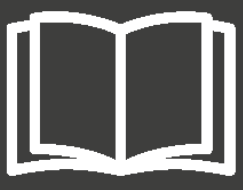
Fabio M. Soares, Alan M. F. Souza

Neural Network Programming with Java

Second Edition

Create and unleash the power of neural networks by implementing professional Java code



Packt

Neural Network Programming with Java

Second Edition

Create and unleash the power of neural networks by implementing professional Java code

Fabio M. Soares

Alan M. F. Souza

BIRMINGHAM - MUMBAI

Neural Network Programming with Java

Second Edition

First published: January 2016

Second edition: March 2017

Production reference:1080317

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

ISBN 978-1-78712-605-3

www.packtpub.com

Credits

Authors
Fabio M. Soares
Alan M. F. Souza

Reviewer
Charles Griffiths

Commissioning Editor
Vedika Naik

Acquisition Editor
Rahul Nair

Content Development Editor
Trusha Shriyan

Technical Editor
Naveenkumar Jain

Copy Editor
Safis Editing

Project Coordinator
Kinjal Bari

Proofreader
Safis Editing

Indexer
Mariammal Chettiyar

Graphics
Kirk D'Penha

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat

About the Authors

Fábio M. Soares is currently a PhD candidate at the Federal University of Pará (Universidade Federal do Pará - UFPA), in northern Brazil. He is very passionate about technology in almost all fields, and designs neural network solutions since 2004 and has applied this technique in several fields like telecommunications, industrial process control and modeling, hydroelectric power generation, financial applications, retail customer analysis and so on. His research topics cover supervised learning for data-driven modeling. As of 2017, he is currently carrying on research projects with chemical process modeling and control in the aluminum smelting and ferronickel processing industries, and has worked as a lecturer teaching subjects involving computer programming and artificial intelligence paradigms. As an active researcher, he has also a number of articles published in English language in many conferences and journals, including four book chapters.

Alan M. F. Souza is computer engineer from Instituto de Estudos Superiores da Amazonia (IESAM). He holds a post-graduate degree in project management software and a master's degree in industrial processes (applied computing) from Universidade Federal do Para (UFPA). He has been working with neural networks since 2009 and has worked with Brazilian IT companies developing in Java, PHP, SQL, and other programming languages since 2006. He is passionate about programming and computational intelligence. Currently, he is a professor at Universidade da Amazonia (UNAMA) and a PhD candidate at UFPA.

About the Reviewer

Charles Griffiths is a software engineer and college professor interested in technology, medicine, economics, and nutrition. He publishes source code on GitHub and the Unity Asset Store.

I'd like to thank my friends for pointing the way forward, and their constant support no matter where I ended up going.

---

www.PacktPub.com

eBooks, discount offers, and more

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

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

`https://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thank you for purchasing this Packt book. We take our commitment to improving our content and products to meet your needs seriously—that's why your feedback is so valuable. Whatever your feelings about your purchase, please consider leaving a review on this book's Amazon page (`https://www.amazon.com/dp/1787126056`). Not only will this help us, more importantly it will also help others in the community to make an informed decision about the resources that they invest in to learn.

You can also review for us on a regular basis by joining our reviewers' club. If you're interested in joining, or would like to learn more about the benefits we offer, please contact us: `customerreviews@packtpub.com`.

Table of Contents

Preface

The life of a programmer can be described as a continual never-ending learning pathway. A programmer always faces challenges regarding new technologies or new approaches. Generally, during our lives, although we become used to repeated things, we are always subjected to new things. The process of learning is one of the most interesting topics in science, and there are a number of attempts to describe or reproduce the human learning process.

The writing of this book was guided by the challenge of facing new content and then mastering it. While the name neural networks may appear strange or even suggest that this book is about neurology, we strived to simplify these nuances by focusing on your reasons for deciding to purchase this book. We intended to build a framework that shows you that neural networks are actually simple and easy to understand, and absolutely no prior knowledge on this topic is required to fully understand the concepts we present here.

So, we encourage you to explore the content of this book to the fullest, beholding the power of neural networks when confronting big problems but always with the point of view of a beginner. Every concept addressed in this book is explained in easy language, and also with a technical background. Our mission in this book is to give you an insight into intelligent applications that can be written using simple language.

What this book covers

Chapter 1, *Getting Started with Neural Networks*, introduces neural networks concepts and shows the very basic neuron structures (Single Layer Perceptrons, Adaline), activation functions, weights, and learning algorithms. Besides, this chapter shows the process of creating basic neural networks in Java from start to finish.

Chapter 2, *Getting Neural Networks to Learn*, presents the details of the neural network learning process. Useful concepts, such as training, test, and validation are introduced. We show how to implement training and validation algorithms. This chapter also shows methods for error evaluation.

Chapter 3, *Perceptrons and Supervised Learning*, gets you acquainted with perceptrons and the supervised learning features. We present training algorithms for these types of Neural Network. The reader will learn how to implement these features in Java.

Chapter 4, *Self-Organizing Maps*, introduces unsupervised learning with self-organizing maps, namely the Kohonen neural network, and their applications, especially with classification and clustering problems.

Chapter 5, *Forecasting Weather*, covers one practical problem using neural networks, which is forecasting weather. You will be presented with a time series dataset of historical weather records from different regions, and learn how to apply preprocessing before presenting them to neural networks.

Chapter 6, *Classifying Disease Diagnosis*, introduces a classification problem, which is also encompassed in supervised learning. Using patient records data, a neural network is built to act as an expert system, being capable of giving a diagnosis based on patients, symptoms.

Chapter 7, *Clustering Customer Profies*, will teach you about clustering using neural networks, as well as applying unsupervised learning algorithms to achieve that goal.

Chapter 8, *Text Recognition*, presents another common task involving neural networks, optical character recognition (OCR), which is a very useful and impressive task that really shows the powerful learning skill neural networks have.

Chapter 9, *Optimizing and Adapting Neural Networks*, demonstrates techniques that help optimizing neural networks, such as input selection, better dataset separation into training, validation and test, as well as data filtering and the choice of number of the hidden neurons.

Chapter 10, *Current Trends in Neural Networks*, will make you aware of the current state of the art in the field of neural networks, enabling you to understand and design new strategies to apply to more complex problems.

Appendix A, *Setting up Netbeans Environment*, This appendix shows a step-by-step procedure for the reader to set up the development environment for the Netbeans IDE.

Appendix B, *Setting up Eclipse Environment*, This appendix shows a step-by-step procedure for the reader to set up your development environment if you want to use Eclipse IDE.

These Appendices are not present in the book but are available for download at the following link: `https://www.packtpub.com/sites/default/files/downloads/Neural_Network_Programming_with_Java_SecondEdition_Appendices.pdf`

What you need for this book

You'll need Netbeans (`www.netbeans.org`) or Eclipse (`www.eclipse.org`). Both are free and available for download from their websites.

Who this book is for

This book is for Java developers who want to know how to develop smarter applications using the power of neural networks. Those who deal with a lot of complex data and want to use it efficiently in their day-to-day apps will find this book quite useful. Some basic experience with statistical computations is expected.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Layers are initialized and calculated as well as the neurons; they also implement the `init()` and `calc()` methods."

A block of code is set as follows:

```
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;
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "After selecting the parameters, the training begins by clicking the **Start Training** button."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—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 book'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 book, 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 book from your account at `http://www.packtpub.com`. If you purchased this book 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 book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book'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 book is also hosted on GitHub at `https://github.com/PacktPublishing/Neural-Network-Programming-with-Java-SecondEdition`. We also have other code bundles from our rich catalog of books and videos 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 books—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 book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, 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 book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

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

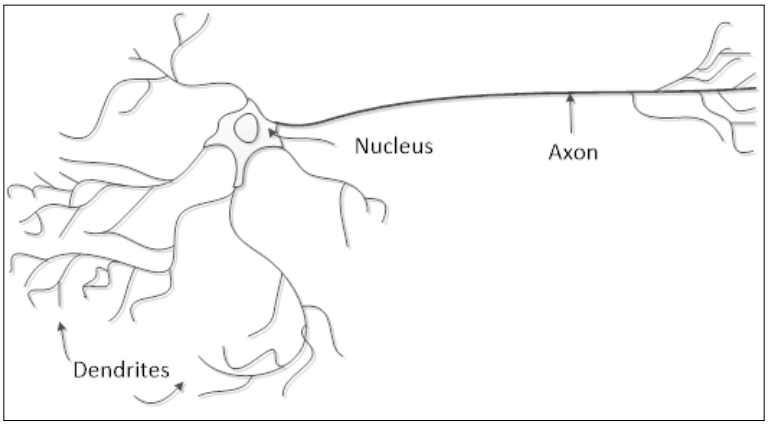
By hearing the term **neural networks** we intuitively create a snapshot of a brain in our minds, and indeed that's right, if we consider the brain to be a big and natural neural network. However, what about **artificial neural networks** (**ANNs**)? Well, here comes an opposite word to natural, and 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.

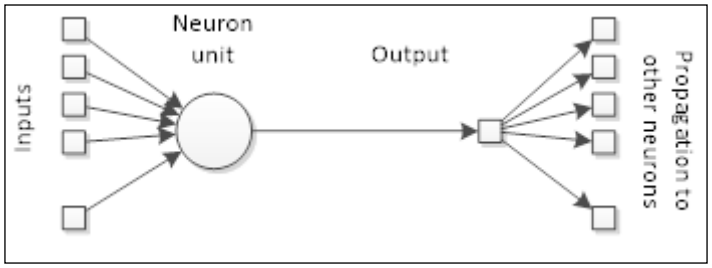
Why artificial neural networks?

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):

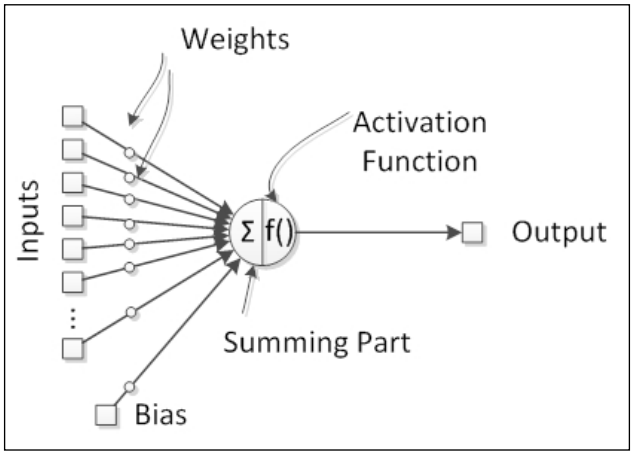
Tasks Quickly Solvable by Humans	**Tasks Quickly Solvable by Computers**
Classification of images	Complex calculation
Voice recognition	Grammatical error correction
Face identification	Signal processing
Forecast events on the basis of experience	Operating system management

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:

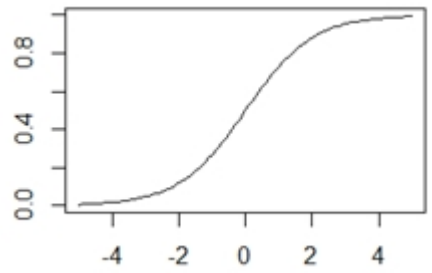
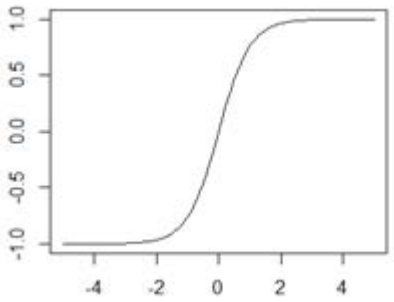
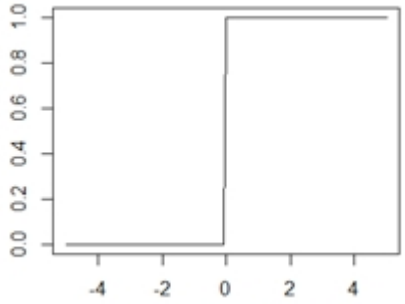


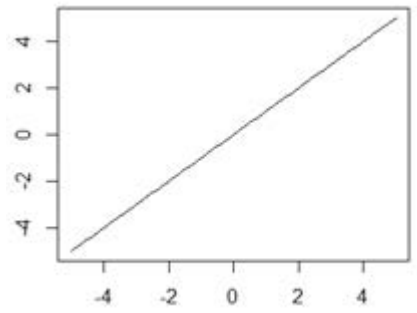
In natural neurons, there is a threshold potential that when reached, fires the axon and propagates the signal to the other neurons. This firing behavior is emulated with activation functions, which has proven to be useful in representing nonlinear behaviors in the neurons.

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	$$f(x) = \begin{pmatrix} 0 \, if \, x < 0 \\ 1 \, if \, x \geq 0 \end{pmatrix}$$	

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

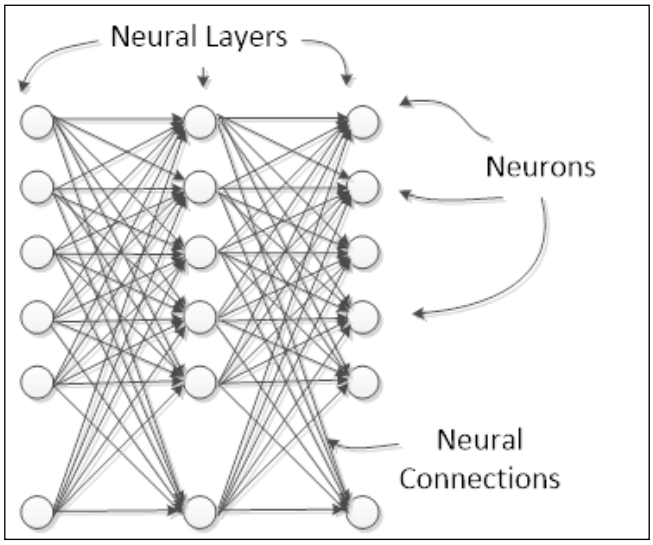
While the neural network structure can be fixed, weights represent the connections between neurons and they have the capability to amplify or attenuate incoming neural signals, thus modifying them and having the power to influence a neuron's output. Hence a neuron's activation will not be dependent on 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

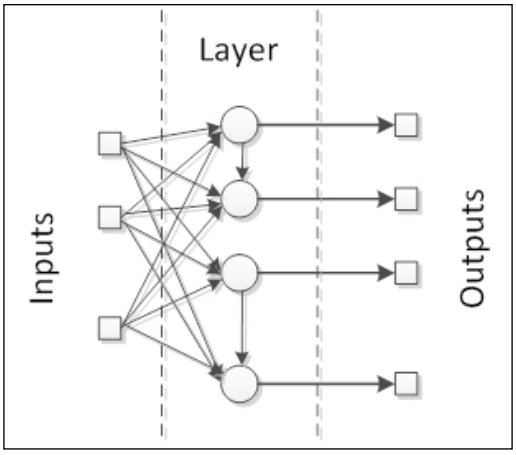
A neural network can have different layouts, depending on how the neurons or layers are connected to each other. Each neural network architecture is designed for a specific goal. Neural networks can be applied to a number of problems, and depending on the nature of the problem, the neural network should 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

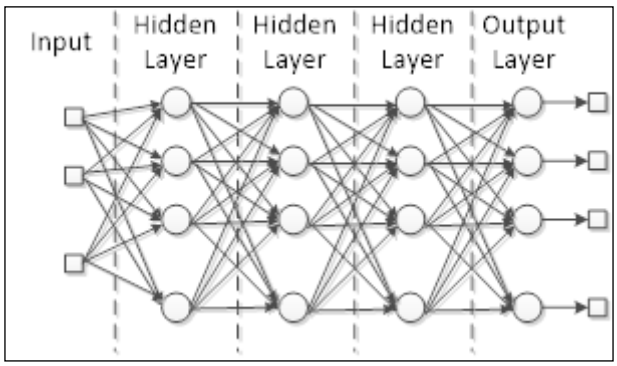
In this architecture, all neurons are laid out in the same level, forming one single layer, as shown in the following figure:



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:



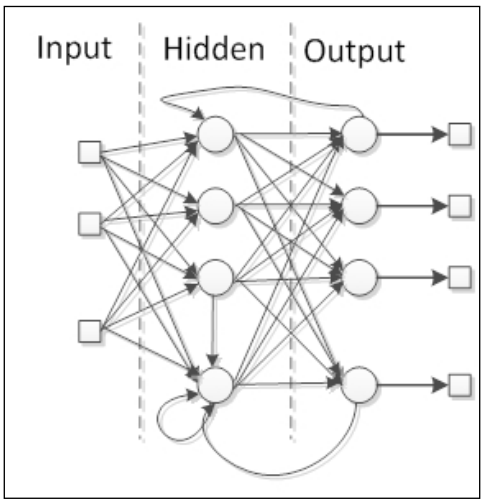
Radial basis functions and multilayer perceptrons are good examples of this architecture. Such networks are really useful for approximating real data to a function especially designed to represent that data. Moreover, because they have multiple layers of processing, these networks are adapted to learn 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

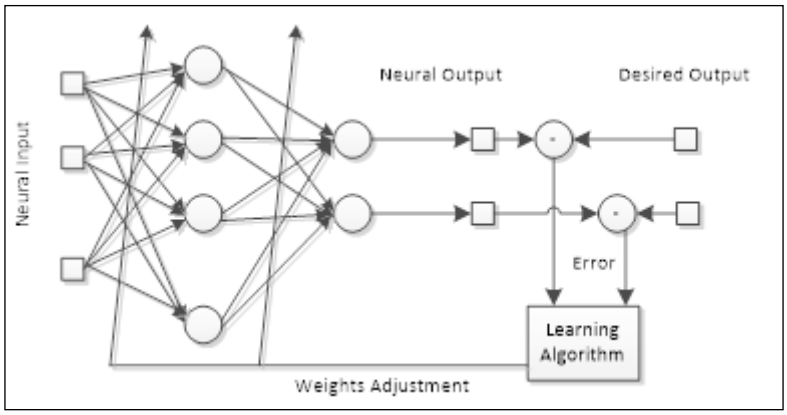
When the neural network has some kind of internal recurrence, it means that the signals are fed back in a neuron or layer that has already received and processed that signal, the network is of the feedback type. See the following figure of feedback networks:



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 numberofoutputs,
            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(numberofoutputs,outputAcFnc,
    hiddenLayer.get(numberOfHiddenLayers-1).
getNumberOfNeuronsInLayer() );
    hiddenLayer.get(numberOfHiddenLayers-1).setNextLayer(outputLayer);
}else{
    outputLayer=new OutputLayer(numberofinputs, outputAcFnc,
numberofoutputs);
    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!

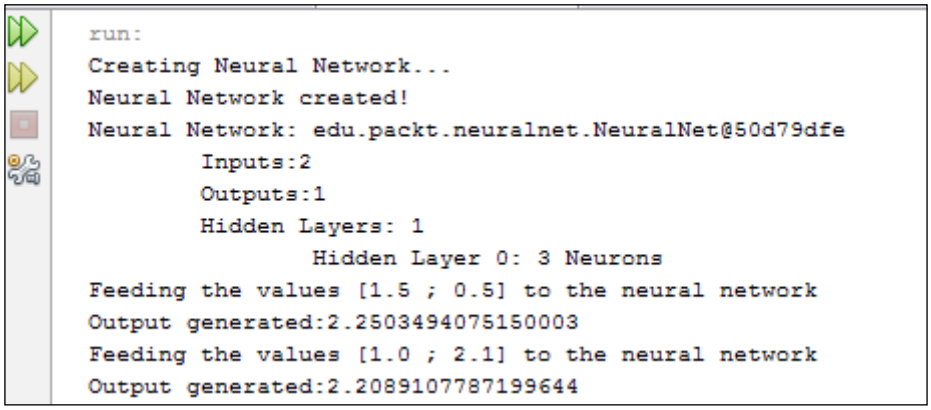
Now let's apply these classes and get some results. The following code has a `test` class, a `main` method with an object of the `NeuralNet` class called `nn`. We are going to define a simple neural network with two inputs, one output, and one hidden layer containing three neurons:

```
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:



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.

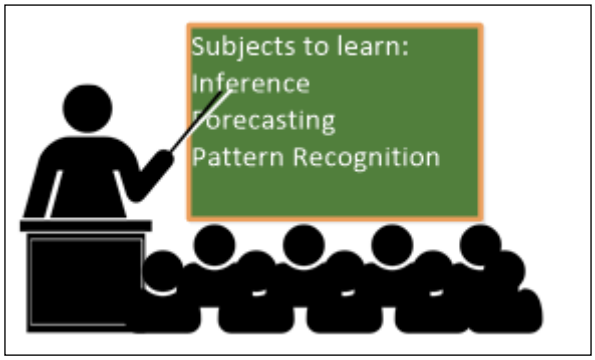
We can then think of the ANN parameters (weights) as the components of a solution. Let's imagine that each weight corresponds to a dimension and one single solution represents a single point in the solution hyperspace. For each single solution, there is an error measure informing how far that solution 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.

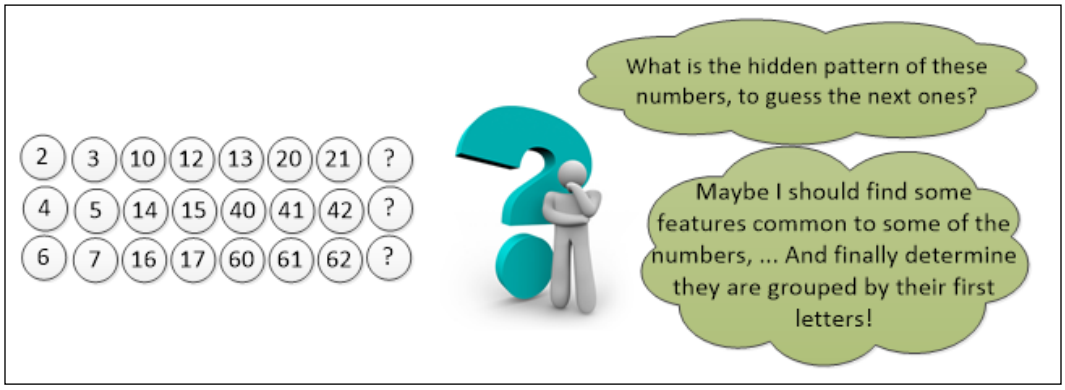
A cost function is just a measurement to be minimized in an optimization problem. That means one seeks to find the parameters that drive the cost function to the lowest possible value.

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.

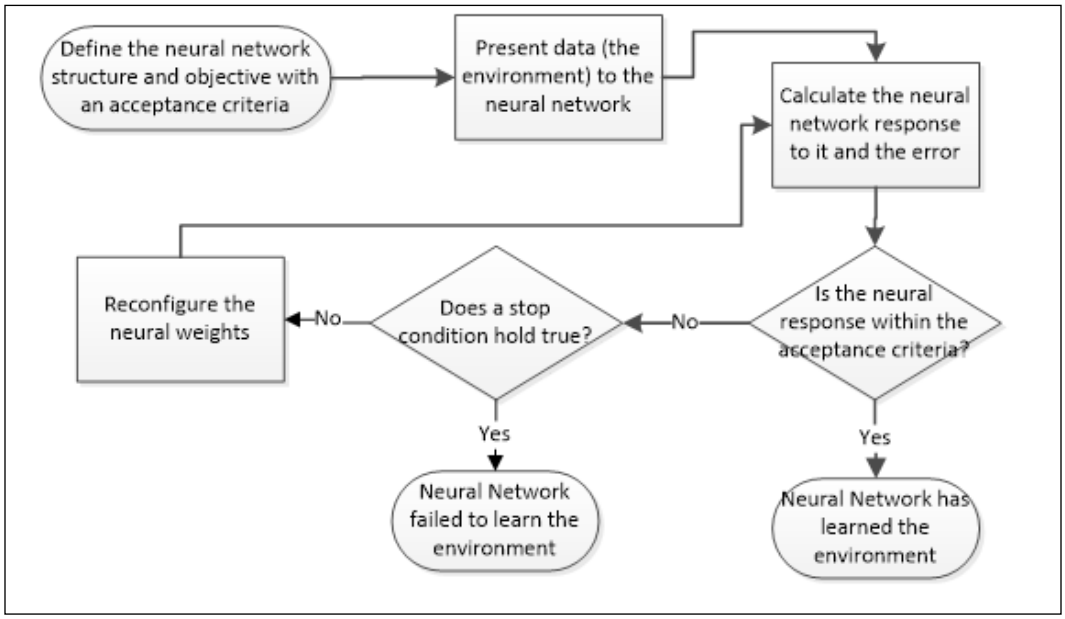
In unsupervised learning, the closer the independent data is, more similar the generated output should be, and this should be considered in the cost function, as opposed to the supervised paradigm.

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

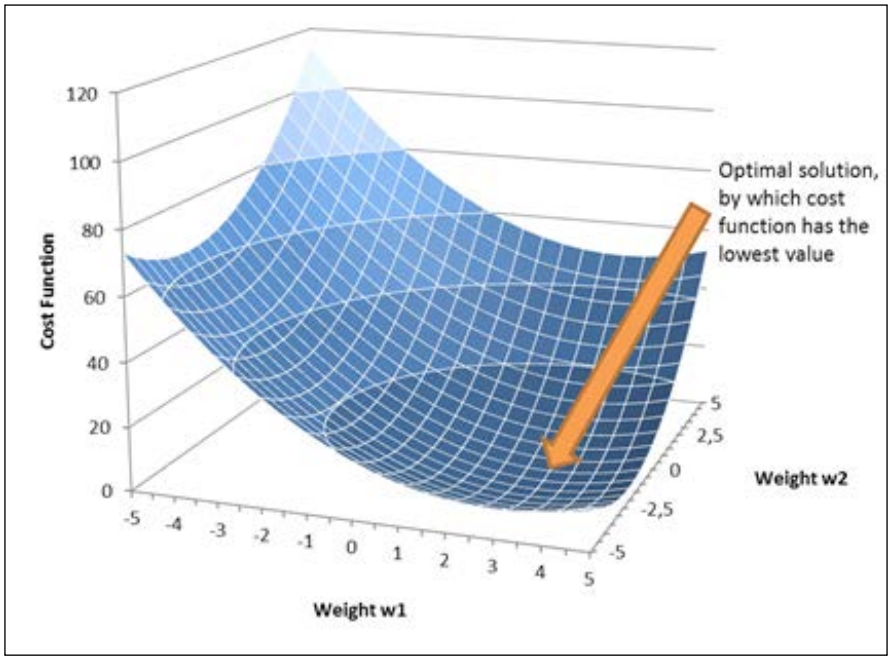
So far, we have theoretically defined the learning process and how it is carried out. But in practice, we must dive a little bit deeper into the mathematical logic, in order to implement the learning algorithm itself. For simplicity, in this chapter, we are basically covering the supervised learning 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 kth iteration and *W(k)* refers to the neural weights at the kth 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.

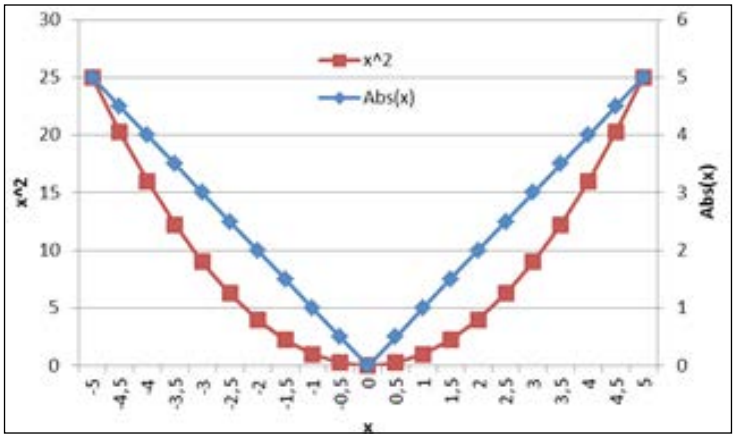
Calculating the cost function

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 (), 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 the 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:

$$E = \begin{pmatrix} (Y_{kj} - \hat{Y}_{kj}) & \cdots & (Y_{kn} - \hat{Y}_{kn}) \\ \vdots & \ddots & \vdots \\ (Y_{Nj} - \hat{Y}_{Nj}) & \cdots & (Y_{Nn} - \hat{Y}_{Nn}) \end{pmatrix}$$

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:

$$e_k = \frac{1}{2}\sum_j^n \left(Y_{kj} - \hat{Y}_{kj}\right)^2$$

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 kth 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 Y is the output vector generated by the neural network. The general error for the kth 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:

$$C\left(X, Y, \hat{Y}\right) = \frac{1}{N}\sum_{k}^{N} E_k^2$$

The weight and bias are updated according to the delta rule, that considers the partial derivatives $\partial C\frac{\left(X,Y,\hat{Y}\right)}{\partial w}$ *and* $\partial C\frac{\left(X,Y,\hat{Y}\right)}{\partial b}$ with respect to the weight and the bias, respectively. For the batch training mode, X and E are vectors:

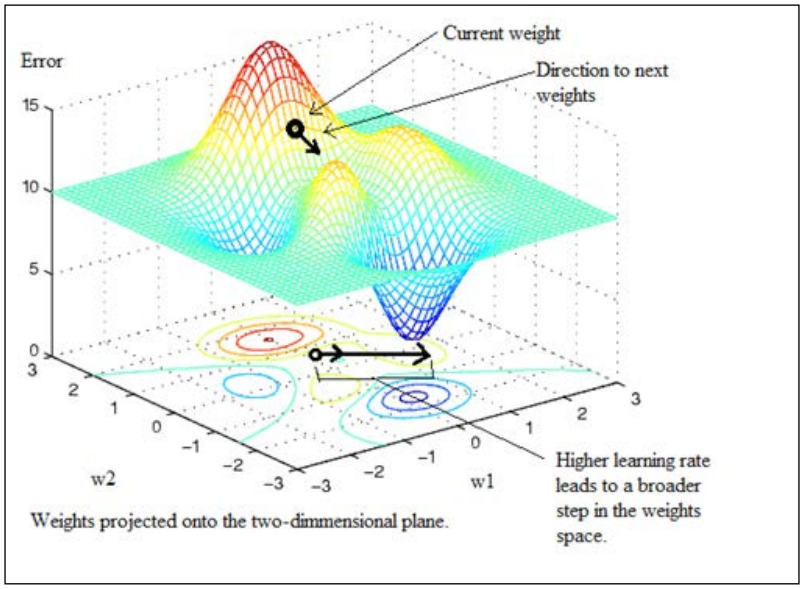
$$\Delta w = \alpha\frac{\partial C(X,Y,\hat{Y})}{\partial w} = \alpha E^T X g'(X\cdot w+b)^T$$
$$\Delta b = \alpha\frac{\partial C(X,Y,\hat{Y})}{\partial b} = \alpha E^T \begin{pmatrix}1\\ \vdots\end{pmatrix} g'(X\cdot w+b)^T$$

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=++k;
          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;
    //…
    }
```

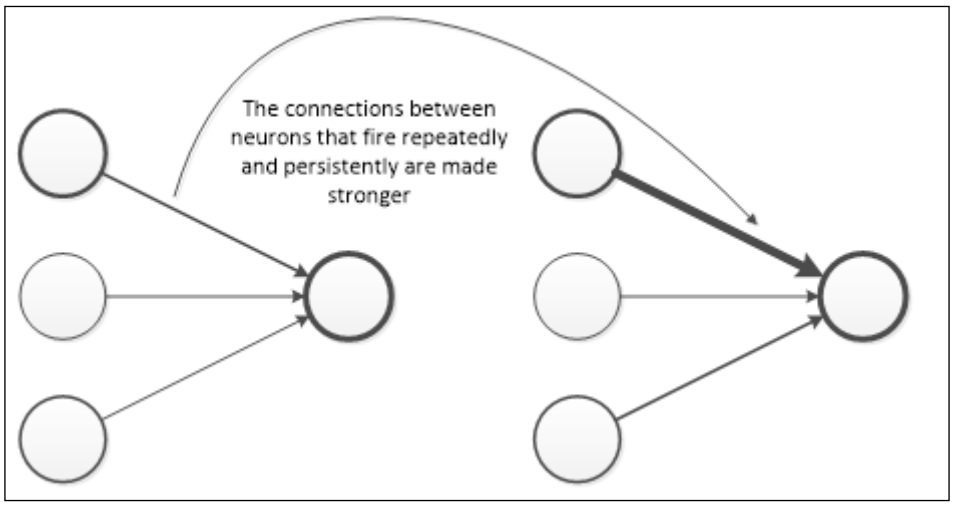
Note that in the weight update, there a call to the derivative of the activation function of the given neuron. This is needed to meet the delta rule. In the activation function interface, we've added this method `derivative( )` to be overridden in each of the implementing classes.

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 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, oj is the output of the neuron j, and oi is the output of the neuron i, also the input i for the neuron j. For the batch training case, oi and oj 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;
}
```

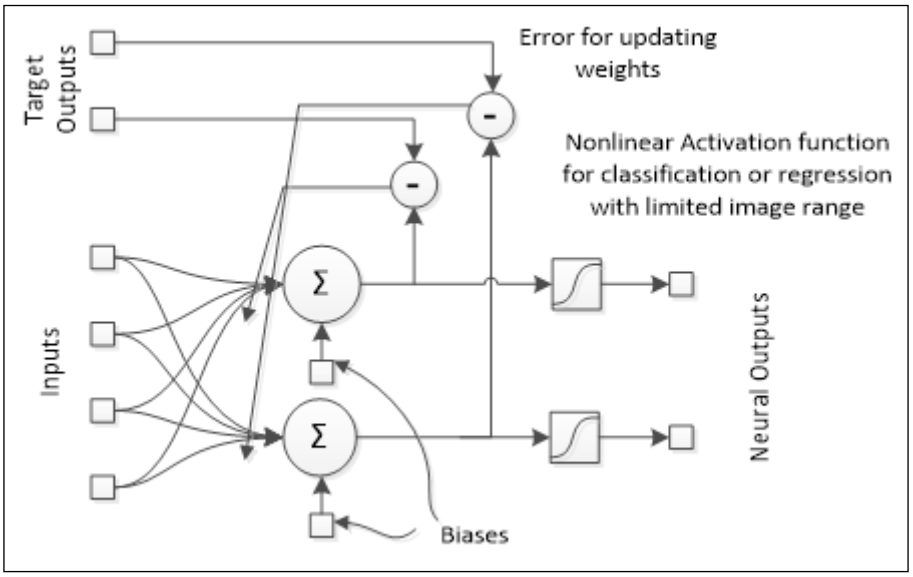
All parameters except for the absent error measures and the new measures of mean are identical to the `DeltaRule` class. The methods are quite similar, except for the `calcNewWeight( )`:

```
@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:

$$\Delta w_{ij} = \alpha(y_j - (\sum_i^n x_i w_{ij} + b))x_i$$

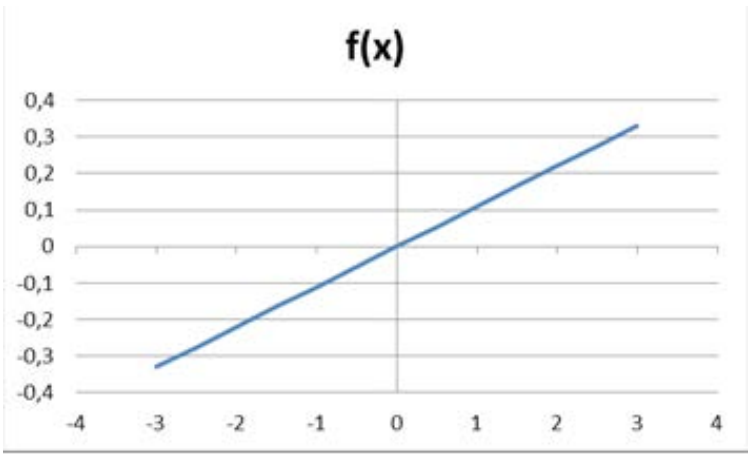
In order to implement Adaline, we create a class called **Adaline** with the following overridden `weight calcNewWeight`. To save space, we're presenting only the online case:

```
@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:

$f(x) = 0.11x$	

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 fncTest(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:

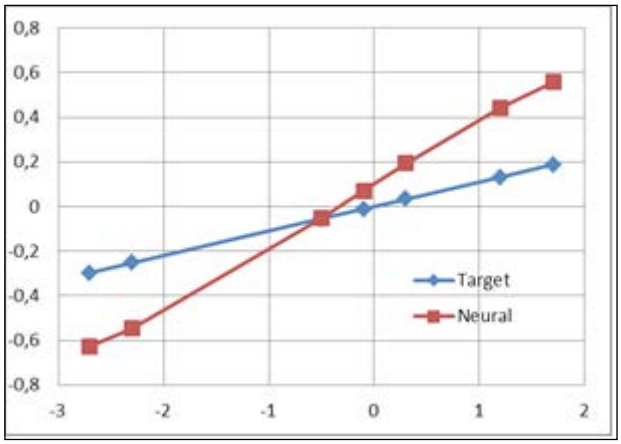
```
DeltaRule deltaRule=new DeltaRule(nn,neuralDataSet
    .LearningAlgorithm.LearningMode.ONLINE);
deltaRule.printTraining=true;
deltaRule.setLearningRate(0.3);
deltaRule.setMaxEpochs(1000);
deltaRule.setMinOverallError(0.00001);
```

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]={ 0.4422476899669740 6}
Neural Output[1]={ 0.1929759240199931 4}
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");
}
```

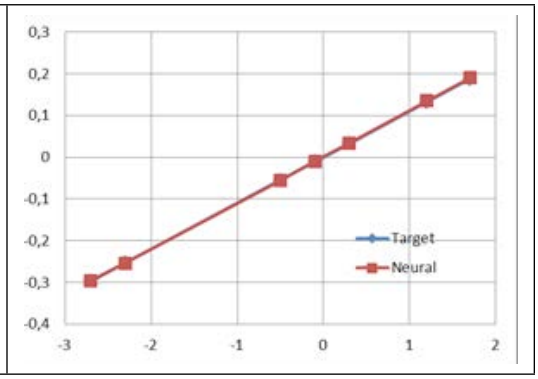
```
Beginning training
Epoch=0; Record=0; Overall Error=0.06586810467152877
Epoch=0; Record=1; Overall Error=0.0642007959986327
Epoch=0; Record=2; Overall Error=0.06427692265805388
Epoch=0; Record=3; Overall Error=0.06102840935680022
Epoch=0; Record=4; Overall Error=0.04880202611839414
Epoch=0; Record=5; 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 succesful!
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:

```
Neural Output after training:
Neural:
Neural Output[0]={ 0.13572519792474516}
Neural Output[1]={ 0.03448715923560529}
Neural Output[2]={ -0.05616629902513205}
Neural Output[3]={ -0.254631922880871}
Neural Output[4]={ 0.19090081884413404}
Neural Output[5]={ -0.010861888406637488}
Neural Output[6]={ -0.29654101894096135}
```



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`:

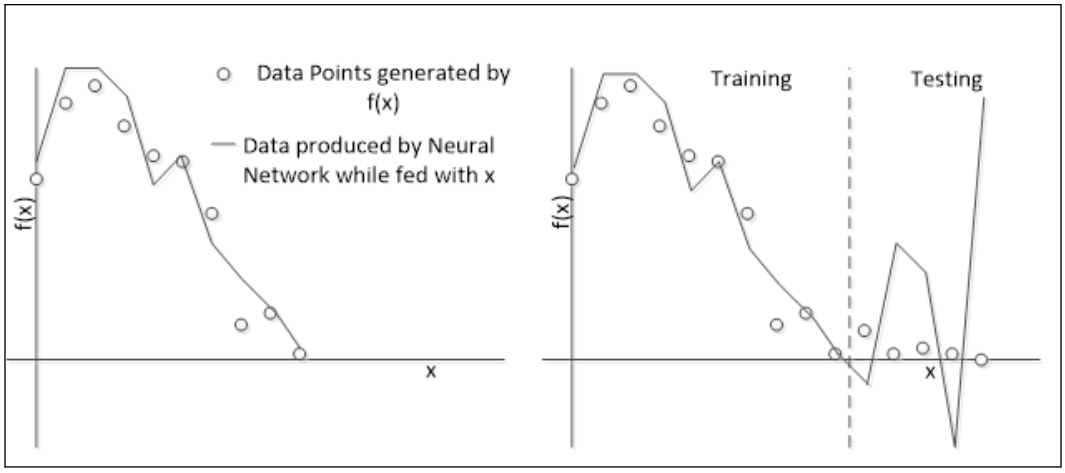
```
weight = nn.getOutputLayer().getWeight(0, 0);
bias = nn.getOutputLayer().getWeight(1, 0);
System.out.println("Weight found:"+String.valueOf(weight));
System.out.println("Bias found:"+String.valueOf(bias));
//Weight found:0.2668421011698528
//Bias found:0.0011258204676042108
```

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):



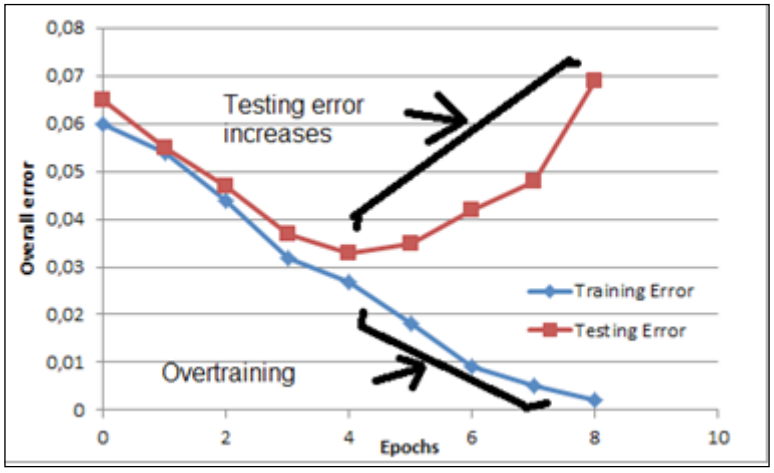
In this case, we see that the neural network failed to learn the whole environment (the function *f(x)*). This happens because of a number of reasons:

- 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:

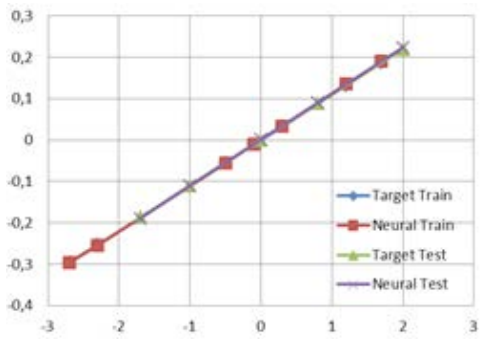


One of ways to prevent overtraining and overfitting is checking the testing error when the training goes on. When the testing error starts to increase, it is time to stop. This will be covered more in detail in the next chapters.

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();
```

```
Weight found:0.2668421011698528
Bias found:0.0011258204676042108
Neural:
Neural Output[0]={ -0.18997857740259588}
Neural Output[1]={ -0.11245179065557878}
Neural Output[2]={ 4.784736622183342E-4}
Neural Output[3]={ 0.09095273671742062}
Neural Output[4]={ 0.22345928013217864}
```



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 YT, then we evaluate a cost function whose aim is to reduce the error between the neural output Y and the target output YT.

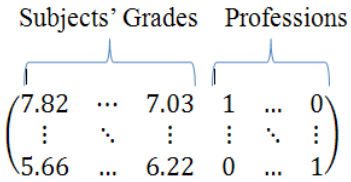
In supervised learning, there are two major categories of tasks involved, which are detailed as follows: classification and regression.

Classification – finding the appropriate class

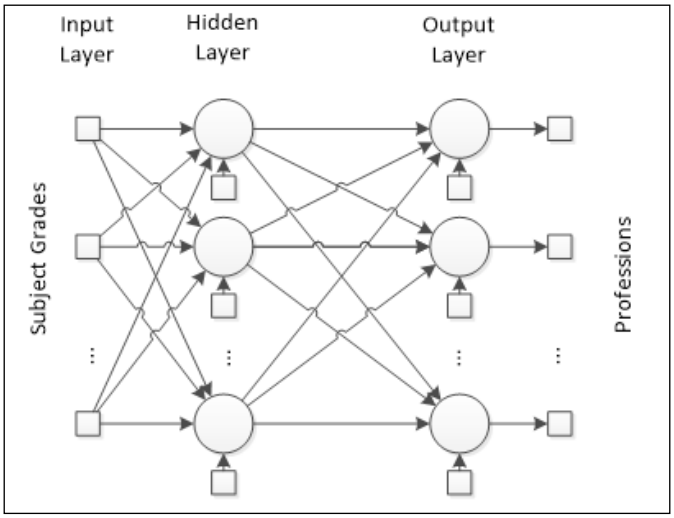
Neural networks also work with categorical data. Given a list of classes and a dataset, one wishes to classify them according to a historical dataset containing records and their respective class. The following table shows an example of this dataset, considering the subjects' average grades between **0** 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:



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:

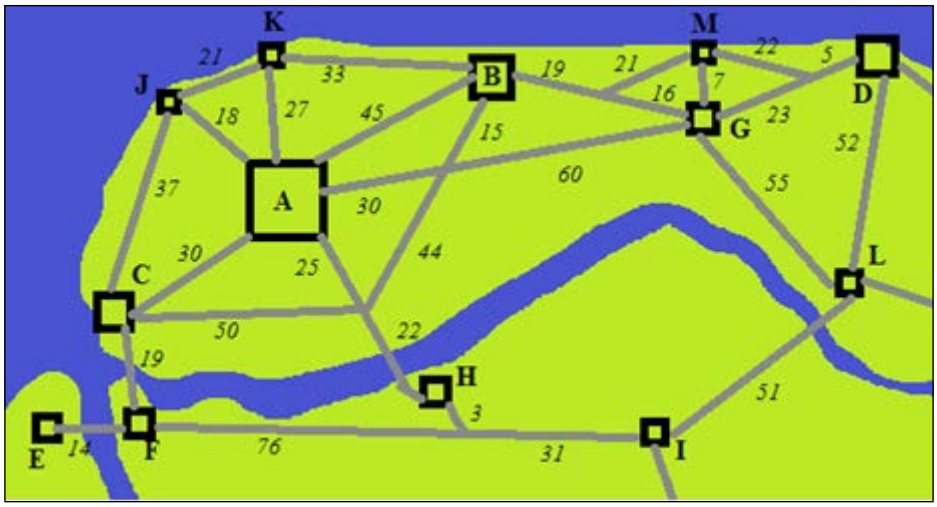
Input independent data				**Output dependent data**			
X1	X2	…	XM	T1	T2	…	TN
x1[0]	x2[0]	…	xm[0]	t1[0]	t2[0]	…	tn[0]
x1[1]	x2[1]	…	xm[1]	t1[1]	t2[1]	…	tn[1]
…	…	…	…	…	…	…	…
x1[k]	x2[k]	…	xm[k]	t1[k]	t2[k]	…	tn[k]

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:



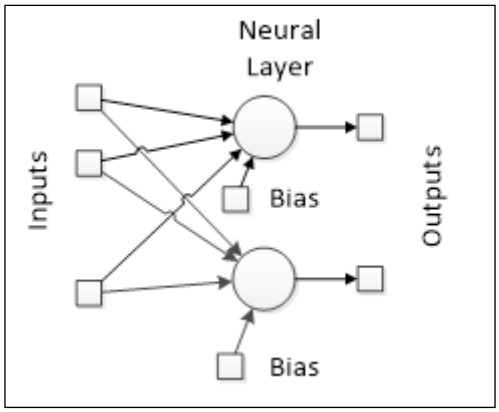
Features city of origin			Features city of destination			Features of the way between			Ticket fare
Population	GDP	Routes	Population	GDP	Routes	Distance	Time	Stops	
500,000	4.5	6	45,000	1.5	5	90	1,5	0	15
120,000	2.6	4	500,000	4.5	6	30	0,8	0	10
30,000	0.8	3	65,000	3.0	3	103	1,6	1	20
35,000	1.4	3	45,000	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:

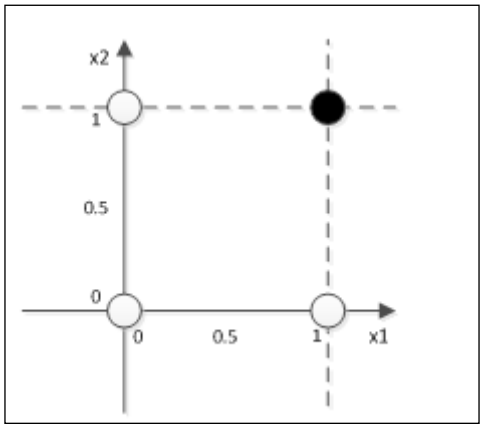
```
int numberOfInputs=3;
int numberOfOutputs=2;

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

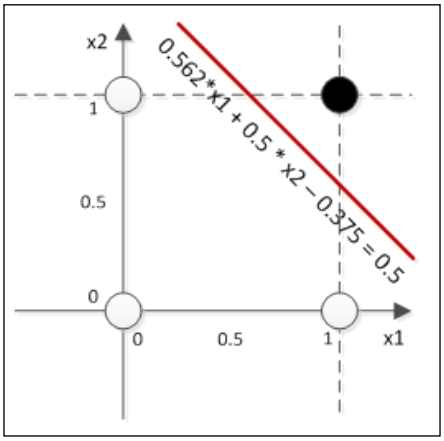
The example consists of an AND function that takes two inputs, **x1** and **x2**. That function can be plotted in a two-dimensional chart as follows:



And now let's examine how the neural network evolves the training using the perceptron rule, considering a pair of two weights, **w1** and **w2**, initially **0.5**, and bias valued **0.5** as well. Assume learning rate η equals **0.2**:

Epoch	**x1**	**x2**	**w1**	**w2**	**b**	**y**	**t**	**E**	**Δw1**	**Δw2**	**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

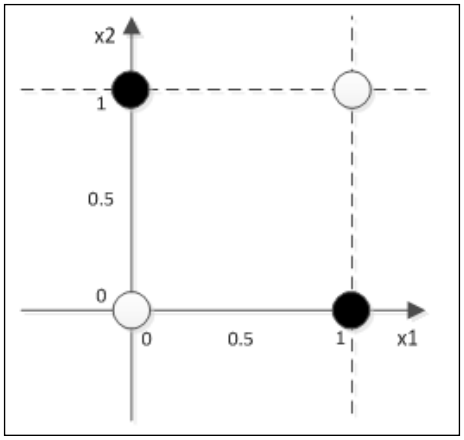
After **89** epochs, we find the network to produce values near to the desired output. Since in this example the outputs are binary (zero or one), we can assume that any value produced by the network that is below **0.5** is considered to be **0** and any value above **0.5** is considered to be **1**. 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 *w1=0.562*, *w2=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

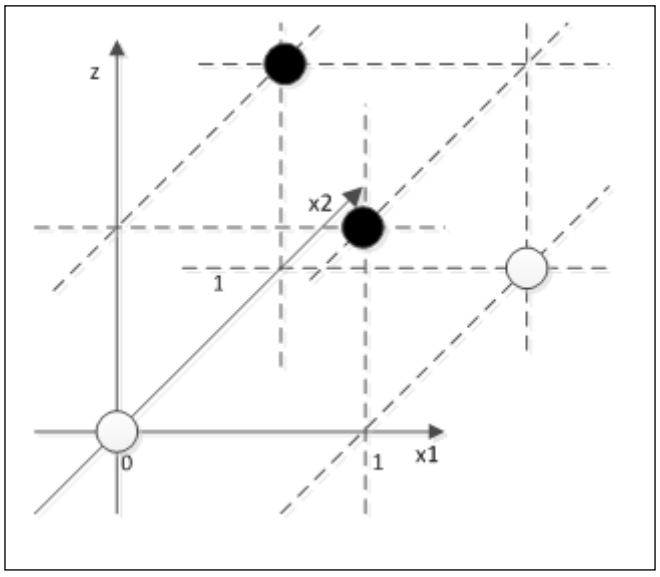
Now let's analyze the XOR case:



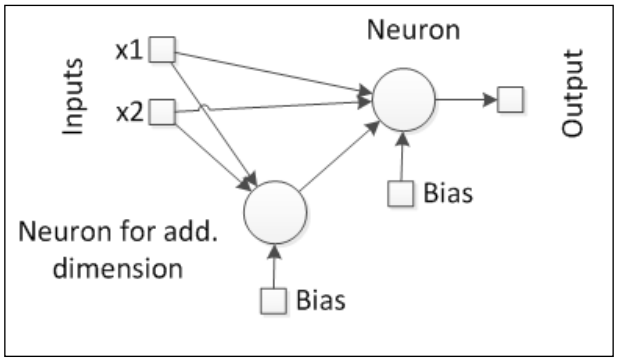
We see that in two dimensions, it is impossible to draw a line to separate the two patterns. What would happen if we tried to train a single layer perceptron to learn this function? Suppose we tried, let's see what happened in the following table:

Epoch	x1	x2	w1	w2	b	y	t	E	Δw1	Δw2	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:

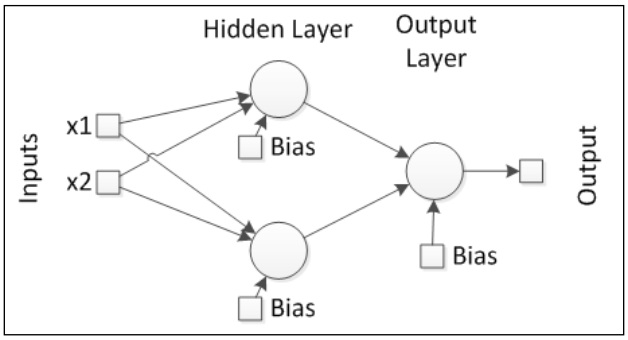


Okay, now the perceptron has three inputs, one of them being a composition of the other. This also leads to a new question: how should that composition be processed? We can see that this component could act as a neuron, so giving the neural network a nested architecture. If so, there would another new 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.

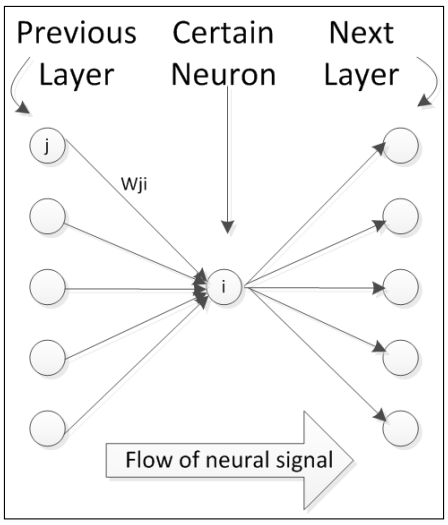
There are also some references that consider the input layer as the nodes that collect input data; therefore, for those cases, the MLP is considered to have at least three layers. For the purpose of this book, let's consider the input layer as a special type of layer which has no weights, and as the 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:

$$y_o = f_o\left(\sum_{i=1}^{n_{h_l}} w_i f_i\left(\sum_{j=1}^{n_{h_{l-1}}} w_{ij} f_j\left(\sum_{k=1}^{n_{h_{l-2}}} w_{jk} f_k(\cdots) + b_j\right) + b_i\right) + b_o\right)$$

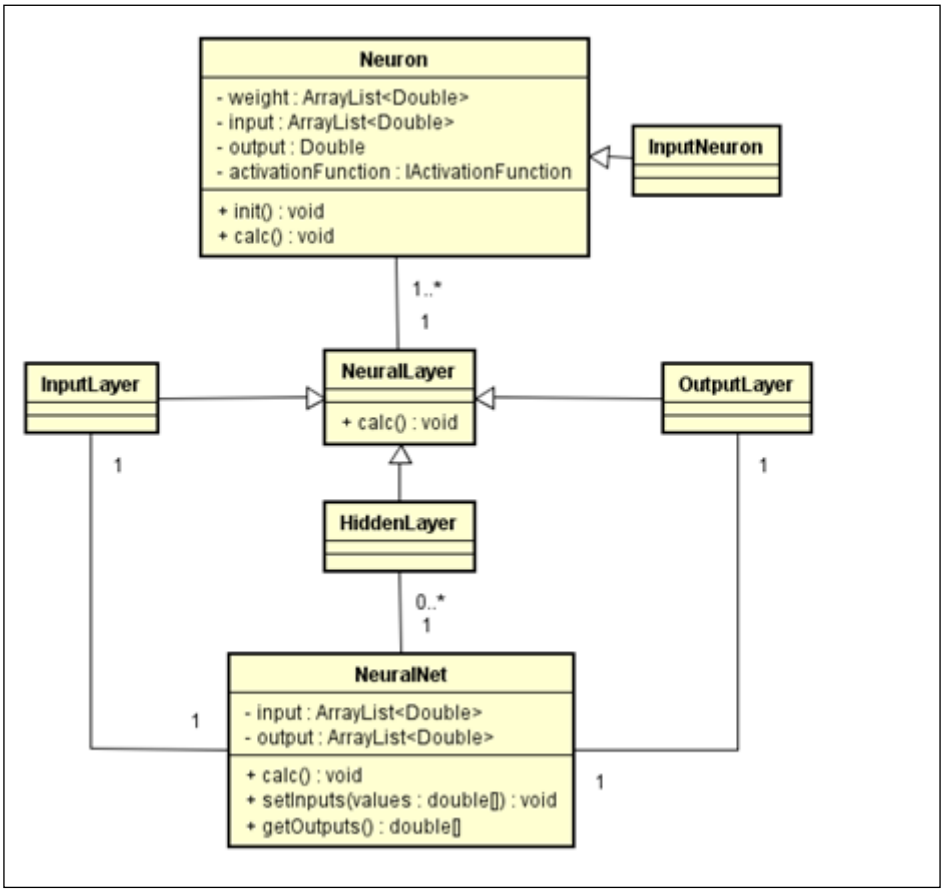
Here, *yo* is the network output (should we have multiple outputs, we can replace *yo* with Y, representing a vector); *fo* is the activation function of the output; *l* is the number of hidden layers; *nhi* is the number of neurons in the hidden layer *i*; *wi* is the weight connecting the *i* th neuron of the last hidden layer to the output; *fi* is the activation function of the neuron *i*; and *bi* 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 *xi*.

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:



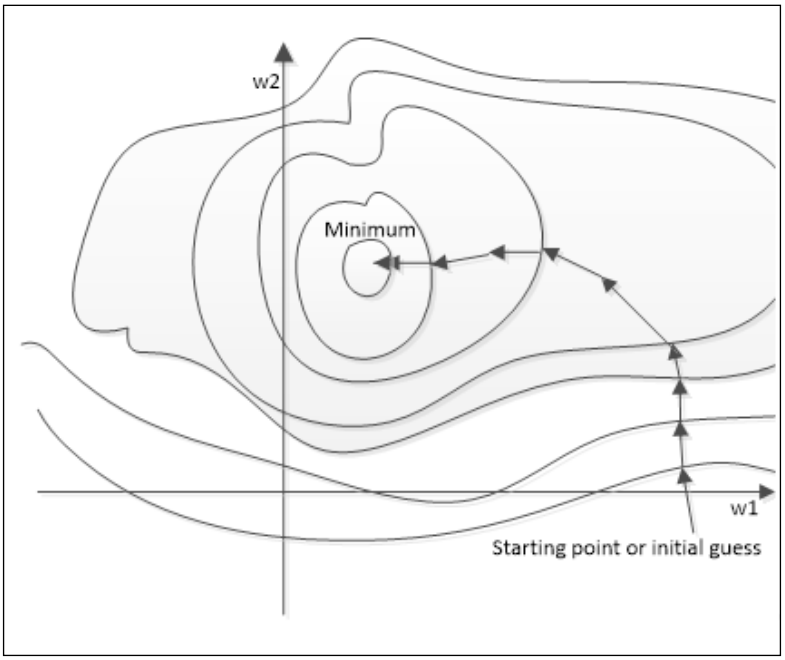
One can see that the neural network structure is hierarchical. A neural network is composed of layers that are composed of neurons. In the MLP architecture, there are three types of layers: input, hidden, and output. So suppose that in Java, we would like to define a neural network consisting of three 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:

$$\Delta w_i = \eta\left(t(k) - y(k)\right)x_i[k]\,g'\left(h_i(k)\right)$$

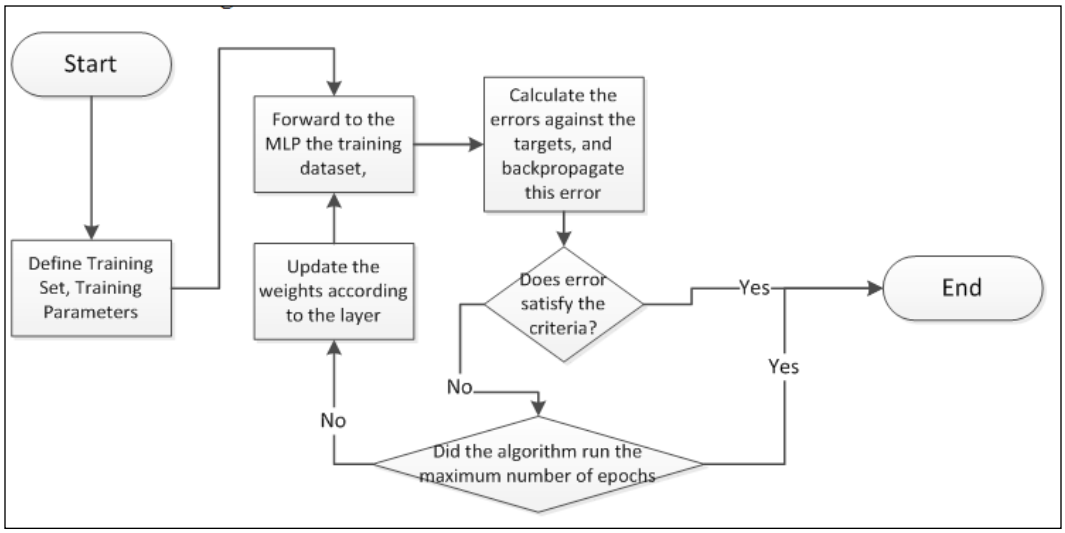
The function that the Delta Rule wants to minimize is the error between the neural network output and the target output, and the parameters to be found are the neural weights. This is an enhanced learning algorithm compared to the perceptron rule, because it takes into account the activation function 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, *wji* is the weight between the neurons *i* and *j*, *oi* is the output of the *ith* neuron, and *hi* is the weighted sum of that neuron's inputs before passing to activation function. Remember that *oi=f(hi)*, *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, oi is the *ith* output, *ti* is the desired *ith* output, *f' (hi)* is the derivative of the output activation function, and *hi* is the weighted sum of the *ith* 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 *wil* is the weight that connects the current neuron to the lth 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=++k; // 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:

$$\Delta W = \left(J^T J + \quad I\right) \quad J^T \left(Y - f\left(X \; W\right)\right)$$

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:

$$J = \begin{bmatrix} \frac{\partial f\left(X[1]\right), W)}{W_1} & \cdots & \frac{\partial f\left(X[1], W\right)}{W_p} \\ \vdots & \ddots & \vdots \\ \frac{\partial f\left(X[k], W\right)}{W_1} & \cdots & \frac{\partial f\left(X[k], W\right)}{W_p} \end{bmatrix}$$

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:

$$E(W) = \sum \left[Y_i - f(X_i, W) \right]^2$$

Converting to the vector and notation:

$$E(W + \Delta W) = \sum \left[Y_i - f(X_i, W) - J_i \Delta W \right]^2$$

Finally, by setting the error E to zero, we get the Levenberg-Marquardt equation after some manipulation:

$$E(W + \Delta W) = \left\| Y - f(X, W) - J\Delta W \right\|^2$$

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 backpropgation
    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:

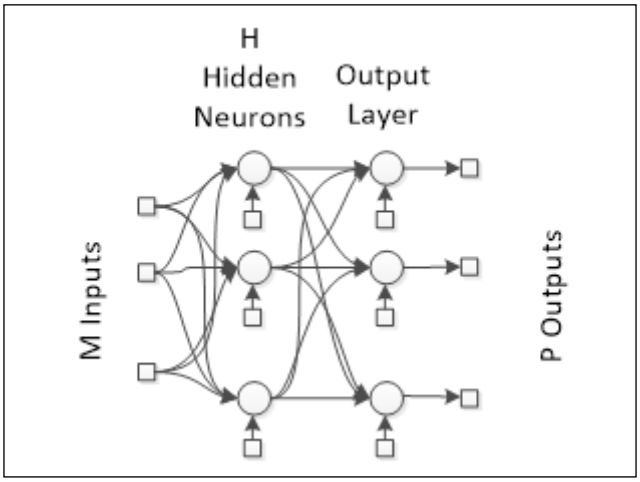
```
Neuron n=nl.getNeuron(k);
double currWeight=n.getWeight(j);
double newWeight=currWeight+deltaWeight.getValue(i++,0);
newWeights.get(l).get(k).set(j,newWeight);
lastWeights.get(l).get(k).set(j,currWeight);
n.updateWeight(j, newWeight);
```

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:

$$H = \begin{pmatrix} g\left(X_0 W_0 + b_0\right) & \cdots & g\left(X_0 W_H + b_H\right) \\ \vdots & \ddots & \vdots \\ g\left(X_N W_0 + b_0\right) & \cdots & g\left(X_N W_H + b_H\right) \end{pmatrix}$$

$$X_i = \begin{pmatrix} x_0[i] & \cdots & x_m[i] \end{pmatrix}^T$$

$$W_j = \begin{pmatrix} w_{0j} & \cdots & w_{mj} \end{pmatrix}^T$$

$$Y = \begin{pmatrix} y_0[0] & \cdots & y_p[0] \\ \vdots & \ddots & \vdots \\ y_0[N] & \cdots & y_p[N] \end{pmatrix} = \begin{pmatrix} H & 1 \end{pmatrix} \begin{pmatrix} \beta_0 & \cdots & \beta_p \\ B_0 & \cdots & B_p \end{pmatrix}$$

Here, *H* is the output of the hidden layer, *g()* is the activation function of the hidden layer, *Xi* is the *ith* input record, *Wj* is the weight vector for the *jth* hidden neuron, *bj* is the bias of the *jth* hidden neuron, βp 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:

$$\beta = \left(H^T H\right)^{-1} H^T T$$

$$T = \begin{pmatrix} YT_0[0] & \cdots & YT_p[0] \\ \vdots & \ddots & \vdots \\ YT_0[N] & \cdots & YT_p[N] \end{pmatrix}$$

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 Percepetron
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 unsuccesful
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=0.0092370946409205 8
End of training
Training successful!
Overall Error:0.0092370946409205 8
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

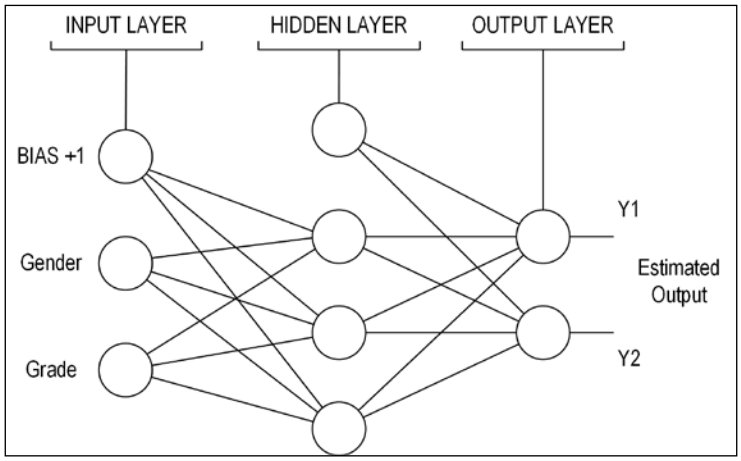
In Brazil, one of the ways for a person to enter university consists of taking an exam and if he/she achieves the minimum grade for the course that he/she is seeking, then he/she can enroll. To demonstrate the backpropagation algorithm, let us consider this scenario. The data shown in the table was collected 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(nnelm,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.

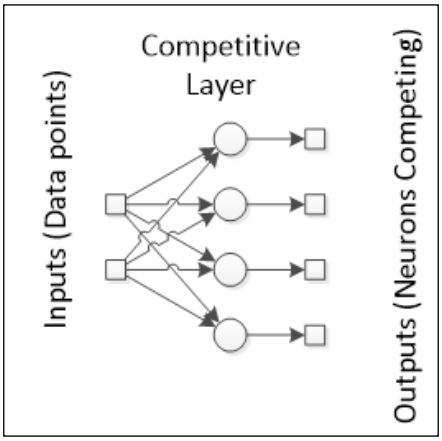
While in supervised learning algorithms, we usually have a smaller number of outputs, for unsupervised learning, there is a need to produce an abstract data representation that may require a high number of outputs, but, except for classification tasks, their meaning is totally different than the one 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.

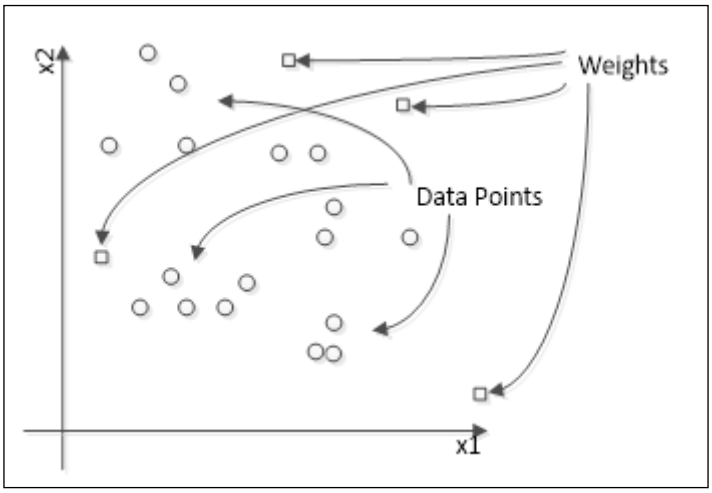
For this learning, the bias is dropped from the neurons, so the neurons will process only the weighted inputs.

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.

Considering that every input neuron *i* is connected to all output neurons *j* through a weight *wij*, in our case, we would have a set of weights:

$$W = \left[w_{11} w_{12} w_{21} w_{22} w_{13} w_{14} w_{23} w_{24} \right]$$

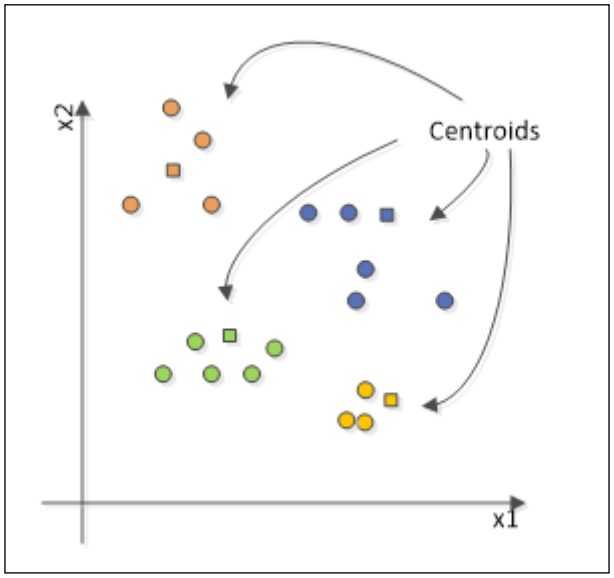
Provided that the weights of every neuron have the same dimensionality of the input data, let's consider all the input data points together in a plot with the weights of each neuron:



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).

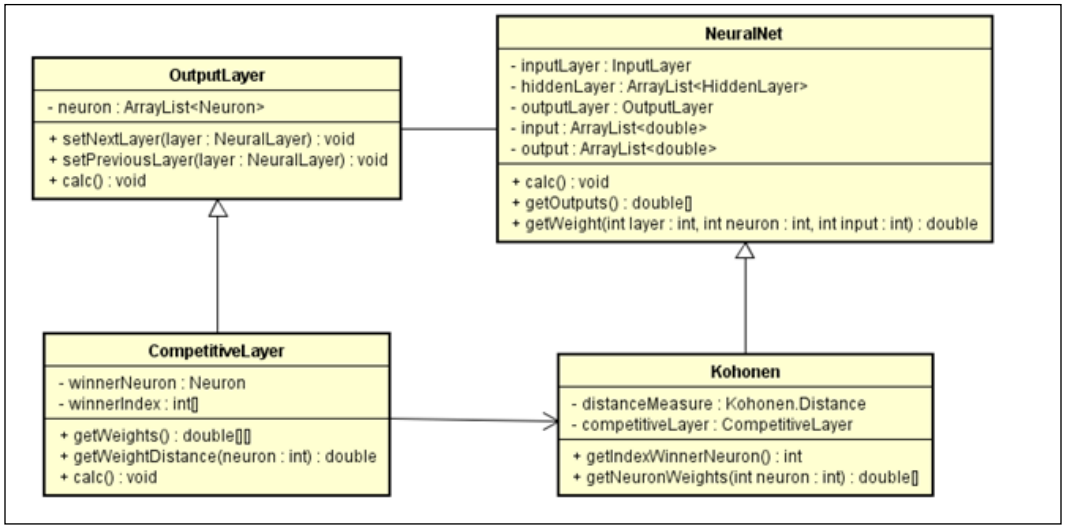
Theoretically, a Kohonen Network would be able to provide a 3-D (or even in more dimensions) representation of the data; however, in printed material such as this book, it is not practicable to show 3-D charts without overlapping some data. Thus in this book, we are going to deal only with 0-D, 1-D, 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.

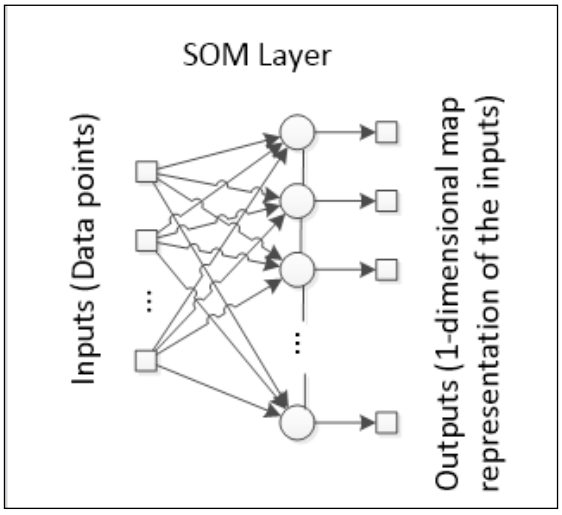
The following code snippet define a zero-dimensional SOM:

```
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

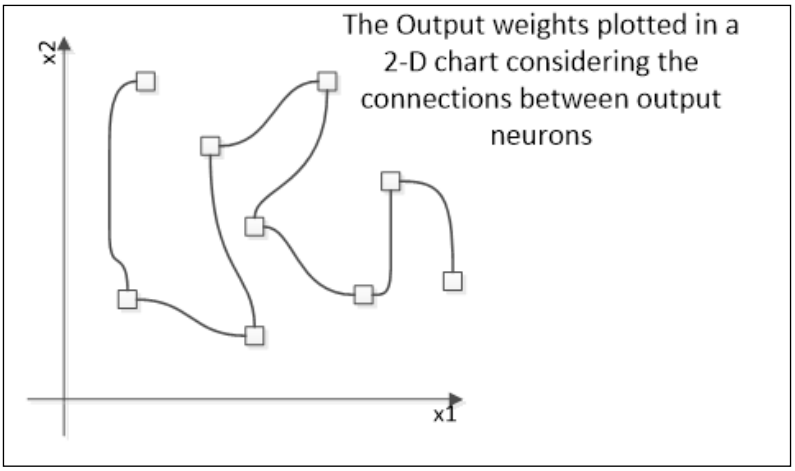
This architecture is similar to the network presented in the last section, **Competitive learning**, with the addition of neighborhood amongst the output neurons:



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:



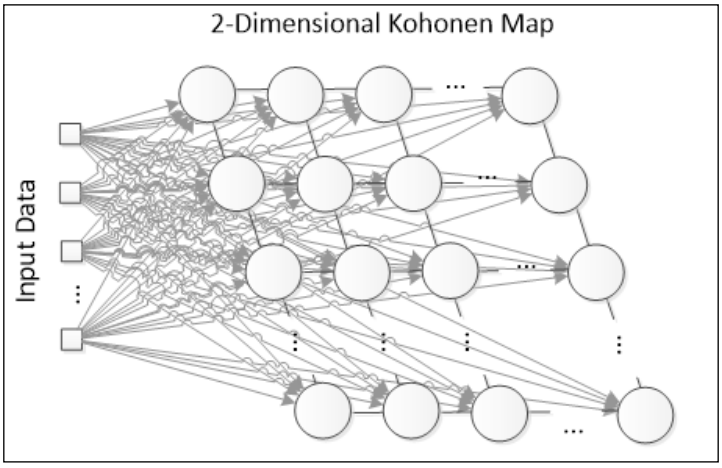
In the chart here presented, for simplicity, we plotted only the output weights to demonstrate how the map is designed in a (in this case) 2-D space. After training over many iterations, the neural network converges to a final shape that represent all data points. Provided that structure, a certain set 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`:

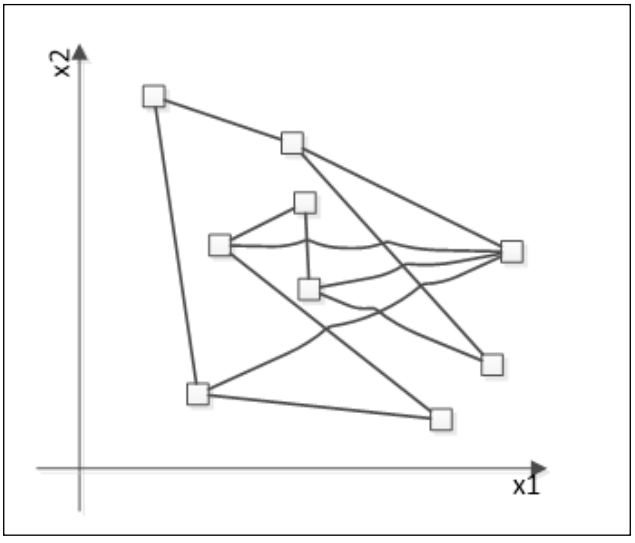
```
Kohonen kn1 = new Kohonen(numberOfInputs,numberOfNeurons,new
UniformInitialization(-1.0,1.0),1);
```

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 x N neurons, interconnected like a grid:

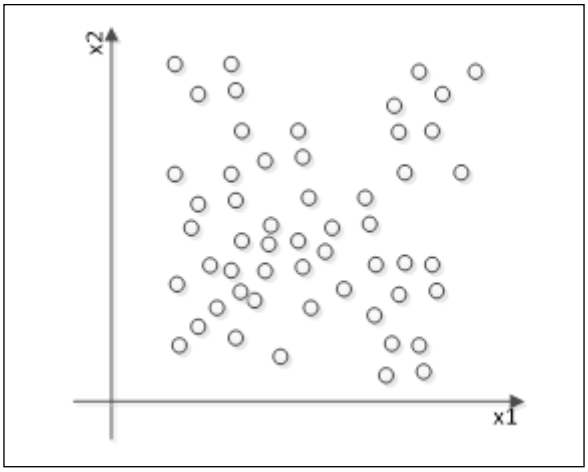


In the 2-D SOMs, every neuron now has up to four neighbors (in the square configuration), although in some representations, the diagonal neurons may also be considered, thus resulting in up to eight neighbors. Hexagonal representations are also useful. Let's see one example of what a 3x3 SOM plot looks like in a 2-D chart (considering two input variables):

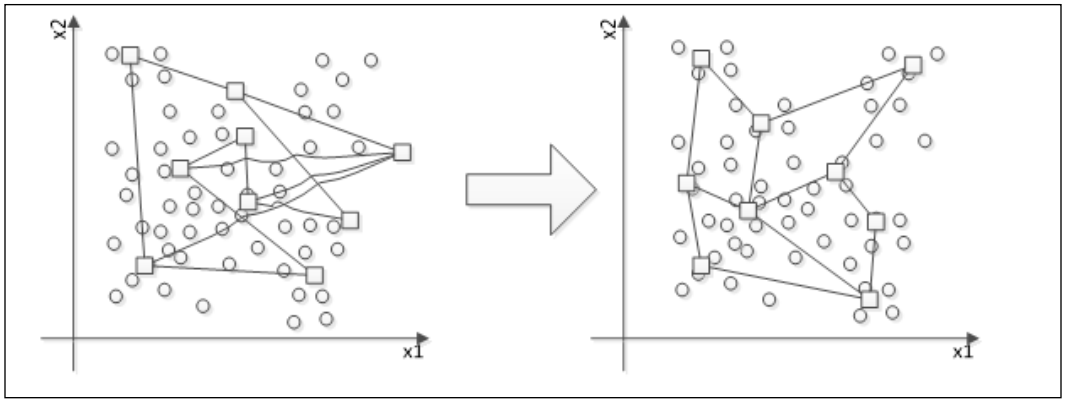


At first, the untrained Kohonen Network shows a very strange and screwed-up shape. The shaping of the weights will depend solely on the input data that is going to be fed to the SOM. Let's see an example of how the map starts to organize itself:

- 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*.

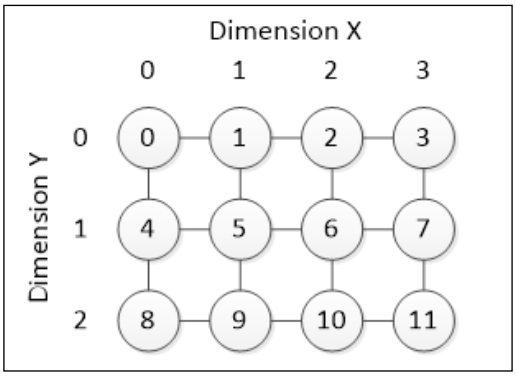
The grid on a chart is just the more used and didactic. There are other ways of showing the SOM diagram, such as the U-matrix and the cluster boundaries.

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 x 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`:



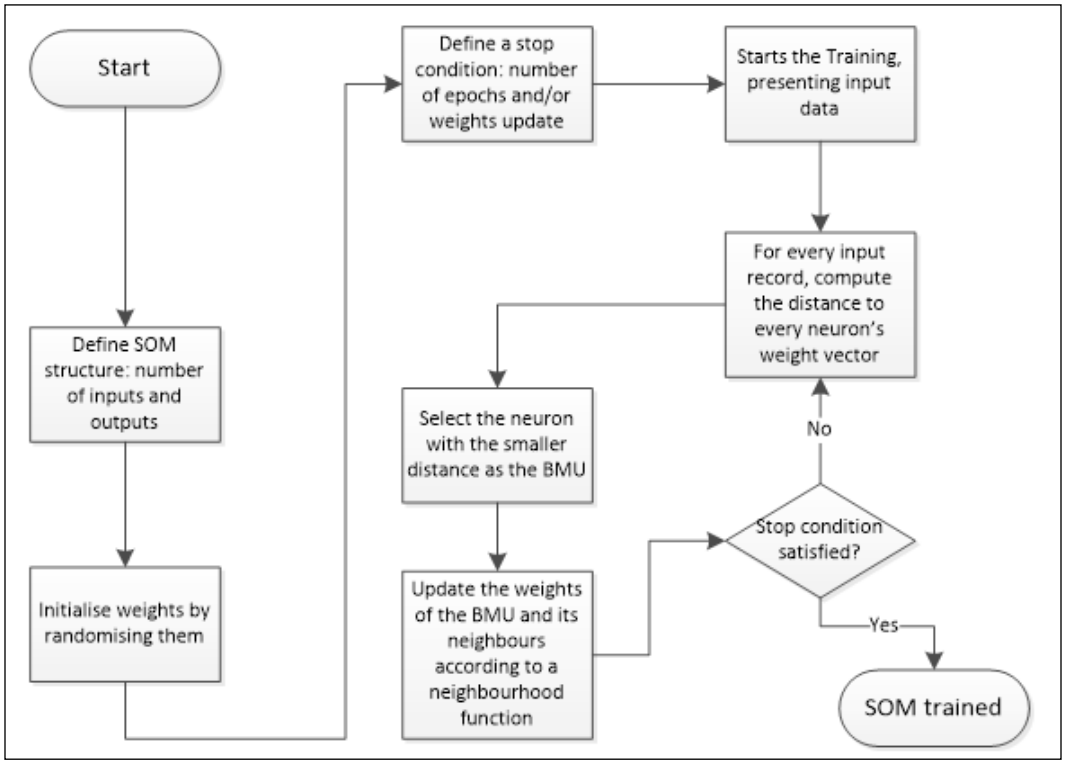
In the illustration above, 12 neurons are arranged in a 3 x 4 grid. Another feature added in this class is the indexing of neurons by the position in the grid. This allows us to get subsets of neurons (and weights), one entire specific row or column of the grid, for example:

```
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;
}
```

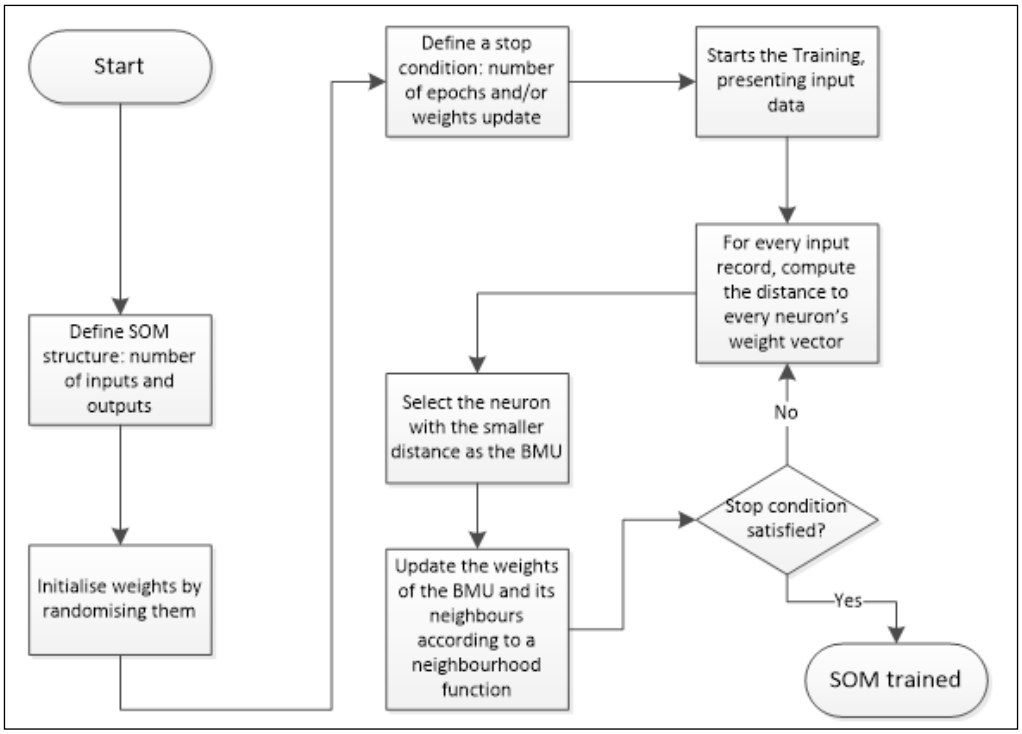
SOM learning algorithm

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 *Θ(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*:

$$\Theta(u,v,s,t) = \exp\left(\frac{d_{u,v}\left(N_{u,}N_{v}\right)^{2}}{2\sigma^{2}(s,t)}\right)$$

Here, *du,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, 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

`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;`
`}`
	`@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;`
	`}`

`public double neuronDistance(int u,int v){`
`  return Math.abs(coordNeuron[u][0]-coordNeuron[v][0]);`
`}`
	`@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);`
	`}`

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 is the initial learning rate. Finally, considering the neighborhood function and the learning rate, the weight update rule is as follows:

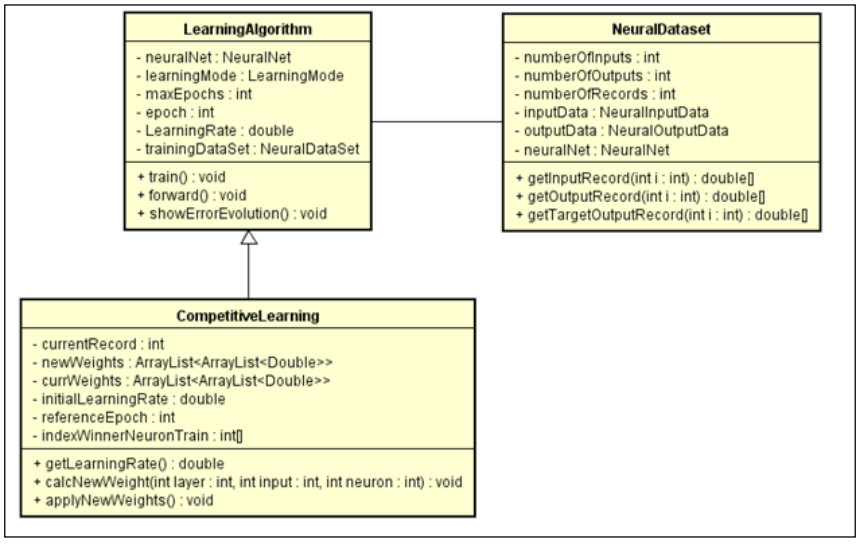
$$a_0$$

$$\Delta W_{kj} = \Theta(i,j,s,t)\,a(s,t)\left(X_k - W_{kj}\right)$$

Here, X_k is the *kth* input, and W_{kj} is the weight connecting the *kth* input to the *jth* 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=++k;
    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.

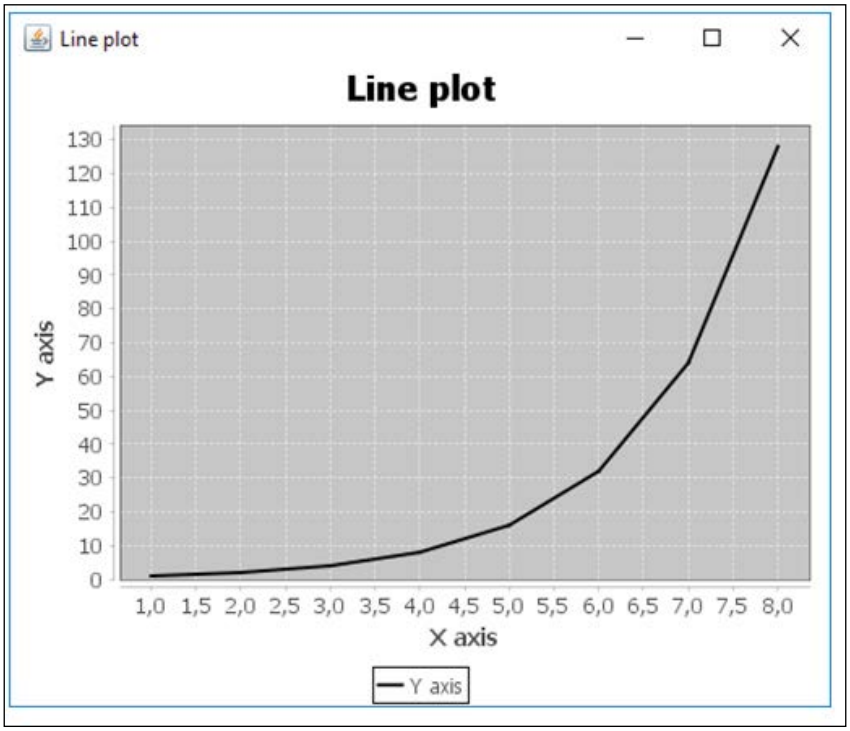
Time to play: SOM applications in action. Now it is time to get hands-on and implement the Kohonen neural network in Java. There are many applications of self-organizing maps, most of them being in the field of clustering, data abstraction, and dimensionality reduction. But the clustering applications 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**:

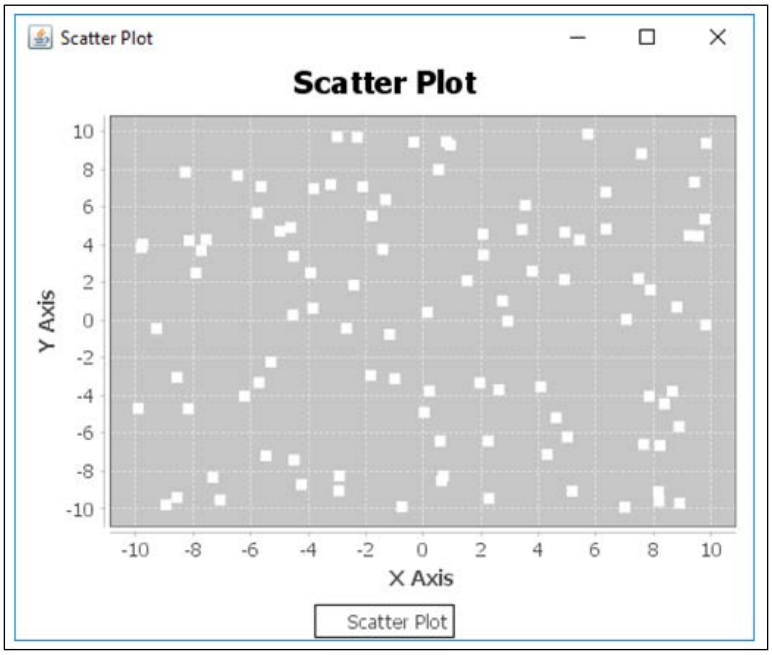
```
public class Chart {
  //title of the chart
  private String chartTitle;
  //datasets to be rendered in the chart
  private ArrayList<XYDataset> dataset = new ArrayList<XYDataset>();
  //the chart object
  private JFreeChart jfChart;
  //colors of each dataseries
  private ArrayList<Paint> seriesColor = 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.setVisibile(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 (`show2DData`) to determine whether the plot is going to be shown for every epoch:

```
protected ChartFrame plot2DData;

public boolean show2DData=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){}
}
```

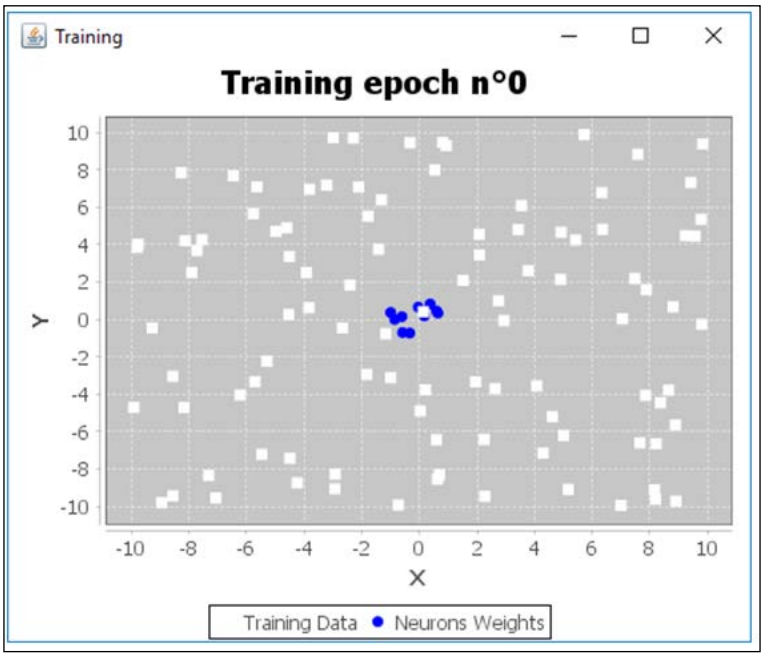
Testing Kohonen learning

Let's now define a Kohonen network and see how it works. First, we're creating a Kohonen with zero dimension:

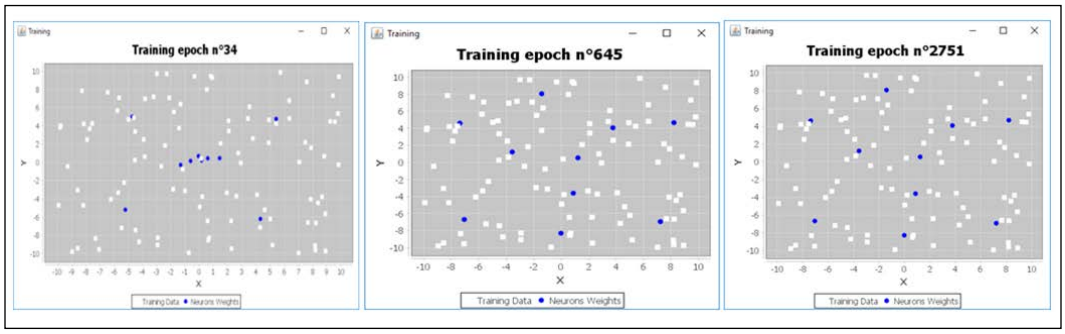
```
RandomNumberGenerator.seed=0;
int numberOfInputs=2;
int numberOfNeurons=10;
```

```
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.show2DData=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:

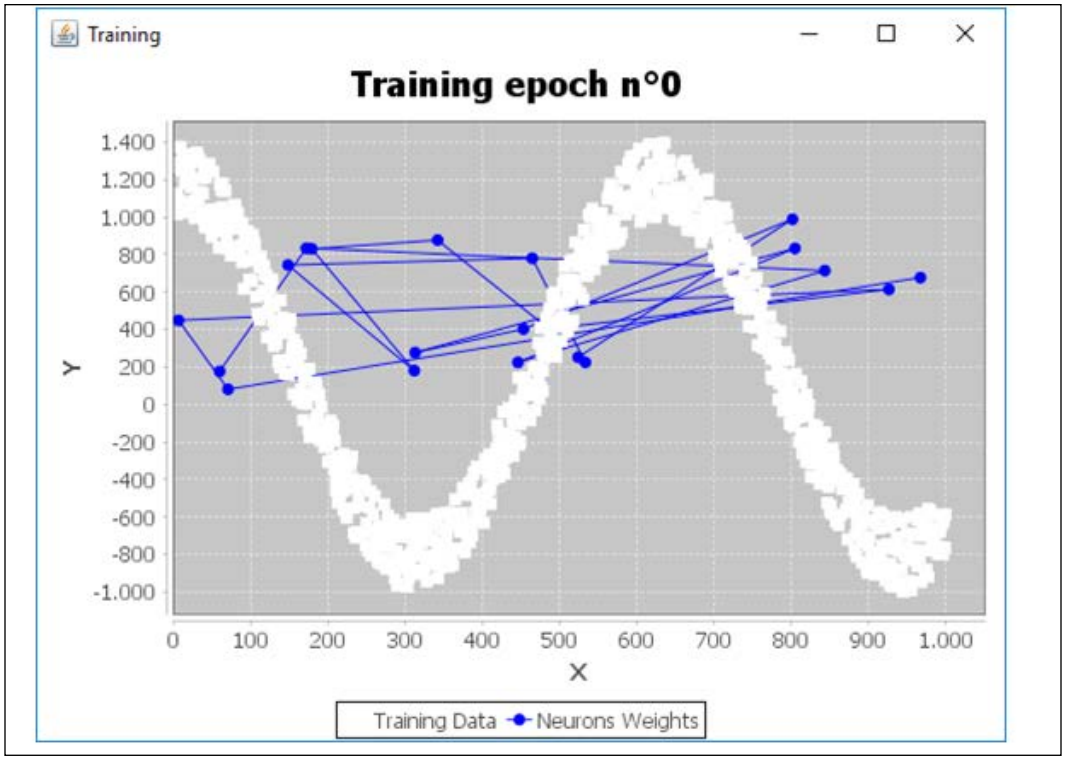


For one dimension, let's try something funkier. Let's create the dataset over a cosine function with random noise:

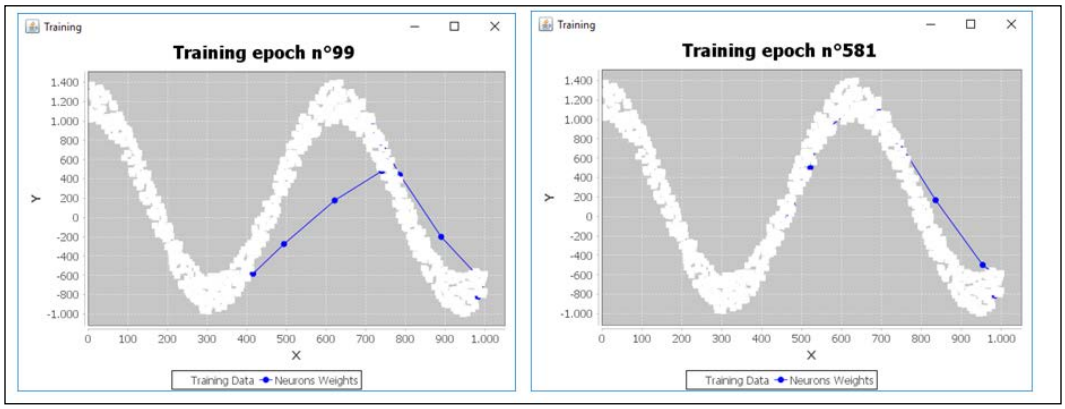
```
int numberOfPoints=1000;
int numberOfInputs=2;
int numberOfNeurons=20;
```

```
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.

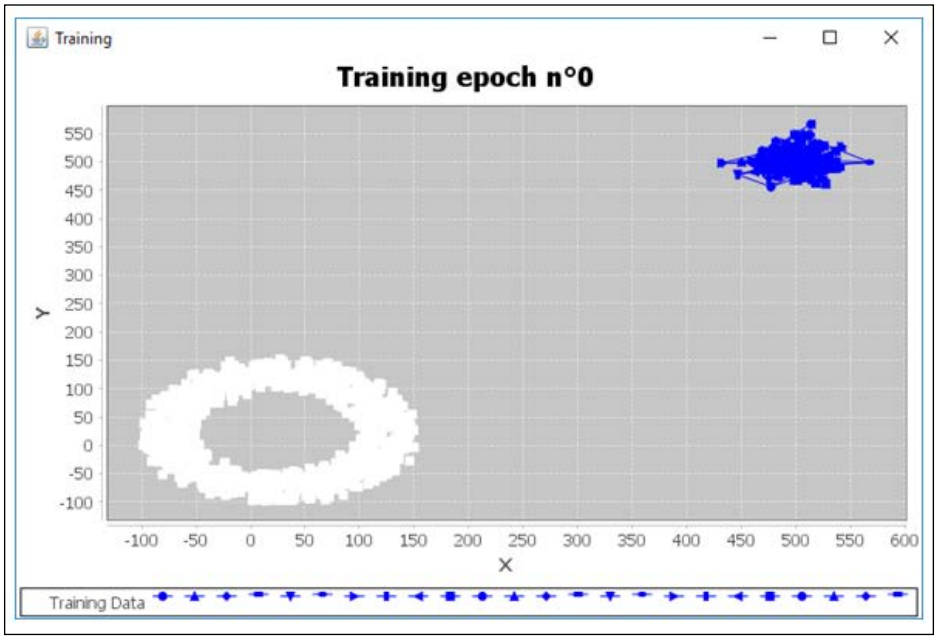
The dataset used here will be a circle with random noise added:

```
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;
}
```

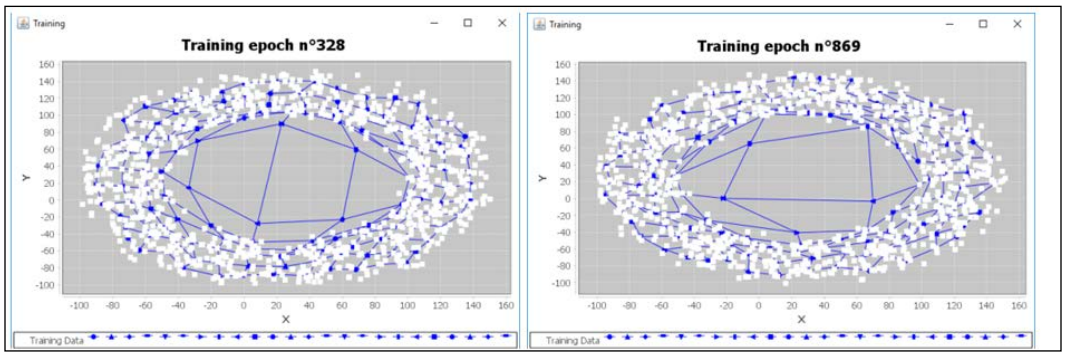
Now let's construct the two-dimensional Kohonen:

```
int numberOfInputs=2;
int neuronsGridX=12;
int neuronsGridY=12;
Kohonen kn2 = new Kohonen(numberOfInputs,neuronsGridX,neuronsGridY,new
GaussianInitialization(500.0,20.0));
```

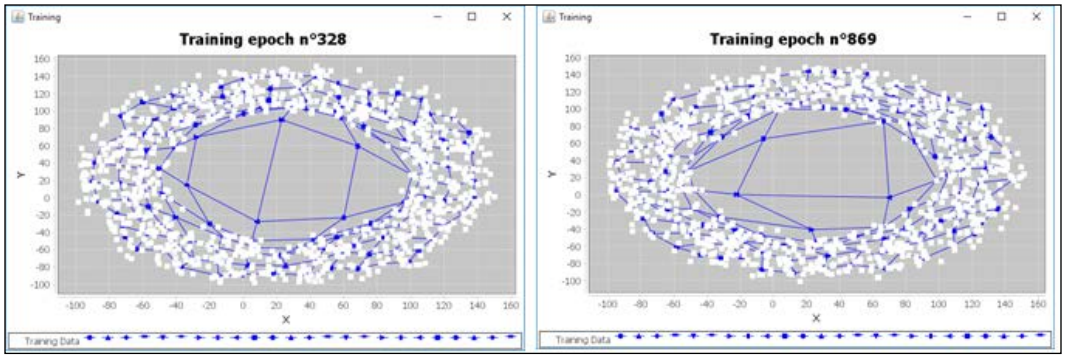
Note that we are using `GaussianInitialization` with mean 500.0 and standard deviation 20.0, that is, the weights will be generated at the position (500.0,500.0) while the data is centered around (50.0,50.0):



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

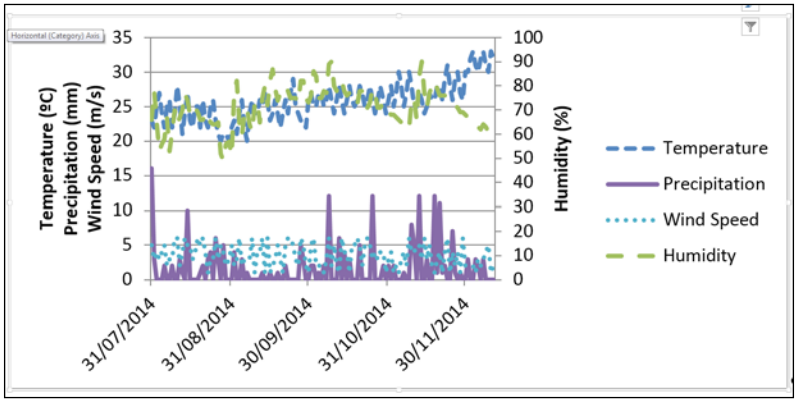
Neural networks for regression problems

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 ºC	880 mbar	66%	16 mm	5 m/s
August 1st	22 ºC	881 mbar	78%	3 mm	3 m/s
August 2nd	25 ºC	884 mbar	65%	0 mm	4 m/s
August 3rd	27 ºC	882 mbar	53%	0 mm	3 m/s
…					
December 11th	32 ºC	890 mbar	64%	0 mm	2 m/s

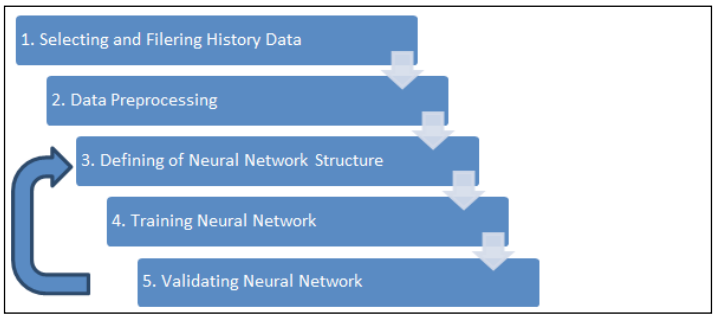
The table above depicts five variables containing hypothetical values of weather data collected from a hypothetical city, only for the purpose of this example. Now let's suppose that each of the variables contains a list of values sequentially taken over time. We can think of each list as a time series. 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 the 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 fullFilePath;
  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 numberOfColumns;
  public int numberOfRecords;

  //creating from Java matrix
  public DataSet(double[][] _data,String[] _columns){
    numberOfRecords=_data.length;
    numberOfColumns=_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);
    numberOfRecords=_data.length;
    numberOfColumns=_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 of 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:

Date (time column)	Temp
42429(29/02/2016)	27.5
42430(01/03/2016)	26.3
42431(02/03/2016)	25.4
42433(04/03/2016)	26.6

Transformed DataSet:

Date (time column)	Temp	Temp__1	Temp__2
42429(29/02/2016)	27.5	NaN	NaN
42430(01/03/2016)	26.3	27.5	NaN
42431(02/03/2016)	25.4	26.3	27.5
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:

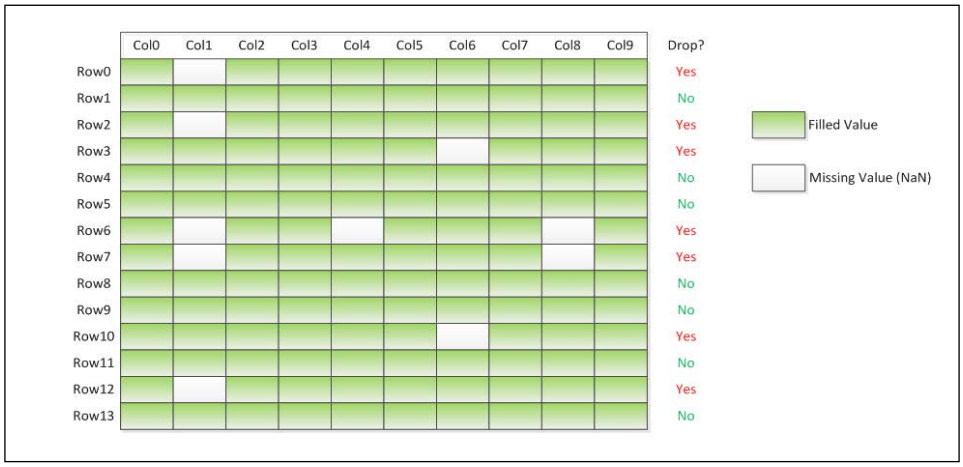
- There may be missing values or no measurements on a specific point of time; this can generate NaNs in the Java matrix.
- Shifting a column over one time period, for example, is not the same as getting the value of the previous row. That's why it is important to choose one column to be the time reference.

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:




```
// 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 (ºC)
- Humidity (%)
- Pressure (mbar)
- Wind speed (m/s)
- Wind direction (º)
- 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

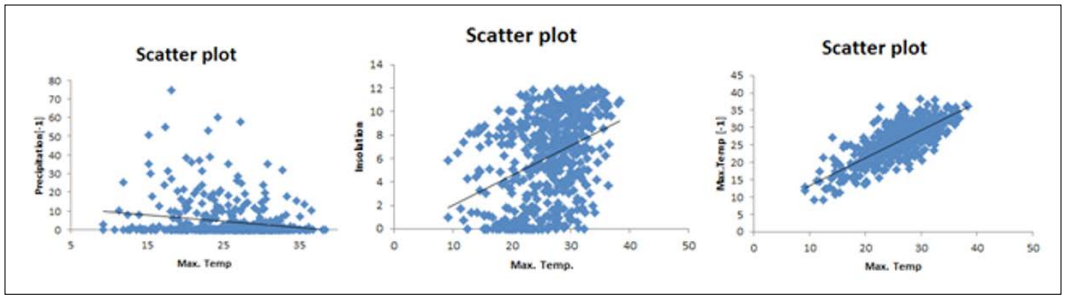
Selecting the appropriate data that fulfils most of the system's dynamics needs to be carefully done. We want the neural network to forecast future weather based on the current and past weather data, but which variables should we choose? Getting an expert opinion on the subject can be really helpful 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:

$$C_{x,y} = \frac{E[X.Y]E[X]E[Y]}{\sigma_x \sigma_y}$$

Here, *E[X.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; *σX* and *σY* are the standard deviation of *X* and *Y* respectively; and finally *Cx,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.

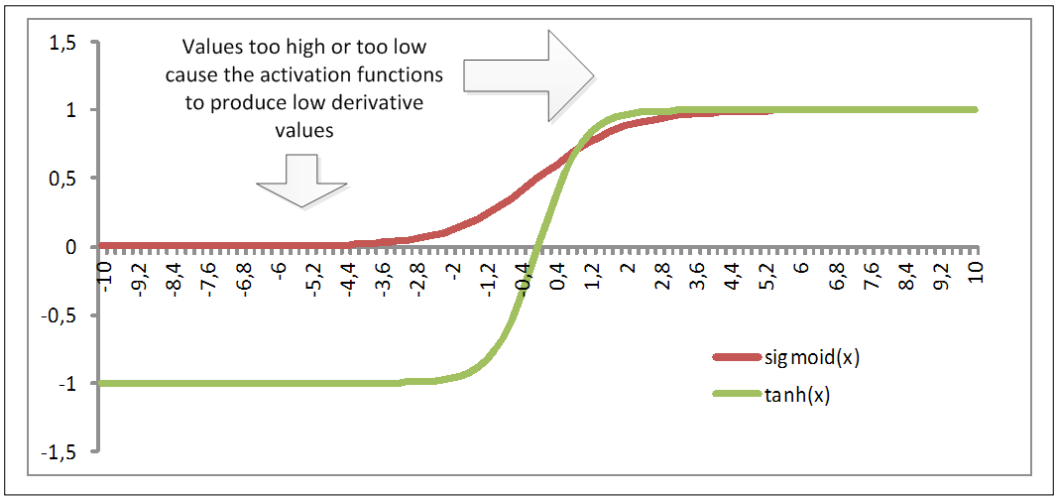
Regarding weather variables, let's take a look at their range, sampling, and type:

Variable	Unit	Range	Sampling	Type
Mean temperature	^{o} 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 in to 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:

$$X_{norm} = \left(N_{\max} - N_{\min}\right)\left[\frac{\left(X - X_{\min}\right)}{\left(X_{\max} - X_{min}\right)}\right] + N_{\min}$$

Here, N_{min} and N_{max} are the normalized minimum and maximum limits respectively, X_{min} and Xmax 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:

$$X_{norm} = \frac{\left(X - X_{\min}\right)}{\left(X_{\max} - X_{min}\right)}$$

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:

$$X = \left(X_{\max} - X_{\min}\right)\left[\frac{\left(X_{norm} - N_{\min}\right)}{\left(N_{\max} - N_{\min}\right)}\right] + X_{\min}$$

Or

$$X = \left(X_{\max} - X_{\min}\right)\left[X_{norm}\right] + X_{\min}$$

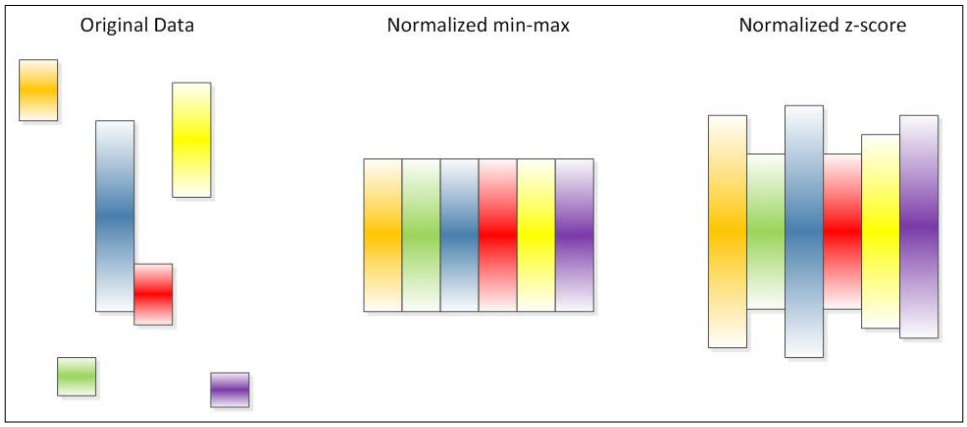
For the normalization between 0 and 1.

Another mode of normalization is the z-score, which takes into account the mean and standard deviation:

$$X_{norm} = S\left[\frac{X - E\left[X\right]}{\sigma_X}\right]$$

Here, S is a scaling constant, *E[X]* is the mean of E, and *σX* 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[] minValues;
  private double[] maxValues;
  private double[] meanValues;
  private double[] stdValues;
  //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	7º37'S	72º40'W	170 m	Tropical Rainforest
2	Picos	7º04'S	41º28'W	208 m	Semi-arid
3	Campos do Jordão	22º45'S	45º36'W	1642 m	Subtropical Highland
4	Porto Alegre	30º01'S	51º13'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.

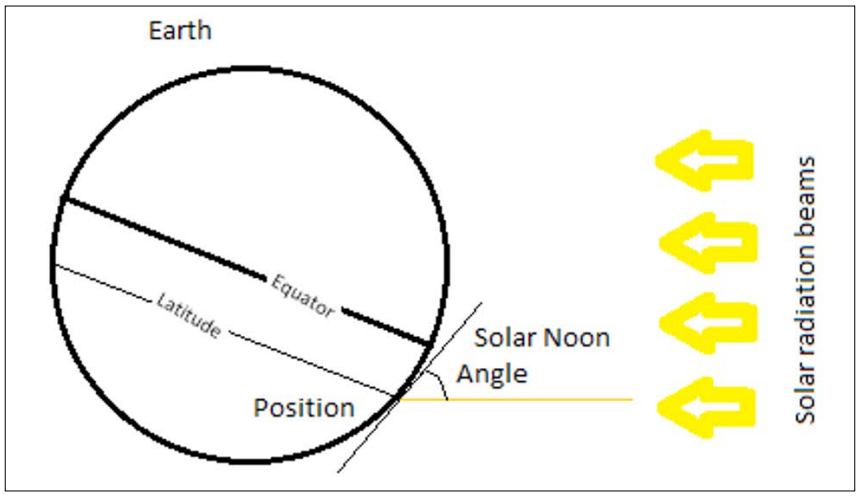
The data collected from the INMET website includes these variables:

- Precipitation (mm)
- Max. temperature (ºC)
- Min. temperature (ºC)
- Insolation (sunny hours)
- Evaporation (mm)
- Avg. temperature (ºC)
- 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 cruzeirodosul;
    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:

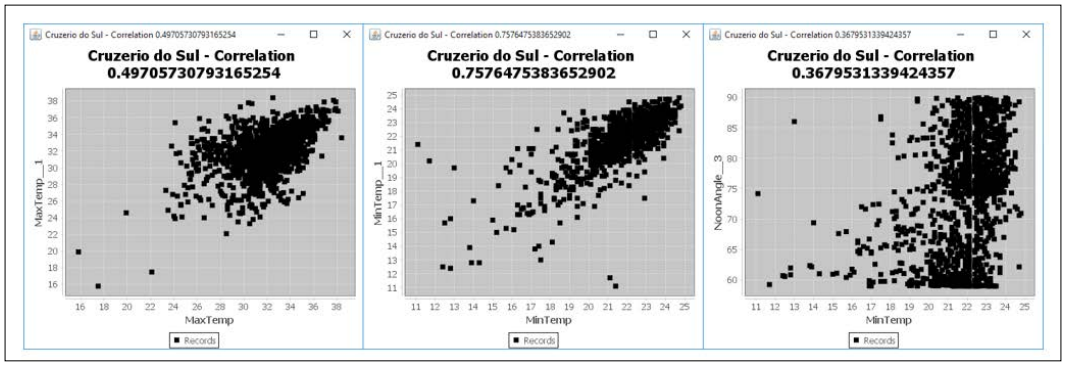
```
public void correlationAnalysis(double minAbsCorr){
  //indexes of output variables (max. and min. temperature)
  int[][] outputs = {
            {2,3}, //cruzeiro do sul
            {2,3}, //picos
            {2,3}, //campos do jordao
            {2,3}}; //porto alegre
  int[][] potentialInputs = { //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: MinTemp
Correlations with the output Variable: 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

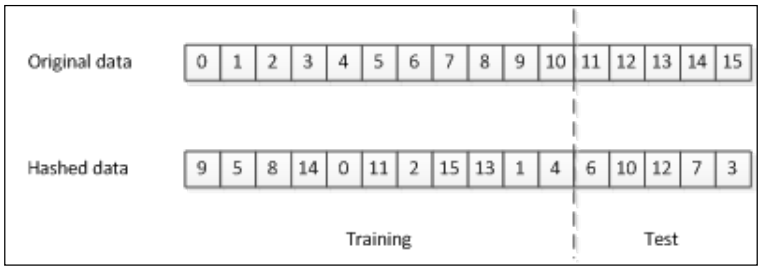
We are using four neural networks to forecast the minimum and maximum temperature. Initially, they will have two hidden layers with 20 and 10 neurons each and hypertan and sigmoid activation functions. We will apply min-max normalization. The following method in the class `WeatherExample` creates the 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`:

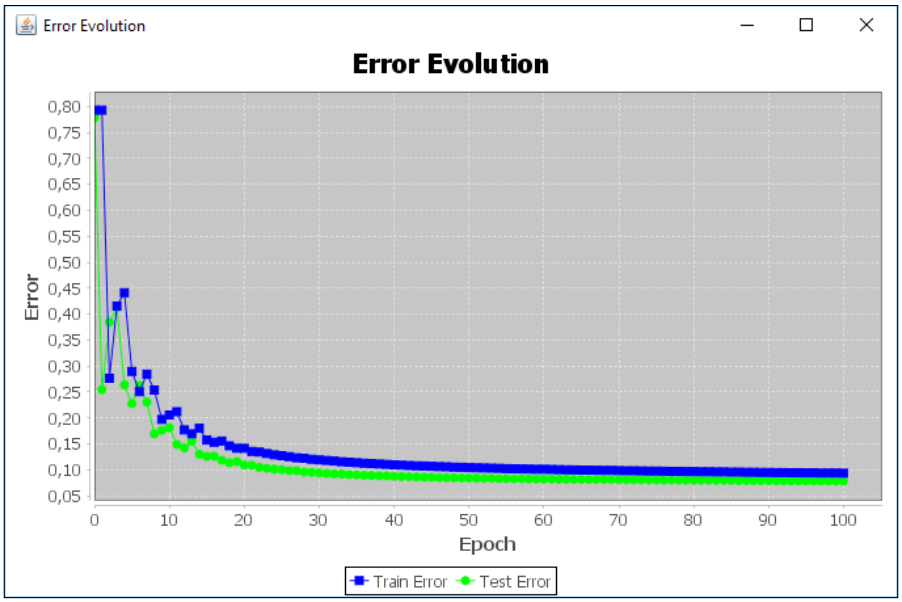
```
Backpropagation bpCS = new Backpropagation(we.nncruzeirosul
               ,we.trainDataCS
               ,LearningAlgorithm.LearningMode.BATCH);
bpCS.setTestingDataSet(we.testDataCS);
bpCS.setLearningRate(0.3);
bpCS.setMaxEpochs(1000);
bpCS.setMinOverallError(0.01); //normalized error
bpCS.printTraining = 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 `LearningAlogrithm` 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

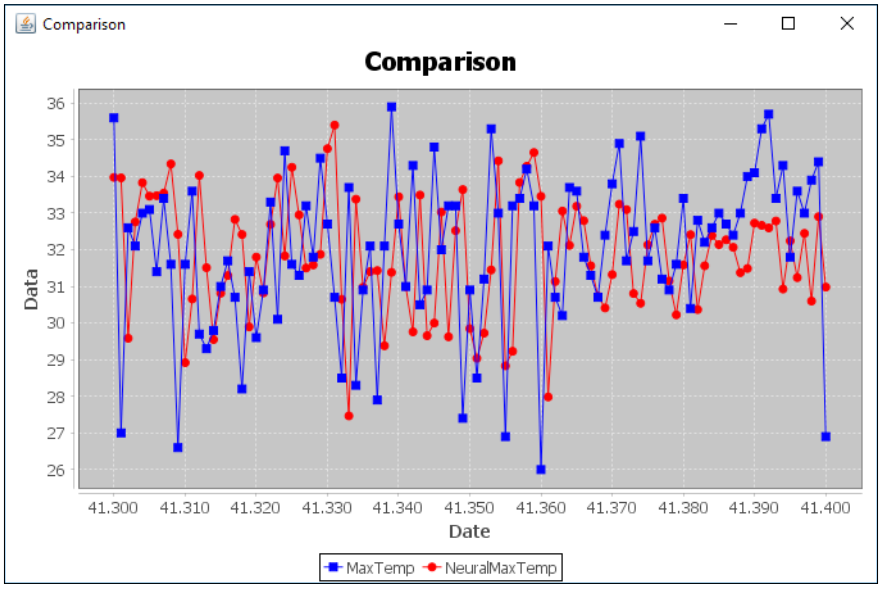
Using this same facility, it is easy to see and compare the neural network output. First, let's transform the neural network output into vector form and add to our dataset using the method `addColumn`. Let's name it `NeuralMinTemp` and `NeuralMaxTemp`:

```
 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

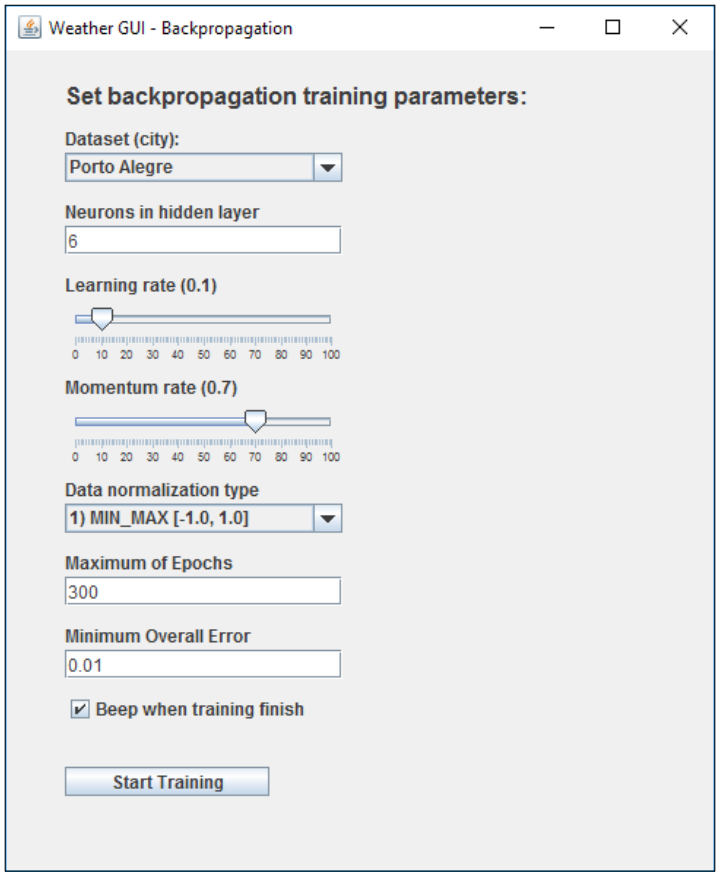
Results and simulations

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:

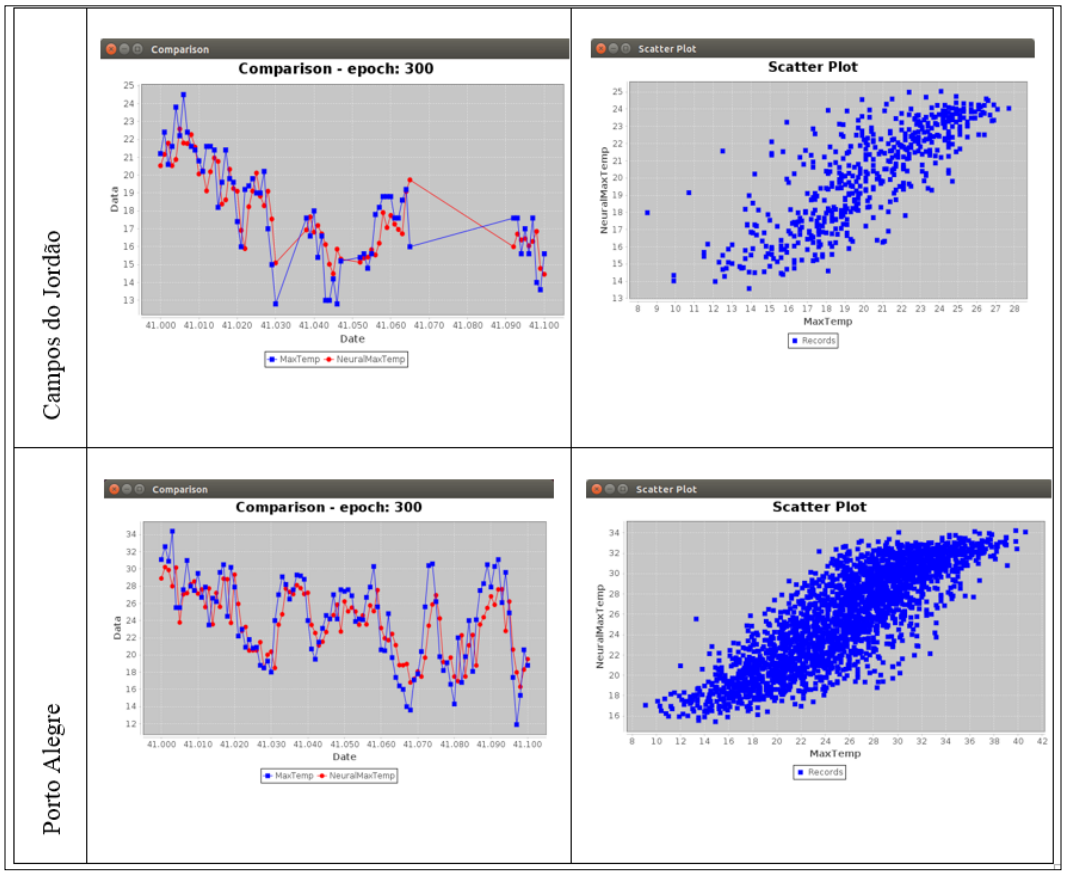


After running 12 experiments, we found the following MSE training errors for each dataset:

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.

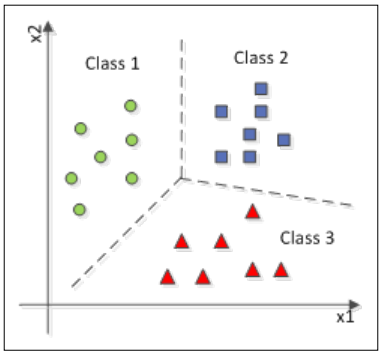
6
Classifying Disease Diagnosis

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

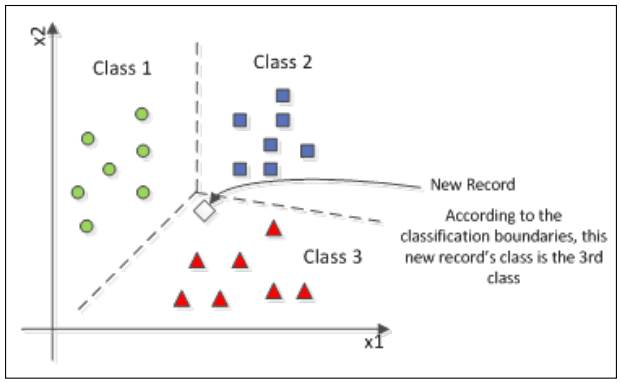
Foundations of classification problems

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 tasks falls in the supervised learning category.

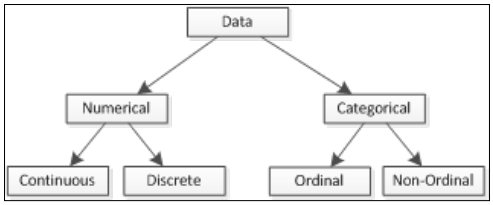
A classification algorithm seeks to find the boundaries between the classes in the data hyperspace. Once the classification boundaries are defined, a new data point, with an unknown class, receives a class label according to the boundaries defined by the classification algorithm. The figure below shows 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:



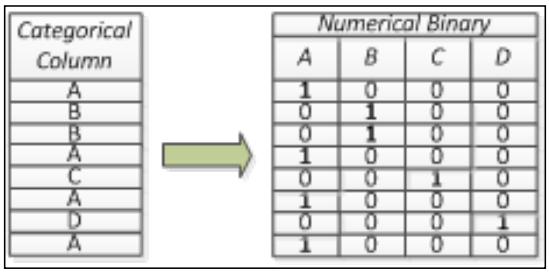
Data can be numerical or categorical or, simply speaking, numbers or words. Numerical data is represented by a numeric value, from which it can be continuous or discrete. This data type has been used so far in this book's applications. Categorical data is a wider class of data that includes words, letters, 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:



Categorical Column
A
B
B
A
C
A
D
A

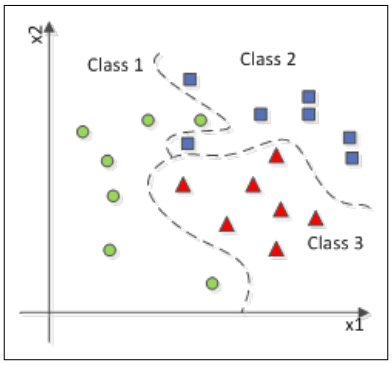
Numerical Binary			
A	B	C	D
1	0	0	0
0	**1**	0	0
0	**1**	0	0
1	0	0	0
0	0	**1**	0
1	0	0	0
0	0	0	**1**
1	0	0	0

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

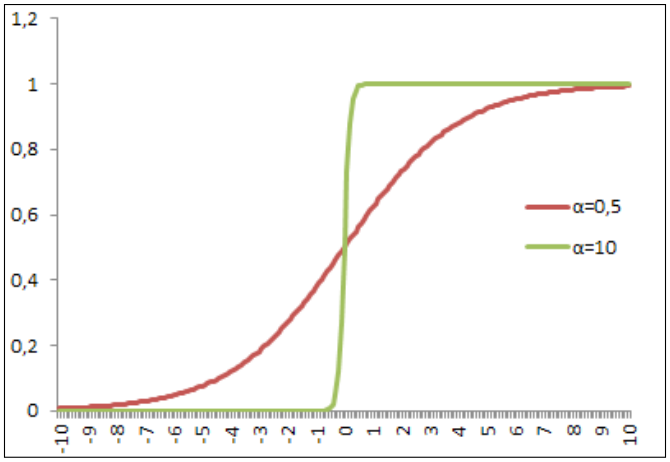
We've covered that Neural Networks can work as data classifiers by establishing decision boundaries onto data in the hyperspace. This boundary can be linear, in the case of perceptrons, or nonlinear, in the case of other neural architectures such as MLPs, Kohonen, or Adaline. The linear case is based 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:

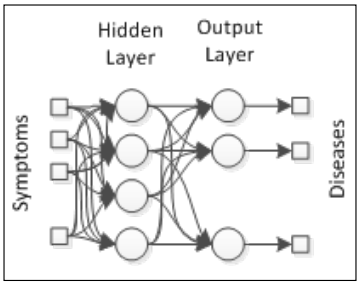


Note that the higher the alpha parameter is, the more the logistic function takes a shape of a hard-limiting threshold function, also known as a step function.

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							
	A	B	C	D	E	F	G	Total
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:

$$Sensitivity = \frac{Number\ of\ True\ Positives}{Total\ of\ Actual\ Positive\ Records}$$

Specificity, in turn, means the true negative rate; it indicates the proportion of negative record identification:

$$Specificity = \frac{Number\ of\ True\ Negatives}{Total\ of\ Actual\ Negative\ Records}$$

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: "+(sensibility*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

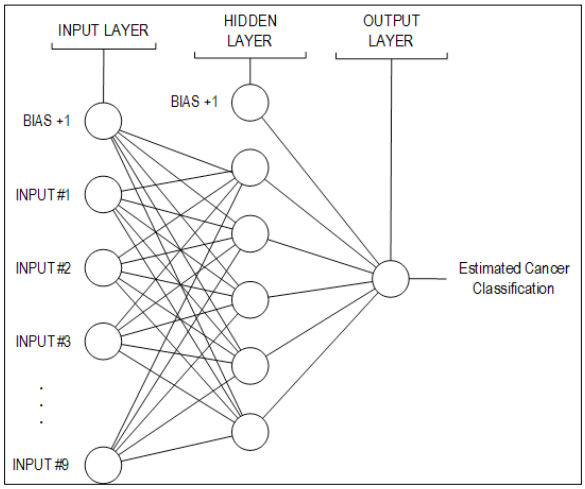
The breast cancer dataset is composed of 10 variables, of which nine are inputs and one is a binary output. The dataset has 699 records, but we excluded from them 16 which were found to be incomplete, thus we used 683 to train and test the neural network.

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%)

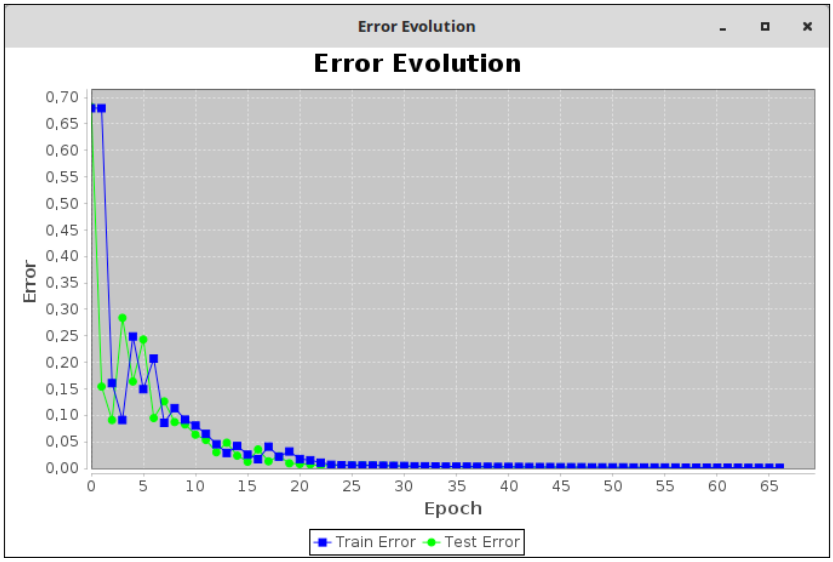
As in the previous cases, we performed many experiments to try to find the best neural network to classify whether cancer is benign or malignant. So we conducted 12 different experiments (1,000 epochs per experiment), wherein MSE and accuracy values were analyzed. After that, the confusion matrix, sensitivity, 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:

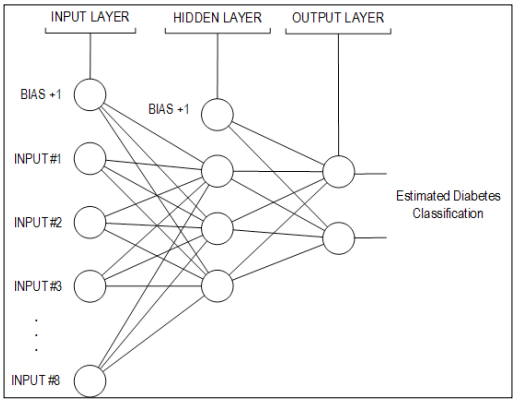
Variable Name	Type	Maximum Value and Minimum Value
Diagnosis result	OUTPUT	[0; 1]
Number of times pregnant	INPUT #1	[0.0; 17]
Plasma glucose concentration a 2 hours in an oral glucose tolerance test	INPUT #2	[0.0; 199]
Diastolic blood pressure (mm Hg)	INPUT #3	[0.0; 122]
Triceps skin fold thickness (mm)	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.

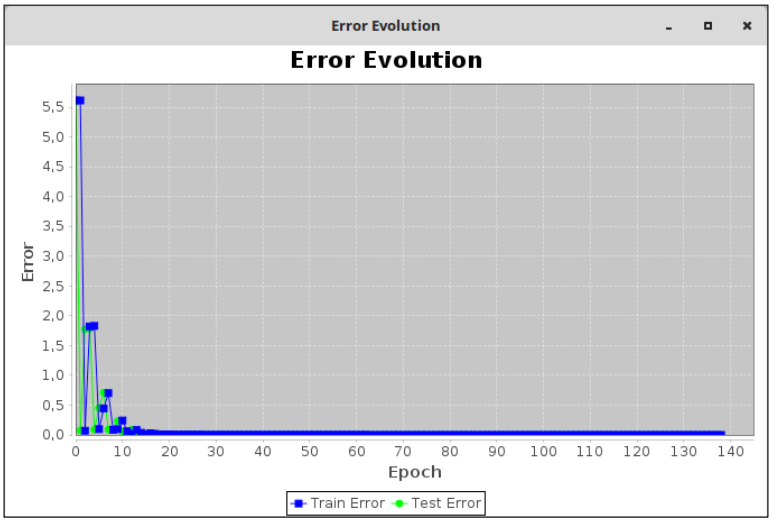
So, the proposed neural architecture looks like that of the following figure:



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%

One more time, we suggest choosing the simplest model. In the diabetes example, it is the artificial neural network generated by experiment #3.

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:

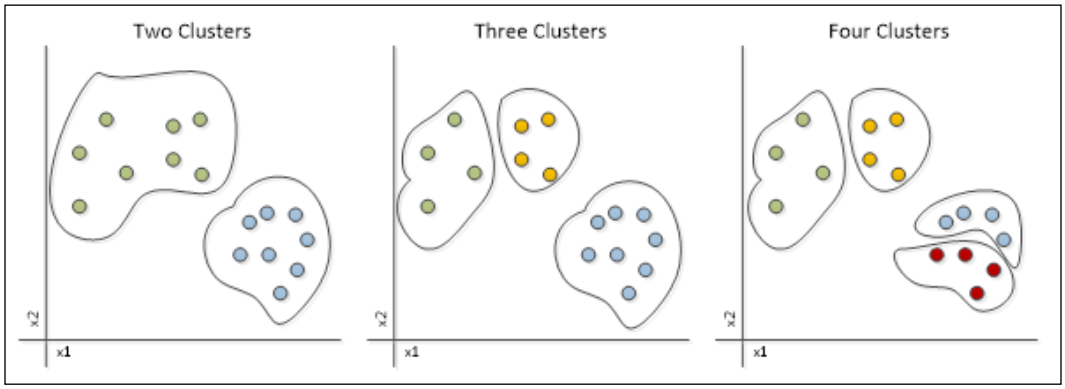
- Clustering tasks
- Cluster analysis
- Cluster evaluation
- Applied unsupervised learning
- Radial basis functions neural network
- Kohonen network for clustering
- Handling with types of data
- Customer profiling
- Preprocessing
- Implementation in Java
- Credit analysis and profiles of customers

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:



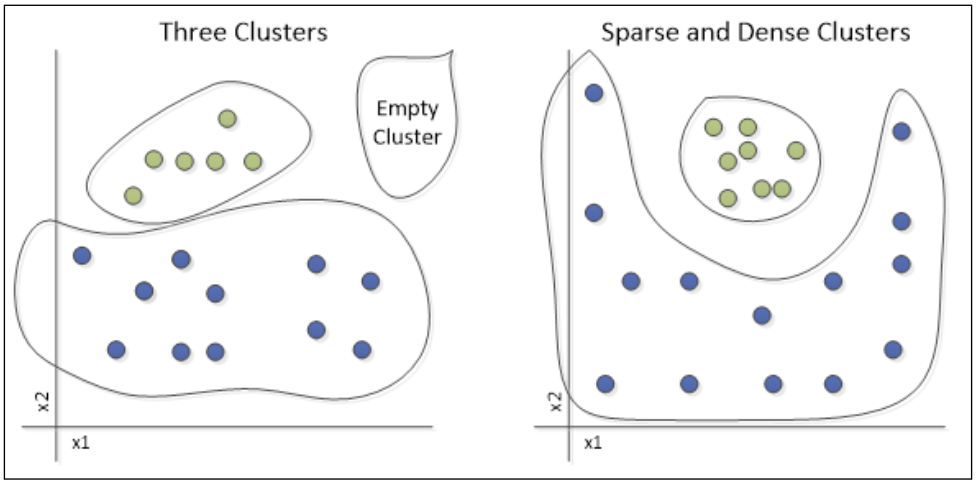
In clustering, the number of clusters is not determined by the data, but by the data analyst who is looking to cluster the data. Here the *boundaries* are a little bit different than those of classification tasks because they depend primarily on the number of clusters.

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:

$$DB = \frac{1}{n}\sum_{i=1}^{n}\max_{j\neq i}\left(\frac{\sigma_i + \sigma_j}{d\left(c_i, c_j\right)}\right)$$

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:

$$D = \frac{\min\limits_{1\leq i<j\leq n} d\left(i,j\right)}{\max\limits_{1\leq k\leq n} d'\left(k\right)}$$

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

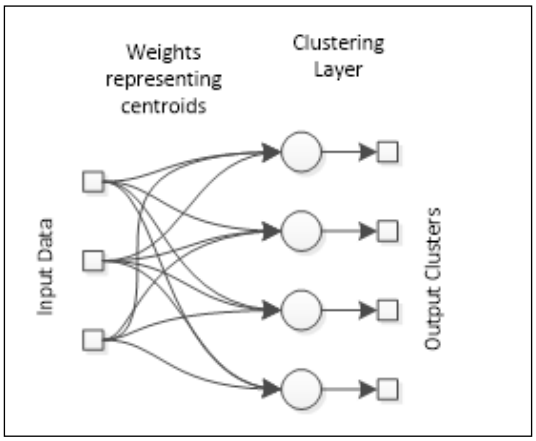
In neural networks, there are a number of architectures implementing unsupervised learning; however, the scope of this book will cover only the Kohonen neural network, developed in *Chapter 4*, *Self-Organizing Maps*.

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

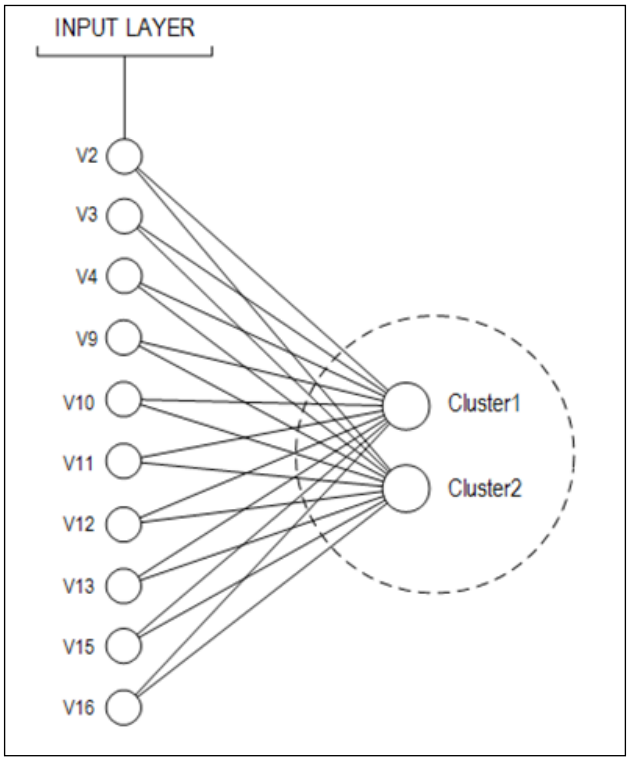
The card dataset is composed of 16 variables in total. 15 are inputs and one is output. For security reasons, all variable names have been changed to meaningless symbols. This dataset brings a good mix of variable types (continuous, categorical with small numbers of values, and categorical with a larger 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	[[14.0, 21.0] [18.0, 17.0]]	44.28%
#2	[[11.0, 24.0] [34.0, 1.0]]	17.14%
#3	[[21.0, 14.0] [17.0, 18.0]]	55.71%
#4	[[24.0, 11.0] [1.0, 34.0]]	82.85%
#5	[[21.0, 14.0] [17.0, 18.0]]	55.71%
#6	[[24.0, 11.0] [1.0, 34.0]]	82.85%
#7	[[8.0, 27.0] [7.0, 28.0]]	51.42%
#8	[[24.0, 11.0] [1.0, 34.0]]	82.85%
#9	[[27.0, 8.0] [28.0, 7.0]]	48.57%
#10	[[24.0, 11.0] [1.0, 34.0]]	82.85%

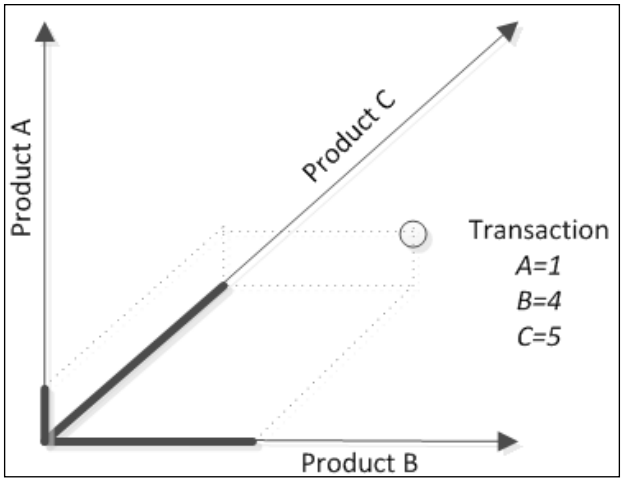
So, neural nets built by experiments #4, #6, #8, or #10 may be used to reach accuracy more than 80% to cluster customers financially.

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 by more than 20 items.

As recommend 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 others ways of analyzing the clustering result.

Summary

In this chapter, we've 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:

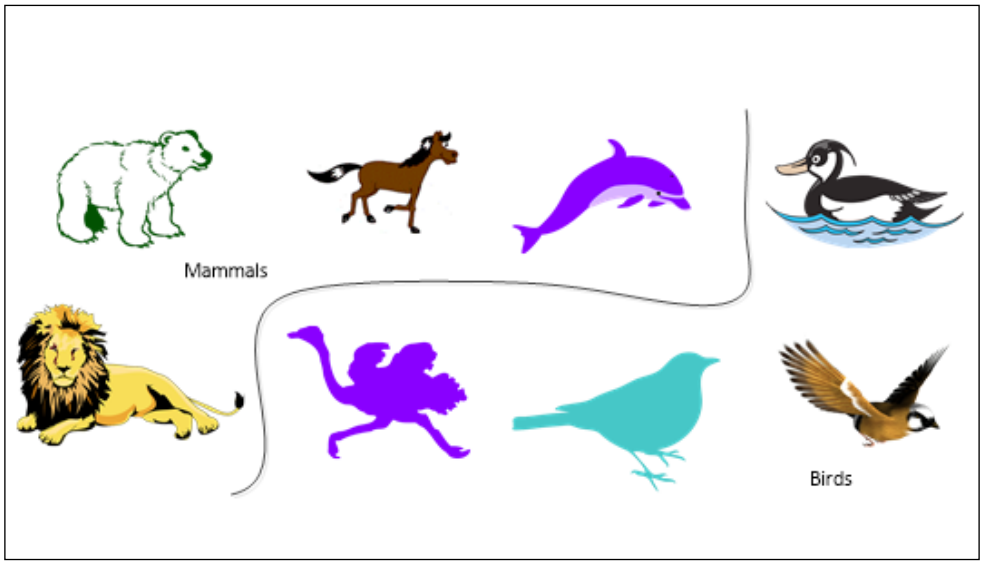
- Shape recognition
- Object classification
- Behavior clustering
- Voice recognition
- OCR
- Chemical reaction taxonomy

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.

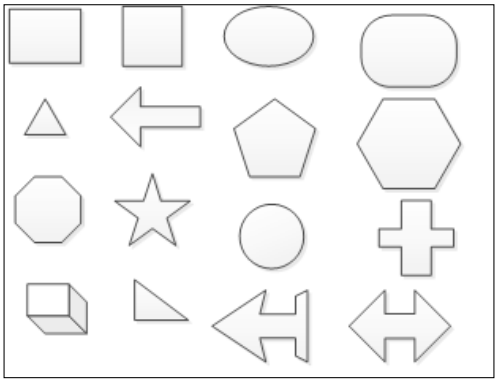
One illustrated example of pattern recognition using defined classes is animal recognition by image, shown in the figure below. The pattern recogniser, however, should be trained to catch all the characteristics that formally define the classes. In the example, eight figures of animals are shown, belonging 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.



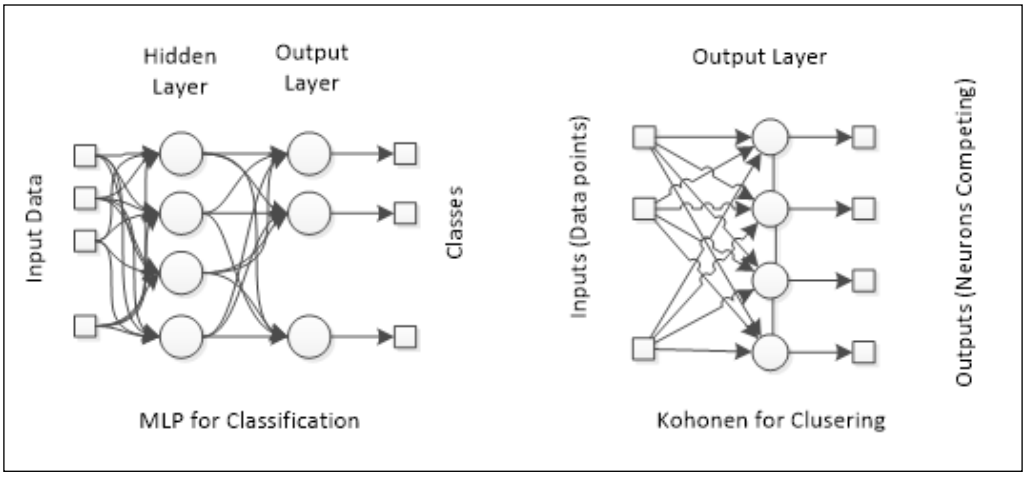
In the image above, we can see some sorts of shapes and we want to arrange them, so that similar ones can be grouped into the same cluster. Based on the shape information that is present in the images, it is likely that the pattern recognizer will classify the rectangle, the square and the, triangle 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

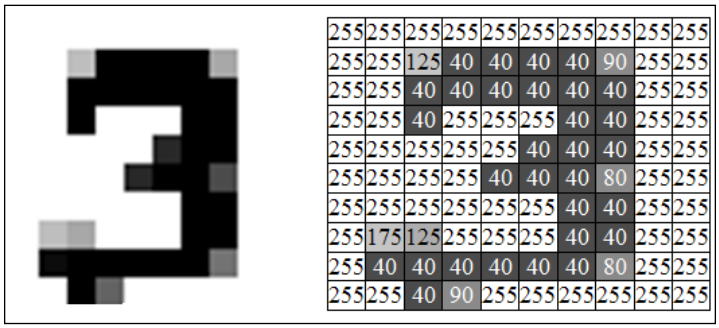
In spite of that, humans can easily interpret and read even texts produced in a bad quality image. This can be explained by the fact that humans are already familiar with text characters and the words in their language. Somehow the algorithm must become acquainted with these elements (characters, digits, 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 in 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	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	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.

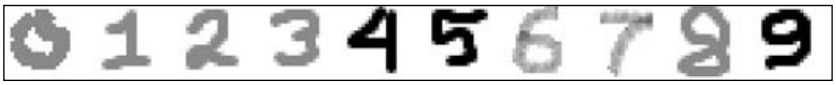
We apply this preprocessing in order to represent all ten digits in this application.

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.

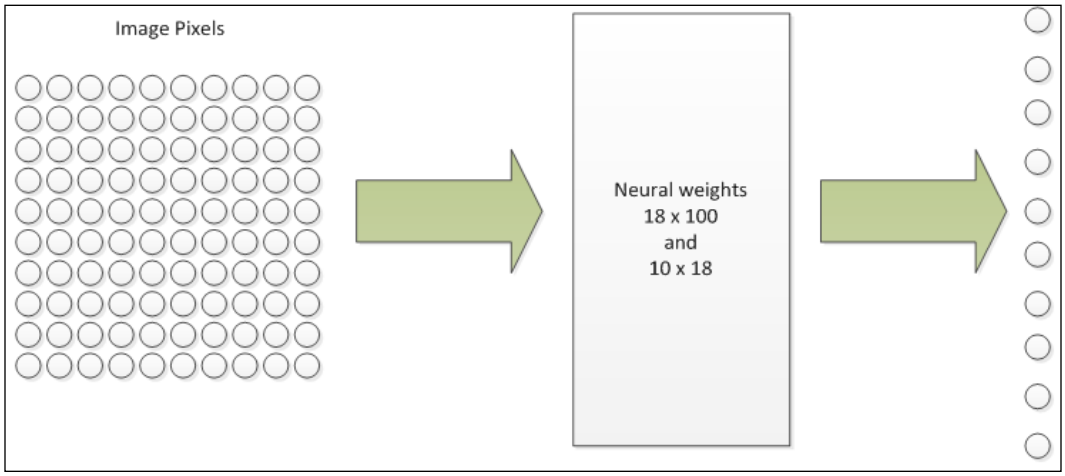
Each matrix row was merged into vectors (D_{train} and D_{test}) to form a pattern that will be used to train and test the neural network. Therefore, the input layer of the neural network will be composed of 101 neurons.

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)
```

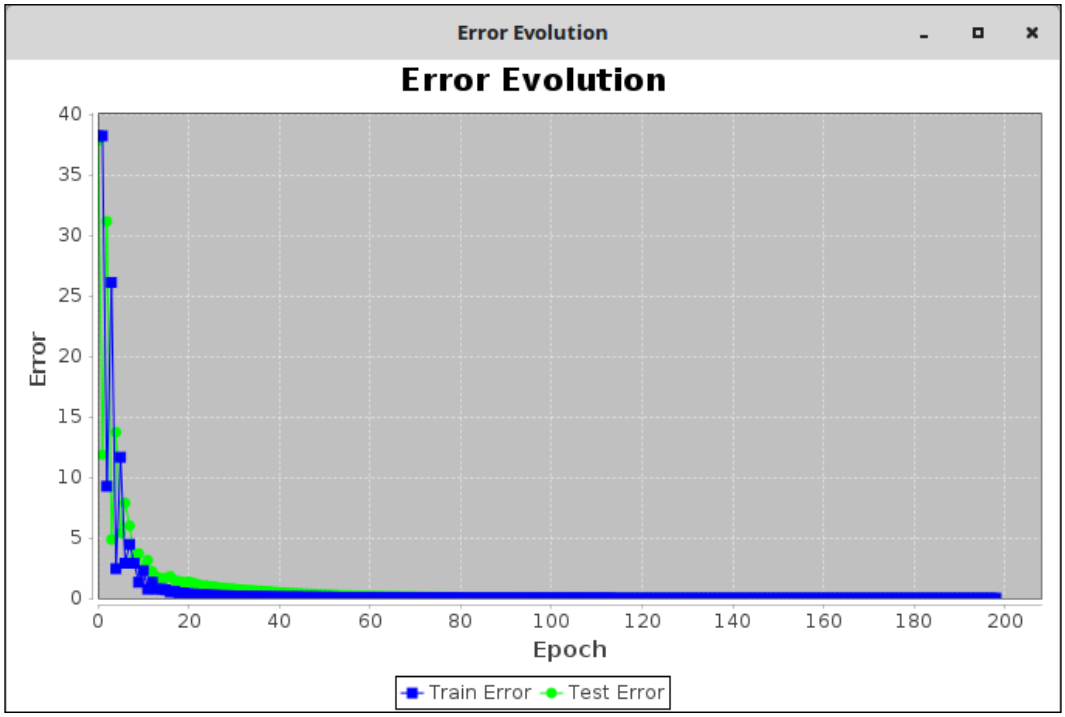
Results

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.

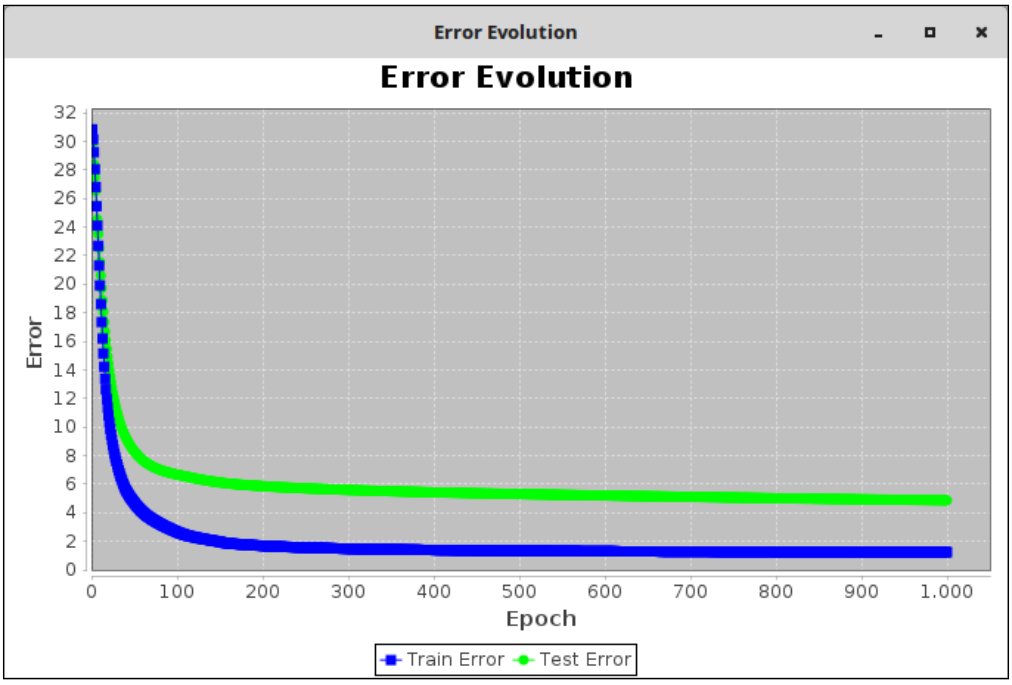
Experiment	Training overall error	Testing overall error	# Right number classifications
#1	9.99918E-4	0.01221	2 by 10
#2	**9.99384E-4**	**0.00140**	**5 by 10**
#3	9.85974E-4	0.00621	4 by 10
#4	9.83387E-4	0.02491	3 by 10

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:



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	0.10	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

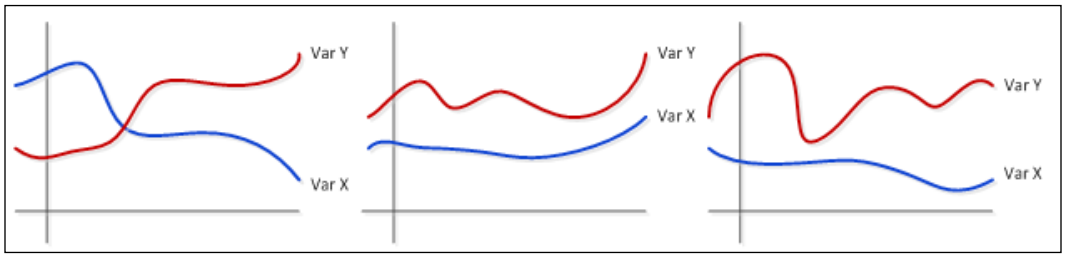
One of the key tasks in designing a neural network application is to select appropriate inputs. For the unsupervised case, one wishes to use only relevant variables on which the neural network will find the patterns. And for the supervised case, there is a need to map the outputs to the inputs, so one 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

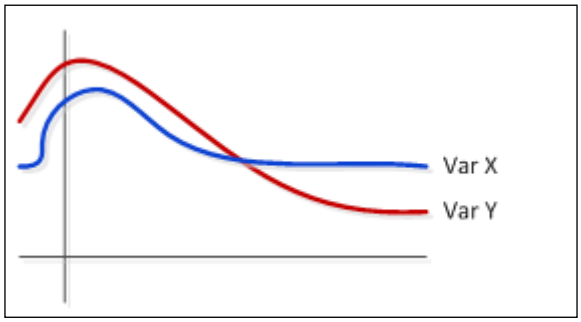
Linear correlation is very good in detecting behaviors between data series when they are presumably linear. However, if two data series form a parable when plotted together, linear correlation won't be able to identify any relation. That's why sometimes we need to transform data into a view that exhibits 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 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:

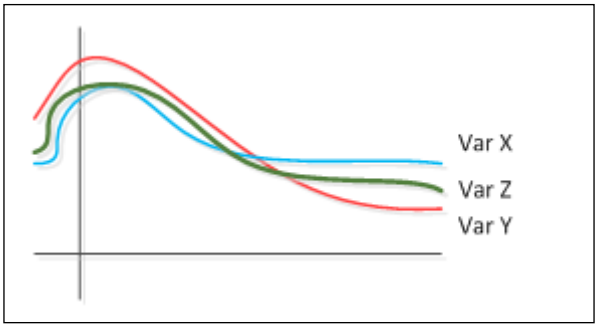
$$S = \begin{bmatrix} S_{XX} & S_{XY} \\ S_{YX} & S_{YY} \end{bmatrix}$$

Here, *SXY* is the covariance between the variables *X* and *Y*.

The derived new data will be then:

$$Z = eig(S)^T \left[X - E[X] Y - E[Y] \right]$$

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:

$$d_i = \left| \frac{X_i - E[X]}{std(X)} \right| \leq 3$$

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){
```

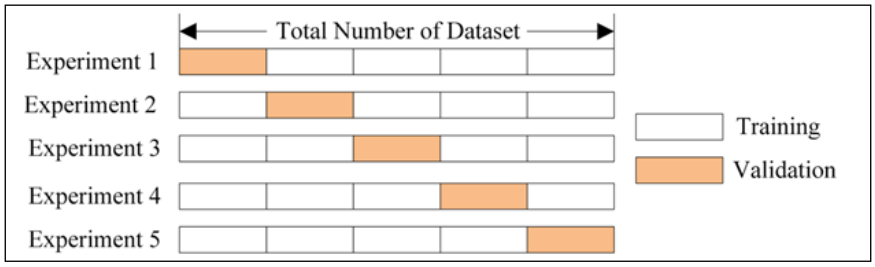
```
    this.originalDS=ds;
    this.thresholdDistance=threshold;
  }

  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:

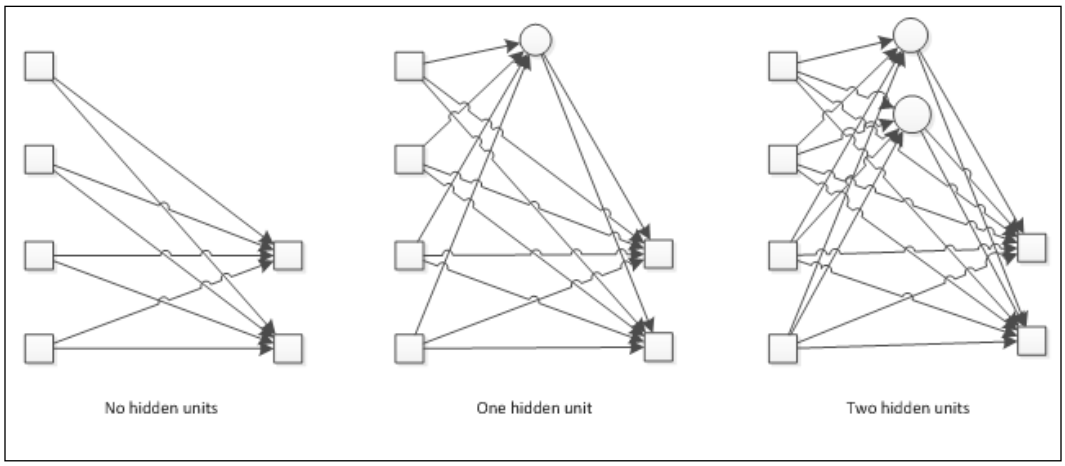
```
public class CrossValidation {
  NeuralDataSet dataSet;
  int numberOfFolds;

  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

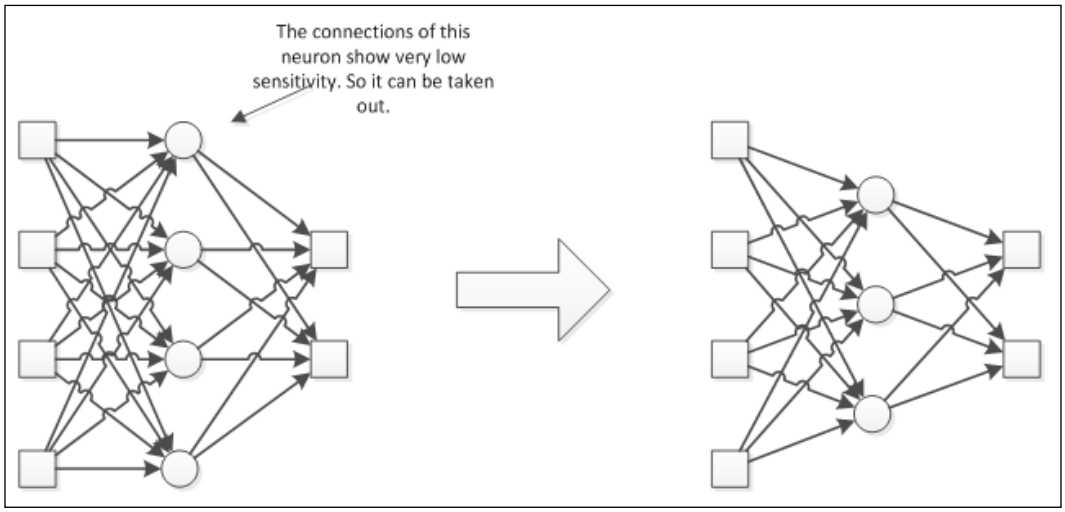
To choose an adequate structure for a neural network is also a very important step. However, this is often done empirically, since there is no rule on how many hidden units a neural network should have. The only measure of how many units are adequate is the neural network performance. One assumes that 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 senstitityThreshold;
}
```

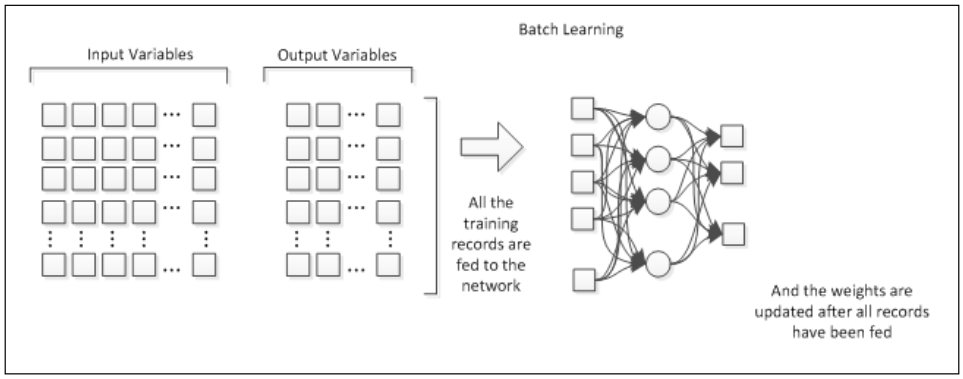
A method called `removeNeuron` in the class `NeuralLayer`, which actually sets all the connections of the neuron to zero, disables weight updating and fires only zero at the neuron`s output. This method is called if the property pruning of the `NeuralNet` object is set to true. The sensitivity calculation is according to the chain rule, as shown in *Chapter 3*, *Perceptrons and Supervised Learning* and implemented in the `calcNewWeigth` 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;
}
```

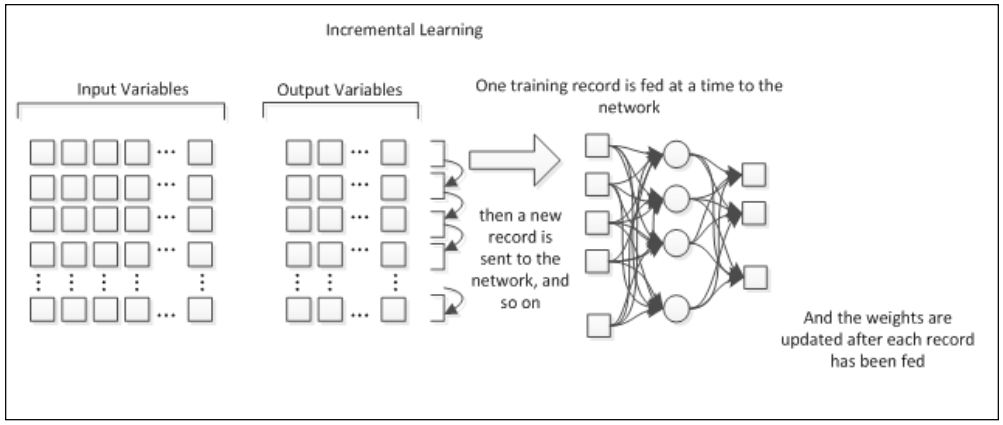
Online retraining

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 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 `BackpropagtionOnline` 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;
  }
```

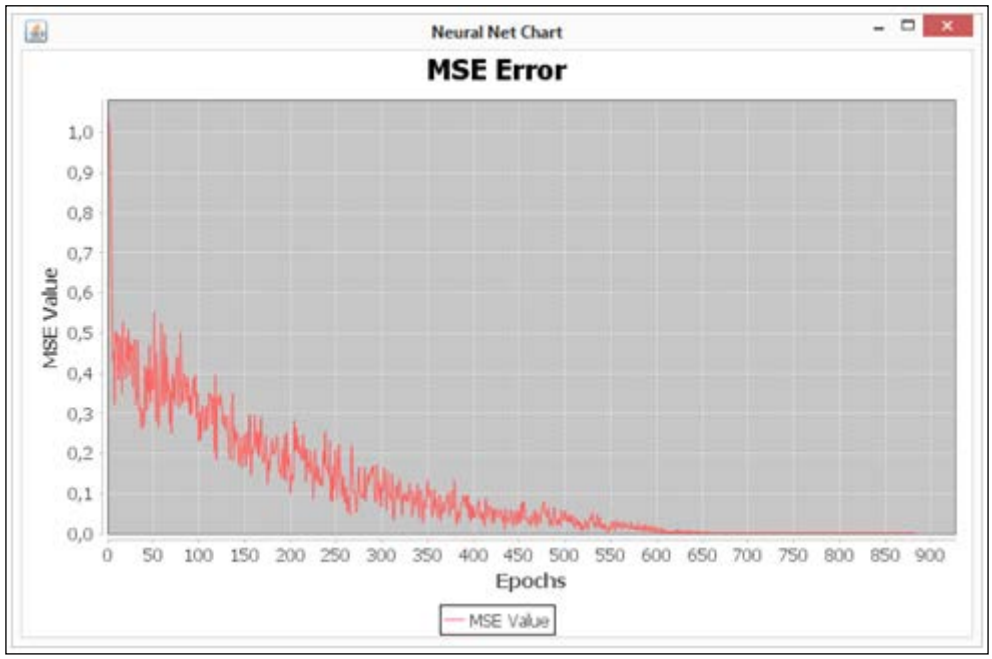
This method will be called at the end of the `train()` method.

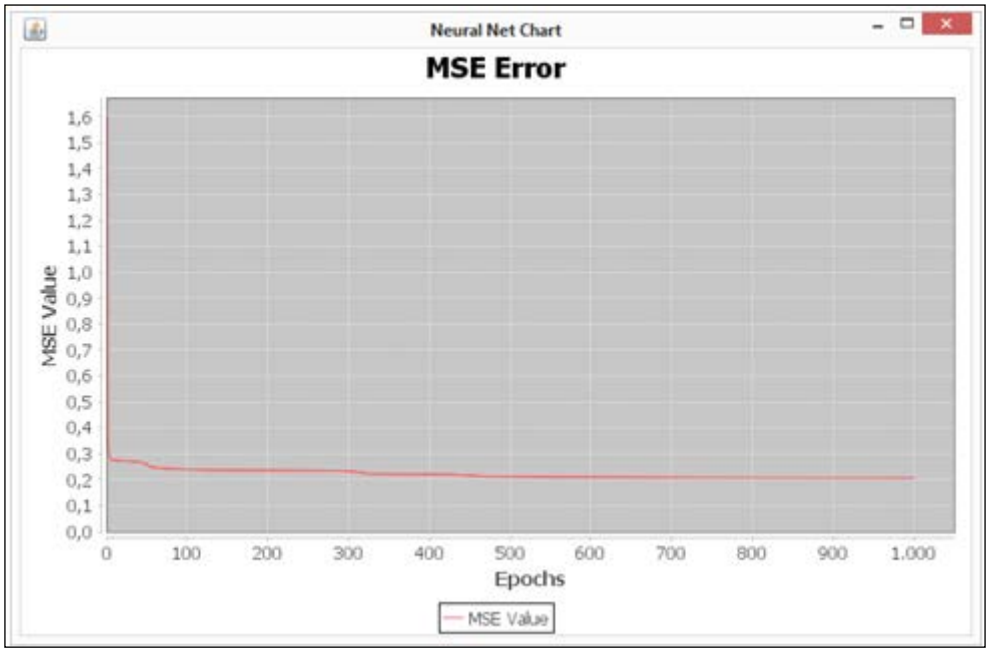
Application

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: ≅ 0.15	Found: ≅ 0.40
On-line Backpropagation MSE Value	0.4618623052	9.977909980E-6

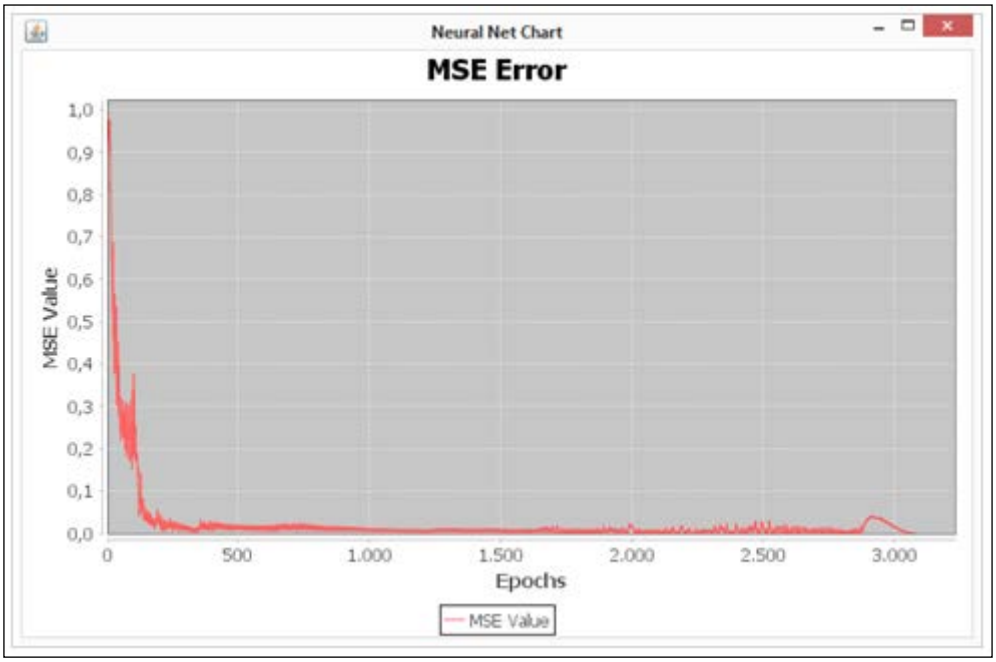
In addition, charts of the MSE evolution have been plotted and are shown here:





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

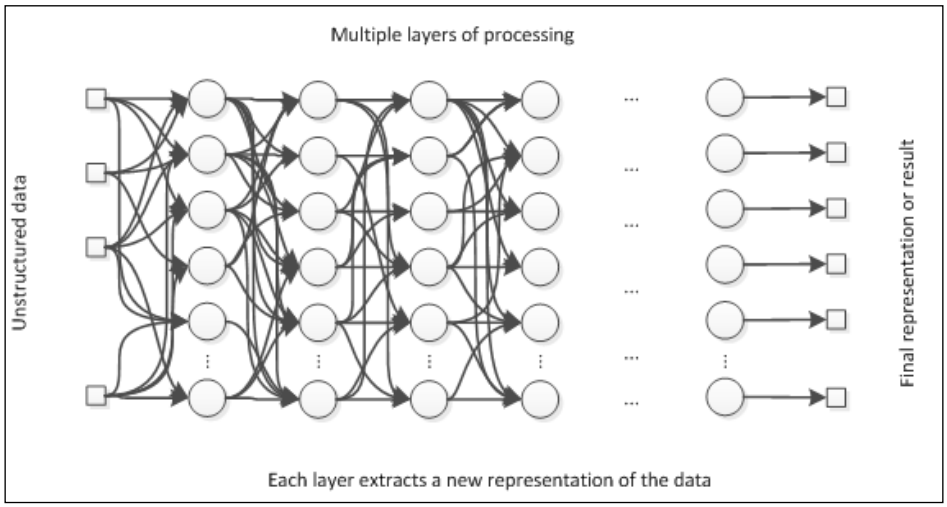
Deep learning

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:



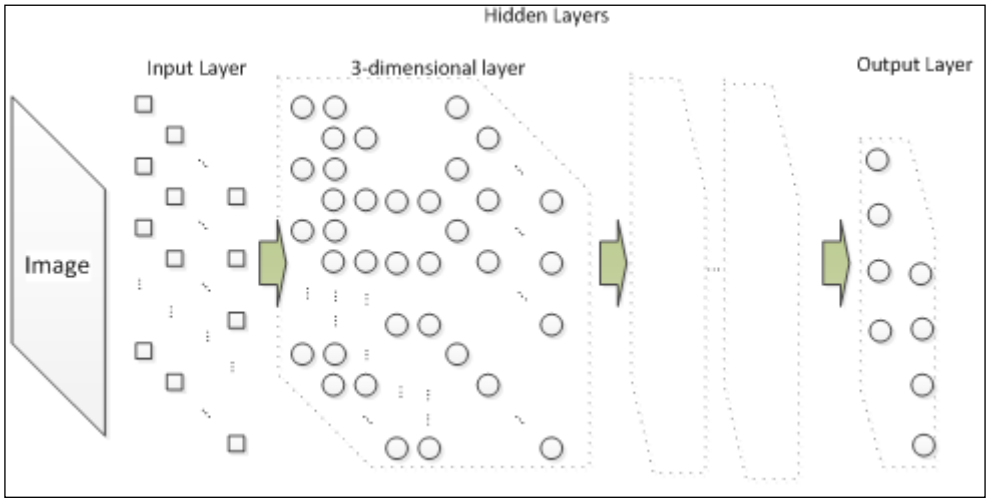
On the other hand, deep neural networks have some problems that need to be overcome. The main problem is overfitting. The many layers that produce new representations of data are very sensitive to the training data, because the deeper the signals reach in the neural layers, the more specific the transformation 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:

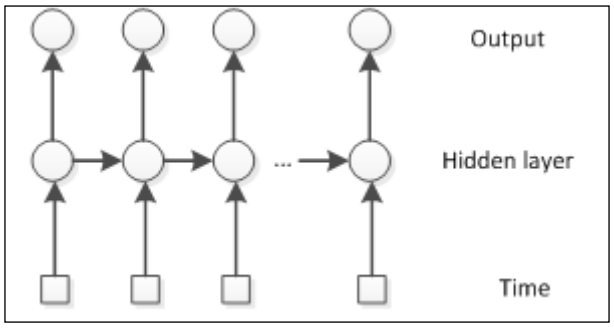
Convolutional neural network

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.

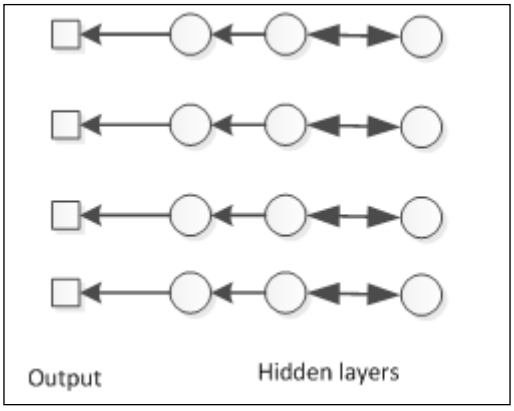


In this architecture, the layers may have multidimensional organization. Inspired by the visual cortex of animals, the typical dimensionality applied to the layers is three-dimensional. In **convolutional neural networks** (**CNNs**), part of the signals of a preceding layer is fed into another part 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.

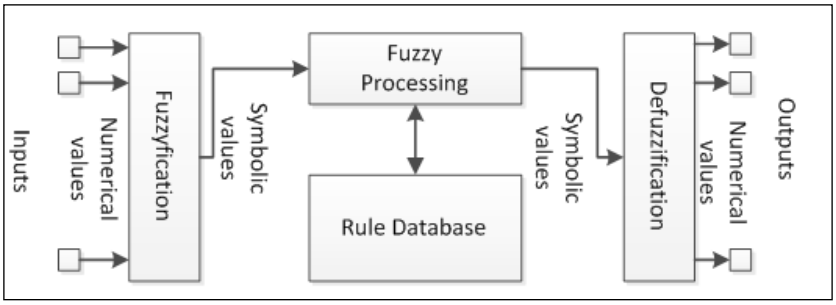
One motivation for using hybrid neural systems is common to one of the foundations of deep learning, which is the feature extraction. Tasks like image recognition become very tough to deal with when resolution is very high; however, if that data can be compacted or reduced, the processing becomes much 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.

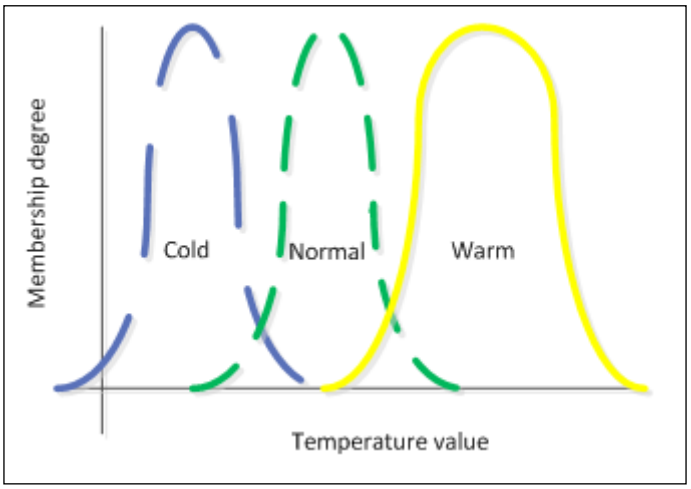
Considering that the concepts addressed in this chapter are advanced, we are not providing full code implementations; instead, we provide only a basic structural snippet on how to start implementing these concepts.

Neuro-fuzzy

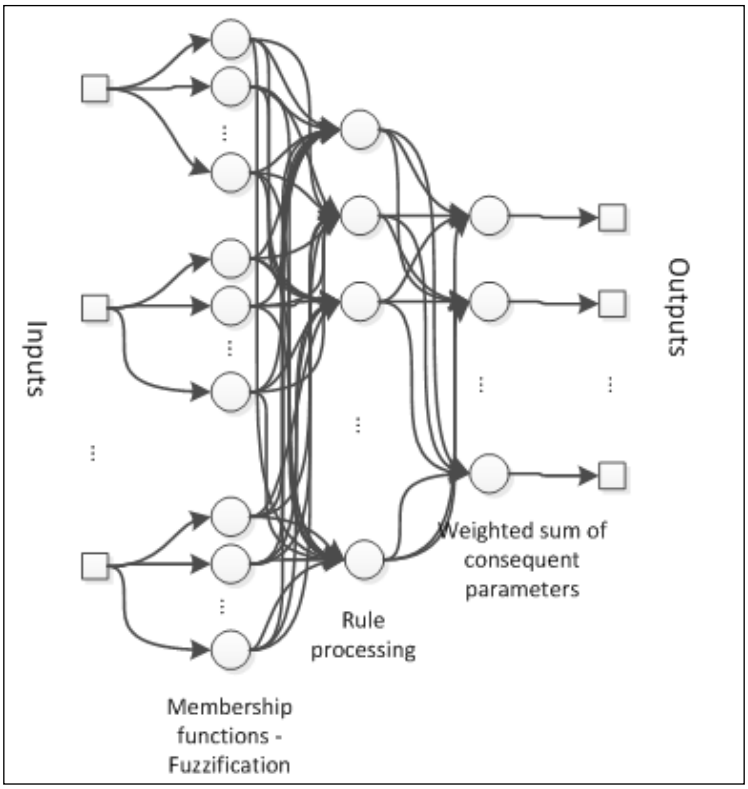
Fuzzy logic is a type of rule-based processing, where every variable is converted to a symbolic value according to a membership function, and then the combination of all variables is queried against an *IF-THEN* rule database.



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:



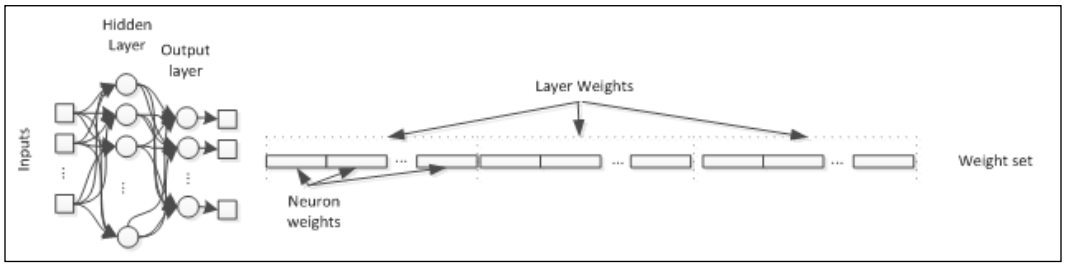
In this architecture, the training finds optimal weights for the rule processing and weighted sum of consequent parameters only, the first hidden layer has no adjustable weights.

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.

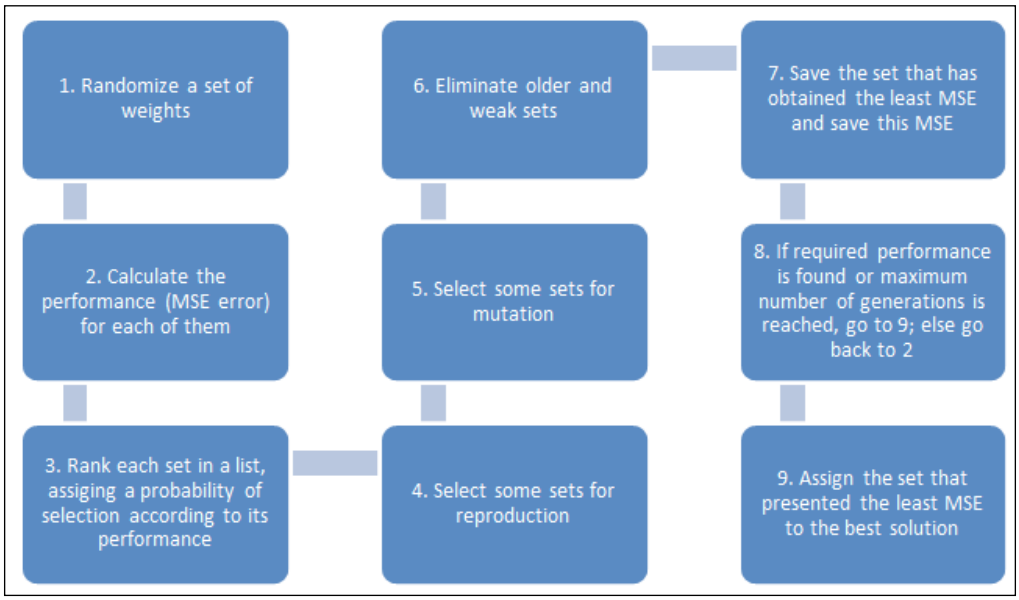
Think of the neural weights as a genetic code (or DNA). If we could generate a finite number of random generated weight sets, and evaluate which produce the best results (smaller errors or other performance measurement), we would select a top N best weight, and then set and apply genetic operations on 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<dou
ble>> 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 elimintation
    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 Leaning 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 Methor for the Solution of Certain Non-Linear Problems in Least Squares, Quaterly 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 Reasearch, Wiley, ISBN-10 0471086029 pp. 230, 1983.

Pearl, Judea, Causality: Models, Reasoning, and Inference, Cambridge University Press, ISBN-10 0521773628, 2000.

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, Cambrige 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.

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 (SIPIJ), 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, Spinger 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.

Index

Symbol

A

B

C

D

E

F

G

H

I

J

S

T

U

W

X

Z