Learning Neural Networks with Java

Ryan Sanders

# Table of Contents

[1 Table of Contents 2](#_Toc365405284)

[2 Introduction 3](#_Toc365405285)

[2.1 Overview and History of Neural Networks 3](#_Toc365405286)

[Part 1 6](#_Toc365405287)

[3 Feedforward Neural Networks 7](#_Toc365405288)

[3.1 Why Feedforward Neural Networks? 7](#_Toc365405289)

[3.2 What Can They Do? 7](#_Toc365405290)

[3.3 Limitations and Criticisms 7](#_Toc365405291)

[4 Basic Feedforward Network Theory 8](#_Toc365405292)

[4.1 Neurons 8](#_Toc365405293)

[4.2 Typical Activation Functions 9](#_Toc365405294)

[4.3 Bias 11](#_Toc365405295)

[4.4 Building a Feedforward Network 12](#_Toc365405296)

[4.5 A Simple Example 13](#_Toc365405297)

[5 Learning Using Back-propagation 16](#_Toc365405298)

[5.1 Objective or Error Function 16](#_Toc365405299)

# Introduction

The aim of this book is to provide the reader with a good understanding of the theory behind feedforward neural networks trained using back-propagation, combined with a walkthrough of implementing such a network in Java. It is meant to be a combination of theory and practice, where the practice helps reinforce the theory and make things a bit more concrete, and hopefully more interesting.

This book is structured as follows:

Introduction This section. Includes an overview and history of neural networks.

Part 1 Goes through the theory underpinning feedforward neural networks and how to train a network using back-propagation. Implements such a network using Java. The network is used to learn a simple function (the XOR function) to prove it works.

Part 2 Delves into real world uses of feedforward neural networks. Builds upon the theory and code from part 1 to create a more interesting neural network that using advanced concepts.

## Overview and History of Neural Networks

A number of terms and concepts are presented here with minimal explanation. Later in this book we will revisit these concepts in detail; they are mentioned only as an overview.

In the late 19th century Alexander Bain, and later William James, began to understand how the brain works. As part of the artificial intelligence field, it made sense to think how the brain could be replicated in software in order to create a system that could learn.

A biological neural network, such as the brain, is made up of a vast number of neurons. A biological neuron is shown in Figure 1.

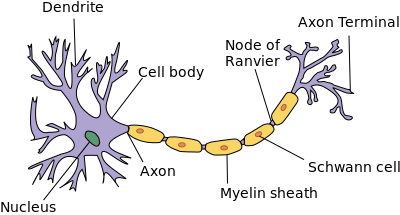


Figure : Biological neuron

For the purpose of comparing to an artificial neuron, we focus only on the dendrites, the cell body and the axon. The dendrites collect the output of other neurons such that they become the input to this neuron. The neuron processes the input and distributes it to the dendrites of other neurons through the axon.

This abstraction makes more sense when compared to an artificial neuron, such as the one in Figure 2.



Figure : Artificial neuron

Warren McCulloch and Walter Pitts created the artificial neuron in 1943. In this particular artificial neuron, there is one input and one output. The input is multiplied by a weight, in this case 1. If the result is greater than the threshold, 2, the neuron fires, or outputs 1. Otherwise the output is 0.

Now that we have an artificial neuron, the next step is to connect a number of them in some way to make an artificial neural network. This book focuses on feedforward neural networks, where the neurons are organized into layers and each neuron in a layer only connects to a neuron in a later layer.

The perceptron is the simplest form of feedforward neural network. It was invented by Frank Rosenblatt in 1957 and is comprised of two layers, and input layer and an output layer—see Figure 3 for an example.



Figure : An example of a perceptron, with two neurons in the input layer and one in the output layer

The alternative to feedforward networks is recurrent networks, where any neuron can connect to any other neuron. An example of a recurrent neural network is a Hopfield network, invented by John Hopfield in 1982.

Of course, an artificial neural network by itself is only part of the story. The goal is to make that network learn. The psychologist Donald Hebb came up with Hebbian theory to describe how biological neurons learn. The theory describes "associative learning", in which simultaneous activation of cells leads to pronounced increases in synaptic strength between those cells. The theory is often summarized as “cells that fire together, wire together”.

Hebbian learning was the first learning technique to be implemented in artificial neural networks. It is implemented such that the weight between two neurons increases if they activate simultaneously and reduces otherwise. Hopfield networks learn in this way and function as associative memory; after the network trains on a set of examples a new input will be classified as the example in the training set that most closely resembles the new input.

Frank Rosenblatt’s perceptron also included an algorithm for learning that is based on stochastic gradient descent. However, it turns out that the perceptron was quite limited in what it could learn; in particular, it can only categorize data that is linearly separable. As an example, this means that while it is capable of learning the AND function, it is incapable of learning something as simple as the XOR function. This limitation caused interest in neural networks to wane.

Learning more complex functions requires additional layers, but training such a network was not possible until the invention of the back-propagation algorithm. The algorithm was first invented in 1969 by Arthur Bryson and Yu-Chi Ho. However, it wasn’t until the mid-1980s that it was used to train artificial neural networks. Now neural networks could learn more complex functions, and interest in neural networks was revived.

# Feedforward Neural Networks

## Why Feedforward Neural Networks?

This book focuses on feedforward neural networks. A reasonable question is, why? There are a number of different neural networks, including recurrent neural networks such as the Hopfield network mentioned in Overview and History of Neural Networks (section 2.1, page 3).

The reason is that feedforward neural networks form a small but significant part of a larger field, and examining them in detail covers a lot of ground. Also, a number of other networks, such as self organizing maps, are extensions or variations of feedforward networks. So focusing on feedforward neural networks gives a good base for further exploration into neural networks.

## What Can They Do?

Another reasonable question is, what can feedforward networks do?

Neural networks are good at solving classification problems and for approximating functions that are tolerant of some imprecision and do not have a known deterministic solution. To be successful they generally need lots of available training data.

## Limitations and Criticisms

In covering what neural networks can do we immediately uncover some limitations. They are limited to certain types of problems. They need a lot of training data to learn a solution. In addition, that training data needs to be diverse.

Other limitations include that once a solution is arrived at, it cannot be examined to understand how the problem was solved. Learning also tends to be slow.

Another common criticism is that, although neural networks were meant to mimic how the human brain learns, they do not yet do so. One counter argument is that when the Wright brothers achieved human flight, it was not by mimicking birds.

# Basic Feedforward Network Theory

This section explains some basic concepts regarding feedforward neural networks. Later, we’ll revisit some of these concepts in more detail.

## Neurons



Figure : Example neuron

We saw the neuron in Figure 4 in History of Neural Networks (section 3.1, page 4). To recap, input is multiplied by a weight, in this case 1. In this example, if the result is greater than the threshold, 2, the neuron fires, or outputs 1; otherwise the output is 0. So an input of 1 would result in an output of 0 () and an input of 3 would result in an output of 1 ().

More formally, the output of a neuron is described as follows:

So how does that formal definition fit with the example we’ve seen? Well, briefly, in the example we only have one input so:

In this case, bias and threshold are effectively the same thing, giving:

Finally, an activation function is applied to the sum plus the bias. The activation function here is a step function, shown in Figure 5.



Figure : Step function

Try and to verify that the earlier examples work with this more formal definition. Don’t worry if this doesn’t make too much sense, we’ll go into more detail in the following sections.

## Typical Activation Functions

We’ve seen one activation function, the step function. The original perceptron used this activation function.

Usually, though, the activation function is something else. For reasons that we’ll go into when we look at Learning Using Back-propagation (section 6, page 17) we normally want an activation function that is differentiable. A more common activation function is the sigmoid function, shown in Figure 6. It is somewhat similar to the step function, but with a region of uncertainty approximately between .



Figure : Sigmoid function

The sigmoid function is defined as:

You’ll notice that the sigmoid function only returns positive values. Another activation function that returns both positive and negative values is the hyperbolic tangent (or tanh) function, shown in Figure 7.



Figure : Hyperbolic tangent function

The hyperbolic tangent function is defined:

## Bias

Earlier, I described bias and threshold as being effectively the same thing. This only really holds true when using the step function. Without a bias, ie, , the threshold is 0. This matches the step function shown in Figure 5.

With the bias used in the example, the step function is effectively shifted by the bias, as shown in Figure 8. The input to this function is now just , where u, the weighted sum of the inputs, needs to be greater than 2 for the shifted activation function to output 1.



Figure : Step function shifted by the bias

The same principle applies to bias and any activation function. Essentially the bias shifts the activation function along the x-axis.

So why is bias useful? We’ll come back to that later.

## Building a Feedforward Network

Now that we have our neurons, we need to build them into a network. We’re specifically interested in building feedforward networks, so there are some rules as how we can combine neurons.

We saw an example of a basic feedforward network, a perceptron, in History of Neural Networks (section 3.1, page 4). A slightly more complex example of a feedforward network is shown in Figure 9.



Figure : Feedforward network with one hidden layer

You can see that the neurons are ordered in layers; in this example there are three layers, the input layer (nodes A and B), the output layer (node E) and a hidden layer (nodes C and D). The hidden layer is so called because it is hidden to the user of the neural network; all interaction is done through the input and output layers.

Since this is a feedforward network, all nodes in one layer must connect with a node in the next layer. For example, node E cannot connect to any of the other nodes and node A cannot connect to node E. Recurrent networks remove this restriction.

Now that we are looking at layers, there’s a caveat that needs to be added to the definition given in section 5.1, Neurons, on page 9. That definition applies to all neurons except those in the input layer. The neurons in the input layer actually do no processing; they simply pass the input through to the next layer. For example, if node B is given an input of 2, then the output of node B will also be 2.

You might be asking when you would want one hidden layer versus two or more, or indeed when you want no hidden layers. The short answer is that it depends on the problem; as a simple example, the XOR function requires one hidden layer but the AND function needs no hidden layers. We’ll go into more detail later on.

## A Simple Example

As we start building our first feedforward neural network we need a problem for it to learn. Although it may not make for the most interesting problem, there is something to be said for selecting something simple.

For one thing, it means we can determine a solution before we begin. This means we know the problem can be solved by a neural network and gives us a structure for the neural network that we can use, ie, how many nodes in each layer. Another benefit is that a simple example only requires a few nodes which allows for easier debugging if we need to step through the calculations.

The problem we are going to start with is the exclusive or function. We have actually already seen a network that can solve this, but here it is again in Figure 10.



Figure : A feedforward neural network that performs the XOR function

Assume we are using the step function as the activation function and a bias of 1.

If 0 and 0 are presented to the network:

1. C receives 0 as input.
2. D receives 0 as input.
3. Both C and D will output 0.
4. E will receive 0 as input.
5. E will output 0.

If 0 and 1 are presented to the network:

1. C receives -1 as input.
2. D receives 1 as input.
3. C will output 0.
4. D will output 1.
5. E will receive 1 as input.
6. E will output 1.

And so on for 1, 0 and 1, 1 (try walking through these to confirm).

# Learning Using Back-propagation

Learning in neural networks can be classified as follows:

1. Supervised learning
2. Unsupervised learning
3. Reinforcement learning
4. Supervised learning with a distal teacher

Back-propagation is a type of supervised learning.

## Supervised Learning

In supervised learning a training set is used where each record is comprised of an input pattern to be presented to the neural network and the desired output. The goal is to find the functional relationship between the input patterns and desired outputs such that inputs not in the training set can be presented to the neural network with the network returning the expected output.

Training is a repetitive, incremental process whereby an input is presented and propagated through the neural network producing an output. The actual output is compared to the desired output using an objective or error function.

The goal of training is to minimize this error. Back-propagation does this by adjusting the network’s weights using gradient descent.

## Objective or Error Function

Now that we have our feedforward neural network, how do determine how well it is performing?

There are a number of error or objective functions that can be used to see how well a network’s output matches the ideal output. The most common is the sum of squared errors (SSE):

where indexes the patterns in the training set, indexes the output nodes, is the target for that training set and output node, and the actual network output.

Later we will look at some other error functions and why we might use one over another.

## Gradient Descent

Gradient descent is an algorithm to find a local minimum of a function. It works by taking steps proportional to the negative of the gradient at that point.

It is probably best illustrated with an example. Let’s say we want to find the local minimum of .



Figure :

Initially we need to guess at a local minimum, let’s say we start with .



Figure : , with gradient at

We calculate the gradient, of course, using . So the gradient at our arbitrary starting point is 12.

Now we take a step proportional to the negative of the gradient:

Say we make :

It is easy to code the algorithm as shown in Code 1 below.

**package** neuralnetwork;

**import** java.text.NumberFormat;

**public class** GradientDescent {

**private static final** NumberFormat NUMBER\_FORMAT = NumberFormat.getInstance();

**public static void** main(String[] args) {

NUMBER\_FORMAT.setMinimumFractionDigits(6);

**double** xOld = 0;

**double** xNew = 6;

**double** proportion = 0.1;

**double** precision = 0.00001;

**while** (Math.abs(xNew - xOld) > precision) {

xOld = xNew;

**double** change = fPrime(xOld);

xNew = xOld - proportion \* change;

System.out.println("Change: " + NUMBER\_FORMAT.format(change) +

", xNew = (" + NUMBER\_FORMAT.format(xNew));

}

System.out.println("Local minimum occurs at: " + NUMBER\_FORMAT.format(xNew));

}

**private static double** fPrime(**double** x) {

**return** 2 \* x;

}

}

Code : Example gradient descent algorithm

The output is as follows:

Change: 12.000000, xNew = 4.800000

Change: 9.600000, xNew = 3.840000

Change: 7.680000, xNew = 3.072000

Change: 6.144000, xNew = 2.457600

Change: 4.915200, xNew = 1.966080

Change: 3.932160, xNew = 1.572864

Change: 3.145728, xNew = 1.258291

Change: 2.516582, xNew = 1.006633

...

Change: 0.000418, xNew = 0.000167

Change: 0.000335, xNew = 0.000134

Change: 0.000268, xNew = 0.000107

Change: 0.000214, xNew = 0.000086

Change: 0.000171, xNew = 0.000069

Change: 0.000137, xNew = 0.000055

Change: 0.000110, xNew = 0.000044

Change: 0.000088, xNew = 0.000035

Local minimum occurs at: 0.000035

## Factors to be Considered