write a function that accepts the path to the data file and the true labels file. The function will load and merge both datasets and remove empty attributes:

```
public static Instances loadData(String pathData, String
    pathLabels) throws Exception {
```

First, we load the data using the CSVLoader() class. Additionally, we specify the \t tab as a field separator and force the last 40 attributes to be parsed as nominal:

```
// Load data
CSVLoader loader = new CSVLoader();
loader.setFieldSeparator("\t");
loader.setNominalAttributes("191-last");
loader.setSource(new File(pathData));
Instances data = loader.getDataSet();
```

The CSVLoader class accepts many additional parameters specifying column separator, string enclosures, whether a header row is present or not, and so on. Complete documentation is available here:

<http://weka.sourceforge.net/doc.dev/weka/core/converters/CSVLoader.html>

Next, some of the attributes do not contain a single value and Weka automatically recognizes them as the String attributes. We actually do not need them, so we can safely remove them using the RemoveType filter. Additionally, we specify the -T parameters, which means remove attribute of specific type and the attribute type that we want to remove:

```
// remove empty attributes identified as String attribute
RemoveType removeString = new RemoveType();
removeString.setOptions(new String[] {"-T", "string"});
removeString.setInputFormat(data);
Instances filteredData = Filter.useFilter(data, removeString);
```

Alternatively, we could use the void deleteStringAttributes() method implemented within the Instances class, which has the same effect, for example, data.removeStringAttributes().

Now, we will load and assign class labels to the data. We will again utilize `CVSLoader`, where we specify that the file does not have any header line, that is, `setNoHeaderRowPresent(true)`:

```
// Load labeles
loader = new CSVLoader();
loader.setFieldSeparator("\t");
loader.setNoHeaderRowPresent(true);
loader.setNominalAttributes("first-last");
loader.setSource(new File(pathLabelles));
Instances labels = loader.getDataSet();
```

Once we have loaded both files, we can merge them together by calling the `Instances.mergeInstances(Instances, Instances)` static method. The method returns a new dataset that has all the attributes from the first dataset plus the attributes from the second set. Note that the number of instances in both datasets must be the same:

```
// Append label as class value
Instances labeledData = Instances.mergeInstances(filteredData,
    labelles);
```

Finally, we set the last attribute, that is, the label attribute that we have just added, as a target variable and return the resulting dataset:

```
// set the label attribute as class
labeledData.setClassIndex(labeledData.numAttributes() - 1);

System.out.println(labeledData.toSummaryString());
return labeledData;
}
```

The function outputs a summary as shown in the following and returns the labeled dataset:

```
Relation Name: orange_small_train.data-weka.filters.unsupervised.
attribute.RemoveType-Tstring_orange_small_train_churn.labels.txt
```

```
Num Instances: 50000
```

```
Num Attributes: 215
```

Name	Type	Nom	Int	Real	Missing	Unique	Dist
1 Var1	Num	0%	1%	0% 49298 / 99%	8 / 0%	18	
2 Var2	Num	0%	2%	0% 48759 / 98%	1 / 0%	2	
3 Var3	Num	0%	2%	0% 48760 / 98%	104 / 0%	146	
4 Var4	Num	0%	3%	0% 48421 / 97%	1 / 0%	4	
...							

Basic modeling

In this section, we will implement our own baseline model by following the approach that the KDD Cup organizers took. However, before we go to the model, let's first implement the evaluation engine that will return AUC on all three problems.

Evaluating models

Now, let's take a closer look at the evaluation function. The evaluation function accepts an initialized model, cross-validates the model on all three problems, and reports the results as an area under the ROC curve (AUC), as follows:

```
public static double[] evaluate(Classifier model)
    throws Exception {

    double results[] = new double[4];

    String[] labelFiles = new String[]{
        "churn", "appetency", "upselling"};

    double overallScore = 0.0;
    for (int i = 0; i < labelFiles.length; i++) {
```

First, we call the `Instance loadData(String, String)` function that we implemented earlier to load the train data and merge it with the selected labels:

```
// Load data
Instances train_data = loadData(
    path + "orange_small_train.data",
    path+"orange_small_train_"+labelFiles[i]+".labels.txt");
```

Next, we initialize the `weka.classifiers.Evaluation` class and pass our dataset (the dataset is used only to extract data properties, the actual data are not considered). We call the `void crossValidateModel(Classifier, Instances, int, Random)` method to begin cross validation and select to create five folds. As validation is done on random subsets of the data, we need to pass a random seed as well:

```
// cross-validate the data
Evaluation eval = new Evaluation(train_data);
eval.crossValidateModel(model, train_data, 5,
    new Random(1));
```

After the evaluation completes, we read the results by calling the `double areUnderROC(int)` method. As the metric depends on the target value that we are interested in, the method expects a class value index, which can be extracted by searching the index of the "1" value in the `class` attribute:

```
// Save results
results[i] = eval.areaUnderROC(
    train_data.classAttribute().indexOfValue("1"));
overallScore += results[i];
}
```

Finally, the results are averaged and returned:

```
// Get average results over all three problems
results[3] = overallScore / 3;
return results;
}
```

Implementing naive Bayes baseline

Now, when we have all the ingredients, we can replicate the naive Bayes approach that we are expected to outperform. This approach will not include any additional data pre-processing, attribute selection, and model selection. As we do not have true labels for test data, we will apply the five-fold cross validation to evaluate the model on a small dataset.

First, we initialize a naive Bayes classifier, as follows:

```
Classifier baselineNB = new NaiveBayes();
```

Next, we pass the classifier to our evaluation function, which loads the data and applies cross validation. The function returns an area under the ROC curve score for all three problems and overall results:

```
double resNB[] = evaluate(baselineNB);
System.out.println("Naive Bayes\n" +
"\tchurn: " + resNB[0] + "\n" +
"\tappetency: " + resNB[1] + "\n" +
"\tup-sell: " + resNB[2] + "\n" +
"\toverall: " + resNB[3] + "\n");
```

In our case, the model achieves the following results:

```
Naive Bayes
churn:      0.5897891153549814
appetency:  0.630778394752436
```

```
up-sell: 0.6686116692438094
overall: 0.6297263931170756
```

These results will serve as a baseline when we tackle the challenge with more advanced modeling. If we process the data with significantly more sophisticated, time-consuming, and complex techniques, we expect the results to be much better. Otherwise, we are simply wasting the resources. In general, when solving machine learning problems, it is always a good idea to create a simple baseline classifier that serves us as an orientation point.

Advanced modeling with ensembles

In the previous section, we implemented an orientation baseline, so let's focus on heavy machinery. We will follow the approach taken by the KDD Cup 2009 winning solution developed by the **IBM Research** team (Niculescu-Mizil and others, 2009).

Their strategy to address the challenge was using the **Ensemble Selection** algorithm (Caruana and Niculescu-Mizil, 2004). This is an ensemble method, which means it constructs a series of models and combines their output in a specific way to provide the final classification. It has several desirable properties as shown in the following list that make it a good fit for this challenge:

- It was proven to be robust, yielding excellent performance
- It can be optimized for a specific performance metric, including AUC
- It allows different classifiers to be added to the library
- It is an anytime method, meaning that, if we run out of time, we have a solution available

In this section, we will loosely follow the steps as described in their report. Note, this is not an exact implementation of their approach, but rather a solution overview that will include the necessary steps to dive deeper.

The general overview of steps is as follows:

1. First, we will preprocess the data by removing attributes that clearly do not bring any value, for example, all the missing or constant values; fixing missing values in order to help machine learning algorithms, which cannot deal with them; and converting categorical attributes to numerical.
2. Next, we will run attributes selection algorithm to select only a subset of attribute that can help in prediction of tasks.
3. In the third step, we will instantiate the Ensemble Selection algorithms with a wide variety of models, and, finally, evaluate the performance.

Before we start

For this task, we will need an additional Weka package, `ensembleLibrary`. Weka 3.7.2 or higher versions support external packages developed mainly by the academic community. A list of **WEKA Packages** is available at <http://weka.sourceforge.net/packageMetaData> as shown at the following screenshot:

The screenshot shows a web browser window with the title "Waikato Environment for Knowledge Analysis". The address bar shows "weka.sourceforge.net/packageMetaData". The page content includes:

- WEKA** logo and text "The University of Waikato".
- pentaho** logo and text "open source business intelligence™".
- ### WEKA Packages
- IMPORTANT:** make sure there are no old versions of Weka (<3.7.2) in your CLASSPATH before starting Weka.
- Installation of Packages**
 - A GUI package manager is available from the "Tools" menu of the GUIChooser.
 - java -jar weka.jar
 - For a command line package manager type:
java weka.core.WekaPackageManager -h
- Running packaged algorithms from the command line**

```
java weka.Run [algorithm name]
```

Substring matching is also supported. E.g. try:

```
java weka.Run Bayes
```
- Available Packages (151)**

Packages	Description	
AnDE	Classification	Averaged N-Dependence Estimators (includes A1DE and A2DE)
ArabicStemmers_LightStemmers	Preprocessing	Arabic Stemmer / Light Stemmer
CAAR	Regression, Ensemble learning	Context Aware Case-Based Regression Learner
CHIRP	Classification	CHIRP: A new classifier based on Composite Hypercubes on Iterated Random Projections
CLOPE	Clustering	CLOPE: a fast and effective clustering algorithm for transactional data
CVAttributeEval	Attribute selection	An Variation degree Algorithm to explore the space of attributes.

Find and download the latest available version of the `ensembleLibrary` package at <http://prdownloads.sourceforge.net/weka/ensembleLibrary1.0.5.zip?download>.

After you unzip the package, locate `ensembleLibrary.jar` and import it to your code, as follows:

```
import weka.classifiers.meta.EnsembleSelection;
```

Data pre-processing

First, we will utilize Weka's built-in `weka.filters.unsupervised.attribute.RemoveUseless` filter, which works exactly as its name suggests. It removes the attributes that do not vary much, for instance, all constant attributes are removed, and attributes that vary too much, almost at random. The maximum variance, which is applied only to nominal attributes, is specified with the `-M` parameter. The default parameter is 99%, which means that if more than 99% of all instances have unique attribute values, the attribute is removed, as follows:

```
RemoveUseless removeUseless = new RemoveUseless();
removeUseless.setOptions(new String[] { "-M", "99" });// threshold
removeUseless.setInputFormat(data);
data = Filter.useFilter(data, removeUseless);
```

Next, we will replace all the missing values in the dataset with the modes (nominal attributes) and means (numeric attributes) from the training data by using the `weka.filters.unsupervised.attribute.ReplaceMissingValues` filter. In general, missing values replacement should be proceeded with caution while taking into consideration the meaning and context of the attributes:

```
ReplaceMissingValues fixMissing = new ReplaceMissingValues();
fixMissing.setInputFormat(data);
data = Filter.useFilter(data, fixMissing);
```

Finally, we will discretize numeric attributes, that is, we transform numeric attributes into intervals using the `weka.filters.unsupervised.attribute.Discretize` filter. With the `-B` option, we set to split numeric attributes into four intervals, and the `-R` option will specify the range of attributes (only numeric attributes will be discretized):

```
Discretize discretizeNumeric = new Discretize();
discretizeNumeric.setOptions(new String[] {
    "-B", "4", // no of bins
    "-R", "first-last"}); //range of attributes
fixMissing.setInputFormat(data);
data = Filter.useFilter(data, fixMissing);
```

Attribute selection

In the next step, we will select only informative attributes, that is, attributes that more likely help with prediction. A standard approach to this problem is to check the information gain carried by each attribute. We will use the `weka.attributeSelection.AttributeSelection` filter, which requires two additional methods: `evaluator`, that is, how attribute usefulness is calculated, and `search` algorithms, that is, how to select a subset of attributes.

In our case, we first initialize `weka.attributeSelection.InfoGainAttributeEval` that implements calculation of information gain:

```
InfoGainAttributeEval eval = new InfoGainAttributeEval();  
Ranker search = new Ranker();
```

To select only top attributes above some threshold, we initialize `weka.attributeSelection.Ranker` to rank the attributes with information gain above a specific threshold. We specify this with the `-T` parameter, while keeping the value of the threshold low to keep the attributes with at least some information:

```
search.setOptions(new String[] { "-T", "0.001" });
```



The general rule for settings this threshold is to sort the attributes by information gain and pick the threshold where the information gain drops to negligible value.

Next, we can initialize the `AttributeSelection` class, set the evaluator and ranker, and apply the attribute selection to our dataset:

```
AttributeSelection attSelect = new AttributeSelection();  
attSelect.setEvaluator(eval);  
attSelect.setSearch(search);  
  
// apply attribute selection  
attSelect.SelectAttributes(data);
```

Finally, we remove the attributes that were not selected in the last run by calling the `reduceDimensionality(Instances)` method.

```
// remove the attributes not selected in the last run  
data = attSelect.reduceDimensionality(data);
```

At the end, we are left with 214 out of 230 attributes.

Model selection

Over the years, practitioners in the field of machine learning have developed a wide variety of learning algorithms and improvements to the existing ones. There are so many unique supervised learning methods that it is challenging to keep track of all of them. As characteristics of the datasets vary, no one method is the best in all the cases, but different algorithms are able to take advantage of different characteristics and relationships of a given dataset. The property the Ensemble Selection algorithm is to try to leverage (Jung, 2005):

Intuitively, the goal of ensemble selection algorithm is to automatically detect and combine the strengths of these unique algorithms to create a sum that is greater than the parts. This is accomplished by creating a library that is intended to be as diverse as possible to capitalize on a large number of unique learning approaches. This paradigm of overproducing a huge number of models is very different from more traditional ensemble approaches. Thus far, our results have been very encouraging.

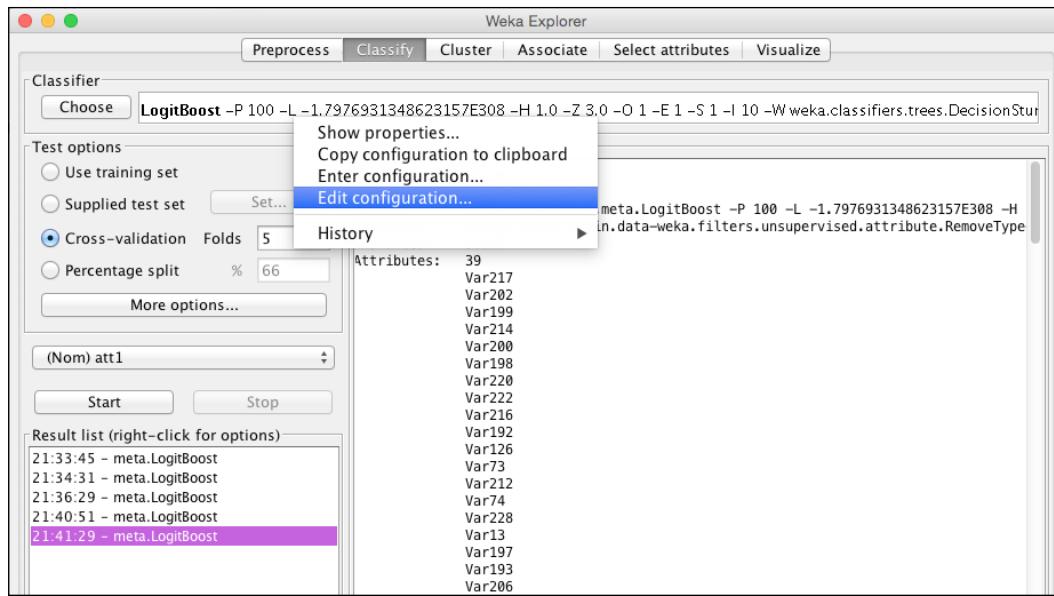
First, we need to create the model library by initializing the `weka.classifiers.EnsembleLibrary` class, which will help us define the models:

```
EnsembleLibrary ensembleLib = new EnsembleLibrary();
```

Next, we add the models and their parameters as strings to the library as string values, for example, we can add three decision tree learners with different parameters, as follows:

```
ensembleLib.addModel ("weka.classifiers.trees.J48 -S -C 0.25 -B -M  
2");  
ensembleLib.addModel ("weka.classifiers.trees.J48 -S -C 0.25 -B -M  
2 -A");
```

If you are familiar with the Weka graphical interface, you can also explore the algorithms and their configurations there and copy the configuration as shown in the following screenshot: right-click on the algorithm name and navigate to **Edit configuration | Copy configuration string**:



To complete the example, we added the following algorithms and their parameters:

- Naive Bayes that was used as default baseline:

```
ensembleLib.addModel ("weka.classifiers.bayes.NaiveBayes");
```
- k-nearest neighbors based on lazy models?:

```
ensembleLib.addModel ("weka.classifiers.lazy.IBk");
```
- Logistic regression as simple logistic with default parameters:

```
ensembleLib.addModel ("weka.classifiers.functions.SimpleLogistic");
```
- Support vector machines with default parameters:

```
ensembleLib.addModel ("weka.classifiers.functions.SMO");
```
- AdaBoost, which is an ensemble method itself:

```
ensembleLib.addModel ("weka.classifiers.meta.AdaBoostM1");
```

- LogitBoost, an ensemble method based on logistic regression:

```
ensembleLib.addModel ("weka.classifiers.meta.LogitBoost");
```
- Decision stump, an ensemble method based on one-level decision trees:

```
ensembleLib.addModel ("classifiers.trees.DecisionStump");
```

As the `EnsembleLibrary` implementation is primarily focused on GUI and console users, we have to save the models into a file by calling the `saveLibrary(File, EnsembleLibrary, JComponent)` method, as follows:

```
EnsembleLibrary.saveLibrary(new
    File(path+"ensembleLib.model.xml"), ensembleLib, null);
System.out.println(ensembleLib.getModels());
```

Next, we can initialize the Ensemble Selection algorithm by instantiating the `weka.classifiers.meta.EnsembleSelection` class. Let's first review the following method options:

- `-L </path/to/modelLibrary>`: This specifies the `modelLibrary` file, continuing the list of all models.
- `-W </path/to/working/directory>`: This specifies the working directory, where all models will be stored.
- `-B <numModelBags>`: This sets the number of bags, that is, the number of iterations to run the Ensemble Selection algorithm.
- `-E <modelRatio>`: This sets the ratio of library models that will be randomly chosen to populate each bag of models.
- `-V <validationRatio>`: This sets the ratio of the training data set that will be reserved for validation.
- `-H <hillClimbIterations>`: This sets the number of hill climbing iterations to be performed on each model bag.
- `-I <sortInitialization>`: This sets the ratio of the ensemble library that the sort initialization algorithm will be able to choose from, while initializing the ensemble for each model bag.
- `-X <numFolds>`: This sets the number of cross validation folds.
- `-P <hillclimbMetric>`: This specifies the metric that will be used for model selection during the hill climbing algorithm. Valid metrics are accuracy, rmse, roc, precision, recall, fscore, and all.

- **-A <algorithm>**: This specifies the algorithm to be used for ensemble selection. Valid algorithms are forward (default) for forward selection, backward for backward elimination, both for both forward and backward elimination, best to simply print the top performer from the ensemble library, and library to only train the models in the ensemble library.
- **-R**: This flags whether or not the models can be selected more than once for an ensemble.
- **-G**: This states whether the sort initialization greedily stops adding models when the performance degrades.
- **-O**: This is a flag for verbose output. This prints the performance of all the selected models.
- **-S <num>**: This is a random number seed (default 1).
- **-D**: If set, the classifier is run in the debug mode and may output additional information to the console.

We initialize the algorithm with the following initial parameters, where we specified optimizing the ROC metric:

```
EnsembleSelection ensambleSel = new EnsembleSelection();
ensambleSel.setOptions(new String[] {
    "-L", path+"ensembleLib.model.xml", // </path/to/modelLibrary>
    "-W", path+"esTmp", // </path/to/working/directory> -
    "-B", "10", // <numModelBags>
    "-E", "1.0", // <modelRatio>.
    "-V", "0.25", // <validationRatio>
    "-H", "100", // <hillClimbIterations>
    "-I", "1.0", // <sortInitialization>
    "-X", "2", // <numFolds>
    "-P", "roc", // <hillclimbMettric>
    "-A", "forward", // <algorithm>
    "-R", "true", // - Flag to be selected more than once
    "-G", "true", // - stops adding models when performance degrades
    "-O", "true", // - verbose output.
    "-S", "1", // <num> - Random number seed.
    "-D", "true" // - run in debug mode
});
```

Performance evaluation

The evaluation is heavy both computationally and memory-wise, so make sure that you initialize the JVM with extra heap space—for instance, **java -Xmx16g**—while the computation can take a couple of hours or days, depending on the number of algorithms you include in the model library. This example took 4 hours and 22 minutes on 12-core Intel Xeon E5-2420 CPU with 32 GB of memory, utilizing 10% CPU and 6 GB of memory on average.

We call our evaluation method and output the results, as follows:

```
double resES[] = evaluate(ensembleSel);
System.out.println("Ensemble Selection\n"
+ "\tchurn:      " + resES[0] + "\n"
+ "\tappetency: " + resES[1] + "\n"
+ "\tup-sell:   " + resES[2] + "\n"
+ "\toverall:   " + resES[3] + "\n");
```

The specific set of classifiers in the model library achieved the following result:

```
Ensamble
churn:      0.7109874158176481
appetency: 0.786325687118347
up-sell:   0.8521363243575182
overall:   0.7831498090978378
```

Overall, the approach has brought us to a significant improvement of more than 15 percentage points compared to the initial baseline that we designed at the beginning of the chapter. While it is hard to give a definite answer, the improvement was mainly due to three factors: data pre-processing and attribute selection, exploration of a large variety of learning methods, and use of an ensemble-building technique that is able to take advantage of the variety of base classifiers without overfitting. However, the improvement requires a significant increase in processing time, as well as working memory.

Summary

In this chapter, we tackled the KDD Cup 2009 challenge on customer-relationship prediction, where we implemented the data pre-processing steps, addressing the missing values and redundant attributes. We followed the winning KDD Cup solution, studying how to leverage ensemble methods using a basket of learning algorithms, which can significantly boost the classification performance.

In the next chapter, we will tackle another problem addressing the customer behavior, that is, the analysis of purchasing behavior, where you will learn how to use algorithms that detect frequently occurring patterns.

5

Affinity Analysis

Affinity analysis is the heart of **Market basket analysis (MBA)**. It can discover co-occurrence relationships among activities performed by specific users or groups. In retail, affinity analysis can help you understand the purchasing behavior of customers. These insights can drive revenue through smart cross-selling and upselling strategies and can assist you in developing loyalty programs, sales promotions, and discount plans.

In this chapter, we will look into the following topics:

- Market basket analysis
- Association rule learning
- Other applications in various domains

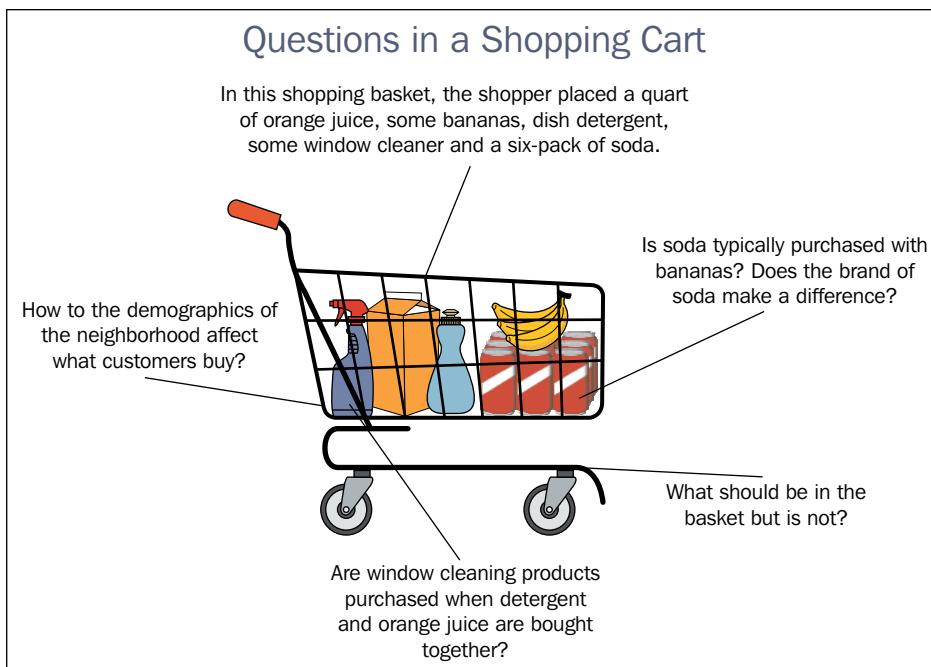
First, we will revise the core association rule learning concepts and algorithms, such as support, lift, **Apriori algorithm**, and **FP-growth algorithm**. Next, we will use Weka to perform our first affinity analysis on supermarket dataset and study how to interpret the resulting rules. We will conclude the chapter by analyzing how association rule learning can be applied in other domains, such as **IT Operations Analytics**, medicine, and others.

Market basket analysis

Since the introduction of electronic point of sale, retailers have been collecting an incredible amount of data. To leverage this data in order to produce business value, they first developed a way to consolidate and aggregate the data to understand the basics of the business. What are they selling? How many units are moving? What is the sales amount?

Affinity Analysis

Recently, the focus shifted to the lowest level of granularity – the market basket transaction. At this level of detail, the retailers have direct visibility into the market basket of each customer who shopped at their store, understanding not only the quantity of the purchased items in that particular basket, but also how these items were bought in conjunction with each other. This can be used to drive decisions about how to differentiate store assortment and merchandise, as well as effectively combine offers of multiple products, within and across categories, to drive higher sales and profits. These decisions can be implemented across an entire retail chain, by channel, at the local store level, and even for a specific customer with so-called personalized marketing, where a unique product offering is made for each customer.



MBA covers a wide variety of analysis:

- **Item affinity:** This defines the likelihood of two (or more) items being purchased together
- **Identification of driver items:** This enables the identification of the items that drive people to the store and always need to be in stock
- **Trip classification:** This analyzes the content of the basket and classifies the shopping trip into a category: weekly grocery trip, special occasion, and so on

- **Store-to-store comparison:** Understanding the number of baskets allows any metric to be divided by the total number of baskets, effectively creating a convenient and easy way to compare the stores with different characteristics (units sold per customer, revenue per transaction, number of items per basket, and so on)
- **Revenue optimization:** This helps in determining the magic price points for this store, increasing the size and value of the market basket
- **Marketing:** This helps in identifying more profitable advertising and promotions, targeting offers more precisely in order to improve ROI, generating better loyalty card promotions with longitudinal analysis, and attracting more traffic to the store
- **Operations optimization:** This helps in matching the inventory to the requirement by customizing the store and assortment to trade area demographics, optimizing store layout

Predictive models help retailers to direct the right offer to the right customer segments/profiles, as well as gain understanding of what is valid for which customer, predict the probability score of customers responding to this offer, and understand the customer value gain from the offer acceptance.

Affinity analysis

Affinity analysis is used to determine the likelihood that a set of items will be bought together. In retail, there are natural product affinities, for example, it is very typical for people who buy hamburger patties to buy hamburger rolls, along with ketchup, mustard, tomatoes, and other items that make up the burger experience.

While there are some product affinities that might seem trivial, there are some affinities that are not very obvious. A classic example is toothpaste and tuna. It seems that people who eat tuna are more prone to brush their teeth right after finishing their meal. So, why is it important for retailers to get a good grasp of the product affinities? This information is critical to appropriately plan promotions as reducing the price for some items may cause a spike on related high-affinity items without the need to further promote these related items.

In the following section, we'll look into the algorithms for association rule learning: Apriori and FP-growth.

Association rule learning

Association rule learning has been a popular approach to discover interesting relations hips between items in large databases. It is most commonly applied in retail to reveal regularities between products.

Asociation rule learning approaches find patterns as interesting strong rules in the database using different measures of interestingness. For example, the following rule would indicate that if a customer buys onions and potatoes together, they are likely to also buy hamburger meat: $\{onions, potatoes\} \rightarrow \{burger\}$

Another classic story probably told in every machine learning class is the beer and diaper story. An analysis of supermarket shoppers' behavior showed that customers, presumably young men, who buy diapers tend also to buy beer. It immediately became a popular example of how an unexpected association rule might be found from everyday data; however, there are varying opinions as to how much of the story is true. Daniel Powers says (DSS News, 2002):

"In 1992, Thomas Blischok, manager of a retail consulting group at Teradata, and his staff prepared an analysis of 1.2 million market baskets from about 25 Osco Drug stores. Database queries were developed to identify affinities. The analysis "did discover that between 5:00 and 7:00 p.m. consumers bought beer and diapers". Osco managers did NOT exploit the beer and diapers relationship by moving the products closer together on the shelves."

In addition to the preceding example from MBA, association rules are today employed in many application areas, including web usage mining, intrusion detection, continuous production, and bioinformatics. We'll take a closer look at these areas later in this chapter.

Basic concepts

Before we dive into algorithms, let's first review the basic concepts.

Database of transactions

In association rule mining, the dataset is structured a bit differently than the approach presented in the first chapter. First, there is no class value, as this is not required for learning association rules. Next, the dataset is presented as a transactional table, where each supermarket item corresponds to a binary attribute. Hence, the feature vector could be extremely large.

Consider the following example. Suppose we have four receipts as shown in the following image. Each receipt corresponds to a purchasing transaction:

WWW.MACHINE-LEARNING-JAVR.COM	WWW.MACHINE-LEARNING-JAVA.COM	WWW.MACHINE-LEARNING-JAVA.COM	WWW.MACHINE-LEARNING-JAVA.COM
GROCERY STORE 921 JAVA AVENUE NEW YORK NY 9999			
PURCHASE:	PURCHASE:	PURCHASE:	PURCHASE:
POTATOES \$4.12 BURGER \$12.04	POTATOES \$4.12 BURGER \$12.04 ONIONS \$3.14 BEER \$27.55	DIPPERS \$29.95 BEER \$27.55	BURGER \$12.04 ONIONS \$3.14 BEER \$27.55
VAT +11% TAX: \$1.77	VAT +11% TAX: \$5.15	VAT +11% TAX: \$6.38	VAT +11% TAX: \$4.78
TOTAL: \$17.98	TOTAL: \$52.08	TOTAL: \$63.88	TOTAL: \$47.48
PAYMENT METHOD: CREDIT CARD TRANSACTION #1450293367 -001 DATE:10/03/2016 9:29:27 AM	PAYMENT METHOD: CREDIT CARD TRANSACTION #1450293420 -001 DATE:10/03/2016 9:30:28 AM	PAYMENT METHOD: CREDIT CARD TRANSACTION #1450293500 -001 DATE:10/03/2016 9:31:48 AM	PAYMENT METHOD: CREDIT CARD TRANSACTION #1450293459 -001 DATE:10/03/2016 9:30:59 AM
THANK YOU	THANK YOU	THANK YOU	THANK YOU

To write these receipts in the form of a transactional database, we first identify all the possible items that appear in the receipts. These items are onions, potatoes, burger, beer, and dippers. Each purchase, that is, transaction, is presented in a row, and there is 1 if an item was purchased within the transaction and 0 otherwise, as shown in the following table:

Transaction ID	Onions	Potatoes	Burger	Beer	Dippers
1	0	1	1	0	0
2	1	1	1	1	0
3	0	0	0	1	1
4	1	0	1	1	0

This example is really small. In practical applications, the dataset often contains thousands or millions of transactions, which allow learning algorithm the discovery of statistically significant patterns.

Itemset and rule

Itemset is simply a set of items, for example, $\{onions, potatoes, burger\}$. A rule consists of two itemsets, X and Y , in the following format $X \rightarrow Y$.

This indicates a pattern that when the X itemset is observed, Y is also observed. To select interesting rules, various measures of significance can be used.

Support

Support, for an itemset, is defined as the proportion of transactions that contain the itemset. The $\{\text{potatoes}, \text{burger}\}$ itemset in the previous table has the following support as it occurs in 50% of transactions (2 out of 4 transactions) $\text{supp}(\{\text{potatoes}, \text{burger}\}) = 2/4 = 0.5$.

Intuitively, it indicates the share of transactions that support the pattern.

Confidence

Confidence of a rule indicates its accuracy. It is defined as

$$\text{Conf}(X \rightarrow Y) = \text{supp}(X \cup Y) / \text{supp}(X).$$

For example, the $\{\text{onions}, \text{burger}\} \rightarrow \{\text{beer}\}$ rule has the confidence $0.5/0.5 = 1.0$ in the previous table, which means that 100% of the times when *onions* and *burger* are bought together, *beer* is bought as well.

Apriori algorithm

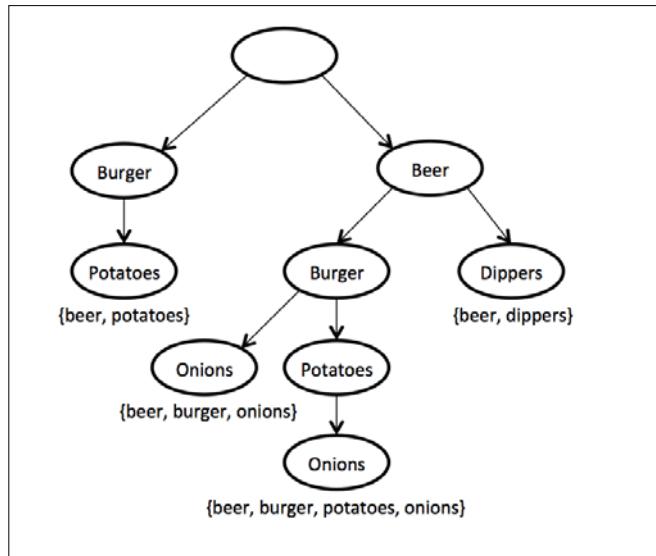
Apriori algorithm is a classic algorithm used for frequent pattern mining and association rule learning over transactional. By identifying the frequent individual items in a database and extending them to larger itemsets, Apriori can determine the association rules, which highlight general trends about a database.

Apriori algorithm constructs a set of itemsets, for example, $\text{itemset1} = \{\text{Item A}, \text{Item B}\}$, and calculates support, which counts the number of occurrences in the database. Apriori then uses a bottom-up approach, where frequent itemsets are extended, one item at a time, and it works by eliminating the largest sets as candidates by first looking at the smaller sets and recognizing that a large set cannot be frequent unless all its subsets are. The algorithm terminates when no further successful extensions are found.

Although, Apriori algorithm is an important milestone in machine learning, it suffers from a number of inefficiencies and tradeoffs. In the following section, we'll look into a more recent FP-growth technique.

FP-growth algorithm

FP-growth, where **frequent pattern (FP)**, represents the transaction database as a prefix tree. First, the algorithm counts the occurrence of items in the dataset. In the second pass, it builds a prefix tree, an ordered tree data structure commonly used to store a string. An example of prefix tree based on the previous example is shown in the following diagram:



If many transactions share most frequent items, prefix tree provides high compression close to the tree root. Large itemsets are grown directly, instead of generating candidate items and testing them against the entire database. Growth starts at the bottom of the tree, by finding all the itemsets matching minimal support and confidence. Once the recursive process has completed, all large itemsets with minimum coverage have been found and association rule creation begins.

FP-growth algorithms have several advantages. First, it constructs an FP-tree, which encodes the original dataset in a substantially compact presentation. Second, it efficiently builds frequent itemsets, leveraging the **FP-tree structure** and **divide-and-conquer** strategy.

The supermarket dataset

The supermarket dataset, located in `datasets/chap5/supermarket.arff`, describes the shopping habits of supermarket customers. Most of the attributes stand for a particular item group, for example, diary foods, beef, potatoes; or department, for example, department 79, department 81, and so on. The following image shows an excerpt of the database, where the value is t if the customer had bought an item and missing otherwise. There is one instance per customer. The dataset contains no class attribute, as this is not required to learn association rules. A sample of data is shown in the following table:

coffee	sauces-gravy-pkle	confectionary	puddings-deserts	dishcloths-scour	deod-disinfectantl	frozen foods	razor blades	fuels-garden aids	spices	jams-spreads
1	1	1	0	1	0	1	1	0	0	0
0	1	0	0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	1
1	1	0	0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0	0	1	0
0	1	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	0	1
1	0	0	0	1	0	1	0	0	0	0
0	0	0	1	0	0	0	0	0	0	1
1	1	0	0	0	0	1	0	0	0	1
0	0	0	1	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	0	0	0
0	1	0	0	1	0	1	0	1	0	0
0	0	0	0	0	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
0	1	1	1	1	0	1	0	0	0	1
0	1	1	0	0	0	0	1	0	0	0
0	1	0	0	0	0	1	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	1	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0
0	1	1	1	1	1	1	0	0	0	0

Discover patterns

To discover shopping patterns, we will use the two algorithms that we have looked into before, Apriori and FP-growth.

Apriori

We will use the Apriori algorithm as implemented in Weka. It iteratively reduces the minimum support until it finds the required number of rules with the given minimum confidence:

```
import java.io.BufferedReader;
import java.io.FileReader;
import weka.core.Instances;
import weka.associations.Apriori;
```

First, we will load the supermarket dataset:

```
Instances data = new Instances(
    new BufferedReader(
        new FileReader("datasets/chap5/supermarket.arff")));
```

Next, we will initialize an Apriori instance and call the `buildAssociations(Instances)` function to start frequent pattern mining, as follows:

```
Apriori model = new Apriori();
model.buildAssociations(data);
```

Finally, we can output the discovered itemsets and rules, as shown in the following code:

```
System.out.println(model);
```

The output is as follows:

```
Apriori
=====
Minimum support: 0.15 (694 instances)
Minimum metric <confidence>: 0.9
Number of cycles performed: 17

Generated sets of large itemsets:
Size of set of large itemsets L(1): 44
Size of set of large itemsets L(2): 380
Size of set of large itemsets L(3): 910
Size of set of large itemsets L(4): 633
Size of set of large itemsets L(5): 105
Size of set of large itemsets L(6): 1

Best rules found:

1. biscuits=t frozen foods=t fruit=t total=high 788 ==> bread and cake=t
723      <conf:(0.92)> lift:(1.27) lev:(0.03) [155] conv:(3.35)
2. baking needs=t biscuits=t fruit=t total=high 760 ==> bread and cake=t
696      <conf:(0.92)> lift:(1.27) lev:(0.03) [149] conv:(3.28)
3. baking needs=t frozen foods=t fruit=t total=high 770 ==> bread and
cake=t 705      <conf:(0.92)> lift:(1.27) lev:(0.03) [150] conv:(3.27)
...
```

The algorithm outputs ten best rules according to confidence. Let's look the first rule and interpret the output, as follows:

```
biscuits=t frozen foods=t fruit=t total=high 788 ==> bread and cake=t 723
<conf: (0.92) > lift: (1.27) lev: (0.03) [155] conv: (3.35)
```

It says that when biscuits, frozen foods, and fruits are bought together and the total purchase price is high, it is also very likely that bread and cake are purchased as well. The {biscuits, frozen foods, fruit, total high} itemset appears in 788 transactions, while the {bread, cake} itemset appears in 723 transactions. The confidence of this rule is 0.92, meaning that the rule holds true in 92% of transactions where the {biscuits, frozen foods, fruit, total high} itemset is present.

The output also reports additional measures such as lift, leverage, and conviction, which estimate the accuracy against our initial assumptions, for example, the 3.35 conviction value indicates that the rule would be incorrect 3.35 times as often if the association was purely a random chance. Lift measures the number of times X and Y occur together than expected if they were statistically independent (lift=1). The 2.16 lift in the X -> Y rule means that the probability of X is 2.16 times greater than the probability of Y.

FP-growth

Now, let's try to get the same results with more efficient FP-growth algorithm. FP-growth is also implemented in the weka.associations package:

```
import weka.associations.FPGrowth;
```

The FP-growth is initialized similarly as we did earlier:

```
FPGrowth fpgModel = new FPGrowth();
fpgModel.buildAssociations(data);
System.out.println(fpgModel);
```

The output reveals that FP-growth discovered 16 rules:

```
FPGrowth found 16 rules (displaying top 10)
```

```
1. [fruit=t, frozen foods=t, biscuits=t, total=high]: 788 ==> [bread and
cake=t]: 723   <conf: (0.92) > lift: (1.27) lev: (0.03) conv: (3.35)
2. [fruit=t, baking needs=t, biscuits=t, total=high]: 760 ==> [bread and
cake=t]: 696   <conf: (0.92) > lift: (1.27) lev: (0.03) conv: (3.28)
...
```

We can observe that FP-growth found the same set of rules as Apriori; however, the time required to process larger datasets can be significantly shorter.

Other applications in various areas

We looked into affinity analysis to demystify shopping behavior patterns in supermarkets. Although the roots of association rule learning are in analyzing point-of-sale transactions, they can be applied outside the retail industry to find relationships among other types of baskets. The notion of a basket can easily be extended to services and products, for example, to analyze items purchased using a credit card, such as rental cars and hotel rooms, and to analyze information on value-added services purchased by telecom customers (call waiting, call forwarding, DSL, speed call, and so on), which can help the operators determine the ways to improve their bundling of service packages.

Additionally, we will look into the following examples of potential cross-industry applications:

- Medical diagnosis
- Protein sequences
- Census data
- Customer relationship management
- IT Operations Analytics

Medical diagnosis

Applying association rules in medical diagnosis can be used to assist physicians while curing patients. The general problem of the induction of reliable diagnostic rules is hard as, theoretically, no induction process can guarantee the correctness of induced hypotheses by itself. Practically, diagnosis is not an easy process as it involves unreliable diagnosis tests and the presence of noise in training examples.

Nevertheless, association rules can be used to identify likely symptoms appearing together. A transaction, in this case, corresponds to a medical case, while symptoms correspond to items. When a patient is treated, a list of symptoms is recorded as one transaction.

Protein sequences

A lot of research has gone into understanding the composition and nature of proteins; yet many things remain to be understood satisfactorily. It is now generally believed that amino-acid sequences of proteins are not random.

Affinity Analysis

With association rules, it is possible to identify associations between different amino acids that are present in a protein. A protein is a sequences made up of 20 types of amino acids. Each protein has a unique three-dimensional structure, which depends on the amino-acid sequence; slight change in the sequence may change the functioning of protein. To apply association rules, a protein corresponds to a transaction, while amino acids and their structure corespond to the items.

Such association rules are desirable for enhancing our understanding of protein composition and hold the potential to give clues regarding the global interactions amongst some particular sets of amino acids occurring in the proteins. Knowledge of these association rules or constraints is highly desirable for synthesis of artificial proteins.

Census data

Censuses make a huge variety of general statistical information about the society available to both researchers and general public. The information related to population and economic census can be forecasted in planning public services (education, health, transport, and funds) as well as in business (for setting up new factories, shopping malls, or banks and even marketing particular products).

To discover frequent patterns, each statistical area (for example, municipality, city, and neighborhood) corresponds to a transaction, and the collected indicators correspond to the items.

Customer relationship management

The **customer relationship management (CRM)**, as we briefly discussed in the previous chapters, is a rich source of data through which companies hope to identify the preference of different customer groups, products, and services in order to enhance the cohesion between their products and services and their customers.

Association rules can reinforce the knowledge management process and allow the marketing personnel to know their customers well in order to provide better quality services. For example, association rules can be applied to detect a change of customer behavior at different time snapshots from customer profiles and sales data. The basic idea is to discover changes from two datasets and generate rules from each dataset to carry out rule matching.

IT Operations Analytics

Based on records of a large number of transactions, association rule learning is well-suited to be applied to the data that is routinely collected in day-to-day IT operations, enabling IT Operations Analytics tools to detect frequent patterns and identify critical changes. IT specialists need to see the big picture and understand, for example, how a problem on a database could impact an application server.

For a specific day, IT operations may take in a variety of alerts, presenting them in a transactional database. Using an association rule learning algorithm, IT Operations Analytics tools can correlate and detect the frequent patterns of alerts appearing together. This can lead to a better understanding about how a component impacts another.

With identified alert patterns, it is possible to apply predictive analytics. For example, a particular database server hosts a web application and suddenly an alert about a database is triggered. By looking into frequent patterns identified by an association rule learning algorithm, this means that the IT staff needs to take action before the web application is impacted.

Association rule learning can also discover alert events originating from the same IT event. For example, every time a new user is added, six changes in the Windows operating system are detected. Next, in the **Application Portfolio Management (APM)**, IT may face multiple alerts, showing that the transactional time in a database is high. If all these issues originate from the same source (such as getting hundreds of alerts about changes that are all due to a Windows update), this frequent pattern mining can help to quickly cut through a number of alerts, allowing the IT operators to focus on truly critical changes.

Summary

In this chapter, you learned how to leverage association rule learning on transactional datasets to gain insight about frequent patterns. We performed an affinity analysis in Weka and learned that the hard work lies in the analysis of results – careful attention is required when interpreting rules, as association (that is, correlation) is not the same as causation.

In the next chapter, we'll look at how to take the problem of item recommendation to the next level using scalable machine learning library, Apache Mahout, which is able to handle big data.

6

Recommendation Engine with Apache Mahout

Recommendation engines are probably one of the most applied data science approaches in startups today. There are two principal techniques for building a recommendation system: **content-based filtering** and **collaborative filtering**. The content-based algorithm uses the properties of the items to find items with similar properties. Collaborative filtering algorithms take user ratings or other user behavior and make recommendations based on what users with similar behavior liked or purchased.

This chapter will first explain the basic concepts required to understand recommendation engine principles and then demonstrate how to utilize Apache Mahout's implementation of various algorithms to quickly get a scalable recommendation engine. This chapter will cover the following topics:

- How to build a recommendation engine
- Getting Apache Mahout ready
- Content-based approach
- Collaborative filtering approach

By the end of the chapter, you will learn the kind of recommendation engine that is appropriate for our problem and how to quickly implement one.

Basic concepts

Recommendation engines aim to show user items of interest. What makes them different from search engines is that the relevant content usually appears on a website without requesting it and users don't have to build queries as recommendation engines observe user's actions and construct query for users without their knowledge.

Arguably, the most well-known example of recommendation engine is www.amazon.com, providing personalized recommendation in a number of ways. The following image shows an example of **Customers Who Bought This Item Also Bought**. As we will see later, this is an example of collaborative item-based recommendation, where items similar to a particular item are recommended:



An example of recommendation engine from www.amazon.com.

In this section, we will introduce key concepts related to understanding and building recommendation engines.

Key concepts

Recommendation engine requires the following four inputs to make recommendations:

- Item information described with attributes
- User profile such as age range, gender, location, friends, and so on
- User interactions in form of ratings, browsing, tagging, comparing, saving, and emailing
- Context where the items will be displayed, for example, item category and item's geographical location

These inputs are then combined together by the recommendation engine to help us answer the following questions:

- Users who bought, watched, viewed, or bookmarked this item also bought, watched, viewed, or bookmarked...
- Items similar to this item...

- Other users you may know...
- Other users who are similar to you...

Now let's have a closer look at how this combining works.

User-based and item-based analysis

Building a recommendation engine depends on whether the engine searches for related items or users when trying to recommend a particular item.

In item-based analysis, the engine focuses on identifying items that are similar to a particular item; while in user-based analysis, users similar to the particular user are first determined. For example, users with the same profile information (age, gender, and so on) or actions history (bought, watched, viewed, and so on) are determined and then the same items are recommended to other similar users.

Both approaches require us to compute a similarity matrix, depending on whether we're analyzing item attributes or user actions. Let's take a deeper look at how this is done.

Approaches to calculate similarity

There are three fundamental approaches to calculate similarity, as follows:

- Collaborative filtering algorithms take user ratings or other user behavior and make recommendations based on what users with similar behavior liked or purchased
- The content-based algorithm uses the properties of the items to find items with similar properties
- A hybrid approach combining collaborative and content-based filtering

Let's take a look at each approach in detail.

Collaborative filtering

Collaborative filtering is based solely on user ratings or other user behavior, making recommendations based on what users with similar behavior liked or purchased.

A key advantage of collaborative filtering is that it does not rely on item content, and therefore, it is capable of accurately recommending complex items such as movies, without understanding the item itself. The underlying assumption is that people who agreed in the past will agree in the future, and that they will like similar kinds of items as they liked in the past.

A major disadvantage of this approach is the so-called cold start, meaning that if we want to build an accurate collaborative filtering system, the algorithm often needs a large amount of user ratings. This usually takes collaborative filtering out of the first version of the product and it is introduced later when a decent amount of data is collected.

Content-based filtering

Content-based filtering, on the other hand, is based on a description of items and a profile of user's preferences combined as follows. First, the items are described with attributes, and to find similar items, we measure the distance between items using a distance measure such as **cosine distance** or **Pearson coefficient** (more about distance measures is in *Chapter 1, Applied Machine Learning Quick Start*). Now, the user profile enters the equation. Given the feedback about the kind of items the user likes, we can introduce weights specifying the importance of a specific item attribute. For instance, Pandora Radio streaming service applies content-based filtering to create stations using more than 400 attributes. A user initially picks a song with specific attributes, and by providing feedback, important song attributes are emphasized.

This approach initially needs very little information on user feedback, thus it effectively avoids the cold-start issue.

Hybrid approach

Now collaborative versus content-based to choose? Collaborative filtering is able to learn user preferences from user's actions regarding one content source and use them across other content types. Content-based filtering is limited to recommending content of the same type that the user is already using. This provides value to different use cases, for example, recommending news articles based on news browsing is useful, but it is much more useful if different sources such as books and movies can be recommended based on news browsing.

Collaborative filtering and content-based filtering are not mutually exclusive; they can be combined to be more effective in some cases. For example, **Netflix** uses collaborative filtering to analyze searching and watching patterns of similar users, as well as content-based filtering to offer movies that share characteristics with films that the user has rated highly.

There is a wide variety of hybridization techniques such as weighted, switching, mixed, feature combination, feature augmentation, cascade, meta-level, and so on. Recommendation systems are an active area in machine learning and data mining community with special tracks on data science conferences. A good overview of techniques is summarized in the paper *Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions* by Adomavicius and Tuzhilin (2005), where the authors discuss different approaches and underlying algorithms and provide references to further papers. To get more technical and understand all the tiny details when a particular approach makes sense, you should look at the book edited by Ricci et al. (2010) *Recommender Systems Handbook* (1st ed.), Springer-Verlag New York.

Exploitation versus exploration

In recommendation system, there is always a tradeoff between recommending items that fall into the user's sweet spot based on what we already know about the user (**exploitation**) and recommending items that don't fall into user's sweet spot with the aim to expose user to some novelties (**exploration**). Recommendation systems with little exploration will only recommend items consistent with the previous user ratings, preventing showing items outside their current bubble. In practice, serendipity of getting new items out of user's sweet spot is often desirable, leading to pleasant surprise and potential discovery of new sweet spots.

In this section, we discussed the essential concepts required to start building recommendation engines. Now, let's take a look at how to actually build one with Apache Mahout.

Getting Apache Mahout

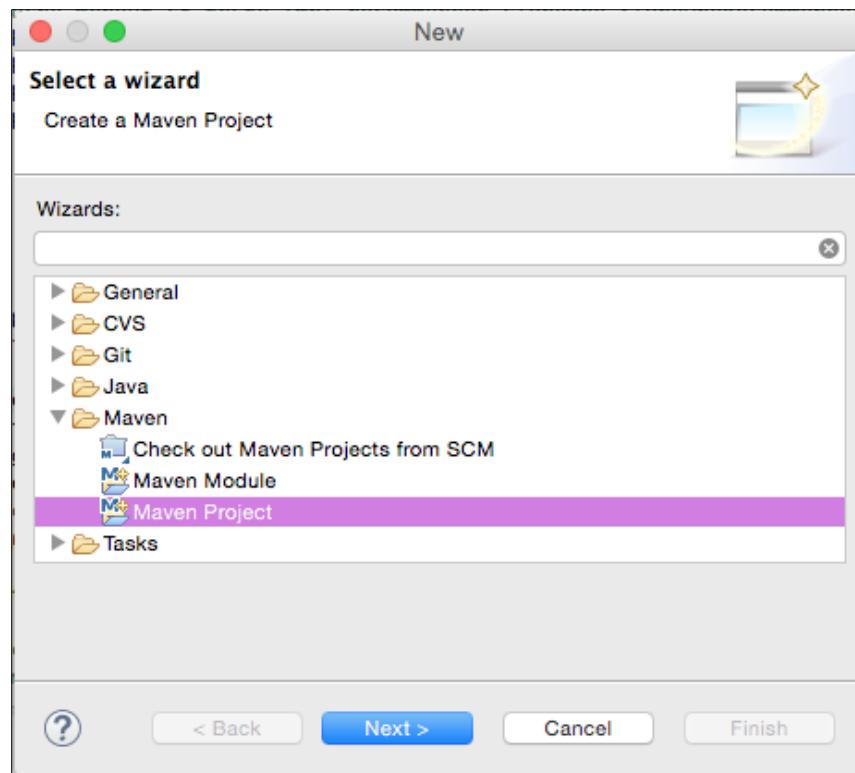
Mahout was introduced in *Chapter 2, Java Tools and Libraries for Machine Learning*, as a scalable machine learning library. It provides a rich set of components with which you can construct a customized recommendation system from a selection of algorithms. The creators of Mahout say it is designed to be enterprise-ready; it's designed for performance, scalability, and flexibility.

Mahout can be configured to run in two flavors: with or without Hadoop for a single machine and distributed processing, correspondingly. We will focus on configuring Mahout without Hadoop. For more advanced configurations and further uses of Mahout, I would recommend two recent books: *Learning Apache Mahout* (Tiwary, 2015) and *Learning Apache Mahout Classification* (Gupta, 2015).

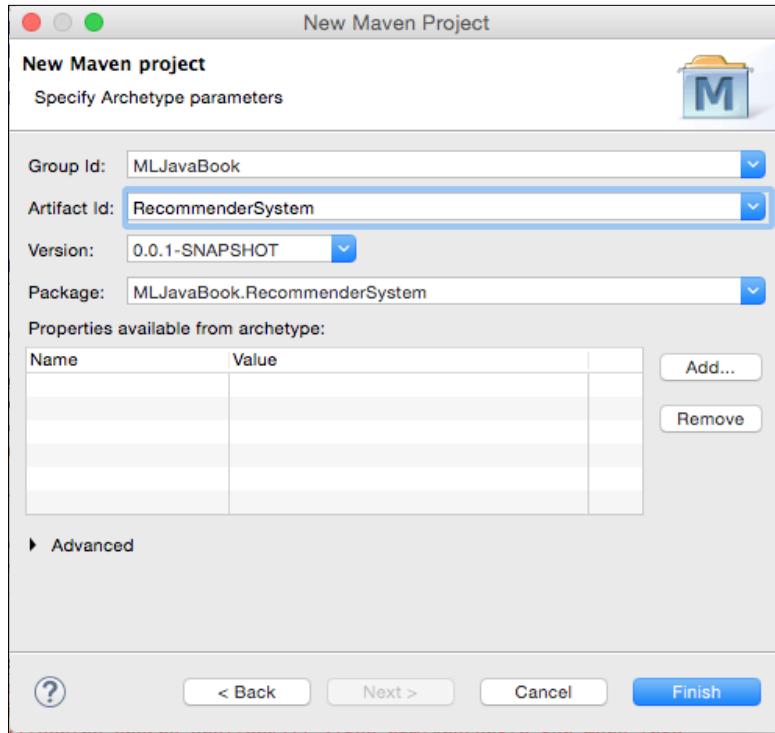
As Apache Mahout's build and release system is based on Maven, we will need to learn how to install it. We will look at the most convenient approach using Eclipse with Maven plugin.

Configuring Mahout in Eclipse with the Maven plugin

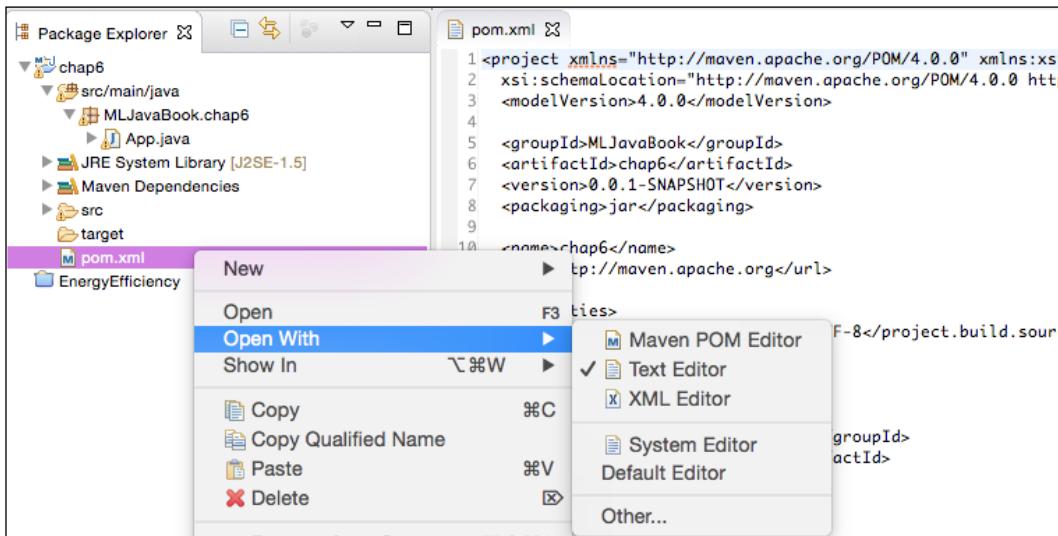
We will need a recent version of Eclipse, which can be downloaded from its home page. We use Eclipse Luna in this book. Open Eclipse and start a new **Maven Project** with default settings as shown in the following screenshot:



The **New Maven project** screen will appear as shown in the following image:



Now, we need to tell the project to add Mahout jar and its dependencies to the project. Locate the `pom.xml` file and open it with the text editor (left click on **Open With | Text Editor**), as shown in the following screenshot:



Locate the line starting with <dependencies> and add the following code in the next line:

```
<dependency>
    <groupId>org.apache.mahout</groupId>
    <artifactId>mahout-mr</artifactId>
    <version>0.10.0</version>
</dependency>
```

That's it, Mahout is added and we are ready to begin now.

Building a recommendation engine

To demonstrate both the content-based filtering and collaborative filtering approaches, we'll build a book-recommendation engine.

Book ratings dataset

In this chapter, we will work with book ratings dataset (Ziegler et al, 2005) collected in a four-week crawl. It contains data on 278,858 members of the **Book-Crossing** website and 1,157,112 ratings, both implicit and explicit, referring to 271,379 distinct ISBNs. User data is anonymized, but with demographic information. The dataset is available at:

<http://www2.informatik.uni-freiburg.de/~cziegler/BX/>.

The Book-Crossing dataset comprises three files described at their website as follows:

- **BX-Users:** This contains the users. Note that user IDs (User-ID) have been anonymized and mapped to integers. Demographic data is provided (Location and Age) if available. Otherwise, these fields contain NULL-values.
- **BX-Books:** Books are identified by their respective ISBN. Invalid ISBNs have already been removed from the dataset. Moreover, some content-based information is given (Book-Title, Book-Author, Year-Of-Publication, and Publisher), obtained from Amazon Web Services. Note that in case of several authors, only the first author is provided. URLs linking to cover images are also given, appearing in three different flavors (Image-URL-S, Image-URL-M, and Image-URL-L), that is, small, medium, and large. These URLs point to the Amazon website.
- **BX-Book-Ratings:** This contains the book rating information. Ratings (Book-Rating) are either explicit, expressed on a scale of 1-10 (higher values denoting higher appreciation), or implicit, expressed by 0.

Loading the data

There are two approaches for loading the data according to where the data is stored: file or database. First, we will take a detailed look at how to load the data from the file, including how to deal with custom formats. At the end, we quickly take a look at how to load the data from a database.

Loading data from file

Loading data from file can be achieved with the `FileDataModel` class, expecting a comma-delimited file, where each line contains a `userID`, `itemID`, optional preference value, and optional timestamp in the same order, as follows:

```
userID, itemID [, preference [, timestamp]]
```

Optional preference accommodates applications with binary preference values, that is, user either expresses a preference for an item or not, without degree of preference, for example, with like/dislike.

A line that begins with hash, #, or an empty line will be ignored. It is also acceptable for the lines to contain additional fields, which will be ignored.

The `DataModel` class assumes the following types:

- `userID`, `itemID` can be parsed as `long`
- preference value can be parsed as `double`
- `timestamp` can be parsed as `long`

If you are able to provide the dataset in the preceding format, you can simply use the following line to load the data:

```
DataModel model = new FileDataModel(new File(path));
```

This class is not intended to be used for very large amounts of data, for example, tens of millions of rows. For that, a JDBC-backed `DataModel` and a database are more appropriate.

In real world, however, we cannot always ensure that the input data supplied to us contain only integer values for `userID` and `itemID`. For example, in our case, `itemID` correspond to ISBN book numbers uniquely identifying items, but these are not integers and the `FileDataModel` default won't be suitable to process our data.

Now, let's consider how to deal with the case where our `itemID` is a string. We will define our custom data model by extending `FileDataModel` and overriding the long `readItemIDFromString(String)` method in order to read `itemID` as a string and convert it into `long` and return a unique long value. To convert `String` to unique `long`, we'll extend another Mahout `AbstractIDMigrator` helper class, which is designed exactly for this task.

Now, let's first look at how `FileDataModel` is extended:

```
class StringItemIdFileDataModel extends FileDataModel {

    //initialize migrator to convert String to unique long
    public ItemMemIDMigrator memIdMigtr;

    public StringItemIdFileDataModel(File dataFile, String regex)
        throws IOException {
        super(dataFile, regex);
    }

    @Override
    protected long readItemIDFromString(String value) {

        if (memIdMigtr == null) {
            memIdMigtr = new ItemMemIDMigrator();
        }

        // convert to long
        long retValue = memIdMigtr.toLongID(value);
        //store it to cache
        if (null == memIdMigtr.toStringID(retValue)) {
            try {
                memIdMigtr.singleInit(value);
            } catch (TasteException e) {
                e.printStackTrace();
            }
        }
        return retValue;
    }

    // convert long back to String
    String getItemIDAsString(long itemId) {
        return memIdMigtr.toStringID(itemId);
    }
}
```

Other useful methods that can be overridden are as follows:

- `readUserIDFromString(String value)` if user IDs are not numeric
- `readTimestampFromString(String value)` to change how timestamp is parsed

Now, let's take a look how `AbstractIDMigrator` is extended:

```
class ItemMemIDMigrator extends AbstractIDMigrator {

    private FastByIDMap<String> longToString;

    public ItemMemIDMigrator() {
        this.longToString = new FastByIDMap<String>(10000);
    }

    public void storeMapping(long longID, String stringID) {
        longToString.put(longID, stringID);
    }

    public void singleInit(String stringID) throws TasteException {
        storeMapping(toLongID(stringID), stringID);
    }

    public String toStringID(long longID) {
        return longToString.get(longID);
    }
}
```

Now, we have everything in place and we can load our dataset with the following code:

```
StringItemIdFileDataModel model = new StringItemIdFileDataModel(
    new File("datasets/chap6/BX-Book-Ratings.csv"), ";");
System.out.println(
    "Total items: " + model.getNumItems() +
    "\nTotal users: " + model.getNumUsers());
```

This outputs the total number of users and items:

```
Total items: 340556
Total users: 105283
```

We are ready to move on and start making recommendations.

Loading data from database

Alternately, we can also load the data from database using one of the JDBC data models. In this chapter, we will not dive into the detailed instructions about how to set up database, connection, and so on, but just give a sketch on how this can be done.

Database connectors have been moved to a separate package `mahout-integration`, hence we have to first add the package to our dependency list. Open the `pom.xml` file and add the following dependency:

```
<dependency>
    <groupId>org.apache.mahout</groupId>
    <artifactId>mahout-integration</artifactId>
    <version>0.7</version>
</dependency>
```

Consider that we want to connect a MySQL database. In this case, we will also need a package that handles database connections. Add the following to the `pom.xml` file:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.35</version>
</dependency>
```

Now, we have all the packages, so we can create a connection. First, let's initialize a `DataSource` class with connection details, as follows:

```
MysqlDataSource dbsource = new MysqlDataSource();
dbsource.setUser("user");
dbsource.setPassword("pass");
dbsource.setServerName("hostname.com");
dbsource.setDatabaseName("db");
```

Mahout integration implements `JDBCDataModel` to various databases that can be accessed via JDBC. By default, this class assumes that there is `DataSource` available under the JNDI name `jdbc/taste`, which gives access to a database with a `taste_preferences` table with the following schema:

```
CREATE TABLE taste_preferences (
    user_id BIGINT NOT NULL,
    item_id BIGINT NOT NULL,
    preference REAL NOT NULL,
    PRIMARY KEY (user_id, item_id)
)
```

```
CREATE INDEX taste_preferences_user_id_index ON taste_preferences
    (user_id);
CREATE INDEX taste_preferences_item_id_index ON taste_preferences
    (item_id);
```

A database-backed data model is initialized as follows. In addition to the DB connection object, we can also specify the custom table name and table column names, as follows:

```
DataModel dataModel = new MySQLJDBCDataModel(dbsource,
    "taste_preferences",
    "user_id", "item_id", "preference", "timestamp");
```

In-memory database

Last, but not least, the data model can be created on the fly and held in memory. A database can be created from an array of preferences holding user ratings for a set of items.

We can proceed as follows. First, we create a `FastByIdMap` hash map of preference arrays, `PreferenceArray`, which stores an array of preferences:

```
FastByIdMap<PreferenceArray> preferences = new FastByIdMap<PreferenceArray>();
```

Next, we can create a new preference array for a user that will hold their ratings. The array must be initialized with a size parameter that reserves that many slots in memory:

```
PreferenceArray prefsForUser1 =
    new GenericUserPreferenceArray(10);
```

Next, we set user ID for current preference at position 0. This will actually set the user ID for all preferences:

```
prefsForUser1.setUserID(0, 1L);
```

Set item ID for current preference at position 0:

```
prefsForUser1.setItemID(0, 101L);
```

Set preference value for preference at 0:

```
prefsForUser1.setValue(0, 3.0f);
```

Continue for other item ratings:

```
prefsForUser1.setItemID (1, 102L);  
prefsForUser1.setValue (1, 4.5F);
```

Finally, add user preferences to the hash map:

```
preferences.put (1L, prefsForUser1); // use userID as the key
```

The preference hash map can be now used to initialize GenericDataModel:

```
DataModel dataModel = new GenericDataModel (preferences);
```

This code demonstrates how to add two preferences for a single user; while in practical application, you'll want to add multiple preferences for multiple users.

Collaborative filtering

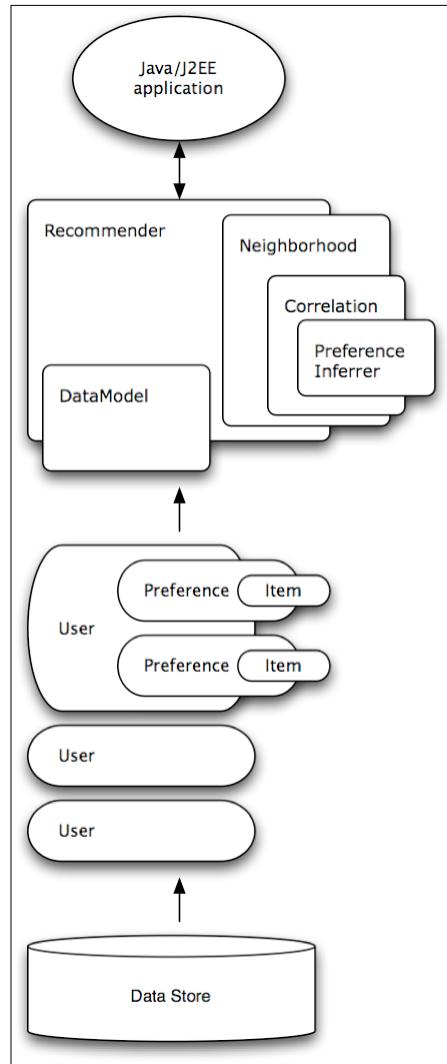
Recommendation engines in Mahout can be built with the `org.apache.mahout.cf.taste` package, which was formerly a separate project called Taste and has continued development in Mahout.

A Mahout-based collaborative filtering engine takes the users' preferences for items (tastes) and returns the estimated preferences for other items. For example, a site that sells books or CDs could easily use Mahout to figure out the CDs that a customer might be interested in listening to with the help of the previous purchase data.

Top-level packages define the Mahout interfaces to the following key abstractions:

- **DataModel:** This represents a repository of information about users and their preferences for items
- **UserSimilarity:** This defines a notion of similarity between two users
- **ItemSimilarity:** This defines a notion of similarity between two items
- **UserNeighborhood:** This computes neighborhood users for a given user
- **Recommender:** This recommends items for user

A general structure of the concepts is shown in the following diagram:



User-based filtering

The most basic user-based collaborative filtering can be implemented by initializing the previously described components as follows.

First, load the data model:

```
StringItemIdFileDataModel model = new StringItemIdFileDataModel(
    new File("/datasets/chap6/BX-Book-Ratings.csv", ";"));
```

Next, define how to calculate how the users are correlated, for example, using Pearson correlation:

```
UserSimilarity similarity =
    new PearsonCorrelationSimilarity(model);
```

Next, define how to tell which users are similar, that is, users that are close to each other according to their ratings:

```
UserNeighborhood neighborhood =
    new ThresholdUserNeighborhood(0.1, similarity, model);
```

Now, we can initialize a GenericUserBasedRecommender default engine with data model, neighborhood, and similar objects, as follows:

```
UserBasedRecommender recommender =
    new GenericUserBasedRecommender(model, neighborhood, similarity);
```

That's it. Our first basic recommendation engine is ready. Let's discuss how to invoke recommendations. First, let's print the items that the user already rated along with ten recommendations for this user:

```
long userID = 80683;
int noItems = 10;

List<RecommendedItem> recommendations = recommender.recommend(
    userID, noItems);

System.out.println("Rated items by user:");
for(Preference preference : model.getPreferencesFromUser(userID)) {
    // convert long itemID back to ISBN
    String itemISBN = model.getItemIDAsString(
        preference.getItemId());
    System.out.println("Item: " + books.get(itemISBN) +
        " | Item id: " + itemISBN +
        " | Value: " + preference.getValue());
}

System.out.println("\nRecommended items:");
for (RecommendedItem item : recommendations) {
    String itemISBN = model.getItemIDAsString(item.getItemId());
    System.out.println("Item: " + books.get(itemISBN) +
        " | Item id: " + itemISBN +
        " | Value: " + item.getValue());
}
```

This outputs the following recommendations along with their scores:

Rated items:

```
Item: The Handmaid's Tale | Item id: 0395404258 | Value: 0.0
Item: Get Clark Smart : The Ultimate Guide for the Savvy Consumer | Item id: 1563526298 | Value: 9.0
Item: Plum Island | Item id: 0446605409 | Value: 0.0
Item: Blessings | Item id: 0440206529 | Value: 0.0
Item: Edgar Cayce on the Akashic Records: The Book of Life | Item id: 0876044011 | Value: 0.0
Item: Winter Moon | Item id: 0345386108 | Value: 6.0
Item: Sarah Bishop | Item id: 059032120X | Value: 0.0
Item: Case of Lucy Bending | Item id: 0425060772 | Value: 0.0
Item: A Desert of Pure Feeling (Vintage Contemporaries) | Item id: 0679752714 | Value: 0.0
Item: White Abacus | Item id: 0380796155 | Value: 5.0
Item: The Land of Laughs : A Novel | Item id: 0312873115 | Value: 0.0
Item: Nobody's Son | Item id: 0152022597 | Value: 0.0
Item: Mirror Image | Item id: 0446353957 | Value: 0.0
Item: All I Really Need to Know | Item id: 080410526X | Value: 0.0
Item: Dreamcatcher | Item id: 0743211383 | Value: 7.0
Item: Perplexing Lateral Thinking Puzzles: Scholastic Edition | Item id: 0806917695 | Value: 5.0
Item: Obsidian Butterfly | Item id: 0441007813 | Value: 0.0
```

Recommended items:

```
Item: Keeper of the Heart | Item id: 0380774933 | Value: 10.0
Item: Bleachers | Item id: 0385511612 | Value: 10.0
Item: Salem's Lot | Item id: 0451125452 | Value: 10.0
Item: The Girl Who Loved Tom Gordon | Item id: 0671042858 | Value: 10.0
Item: Mind Prey | Item id: 0425152898 | Value: 10.0
Item: It Came From The Far Side | Item id: 0836220730 | Value: 10.0
Item: Faith of the Fallen (Sword of Truth, Book 6) | Item id: 081257639X | Value: 10.0
Item: The Talisman | Item id: 0345444884 | Value: 9.86375
Item: Hamlet | Item id: 067172262X | Value: 9.708363
Item: Untamed | Item id: 0380769530 | Value: 9.708363
```

Item-based filtering

The `ItemSimilarity` is the most important point to discuss here. Item-based recommenders are useful as they can take advantage of something very fast: they base their computations on item similarity, not user similarity, and item similarity is relatively static. It can be precomputed, instead of recomputed in real time.

Thus, it's strongly recommended that you use `GenericItemSimilarity` with precomputed similarities if you're going to use this class. You can use `PearsonCorrelationSimilarity` too, which computes similarities in real time, but you will probably find this painfully slow for large amounts of data:

```
StringItemIdFileDataModel model = new StringItemIdFileDataModel(  
    new File("datasets/chap6/BX-Book-Ratings.csv"), ";");  
  
ItemSimilarity itemSimilarity = new  
    PearsonCorrelationSimilarity(model);  
  
ItemBasedRecommender recommender = new  
    GenericItemBasedRecommender(model, itemSimilarity);  
  
String itemISBN = "0395272238";  
long itemID = model.readItemIDFromString(itemISBN);  
int noItems = 10;  
List<RecommendedItem> recommendations =  
    recommender.mostSimilarItems(itemID, noItems);  
  
System.out.println("Recommendations for item:  
    "+books.get(itemISBN));  
  
System.out.println("\nMost similar items:");
for (RecommendedItem item : recommendations) {
    itemISBN = model.getItemIDAsString(item.getItemId());
    System.out.println("Item: " + books.get(itemISBN) + " | Item id:  

        " + itemISBN + " | Value: " + item.getValue());
}
```

Recommendations for item: Close to the Bone

```
Most similar items:  
Item: Private Screening | Item id: 0345311396 | Value: 1.0  
Item: Heartstone | Item id: 0553569783 | Value: 1.0  
Item: Clockers / Movie Tie In | Item id: 0380720817 | Value: 1.0  
Item: Rules of Prey | Item id: 0425121631 | Value: 1.0  
Item: The Next President | Item id: 0553576666 | Value: 1.0  
Item: Orchid Beach (Holly Barker Novels (Paperback)) | Item id:  
0061013412 | Value: 1.0
```

```
Item: Winter Prey | Item id: 0425141233 | Value: 1.0
Item: Night Prey | Item id: 0425146413 | Value: 1.0
Item: Presumed Innocent | Item id: 0446359866 | Value: 1.0
Item: Dirty Work (Stone Barrington Novels (Paperback)) | Item id:
0451210158 | Value: 1.0
```

The resulting list returns a set of items similar to particular item that we selected.

Adding custom rules to recommendations

It often happens that some business rules require us to boost the score of the selected items. In the book dataset, for example, if a book is recent, we want to give it a higher score. That's possible using the `IDRescorer` interface implementing, as follows:

- `rescore(long, double)` that takes `itemId` and original score as an argument and returns a modified score
- `isFiltered(long)` that may return `true` to exclude a specific item from recommendation or `false` otherwise

Our example could be implemented as follows:

```
class MyRescorer implements IDRescorer {

    public double rescore(long itemId, double originalScore) {
        double newScore = originalScore;
        if(bookIsNew(itemId)){
            originalScore *= 1.3;
        }
        return newScore;
    }

    public boolean isFiltered(long arg0) {
        return false;
    }

}
```

An instance of `IDRescorer` is provided when invoking `recommender.recommend`:

```
IDRescorer rescorer = new MyRescorer();
List<RecommendedItem> recommendations =
recommender.recommend(userID, noItems, rescorer);
```

Evaluation

You might wonder how to make sure that the returned recommendations make any sense? The only way to be really sure about how effective recommendations are is to use A/B testing in a live system with real users. For example, the A group receives a random item as a recommendation, while the B group receives an item recommended by our engine.

As this is neither always possible nor practical, we can get an estimate with offline statistical evaluation. One way to proceed is to use the k-fold cross validation introduced in *Chapter 1, Applied Machine Learning Quick Start*. We partition dataset into multiple sets, some are used to train our recommendation engine and the rest to test how well it recommends items to unknown users.

Mahout implements the `RecommenderEvaluator` class that splits a dataset in two parts. The first part, 90% by default, is used to produce recommendations, while the rest of the data is compared against estimated preference values to test the match. The class does not accept a `recommender` object directly, you need to build a class implementing the `RecommenderBuilder` interface instead, which builds a `recommender` object for a given `DataModel` object that is then used for testing. Let's take a look at how this is implemented.

First, we create a class that implements the `RecommenderBuilder` interface. We need to implement the `buildRecommender` method, which will return a `recommender`, as follows:

```
public class BookRecommender implements RecommenderBuilder {
    public Recommender buildRecommender(DataModel dataModel) {
        UserSimilarity similarity =
            new PearsonCorrelationSimilarity(model);
        UserNeighborhood neighborhood =
            new ThresholdUserNeighborhood(0.1, similarity, model);
        UserBasedRecommender recommender =
            new GenericUserBasedRecommender(
                model, neighborhood, similarity);
        return recommender;
    }
}
```

Now that we have class that returns a `recommender` object, we can initialize a `RecommenderEvaluator` instance. Default implementation of this class is the `AverageAbsoluteDifferenceRecommenderEvaluator` class, which computes the average absolute difference between the predicted and actual ratings for users. The following code shows how to put the pieces together and run a hold-out test.

First, load a data model:

```
DataModel dataModel = new FileDataModel(  
    new File("/path/to/dataset.csv"));
```

Next, initialize an evaluator instance, as follows:

```
RecommenderEvaluator evaluator =  
    new AverageAbsoluteDifferenceRecommenderEvaluator();
```

Initialize the BookRecommender object, implementing the RecommenderBuilder interface:

```
RecommenderBuilder builder = new MyRecommenderBuilder();
```

Finally, call the `evaluate()` method, which accepts the following parameters:

- `RecommenderBuilder`: This is the object implementing `RecommenderBuilder` that can build recommender to test
- `DataModelBuilder`: `DataModelBuilder` to use, or if null, a default `DataModel` implementation will be used
- `DataModel`: This is the dataset that will be used for testing
- `trainingPercentage`: This indicates the percentage of each user's preferences to use to produced recommendations; the rest are compared to estimated preference values to evaluate the recommender performance
- `evaluationPercentage`: This is the percentage of users to be used in evaluation

The method is called as follows:

```
double result = evaluator.evaluate(builder, null, model, 0.9,  
    1.0);  
System.out.println(result);
```

The method returns a double, where 0 presents the best possible evaluation, meaning that the recommender perfectly matches user preferences. In general, lower the value, better the match.

Online learning engine

What about the online aspect? The above will work great for existing users; but what about new users which register in the service? For sure, we want to provide some reasonable recommendations for them as well. Creating a recommendation instance is expensive (it definitely takes longer than a usual network request), so we can't just create a new recommendation each time.

Luckily, Mahout has a possibility of adding temporary users to a data model. The general set up is as follows:

- Periodically recreate the whole recommendation using current data (for example, each day or hour, depending on how long it takes)
- When doing a recommendation, check whether the user exists in the system
- If yes, complete the recommendation as always
- If no, create a temporary user, fill in the preferences, and do the recommendation

The first part (periodically recreating the recommender) may be actually quite tricky if you are limited on memory: when creating the new recommender, you need to hold two copies of the data in memory (in order to still be able to serve requests from the old one). However, as this doesn't really have anything to do with recommendations, I won't go into details here.

As for the temporary users, we can wrap our data model with a `PlusAnonymousConcurrentUserDataModel` instance. This class allows us to obtain a temporary user ID; the ID must be later released so that it can be reused (there's a limited number of such IDs). After obtaining the ID, we have to fill in the preferences, and then, we can proceed with the recommendation as always:

```
class OnlineRecommendation{  
  
    Recommender recommender;  
    int concurrentUsers = 100;  
    int noItems = 10;  
  
    public OnlineRecommendation() throws IOException {  
  
        DataModel model = new StringItemIdFileDataModel(  
            new File "/chap6/BX-Book-Ratings.csv"), ";");  
        PlusAnonymousConcurrentUserDataModel plusModel = new  
            PlusAnonymousConcurrentUserDataModel  
                (model, concurrentUsers);  
        recommender = ...;  
    }  
  
    public List<RecommendedItem> recommend(long userId,  
        PreferenceArray preferences){  
  
        if(userExistsInDataModel(userId)){  
    }
```

```

        return recommender.recommend(userID, noItems);
    }

    else{

        PlusAnonymousConcurrentUserDataModel plusModel =
            (PlusAnonymousConcurrentUserDataModel)
                recommender.getDataModel();

        // Take an available anonymous user from the poll
        Long anonymousUserID = plusModel.takeAvailableUser();

        // Set temporary preferences
        PreferenceArray tempPrefs = preferences;
        tempPrefs.setUserID(0, anonymousUserID);
        tempPrefs.setItemID(0, itemID);
        plusModel.setTempPrefs(tempPrefs, anonymousUserID);

        List<RecommendedItem> results =
            recommender.recommend(anonymousUserID, noItems);

        // Release the user back to the poll
        plusModel.releaseUser(anonymousUserID);

        return results;
    }
}
}

```

Content-based filtering

Content-based filtering is out of scope in the Mahout framework, mainly because it is up to you to decide how to define similar items. If we want to do a content-based item-item similarity, we need to implement our own `ItemSimilarity`. For instance, in our book's dataset, we might want to make up the following rule for book similarity:

- If genres are the same, add 0.15 to similarity
- If author is the same, add 0.50 to similarity

We could now implement our own similarity measure as follows:

```
class MyItemSimilarity implements ItemSimilarity {  
    ...  
    public double itemSimilarity(long itemID1, long itemID2) {  
        MyBook book1 = lookupMyBook (itemID1);  
        MyBook book2 = lookupMyBook (itemID2);  
        double similarity = 0.0;  
        if (book1.getGenre () .equals(book2.getGenre ()) )  
            similarity += 0.15;  
        }  
        if (book1.getAuthor () .equals(book2. getAuthor ())) {  
            similarity += 0.50;  
        }  
        return similarity;  
    }  
    ...  
}
```

We then use this `ItemSimilarity` instead of something like `LogLikelihoodSimilarity` or other implementations with a `GenericItemBasedRecommender`. That's about it. This is as far as we have to go to perform content-based recommendation in the Mahout framework.

What we saw here is one of the simplest forms of content-based recommendation. Another approach could be to create a content-based profile of users, based on a weighted vector of item features. The weights denote the importance of each feature to the user and can be computed from individually-rated content vectors.

Summary

In this chapter, you learned the basic concepts of recommendation engines, the difference between collaborative and content-based filtering, and how to use Apache Mahout, which is a great basis to create recommenders as it is very configurable and provides many extension points. We looked at how to pick the right configuration parameter values, set up resoring, and evaluate the recommendation results.

With this chapter, we completed data science techniques to analyze customer behavior that started with customer-relationship prediction in *Chapter 4, Customer Relationship Prediction with Ensembles*, and continued with affinity analytics in *Chapter 5, Affinity Analysis*. In the next chapter, we will move on to other topics, such as fraud and anomaly detection.

7

Fraud and Anomaly Detection

Outlier detection is used to identify exceptions, rare events, or other anomalous situations. Such anomalies may be hard-to-find needles in a haystack, but their consequences may nonetheless be quite dramatic, for instance, credit card fraud detection, identifying network intrusion, faults in a manufacturing processes, clinical trials, voting activities, and criminal activities in e-commerce. Therefore, discovered anomalies represent high value when they are found or high costs if they are not found. Applying machine learning to outlier detection problems brings new insight and better detection of outlier events. Machine learning can take into account many disparate sources of data and find correlations that are too obscure for human analysis to identify.

Take the example of e-commerce fraud detection. With machine learning algorithm in place, the purchaser's online behavior, that is, website browsing history, becomes a part of the fraud detection algorithm rather than simply considering the history of purchases made by the cardholder. This involves analyzing a variety of data sources, but it is also a far more robust approach to e-commerce fraud detection.

In this chapter, we will cover the following topics:

- Problems and challenges
- Suspicious pattern detection
- Anomalous pattern detection
- Working with unbalanced datasets
- Anomaly detection in time series

Suspicious and anomalous behavior detection

The problem of learning patterns from sensor data arises in many applications, including e-commerce, smart environments, video surveillance, network analysis, human-robot interaction, ambient assisted living, and so on. We focus on detecting patterns that deviate from regular behaviors and might represent a security risk, health problem, or any other abnormal behavior contingency.

In other words, deviant behavior is a data pattern that either does not conform to the expected behavior (anomalous behavior) or matches a previously defined unwanted behavior (suspicious behavior). Deviant behavior patterns are also referred to as outliers, exceptions, peculiarities, surprise, misuse, and so on. Such patterns relatively occur infrequently; however, when they do occur, their consequences can be quite dramatic, and often negatively so. Typical examples include credit card fraud detection, cyber-intrusions, and industrial damage. In e-commerce, fraud is estimated to cost merchants more than \$200 billion a year; in healthcare, fraud is estimated to cost taxpayers \$60 billion a year; for banks, the cost is over \$12 billion.

Unknown-unknowns

When Donald Rumsfeld, US Secretary of Defense, had a news briefing on February 12, 2002, about the lack of evidence linking the government of Iraq to the supply of weapons of mass destruction to terrorist groups, it immediately became a subject of much commentary. Rumsfeld stated (DoD News, 2012):

"Reports that say that something hasn't happened are always interesting to me, because as we know, there are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns – the ones we don't know we don't know. And if one looks throughout the history of our country and other free countries, it is the latter category that tend to be the difficult ones."

The statement might seem confusing at first, but the idea of unknown unknowns was well studied among scholars dealing with risk, NSA, and other intelligence agencies. What the statement basically says is the following:

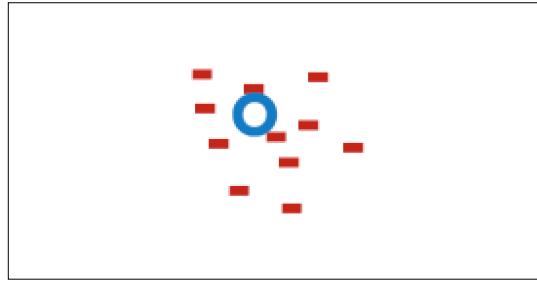
- **Known-knowns:** These are well-known problems or issues we know how to recognize them and how deal with them
- **Known-unknowns:** These are expected or foreseeable problems, which can be reasonably anticipated, but have not occurred before

- **Unknown-unknowns:** These are unexpected and unforeseeable problems, which pose significant risk as they cannot be anticipated, based on previous experience

In the following sections, we will look into two fundamental approaches dealing with the first two types of knowns and unknowns: suspicious pattern detection dealing with known-knowns and anomalous pattern detection targeting known-unknowns.

Suspicious pattern detection

The first approach assumes a behavior library that encodes negative patterns shown as red minus signs in the following image, and thus recognizing that observed behavior corresponds to identifying a match in the library. If a new pattern (blue circle) can be matched against negative patterns, then it is considered suspicious:



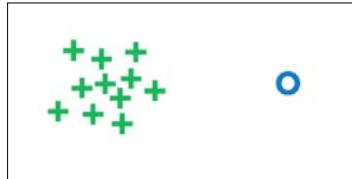
For example, when you visit a doctor, she inspects various health symptoms (body temperature, pain levels, affected areas, and so on) and matches the symptoms to a known disease. In machine learning terms, the doctor collects attributes and performs classifications.

An advantage of this approach is that we immediately know what is wrong; for example, assuming we know the disease, we can select appropriate treatment procedure.

A major disadvantage of this approach is that it can detect only suspicious patterns that are known in advance. If a pattern is not inserted into a negative pattern library, then we will not be able to recognize it. This approach is, therefore, appropriate for modeling known-knowns.

Anomalous pattern detection

The second approach uses the pattern library in an inverse fashion, meaning that the library encodes only positive patterns marked with green plus signs in the following image. When an observed behavior (blue circle) cannot be matched against the library, it is considered anomalous:



This approach requires us to model only what we have seen in the past, that is, normal patterns. If we return to the doctor example, the main reason we visited the doctor in the first place was because we did not feel fine. Our perceived state of feelings (for example, headache, sore skin) did not match our usual feelings, therefore, we decided to seek doctor. We don't know which disease caused this state nor do we know the treatment, but we were able to observe that it doesn't match the usual state.

A major advantage of this approach is that it does not require us to say anything about non-normal patterns; hence, it is appropriate for modeling known-unknowns and unknown-unknowns. On the other hand, it does not tell us what exactly is wrong.

Analysis types

Several approaches have been proposed to tackle the problem either way. We broadly classify anomalous and suspicious behavior detection in the following three categories: pattern analysis, transaction analysis, and plan recognition. In the following sections, we will quickly look into some real-life applications.

Pattern analysis

An active area of anomalous and suspicious behavior detection from patterns is based on visual modalities such as camera. Zhang et al (2007) proposed a system for a visual human motion analysis from a video sequence, which recognizes unusual behavior based on walking trajectories; Lin et al (2009) described a video surveillance system based on color features, distance features, and a count feature, where evolutionary techniques are used to measure observation similarity. The system tracks each person and classifies their behavior by analyzing their trajectory patterns. The system extracts a set of visual low-level features in different parts of the image, and performs a classification with SVMs to detect aggressive, cheerful, intoxicated, nervous, neutral, and tired behavior.

Transaction analysis

Transaction analysis assumes discrete states/transactions in contrast to continuous observations. A major research area is **Intrusion Detection (ID)** that aims at detecting attacks against information systems in general. There are two types of ID systems, signature-based and anomaly-based, that broadly follow the suspicious and anomalous pattern detection as described in the previous sections. A comprehensive review of ID approaches was published by Gyanchandani et al (2012).

Furthermore, applications in ambient-assisted living that are based on wearable sensors also fit to transaction analysis as sensing is typically event-based.

Lymberopoulos et al (2008) proposed a system for automatic extraction of the users' **spatio-temporal** patterns encoded as sensor activations from the sensor network deployed inside their home. The proposed method, based on location, time, and duration, was able to extract frequent patterns using the Apriori algorithm and encode the most frequent patterns in the form of a Markov chain. Another area of related work includes **Hidden Markov Models (HMMs)** (Rabiner, 1989) that are widely used in traditional activity recognition for modeling a sequence of actions, but these topics are already out of scope of this book.

Plan recognition

Plan recognition focuses on a mechanism for recognizing the unobservable state of an agent, given observations of its interaction with its environment (Avrahami-Zilberbrand, 2009). Most existing investigations assume discrete observations in the form of activities. To perform anomalous and suspicious behavior detection, plan recognition algorithms may use a hybrid approach, a symbolic plan recognizer is used to filter consistent hypotheses, passing them to an evaluation engine, which focuses on ranking.

These were advanced approaches applied to various real-life scenarios targeted at discovering anomalies. In the following sections, we'll dive into more basic approaches for suspicious and anomalous pattern detection.

Fraud detection of insurance claims

First, we'll take a look at suspicious behavior detection, where the goal is to learn known patterns of frauds, which correspond to modeling known-knowns.

Dataset

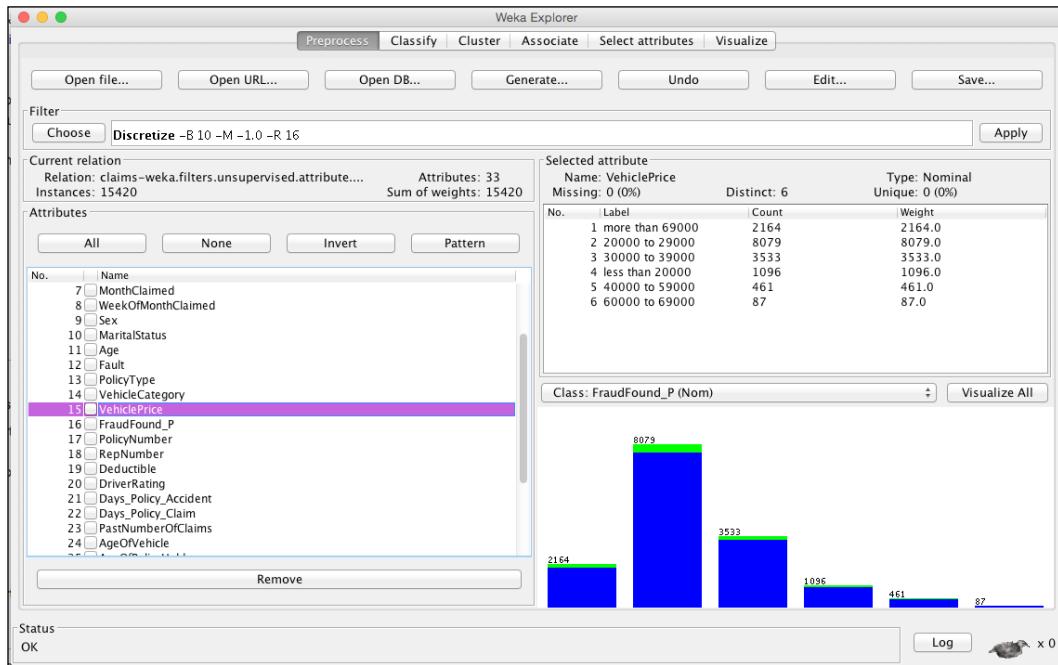
We'll work with a dataset describing insurance transactions publicly available at **Oracle Database Online Documentation** (2015), as follows:

http://docs.oracle.com/cd/B28359_01/datamine.111/b28129/anomalies.htm

The dataset describes insurance vehicle incident claims for an undisclosed insurance company. It contains 15,430 claims; each claim comprises 33 attributes describing the following components:

- Customer demographic details (**Age**, **Sex**, **MartialStatus**, and so on)
- Purchased policy (**PolicyType**, **VehicleCategory**, number of supplements, agent type, and so on)
- Claim circumstances (day/month/week claimed, policy report filed, witness present, past days between incident-policy report, incident-claim, and so on)
- Other customer data (number of cars, previous claims, **DriverRating**, and so on)
- Fraud found (yes and no)

A sample of the database shown in the following screenshot depicts the data loaded into Weka:



Now the task is to create a model that will be able to identify suspicious claims in future. The challenging thing about this task is the fact that only 6% of claims are suspicious. If we create a dummy classifier saying no claim is suspicious, it will be accurate in 94% cases. Therefore, in this task, we will use different accuracy measures: precision and recall.

Recall the outcome table from *Chapter 1, Applied Machine Learning Quick Start*, where there are four possible outcomes denoted as true positive, false positive, false negative, and true negative:

		Classified as	
Actual		Fraud	No fraud
	Fraud	TP – true positive	FN – false negative
	No fraud	FP – false positive	TN – true negative

Precision and recall are defined as follows:

- Precision is equal to the proportion of correctly raised alarms, as follows:

$$\text{Pr} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

- Recall is equal to the proportion of deviant signatures, which are correctly identified as such:

$$\text{Re} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

- With these measures, our dummy classifier scores $\text{Pr} = 0$ and $\text{Re} = 0$ as it never marks any instance as fraud ($\text{TP}=0$). In practice, we want to compare classifiers by both numbers, hence we use *F-measure*. This is a de-facto measure that calculates a harmonic mean between precision and recall, as follows:

$$F - \text{measure} = \frac{2 * \text{Pr} * \text{Re}}{\text{Pr} + \text{Re}}$$

Now let's move on to designing a real classifier.

Modeling suspicious patterns

To design a classifier, we can follow the standard supervised learning steps as described in *Chapter 1, Applied Machine Learning Quick Start*. In this recipe, we will include some additional steps to handle unbalanced dataset and evaluate classifiers based on precision and recall. The plan is as follows:

- Load the data in the .csv format
- Assign the class attribute
- Convert all the attributes from numeric to nominal in order to make sure there are no incorrectly loaded numerical values
- **Experiment 1:** Evaluate models with k-fold cross validation
- **Experiment 2:** Rebalance dataset to a more balanced class distribution and manually perform cross validation
- Compare classifiers by recall, precision, and f-measure

First, let's load the data using the CSVLoader class, as follows:

```
String filePath = "/Users/bostjan/Dropbox/ML Java Book/book/datasets/
chap07/claims.csv";

CSVLoader loader = new CSVLoader();
loader.setFieldSeparator(",");
loader.setSource(new File(filePath));
Instances data = loader.getDataSet();
```

Next, we need to make sure all the attributes are nominal. During the data import, Weka applies some heuristics to guess the most probable attribute type, that is, numeric, nominal, string, or date. As heuristics cannot always guess the correct type, we can set types manually, as follows:

```
NumericToNominal toNominal = new NumericToNominal();
toNominal.setInputFormat(data);
data = Filter.useFilter(data, toNominal);
```

Before we continue, we need to specify the attribute that we will try to predict. We can achieve this by calling the `setClassIndex(int)` function:

```
int CLASS_INDEX = 15;
data.setClassIndex(CLASS_INDEX);
```

Next, we need to remove an attribute describing the policy number as it has no predictive value. We simply apply the `Remove` filter, as follows:

```
Remove remove = new Remove();
remove.setInputFormat(data);
remove.setOptions(new String[] {"-R", "+POLICY_INDEX"});
data = Filter.useFilter(data, remove);
```

Now we are ready to start modeling.

Vanilla approach

The vanilla approach is to directly apply the lesson as demonstrated in *Chapter 3, Basic Algorithms – Classification, Regression, Clustering*, without any pre-processing and not taking into account dataset specifics. To demonstrate drawbacks of vanilla approach, we will simply build a model with default parameters and apply k-fold cross validation.

First, let's define some classifiers that we want to test:

```
ArrayList<Classifier>models = new ArrayList<Classifier>();
models.add(new J48());
models.add(new RandomForest());
models.add(new NaiveBayes());
models.add(new AdaBoostM1());
models.add(new Logistic());
```

Next, we create an `Evaluation` object and perform k-fold cross validation by calling the `crossValidate(Classifier, Instances, int, Random, String[])` method, outputting precision, recall, and fMeasure:

```
int FOLDS = 3;
Evaluation eval = new Evaluation(data);

for(Classifier model : models){
    eval.crossValidateModel(model, data, FOLDS,
    new Random(1), new String[] {});
    System.out.println(model.getClass().getName() + "\n"+
    "\tRecall: " + eval.recall(FRAUD) + "\n"+
    "\tPrecision: " + eval.precision(FRAUD) + "\n"+
    "\tF-measure: " + eval.fMeasure(FRAUD));
}
```

The evaluation outputs the following scores:

```
weka.classifiers.trees.J48
Recall:      0.03358613217768147
Precision:   0.9117647058823529
F-measure:   0.06478578892371996
...
weka.classifiers.functions.Logistic
Recall:      0.037486457204767065
Precision:   0.2521865889212828
F-measure:   0.06527070364082249
```

We can see the results are not very promising. Recall, that is, the share of discovered frauds among all frauds is only 1-3%, meaning that only 1-3/100 frauds are detected. On the other hand, precision, that is, the accuracy of alarms is 91%, meaning that in 9/10 cases, when a claim is marked as fraud, the model is correct.

Dataset rebalancing

As the number of negative examples, that is, frauds, is very small, compared to positive examples, the learning algorithms struggle with induction. We can help them by giving them a dataset, where the share of positive and negative examples is comparable. This can be achieved with dataset rebalancing.

Weka has a built-in filter, **Resample**, which produces a random subsample of a dataset using either sampling with replacement or without replacement. The filter can also bias distribution towards a uniform class distribution.

We will proceed by manually implementing k-fold cross validation. First, we will split the dataset into k equal folds. Fold k will be used for testing, while the other folds will be used for learning. To split dataset into folds, we'll use the **StratifiedRemoveFolds** filter, which maintains the class distribution within the folds, as follows:

```
StratifiedRemoveFolds kFold = new StratifiedRemoveFolds();
kFold.setInputFormat(data);

double measures [][] = new double [models.size ()] [3];

for(int k = 1; k <= FOLDS; k++){
    // Split data to test and train folds
    kFold.setOptions (new String [] {
```

```
"-N", "+FOLDS, "-F", "+k, -S", "1"});  
Instances test = Filter.useFilter(data, kFold);  
  
kFold.setOptions(new String[] {  
    "-N", "+FOLDS, -F", "+k, -S", "1", "-V"});  
    // select inverse "-V"  
Instances train = Filter.useFilter(data, kFold);
```

Next, we can rebalance train dataset, where the `-Z` parameter specifies the percentage of dataset to be resampled, and `-B` bias the class distribution towards uniform distribution:

```
Resample resample = new Resample();  
resample.setInputFormat(data);  
resample.setOptions(new String[] {"-Z", "100", "-B", "1"}); //with  
    replacement  
Instances balancedTrain = Filter.useFilter(train, resample);
```

Next, we can build classifiers and perform evaluation:

```
for(ListIterator<Classifier>it = models.listIterator();  
    it.hasNext();){  
    Classifier model = it.next();  
    model.buildClassifier(balancedTrain);  
    eval = new Evaluation(balancedTrain);  
    eval.evaluateModel(model, test);  
  
    // save results for average  
    measures[it.previousIndex()][0] += eval.recall(FRAUD);  
    measures[it.previousIndex()][1] += eval.precision(FRAUD);  
    measures[it.previousIndex()][2] += eval.fMeasure(FRAUD);  
}
```

Finally, we calculate the average and output the best model:

```
// calculate average  
for(int i = 0; i < models.size(); i++){  
    measures[i][0] /= 1.0 * FOLDS;  
    measures[i][1] /= 1.0 * FOLDS;  
    measures[i][2] /= 1.0 * FOLDS;  
}  
  
// output results and select best model  
Classifier bestModel = null; double bestScore = -1;  
for(ListIterator<Classifier> it = models.listIterator();  
    it.hasNext();){
```

```
Classifier model = it.next();
double fMeasure = measures[it.previousIndex()][2];
System.out.println(
    model.getClass().getName() + "\n"+
    "\tRecall: " + measures[it.previousIndex()][0] + "\n"+
    "\tPrecision: " + measures[it.previousIndex()][1] + "\n"+
    "\tF-measure: " + fMeasure);
if(fMeasure > bestScore) {
    bestScore = fMeasure;
    bestModel = model;

}
System.out.println("Best model:" + bestModel.getName());
```

Now the performance of the models has significantly improved, as follows:

```
weka.classifiers.trees.J48
Recall: 0.44204845100610574
Precision: 0.14570766048577555
F-measure: 0.21912423640160392
...
weka.classifiers.functions.Logistic
Recall: 0.7670657247204478
Precision: 0.13507459756495374
F-measure: 0.22969038530557626
Best model: weka.classifiers.functions.Logistic
```

What we can see is that all the models have scored significantly better; for instance, the best model, Logistic Regression, correctly discovers 76% of frauds, while producing a reasonable amount of false alarms – only 13% of claims marked as fraud are indeed fraudulent. If an undetected fraud is significantly more expensive than investigation of false alarms, then it makes sense to deal with an increased number of false alarms.

The overall performance has most likely still some room for improvement; we could perform attribute selection and feature generation and apply more complex model learning that we discussed in *Chapter 3, Basic Algorithms – Classification, Regression, Clustering*.

Anomaly detection in website traffic

In the second example, we'll focus on modeling the opposite of the previous example. Instead of discussing what typical fraud-less cases are, we'll discuss the normal expected behavior of the system. If something cannot be matched against our expected model, it will be considered anomalous.

Dataset

We'll work with a publicly available dataset released by Yahoo Labs that is useful for discussing how to detect anomalies in time series data. For Yahoo, the main use case is in detecting unusual traffic on Yahoo servers.

Even though Yahoo announced that their data is publicly available, you have to apply to use it, and it takes about 24 hours before the approval is granted. The dataset is available here:

<http://webscope.sandbox.yahoo.com/catalog.php?datatype=s&did=70>

The data set comprises real traffic to Yahoo services, along with some synthetic data. In total, the dataset contains 367 time series, each of which contain between 741 and 1680 observations, recorded at regular intervals. Each series is written in its own file, one observation per line. A series is accompanied by a second column indicator with a one if the observation was an anomaly, and zero otherwise. The anomalies in real data were determined by human judgment, while those in the synthetic data were generated algorithmically. A snippet of the synthetic times series data is shown in the following table:

timestamp	value	anomaly	change point	trend	noise	12 hour seasonality	daily seasonality	weekly seasonality
1422237600	4333.43	0	0	4599	1.81	-190.95	-128.86	52.44
1422241200	4316.14	0	0	4602	-14.65	-220.5	-105.21	54.51
1422244800	4403.20	0	0	4605	7.04	-190.95	-74.39	56.51
1422248400	4531.20	0	0	4608	13.52	-110.25	-38.51	58.43
1422252000	4967.50	1	0	4911	-3.77	-6.91	-2.33	60.27

Snippet of the synthetic time-series data

In the following section, we'll learn how to transform time series data to attribute presentation that allows us to apply machine learning algorithms.

Anomaly detection in time series data

Detecting anomalies in raw, streaming time series data requires some data transformations. The most obvious way is to select a time window and sample time series with fixed length. In the next step, we want to compare a new time series to our previously collected set to detect if something is out of the ordinary.

The comparison can be done with various techniques, as follows:

- Forecasting the most probable following value, as well as confidence intervals (for example, Holt-Winters exponential smoothing). If a new value is out of forecasted confidence interval, it is considered anomalous.
- Cross correlation compares new sample to library of positive samples, it looks for exact match. If the match is not found, it is marked as anomalous.
- Dynamic time wrapping is similar to cross correlation, but allows signal distortion in comparison.
- Discretizing signal to bands, where each band corresponds to a letter. For example, $A=[\min, \text{mean}/3]$, $B=[\text{mean}/3, \text{mean}^*2/3]$, and $C=[\text{mean}^*2/3, \max]$ transforms the signal to a sequence of letters such as *aABAACAAABBA....* This approach reduces the storage and allows us to apply text-mining algorithms that we will discuss in *Chapter 10, Text Mining with Mallet – Topic Modeling and Spam Detection*.
- Distribution-based approach estimates distribution of values in a specific time window. When we observe a new sample, we can compare whether distribution matches to the previously observed one.

This list is, by no means, exhaustive. Different approaches are focused on detecting different anomalies (for example, in value, frequency, and distribution). We will focus on a version of distribution-based approaches.

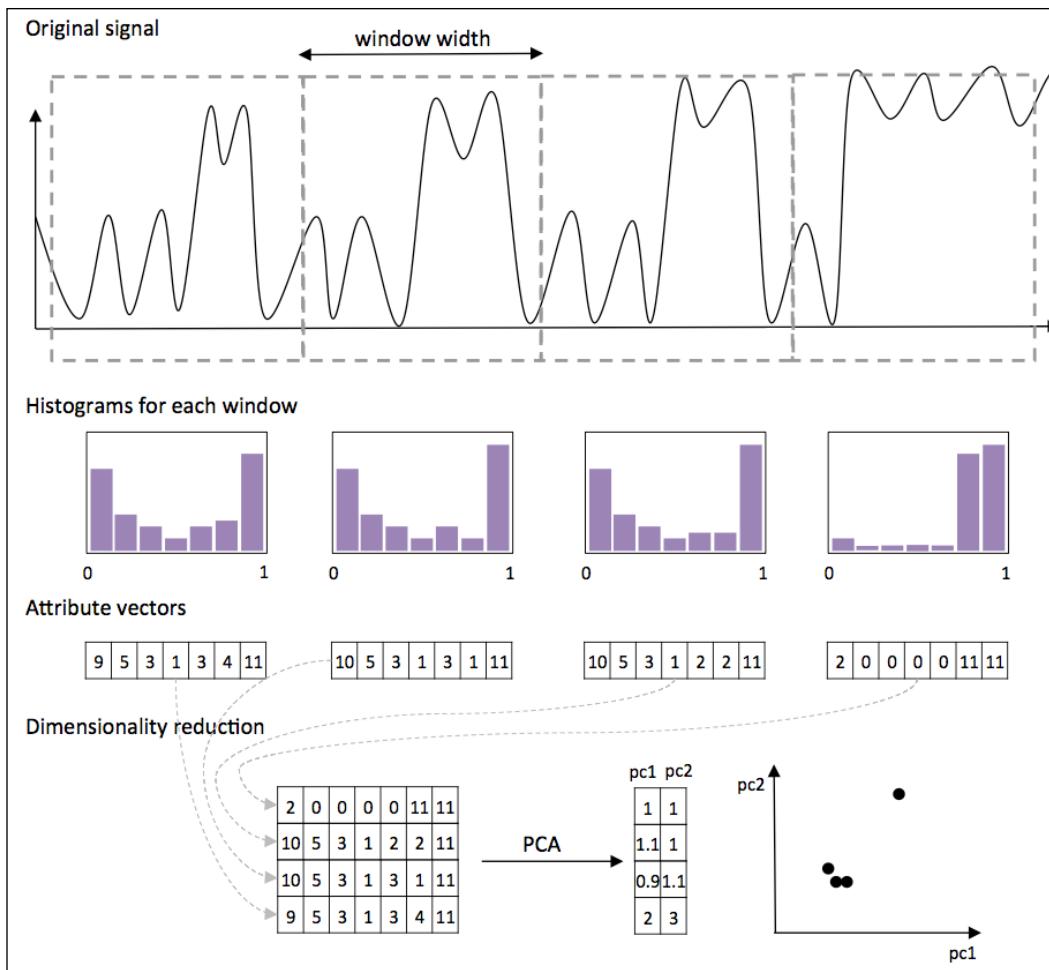
Histogram-based anomaly detection

In histogram-based anomaly detection, we split signals by some selected time window as shown in the following image.

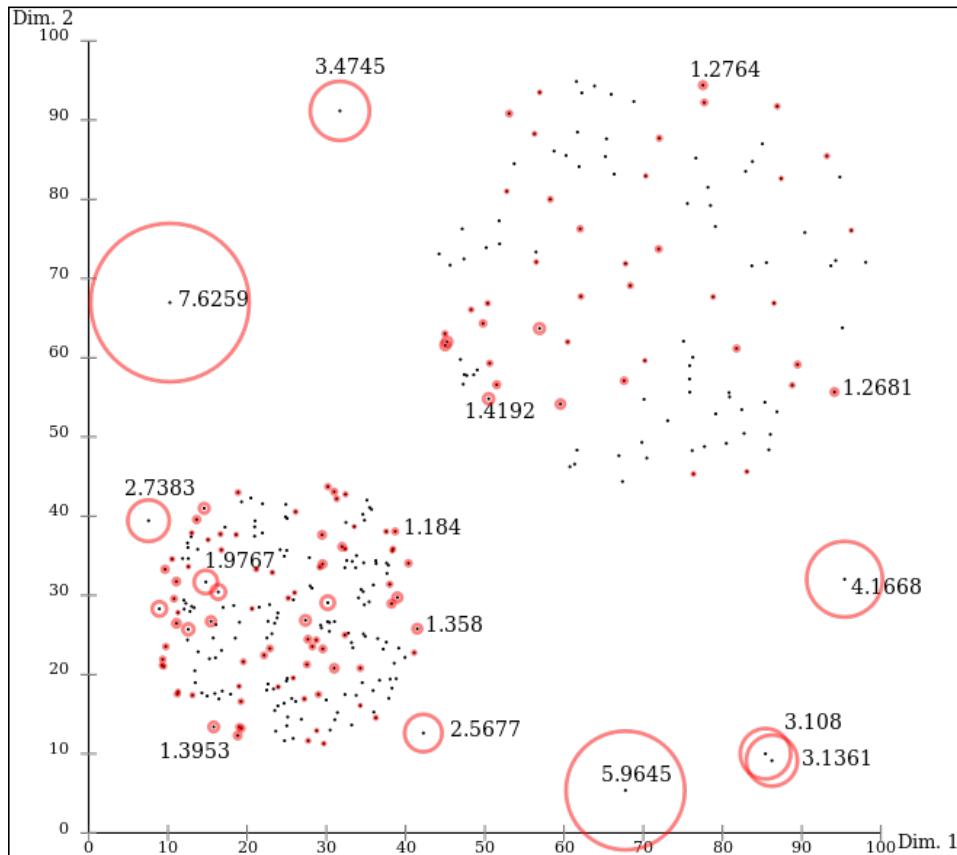
For each window, we calculate the histogram, that is, for a selected number of buckets, we count how many values fall into each bucket. The histogram captures basic distribution of values in a selected time window as shown at the center of the diagram.

Histograms can be then directly presented as instances, where each bin corresponds to an attribute. Further, we can reduce the number of attributes by applying a dimensionality-reduction technique such as **Principal Component Analysis (PCA)**, which allows us to visualize the reduced-dimension histograms in a plot as shown at the bottom-right of the diagram, where each dot corresponds to a histogram.

In our example, the idea is to observe website traffic for a couple of days and then to create histograms, for example, four-hour time windows to build a library of positive behavior. If a new time window histogram cannot be matched against positive library, we can mark it as an anomaly:



For comparing a new histogram to a set of existing histograms, we will use a density-based k-nearest neighbor algorithm, **Local Outlier Factor (LOF)** (Breunig et al, 2000). The algorithm is able to handle clusters with different densities as shown in the following image. For example, the upper-right cluster is large and widespread as compared to the bottom-left cluster, which is smaller and denser:



Let's get started!

Loading the data

In the first step, we'll need to load the data from text files to a Java object. The files are stored in a folder, each file contains one-time series with values per line. We'll load them into a list of Double:

```
String filePath = "chap07/ydata/A1Benchmark/real";
List<List<Double>> rawData = new ArrayList<List<Double>>();
```

We will need the `min` and `max` value for histogram normalization, so let's collect them in this data pass:

```
double max = Double.MIN_VALUE;
double min = Double.MAX_VALUE;

for(int i = 1; i<= 67; i++){
    List<Double> sample = new ArrayList<Double>();
    BufferedReader reader = new BufferedReader(new
        FileReader(filePath+i+".csv"));

    boolean isAnomaly = false;
    reader.readLine();
    while(reader.ready()){
        String line[] = reader.readLine().split(",");
        double value = Double.parseDouble(line[1]);
        sample.add(value);

        max = Math.max(max, value);
        min = Double.min(min, value);

        if(line[2] == "1")
            isAnomaly = true;
    }
    System.out.println(isAnomaly);
    reader.close();

    rawData.add(sample);
}
```

The data is loaded, now let's move on to histograms.

Creating histograms

We will create a histogram for a selected time window with the `WIN_SIZE` width. The histogram will hold the `HIST_BINS` value buckets. The histograms consisting of list of doubles will be stored into an array list:

```
int WIN_SIZE = 500;
int HIST_BINS = 20;
int current = 0;

List<double[]> dataHist = new ArrayList<double[]>();
for(List<Double> sample : rawData){
```

```
double[] histogram = new double[HIST_BINS];
for(double value : sample){
    int bin = toBin(normalize(value, min, max), HIST_BINS);
    histogram[bin]++;
    current++;
    if(current == WIN_SIZE) {
        current = 0;
        dataHist.add(histogram);
        histogram = new double[HIST_BINS];
    }
}
dataHist.add(histogram);
}
```

Histograms are now completed. The last step is to transform them into Weka's `Instance` objects. Each histogram value will correspond to one Weka attribute, as follows:

```
ArrayList<Attribute> attributes = new ArrayList<Attribute>();
for(int i = 0; i < HIST_BINS; i++) {
    attributes.add(new Attribute("Hist-"+i));
}
Instances dataset = new Instances("My dataset", attributes,
    dataHist.size());
for(double[] histogram: dataHist){
    dataset.add(new Instance(1.0, histogram));
}
```

The dataset is now loaded and ready to be plugged into an anomaly-detection algorithm.

Density based k-nearest neighbors

To demonstrate how LOF calculates scores, we'll first split the dataset into training and testing set using the `testCV(int, int)` function. The first parameter specifies the number of folds, while the second parameter specifies which fold to return.

```
// split data to train and test
Instances trainData = dataset.testCV(2, 0);
Instances testData = dataset.testCV(2, 1);
```

The LOF algorithm is not a part of the default Weka distribution, but it can be downloaded through Weka's package manager:

<http://weka.sourceforge.net/packageMetaData/localOutlierFactor/index.html>

LOF algorithm has two implemented interfaces: as an unsupervised filter that calculates LOF values (known-unknowns) and as a supervised k-nn classifier (known-knowns). In our case, we want to calculate the outlier-ness factor, therefore, we'll use the unsupervised filter interface:

```
import weka.filters.unsupervised.attribute.LOF;
```

The filter is initialized the same way as a usual filter. We can specify the `k` number of neighbors, for example, `k=3`, with `-min` and `-max` parameters. LOF allows us to specify two different `k` parameters, which are used internally as the upper and lower bound to find the minimal/maximal number `lof` values:

```
LOF lof = new LOF();
lof.setInputFormat(trainData);
lof.setOptions(new String[] {"-min", "3", "-max", "3"});
```

Next, we load training instances into the filter that will serve as a positive example library. After we complete the loading, we call the `batchFinished()` method to initialize internal calculations:

```
for(Instance inst : trainData) {
    lof.input(inst);
}
lof.batchFinished();
```

Finally, we can apply the filter to test data. Filter will process the instances and append an additional attribute at the end containing the LOF score. We can simply output the score on the console:

```
Instances testDataLofScore = Filter.useFilter(testData, lof);

for(Instance inst : testDataLofScore) {
    System.out.println(inst.value(inst.numAttributes() - 1));
}
```

The LOF score of the first couple of test instances is as follows:

```
1.306740014927325
1.318239332210458
1.0294812291949587
1.1715039094530768
```

To understand the LOF values, we need some background on the LOF algorithm. It compares the density of an instance to the density of its nearest neighbors. The two scores are divided, producing the LOF score. The LOF score around 1 indicates that the density is approximately equal, while higher LOF values indicate that the density of the instance is substantially lower than the density of its neighbors. In such cases, the instance can be marked as anomalous.

Summary

In this chapter, we looked into detecting anomalous and suspicious patterns. We discussed the two fundamental approaches focusing on library encoding either positive or negative patterns. Next, we got our hands on two real-life datasets, where we discussed how to deal with unbalanced class distribution and perform anomaly detection in time series data.

In the next chapter, we'll dive deeper into patterns and more advanced approaches to build pattern-based classifier, discussing how to automatically assign labels to images with deep learning.

8

Image Recognition with Deeplearning4j

Images have become ubiquitous in web services, social networks, and web stores. In contrast to humans, computers have great difficulty in understanding what is in the image and what does it represent. In this chapter, we'll first look at the challenges required to teach computers how to understand images, and then focus on an approach based on deep learning. We'll look at a high-level theory required to configure a deep learning model and discuss how to implement a model that is able to classify images using a Java library, Deeplearning4j.

This chapter will cover the following topics:

- Introducing image recognition
- Discussing deep learning fundamentals
- Building an image recognition model

Introducing image recognition

A typical goal of image recognition is to detect and identify an object in a digital image. Image recognition is applied in factory automation to monitor product quality; surveillance systems to identify potentially risky activities, such as moving persons or vehicles; security applications to provide biometric identification through fingerprints, iris, or facial features; autonomous vehicles to reconstruct conditions on the road and environment and so on.

Digital images are not presented in a structured way with attribute-based descriptions; instead, they are encoded as the amount of color in different channels, for instance, black-white and red-green-blue channels. The learning goal is to then identify patterns associated with a particular object. The traditional approach for image recognition consists of transforming an image into different forms, for instance, identify object corners, edges, same-color blobs, and basic shapes. Such patterns are then used to train a learner to distinguish between objects. Some notable examples of traditional algorithms are:

- Edge detection finds boundaries of objects within an image
- Corner detection identifies intersections of two edges or other interesting points, such as line endings, curvature maxima/minima, and so on
- Blob detection identifies regions that differ in a property, such as brightness or color, compared to its surrounding regions
- Ridge detection identifies additional interesting points at the image using smooth functions
- **Scale Invariant Feature Transform (SIFT)** is a robust algorithm that can match objects even if their scale or orientation differs from the representative samples in database
- Hough transform identifies particular patterns in the image

A more recent approach is based on deep learning. Deep learning is a form of neural network, which mimics how the brain processes information. The main advantage of deep learning is that it is possible to design neural networks that can automatically extract relevant patterns, which in turn, can be used to train a learner. With recent advances in neural networks, image recognition accuracy was significantly boosted. For instance, the **ImageNet** challenge (ImageNet, 2016), where competitors are provided more than 1.2 million images from 1,000 different object categories, reports that the error rate of the best algorithm was reduced from 28% in 2010, using SVM, to only 7% in 2014, using deep neural network.

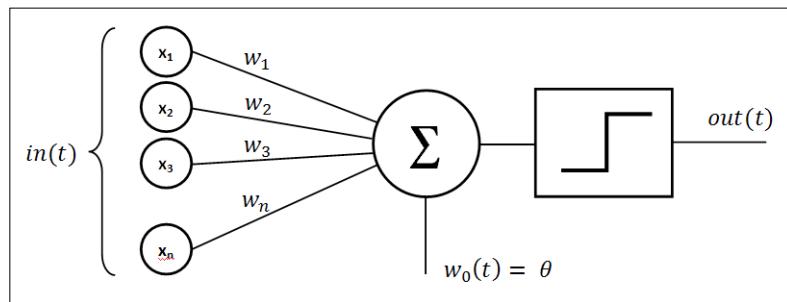
In this chapter, we'll take a quick look at neural networks, starting from the basic building block – **perceptron** – and gradually introducing more complex structures.

Neural networks

The first neural networks, introduced in the sixties, are inspired by biological neural networks. Recent advances in neural networks proved that deep neural networks fit very well in pattern recognition tasks, as they are able to automatically extract interesting features and learn the underlying presentation. In this section, we'll refresh the fundamental structures and components from a single perceptron to deep networks.

Perceptron

Perceptron is a basic neural network building block and one of the earliest supervised algorithms. It is defined as a sum of features, multiplied by corresponding weights and a bias. The function that sums all of this together is called **sum transfer function** and it is fed into an **activation function**. If the binary step activation function reaches a threshold, the output is 1, otherwise 0, which gives us a binary classifier. A schematic illustration is shown in the following diagram:

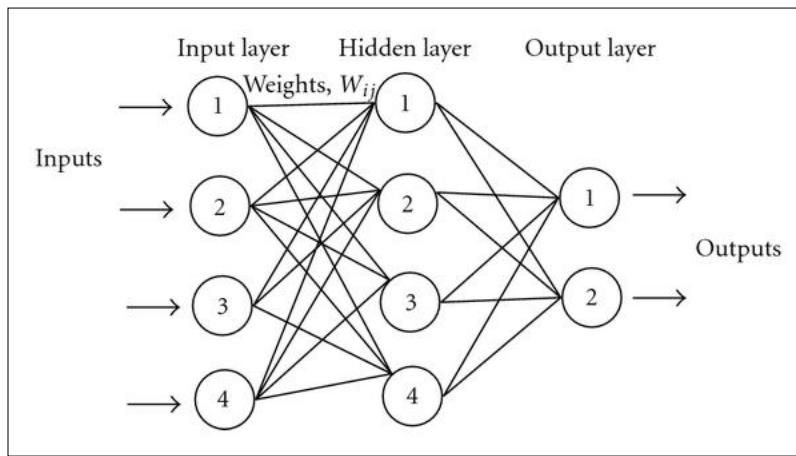


Training perceptrons involves a fairly simple learning algorithm that calculates the errors between the calculated output values and correct training output values, and uses this to create an adjustment to the weights; thus implementing a form of gradient descent. This algorithm is usually called the **delta rule**.

Single-layer perceptron is not very advanced, and nonlinearly separable functions, such as XOR, cannot be modeled using it. To address this issue, a structure with multiple perceptrons was introduced, called **multilayer perceptron**, also known as **feedforward neural network**.

Feedforward neural networks

A feedforward neural network is an artificial neural network that consists of several perceptrons, which are organized by layers, as shown in the following diagram: input layer, output layer, and one or more hidden layers. Each layer perceptron, also known as neuron, has direct connections to the perceptrons in the next layer; whereas, connections between two neurons carry a weight similar to the perceptron weights. The diagram shows a network with a four-unit **Input layer**, corresponding to the size of feature vector of length 4, a four-unit **Hidden layer**, and a two-unit **Output layer**, where each unit corresponds to one class value:



The most popular approach to train multilayer networks is backpropagation. In backpropagation, the calculated output values are compared with the correct values in the same way as in delta rule. The error is then fed back through the network by various techniques, adjusting the weights of each connection in order to reduce the value of the error. The process is repeated for sufficiently large number of training cycles, until the error is under a certain threshold.

Feedforward neural network can have more than one hidden layer; whereas, each additional hidden layer builds a new abstraction atop the previous layers. This often leads to more accurate models; however, increasing the number of hidden layers leads to the following two known issues:

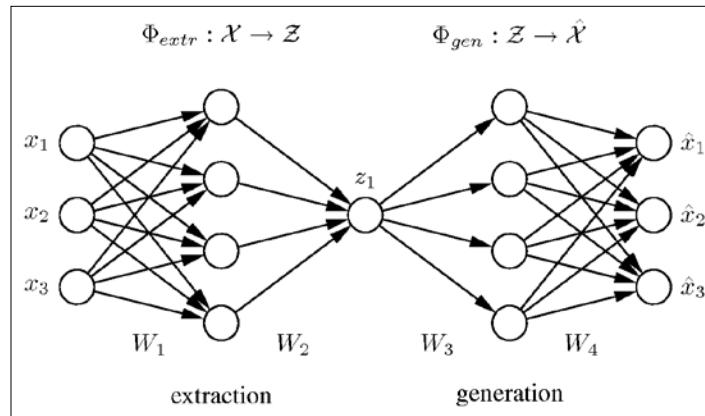
- **Vanishing gradients problem:** With more hidden layers, the training with backpropagation becomes less and less useful for passing information to the front layers, causing these layers to train very slowly
- **Overfitting:** The model fits the training data too well and performs poorly on real examples

Let's look at some other networks structures that address these issues.

Autoencoder

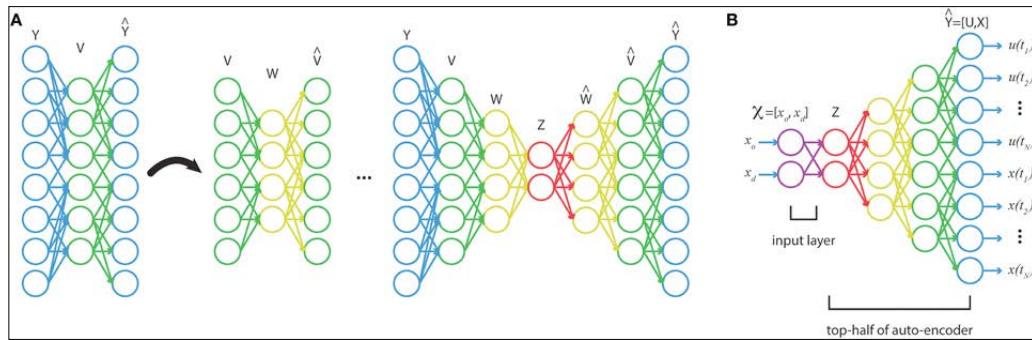
Autoencoder is a feedforward neural network that aims to learn how to compress the original dataset. Therefore, instead of mapping features to input layer and labels to output layer, we will map the features to both input and output layers. The number of units in hidden layers is usually different from the number of units in input layers, which forces the network to either expand or reduce the number of original features. This way the network will learn the important features, while effectively applying dimensionality reduction.

An example network is shown below. The three-unit input layer is first expanded into a four-unit layer and then compressed into a single-unit layer. The other side of the network restores the single-layer unit back to the four-unit layer, and then to the original three-input layer:



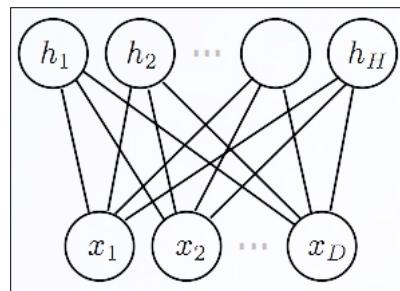
Once the network is trained, we can take the left-hand side to extract image features as we would with traditional image processing.

The autoencoders can be also combined into **stacked autoencoders**, as shown in the following image. First, we will discuss the hidden layer in a basic autoencoder, as described previously. Then we will take the learned hidden layer (green circles) and repeat the procedure, which in effect, learns a more abstract presentation. We can repeat the procedure multiple times, transforming the original features into increasingly reduced dimensions. At the end, we will take all the hidden layers and stack them into a regular feedforward network, as shown at the top-right part of the diagram:



Restricted Boltzmann machine

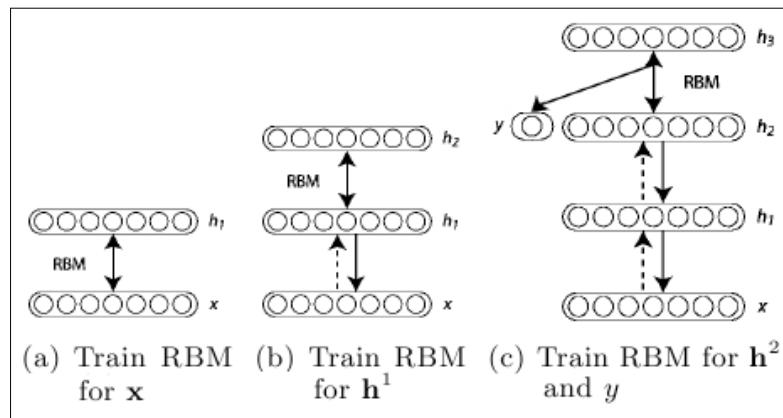
Restricted Boltzman machine is an undirected neural network, also denoted as **Generative Stochastic Networks (GSNs)**, and can learn probability distribution over its set of inputs. As the name suggests, they originate from Boltzman machine, a recurrent neural network introduced in the eighties. Restricted means that the neurons must form two fully connected layers – input layer and hidden layer – as show in the following diagram:



Unlike feedforward networks, the connections between the visible and hidden layers are undirected, hence the values can be propagated in both visible-to-hidden and hidden-to-visible directions.

Training Restricted Boltzmann machines is based on the **Contrastive Divergence** algorithm, which uses a gradient descent procedure, similar to backpropagation, to update weights, and **Gibbs sampling** is applied on the **Markov chain** to estimate the gradient—the direction on how to change the weights.

Restricted Boltzmann machines can also be stacked to create a class known as **Deep Belief Networks (DBNs)**. In this case, the hidden layer of RBM acts as a visible layer for the RBM layer, as shown in the following diagram:



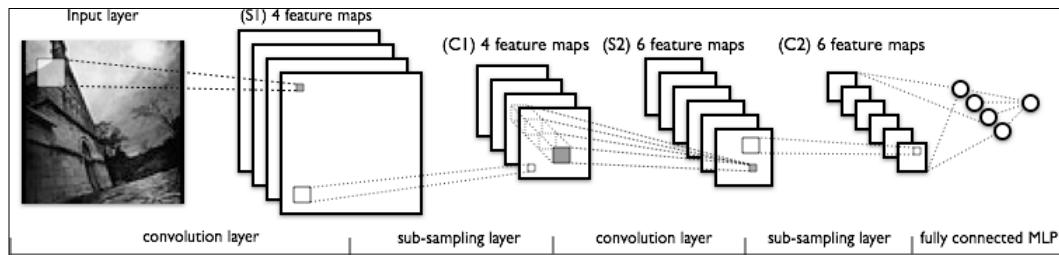
The training, in this case, is incremental; training layer by layer.

Deep convolutional networks

A network structure that recently achieves very good results at image recognition benchmarks is **Convolutional Neural Network (CNN)** or ConvNet. CNNs are a type of feedforward neural network that are structured in such a way that emulates behavior of the visual cortex, exploiting 2D structures of an input image, that is, patterns that exhibit spatially local correlation.

A CNN consists of a number of convolutional and subsampling layers, optionally followed by fully connected layers. An example is shown in the following image. The input layer reads all the pixels at an image and then we apply multiple filters. In the following image, four different filters are applied. Each filter is applied to the original image, for example, one pixel of a 6×6 filter is calculated as the weighted sum of a 6×6 square of input pixels and corresponding 6×6 weights. This effectively introduces filters similar to the standard image processing, such as smoothing, correlation, edge detection, and so on. The resulting image is called **feature map**. In the example in the image, we have four feature maps, one for each filter.

The next layer is the subsampling layer, which reduces the size of the input. Each feature map is subsampled typically with mean or max pooling over a contiguous region of 2×2 (up to 5×5 for large images). For example, if the feature map size is 16×16 and the subsampling region is 2×2 , the reduced feature map size is 8×8 , where 4 pixels (2×2 square) are combined into a single pixel by calculating max, min, mean, or some other functions:



The network may contain several consecutive convolution and subsampling layers, as shown in the preceding diagram. A particular feature map is connected to the next reduced/convoluted feature map, while feature maps at the same layer are not connected to each other.

After the last subsampling or convolutional layer, there is usually a fully connected layer, identical to the layers in a standard multilayer neural network, which represents the target data.

CNN is trained using a modified backpropagation algorithm that takes the subsampling layers into account and updates the convolutional filter weights based on all the values where this filter is applied.

Some good CNN designs can be found at the ImageNet competition results page:
<http://www.image-net.org/>
An example is AlexNet, described in the *ImageNet Classification with Deep Covolutional Neural Networks* paper by A. Krizhevsky et al.

This concludes our review of main neural network structures. In the following section, we'll move to the actual implementation.

Image classification

In this section, we will discuss how to implement some of the neural network structures with the deeplearning4j library. Let's start.

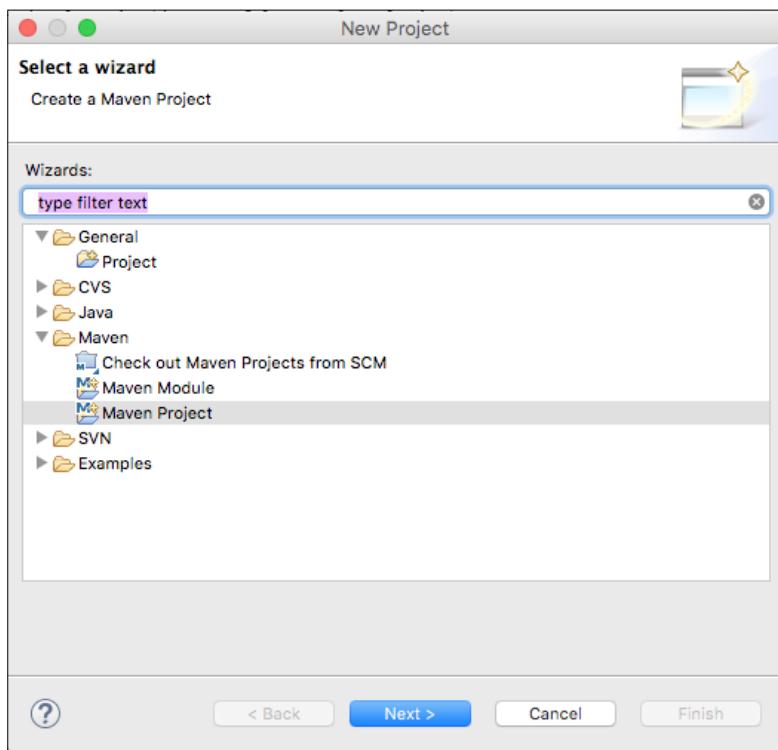
Deeplearning4j

As we discussed in *Chapter 2, Java Libraries and Platforms for Machine Learning*, deeplearning4j is an open source, distributed deep learning project in Java and Scala. Deeplearning4j relies on Spark and Hadoop for MapReduce, trains models in parallel, and iteratively averages the parameters they produce in a central model. A detailed library summary is presented in *Chapter 2, Java Libraries and Platforms for Machine Learning*.

Getting DL4J

The most convenient way to get deeplearning4j is through the Maven repository:

1. Start a new Eclipse project and pick **Maven Project**, as shown in the following screenshot:



2. Open the `pom.xml` file and add the following dependencies under the `<dependencies>` section:

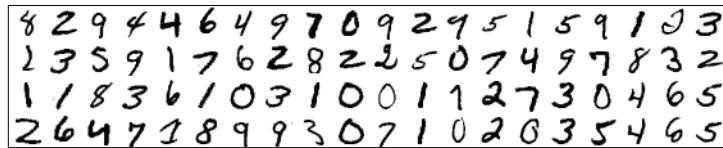
```
<dependency>
    <groupId>org.deeplearning4j</groupId>
    <artifactId>deeplearning4j-nlp</artifactId>
    <version>${dl4j.version}</version>
</dependency>

<dependency>
    <groupId>org.deeplearning4j</groupId>
    <artifactId>deeplearning4j-core</artifactId>
    <version>${dl4j.version}</version>
</dependency>
```

3. Finally, right-click on **Project**, select **Maven**, and pick **Update project**.

MNIST dataset

One of the most famous datasets is MNIST dataset, which consists of handwritten digits, as shown in the following image. The dataset comprises 60,000 training and 10,000 testing images:



The dataset is commonly used in image recognition problems to benchmark algorithms. The worst recorded error rate is 12%, with no preprocessing and using a SVM in one-layer neural network. Currently, as of 2016, the lowest error rate is only 0.21%, using the **DropConnect** neural network, followed by **deep convolutional network** at 0.23%, and deep feedforward network at 0.35%.

Now, let's see how to load the dataset.

Loading the data

DeepLearning4j provides the MNIST dataset loader out of the box. The loader is initialized as `DataSetIterator`. Let's first import the `DataSetIterator` class and all the supported datasets that are part of the `impl` package, for example, Iris, MNIST, and others:

```
import org.deeplearning4j.datasets.iterator.DataSetIterator;
import org.deeplearning4j.datasets.iterator.impl.*;
```

Next, we'll define some constants, for instance, the images consist of 28×28 pixels and there are 10 target classes and 60,000 samples. Initialize a new `MnistDataSetIterator` class that will download the dataset and its labels. The parameters are iteration batch size, total number of examples, and whether the datasets should be binarized or not:

```
int numRows = 28;
int numColumns = 28;
int outputNum = 10;
int numSamples = 60000;
int batchSize = 100;
DataSetIterator iter = new MnistDataSetIterator(batchSize,
numSamples,true);
```

Having an already-implemented data importer is really convenient, but it won't work on your data. Let's take a quick look at how it is implemented and what needs to be modified to support your dataset. If you're eager to start implementing neural networks, you can safely skip the rest of this section and return to it when you need to import your own data.

To load the custom data, you'll need to implement two classes: `DataSetIterator` that holds all the information about the dataset and `BaseDataFetcher` that actually pulls the data either from file, database, or web. Sample implementations are available on GitHub at <https://github.com/deeplearning4j/deeplearning4j/tree/master/deeplearning4j-core/src/main/java/org/deeplearning4j/datasets/iterator/impl>.

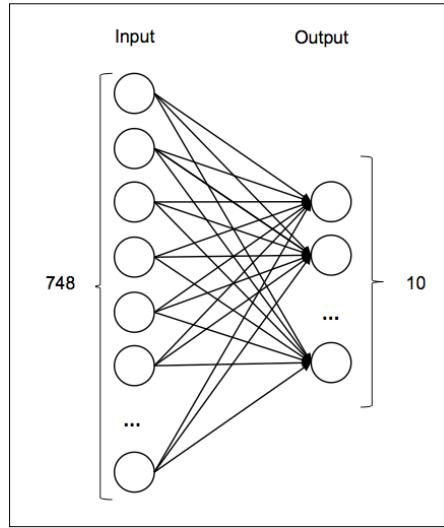
Another option is to use the **Canova** library, which is developed by the same authors, at <http://deeplearning4j.org/canovadoc/>.

Building models

In this section, we'll discuss how to build an actual neural network model. We'll start with a basic single-layer neural network to establish a benchmark and discuss the basic operations. Later, we'll improve this initial result with DBN and **Multilayer Convolutional Network**.

Building a single-layer regression model

Let's start by building a single-layer regression model based on the softmax activation function, as shown in the following diagram. As we have a single layer, **Input** to the neural network will be all the figure pixels, that is, $28 \times 28 = 748$ neurons. The number of **Output** neurons is **10**, one for each digit. The network layers are fully connected, as shown in the following diagram:



A neural network is defined through a `NeuralNetConfiguration` Builder object as follows:

```
MultiLayerConfiguration conf = new  
    NeuralNetConfiguration.Builder()
```

We will define the parameters for gradient search in order to perform iterations with the **conjugate gradient optimization** algorithm. The momentum parameter determines how fast the optimization algorithm converges to a local optimum – the higher the momentum, the faster the training; but higher speed can lower model's accuracy, as follows:

```
.seed(seed)  
.gradientNormalization(GradientNormalization.ClipElementWiseAbsolu  
teValue)  
.gradientNormalizationThreshold(1.0)  
.iterations(iterations)  
.momentum(0.5)  
.momentumAfter(Collections.singletonMap(3, 0.9))  
.optimizationAlgo(OptimizationAlgorithm.CONJUGATE_GRADIENT)
```

Next, we will specify that the network will have one layer and define the error function (`NEGATIVELOGLIKELIHOOD`), internal perceptron activation function (`softmax`), and the number of input and output layers that correspond to total image pixels and the number of target variables:

```
.list(1)
.layer(0, new
OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
.activation("softmax")
.nIn(numRows*numColumns).nOut(outputNum).build())
```

Finally, we will set the network to pretrain, disable backpropagation, and actually build the untrained network structure:

```
.pretrain(true).backprop(false)
.build();
```

Once the network structure is defined, we can use it to initialize a new `MultiLayerNetwork`, as follows:

```
MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
```

Next, we will point the model to the training data by calling the `setListeners` method, as follows:

```
model.setListeners(Collections.singletonList((IterationListener)
new ScoreIterationListener(listenerFreq)));
```

We will also call the `fit(int)` method to trigger an end-to-end network training:

```
model.fit(iter);
```

To evaluate the model, we will initialize a new `Evaluation` object that will store batch results:

```
Evaluation eval = new Evaluation(outputNum);
```

We can then iterate over the dataset in batches in order to keep the memory consumption at a reasonable rate and store the results in an `eval` object:

```
DataSetIterator testIter = new MnistDataSetIterator(100,10000);
while(testIter.hasNext()) {
    DataSet testMnist = testIter.next();
    INDArray predict2 =
    model.output(testMnist.getFeatureMatrix());
    eval.eval(testMnist.getLabels(), predict2);
}
```

Finally, we can get the results by calling the `stats()` function:

```
log.info(eval.stats());
```

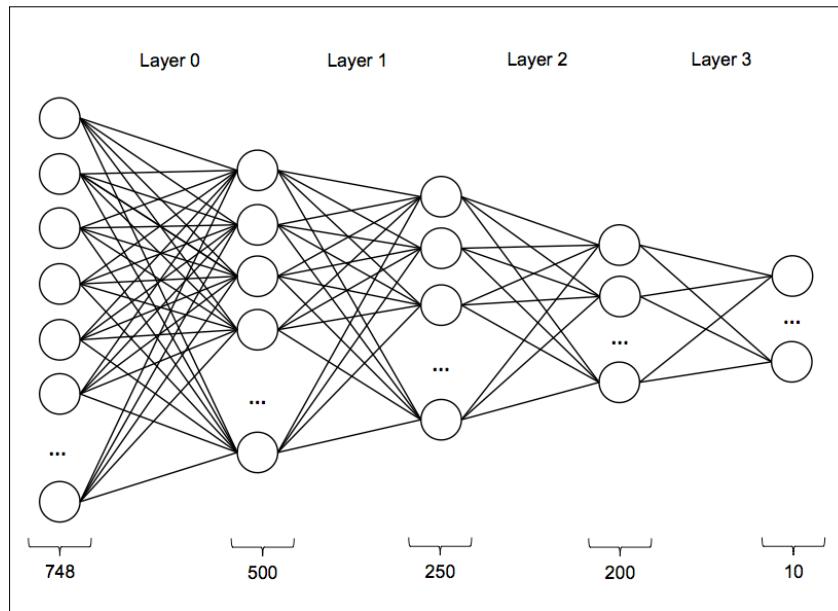
A basic one-layer model achieves the following accuracy:

```
Accuracy: 0.8945
Precision: 0.8985
Recall: 0.8922
F1 Score: 0.8953
```

Getting 89.22% accuracy, that is, 10.88% error rate, on MNIST dataset is quite bad. We'll improve that by going from a simple one-layer network to the moderately sophisticated deep belief network using Restricted Boltzmann machines and Multilayer Convolutional Network.

Building a deep belief network

In this section, we'll build a deep belief network based on Restricted Boltzmann machine, as shown in the following diagram. The network consists of four layers: the first layer receives the **748** inputs to **500** neurons, then to **250**, followed by **200**, and finally to the last **10** target values:



As the code is the same as in the previous example, let's take a look at how to configure such a network:

```
MultiLayerConfiguration conf = new  
    NeuralNetConfiguration.Builder()
```

We defined the gradient optimization algorithm, as shown in the following code:

```
.seed(seed)  
.gradientNormalization(  
    GradientNormalization.ClipElementWiseAbsoluteValue)  
.gradientNormalizationThreshold(1.0)  
.iterations(iterations)  
.momentum(0.5)  
.momentumAfter(Collections.singletonMap(3, 0.9))  
.optimizationAlgo(OptimizationAlgorithm.CONJUGATE_GRADIENT)
```

We will also specify that our network will have four layers:

```
.list(4)
```

The input to the first layer will be 748 neurons and the output will be 500 neurons. We'll use the root mean squared-error cross entropy, Xavier algorithm, to initialize weights by automatically determining the scale of initialization based on the number of input and output neurons, as follows:

```
.layer(0, new RBM.Builder()  
.nIn(numRows*numColumns)  
.nOut(500)  
.weightInit(WeightInit.XAVIER)  
.lossFunction(LossFunction.RMSE_XENT)  
.visibleUnit(RBM.VisibleUnit.BINARY)  
.hiddenUnit(RBM.HiddenUnit.BINARY)  
.build())
```

The next two layers will have the same parameters, except the number of input and output neurons:

```
.layer(1, new RBM.Builder()  
.nIn(500)  
.nOut(250)  
.weightInit(WeightInit.XAVIER)  
.lossFunction(LossFunction.RMSE_XENT)  
.visibleUnit(RBM.VisibleUnit.BINARY)  
.hiddenUnit(RBM.HiddenUnit.BINARY)  
.build())
```

```
.layer(2, new RBM.Builder()
    .nIn(250)
    .nOut(200)
    .weightInit(WeightInit.XAVIER)
    .lossFunction(LossFunction.RMSE_XENT)
    .visibleUnit(RBM.VisibleUnit.BINARY)
    .hiddenUnit(RBM.HiddenUnit.BINARY)
    .build())
```

Now the last layer will map the neurons to outputs, where we'll use the `softmax` activation function, as follows:

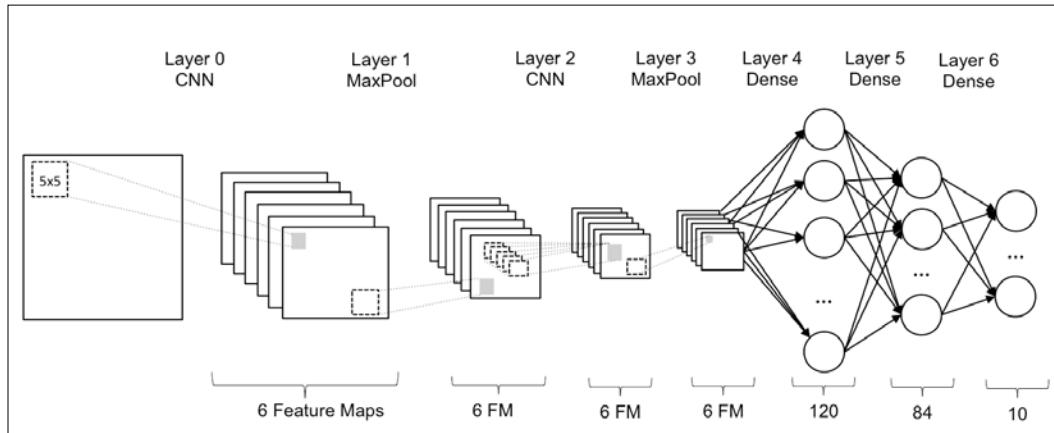
```
.layer(3, new OutputLayer.Builder()
    .nIn(200)
    .nOut(outputNum)
    .lossFunction(LossFunction.NEGATIVELOGLIKELIHOOD)
    .activation("softmax")
    .build())
    .pretrain(true).backprop(false)
    .build();
```

The rest of the training and evaluation is the same as in the single-layer network example. Note that training deep network might take significantly more time compared to a single-layer network. The accuracy should be around 93%.

Now let's take a look at another deep network.

Build a Multilayer Convolutional Network

In the final example, we'll discuss how to build a convolutional network, as shown in the following diagram. The network will consist of seven layers: first, we'll repeat two pairs of convolutional and subsampling layers with max pooling. The last subsampling layer is then connected to a densely connected feedforward neuronal network, comprising 120 neurons, 84 neurons, and 10 neurons in the last three layers, respectively. Such a network effectively forms the complete image recognition pipeline, where the first four layers correspond to feature extraction and the last three layers correspond to the learning model:



Network configuration is initialized as we did earlier:

```
MultiLayerConfiguration.Builder conf = new
    NeuralNetConfiguration.Builder()
```

We will specify the gradient descent algorithm and its parameters, as follows:

```
.seed(seed)
.iterations(iterations)
.activation("sigmoid")
.weightInit(WeightInit.DISTRIBUTION)
.dist(new NormalDistribution(0.0, 0.01))
.learningRate(1e-3)
.learningRateScoreBasedDecayRate(1e-1)
.optimizationAlgo(
    OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
```

We will also specify the seven network layers, as follows:

```
.list(7)
```

The input to the first convolutional layer is the complete image, while the output is six feature maps. The convolutional layer will apply a 5 x 5 filter, and the result will be stored in a 1 x 1 cell:

```
.layer(0, new ConvolutionLayer.Builder(
    new int[]{5, 5}, new int[]{1, 1})
    .name("cnn1")
    .nIn(numRows*numColumns)
    .nOut(6)
    .build())
```

The second layer is a subsampling layer that will take a 2×2 region and store the max result into a 2×2 element:

```
.layer(1, new SubsamplingLayer.Builder()  
    SubsamplingLayer.PoolingType.MAX,  
    new int[]{2, 2}, new int[]{2, 2})  
    .name("maxpool1")  
    .build())
```

The next two layers will repeat the previous two layers:

```
.layer(2, new ConvolutionLayer.Builder(new int[]{5, 5}, new  
    int[]{1, 1})  
    .name("cnn2")  
    .nOut(16)  
    .biasInit(1)  
    .build())  
.layer(3, new SubsamplingLayer.Builder  
    (SubsamplingLayer.PoolingType.MAX, new  
    int[]{2, 2}, new int[]{2, 2})  
    .name("maxpool2")  
    .build())
```

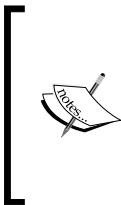
Now we will wire the output of the subsampling layer into a dense feedforward network, consisting of 120 neurons, and then through another layer, into 84 neurons, as follows:

```
.layer(4, new DenseLayer.Builder()  
    .name("ffn1")  
    .nOut(120)  
    .build())  
.layer(5, new DenseLayer.Builder()  
    .name("ffn2")  
    .nOut(84)  
    .build())
```

The final layer connects 84 neurons with 10 output neurons:

```
.layer(6, new OutputLayer.Builder  
    (LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)  
    .name("output")  
    .nOut(outputNum)  
    .activation("softmax") // radial basis function required  
    .build())  
.backprop(true)  
.pretrain(false)  
.cnnInputSize(numRows, numColumns, 1);
```

To train this structure, we can reuse the code that we developed in the previous two examples. Again, the training might take some time. The network accuracy should be around 98%.



As model training significantly relies on linear algebra, training can be significantly sped up by using **Graphics Processing Unit (GPU)** for an order of magnitude. As GPU backend is at the time of writing undergoing a rewrite, please check the latest documentation at <http://deeplearning4j.org/documentation>

As we saw in different examples, increasingly more complex neural networks allow us to extract relevant features automatically, thus completely avoiding traditional image processing. However, the price we pay for this is an increased processing time and a lot of learning examples to make this approach efficient.

Summary

In this chapter, we discussed how to recognize patterns in images in order to distinguish between different classes by covering fundamental principles of deep learning and discussing how to implement them with the deeplearning4j library. We started by refreshing the basic neural network structure and discussed how to implement them to solve handwritten digit recognition problem.

In the next chapter, we'll look further into patterns; however, instead of patterns in images, we'll tackle patterns with temporal dependencies that can be found in sensor data.

9

Activity Recognition with Mobile Phone Sensors

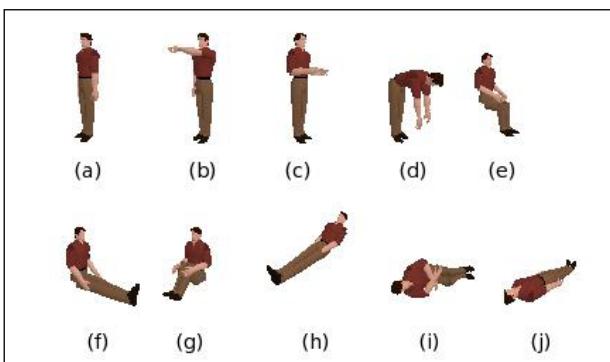
While the previous chapter focused on pattern recognition in images, this chapter is all about recognizing patterns in sensor data, which, in contrast to images, has temporal dependencies. We will discuss how to recognize granular daily activities such as walking, sitting, and running using mobile phone inertial sensors. The chapter also provides references to related research and emphasizes best practices in the activity recognition community.

The topics covered in this chapter will include the following:

- Introducing activity recognition, covering mobile phone sensors and activity recognition pipeline
- Collecting sensor data from mobile devices
- Discussing activity classification and model evaluation
- Deploying activity recognition model

Introducing activity recognition

Activity recognition is an underpinning step in behavior analysis, addressing healthy lifestyle, fitness tracking, remote assistance, security applications, elderly care, and so on. Activity recognition transforms low-level sensor data from sensors, such as accelerometer, gyroscope, pressure sensor, and GPS location, to a higher-level description of behavior primitives. In most cases, these are basic activities, for example, walking, sitting, lying, jumping, and so on, as shown in the following image, or they could be more complex behaviors, such as going to work, preparing breakfast, shopping, and so on:



In this chapter, we will discuss how to add the activity recognition functionality into a mobile application. We will first look at what does an activity recognition problem looks like, what kind of data do we need to collect, what are the main challenges are, and how to address them?

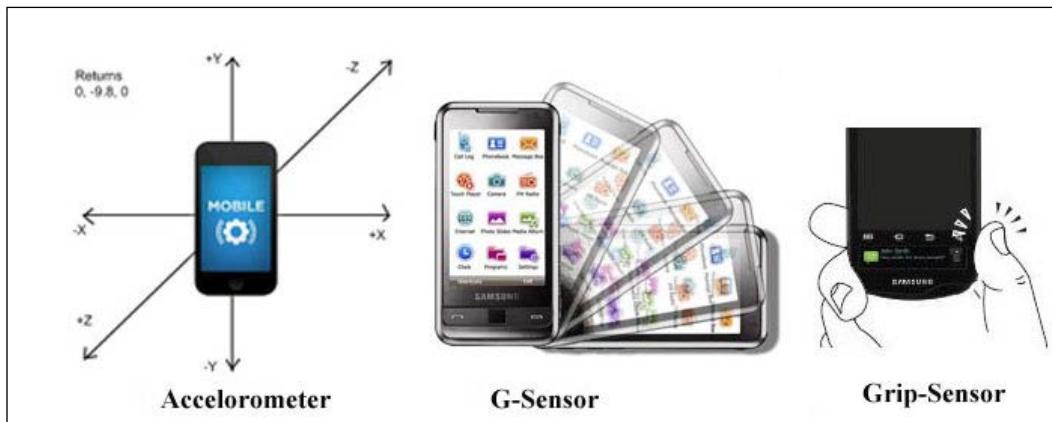
Later, we will follow an example to see how to actually implement activity recognition in an Android application, including data collection, data transformation, and building a classifier.

Let's start!

Mobile phone sensors

Let's first review what kinds of mobile phone sensors there are and what they report. Most smart devices are now equipped with a several built-in sensors that measure the motion, position, orientation, and conditions of the ambient environment. As sensors provide measurements with high precision, frequency, and accuracy, it is possible to reconstruct complex user motions, gestures, and movements. Sensors are often incorporated in various applications; for example, gyroscope readings are used to steer an object in a game, GPS data is used to locate the user, and accelerometer data is used to infer the activity that the user is performing, for example, cycling, running, or walking.

The next image shows a couple of examples what kind of interactions the sensors are able to detect:



Mobile phone sensors can be classified into the following three broad categories:

- **Motion sensors** measure acceleration and rotational forces along the three perpendicular axes. Examples of sensors in this category include accelerometers, gravity sensors, and gyroscopes.
- **Environmental sensors** measure a variety of environmental parameters, such as illumination, air temperature, pressure, and humidity. This category includes barometers, photometers, and thermometers.
- **Position sensors** measure the physical position of a device. This category includes orientation sensors and magnetometers.

More detailed descriptions for different mobile platforms are available at the following links:

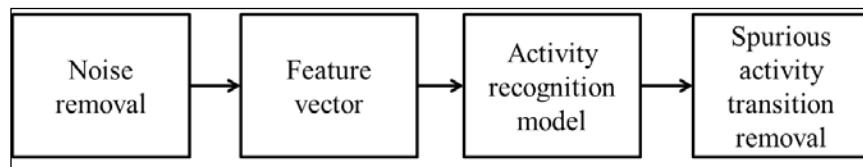
- Android sensors framework: http://developer.android.com/guide/topics/sensors/sensors_overview.html
- iOS Core Motion framework: https://developer.apple.com/library/ios/documentation/CoreMotion/Reference/CoreMotion_Reference/
- Windows Phone: [https://msdn.microsoft.com/en-us/library/windows/apps/hh202968\(v=vs.105\).aspx](https://msdn.microsoft.com/en-us/library/windows/apps/hh202968(v=vs.105).aspx)

In this chapter, we will work only with Android's sensors framework.

Activity recognition pipeline

Classifying multidimensional time-series sensor data is inherently more complex compared to classifying traditional, nominal data as we saw in the previous chapters. First, each observation is temporally connected to the previous and following observations, making it very difficult to apply a straightforward classification of a single set of observations only. Second, the data obtained by sensors at different time points stochastic, that is unpredictable due to influence of sensor noise, environmental disturbances, and many other reasons. Moreover, an activity can comprise various sub-activities executed in different manner and each person performs the activity a bit differently, which results in high intraclass differences. Finally, all these reasons make an activity recognition model imprecise, resulting in new data being often misclassified. One of the highly desirable properties of an activity recognition classifier is to ensure continuity and consistency in the recognized activity sequence.

To deal with these challenges, activity recognition is applied to a pipeline as shown in the following:



In the first step, we attenuate as much noise as we can, for example, by reducing sensor sampling rate, removing outliers, applying high/low-pass filters, and so on. In the next phase, we construct a feature vector, for instance, we convert sensor data from time domain to frequency domain by applying **Discrete Fourier Transform (DFT)**. DFT is a method that takes a list of samples as an input and returns a list of sinusoid coefficients ordered by their frequencies. They represent a combination of frequencies that are present in the original list of samples.

[ An gentle introduction of Fourier transform is written by Pete Bevelacqua at <http://www.thefouriertransform.com/>. If you want to get more technical and theoretical background on the Fourier transform, take a look at the eighth and ninth lectures in the class by Robert Gallager and Lizhong Zheng at MIT open course:
<http://theopenacademy.com/content/principles-digital-communication>]

Next, based on the feature vector and set of training data, we can build an activity recognition model that assigns an atomic action to each observation. Therefore, for each new sensor reading, the model will output the most probable activity label. However, models make mistakes. Hence, the last phase smooths the transitions between activities by removing transitions that cannot occur in reality, for example, it is not physically feasible that the transition between activities lying-standing-lying occurs in less than half a second, hence such transition between activities is smoothed as lying-lying-lying.

The activity recognition model is constructed with a supervised learning approach, which consists of training and classification steps. In the training step, a set of labeled data is provided to train the model. The second step is used to assign a label to the new unseen data by the trained model. The data in both phases must be pre-processed with the same set of tools, such as filtering and feature-vector computation.

The post-processing phase, that is, spurious activity removal, can also be a model itself and, hence, also requires a learning step. In this case, the pre-processing step also includes activity recognition, which makes such arrangement of classifiers a meta-learning problem. To avoid overfitting, it is important that the dataset used for training the post-processing phase is not the same as that used for training the activity recognition model.

We will roughly follow a lecture on smartphone programming by professor Andrew T. Campbell from Dartmouth University and leverage data collection mobile app that they developed in the class (Campbell, 2011).

The plan

The plan consists of training phase and deployment phase. Training phase shown in the following image boils down to the following steps:

1. Install Android Studio and import `MyRunsDataCollector.zip`.
2. Load the application in your Android phone.
3. Collect your data, for example, standing, walking, and running, and transform the data to a feature vector comprising of FFT transforms. Don't panic, low-level signal processing functions such as FFT will not be written from scratch as we will use existing code to do that. The data will be saved on your phone in a file called `features.arff`.
4. Create and evaluate an activity recognition classifier using exported data and implement filter for spurious activity transitions removal.
5. Plug the classifier back into the mobile application.

If you don't have an Android phone, or if you want to skip all the steps related to mobile application, just grab an already-collected dataset located in `data/features.arff` and jump directly to the *Building a classifier* section.

Collecting data from a mobile phone

This section describes the first three steps from the plan. If you want to directly work with the data, you can just skip this section and continue to the following *Building a classifier* section. There are many open source mobile apps for sensor data collection, including an app by Prof. Campbell that we will use in this chapter. The application implements the essentials to collect sensor data for different activity classes, for example, standing, walking, running, and others.

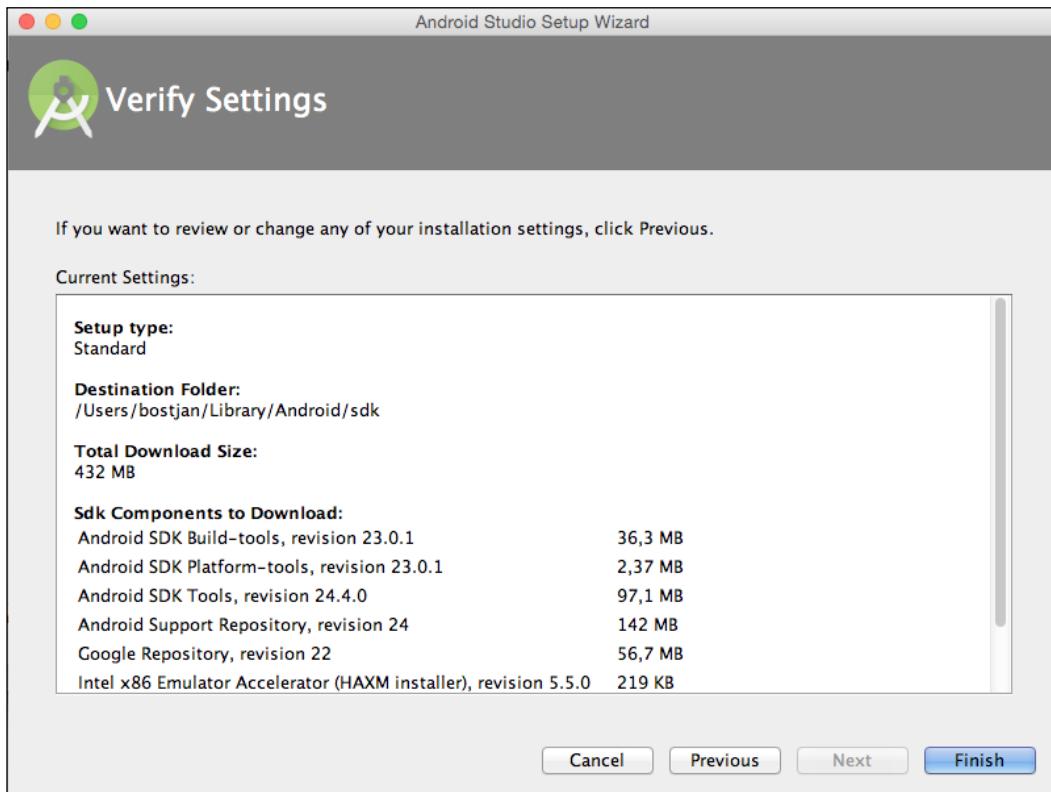
Let's start by preparing the Android development environment. If you have already installed it, jump to the *Loading the data collector* section.

Installing Android Studio

Android Studio is a development environment for Android platform. We will quickly review installation steps and basic configurations required to start the app on a mobile phone. For more detailed introduction to Android development, I would recommend an introductory book, *Android 5 Programming by Example* by Kyle Mew.

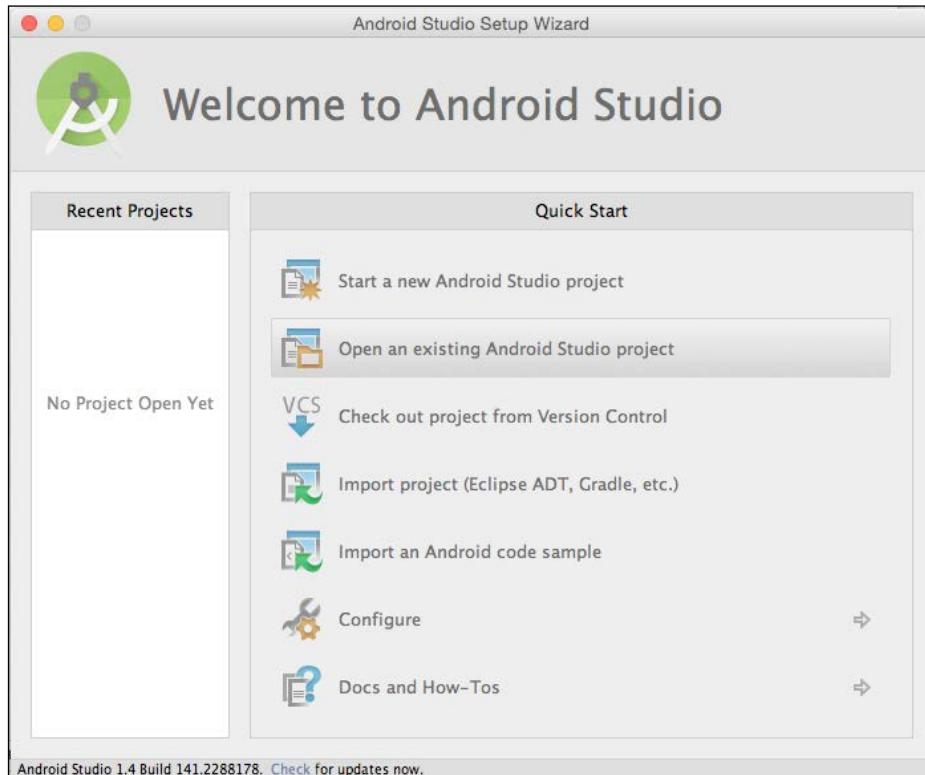
Grab the latest Android Studio for developers at <http://developer.android.com/sdk/installing/index.html?pkg=studio> and follow the installation instructions. The installation will take over 10 minutes, occupying approximately 0.5 GB of space:

Then:

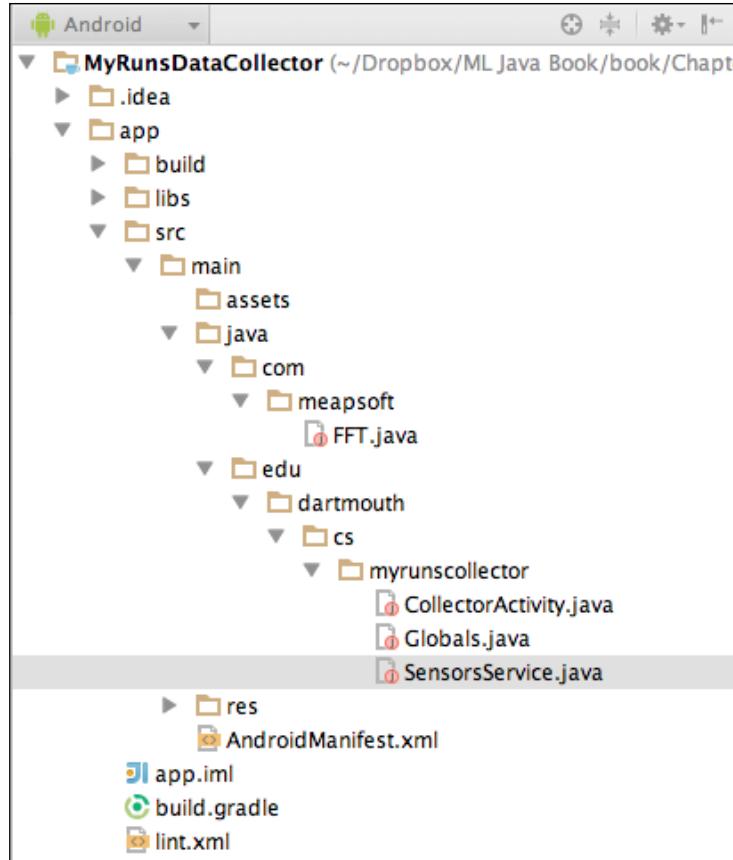


Loading the data collector

First, grab source code of MyRunsDataCollector from <http://www.cs.dartmouth.edu/~campbell/cs65/code/myrunsdatacollector.zip>. Once the Android Studio is installed, choose to open an existing Android Studio project as shown in the following image and select the MyRunsDataCollector folder. This will import the project to Android Studio:



After the project import is completed, you should be able to see the project files structure, as shown in the following image. As shown in the following, the collector consists of `CollectorActivity.java`, `Globals.java`, and `SensorsService.java`. The project also shows `FFT.java` implementing low-level signal processing:



The main `myrunscollector` package contains the following classes:

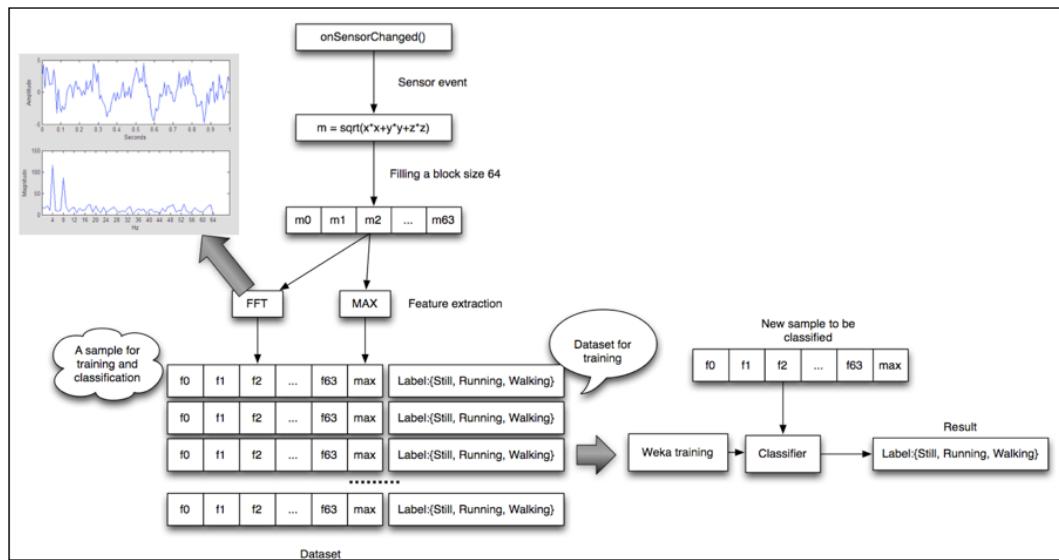
- `Globals.java`: This defines global constants such as activity labels and IDs, data filenames, and so on
- `CollectorActivity.java`: This implements user interface actions, that is, what happens when specific button is pressed
- `SensorsService.java`: This implements a service that collects data, calculates the feature vector as we will discuss in the following sections, and stores the data into a file on the phone

The next question that we will address is how to design features.

Feature extraction

Finding an appropriate representation of the person's activities is probably the most challenging part of activity recognition. The behavior needs to be represented with simple and general features so that the model using these features will also be general and work well on behaviors different from those in the learning set.

In fact, it is not difficult to design features specific to the captured observations in a training set; such features would work well on them. However, as the training set captures only a part of the whole range of human behavior, overly specific features would likely fail on general behavior:



Let's see how it is implemented in `MyRunsDataCollector`. When the application is started, a method called `onSensorChanged()` gets a triple of accelerometer sensor readings (x , y , and z) with a specific time stamp and calculates the magnitude from the sensor readings. The methods buffers up to 64 consecutive magnitudes marked before computing the FFT coefficients (Campbell, 2015):

"As shown in the upper left of the diagram, FFT transforms a time series of amplitude over time to magnitude (some representation of amplitude) across frequency; the example shows some oscillating system where the dominant frequency is between 4-8 cycles/second called Hertz (H) – imagine a ball attached to an elastic band that this stretched and oscillates for a short period of time, or your gait while walking, running -- one could look at these systems in the time and frequency domains. The x,y,z accelerometer readings and the magnitude are time domain variables. We transform these time domain data into the frequency domain because the can represent the distribution in a nice compact manner that the classifier will use to build a decision tree model. For example, the rate of the amplitude transposed to the frequency domain may look something like the figure bottom plot -- the top plot is time domain and the bottom plot a transformation of the time to the frequency domain.

The training phase also stores the maximum (MAX) magnitude of the ($m0..m63$) and the user supplied label (e.g., walking) using the collector. The individual features are computed as magnitudes ($f0..f63$), the MAX magnitude and the class label."

Now let's move on to the actual data collection.

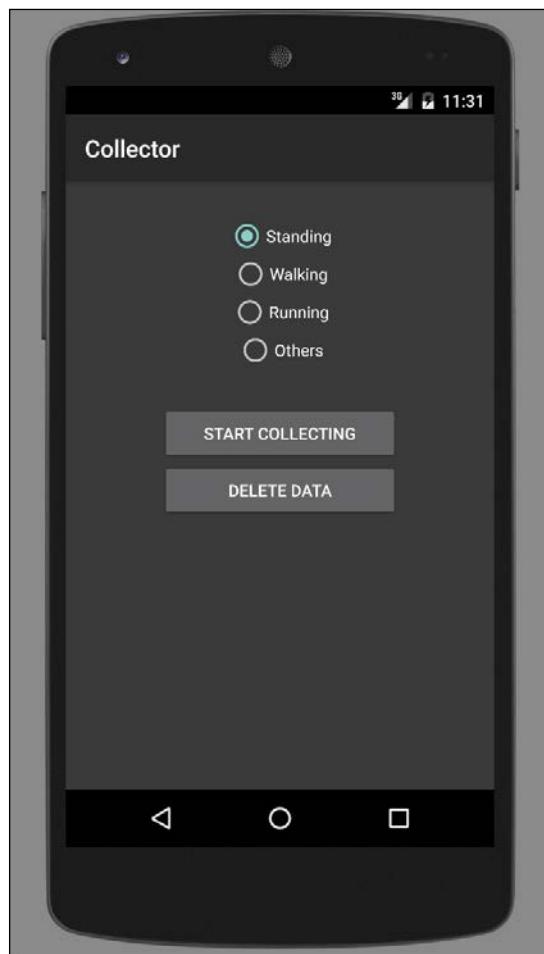
Collecting training data

We can now use the collector to collect training data for activity recognition. The collector supports three activities by default: standing, walking, and running, as shown in the application screenshot in the following figure.

You can select an activity, that is, target class value, and start recording the data by clicking the **START COLLECTING** button . Make sure that each activity is recorded for at least three minutes, for example, if the **Walking** activity is selected, press **START COLLECTING** and walk around for at least three minutes. At the end of the activity, press stop collecting. Repeat this for each of the activities.

You could also collect different scenarios involving these activities, for example, walking in the kitchen, walking outside, walking in a line, and so on. By doing so, you will have more data for each activity class and a better classifier. Makes sense, right? The more data, the less confused the classifier will be. If you only have a little data, overfitting will occur and the classifier will confuse classes—standing with walking, walking with running. However, the more data, the less they get confused. You might collect less than three minutes per class when you are debugging, but for your final polished product, the more data, the better it is. Multiple recording instances will simply be accumulated in the same file.

Note, the delete button removes the data that is stored in a file on the phone. If you want to start over again, hit delete before starting otherwise, the new collected data will be appended at the end of the file:

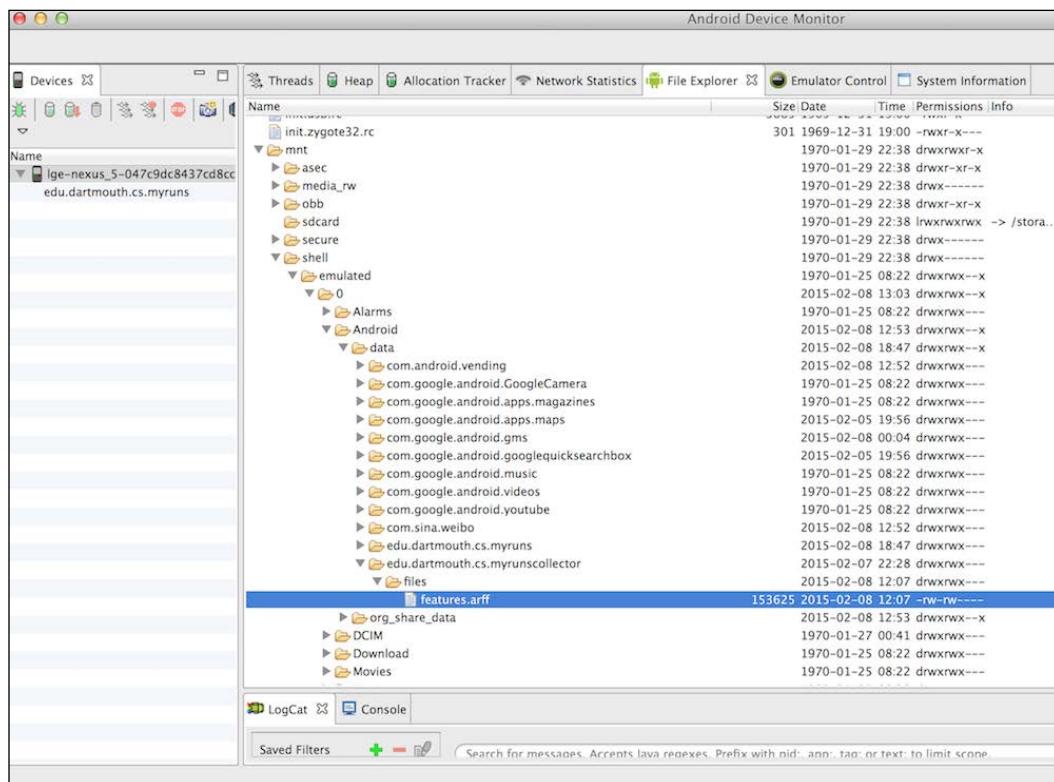


The collector implements the diagram as discussed in the previous sections: it collects accelerometer samples, computes the magnitudes, uses the `FFT.java` class to compute the coefficients, and produces the feature vectors. The data is then stored in a Weka formatted `features.arff` file. The number of feature vectors will vary as you will collect a small or large amount of data. The longer you collect the data, the more feature vectors are accumulated.

Once you stop collecting the training data using the collector tool, we need to grab the data to carry on the workflow. We can use the file explorer in **Android Device Monitor** to upload the `features.arff` file from the phone and to store it on the computer. You can access your Android Device Monitor by clicking on the Android robot icon as shown in the following image:



By selecting your device on the left, your phone storage content will be shown on the right-hand side. Navigate through `/mnt/shell/emulated/Android/data/edu.dartmouth.cs.myrunscollector/files/features.arff`:



To upload this file to your computer, you need to select the file (it is highlighted) and click **Upload**.

Now we are ready to build a classifier.

Building a classifier

Once sensor samples are represented as feature vectors having the class assigned, it is possible to apply standard techniques for supervised classification, including feature selection, feature discretization, model learning, k-fold cross validation, and so on. The chapter will not delve into the details of the machine learning algorithms. Any algorithm that supports numerical features can be applied, including SVMs, random forest, AdaBoost, decision trees, neural networks, multi-layer perceptrons, and others.

Therefore, let's start with a basic one, decision trees: load the dataset, build set class attribute, build a decision tree model, and output the model:

```
String databasePath = "/Users/bostjan/Dropbox/ML Java Book/book/
datasets/chap9/features.arff";

// Load the data in arff format
Instances data = new Instances(new BufferedReader(new
    FileReader(databasePath)));

// Set class the last attribute as class
data.setClassIndex(data.numAttributes() - 1);

// Build a basic decision tree model
String[] options = new String[] {};
J48 model = new J48();
model.setOptions(options);
model.buildClassifier(data);

// Output decision tree
System.out.println("Decision tree model:\n"+model);
```

The algorithm first outputs the model, as follows:

Decision tree model:

J48 pruned tree

max <= 10.353474

```

|   fft_coef_0000 <= 38.193106: standing (46.0)
|   fft_coef_0000 > 38.193106
|   |   fft_coef_0012 <= 1.817792: walking (77.0/1.0)
|   |   fft_coef_0012 > 1.817792
|   |   |   max <= 4.573082: running (4.0/1.0)
|   |   |   max > 4.573082: walking (24.0/2.0)
max > 10.353474: running (93.0)

```

Number of Leaves : 5

Size of the tree : 9

The tree is quite simplistic and seemingly accurate as majority class distributions in the terminal nodes are quite high. Let's run a basic classifier evaluation to validate the results:

```

// Check accuracy of model using 10-fold cross-validation
Evaluation eval = new Evaluation(data);
eval.crossValidateModel(model, data, 10, new Random(1), new
String[] {});
System.out.println("Model performance:\n"+
eval.toSummaryString());

```

This outputs the following model performance:

Correctly Classified Instances	226	92.623 %
Incorrectly Classified Instances	18	7.377 %
Kappa statistic	0.8839	
Mean absolute error	0.0421	
Root mean squared error	0.1897	
Relative absolute error	13.1828 %	
Root relative squared error	47.519 %	
Coverage of cases (0.95 level)	93.0328 %	
Mean rel. region size (0.95 level)	27.8689 %	
Total Number of Instances	244	

The classification accuracy scores very high, 92.62%, which is an amazing result. One important reason why the result is so good lies in our evaluation design. What I mean here is the following: sequential instances are very similar to each other, if we split them randomly during a 10-fold cross validation, there is a high chance that we use almost identical instances for both training and testing; hence, straightforward k-fold cross validation produces an optimistic estimate of model performance.

A better approach is to use folds that correspond to different sets of measurements or even different people. For example, we can use the application to collect learning data of five people. Then, it makes sense to run k-person cross validation, where the model is trained on four people and tested on the fifth person. The procedure is repeated for each person and the results are averaged. This will give us a much more realistic estimate of the model performance.

Leaving evaluation comment aside, let's look at how to deal with classifier errors.

Reducing spurious transitions

At the end of the activity recognition pipeline, we want to make sure that the classifications are not too volatile, that is, we don't want activities to change every millisecond. A basic approach is to design a filter that ignores quick changes in the activity sequence.

We build a filter that remembers the last window activities and returns the most frequent one. If there are multiple activities with the same score, it returns the most recent one.

First, we create a new `SpuriousActivityRemoval` class that will hold a list of activities and the `window` parameter:

```
class SpuriousActivityRemoval {  
  
    List<Object> last;  
    int window;  
  
    public SpuriousActivityRemoval(int window) {  
        this.last = new ArrayList<Object>();  
        this.window = window;  
    }  
}
```

Next, we create the `Object filter(Object)` method that will take an activity and return a filtered activity. The method first checks whether we have enough observations. If not, it simply stores the observation and returns the same value, as shown in the following code:

```
public Object filter(Object obj) {  
    if(last.size() < window) {  
        last.add(obj);  
        return obj;  
    }  
}
```

If we already collected window observations, we simply return the most frequent observation, remove the oldest observation, and insert the new observation:

```
Object o = getMostFrequentElement(last);
last.add(obj);
last.remove(0);
return o;
}
```

What is missing here is a function that returns the most frequent element from a list of objects. We implement this with a hash map, as follows:

```
private Object getMostFrequentElement(List<Object> list) {

    HashMap<String, Integer> objectCounts = new HashMap<String,
        Integer>();
    Integer frequentCount = 0;
    Object frequentObject = null;

    for(Object obj : list) {
        String key = obj.toString();
        Integer count = objectCounts.get(key);
        if(count == null) {
            count = 0;
        }
        objectCounts.put(key, ++count);

        if(count >= frequentCount) {
            frequentCount = count;
            frequentObject = obj;
        }
    }

    return frequentObject;
}
```

Let's run a simple example:

```
String[] activities = new String[]{"Walk", "Walk", "Walk", "Run",
    "Walk", "Run", "Run", "Sit", "Sit", "Sit"};
SpuriousActivityRemoval dlpFilter = new
    SpuriousActivityRemoval(3);
```

```
for(String str : activities){  
    System.out.println(str + " -> " + dlpFilter.filter(str));  
}
```

The example outputs the following activities:

```
Walk -> Walk  
Walk -> Walk  
Walk -> Walk  
Run -> Walk  
Walk -> Walk  
Run -> Walk  
Run -> Run  
Sit -> Run  
Sit -> Run  
Sit -> Sit
```

The result is a continuous sequence of activities, that is, we do not have quick changes. This adds some delay, but unless this is absolutely critical for the application, it is acceptable.

Activity recognition may be enhanced by appending n previous activities as recognized by the classifier to the feature vector. The danger of appending previous activities is that the machine learning algorithm may learn that the current activity is always the same as the previous one, as this will often be the case. The problem may be solved by having two classifiers, A and B: the classifier B's attribute vector contains n previous activities as recognized by the classifier A. The classifier A's attribute vector does not contain any previous activities. This way, even if B gives a lot of weight to the previous activities, the previous activities as recognized by A will change as A is not burdened with B's inertia.

All that remains to do is to embed the classifier and filter into our mobile application.

Plugging the classifier into a mobile app

There are two ways to incorporate a classifier into a mobile application. The first one involves exporting a model in the Weka format, using the Weka library as a dependency in our mobile application, loading the model, and so on. The procedure is identical to the example we saw in *Chapter 3, Basic Algorithms – Classification, Regression, and Clustering*. The second approach is more lightweight; we export the model as a source code, for example, we create a class implementing the decision tree classifier. Then we can simply copy and paste the source code into our mobile app, without even importing any Weka dependencies.

Fortunately, some Weka models can be easily exported to source code by the `toSource(String)` function:

```
// Output source code implementing the decision tree
System.out.println("Source code:\n" +
    model.toSource("ActivityRecognitionEngine"));
```

This outputs an `ActivityRecognitionEngine` class that corresponds to our model. Now, let's take a closer look at the outputted code:

```
class ActivityRecognitionEngine {

    public static double classify(Object[] i)
        throws Exception {

        double p = Double.NaN;
        p = ActivityRecognitionEngine.N17a7cec20(i);
        return p;
    }
    static double N17a7cec20(Object []i) {
        double p = Double.NaN;
        if (i[64] == null) {
            p = 1;
        } else if (((Double) i[64]).doubleValue() <= 10.353474) {
            p = ActivityRecognitionEngine.N65b3120a1(i);
        } else if (((Double) i[64]).doubleValue() > 10.353474) {
            p = 2;
        }
        return p;
    }
    ...
}
```

The outputted `ActivityRecognitionEngine` class implements the decision tree that we discussed earlier. The machine-generated function names, such as `N17a7cec20(Object [])`, correspond to decision tree nodes. The classifier can be called by the `classify(Object [])` method, where we should pass a feature vector obtained by the same procedure as we discussed in the previous sections. As usual, it returns a double, indicating a class label index.

Summary

In this chapter, we discussed how to implement an activity recognition model for mobile applications. We looked into the completed process, including data collection, feature extraction, model building, evaluation, and model deployment.

In the next chapter, we will move on to another Java library targeted at text analysis—Mallet.

10

Text Mining with Mallet – Topic Modeling and Spam Detection

In this chapter, we will first discuss what text mining is, what kind of analysis is it able to offer, and why you might want to use it in your application. We will then discuss how to work with Mallet, a Java library for natural language processing, covering data import and text pre-processing. Afterwards, we will look into two text mining applications: topic modeling, where we will discuss how text mining can be used to identify topics found in the text documents without reading them individually; and spam detection, where we will discuss how to automatically classify text documents into categories.

This chapter will cover the following topics:

- Introducing text mining
- Installing and working with Mallet
- Topic modeling
- Spam detection

Introducing text mining

Text mining, or text analytics, refers to the process of automatically extracting high-quality information from text documents, most often written in natural language, where high-quality information is considered to be relevant, novel, and interesting.

While a typical text-analytics application is to scan a set of documents to generate a search index, text mining can be used in many other applications, including text categorization into specific domains; text clustering to automatically organize a set of documents; sentiment analysis to identify and extract subjective information in documents; concept/entity extraction that is capable of identifying people, places, organizations, and other entities from documents; document summarization to automatically provide the most important points in the original document; and learning relations between named entities.

The process based on statistical pattern mining usually involves the following steps:

1. Information retrieval and extraction.
2. Transforming unstructured text data into structured; for example, parsing, removing noisy words, lexical analysis, calculating word frequencies, deriving linguistic features, and so on.
3. Discovery of patterns from structured data and tagging/annotation.
4. Evaluation and interpretation of the results.

Later in this chapter, we will look at two application areas: topic modeling and text categorization. Let's examine what they bring to the table.

Topic modeling

Topic modeling is an unsupervised technique and might be useful if you need to analyze a large archive of text documents and wish to understand what the archive contains, without necessarily reading every single document by yourself. A text document can be a blog post, e-mail, tweet, document, book chapter, diary entry, and so on. Topic modeling looks for patterns in a corpus of text; more precisely, it identifies topics as lists of words that appear in a statistically meaningful way. The most well-known algorithm is **Latent Dirichlet Allocation** (Blei et al, 2003), which assumes that author composed a piece of text by selecting words from possible baskets of words, where each basket corresponds to a topic. Using this assumption, it becomes possible to mathematically decompose text into the most likely baskets from where the words first came. The algorithm then iterates over this process until it converges to the most likely distribution of words into baskets, which we call topics.

For example, if we use topic modeling on a series of news articles, the algorithm would return a list of topics and keywords that most likely comprise of these topics. Using the example of news articles, the list might look similar to the following:

- Winner, goal, football, score, first place
- Company, stocks, bank, credit, business
- Election, opponent, president, debate, upcoming

By looking at the keywords, we can recognize that the news articles were concerned with sports, business, upcoming election, and so on. Later in this chapter, we will learn how to implement topic modeling using the news article example.

Text classification

In text classification, or text categorization, the goal is to assign a text document according to its content to one or more classes or categories, which tend to be a more general subject area such as vehicles or pets. Such general classes are referred to as topics, and the classification task is then called text classification, text categorization, topic classification, or topic spotting. While documents can be categorized according to other attributes such as document type, author, printing year, and so on, the focus in this chapter will be on the document content only. Examples of text classification include the following components:

- Spam detection in e-mail messages, user comments, webpages, and so on
- Detection of sexually-explicit content
- Sentiment detection, which automatically classifies a product/service review as positive or negative
- E-mail sorting according to e-mail content
- Topic-specific search, where search engines restrict searches to a particular topic or genre, thus providing more accurate results

These examples show how important text classification is in information retrieval systems, hence most modern information retrieval systems use some kind of text classifier. The classification task that we will use as an example in this book is text classification for detecting e-mail spam.

We continue this chapter with an introduction to Mallet, a Java-based package for statistical natural language processing, document classification, clustering, topic modeling, information extraction, and other machine learning applications to text. We will then cover two text-analytics applications, namely, topics modeling and spam detection as text classification.

Installing Mallet

Mallet is available for download at UMass Amherst University website at <http://mallet.cs.umass.edu/download.php>. Navigate to the **Download** section as shown in the following image and select the latest stable release (2.0.8, at the time of writing this book):

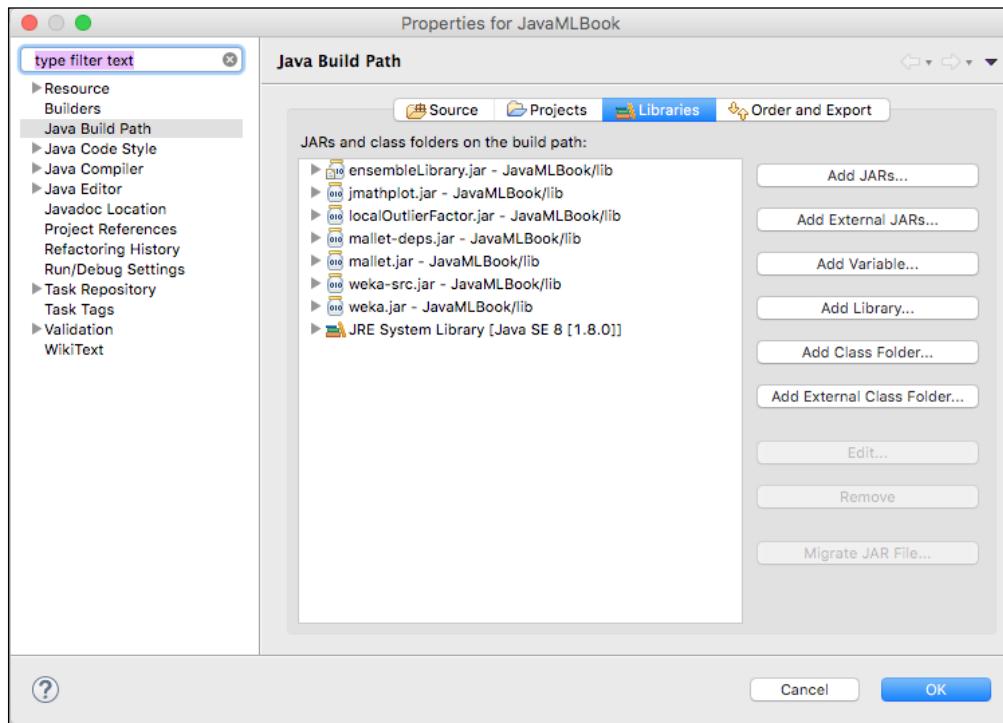
The screenshot shows the Mallet website's download page. The header features the Mallet logo and the text "MAchine Learning for LanguagE Toolkit". On the left, a sidebar menu includes links for Home, Tutorial slides / video, Download, API, Quick Start, Sponsors, Mailing List, About, Importing Data, Classification, Sequence Tagging, Topic Modeling, Optimization, and Graphical Models. A note at the bottom of the sidebar states: "MALLET is open source software [License]. For research use, please remember to cite MALLET". The main content area starts with a "Current release" section: "The following packaged release of MALLET 2.0 is available: mallet-2.0.8RC3.tar.gz mallet-2.0.8RC3.zip". It notes that 2.0.8RC3 is an official release and 2.0.7 is still available. Below this is a "Windows installation" section with instructions to set the environment variable %MALLET_HOME% to point to the MALLET directory. It also provides instructions for a "Development release" via GitHub and details for building the package using Apache Ant. At the bottom, a list of older releases is provided:

- [mallet-2.0.6.tar.gz](#)
- [mallet-2.0.5.tar.gz \(notes\)](#)
- [mallet-2.0-RC4.tar.gz \(notes\)](#)
- [mallet-2.0-RC3.tar.gz \(notes\)](#)
- [mallet-2.0-RC2.tar.gz](#)
- [mallet-2.0-RC1.tar.gz](#)
- [mallet-0.4.tar.gz](#)

Download the ZIP file and extract the content. In the extracted directory, you should find a folder named `dist` with two JAR files: `mallet.jar` and `mallet-deps.jar`. The first one contains all the packaged Mallet classes, while the second one packs all the dependencies. Include both JARs in your project as referenced libraries, as shown in the following image:

Name	Size	Kind
▶ bin	--	Folder
build.xml	3 KB	XML
▶ class	--	Folder
dist	--	Folder
mallet-deps.jar	2,6 MB	Java JAR file
mallet.jar	2,2 MB	Java JAR file
▶ lib	--	Folder
LICENSE	12 KB	TextEd...ument
Makefile	4 KB	TextEd...ument
pom.xml	3 KB	XML
README.md	2 KB	Markd...cument
▶ sample-data	--	Folder
▶ src	--	Folder
▶ stoplists	--	Folder
▶ test	--	Folder

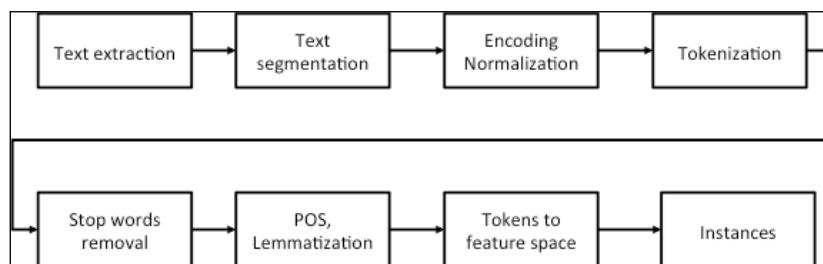
If you are using Eclipse, right click on **Project**, select **Properties**, and pick **Java Build Path**. Select the **Libraries** tab and click **Add External JARs**. Now, select the two JARs and confirm, as shown in the following screenshot:



Now we are ready to start using Mallet.

Working with text data

One of the main challenges in text mining is transforming unstructured written natural language into structured attribute-based instances. The process involves many steps as shown in the following image:



First, we extract some text from the Internet, existing documents, or databases. At the end of the first step, the text could still be presented in the XML format or some other proprietary format. The next step is to, therefore, extract the actual text only and segment it into parts of the document, for example, title, headline, abstract, body, and so on. The third step is involved with normalizing text encoding to ensure the characters are presented the same way, for example, documents encoded in formats such as ASCII, ISO 8859-1, and Windows-1250 are transformed into Unicode encoding. Next, tokenization splits the document into particular words, while the following step removes frequent words that usually have low predictive power, for example, the, a, I, we, and so on.

The **part-of-speech (POS)** tagging and lemmatization step could be included to transform each token (that is, word) to its basic form, which is known as lemma, by removing word endings and modifiers. For example, running becomes run, better becomes good, and so on. A simplified approach is stemming, which operates on a single word without any context of how the particular word is used, and therefore, cannot distinguish between words having different meaning, depending on the part of speech, for example, axes as plural of axe as well as axis.

The last step transforms tokens into a feature space. Most often feature space is a **bag-of-words (BoW)** presentation. In this presentation, a set of all words appearing in the dataset is created, that is, a bag of words. Each document is then presented as a vector that counts how many times a particular word appears in the document.

Consider the following example with two sentences:

- Jacob likes table tennis. Emma likes table tennis too.
- Jacob also likes basketball.

The bag of words in this case consists of {Jacob, likes, table, tennis, Emma, too, also, basketball}, which has eight distinct words. The two sentences could be now presented as vectors using the indexes of the list, indicating how many times a word at a particular index appears in the document, as follows:

- [1, 2, 2, 2, 1, 0, 0, 0]
- [1, 1, 0, 0, 0, 0, 1, 1]

Such vectors finally become instances for further learning.

 Another very powerful presentation based on the BoW model is **word2vec**. Word2vec was introduced in 2013 by a team of researchers led by Tomas Mikolov at Google. Word2vec is a neural network that learns distributed representations for words. An interesting property of this presentation is that words appear in clusters, such that some word relationships, such as analogies, can be reproduced using vector math. A famous example shows that king - man + woman returns queen.

Further details and implementation are available at the following link:

<https://code.google.com/archive/p/word2vec/>

Importing data

In this chapter, we will not look into how to scrap a set of documents from a website or extract them from database. Instead, we will assume that we already collected them as set of documents and store them in the .txt file format. Now let's look at two options how to load them. The first option addresses the situation where each document is stored in its own .txt file. The second option addresses the situation where all the documents are stored in a single file, one per line.

Importing from directory

Mallet supports reading from directory with the `cc.mallet.pipe.iterator.FileIterator` class. File iterator is constructed with the following three parameters:

- A list of `File []` directories with text files
- File filter that specifies which files to select within a directory
- A pattern that is applied to a filename to produce a class label

Consider the data structured into folders as shown in the following image. We have documents organized in five topics by folders (`tech`, `entertainment`, `politics`, and `sport`, `business`). Each folder contains documents on particular topics, as shown in the following image:

Name
► tech
► entertainment
► politics
► sport
▼ business
003.txt
004.txt
008.txt
009.txt
014.txt
015.txt

In this case, we initialize iterator as follows:

```
FileIterator iterator =
    new FileIterator(new File[]{new File("path-to-my-dataset")},
    new TxtFilter(),
    FileIterator.LAST_DIRECTORY);
```

The first parameter specifies the path to our root folder, the second parameter limits the iterator to the .txt files only, while the last parameter asks the method to use the last directory name in the path as class label.

Importing from file

Another option to load the documents is through `cc.mallet.pipe.iterator.CsvIterator.CsvIterator(Reader, Pattern, int, int, int)`, which assumes all the documents are in a single file and returns one instance per line extracted by a regular expression. The class is initialized by the following components:

- Reader: This is the object that specifies how to read from a file
- Pattern: This is a regular expression, extracting three groups: data, target label, and document name
- int, int, int: These are the indexes of data, target, and name groups as they appear in a regular expression

Consider a text document in the following format, specifying document name, category and content:

```
AP881218 local-news A 16-year-old student at a private  
Baptist...  
AP880224 business The Bechtel Group Inc. offered in 1985 to...  
AP881017 local-news A gunman took a 74-year-old woman hostage...  
AP900117 entertainment Cupid has a new message for lovers  
this...  
AP880405 politics The Reagan administration is weighing w...
```

To parse a line into three groups, we can use the following regular expression:

```
^(\\S*) [\\s,]* (\\S*) [\\s,]* (.*)$
```

There are three groups that appear in parenthesis, (), where the third group contains the data, the second group contains the target class, and the first group contains the document ID. The iterator is initialized as follows:

```
CsvIterator iterator = new CsvIterator (   
    fileReader,  
    Pattern.compile("^(\\S*) [\\s,]* (\\S*) [\\s,]* (.*)$") ,  
    3, 2, 1));
```

Here the regular expression extracts the three groups separated by an empty space and their order is 3, 2, 1.

Now let's move to data pre-processing pipeline.

Pre-processing text data

Once we initialized an iterator that will go through the data, we need to pass the data through a sequence of transformations as described at the beginning of this section. Mallet supports this process through a pipeline and a wide variety of steps that could be included in a pipeline, which are collected in the `cc.mallet.pipe` package. Some examples are as follows:

- `Input2CharSequence`: This is a pipe that can read from various kinds of text sources (either URI, File, or Reader) into `CharSequence`
- `CharSequenceRemoveHTML`: This pipe removes HTML from `CharSequence`
- `MakeAmpersandXMLFriendly`: This converts & to & in tokens of a token sequence
- `TokenSequenceLowercase`: This converts the text in each token in the token sequence in the data field to lower case

- `TokenSequence2FeatureSequence`: This converts the token sequence in the data field of each instance to a feature sequence
- `TokenSequenceNGrams`: This converts the token sequence in the data field to a token sequence of ngrams, that is, combination of two or more words

 The full list of processing steps is available in the following Mallet documentation:
<http://mallet.cs.umass.edu/api/index.html?cc/mallet/pipe/iterator/package-tree.html>

Now we are ready to build a class that will import our data.

First, let's build a pipeline, where each processing step is denoted as a pipeline in Mallet. Pipelines can be wired together in a serial fashion with a list of `ArrayList<Pipe>` objects:

```
ArrayList<Pipe> pipeList = new ArrayList<Pipe>();
```

Begin by reading data from a file object and converting all the characters into lower case:

```
pipeList.add(new Input2CharSequence("UTF-8"));
pipeList.add(new CharSequenceLowercase());
```

Next, tokenize raw strings with a regular expression. The following pattern includes Unicode letters and numbers and the underscore character:

```
Pattern tokenPattern =
    Pattern.compile("[\\p{L}\\p{N}_]+");

pipeList.add(new CharSequence2TokenSequence(tokenPattern));
```

Remove stop words, that is, frequent words with no predictive power, using a standard English stop list. Two additional parameters indicate whether stop word removal should be case-sensitive and mark deletions instead of just deleting the words. We'll set both of them to `false`:

```
pipeList.add(new TokenSequenceRemoveStopwords(false, false));
```

Instead of storing the actual words, we can convert them into integers, indicating a word index in the bag of words:

```
pipeList.add(new TokenSequence2FeatureSequence());
```

We'll do the same for the class label; instead of label string, we'll use an integer, indicating a position of the label in our bag of words:

```
pipeList.add(new Target2Label());
```

We could also print the features and the labels by invoking the `PrintInputAndTarget` pipe:

```
pipeList.add(new PrintInputAndTarget());
```

Finally, we store the list of pipelines in a `SerialPipes` class that will covert an instance through a sequence of pipes:

```
SerialPipes pipeline = new SerialPipes(pipeList);
```

Now let's take a look at how apply this in a text mining application!

Topic modeling for BBC news

As discussed earlier, the goal of topic modeling is to identify patterns in a text corpus that correspond to document topics. In this example, we will use a dataset originating from BBC news. This dataset is one of the standard benchmarks in machine learning research, and is available for non-commercial and research purposes.

The goal is to build a classifier that is able to assign a topic to an uncategorized document.

BBC dataset

Greene and Cunningham (2006) collected the BBC dataset to study a particular document-clustering challenge using support vector machines. The dataset consists of 2,225 documents from the BBC News website from 2004 to 2005, corresponding to the stories collected from five topical areas: business, entertainment, politics, sport, and tech. The dataset can be grabbed from the following website:

<http://mlg.ucd.ie/datasets/bbc.html>

Download the raw text files under the **Dataset: BBC** section. You will also notice that the website contains already processed dataset, but for this example, we want to process the dataset by ourselves. The ZIP contains five folders, one per topic. The actual documents are placed in the corresponding topic folder, as shown in the following screenshot:

The screenshot shows a website interface with a navigation bar at the top. The 'Datasets' tab is selected. Below the navigation bar, there are two main sections: 'BBC Datasets' and 'Dataset: BBCSport'. Each section contains a brief description, a list of bullet points, and two download links.

BBC Datasets

Two news article datasets, originating from [BBC News](#), provided for use as benchmarks for machine learning research. These datasets are made available for non-commercial and research purposes only, and all data is provided in pre-processed matrix format. If you make use of these datasets please consider citing the publication:

D. Greene and P. Cunningham. "Practical Solutions to the Problem of Diagonal Dominance in Kernel Document Clustering", Proc. ICML 2006. [\[PDF\]](#) [\[BibTeX\]](#).

Dataset: BBC

All rights, including copyright, in the content of the original articles are owned by the BBC.

- Consists of 2225 documents from the [BBC news](#) website corresponding to stories in five topical areas from 2004-2005.
- Class Labels: 5 (business, entertainment, politics, sport, tech)

>> Download pre-processed dataset
>> Download raw text files

Dataset: BBCSport

All rights, including copyright, in the content of the original articles are owned by the BBC.

- Consists of 737 documents from the [BBC Sport](#) website corresponding to sports news articles in five topical areas from 2004-2005.
- Class Labels: 5 (athletics, cricket, football, rugby, tennis)

>> Download pre-processed dataset
>> Download raw text files

Now, let's build a topic classifier.

Modeling

Start by importing the dataset and processing the text:

```
import cc.mallet.types.*;
import cc.mallet.pipe.*;
import cc.mallet.pipe.iterator.*;
import cc.mallet.topics.*;

import java.util.*;
import java.util.regex.*;
import java.io.*;
```

```
public class TopicModeling {  
  
    public static void main(String[] args) throws Exception {  
  
        String dataFolderPath = args[0];  
        String stopListFilePath = args[1];
```

Create a default pipeline as previously described:

```
ArrayList<Pipe> pipeList = new ArrayList<Pipe>();  
pipeList.add(new Input2CharSequence("UTF-8"));  
Pattern tokenPattern = Pattern.compile("[\\p{L}\\p{N}_]+");  
pipeList.add(new CharSequence2TokenSequence(tokenPattern));  
pipeList.add(new TokenSequenceLowercase());  
pipeList.add(new TokenSequenceRemoveStopwords(new  
    File(stopListFilePath), "utf-8", false, false, false));  
pipeList.add(new TokenSequence2FeatureSequence());  
pipeList.add(new Target2Label());  
SerialPipes pipeline = new SerialPipes(pipeList);
```

Next, initialize folderIterator:

```
FileIterator folderIterator = new FileIterator(  
    new File[] {new File(dataFolderPath)},  
    new TxtFilter(),  
    FileIterator.LAST_DIRECTORY);
```

Construct a new instance list with the pipeline that we want to use to process the text:

```
InstanceList instances = new InstanceList(pipeline);
```

Finally, process each instance provided by the iterator:

```
instances.addThruPipe(folderIterator);
```

Now let's create a model with five topics using the `cc.mallet.topics.ParallelTopicModel`. `ParallelTopicModel` class that implements a simple threaded **Latent Dirichlet Allocation (LDA)** model. LDA is a common method for topic modeling that uses Dirichlet distribution to estimate the probability that a selected topic generates a particular document. We will not dive deep into the details in this chapter; the reader is referred to the original paper by D. Blei et al. (2003). Note that there is another classification algorithm in machine learning with the same acronym that refers to **Linear Discriminant Analysis (LDA)**. Beside the common acronym, it has nothing in common with the LDA model.

The class is instantiated with parameters alpha and beta, which can be broadly interpreted, as follows:

- High alpha value means that each document is likely to contain a mixture of most of the topics, and not any single topic specifically. A low alpha value puts less of such constraints on documents, and this means that it is more likely that a document may contain mixture of just a few, or even only one, of the topics.
- A high beta value means that each topic is likely to contain a mixture of most of the words, and not any word specifically; while a low value means that a topic may contain a mixture of just a few of the words.

In our case, we initially keep both parameters low ($\alpha_t = 0.01$, $\beta_w = 0.01$) as we assume topics in our dataset are not mixed much and there are many words for each of the topics:

```
int numTopics = 5;
ParallelTopicModel model =
    new ParallelTopicModel(numTopics, 0.01, 0.01);
```

Next, add instances to the model, and as we are using parallel implementation, specify the number of threads that will run in parallel, as follows:

```
model.addInstances(instances);
model.setNumThreads(4);
```

Run the model for a selected number of iterations. Each iteration is used for better estimation of internal LDA parameters. For testing, we can use a small number of iterations, for example, 50; while in real applications, use 1000 or 2000 iterations. Finally, call the void `estimate()` method that will actually build an LDA model:

```
model.setNumIterations(1000);
model.estimate();
```

The model outputs the following result:

```
0 0,06654 game england year time win world 6
1 0,0863 year 1 company market growth economy firm
2 0,05981 people technology mobile mr games users music
3 0,05744 film year music show awards award won
4 0,11395 mr government people labour election party blair
```

```
[beta: 0,11328]
<1000> LL/token: -8,63377
```

```
Total time: 45 seconds
```

LL/token indicates the model's log-likelihood, divided by the total number of tokens, indicating how likely the data is given the model. Increasing values mean the model is improving.

The output also shows the top words describing each topic. The words correspond to initial topics really well:

- **Topic 0:** game, England, year, time, win, world, 6 → sport
- **Topic 1:** year, 1, company, market, growth, economy, firm → finance
- **Topic 2:** people, technology, mobile, mr, games, users, music → tech
- **Topic 3:** film, year, music, show, awards, award, won → entertainment
- **Topic 4:** mr, government, people, labor, election, party, blair → politics

There are still some words that don't make much sense, for instance, mr, 1, and 6. We could include them in the stop word list. Also, some words appear twice, for example, award and awards. This happened because we didn't apply any stemmer or lemmatization pipe.

In the next section, we'll take a look to check whether the model is of any good.

Evaluating a model

As statistical topic modeling has unsupervised nature, it makes model selection difficult. For some applications, there may be some extrinsic tasks at hand, such as information retrieval or document classification, for which performance can be evaluated. However, in general, we want to estimate the model's ability to generalize topics regardless of the task.

Wallach et al. (2009) introduced an approach that measures the quality of a model by computing the log probability of held-out documents under the model. Likelihood of unseen documents can be used to compare models – higher likelihood implies a better model.

First, let's split the documents into training and testing set (that is, held-out documents), where we use 90% for training and 10% for testing:

```
// Split dataset
InstanceList[] instanceSplit= instances.split(new Randoms(), new
    double[] {0.9, 0.1, 0.0});
```

Now, let's rebuild our model using only 90% of our documents:

```
// Use the first 90% for training
model.addInstances(instanceSplit[0]);
```

```
model.setNumThreads(4);
model.setNumIterations(50);
model.estimate();
```

Next, initialize an estimator that implements Wallach's log probability of held-out documents, MarginalProbEstimator:

```
// Get estimator
MarginalProbEstimator estimator = model.getProbEstimator();
```

An intuitive description of LDA is summarized by Annalyn Ng in her blog:

<https://annalyzin.wordpress.com/2015/06/21/laymans-explanation-of-topic-modeling-with-lda-2/>



To get deeper insight into the LDA algorithm, its components, and its working, take a look at the original paper LDA by David Blei et al. (2003) at <http://jmlr.csail.mit.edu/papers/v3/blei03a.html> or take a look at the summarized presentation by D. Santhanam of Brown University at http://www.cs.brown.edu/courses/csci2950-p/spring2010/lectures/2010-03-03_santhanam.pdf.

The class implements many estimators that require quite deep theoretical knowledge of how the LDA method works. We'll pick the left-to-right evaluator, which is appropriate for a wide range of applications, including text mining, speech recognition, and others. The left-to-right evaluator is implemented as the double evaluateLeftToRight method, accepting the following components:

- Instances heldOutDocuments: This tests the instances
- int numParticles: This algorithm parameter indicates the number of left-to-right tokens, where default value is 10
- boolean useResampling: This states whether to resample topics in left-to-right evaluation; resampling is more accurate, but leads to quadratic scaling in the length of documents
- PrintStream docProbabilityStream: This is the file or stdout in which we write the inferred log probabilities per document

Let's run the estimator, as follows:

```
double loglike = estimator.evaluateLeftToRight(
    instanceSplit[1], 10, false, null);
System.out.println("Total log likelihood: "+loglike);
```

In our particular case, the estimator outputs the following log likelihood, which makes sense when it is compared to other models that are either constructed with different parameters, pipelines, or data – the higher the log likelihood, the better the model is:

```
Total time: 3 seconds
Topic Evaluator: 5 topics, 3 topic bits, 111 topic mask
Total log likelihood: -360849.4240795393
Total log likelihood
```

Now let's take a look at how to make use of this model.

Reusing a model

As we are usually not building models on the fly, it often makes sense to train a model once and use it repeatedly to classify new data.

Note that if you'd like to classify new documents, they need go through the same pipeline as other documents – the pipe needs to be the same for both training and classification. During training, the pipe's data alphabet is updated with each training instance. If you create a new pipe with the same steps, you don't produce the same pipeline as its data alphabet is empty. Therefore, to use the model on new data, save/load the pipe along with the model and use this pipe to add new instances.

Saving a model

Mallet supports a standard method for saving and restoring objects based on serialization. We simply create a new instance of `ObjectOutputStream` class and write the object into a file as follows:

```
String modelPath = "myTopicModel";

//Save model
ObjectOutputStream oos = new ObjectOutputStream(
new FileOutputStream (new File(modelPath+".model")));
oos.writeObject(model);
oos.close();

//Save pipeline
oos = new ObjectOutputStream(
new FileOutputStream (new File(modelPath+".pipeline")));
oos.writeObject(pipeline);
oos.close();
```

Restoring a model

Restoring a model saved through serialization is simply an inverse operation using the `ObjectInputStream` class:

```
String modelPath = "myTopicModel";

//Load model
ObjectInputStream ois = new ObjectInputStream(
    new FileInputStream (new File(modelPath+".model")));
ParallelTopicModel model = (ParallelTopicModel) ois.readObject();
ois.close();

// Load pipeline
ois = new ObjectInputStream(
    new FileInputStream (new File(modelPath+".pipeline")));
SerialPipes pipeline = (SerialPipes) ois.readObject();
ois.close();
```

We discussed how to build an LDA model to automatically classify documents into topics. In the next example, we'll look into another text mining problem—text classification.

E-mail spam detection

Spam or electronic spam refers to unsolicited messages, typically carrying advertising content, infected attachments, links to phishing or malware sites, and so on. While the most widely recognized form of spam is e-mail spam, spam abuses appear in other media as well: website comments, instant messaging, Internet forums, blogs, online ads, and so on.

In this chapter, we will discuss how to build naive Bayesian spam filtering, using bag-of-words representation to identify spam e-mails. The naive Bayes spam filtering is one of the basic techniques that was implemented in the first commercial spam filters; for instance, **Mozilla Thunderbird** mail client uses native implementation of such filtering. While the example in this chapter will use e-mail spam, the underlying methodology can be applied to other type of text-based spam as well.

E-mail spam dataset

Androulidakis et al. (2000) collected one of the first e-mail spam datasets to benchmark spam-filtering algorithms. They studied how the naive Bayes classifier can be used to detect spam, if additional pipes such as stop list, stemmer, and lemmatization contribute to better performance. The dataset was reorganized by Andrew Ng in OpenClassroom's machine learning class, available for download at <http://openclassroom.stanford.edu/MainFolder/DocumentPage.php?course=MachineLearning&doc=exercises/ex6/ex6.html>.

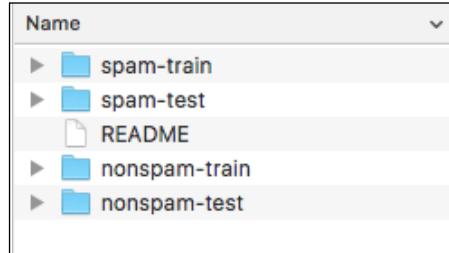
Select and download the second option, `ex6DataEmails.zip`, as shown in the following image:

The screenshot shows the OpenClassroom platform interface. At the top, it says "OpenClassroom". Below that, there's a sidebar with a book icon labeled "Machine Learning" and "Andrew Ng". The main content area has a title "Exercise 6: Naive Bayes". Below the title, there's a paragraph about the exercise, mentioning the Ling-Spam Dataset and its 960 real email messages. It also describes two ways to complete the exercise: using a pre-generated Matlab/Octave pack or generating features from emails and implementing Naive Bayes. A "RESOURCES" sidebar on the right lists Syllabus, FAQ, and Credits/Acknowledgments.

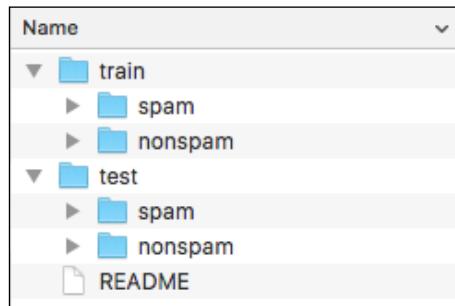
The ZIP contains four folders (Ng, 2015):

- The `nonspam-train` and `spam-train` folders contain the pre-processed e-mails that you will use for training. They have 350 e-mails each.
- The `nonspam-test` and `spam-test` folders constitute the test set, containing 130 spam and 130 nonspam e-mails. These are the documents that you will make predictions on. Notice that even though separate folders tell you the correct labeling, you should make your predictions on all the test documents without this knowledge. After you make your predictions, you can use the correct labeling to check whether your classifications were correct.

To leverage Mallet's folder iterator, let's reorganize the folder structure as follows. Create two folders, `train` and `test`, and put the `spam/nospam` folders under the corresponding folders. The initial folder structure is as shown in the following image:



The final folder structure will be as shown in the following image:



The next step is to transform e-mail messages to feature vectors.

Feature generation

Create a default pipeline as described previously:

```
ArrayList<Pipe> pipeList = new ArrayList<Pipe>();
pipeList.add(new Input2CharSequence("UTF-8"));
Pattern tokenPattern = Pattern.compile("[\p{L}\p{N}_]+");
pipeList.add(new CharSequence2TokenSequence(tokenPattern));
pipeList.add(new TokenSequenceLowercase());
pipeList.add(new TokenSequenceRemoveStopwords(new
    File(stopListFilePath), "utf-8", false, false, false));
pipeList.add(new TokenSequence2FeatureSequence());
pipeList.add(new FeatureSequence2FeatureVector());
pipeList.add(new Target2Label());
SerialPipes pipeline = new SerialPipes(pipeList);
```

Note that we added an additional `FeatureSequence2FeatureVector` pipe that transforms a feature sequence into a feature vector. When we have data in a feature vector, we can use any classification algorithm as we saw in the previous chapters. We'll continue our example in Mallet to demonstrate how to build a classification model.

Next, initialize a folder iterator to load our examples in the `train` folder comprising e-mail examples in the `spam` and `nonspam` subfolders, which will be used as example labels:

```
FileIterator folderIterator = new FileIterator()  
    new File[] {new File(dataFolderPath)},  
    new TxtFilter(),  
    FileIterator.LAST_DIRECTORY);
```

Construct a new instance list with the pipeline that we want to use to process the text:

```
InstanceList instances = new InstanceList(pipeline);
```

Finally, process each instance provided by the iterator:

```
instances.addThruPipe(folderIterator);
```

We have now loaded the data and transformed it into feature vectors. Let's train our model on the training set and predict the `spam/nonspam` classification on the test set.

Training and testing

Mallet implements a set of classifiers in the `cc.mallet.classify` package, including decision trees, naive Bayes, AdaBoost, bagging, boosting, and many others. We'll start with a basic classifier, that is, a naive Bayes classifier. A classifier is initialized by the `ClassifierTrainer` class, which returns a classifier when we invoke its `train(Instances)` method:

```
ClassifierTrainer classifierTrainer = new NaiveBayesTrainer();  
Classifier classifier = classifierTrainer.train(instances);
```

Now let's see how this classifier works and evaluate its performance on a separate dataset.

Model performance

To evaluate the classifier on a separate dataset, let's start by importing the e-mails located in our `test` folder:

```
InstanceList testInstances = new  
    InstanceList(classifier.getInstancePipe());  
folderIterator = new FileIterator(  
    new File[] {new File(testFolderPath)},  
    new TxtFilter(),  
    FileIterator.LAST_DIRECTORY);
```

We will pass the data through the same pipeline that we initialized during training:

```
testInstances.addThruPipe(folderIterator);
```

To evaluate classifier performance, we'll use the `cc.mallet.classify.Trial` class, which is initialized with a classifier and set of test instances:

```
Trial trial = new Trial(classifier, testInstances);
```

The evaluation is performed immediately at initialization. We can then simply take out the measures that we care about. In our example, we'd like to check the precision and recall on classifying spam e-mail messages, or F-measure, which returns a harmonic mean of both values, as follows:

```
System.out.println(  
    "F1 for class 'spam': " + trial.getF1("spam"));  
System.out.println(  
    "Precision: " + trial.getPrecision(1));  
System.out.println(  
    "Recall: " + trial.getRecall(1));
```

The evaluation object outputs the following results:

```
F1 for class 'spam': 0.9731800766283524  
Precision: 0.9694656488549618  
Recall: 0.9769230769230769
```

The results show that the model correctly discovers 97.69% of spam messages (recall), and when it marks an e-mail as spam, it is correct in 96.94% cases. In other words, it misses approximately 2 per 100 spam messages and marks 3 per 100 valid messages as spam. Not really perfect, but it is more than a good start!

Summary

In this chapter, we discussed how text mining is different from traditional attribute-based learning, requiring a lot of pre-processing steps in order to transform written natural language into feature vectors. Further, we discussed how to leverage Mallet, a Java-based library for natural language processing by applying it to two real life problems. First, we modeled topics in news corpus using the LDA model to build a model that is able to assign a topic to new document. We also discussed how to build a naive Bayesian spam-filtering classifier using the bag-of-words representation.

This chapter concludes the technical demonstrations of how to apply various libraries to solve machine learning tasks. As we were not able to cover more interesting applications and give further details at many points, the next chapter gives some further pointers on how to continue learning and dive deeper into particular topics.

11

What is Next?

This chapter brings us to the end of our journey of reviewing machine learning Java libraries and discussing how to leverage them to solve real-life problems. However, this should not be the end of your journey by all means. This chapter will give you some practical advice on how to start deploying your models in the real world, what are the catches, and where to go to deepen your knowledge. It also gives you further pointers about where to find additional resources, materials, venues, and technologies to dive deeper into machine learning.

This chapter will cover the following topics:

- Important aspects of machine learning in real life
- Standards and markup languages
- Machine learning in the cloud
- Web resources and competitions

Machine learning in real life

Papers, conference presentations, and talks often don't discuss how the models were actually deployed and maintained in production environment. In this section, we'll look into some aspects that should be taken into consideration.

Noisy data

In practice, data typically contains errors and imperfections due to various reasons such as measurement errors, human mistakes, errors of expert judgment in classifying training examples, and so on. We refer to all of these as noise. Noise can also come from the treatment of missing values when an example with unknown attribute value is replaced by a set of weighted examples corresponding to the probability distribution of the missing value. The typical consequences of noise in learning data are low prediction accuracy of learned model in new data and complex models that are hard to interpret and to understand by the user.

Class unbalance

Class unbalance is a problem we come across in *Chapter 7, Fraud and Anomaly Detection*, where the goal was to detect fraudulent insurance claims. The challenge is that a very large part of the dataset, usually more than 90%, describes normal activities and only a small fraction of the dataset contains fraudulent examples. In such a case, if the model always predicts normal, then it is correct 90% of the time. This problem is extremely common in practice and can be observed in various applications, including fraud detection, anomaly detection, medical diagnosis, oil spillage detection, facial recognition, and so on.

Now knowing what the class unbalance problem is and why is it a problem, let's take a look at how to deal with this problem. The first approach is to focus on measures other than classification accuracy, such as recall, precision, and f-measure. Such measures focus on how accurate a model is at predicting minority class (recall) and what is the share of false alarms (precision). The other approach is based on resampling, where the main idea is to reduce the number of overrepresented examples in such way that the new set contains a balanced ratio of both the classes.

Feature selection is hard

Feature selection is arguably the most challenging part of modeling that requires domain knowledge and good insights into the problem at hand. Nevertheless, properties of well-behaved features are as follows:

- **Reusability:** Features should be available for reuse in different models, applications, and teams
- **Transformability:** You should be able to transform a feature with an operation, for example, `log()`, `max()`, or combine multiple features together with a custom calculation

- **Reliability:** Features should be easy to monitor and appropriate unit tests should exist to minimize bugs/issues
- **Interpretability:** In order to perform any of the previous actions, you need to be able to understand the meaning of features and interpret their values

The better you are able to capture the features, the more accurate your results will be.

Model chaining

Some models might produce an output, which is used as the feature in another model. Moreover, we can use multiple models – ensembles – turning any model into a feature. This is a great way to get better results, but this can lead to problems too. Care must be taken that the output of your model is ready to accept dependencies. Also, try to avoid feedback loops, as they can create dependencies and bottlenecks in pipeline.

Importance of evaluation

Another important aspect is model evaluation. Unless you apply your models to actual new data and measure a business objective, you're not doing predictive analytics. Evaluation techniques, such as cross-validation and separated train/test set, simply split your test data, which can give only you an estimate of how your model will perform. Life often doesn't hand you a train dataset with all the cases defined, so there is a lot of creativity involved in defining these two sets in a real-world dataset.

At the end of the day, we want to improve a business objective, such as improve ad conversion rate, get more clicks on recommended items, and so on. To measure the improvement, execute A/B tests, measure differences in metrics across statistically identical populations that each experience a different algorithm. Decisions on the product are always data-driven.

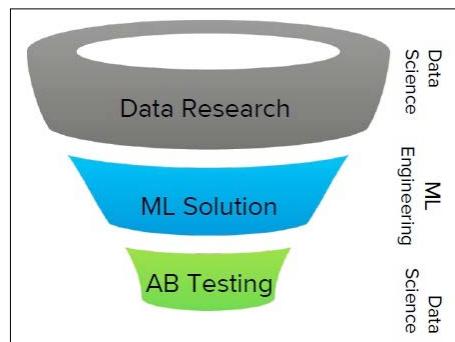
A/B testing is a method for a randomized experiment with two variants: A, which corresponds to the original version, controlling the experiment; and B, which corresponds to a variation. The method can be used to determine whether the variation outperforms the original version. It can be used to test everything from website changes to sales e-mails to search ads.

Udacity offers a free course, covering design and analysis of A/B tests at <https://www.udacity.com/course/ab-testing--ud257>.

Getting models into production

The path from building an accurate model in a lab to deploying it in a product involves collaboration of data science and engineering, as shown in the following three steps and diagram:

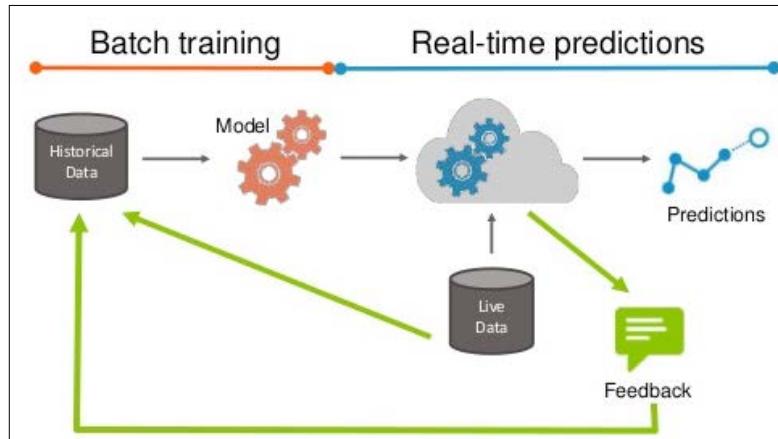
1. **Data research and hypothesis building** involves modeling the problem and executing initial evaluation.
2. **Solution building and implementation** is where your model finds its way into the product flow by rewriting it into more efficient, stable, and scalable code.
3. **Online evaluation** is the last stage where the model is evaluated with live data using A/B testing on business objectives.



Model maintenance

Another aspect that we need to address is how the model will be maintained. Is this a model that will not change over time? Is it modeling a dynamic phenomenon requiring the model to adjust its prediction over time?

The model is usually built in an of offline batch training and then used on live data to serve predictions as shown in the following figure. If we are able to receive feedback on model predictions; for instance, whether the stock went up as model predicted, whether the candidate responded to campaign, and so on, the feedback should be used to improve the initial model.



The feedback could be really useful to improve the initial model, but make sure to pay attention to the data you are sampling. For instance, if you have a model that predicts who will respond to a campaign, you will initially use a set of randomly contacted clients with specific responded/not responded distribution and feature properties. The model will focus only on a subset of clients that will most likely respond and your feedback will return you a subset of clients that responded. By including this data, the model is more accurate in a specific subgroup, but might completely miss some other group. We call this problem exploration versus exploitation. Some approaches to address this problem can be found in Osugi et al (2005) and Bondu et al (2010).

Standards and markup languages

As predictive models become more pervasive, the need for sharing the models and completing the modeling process leads to formalization of development process and interchangeable formats. In this section, we'll review two de facto standards, one covering data science processes and the other specifying an interchangeable format for sharing models between applications.

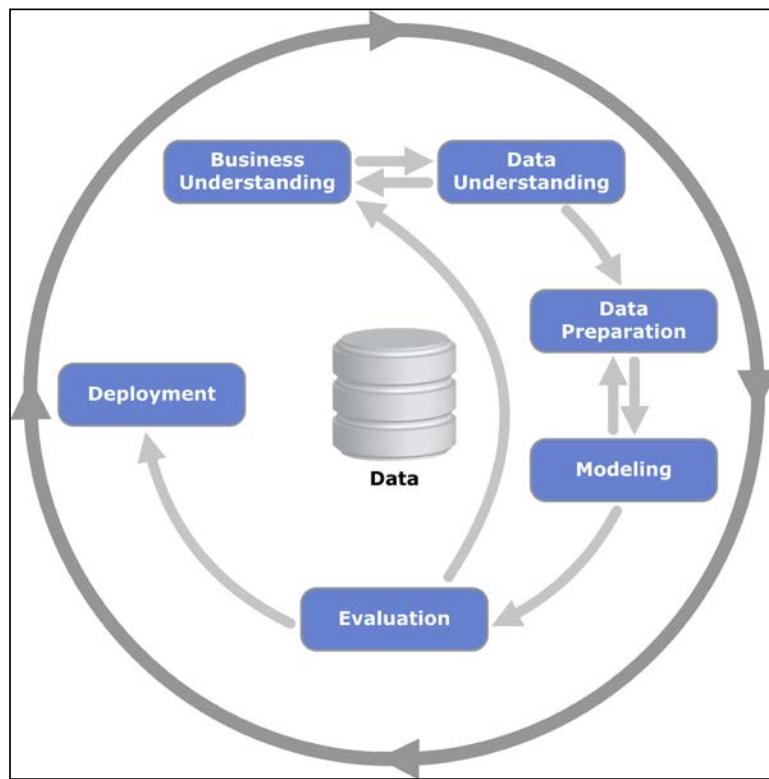
CRISP-DM

Cross Industry Standard Process for Data Mining (CRISP-DM) describing a data mining process commonly used by data scientists in industry. CRISP-DM breaks the data mining science process into the following six major phases:

- **Business understanding**
- **Data understanding**

- **Data preparation**
- **Modeling**
- **Evaluation**
- **Deployment**

In the following diagram, the arrows indicate the process flow, which can move back and forth through the phases. Also, the process doesn't stop with model deployment. The outer arrow indicates the cyclic nature of data science. Lessons learned during the process can trigger new questions and repeat the process while improving previous results:



SEMMA methodology

Another methodology is **Sample, Explore, Modify, Model, and Assess (SEMMA)**. SEMMA describes the main modeling tasks in data science, while leaving aside business aspects such as data understanding and deployment. SEMMA was developed by SAS institute, which is one of the largest vendors of statistical software, aiming to help the users of their software to carry out core tasks of data mining.

Predictive Model Markup Language

Predictive Model Markup Language (PMML) is an XML-based interchange format that allows machine learning models to be easily shared between applications and systems. Supported models include logistic regression, neural networks, decision trees, naïve Bayes, regression models, and many others. A typical PMML file consists of the following sections:

- Header containing general information
- Data dictionary, describing data types
- Data transformations, specifying steps for normalization, discretization, aggregations, or custom functions
- Model definition, including parameters
- Mining schema listing attributes used by the model
- Targets allowing post-processing of the predicted results
- Output listing fields to be outputted and other post-processing steps

The generated PMML files can be imported to any PMML-consuming application, such as Zementis **Adaptive Decision and Predictive Analytics (ADAPA)** and **Universal PMML Plug-in (UPPI)** scoring engines; Weka, which has built-in support for regression, general regression, neural network, **TreeModel**, **RuleSetModel**, and **Support Vector Machine (SVM)** model; Spark, which can export k-means clustering, linear regression, ridge regression, lasso model, binary logistic model, and SVM; and cascading, which can transform PMML files into an application on Apache Hadoop.

The next generation of PMML is an emerging format called **Portable Format for Analytics (PFA)**, providing a common interface to deploy the complete workflows across environments.

Machine learning in the cloud

Setting up a complete machine learning stack that is able to scale with the increasing amount of data could be challenging. Recent wave of **Software as a Service (SaaS)** and **Infrastructure as a Service (IaaS)** paradigm was spilled over to machine learning domain as well. The trend today is to move the actual data preprocessing, modeling, and prediction to cloud environments and focus on modeling task only.

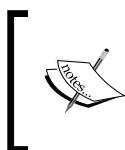
In this section, we'll review some of the promising services offering algorithms, predictive models already train in specific domain, and environments empowering collaborative workflows in data science teams.

Machine learning as a service

The first category is algorithms as a service, where you are provided with an API or even graphical user interface to connect pre-programmed components of data science pipeline together:

- **Google Prediction API** was one of the first companies that introduced prediction services through its web API. The service is integrated with **Google Cloud Storage** serving as data storage. The user can build a model and call an API to get predictions.
- **BigML** implements a user-friendly graphical interface, supports many storage providers (for instance, Amazon S3) and offers a wide variety of data processing tools, algorithms, and powerful visualizations.
- **Microsoft Azure Machine Learning** provides a large library of machine learning algorithms and data processing functions, as well as graphical user interface, to connect these components to an application. Additionally, it offers a fully-managed service that you can use to deploy your predictive models as ready-to-consume web services.
- **Amazon Machine Learning** entered the market quite late. It's main strength is seamless integration with other Amazon services, while the number of algorithms and user interface needs further improvements.
- **IBM Watson Analytics** focuses on providing models that are already hand-crafted to a particular domain such as speech recognition, machine translations, and anomaly detection. It targets a wide range of industries by solving specific use cases.
- **Prediction.IO** is a self-hosted open source platform, providing the full stack from data storage to modeling to serving the predictions. Prediction.IO can talk to Apache Spark to leverage its learning algorithms. In addition, it is shipped with a wide variety of models targeting specific domains, for instance, recommender system, churn prediction, and others.

Predictive API is an emerging new field, so these are just some of the well-known examples; **KDNuggets** compiled a list of 50 machine learning APIs at <http://www.kdnuggets.com/2015/12/machine-learning-data-science-apis.html>.



To learn more about it, you can visit PAPI, the International Conference on Predictive APIs and Apps at <http://www.papi.io> or take a look at a book by *Louis Dorard, Bootstrapping Machine Learning* (L. Dorard, 2014).

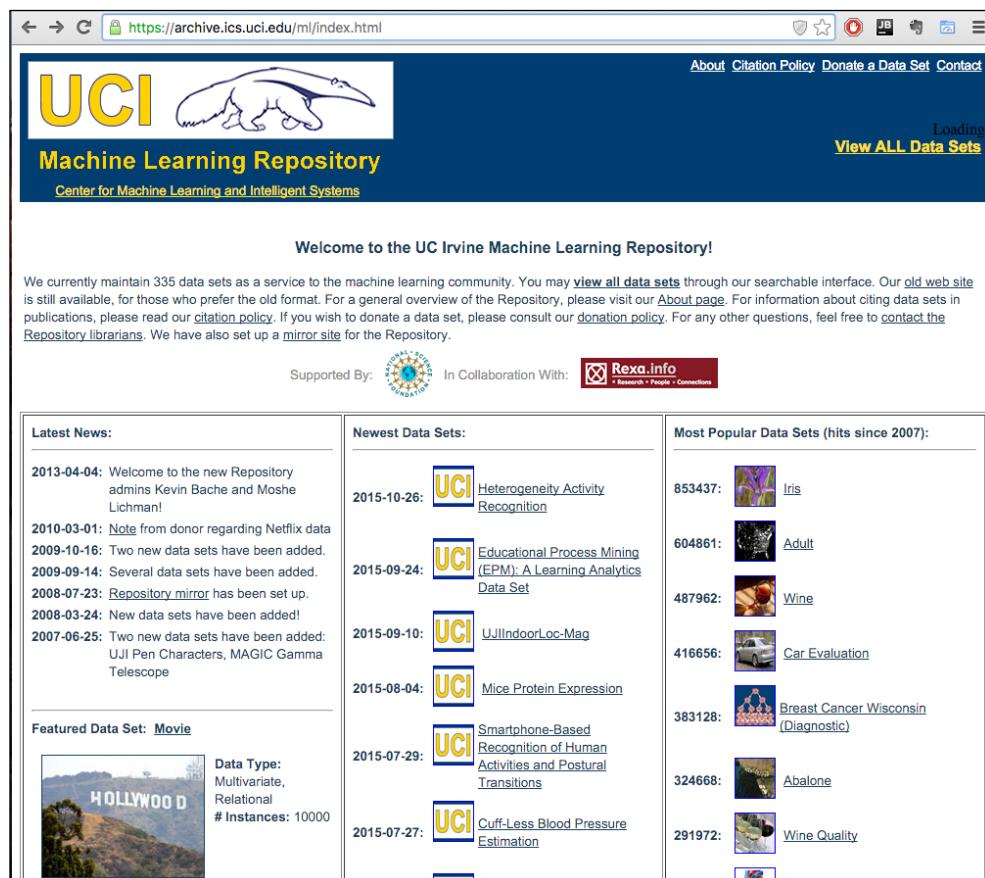
Web resources and competitions

In this section, we'll review where to find additional resources for learning, discussing, presenting, or sharpening our data science skills.

Datasets

One of the most well-known repositories of machine learning datasets is hosted by the University of California, Irvine. The UCI repository contains over 300 datasets covering a wide variety of challenges, including poker, movies, wine quality, activity recognition, stocks, taxi service trajectories, advertisements, and many others. Each dataset is usually equipped with a research paper where the dataset was used, which can give you a hint on how to start and what is the prediction baseline.

The UCI machine learning repository can be accessed at <https://archive.ics.uci.edu/ml/index.html>, as follows:



The screenshot shows the homepage of the UC Irvine Machine Learning Repository. At the top, there's a navigation bar with links for About, Citation Policy, Donate a Data Set, and Contact. Below the navigation is a banner with the text "View ALL Data Sets". The main content area features a "Welcome to the UC Irvine Machine Learning Repository!" message. It includes a note about maintaining 335 data sets and links to various resources like old web sites, citation policy, and contact information. Below this, there are three columns: "Latest News", "Newest Data Sets", and "Most Popular Data Sets (hits since 2007)". The "Latest News" column lists recent updates. The "Newest Data Sets" column lists datasets added in 2015. The "Most Popular Data Sets" column lists datasets based on hits since 2007, including Iris, Adult, Wine, Car Evaluation, Breast Cancer Wisconsin (Diagnostic), Abalone, and Wine Quality. A sidebar on the left shows a "Featured Data Set: Movie" with a thumbnail of the Hollywood sign.

Latest News:		
<p>2013-04-04: Welcome to the new Repository admins Kevin Bache and Moshe Lichman!</p> <p>2010-03-01: Note from donor regarding Netflix data</p> <p>2009-10-16: Two new data sets have been added.</p> <p>2009-09-14: Several data sets have been added.</p> <p>2008-07-23: Repository mirror has been set up.</p> <p>2008-03-24: New data sets have been added!</p> <p>2007-06-25: Two new data sets have been added: UJI Pen Characters, MAGIC Gamma Telescope</p>		
Featured Data Set: Movie		
 <p>Data Type: Multivariate, Relational # Instances: 10000</p>		

Newest Data Sets:		
2015-10-26:	 Heterogeneity Activity Recognition	
2015-09-24:	 Educational Process Mining (EPM): A Learning Analytics Data Set	
2015-09-10:	 UJIIndoorLoc-Mag	
2015-08-04:	 Mice Protein Expression	
2015-07-29:	 Smartphone-Based Recognition of Human Activities and Postural Transitions	
2015-07-27:	 Cuff-Less Blood Pressure Estimation	

Most Popular Data Sets (hits since 2007):		
853437:		Iris
604861:		Adult
487962:		Wine
416656:		Car Evaluation
383128:		Breast Cancer Wisconsin (Diagnostic)
324668:		Abalone
291972:		Wine Quality

Another well-maintained collection by Xiaming Chen is hosted on GitHub:

<https://github.com/caesar0301/awesome-public-datasets>

The Awesome Public Datasets repository maintains links to more than 400 data sources from a variety of domains, ranging from agriculture, biology, economics, psychology, museums, and transportation. Datasets, specifically targeting machine learning, are collected under the image processing, machine learning, and data challenges sections.

Online courses

Learning how to become a data scientist has became much more accessible due to the availability of online courses. The following is a list of free resources to learn different skills online:

- Online courses for learning Java:
 - **Udemy: Learn Java Programming From Scratch** at <https://www.udemy.com/learn-java-programming-from-scratch>
 - **Udemy: Java Tutorial for Complete Beginners** at <https://www.udemy.com/java-tutorial>
 - **LearnJAvAOnline.org: Interactive Java tutorial** at <http://www.learnjavaonline.org/>
- Online courses to learn more about machine learning:
 - **Coursera: Machine Learning (Stanford) by Andrew Ng:** This teaches you the math behind many the machine learning algorithms, explains how they work, and explores why they make sense at <https://www.coursera.org/learn/machine-learning>.
 - **Statistics 110 (Harvard) by Joe Blitzstein:** This course lets you discover the probability of related terms that you will hear many times in your data science journey. Lectures are available on YouTube at <http://projects.iq.harvard.edu/stat110/youtube>.
 - **Data Science CS109 (Harvard) by John A. Paulson:** This is a hands-on course where you'll learn about Python libraries for data science, as well as how to handle machine-learning algorithms at <http://cs109.github.io/2015/>.

Competitions

The best way to sharpen your knowledge is to work on real problems; and if you want to build a proven portfolio of your projects, machine learning competitions are a viable place to start:

- **Kaggle:** This is the number one competition platform, hosting a wide variety of challenges with large prizes, strong data science community, and lots of helpful resources. You can check it out at <https://www.kaggle.com/>.
- **CrowdANALYTIX:** This is a crowdsourced data analytics service that is focused on the life sciences and financial services industries at <https://www.crowdanalytix.com>.
- **DrivenData:** This hosts data science competitions for social good at <http://www.drivendata.org/>.

Websites and blogs

In addition to online courses and competitions, there are numerous websites and blogs publishing the latest developments in the data science community, their experience in attacking different problems, or just best practices. Some good starting points are as follows:

- **KDnuggets:** This is the de facto portal for data mining, analytics, big data, and data science, covering the latest news, stories, events, and other relevant issues at <http://www.kdnuggets.com/>.
- **Machine learning mastery:** This is an introductory-level blog with practical advice and pointers where to start. Check it out at <http://machinelearningmastery.com/>.
- **Data Science Central:** This consists of practical community articles on a variety of topics, algorithms, caches, and business cases at <http://www.datasciencentral.com/>.
- **Data Mining Research** by Sandro Saitta at <http://www.dataminingblog.com/>.
- **Data Mining: Text Mining, Visualization and Social Media** by Matthew Hurst, covering interesting text and web mining topics, frequently with applications to Bing and Microsoft at http://datamining.typepad.com/data_mining/.
- **Geeking with Greg** by Greg Linden, inventor of Amazon recommendation engine and Internet entrepreneur. You can check it out at <http://glinden.blogspot.si/>.
- **DSGuide:** This is a collection of over 150 data science blogs at <http://dsguide.biz/reader/sources>.

Venues and conferences

The following are a few top-tier academic conferences with the latest algorithms:

- **Knowledge Discovery in Databases (KDD)**
- **Computer Vision and Pattern Recognition (CVPR)**
- **Annual Conference on Neural Information Processing Systems (NIPS)**
- **International Conference on Machine Learning (ICML)**
- **IEEE International Conference on Data Mining (ICDM)**
- **International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)**
- **International Joint Conference on Artificial Intelligence (IJCAI)**

Some business conferences are as follows:

- O'Reilly Strata Conference
- The Strata + Hadoop World Conferences
- Predictive Analytics World
- MLconf

You can also check local meetup groups.

Summary

In this chapter, we concluded the book by discussing some aspects of model deployment, we also looked into standards for data science process and interchangeable predictive model format PMML. We also reviewed online courses, competitions, web resources, and conferences that could help you in your journey towards mastering the art of machine learning.

I hope this book inspired you to dive deeper into data science and has motivated you to get your hands dirty, experiment with various libraries and get a grasp of how different problems could be attacked. Remember, all the source code and additional resources are available at the supplementary website <http://www.machine-learning-in-java.com>.

References

The following are the references for all the citations throughout the book:

- Adomavicius, G. and Tuzhilin, A.. Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17(6), 734-749. 2005.
- Bengio, Y.. Learning Deep Architectures for AI. *Foundations and Trends in Machine Learning* 2(1), 1-127. 2009. Retrieved from <http://www.iro.umontreal.ca/~bengioy/papers/ftml.pdf>.
- Blei, D. M., Ng, A. Y., and Jordan, M. I.. Latent dirichlet allocation. *Journal of Machine Learning Research*. 3, 993–1022. 2003. Retrieved from: <http://www.jmlr.org/papers/volume3/blei03a/blei03a.pdf>.
- Bondu, A., Lemaire, V., Boulle, M.. Exploration vs. exploitation in active learning: A Bayesian approach. *The 2010 International Joint Conference on Neural Networks (IJCNN)*, Barcelona, Spain. 2010.
- Breunig, M. M., Kriegel, H.-P., Ng, R. T., Sander, J.. LOF: Identifying Density-based Local Outliers (PDF). *Proceedings from the 2000 ACM SIGMOD International Conference on Management of Data*, 29(2), 93–104. 2000
- Campbell, A. T. (n.d.). Lecture 21 - Activity Recognition. Retrieved from <http://www.cs.dartmouth.edu/~campbell/cs65/lecture22/lecture22.html>.
- Chandra, N.S. Unraveling the Customer Mind. 2012. Retrieved from <http://www.cognizant.com/InsightsWhitepapers/Unraveling-the-Customer-Mind.pdf>.

References

- Dror, G., Boulle □, M., Guyon, I., Lemaire, V., and Vogel, D.. The 2009 Knowledge Discovery in Data Competition (KDD Cup 2009) Volume 3, Challenges in Machine Learning, Massachusetts, US. Microtome Publishing. 2009.
- Gelman, A. and Nolan, D.. Teaching Statistics a bag of tricks. Cambridge, MA. Oxford University Press. 2002.
- Goshtasby, A. A. *Image Registration Principles, Tools and Methods*. London, Springer. 2012.
- Greene, D. and Cunningham, P.. Practical Solutions to the Problem of Diagonal Dominance in Kernel Document Clustering. Proceedings from the 23rd International Conference on Machine Learning, Pittsburgh, PA. 2006. Retrieved from http://www.autonlab.org/icml_documents/camera-ready/048_Practical_Solutions.pdf.
- Gupta, A.. *Learning Apache Mahout Classification*, Birmingham, UK. Packt Publishing. 2015.
- Gutierrez, N.. Demystifying Market Basket Analysis. 2006. Retrieved from <http://www.information-management.com/specialreports/20061031/1067598-1.html>.
- Hand, D., Manilla, H., and Smith, P.. *Principles of Data Mining*. USA. MIT Press. 2001. Retrieved from ftp://gamma.sbin.org/pub/doc/books/Principles_of_Data_Mining.pdf.
- Intel. What Happens in an Internet Minute?. 2013. Retrieved from <http://www.intel.com/content/www/us/en/communications/internet-minute-infographic.html>.
- Kaluža, B.. *Instant Weka How-To*. Birmingham. Packt Publishing. 2013.
- Karkera, K. R.. *Building Probabilistic Graphical Models with Python*. Birmingham, UK. Packt Publishing. 2014.
- KDD (n.d.). KDD Cup 2009: Customer relationship prediction. Retrieved from <http://www.kdd.org/kdd-cup/view/kdd-cup-2009>.
- Koller, D. and Friedman, N.. *Probabilistic Graphical Models Principles and Techniques*. Cambridge, Mass. MIT Press. 2012.
- Kurucz, M., Siklósi, D., Bíró, I., Csizsek, P., Fekete, Z., Iwatt, R., Kiss, T., and Szabó, A.. KDD Cup 2009 @ Budapest: feature partitioning and boosting 61. JMLR W&CP 7, 65–75. 2009.
- Laptev, N., Amizadeh, S., and Billawala, Y. (n.d.). A Benchmark Dataset for Time Series Anomaly Detection. Retrieved from <http://yahoolabs.tumblr.com/post/114590420346/a-benchmark-dataset-for-time-series-anomaly>.

- LKurgan, L.A. and Musilek, P.. A survey of Knowledge Discovery and Data Mining process models. *The Knowledge Engineering Review*, 21(1), 1–24. 2006.
- Lo, H.-Y., Chang, K.-W., Chen, S.-T., Chiang, T.-H., Ferng, C.-S., Hsieh, C.-J., Ko, Y.-K., Kuo, T.-T., Lai, H.-C., Lin, K.-Y., Wang, C.-H., Yu, H.-F., Lin, C.-J., Lin, H.-T., and Lin, S.-de. An Ensemble of Three Classifiers for KDD Cup 2009: Expanded Linear Model, Heterogeneous Boosting, and Selective Naive Bayes. *JMLR W&CP* 7, 57–64. 2009.
- Magalhães, P. Incorrect information provided by your website. 2010. Retrevied from <http://www.best.eu.org/aboutBEST/helpdeskRequest.jsp?req=f5wpxc8&auth=Paulo>.
- Mariscal, G., Marban, O., and Fernandez, C.. A survey of data mining and knowledge discovery process models and methodologies. *The Knowledge Engineering Review*, 25(2), 137–166. 2010.
- Mew, K. (2015). *Android 5 Programming by Example*. Birmingham, UK. Packt Publishing.
- Miller, H., Clarke, S., Lane, S., Lonie, A., Lazaridis, D., Petrovski, S., and Jones, O.. Predicting customer behavior: The University of Melbourne's KDD Cup report, *JMLR W&CP* 7, 45–55. 2009.
- Niculescu-Mizil, A., Perlich, C., Swirszcz, G., Sind- hwani, V., Liu, Y., Melville, P., Wang, D., Xiao, J., Hu, J., Singh, M., Shang, W. X., and Zhu, Y.. Winning the KDD Cup Orange Challenge with Ensemble Selection. *JMLR W&CP*, 7, 23–34. 2009. Retrieved from <http://jmlr.org/proceedings/papers/v7/niculescu09/niculescu09.pdf>.
- Oracle (n.d.). Anomaly Detection. Retrieved from http://docs.oracle.com/cd/B28359_01/datamine.111/b28129/anomalies.htm.
- Osugi, T., Deng, K., and Scott, S.. Balancing exploration and exploitation: a new algorithm for active machine learning. Fifth IEEE International Conference on Data Mining, Houston, Texas. 2005.
- Power, D. J. (ed.). DSS News. DSSResources.com, 3(23). 2002. Retreived from <http://www.dssresources.com/newsletters/66.php>.
- Quinlan, J. R. *C4.5: Programs for Machine Learning*. San Francisco, CA. Morgan Kaufmann Publishers. 1993.
- Rajak, A.. Association Rule Mining-Applications in Various Areas. 2008. Retrieved from https://www.researchgate.net/publication/238525379_Association_rule_mining-_Applications_in_various_areas.
- Ricci, F., Rokach, L., Shapira, B., and Kantor, P. B.. (eds.). *Recommender Systems Handbook*. New York, Springer. 2010.

References

- Rumsfeld, D. H. and Myers, G.. DoD News Briefing – Secretary Rumsfeld and Gen. Myers. 2002. Retrieved from <http://archive.defense.gov/transcripts/transcript.aspx?transcriptid=2636>.
- Stevens, S. S.. On the Theory of Scales of Measurement. *Science*, 103 (2684), 677–680. 1946.
- Sutton, R. S. and Barto, A. G.. *Reinforcement Learning An Introduction*. Cambridge, MA: MIT Press. 1998.
- Tiwary, C.. *Learning Apache Mahout*. Birmingham, UK. Packt Publishing. 2015.
- Tsai, J., Kaminka, G., Epstein, S., Zilka, A., Rika, I., Wang, X., Ogden, A., Brown, M., Fridman, N., Taylor, M., Bowring, E., Marsella, S., Tambe, M., and Sheel, A.. ESCAPES - Evacuation Simulation with Children, Authorities, Parents, Emotions, and Social comparison. Proceedings from 10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011) 2 (6), 457–464. 2011. Retrieved from http://www.aamas-conference.org/Proceedings/aamas2011/papers/D3_G57.pdf.
- Tsanas, A. and Xifara. Accurate quantitative estimation of energy performance of residential buildings using statistical machine learning tools. *Energy and Buildings*, 49, 560-567. 2012.
- Utts, J.. What Educated Citizens Should Know About Statistics and Probability. *The American Statistician*, 57 (2), 74-79. 2003.
- Wallach, H. M., Murray, I., Salakhutdinov, R., and Mimno, D.. Evaluation Methods for Topic Models. Proceedings from the 26th International conference on Machine Learning, Montreal, Canada. 2009. Retrieved from <http://mimno.infosci.cornell.edu/papers/wallach09evaluation.pdf>.
- Witten, I. H. and Frank, E.. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. USA. Morgan Kaufmann Publishers. 2000.
- Xie, J., Rojkova, V., Pal, S., and Coggeshall, S.. A Combination of Boosting and Bagging for KDD Cup 2009. *JMLR W&CP*, 7, 35–43. 2009.
- Zhang, H.. The Optimality of Naive Bayes. Proceedings from FLAIRS 2004 conference. 2004. Retrieved from <http://www.cs.unb.ca/~hzhang/publications/FLAIRS04ZhangH.pdf>.
- Ziegler, C-N., McNee, S. M., Konstan, J. A., and Lausen, G.. Improving Recommendation Lists Through Topic Diversification. Proceedings from the 14th International World Wide Web Conference (WWW '05), Chiba, Japan. 2005. Retrieved from <http://www2.informatik.uni-freiburg.de/~cziegler/papers/www-05-CR.pdf>.

Module 3

Neural Network Programming with Java, Second Edition

Create and unleash the power of neural networks by implementing professional Java code

1

Getting Started with Neural Networks

In this chapter, we will introduce neural networks and what they are designed for. This chapter serves as a foundation layer for the subsequent chapters, while presenting the basic concepts for neural networks. In this chapter, we will cover the following:

- Artificial neurons
- Weights and biases
- Activation functions
- Layers of neurons
- Neural network implementation in Java

Discovering neural networks

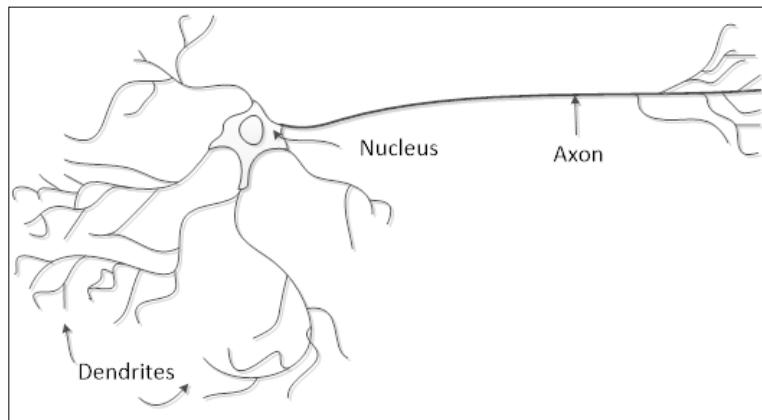
By hearing the term **neural networks** we intuitively create a snapshot of a brain in our minds, and indeed that's right, if we consider the brain to be a big and natural neural network. However, what about **artificial neural networks (ANNs)**? Well, here comes an opposite word to natural, and the first thing now that comes into our head is an image of an artificial brain or a robot, given the term *artificial*. In this case, we also deal with creating a structure similar to and inspired by the human brain; therefore, this can be called artificial intelligence.

Now the reader newly introduced to ANN may be thinking that this book teaches how to build intelligent systems, including an artificial brain, capable of emulating the human mind using Java codes, isn't it? The amazing answer is yes, but of course, we will not cover the creation of artificial thinking machines such as those from the Matrix trilogy movies; the reader will be provided a walkthrough on the process of designing artificial neural network solutions capable of abstracting knowledge in raw data, taking advantage of the entire Java programming language framework.

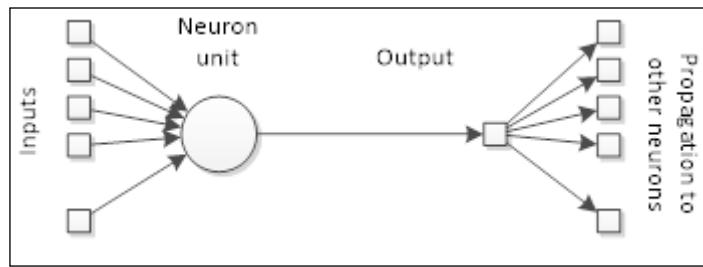
Why artificial neural networks?

We cannot begin talking about neural networks without understanding their origins, including the term as well. The terms neural networks (NN) and ANN are used as synonyms in this book, despite NNs being more general, covering the natural neural networks as well. So, what actually is an ANN? Let's explore a little of the history of this term.

In the 1940s, the neurophysiologist Warren McCulloch and the mathematician Walter Pitts designed the first mathematical implementation of an artificial neuron combining the neuroscience foundations with mathematical operations. At that time, the human brain was being studied largely to understand its hidden and mystery behaviors, yet within the field of neuroscience. The natural neuron structure was known to have a nucleus, dendrites receiving incoming signals from other neurons, and an axon activating a signal to other neurons, as shown in the following figure:



The novelty of McCulloch and Pitts was the math component included in the neuron model, supposing a neuron as a simple processor summing all incoming signals and activating a new signal to other neurons:



Furthermore, considering that the brain is composed of billions of neurons, each one interconnected with another tens of thousands, resulting in some trillions of connections, we are talking about a giant network structure. On the basis of this fact, McCulloch and Pitts designed a simple model for a single neuron, initially to simulate the human vision. The available calculators or computers at that time were very rare, but capable of dealing with mathematical operations quite well; on the other hand, even tasks today such as vision and sound recognition are not easily programmed without the use of special frameworks, as opposed to the mathematical operations and functions. Nevertheless, the human brain can perform sound and image recognition more efficiently than complex mathematical calculations, and this fact really intrigues scientists and researchers.

However, one known fact is that all complex activities that the human brain performs are based on learned knowledge, so as a solution to overcome the difficulty that conventional algorithmic approaches face in addressing these tasks easily solved by humans, an ANN is designed to have the capability to learn how to solve some task by itself, based on its stimuli (data):

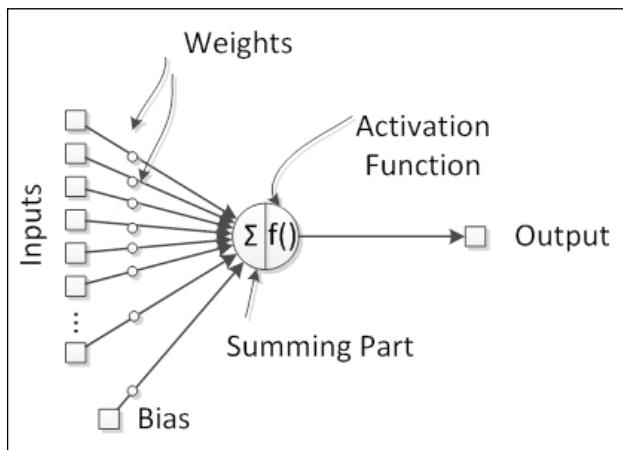
Tasks Quickly Solvable by Humans	Tasks Quickly Solvable by Computers
Classification of images	Complex calculation
Voice recognition	Grammatical error correction
Face identification	Signal processing
Forecast events on the basis of experience	Operating system management

How neural networks are arranged

By taking into account the human brain characteristics, it can be said that the ANN is a nature-inspired approach, and so is its structure. One neuron connects to a number of others that connect to another number of neurons, thus being a highly interconnected structure. Later in this book, it will be shown that this connectivity between neurons accounts for the capability of learning, since every connection is configurable according to the stimuli and the desired goal.

The very basic element – artificial neuron

Let's explore the most basic artificial neural element – the artificial neuron. Natural neurons have proven to be signal processors since they receive micro signals in the dendrites that can trigger a signal in the axon depending on their strength or magnitude. We can then think of a neuron as having a signal collector in the inputs and an activation unit in the output that can trigger a signal that will be forwarded to other neurons, as shown in the following figure:



[ In natural neurons, there is a threshold potential that when reached, fires the axon and propagates the signal to the other neurons. This firing behavior is emulated with activation functions, which has proven to be useful in representing nonlinear behaviors in the neurons.]

Giving life to neurons – activation function

This activation function is what fires the neuron's output, based on the sum of all incoming signals. Mathematically it adds nonlinearity to neural network processing, thereby providing the artificial neuron nonlinear behaviors, which will be very useful in emulating the nonlinear nature of natural neurons. An activation function is usually bounded between two values at the output, therefore being a nonlinear function, but in some special cases, it can be a linear function.

Although any function can be used as activation, let's concentrate on common used ones:

Function	Equation	Chart
Sigmoid	$f(x) = \frac{1}{1 + e^{-ax}}$	
Hyperbolic tangent	$f(x) = \frac{1 - e^{-ax}}{1 + e^{-ax}}$	
Hard limiting threshold	$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	

Function	Equation	Chart
Linear	$f(x) = ax$	

In these equations and charts the coefficient a can be chosen as a setting for the activation function.

The flexible values – weights

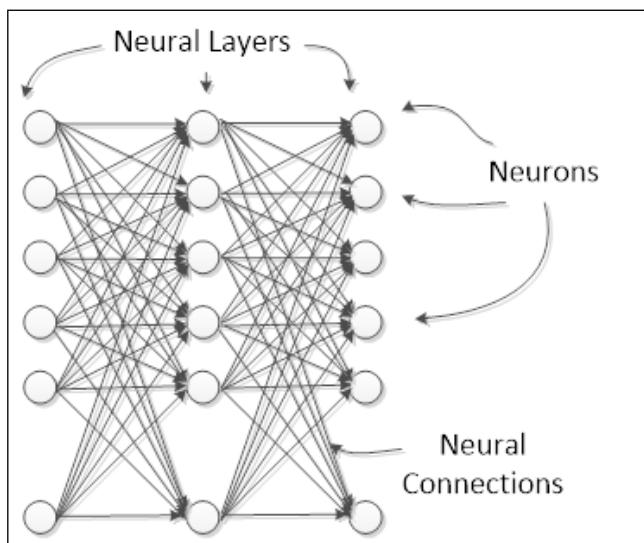
While the neural network structure can be fixed, weights represent the connections between neurons and they have the capability to amplify or attenuate incoming neural signals, thus modifying them and having the power to influence a neuron's output. Hence a neuron's activation will not be dependent on only the inputs, but on the weights too. Provided that the inputs come from other neurons or from the external world (stimuli), the weights are considered to be a neural network's established connections between its neurons. Since the weights are an internal neural network component and influence its outputs, they can be considered as neural network knowledge, provided that changing the weights will change the neural network's outputs, that is, its answers to external stimuli.

An extra parameter – bias

It is useful for the artificial neuron to have an independent component that adds an extra signal to the activation function: the **bias**. This parameter acts like an input, except for the fact that it is stimulated by one fixed value (usually 1), which is multiplied by an associated weight. This feature helps in the neural network knowledge representation as a more purely nonlinear system, provided that when all inputs are zero, that neuron won't necessarily produce a zero at the output, instead it can fire a different value according to the bias associated weight.

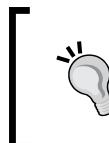
The parts forming the whole – layers

In order to abstract levels of processing, as our mind does, neurons are organized in layers. The input layer receives direct stimuli from the outside world, and the output layers fire actions that will have a direct influence on the outside world. Between these layers, there are a number of hidden layers, in the sense that they are invisible (hidden) from the outside world. In artificial neural networks, a layer has the same inputs and activation function for all its composing neurons, as shown in the following figure:



Neural networks can be composed of several linked layers, forming the so-called **multilayer networks**. Neural layers can then be classified as *Input*, *Hidden*, or *Output*.

In practice, an additional neural layer enhances the neural network's capacity to represent more complex knowledge.



Every neural network has at least an input/output layer irrespective of the number of layers. In the case of a multilayer network, the layers between the input and the output are called **hidden**

Learning about neural network architectures

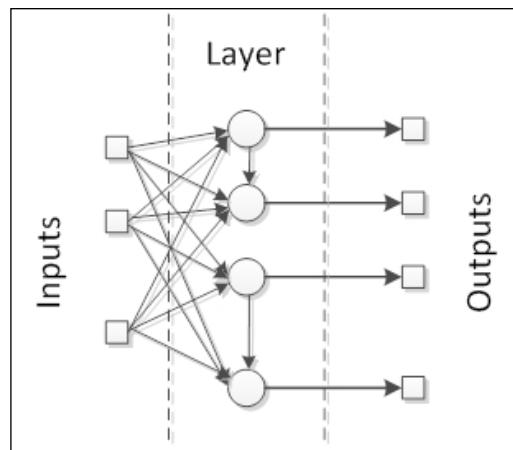
A neural network can have different layouts, depending on how the neurons or layers are connected to each other. Each neural network architecture is designed for a specific goal. Neural networks can be applied to a number of problems, and depending on the nature of the problem, the neural network should be designed in order to address this problem more efficiently.

Neural network architectures classification is two-fold:

- Neuron connections
- Monolayer networks
- Multilayer networks
- Signal flow
- Feedforward networks
- Feedback networks

Monolayer networks

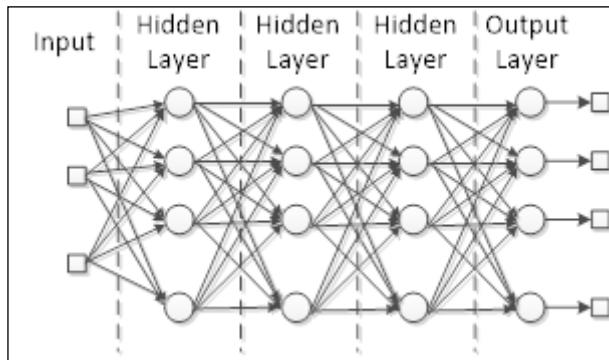
In this architecture, all neurons are laid out in the same level, forming one single layer, as shown in the following figure:



The neural network receives the input signals and feeds them into the neurons, which in turn produce the output signals. The neurons can be highly connected to each other with or without recurrence. Examples of these architectures are the single-layer perceptron, Adaline, self-organizing map, Elman, and Hopfield neural networks.

Multilayer networks

In this category, neurons are divided into multiple layers, each layer corresponding to a parallel layout of neurons that shares the same input data, as shown in the following figure:



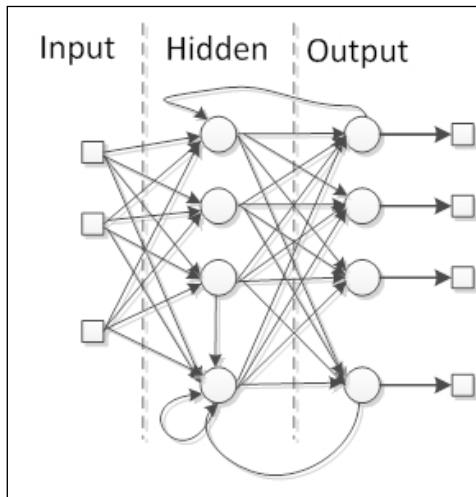
Radial basis functions and multilayer perceptrons are good examples of this architecture. Such networks are really useful for approximating real data to a function especially designed to represent that data. Moreover, because they have multiple layers of processing, these networks are adapted to learn from nonlinear data, being able to separate it or determine more easily the knowledge that reproduces or recognizes this data.

Feedforward networks

The flow of the signals in neural networks can be either in only one direction or in recurrence. In the first case, we call the neural network architecture feedforward, since the input signals are fed into the input layer; then, after being processed, they are forwarded to the next layer, just as shown in the figure in the multilayer section. Multilayer perceptrons and radial basis functions are also good examples of feedforward networks.

Feedback networks

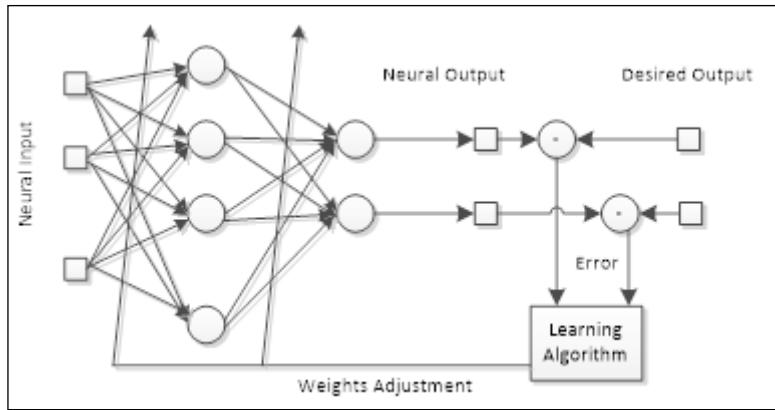
When the neural network has some kind of internal recurrence, it means that the signals are fed back in a neuron or layer that has already received and processed that signal, the network is of the feedback type. See the following figure of feedback networks:



The special reason to add recurrence in the network is the production of a dynamic behavior, particularly when the network addresses problems involving time series or pattern recognition, which require an internal memory to reinforce the learning process. However, such networks are particularly difficult to train, because there will eventually be a recursive behavior during the training (for example, a neuron whose outputs are fed back into its inputs), in addition to the arrangement of data for training. Most of the feedback networks are single layer, such as Elman and Hopfield networks, but it is possible to build a recurrent multilayer network, such as echo and recurrent multilayer perceptron networks.

From ignorance to knowledge – learning process

Neural networks learn by adjusting the connections between the neurons, namely the weights. As mentioned in the neural structure section, weights represent the neural network knowledge. Different weights cause the network to produce different results for the same inputs. So, a neural network can improve its results by adapting its weights according to a learning rule. The general schema of learning is depicted in the following figure:



The process depicted in the previous figure is called **supervised learning** because there is a desired output, but neural networks can also learn by the input data, without any desired output (supervision). In *Chapter 2, Getting Neural Networks to Learn*, we are going to dive deeper into the neural network learning process.

Let the coding begin! Neural networks in practice

In this book, we will cover the entire process of implementing a neural network by using the Java programming language. Java is an object-oriented programming language that was created in the 1990s by a small group of engineers from Sun Microsystems, later acquired by Oracle in the 2010s. Nowadays, Java is present in many devices that are part of our daily life.

In an object-oriented language, such as Java, we deal with classes and objects. A class is a blueprint of something in the real world, and an object is an instance of this blueprint, something like a car (class referring to all and any car) and my car (object referring to a specific car – mine). Java classes are usually composed of attributes and methods (or functions), that include **objects-oriented programming (OOP)** concepts. We are going to briefly review all of these concepts without diving deeper into them, since the goal of this book is just to design and create neural networks from a practical point of view. Four concepts are relevant and need to be considered in this process:

- **Abstraction:** The transcription of a real-world problem or rule into a computer programming domain, considering only its relevant features and dismissing the details that often hinder development.

- **Encapsulation:** Analogous to a product encapsulation by which some relevant features are disclosed openly (public methods), while others are kept hidden within their domain (private or protected), therefore avoiding misuse or excess of information.
- **Inheritance:** In the real world, multiple classes of objects share attributes and methods in a hierarchical manner; for example, a vehicle can be a superclass for car and truck. So, in OOP, this concept allows one class to inherit all features from another one, thereby avoiding the rewriting of code.
- **Polymorphism:** Almost the same as inheritance, but with the difference that methods with the same signature present different behaviors on different classes.

Using the neural network concepts presented in this chapter and the OOP concepts, we are now going to design the very first class set that implements a neural network. As could be seen, a neural network consists of layers, neurons, weights, activation functions, and biases. About layers, there are three types of them: input, hidden, and output. Each layer may have one or more neurons. Each neuron is connected either to a neural input/output or to another neuron, and these connections are known as weights.

It is important to highlight that a neural network may have many hidden layers or none, because the number of neurons in each layer may vary. However, the input and output layers have the same number of neurons as the number of neural inputs/outputs, respectively.

So, let's start implementing. Initially, we are going to define the following classes:

- **Neuron:** Defines the artificial neuron
- **NeuralLayer:** Abstract class that defines a layer of neurons
- **InputLayer:** Defines the neural input layer
- **HiddenLayer:** Defines the layers between input and output
- **OutputLayer:** Defines the neural output layer
- **InputNeuron:** Defines the neuron that is present at the neural network input
- **NeuralNet:** Combines all previous classes into one ANN structure

In addition to these classes, we should also define an `IActivationFunction` interface for activation functions. This is necessary because `Activation` functions will behave like methods, but they will need to be assigned as a neuron property. So we are going to define classes for activation functions that implement this interface:

- Linear
- Sigmoid
- Step
- HyperTan

Our first chapter coding is almost complete. We need to define two more classes. One for handling eventually thrown exceptions (`NeuralException`) and another to generate random numbers (`RandomNumberGenerator`). Finally, we are going to separate these classes into two packages:

- `edu.packt.neuralnet`: For the neural network related classes (`NeuralNet`, `Neuron`, `NeuralLayer`, and so on)
- `edu.packt.neuralnet.math`: For the math related classes (`IActivationFunction`, `Linear`, and so on)

To save space, we are not going to write the full description of each class, instead we are going to address the key features of most important classes. However, the reader is welcomed to take a glance at the Javadoc documentation of the code, in order to get more details on the implementation.

The neuron class

This is the very foundation class for this chapter's code. According to the theory, an artificial neuron has the following attributes:

- Inputs
- Weights
- Bias
- Activation function
- Output

It is also important to define one attribute that will be useful in future examples, that is the output before activation function. We then have the implementation of the following properties:

```
public class Neuron {  
    protected ArrayList<Double> weight;  
    private ArrayList<Double> input;  
    private Double output;  
    private Double outputBeforeActivation;  
    private int numberofInputs = 0;  
    protected Double bias = 1.0;  
    private IActivationFunction activationFunction;  
    ...  
}
```

When instantiating a neuron, we need to specify how many inputs are going to feed values to it, and what should be its activation function. So let's take a look on the constructor:

```
public Neuron(int numberofinputs,IActivationFunction iaf){  
    numberofInputs=numberofinputs;  
    weight=new ArrayList<>(numberofinputs+1);  
    input=new ArrayList<>(numberofinputs);  
    activationFunction=iaf;  
}
```

Note that we define one extra weight for the bias. One important step is the initialization of the neuron, that is, how the weights receive their first values. This is defined in the `init()` method, by which weights receive randomly generated values by the `RandomNumberGenerator` static class. Note the need to prevent an attempt to set a value outside the bounds of the weight array:

```
public void init(){  
    for(int i=0;i<=numberofInputs;i++){  
        double newWeight = RandomNumberGenerator.GenerateNext();  
        try{  
            this.weight.set(i, newWeight);  
        }  
        catch(IndexOutOfBoundsException iobe){  
            this.weight.add(newWeight);  
        }  
    }  
}
```

Finally, let's take a look on how the output values are calculated in the `calc()` method:

```
public void calc() {
    outputBeforeActivation=0.0;
    if(numberOfInputs>0){
        if(input!=null && weight!=null){
            for(int i=0;i<=numberOfInputs;i++){
                outputBeforeActivation+=(i==numberOfInputs?bias:input.
get(i))*weight.get(i);
            }
        }
        output=activationFunction.calc(outputBeforeActivation);
    }
}
```

Note that first, the products of all inputs and weights are summed (the bias multiplies the last weight - `i==numberOfInputs`), and this value is saved in the `outputBeforeActivation` property. The activation function calculates the neuron's output with this value.

The NeuralLayer class

In this class we are going to group the neurons that are aligned in the same layer. Also, there is a need to define links between layers, since one layer forwards values to another. So the class will have the following properties:

```
public abstract class NeuralLayer {
    protected int numberOfNeuronsInLayer;
    private ArrayList<Neuron> neuron;
    protected IActivationFunction activationFnc;
    protected NeuralLayer previousLayer;
    protected NeuralLayer nextLayer;
    protected ArrayList<Double> input;
    protected ArrayList<Double> output;
    protected int numberOfInputs;
    ...
}
```

Note that this class is abstract, the layer classes that can be instantiated are `InputLayer`, `HiddenLayer`, and `OutputLayer`. In order to create one layer, one must use one of these classes' constructors that work quite similar:

```
public InputLayer(int numberofinputs);
public HiddenLayer(int numberofneurons, IActivationFunction iaf,
int numberofinputs);
public OutputLayer(int numberofneurons, IActivationFunction iaf,
int numberofinputs);
```

Layers are initialized and calculated as well as the neurons, they also implement the methods `init()` and `calc()`. The signature protected guarantees that only the subclasses can call or override these methods:

```
protected void init(){
    for(int i=0;i<numberOfNeuronsInLayer;i++){
        try{
            neuron.get(i).setActivationFunction(activationFnc);
            neuron.get(i).init();
        }
        catch(IndexOutOfBoundsException iobe){
            neuron.add(new Neuron(numberOfInputs,activationFnc));
            neuron.get(i).init();
        }
    }
}
protected void calc(){
    for(int i=0;i<numberOfNeuronsInLayer;i++){
        neuron.get(i).setInputs(this.input);
        neuron.get(i).calc();
        try{
            output.set(i,neuron.get(i).getOutput());
        }
        catch(IndexOutOfBoundsException iobe){
            output.add(neuron.get(i).getOutput());
        }
    }
}
```

The ActivationFunction interface

Before we define the `NeuralNetwork` class, let's take a look at an example of Java code with interface:

```
public interface IActivationFunction {
    double calc(double x);
```

```
public enum ActivationFunctionENUM {
    STEP, LINEAR, SIGMOID, HYPERTAN
}
```

The `calc()` signature method is used by a specific Activation Function that implements this interface, the `Sigmoid` function, for example:

```
public class Sigmoid implements IActivationFunction {
    private double a=1.0;
    public Sigmoid(double _a){this.a=_a;}
    @Override
    public double calc(double x){
        return 1.0/(1.0+Math.exp(-a*x));
    }
}
```

This is one example of polymorphism, whereby a class or method may present different behavior, but yet under the same signature, allowing a flexible application.

The neural network class

Finally, let's define the neural network class. It has been known so far that neural networks organize neurons in layers, and every neural network has at least two layers, one for gathering the inputs and one for processing the outputs, and a variable number of hidden layers. Therefore our `NeuralNet` class will have these properties, in addition to other properties similar to the `neuron` and the `NeuralLayer` classes, such as `numberOfInputs`, `numberOfOutputs`, and so on:

```
public class NeuralNet {
    private InputLayer inputLayer;
    private ArrayList<HiddenLayer> hiddenLayer;
    private OutputLayer outputLayer;
    private int numberOfHiddenLayers;
    private int numberOfInputs;
    private int numberOfOutputs;
    private ArrayList<Double> input;
    private ArrayList<Double> output;
    ...
}
```

The constructor of this class has more arguments than the previous classes:

```
public NeuralNet(int numberofinputs,int numberoffoutputs,  
                 int [] numberofhiddenneurons,IActivationFunction []  
                 hiddenAcFnc,  
                 IActivationFunction outputAcFnc)
```

Provided that the number of hidden layers is variable, we should take into account that there may be many hidden layers or none, and in each of them there will be a variable number of hidden neurons. So the best way to deal with this variability is to represent the quantity of neurons in each hidden layer as a vector of integers (argument `numberofhiddenlayers`). Moreover, one needs to define the activation functions for each hidden layer, and for the output layer as well, to that goal serve the arguments `hiddenActivationFnc` and `outputAcFnc`.

To save space in this chapter we are not going to show the full implementation of this constructor, but we can show the example for the definition of layers and the links between them. First, the input layer is defined observing the number of inputs:

```
input=new ArrayList<>(numberofinputs);  
inputLayer=new InputLayer(numberofinputs);
```

A hidden layer will be defined depending on its position, if it is right after the input layer, the definition is as follows:

```
hiddenLayer.set(i,new HiddenLayer(numberofhiddenneurons[i],  
                                 hiddenAcFnc[i],  
                                 inputLayer.getNumberOfNeuronsInLayer()));  
inputLayer.setNextLayer(hiddenLayer.get(i));
```

Or else it will get the reference of the previous hidden layer:

```
hiddenLayer.set(i, new HiddenLayer(numberofhiddenneurons[i],  
                                 hiddenAcFnc[i],hiddenLayer.get(i-1).getNumberOfNeuronsInLayer()));  
hiddenLayer.get(i-1).setNextLayer(hiddenLayer.get(i));
```

As for the output layer, the definition is very similar to the latter case, except for the `OutputLayer` class and the fact that there may be no hidden layers:

```
if(numberOfHiddenLayers>0){  
    outputLayer=new OutputLayer(numberoffoutputs,outputAcFnc,  
                               hiddenLayer.get(numberOfHiddenLayers-1).  
                               getNumberOfNeuronsInLayer());  
    hiddenLayer.get(numberOfHiddenLayers-1).setNextLayer(outputLayer);  
}else{  
    outputLayer=new OutputLayer(numberofinputs, outputAcFnc,  
                               numberoffoutputs);  
    inputLayer.setNextLayer(outputLayer);  
}
```

The `calc()` method executes the forwarding flow of signals from the input to the output end:

```
public void calc() {
    inputLayer.setInputs(input);
    inputLayer.calc();
    for(int i=0;i<numberOfHiddenLayers;i++) {
        HiddenLayer hl = hiddenLayer.get(i);
        hl.setInputs(hl.getPreviousLayer().getOutputs());
        hl.calc();
    }
    outputLayer.setInputs(outputLayer.getPreviousLayer().getOutputs());
    outputLayer.calc();
    this.output=outputLayer.getOutputs();
}
```

In appendix C, we present the reader the full documentation of the classes along with their UML class and package diagrams that will surely help as a reference for this book.

Time to play!

Now let's apply these classes and get some results. The following code has a test class, a main method with an object of the `NeuralNet` class called `nn`. We are going to define a simple neural network with two inputs, one output, and one hidden layer containing three neurons:

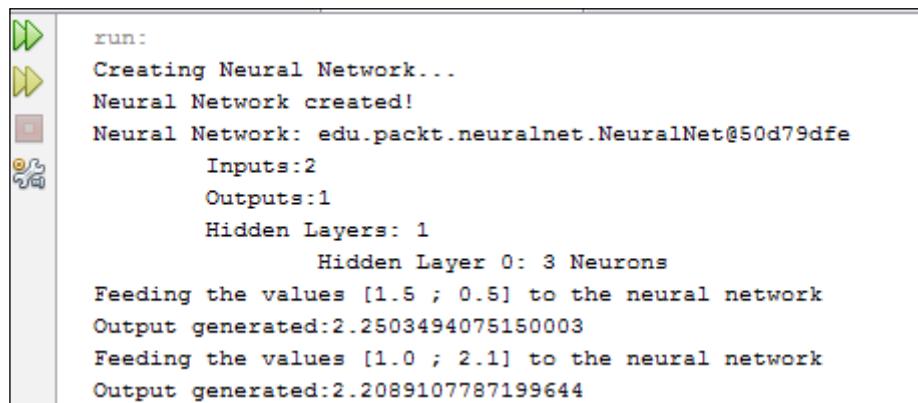
```
public class NeuralNetConsoleTest {
    public static void main(String[] args) {
        RandomNumberGenerator.seed=0;

        int numberOfInputs=2;
        int numberOfOutputs=1;
        int[] numberOfHiddenNeurons= { 3 };
        IActivationFunction[] hiddenAcFnc = { new Sigmoid(1.0) } ;
        Linear outputAcFnc = new Linear(1.0);
        System.out.println("Creating Neural Network...");
        NeuralNet nn = new NeuralNet(numberOfInputs,numberOfOutputs,
            numberOfHiddenNeurons,hiddenAcFnc,outputAcFnc);
        System.out.println("Neural Network created!");
        nn.print();
        ...
    }
}
```

Still in this code, let's feed to the neural network two sets of data, and let's see what output it is going to produce:

```
double [] neuralInput = { 1.5 , 0.5 };  
double [] neuralOutput;  
System.out.println("Feeding the values ["+String.  
valueOf(neuralInput[0])+" ; "+  
String.valueOf(neuralInput[1])+"] to the neural  
network");  
nn.setInputs(neuralInput);  
nn.calc();  
neuralOutput=nn.getOutputs();  
  
neuralInput[0] = 1.0;  
neuralInput[1] = 2.1;  
...  
nn.setInputs(neuralInput);  
nn.calc();  
neuralOutput=nn.getOutputs();
```

This code gives the following output:



The screenshot shows the Java IDE interface with the code execution results. The output window displays the following text:

```
run:  
Creating Neural Network...  
Neural Network created!  
Neural Network: edu.packt.neuralnet.NeuralNet@50d79dfe  
    Inputs:2  
    Outputs:1  
    Hidden Layers: 1  
        Hidden Layer 0: 3 Neurons  
Feeding the values [1.5 ; 0.5] to the neural network  
Output generated:2.2503494075150003  
Feeding the values [1.0 ; 2.1] to the neural network  
Output generated:2.2089107787199644
```

It's relevant to remember that each time that the code runs, it generates new pseudo random weight values, unless you work with the same seed value. If you run the code exactly as provided here, the same values will appear in console:

Summary

In this chapter, we've seen an introduction to the neural networks, what they are, what they are used for, and their basic concepts. We've also seen a very basic implementation of a neural network in the Java programming language, wherein we applied the theoretical neural network concepts in practice, by coding each of the neural network elements. It's important to understand the basic concepts before we move on to advanced concepts. The same applies to the code implemented with Java.

In the next chapter, we will delve into the learning process of a neural network and explore the different types of leaning with simple examples.

2

Getting Neural Networks to Learn

Now that you have been introduced to neural networks, it is time to learn about their learning process. In this chapter, we're going to explore the concepts involved with neural network learning, along with their implementation in Java. We will make a review on the foundations and inspirations for the neural learning process that will guide us in implementation of learning algorithms in Java to be applied on our neural network code. In summary, these are the concepts addressed in this chapter:

- Learning ability
- How learning helps
- Learning paradigms
- Supervised
- Unsupervised
- The learning process
- Optimization foundations
- The cost function
- Error measurement
- Learning algorithms
- Delta rule
- Hebbian rule
- Adaline/perceptron
- Training, test, and validation

- Dataset splitting
- Overfitting and overtraining
- Generalization

Learning ability in neural networks

What is really amazing in neural networks is their capacity to learn from the environment, just like brain-gifted beings are able to do so. We, as humans, experience the learning process through observations and repetitions, until some task, or concept is completely mastered. From the physiological point of view, the learning process in the human brain is a reconfiguration of the neural connections between the nodes (neurons), which results in a new thinking structure.

While the connectionist nature of neural networks distributes the learning process all over the entire structure, this feature makes this structure flexible enough to learn a wide variety of knowledge. As opposed to ordinary digital computers that can execute only tasks they are programmed to do, neural systems are able to improve and perform new activities according to some satisfaction criteria. In other words, neural networks don't need to be programmed; they learn the program by themselves.

How learning helps solving problems

Considering that every task to solve may have a huge number of theoretically possible solutions, the learning process seeks to find an optimal solution that can produce a satisfying result. The use of structures such as **artificial neural networks (ANN)** is encouraged due to their ability to acquire knowledge of any type, strictly by receiving input stimuli, that is, data relevant to the task/problem. At first, the ANN will produce a random result and an error, and based on this error, the ANN parameters are adjusted.



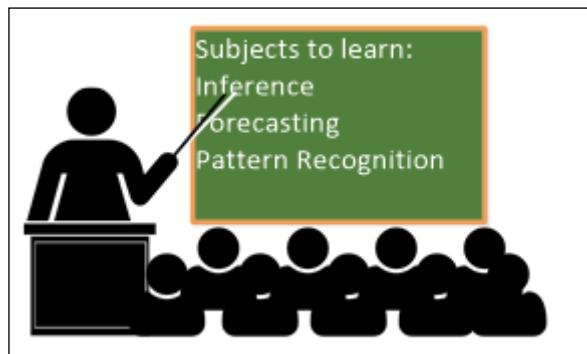
We can then think of the ANN parameters (weights) as the components of a solution. Let's imagine that each weight corresponds to a dimension and one single solution represents a single point in the solution hyperspace. For each single solution, there is an error measure informing how far that solution is from the satisfaction criteria. The learning algorithm then iteratively seeks a solution closer to the satisfaction criteria.

Learning paradigms

There are basically two types of learning for neural networks, namely supervised, and unsupervised. The learning in the human mind, for example, also works in this way. We are able to build knowledge from observations without any target (unsupervised) or we can have a teacher who shows us the right pattern to follow (supervised). The difference between these two paradigms relies mainly on the relevancy of a target pattern, and varies from problem to problem.

Supervised learning

This learning type deals with pairs of xs (independent values), and ys (dependent values) with the objective to map them in a function . Here the Y data is the *supervisor*, the target desired outputs, and the X are the source independent data that jointly generate the Y data. It is analogous to a teacher who is teaching somebody a certain task to be performed:



One particular feature of this learning paradigm is that there is a direct error reference which is just the comparison between the target and the current actual result. The network parameters are fed into a cost function which quantifies the mismatch between desired and actual outputs.



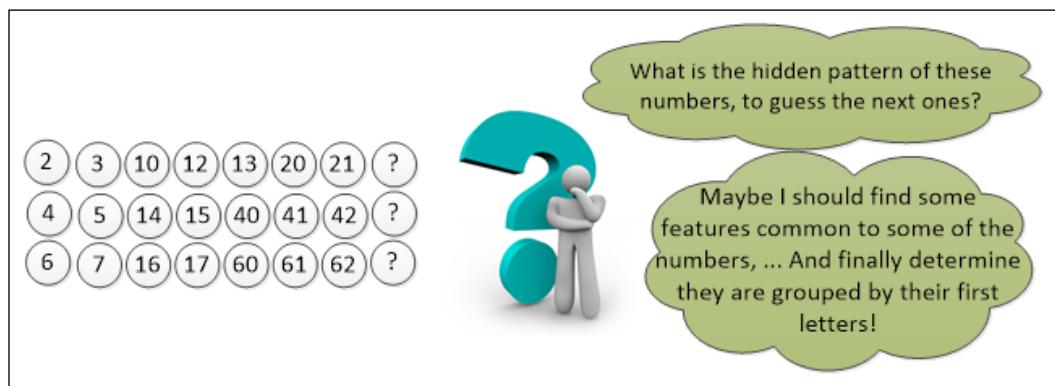
A cost function is just a measurement to be minimized in an optimization problem. That means one seeks to find the parameters that drive the cost function to the lowest possible value.

The cost function will be covered in detail later in this chapter

The supervised learning is suitable for tasks having a defined pattern to be reproduced. Some examples include classification of images, speech recognition, function approximation, and forecasting. Note that the neural network should be provided a previous knowledge of both input independent values (X) and the output dependent values (Y). The presence of a dependent output value is a necessary condition for the learning to be supervised.

Unsupervised learning

In unsupervised learning, we deal only with data without any labeling or classification. Instead, one tries to make an inference and extract knowledge by taking into account only the independent data X :



This is analogous to self-learning, when someone takes into account his/her own experience and a set of supporting criteria. In unsupervised learning, we don't have a defined desired pattern; instead, we use the provided data to infer a dependent output Y without any supervision.

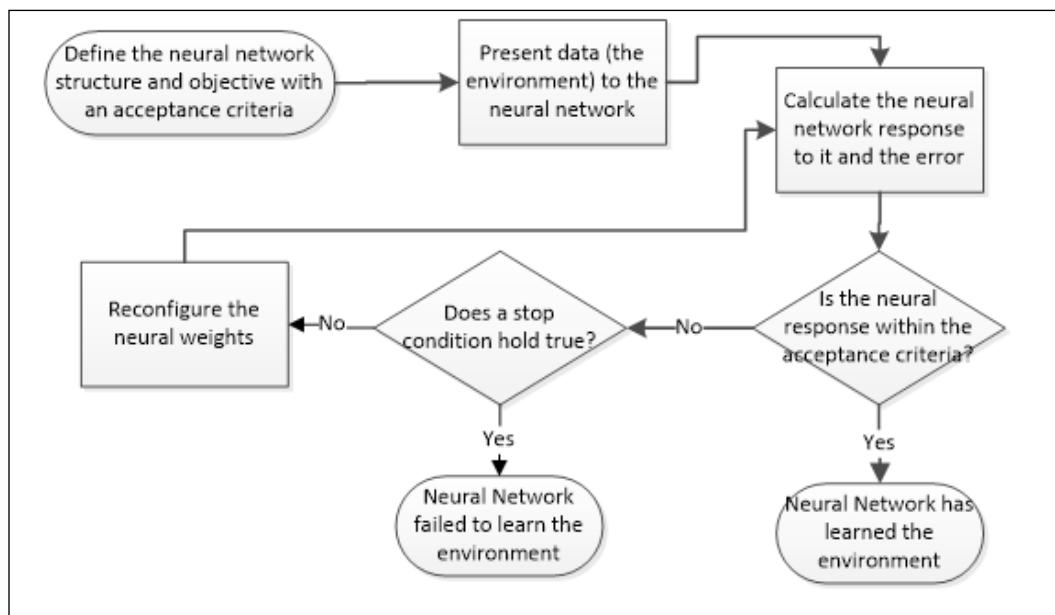
[] In unsupervised learning, the closer the independent data is, more similar the generated output should be, and this should be considered in the cost function, as opposed to the supervised paradigm. []

Examples of tasks that unsupervised learning can be applied to are clustering, data compression, statistical modeling, and language modeling. This learning paradigm will be covered in more detail in *Chapter 4, Self-Organizing Maps*.

The learning process

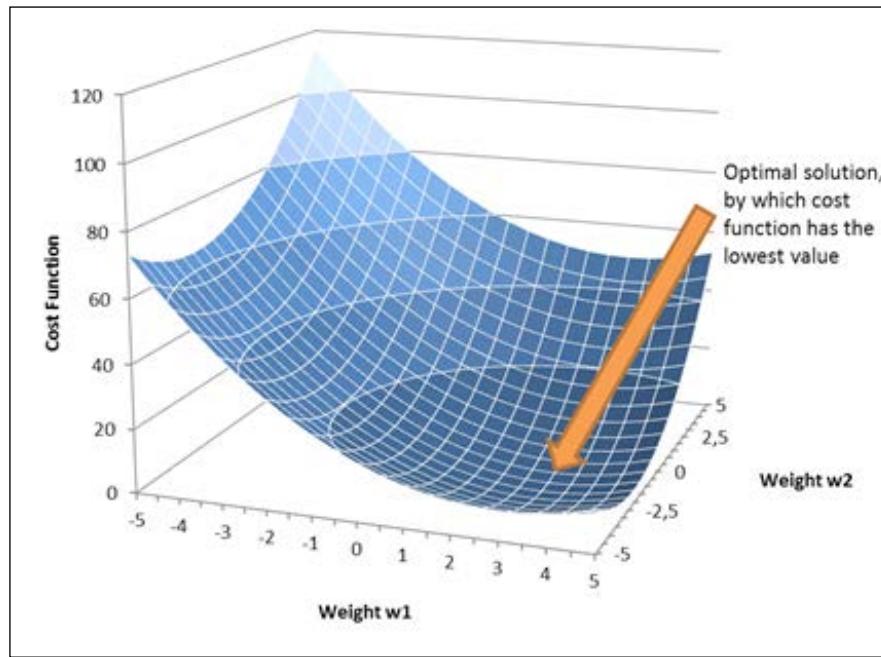
So far, we have theoretically defined the learning process and how it is carried out. But in practice, we must dive a little bit deeper into the mathematical logic, in order to implement the learning algorithm itself. For simplicity, in this chapter, we are basically covering the supervised learning case; however, we will present here a rule for updating weights in unsupervised learning. A learning algorithm is a procedure that drives the learning process of neural networks, and it is strongly determined by the neural network architecture. From the mathematical point of view, one wishes to find the optimal weights W that can drive the cost function $C(X, Y)$ to the lowest possible value. However, sometimes the learning process cannot find a good set of weights capable of meeting the acceptance criteria, but a stop condition must be set to prevent the neural network from learning forever and thereby causing the Java program to freeze.

In general, this process is carried out in the fashion presented in the following flowchart:



The cost function finding the way down to the optimum

Now let's find out in detail what role the cost function plays. Let's think of cost function as a two-variable function whose shape is represented by a hypersurface. For simplicity, let's consider for now only two weights (two-dimensional space plus height representing cost function). Suppose our cost function has the following shape:



Visually, we can see that there is an optimum, by which the cost function roughly approaches zero. But how can we make this programmatically? The answer lies in the mathematical optimization, whereby the cost function is defined as an optimization problem:

$$\min_{X, Y, W \in R^N} C(X, Y, W)$$

By recalling the optimization Fermat's theorems, the optimal solution lies in a place where the surface slope should be zero at all dimensions, that is, the partial derivative should be zero, and it should be convex (for the minimum case). Considering that one starts with an arbitrary solution W , the search for the optimum should take into account the direction to which the surface height is going down. This is the so-called gradient method.

Learning in progress - weight update

According to the cost function used, an update rule will dictate how the weights, the neural flexible parameters, should be changed, so the cost function will have a lower value at the new weights:

$$W(k + 1) = W(k) + \Delta W$$

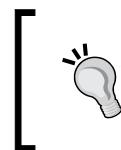
Here, k refers to the k th iteration and $W(k)$ refers to the neural weights at the k th iteration, and subsequently $k+1$ refers to the next iteration.

The weight update operation can be performed in online or batch mode. Online here implies that the weights are updated after every single record from the dataset. Batch update means that first all the records from the dataset are presented to the neural network before it starts updating its weights. This will be explored in detail in the code at the end of this chapter.

Calculating the cost function

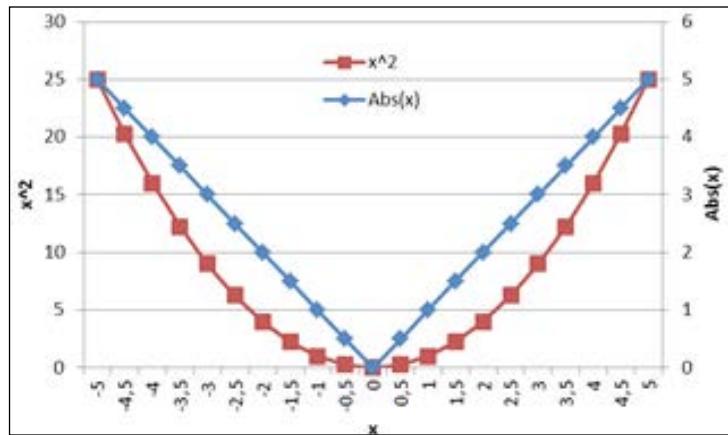
When a neural network learns, it receives data from an environment and adapts its weights according to the objective. This data is referred to as the training dataset and has several samples. The idea behind the word training lies in the process of adapting the neural weights, as if they were *training* to give the desired response in the neural network. While the neural network is still learning, there is an error between the target outputs (Y) and the neural outputs (\hat{Y}), in the supervised case:

$$e = Y - \hat{Y}$$



Some literature about neural networks identifies the target variable with the letter T , and the neural output as Y , while in this book we are going to denote it as Y and , to not confuse the reader, since it was presented initially as Y .

Well, given that the training dataset has multiple values, there will be N values of errors for each single record. So, how to get an overall error? One intuitive approach is to get an average of all errors, but this is misleading. The error vector can take on both positive and negative values, therefore an average of all error values is very likely to be closer to zero, regardless of how big the error measurements may be. Using the absolute value to generate an average seems to be a smarter approach, but this function has a discontinuity at the origin, what is awkward in calculating its derivative:



So, the reasonable option we have is to use the average of a quadratic sum of the error, also known as **mean squared error (MSE)**:

$$MSE(Y, \hat{Y}) = \frac{1}{N} \sum_i^N (Y_i - \hat{Y}_i)^2$$

General error and overall error

We need to clarify one thing before going further. The neural network being a multiple output structure, we have to deal with the multiple output case, when instead of an error vector, we will have an error matrix:

$$E = \begin{pmatrix} (Y_{kj} - \hat{Y}_{kj}) & \dots & (Y_{kn} - \hat{Y}_{kn}) \\ \vdots & \ddots & \vdots \\ (Y_{Nj} - \hat{Y}_{Nj}) & \dots & (Y_{Nn} - \hat{Y}_{Nn}) \end{pmatrix}$$

Well, in such cases, there may be a huge number of errors to work with, whether regarding one specific output, a specific record, or the whole dataset. To facilitate understanding, let's call the specific-to-record error the general error, by which all output errors are given one scalar for the general output error; and the error referring to the whole data as overall error.

The general error for single output network is a mere difference between target and output, but in the multiple output case, it needs be composed of each output error. As we saw, the squared error is a suitable approach to summarize error measures, therefore the general error can be calculated using the square of each output error:

$$e_k = \frac{1}{2} \sum_j^n (Y_{kj} - \hat{Y}_{kj})^2$$

As for the overall error, it actually considers the general error but for all records in the dataset. Since the dataset can be huge, it is better to calculate the overall error using the MSE of the quadratic general errors.

Can the neural network learn forever? When is it good to stop?

As the learning process is run, the neural network must give results closer and closer to the expectation, until finally it reaches the acceptation criteria or one limitation in learning iterations, that we'll call epochs. The learning process is then considered to be finished when one of these conditions is met:

- **Satisfaction criterion:** minimum overall error or minimum weight distance, according to the learning paradigm
- **Maximum number of epochs**

Examples of learning algorithms

Let's now merge the theoretical content presented so far together into simple examples of learning algorithms. In this chapter, we are going to explore a couple of learning algorithms in single layer neural networks; multiple layers will be covered in the next chapter.

In the Java code, we will create one new superclass `LearningAlgorithm` in a new package `edu.packt.neural.learn`. Another useful package called `edu.packt.neural.data` will be created to handle datasets that will be processed by the neural network, namely the classes `NeuralInputData`, and `NeuralOutputData`, both referenced by the `NeuralDataSet` class. We recommend the reader takes a glance at the code documentation to understand how these classes are organized, to save text space here.

The `LearningAlgorithm` class has the following attributes and methods:

```
public abstract class LearningAlgorithm {  
    protected NeuralNet neuralNet;  
    public enum LearningMode {ONLINE,BATCH};  
    protected enum LearningParadigm {SUPERVISED,UNSUPERVISED};  
    //...  
    protected int MaxEpochs=100;  
    protected int epoch=0;  
    protected double MinOverallError=0.001;  
    protected double LearningRate=0.1;  
    protected NeuralDataSet trainingDataSet;  
    protected NeuralDataSet testingDataSet;  
    protected NeuralDataSet validatingDataSet;  
    public boolean printTraining=false;  
    public abstract void train() throws NeuralException;  
    public abstract void forward() throws NeuralException;  
    public abstract void forward(int i) throws NeuralException;  
    public abstract Double calcNewWeight(int layer,int input,int  
neuron) throws NeuralException;  
    public abstract Double calcNewWeight(int layer,int input,int  
neuron,double error) throws NeuralException;  
    //...  
}
```

The `neuralNet` object is a reference to the neural network that will be trained by this learning algorithm. The enums define the learning mode and learning paradigm. The learning executing parameters are defined (`MaxEpochs`, `MinOverallError`, `LearningRate`), and the datasets that will be taken into account during the learning process.

The method `train()` should be overridden by each learning algorithm implementation. All the training process will occur in this method. The methods `forward()` and `forward(int k)` process the neural network with all input data and with the k th input data record, respectively. And finally, the method `calcNewWeight()` will perform the weight update for the weight connecting an input to a neuron in a specific layer. A variation in the `calcNewWeight()` method allows providing a specific error to be taken in the update operation.

The delta rule

This algorithm updates the weights according to the cost function. Following the gradient approach, one wants to know which weights can drive the cost function to a lower value. Note that we can find the direction by computing the partial derivative of the cost function to each of the weights. To help in understanding, let's consider one simple approach with only one neuron, one weight, and one bias, and therefore one input. The output will be as follows:

$$\hat{Y} = g(X \cdot w + b)$$

Here, g is the activation function, X is the vector containing x values, and \hat{Y} is the output vector generated by the neural network. The general error for the k th sample is quite simple:

$$E_k = y_k - \hat{y}_k$$

However, it is possible to define this error as square error, N-degree error, or MSE. But, for simplicity, let's consider the simple error difference for the general error. Now the overall error, that will be the cost function, should be computed as follows:

$$C(X, Y, \hat{Y}) = \frac{1}{N} \sum_k^N E_k^2$$

The weight and bias are updated according to the delta rule, that considers the partial derivatives $\frac{\partial C(X, Y, \hat{Y})}{\partial w}$ and $\frac{\partial C(X, Y, \hat{Y})}{\partial b}$ with respect to the weight and the bias, respectively. For the batch training mode, X and E are vectors:

$$\Delta w = \alpha \frac{\partial C(X, Y, \hat{Y})}{\partial w} = \alpha E^T X g'(X \cdot w + b)^T$$

$$\Delta b = \alpha \frac{\partial C(X, Y, \hat{Y})}{\partial b} = \alpha E^T \begin{pmatrix} 1 \\ \vdots \end{pmatrix} g'(X \cdot w + b)^T$$

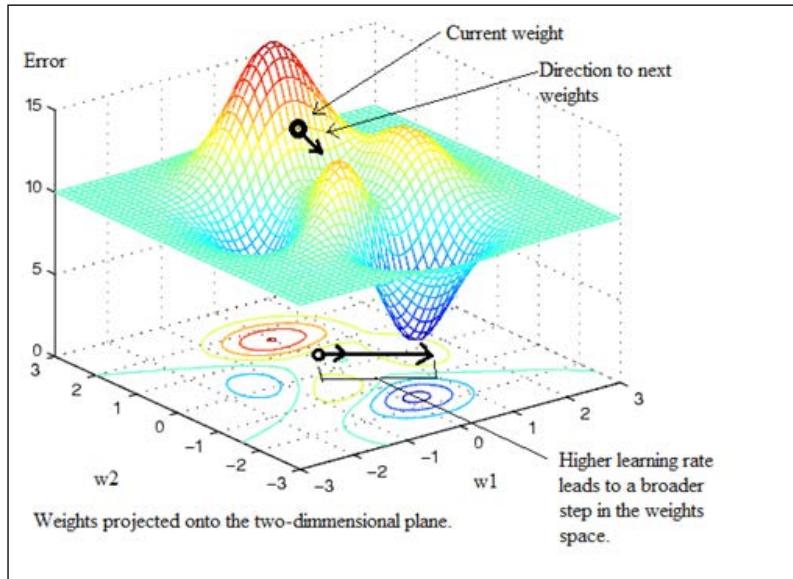
If the training mode is online, we don't need to perform dot product:

$$\Delta w = \alpha \frac{\partial C(X, Y, \hat{Y})}{\partial w} = \alpha E_k X_k g'(X_k \cdot w + b)$$

$$\Delta b = \alpha \frac{\partial C(X, Y, \hat{Y})}{\partial b} = \alpha E_k g'(X_k \cdot w + b)$$

The learning rate

Note in the preceding equations the presence of the term α that indicates the learning rate. It plays an important role in weight update, because it can drive faster or slower to the minimum cost value. Let's see a cost error surface in relation to two weights:



Implementing the delta rule

We will implement the delta rule in a class called `DeltaRule`, that will extend the `LearningAlgorithm` class:

```
public class DeltaRule extends LearningAlgorithm {
    public ArrayList<ArrayList<Double>> error;
    public ArrayList<Double> generalError;
    public ArrayList<Double> overallError;
    public double overallGeneralError;
    public double degreeGeneralError=2.0;
    public double degreeOverallError=0.0;
    public enum ErrorMeasurement {SimpleError,
        SquareError,NDegreeError,MSE}

    public ErrorMeasurement generalErrorMeasurement=ErrorMeasurement.
        SquareError;
    public ErrorMeasurement overallErrorMeasurement=ErrorMeasurement.
        MSE;
    private int currentRecord=0;
    private ArrayList<ArrayList<ArrayList<Double>>> newWeights;
//...
}
```

The errors discussed in the error measurement section (general and overall errors) are implemented in the `DeltaRule` class, because the delta rule learning algorithm considers these errors during the training. They are arrays because there will be a general error for each dataset record, and there will be an overall error for each output. An attribute `overallGeneralError` takes on the cost function result, or namely the overall error for all outputs and records. A matrix called `error`, stores the errors for each output record combination.

This class also allows multiple ways of calculating the overall and general errors. The attributes `generalErrorMeasurement` and `overallErrorMeasurement` can take on one of the input values for simple error, square error calculation, Nth degree error (cubic, quadruple, and so on), or the MSE. The default will be simple error for the general error and MSE for the overall.

Two important attributes are worth noting in this code: `currentRecord` refers to the index of the record being fed into the neural network during training, and the `newWeights` cubic matrix is a collection of all new values of weights that will be updated in the neural network. The `currentRecord` attribute is useful in the online training, and the `newWeights` matrix helps the neural network to keep all of its original weights until all new weights calculation is finished, preventing new weights to be updated during the forward processing stage, what could compromise the training quality significantly.

The core of the delta rule learning - train and calcNewWeight methods

To save space, we will not detail here the implementation of the forward methods. As described in the previous section, forward means that neural dataset records should be fed into the neural network and then the error values are calculated:

```
@Override
public void train() throws NeuralException{
//...
switch(learningMode) {
    case BATCH: //this is the batch training mode
        epoch=0;
        forward(); //all data are presented to the neural network
        while(epoch<MaxEpochs &&
overallGeneralError>MinOverallError){ //continue condition
            epoch++; //new epoch
            for(int j=0;j<neuralNet.getNumberOfOutputs();j++) {
                for(int i=0;i<=neuralNet.getNumberOfInputs();i++) {
                    //here the new weights are calculated
                    newWeights.get(0).get(j).set(i,calcNewWeight(0,i,j));
                }
            }
        }
        //only after all weights are calculated, they are applied
        applyNewWeights();
        // the errors are updated with the new weights
        forward();
    }
    break;
case ONLINE://this is the online training
    epoch=0;
    int k=0;
    currentRecord=0; //this attribute is used in weight update
    forward(k); //only the k-th record is presented
```

```

        while(epoch<MaxEpochs && overallGeneralError>MinOverallError) {
            for(int j=0;j<neuralNet.getNumberOfOutputs();j++) {
                for(int i=0;i<=neuralNet.getNumberOfInputs();i++) {
                    newWeights.get(0).get(j).set(i,calcNewWeight(0,i,j));
                }
            }
            //the new weights will be considered for the next record
            applyNewWeights();
            currentRecord++;
            if(k>=trainingDataSet.numberOfRecords) {
                k=0; //if it was the last record, again the first
                currentRecord=0;
                epoch++; //epoch completes after presenting all records
            }
            forward(k); //presenting the next record
        }
        break;
    }
}

```

We note that in the `train()` method, there is a loop with a condition to continue training. This means that while the training will stop when this condition no longer holds true. The condition checks the epoch number and the overall error. When the epoch number reaches the maximum or the error reaches the minimum, the training is finished. However, there are some cases in which the overall error fails to meet the minimum requirement, and the neural network needs to stop training.

The new weight is calculated using the `calcNewWeight()` method:

```

@Override
public Double calcNewWeight(int layer,int input,int neuron)
    throws NeuralException{
//...
Double deltaWeight=LearningRate;
Neuron currNeuron=neuralNet.getOutputLayer().getNeuron(neuron);
switch(learningMode){
    case BATCH: //Batch mode
        ArrayList<Double> derivativeResult=currNeuron
            .derivativeBatch(trainingDataSet.getArrayInputData());
        ArrayList<Double> _ithInput;
        if(input<currNeuron.getNumberOfInputs()) { // weights
            _ithInput=trainingDataSet.getIthInputArrayList(input);
        }
        else{ // bias

```

```
_ithInput=new ArrayList<>();
for(int i=0;i<trainingDataSet.numberOfRecords;i++) {
    _ithInput.add(1.0);
}
}
Double multDerivResultIthInput=0.0; // dot product
for(int i=0;i<trainingDataSet.numberOfRecords;i++) {
    multDerivResultIthInput+=error.get(i).get(neuron)*
        derivativeResult.get(i)*_ithInput.get(i);
}
deltaWeight*=multDerivResultIthInput;
break;
case ONLINE:
    deltaWeight*=error.get(currentRecord).get(neuron);
    deltaWeight*=currNeuron.derivative(neuralNet.getInputs());
    if(input<currNeuron.getNumberOfInputs()){
        deltaWeight*=neuralNet.getInput(input);
    }
    break;
}
return currNeuron.getWeight(input)+deltaWeight;
//...
}
```

Note that in the weight update, there is a call to the derivative of the activation function of the given neuron. This is needed to meet the delta rule. In the activation function interface, we've added this method `derivative()` to be overridden in each of the implementing classes.

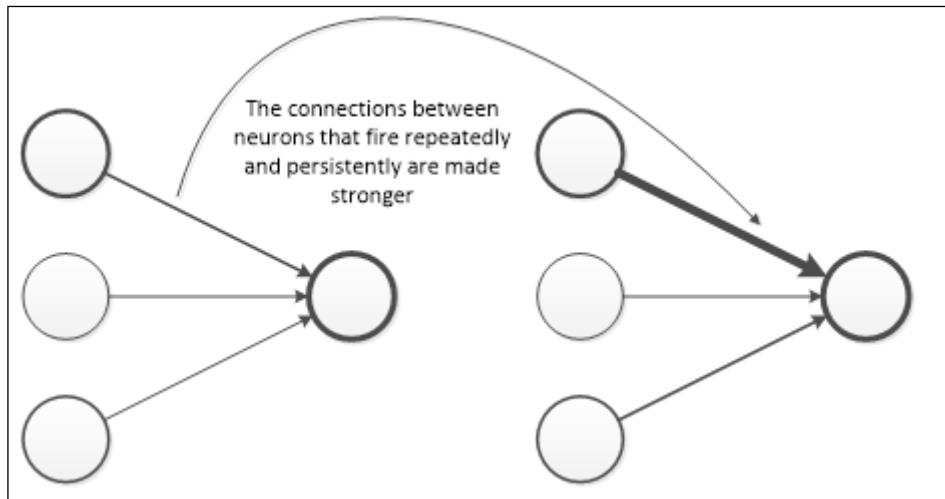


Note: For the batch mode the call to the `derivativeBatch()`, that receives and returns an array of values, instead of a single scalar.

In the `train()` method, we've seen that new weights are stored in the `newWeights` attribute, to not influence the current learning process, and are only applied after the training iteration has finished.

Another learning algorithm - Hebbian learning

In the 1940s, the neuropsychologist Donald Hebb postulated that the connections between neurons that activate or fire simultaneously, or using his words, repeatedly or persistently, should be increased. This is one approach of unsupervised learning, since no target output is specified for Hebbian learning:



In summary, the weight update rule for Hebbian learning takes into account only the input and outputs of the neuron. Given a neuron j whose connection to neuron i (weight w_{ij}) is to be updated, the update is given by the following equation:

$$\Delta w_{ij} = \alpha o_j o_i$$

Here, α is a learning rate, o_j is the output of the neuron j , and o_i is the output of the neuron i , also the input i for the neuron j . For the batch training case, o_i and o_j will be vectors, and we'll need to perform a dot product.

Since we don't include error measurement in Hebbian learning, a stop condition can be determined by the maximum number of epochs or the increase in the overall average of neural outputs. Given N records, we compute the expectancy or average of all outputs produced by the neural network. When this average increases over a certain level, it is time to stop the training, to prevent the neural outputs from blowing up.

We'll develop a new class for Hebbian learning, also inheriting from LearningAlgorithm:

```
public class Hebbian extends LearningAlgorithm {  
    //...  
    private ArrayList<ArrayList<ArrayList<Double>>> newWeights;  
    private ArrayList<Double> currentOutputMean;  
    private ArrayList<Double> lastOutputMean;  
}
```

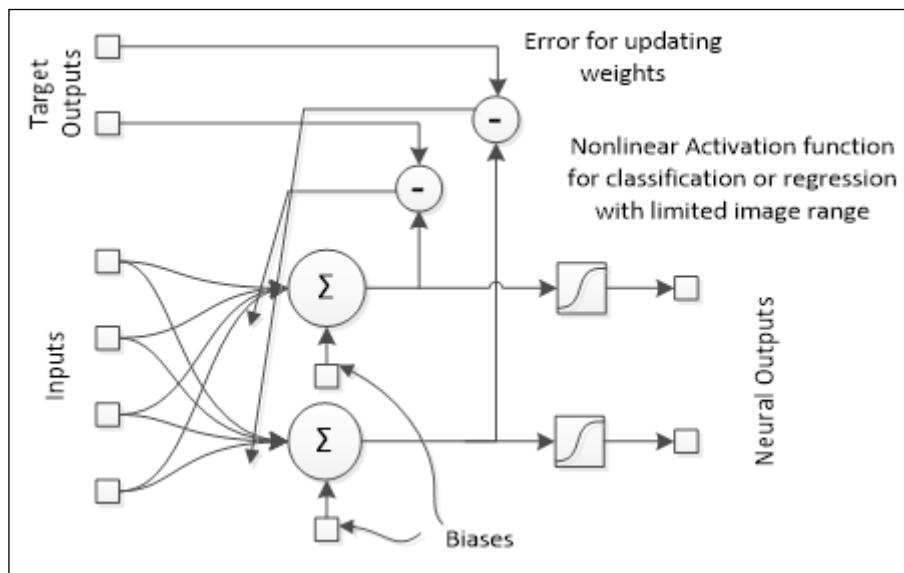
All parameters except for the absent error measures and the new measures of mean are identical to the DeltaRule class. The methods are quite similar, except for the calcNewWeight():

```
@Override  
public Double calcNewWeight(int layer,int input,int neuron)  
    throws NeuralException{  
    //...  
    Double deltaWeight=LearningRate;  
    Neuron currNeuron=neuralNet.getOutputLayer().getNeuron(neuron);  
    switch(learningMode){  
        case BATCH:  
        //...  
        //the batch case is analogous to the implementation in Delta Rule  
        //but with the neuron's output instead of the error  
        //we're suppressing here to save space  
        break;  
        case ONLINE:  
            deltaWeight*=currNeuron.getOutput();  
            if(input<currNeuron.getNumberOfInputs()){  
                deltaWeight*=neuralNet.getInput(input);  
            }  
            break;  
    }  
    return currNeuron.getWeight(input)+deltaWeight;  
}
```

Adaline

Adaline is an architecture standing for Adaptive Linear Neuron, developed by Bernard Widrow and Ted Hoff, based on the McCulloch, and Pitts neuron. It has only one layer of neurons and can be trained similarly to the delta rule. The main difference lies in the fact that the update rule is given by the error between the weighted sum of inputs and biases and the target output, instead of updating based on the neuron output after the activation function. This may be desirable when one wants to perform continuous learning for classification problems, which tend to use discrete values instead of continuous.

The following figure illustrates how Adaline learns:



So the weights are updated by the following equation:

$$\Delta w_{ij} = \alpha(y_j - (\sum_i^n x_i w_{ij} + b))x_i$$

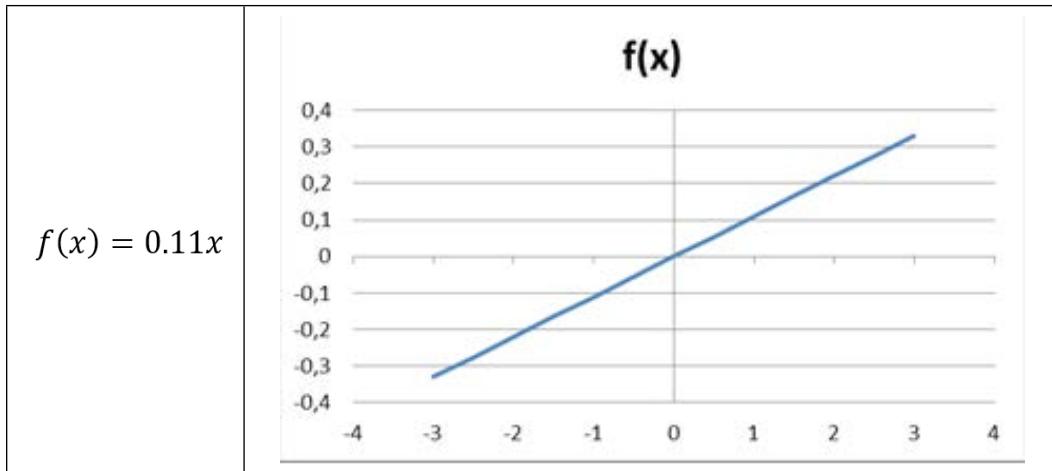
In order to implement Adaline, we create a class called **Adaline** with the following overridden weight calcNewWeight. To save space, we're presenting only the online case:

```
@Override  
public Double calcNewWeight(int layer,int input,int neuron)  
    throws NeuralException{  
//...  
    Double deltaWeight=LearningRate;  
    Neuron currNeuron=neuralNet.getOutputLayer().getNeuron(neuron);  
    switch(learningMode){  
        case BATCH:  
//...  
        break;  
        case ONLINE:  
            deltaWeight*=error.get(currentRecord).get(neuron)  
                *currNeuron.getOutputBeforeActivation();  
            if(input<currNeuron.getNumberOfInputs()){  
                deltaWeight*=neuralNet.getInput(input);  
            }  
            break;  
    }  
    return currNeuron.getWeight(input)+deltaWeight;  
}
```

Note the method `getOutputBeforeActivation()`; we mentioned in the last chapter that this property would be useful in the future.

Time to see the learning in practice!

Let's work on a very simple yet illustrative example. Suppose you want a single neuron neural network to learn how to fit a simple linear function such as the following:



This is quite easy even for those who have little math background, so guess what? It is a nice start for our simplest neural network to prove its learning ability!

Teaching the neural network – the training dataset

We're going to structure the dataset for the neural network to learn using the following code, which you can find in the main method of the file NeuralNetDeltaRuleTest:

```
Double[][] _neuralDataSet = {
    {1.2 , fncTest(1.2)}
,   {0.3 , fncTest(0.3)}
,   {-0.5 , fncTest(-0.5)}
,   {-2.3 , fncTest(-2.3)}
,   {1.7 , fncTest(1.7)}
,   {-0.1 , fncTest(-0.1)}
,   {-2.7 , fncTest(-2.7)}  };
int[] inputColumns = {0};
int[] outputColumns = {1};
NeuralDataSet neuralDataSet = new
    NeuralDataSet(_neuralDataSet,inputColumns,outputColumns);
```

The `funcTest` function is defined as the function we mentioned:

```
public static double funcTest(double x) {  
    return 0.11*x;  
}
```

Note that we're using the class `NeuralDataSet` to structure all this data in such a way that they will be fed into the neural network the right way. Now let's link this dataset to the neural network. Remember that this network has a single neuron in the output. Let's use a nonlinear activation function such as hyperbolic tangent at the output with a coefficient 0.85:

```
int numberOfInputs=1;  
int numberOfOutputs=1;  
HyperTan htAcFnc = new HyperTan(0.85);  
NeuralNet nn = new NeuralNet(numberOfInputs,numberOfOutputs,  
htAcFnc);
```

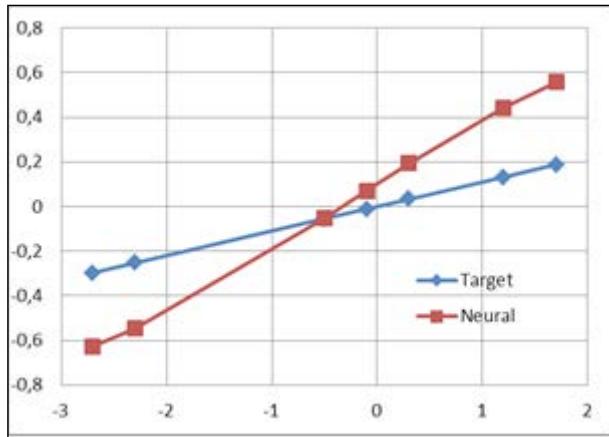
Let's now instantiate the `DeltaRule` object and link it to the neural network created. Then we'll set the learning parameters such as learning rate, minimum overall error, and maximum number of epochs:

```
DeltaRule deltaRule=new DeltaRule(nn,neuralDataSet  
.LearningAlgorithm.LearningMode.ONLINE);  
deltaRule.printTraining=true;  
deltaRule.setLearningRate(0.3);  
deltaRule.setMaxEpochs(1000);  
deltaRule.setMinOverallError(0.00001);
```

Now let's see the first neural output of the untrained neural network, after calling the method `forward()` of the `deltaRule` object:

```
deltaRule.forward();  
neuralDataSet.printNeuralOutput();  
  
    . . .  
Getting the first output of the neural network  
Neural:  
Neural Output[0]={ 0.44224768996697406}  
Neural Output[1]={ 0.19297592401999314}  
Neural Output[2]={ -0.053052819745845956}  
Neural Output[3]={ -0.5457392112113837}  
Neural Output[4]={ 0.5582946870603158}  
Neural Output[5]={ 0.07104196712543694}  
Neural Output[6]={ -0.6270608140997245}
```

Plotting a chart, we find that the output generated by the neural network is a little bit different:



We will start training the neural network in the online mode. We've set the `printTraining` attribute as true, so we will receive in the screen an update. The following piece of code will produce the subsequent screenshot:

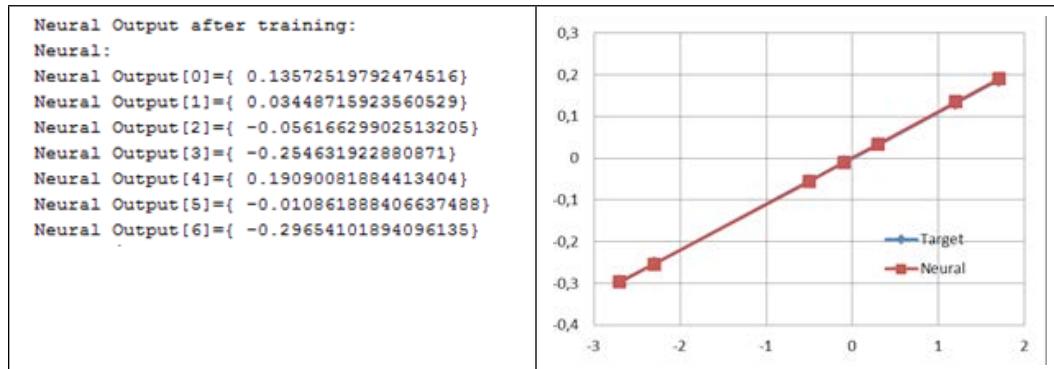
```
System.out.println("Beginning training");
    deltaRule.train();
System.out.println("End of training");
if(deltaRule.getMinOverallError() >=deltaRule
    .getOverallGeneralError()){
    System.out.println("Training succesful!");
}
else{
    System.out.println("Training was unsuccesful");
}
```

```
Beginning training
Epoch=0; Record=0; Overall Error=0.06586810467152877
Epoch=0; Record=1; Overall Error=0.0642007959986327
Epoch=0; Record=2; Overall Error=0.06427692265805388
Epoch=0; Record=3; Overall Error=0.06102840935680022
Epoch=0; Record=4; Overall Error=0.04880202611839414
Epoch=0; Record=5; Overall Error=0.04823124833461199
Epoch=0; Record=6; Overall Error=0.03388528328163059
Epoch=1; Record=0; Overall Error=0.021528920861072397
```

The training begins and the overall error information is updated after every weight update. Note the error is decreasing:

```
Epoch=4; Record=4; Overall Error=1.805118258834097E-5
Epoch=4; Record=5; Overall Error=1.6070522271176285E-5
Epoch=4; Record=6; Overall Error=1.3418163790928809E-5
Epoch=5; Record=0; Overall Error=8.344050623786828E-6
End of training
Training successful!
Overall Error:8.344050623786828E-6
Min Overall Error:1.0E-5
Epochs of training:5
```

After five epochs, the error reaches the minimum; now let's see the neural outputs and the plot:



Quite amazing, isn't it? The target and the neural output are practically the same.
Now let's take a look at the final weight and bias:

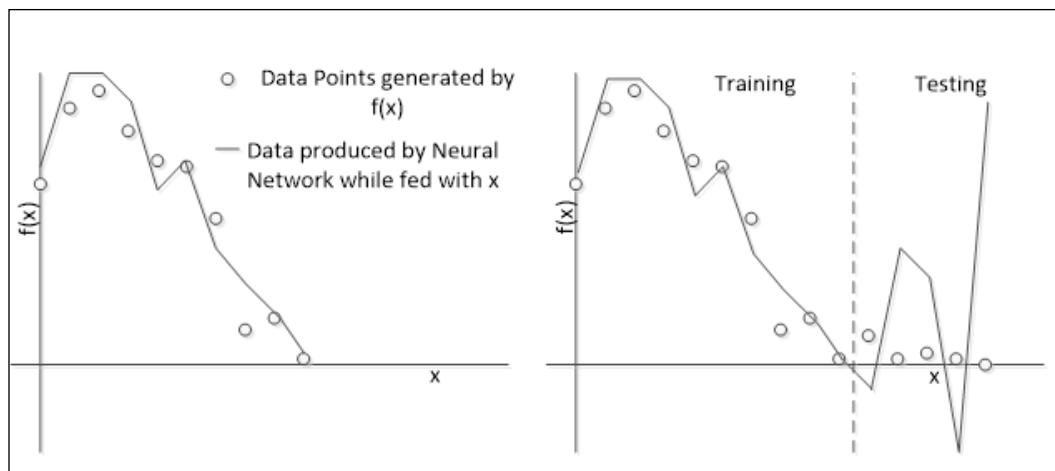
```
weight = nn.getOutputLayer().getWeight(0, 0);
bias = nn.getOutputLayer().getWeight(1, 0);
System.out.println("Weight found:"+String.valueOf(weight));
System.out.println("Bias found:"+String.valueOf(bias));
//Weight found:0.2668421011698528
//Bias found:0.0011258204676042108
```

Amazing, it learned! Or, did it really? A further step – testing

Well, we might ask now: so the neural network has already learned from the data; how can we attest it has effectively learned? Just like in exams students are subjected to, we need to check the network response after training. But wait! Do you think it is likely a teacher would put in an exam the same questions he/she has presented in class? There is no sense in evaluating somebody's learning with examples that are already known, or a suspecting teacher would conclude the student might have memorized the content, instead of having learned it.

Okay, let's now explain this part. What we are talking about here is testing. The learning process we have covered is called training. After training a neural network, we should test whether it has really learnt. For testing, we must present to the neural network another fraction of data from the same environment it has learnt from. This is necessary because, just like the student, the neural network could respond properly to only the data points it has been exposed to; this is called overtraining. To check whether the neural network has not passed on overtraining, we must check its response to other data points.

The following figure illustrates the overtraining problem. Imagine that our network is designed to approximate some function $f(x)$ whose definition is unknown. The neural network was fed with some data from that function and produced the result shown on the left in the following figure. But when expanding to a wider domain, for example, adding a testing dataset, we note the neural response does not follow the data (on the right in the figure):



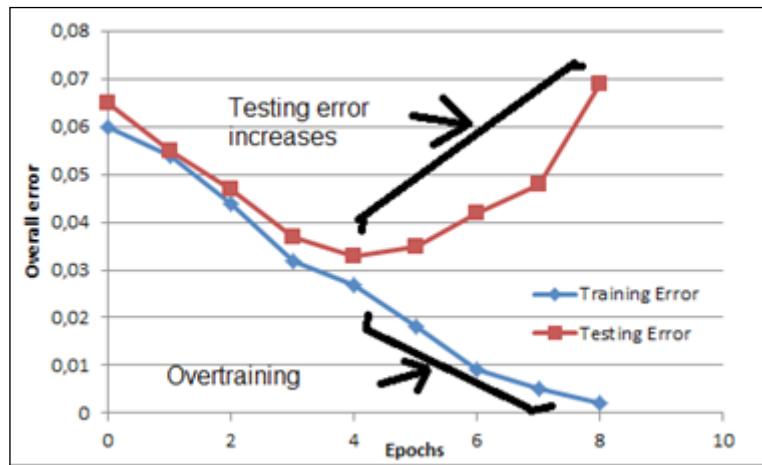
In this case, we see that the neural network failed to learn the whole environment (the function $f(x)$). This happens because of a number of reasons:

- The neural network didn't receive enough information from the environment
- The data from the environment is nondeterministic
- The training and testing datasets are poorly defined
- The neural network has learnt so much on the training data, it has *forgotten* about the testing data

Throughout this book, we are going to cover the process to prevent this and other issues that may arise during training.

Overfitting and overtraining

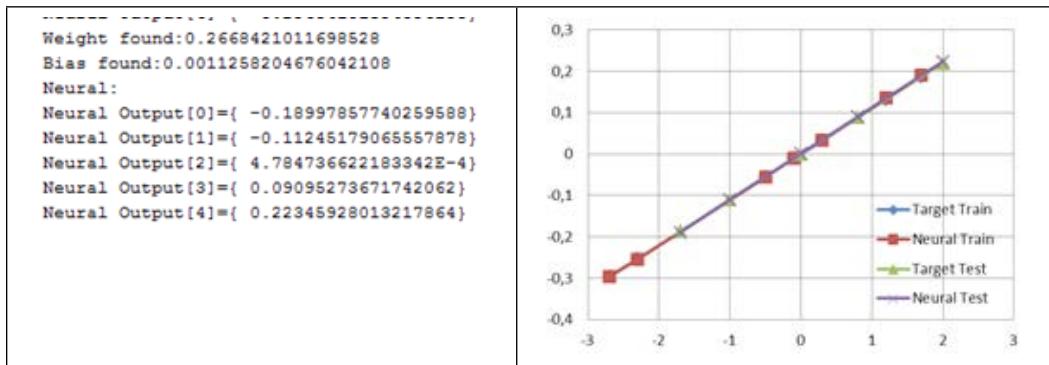
In our previous example, the neural network seemed to have learned amazingly well. However, there is a risk of overfitting and overtraining. The difference between these two concepts is very subtle. An overfitting occurs when the neural network memorizes the problem's behavior, so that it can provide good values only on training points, therefore losing a generalization capacity. Overtraining, which can be a cause for overfitting, occurs when the training error becomes much smaller than the testing error, or actually, the testing error starts to increase as the neural network continues (over)training:



One of ways to prevent overtraining and overfitting is checking the testing error when the training goes on. When the testing error starts to increase, it is time to stop. This will be covered more in detail in the next chapters.

Now, let's see if there is the case in our example. Let's now add some more data and test it:

```
Double[][] _testDataSet = {
    {-1.7 , fncTest(-1.7) }
, {-1.0 , fncTest(-1.0) }
, {0.0 , fncTest(0.0) }
, {0.8 , fncTest(0.8) }
, {2.0 , fncTest(2.0) }
};
NeuralDataSet testDataSet = new NeuralDataSet(_testDataSet,
....inputColumns, outputColumns);
deltaRule.setTestingDataSet(testDataSet);
deltaRule.test();
testDataSet.printNeuralOutput();
```



As can be seen, the neural network presents a generalization capacity in this case. In spite of the simplicity of this example, we can still see the learning skill of the neural network.

Summary

This chapter presented the reader with the whole learning process of which neural networks are capable. We presented the very basic foundations of learning, inspired by human learning itself. To illustrate this process in practice, we have implemented two learning algorithms in Java, and applied them in two examples. With this, the reader can have a basic but useful understanding on how neural networks learn and even how one can systematically describe the process of learning. This will be the foundation for the next chapters, which will present more complex examples.

3

Perceptrons and Supervised Learning

In this chapter, we are going to explore in more detail supervised learning, which is very useful in finding relations between two datasets. Also, we introduce perceptrons, a very popular neural network architecture that implements supervised learning. This chapter also presents their extended generalized version, the so-called multi-layer perceptrons, as well as their features, learning algorithms, and parameters. Also, the reader will learn how to implement them in Java and how to use them in solving some basic problems. This chapter will cover the following topics:

- Supervised learning
- Regression tasks
- Classification tasks
- Perceptrons
- Linear separation
- Limitations: the XOR problem
- Multilayer perceptrons
- Generalized delta rule – backpropagation algorithm
- Levenberg–Marquardt algorithm
- Single hidden layer neural networks
- Extreme learning machines

Supervised learning – teaching the neural net

In the previous chapter, we introduced the learning paradigms that apply to neural networks, where supervised learning implies that there is a goal or a defined target to reach. In practice, we present a set of input data X , and a set of desired output data Y_T , then we evaluate a cost function whose aim is to reduce the error between the neural output Y and the target output Y_T .

In supervised learning, there are two major categories of tasks involved, which are detailed as follows: classification and regression.

Classification – finding the appropriate class

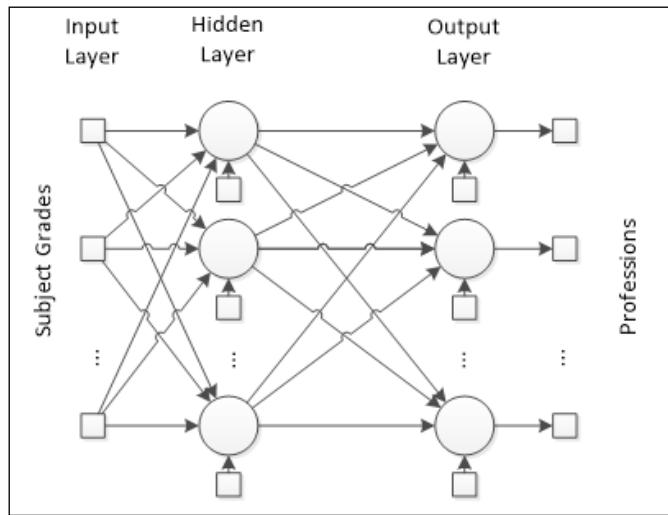
Neural networks also work with categorical data. Given a list of classes and a dataset, one wishes to classify them according to a historical dataset containing records and their respective class. The following table shows an example of this dataset, considering the subjects' average grades between 0 and 10:

Student Id	Subjects								Profession
	English	Math	Physics	Chemistry	Geography	History	Literature	Biology	
89543	7.82	8.82	8.35	7.45	6.55	6.39	5.90	7.03	Electrical Engineer
93201	8.33	6.75	8.01	6.98	7.95	7.76	6.98	6.84	Marketer
95481	7.76	7.17	8.39	8.64	8.22	7.86	7.07	9.06	Doctor
94105	8.25	7.54	7.34	7.65	8.65	8.10	8.40	7.44	Lawyer
96305	8.05	6.75	6.54	7.20	7.96	7.54	8.01	7.86	School Principal
92904	6.95	8.85	9.10	7.54	7.50	6.65	5.86	6.76	Programmer

One example is the prediction of profession based on scholar grades. Let's consider a dataset of former students who are now working. We compile a data set containing each student's average grade on each subject and his/her current profession. Note that the output would be the name of professions, which neural networks are not able to give directly. Instead, we need to make one column (one output) for each known profession. If that student chose a certain profession, the column corresponding to that profession would have the value one, otherwise it would be zero:

$$\begin{array}{cc}
 \text{Subjects' Grades} & \text{Professions} \\
 \left(\begin{array}{ccc|cc}
 7.82 & \dots & 7.03 & 1 & \dots & 0 \\
 \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
 5.66 & \dots & 6.22 & 0 & \dots & 1
 \end{array} \right)
 \end{array}$$

Now we want to find a model - based on a neural network - to predict which profession a student will be likely to choose based on his/her grades. To that end, we structure a neural network containing the number of scholar subjects as the input and the number of known professions as the output, and an arbitrary number of hidden neurons in the hidden layer:



For classification problems, there is usually only one class for each data point. So in the output layer, the neurons are fired to produce either zero or one, it being better to use activation functions that are output bounded between these two values. However, we must consider the case in which more than one neuron fires, giving two classes for a record. There are a number of mechanisms to prevent this case, such as the softmax function or the winner-takes-all algorithm, for example. These mechanisms are going to be detailed in the practical application in *Chapter 6, Classifying Disease Diagnosis*.

After being trained, the neural network has learned what will be the most probable profession for a given student given his/her grades.

Regression – mapping real inputs to outputs

Regression consists in finding some function that maps a set of inputs to a set of outputs. The following table shows how a dataset containing k records of m independent inputs X are known to be bound to n dependent outputs:

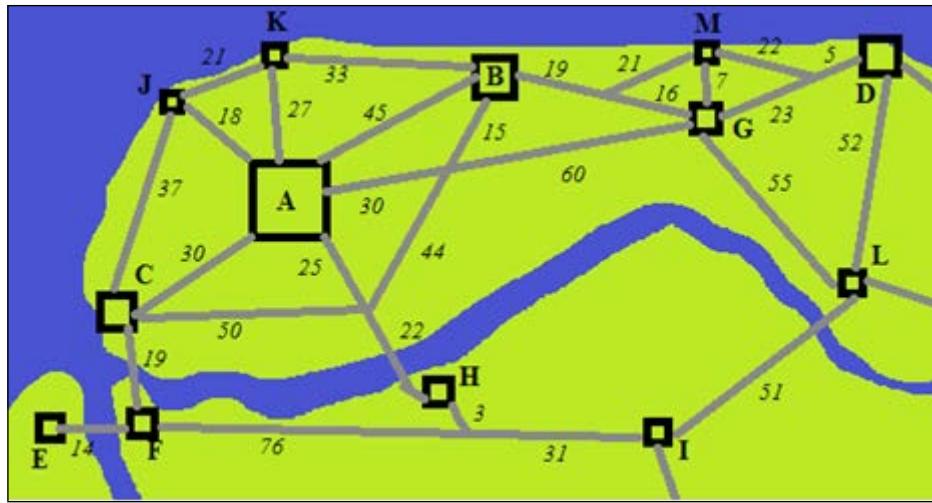
Input independent data				Output dependent data			
X1	X2	...	XM	T1	T2	...	TN
x1[0]	x2[0]	...	xm[0]	t1[0]	t2[0]	...	tn[0]
x1[1]	x2[1]	...	xm[1]	t1[1]	t2[1]	...	tn[1]
...
x1[k]	x2[k]	...	xm[k]	t1[k]	t2[k]	...	tn[k]

The preceding table can be compiled in matrix format:

$$X_{k,m} = \begin{pmatrix} x_1(0) & x_2(0) & \cdots & x_m(0) \\ x_1(1) & x_2(1) & \cdots & x_m(1) \\ \vdots & \vdots & \ddots & \vdots \\ x_1(k) & x_2(k) & \cdots & x_m(k) \end{pmatrix}$$

$$T_{k,n} = \begin{pmatrix} t_1[0] & t_2[0] & \cdots & t_n[0] \\ t_1[1] & t_2[1] & \cdots & t_n[1] \\ \vdots & \vdots & \ddots & \vdots \\ t_1[k] & t_2[k] & \cdots & t_n[k] \end{pmatrix}$$

Unlike the classification, the output values are numerical instead of labels or classes. There is also a historical database containing records of some behavior we would like the neural network to learn. One example is the prediction of bus ticket prices between two cities. In this example, we collect information from a list of cities and the current ticket prices of buses departing from one and arriving to another. We structure the city features as well as the distance and/or time between them as the input and the bus ticket price as the output:



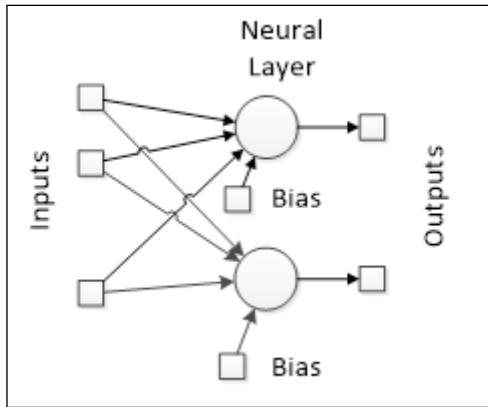
Features city of origin			Features city of destination			Features of the way between			Ticket fare
Population	GDP	Routes	Population	GDP	Routes	Distance	Time	Stops	
500,000	4.5	6	45,000	1.5	5	90	1,5	0	15
120,000	2.6	4	500,000	4.5	6	30	0,8	0	10
30,000	0.8	3	65,000	3.0	3	103	1,6	1	20
35,000	1.4	3	45,000	1.5	5	7	0.4	0	5
...									
120,000	2.6	4	12,000	0.3	3	37	0.6	0	7

Having structured the dataset, we define a neural network containing the exact number of features (multiplied by two, provided two cities) plus the route features in the input, one output, and an arbitrary number of neurons in the hidden layer. In the case presented in the preceding table, there would be nine inputs. Since the output is numerical, there is no need to convert output data.

This neural network would give an estimate price for a route between two cities, which currently is not served by any bus transportation company.

A basic neural architecture – perceptrons

Perceptron is the most simple neural network architecture. Projected by *Frank Rosenblatt* in 1957, it has just one layer of neurons, receiving a set of inputs and producing another set of outputs. This was one of the first representations of neural networks to gain attention, especially because of their simplicity:



In our Java implementation, this is illustrated with one neural layer (the output layer). The following code creates a perceptron with three inputs and two outputs, having the linear function at the output layer:

```
int numberOfInputs=3;
int numberOfOutputs=2;

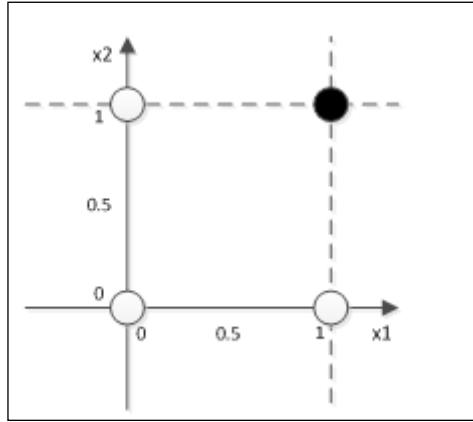
Linear outputAcFnc = new Linear(1.0);
NeuralNet perceptron = new NeuralNet(numberOfInputs,numberOfOutputs,
                                     outputAcFnc);
```

Applications and limitations

However, scientists did not take long to conclude that a perceptron neural network could only be applied to simple tasks, according to that simplicity. At that time, neural networks were being used for simple classification problems, but perceptrons usually failed when faced with more complex datasets. Let's illustrate this with a very basic example (an AND function) to understand better this issue.

Linear separation

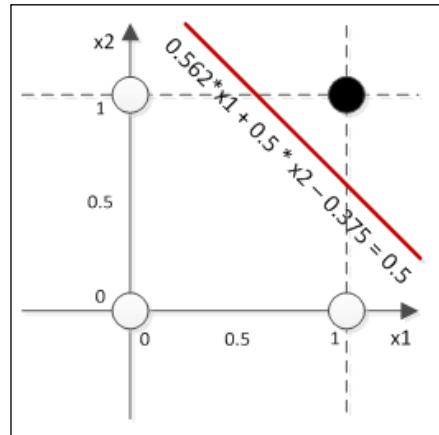
The example consists of an AND function that takes two inputs, x_1 and x_2 . That function can be plotted in a two-dimensional chart as follows:



And now let's examine how the neural network evolves the training using the perceptron rule, considering a pair of two weights, w_1 and w_2 , initially **0.5**, and bias valued **0.5** as well. Assume learning rate η equals **0.2**:

Epoch	x_1	x_2	w_1	w_2	b	y	t	E	Δw_1	Δw_2	Δb
1	0	0	0.5	0.5	0.5	0.5	0	-0.5	0	0	-0.1
1	0	1	0.5	0.5	0.4	0.9	0	-0.9	0	-0.18	-0.18
1	1	0	0.5	0.32	0.22	0.72	0	-0.72	-0.144	0	-0.144
1	1	1	0.356	0.32	0.076	0.752	1	0.248	0.0496	0.0496	0.0496
2	0	0	0.406	0.370	0.126	0.126	0	-0.126	0.000	0.000	-0.025
2	0	1	0.406	0.370	0.100	0.470	0	-0.470	0.000	-0.094	-0.094
2	1	0	0.406	0.276	0.006	0.412	0	-0.412	-0.082	0.000	-0.082
2	1	1	0.323	0.276	-0.076	0.523	1	0.477	0.095	0.095	0.095
...	...										
89	0	0	0.625	0.562	-0.312	-0.312	0	0.312	0	0	0.062
89	0	1	0.625	0.562	-0.25	0.313	0	-0.313	0	-0.063	-0.063
89	1	0	0.625	0.500	-0.312	0.313	0	-0.313	-0.063	0	-0.063
89	1	1	0.562	0.500	-0.375	0.687	1	0.313	0.063	0.063	0.063

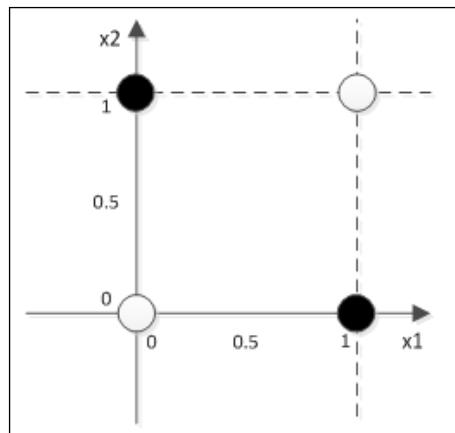
After 89 epochs, we find the network to produce values near to the desired output. Since in this example the outputs are binary (zero or one), we can assume that any value produced by the network that is below 0.5 is considered to be 0 and any value above 0.5 is considered to be 1. So, we can draw a function $Y = x_1 w_1 + x_2 w_2 + b = 0.5$, with the final weights and bias found by the learning algorithm $w_1=0.562$, $w_2=0.5$ and $b=-0.375$, defining the linear boundary in the chart:



This boundary is a definition of all classifications given by the network. You can see that the boundary is linear, given that the function is also linear. Thus, the perceptron network is really suitable for problems whose patterns are linearly separable.

The XOR case

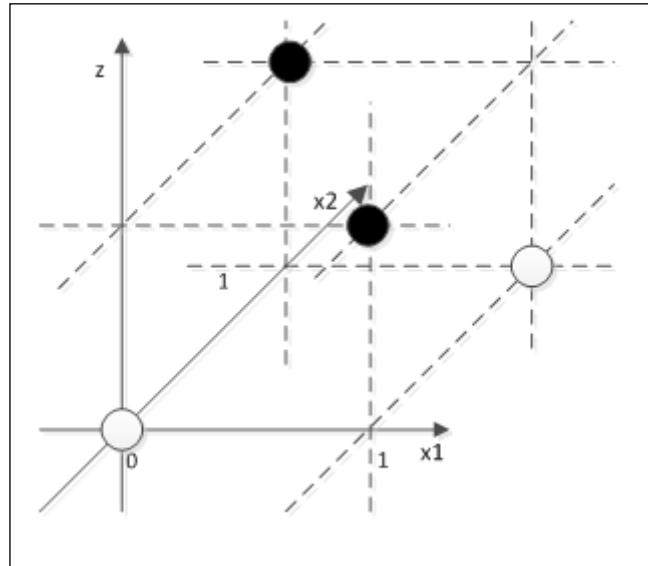
Now let's analyze the XOR case:



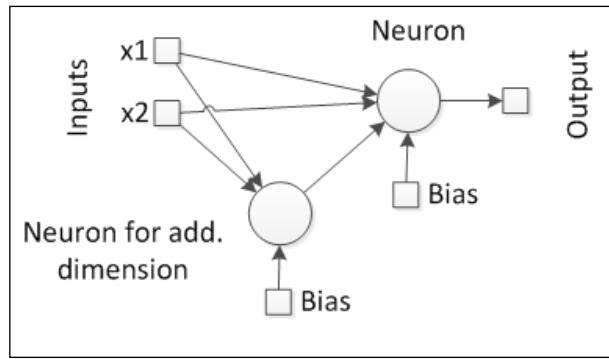
We see that in two dimensions, it is impossible to draw a line to separate the two patterns. What would happen if we tried to train a single layer perceptron to learn this function? Suppose we tried, let's see what happened in the following table:

Epoch	x1	x2	w1	w2	b	y	t	E	Δw_1	Δw_2	Δb
1	0	0	0.5	0.5	0.5	0.5	0	-0.5	0	0	-0.1
1	0	1	0.5	0.5	0.4	0.9	1	0.1	0	0.02	0.02
1	1	0	0.5	0.52	0.42	0.92	1	0.08	0.016	0	0.016
1	1	1	0.516	0.52	0.436	1.472	0	-1.472	-0.294	-0.294	-0.294
2	0	0	0.222	0.226	0.142	0.142	0	-0.142	0.000	0.000	-0.028
2	0	1	0.222	0.226	0.113	0.339	1	0.661	0.000	0.132	0.132
2	1	0	0.222	0.358	0.246	0.467	1	0.533	0.107	0.000	0.107
2	1	1	0.328	0.358	0.352	1.038	0	-1.038	-0.208	-0.208	-0.208
...	...										
127	0	0	-0.250	-0.125	0.625	0.625	0	-0.625	0.000	0.000	-0.125
127	0	1	-0.250	-0.125	0.500	0.375	1	0.625	0.000	0.125	0.125
127	1	0	-0.250	0.000	0.625	0.375	1	0.625	0.125	0.000	0.125
127	1	1	-0.125	0.000	0.750	0.625	0	-0.625	-0.125	-0.125	-0.125

The perceptron just could not find any pair of weights that would drive the following error 0.625. This can be explained mathematically as we already perceived from the chart that this function cannot be linearly separable in two dimensions. So what if we add another dimension? Let's see the chart in three dimensions:



In three dimensions, it is possible to draw a plane that would separate the patterns, provided that this additional dimension could properly transform the input data. Okay, but now there is an additional problem: how could we derive this additional dimension since we have only two input variables? One obvious, but also workaround, answer would be adding a third variable as a derivation from the two original ones. And being this third variable a (derivation), our neural network would probably get the following shape:

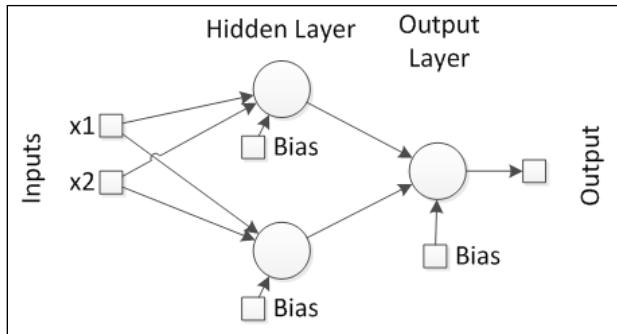


Okay, now the perceptron has three inputs, one of them being a composition of the other. This also leads to a new question: how should that composition be processed? We can see that this component could act as a neuron, so giving the neural network a nested architecture. If so, there would another new question: how would the weights of this new neuron be trained, since the error is on the output neuron?

Multi-layer perceptrons

As we can see, one simple example in which the patterns are not linearly separable has led us to more and more issue using the perceptron architecture. That need led to the application of multilayer perceptrons. In *Chapter 1, Getting Started with Neural Networks* we dealt with the fact that the natural neural network is structured in layers as well, and each layer captures pieces of information from a specific environment. In artificial neural networks, layers of neurons act in this way, by extracting and abstracting information from data, transforming them into another dimension or shape.

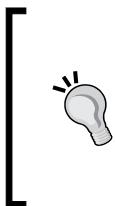
In the XOR example, we found the solution to be the addition of a third component that would make possible a linear separation. But there remained a few questions regarding how that third component would be computed. Now let's consider the same solution as a two-layer perceptron:



Now we have three neurons instead of just one, but in the output the information transferred by the previous layer is transformed into another dimension or shape, whereby it would be theoretically possible to establish a linear boundary on those data points. However, the question on finding the weights for the first layer remains unanswered, or can we apply the same training rule to neurons other than the output? We are going to deal with this issue in the Generalized delta rule section.

MLP properties

Multi-layer perceptrons can have any number of layers and also any number of neurons in each layer. The activation functions may be different on any layer. An MLP network is usually composed of at least two layers, one for the output and one hidden layer.



There are also some references that consider the input layer as the nodes that collect input data; therefore, for those cases, the MLP is considered to have at least three layers. For the purpose of this book, let's consider the input layer as a special type of layer which has no weights, and as the effective layers, that is, those enabled to be trained, we'll consider the hidden and output layers.

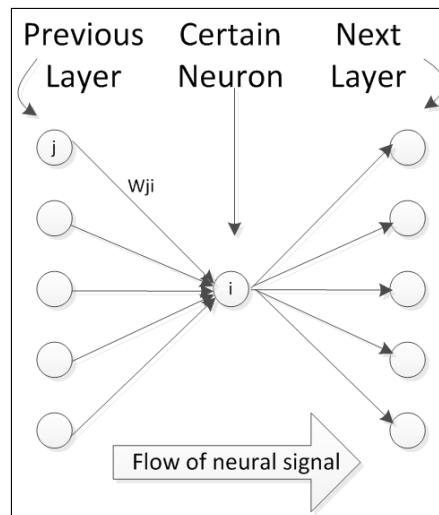
A hidden layer is called that because it actually hides its outputs from the external world. Hidden layers can be connected in series in any number, thus forming a deep neural network. However, the more layers a neural network has, the slower would be both training and running, and according to mathematical foundations, a neural network with one or two hidden layers at most may learn as well as deep neural networks with dozens of hidden layers. But it depends on several factors.



It is really recommended for the activation functions to be nonlinear in the hidden layers, especially if in the output layer the activation function is linear. According to linear algebra, having a linear activation function in all layers is equivalent to having only one output layer, provided that the additional variables introduced by the layers would be mere linear combinations of the previous ones or the inputs. Usually, activation functions such as hyperbolic tangent or sigmoid are used, because they are derivable.

MLP weights

In an MLP feedforward network, one particular neuron i receives data from a neuron j of the previous layer and forwards its output to a neuron k of the next layer:



The mathematical description of a neural network is recursive:

$$y_o = f_o \left(\sum_{i=1}^{n_{h_l}} w_i f_i \left(\sum_{j=1}^{n_{h_{l-1}}} w_{ij} f_j \left(\sum_{k=1}^{n_{h_{l-2}}} w_{jk} f_k (\dots) + b_j \right) + b_i \right) + b_o \right)$$

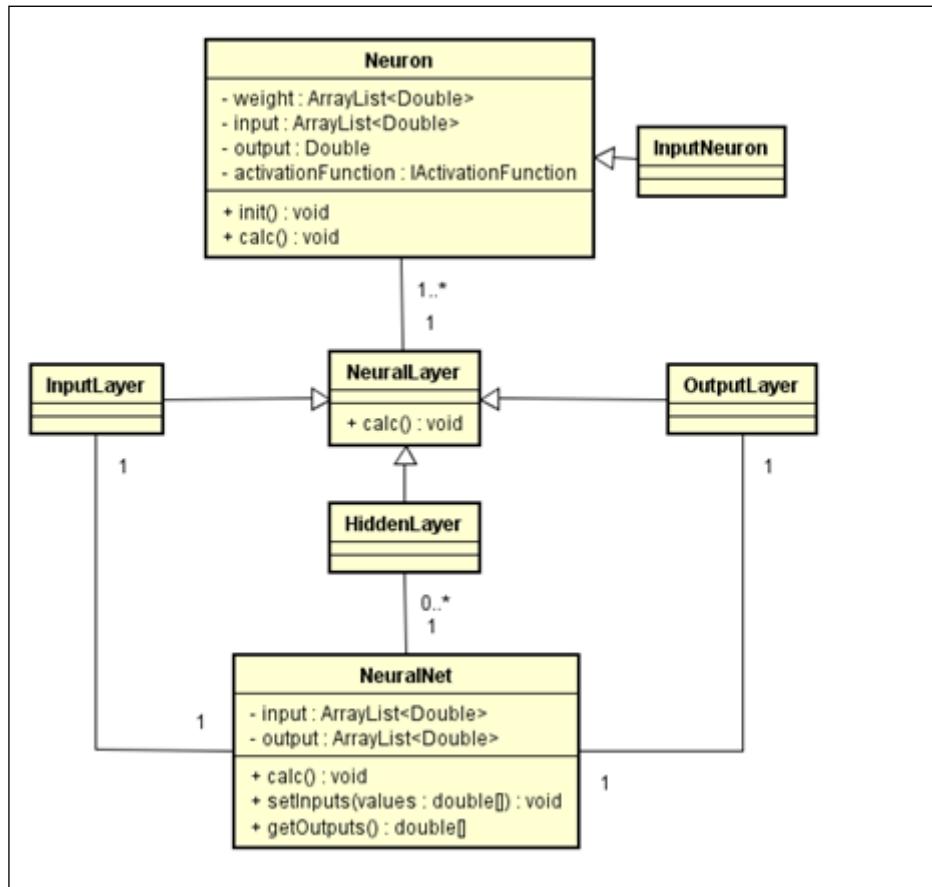
Here, y_o is the network output (should we have multiple outputs, we can replace y_o with \mathbf{Y} , representing a vector); f_o is the activation function of the output; l is the number of hidden layers; n_{hi} is the number of neurons in the hidden layer i ; w_i is the weight connecting the i th neuron of the last hidden layer to the output; f_i is the activation function of the neuron i ; and b_i is the bias of the neuron i . It can be seen that this equation gets larger as the number of layers increases. In the last summing operation, there will be the inputs x_i .

Recurrent MLP

The neurons on an MLP may feed signals not only to neurons in the next layers (feedforward network), but also to neurons in the same or previous layers (feedback or recurrent). This behavior allows the neural network to maintain state on some data sequence, and this feature is especially exploited when dealing with time series or handwriting recognition. Recurrent networks are usually harder to train, and eventually the computer may run out of memory while executing them. In addition, there are recurrent network architectures better than MLPs, such as Elman, Hopfield, Echo state, Bidirectional RNNs (recurrent neural networks). But we are not going to dive deep into these architectures, because this book focuses on the simplest applications for those who have minimal experience in programming. However, we recommend good literature on recurrent networks for those who are interested in it.

Coding an MLP

Bringing these concepts into the OOP point of view, we can review the classes already designed so far:



One can see that the neural network structure is hierarchical. A neural network is composed of layers that are composed of neurons. In the MLP architecture, there are three types of layers: input, hidden, and output. So suppose that in Java, we would like to define a neural network consisting of three inputs, one output (linear activation function) and one hidden layer (sigmoid function) containing five neurons. The resulting code would be as follows:

```

int numberOfInputs=3;
int numberOfOutputs=1;
  
```

```

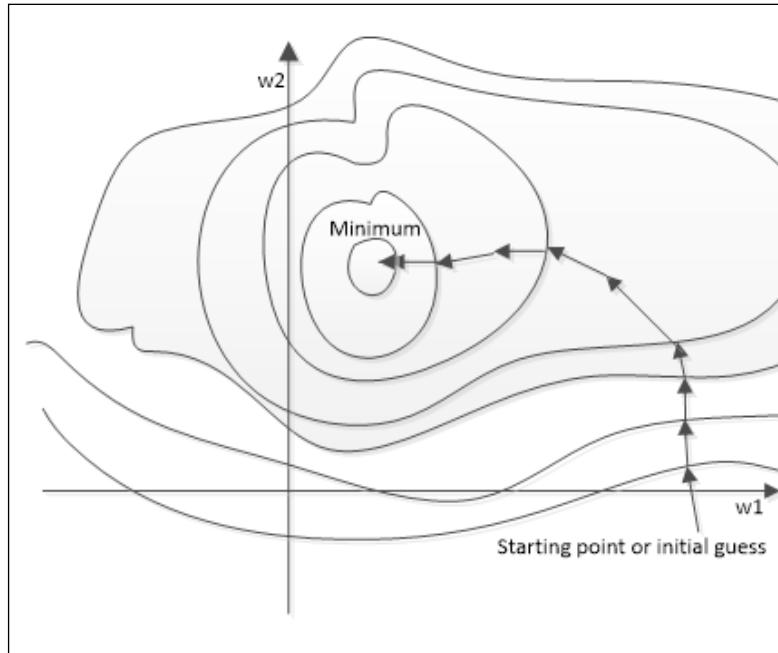
int [] numberOfHiddenNeurons={5};

Linear outputAcFnc = new Linear(1.0);
Sigmoid hiddenAcFnc = new Sigmoid(1.0);
NeuralNet neuralnet = new NeuralNet(numberOfInputs, numberOfOutputs,
numberOfHiddenNeurons, hiddenAcFnc, outputAcFnc);

```

Learning in MLPs

The multi-layer perceptron network learns based on the Delta Rule, which is also inspired by the gradient descent optimization method. The gradient method is broadly applied to find minima or maxima of a given function:



This method is applied at *walking* the direction where the function's output is higher or lower, depending on the criteria. This concept is explored in the Delta Rule:

$$\Delta w_i = \eta(t(k) - y(k)) x_i[k] g'(h_i(k))$$

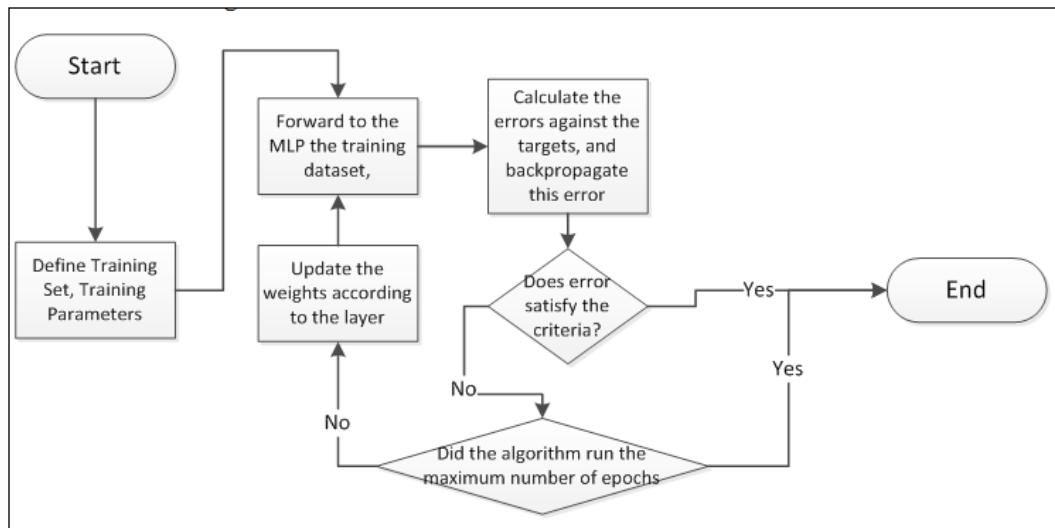
The function that the Delta Rule wants to minimize is the error between the neural network output and the target output, and the parameters to be found are the neural weights. This is an enhanced learning algorithm compared to the perceptron rule, because it takes into account the activation function derivative $g'(h)$, which in mathematical terms indicates the direction where the function is decreasing most.

Backpropagation algorithm

Although the Delta Rule works well for the neural networks having only output and input layers, for the MLP networks, the pure Delta Rule cannot be applied because of the hidden layer neurons. To overcome this issue, in the 1980s, *Rummelhart* and others proposed a new algorithm, also inspired by the gradient method, called backpropagation.

This algorithm is indeed a generalization of the Delta Rule for MLPs. The benefits of having additional layers to abstract more data from the environment have motivated the development of a training algorithm that could properly adjust the weights of the hidden layer. Based on the gradient method, the error from output would be (back) propagated to the previous layers, so making possible the weight update using the same equation as the Delta Rule.

The algorithm runs as follows:



The second step is the backpropagation itself. What it does is to find the weight variation according to the gradient, which is the base for the Delta Rule:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial o_i} \frac{\partial o_i}{\partial h_i} \frac{\partial h_i}{\partial w_{ji}} = (t - y) f'(h_i) x_j$$

Here, E is the error, w_{ji} is the weight between the neurons i and j , o_i is the output of the i th neuron, and h_i is the weighted sum of that neuron's inputs before passing to activation function. Remember that $o_i=f(h_i)$, f being the activation function.

For updating in the hidden layers, it is a bit more complicated as we consider the error as function of all neurons between the weight to be updated and the output. To facilitate this process, we should compute the sensibility or backpropagation error:

$$\delta_i = \frac{\partial E}{\partial o_i} \frac{\partial o_i}{\partial h_i}$$

And the weight update is as follows:

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}} = \eta \delta_i x_j$$

The calculation of the backpropagation error varies for the output and for the hidden layers:

- Backpropagation for the output layer:

$$\delta_i = (o_i - t_i) f'(h_i)$$

- Here, o_i is the i th output, t_i is the desired i th output, $f'(h_i)$ is the derivative of the output activation function, and h_i is the weighted sum of the i th neuron inputs

- Backpropagation for the hidden layer:

$$\delta_i = \sum_l \delta_l w_{il} f'(h_i)$$

- Here, l is a neuron of the layer ahead and w_{il} is the weight that connects the current neuron to the l th neuron of the layer immediately ahead.

For simplicity reasons, we did not demonstrate fully how the backpropagation equation was developed. Anyway, if you are interested in the details, you may consult the book neural networks – a comprehensive foundation by Simon Haykin.

The momentum

Like any gradient-based method, there is a risk of falling into a local minimum. To mitigate this risk, we can add another term to the weight update rule called momentum, which takes into consideration the last variation of weight:

$$w_{ji}(k+1) = w_{ji}(k) + \Delta w_{ji}(k) + \mu \Delta w_{ji}(k-1)$$

Here, μ is a momentum rate and $\Delta w_{ji}(k-1)$ is the last delta weight. This gives an additional step to the update, therefore attenuating the oscillations in the error hyperspace.

Coding the backpropagation

Let's define the class backpropagation in the package `edu.packt.neural.learn`. Since this learning algorithm is a generalization of the `DeltaRule`, this class may inherit and override the features already defined in Delta Rule. Three additional attributes included in this class are the momentum rate, the delta neuron, and the last delta weight arrays:

```
public class Backpropagation extends DeltaRule {  
    private double MomentumRate=0.7;  
    public ArrayList<ArrayList<Double>> deltaNeuron;  
    public ArrayList<ArrayList<ArrayList<Double>>> lastDeltaWeights;  
    ...  
}
```

The constructor will have the same arguments as for the `DeltaRule` class, adding the calls to methods for initialization of the `deltaNeuron` and `lastDeltaWeights` arrays:

```
public Backpropagation(NeuralNet _neuralNet, NeuralDataSet _trainDataSet, DeltaRule.LearningMode _learningMode) {
    super(_neuralNet,_trainDataSet,_learningMode);
    initializeDeltaNeuron();
    initializeLastDeltaWeights();
}
```

The `train()` method will work in a similar way as in the `DeltaRule` class; the additional component is the backward step, whereby the error is backpropagated throughout the neural layers up to the input:

```
@Override
public void train() throws NeuralException{
    neuralNet.setNeuralNetMode(NeuralNet.NeuralNetMode.TRAINING);
    epoch=0; // initialization of epoch
    int k=0; // first training record
    currentRecord=0; // this attribute keeps track of which record
                     // is currently under processing in the training
    forward(); // initial forward step to determine the error
    forward(k); // forward for backpropagation of first record error
    while(epoch<MaxEpochs && overallGeneralError>MinOverallError) {
        backward(); // backward step
        switch(learningMode) {
            case BATCH:
                if(k==trainingDataSet.numberOfRecords-1)
                    applyNewWeights(); // batch update
                break;
            case ONLINE:
                applyNewWeights(); //online update
        }
        currentRecord++; // moving on to the next record
        if(k>=trainingDataSet.numberOfRecords){ //if it was the last
            k=0;
            currentRecord=0; // reset to the first
            epoch++; // and increase the epoch
        }
        forward(k); // forward the next record
    }
    neuralNet.setNeuralNetMode(NeuralNet.NeuralNetMode.RUN);
}
```

The role of the backward step is to determine the delta weights by means of error backpropagation, from the output layer down to the first hidden layer:

```
public void backward() {
    int numberofLayers=neuralNet.getNumberOfHiddenLayers();
    for(int l=numberofLayers;l>=0;l--) {
        int numberofNeuronsInLayer=deltaNeuron.get(l).size();
        for(int j=0;j<numberofNeuronsInLayer;j++) {
            for(int i=0;i<newWeights.get(l).get(j).size();i++) {

                // get the current weight of the neuron
                double currNewWeight = this.newWeights.get(l).get(j).get(i);
                //if it is the first epoch, get directly from the neuron
                if(currNewWeight==0.0 && epoch==0.0)
                    if(l==numberofLayers)
                        currNewWeight=neuralNet.getOutputLayer().getWeight(i, j);
                    else
                        currNewWeight=neuralNet.getHiddenLayer(l).
                            getWeight(i, j);
                // calculate the delta weight
                double deltaWeight=calcDeltaWeight(l, i, j);
                // store the new calculated weight for subsequent update
                newWeights.get(l).get(j).set(i,currNewWeight+deltaWeight);
            }
        }
    }
}
```

The backpropagation step is performed in the method `calcDeltaWeight()`. The momentum will be added only before updating the weights because it should recall the last delta weight determined:

```
public Double calcDeltaWeight(int layer,int input,int neuron) {
    Double deltaWeight=1.0;
    NeuralLayer currLayer;
    Neuron currNeuron;
    double _deltaNeuron;
    if(layer==neuralNet.getNumberOfHiddenLayers()) { //output layer
        currLayer=neuralNet.getOutputLayer();
        currNeuron=currLayer.getNeuron(neuron);
        _deltaNeuron=error.get(currentRecord).get(neuron)
                    *currNeuron.derivative(currLayer.getInputs());
    }
    else{ //hidden layer
```

```

currLayer=neuralNet.getHiddenLayer(layer);
currNeuron=currLayer.getNeuron(neuron);
double sumDeltaNextLayer=0;
NeuralLayer nextLayer=currLayer.getNextLayer();
for(int k=0;k<nextLayer.getNumberOfNeuronsInLayer();k++) {
    sumDeltaNextLayer+=nextLayer.getWeight(neuron, k)
        *deltaNeuron.get(layer+1).get(k);
}
_deltaNeuron=sumDeltaNextLayer*
currNeuron.derivative(currLayer.getInputs());
}

deltaNeuron.get(layer).set(neuron, _deltaNeuron);
deltaWeight*=_deltaNeuron;
if(input<currNeuron.getNumberOfInputs()){
    deltaWeight*=currNeuron.getInput(input);
}

return deltaWeight;
}

```

Note the calculation of the `_deltaNeuron` is different for the output and the hidden layers, but for both of them the derivative is used. To facilitate this task, we've added the `derivative()` method to the class `Neuron`. Details can be found in *Annex III* documentation. At the end, the input corresponding to the weight is multiplied to the delta weight calculated.

The weight update is performed by the method `applyNewWeights()`. To save space, we are not going to write here the whole method body, but only the core where the weight update is performed:

```

HiddenLayer hl = this.neuralNet.getHiddenLayer(l);
Double lastDeltaWeight=lastDeltaWeights.get(l).get(j).get(i);
// determine the momentum
double momentum=MomentumRate*lastDeltaWeight;
//the momentum is then added to the new weight
double newWeight=this.newWeights.get(l).get(j).get(i)
    -momentum;
this.newWeights.get(l).get(j).set(i,newWeight);
Neuron n=hl.getNeuron(j);
// save the current delta weight for the next step
double deltaWeight=(newWeight-n.getWeight(i));

```

```
lastDeltaWeights.get(l).get(j).set(i, (double)deltaWeight);
// finally the weight is updated
hl.getNeuron(j).updateWeight(i, newWeight);
```

In the code listing, l represents the layer, j the neuron, and i the input to the weight. For the output layer, l will be equal to the number of hidden layers (exceeding the Java array limits), so the NeuralLayer called is as follows:

```
OutputLayer ol = this.neuralNet.getOutputLayer();
Neuron n=ol.getNeuron(j);
ol.getNeuron(j).updateWeight(i, newWeight);
```

This class can be used exactly the same way as DeltaRule:

```
int numberOfInputs=2;
int numberOfOutputs=1;

int[] numberOfHiddenNeurons={2};

Linear outputAcFnc = new Linear(1.0);
Sigmoid hdAcFnc = new Sigmoid(1.0);
IActivationFunction[] hiddenAcFnc={hdAcFnc };

NeuralNet mlp = new NeuralNet(numberOfInputs,numberOfOutputs
,numberOfHiddenNeurons,hiddenAcFnc,outputAcFnc);

Backpropagation backprop = new Backpropagation(mlp,neuralDataSet
,LearningAlgorithm.LearningMode.ONLINE);
```

At the end of this chapter, we will make a comparison of the Delta Rule for the perceptrons with the backpropagation with a multi-layer perceptron, trying to solve the XOR problem.

Levenberg-Marquardt algorithm

The backpropagation algorithm, like all gradient-based methods, usually presents slow convergence, especially when it falls in a zig-zag situation, when the weights are changed to almost the same value every two iterations. This drawback was studied in problems such as curve-fitting interpolation by Kenneth Levenberg in 1944, and later by Donald Marquart in 1963, who developed a method for finding coefficients based on the Gauss Newton algorithm and the gradient descent, so therefrom comes the name of the algorithm.

The LM algorithm deals with some optimization terms which are beyond the scope of this book, but in the references section, the reader will find good resources to learn more about these concepts, so we will present the method in a simpler way. Let's suppose we have a list of inputs x and outputs t :

$$\begin{pmatrix} x_1(0) & x_2(0) & \cdots & x_m(0)t_1[0] & t_2[0] & \cdots & t_n[0] \\ x_1(1) & x_2(1) & \cdots & x_m(1)t_1[1] & t_2[1] & \cdots & t_n[1] \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ x_1(k) & x_2(k) & \cdots & x_m(k)t_1[k] & t_2[k] & \cdots & t_n[k] \end{pmatrix}$$

We have seen that a neural network has the property to map inputs to outputs just like a nonlinear function f with coefficients W (weights and bias):

$$Y = f(X, W)$$

The nonlinear function will produce values different than the outputs T ; that's because we marked the variable Y in the equation. The Levenberg-Marquardt algorithm works over a Jacobian matrix, which is a matrix of all partial derivatives in regard to each weight and bias for each data row. So the Jacobian matrix has the following format:

$$\Delta W = (J^T J + I) J^T (Y - f(X, W))$$

Here, k is the total number of data points and p is the total number of weights and bias. In the Jacobian matrix, all weights and bias are stored serially in a single row. The elements of the Jacobian Matrix are calculated from the gradients:

$$J = \begin{bmatrix} \frac{\partial f(X[1], W)}{W_1} & \dots & \frac{\partial f(X[1], W)}{W_p} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(X[k], W)}{W_1} & \dots & \frac{\partial f(X[k], W)}{W_p} \end{bmatrix}$$

The partial derivative of the error E in relation to each weight is calculated in the backpropagation algorithm, so this algorithm is going to run the backpropagation step as well.

In every optimization problem, one wishes to minimize the total error:

$$\frac{\partial E}{\partial w_{ji}} = (t - y) \frac{\partial f(x_i, W)}{\partial w_{ji}} \Rightarrow \frac{\partial f(x_i, W)}{\partial w_{ji}} = \frac{\partial E}{\partial w_{ji}} (t - y)^2$$

Here, W (weights and bias in the NN case) are the variables to optimize. The optimization algorithm updates W by adding ΔW . By applying some algebra, the last equation can be extended to this one:

$$E(W) = \sum [Y_i - f(X_i, W)]^2$$

Converting to the vector and notation:

$$E(W + \Delta W) = \sum [Y_i - f(X_i, W) - J_i \Delta W]^2$$

Finally, by setting the error E to zero, we get the Levenberg-Marquardt equation after some manipulation:

$$E(W + \Delta W) = \|Y - f(X, W) - J \Delta W\|^2$$

This is the weight update rule. As can be seen, it involves matrix operations such as transposition and inversion. The Greek letter λ is the damping factor, an equivalent for the learning rate.

Coding the Levenberg-Marquardt with matrix algebra

In order to effectively implement the LM algorithm, it is very useful to work with matrix algebra. To address that, we defined a class called `Matrix` in the package `edu.packt.neuralnet.math`, including all the matrix operations, such as multiplication, inverse, and LU decomposition, among others. The reader may refer to the documentation to find out more about this class.

The Levenberg-Marquardt algorithm uses many features of the backpropagation algorithm; that's why we inherit this class from backpropagation. Some new attributes are included:

- **Jacobian matrix:** This is the matrix containing the partial derivatives to each weight for all training records
- **Damping factor**
- **Error backpropagation:** This array has the same function of `deltaNeuron`, but its calculation differs a little to each neural output; that's why we defined it in a separate attribute
- **Error LMA:** The error in the matrix form:

```
public class LevenbergMarquardt extends Backpropagation {

    private Matrix jacobian = null;
    private double damping=0.1;

    private ArrayList<ArrayList<ArrayList<Double>>>
    errorBackpropagation;
    private Matrix errorLMA;

    public ArrayList<ArrayList<ArrayList<Double>>> lastWeights;
}
```

Basically, the `train` function is the same as that of the backpropagation, except for the following calculation of the Jacobian and error matrices and the damping update:

```
@Override
public void train() throws NeuralException{
    neuralNet.setNeuralNetMode(NeuralNet.NeuralNetMode.TRAINING) ;
    forward();
    double currentOverallError=overallGeneralError;
    buildErrorVector(); // copy the error values to the error matrix
    while(epoch<MaxEpochs && overallGeneralError>MinOverallError
        && damping<=10000000000.0){ // to prevent the damping from
        growing up to infinity
        backward(); // to determine the error backpropagation
        calculateJacobian(); // copies the derivatives to the jacobian
        matrix
        applyNewWeights(); //update the weights
        forward(); //forward all records to evaluate new overall error
        if(overallGeneralError<currentOverallError){
            if the new error is less than current
            damping/=10.0; // the damping factor reduces
```

```
        currentOverallError=overallGeneralError;
    }
    else{ // otherwise, the damping factor grows
        damping*=10.0;
        restoreLastWeights(); // the last weights are recovered
        forward();
    }
    buildErrorVector(); reevaluate the error matrix
}
neuralNet.setNeuralNetMode(NeuralNet.NeuralNetMode.RUN);

}
```

The loop where it goes over the training dataset calls the method calculateJacobian. This method works on the error backpropagation evaluated in the method backward:

```
double input;
if(p==numberOfInputs)
    input=1.0;
else
    input = n.getInput(p);
double deltaBackprop = errorBackpropagation.get(m).get(l).get(k);
jacobian.setValue(i, j++, deltaBackprop*input);
```

In the code listing, p is the input connecting to the neuron (when it is equal to the number of neuron inputs, it represents the bias), k is the neuron, l is the layer, m is the neural output, i is a sequential index of the record, and j is the sequential index of the weight or bias, according to the layer and neuron in which it is located. Note that after setting the value in the Jacobian matrix, j is incremented.

The weight update is performed by means of determining the deltaWeight matrix:

```
Matrix jacob=jacobian.subMatrix(rowi, rowe, 0, numberOfWeights-1);
Matrix errorVec = errorLMA.subMatrix(rowi, rowe, 0, 0);
Matrix pseudoHessian=jacob.transpose().multiply(jacob);
Matrix miIdent = new IdentityMatrix(numberOfWeights)
    .multiply(damping);
Matrix inverseHessianMi = pseudoHessian.add(miIdent).inverse();
Matrix deltaWeight = inverseHessianMi.multiply(jacob.transpose())
    .multiply(errorVec);
```

The previous code refers to the matrix algebra shown in the section presenting the algorithm. The matrix `deltaWeight` contains the steps for each weight in the neural network. In the following code, `k` is the neuron, `j` is the input, and `l` is the layer:

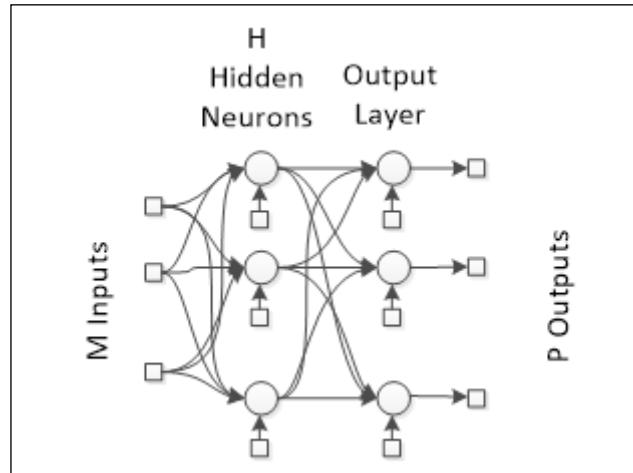
```
Neuron n=nL.getNeuron(k);
double currWeight=n.getWeight(j);
double newWeight=currWeight+deltaWeight.getValueAt(i++,0);
newWeights.get(l).get(k).set(j,newWeight);
lastWeights.get(l).get(k).set(j,currWeight);
n.updateWeight(j, newWeight);
```

Note that the weights are saved in the `lastWeights` array, so they can be recovered if the error gets worse.

Extreme learning machines

Taking advantage of the matrix algebra, **extreme learning machines** (ELMs) are able to converge learning very fast. This learning algorithm has one limitation, since it is applied only on neural networks containing one single hidden layer. In practice, one hidden layer works pretty fine for most applications.

Representing the neural network in matrix algebra, for the following neural network:



We have the corresponding equations:

$$H = \begin{pmatrix} g(X_0W_0 + b_0) & \cdots & g(X_HW_H + b_H) \\ \vdots & \ddots & \vdots \\ g(X_NW_0 + b_0) & \cdots & g(X_NW_H + b_H) \end{pmatrix}$$

$$X_i = (x_0[i] \ \cdots \ x_m[i])^T$$

$$W_j = (w_{0j} \ \cdots \ w_{mj})^T$$

$$Y = \begin{pmatrix} y_0[0] & \cdots & y_p[0] \\ \vdots & \ddots & \vdots \\ y_0[N] & \cdots & y_p[N] \end{pmatrix} = (H^{-1})(\beta_0 \ \cdots \ \beta_p)$$

Here, H is the output of the hidden layer, $g()$ is the activation function of the hidden layer, X_i is the i th input record, W_j is the weight vector for the j th hidden neuron, b_j is the bias of the j th hidden neuron, β_p is the weight vector for the output p , and Y is the output generated by the neural network.

In the ELM algorithm, the hidden layer weights are generated randomly, while the output weights are adjusted according to a least squares approach:

$$\beta = (H^T H)^{-1} H^T T$$

$$T = \begin{pmatrix} YT_0[0] & \cdots & YT_p[0] \\ \vdots & \ddots & \vdots \\ YT_0[N] & \cdots & YT_p[N] \end{pmatrix}$$

Here, T is the target output training dataset.

This algorithm is implemented in a class called `ELM` in the same package as the other training algorithms. This class will inherit from `DeltaRule`, which has all the basic properties for supervised learning algorithms:

```
public class ELM extends DeltaRule {

    private Matrix H;
    private Matrix T;
```

```

public ELM(NeuralNet _neuralNet,NeuralDataSet _trainDataSet) {
    super(_neuralNet,_trainDataSet);
    learningMode=LearningMode.BATCH;
    initializeMatrices();
}
}

```

In this class, we define the matrices H and T , which will be later used for output weight calculation. The constructor is similar to the other training algorithms, except for the fact that this algorithm works only on batch mode.

Since this training algorithm takes only one epoch, the train method forwards all training records to build the H matrix. Then, it calculates the output weights:

```

@Override
public void train() throws NeuralException{
    if(neuralNet.getNumberOfHiddenLayers()!=1)
        throw new NeuralException("The ELM learning algorithm can be
performed only on Single Hidden Layer Neural Network");
    neuralNet.setNeuralNetMode(NeuralNet.NeuralNetMode.TRAINING);
    int k=0;
    int N=trainingDataSet.numberOfRecords;
    currentRecord=0;
    forward();
    double currentOverallError=overallGeneralError;
    while(k<N) {
        forward(k);
        buildMatrices();
        currentRecord=++k;
    }
    applyNewWeights();
    forward();
    currentOverallError=overallGeneralError;
    neuralNet.setNeuralNetMode(NeuralNet.NeuralNetMode.RUN);
}

```

The `buildMatrices` method only places the output of the hidden layer to its corresponding row in the H matrix. The output weights are adjusted in the `applyNewWeights` method:

```

@Override
public void applyNewWeights(){
    Matrix Ht = H.transpose();
    Matrix HtH = Ht.multiply(H);
    Matrix invH = HtH.inverse();
    Matrix invHt = invH.multiply(Ht);
}

```

```
Matrix beta = invHt.multiply(T);

OutputLayer ol = this.neuralNet.getOutputLayer();
HiddenLayer hl = (HiddenLayer)ol.getPreviousLayer();
int h = hl.getNumberOfNeuronsInLayer();
int n = ol.getNumberOfNeuronsInLayer();
for(int i=0;i<=h;i++){
    for(int j=0;j<n;j++){
        if(i<h || outputBiasActive)
            ol.getNeuron(j).updateWeight(i, beta.getValue(i, j));
    }
}
```

Practical example 1 – the XOR case with delta rule and backpropagation

Now let's see the multilayer perceptron in action. We coded the example `XORTest.java`, which basically creates two neural networks with the following features:

Neural Network	Perceptron	Multi-layer Perceptron
Inputs	2	2
Outputs	1	1
Hidden Layers	0	1
Hidden Neurons in each layer	0	2
Hidden Layer Activation Function	Non	Sigmoid
Output Layer Activation Function	Linear	Linear
Training Algorithm	Delta Rule	Backpropagation
Learning Rate	0.1	0.3 Momentum 0.6
Max Epochs	4000	4000
Min. overall error	0.1	0.01

In Java, this is coded as follows:

```
public class XORTest {
    public static void main(String[] args) {
        RandomNumberGenerator.seed=0;
```

```

int numberOfInputs=2;
int numberOfOutputs=1;

int[] numberOfHiddenNeurons={2};

Linear outputAcFnc = new Linear(1.0);
Sigmoid hdAcFnc = new Sigmoid(1.0);
IActivationFunction[] hiddenAcFnc={hdAcFnc};

NeuralNet perceptron = new NeuralNet(numberOfInputs,
    numberOfOutputs,outputAcFnc);

NeuralNet mlp = new NeuralNet(numberOfInputs,numberOfOutputs
    ,numberOfHiddenNeurons,hiddenAcFnc,outputAcFnc);
}
}

```

Then, we define the dataset and the learning algorithms:

```

Double[][] _neuralDataSet = {
    {0.0 , 0.0 , 1.0 }
    ,
    {0.0 , 1.0 , 0.0 }
    ,
    {1.0 , 0.0 , 0.0 }
    ,
    {1.0 , 1.0 , 1.0 }
};

int[] inputColumns = {0,1};
int[] outputColumns = {2};

NeuralDataSet neuralDataSet = new NeuralDataSet(_neuralDataSet,inputCo
lumns,outputColumns);

DeltaRule deltaRule=new DeltaRule(perceptron,neuralDataSet
    ,LearningAlgorithm.LearningMode.ONLINE);

deltaRule.printTraining=true;
deltaRule.setLearningRate(0.1);
deltaRule.setMaxEpochs(4000);
deltaRule.setMinOverallError(0.1);

Backpropagation backprop = new Backpropagation(mlp,neuralDataSet
    ,LearningAlgorithm.LearningMode.ONLINE);
backprop.printTraining=true;

```

```
backprop.setLearningRate(0.3);  
backprop.setMaxEpochs(4000);  
backprop.setMinOverallError(0.01);  
backprop.setMomentumRate(0.6);
```

The training is then performed for both algorithms. As expected, the XOR case is not linearly separable by one single layer perceptron. The neural network runs the training but unsuccessfully:

```
deltaRule.train();  
  
Epoch=3997; Record=3; Overall Error=0.308641975308642  
Epoch=3998; Record=0; Overall Error=0.308641975308642  
Epoch=3998; Record=1; Overall Error=0.308641975308642  
Epoch=3998; Record=2; Overall Error=0.308641975308642  
Epoch=3998; Record=3; Overall Error=0.308641975308642  
Epoch=3999; Record=0; Overall Error=0.308641975308642  
Epoch=3999; Record=1; Overall Error=0.308641975308642  
Epoch=3999; Record=2; Overall Error=0.308641975308642  
Epoch=3999; Record=3; Overall Error=0.308641975308642  
Epoch=4000; Record=0; Overall Error=0.308641975308642  
End of Delta Rule training  
Training was unsuccessful  
Overall Error:0.308641975308642  
Min Overall Error:0.1  
Epochs of training:4000  
Target Outputs:  
Targets:  
Target Output[0]={ 1.0}  
Target Output[1]={ 0.0}  
Target Output[2]={ 0.0}  
Target Output[3]={ 1.0}  
Neural Output after training:  
Neural:  
Neural Output[0]={ 0.4444444444444441}  
Neural Output[1]={ 0.4999999999999999}  
Neural Output[2]={ 0.5555555555555555}  
Neural Output[3]={ 0.6111111111111112}
```

But the backpropagation algorithm for the multilayer perceptron manages to learn the XOR function after 39 epochs:

```
backprop.train();

Epoch=37; Record=3; Overall Error=0.014316276276702164
Epoch=38; Record=0; Overall Error=0.014205076880259711
Epoch=38; Record=1; Overall Error=0.013967510242490638
Epoch=38; Record=2; Overall Error=0.012818601428508655
Epoch=38; Record=3; Overall Error=0.011524501836923705
Epoch=39; Record=0; Overall Error=0.011442871267615572
Epoch=39; Record=1; Overall Error=0.011242993333301898
Epoch=39; Record=2; Overall Error=0.010319591111492887
Epoch=39; Record=3; Overall Error=0.00923709464092058
End of training
Training successful!
Overall Error:0.00923709464092058
Min Overall Error:0.01
Epochs of training:39
Target Outputs:
Targets:
Target Output[0]={ 1.0}
Target Output[1]={ 0.0}
Target Output[2]={ 0.0}
Target Output[3]={ 1.0}
Neural Output after training:
Neural:
Neural Output[0]={ 0.8635135822257722}
Neural Output[1]={ -0.034317801910536794}
Neural Output[2]={ -0.004429142261391461}
Neural Output[3]={ 0.8635135822257722}
```

Practical example 2 – predicting enrolment status

In Brazil, one of the ways for a person to enter university consists of taking an exam and if he/she achieves the minimum grade for the course that he/she is seeking, then he/she can enroll. To demonstrate the backpropagation algorithm, let us consider this scenario. The data shown in the table was collected from a university database. The second column represents the person's gender (one means female and zero means male); the third column has grades scaled by 100 and the last column is formed by two neurons (1,0 means performed enrollment and 0,1 means waiver enrollment):

Sample	Gender	Grade	Enrollment Status
1	1	0.73	1 0
2	1	0.81	1 0

Sample	Gender	Grade	Enrollment Status
3	1	0.86	1 0
4	0	0.65	1 0
5	0	0.45	1 0
6	1	0.70	0 1
7	0	0.51	0 1
8	1	0.89	0 1
9	1	0.79	0 1
10	0	0.54	0 1

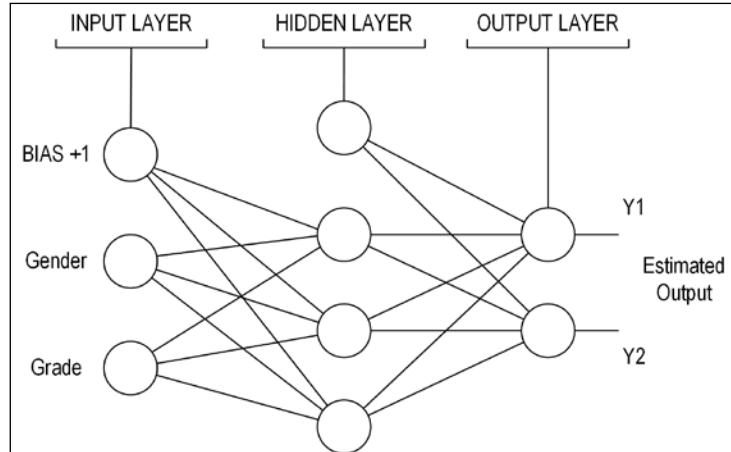
```
Double[][] _neuralDataSet = {
    {1.0,    0.73,    1.0,     -1.0},
    , {1.0,    0.81,    1.0,     -1.0}
    , {1.0,    0.86,    1.0,     -1.0}
    , {0.0,    0.65,    1.0,     -1.0}
    , {0.0,    0.45,    1.0,     -1.0}
    , {1.0,    0.70,   -1.0,     1.0}
    , {0.0,    0.51,   -1.0,     1.0}
    , {1.0,    0.89,   -1.0,     1.0}
    , {1.0,    0.79,   -1.0,     1.0}
    , {0.0,    0.54,   -1.0,     1.0}

};

int[] inputColumns = {0,1};
int[] outputColumns = {2,3};

NeuralDataSet neuralDataSet = new NeuralDataSet(_neuralDataSet,inputCo
lumns,outputColumns);
```

We create a neural network containing three neurons in the hidden layer, as shown in the following figure:



```

int numberOfInputs = 2;
int numberOfOutputs = 2;
int[] numberOfHiddenNeurons={5};

Linear outputAcFnc = new Linear(1.0);
Sigmoid hdAcFnc = new Sigmoid(1.0);
IActivationFunction[] hiddenAcFnc={hdAcFnc};

NeuralNet nnlm = new NeuralNet(numberOfInputs,numberOfOutputs
                               ,numberOfHiddenNeurons,hiddenAcFnc,outputAcFnc);

NeuralNet nnelm = new NeuralNet(numberOfInputs,numberOfOutputs
                               ,numberOfHiddenNeurons,hiddenAcFnc,outputAcFnc);
  
```

We've also set up the learning algorithms Levenberg-Marquardt and extreme learning machines:

```

LevenbergMarquardt lma = new LevenbergMarquardt(nnlm,
neuralDataSet,
LearningAlgorithm.LearningMode.BATCH);
lma.setDamping(0.001);
lma.setMaxEpochs(100);
lma.setMinOverallError(0.0001);

ELM elm = new ELM(neelm,neuralDataSet);
elm.setMinOverallError(0.0001);
elm.printTraining=true;
  
```

Running the training, we find that the training was successful. For the Levenberg-Marquardt algorithm, the minimum satisfied error was found after nine epochs:

```
Epoch= 0; Overall Error =27.504416063008975
Epoch= 1; Overall Error =0.0015829236005955073
Epoch= 2; Overall Error =0.0015829236005955073
Epoch= 3; Overall Error =0.0015829236005955073
Epoch= 4; Overall Error =6.565629615058485E-4
Epoch= 5; Overall Error =6.565629615058485E-4
Epoch= 6; Overall Error =3.0834297944887663E-4
Epoch= 7; Overall Error =3.0834297944887663E-4
Epoch= 8; Overall Error =3.0834297944887663E-4
Epoch= 9; Overall Error =2.395928627973254E-5
End of training
Training successful!
Overall Error:2.395928627973254E-5
Min Overall Error:1.0E-4
```

And the extreme learning machine found an error near to zero:

```
Epoch= 0; Overall Error =19.074542981619114
Epoch= 1; Overall Error =0.0
End of training
Training successful!
Overall Error:0.0
Min Overall Error:1.0E-4
```

Summary

In this chapter, we've seen how perceptrons can be applied to solve linear separation problems, but also their limitations in classifying nonlinear data. To suppress those limitations, we presented **multi-layer perceptrons (MLPs)** and new training algorithms: backpropagation, Levenberg-Marquardt, and extreme learning machines. We've also seen some classes of problems which MLPs can be applied to, such as classification and regression. The Java implementation explored the power of the backpropagation algorithm in updating the weights both in the output layer and the hidden layer. Two practical applications were shown to demonstrate the MLPs for the solution of problems with the three learning algorithms.

4

Self-Organizing Maps

In this chapter, we present a neural network architecture that is suitable for unsupervised learning: self-organizing maps, also known as Kohonen networks. This particular type of neural network is able to categorize records of data without any target output or find a representation of the data in a smaller dimension. Throughout this chapter, we are going to explore how this is achieved, as well as examples to attest to its capacity. The subtopics of this chapter are as follows:

- Neural networks unsupervised learning
- Competitive learning
- Kohonen self-organizing maps
- One-dimensional SOMs
- Two-dimensional SOMs
- Problems solved with unsupervised learning
- Java implementation
- Data visualization
- Practical problems

Neural networks unsupervised learning

In *Chapter 2, Getting Neural Networks to Learn* we've been acquainted with unsupervised learning, and now we are going to explore the features of this learning paradigm in more detail. The mission of unsupervised learning algorithms is to find patterns in datasets, where the parameters (weights in the case of neural networks) are adjusted without any error measure (there are no target values).

While the supervised algorithms provide an output comparable to the dataset that was presented, the unsupervised algorithms do not need to know the output values. The fundamentals of unsupervised learning are inspired by the fact that, in neurology, similar stimuli produce similar responses. So applying this to artificial neural networks, we can say that similar data produces similar outputs, so those outputs can be grouped or clustered.

Although this learning may be used in other mathematical fields, such as statistics, its core functionality is intended and designed for machine learning problems such as data mining, pattern recognition, and so on. Neural networks are a subfield in the machine learning discipline, and provided that their structure allows iterative learning, they serve as a good framework to apply this concept to.

Most of unsupervised learning applications are aimed at clustering tasks, which means that similar data points are to be clustered together, while different data points form different clusters. Also, one application that unsupervised learning is suitable for is dimensionality reduction or data compression, as long as simpler and smaller representations of the data can be found among huge datasets.

Unsupervised learning algorithms

Unsupervised algorithms are not unique to neural networks, as K-means, expectation maximization, and methods of moments are also examples of unsupervised learning algorithms. One common feature of all learning algorithms is the absence of mapping among variables in the current dataset; instead, one wishes to find a different meaning of this data, and that's the goal of any unsupervised learning algorithm.

While in supervised learning algorithms, we usually have a smaller number of outputs, for unsupervised learning, there is a need to produce an abstract data representation that may require a high number of outputs, but, except for classification tasks, their meaning is totally different than the one presented in the supervised learning. Usually, each output neuron is responsible for representing a feature or a class present in the input data. In most architectures, not all output neurons need to be activated at a time; only a restricted set of output neurons may fire, meaning that that neuron is able to better represent most of the information being fed at the neural input.

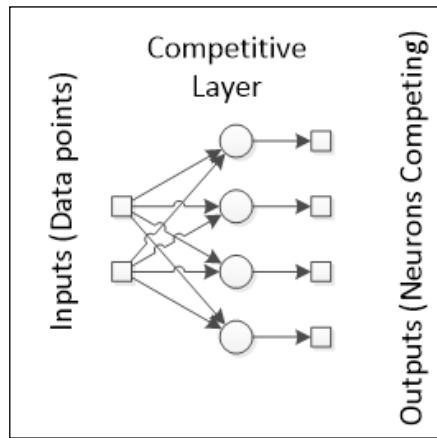


One advantage of unsupervised learning over supervised learning is that less computational power required by the first for the learning of huge datasets. Time consumption grows linearly while for the supervised learning it grows exponentially.

In this chapter, we are going to explore two unsupervised learning algorithms: competitive learning and Kohonen self-organizing maps.

Competitive learning

As the name implies, competitive learning handles a competition between the output neurons to determine which one is the winner. In competitive learning, the winning neuron is usually determined by comparing the weights against the inputs (they have the same dimensionality). To facilitate understanding, suppose we want to train a single layer neural network with two inputs and four outputs:



Every output neuron is then connected to these two inputs, hence for each neuron there are two weights.

[ For this learning, the bias is dropped from the neurons, so the neurons will process only the weighted inputs.]

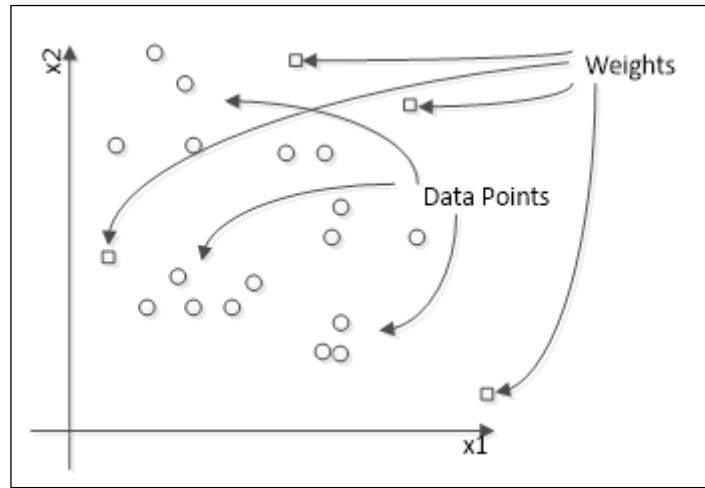
The competition starts after the data has been processed by the neurons. The winner neuron will be the one whose weights are *near* to the input values. One additional difference compared to the supervised learning algorithm is that only the winner neuron may update their weights, while the other ones remain unchanged. This is the so-called **winner-takes-all** rule. This intention to bring the neuron *nearer* to the input that caused it to win the competition.

Self-Organizing Maps

Considering that every input neuron i is connected to all output neurons j through a weight w_{ij} , in our case, we would have a set of weights:

$$W = [w_{11} w_{12} w_{21} w_{22} w_{13} w_{14} w_{23} w_{24}]$$

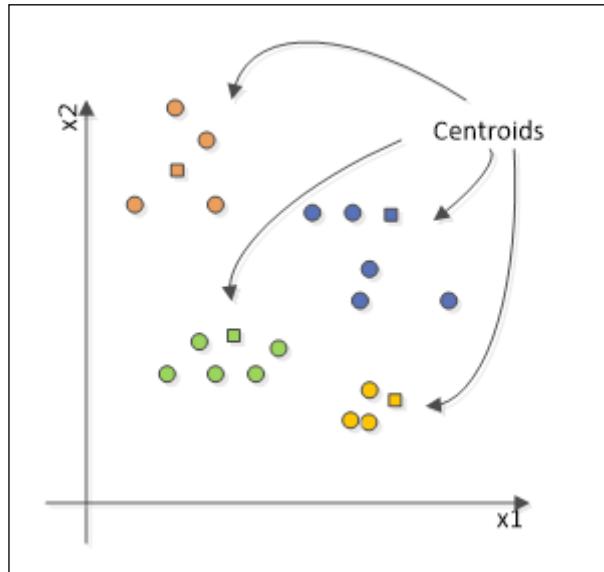
Provided that the weights of every neuron have the same dimensionality of the input data, let's consider all the input data points together in a plot with the weights of each neuron:



In this chart, let's consider the circles as the data points and the squares as the neuron weights. We can see that some data points are closer to certain weights, while others are farther but nearer to others. The neural network performs computations on the distance on the inputs and the weights:

$$o_j(X) = f_j \left(\frac{1}{\|w_j - x\|} \right)$$

The result of this equation will determine how much *stronger* a neuron is against its competitors. The neuron whose weight distance to the input is the smaller is considered the winner. After many iterations, the weights are driven near enough to the data points that give more cause the corresponding neuron to win that the changes are either too small or the weights fall in a zig-zag setting. Finally, when the network is already trained, the chart takes another shape:



As can be seen, the neurons form centroids surrounding the points capable of making the corresponding neuron stronger than its competitors.

In an unsupervised neural network, the number of outputs is completely arbitrary. Sometimes only some neurons are able to change their weights, while in other cases, all the neurons may respond differently to the same input, causing the neural network to never learn. In these cases, it is recommended either to review the number of output neurons, or consider another type of unsupervised learning.

Two stopping conditions are preferable in competitive learning:

- Predefined number of epochs: This prevents our algorithm from running for a longer time without convergence
- Minimum value of weight update: Prevents the algorithm from running longer than necessary

Competitive layer

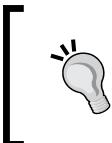
This type of neural layer is particular, as the outputs won't be necessarily the same as its neuron's outputs. Only one neuron will be fired at a time, thereby requiring a special rule to calculate the outputs. So, let's create a new class called `CompetitiveLayer` that will inherit from `OutputLayer` and starting with two new attributes: `winnerNeuron` and `winnerIndex`:

```
public class CompetitiveLayer extends OutputLayer {  
    public Neuron winnerNeuron;  
    public int[] winnerIndex;  
    //...  
}
```

This new class of neural layer will override the method `calc()` and add some new particular methods to get the weights:

```
@Override  
public void calc(){  
    if(input!=null && neuron!=null){  
        double[] result = new double[numberOfNeuronsInLayer];  
        for(int i=0;i<numberOfNeuronsInLayer;i++){  
            neuron.get(i).setInputs(this.input);  
            //perform the normal calculation  
            neuron.get(i).calc();  
            //calculate the distance and store in a vector  
            result[i]=getWeightDistance(i);  
            //sets all outputs to zero  
            try{  
                output.set(i,0.0);  
            }  
            catch(IndexOutOfBoundsException iobe){  
                output.add(0.0);  
            }  
        }  
        //determine the index and the neuron that was the winner  
        winnerIndex[0]=ArrayOperations.indexmin(result);  
        winnerNeuron=neuron.get(winnerIndex[0]);  
        // sets the output of this particular neuron to 1.0  
        output.set(winnerIndex[0], 1.0);  
    }  
}
```

In the next sections, we will define the class `Kohonen` for the Kohonen neural network. In this class, there will be an enum called `distanceCalculation`, which will have the different methods to calculate distance. In this chapter (and book), we'll stick to the Euclidian distance.



A new class called `ArrayOperations` was created to provide methods that facilitate operations with arrays. Functionalities such as getting the index of the maximum or minimum or getting a subset of the array are implemented in this class.

The distance between the weights of a particular neuron and the input is calculated by the method `getWeightDistance()`, which is called inside the `calc` method:

```
public double getWeightDistance(int neuron) {
    double[] inputs = this.getInputs();
    double[] weights = this.getNeuronWeights(neuron);
    int n=this.numberOfInputs;
    double result=0.0;
    switch(distanceCalculation){
        case EUCLIDIAN:
            //for simplicity, let's consider only the euclidian distance
        default:
            for(int i=0;i<n;i++){
                result+=Math.pow(inputs[i]-weights[i],2);
            }
            result=Math.sqrt(result);
    }
    return result;
}
```

The method `getNeuronWeights()` returns the weights of the neuron corresponding to the index passed in the array. Since it is simple and to save space here, we invite the reader to see the code to check its implementation.

Kohonen self-organizing maps

This network architecture was created by the Finnish professor Teuvo Kohonen at the beginning of the 80s. It consists of one single layer neural network capable of providing a *visualization* of the data in one or two dimensions.

In this book, we are going to use Kohonen networks also as a basic competitive layer with no links between the neurons. In this case, we are going to consider it as zero dimension (0-D).

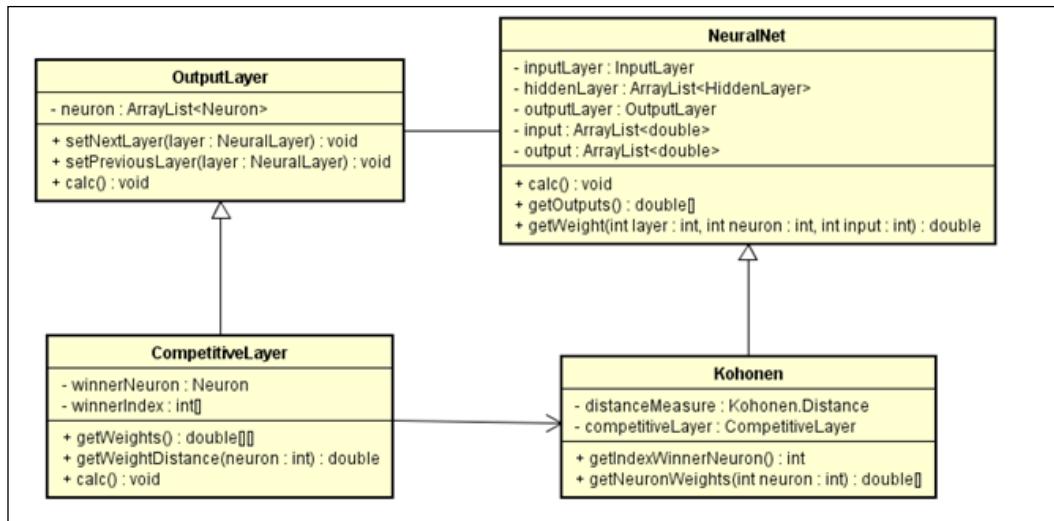
Theoretically, a Kohonen Network would be able to provide a 3-D (or even in more dimensions) representation of the data; however, in printed material such as this book, it is not practicable to show 3-D charts without overlapping some data. Thus in this book, we are going to deal only with 0-D, 1-D, and 2-D Kohonen networks.

Kohonen **Self-Organizing Maps (SOMs)**, in addition to the traditional single layer competitive neural networks (in this book, the 0-D Kohonen network), add the concept of neighborhood neurons. A dimensional SOM takes into account the index of the neurons in the competitive layer, letting the neighborhood of neurons play a relevant role during the learning phase.

An SOM has two modes of functioning: mapping and learning. In the mapping mode, the input data is classified in the most appropriate neuron, while in the learning mode, the input data helps the learning algorithm to build the *map*. This map can be interpreted as a lower-dimension representation from a certain dataset.

Extending the neural network code to Kohonen

In our code, let's create a new class inherited from `NeuralNet`, since it will be a particular type of neural network. This class will be called `Kohonen`, which will use the class `CompetitiveLayer` as the output layer. The following class diagram shows how these new classes are arranged:



Three types of SOMs are covered in this chapter: zero-, one- and two-dimensional. These configurations are defined in an enum `MapDimension`:

```
public enum MapDimension {ZERO, ONE_DIMENSION, TWO_DIMENSION};
```

The Kohonen constructor defines the dimension of the Kohonen neural network:

```
public Kohonen(int numberofinputs, int numberofoutputs,
WeightInitialization _weightInitialization, int dim){
    weightInitialization=_weightInitialization;
    activeBias=false;
    numberOfHiddenLayers=0; //no hidden layers
    /**
     * ...
     */
    numberofInputs=numberofinputs;
    numberofOutputs=numberofoutputs;
    input=new ArrayList<>(numberofinputs);
    inputLayer=new InputLayer(this,numberofinputs);
    // the competitive layer will be defined according to the dimension
    // passed in the argument dim
    outputLayer=new CompetitiveLayer(this,numberofoutputs,
    numberofinputs,dim);
    inputLayer.setNextLayer(outputLayer);
    setNeuralNetMode(NeuralNetMode.RUN);
    deactivateBias();
}
```

Zero-dimensional SOM

This is the pure competitive layer, where the order of the neurons is irrelevant. Features such as neighborhood functions are not taken into account. Only the winner neuron weights are affected during the learning phase. The map will be composed only of unconnected dots.

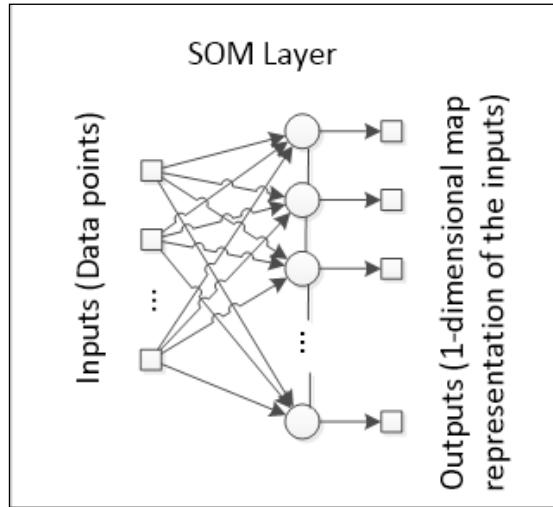
The following code snippet define a zero-dimensional SOM:

```
int numberofInputs=2;
int numberofNeurons=10;
Kohonen kn0 = new Kohonen(numberofInputs,numberofNeurons,new
UniformInitialization(-1.0,1.0),0);
```

Note the value 0 passed in the argument dim (the last of the constructor).

One-dimensional SOM

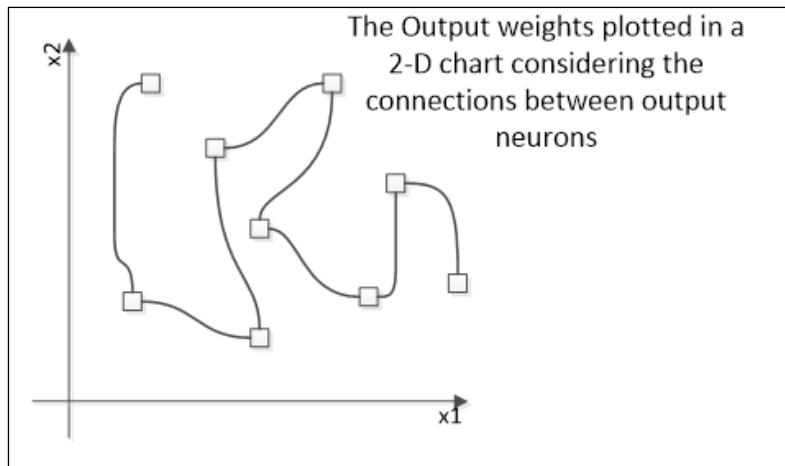
This architecture is similar to the network presented in the last section, **Competitive learning**, with the addition of neighborhood amongst the output neurons:



Note that every neuron on the output layer has one or two neighbors. Similarly, the neuron that fires the greatest value updates its weights, but in a SOM, the neighbor neurons also update their weights in a smaller rate.

The effect of the neighborhood extends the activation area to a wider area of the map, provided that all the output neurons must observe an organization, or a path in the one-dimensional case. The neighborhood function also allows for a better exploration of the properties of the input space, since it forces the neural network to keep the connections between neurons, therefore resulting in more information in addition to the clusters that are formed.

In a plot of the input data points with the neural weights, we can see the path formed by the neurons:



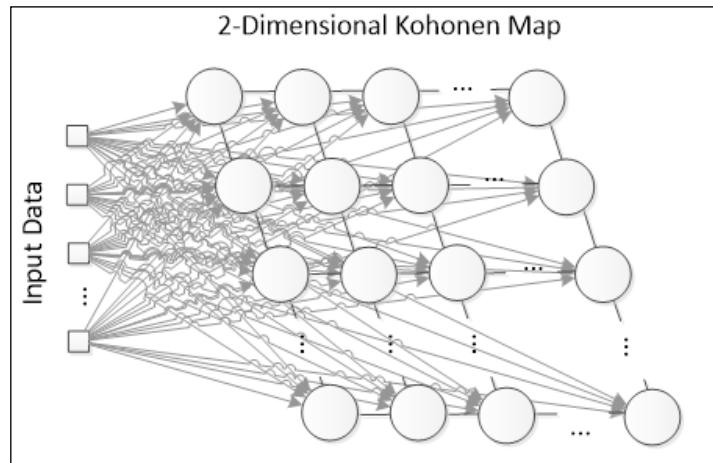
In the chart here presented, for simplicity, we plotted only the output weights to demonstrate how the map is designed in a (in this case) 2-D space. After training over many iterations, the neural network converges to a final shape that represent all data points. Provided that structure, a certain set of data may cause the Kohonen Network to design another shape in the space. This is a good example of dimensionality reduction, since a multidimensional dataset when presented to the Self-Organizing Map is able to produce one single line (in the 1-D SOM) that summarizes the entire dataset.

To define a one-dimensional SOM, we need to pass the value 1 as the argument `dim`:

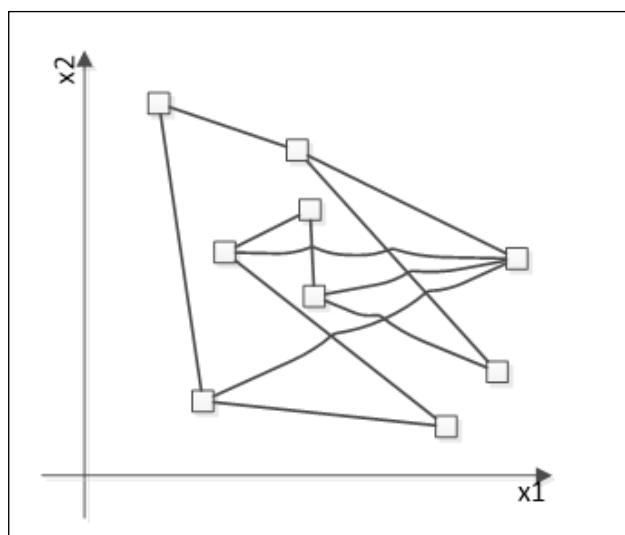
```
Kohonen kn1 = new Kohonen(numberOfInputs,numberOfNeurons,new  
UniformInitialization(-1.0,1.0),1);
```

Two-dimensional SOM

This is the most used architecture to demonstrate the Kohonen neural network power in a visual way. The output layer is one matrix containing $M \times N$ neurons, interconnected like a grid:

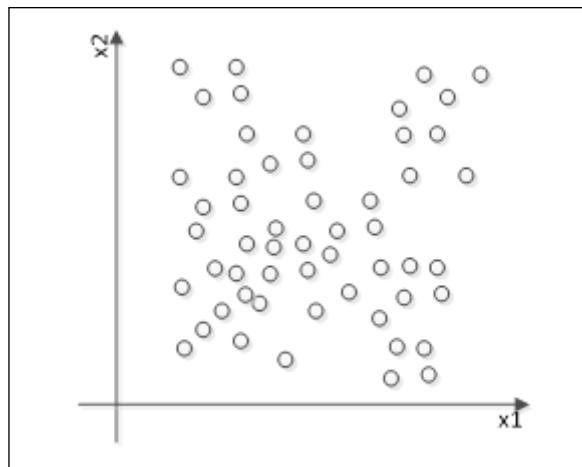


In the 2-D SOMs, every neuron now has up to four neighbors (in the square configuration), although in some representations, the diagonal neurons may also be considered, thus resulting in up to eight neighbors. Hexagonal representations are also useful. Let's see one example of what a 3×3 SOM plot looks like in a 2-D chart (considering two input variables):

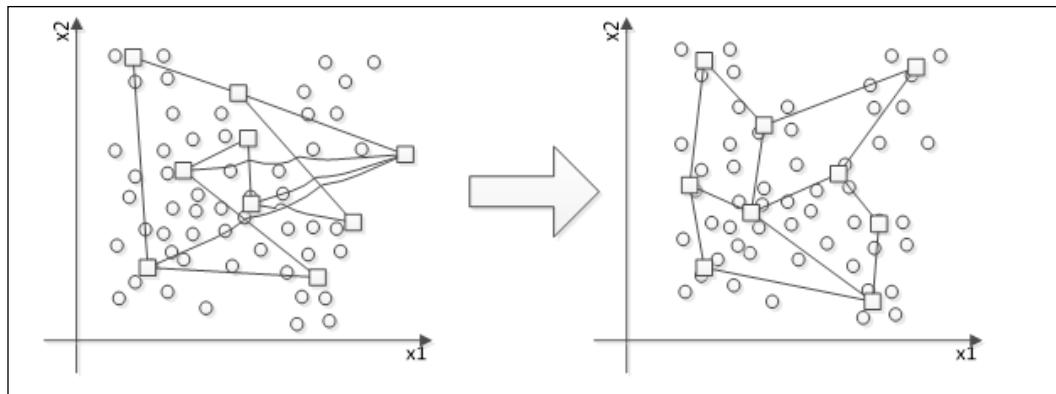


At first, the untrained Kohonen Network shows a very strange and screwed-up shape. The shaping of the weights will depend solely on the input data that is going to be fed to the SOM. Let's see an example of how the map starts to organize itself:

- Suppose we have the dense data set shown in the following plot:



- Applying SOM, the 2-D shape gradually changes, until it achieves the final configuration:



The final shape of a 2-D SOM may not always be a perfect square; instead, it will resemble a shape that could be drawn from the dataset. The neighborhood function is one important component in the learning process because it approximates the neighbor neurons in the plot, and the structure moves to a configuration that is more organized.



The grid on a chart is just the more used and didactic. There are other ways of showing the SOM diagram, such as the U-matrix and the cluster boundaries.

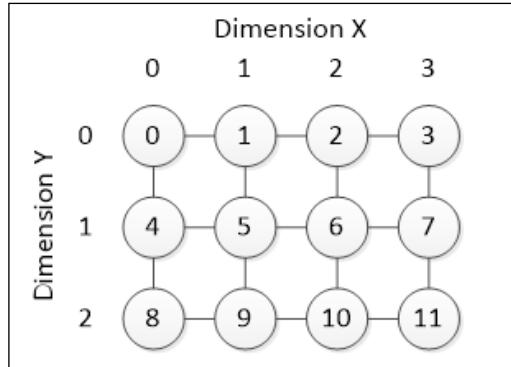


2D competitive layer

In order to better represent the neurons of a 2D competitive layer in a grid form, we're creating the CompetitiveLayer2D class, which inherits from CompetitiveLayer. In this class, we can define the number of neurons in the form of a grid of $M \times N$ neurons:

```
public class CompetitiveLayer2D extends CompetitiveLayer {  
  
    protected int sizeMapX; // neurons in dimension X  
    protected int sizeMapY; // neurons in dimension Y  
  
    protected int[] winner2DIndex;// position of the neuron in grid  
  
    public CompetitiveLayer2D(NeuralNet _neuralNet,int  
        numberOfNeuronsX,int numberOfNeuronsY,int numberOfInputs){  
        super(_neuralNet,numberOfNeuronsX*numberOfNeuronsY,  
        numberOfInputs);  
        this.dimension=Kohonen.MapDimension.TWO_DIMENSION;  
        this.winnerIndex=new int[1];  
        this.winner2DIndex=new int[2];  
        this.coordNeuron=new int [numberOfNeuronsX*numberOfNeuronsY]  
        [2];  
        this.sizeMapX=numberOfNeuronsX;  
        this.sizeMapY=numberOfNeuronsY;  
        //each neuron is assigned a coordinate in the grid  
        for(int i=0;i<numberOfNeuronsY;i++){  
            for(int j=0;j<numberOfNeuronsX;j++){  
                coordNeuron[i*numberOfNeuronsX+j] [0]=i;  
                coordNeuron[i*numberOfNeuronsX+j] [1]=j;  
            }  
        }  
    }  
}
```

The coordinate system in the 2D competitive layer is analogous to the Cartesian. Every neuron is assigned a position in the grid, with indexes starting from 0:



In the illustration above, 12 neurons are arranged in a 3×4 grid. Another feature added in this class is the indexing of neurons by the position in the grid. This allows us to get subsets of neurons (and weights), one entire specific row or column of the grid, for example:

```

public double[] getNeuronWeights(int x, int y){
    double[] nweights = neuron.get(x*sizeMapX+y).getWeights();
    double[] result = new double[nweights.length-1];
    for(int i=0;i<result.length;i++){
        result[i]=nweights[i];
    }
    return result;
}

public double[][] getNeuronWeightsColumnGrid(int y){
    double[][] result = new double[sizeMapY][numberOfInputs];
    for(int i=0;i<sizeMapY;i++){
        result[i]=getNeuronWeights(i,y);
    }
    return result;
}

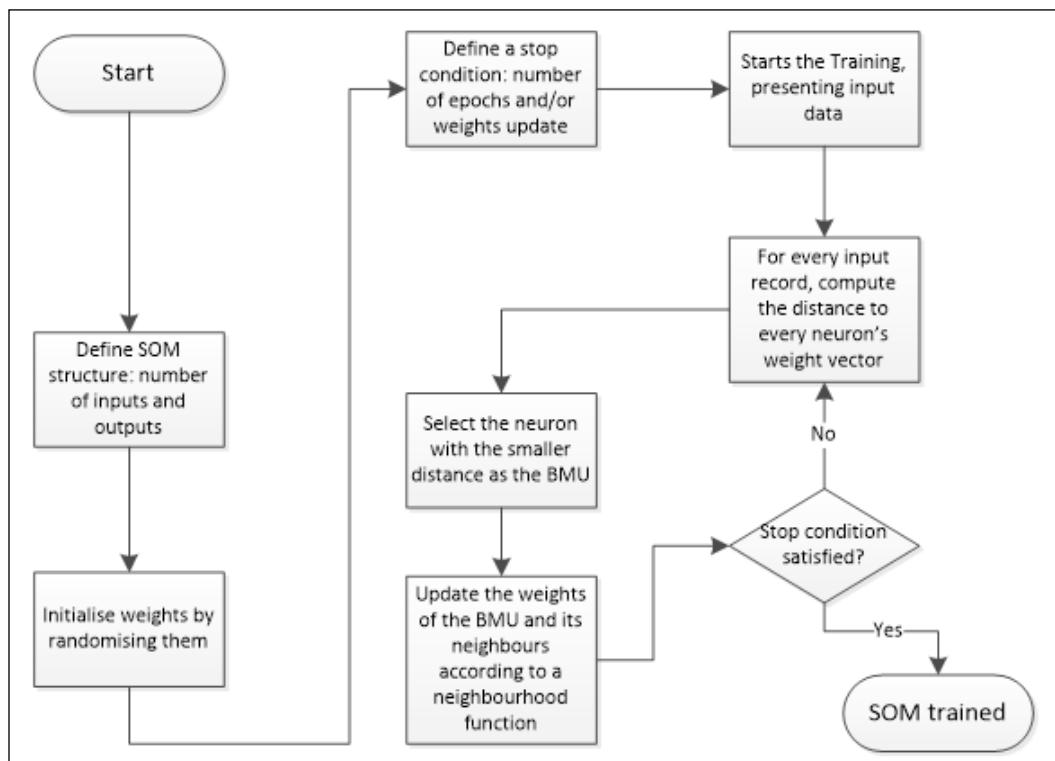
public double[][] getNeuronWeightsRowGrid(int x){
    double[][] result = new double[sizeMapX][numberOfInputs];
    for(int i=0;i<sizeMapX;i++){
        result[i]=getNeuronWeights(x,i);
    }
    return result;
}

```

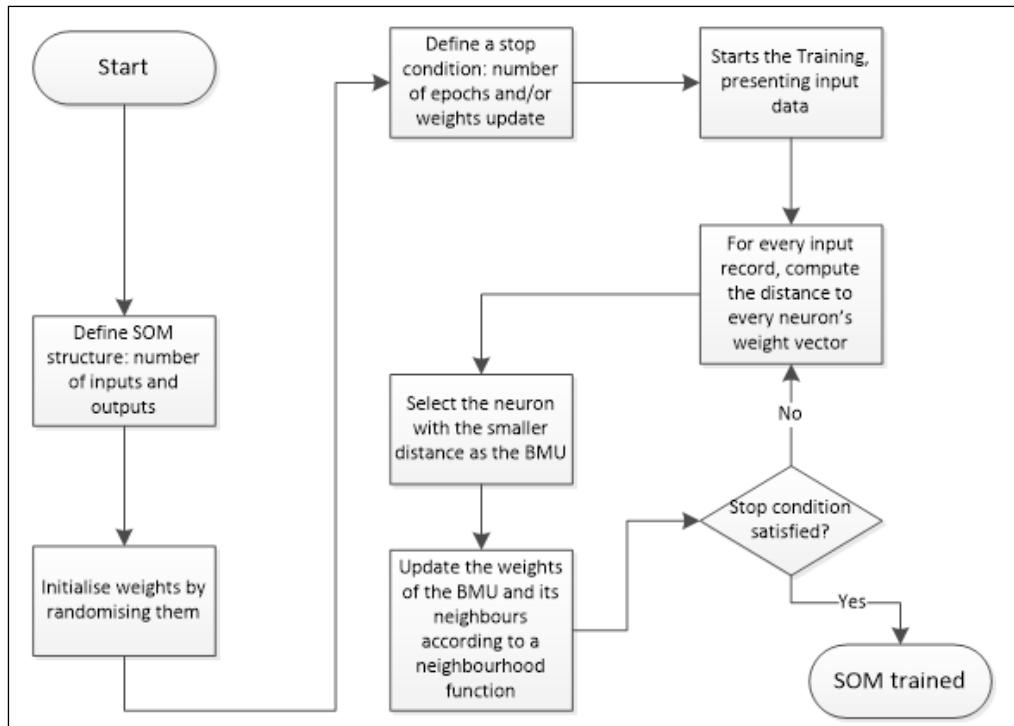
SOM learning algorithm

A self-organizing map aims at classifying the input data by clustering those data points that trigger the same response on the output. Initially, the untrained network will produce random outputs, but as more examples are presented, the neural network identifies which neurons are activated more often and then their *position* in the SOM output space is changed. This algorithm is based on competitive learning, which means a winner neuron (also known as best matching unit, or BMU) will update its weights and its neighbor weights.

The following flowchart illustrates the learning process of a SOM Network:



The learning resembles a bit those algorithms addressed in *Chapter 2, Getting Neural Networks to Learn* and *Chapter 3, Perceptrons and Supervised Learning*. Three major differences are the determination of the BMU with the distance, the weight update rule, and the absence of an error measure. The distance implies that nearer points should produce similar outputs, thus here the criterion to determine the lowest BMU is the neuron which presents a lower distance to some data point. This Euclidean distance is usually used, and in this book we will apply it for simplicity:



The weight-to-input distance is calculated by the method `getWeightDistance()` of the `CompetitiveLayer` class for a specific neuron *i* (argument *neuron*). This method was described above.

Effect of neighboring neurons – the neighborhood function

The weight update rule uses a neighborhood function $\Theta(u,v,s,t)$ which states how much a neighbor neuron u (BMU unit) is close to a neuron v . Remember that in a dimensional SOM, the BMU neuron is updated together with its neighbor neurons. This update is also dependent on a neighborhood radius, which takes into account the number of epoch's s and a reference epoch t :

$$\Theta(u,v,s,t) = \exp\left(\frac{d_{u,v}(N_u, N_v)^2}{2\sigma^2(s,t)}\right)$$

Here, $d_{u,v}$ is the neuron distance between neurons u and v in the grid. The radius is calculated as follows:

$$\sigma(s,t) = \sigma_0 \delta^{-\frac{s}{t}}$$

Here, σ_0 is the initial radius. The effect of the number of epochs (s) and the reference epoch (t) is the decreasing of the neighborhood radius and thereby the effect of neighborhood. This is useful because in the beginning of the training, the weights need to be updated more often, because they are usually randomly initialized. As the training process continues, the updates need to be weaker, otherwise the neural network will continue to change its weights forever and will never converge.

$$\sigma_0$$

The neighborhood function and the neuron distance are implemented in the `CompetitiveLayer` class, with overridden versions for the `CompetitiveLayer2D` class:

CompetitiveLayer	CompetitiveLayer2D
<pre>public double neighborhood(int u, int v, int s,int t){ double result; switch(dimension) { case ZERO: if(u==v) result=1.0; else result=0.0; break; case ONE_DIMENSION: default: double exponent=- (neuronDistance(u,v) /neighborhoodRadius(s,t)); result=Math.exp(exponent); } return result; }</pre>	<pre>@Override public double neighborhood(int u, int v, int s,int t){ double result; double exponent=- (neuronDistance(u,v) /neighborhoodRadius(s,t)); result=Math.exp(exponent); return result; }</pre>
<pre>public double neuronDistance(int u,int v){ return Math. abs(coordNeuron[u] [0]-coordNeuron[v] [0]); }</pre>	<pre>@Override public double neuronDistance(int u,int v){ double distance= Math.pow(coordNeuron[u] [0] -coordNeuron[v] [0],2); distance+= Math.pow(coordNeuron[u] [1]-coordNeuron[v] [1],2); return Math.sqrt(distance); }</pre>

The neighborhood radius function is the same for both classes:

```
public double neighborhoodRadius(int s,int t){
    return this.initialRadius*Math.exp(-((double)s/(double)t));
}
```

The learning rate

The learning rate also becomes weaker as the training goes on:

$$a(s,t)$$

$$a(s,t) = a_0 \exp(-s/t)$$

The parameter a_0 is the initial learning rate. Finally, considering the neighborhood function and the learning rate, the weight update rule is as follows:

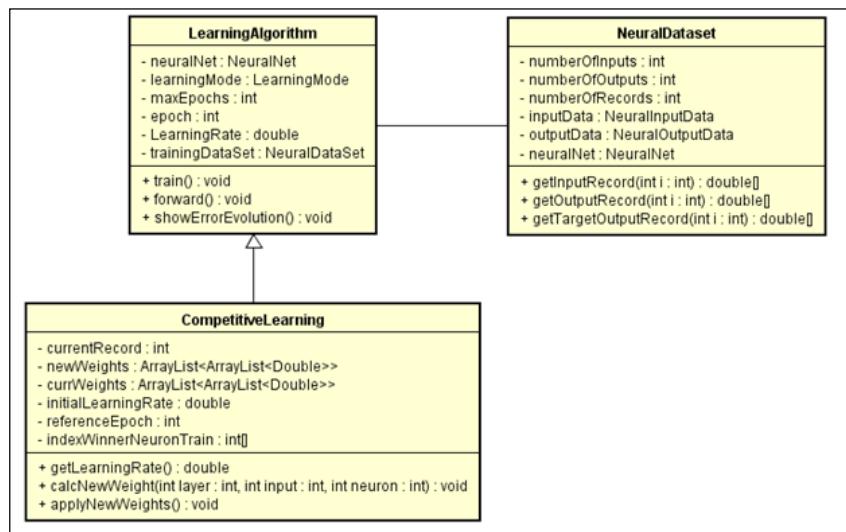
$$a_0$$

$$\Delta W_{kj} = \Theta(i, j, s, t) a(s, t) (X_k - W_{kj})$$

Here, X_k is the k th input, and W_{kj} is the weight connecting the k th input to the j th output.

A new class for competitive learning

Now that we have a competitive layer, a Kohonen neural network, and defined the methods for neighboring functions, let's create a new class for competitive learning. This class will inherit from `LearningAlgorithm` and will receive Kohonen objects for learning:



As seen in *Chapter 2, Getting Neural Networks to Learn a LearningAlgorithm* object receives a neural dataset for training. This property is inherited by the `CompetitiveLearning` object, which implements new methods and properties to realize the competitive learning procedure:

```
public class CompetitiveLearning extends LearningAlgorithm {
    // indicates the index of the current record of the training
    dataset
    private int currentRecord=0;
    //stores the new weights until they will be applied
    private ArrayList<ArrayList<Double>> newWeights;
    //saves the current weights for update
    private ArrayList<ArrayList<Double>> currWeights;
    // initial learning rate
    private double initialLearningRate = 0.3;
    //default reference epoch
    private int referenceEpoch = 30;
    //saves the index of winner neurons for each training record
    private int[] indexWinnerNeuronTrain;
    /**
    */
}
```

The learning rate, as opposed to the previous algorithms, now changes over the training process, and it will be returned by the method `getLearningRate()`:

```
public double getLearningRate(int epoch) {
    double exponent=(double)(epoch)/(double)(referenceEpoch);
    return initialLearningRate*Math.exp(-exponent);
}
```

This method is used in the `calcWeightUpdate()`:

```
@Override
public double calcNewWeight(int layer,int input,int neuron)
    throws NeuralException{
//...
    Double deltaWeight=getLearningRate(epoch);
    double xi=neuralNet.getInput(input);
    double wi=neuralNet.getOutputLayer().getWeight(input, neuron);
    int wn = indexWinnerNeuronTrain[currentRecord];
    CompetitiveLayer cl = ((CompetitiveLayer)((Kohonen)(neuralNet))
        .getOutputLayer()));
    switch(learningMode){
        case BATCH:
        case ONLINE: //The same rule for batch and online modes
```

```
        deltaWeight*=cl.neighborhood(wn, neuron, epoch, referenceEpoch)
        *(xi-wi);
        break;
    }
    return deltaWeight;
}
```

The `train()` method is adapted as well for competitive learning:

```
@Override
public void train() throws NeuralException{
//...
epoch=0;
int k=0;
forward();
//...
currentRecord=0;
forward(currentRecord);
while(!stopCriteria()){
    // first it calculates the new weights for each neuron and input
    for(int j=0;j<neuralNet.getNumberOfOutputs();j++) {
        for(int i=0;i<neuralNet.getNumberOfInputs();i++) {
            double newWeight=newWeights.get(j).get(i);
            newWeights.get(j).set(i,newWeight+calcNewWeight(0,i,j));
        }
    }
    //the weights are promptly updated in the online mode
    switch(learningMode) {
        case BATCH:
            break;
        case ONLINE:
        default:
            applyNewWeights();
    }
    currentRecord++;
    if(k>=trainingDataSet.numberOfRecords) {
        //for the batch mode, the new weights are applied once an epoch
        if(learningMode==LearningAlgorithm.LearningMode.BATCH) {
            applyNewWeights();
        }
        k=0;
        currentRecord=0;
        epoch++;
    }
}
```

```
    forward(k);
  ...
}
}
}
```

The implementation for the method `appliedNewWeights()` is analogous to the one presented in the previous chapter, with the exception that there is no bias and there is only one output layer.

Time to play: SOM applications in action. Now it is time to get hands-on and implement the Kohonen neural network in Java. There are many applications of self-organizing maps, most of them being in the field of clustering, data abstraction, and dimensionality reduction. But the clustering applications are the most interesting because of the many possibilities one may apply them on. The real advantage of clustering is that there is no need to worry about input/output relationship, rather the problem solver may concentrate on the input data. One example of clustering application will be explored in *Chapter 7, Clustering Customer Profiles*.

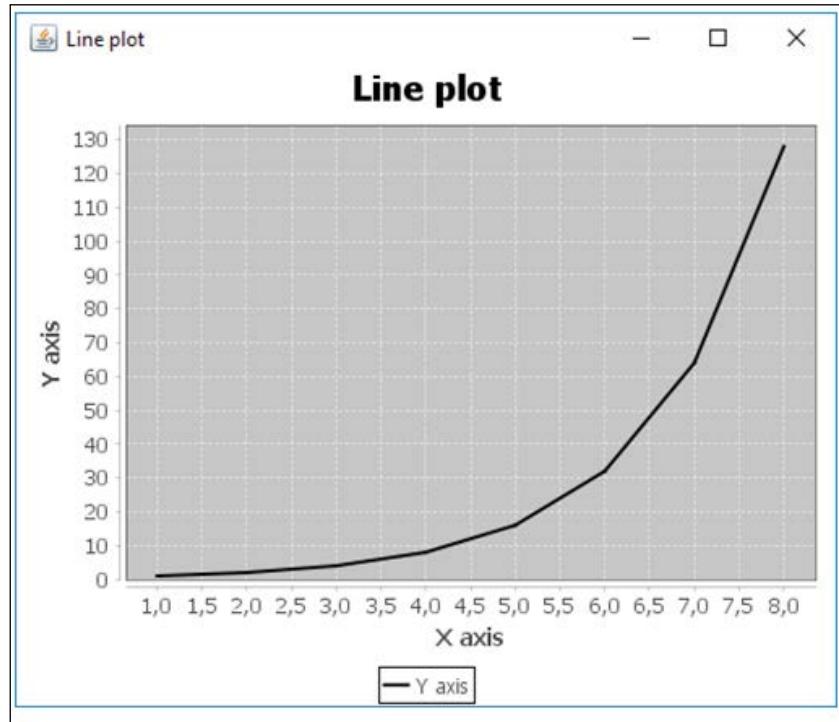
Visualizing the SOMs

In this section, we are going to introduce the plotting feature. Charts can be drawn in Java by using the freely available package JFreeChart (which can be downloaded from <http://www.jfree.org/jfreechart/>). This package is attached with this chapter's source code. So, we designed a class called **Chart**:

```
public class Chart {
  //title of the chart
  private String chartTitle;
  //datasets to be rendered in the chart
  private ArrayList<XYDataset> dataset = new ArrayList<XYDataset>();
  //the chart object
  private JFreeChart jfChart;
  //colors of each dataseries
  private ArrayList<Paint> seriesColor = new ArrayList<>();
  //types of series (dots or lines for now)
  public enum SeriesType {DOTS,LINES};
  //collections of types for each series
  public ArrayList<SeriesType> seriesTypes = new
  ArrayList<SeriesType>();

  ...
}
```

The methods implemented in this class are for plotting line and scatter plots. The main difference between them lies in the fact that line plots take all data series over one x-axis (usually the time axis) where each data series is a line; scatter plots, on the other hand, show dots in a 2D plane indicating its position in relation to each of the axis. Charts below show graphically the difference between them and the codes to generate them:



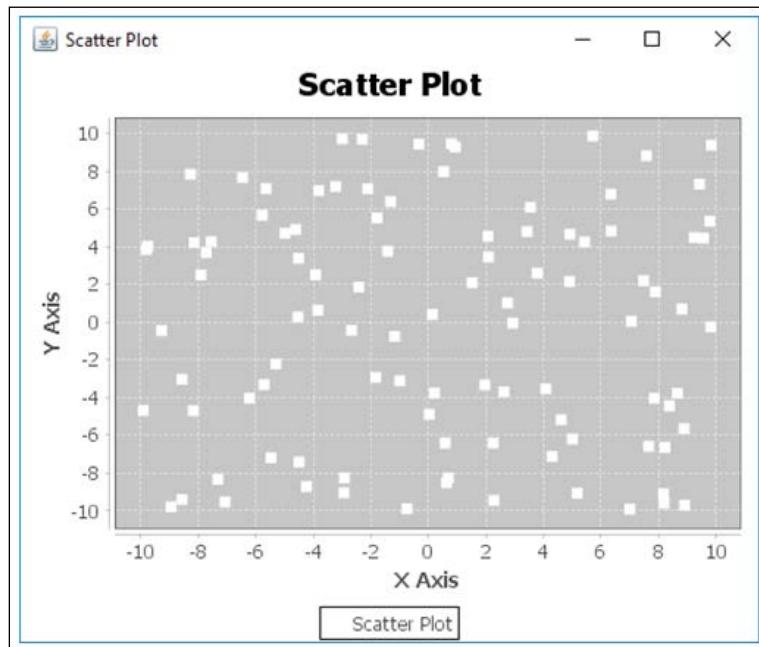
```
int numberOfPoints=10;

double[][] dataSet = {
    {1.0, 1.0}, {2.0, 2.0}, {3.0, 4.0}, {4.0, 8.0}, {5.0, 16.0}, {6.0, 32.0},
    {7.0, 64.0}, {8.0, 128.0}};

String[] seriesNames = {"Line Plot"};
Paint[] seriesColor = {Color.BLACK};

Chart chart = new Chart("Line Plot", dataSet, seriesNames, 0,
seriesColor, Chart.SeriesType.LINE);
```

```
ChartFrame frame = new ChartFrame("Line Plot", chart.linePlot("X  
Axis", "Y Axis"));  
  
frame.pack();  
frame.setVisible(true);
```



```
int numberOfInputs=2;  
int numberOfPoints=100;  
  
double[][] rndDataSet =  
RandomNumberGenerator  
.GenerateMatrixBetween  
(numberOfPoints  
, numberOfInputs, -10.0, 10.0);  
String[] seriesNames = {"Scatter Plot"};  
Paint[] seriesColor = {Color.WHITE};  
  
Chart chart = new Chart("Scatter Plot", rndDataSet, seriesNames, 0,  
seriesColor, Chart.SeriesType.DOTS);  
  
ChartFrame frame = new ChartFrame("Scatter Plot", chart.scatterPlot("X  
Axis", "Y Axis"));  
  
frame.pack();
```

We're suppressing the codes for chart generation (methods `linePlot()` and `scatterPlot()`); however, in the file `Chart.java`, the reader can find their implementation.

Plotting 2D training datasets and neuron weights

Now that we have the methods for plotting charts, let's plot the training dataset and neuron weights. Any 2D dataset can be plotted in the same way shown in the diagram of the last section. To plot the weights, we need to get the weights of the Kohonen neural network using the following code:

```
CompetitiveLayer cl = ((CompetitiveLayer) (neuralNet.  
getOutputLayer()));  
double[][] neuronsWeights = cl.getWeights();
```

In competitive learning, we can check visually how the weights *move* around the dataset space. So we're going to add a method (`showPlot2DData()`) to plot the dataset and the weights, a property (`plot2DData`) to hold the reference to the `ChartFrame`, and a flag (`show2DDData`) to determine whether the plot is going to be shown for every epoch:

```
protected ChartFrame plot2DData;  
  
public boolean show2DDData=false;  
  
public void showPlot2DData(){  
    double[][] data= ArrayOperations. arrayListToDoubleMatrix(  
    trainingDataSet.inputData.data);  
    String[] seriesNames = {"Training Data"};  
    Paint[] seriesColor = {Color.WHITE};  
  
    Chart chart = new Chart("Training epoch n°"+String.valueOf(epoch)+"  
",data,seriesNames,0,seriesColor,Chart.SeriesType.DOTS);  
    if(plot2DData ==null){  
        plot2DData = new ChartFrame("Training",chart.  
scatterPlot("X","Y"));  
    }  
  
    Paint[] newColor={Color.BLUE};  
    String[] neuronsNames={""};  
    CompetitiveLayer cl = ((CompetitiveLayer) (neuralNet.  
getOutputLayer()));
```

```

double[][] neuronsWeights = cl.getWeights();
switch(cl.dimension){
    case TWO_DIMENSION:
        ArrayList<double[][]> gridWeights = ((CompetitiveLayer2D)(cl)).getGridWeights();
        for(int i=0;i<gridWeights.size();i++){
            chart.addSeries(gridWeights.get(i),neuronsNames, 0,newColor,
Chart.SeriesType.LINES);
        }
        break;
    case ONE_DIMENSION:
        neuronsNames[0] ="Neurons Weights";
        chart.addSeries(neuronsWeights, neuronsNames, 0, newColor,
Chart.SeriesType.LINES);
        break;
    case ZERO:
        neuronsNames[0] ="Neurons Weights";
        default:
            chart.addSeries(neuronsWeights, neuronsNames, 0,newColor, Chart.
SeriesType.DOTS);
        }
    plot2DData.getChartPanel().setChart(chart.scatterPlot("X", "Y"));
}

```

This method will be called from the `train` method at the end of each epoch. A property called **sleep** will determine for how many milliseconds the chart will be displayed until the next epoch's chart replaces it:

```

if(show2DData){
    showPlot2DData();
    if(sleep!=-1)
        try{ Thread.sleep(sleep); }
        catch(Exception e){}
}

```

Testing Kohonen learning

Let's now define a Kohonen network and see how it works. First, we're creating a Kohonen with zero dimension:

```

RandomNumberGenerator.seed=0;
int numberofInputs=2;
int numberofNeurons=10;

```

Self-Organizing Maps

```
int numberOfPoints=100;

// create a random dataset between -10.0 and 10.0
double[][] rndDataSet = RandomNumberGenerator.GenerateMatrixBetween(n
umberOfPoints, numberOfInputs, -10.0, 10.0);

// create the Kohonen with uniform initialization of weights
Kohonen kn0 = new Kohonen(numberOfInputs,numberOfNeurons,new
UniformInitialization(-1.0,1.0),0);

//add the dataset to the neural dataset
NeuralDataSet neuralDataSet = new NeuralDataSet(rndDataSet,2);

//create an instance of competitive learning in the online mode
CompetitiveLearning complrn=new CompetitiveLearning(kn0,neuralDataSet,
LearningAlgorithm.LearningMode.ONLINE);

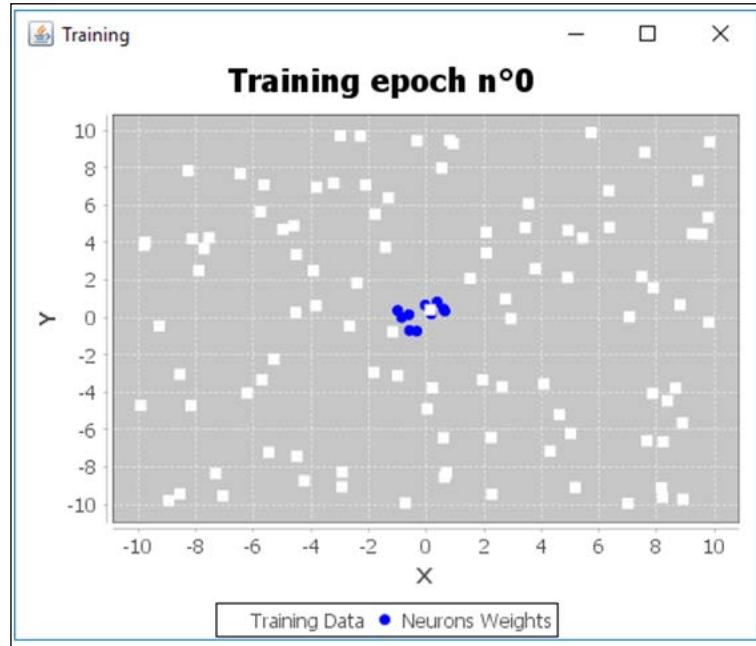
//sets the flag to show the plot
complrn.show2DDData=true;

try{
// give names and colors for the dataset
String[] seriesNames = {"Training Data"};
Paint[] seriesColor = {Color.WHITE};
//this instance will create the plot with the random series
Chart chart = new Chart("Training",rndDataSet,seriesNames,0,
seriesColor);
ChartFrame frame = new ChartFrame("Training", chart.scatterPlot("X",
"Y"));
frame.pack();
frame.setVisible(true);

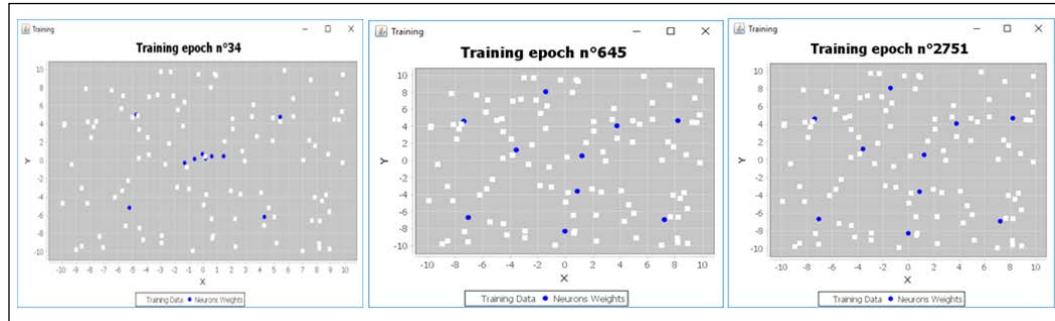
// we pass the reference of the frame to the complrn object
complrn.setPlot2DFrame(frame);
// show the first epoch
complrn.showPlot2DData();
//wait for the user to hit an enter
System.in.read();
//training begins, and for each epoch a new plot will be shown
complrn.train();
}
catch(Exception ne){

}
```

By running this code, we get the first plot:



As the training starts, the weights begin to distribute over the input data space, until finally it converges by being distributed uniformly along the input data space:



For one dimension, let's try something funkier. Let's create the dataset over a cosine function with random noise:

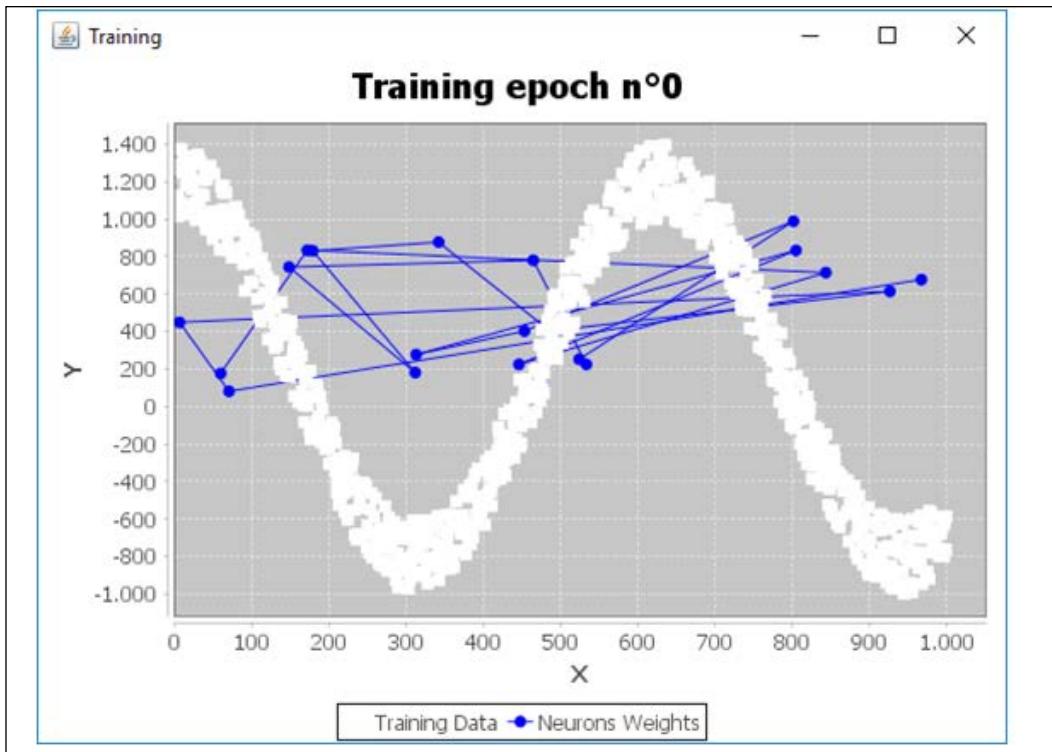
```
int numberOfPoints=1000;
int numberOfInputs=2;
int numberOfNeurons=20;
```

Self-Organizing Maps

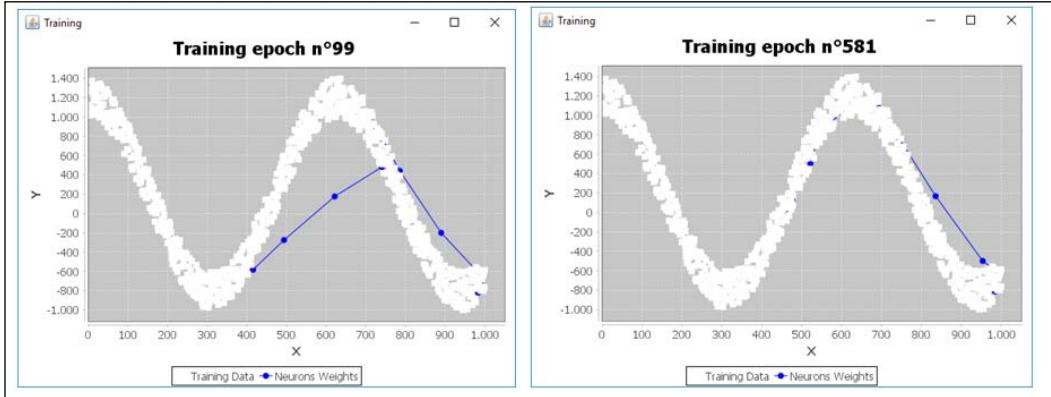
```
double[][] rndDataSet;

for (int i=0;i<numberOfPoints;i++) {
    rndDataSet[i][0]=i;
    rndDataSet[i][0]+=RandomNumberGenerator.GenerateNext();
    rndDataSet[i][1]=Math.cos(i/100.0)*1000;
    rndDataSet[i][1]+=RandomNumberGenerator.GenerateNext()*400;
}
Kohonen kn1 = new Kohonen(numberOfInputs,numberOfNeurons,new
UniformInitialization(0.0,1000.0),1);
```

By running the same previous code and changing the object to *kn1*, we get a line connecting all the weight points:



As the training continues, the lines tend to be organized along the data wave:



See the file `Kohonen1DTest.java` if you want to change the initial learning rate, maximum number of epochs, and other parameters.

Finally, let's see the two-dimensional Kohonen chart. The code will be a little bit different, since now, instead of giving the number of neurons, we're going to inform the Kohonen constructor the dimensions of our neural grid.

The dataset used here will be a circle with random noise added:

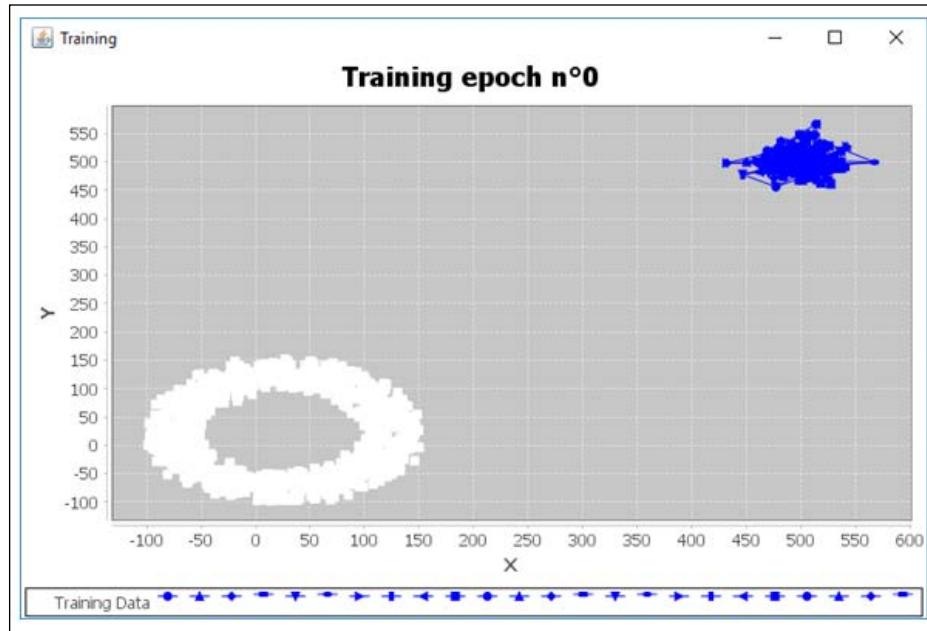
```
int numberOfPoints=1000;
for (int i=0;i<numberOfPoints;i++) {
    rndDataSet[i][0]*=Math.sin(i);
    rndDataSet[i][0]+=RandomNumberGenerator.GenerateNext()*50;
    rndDataSet[i][1]*=Math.cos(i);
    rndDataSet[i][1]+=RandomNumberGenerator.GenerateNext()*50;
}
```

Now let's construct the two-dimensional Kohonen:

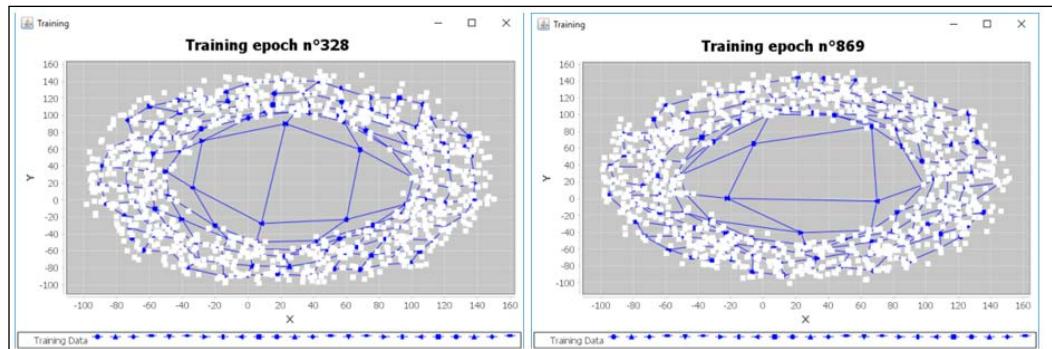
```
int numberOfInputs=2;
int neuronsGridX=12;
int neuronsGridY=12;
Kohonen kn2 = new Kohonen(numberOfInputs,neuronsGridX,neuronsGridY,new
GaussianInitialization(500.0,20.0));
```

Self-Organizing Maps

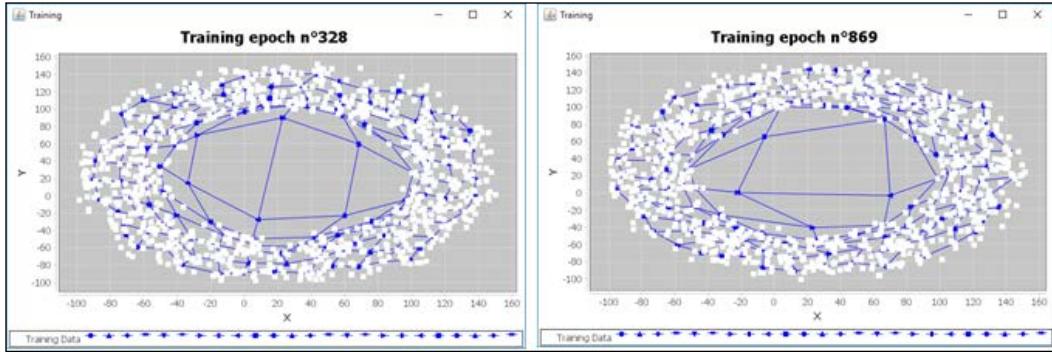
Note that we are using GaussianInitialization with mean 500.0 and standard deviation 20.0, that is, the weights will be generated at the position (500.0,500.0) while the data is centered around (50.0,50.0):



Now let's train the neural network. The neuron weights quickly move to the circle in the first epochs:



By the end of the training, the majority of the weights will be distributed over the circle, while in the center there will be an empty space, as the grid will be totally stretched out:



Summary

In this chapter, we've seen how to apply unsupervised learning algorithms on neural networks. We've been presented a new and suitable architecture for that end, the self-organizing maps of Kohonen. Unsupervised learning has proved to be as powerful as the supervised learning methods, because they concentrate only on the input data, without need to make input-output mappings. We've seen graphically how the training algorithms are able to drive the weights nearer to the input data, thereby playing a role in clustering and dimensionality reduction. In addition to these examples, Kohonen SOMs are also able to classify clusters of data, as each neuron will provide better responses for a particular set of inputs.

5

Forecasting Weather

This chapter presents one well-known application in daily life to which neural networks can perfectly be applied: forecasting weather. We are going to walk through the entire process of designing a neural network solution to this problem: how to choose the neural architecture and the number of neurons, as well as selecting and preprocessing data. Then the reader will be presented with techniques to handle time series datasets, from which our neural network is going to make predictions on weather variables using the Java programming language. The topics covered in this chapter are as follows:

- Neural networks for regression problems
- Loading/selecting data
- Input/output variables
- Choosing inputs
- Preprocessing
- Normalization
- Empirical design of neural networks

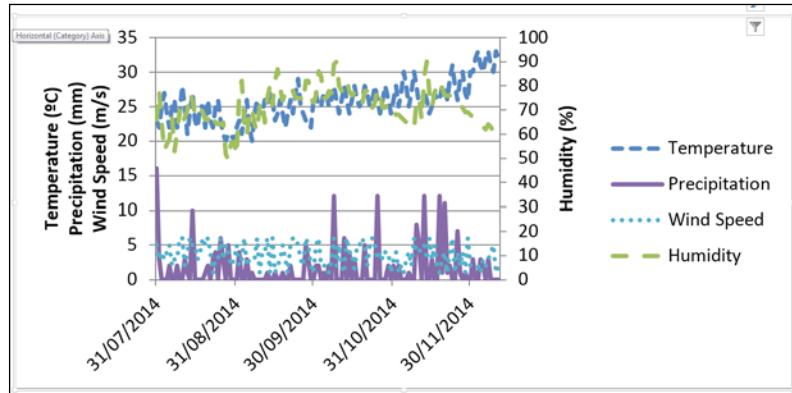
Neural networks for regression problems

So far, the reader has been presented with a number of neural network implementations and architectures, so now it is time to get into more complex cases. The power of neural networks in predictions is really astonishing since they can perform *learning* from historical data in such a way that neural connections are adapted to produce the same results according to some input data. For example, for a given situation (cause), there is a consequence (result) and this is represented as data; neural networks are capable of learning the nonlinear function that maps the situation to the consequence (or the cause to the result).

Prediction and regression problems are an interesting category to apply neural networks to. Let's take a look at a sample table containing weather data:

Date	Avg. Temperature	Pressure	Humidity	Precipitation	Wind Speed
July 31st	23 °C	880 mbar	66%	16 mm	5 m/s
August 1st	22 °C	881 mbar	78%	3 mm	3 m/s
August 2nd	25 °C	884 mbar	65%	0 mm	4 m/s
August 3rd	27 °C	882 mbar	53%	0 mm	3 m/s
...					
December 11th	32 °C	890 mbar	64%	0 mm	2 m/s

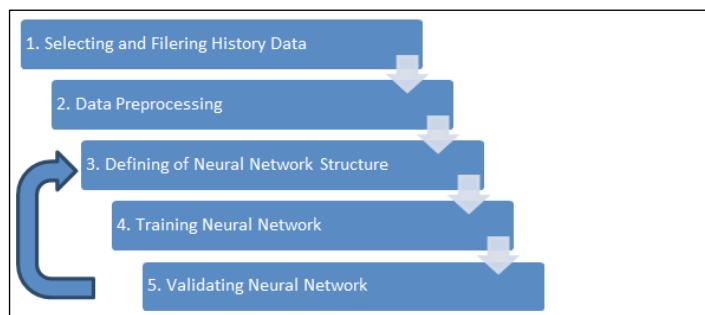
The table above depicts five variables containing hypothetical values of weather data collected from a hypothetical city, only for the purpose of this example. Now let's suppose that each of the variables contains a list of values sequentially taken over time. We can think of each list as a time series. On a time series chart, one can see how they evolve with time:



The relationship between these time series denotes a dynamic representation of weather in a certain city, as depicted in the chart above. We want the neural network to learn this dynamic representation; however, we need to structure this data in a way neural networks can process, that is, identifying which data series (variables) are the cause and which are the effect. Dynamic systems have variables whose value depends on past values, so neural network applications can rely not only on the present situation, but also on the past. This is very important because historical events influence the present and future.

Only after structuring data can we structure the neural network, that is, the number of inputs, outputs, and hidden nodes. However, there are many other architectures that may be suitable for prediction problems, such as radial basis functions and feedback networks, among others. In this chapter, we are dealing with a feedforward multilayer perceptron with the Backpropagation learning algorithm, to demonstrate how this architecture can be simply exploited to predict weather variables; also, this architecture presents very good generalized results with good selected data and there is little complexity involved in the design process.

The overall process for designing neural networks for prediction processes is depicted in the figure below:



If the neural network fails to be validated (**step 5**), usually a new structure (**step 3**) is defined, although sometimes **steps 1** and **step 2** may be repeated. Each of the steps in the figure will be addressed in the next sections in this chapter.

Loading/selecting data

First we need to load raw data into our Java environment. Data can be stored in a variety of data sources, from text files to structured database systems. One basic and simple type used is **CSV (Comma-Separated Values)**, which is simple and in general use. In addition, we'll need to transform this data and perform selection before presenting it to the neural network.

Building auxiliary classes

To deal with these tasks, we need some auxiliary classes in the package `edu.packt.neuralnet.data`. The first will be `LoadCsv` to read CSV files:

```

public class LoadCsv {
    //Path and file name separated for compatibility
    private String PATH;
  
```

```
private String FILE_NAME;
private double[][] dataMatrix;
private boolean columnsInFirstRow=false;
private String separator = ",";
private String fullPath;
private String[] columnNames;
final double missingValue=Double.NaN;

//Constructors
public LoadCsv(String path, String fileName)
//...
public LoadCsv(String fileName, boolean _columnsInFirstRow, String
_separator)
//...

//Method to load data from file returning a matrix
public double[][] getDataMatrix(String fullPath, boolean _
columnsInFirstRow, String _separator)
//...

//Static method for calls without instantiating LoadCsv object
public static double[][] getData(String fullPath, boolean _
columnsInFirstRow, String _separator)
//...

Method for saving data into csv file
public void save()
//...

//...
}
```



To save space here, we are not showing the full code. For more details and the full list of methods, please refer to the code and documentation in *Appendix C*.

We are also creating a class to store the raw data loaded from CSV into a structure containing not only the data but the information on this data, such as column names. This class will be called **DataSet**, inside the same package:

```
public class DataSet {
    //column names list
    public ArrayList<String> columns;
    //data matrix
```

```
public ArrayList<ArrayList<Double>> data;

public int numberOfRows;
public int numberOfRecords;

//creating from Java matrix
public DataSet(double[][] _data, String[] _columns) {
    numberOfRows=_data.length;
    numberOfRows=_data[0].length;
    columns = new ArrayList<>();
    for(int i=0;i<numberOfColumns;i++){
        //...
        columns.add(_columns[i]);
        //...
    }
    data = new ArrayList<>();
    for(int i=0;i<numberOfRecords;i++){
        data.add(new ArrayList<Double>());
        for(int j=0;j<numberOfColumns;j++) {
            data.get(i).add(_data[i][j]);
        }
    }
}

//creating from csv file
public DataSet(String filename,boolean columnsInFirstRow,String
separator){
    LoadCsv lcsv = new LoadCsv(filename,columnsInFirstRow,separator);
    double[][] _data= lcsv.getDataMatrix(filename, columnsInFirstRow,
separator);
    numberOfRows=_data.length;
    numberOfRows=_data[0].length;
    columns = new ArrayList<>();
    if(columnsInFirstRow){
        String[] columnNames = lcsv.getColumnNames();
        for(int i=0;i<numberOfColumns;i++){
            columns.add(columnNames[i]);
        }
    }
    else{ //default column names: Column0, Column1, etc.
        for(int i=0;i<numberOfColumns;i++){
            columns.add("Column"+String.valueOf(i));
        }
    }
}
```

```
data = new ArrayList<>();
for(int i=0;i<numberOfRecords;i++) {
    data.add(new ArrayList<Double>());
    for(int j=0;j<numberOfColumns;j++) {
        data.get(i).add(_data[i][j]);
    }
}
//...
//method for adding new column
public void addColumn(double[] _data, String name)
//...
//method for appending new data, number of columns must correspond
public void appendData(double[][] _data)
//...
//getting all data
public double[][] getData() {
    return ArrayOperations.arrayListToDoubleMatrix(data);
}
//getting data from specific columns
public double[][] getData(int[] columns) {
    return ArrayOperations.getMultipleColumns(getData(), columns);
}
//getting data from one column
public double[] getData(int col) {
    return ArrayOperations.getColumn(getData(), col);
}
//method for saving the data in a csv file
public void save(String filename, String separator)
//...
}
```

In *Chapter 4, Self-Organizing Maps*, we've created a class `ArrayOperations` in the package `edu.packt.neuralnet.math` to handle operations involving arrays of data. This class has a large number of static methods and it would be impractical to depict all of them here; however, information can be found in *Appendix C*.

Getting a dataset from a CSV file

To make the task easier, we've implemented a static method in the class `LoadCsv` to load a CSV file and automatically convert it into the structure of a `DataSet` object:

```
public static DataSet getDataSet(String fullPath, boolean _columnsInFirstRow, String _separator){
```

```

LoadCsv lcsv = new LoadCsv(fullPath,_columnsInFirstRow,_separator);
lcsv.columnsInFirstRow=_columnsInFirstRow;
lcsv.separator=_separator;
try{
    lcsv.dataMatrix=lcsv.csvData2Matrix(fullPath);
    System.out.println("File "+fullPath+" loaded!");
}
catch(IOException ioe){
    System.err.println("Error while loading CSV file. Details: " +
ioe.getMessage());
}
return new DataSet(lcsv.dataMatrix, lcsv.columnNames);
}

```

Building time series

A time series structure is essential for all problems involving time dimensions or domains, such as forecasting and prediction. A class called `TimeSeries` implements some time-related attributes such as time column and delay. Let's take a look at the structure of this class:

```

public class TimeSeries extends DataSet {
    //index of the column containing time information
    private int indexTimeColumn;

    public TimeSeries(double[][] _data,String[] _columns) {
        super(_data,_columns); //just a call to superclass constructor
    }
    public TimeSeries(String path, String filename){
        super(path,filename);
    }
    public TimeSeries(DataSet ds){
        super(ds.getData(),ds.getColumns());
    }
    public void setIndexColumn(int col){
        this.indexTimeColumn=col;
        this.sortBy(indexTimeColumn);
    }
//...
}

```

In time series, one frequent operation is the delay or shift of values. For example, we want to include in the processing not the current but the two past values for the daily temperature. Considering temperature as a time series with a time column (date), we must shift the values in the number of cycles desired (one and two, in this example):

Original DataSet:		Transformed DataSet:			
Date (time column)	Temp	Date (time column)	Temp	Temp_1	Temp_2
42429(29/02/2016)	27.5	42429(29/02/2016)	27.5	NaN	NaN
42430(01/03/2016)	26.3	42430(01/03/2016)	26.3	27.5	NaN
42431(02/03/2016)	25.4	42431(02/03/2016)	25.4	26.3	27.5
42433(04/03/2016)	26.6	42432(04/03/2016)	26.6	NaN	25.4

[ We have used **Microsoft Excel®** to convert the `datetime` values into real values. Working with numerical values is always preferred to working with structures such as dates or categories. So in this chapter, we are using numerical values to represent date.]

While working with time series, one should pay attention to two points:

- There may be missing values or no measurements on a specific point of time; this can generate NaNs in the Java matrix.
- Shifting a column over one time period, for example, is not the same as getting the value of the previous row. That's why it is important to choose one column to be the time reference.

In the `ArrayOperations` class, we implemented a method `shiftColumn` to shift the column of a matrix considering a time column for reference. This method is called from another method of the same name in the `TimeSeries` class, and then used in the `shift` method:

```
public double[] shiftColumn(int col,int shift){
    double[][] _data = ArrayOperations.arrayListToDoubleMatrix(data);
    return ArrayOperations.shiftColumn(_data, indexTimeColumn, shift,
    col);
}
public void shift(int col,int shift){
    String colName = columns.get(col);
    if(shift>0) colName=colName+"_"+String.valueOf(shift);
    else colName=colName+"__"+String.valueOf(-shift);
   addColumn(shiftColumn(col,shift),colName);
}
```

Dropping NaNs

NANs are undesired values often present after loading or transforming data. They are undesired because we cannot operate over them. If we feed a NaN value into a neural network, the output will definitely be NaN, just consuming more computational power. That's why it is better to drop them out. In the class `DataSet`, we've implemented two methods to drop NaNs: one just substitutes a value for them, and the other drops the entire row if it has at least one missing value, as shown in the following figure:

	Col0	Col1	Col2	Col3	Col4	Col5	Col6	Col7	Col8	Col9	Drop?
Row0	filled	NaN	filled	Yes							
Row1	filled	No									
Row2	filled	Yes									
Row3	filled	filled	filled	filled	filled	filled	NaN	filled	filled	filled	Yes
Row4	filled	No									
Row5	filled	No									
Row6	filled	filled	filled	NaN	filled	filled	filled	filled	filled	filled	Yes
Row7	filled	NaN	filled	filled	Yes						
Row8	filled	No									
Row9	filled	No									
Row10	filled	filled	filled	filled	filled	filled	NaN	filled	filled	filled	Yes
Row11	filled	No									
Row12	filled	NaN	filled	Yes							
Row13	filled	No									

```
// dropping with a substituting value
public void dropNaN(double substvalue)
//...
// dropping the entire row
public void dropNaN()
//...
```

Getting weather data

Now that we have the tools to get the data, let's find some datasets on the Internet. In this chapter, we are going to use the data from the Brazilian Institute of Meteorology (INMET: <http://www.inmet.gov.br> in Portuguese), which is freely available on the Internet; we have the rights to use it in this book. However, the reader may use any free weather database on the Internet while developing applications.

Some examples from English language sources are listed below:

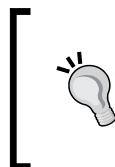
- Wunderground (<http://wunderground.com/>)
- Open Weather Map (<http://openweathermap.org/api>)
- Yahoo weather API (<https://developer.yahoo.com/weather/>)
- US National Climatic Data Center (<http://www.ncdc.noaa.gov/>)

Weather variables

Any weather database will have almost the same variables:

- Temperature (°C)
- Humidity (%)
- Pressure (mbar)
- Wind speed (m/s)
- Wind direction (°)
- Precipitation (mm)
- Sunny hours (h)
- Sun energy (W/m²)

This data is usually collected from meteorological stations, satellites, or radar, on an hourly or daily basis.



Depending on the collection frequency, some variables may be summarized with average, minimum, or maximum values. Data units may also vary from source to source; that's why units should always be observed.

Choosing input and output variables

Selecting the appropriate data that fulfills most of the system's dynamics needs to be carefully done. We want the neural network to forecast future weather based on the current and past weather data, but which variables should we choose? Getting an expert opinion on the subject can be really helpful in understanding the relationship between variables.

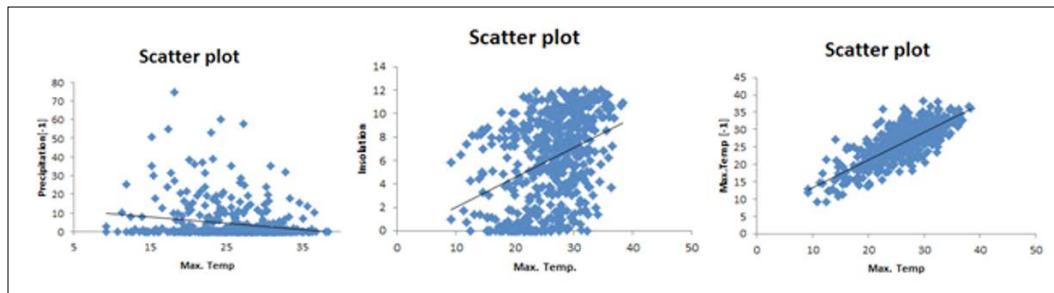


Regarding time series variables, one can derive new variables by applying historical data. That means, given a certain date, one may consider this date's values and the data collected (and/or summarized) from past dates, therefore extending the number of variables.

While defining a problem to use neural networks on, there are one or more predefined target variables: predict the temperature, forecast precipitation, measure insolation, and so on. But, in some cases, one wants to model all the variables and therefore to find causal relationships between them. Causal relationships can be identified by statistical tools, of which Pearson cross-correlation is the most used:

$$C_{x,y} = \frac{E[X \cdot Y] - E[X]E[Y]}{\sigma_x \sigma_y}$$

Here, $E[X \cdot Y]$ is the mean of the multiplication of variables X and Y ; $E[X]$ and $E[Y]$ are the means of X and Y respectively; σ_x and σ_y are the standard deviation of X and Y respectively; and finally $C_{x,y}$ is the Pearson coefficient of X related to Y , whose values lie between -1 and 1. This coefficient shows how much a variable X is correlated with a variable Y . Values near 0 denote weak or no correlation at all, while values close to -1 or 1 denote negative or positive correlation, respectively. Graphically, it can be seen by a scatter plot, as shown below:



In the chart on the left, the correlation between the precipitation of the last day (indicated as [-1]) and the maximum temperature is -0.202, which is a weak value of negative correlation. In the middle chart, the correlation between insolation and maximum temperature is 0.376, which is a fair correlation, yet not very significant; one can see a slight positive trend. An example of strong positive correlation is shown in the chart on the right, which is between the last day's maximum temperature and the maximum temperature of the day. This correlation is 0.793, and we can see a thinner cloud of dots indicating the trend.

We are going to use correlation to choose the most appropriate inputs for our neural network. First, we need to code a method in the class `DataSet`, called `correlation`. Please note that operations such as mean and standard deviation are implemented in our class `ArrayOperations`:

```
public double correlation(int colx,int coly){  
    double[] arrx = ArrayOperations.getColumn(data,colx);  
    double[] arry = ArrayOperations.getColumn(data,coly);  
    double[] arrxy = ArrayOperations.elementWiseProduct(arrx, arry);  
    double meanxy = ArrayOperations.mean(arrxy);  
    double meanx = ArrayOperations.mean(arrx);  
    double meany = ArrayOperations.mean(arry);  
    double stdx = ArrayOperations.stdev(arrx);  
    double stdy = ArrayOperations.stdev(arry);  
    return (meanxy*meanx*meany)/(stdx*stdy);  
}
```

We will not delve too deeply into statistics in this book, so we recommend a number of references if the reader is interested in more details on this topic.

Preprocessing

Raw data collected from a data source usually presents different particularities, such as data range, sampling, and category. Some variables result from measurements while others are summarized or even calculated. Preprocessing means to adapt these variable values to a range that neural networks can handle properly.

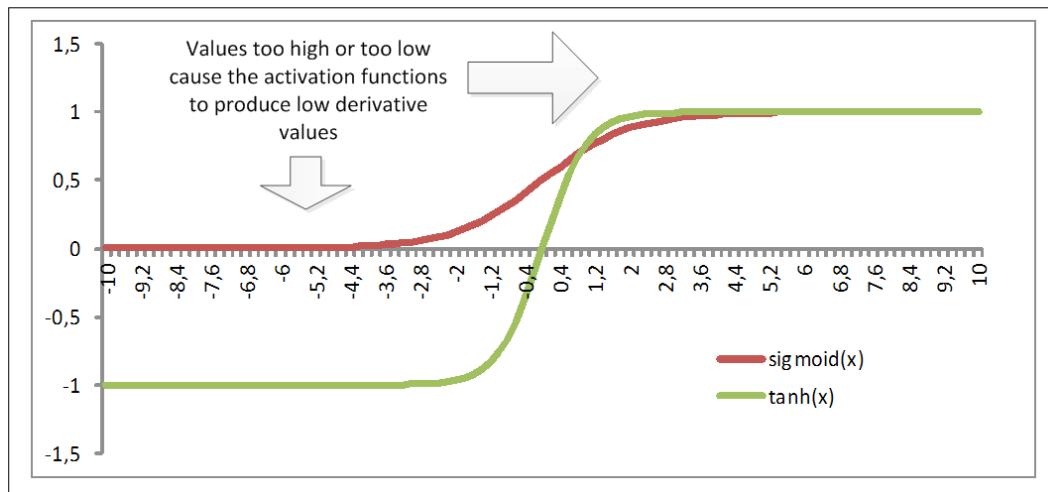
Regarding weather variables, let's take a look at their range, sampling, and type:

Variable	Unit	Range	Sampling	Type
Mean temperature	°C	10.86 – 29.25	Hourly	Average of hourly measurements
Precipitation	mm	0 – 161.20	Daily	Accumulation of daily rain
Insolation	hours	0 – 10.40	Daily	Count of hours receiving sun radiation
Mean humidity	%	45.00 – 96.00	Hourly	Average of hourly measurements
Mean wind speed	km/h	0.00 – 3.27	Hourly	Average of hourly measurements

Except for insolation and precipitation, the variables are all measured and share the same sampling, but if we wanted, for example, to use an hourly dataset, we would have to preprocess all the variables to use the same sample rate. Three of the variables are summarized, using daily average values, but if we wanted to we could use hourly data measurements. However, the range would certainly be larger.

Normalization

Normalization is the process of getting all variables into the same data range, usually with smaller values, between 0 and 1 or -1 and 1. This helps the neural network to present values within the variable zone in activation functions such as sigmoid or hyperbolic tangent:



Values too high or too low may drive neurons into producing values too high or too low as well for the activation functions, therefore leading to the derivative for these neurons being too small, near zero. In this book, we implemented two modes of normalization: min-max and z-score.

The min-max normalization should consider a predefined range of the dataset. It is performed right away:

$$X_{norm} = (N_{\max} - N_{\min}) \left[\frac{(X - X_{\min})}{(X_{\max} - X_{\min})} \right] + N_{\min}$$

Here, N_{\min} and N_{\max} are the normalized minimum and maximum limits respectively, X_{\min} and X_{\max} are the variable X's minimum and maximum limits respectively, X is the original value, and X_{norm} is the normalized value. If we want the normalization to be between 0 and 1, for example, the equation is simplified to the following:

$$X_{\text{norm}} = \frac{(X - X_{\min})}{(X_{\max} - X_{\min})}$$

By applying the normalization, a new *normalized* dataset is produced and is fed to the neural network. One should also take into account that a neural network fed with normalized values will be trained to produce normalized values on the output, so the inverse (denormalization) process becomes necessary as well:

$$X = (X_{\max} - X_{\min}) \left[\frac{(X_{\text{norm}} - N_{\min})}{(N_{\max} - N_{\min})} \right] + X_{\min}$$

Or

$$X = (X_{\max} - X_{\min}) [X_{\text{norm}}] + X_{\min}$$

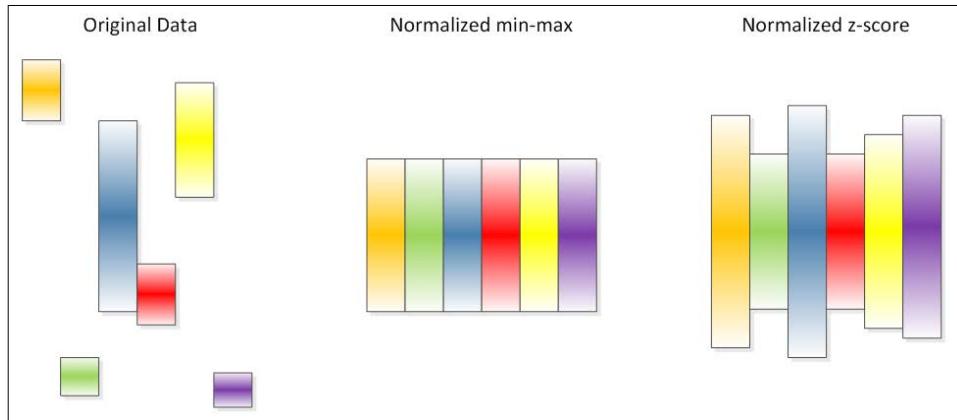
For the normalization between 0 and 1.

Another mode of normalization is the z-score, which takes into account the mean and standard deviation:

$$X_{\text{norm}} = S \left[\frac{X - E[X]}{\sigma_X} \right]$$

Here, S is a scaling constant, $E[X]$ is the mean of E, and σ_X is the standard deviation of X. The main difference in this normalization mode is that there will be no limit defined for the range of variables; however, the variables will have values on the same range centered on zero with standard deviation equal to the scaling constant S.

The figure below shows what both normalization modes do with the data:



A class called `DataNormalization` is implemented to handle the normalization of data. Since normalization considers the statistical properties of the data, we need to store this statistical information in a `DataNormalization` object:

```
public class DataNormalization {
    //ENUM normalization types
    public enum NormalizationTypes { MIN_MAX, ZSCORE }
    // normalization type
    public NormalizationTypes TYPE;
    //statistical properties of the data
    private double[] minValue;
    private double[] maxValue;
    private double[] meanValue;
    private double[] stdValue;
    //normalization properties
    private double scaleNorm=1.0;
    private double minNorm=-1.0;
    //...
    //constructor for min-max norm
    public DataNormalization(double[][] data,double _minNorm, double
    _maxNorm) {
        this.TYPE=NormalizationTypes.MIN_MAX;
        this.minNorm=_minNorm;
        this.scaleNorm=_maxNorm-_minNorm;
        calculateReference(data);
    }
    //constructor for z-score norm
    public DataNormalization(double[][] data,double _zscale) {
```

```
    this.TYPE=NormalizationTypes.ZSCORE;
    this.scaleNorm=_zscale;
    calculateReference(data);
}
//calculation of statistical properties
private void calculateReference(double[][] data) {
    minValues=ArrayOperations.min(data);
    maxValues=ArrayOperations.max(data);
    meanValues=ArrayOperations.mean(data);
    stdValues=ArrayOperations.stdev(data);
}
//...
}
```

The normalization procedure is performed on a method called `normalize`, which has a denormalization counterpart called `denormalize`:

```
public double[][] normalize( double[][] data ) {
    int rows = data.length;
    int cols = data[0].length;
    //...
    double[][] normalizedData = new double[rows][cols];
    for(int i=0;i<rows;i++) {
        for(int j=0;j<cols;j++) {
            switch (TYPE) {
                case MIN_MAX:
                    normalizedData[i][j]=(minNorm) + ((data[i][j] -
minValues[j]) / ( maxValues[j] - minValues[j] )) * (scaleNorm);
                    break;
                case ZSCORE:
                    normalizedData[i][j]=scaleNorm * (data[i][j] -
meanValues[j]) / stdValues[j];
                    break;
            }
        }
    }
    return normalizedData;
}
```

Adapting NeuralDataSet to handle normalization

The already implemented `NeuralDataSet`, `NeuralInputData`, and `NeuralOutputData` will now have `DataNormalization` objects to handle normalization operations. In the `NeuralDataSet` class, we've added objects for input and output data normalization:

```
public DataNormalization inputNorm;
public DataNormalization outputNorm;
//zscore normalization
public void setNormalization(double _scaleNorm) {
    inputNorm = new DataNormalization(_scaleNorm);
    inputData.setNormalization(inputNorm);
    outputNorm = new DataNormalization(_scaleNorm);
    outputData.setNormalization(outputNorm);
}
//min-max normalization
public void setNormalization(double _minNorm,double _maxNorm) {
    inputNorm = new DataNormalization(_minNorm,_maxNorm);
    inputData.setNormalization(inputNorm);
    outputNorm = new DataNormalization(_minNorm,_maxNorm);
    outputData.setNormalization(outputNorm);
}
```

`NeuralInputData` and `NeuralOutputData` will now have `normdata` properties to store the normalized data. The methods to retrieve data from these classes will have a Boolean parameter, `isNorm`, to indicate whether the value to be retrieved should be normalized or not.

Considering that `NeuralInputData` will provide the neural network with input data, this class will only perform normalization before feeding data into the neural network. The method `setNormalization` is implemented in this class to that end:

```
public ArrayList<ArrayList<Double>> normdata;
public DataNormalization norm;
public void setNormalization(DataNormalization dn) {
    //getting the original data into java matrix
    double[][] origData = ArrayOperations.
    arrayListToDoubleMatrix(data);
    //perform normalization
    double[][] normData = dn.normalize(origData);
    normdata=new ArrayList<>();
    //store the normalized values into ArrayList normdata
    for(int i=0;i<normData.length;i++) {
```

```
        normData.add(new ArrayList<Double>());
        for(int j=0;j<normData[0].length;j++) {
            normData.get(i).add(normData[i][j]);
        }
    }
}
```

In `NeuralOutputData`, there are two datasets, one for the target and one for the neural network output. The target dataset is normalized to provide the training algorithm with normalized values. However, the neural output dataset is the output of the neural network, that is, it will be normalized first. We need to perform denormalization after setting the neural network output dataset:

```
public ArrayList<ArrayList<Double>> normTargetData;
public ArrayList<ArrayList<Double>> normNeuralData;
public void setNeuralData(double[][] _data,boolean isNorm) {
    if(isNorm){ //if is normalized
        this.normNeuralData=new ArrayList<>();
        for(int i=0;i<numberOfRecords;i++){
            this.normNeuralData.add(new ArrayList<Double>());
            //... save in the normNeuralData
            for(int j=0;j<numberOfOutputs;j++){
                this.normNeuralData.get(i).add(_data[i][j]);
            }
        }
        double[][] deNorm = norm.denormalize(_data);
        for(int i=0;i<numberOfRecords;i++)
            for(int j=0;j<numberOfOutputs;j++) //then in neuralData
                this.neuralData.get(i).set(j,deNorm[i][j]);
    }
    else setNeuralData(_data);
}
```

Adapting the learning algorithm to normalization

Finally, the `LearningAlgorithm` class needs to include the normalization property:

```
protected boolean normalization=false;
```

Now during the training, on every call to the `NeuralDataSet` methods that retrieve or write data, the normalization property should be passed in the parameter `isNorm`, as in the method `forward` of the class `Backpropagation`:

```
@Override
public void forward(){
```

```

for(int i=0;i<trainingDataSet.numberOfRecords;i++) {
    neuralNet.setInputs(trainingDataSet.
        getInputRecord(i,normalization));
    neuralNet.calc();
    trainingDataSet.setNeuralOutput(i, neuralNet.getOutputs(),
        normalization);
//...
}
}

```

Java implementation of weather forecasting

In Java, we are going to use the package `edu.packt.neuralnet.chart` to plot some charts and visualize data. We're also downloading historical meteorology data from INMET, the Brazilian Institute of Meteorology. We've downloaded data from several cities, so we could have a variety of climates included in our weather forecasting case.



In order to run the training expeditiously, we have selected a small period (5 years), which has more than 2,000 samples.



Collecting weather data

In this example, we wanted to collect a variety of data from different places, to attest to the capacity of the neural network to forecast it. Since we downloaded it from the INMET website, which covers only Brazilian territory, only Brazilian cities are covered. However, it is a very vast territory with a great variety of climates. Below is a list of places we collected data from:

#	City Name	Latitude	Longitude	Altitude	Climate Type
1	Cruzeiro do Sul	7°37'S	72°40'W	170 m	Tropical Rainforest
2	Picos	7°04'S	41°28'W	208 m	Semi-arid
3	Campos do Jordão	22°45'S	45°36'W	1642 m	Subtropical Highland
4	Porto Alegre	30°01'S	51°13'W	48 m	Subtropical Humid

The location of these four cities is indicated on the map below:



Source: Wikipedia, user NordNordWest using United States National Imagery and Mapping Agency data, World Data Base II data

The weather data collected is from January 2010 until November 2016 and is saved in the data folder with the name corresponding to the city.

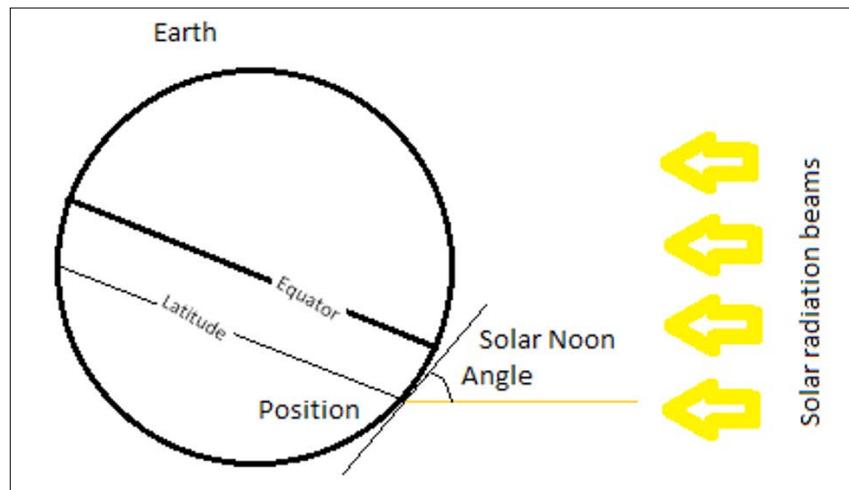
The data collected from the INMET website includes these variables:

- Precipitation (mm)
- Max. temperature (°C)
- Min. temperature (°C)
- Insolation (sunny hours)
- Evaporation (mm)
- Avg. temperature (°C)
- Avg. humidity (%)
- Avg. wind speed (mph)
- Date (converted into Excel number format)
- Position of the station (latitude, longitude, and altitude)

For each city, we are going to build a neural network to forecast the weather based on the past. But first, we need to point out two important facts:

- Cities located in high latitudes experience high weather variations due to the seasons; that is, the weather will be dependent on the date
- The weather is a very dynamic system whose variables are influenced by past values

To overcome the first issue, we may derive a new column from the date to indicate the solar noon angle, which is the angle at which the solar rays reach the surface at the city at the highest point in the sky (noon). The greater this angle, the more intense and warm the solar radiation is; on the other hand, when this angle is small, the surface will receive a small fraction of the solar radiation:



The solar noon angle is calculated by the following formula and Java implementation in the class `WeatherExample`, which will be used in this chapter:

$$\alpha_{noon} = 90 - \left| -23.44 \cos\left(\left(\frac{2\pi}{365.25}\right)(D+8.5)\right) - lat \right|$$

```
public double calcSolarNoonAngle(double date, double latitude) {
    return 90 - Math.abs(-23.44 * Math.cos((2 * Math.PI / 365.25) * (date + 8.5)) -
        latitude);
}
public void addSolarNoonAngle(TimeSeries ts, double latitude) { // to add
    column
```

```
double[] sna = new double[ts.numberOfRecords];
for(int i=0;i<ts.numberOfRecords;i++)
    sna[i]=calcSolarNoonAngle(
        ts.data.get(i).get(ts.getIndexColumn()), latitude);
ts.addColumn(sna, "NoonAngle");
}
```

Delaying variables

In the class `WeatherExample`, let's place a method called `makeDelays`, which will later be called from the main method. The delays will be made on a given `TimeSeries` and up to a given number for all columns of the time series except that of the index column:

```
public void makeDelays(TimeSeries ts,int maxdelays){
    for(int i=0;i<ts.numberOfColumns;i++)
        if(i!=ts.getIndexColumn())
            for(int j=1;j<=maxdelays;j++)
                ts.shift(i, -j);
}
```



Be careful not to call this method multiple times; it may delay the same column over and over again.



Loading the data and beginning to play!

In the `WeatherExample` class, we are going to add four `TimeSeries` properties and four `NeuralNet` properties for each case:

```
public class WeatherExample {

    TimeSeries cruceirodosul;
    TimeSeries picos;
    TimeSeries camposdojordao;
    TimeSeries portoalegre;

    NeuralNet nncruzeirosul;
    NeuralNet nnpicos;
    NeuralNet nncamposjordao;
    NeuralNet nnportoalegre;
    //...
}
```

In the `main` method, we load data to each of them and delay the columns up to three days before:

```
public static void main(String[] args) {
    WeatherExample we = new WeatherExample();
    //load weather data
    we.cruzeirodosul = new TimeSeries(LoadCsv.getDataSet("data",
"cruzeirodosul2010daily.txt", true, ";"));
    we.cruzeirodosul.setIndexColumn(0);
    we.makeDelays(we.cruzeirodosul, 3);

    we.picos = new TimeSeries(LoadCsv.getDataSet("data",
"picos2010daily.txt", true, ";"));
    we.picos.setIndexColumn(0);
    we.makeDelays(we.picos, 3);

    we.camposdojordao = new TimeSeries(LoadCsv.getDataSet("data",
"camposdojordao2010daily.txt", true, ";"));
    we.camposdojordao.setIndexColumn(0);
    we.makeDelays(we.camposdojordao, 3);

    we.portoalegre = new TimeSeries(LoadCsv.getDataSet("data",
"portoalegre2010daily.txt", true, ";"));
    we.portoalegre.setIndexColumn(0);
    we.makeDelays(we.portoalegre, 3);
//...
```



This piece of code can take a couple of minutes to execute, given that each file may have more than 2,000 rows.

After loading, we need to remove the NaNs, so we call the method `dropNaN` from each time series object:

```
//...
we.cruzeirodosul.dropNaN();
we.camposdojordao.dropNaN();
we.picos.dropNaN();
we.portoalegre.dropNaN();
//...
```

To save time and effort for future executions, let's save these datasets:

```
we.cruzeirodosul.save("data", "cruzeirodosul2010daily_delays_clean.  
txt", ";");  
//...  
we.portoalegre.save("data", "portoalegre2010daily_delays_clean.  
txt", ";");
```

Now, for all-time series, each column has three delays, and we want the neural network to forecast the maximum and minimum temperature of the next day. We can forecast the future by taking into account only the present and the past, so for inputs we must rely on the delayed data (from -1 to -3 days before), and for outputs we may consider the current temperature values. Each column in the time series dataset is indicated by an index, where zero is the index of the date. Since some of the datasets had missing data on certain columns, the index of a column may vary. However, the index for output variables is the same through all datasets (indexes 2 and 3).

Let's perform a correlation analysis

We are interested in finding patterns between the delayed data and the current maximum and minimum temperature. So we perform a cross-correlation analysis combining all output and potential input variables, and select the variables that present at least a minimum absolute correlation as a threshold. So we write a method `correlationAnalysis` taking the minimum absolute correlation as the argument. To save space, we have trimmed the code here:

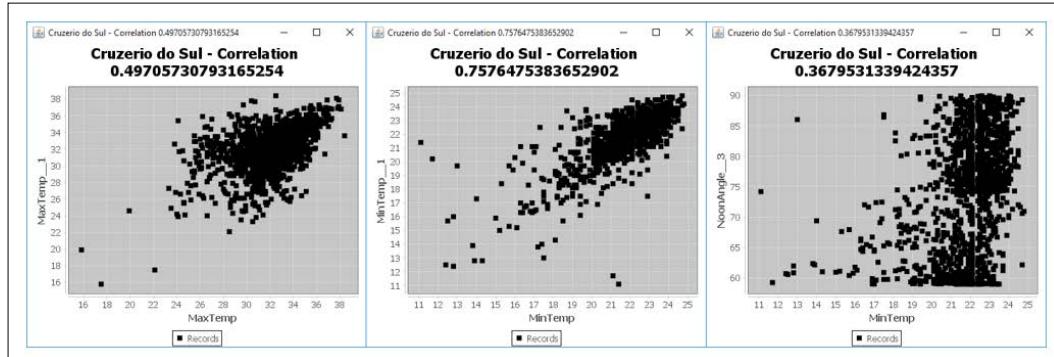
```
public void correlationAnalysis(double minAbsCorr) {  
    //indexes of output variables (max. and min. temperature)  
    int[][] outputs = {  
        {2,3}, //cruzeiro do sul  
        {2,3}, //picos  
        {2,3}, //campos do jordao  
        {2,3}}; //porto alegre  
    int[][] potentialInputs = { //indexes of input variables (delayed)  
        {10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,  
        29,30,31,32,33,34,38,39,40}, //cruzeiro do sul  
        //... and all others  
    };
```

```
ArrayList<ArrayList<ArrayList<Double>>> chosenInputs = new
ArrayList<>();
TimeSeries[] tscollect = {this.cruzeirodosul,this.picos,this.
camposdojordao,this.portoalegre};
double[][][] correlation = new double[4][][];
for(int i=0;i<4;i++){
    chosenInputs.add(new ArrayList<ArrayList<Double>>());
    correlation[i]=new double[outputs[i].length][potentialInputs[i].
length];
    for(int j=0;j<outputs[i].length;j++){
        chosenInputs.get(i).add(new ArrayList<Double>());
        for(int k=0;k<potentialInputs[i].length;k++){
            correlation[i][j][k]=tscollect[i].correlation(outputs[i][j],
potentialInputs[i][k]);
            //if the absolute correlation is above the threshold
            if(Math.abs(correlation[i][j][k])>minAbsCorr){
                //it is added to the chosen inputs
                chosenInputs.get(i).get(j).add(correlation[i][j][k]);
                //and we see the plot
                tscollect[i].getScatterChart("Correlation "+String.
valueOf(correlation[i][j][k]), outputs[i][j], potentialInputs[i][k],
Color.BLACK).setVisible(true);
            }
        }
    }
}
```

By running this analysis, we receive the following result for Cruzeiro do Sul (the bold columns are chosen as neural network inputs):

Correlation Analysis for data from Cruzeiro do Sul	Correlations with the output Variable:
Correlations with the output Variable:	MinTemp
MaxTemp	NoonAngle:0.346545
NoonAngle:0.0312808	Precipitation__1:0.012696
Precipitation__1:-0.115547	Precipitation__2:0.063303
Precipitation__2:-0.038969	Precipitation__3:0.112842
Precipitation__3:-0.062173	MaxTemp__1:0.311005
MaxTemp__1:0.497057	MaxTemp__2:0.244364
MaxTemp__2:0.252831	MaxTemp__3:0.123838
MaxTemp__3:0.159098	MinTemp__1:0.757647
MinTemp__1:-0.033339	MinTemp__2:0.567563
MinTemp__2:-0.123063	MinTemp__3:0.429669
MinTemp__3:-0.125282	Insolation__1:-0.10192
Insolation__1:0.395741	Insolation__2:-0.101146
Insolation__2:0.197949	Insolation__3:-0.151896
Insolation__3:0.134345	Evaporation__1:-0.115236
Evaporation__1:0.21548	Evaporation__2:-0.160718
Evaporation__2:0.161384	Evaporation__3:-0.160536
Evaporation__3:0.199385	AvgTemp__1:0.633741
AvgTemp__1:0.432280	AvgTemp__2:0.487609
AvgTemp__2:0.152103	AvgTemp__3:0.312645
AvgTemp__3:0.060368	AvgHumidity__1:0.151009
AvgHumidity__1:-0.415812	AvgHumidity__2:0.155019
AvgHumidity__2:-0.265189	AvgHumidity__3:0.177833
AvgHumidity__3:-0.214624	WindSpeed__1:-0.198555
WindSpeed__1:-0.166418	WindSpeed__2:-0.227227
WindSpeed__2:-0.056825	WindSpeed__3:-0.185377
WindSpeed__3:-0.001660	NoonAngle__1:0.353834
NoonAngle__1:0.0284473	NoonAngle__2:0.360943
NoonAngle__2:0.0256710	NoonAngle__3:0.367953
NoonAngle__3:0.0227864	

The scatter plots show how this data is related:



On the left, there is a fair correlation between the last day's maximum temperature and the current; in the center, a strong correlation between the last day's minimum temperature and the current; and on the right, a weak correlation between NoonAngle of 3 days before and the current minimum temperature. By running this analysis for all other cities, we determine the inputs for the other neural networks:

Cruzeiro do Sul	Picos	Campos do Jordão	Porto Alegre
NoonAngle	MaxTemp	NoonAngle	MaxTemp
MaxTemp_1	MaxTemp_1	MaxTemp_1	NoonAngle
MinTemp_1	MaxTemp_2	MaxTemp_2	MaxTemp_1
MinTemp_2	MaxTemp_3	MaxTemp_3	MaxTemp_2
MinTemp_3	MinTemp_1	MinTemp_1	MaxTemp_3
Insolation_1	MinTemp_2	MinTemp_2	MinTemp_1
AvgTemp_1	MinTemp_3	MinTemp_3	MinTemp_2
AvgTemp_2	Insolation_1	Evaporation_1	MinTemp_3
AvgHumidity_1	Insolation_2	AvgTemp_1	Insolation_1
NoonAngle_1	Evaporation_1	AvgTemp_2	Insolation_2
NoonAngle_2	Evaporation_2	AvgTemp_3	Insolation_3
NoonAngle_3	Evaporation_3	AvgHumidity_1	Evaporation_1
	AvgTemp_1	NoonAngle_1	Evaporation_2
	AvgTemp_2	NoonAngle_2	Evaporation_3
	AvgTemp_3	NoonAngle_3	AvgTemp_1
	AvgHumidity_1		AvgTemp_2
	AvgHumidity_2		AvgTemp_3
	AvgHumidity_3		AvgHumidity_1
			AvgHumidity_2
			NoonAngle_1
			NoonAngle_2
			NoonAngle_3

Creating neural networks

We are using four neural networks to forecast the minimum and maximum temperature. Initially, they will have two hidden layers with 20 and 10 neurons each and hypertan and sigmoid activation functions. We will apply min-max normalization. The following method in the class `WeatherExample` creates the neural networks with this configuration:

```
public void createNNs() {
    //fill a vector with the indexes of input and output columns
    int[] inputColumnsCS = {10,14,17,18,19,20,26,27,29,38,39,40};
    int[] outputColumnsCS = {2,3};
    //this static method hashes the dataset
    NeuralDataSet[] nnttCS = NeuralDataSet.randomSeparateTrainTest(this.
        cruzeirodosul, inputColumnsCS, outputColumnsCS, 0.7);
    //setting normalization
    DataNormalization.setNormalization(nnttCS, -1.0, 1.0);

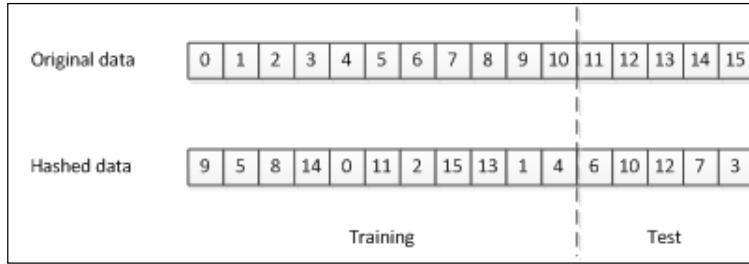
    this.trainDataCS = nnttCS[0]; // 70% for training
    this.testDataCS = nnttCS[1]; // rest for test

    //setup neural net parameters:
    this.nncruzeirosul = new NeuralNet( inputColumnsCS.length,
        outputColumnsCS.length, new int[]{20,10}
            , new IActivationFunction[] {new HyperTan(1.0),new Sigmoid(1.0)}
            , new Linear()
            , new UniformInitialization(-1.0, 1.0) );
    //...
}
```

Training and test

In *Chapter 2, Getting Neural Networks to Learn* we have seen that a neural network should be tested to verify its learning, so we divide the dataset into training and testing subsets. Usually about 50-80% of the original filtered dataset is used for training and the remaining fraction is for testing.

A static method `randomSeparateTrainTest` in the class `NeuralDataSet` separates the dataset into these two subsets. In order to ensure maximum generalization, the records of this dataset are hashed, as shown in the following figure:



The records may be originally sequential, as in weather time series; if we hash them in random positions, the training and testing sets will contain records from all periods.

Training the neural network

The neural network will be trained using the basic backpropagation algorithm. The following is a code sample for the dataset Cruzeiro do Sul:

```

Backpropagation bpCS = new Backpropagation(we.nncruzeirosul
    ,we.trainDataCS
    ,LearningAlgorithm.LearningMode.BATCH);
bpCS.setTestingDataSet(we.testDataCS);
bpCS.setLearningRate(0.3);
bpCS.setMaxEpochs(1000);
bpCS.setMinOverallError(0.01); //normalized error
bpCS.printTraining = true;
bpCS.setMomentumRate( 0.3 );

try{
    bpCS.forward();
    bpCS.train();

    System.out.println("Overall Error:" + String.valueOf(bpCS.
getOverallGeneralError()));
    System.out.println("Testing Error:" + String.valueOf(bpCS.
getTestingOverallGeneralError()));
    System.out.println("Min Overall Error:" + String.valueOf(bpCS.
getMinOverallError()));
    System.out.println("Epochs of training:" + String.valueOf(bpCS.
getEpoch()));
}
catch(NeuralException ne){ }

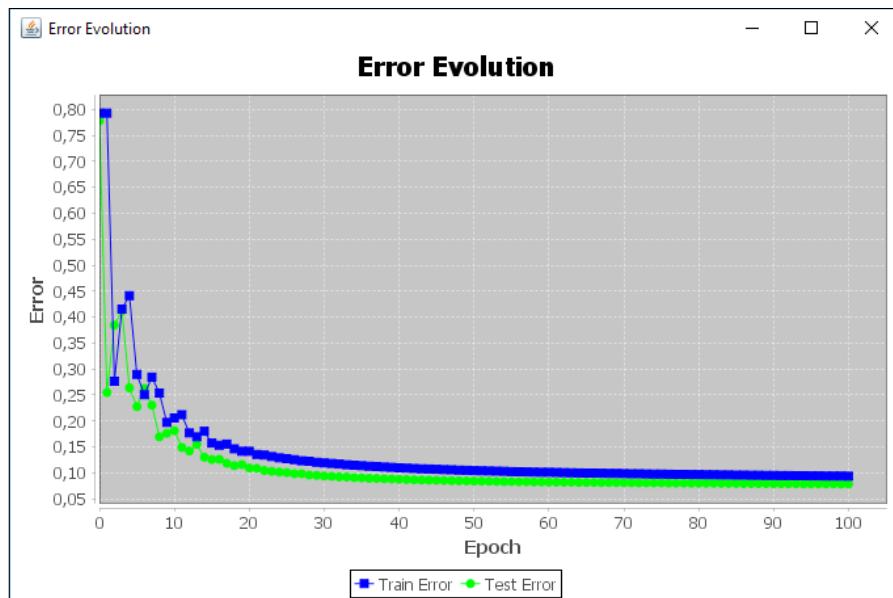
```

Plotting the error

Using the JFreeCharts framework, we can plot error evolution for the training and testing datasets. There is a new method in the class `LearningAlgorithm` called `showErrorEvolution`, which is inherited and overridden by `BackPropagation`. To see the chart, just call as in the example:

```
//plot list of errors by epoch  
bpCS.showErrorEvolution();
```

This will show a plot like the one shown in the following figure:



Viewing the neural network output

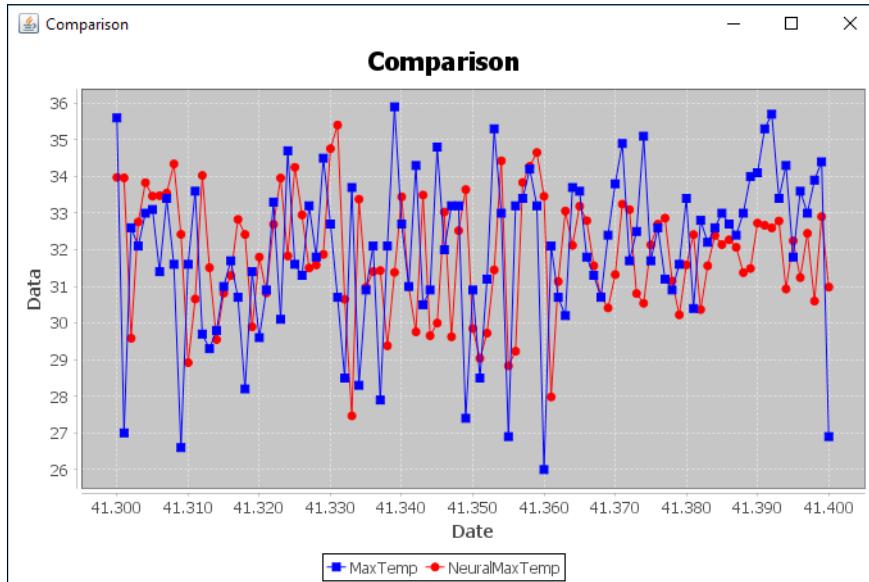
Using this same facility, it is easy to see and compare the neural network output. First, let's transform the neural network output into vector form and add to our dataset using the method `addColumn`. Let's name it `NeuralMinTemp` and `NeuralMaxTemp`:

```
String[] neuralOutputs = { "NeuralMaxTemp", "NeuralMinTemp" };  
we.cruzeirodosul.addColumn(we.fullDataCS.getIthNeuralOutput(0),  
neuralOutputs[0]);  
we.cruzeirodosul.addColumn(we.fullDataCS.getIthNeuralOutput(1),  
neuralOutputs[1]);  
String[] comparison = { "MaxTemp", "NeuralMaxTemp" };  
Paint[] comp_color = {Color.BLUE, Color.RED};
```

```
final double minDate = 41200.0;
final double maxDate = 41300.0;
```

The class `TimeSeries` has a method called `getTimePlot`, which is used to plot variables over a specified range:

```
ChartFrame viewChart = we.cruzeirodosul.getTimePlot("Comparison",
comparison, comp_color, minDate, maxDate);
```



Empirical design of neural networks

While using neural networks in regression problems (that include prediction), there is no fixed number of hidden neurons, so usually the solver chooses an arbitrary number of neurons and then varies it according to the results produced by the networks created. This procedure may be repeated a number of times until a network with a satisfying criterion is found.

Designing experiments

Experiments can be made on the same training and test datasets, while varying other network parameters, such as learning rate, normalization, and the number of hidden units. The objective is to choose the neural network that presents the best performance from the experiments. The best performance is assigned to the network that presents a lower MSE error, but an analysis of generalization with test data is also useful.



While designing experiments, consider always starting from a lower number of hidden neurons, since it is desirable to have a lower computational processing consumption.

The table below shows the experiments have that been run for all cities:

Experiment	Number of neurons in hidden layer	Learning rate	Data normalization type
#1	5	0.1	MIN_MAX [-1, 1]
#2			Z-SCORE
#3		0.5	MIN_MAX [-1, 1]
#4			Z-SCORE
#5		0.9	MIN_MAX [-1, 1]
#6			Z-SCORE
#7	10	0.1	MIN_MAX [-1, 1]
#8			Z-SCORE
#9		0.5	MIN_MAX [-1, 1]
#10			Z-SCORE
#11		0.9	MIN_MAX [-1, 1]
#12			Z-SCORE

Results and simulations

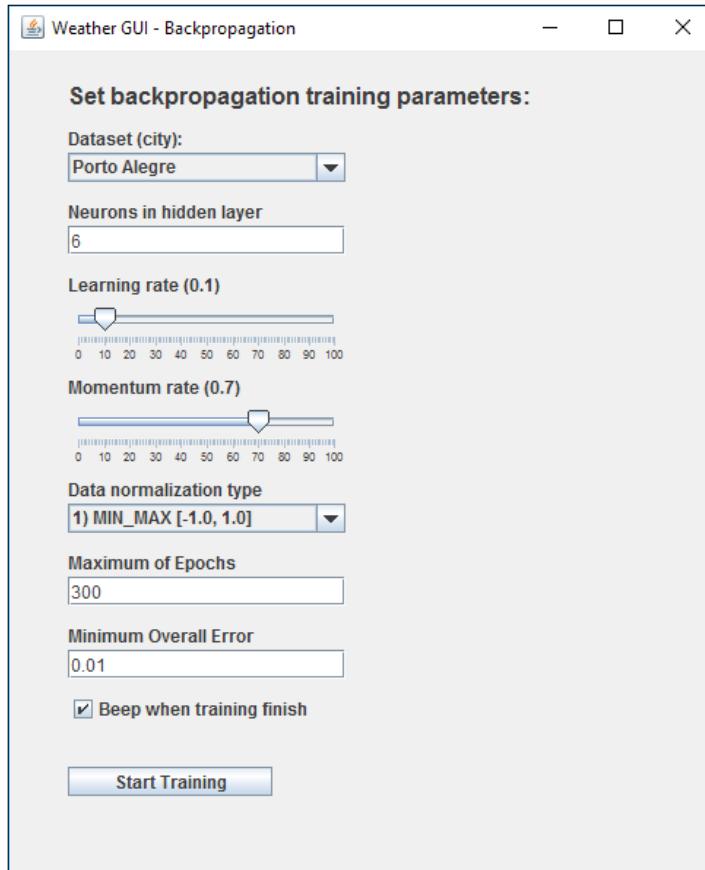
In order to facilitate the execution of experiments, we've designed a Java Swing **Graphical User Interface (GUI)**, with which it is possible to select neural network parameters for training and the dataset.



This interface covers only neural networks with just one hidden layer; however, since the code is open, the implementation of a multilayer perceptron with multiple hidden layers is suggested as an exercise, as well as the choice of other algorithms for the training.

The charts show only the predicted maximum temperature; therefore, implementing an option for displaying the minimum temperature is also suggested.

After selecting the parameters, training begins when you click the **Start Training** button:

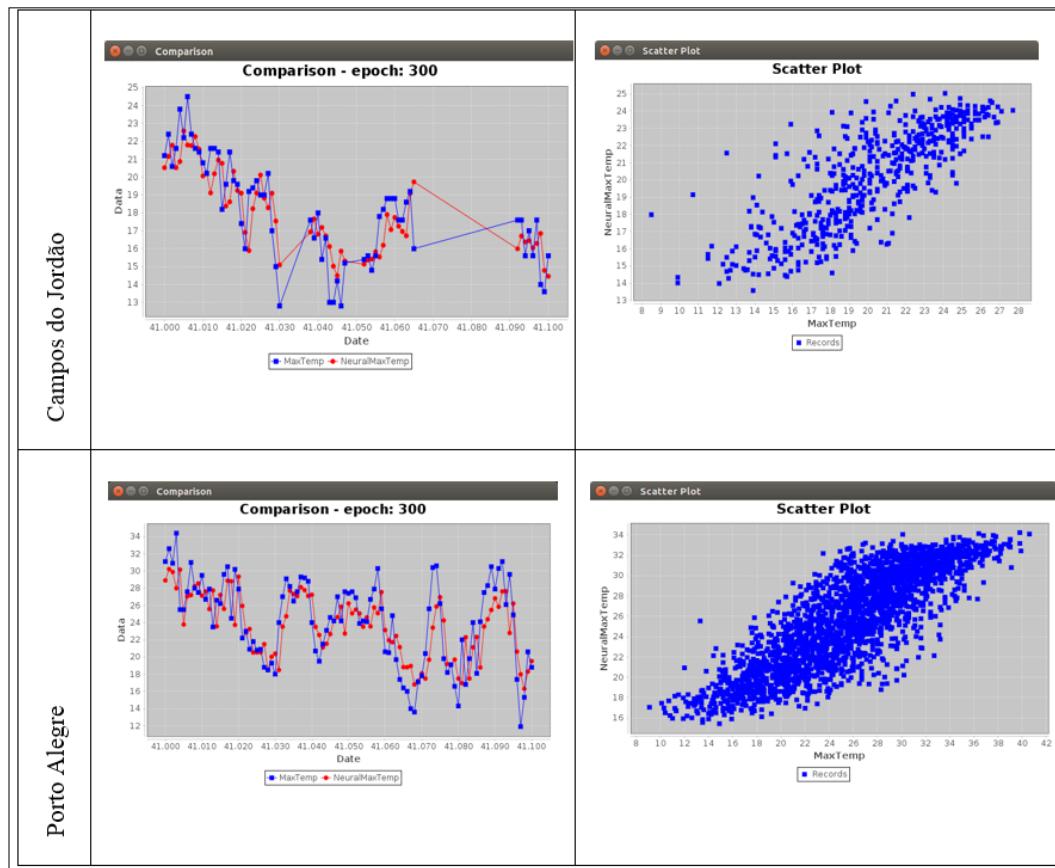


After running 12 experiments, we found the following MSE training errors for each dataset:

Experiment	Cruzeiro do Sul	Picos	Campos do Jordão	Porto Alegre
#1	0.130156	0.147111	0.300437	0.323342
#2	0.512389	0.572588	0.428692	0.478379
#3	0.08659	0.094822	0.124752	0.114486
#4	0.360728	0.258596	0.168351	0.192012
#5	0.076476	0.074777	0.108991	0.085029
#6	0.328493	0.186793	0.152499	0.151248

Experiment	Cruzeiro do Sul	Picos	Campos do Jordão	Porto Alegre
#7	0.146801	0.130004	0.277765	0.19076
#8	0.431811	0.29629	0.364418	0.278864
#9	0.071135	0.081159	0.117634	0.091174
#10	0.332534	0.210107	0.170179	0.164179
#11	0.07247	0.089069	0.102137	0.076578
#12	0.33342	0.19835	0.155036	0.145843

The MSE error information only gives us an idea of how much the neural network output could match real data in the overall context. This performance can be verified by viewing the time series comparison and scatter plots:



These charts show that, although in many cases the temperature cannot be accurately predicted, a trend is being followed. This can be attested to by the correlation visible in the scatter plots. The last row of the table, showing the prediction for Porto Alegre, which has a subtropical climate and high temperature variations, shows a good prediction even for the extreme temperature variations. However, we remind the reader that forecasting weather needs to consider many additional variables, which could not be included in this example due to availability constraints. Anyway, the results show we've made a good start to search for a neural network configuration that can outperform these ones found.

Summary

In this chapter, we've seen an interesting practical use case for the application of neural networks. Weather forecasting has always been a rich research field, and indeed neural networks are widely used for this purpose. In this chapter, the reader also learned how to prepare similar experiments for prediction problems. The correct application of techniques for data selection and preprocessing can save a lot of time while designing neural networks for prediction. This chapter also served as a foundation for the next ones, since all of them will focus on practical cases; thus, the concepts learned herein will be explored widely in the rest of the book.

6

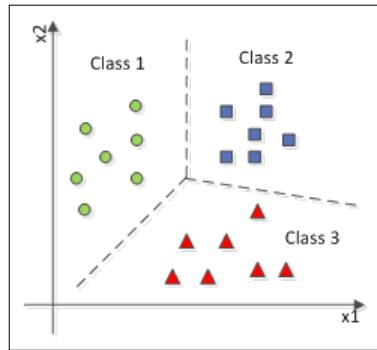
Classifying Disease Diagnosis

So far, we have been working with supervised learning for predicting numerical values; however, in the real world, numbers are just part of the data addressed. Real variables also contain categorical values, which are not purely numerical, but describe important features that have influence on the problems neural networks are applied to solve. In this chapter, the reader will be presented with a very didactic but interesting application involving categorical values and classification: disease diagnosis. This chapter digs deeper into classification problems and how to represent categorical data, as well as showing how to design a classification algorithm using neural networks. The topics covered in this chapter are as follows:

- Foundations of classification problems
- Categorical data
- Logistic regression
- Confusion matrix
- Sensibility and specificity
- Neural networks for classification
- Disease diagnosis using neural networks
- Diagnosis for cancer
- Diagnosis for diabetes

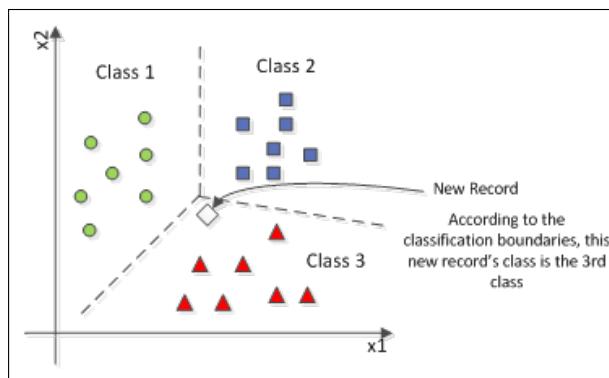
Foundations of classification problems

One thing neural networks are really good at is classifying records. The very simple perceptron network draws a decision boundary, defining whether a data point belongs to one region or another, whereas a region denotes a class. Let's take a look visually on an x - y scatter chart:



The dashed lines explicitly separate the points into classes. These points represent data records which originally had the corresponding class labels. That means their classes were already known, therefore this classification task falls in the supervised learning category.

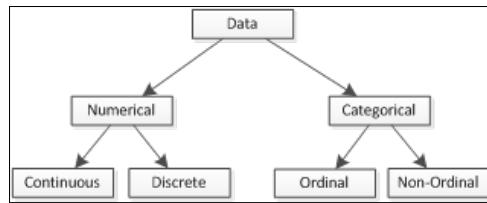
A classification algorithm seeks to find the boundaries between the classes in the data hyperspace. Once the classification boundaries are defined, a new data point, with an unknown class, receives a class label according to the boundaries defined by the classification algorithm. The figure below shows how a new record is classified:



Based on the current class configuration, the new record's class is the third class.

Categorical data

Applications usually lead with the types of data shown in the following figure:



Data can be numerical or categorical or, simply speaking, numbers or words. Numerical data is represented by a numeric value, from which it can be continuous or discrete. This data type has been used so far in this book's applications. Categorical data is a wider class of data that includes words, letters, or even numbers, but with a quite different meaning. While numerical data can support arithmetic operations, categorical data is only descriptive and cannot be processed like numbers, even if the value is a number. An example is the severity degree of a disease in a scale (from zero to five, for example). Another property of categorical data is that a certain variable has a finite number of values; in other words, only a defined set of values can be assigned to a categorical variable. A subclass of data inside the categorical is ordinal data. This class is particular because the defined values can be sorted in a predefined order. An example is adjectives indicating the state or quality of something (bad, fair, good, excellent):

Numerical		Categorical	
Only numbers		Numbers, words, letters, signs	
Can support arithmetic operations		Do not support arithmetic operations	
Infinite or undefined range of values		Finite or defined set of values	
Continuous	Discrete	Ordinal	Non-ordinal
Real values	Integers, decimal	Can be ordered	Cannot be ordered
Any possible value	Predefined intervals	Can be assigned numbers	Each possible value is a flag

[ Note that here we are addressing structured data only. In the real world, most data is unstructured, including text and multimedia content. Although these types of data are also processed in learning from data applications, neural networks require them to be transformed into structured data types.]

Working with categorical data

Structured data files, such as those used in CSV or Excel, usually contain columns of numerical and categorical data. In *Chapter 5, Forecasting Weather* we have created the classes `LoadCsv` (for loading csv files) and `DataSet` (for storing data from csv), but these classes are prepared only for working with numerical data. The simplest way of representing categorical value is converting each possible value into a binary column, whereby if the given value is presented in the original column, the corresponding binary column will have a one as the converted value, otherwise it will be zero:

The diagram illustrates the transformation of a categorical column into a numerical binary matrix. On the left, a vertical table is labeled "Categorical Column". It contains eight rows with values A, B, B, A, C, A, D, and A. An arrow points from this table to a larger table on the right labeled "Numerical Binary". This second table has four columns labeled A, B, C, and D. It contains eight rows of binary values (0 or 1). The first row has 1 in the A column and 0 in others. The second row has 0 in the A column and 1 in the B column. The third row has 0 in the A column and 1 in the C column. The fourth row has 1 in the A column and 0 in others. The fifth row has 0 in the A column and 1 in the C column. The sixth row has 1 in the A column and 0 in others. The seventh row has 0 in the A column and 1 in the D column. The eighth row has 1 in the A column and 0 in others.

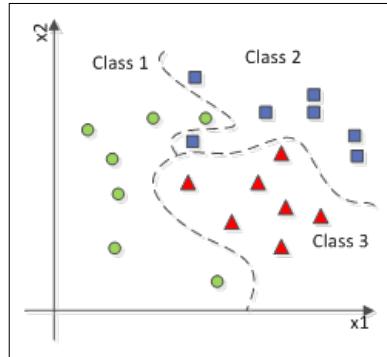
Numerical Binary							
A	B	C	D	A	B	C	D
1	0	0	0	0	1	0	0
0	1	0	0	1	0	0	0
0	1	0	0	0	1	0	0
1	0	0	0	1	0	0	0
0	0	1	0	0	0	1	0
1	0	0	0	0	0	0	1
0	0	0	1	1	0	0	0
1	0	0	0	0	0	0	0

Ordinal columns can assume the defined values as numerical in the same column; however, if the original values are letters or words, they need to be converted into numbers via a Java Dictionary.

The strategy described above may be implemented by you as an exercise. Otherwise, you would have to handle this manually. In this case, depending on the number of data rows, it can be time-consuming.

Logistic regression

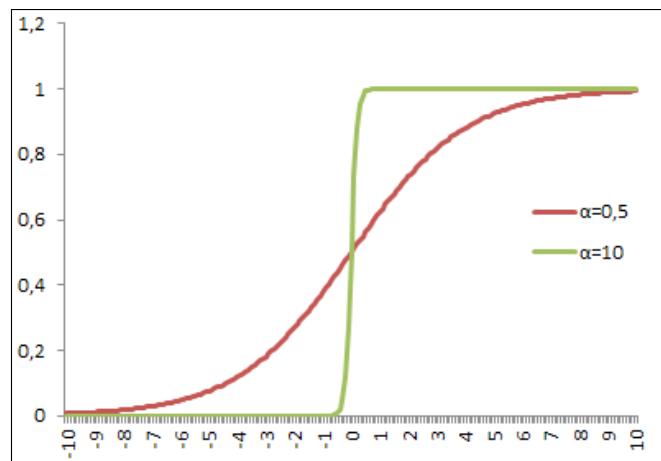
We've covered that Neural Networks can work as data classifiers by establishing decision boundaries onto data in the hyperspace. This boundary can be linear, in the case of perceptrons, or nonlinear, in the case of other neural architectures such as MLPs, Kohonen, or Adaline. The linear case is based on linear regression, on which the classification boundary is a literally a line, as shown in the previous figure. If the scatter chart of the data looks like that of the following figure, then a nonlinear classification boundary is needed:



Neural Networks are in fact a great nonlinear classifier, and this is achieved by the usage of nonlinear activation functions. One nonlinear function that actually works well for nonlinear classification is the sigmoid function, whereas the procedure for classification using this function is called logistic regression:

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

This function returns values bounded between zero and one. In this function α parameter denotes how hard the transition from zero and 1 occurs. The following chart shows the difference:

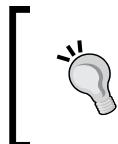
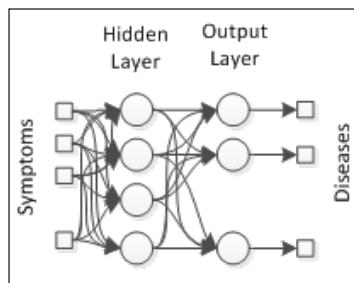


Note that the higher the alpha parameter is, the more the logistic function takes a shape of a hard-limiting threshold function, also known as a step function.

Multiple classes versus binary classes

Classification problems usually deal with a multiple class's case, where each class is assigned a label. However, a binary classification schema is useful to be applied in neural networks. This is because a neural network with a logistic function at the output layer can produce only values between 0 and 1, meaning it belongs (1) or does not belong (0) to some class.

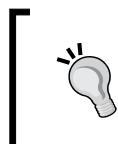
Nevertheless, there is one approach for multiple classes using binary functions. Consider that every class is represented by an output neuron, and whenever that output neuron fires, that neuron's corresponding class is applied on the input data record. So let's suppose a network to classify diseases; each neuron output will represent a disease to be applied to some symptom:



Note that in that configuration, it would be possible to have multiple diseases with the same symptoms, which can happen. However, if only one class would be desirable to be chosen, then a schema as the competitive learning algorithm would suit more in that case.

Confusion matrix

There is no perfect classifier algorithm; all of them are subject to errors and biases; however, it is expected that a classification algorithm can correctly classify 70-90% of the records.



Very high correct classification rates are not always desirable, because of possible biases presented in the input data that might affect the classification task, and also there is a risk of overtraining, when only the training data is correctly classified.

A confusion matrix shows how much of a given class's records were correctly classified and thereby how much were wrongly classified. The following table depicts what a confusion matrix may look like:

Actual Class	Inferred Class							Total
	A	B	C	D	E	F	G	
A	92%	1%	0%	4%	0%	1%	2%	100%
B	0%	83%	5%	6%	2%	3%	1%	100%
C	1%	3%	85%	0%	2%	5%	4%	100%
D	0%	3%	0%	92%	2%	1%	1%	100%
E	0%	10%	2%	1%	78%	1%	8%	100%
F	22%	2%	2%	3%	3%	65%	3%	100%
G	9%	6%	0%	16%	0%	3%	66%	100%

Note that the main diagonal is expected to have the higher values, as the classification algorithm will always try to extract meaningful information from the input dataset. The sum of all rows must be equal to 100%, because all elements of a given class are to be classified in one of the available classes. Note that some classes may receive more classifications than expected.

The more a confusion matrix looks like an identity matrix, the better the classification algorithm will be.

Sensitivity and specificity

When the classification is binary, the confusion matrix is found to be a simple 2x2 matrix, and therefore its positions are specially named:

Actual Class	Inferred Class	
	Positive (1)	Negative (0)
Positive (1)	True Positive	False Negative
Negative (0)	False Positive	True Negative

In disease diagnosis, which is the subject of this chapter, the concept of a binary confusion matrix is applied in the sense that a false diagnosis may be either false positive or false negative. The rate of false results can be measured by sensitivity and specificity indexes.

Sensitivity means the true positive rate; it measures how many of the records are correctly classified positively:

$$\text{Sensitivity} = \frac{\text{Number of True Positives}}{\text{Total of Actual Positive Records}}$$

Specificity, in turn, means the true negative rate; it indicates the proportion of negative record identification:

$$\text{Specificity} = \frac{\text{Number of True Negatives}}{\text{Total of Actual Negative Records}}$$

High values of both sensitivity and specificity are desired; however, depending on the application field, the sensitivity may carry more meaning.

Implementing a confusion matrix

In our code, let's implement the confusion matrix in the class `NeuralOutputData`. The method `calculateConfusionMatrix` below is programmed to consider two neurons in the output layer. If the output is 10, then it is yes to a confusion matrix; if the output is 01, then it is no:

```
public double[][] calculateConfusionMatrix(double[][]  
    dataOutputTestAdapted, double[][] dataOutputTargetTestAdapted) {  
    int TP = 0;  
    int TN = 0;  
    int FP = 0;  
    int FN = 0;  
    for (int m = 0; m < getTargetData().length; m++) {  
        if ( ( dataOutputTargetTestAdapted[m][0] == 1.0 &&  
            dataOutputTargetTestAdapted[m][1] == 0.0 )  
            && ( dataOutputTestAdapted[m][0] == 1.0 &&  
            dataOutputTestAdapted[m][1] == 0.0 ) ) {  
            TP++;  
        } else if ( ( dataOutputTargetTestAdapted[m][0] == 0.0 &&  
            dataOutputTargetTestAdapted[m][1] == 1.0 )  
            && ( dataOutputTestAdapted[m][0] == 0.0 &&  
            dataOutputTestAdapted[m][1] == 1.0 ) ) {  
            TN++;  
        } else if ( ( dataOutputTargetTestAdapted[m][0] == 1.0 &&  
            dataOutputTargetTestAdapted[m][1] == 0.0 )  
            && ( dataOutputTestAdapted[m][0] == 0.0 &&  
            dataOutputTestAdapted[m][1] == 1.0 ) ) {  
    }
```

```

        FP++;
    } else if ( ( dataOutputTargetTestAdapted[m][0] == 0.0 &&
dataOutputTargetTestAdapted[m][1] == 1.0 )
    && ( dataOutputTestAdapted[m][0] == 1.0 &&
dataOutputTestAdapted[m][1] == 0.0 ) ) {
        FN++;
    }
}

return new double[][] { {TP, FN}, {FP, TN} };
}

```

Another method implemented in the NeuralOutputData class is called calculatePerformanceMeasures. It receives as parameter the confusion matrix and it calculates and prints the following performance measures of classification:

- Positive class error rate
- Negative class error rate
- Total error rate
- Total accuracy
- Precision
- Sensibility
- Specificity

This method is shown below:

```

public void calculatePerformanceMeasures(double[][] confMat) {
    double errorRatePositive = confMat[0][1] / (confMat[0]
[0]+confMat[0][1]);
    double errorRateNegative = confMat[1][0] / (confMat[1]
[0]+confMat[1][1]);
    double totalErrorRate = (confMat[0][1] + confMat[1][0]) /
(confMat[0][0] + confMat[0][1] + confMat[1][0] + confMat[1][1]);
    double totalAccuracy = (confMat[0][0] + confMat[1][1]) /
(confMat[0][0] + confMat[0][1] + confMat[1][0] + confMat[1][1]);
    double precision = confMat[0][0] / (confMat[0][0]+confMat[1][0]);
    double sensibility = confMat[0][0] / (confMat[0][0]+confMat[0]
[1]);
    double specificity = confMat[1][1] / (confMat[1][0]+confMat[1]
[1]);
}

```

```
System.out.println("### PERFORMANCE MEASURES ###");
System.out.println("positive class error rate:
"+(errorRatePositive*100.0)+"%");
System.out.println("negative class error rate:
"+(errorRateNegative*100.0)+"%");
System.out.println("total error rate:
"+(totalErrorRate*100.0)+"%");
System.out.println("total accuracy: "+(totalAccuracy*100.0)+"%");
System.out.println("precision: "+(precision*100.0)+"%");
System.out.println("sensibility: "+(sensitivity*100.0)+"%");
System.out.println("specificity: "+(specificity*100.0)+"%");

}
```

Neural networks for classification

Classification tasks can be done by any of the supervised neural networks this book has covered so far. However, it is recommended that you use more complex architectures such as MLPs. In this chapter, we are going to use the `NeuralNet` class to build an MLP with one hidden layer and the sigmoid function at the output. Every output neuron will mean a class.

The code used to implement the examples is very similar to the test class (`BackpropagationTest`). However, the class `DiagnosisExample` asks which dataset the user would like to use and other neural network parameters, such as number of epochs, number of neurons in hidden layer, and learning rate.

Disease diagnosis with neural networks

For disease diagnosis, we are going to use the free dataset `proben1`, which is available on the Web (<http://www.filewatcher.com/m/proben1.tar.gz.1782734-0.html>). `Proben1` is a benchmark set of several datasets from different domains. We are going to use the cancer and the diabetes datasets. We add a class to run the experiments of each case: `DiagnosisExample`.

Breast cancer

The breast cancer dataset is composed of 10 variables, of which nine are inputs and one is a binary output. The dataset has 699 records, but we excluded from them 16 which were found to be incomplete, thus we used 683 to train and test the neural network.

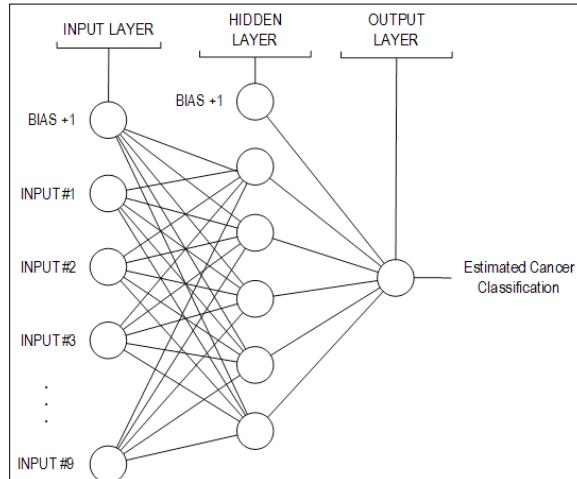


In real practical problems, it is common to have missing or invalid data. Ideally, the classification algorithm must handle these records, but sometimes it is recommended that you exclude them, since there would be not enough information to produce an accurate result.

The following table shows a configuration of this dataset:

Variable Name	Type	Maximum Value and Minimum Value
Diagnosis result	OUTPUT	[0; 1]
Clump Thickness	INPUT #1	[1; 10]
Uniformity of Cell Size	INPUT #2	[1; 10]
Uniformity of Cell Shape	INPUT #3	[1; 10]
Marginal Adhesion	INPUT #4	[1; 10]
Single Epithelial Cell Size	INPUT #5	[1; 10]
Bare Nuclei	INPUT #6	[1; 10]
Bland Chromatin	INPUT #7	[1; 10]
Normal Nucleoli	INPUT #8	[1; 10]
Mitoses	INPUT #9	[1; 10]

So, the proposed neural topology will be that of the following figure:



The dataset division was made as follows:

- **Training:** 549 records (80%);
- **Testing:** 134 records (20%)

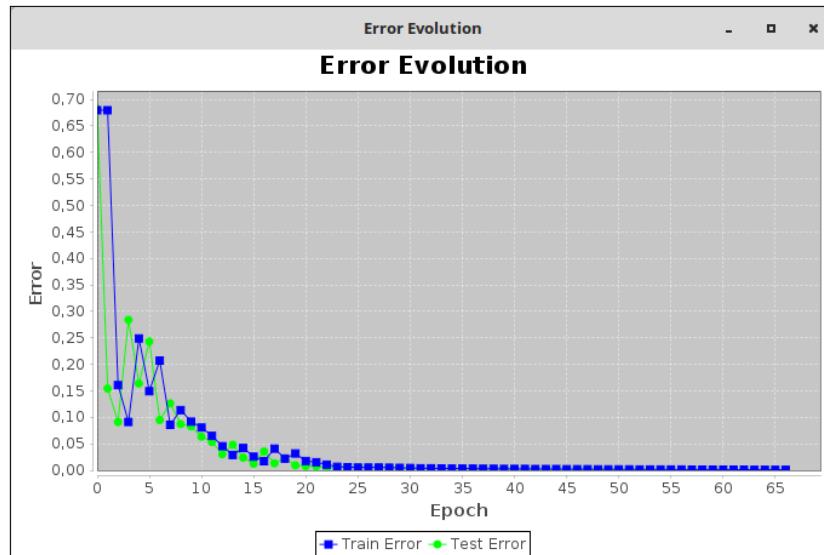
As in the previous cases, we performed many experiments to try to find the best neural network to classify whether cancer is benign or malignant. So we conducted 12 different experiments (1,000 epochs per experiment), wherein MSE and accuracy values were analyzed. After that, the confusion matrix, sensitivity, and specificity were generated with the test dataset and analysis was done. Finally, an analysis of generalization was taken. The neural networks involved in the experiments are shown in the following table:

Experiment	Number of neurons in hidden layer	Learning rate	Activation Function
#1	3	0.1	Hidden Layer: SIGLOG Output Layer: LINEAR
#2			Hidden Layer: HYPERTAN Output Layer: LINEAR
#3		0.5	Hidden Layer: SIGLOG Output Layer: LINEAR
#4			Hidden Layer: HYPERTAN Output Layer: LINEAR
#5		0.9	Hidden Layer: SIGLOG Output Layer: LINEAR
#6			Hidden Layer: HYPERTAN Output Layer: LINEAR
#7	5	0.1	Hidden Layer: SIGLOG Output Layer: LINEAR
#8			Hidden Layer: HYPERTAN Output Layer: LINEAR
#9		0.5	Hidden Layer: SIGLOG Output Layer: LINEAR
#10			Hidden Layer: HYPERTAN Output Layer: LINEAR
#11		0.9	Hidden Layer: SIGLOG Output Layer: LINEAR
#12			Hidden Layer: HYPERTAN Output Layer: LINEAR

After each experiment, we collected MSE values (Table X); experiments #4, #8, #9, #10, and #11 were equivalents, because they have low MSE values and same total accuracy measure (92.25%). Therefore, we selected experiments #4 and #11, because they have low MSE values among the five experiments mentioned before:

Experiment	MSE training rate	Total accuracy
#1	0.01067	96.29%
#2	0.00443	98.50%
#3	9.99611E-4	97.77%
#4	9.99913E-4	99.25%
#5	9.99670E-4	96.26%
#6	9.92578E-4	97.03%
#7	0.01392	98.49%
#8	0.00367	99.25%
#9	9.99928E-4	99.25%
#10	9.99951E-4	99.25%
#11	9.99926E-4	99.25%
#12	NaN	3.44%

Graphically, the MSE evolution over time is very fast, as can be seen in the following chart of the fourth experiment. Although we used 1,000 epochs to train, the experiment stopped earlier, because the minimum overall error (0.001) was reached:



The confusion matrix is shown in the table with the sensibility and specificity for both experiments. It is possible to check that measures are the same for both experiments:

Experiment	Confusion Matrix	Sensibility	Specificity
#4	$[[34.0, 1.0]$ $[0.00, 99.0]]$	97.22%	100.0%
#11	$[[34.0, 1.0]$ $[0.00, 99.0]]$	97.22%	100.0%

If we had to choose between models generated by experiments #4 or #11, we recommend selecting #4, because it's simpler than #11 (it has fewer neurons in the hidden layer).

Diabetes

An additional example to be explored is the diagnosis of diabetes. This dataset has eight inputs and one output, shown in the table below. There are 768 records, all complete. However, proben1 states that there are several senseless zero values, probably indicating missing data. We're handling this data as if it was real anyway, thereby introducing some errors (or noise) into the dataset:

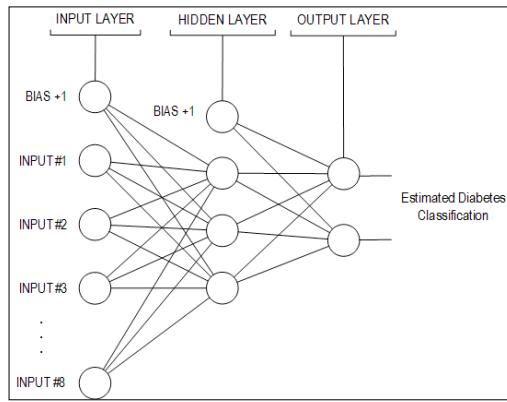
Variable Name	Type	Maximum Value and Minimum Value
Diagnosis result	OUTPUT	[0; 1]
Number of times pregnant	INPUT #1	[0.0; 17]
Plasma glucose concentration a 2 hours in an oral glucose tolerance test	INPUT #2	[0.0; 199]
Diastolic blood pressure (mm Hg)	INPUT #3	[0.0; 122]
Triceps skin fold thickness (mm)	INPUT #4	[0.0; 99]
2-Hour serum insulin (mu U/ml)	INPUT #5	[0.0; 744]
Body mass index (weight in kg/(height in m)^2)	INPUT #6	[0.0; 67.1]
Diabetes pedigree function	INPUT #7	[0.078; 2420]
Age (years)	INPUT #8	[21; 81]

The dataset division was made as follows:

- **Training:** 617 records (80%)
- **Test:** 151 records (20%)

To discover the best neural net topology to classify diabetes, we used the same schema of neural networks with the same analysis described in the last section. However, we're using multiple class classification in the output layer: two neurons in this layer will be used, one for the presence of diabetes and one for absence.

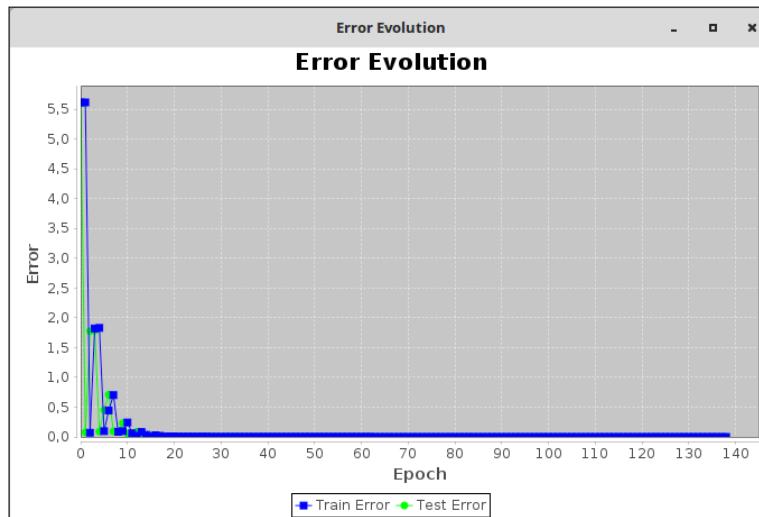
So, the proposed neural architecture looks like that of the following figure:



The table below shows the MSE training value and accuracy of the first six experiments and of the last six experiments:

Experiment	MSE training rate	Total accuracy
#1	0.00807	60.54%
#2	0.00590	71.03%
#3	9.99990E-4	75.49%
#4	9.98840E-4	74.17%
#5	0.00184	61.58%
#6	9.82774E-4	59.86%
#7	0.00706	63.57%
#8	0.00584	72.41%
#9	9.99994E-4	74.66%
#10	0.01047	72.14%
#11	0.00316	59.86%
#12	0.43464	40.13%

The fall of the MSE is fast in both cases. However, experiment #9 generates an increase of error rate in the first values. It is shown in the following figure:



Analyzing the confusion matrixes, it can be seen that the measures are very similar:

Experiment	Confusion Matrix	Sensibility	Specificity
#3	[[35.0, 12.0] [25.0, 79.0]]	74.46%	75.96%
#9	[[34.0, 12.0] [26.0, 78.0]]	73.91%	75.00%



It is recommended you explore the class `DiagnosisExample` and create a GUI to become easy select neural net parameters, as was done in the previous chapter. You should try to reuse code already programmed through the inheritance concept.

Summary

In this chapter, we've seen two examples of the application of disease diagnosis using neural networks. The fundamentals of classification problems were briefly reviewed in order to level the knowledge explored in this chapter. Classification tasks belong to one of the most used types of supervised tasks in the machine learning / data mining fields, and Neural Networks proved to be very appropriate to be applied to this type of problem. The reader was also presented with the concepts that evaluate the classification tasks, such as sensitivity, specificity, and the confusion matrix. These notations are very useful for all classification tasks, including those which are handled with other algorithms besides neural networks. The next chapter will explore a similar kind of task but using unsupervised learning – that means, without expected output data – but the fundamentals presented in this chapter will be somewhat helpful.

7

Clustering Customer Profiles

One of the amazing capabilities of neural networks applying unsupervised learning is their ability to find hidden patterns which even experts may not have any clue about. In this chapter, we're going to explore this fascinating feature through a practical application to find customer and product clusters provided in transactions database. We'll go through a review on unsupervised learning and the clustering task. To demonstrate this application, the reader will be provided with a practical example on customer profiling and it's implementation in Java. The topics of this chapter are:

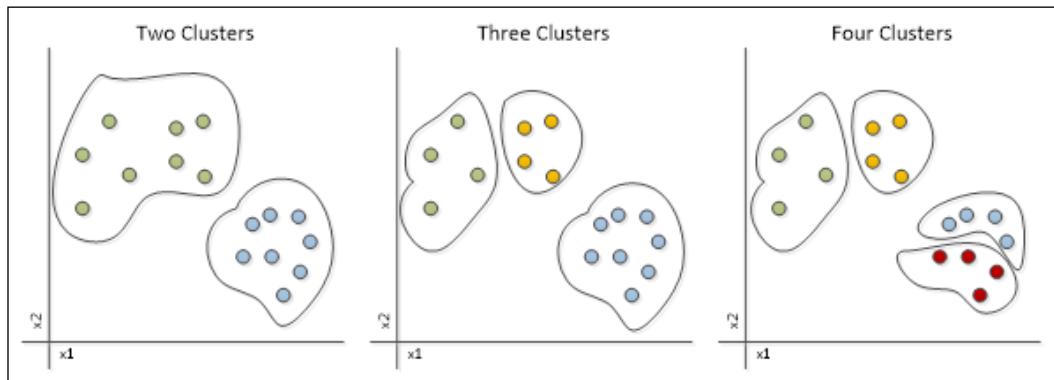
- Clustering tasks
- Cluster analysis
- Cluster evaluation
- Applied unsupervised learning
- Radial basis functions neural network
- Kohonen network for clustering
- Handling with types of data
- Customer profiling
- Preprocessing
- Implementation in Java
- Credit analysis and profiles of customers

Clustering tasks

Clustering is part of a broader set of tasks in data analysis, whose objective is to group elements that look alike, more similar to each other, into clusters or groups. Clustering tasks are fully based on unsupervised learning since there is no need to include any target output data in order to find clusters; instead, the solution designer may choose a number of clusters that they want to group the records into and check the response of the algorithm to it.

 Clustering tasks may seem to overlap with classification tasks with the crucial difference that in clustering there is no need to have a predefined set of classes before the clustering algorithm is run.

One may wish to apply clustering when there is little or no information at all about how the data can be gathered into groups. Provided with dataset, we wish our neural network to identify both the groups and their members. While this may seem easy and straightforward to perform visually in a two-dimensional dataset, as shown in the following figure, with a higher number of dimensions, this task becomes not so trivial to perform and needs an algorithmic solution:



In clustering, the number of clusters is not determined by the data, but by the data analyst who is looking to cluster the data. Here the *boundaries* are a little bit different than those of classification tasks because they depend primarily on the number of clusters.

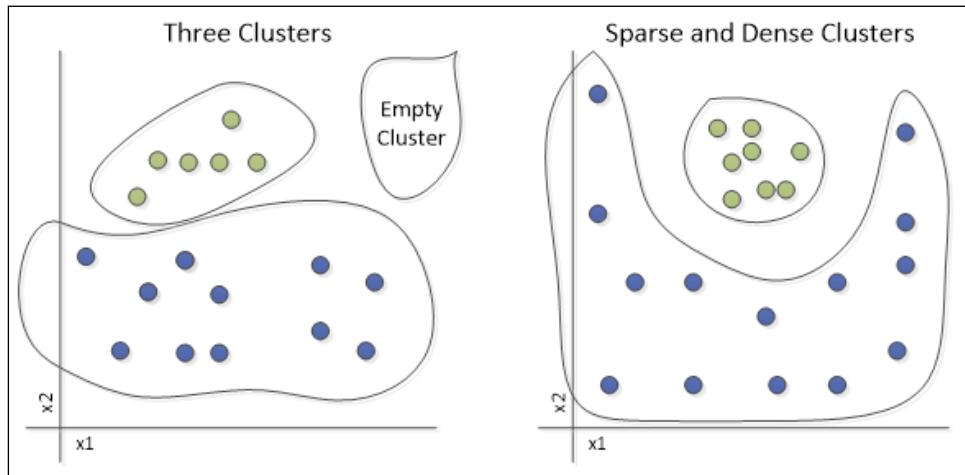
Cluster analysis

One difficulty in the clustering tasks, and also in unsupervised learning tasks, is the accurate interpretation of the results. While in supervised learning there is a defined target, from which we can derive an error measure or confusion matrix, in unsupervised learning the evaluation of quality is totally different, and also totally dependent on the data itself. The validation criteria involves indexes that assert how well the data distributed across the clusters is, as well as external opinions from experts on the data that are also a measure of quality.



To illustrate an example, let's suppose a task of clustering of plants given their characteristics (sizes, leave colors, period of fruiting, and so on), and a neural network mistakenly groups cactus with pine trees in the same cluster. A botanist would certainly not endorse the classification based on their specific knowledge on the field that this grouping does not make any sense.

Two major issues happen in clustering. One is the fact that one neural network's output is never activated, meaning that one cluster does not have any data point associated with it. Another one is the case of nonlinear or sparse clusters, which could be erroneously grouped into several clusters while actually there might be only one.



Cluster evaluation and validation

Unfortunately, if the neural network clusters badly, one needs either to redefine the number of clusters or perform additional data preprocessing. To evaluate how well the clustered data is, the Davies-Bouldin and Dunn indexes may be applied.

The Davies-Boudin index takes into account the cluster's centroids in order to find inter and intra-distances between clusters and cluster members:

$$DB = \frac{1}{n} \sum_{i=1}^n \max_{j \neq i} \left(\frac{\sigma_i + \sigma_j}{d(c_i, c_j)} \right)$$

Where n is the number of clusters, c_i is the centroid of cluster i , σ_i is the average distance of all elements in cluster i , and $d(c_i, c_j)$ is the distance between clusters i and j . The smaller the value of DB index, the better the neural network will be considered to the cluster.

However, for dense and sparse clusters, the DB index will not give much useful information. This limitation can be overcome with the Dunn index:

$$D = \frac{\min_{1 \leq i < j \leq n} d(i, j)}{\max_{1 \leq k \leq n} d'(k)}$$

Where $d(i, j)$ is the inter cluster distance between i and j , and $d'(k)$ is the intra cluster distance of cluster k . Here the higher the Dunn index is, the better the clustering will be because although the clusters may be sparse, they still need to be grouped together, and high intra-cluster distances will denote a bad grouping of data.

Implementation

In the CompetitiveLearning class, we are going to implement these indexes:

```
public double DBIndex() {
    int numberOfClusters = this.neuralNet.getNumberOfOutputs();
    double sum=0.0;
    for(int i=0;i<numberOfClusters;i++){
        double[] index = new double[numberOfClusters];
        for(int j=0;j<numberOfClusters;j++){
            if(i!=j){
                //calculate the average distance for cluster i
                ...
            }
        }
    }
}
```

```

        Double Sigmai=averageDistance(i,trainingDataSet);
        Double Sigmaj=averageDistance(j,trainingDataSet);
        Double[] Centeri=neuralNet.getOutputLayer().getNeuron(i).
        getWeights();
        Double[] Centerj=neuralNet.getOutputLayer().getNeuron(j).
        getWeights();
        Double distance = getDistance(Centeri,Centerj);
        index[j]=((Sigmai+Sigmaj)/distance);
    }
}
sum+=ArrayOperations.max(index);
}
return sum/numberOfClusters;
}

public double Dunn(){
    int numberOfClusters = this.neuralNet.getNumberOfOutputs();
    ArrayList<double> interclusterDistance;
    for(int i=0;i<numberOfClusters;i++){
        for(int j=i+1;j<numberOfClusters;j++){
            interClusterDistance.add(minInterClusterDistance
(i,j,trainingDataSet));
        }
    }
    ArrayList<double> intraclusterDistance;
    for(int k=0;k<numberOfClusters;k++){
        intraclusterDistance.add(maxIntraClusterDistance(k,
trainingDataSet));
    }
    return ArrayOperations.Min(interclusterDistance) / ArrayOperations.
Max(intraclusterDistance);
}

```

External validation

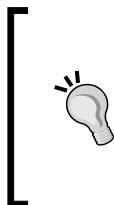
In some cases, there is already an expected result for clustering, as in the example of plants clustering. This is called external validation. One may apply a neural network with unsupervised learning to cluster data that is already assigned a value. The major difference against the classification lies in the fact that the target outputs are not considered, so the algorithm itself is expected to draw a borderline based only on the data.

Applied unsupervised learning

In neural networks, there are a number of architectures implementing unsupervised learning; however, the scope of this book will cover only the Kohonen neural network, developed in *Chapter 4, Self-Organizing Maps*.

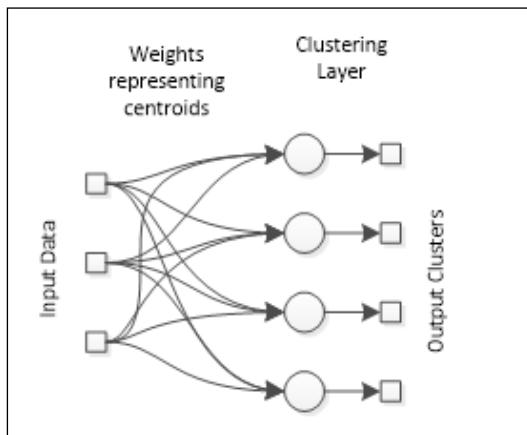
Kohonen neural network

Kohonen Networks, which have been covered in *Chapter 4, Self-Organizing Maps* are now used in a modified fashion. Kohonen can produce a shape in one or two dimensions at the output, but here we are interested in clustering, which can be reduced in only one dimension.



Actually the Kohonen neural network implemented in this framework considers the dimensions zero, one, and two, where zero means no connections between the output neurons and one means they form a line, and two means a grid. For this chapter's example, we will need a Kohonen network with no connected output neurons, therefore, the dimension will be zero.

In addition, clusters may be related or not to each other, so the vicinity of neurons can be ignored for now in this chapter, which means only one neuron will be activated and their neighbors will remain unchanged. And so, the neural network will adjust its weights to match data to an array of clusters:



The training algorithm will be the competitive learning, whereby the neuron with the greatest output has its weights adjusted. By the end of training, all the clusters of a neural network are expected to be defined. Note that there are no links between output neurons, meaning that only one input is active at the output.

Profiling

One of the interesting tasks in unsupervised learning is the profiling or clustering of information, in this chapter, customers and products. Given one dataset, one wants to find groups of records that share similar characteristics. Examples are customers that buy the same products or products that are usually bought together. This task results in a number of benefits for business owners because they are provided the information on which groups of customers and products they have, whereby they are enabled to address them more accurately.

Pre-processing

As seen in *Chapter 6, Classifying Disease Diagnosis* transactional databases can contain both numerical and categorical data. Whenever we face a categorical unscaled variable, we need to split it into the number of values the variable may take, using the `CategoricalDataset` class. For example, let's suppose we have the following transaction list of customer purchases:

Transaction ID	Customer ID	Products	Discount	Total
1399	56	Milk, Bread, Butter	0.00	4.30
1400	991	Cheese, Milk	2.30	5.60
1401	406	Bread, Sausage	0.00	8.80
1402	239	Chipotle Sauce, Spice	0.00	6.70
1403	33	Turkey	0.00	4.50
1404	406	Turkey, Butter, Spice	1.00	9.00

It can easily be seen that the products are unscaled categorical data and for each transaction there is an undefined number of products purchased, the customer may purchase one or several. In order to transform that dataset into a numerical dataset, preprocessing is needed. For each product there will be a variable added to the dataset, resulting in the following:

Cust. Id	Milk	Bread	Butter	Cheese	Sausage	Chipotle Sauce	Spice	Turkey
56	1	1	1	0	0	0	0	0
991	1	0	0	1	0	0	0	0
406	0	1	1	0	1	0	1	1
239	0	0	0	0	0	1	1	0
33	0	0	0	0	0	0	0	1

In order to save space, we ignored the numerical variables and considered the presence of the product purchased by a client as *1* and the absence as *0*. Alternative preprocessing may consider the number of occurrences of a value, therefore becoming no longer binary, but discrete.

Implementation in Java

In this chapter, we are going to explore the usage of Kohonen neural network applied to customer clustering based on customer information collected from Proben1 (Card dataset).

Card – credit analysis for customer profiling

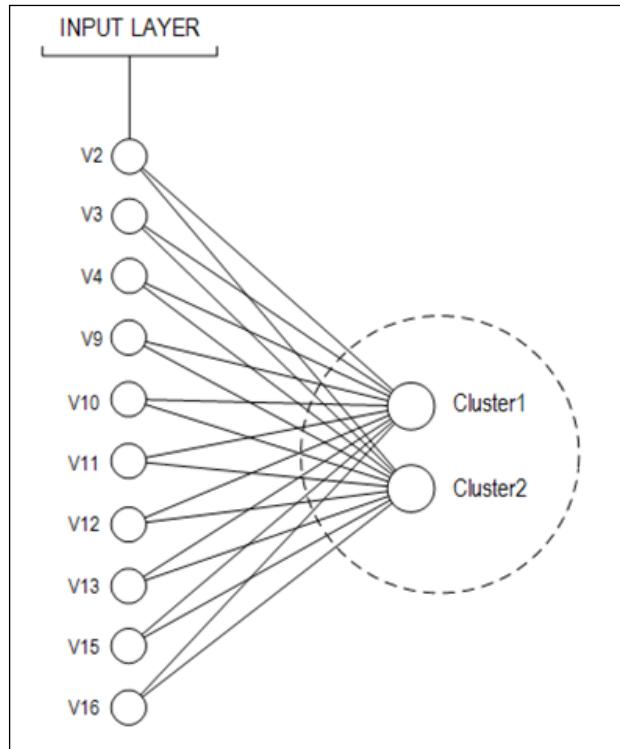
The card dataset is composed of 16 variables in total. 15 are inputs and one is output. For security reasons, all variable names have been changed to meaningless symbols. This dataset brings a good mix of variable types (continuous, categorical with small numbers of values, and categorical with a larger number of values). The following table shows a summary of data:

Variable	Type	Values
V1	OUTPUT	0; 1
V2	INPUT #1	b, a
V3	INPUT #2	continuous
V4	INPUT #3	continuous
V5	INPUT #4	u, y, l, t.
V6	INPUT #5	g, p, gg
V7	INPUT #6	c, d, cc, i, j, k, m, r, q, w, x, e, aa, ff
V8	INPUT #7	v, h, bb, j, n, z, dd, ff, o
V9	INPUT #8	continuous
V10	INPUT #9	t, f
V11	INPUT #10	t, f
V12	INPUT #11	continuous
V13	INPUT #12	t, f
V14	INPUT #13	g, p, s
V15	INPUT #14	continuous
V16	INPUT #15	continuous

For simplicity we didn't use the inputs $v5-v8$ and $v14$, in order to not inflate the number of inputs very much. We applied the following transformation:

Variable	Type	Values	Conversion
V1	OUTPUT	0; 1	-
V2	INPUT #1	b, a	$b = 1, a = 0$
V3	INPUT #2	continuous	-
V4	INPUT #3	continuous	-
V9	INPUT #8	continuous	-
V10	INPUT #9	t, f	$t = 1, f = 0$
V11	INPUT #10	t, f	$t = 1, f = 0$
V12	INPUT #11	continuous	-
V13	INPUT #12	t, f	$t = 1, f = 0$
V15	INPUT #14	continuous	-
V16	INPUT #15	continuous	-

The neural net topology proposed is shown in the following figure:



The number of examples stored is 690, but 37 of them have missing values. These 37 records were discarded. Therefore, 653 examples were used to train and test the neural network. The dataset division was made as follows:

- **Training:** 583 records
- **Test:** 70 records

The Kohonen training algorithm used to cluster similar behavior depends on some parameters, such as:

- Normalization type
- Learning rate

It is important to consider that the Kohonen training algorithm is unsupervised. So, this algorithm is used when the output is not known. In the card example there are output values in the dataset and they will be used here only to attest clustering. But in traditional clustering cases, the output values are not available.

In this specific case, because output is known, as classification, the clustering quality may be attested by:

- Sensibility (true positive rate)
- Specificity (true negative rate)
- Total accuracy

In Java projects, the calculations of these values are done through a class named `NeuralOutputData`, previously developed in *Chapter 6, Classifying Disease Diagnosis*.

It is good practice to do many experiments to try to find the best neural net to cluster customers' profiles. Ten different experiments will be generated and each will be analyzed with the quality rates mentioned previously. The following table summarizes the strategy that will be followed:

Experiment	Learning rate	Normalization type
#1	0.1	MIN_MAX
#2		Z_SCORE
#3	0.3	MIN_MAX
#4		Z_SCORE
#5	0.5	MIN_MAX
#6		Z_SCORE
#7	0.7	MIN_MAX
#8		Z_SCORE

Experiment	Learning rate	Normalization type
#9	0.9	MIN_MAX
#10		Z_SCORE

The `ClusterExamples` class was created to run each experiment. In addition to processing data in *Chapter 4, Self-Organizing Maps* it was also explained how to create a Kohonen net and how to train it via the Euclidian distance algorithm.

The following piece of code shows a bit of its implementation:

```
// enter neural net parameter via keyboard (omitted)

// load dataset from external file (omitted)

// data normalization (omitted)

// create ANN and define parameters to TRAIN:
CompetitiveLearning cl = new CompetitiveLearning(kn1,
neuralDataSetToTrain, LearningAlgorithm.LearningMode.ONLINE);
cl.show2DData=false;
cl.printTraining=false;
cl.setLearningRate( typedLearningRate );
cl.setMaxEpochs( typedEpochs );
cl.setReferenceEpoch( 200 );
cl.setTestingDataSet(neuralDataSetToTest);

// train ANN
try {
System.out.println("Training neural net... Please, wait...");
cl.train();
System.out.println("Winner neurons (clustering result [TRAIN]):");
System.out.println( Arrays.toString( cl.getIndexWinnerNeuronTrain()
) );
}

} catch (NeuralException ne) {
ne.printStackTrace();
}
```

After running each experiment using the `ClusteringExamples` class and saving the confusion matrix and total accuracy rates, it is possible to observe that experiments #4, #6, #8, and #10 have the same confusion matrix and accuracy. These experiments used z-score to normalize data:

Experiment	Confusion matrix	Total accuracy
#1	<code>[[14.0, 21.0] [18.0, 17.0]]</code>	44.28%
#2	<code>[[11.0, 24.0] [34.0, 1.0]]</code>	17.14%
#3	<code>[[21.0, 14.0] [17.0, 18.0]]</code>	55.71%
#4	<code>[[24.0, 11.0] [1.0, 34.0]]</code>	82.85%
#5	<code>[[21.0, 14.0] [17.0, 18.0]]</code>	55.71%
#6	<code>[[24.0, 11.0] [1.0, 34.0]]</code>	82.85%
#7	<code>[[8.0, 27.0] [7.0, 28.0]]</code>	51.42%
#8	<code>[[24.0, 11.0] [1.0, 34.0]]</code>	82.85%
#9	<code>[[27.0, 8.0] [28.0, 7.0]]</code>	48.57%
#10	<code>[[24.0, 11.0] [1.0, 34.0]]</code>	82.85%

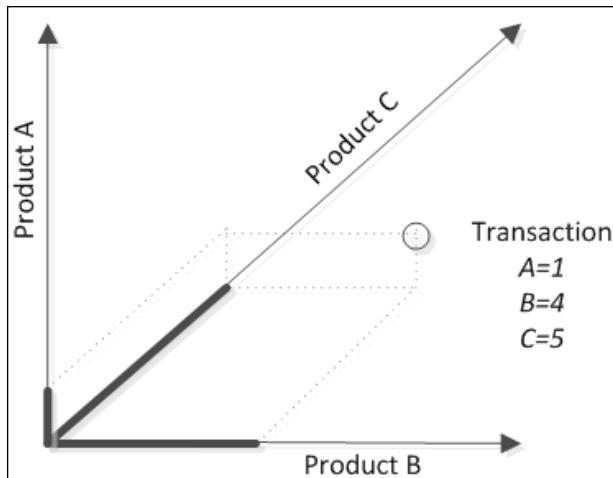
So, neural nets built by experiments #4, #6, #8, or #10 may be used to reach accuracy more than 80% to cluster customers financially.

Product profiling

Using a transactional database provided with the code, we've compiled about 650 purchase transactions into a big matrix transactions x products, where in each cell there is the quantity of the corresponding product that has been bought on the corresponding transaction:

#Trns.	Prd.1	Prd.2	Prd.3	Prd.4	Prd.5	Prd.6	Prd.7	...	Prd.N
1	56	0	0	3	2	0	0	...	0
2	0	0	40	0	7	0	19	...	0
...
n	0	0	0	0	0	0	0	...	1

Let's consider that this matrix is a representation in an N-dimensional hyperspace taking each product as a dimension and the transactions as points. For simplicity, let's consider an example on three dimensions. A given transaction with the quantities bought for each product will be placed in a point corresponding to the quantities at each dimension.



The idea is to cluster these transactions in order to find which products are usually bought together. So, we are going to use a Kohonen neural network in order to find the positions of the products that the clusters centers will be located at.

Our database consists of a clothing store and a sample of 27 registered products:

1 Long Dress A	19 Overall with zipper	43 Bermuda M
3 Long Dress B	22 Shoulder overall	48 Stripped skirt
7 Short Dress A	23 Long stamped skirt	67 Camisole shoulder strap
8 Stamped Dress	24 Stamped short dress	68 Jeans M
9 Women Camisole	28 Pants M	69 XL Short dress
13 Pants S	31 Sleeveless short dress	74 Stripped camisole S
16 Overall for children	32 Short dress shoulder	75 Stripped camisole M
17 Shorts	34 Short dress B	76 Stripped camisole L
18 Stamped overall	42 Two blouse overall	106 Straight skirt

How many clusters?

Sometimes it may be difficult to choose how many clusters to find in a clustering algorithm. Some approaches to determine an optimal choice include information criteria such as **Akaike Information Criteria (AIC)**, **Bayesian Information Criteria (BIC)**, and the Mahalanobis distance from the center to the data. We suggest to the reader to check the references if interested in further details on these criteria.

To make tests to product example, we also should use the `ClusteringExamples` class. For simplicity, we run tests with three and five clusters. For each experiment, the number of epochs was 1000, the learning rate was 0.5, and the normalization type was `MIN_MAX (-1; 1)`. Some results are shown in the following table:

Number of clusters	Clusters of the first 15 elements	Sum of products bought
3	0, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 2,	973, 585, 11, 5, 2, 4, 11, 6, 3, 2, 2, 2, 669, 672, 7,
5	0, 1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 0, 0, 4,	973, 585, 11, 5, 2, 4, 11, 6, 3, 2, 2, 2, 669, 672, 7,

Observing the preceding table, we note when the sum of products acquired is more than 600, then it's clustered together. Otherwise, when the sum is in the range of 500 to 599, another cluster is formed. Lastly, if the sum is low, a large cluster is created, because the dataset is compound by many cases that customers doesn't buy more than 20 items.



As recommended in the previous chapter, we suggest you explore the `ClusteringExamples` class and create a GUI to easily select the neural net parameters. You should try to reuse code through the inheritance concept.

Another tip is to further explore the product profiling example: varying the neural network training parameters, the number of clusters, and/or develop other ways of analyzing the clustering result.

Summary

In this chapter, we've seen an application of customer profiling using the Kohonen neural network. Unlike the classification task, the clustering task does not consider the previous knowledge on the desired output; instead it is desirable for the clusters to be found by the neural network. However, we've seen that validation techniques may include external validation, which is a comparison with what could be understood as *target output*. Customer profiling is important because it gives a business owner more accurate and clean information about their customers, without the *human interference* in pointing which customers are in some groups or in others, as occurs in supervised learning. That's the advantage of unsupervised learning, enabling the data to draw results solely by themselves.

8

Text Recognition

We all know that humans can read and recognize images faster than any supercomputer. However, we have seen so far that neural networks show amazing capabilities of learning through data in both a supervised and an unsupervised way. In this chapter, we present an additional case of pattern recognition involving an example of optical character recognition. Neural networks can be trained to strictly recognize digits written in an image file. The topics of this chapter are:

- Pattern recognition
- Defined classes
- Undefined classes
- Neural networks in pattern recognition
- MLP
- The OCR problem
- Preprocessing and classes definition
- Implementation in Java
- Digit recognition

Pattern recognition

Patterns are a bunch of data and elements that look similar to each other, in such a way that they can occur systematically and repeat from time to time. This is a task that can be solved mainly by unsupervised learning by clustering; however, when there is labelled data or there are defined classes of data, this task can be solved by supervised methods. We, as humans, perform this task more often than we can imagine. When we see objects and recognise them as belonging to a certain class, we are indeed recognising a pattern. Also, when we analyse charts, discrete events, and time series, we might find evidence of some sequence of events that repeat systematically under certain conditions. In summary, patterns can be learned by data observations.

Examples of pattern recognition tasks include, but are not limited to:

- Shape recognition
- Object classification
- Behavior clustering
- Voice recognition
- OCR
- Chemical reaction taxonomy

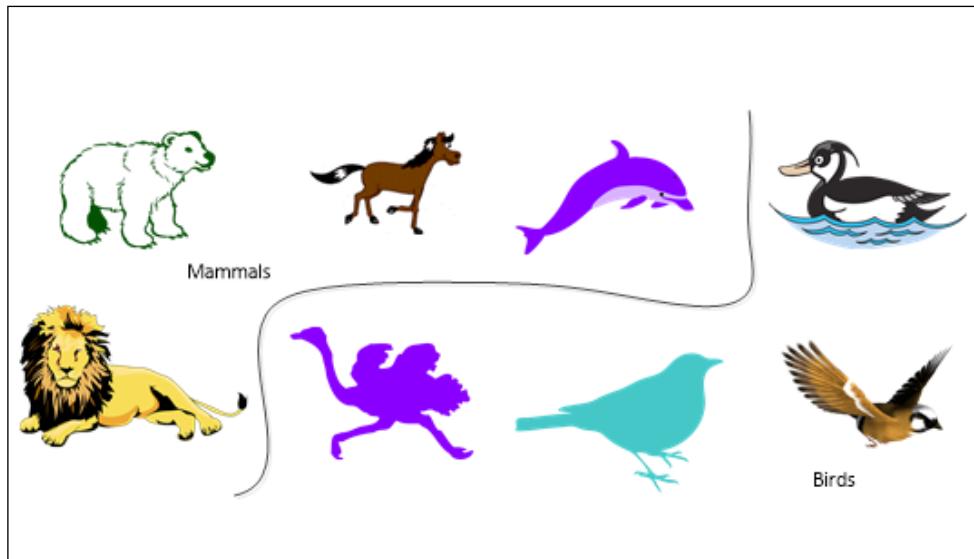
Defined classes

In a list of classes that has been predefined for a specific domain, each class is considered to be a pattern; therefore every data record or occurrence is assigned one of these predefined classes.



The predefinition of classes can usually be performed by an expert or based on previous knowledge of the application domain. Also, it is desirable to apply defined classes when we want the data to be classified strictly into one of the predefined classes.

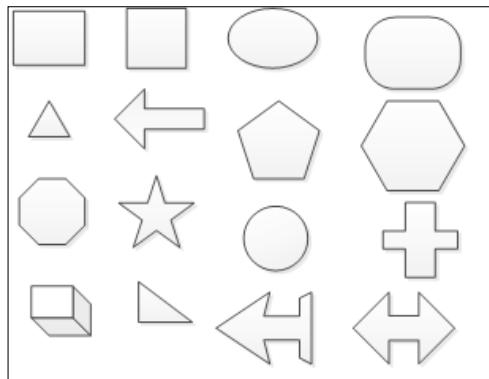
One illustrated example of pattern recognition using defined classes is animal recognition by image, shown in the figure below. The pattern recogniser, however, should be trained to catch all the characteristics that formally define the classes. In the example, eight figures of animals are shown, belonging to two classes: mammals and birds. Since this is a supervised mode of learning, the neural network should be provided with a sufficient number of images to allow it to properly classify new images.



Of course, sometimes the classification may fail, mainly due to similar hidden patterns in the images that neural networks may catch and also due to small nuances present in the shapes. For example, the dolphin has flippers but it is still a mammal. Sometimes, in order to obtain a more accurate classification, it is necessary to apply preprocessing and ensure that the neural network will receive the appropriate data that will allow for classification.

Undefined classes

When data is unlabelled and there is no predefined set of classes, it is an unsupervised learning scenario. Shape recognition is a good example, since the shapes may be flexible and have an infinite number of edges, vertices, or bindings.



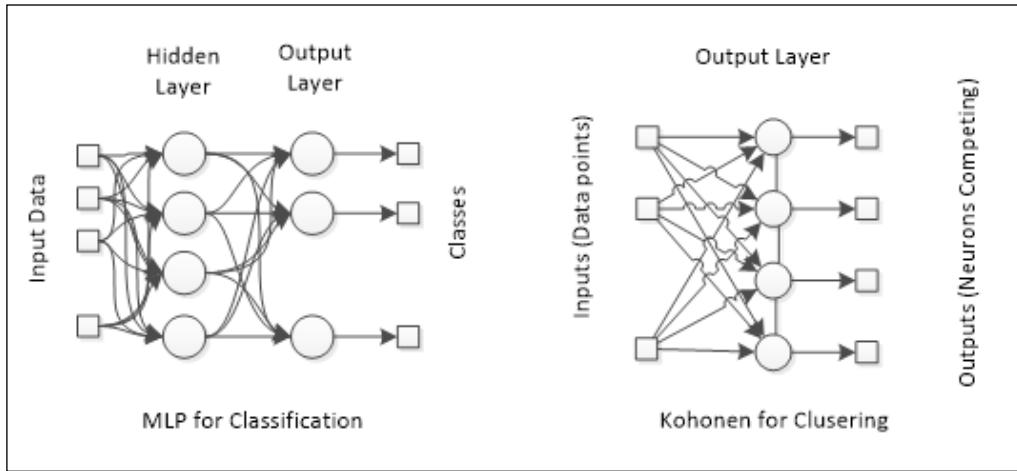
In the image above, we can see some sorts of shapes and we want to arrange them, so that similar ones can be grouped into the same cluster. Based on the shape information that is present in the images, it is likely that the pattern recognizer will classify the rectangle, the square and the triangle into the same group. However, if the information were presented to the pattern recognizer, not as an image, but as a graph with edges and vertices coordinates, the classification might change a little.

In summary, the pattern recognition task may use both supervised and unsupervised modes of learning, basically depending of the objective of recognition.

Neural networks in pattern recognition

For pattern recognition, the neural network architectures that can be applied are MLPs (supervised) and the Kohonen Network (unsupervised). In the first case, the problem should be set up as a classification problem, that is, the data should be transformed into the X - Y dataset, where for every data record in X there should be a corresponding class in Y . As stated in *Chapter 3, Perceptrons and Supervised Learning* and *Chapter 6, Classifying Disease Diagnosis* the output of the neural network for classification problems should have all of the possible classes, and this may require preprocessing of the output records.

For the other case, unsupervised learning, there is no need to apply labels to the output, but the input data should be properly structured. To remind you, the schema of both neural networks are shown in the next figure:



Data pre-processing

As previously seen in *Chapter 6, Classifying Disease Diagnosis* and *Chapter 7, Clustering Customer Profiles* we have to deal with all possible types of data, i.e., numerical (continuous and discrete) and categorical (ordinal or unscaled).

However, here we have the possibility of performing pattern recognition on multimedia content, such as images and videos. So, can multimedia could be handled? The answer to this question lies in the way these contents are stored in files. Images, for example, are written with a representation of small colored points called pixels. Each color can be coded in an RGB notation where the intensity of red, green, and blue define every color the human eye is able to see. Therefore an image of dimension 100x100 would have 10,000 pixels, each one having three values for red, green and blue, yielding a total of 30,000 points. That is the challenge for image processing in neural networks.

Some methods, which we'll review in the next chapter, may reduce this huge number of dimensions. Afterwards an image can be treated as big matrix of numerical continuous values.

For simplicity, we are applying only gray-scale images with small dimensions in this chapter.

Text recognition (optical character recognition)

Many documents are now being scanned and stored as images, making it necessary to convert these documents back into text, for a computer to apply edition and text processing. However, this feature involves a number of challenges:

- Variety of text font
- Text size
- Image noise
- Manuscripts

In spite of that, humans can easily interpret and read even texts produced in a bad quality image. This can be explained by the fact that humans are already familiar with text characters and the words in their language. Somehow the algorithm must become acquainted with these elements (characters, digits, signalization, and so on), in order to successfully recognize texts in images.

Digit recognition

Although there are a variety of tools available on the market for OCR, it still remains a big challenge for an algorithm to properly recognize texts in images. So, we will be restricting our application to a smaller domain, so that we'll face simpler problems. Therefore, in this chapter, we are going to implement a neural network to recognize digits from 0 to 9 represented on images. Also, the images will have standardized and small dimensions, for the sake of simplicity.

Digit representation

We applied the standard dimension of 10x10 (100 pixels) in gray scaled images, resulting in 100 values of gray scale for each image:

	255	255	255	255	255	255	255	255	255	255	255
	255	255	125	40	40	40	40	90	255	255	
	255	255	40	40	40	40	40	40	255	255	
	255	255	40	255	255	255	40	40	255	255	
	255	255	255	255	255	40	40	40	255	255	
	255	255	255	255	255	40	40	40	80	255	255
	255	255	255	255	255	255	40	40	255	255	
	255	175	125	255	255	255	40	40	255	255	
	255	40	40	40	40	40	40	80	255	255	
	255	255	40	90	255	255	255	255	255	255	

In the preceding image we have a sketch representing the digit 3 at the left and a corresponding matrix with gray values for the same digit, in gray scale.

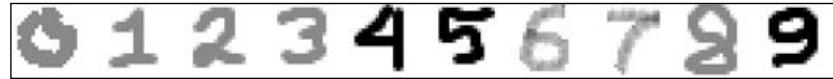
We apply this preprocessing in order to represent all ten digits in this application.

Implementation in Java

To recognize optical characters, data to train and to test neural network was produced by us. In this example, digits from 0 (super black) to 255 (super white) were considered. According to pixel disposal, two versions of each digit data were created: one to train and another to test. Classification techniques presented in *Chapter 3, Perceptrons and Supervised Learning* and *Chapter 6, Classifying Disease Diagnosis* will be used here.

Generating data

Numbers from zero to nine were drawn in the Microsoft Paint ®. The images have been converted into matrices, from which some examples are shown in the following image. All pixel values between zero and nine are grayscale:



For each digit we generated five variations, where one is the perfect digit, and the others contain noise, either by the drawing, or by the image quality.

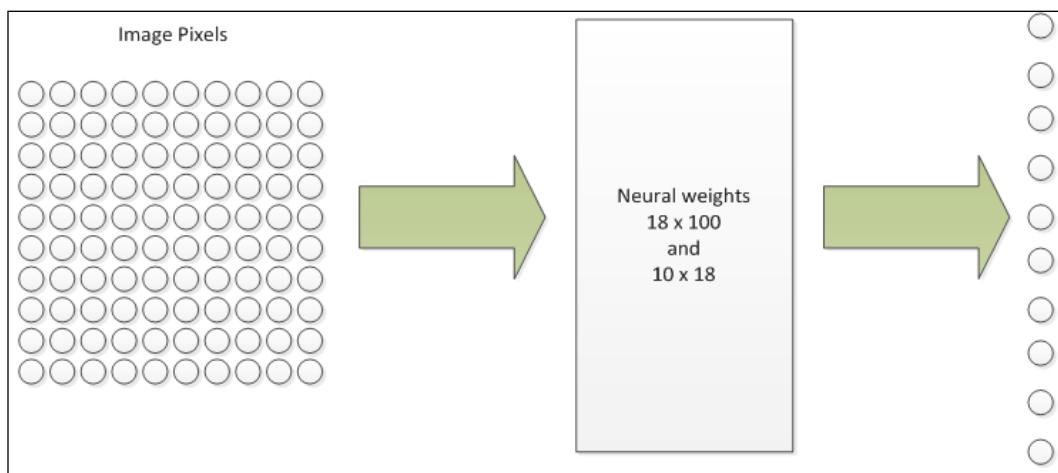
Each matrix row was merged into vectors (D_{train} and D_{test}) to form a pattern that will be used to train and test the neural network. Therefore, the input layer of the neural network will be composed of 101 neurons.

The output dataset was represented by ten patterns. Each one has a more expressive value (one) and the rest of the values are zero. Therefore, the output layer of the neural network will have ten neurons.

Neural architecture

So, in this application our neural network will have 100 inputs (for images that have a 10x10 pixel size) and ten outputs, the number of hidden neurons remaining unrestricted. We created a class called `DigitExample` in the package `examples.chapter08` to handle this application. The neural network architecture was chosen with these parameters:

- **Neural network type:** MLP
- **Training algorithm:** Backpropagation
- **Number of hidden layers:** 1
- **Number of neurons in the hidden layer:** 18
- **Number of epochs:** 1000
- **Minimum overall error:** 0.001



Experiments

Now, as has been done in other cases previously presented, let's find the best neural network topology training several nets. The strategy to do that is summarized in the following table:

Experiment	Learning rate	Activation Functions
#1	0.3	Hidden Layer: SIGLOG
		Output Layer: LINEAR

Experiment	Learning rate	Activation Functions
#2	0.5	Hidden Layer: SIGLOG
		Output Layer: LINEAR
#3	0.8	Hidden Layer: SIGLOG
		Output Layer: LINEAR
#4	0.3	Hidden Layer: HYPERTAN
		Output Layer: LINEAR
#5	0.5	Hidden Layer: SIGLOG
		Output Layer: LINEAR
#6	0.8	Hidden Layer: SIGLOG
		Output Layer: LINEAR
#7	0.3	Hidden Layer: HYPERTAN
		Output Layer: SIGLOG
#8	0.5	Hidden Layer: HYPERTAN
		Output Layer: SIGLOG
#9	0.8	Hidden Layer: HYPERTAN
		Output Layer: SIGLOG

The following `DigitExample` class code defines how to create a neural network to read from digit data:

```
// enter neural net parameter via keyboard (omitted)

// load dataset from external file (omitted)

// data normalization (omitted)

// create ANN and define parameters to TRAIN:
Backpropagation backprop = new Backpropagation(nn,
neuralDataSetToTrain, LearningAlgorithm.LearningMode.BATCH);
backprop.setLearningRate( typedLearningRate );
backprop.setMaxEpochs( typedEpochs );
backprop.setGeneralErrorMeasurement(Backpropagation.ErrorMeasurement.
SimpleError);
backprop.setOverallErrorMeasurement(Backpropagation.ErrorMeasurement.
MSE);
backprop.setMinOverallError(0.001);
backprop.setMomentumRate(0.7);
backprop.setTestingDataSet(neuralDataSetToTest);
backprop.printTraining = true;
```

```
backprop.showPlotError = true;

// train ANN:
try {
    backprop.forward();
    //neuralDataSetToTrain.printNeuralOutput();

    backprop.train();
    System.out.println("End of training");
    if (backprop.getMinOverallError() >= backprop.
getOverallGeneralError()) {
        System.out.println("Training successful!");
    } else {
        System.out.println("Training was unsuccessful");
    }
    System.out.println("Overall Error:" + String.valueOf(backprop.
getOverallGeneralError()));
    System.out.println("Min Overall Error:" + String.valueOf(backprop.
getMinOverallError()));
    System.out.println("Epochs of training:" + String.
valueOf(backprop.getEpoch()));

} catch (NeuralException ne) {
    ne.printStackTrace();
}

// test ANN (omitted)
```

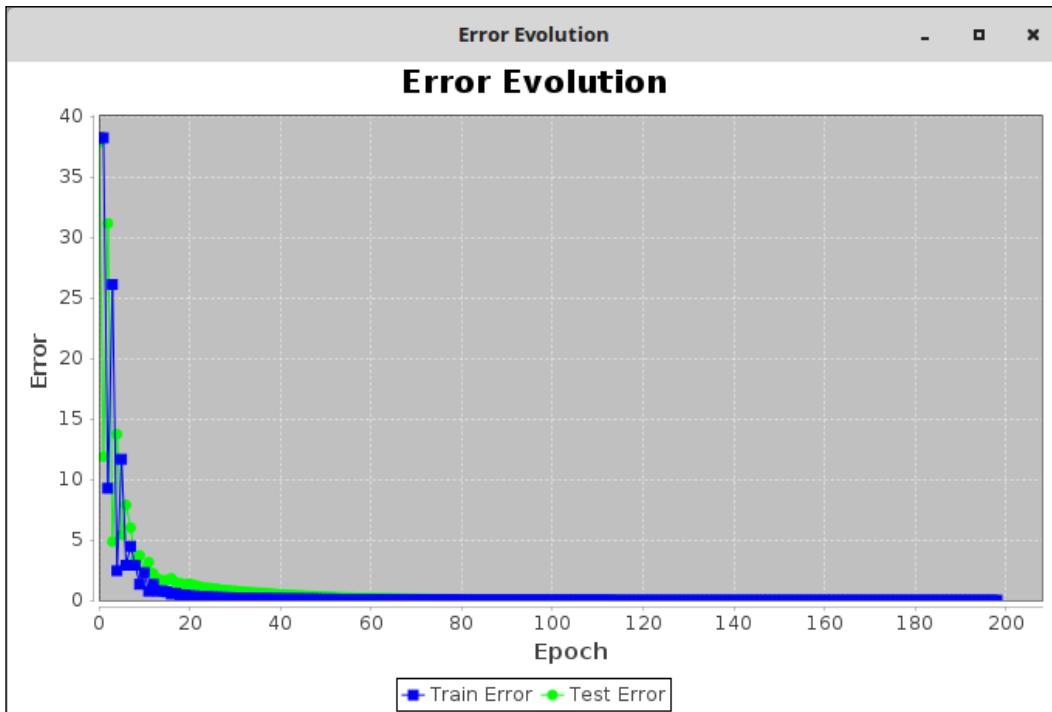
Results

After running each experiment using the `DigitExample` class, excluding training and testing overall errors and the quantity of right number classifications using the test data (table above), it is possible observe that experiments #2 and #4 have the lowest MSE values. The differences between these two experiments are learning rate and activation function used in the output layer.

Experiment	Training overall error	Testing overall error	# Right number classifications
#1	9.99918E-4	0.01221	2 by 10
#2	9.99384E-4	0.00140	5 by 10
#3	9.85974E-4	0.00621	4 by 10
#4	9.83387E-4	0.02491	3 by 10

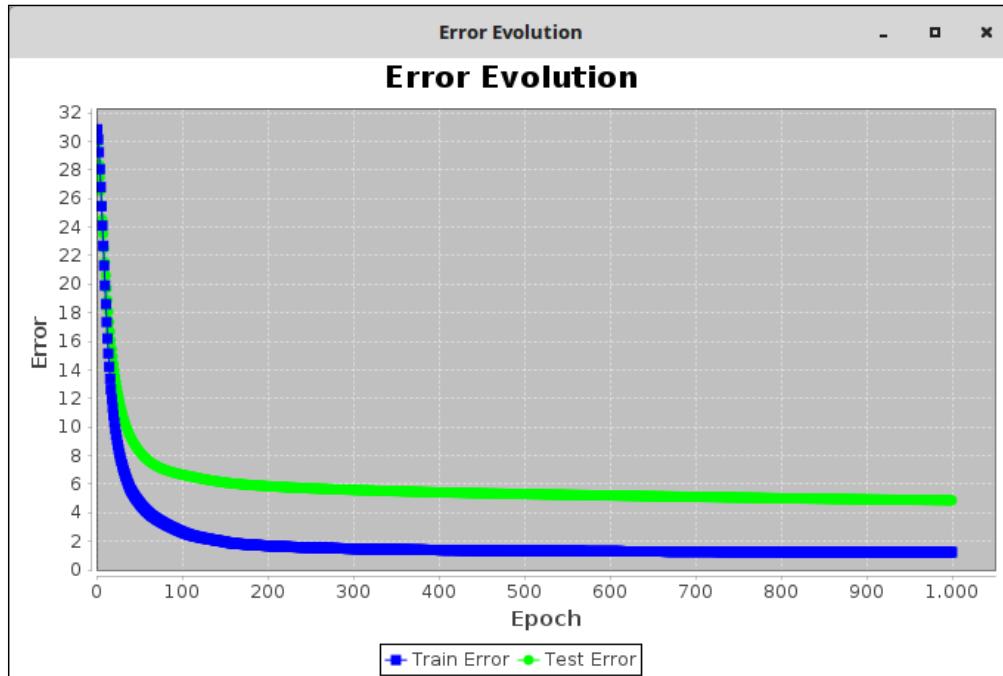
Experiment	Training overall error	Testing overall error	# Right number classifications
#5	9.99349E-4	0.00382	3 by 10
#6	273.70	319.74	2 by 10
#7	1.32070	6.35136	5 by 10
#8	1.24012	4.87290	7 by 10
#9	1.51045	4.35602	3 by 10

The figure above shows the MSE evolution (train and test) by each epoch graphically by experiment #2. It is interesting to notice the curve stabilizes near the 30th epoch:



Text Recognition

The same graphic analysis was performed for experiment #8. It is possible to check the MSE curve stabilizes near the 200th epoch.



As already explained, only MSE values might not be considered to attest neural net quality. Accordingly, the test dataset has verified the neural network generalization capacity. The next table shows the comparison between real output with noise and the neural net estimated output of experiment #2 and #8. It is possible to conclude that the neural network weights by experiment #8 can recognize seven digits patterns better than #2's:

Output comparison											
Real output (test dataset)											Digit
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0		0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0		1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0		2
0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0		3
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0		4
0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0		5
0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0		6
0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0		7
0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0		8
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0		9
Estimated output (test dataset) - Experiment #2											Digit
0.20	0.26	0.09	-0.09	0.39	0.24	0.35	0.30	0.24	1.02		0 (OK)
0.42	-0.23	0.39	0.06	0.11	0.16	0.43	0.25	0.17	-0.26		1 (ERR)
0.51	0.84	-0.17	0.02	0.16	0.27	-0.15	0.14	-0.34	-0.12		2 (ERR)
-0.20	-0.05	-0.58	0.20	-0.16	0.27	0.83	-0.56	0.42	0.35		3 (OK)
0.24	0.05	0.72	-0.05	-0.25	-0.38	-0.33	0.66	0.05	-0.63		4 (ERR)
0.08	0.41	-0.21	0.41	0.59	-0.12	-0.54	0.27	0.38	0.00		5 (OK)
-0.76	-0.35	-0.09	1.25	-0.78	0.55	-0.22	0.61	0.51	0.27		6 (OK)
-0.15	0.11	0.54	-0.53	0.55	0.17	0.09	-0.72	0.03	0.12		7 (ERR)
0.03	0.41	0.49	-0.44	-0.01	0.05	-0.05	-0.03	-0.32	-0.30		8 (ERR)
0.63	-0.47	-0.15	0.17	0.38	-0.24	0.58	0.07	-0.16	0.54		9 (OK)

Output comparison											
Estimated output (test dataset) – Experiment #8											Digit
0.10	0.10	0.12	0.10	0.12	0.13	0.13	0.26	0.17	0.39		0 (OK)
0.13	0.10	0.11	0.10	0.11	0.10	0.29	0.23	0.32	0.10		1 (OK)
0.26	0.38	0.10	0.10	0.12	0.10	0.10	0.17	0.10	0.10		2 (OK)
0.10	0.10	0.10	0.10	0.10	0.17	0.39	0.10	0.38	0.10		3 (ERR)
0.15	0.10	0.24	0.10	0.10	0.10	0.39	0.37	0.10			4 (OK)
0.20	0.12	0.10	0.10	0.37	0.10	0.10	0.10	0.17	0.12		5 (ERR)
0.10	0.10	0.10	0.39	0.10	0.16	0.11	0.30	0.14	0.10		6 (OK)
0.10	0.11	0.39	0.10	0.10	0.15	0.10	0.10	0.17	0.10		7 (OK)
0.10	0.25	0.34	0.10	0.10	0.10	0.10	0.10	0.10	0.10		8 (ERR)
0.39	0.10	0.10	0.10	0.28	0.10	0.27	0.11	0.10	0.21		9 (OK)



The experiments showed in this chapter have taken in consideration 10x10 pixel information images. We recommend that you try to use 20x20 pixel datasets to build a neural net able to classify digit images of this size.

You should also change the training parameters of the neural net to achieve better classifications.

Summary

In this chapter we've seen the power of neural networks in recognizing digits from 0 to 9 in images. Although the coding of the digits was very small in 10x10 images, it was important to understand the concept in practice. Neural networks are capable of learning from data, and provided that real-world representations can be transformed into data, it is reasonable to take into account that character recognition can be a very good example of the application in pattern recognition. The application here can be extended to any type of characters, under the condition that the neural network should all be presented with the predefined characters.

9

Optimizing and Adapting Neural Networks

In this chapter, the reader will be presented with techniques that help to optimize neural networks, in order to get the best performance. Tasks such as input selection, dataset separation and filtering, choosing the number of hidden neurons, and cross-validation strategies are examples of what can be adjusted to improve a neural network's performance. Furthermore, this chapter focuses on methods for adapting neural networks to real-time data. Two implementations of these techniques are presented here. Application problems will be selected for exercises. This chapter deals with the following topics:

- Input selection
- Dimensionality reduction
- Data filtering
- Structure selection
- Pruning
- Validation strategies
- Cross-validation
- Online retraining
- Stochastic online learning
- Adaptive neural networks
- Adaptive resonance theory

Common issues in neural network implementations

When developing a neural network application, it is quite common to face problems regarding how accurate the results are. The source of these problems can be various:

- Bad input selection
- Noisy data
- Too big a dataset
- Unsuitable structure
- Inadequate number of hidden neurons
- Inadequate learning rate
- Insufficient stop condition
- Bad dataset segmentation
- Bad validation strategy

The design of a neural network application sometimes requires a lot of patience and the use of trial and error methods. There is no methodology stating specifically which number of hidden units and/or architecture should be used, but there are recommendations on how to choose these parameters properly. Another issue programmers may face is a long training time, which often causes the neural network to not learn the data. No matter how long the training runs, the neural network won't converge.



Designing a neural network requires the programmer or designer to test and redesign the neural structure as many times as needed, until an acceptable result is obtained.



On the other hand, the neural network solution designer may wish to improve the results. Because a neural network can learn until the learning algorithm reaches the stop condition, the number of epochs or the mean squared error, the results are not accurate enough or not generalized. This will require a redesign of the neural structure, or a new dataset selection.

Input selection

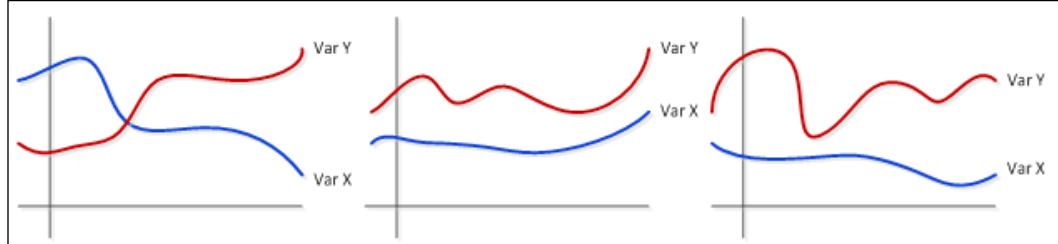
One of the key tasks in designing a neural network application is to select appropriate inputs. For the unsupervised case, one wishes to use only relevant variables on which the neural network will find the patterns. And for the supervised case, there is a need to map the outputs to the inputs, so one needs to choose only the input variables which somewhat have influence on the output.

Data correlation

One strategy that helps in selecting good inputs in the supervised case is the correlation between data series, which is implemented in *Chapter 5, Forecasting Weather*. A correlation between data series is a measure of how one data sequence reacts or influences the other. Suppose we have one dataset containing a number of data series, from which we choose one to be an output. Now we need to select the inputs from the remaining variables.

The correlation takes values from -1 to 1, where values near to +1 indicate a positive correlation, values near -1 indicate a negative correlation, and values near 0 indicate no correlation at all.

As an example, let's see three charts of two variables X and Y:



In the first chart, to the left, visually one can see that as one variable decreases, the other increases its value (corr. -0.8). The middle chart shows the case when the two variables vary in the same direction, therefore positive correlation (corr. +0.7). The third chart, to the right, shows a case where there is no correlation between the variables (corr. -0.1).

There is no threshold rule as to which correlation should be taken into account as a limit; it depends on the application. While absolute correlation values greater than 0.5 may be suitable for one application, in others, values near 0.2 may add a significant contribution.

Transforming data

Linear correlation is very good in detecting behaviors between data series when they are presumably linear. However, if two data series form a parable when plotted together, linear correlation won't be able to identify any relation. That's why sometimes we need to transform data into a view that exhibits a linear correlation.

Data transformation depends on the problem that is being faced. It consists of inserting an additional data series with processed data from one or more data series. One example is an equation (possibly nonlinear) that includes one or more parameters. Some behaviors are more detectable under a transformed view of the data.

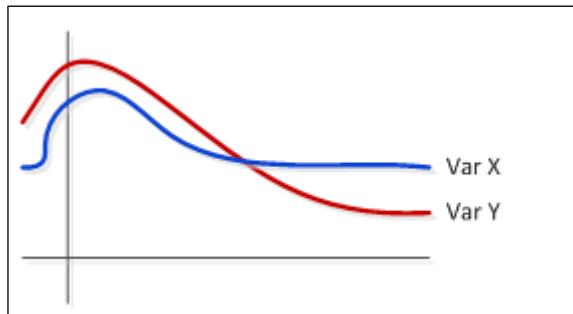


Data transformation also involves a bit of knowledge about the problem. However, by seeing the scatter plot of two data series, it becomes easier to choose which transformation to apply.



Dimensionality reduction

Another interesting point is regarding removing redundant data. Sometimes this is desired when there is a lot of available data in both unsupervised and supervised learning. As an example, let's see a chart of two variables:



It can be seen that both X and Y variables share the same shape, so this can be interpreted as a redundancy, as both variables are carrying almost the same information due to the high positive correlation. Thus, one can consider a technique called **Principal Component Analysis (PCA)** which gives a good approach for dealing with these cases.

The result of PCA will be a new variable summarizing the previous two (or more). Basically, the original data series are subtracted by the mean and then multiplied by the transposed eigenvectors of the covariance matrix:

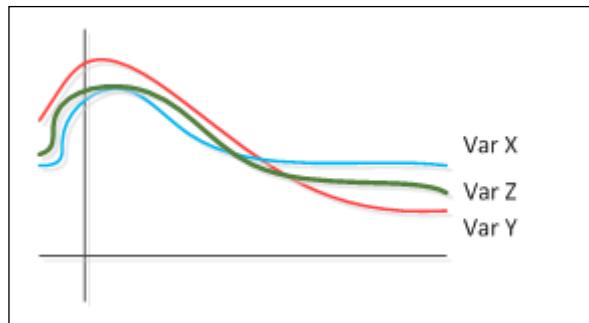
$$S = \begin{bmatrix} S_{XX} & S_{XY} \\ S_{YX} & S_{YY} \end{bmatrix}$$

Here, S_{XY} is the covariance between the variables X and Y .

The derived new data will be then:

$$Z = \text{eig}(S)^T [X - E[X] Y - E[Y]]$$

Let's see now what a new variable would look like in a chart, compared to the original ones:



In our framework, we are going to add the class `PCA` that will perform this transformation and preprocessing before applying the data into a neural network:

```
public class PCA {

    DataSet originalDS;
    int numberOfDimensions;
    DataSet reducedDS;

    DataNormalization normalization = new DataNormalization(DataNormaliz
    ation.NormalizationTypes.ZSCORE);

    public PCA(DataSet ds, int dimensions) {
        this.originalDS=ds;
        this.numberOfDimensions=dimensions;
```

```
    }

    public DataSet reduceDS() {
        //matrix algebra to calculate transformed data in lower dimension
        ...
    }

    public DataSet reduceDS(int numberOfDimensions) {
        this.numberOfDimensions = numberOfDimensions;
        return reduceDS;
    }

}
```

Data filtering

Noisy data and bad data are also sources of problems in neural network applications; that's why we need to filter data. One of the common data filtering techniques can be performed by excluding the records that exceed the usual range. For example, temperature values are between -40 and 40, so a value such as 50 would be considered an outlier and could be taken out.

The 3-sigma rule is a good and effective measure for filtering. It consists in filtering the values that are beyond three times the standard deviation from the mean:

$$d_i = \left| \frac{X_i - E[X]}{\text{std}(X)} \right| \leq 3$$

Let's add a class to deal with data filtering:

```
public abstract class DataFiltering {

    DataSet originalDS;
    DataSet filteredDS;

}

public class ThreeSigmaRule extends DataFiltering {

    double thresholdDistance = 3.0;

    public ThreeSigmaRule(DataSet ds, double threshold) {
```

```

        this.originalDS=ds;
        this.thresholdDistance=threshold;
    }

    public DataSet filterDS(){
        //matrix algebra to calculate the distance of each point in each
        column
        ...
    }
}

```

These classes can be called in `DataSet` by the following methods, which are then called elsewhere for filtering and reducing dimensionality:

```

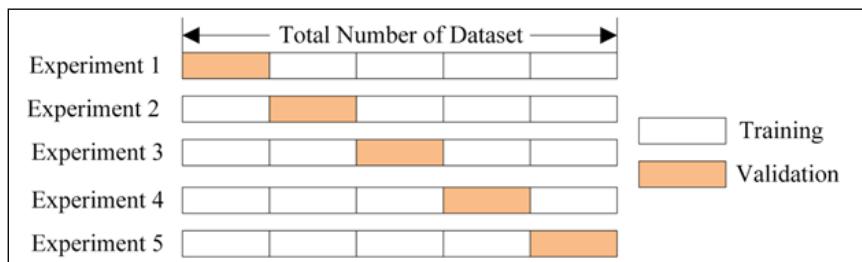
public DataSet applyPCA(int dimensions){
    PCA pca = new PCA(this,dimensions);
    return pca.reduceDS();
}

public DataSet filter3Sigma(double threshold){
    ThreeSigmaRule df = new ThreeSigmaRule(this,threshold);
    return df.filterDS();
}

```

Cross-validation

Among a number of strategies for validating a neural network, one very important one is cross-validation. This strategy ensures that all data has been presented to the neural network as training and test data. The dataset is partitioned into K groups, of which one is separated for testing while the others are for training:



In our code, let's create a class called `CrossValidation` to manage cross-validation:

```
public class CrossValidation {
    NeuralDataSet dataSet;
    int numberOfFolds;

    public LearningAlgorithm la;

    double[] errorsMSE;

    public CrossValidation(LearningAlgorithm _la, NeuralDataSet _nds, int
    _folds){
        this.dataSet=_nds;
        this.la=_la;
        this.numberOfFolds=_folds;
        this.errorsMSE=new double[_folds];
    }

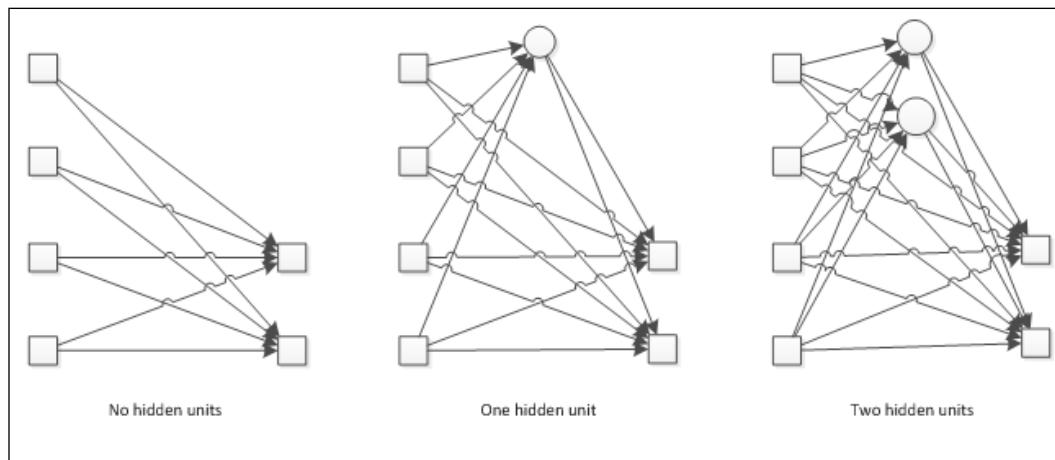
    public void performValidation() throws NeuralException{
        //shuffle the dataset
        NeuralDataSet shuffledDataSet = dataSet.shuffle();
        int subSize = shuffledDataSet.numberOfRecords/numberOfFolds;
        NeuralDataSet[] foldedDS = new NeuralDataSet[numberOfFolds];
        for(int i=0;i<numberOfFolds;i++){
            foldedDS[i]=shuffledDataSet.subDataSet(i*subSize,(i+1)*su
            bSize-1);
        }
        //run the training
        for(int i=0;i<numberOfFolds;i++){
            NeuralDataSet test = foldedDS[i];
            NeuralDataSet training = foldedDS[i==0?1:0];
            for(int k=1;k<numberOfFolds;k++){
                if((i>0)&&(k!=i)){
                    training.append(foldedDS[k]);
                }
                else if(k>1) training.append(foldedDS[k]);
            }
            la.setTrainingDataSet(training);
            la.setTestingDataSet(test);
            la.train();
            errorsMSE[i]=la.getMinOverallError();
        }
    }
}
```

Structure selection

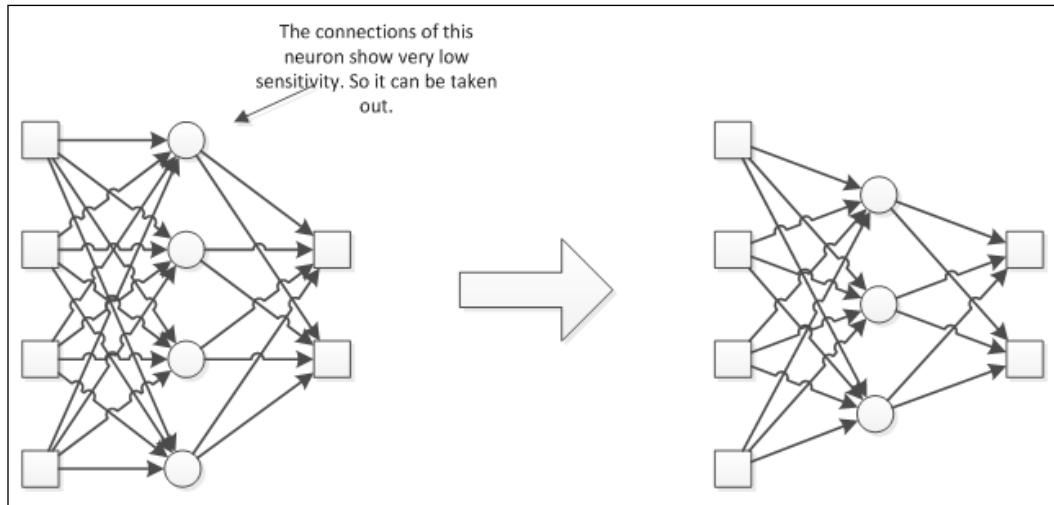
To choose an adequate structure for a neural network is also a very important step. However, this is often done empirically, since there is no rule on how many hidden units a neural network should have. The only measure of how many units are adequate is the neural network performance. One assumes that if the general error is low enough, then the structure is suitable. Nevertheless, there might be a smaller structure that could yield the same result.

In this context, there are basically two methodologies: constructive and pruning. The constructive consists in starting with only the input and output layers, then adding new neurons at a hidden layer, until a good result can be obtained. The destructive approach, also known as pruning, works on a bigger structure on which the neurons having few contributions to the output are taken out.

The constructive approach is depicted in the following figure:



Pruning is the way back: when given a high number of neurons, one wishes to *prune* those whose sensitivity is very low, that is, whose contribution to the error is minimal:



To implement pruning, we've added the following properties in the class NeuralNet:

```
public class NeuralNet{
//...
    public Boolean pruning;
    public double sensitivityThreshold;
}
```

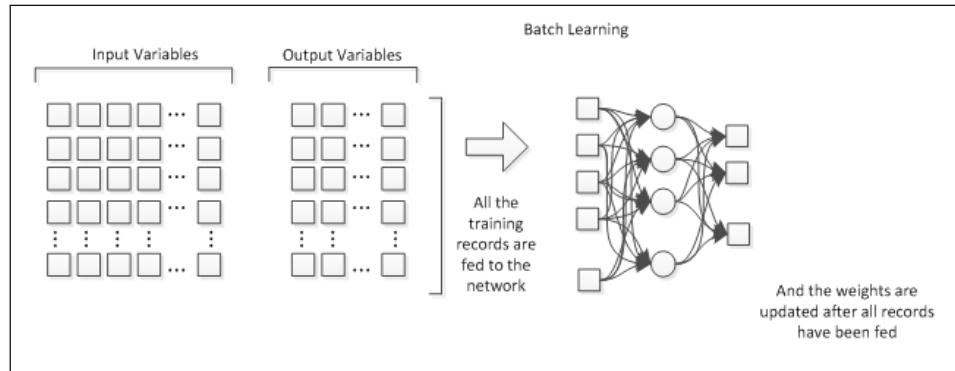
A method called `removeNeuron` in the class `NeuralLayer`, which actually sets all the connections of the neuron to zero, disables weight updating and fires only zero at the neuron's output. This method is called if the property `pruning` of the `NeuralNet` object is set to true. The sensitivity calculation is according to the chain rule, as shown in *Chapter 3, Perceptrons and Supervised Learning* and implemented in the `calcNewWeight` method:

```
@Override
public Double calcNewWeight(int layer,int input,int neuron) {
    Double deltaWeight=calcDeltaWeight(layer,input,neuron);
    if(this.neuralNet.pruning){
        if(deltaWeight<this.neuralNet.sensitivityThreshold)
            neuralNet.getHiddenLayer(layer).remove(neuron);
    }
    return newWeights.get(layer).get(neuron).get(input)+deltaWeight;
}
```

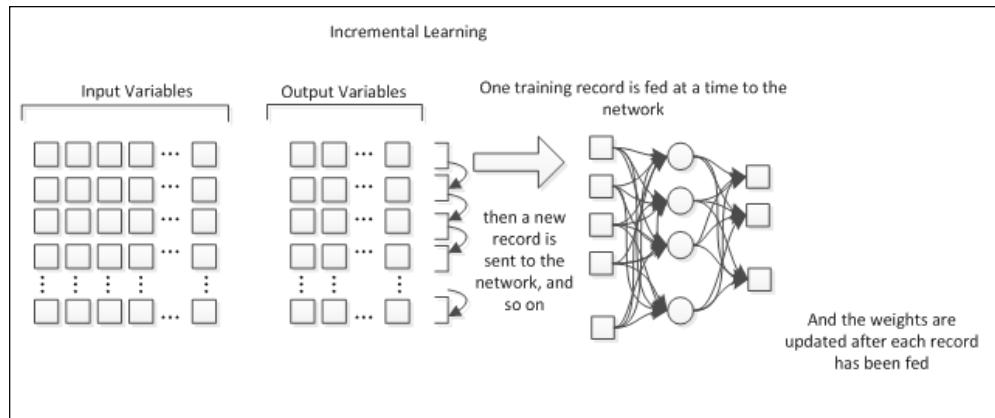
Online retraining

During the learning process, it is important to design how the training should be performed. Two basic approaches are batch and incremental learning.

In batch learning, all the records are fed to the network, so it can evaluate the error and then update the weights:



In incremental learning, the update is performed after each record has been sent to the network:



Both approaches work well and have advantages and disadvantages. While batch learning can be used for a less frequent, though more directed, weight update, incremental learning provides a method for fine-tuned weight adjustment. In that context, it is possible to design a mode of learning that enables the network to learn continually.



As a suggested exercise, the reader may pick one of the datasets available in the code and design a training using part of the records, and then train using another part in both modes, online and batch. See the `IncrementalLearning.java` file for details.

Stochastic online learning

Offline learning means that the neural network learns while not in *operation*. Every neural network application is supposed to work in an environment, and in order to be at production, it should be properly trained. Offline training is suitable for putting the network into operation, since its outputs may be varied over large ranges of values, which would certainly compromise the system, if it is in operation. But when it comes to online learning, there are restrictions. While in offline learning, it's possible to use cross-validation and bootstrapping to predict errors, in online learning, this can't be done since there's no "training dataset" anymore. However, one would need online training when some improvement in the neural network's performance is desired.

A stochastic method is used when online learning is performed. This algorithm to improve neural network training is composed of two main features: random choice of samples for training and variation of learning rate in runtime (online). This training method has been used when noise is found in the objective function. It helps to escape the local minimum (one of the best solutions) and to reach the global minimum (the best solution).

The pseudo-algorithm is displayed below (source: ftp://ftp.sas.com/pub/neural/FAQ2.html#A_styles):

```
Initialize the weights.  
Initialize the learning rate.  
Repeat the following steps:  
  Randomly select one (or possibly more) case(s)  
    from the population.  
  Update the weights by subtracting the gradient  
    times the learning rate.  
  Reduce the learning rate according to an  
    appropriate schedule.
```

Implementation

The Java project has created the class `BackpropagationOnline` inside the `learn` package. The differences between this algorithm and classic Backpropagation was programmed by changing the `train()` method, by adding two new methods: `generateIndexRandomList()` and `reduceLearningRate()`. The first one generates a random list of indexes to be used in the training step and the second one executes the learning rate online variation according to the following heuristic:

```
private double reduceLearningRate(NeuralNet n, double percentage) {
    double newLearningRate = n.getLearningRate() *
        ((100.0 - percentage) / 100.0);

    if(newLearningRate < 0.1) {
        newLearningRate = 1.0;
    }

    return newLearningRate;
}
```

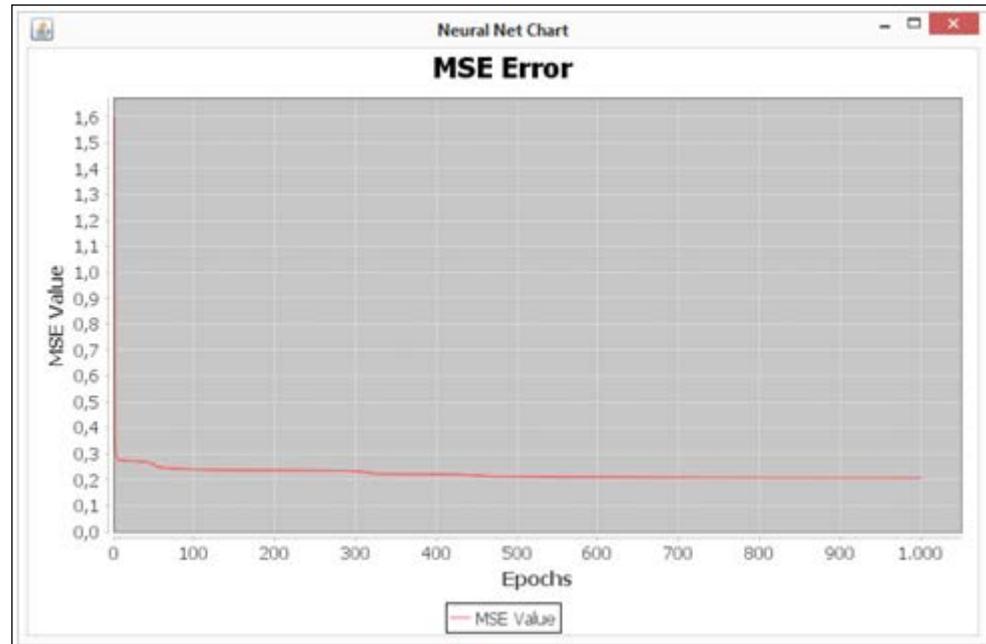
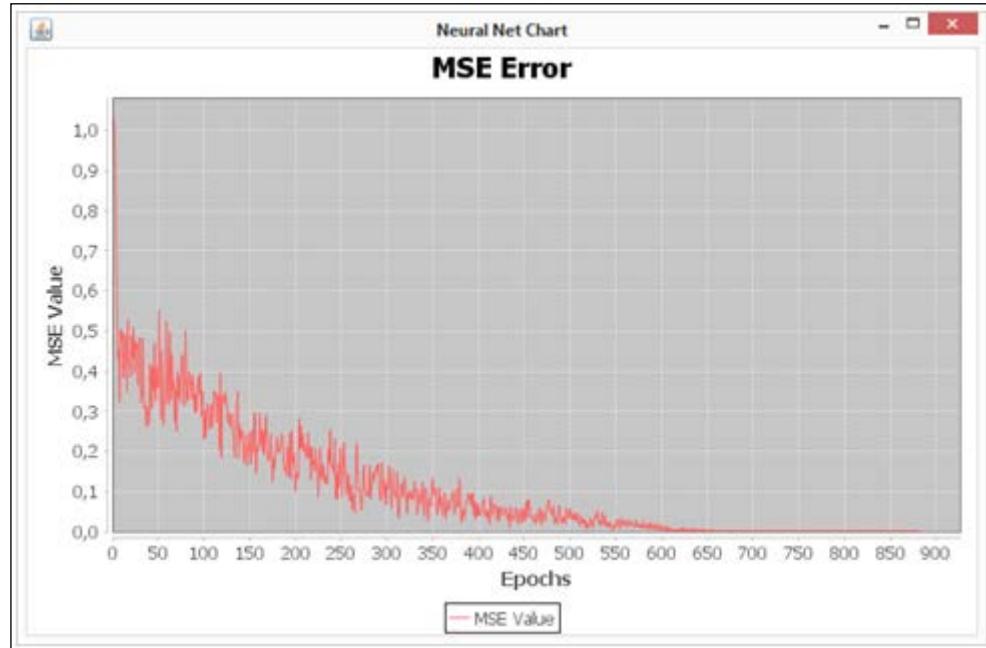
This method will be called at the end of the `train()` method.

Application

It has used data from previous chapters to test this new way to train neural nets. The same neural net topology defined in each chapter (*Chapter 5, Forecasting Weather* and *Chapter 8, Text Recognition*) has been used to train the nets of this chapter. The first one is the weather forecasting problem and the second one is the OCR. The following table shows the comparison of results:

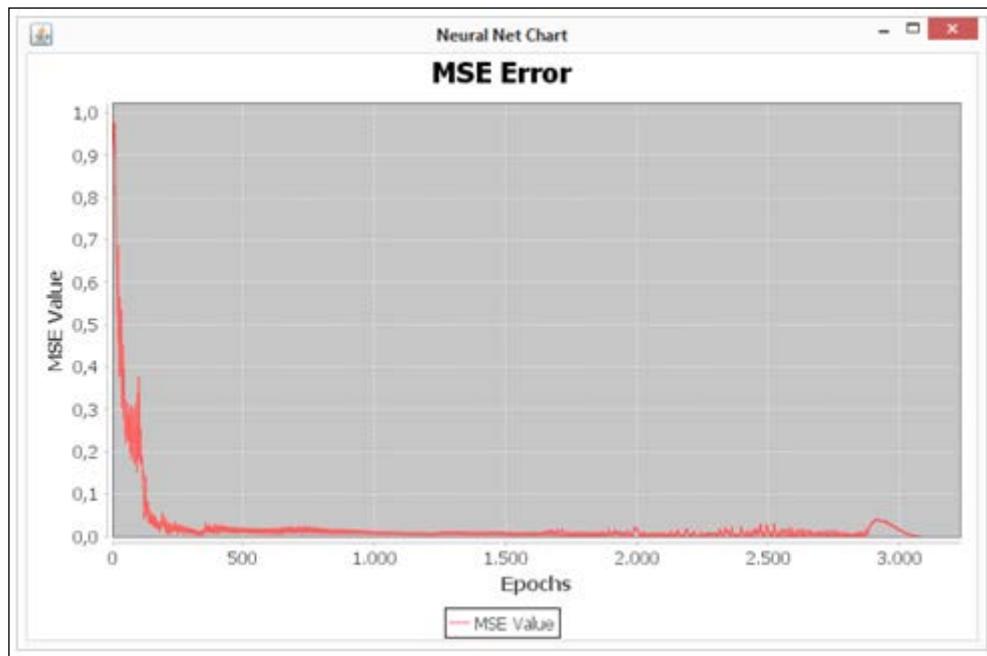
Values	Forecast Weather	OCR
Classic Backpropagation Learning Rate	0.5	0.5
Classic Backpropagation MSE Value	0.2877786584	0.0011981712
On-line Backpropagation Learning Rate	Found: $\cong 0.15$	Found: $\cong 0.40$
On-line Backpropagation MSE Value	0.4618623052	9.977909980E-6

In addition, charts of the MSE evolution have been plotted and are shown here:



The curve showed in the first chart (Weather Forecast) has a saw shape, because of the variation of learning rate. Besides, it's very similar to the curve, as shown in *Chapter 5, Forecasting Weather*. On the other hand, the second chart (OCR) shows that the training process was faster and stops near the 900th epoch because it reached a very small MSE error.

Other experiments were made: training neural nets with a backpropagation algorithm, and considering the learning rate found by the online approach. The MSE values reduced in both problems:



Another important observation consists in the fact that training process demonstrated by the training terminated almost in the 3,000th epoch. Therefore, it's faster and better than the training process seen in *Chapter 8, Text Recognition* using the same algorithm.

Adaptive neural networks

Analogous to human learning, neural networks may also work in order not to forget previous knowledge. Using the traditional approaches for neural learning, this is nearly impossible, due to the fact that every training implies replacing all the connections already made by new ones, thereby *forgetting* the previous knowledge. Thus a need arises to make the neural networks adapt to new knowledge by incrementing instead of replacing their current knowledge. To address that issue, we are going to explore one method called **adaptive resonance theory (ART)**.

Adaptive resonance theory

The question that drove the development of this theory was: *How can an adaptive system remain plastic to a significant input and yet keep stability for irrelevant inputs?* In other words: *How can it retain previously learned information while learning new information?*

We've seen that competitive learning in unsupervised learning deals with pattern recognition, whereby similar inputs yield similar outputs or fire the same neurons. In an ART topology, the resonance comes in when the information is being retrieved from the network, by providing a feedback from the competitive layer and the input layer. So, while the network receives data to learn, there is an oscillation resulting from the feedback between the competitive and input layers. This oscillation stabilizes when the pattern is fully developed inside the neural network. This resonance then reinforces the stored pattern.

Implementation

A new class called ART has created into the some package, inheriting from CompetitiveLearning. Besides other small contributions, its great change is the vigilance test:

```
public class ART extends CompetitiveLearning{  
  
    private boolean vigilanceTest(int row_i) {  
        double v1 = 0.0;  
        double v2 = 0.0;  
  
        for (int i = 0; i < neuralNet.getNumberOfInputs(); i++) {  
            double weightIn  = neuralNet.getOutputLayer().getWeight(i);  
            double trainPattern = trainingDataSet.getithInput(row_i)[i];  
        }  
    }  
}
```

```

    v1 = v1 + (weightIn * trainPattern) ;

    v2 = v2 + (trainPattern * trainPattern) ;
}

double vigilanceValue = v1 / v2;

if(vigilanceValue > neuralNet.getMatchRate()) {
    return true;
} else {
    return false;
}

}

```

The training method is shown below. It's possible to notice that, firstly, global variables and the neural net are initialized; after that, the number of training sets and the training patterns are stored; then the training process begins. The first step of this process is to calculate the index of the winner neuron; the second is make attribution of the neural net output. The next step consists of verifying whether the neural net has learned or not, whether it has learned that weights are fixed; if not, another training sample is presented to the net:

```

epoch=0;
int k=0;
forward();
//...
currentRecord=0;
forward(currentRecord);
while(!stopCriteria()){
    //...
    boolean isMatched = this.vigilanceTest(currentRecord);
    if ( isMatched ) {
        applyNewWeights();
    }
}

```

Summary

In this chapter, we've seen a few topics that make a neural network work better, either by improving its accuracy or by extending its knowledge. These techniques help a lot in designing solutions with artificial neural networks. The reader is welcome to apply this framework in any desired task that neural networks can be used on, in order to explore the enhanced power that these structures can have. Even simple details such as selecting input data may influence the entire learning process, as well as filtering bad data or eliminating redundant variables. We demonstrated two implementations, two strategies that help to improve the performance of a neural network: stochastic online learning and adaptive resonance theory. These methodologies enable the network to extend its knowledge and therefore adapt to new, changing environments.

10

Current Trends in Neural Networks

This final chapter shows the reader the most recent trends in neural networks. Although this book is introductory, it is always useful to be aware of the latest developments and where the science behind this theory is going to. Among the latest advancements is the so-called **deep learning**, a very popular research field for many data scientists; this type of network is briefly covered in this chapter. Convolutional and cognitive architectures are also in this trend and gaining popularity for multimedia data recognition. Hybrid systems that combine different architectures are a very interesting strategy for solving more complex problems, as well as applications that involve analytics, data visualization, and so on. Being more theoretical, there is no actual implementation of the architectures, although an example of implementation for a hybrid system is provided. Topics covered in this chapter include:

- Deep learning
- Convolutional neural networks
- Long short term memory networks
- Hybrid systems
- Neuro-Fuzzy
- Neuro-Genetic
- Implementation of a hybrid neural network

Deep learning

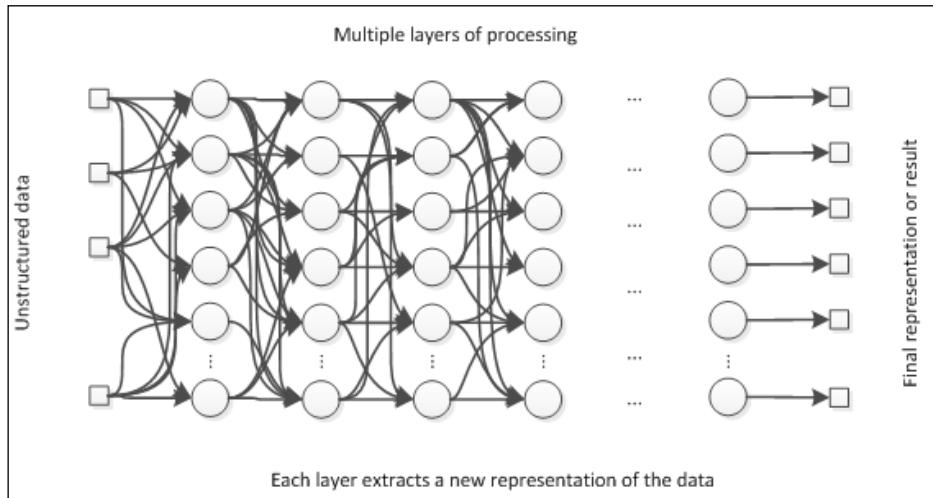
One of the latest advancements in neural networks is the so-called deep learning. Nowadays it is nearly impossible to talk about neural networks without mentioning deep learning, because the recent research on feature extraction, data representation, and transformation has found that many layers of processing information are able to abstract and produce better representations of data for learning. Throughout this book we have seen that neural networks require input data in numerical form, no matter if the original data is categorical or binary, neural networks cannot process non-numerical data directly. But it turns out that in the real world most of the data is non-numerical or is even unstructured, such as images, videos, audios, texts, and so on.

In this sense a deep network would have many layers that could act as data processing units to transform this data and provide it to the next layer for subsequent data processing. This is analogous to the process that happens in the brain, from the nerve endings to the cognitive core; in this long path the signals are processed by multiple layers before resulting in signals that control the human body. Currently, most of the research on deep learning has been on the processing of unstructured data, particularly image and sound recognition and natural language processing.



Deep learning is still under development and much has changed since 2012. Big companies such as Google and Microsoft have teams for research on this field and much is likely to change in the next couple of years.

A scheme of a deep learning architecture is shown in the following figure:



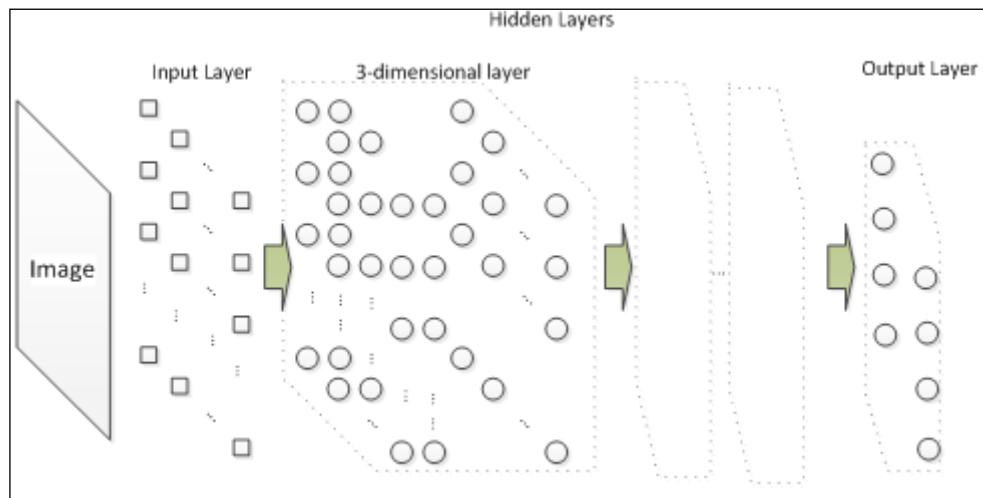
On the other hand, deep neural networks have some problems that need to be overcome. The main problem is overfitting. The many layers that produce new representations of data are very sensitive to the training data, because the deeper the signals reach in the neural layers, the more specific the transformation will be for the input data. Regularization methods and pruning are often applied to prevent overfitting. Computation time is another common issue in training deep networks. The standard backpropagation algorithm can take a very long time to train a deep neural network, although strategies such as selecting a smaller training dataset can speed up the training time. In addition, in order to train a deep neural network, it is often recommended to use a faster machine and parallelize the training as much as possible.

Deep architectures

There is a great variety of deep neural architectures with both feedforward and feedback flows, although they are typically feedforward. Main architectures are, without limitation to:

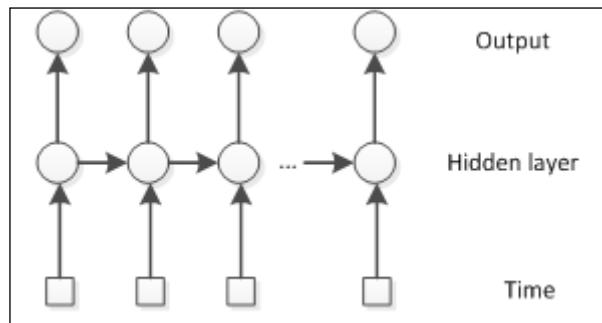
Convolutional neural network

In this architecture, the layers may have multidimensional organization. Inspired by the visual cortex of animals, the typical dimensionality applied to the layers is three-dimensional. In **convolutional neural networks (CNNs)**, part of the signals of a preceding layer is fed into another part of neurons in the following layer. This architecture is feedforward and is well applied for image and sound recognition. The main feature that distinguishes this architecture from Multilayer Perceptrons is the partial connectivity between layers. Considering the fact that not all neurons are relevant for a certain neuron in the next layer, the connectivity is local and respects the correlation between neurons. This prevents both long time training and overfitting, provided that a fully connected MLP blows up the number of weight as the dimension of images grows, for example. In addition, neurons in layers are arranged in dimensions, typically three, thereby staked in an array in width, height, and depth.

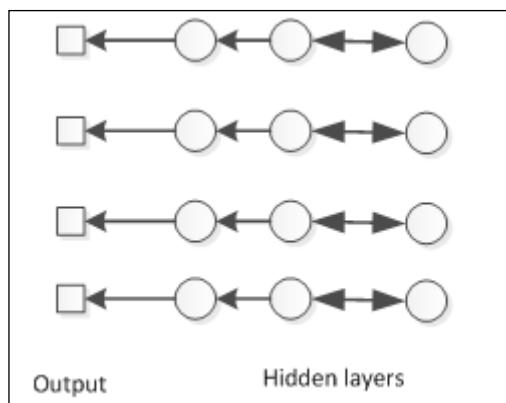


In this architecture, the layers may have multidimensional organization. Inspired by the visual cortex of animals, the typical dimensionality applied to the layers is three-dimensional. In **convolutional neural networks** (CNNs), part of the signals of a preceding layer is fed into another part of neurons in the following layer. This architecture is feedforward and is well applied for image and sound recognition. The main feature that distinguishes this architecture from multilayer perceptrons is the partial connectivity between layers. Considering the fact that not all neurons are relevant for a certain neuron in the next layer, the connectivity is local and respects the correlation between neurons. This prevents both long time training and overfitting, provided that a fully connected MLP blows up the number of weight as the dimension of images grows, for example. In addition, neurons in layers are arranged in dimensions, typically three, thereby staked in an array in width, height, and depth.

Long short-term memory: This is a recurrent type of neural network that takes into account always the last value of the hidden layer, exactly like a **hidden Markov model (HMM)**. A **Long Short Time Memory network (LSTM)** has LSTM units instead of traditional neurons, and these units implement operations such as store and forget a value to control the flow in a deep network. This architecture is well applied to natural language processing, due to the capacity of retaining information for a long time while receiving completely unstructured data such as audio or text files. One way to train this type of network is the backpropagation through time (BPTT) algorithm, but there are also other algorithms such as reinforcement learning or evolution strategies.



Deep belief network: Deep belief networks (DBN's) are probabilistic models where layers are classified into visible and hidden. This is also a type of recurrent neural network based on a **restricted Boltzmann machine (RBM)**. It is typically used as a first step in the training of a **deep neural network (DNN)**, which is further trained by other supervised algorithms such as backpropagation. In this architecture each layer acts like a feature detector, abstracting new representations of data. The visible layer acts both as an output and as an input, and the deepest hidden layer represents the highest level of abstraction. Applications of this architecture are typically the same as those of convolutional neural networks.



How to implement deep learning in Java

Because this book is introductory, we are not diving into further details on deep learning in this chapter. However, some recommendations of code for a deep architecture are provided. An example on how a convolutional neural network would be implemented is provided here. One needs to implement a class called `ConvolutionalLayer` to represent a multidimensional layer, and a `CNN` class for the convolutional neural network itself:

```
public class ConvolutionalLayer extends NeuralLayer{
    int height, width, depth;
    //...
    ArrayList<ArrayList<ArrayList<Neuron>>> neurons;
    Map<Neuron, Neuron> connections;
    ConvolutionalLayer previousLayer;

    //the method call should take into account the mapping
```

```
// between neurons from different layers
@Override
public void calc(){
    ArrayList<ArrayList<ArrayList<double>>> inputs;
    foreach(Neuron n:neurons){
        foreach(Neuron m:connections.keySet()) {
            // here we get only the inputs that are connected to the neuron
        }
    }
}

public class CNN : NeuralNet{
    int depth;
    ArrayList<ConvolutionalLayer> layers;
//...
@Override
public void calc(){
    //here we perform the calculation for each layer,
    //taking into account the connections between layers
}
}
```

In this class, the neurons are organized in dimensions and methods for pruning are used to make the connections between the layers. Please see the files `ConvolutionalLayer.java` and `CNN.java` for further details.

Since the other architectures are recurrent and this book does not cover the recurrent neural networks (for simplicity purposes in an introductory book) they are provided only for the reader's information. We suggest the reader to take a look at the references provided to find out more on these architectures.

Hybrid systems

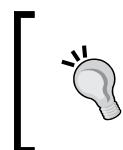
In machine learning, or even in the artificial intelligence field, there are many other algorithms and techniques other than neural networks. Each technique has its strengths and drawbacks, and that inspires many researchers to combine them into a single structure. Neural networks are part of the connectionist approach for artificial intelligence, whereby operations are performed on numerical and continuous values; but there are other approaches that include cognitive (rule-based systems) and evolutionary computation.

Connectionist	Cognitive	Evolutionary
Numerical processing	Symbolic processing	Numerical and symbolic processing
Large network structures	Large rule bases and premises	Large quantity of solutions
Performance by statistics	Design by experts/statistics	Better solutions are produced every iteration
Highly sensitive to data	Highly sensitive to theory	Local minima proof

The main representative of connectionism is the neural network, which has a lot of different architectures for a variety of purposes. Some neural networks, such as **multilayer perceptrons (MLP)**, are good at mapping nonlinear input-output behaviors, while others such as **self-organizing maps (SOM)** are good at finding patterns in the data. Some architecture, such as **radial basis function (RBF)** networks, combines multiple features in different steps of training and processing.

One motivation for using hybrid neural systems is common to one of the foundations of deep learning, which is the feature extraction. Tasks like image recognition become very tough to deal with when resolution is very high; however, if that data can be compacted or reduced, the processing becomes much simpler.

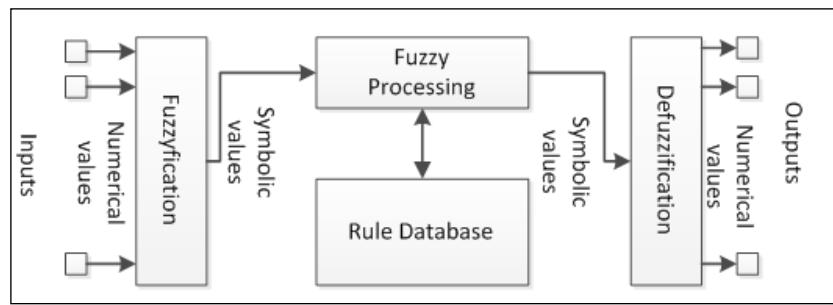
Combining multiple approaches for artificial intelligence is also interesting, although it becomes more complex. In this context, let's review two strategies: neuro-fuzzy and neuro-genetic.



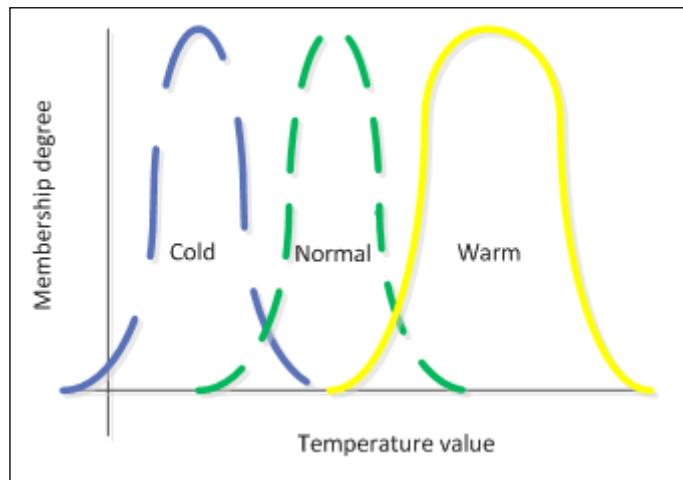
Considering that the concepts addressed in this chapter are advanced, we are not providing full code implementations; instead, we provide only a basic structural snippet on how to start implementing these concepts.

Neuro-fuzzy

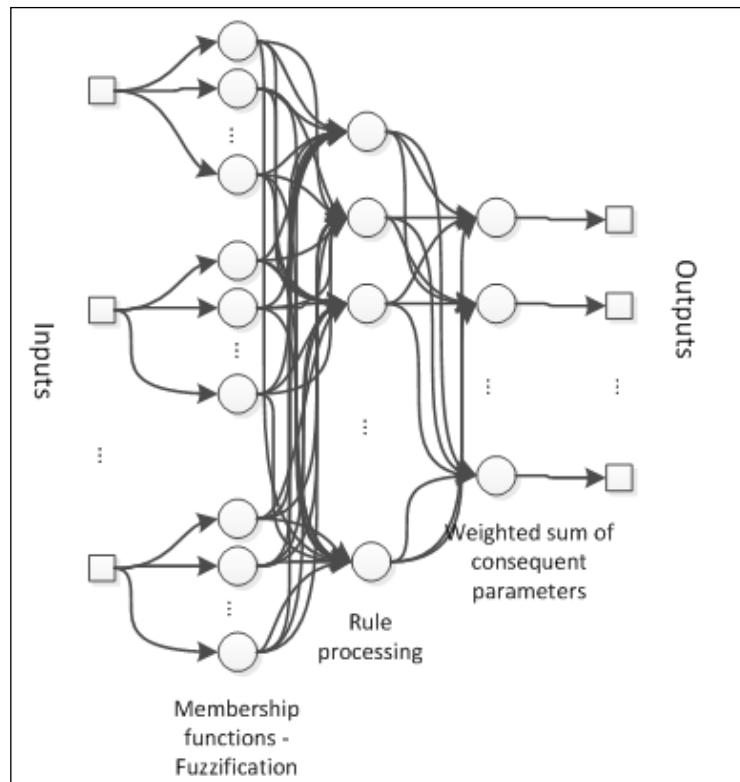
Fuzzy logic is a type of rule-based processing, where every variable is converted to a symbolic value according to a membership function, and then the combination of all variables is queried against an *IF-THEN* rule database.



A membership function usually has a Gaussian bell shape, which tells us how much a given value is a *member* of that class. Let's take, for example, temperature, which may take on three different classes (cold, normal, and warm). A membership value will be higher the more the temperature is closer to the bell shape centers.



Furthermore, the fuzzy processing finds which rules are fired by every input record and which output values are produced. A neuro-fuzzy architecture treats each input differently, so the first hidden layer has a set of neurons for each input corresponding for each membership function:



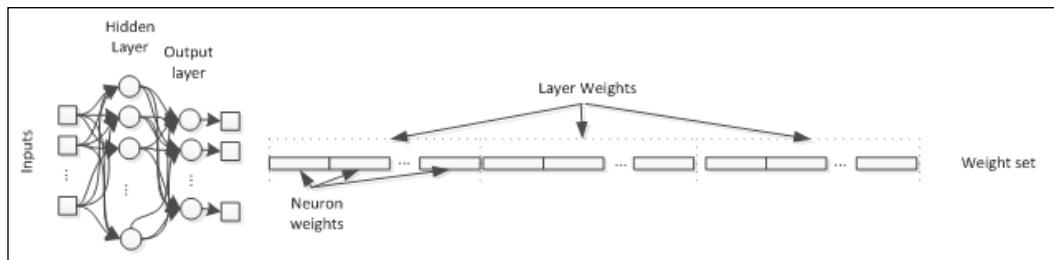
In this architecture, the training finds optimal weights for the rule processing and weighted sum of consequent parameters only, the first hidden layer has no adjustable weights.

In fuzzy logic architecture, the experts define a rule database that may become huge as the number of variables increase. The neuro-fuzzy architecture releases the designer from defining the rules, and lets this task be performed by the neural network. The training of a neuro-fuzzy can be performed by gradient type algorithms such as backpropagation or matrix algebra such as least squares, both in the supervised mode. Neuro-fuzzy systems are suitable for control of dynamic systems and diagnostics.

Neuro-genetic

In the evolutionary artificial intelligence approach, one common strategy is genetic algorithms. This name is inspired by natural evolution, which states that beings more adapted to the environment are able to produce new generations of better adapted beings. In the computing intelligence field, the *beings* or *individuals* are candidate solutions or hypotheses that can solve an optimization problem. Supervised neural networks are used for optimization, since there is an error measure that we want to minimize by adjusting the neural weights. While the training algorithms are able to find better weights by gradient methods, they often fall in local minima. Although some mechanisms, such as regularization and momentum, may improve the results, once the weights fall in a local minimum, it is very unlikely that a better weight will be found, and in this context genetic algorithms are very good at it.

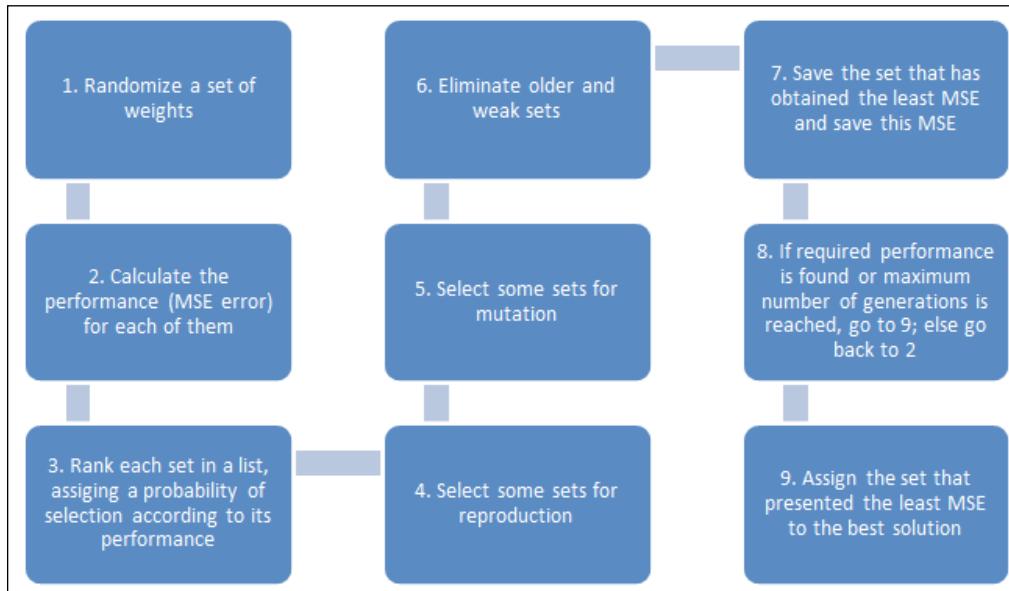
Think of the neural weights as a genetic code (or DNA). If we could generate a finite number of random generated weight sets, and evaluate which produce the best results (smaller errors or other performance measurement), we would select a top N best weight, and then set and apply genetic operations on them, such as reproduction (interchange of weights) and mutation (random change of weights).



This process is repeated until some acceptable solution is found.

Another strategy is to use genetic operations on neural network parameters, such as number of neurons, learning rate, activation functions, and so on. Considering that, there is always a need to adjust parameters or train multiple times to ensure we've found a good solution. So, one may code all parameters in a genetic code (parameter set) and generate multiple neural networks for each parameter set.

The scheme of a genetic algorithm is shown in the following figure:



Genetic algorithms are broadly used for many optimization problems, but in this book we are sticking with these two classes of problems, weight and parameter optimization.

Implementing a hybrid neural network

Now, let's implement a simple code that can be used in the neuro-fuzzy and neuro-genetic networks. First, we need to define Gaussian functions for activation that will be the membership functions:

```
public class Gaussian implements ActivationFunction{
    double A=1.0,B=0.0,C=1.0;
    public Gaussian(double A){ //...
    }
    public double calc(double x){
        return this.A*Math.exp(-Math.pow(x-this.B,2.0) / 2*Math.
        pow(this.C,2.0));
    }
}
```

The fuzzy sets and rules need to be represented in a way that a neural network can understand and drive the execution. This representation includes the quantity of sets per input, therefore having the information on how the neurons are connected; and the membership functions for each set. A simple way to represent the quantity is an array. The array of sets just indicates how many sets there are for each variable; and the array of rules is a matrix, where each row represents a rule and each column represents a variable; each set can be assigned a numerical integer value for reference in the rule array. An example of three variables, each having three sets, is defined in the following snippet, along with the rules:

```
int[] setsPerVariable = {3,3,3};  
int[][] rules = {{0,0,0},{0,1,0},{1,0,1},{1,1,0},{2,0,2},{2,1,1},  
{2,2,2}};
```

The membership functions can be referenced in a serialized array:

```
ActivationFunction[] fuzzyMembership = {new Gausian(1.0),//...  
};
```

We need also to create classes for the layers of a neuro fuzzy architecture, such as `InputFuzzyLayer` and `RuleLayer`. They can be children of a `NeuroFuzzyLayer` superclass, which can inherit from `NeuralLayer`. These classes are necessary because they work differently from the already defined neural layer:

```
public class NeuroFuzzyLayer extends NeuralLayer{  
    double[] inputs;  
    ArrayList<Neuron> neurons;  
    Double[] outputs;  
    NeuroFuzzyLayer previousLayer;  
    //...  
}  
  
public class InputFuzzyLayer extends NeuroFuzzyLayer{  
  
    int[] setsPerVariable;  
    ActivationFunction[] fuzzyMembership;  
    //...  
}  
  
public class RuleLayer extends NeuroFuzzyLayer{  
    int[][] rules;  
    //...  
}
```

A NeuroFuzzy class will inherit from NeuralNet, having references to the other fuzzy layer classes. The calc() methods of the NeuroFuzzyLayer will also be different, taking into account the membership functions centers:

```
public class NeuroFuzzy extends NeuralNet{
    InputFuzzyLayer inputLayer;
    RuleLayer ruleLayer;
    NeuroFuzzyLayer outputLayer;
    //...
}
```

For more details, see the files in the `edu.packt.neuralnet.neurofuzzy` package.

To code a neuro-genetic for weight sets, one needs to define the genetic operations. Let's create a class called `NeuroGenetic` to implement reproduction and mutation:

```
public class NeuroGenetic{
    // each element ArrayList<double> is a solution, i.e.
    // a set of weights
    ArrayList<ArrayList<double>> population;
    ArrayList<double> score;

    NeuralNet neuralNet;
    NeuralDataSet trainingDataSet;
    NeuralDataSet testDataSet;

    public ArrayList<ArrayList<double>> reproduction(ArrayList<ArrayList<double>> solutions) {
        // a set of weights is passed as an argument
        // the weights are just swapped between them in groups of two
    }

    public ArrayList<ArrayList<double>> mutation(ArrayList<ArrayList<double>> solutions) {
        // a random weight can suddenly change its value
    }
}
```

The next step is to define the evaluation of each weight on each iteration:

```
public double evaluation(ArrayList<double> solution){  
    neuralNet.setAllWeights(solution);  
    LearningAlgorithm la = new LearningAlgorithm(neuralNet,trainingData  
Set);  
    la.forward();  
    return la.getOverallGeneralError();  
}
```

Finally, we can just call a neuro-genetic algorithm by using the following code:

```
public void run{  
    generatePopulation();  
    int generation=0;  
    while(generation<MaxGenerations && bestMSError>MinMSError){  
        //evaluate all  
        foreach(ArrayList<double> solution:population){  
            score.set(i,evaluation(solution));  
        }  
        //make a rank  
        int[] rank = rankAll(score);  
        //check the best MSE  
        if(ArrayOperations.min(score)<bestMSError){  
            bestMSError = ArrayOperations.min(score);  
            bestSolution = population.get(ArrayOperations.indexMin(score));  
        }  
        //perform a selection for reproduction  
        ArrayList<ArrayList<double>> newSolutions = reproduction(  
            selectionForReproduction(rank,score,population));  
        //perform selection for mutation  
        ArrayList<ArrayList<double>> mutated = mutation(selectionForMutati  
on(rank,score,population));  
        //perform selection for elimination  
        if(generation>5)  
            eliminateWorst(rank,score,population);  
        //add the new elements  
        population.append(newSolutions);  
        population.append(mutated);  
    }  
    System.out.println("Best MSE found:"+bestMSError);  
}
```

Summary

In this final chapter, we gave the reader a glimpse of what to do next in this field. Being more theoretical, this chapter has focused more on the functionality and information than on practical implementation, because this would be very heavy for an introductory book. In every case, a simple code is provided to give a hint on how to further implement deep neural networks. The reader is then encouraged to modify the codes of the previous chapters, adapting them to the hybrid neural networks and comparing the results. Being a very dynamic and novel field of research, at every moment new approaches and algorithms are under development, and we provide in the references a list of publications to stay up to date on this subject.

References

Here are some references for the reader to check if wanting to know more about a specific topic covered in this book.

Chapter 1: Getting Started with Neural Networks

Priddy, Kevin L., Keller, Paul. E., Artificial Neural Networks: An introduction, SPIE Press, ISBN-13 9780819459879, Jan, 1, 2005.

Levenick, J., Simply Java: An introduction to Java Programming, Charles River Media; 1 ed., ISBN-13 9781584504269, Sep, 8, 2005.

Chapter 2: Getting Neural Networks to Learn

Sejnowski, Terrence, J., Neural Network Learning Algorithms, Neural Computers Vol. 41, Springer Study Edition, pp. 291-300, 1989.

Hguyen, Derrick H., Widrow, Bernard, Neural Networks for Self-Learning Control Systems, IEEE Control Systems Magazine, April 1990.

Chapter 3: Perceptrons and Supervised Learning

Haykin, Simon O., Neural Networks and Learning Machines, Prentice Hall, 3rd ed., ISBN-13 9780131471399, Nov, 28, 2008.

Rumelhart, David E., Hinton, Geoffrey E., Williams, Ronald J., "Learning Representations by back-propagating errors", Nature v. 323 (6088), pp. 533-536, Oct, 8, 1986.

Levenberg K., A Method for the Solution of Certain Non-Linear Problems in Least Squares, Quarterly of Applied Mathematics, vol 2, pp. 164-168, 1944.

Marquardt, D., An Algorithm for Least-Squares Estimation of Nonlinear Parameters, SIAM Journal on Applied Mathematics, vol 11 (2), pp. 431-441, 1963.

Huang, Guang B., Zhu, Qin Y., Siew, Chee K., Extreme learning machine: A new learning scheme of feedforward neural networks, Proceedings of IEEE International Joint Conference on Neural Networks, 2004.

Chapter 4: Self-Organizing Maps

Duda, Richard O, Hart, Peter E., Stork, David G., Unsupervised Learning and Clustering, Pattern Classification 2nd ed., Wiley, ISBN-10 0471056693, 2001.

Van Hulle, Marc M., Self Organizing Maps, Handbook of Natural Computing, pp. 585-622, ISBN-13 978-3-540-92910-9, 2012.

Rummelhart, David E., Zipser David, Feature discovery by competitive learning, Cognitive science 9.1, pp. 75-112, 1985.

Kohonen, Teuvo, Self-Organized Formation of Topologically Correct Feature Maps, Biological Cybernetics, v. 43 (1), pp. 59-69, 1982.

Chapter 5: Forecasting Weather

Dowdy, S., Wearden S., Statistics for Research, Wiley, ISBN-10 0471086029 pp. 230, 1983.

Pearl, Judea, Causality: Models, Reasoning, and Inference, Cambridge University Press, ISBN-10 0521773628, 2000.

Appendix

Fortuna, Luigi, Graziani, Salvatore, Rizzo, Alessandro, Xibilia, Maria G., Soft Sensors for Monitoring and Control of Industrial Processes, Springer Advances in Industrial Control, ISBN-13 9781846284793, 2007.

Cohen, Jacob, Cohen, Patricia, West, Stephen G., Aiken, Leona S., Applied Multiple Regression/Correlation Analysis for the behavioral sciences, Routledge, ISBN 9781134800940, 2013.

Spatz, Chris, Basic Statistics: Tales of Distributions, Cengage Learning, ISBN 9780495383932, 2007.

Chapter 6: Classifying Disease Diagnosis

Altman, Edward I., Marco, Giancarlo, Varetto, Franco, Corporate distress diagnosis: Comparison using linear discriminant analysis and neural networks (the Italian experience), Journal of Banking and Finance v. 18, pp. 505-529, 1994.

Bishop, C.M. Neural Networks for Pattern Recognition, Oxford University Press, ISBN-10 0198538499, 1995.

Al-Shayea, Qeethara K., Artificial Neural Networks in Medical Diagnosis, International Journal of Computer Science Issues, Vol. 8, Issue 2, pp. 150-154, March 2011.

Freedman, David A., Statistical Models: Theory and Practice, Cambridge University Press, 2009.

Fawcett, Tom, An Introduction to ROC Analysis, Pattern Recognition Letters, vol. 27, is. 8 pp. 861-874, 2006.

Chapter 7: Clustering Customer Profiles

Du, K.L., Clustering: A Neural Network Approach, Neural Networks, Vol. 23, Is. 1, pp. 89-107, January 2010.

Park, J, Sandberg, I.W., Universal Approximation using Radial-Basis-Function Networks, Neural Computation, vol. 3 is. 2, pp. 246-257, 1991.

Wall, Michael E., Rechtsteiner Andreas, Rocha, Luis M., Singular value decomposition and principal component analysis, A Practical Approach to Microarray Data Analysis, pp. 91-109, 2003.

References

- Cross Glendon, Thompson Wayne, Understanding your Customer: Segmentation Techniques for Gaining Customer Insight and Predicting Risk in the Telecom Industry, SAS Global Forum, 2008.
- Bozdogan, Hamparsun, Akaike's Information Criterion and Recent Developments in Information Complexity, Journal of Mathematical Psychology, Vol. 44, Is. 1, pp. 62-91, March 2000.

Chapter 8: Text Recognition

Basu, Jayanta K., Bhattacharyya Debnath, Kim, Tai-hoon, Use of Artificial Neural Network in Pattern Recognition, International Journal of Software Engineering and Its Applications, Vol. 4, No. 2, April 2010.

Shrivastava, Vivek, Sharma, Navdeep, Artificial Neural Network Based Optical Character Recognition, Signal and Image Processing: An International Journal (SIPJ), Vol. 3, No. 5, October 2012.

Chapter 9: Optimizing and Adapting Neural Networks

Utrans, J, Moody J., Rehfuss, S., Siegelmann, H., Input variable selection for neural networks: application to predicting the U.S. business cycle, Computational Intelligence for Financial Engineering, Proceedings of the IEEE/IAFE 1995.

Saxén, H., Pettersson, F., Method for the selection of inputs and structure of feedforwaed neural networks, Computers and Chemical Enginnering, Vol. 30, Is. 6-7, pp. 1048-1045, 15May 2006.

Souza, Alan M.F., Affonso, Carolina M., Soares, Fábio M., De Oliveira, Roberto C.L., Soft Sensor for Fluoridated Alumina Inference in Gas Treatment Centers, Intelligent Data Engineering and Automated Learning 2012, Lecture Notes in Computer Science v. 7435, pp. 294-302, Springer Verlab Berlin Heidelberg, 2012.

Jollife, I.T. Principal Component Analysis, 2nd ed. Springer Wiley, 2002.

Karmin E.D., A simple procedure for pruning back-propagation trained neural networks, IEEE transactions on Neural Networks, pp. 239-242, June 1990

Gill, P.E., Murray, W. Wright, M.H., Practical Optimization, Academic Press: London, 1981.

Carpenter, Gail A., Grossberg, Stephen, Adaptive Resonance Theory, The Handbook of Brain Theory and Neural Networks, 2nd ed., pp. 1-11, 2002.

Zhang, Guoqiang, Hu, Michael Y., Patuwo, Eddy B., Indro, Daniel C., Artificial neural networks in bankruptcy prediction: General framework and cross-validation analysis, European Journal of Operational Research, vol. 116, Is. 1, pp. 16-32, July 1999.

Chapter 10: Current Trends in Neural Networks

Schmidhuber, Juergen, Deep learning in neural networks: An overview, *Neural Networks*, Elsevier, Vol. 61, pp. 85-117, January 2015.

Krizhevsky, Alex, Sutskever, Ilya, Hinton, Geoffrey, ImageNet Classification with Deep Convolutional Neural Networks, *Advances in Neural Information Processing Systems*, pp. 85-117, 2012.

Nauck, Detlef, Klawonn, Frank, Kruse, Rudolf, Foundations of neuro-fuzzy systems, John Wiley and Sons, ISBN 047197150, 1997.

Montana, David J., Davis, Lawrence, Training Feedforward Neural Network Using Genetic Algorithms, *IJCAI*. Vol. 89, pp. 762-767, 1989.

Bibliography

This course is a blend of text and quizzes, all packaged up keeping your journey in mind. It includes content from the following Packt products:

- *Java Deep Learning Essentials, Yusuke Sugomori*
- *Machine Learning in Java, Boštjan Kaluža*
- *Neural Network Programming with Java, Second Edition, Fábio M. Soares and Alan M. F. Souza*

Index

Symbols

2D competitive layer
creating 562, 563

A

abstraction 473
A/B tests
URL 447
activation function
about 383
using 467, 468
ActivationFunction interface
defining 478
activity recognition
about 402
activity-recognition pipeline 404, 405
mobile phone sensors 402, 403
plan 405
AdaBoost M1 method 292
Adaline 503, 504
adaptive neural networks
about 682
adaptive resonance theory (ART) 682
implementation 682, 683
adaptive resonance theory (ART) 682
advanced modeling
attribute selection 316
data, pre-processing 315
ensembleLibrary package, using 314
model selection 317-320
performance, evaluation 321
with ensembles 313
affinity analysis
about 323, 325

cross-industry applications 333
agglomerative clustering 252
AI transition
defining 4
Akaike Information Criteria (AIC) 650
AlphaGo 224
Amazon Machine Learning 452
analysis types
about 364
pattern analysis 364
transaction analysis 365
Android Device Monitor 413
Android Studio
installing 406, 407
URL 406
anomalous behavior detection
about 362
unknown-unknowns 362
anomalous pattern detection
about 364
analysis types 364
plan recognition 365
anomaly detection, in time series data
about 374
data, loading 376, 377
density based k-nearest neighbors 378-380
histogram-based anomaly
detection 374-376
histograms, creating 377, 378
anomaly detection, in website traffic
about 373
dataset, using 373
Apache Mahout
about 271, 272
configuring 341

configuring, in Eclipse with Maven plugin 342, 343

Apache Spark
about 272-274
URL 272

Application Portfolio Management (APM) 335

Applied Machine Learning
about 237
workflow 239, 240

Apriori 268

Apriori algorithm
about 323, 328
used, for discovering shopping patterns 330-332

architecture, neural networks
about 470
feedback networks 472
feedforward networks 471
monolayer networks 470
multilayer networks 471
perceptrons 518

Artificial Intelligence (AI)
about 3
and deep learning 16-24, 226-231
defining 4, 5
history 5-10

artificial neural networks (ANNs)
about 254, 463, 486
activation function, using 467, 468
artificial neuron 466
bias 468
layers 469
need for 464, 465
neural networks, arrangements 466
weights 468

artificial neuron 466

association rule learning
about 326
Apriori algorithm 328
confidence 328
database, of transactions 326, 327
FP-growth algorithm 329
itemset 327
rule 327
support 328

autoencoder 385, 386

automatic colorization
reference 230

automatic differentiation 209

B

backpropagated error 63

backpropagation algorithm
about 528-530
coding 530-534

backpropagation formula 63

Backpropagation through Time (BPTT) 189

bag-of-word (BoW) 427

basic modeling
about 311
models, evaluating 311, 312
naive Bayes baseline,
implementing 312, 313

basic naive Bayes classifier baseline
about 307
data, loading 309, 310
data, obtaining 308, 309

Bayesian Information Criteria (BIC) 650

benchmark tests
URL 222

Bernoulli RBM 81

bias 468

big data
dealing with 279
variety 279
velocity 279
volume 279

big data application
architecture 279

BigML 452

bigram 183

binary classes
versus multiple classes 624

Boltzmann Machines (BMs) 78

Book-Crossing dataset
BX-Book-Ratings file 344
BX-Books file 344
BX-Users file 344
URL 344

- book-recommendation engine**
book ratings dataset, using 344
building 344
collaborative filtering, implementing 350
content-based filtering,
 implementing 359, 360
custom rules, adding 355
data, loading 345
data, loading from database 348, 349
data, loading from file 345-347
evaluation 356, 357
in-memory database, creating 349, 350
online learning engine 357, 358
- Bot Store**
URL 229
- Brazilian Institute of Meteorology (INMET)**
URL 591
- breadth-first search (BFS)** 6
- breakdown-oriented approach, deep learning** 200, 204-207
- C**
- Caffe**
about 219, 221
URL 220
- Canova library**
URL 391
- Cassandra**
about 280
URL 280
- categorical data**
about 621
working with 622
- cc.mallet.pipe package**
CharSequenceRemoveHTML pipeline 430
Input2CharSequence pipeline 430
MakeAmpersandXMLFriendly
 pipeline 430
TokenSequence2FeatureSequence
 pipeline 431
TokenSequenceLowercase pipeline 430
TokenSequenceNGrams pipeline 431
- Chainer**
URL 222
- Chart class** 571
- Chebyshev distance** 270
- classification**
about 253, 282, 514, 515
artificial neural networks 254
classification algorithm, selecting 291, 292
confusion matrix, examining 290, 291
data, classifying 289
data, loading 284
data, using 283
decision trees learning 254
ensemble learning 255
evaluating 255, 256
evaluation 290
feature selection 285
kernel methods 254
learning algorithms, selecting 286-288
precision 256
prediction error metrics 290
probabilistic classifiers 254
recall 256
Roc curves 256, 257
with neural networks 628
- classification algorithms**
weka.classifiers.bayes.NaiveBayes 292
weka.classifiers.functions.Multilayer
 Perceptron 292
weka.classifiers.lazy.IBk 292
weka.classifiers.meta.AdaboostM1 292
weka.classifiers.meta.Bagging 292
weka.classifiers.rules.ZeroR 291
weka.classifiers.trees.RandomForest 292
weka.classifiers.trees.RandomTree 291
- classification problems**
categorical data 621
foundations 620
- classifier**
building 414, 415
plugging, into mobile app 418, 419
spurious transitions, reducing 416-418
- class implementation**
reference link 391
- class unbalance** 446
- clustering**
about 30, 252, 253, 299, 300
clustering algorithms 300, 301
evaluation 302

clustering algorithms
implementing 300, 301

clustering tasks
about 638
cluster analysis 639
cluster evaluation 640
external validation 641
implementation 640
validation 640

collaborative filtering
about 337, 339
implementing, with book-recommendation engine 350
item-based 354, 355
user-based 351, 352, 353

Comma Separated Value (CSV) 293, 585

competitive layer 554, 555

competitive learning
about 551-553
class, creating 568-571

computational differentiation 209

confusion matrix
about 624
implementing 626, 627

conjugate gradient optimization algorithm
building 392

constant error carousel (CEC) 193

content-based filtering
about 337, 340
implementing, with book-recommendation engine 359, 360

Contrastive Divergence algorithm 387

Contrastive Divergence (CD) 85

Convolutional neural networks (CNN)
about 122, 123, 387, 688, 689
convolution layers 124-127
equations 128-143
feature maps 124
implementations 128-143
kernels 124
local receptive field 125
pooling layers 127, 128
translation invariance 125

convolution layers 126

Core Motion framework, iOS
URL 403

correlation coefficient 259

cosine distance 340

cost function
about 253
calculating 491, 492
using 490

cross-industry applications, of affinity analysis
about 333
census data 334
customer relationship management (CRM) 334
IT Operations Analytics 335
medical diagnosis 333
protein sequences 333

Cross Industry Standard Process for Data Mining (CRISP-DM) 449

cross-validation 262, 673

CrowdANALYTIX
URL 455

CSVLoader class
URL 309

curse of dimensionality 251

customer profiling
credit analysis, performing 644-648

customer relationship database
about 304
challenge 304, 305
dataset 305, 306
evaluation 307

Customer Relationship Management (CRM) 303

Cyc
URL 9

D

data 240

data and problem definition
about 240
measurement scales 241, 242

data cleaning 245

data collection
about 242
Android Studio, installing 406, 407
data collector, loading 408, 409
data, generating 244

data, observing 243, 244
data, searching 243, 244
from mobile phone 406
training data, collecting 411-414
traps, sampling 245

data collector
feature extraction 410, 411
loading 408, 409
URL 408

data correlation 669

data filtering 672, 673

Data Mining
URL 455

Data Mining Research
URL 455

data pre-processing
about 245
data cleaning 245
data reduction 248, 249
data transformation 247, 248
missing values, filling 246
outliers, removing 247

data reduction 248, 249

data science 237, 238

Data Science Central
URL 455

dataset rebalancing 370-372

datasets 453, 454

data transformation 247, 248, 670

Davies-Boudin index 640

Decision and Predictive Analytics (ADAPA) 451

decision boundary 29

decision trees 261

decision trees learning 254

Decode 99

deep architectures
about 688
convolutional neural network (CNN) 688, 689
deep belief network (DBN) 690
hybrid systems 692
long short time memory (LSTM) 689

Deep Belief Nets (DBNs)
about 17, 69
defining 92-98

deep belief network (DBN)
about 254, 387, 690
building 394-396

deep convolutional networks 387-390

Deep Dream
URL 20

deep learning
abilities, maximizing 200
about 3, 223-225, 685-687
active, on fields 180
algorithms 78
and AI 16-24, 226-231
breakdown-oriented approach 204-207
Deep Belief Nets (DBNs) 92-98
defining 71
Denoising Autoencoders (DA) 98-105
field-oriented approach 201
image recognition field 180, 181
implementing, in Java 690, 691
issues 198, 199
natural language processing (NLP) 182
news sources 231-234
output-oriented approach 207, 208
possibilities, maximizing 200
references 71
Restricted Boltzmann Machines (RBM) 78-91
Stacked Denoising Autoencoders (SDA) 105, 106
with pre-training 72-77

Deeplearning4j (DL4J)
about 274, 275
org.deeplearning4j.base 274
org.deeplearning4j.berkeley 274
org.deeplearning4j.clustering 274
org.deeplearning4j.datasets 274
org.deeplearning4j.distributions 274
org.deeplearning4j.eval 274
org.deeplearning4j.exceptions 274
org.deeplearning4j.models 274
org.deeplearning4j.nn 275
org.deeplearning4j.optimize 275
org.deeplearning4j.plot 275
org.deeplearning4j.rng 275
org.deeplearning4j.util 275
URL 274

deeplearning4java
about 389
obtaining 389, 390

deep learning algorithm
URL 180
without pre-training 109, 110

deep learning group
URL 234

Deep Learning News
about 233
URL 233

DeepMind
URL 224

defined classes 654, 655

delta rule
about 383, 495, 496
implementing 497

Denoising Autoencoders (DA)
defining 98-105

denormalize 598

depth-first search (DFS) 6

digit representation 658, 659

digits recognition 658

dimensionality reduction 670, 671

directory
text data, importing 428, 429

Discrete Fourier Transform (DFT) 404

disease diagnosis
experimenting, for breast cancer 628-631
experimenting, for diabetes 632-634
with neural networks 628

distance measures
Euclidean distances 249
non-Euclidean distances 250, 251

divide-and-conquer strategy 329

DL4J
building 159
CNNMnistExample.java /
 LenetMnistExample.java 168-173
CSVExample.java 165-167
DBNIrisExample.java 159-164
implementations 156
implementing, with ND4J 148, 149
learning rate, optimization 174-176
set up 156-158
URL 148

double evaluateLeftToRight method
boolean useResampling component 437
Instances heldOutDocuments
 component 437
int numParticles component 437
PrintStream docProbabilityStream
 component 437

DrivenData 455

DropConnect neural network 390

dropout 21, 22

dropout algorithm
about 110-122
Rectified Linear Unit (ReLU) 113
rectifier 113
softplus function 114

DSGuide
URL 455

Dunn index 640

dynamic time wrapping (DTW) 270

E

Eclipse
Apache Mahout, configuring with Maven
plugin 342, 343

Eclipse IDE
using 282

Edit distance 251

elbow method 300

email spam dataset
URL 440

email spam detection
about 439
default pipeline, creating 441, 442
email spam dataset, collecting 440, 441
model performance, evaluating 443
testing 442
training 442

empirical design, neural networks
experiments, designing 613, 614
results 614-617
simulations 614-617

encapsulation 474

Encode 99

enrolment status prediction
case study 545- 548

ensambleSel.setOptions () method
 -A <algorithm> option 320
 -B <numModelBags> option 319
 -D option 320
 -E <modelRatio> option 319
 -G option 320
 -H <hillClimbIterations> option 319
 -I <sortInitialization> option 319
 -L </path/to/modellLibrary> option 319
 -O option 320
 -P <hillclimbMettric> option 319
 -R option 320
 -S <num> option 320
 -V <validationRatio> option 319
 -W </path/to/working/directory> option 319
 -X <numFolds> option 319

ensemble learning 255

ensembleLibrary package
 URL 314
 using 314

ensembles
 used, for advanced modeling 313

Ensemble Selection algorithm 313

environmental sensors 403

Euclidean distances 249

evaluate() method, parameters
 DataModel 357
 DataModelBuilder 357
 evaluationPercentage 357
 RecommenderBuilder 357
 trainingPercentage 357

evaluation 260

Expectation Maximization (EM)
 clustering 299

exploitation 341

exploration 341

external validation 641

extreme learning machines (ELMs) 539-541

F

feature extraction 278

feature map 124, 387

feature selection 248

feedback networks 472

feedforward networks 471

feed-forward neural networks 182-187, 384

field-oriented approach, deep learning
 about 200
 advert technologies 203
 automobiles 202, 203
 medicine 201
 profession or practice 203
 sports 204

file
 text data, importing 429, 430

fine-tuning 72

forget gate 195

Fourier transform
 reference 404

FP-growth algorithm
 about 268, 323, 329
 used, for discovering shopping patterns 332

FP-tree structure 329

frame problem 7

fraud detection, of insurance claims
 about 365
 dataset, using 366, 367
 suspicious patterns, modeling 368

frequent pattern (FP) 329

G

generalization
 about 260
 cross-validation 262
 leave-one-out validation 262
 overfitting 260, 261
 stratification 263
 test set 262
 train set 262
 underfitting 260, 261

Generalized Sequential Patterns (GSP) 268

Generative Stochastic Networks (GSNs) 386

Gibbs sampling 387

GitXiv
 about 232
 URL 232

GNU General Public License (GNU GPL) 267

Google Prediction API 452

gradient method 491

Graphical User Interface (GUI) 614

Graphics Processing Unit (GPU)

about 399

reference link 399

GraphX 272

H

Hacker News

about 233

URL 233

Hadoop

about 280

URL 280

Hadoop Distributed File System (HDFS) 273

Hamming distance 251

HBase

about 280

URL 280

Hebbian learning 501

Hidden layer 384

Hidden layer, issues

overfitting 384

vanishing gradients problem 384

hidden layers 469

Hidden Markov Model

(HMM) 33, 272, 365, 689

hidden units 80

hierarchical clustering 252

histogram-based anomaly detection 374-376

Hopfield network 80

Hotspot 268

hybrid approach 340

hybrid neural network

implementing 696-698

hybrid systems

about 692

neuro-fuzzy 693, 694

neuro-genetic 695, 696

I

IBM Watson Analytics 452

image classification

about 389

data, loading 390, 391

deeplearning4java 389

MNIST dataset 390

models, building 391

ImageNet

about 382

URL 388

Imagenet Large Scale Visual Recognition

Challenge (ILSVRC) 18

image recognition

about 180, 181, 381, 382

neural networks 383

Inceptionism

about 20

URL 20

Infrastructure as a Service (IaaS) 451

inheritance 474

input gate 194

Input layer 384

input method editor (IME) 12

input selection

about 669

cross-validation 673

data correlation 669

data filtering 672, 673

data transformation 670

dimensionality reduction 670, 671

structure selection 675, 676

insurance claims

fraud detection 365

interval data 241

Intrusion Detection (ID) 365

item-based analysis 339

item-based collaborative filtering 354, 355

IT Operations Analytics 323

J

Jaccard distance 250

Java

deep learning, implementing 690, 691

need for 266

pattern recognition, implementing 659

profiling, implementing 644

used, for implementing neural

networks 473-475

Java API packages, Weka

weka.associations 268

weka.classifiers 268

- weka.clusterers 269
- weka.core 269
- weka.datagenerators 269
- weka.estimators 269
- weka.experiment 269
- weka.filters 269
- weka.gui 269
- Java machine learning (Java-ML)**
 - about 270
 - URL 270
- Java-ML packages**
 - net.sf.javaml.classification 270
 - net.sf.javaml.clustering 270
 - net.sf.javaml.core 270
 - net.sf.javaml.distance 270
 - net.sf.javaml.featureselection 271
 - net.sf.javaml.filter 271
 - net.sf.javaml.matrix 271
 - net.sf.javaml.sampling 271
 - net.sf.javaml.tools 271
 - net.sf.javaml.utils 271
- java -Xmx16g 321**
- JFreeChart package**
 - URL 571
- K**
 - Kaggle 455**
 - KDD Cup**
 - about 304
 - URL 304
 - KDnuggets**
 - about 452
 - URL 455
 - kernel methods 254**
 - kernels 124**
 - k-fold cross-validation 41**
 - k-means clustering 252**
 - k-nearest neighbors 261**
 - knowledge base 9**
 - Knowledge Discovery and Data Science (KDD) 303**
 - Knowledge Representation (KR) 8**
 - known-knowns 362**
 - known-unknowns 362**
 - Kohonen neural network 642**
- Kohonen self-organizing maps (SOMs)**
 - 2D competitive layer, creating 562, 563
 - 2D training datasets, plotting 574, 575
 - about 555, 556
 - class, creating for competitive learning 568-571
 - learning rate 568
 - neighborhood function, using 566, 567
 - neural network code, extending 556, 557
 - neuron weights, plotting 574, 575
 - one-dimensional SOM, defining 558, 559
 - SOM learning algorithm 564, 565
 - testing 575-581
 - two-Dimensional SOM, defining 560-562
 - visualizing 571-574
 - zero-dimensional SOM, defining 557
- L**
 - Latent Dirichlet Allocation (LDA) 422, 434**
 - layers 469**
 - layer-wise training 71**
 - learning ability**
 - for solving problems 486
 - illustrative example 504
 - in neural networks 486
 - testing 509, 510
 - training dataset, using 505-508
 - learning algorithms**
 - Adaline 503, 504
 - calcNewWeight method,
 - implementing 498-500
 - delta rule 495, 496
 - delta rule, implementing 497
 - examples 493, 495
 - Hebbian learning 501
 - learning rate 496
 - train method, implementing 498-500
 - learning paradigms**
 - about 487
 - supervised learning 487
 - unsupervised learning 488
 - learning process**
 - about 472, 473, 489
 - cost function, calculating 491, 492
 - cost function, using 490

general error 492, 493
overall error 492, 493
terminating 493
weights, updating 491
learning rate 496
leave-one-out validation 262
Levenberg-Marquardt algorithm
 about 534-536
 coding, with matrix algebra 536-539
 error backpropagation 537
 error LMA 537
 Jacobian matrix 537
library/framework
 versus scratch implementations 146-148
Linear Discriminant Analysis (LDA)
 about 434
 reference link 437
linear regression 295, 296
linear separation 519, 520
Local Outlier Factor (LOF) 376
LOF algorithm
 URL 378
logistic regression
 about 622, 623
 confusion matrix 624
 confusion matrix, implementing 626, 627
 defining 35, 50-53
 multiple classes, versus binary classes 624
 sensitivity 625, 626
 specificity 625, 626
long short term memory (LSTM)
 network 193-198, 689
LSTM block 195
LSTM memory block 195

M

machine
 and human, comparing 15, 16
machine learning
 about 237, 238
 advantages 238, 239
 application flow 36-41
 as service 452
 class unbalance 446
 defining 10-12
 drawbacks 13, 14

evaluation 447
feature selection 446
in cloud 451
in real life 445
model chaining 447
models, in production 448
models, maintaining 448, 449
need for training 26-29
noisy data 446
reinforcement learning 239
supervised learning 238
unsupervised learning 239
machine learning application
 big data, dealing with 279
 building 278
 traditional machine learning 278
Machine Learning for Language Toolkit (MALLET)
 about 275, 276
 URL 275
machine learning libraries
 about 266
 Apache Mahout 271, 272
 Apache Spark 272-274
 comparing 277
 Deeplearning4j 274, 275
 Java machine learning (Java-ML) 270
 Machine Learning for Language Toolkit (MALLET) 275, 276
 Waikato Environment for Knowledge Analysis (Weka) 266-269
Machine learning mastery
 URL 455
Mahalanobis distance 251, 270
Mahout interfaces, abstractions
 DataModel 350
 ItemSimilarity 350
 Recommender 350
 UserNeighborhood 350
 UserSimilarity 350
Mahout libraries
 org.apache.mahout.cf.taste 272
 org.apache.mahout.classifier 272
 org.apache.mahout.clustering 272
 org.apache.mahout.common 272
 org.apache.mahout.ep 272

org.apache.mahout.math 272
org.apache.mahout.vectorizer 272

Mallet
installing 424-426
URL 424

MALLET, packages
cc.mallet.classify 276
cc.mallet.cluster 276
cc.mallet.extract 276
cc.mallet.fst 276
cc.mallet.grmm 276
cc.mallet.optimize 276
cc.mallet.pipe 276
cc.mallet.topics 276
cc.mallet.types 276
cc.mallet.util 276

Manhattan distance 270

market basket analysis (MBA)
about 323, 324, 325
affinity analysis 325
identification, of driver items 324
item affinity 324
marketing 325
operations optimization 325
revenue optimization 325
store-to-store comparison 325
trip classification 324

Markov chain 387

Markov model 184

Markov process 33

matrix algebra
Levenberg-Marquardt algorithm,
coding 536-539

Maven plugin
Apache Mahout, configuring with 342-344

maximizing the margin 30

maximum likelihood estimation (MLE) 185

mean absolute error 259

mean squared error (MSE) 259, 492

measurement scales
about 241, 242
interval data 241
nominal data 241
ordinal data 241
ratio data 242

Microsoft Azure Machine Learning 452

Microsoft Excel® 590

mini-batch 53

mini-batch stochastic gradient descent (MSGD) 53

Minkowski distance 270

missing values
filling 246

MLLib API library
org.apache.spark.mllib.classification 273
org.apache.spark.mllib.clustering 273
org.apache.spark.mllib.linalg 273
org.apache.spark.mllib.optimization 273
org.apache.spark.mllib.recommendation 273
org.apache.spark.mllib.regression 274
org.apache.spark.mllib.stat 274
org.apache.spark.mllib.tree 274
org.apache.spark.mllib.util 274

MLP implementation
URL 211

MNIST classifications
URL 215

MNIST database 17

MNIST dataset 390

mobile app
classifier, plugging into 418, 419

mobile phone
data, collecting 406

mobile phone sensors
about 402
environmental sensors 403
motion sensors 403
position sensors 403
URL, for Android 403
URL, for Windows Phone 403

models
building 391
chaining 447
deep belief network, building 394-396
in production 448
maintenance 448, 449
Multilayer Convolutional Network,
building 396-399
single layer regression model,
building 392-394

momentum 530

momentum coefficient 174
MongoDB
 about 280
 URL 280
monolayer networks 470
motion sensors 403
Mozilla Thunderbird 439
multi-class logistic regression
 defining 53-58
Multilayer Convolutional Network
 about 391
 building 396-399
multilayer networks 469-471
multi-layer neural networks (MLP) 60
multi-layer perceptrons
 about 522, 523
 backpropagation algorithm 528-530
 backpropagation algorithm, coding 530-534
 coding 526
 defining 59-67
 extreme learning machines (ELMs) 539-541
 learning process 527, 528
 Levenberg-Marquardt algorithm 534-536
 momentum 530
 properties 523, 524
 recurrent MLP 525
 weights 524, 525
Multilayer Perceptrons (MLP) 692
multiple classes
 versus binary classes 624
myrunscollector package
 CollectorActivity.java class 409
 Globals.java class 409
 SensorsService.java class 409

N

naive Bayes 261
naive Bayes baseline
 implementing 312, 313
NaNs 591
natural language processing (NLP)
 about 33, 182
 deep learning for 188
 feed-forward neural networks 182-187
 long short term memory networks 193
 recurrent neural networks 188-193
N-Dimensional Arrays for Java (ND4J)
 implementations 150-154
 URL 149, 152
neighborhood function
 using 566, 567
Nervana
 URL 222
Nesterov's Accelerated Gradient Descent 176
NeuralLayer class
 defining 477, 478
neural network class
 defining 479-481
Neural Network Language Model (NLMM)
 URL 186
neural networks
 about 261, 383
 architecture 470
 autoencoder 385, 386
 coding 481, 482
 common issues, with implementations 668
 deep convolutional networks 387, 388
 defining 34
 discovering 463
 empirical design 613
 feedforward neural networks 384
 for classification 628
 for regression problems 583-585
 for unsupervised learning 549, 550
 implementing, with Java 473-475
 in pattern recognition 656
 learning ability 486
 logistic regression 50-53
 multi-class logistic regression 53-58
 multi-layer perceptrons 59-67
 perceptron 383
 perceptron algorithm 42-50
 problems 69, 70
 Restricted Boltzman machine 386, 387
 theories and algorithms 42
 used, for disease diagnosis 628
 with unsupervised learning 642

neural networks, classes
HiddenLayer 474
InputLayer 474
InputNeuron 474
NeuralLayer 474
NeuralNet 474
Neuron 474
OutputLayer 474
neural-storyteller
URL 231
neuro-fuzzy 693, 694
neuro-genetic 695, 696
neuron class
defining 475-477
N-gram 183
No Free Lunch Theorem (NFLT) 181
nominal data 241
non-Euclidean distance 250, 251
normalization 595-597

O

object-oriented programming (OOP) 473
one-dimensional SOM
defining 558, 559
online courses 454
online learning engine 357, 358
online retraining
about 677
application 679, 681
batch learning 677
implementation 679
incremental learning 677
stochastic online learning 678
Oracle Database Online Documentation
URL 366
ordinal data 241
outliers
removing 247
output gate 194
Output layer 384
output-oriented approach, deep learning 200, 207, 208
overfits 240
overfitting 260, 261, 510, 511

overfitting problem 41
overtraining 509-511

P

paper, deep belief nets (DBN)
URL 17
PAPI
URL 452
part-of-speech (POS) 427
pattern analysis 364
pattern recognition
about 654
data, generating 659
data, pre-processing 657
defined classes 654, 655
digits, recognizing 658
digits, representing 658, 659
experimenting 660, 661
implementing, in Java 659
neural architecture 660
results 662-666
text, recognizing 658
undefined classes 656
with neural networks 656
Pearson coefficient 340
Pearson correlation coefficient 270
peephole connections 196
perceptron algorithm 42-50
perceptrons
about 42, 254, 382, 383, 518
applications 518
limitations 518
linear separation 519, 520
XOR case 520, 521, 522
plan recognition 365
p-norm distance 249
polymorphism 474
pooling layers 127, 128
Portable Format for Analytics (PFA) 451
position sensors 403
precision 256
Prediction.IO 452
predictive apriori 268

Predictive Model Markup Language (PMML) 451
pre-processing phase 278
pre-training 72
Principal Component Analysis (PCA) 248, 254, 375, 670
probabilistic classifiers 254
probabilistic statistical model 11
profiling
 about 643
 cluster count, obtaining 650
 credit analysis, performing for customer profiling 644-648
 implementing, in Java 644
 pre-processing 643
 product, profiling 649, 650
protocol file 221
pseudo-algorithm
 URL 678
Pylearn2
 URL 222

R

Radial Basis Function (RBF) 692
ratio data 242
recall 256
Receiver Operating Characteristics (ROC) 257
recommendation engine
 basic concepts 337, 338
 book-recommendation engine,
 building 344
 exploitation 341
 exploration 341
 item-based analysis 339
 key concepts 338, 339
 similarity, calculating 339
 user-based analysis 339
Rectified Linear Unit (ReLU) 113
recurrent MLP 525
recurrent neural network language model (RNNLM)
 URL 190

recurrent neural network (RNN) 188
regression
 about 257, 261, 292, 516, 517
 attributes, analyzing 294
 correlation coefficient 259
 data, loading 292-294
 evaluating 258
 linear regression 258
 mean absolute error 259
 mean squared error 259
 regression model, building 295
 regression model, evaluating 295
 tips 299
regression model
 building 295
 evaluating 295
 linear regression 295, 296
 regression trees 296, 298
regression problems
 solving, with neural networks 583-585
regression trees 296-298
reinforcement learning
 about 239
 defining 35, 36
Resilient Distributed Dataset (RDD) 273
Restricted Boltzmann Machines (RBM) 78-91, 254, 274, 386, 387, 690
RMSProp 176
RMSProp + momentum 176
Roc curves 256, 257
RuleSetModel 451

S

Sample, Explore, Modify, Model, and Assess (SEMMA) 450
Scale Invariant Feature Transform (SIFT) 382
score function 253
scratch implementations
 versus library/framework 146-148
Self-Organizing Maps (SOM) 692
signe 15
similar items
 searching 249

similarity calculation
about 339
collaborative filtering 339
content-based filtering 340
hybrid approach 340
SimRank 251
single layer regression model
building 392-394
Singular value decomposition (SVD) 248
Skymind
URL 148
softplus function 114
SOM learning algorithm 564, 565
Spark Streaming 272
spatio-temporal patterns 365
Spearman's footrule distance 270
stacked autoencoders 386
Stacked Denoising Autoencoders (SDA)
about 69
defining 105, 106
standards and markup languages 449
stochastic gradient descent (SGD) 53
stochastic online learning 678
stratification 263
strong AI 5
structure selection 675, 676
sum transfer function 383
supermarket dataset
about 330
shopping patterns, discovering 330
shopping patterns, discovering with
Apriori algorithm 330-332
shopping patterns, discovering with
FP-growth algorithm 332
supervised learning
about 29, 238, 253, 473, 487, 514
classification 253, 514, 515
regression 257, 516, 517
Support Vector Machines (SVM) 30-32, 254, 451
support vectors 30
survivorship bias 245
suspicious behavior detection 362
suspicious pattern detection 363
suspicious patterns, modeling
about 368
dataset rebalancing 370-372
vanilla approach 369, 370
symbol content 15
symbol grounding problem 10
symbol representation 15

T

target variables
appetency probability 305
churn probability 304
upselling probability 305
Tay
URL 227
Technical Singularity 23
TensorFlow
about 214-219
URL 148, 214
Tertius 268
testing
of learning ability 509, 510
overfitting 510, 511
overtraining 510, 511
test set 262
text classification
about 423
examples 423
text data
extracting 426-428
importing 428
importing, from directory 428
importing, from file 429, 430
pre-processing 430-432
text mining
about 421, 422
text classification 423
topic modeling 422, 423
text recognition 658
Theano
about 209-213
deep learning algorithms, URL 211
URL 209

time series data
anomaly detection 374
topic modeling 422, 423
topic modeling, for BBC news
about 432
BBC dataset, collecting 432, 433
model, evaluating 436, 437
modeling 433, 434, 435, 436
model, restoring 439
model, reusing 438
model, saving 438
traditional machine learning
architecture 278
training 509
training data
about 278
collecting 411-414
train set 262
transaction analysis 365
TreeModel 451
trigram 183
truncated BPTT 191
two-dimensional SOM
defining 560-562

U

UCI machine learning repository
URL 453
Udemy
URL 454
undefined classes 656
underfits 240
underfitting 260, 261
unigram 183
Universal PMML Plug-in (UPPI) 451
unknown-unknowns 362, 363
unsupervised learning
about 29, 239, 249, 488
clustering 252, 253
Hidden Markov Model (HMM) 33
implementing, with neural networks 642
Kohonen neural network 642
similar items, searching 249
with neural networks 549, 550

unsupervised learning algorithms
about 550
competitive layer 554, 555
competitive learning 551-553
user-based analysis 339
user-based collaborative filtering 351-353

V

vanilla approach 369, 370
vanishing gradient problem 70
visible layer 80
visible units 80

W

Waikato Environment for Knowledge Analysis (Weka)
about 266-269
URL 266
weak AI 5
weather database
references 592
weather forecasting
correlation analysis, performing 606-609
data, loading 585, 604-606
data, selecting 585
error, plotting 612
executing 604-606
input variable, selecting 592-594
Java implementation 601
learning algorithm, adapting for
normalization 600
neural network output, viewing 612
neural networks, creating 610
neural network, training 611
normalization, handling with
 NeuralDataSet 599, 600
normalization, implementing 595-598
output variable, selecting 592-594
preprocessing 594
testing 610
training 610
variables, delaying 604
weather data, collecting 601-603

weather forecasting, data selection
auxiliary classes, building 585-588
dataset, obtaining from CSV file 588
NaNs, dropping 591
time series, building 589, 590
weather data, obtaining 591
weather variables 592

web resources and competitions
about 453-455
datasets 453, 454
online courses 454
venues and conferences 456
websites and blogs 455

website traffic
anomaly detection 373

Weka 3.6
downloading 282
URL 282

weka.classifiers package
weka.classifiers.bayes 268
weka.classifiers.evaluation 268
weka.classifiers.functions 268
weka.classifiers.lazy 268
weka.classifiers.meta 269
weka.classifiers.mi 269
weka.classifiers.rules 269
weka.classifiers.trees 269

WEKA Packages
URL 314

winner-takes-all rule 551

word2vec
about 428
URL 428

workflow, Applied Machine Learning
data analysis and modeling 240
data and problem definition 239
data collection 240
data preprocessing 240
evaluation 240

X

XOR case
about 520-522
with Delta Rule and
backpropagation 542-544

Z

zero-dimensional SOM
defining 557