Artificial Neural Networks

Extended Project
Henry Thompson

Introduction

Artificial Neural Networks (ANNs), often abbreviated to neural nets, are a technique often used in machine learning, a part of artificial intelligence (AI), usually for solving classification or function-fitting problems. They are able to learn via supervised, unsupervised and reinforcement learning, depending on their implementation. Notably, ANNs can produce emergent behaviour in agents responding to environment sensors, and are especially useful for pattern recognition (e.g. hand-writing recognition), time series prediction (e.g. predicting stock movement), signal processing (e.g. noise reduction), data processing (e.g. data compression) and controlling machinery (e.g. self-driving cars), amongst many other uses.

As a soft computing technique (i.e. a computational technique derived from biology), ANNs are based upon biological neurons. They offer solutions to many problems with which classical computing (conventional, rule-based computing) struggles – in particular, it is their ability to learn which is especially useful. Computers can solve many problems incredibly fast, such as calculating the square root of a nine-digit number, but fall considerably short at certain tasks including speech recognition and visual object recognition ("seeing"). Many researchers have dedicated entire careers to such problems, but computers can still not reliably distinguish between the image of a dog and a cat, a task children successfully complete instantaneously. ANNs have provided much progress in such fields, proving to be useful and effective in many.

This aim of this extended project is to both explore how neural networks work and have as much practical experience with them as possible, writing neural networks to perform a variety of tasks from scratch. As such, the extended project is split into a theory section, then a programming section. The programming language I will use is Java, as it is a language I am very familiar with, which runs ubiquitously, and whose object-oriented architecture is ideal for this project – a downside is the overhead from the Java Virtual Machine, but the neural networks will not be that large and consequently running time should not be an issue. Although writing neural networks from scratch may seem unnecessary when free ANN Java libraries such as the open-source JOOME, Neuroph and Encog projects exist, doing so will be both interesting as an exercise and excellent programming practice. Relevant code for the project will be included in this essay.

PART I – Theory

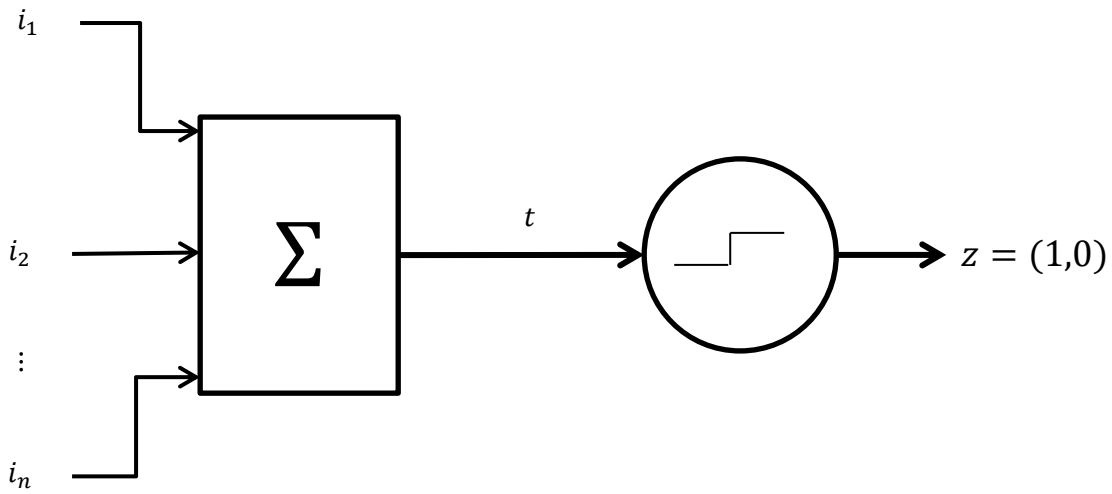
The Biological Neuron

Artificial Intelligence seeks to endow computers with many humanlike qualities, and so it stands to reason that mimicking the design of the human brain may hold the key to effective AI. The human brain is an extreme example of a neural network – though it is more than just this – with 100 billion neurons each connected to thousands of others. The fundamental computing unit of the brain is a cell called the neuron, and it is this we must first seek to imitate.

Despite the complexity of an entire brain, the computation performed by individual neurons is relatively straightforward. They have protrusions called dendrites from one end of the cell and axons on the other, which connect to other neurons. The points where they meet are called synapses. A

neuron connected to n other neurons via its dendrites will receive n input signals $i_1, i_2, \dots, i_n$ from each of these neurons. These inputs are weighted then combined to a value t. If t is greater than a threshold value T then the neuron will fire an output signal, or an "activation potential, z; if $t < T$ output remains as $z = 0$. The output will be passed to other neurons via the axons as an input to the other neurons, which will process this input using the same method as above.

Modelling a Neuron

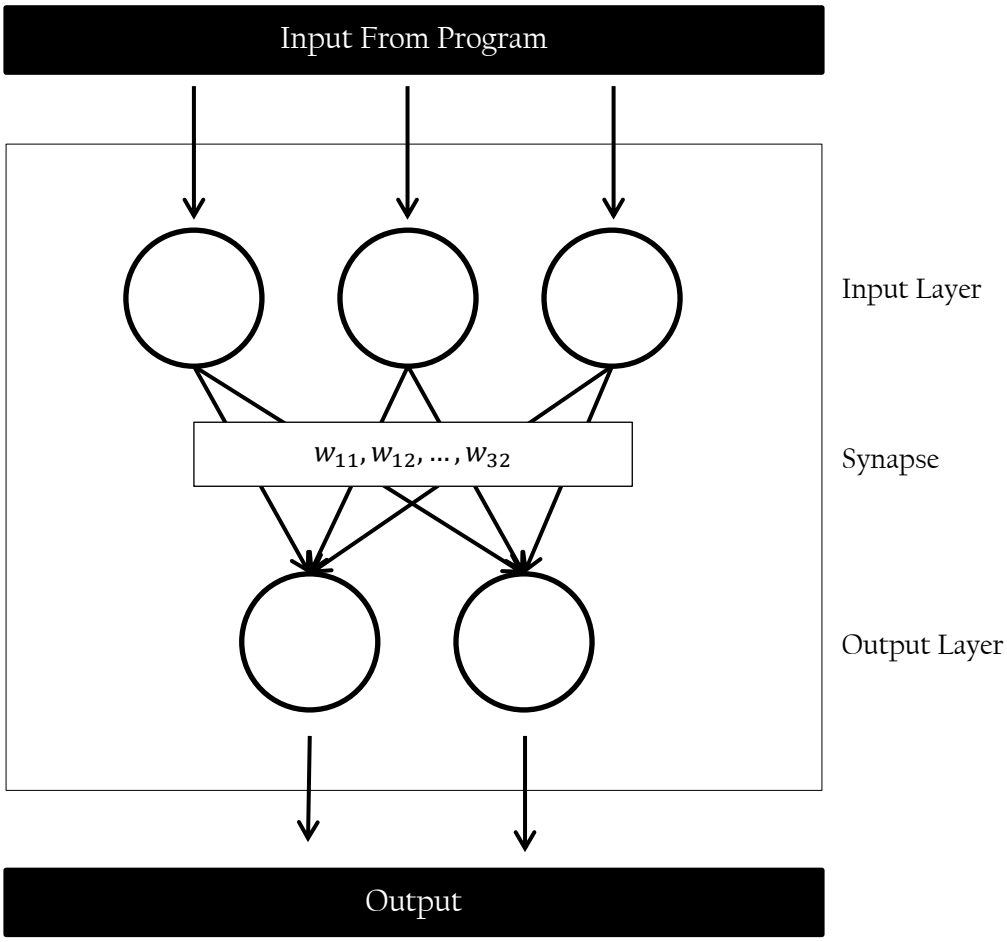


It is not very difficult to produce a crude model of this, called the "McCulloch-Pitts Neuron" (MPN):

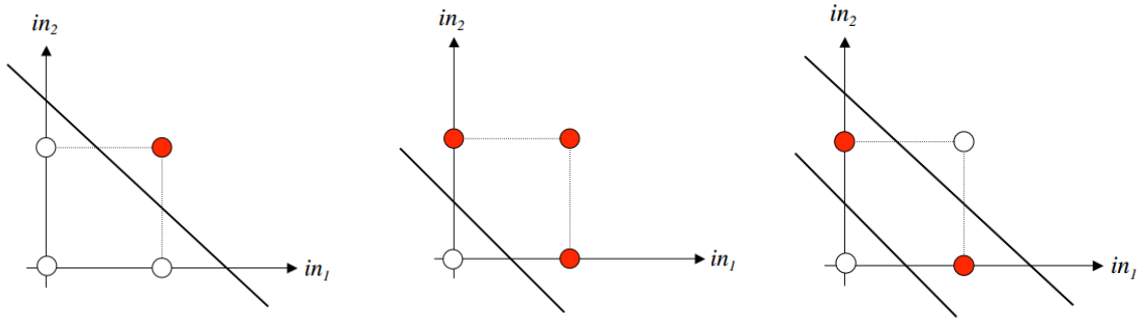
1. n inputs $i_1, i_2, \dots, i_n$, are fed in
2. The inputs are summed, $t = \sum_n i_n$
3. If $t < T$ then the neuron outputs $z = 1$; otherwise, the perceptron outputs $z = 0$

MPNs can be connected together. An arrangement of an "input" layer of MPNs I feeding into another "output" layer of MPNs J is called a perceptron (or "Single-Layer Feed-Forward ANN"). Just like in real neural network each connection, or synapse, between a neuron i and a neuron j is weighted by a weight w_{ij}. The weights within a synapse are most conveniently represented by a matrix:

$$\boldsymbol{w} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1j} \\ w_{21} & w_{22} & \cdots & w_{2j} \\ \vdots & \vdots & \ddots & \vdots \\ w_{i1} & w_{i2} & \cdots & w_{ij} \end{bmatrix}$$



Perceptrons can perform very basic tasks, such as classifying whether a point lies above a below a line on a plane, or performing binary AND, OR, and NOT operations. We shall model a perceptron later on to perform these tasks. Ultimately, perceptrons determine output boundaries, but are limited as they can only represent a single $(n-1)$-dimensional flat hyperplane in a n-dimensional input space. As a result, perceptrons can only divide the space into two regions, so can only classify any given input into two different categories. Problems solvable like this are said to be linearly separable. As a result, perceptrons are unable to model operations like XOR:



AND
$w_1 = 1, w_2 = 1, T = 1.5$

OR
$w_1 = 1, w_2 = 1, T = 0.5$

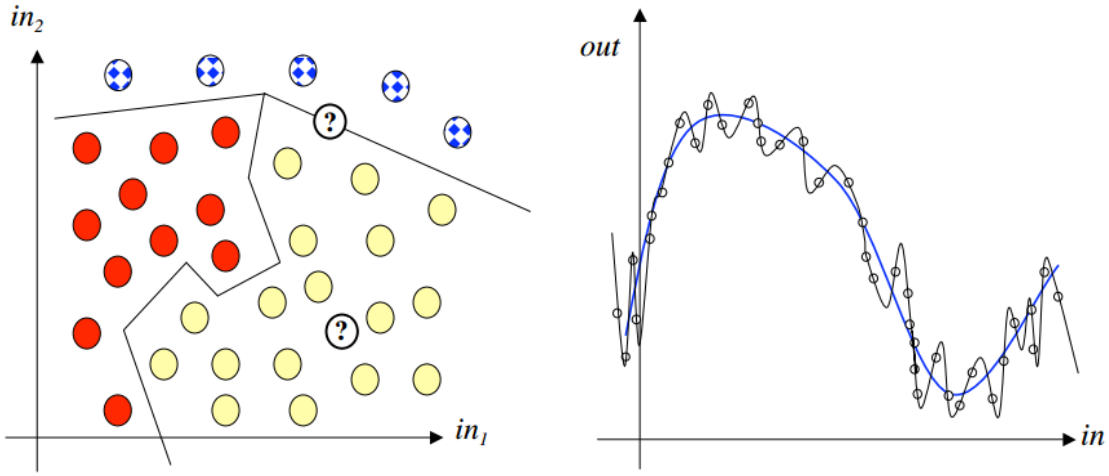
XOR

Likeness To Biological Neurons

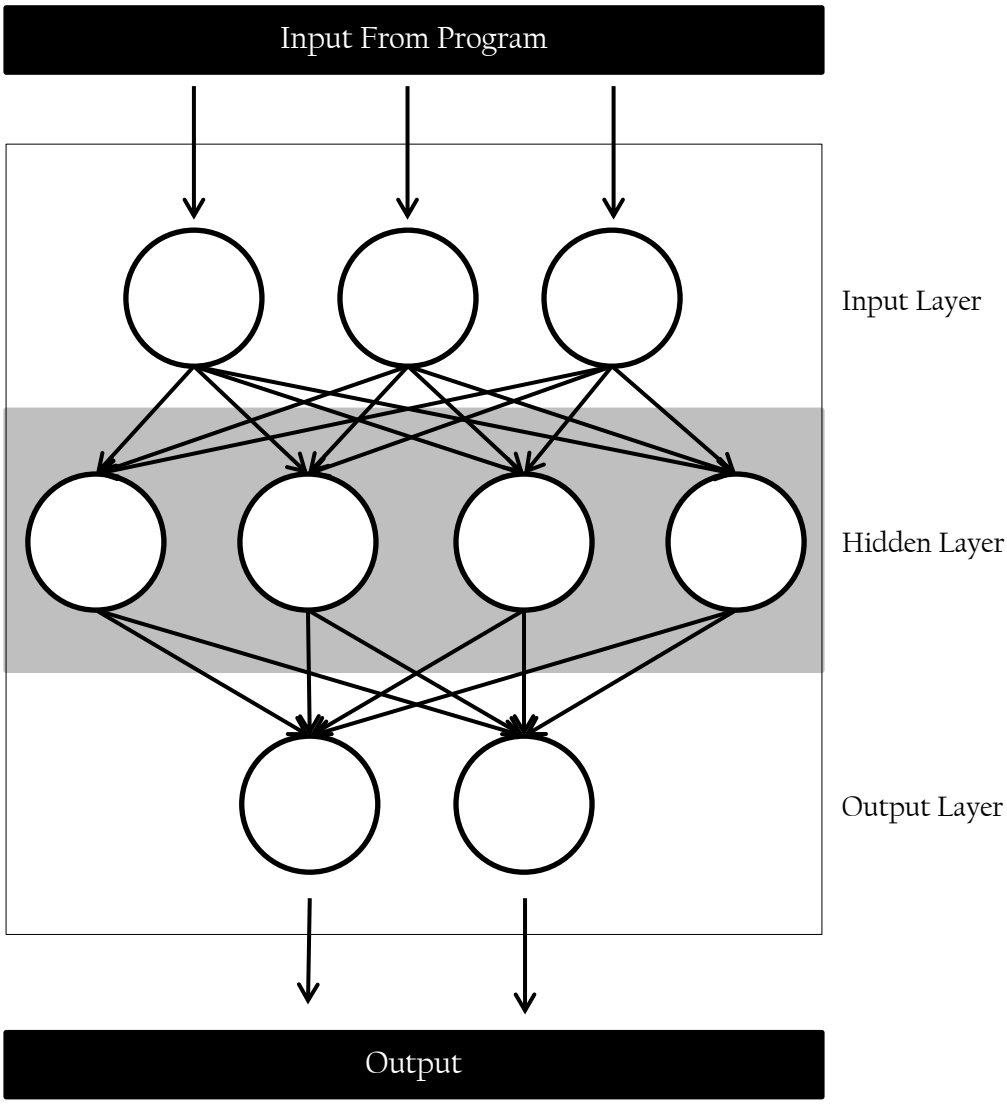
The McCulloch-Pitts Neuron is an extremely crude approximation of biological neurons. For example, in reality, the timings of outputs are very important, so a "spike time coding", which takes this into account, is more realistic. However, this greatly increases the complexity of computer models and is therefore not taken into account. The model above instead performs "rate coding" where time is not taken into account. It is useful for producing differentiable outputs, essential for efficient learning algorithms such as backpropagation as will be seen later. MPNs also lack non-linear summation, smooth thresholding, stochasticity, and temporal information processing, all of which real neurons do.

Multi-Layer Feed-Forward ANNs

Perceptrons are rather limited in their usefulness, so other ANN models are used. Often, we need multiple complex regions in an input space for more complex classification, or to approximate a complex function to fit a set of noisy input values:



One solution is MLPs (Multi-Layer Perceptrons), or "multi-layer feed-forward networks", which are similar to perceptrons but have "hidden" layers between the input and output layers. Each input neuron is set as a variable for the calculation being performed – for example, if this ANN is used in visual object recognition, each input neuron may represent an individual pixel's value – and these values feed forward through the neural network into subsequent "hidden" layers via synapses, where they are weighted each time. Eventually it reaches the "output layer" where each output neuron represents the value of a component of the solution.



Calculating Appropriate Weights – Supervised Learning

Currently, there is no method of knowing what the appropriate weights are – instead a computer typically starts with random weights then "teaches" itself the correct weights using a learning algorithm because computing weights directly is intractable in all but the simplest ANNs.

Whilst "learning" implies a process similar to human learning, this is misleading – perhaps a better term is "calculating". Commonly, supervised learning is used, where known sample inputs (the training samples) are fed into the ANN and the actual output compared with the correct expected output. Each time, the weights are appropriately altered, and this process is repeated until the output of the ANN is consistently within a specified maximum error given any input – in effect, the weights and thresholds form an ANN's memory. One pass of all the training samples is called an epoch.

To understand how this works, take a neural network which accepts n inputs and produces m outputs. The values of the inputs can be represented by the n-dimensional vector $\boldsymbol{x}$. The values of the output can be represented by the m-dimensional vector $\boldsymbol{z}$. Mathematically speaking, any given neural network with fixed structure maps $\boldsymbol{x}$ and the set of weights $\boldsymbol{w}$ to the output $\boldsymbol{z}$,

$$\boldsymbol{z} = f(\boldsymbol{x}, \boldsymbol{w})$$

Therefore, ANNs may be viewed as a way of defining the mapping f, useful when f cannot be calculated by hand or no other way to define it is known. If we train a neural network through supervised learning using a set of training samples, we have a desired output vector $\boldsymbol{d}$ for every $\boldsymbol{x}$ where

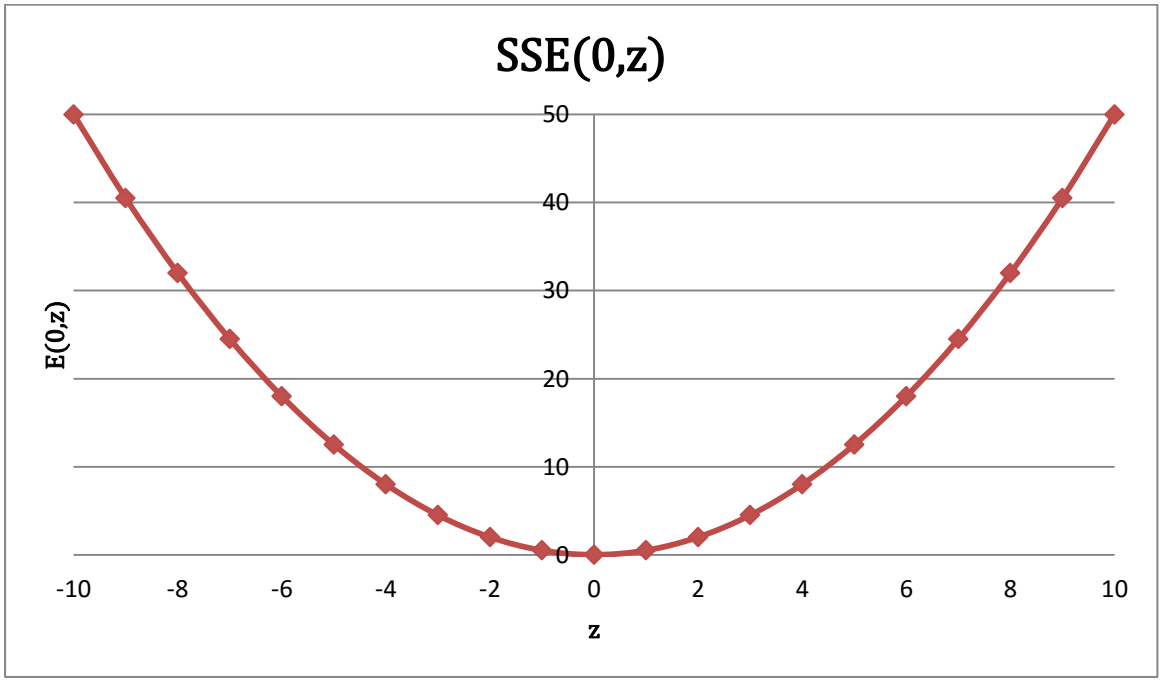
$$\boldsymbol{d} = g(\boldsymbol{x})$$

Therefore, the neural net problem can be stated as calculating weights $\boldsymbol{w}$ which brings $g(\boldsymbol{x})$ into alignment with $f(\boldsymbol{x}, \boldsymbol{w})$ – i.e. calculate a set of weights which produces the desired output for any input.

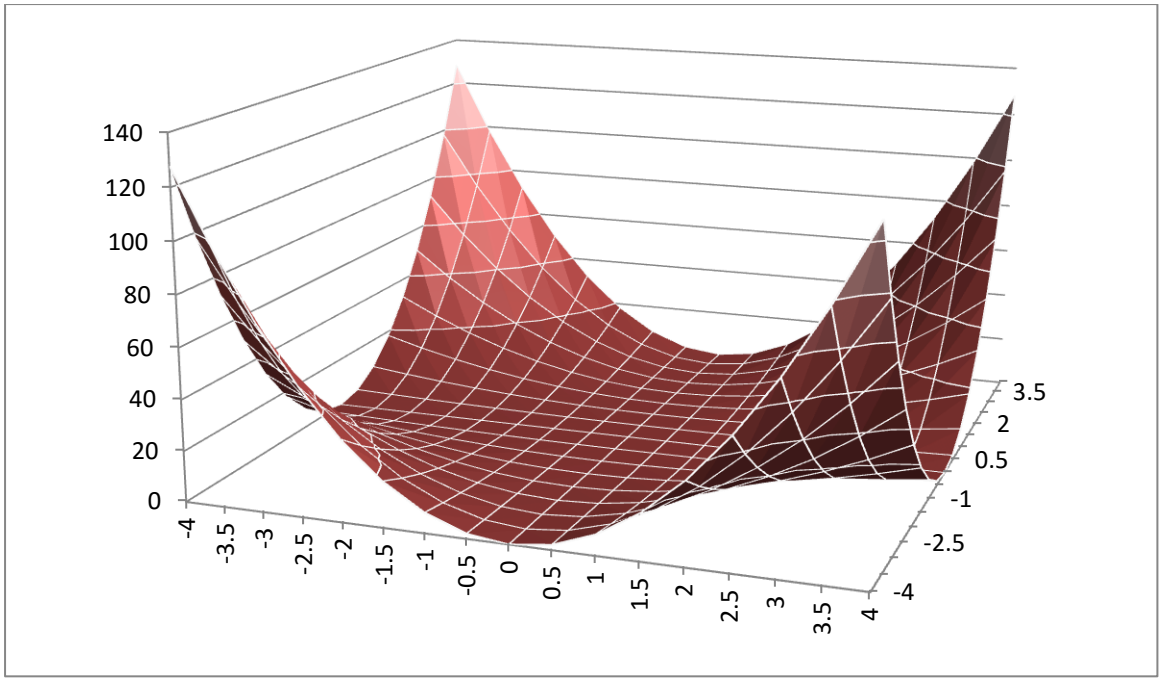
In order to do this, we first need a measure of the error between the actual output $\boldsymbol{z}$ and the desired output $\boldsymbol{d}$ using an error function, $E(\boldsymbol{d}, \boldsymbol{z})$. A common one is the Sum Squared Error (SSE) function (the Cost Entropy Error (CEE) will be discussed later):

$$E(\boldsymbol{d}, \boldsymbol{z}) = \frac{1}{2}(\boldsymbol{d} - \boldsymbol{z})^2$$

We square the error because we are interested in its magnitude, not its sign, and the $\frac{1}{2}$ makes calculus a little cleaner later on. If $\boldsymbol{d}$ and $\boldsymbol{z}$ are scalars, the graph of z against $E(d, z)$ for $d = 0$ looks like:



As can be seen in the graph above, since we want we want $E(d, z) = 0$, we must minimise $E(d, z)$ in order to find the optimal $\boldsymbol{w}$. If you have two weights w_1 and w_2, the error space for a linear perceptron with two weights w_1 and w_2 can be visualised as:



In this example it is easy to simply try moving along the w_1 and w_2 axes until $E(\boldsymbol{d}, \boldsymbol{z})$ is minimised – but in cases where there are many weights, we will be dealing with a many-dimensional search space in which this is impractical. Therefore, we will use a gradient descent method, which involves moving along the search space's surface in the direction opposite to that of steepest gradient. So, in general, if you have n weights, the required difference in the weights is proportional to the partial derivatives of E with respect to each weight,

$$\Delta \boldsymbol{w} = \begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix} \propto - \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \vdots \\ \frac{\partial E}{\partial w_n} \end{bmatrix}$$

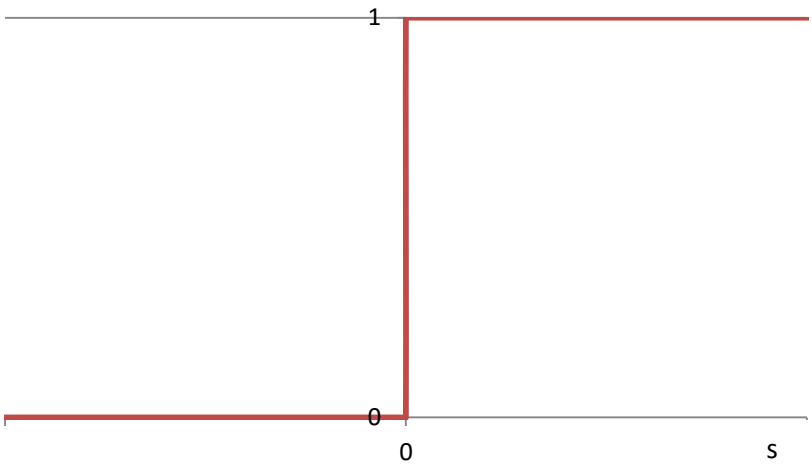
Note however that there is no place for the threshold in this mathematical model. To get rid of this, for each neuron with threshold value T, we can add a 'bias' input with a constant value $i_0 = T$ and constant weight $w_0 = -1$ to eliminate this by effectively translating the activation function by $-T$. The activation function now outputs 0 when the sum of the weighted inputs equals 0, and 1 when the sum is greater than 0.

The main issue, however, is that, clearly, gradient ascent can only work on a differentiable function. Since our activation function is a step function, we use a differentiable function which behaves similarly.

Choosing The Appropriate Activation Function

There are a number of activation functions which are commonly used. It should be noted that different layers can have different activation functions (more rarely, different neurons within a layer may differ as well). The most common functions and their uses are discussed below.

The Step/Signum Function:



This function cannot be differentiated, and can only be used in the most simple perceptrons.

$$\beta(\alpha) = \begin{cases} 0 & if\ \alpha \leq 0 \\ 1 & if\ \alpha > 0 \end{cases}$$

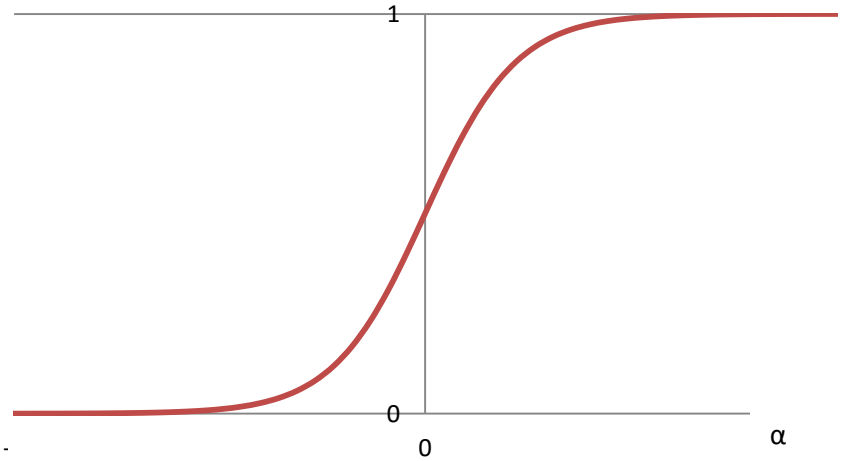
The Linear Function:

$$\beta(\alpha) = \alpha$$

This function is used on the output layer for non-binary regression or function-fitting problems, as it makes no sense to have a sigmoid function when the values can take any value. Such layers do no computation on their inputs, but pass the input values on unaffected to subsequent layers. This cannot be used in all layers of a network because it would not be able to solve non-linearly separable problems (it would be equivalent to a perceptron).

The Logistic Sigmoid Function:

$$\beta(\alpha) = \frac{1}{1 + e^{-\alpha}}$$



The logistic function can be considered a "smoothed out" variation of the threshold function. It can be shown (using mathematics well beyond the scope of this project) that this function can be used in the output layer of two-class classification problems involving one output unit where an output of **0**

represents one class and an output of 1 represents the other, and is used also in the hidden layers of other networks. It is very convenient as its derivative is extremely computationally simple:

$$\begin{aligned}\frac{d\beta}{d\alpha} &= -(1+e^{-\alpha})^{-2} \times -e^{-\alpha} \\ &= \frac{e^{-\alpha}}{(1+e^{-\alpha})^2} \\ &= \frac{1+e^{-\alpha}-1}{(1+e^{-\alpha})^2} \\ &= \frac{1}{1+e^{-\alpha}}\left[\frac{1+e^{-\alpha}}{1+e^{-\alpha}} - \frac{1}{1+e^{-\alpha}}\right] \\ &= \beta(1-\beta)\end{aligned}$$

The Softmax (or Normalized Exponential) Function:

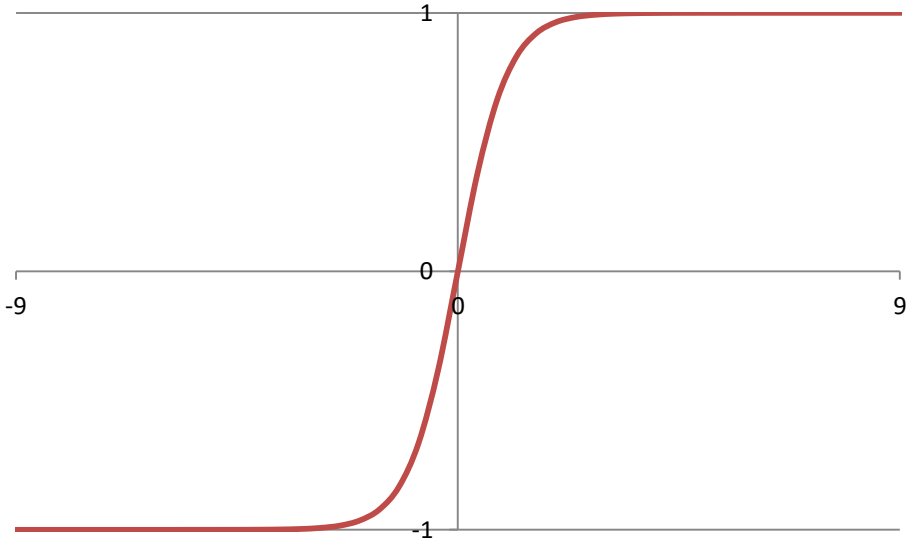
$$\beta(\alpha)_j = \frac{e^{\alpha_j}}{\sum_{k=1}^{K} e^{\alpha_k}}$$

This is a generalisation of the logistic function above. For reasons again well beyond this project's scope, it can be used in the output layer of a multiclass classification network because it represents a categorical probability distribution – so, for example, if a unit corresponding to a particular class outputs 0.9, there is a 90% chance that the input data represents a member of that class (this is also true of the logistic function for two-class problems). Nonetheless, it should be clear that the outputs must all sum to one, which allows this to happen. It should also be clear that with a single output neuron the function simplifies down to the logistic function. The derivative is the same as for the logistic function:

$$\frac{d\beta}{d\alpha} = \beta(1-\beta)$$

The Hyperbolic Tangent:

$$\beta(\alpha) = \tanh(\alpha)$$



Some empirical evidence suggests that anti-symmetric activation functions (i.e. ones which satisfy $f(x) = -f(x)$) can in fact enable machines to learn faster. The hyperbolic tangent is one such function, and is closely related to the logistic function:

$$\tanh(\alpha) = 2logistic(2x) - 1$$

This function is also useful because it has a very computationally simple derivative, just like the logistic sigmoid:

$$\frac{\mathrm{d}\beta}{\mathrm{d}\alpha} = 1 - \beta(\alpha)^2$$

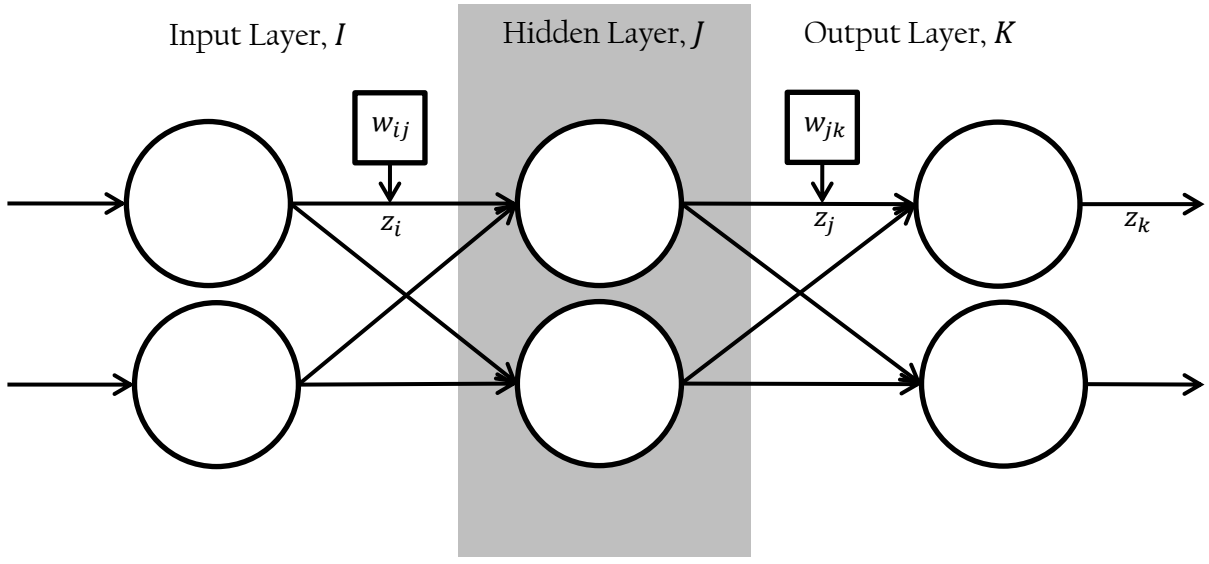
Obviously, in this case, binary values will have to use targets of -1 and 1 instead of 0 and 1.

The Backpropagation Algorithm

The Backpropagation Algorithm is the most common learning algorithm used, and can be used on any feed-forward ANN whose activation and error functions are differentiable. It is named so because it adjusts weights starting at the output layer, working its way back until it reaches the input layer.

Consider an ANN with an input layer I, a hidden layer J and an output layer K. Take any one neuron from each layer. We can say that the input neuron outputted a value of z_i; the hidden neuron received a net input of $t_{ij} = w_{ij}z_i$ from this connection and outputted a value of z_j; the output neuron received a net input of $t_{jk} = w_{jk}z_j$ and outputted a value of z_k, though its expected output was d. If a weight w_{jk} between the hidden and output neurons exists, we must calculate the appropriate adjustment Δw_{jk} for that weight. We similarly must adjust the weight w_{ij} between the input and hidden neurons by a value of Δw_{ij}.

Whilst there will be many other weights and outputs in the ANN, we can ignore them: the method of calculating the adjustment for a weight leading to any output neuron must be the same as that for calculating Δw_{jk} , and the method of calculating the adjustment for any weight leading to a hidden neuron must similarly be the same as that for calculating $\Delta \mathrm{w}_{ij}$. To keep the maths simple, we will use the SSE error function.



Let's being with $\Delta w_{jk} \propto -\frac{\partial E}{\partial w_{jk}}$. Since E is not a direct function of w_{jk} we must expand it using the chain rule:

$$\begin{aligned}\frac{\partial E}{\partial w_{jk}} &= \frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial w_{jk}} \\ &= \frac{\partial E}{\partial z_k}\frac{\partial t_{jk}}{\partial w_{jk}}\frac{\partial z_k}{\partial t_{jk}}\end{aligned}$$

The derivative of the error function with respect to z_k is easily calculated:

$$\begin{aligned}\frac{\partial E}{\partial z_k} &= \frac{\partial}{\partial z_k}\left(\frac{1}{2}(d - z_k)^2\right) \\ &= -(d - z_k)\end{aligned}$$

And since $t_{jk} = z_j w_{jk}$ therefore

$$\frac{\partial t_{jk}}{\partial w_{jk}} = z_j$$

Therefore:

$$\frac{\partial E}{\partial w_{jk}} = -(d - z_k)\frac{\partial z_k}{\partial t_{jk}} z_j$$

We can now do a similar process for w_{ij}. This depends on each of the n neurons it is connection to:

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}} &= \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}} \\ &= \left[\sum_{k=1}^{n}\frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial z_j}\right]\frac{\partial z_j}{\partial t_{ij}}\frac{t_{ij}}{\partial w_{ij}} \\ &= \left[\sum_{k=1}^{n}\frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial t_{jk}}\frac{\partial t_{jk}}{\partial z_j}\right]\frac{\partial z_j}{\partial t_{ij}}\frac{\partial t_{ij}}{\partial w_{ij}}\end{aligned}$$

We know already that $\frac{\partial E}{\partial z_k} = -(d - z_k)$. Since $t_{jk} = w_{jk} z_j$

$$\frac{\partial t_{jk}}{\partial z_j} = w_{jk}$$

Since $t_{ij} = w_{ij} z_i$ therefore

$$\frac{\partial t_{ij}}{\partial w_{ij}} = z_i$$

Therefore:

$$\frac{\partial E}{\partial w_{ij}} = -\left[\sum_{k=1}^{n}(d - z_k)\frac{\partial z_k}{\partial t_{jk}} w_{jk}\right]\frac{dz_j}{dt_{ij}} z_i$$

The remaining derivatives are derivatives of the activation function, so should be substituted accordingly. For example, if the logistic function was used:

$$\Delta w_{ij} \propto -\frac{\partial E}{\partial w_{ij}} = \sum_{k=1}^{n} \left[(d_k - z_k) z_k (1 - z_k) w_{jk}\right] z_j (1 - z_j) z_i$$

$$\Delta w_{jk} \propto -\frac{\partial E}{\partial w_{jk}} = z_j (d - z_k) z_k (1 - z_k)$$

Looked at more closely, we see that $\frac{\partial E}{\partial w_{jk}}$ takes the derivative of the error at the output neuron, (in this case $(d_k - z_k)$), and multiplies it by the derivative of the activation function (in this case $z_k(1 - z_k)$) Consequently, if the error is negative, the weight is altered such that we move down the gradient; if the error is positive, we move up the gradient, reducing the error. Therefore this "error gradient" for an output neuron k with error function E and activation function β is

$$\delta_k = E'(z_k)\beta'(t_{jk})$$

In layer J the error gradient depends on the sum of the weighted error gradients of the next layer (which is not necessarily the output layer in a many-layered ANN):

$$\left[\sum_{k=1}^{n} E'(z_k)\beta'(t_{jk})\, w_{jk}\right]\beta'(t_{ij}) = \left[\sum_{k=1}^{n} \delta_k w_{jk}\right]\beta'(t_{ij})$$

And so, it makes sense to say that:

$$\delta_j = \beta'(t_{ij}) \left[\sum_{k=1}^{n} \delta_k w_{jk}\right]$$

Also, we must have a coefficient of proportionality η, or "rate constant". If we apply these observations we are left with the "generalized delta rule":

- For any weight between a neuron i and a hidden neuron j:

$$\Delta w_{ij} = \eta \delta_j z_i$$

- For any weight between a neuron j and an output neuron k:

$$\Delta w_{jk} = \eta \delta_k z_j$$

Analysing this reveals that it provides an extremely efficient and simple way to train the neural network because both formulae depend only on numbers which are either precalculated or in the local vicinity, i.e. numbers associated with neurons directly connected to it. Δw_{jk} depends only z_k, which was calculated previously; and w_{jk} and d_k, which were known beforehand. Δw_{jk} only depends on the error gradients of the layer in front of it – which must have been previously calculated; the weights w_{jk} for the following layer, which were known beforehand; and z_j, which was calculated previously.

So, any computation performed is relatively simple and on precalculated values, making this very efficient; moreover, only a local computation is required on each layer, so the computation time for

each weight, be it on the 1st layer or the 99th layer, is always the same. For an implementation with a fixed number of neurons in each layer, computation time is linear to the number of layers.

Weight Updates In A Perceptron

To teach a perceptron with a linear activation function using SSE it should now be clear that:

$$\Delta w_{ij} = \eta \left(d_j - z_j\right) z_i$$

Choosing Appropriate Learning Rates

The learning rate η must be chosen carefully. If too small, gradient ascent will take small steps and the neural network will take a long time and many epochs to train. Too large and it will overshoot the top of the curve; in the next epoch it will move back towards the peak but overshoot again. This process will repeat itself continually, forming a positive feedback loop where the neural network will never stabilise on an appropriate set of weights, but oscillate or even diverge. In fact, the value of η depends on each and every implementation of the feed-forward neural network.

A degree of experimentation is required, trying it out with learning rates of 0.001, 0.01, 0.1 etc. and using the results as a guide. It is also sometimes advantageous to have an epoch dependent learning rate – two common approaches for epoch t are:

$$\eta(t) = \frac{\eta(1)}{t}$$

$$\eta(t) = \frac{\eta(0)}{1 + t/\tau}$$

Age-dependent learning rates like these are found in human children. Different layers may also have different learning rates, but calculating the optimal values for each layer is time consuming and often not worth it. Evolutionary algorithms can be beneficial, however, in producing optimal values.

Choosing an Appropriate Error/Cost Function

As mentioned a number of cost functions exist, and the most appropriate depends on the type of problem being solved. In general, we must find the neural network which maximises the likelihood L of correctly observing the training data, which is the equivalent of minimising the error/cost function $E = -\ln(\mathrm{L})$.

The Cross-Entropy Error (CEE):

In a classification problem, the most likely class that an input pattern belongs to must be determined. (More formally, the posterior probabilities of class membership conditioned on the input variables must be modelled). Here the Cross-Entropy Error (CEE) should be used.

For a two-class classification problem with a single output using S training sets:

$$E_{CE}(z) = -\sum_{s} [d.\log(z) + (1 - d).\log(1 - d)]$$

For multiple-class classification problems with k output nodes:

$$E_{CE} = -\sum_{s}\sum_{k} d_k \,.log\,(z_k)$$

The derivation of these is beyond the scope of this project. If the Sigmoid activation function is used in the two-class problem and the Softmax activation function is used in the multiple-class problem (see section on choosing appropriate activation functions) then, for both problems, the weight update is very simple because the derivative of the activation function cancels with that of the CEE:

$$\Delta w_{jk} = \eta \sum_{s} (d_k - z_k)\, w_j$$

The Sum Squared Error (SSE):

Regression or Function Approximation problems involve approximating the underlying function from a set of noisy inputs. In these cases the Sum Squared Error (SSE) is most appropriate.

For an output layer using S normally distributed training samples and K output nodes:

$$E_{SS}(z) = -\frac{1}{2}\sum_{s} (d_{ks} - z_k)^2$$

If and a linear output function is used (see section on choosing appropriate activation functions) this must be the perceptron weight update equation:

$$\Delta w_{jk} = \eta \sum_{s} (d_k - z_k)\, w_j$$

Notice that the weight update equation is the same for regression, two-class classification and multiple-class classification provided the appropriate cost and activation functions for each case are used!

In general:

- **Regression/Function Approximation Problems:** Use SSE cost function, linear output activations, sigmoid hidden activations
- **Classification Problems (2 classes, 1 output):** Use CE cost function, sigmoid output and hidden activations
- **Classification Problems (multiple-classes, 1 output per class):** Use CE cost function, softmax outputs, sigmoid hidden activations

When to Stop Training

The sigmoid function can only reach values of 0 or 1 at $\pm\infty$. Clearly this can never be reached with finite step sizes. Even with offset targets, smaller gradient ascent steps mean the target will never be reached. Regression problems with real continuous values and linear activations also fail to ever achieve their targets. And so, we must instead decide when the outputs are "good enough".

In binary output systems, it may be that beyond a certain tolerance target (e.g. within 0.1 and 0.9) is considered good enough. It may be easier and more suitable however to stop when the SSE or CEE reaches a certain small value over all training samples (e.g. $SSE = 0.2$). Whilst this often useful in regression problems, for classification problems it is often useful to use the Root-Mean Squared

Deviation of the CEE instead, or calculate the percentage of times the correct output unit has the highest value, or a value beyond a threshold.

If a set of noisy input data is provided, generalisation gets worse as training is continued beyond a certain point. Therefore, it is necessary to estimate how well the network is generalising after each epoch and stop when this reaches a minimum.

Other Considerations

1. **Training Methods:** There are two common approaches to learning:

    a. **Batch Training:** The equations in the section above summed the errors from all of the sample inputs then performed the weight changes. A smaller η is required but this is usually made up for by the smaller number of weight updates required.
    b. **Online Training:** Weight updates are performed after each sample input is fed in. This is in reality stochastic gradient descent and so the weight changes can be erratic.

2. **Local Minima:** Error functions often have multiple minima. If training starts of near one of these then the global minimum may not be found. Starting with different initial weight sets can reduce the chances of this happening.

3. **Flat Spots:** If the error function has flat spots it can take a long time to pass through them. In particular, if the SSE is used with the sigmoid function (which should never be done anyway), for large or small values the gradient tends to zero. As a result, if the outputs become completely wrong during training then the weight updates become tiny and the learning takes extremely long. As a result initial weights must be small to avoid saturating the sigmoids before training even begins. Three other techniques remedy this:

    a. **Target Offsets:** If the target is offset by a small amount (e.g. 0.1 and 0.9 instead of 0 and 1) then learning can no longer saturate the sigmoids.
    b. **Sigmoid Prime Offsets:** Add a small offset to the sigmoid derivative so it is never zero when it saturates.
    c. **Use a CEE error function:** This cancels out the sigmoid derivative so saturation no longer matters. Evolutionary computation shows that this is the best approach.

4. **Preprocessing Data:** Data used for training should be representative (i.e. not contain too many examples of one type over another). Empirical evidence suggests that performance is better for continuous data if it is rescaled for a mean around zero and each input has a similar standard deviation. If online learning is used, the data should be shuffled after each epoch.

5. **How Many Neurons To Use:** This can only be sensibly determined from experimentation. It depends on the noisiness of and generalisation required from the data, as well as the activation functions used, the training algorithm used, and many other variables.

6. **Initial Weights:** If these are all the same, then the gradient descent algorithm will treat them the same and the network will not learn properly. Instead, they should be set to random values within a small range of $-x$ to x to avoid this.

Although there are numerous other techniques, they will not be included here. They are simply too numerous, and oftentimes complex. In any case, there should now be enough information to produce a basic neural network.

The Benefits And Limitations of ANNs

It should be clear now that ANNs have many benefits. Their ability to learn makes them invaluable, and because they are able to generalise the patterns found within training sets, they solve many problems where classical techniques would require enormous complexity. They are also extremely powerful computational devices which are fully Turing equivalent, universal computers and whose massive parallelism makes them inherently extremely efficient. Furthermore, they are very fault tolerant and noise resilient, so that even if low quality or corrupted data is inputted, the neural network can still process it.

Nonetheless, ultimately, an ANN is simply a parallel computational system which consists of simple processing elements connected in a specific way to perform a certain task. The models used are extremely crude approximations of how real, biological neural networks behave, and so ANNs cannot fully match their complexity and capabilities. Furthermore, it is often difficult to encode problems into values to be passed into the input neurons of an ANN.

PART II – Programming

Layers and Synapses

With this information, it was possible to start coding some neural networks. For perceptrons and feed-forward ANNs, it was unnecessary to create an object to represent every neuron as they are arranged in layers. Instead, the program is simpler and more efficient with just a class `Layer`, an instance of which represents a single layer, which behaves as follows:

- Receive the weighted, summed input values (the "net input") for each neuron in a `double[]` array from the previous layer;
- Run each net input value through an activation function. `Layer` is an abstract class so subclasses can implement their own `activationFunction(double[] netInput)`;
- Return the output as another `double[]` array. There must be an interface `OutputListener` which will allow other classes such as `Synapse` to receive the result when the layer has completed a computation and has produced an output.

Here is Layer.java:

`/src/com/henrythompson/neuralnets/Layer.java:`

```
public abstract class Layer {
	/** Number of neurons in this layer */
	private int mSize;

	/** The error gradients for each neuron in this layer */
	private double[] mErrorGradients;

	/** List of output listeners registered to receive any
	 * outputs from this layer */
	private ArrayList<OutputListener> mOutputListeners = new
ArrayList<OutputListener>();
	private double[] mLastOutput;

	/** Instantiates a new Layer object with the specified
```

```
     * number of neurons
     * @param size The number of neurons in this layer
     */
    public Layer(int size) {
            mSize = size;
            mErrorGradients = new double[size];
    }

    /** Returns the number of neurons in the layer */
    public int size() {
            return mSize;
    }

    /** Registers a new output listener with this layer */
    public void addOutputListener(OutputListener listener) {
            mOutputListeners.add(listener);
    }

    /** Removes the specified output listener, if it is registered
     * with this layer */
    public void removeOutputListener(OutputListener listener) {
            mOutputListeners.remove(listener);
    }


    /** Processes the specified input
     * @param input The net input values of each of the neurons
     * in this layer
     * @return The output of this layer when that set of inputs
     * was run through it
     */
    public double[] processInput(double[] input) {
            mLastOutput = activationFunction(input);

            for (OutputListener ol: mOutputListeners) {
                    ol.onOutput(mLastOutput);
            }

            return mLastOutput;
    }

    /** @return The most recent output of the layer, or
     * {@code null} if it has not outputted anything yet
     */
    public double[] getLastOutput() {
            return mLastOutput;
    }

    /** Sets the error gradient for an individual neuron
     * @param index The index of the neuron
     * @param gradient The value for the error gradient
     */
    public void setErrorGradient(int index, double gradient) {
            if (index < mSize) {
                    mErrorGradients[index] = gradient;
            }
    }

    /** Sets the error gradients of all the neurons in this
     * layer
     * @param gradients The values for the error gradients,
     * such that the value at {@code gradients[i]} is the
     * value of the error for neuron {@code i}
     */
    public void setErrorGradients(double[] gradients) {
            mErrorGradients = gradients;
    }
```

```
        /** @param index The index of the neuron
         * @returns The error gradient for the specified neuron
         */
        public double getErrorGradient(int index) {
                return mErrorGradients[index];
        }

        /** @return The error gradient for each neuron, such that
         * the error {@code i} corresponds to the error for
         * neuron {@code i}
         */
        public double[] getErrorGradients() {
                return mErrorGradients;
        }

        /** Performs the activation function on a particular net input
         * @param netInputs The net input value
         * @return The output of this neuron when this net input value
         * is run through it
         */
        protected abstract double[] activationFunction(double[] netInputs);

        /** Calculates the derivatives of the activation function based
         * on the outputs
         * @param outputs The last output of the layer
         * @return The derivatives of the activation function based on
         * the last outputs
         */
        protected abstract double[] getActivationDerivative(double[] outputs);

        /** @return A {@code String} which uniquely identifies this
         * type of neuron. Used when exporting and importing networks.
         */
        public abstract String typeName();
}
```

/src/henrythompson/neuralnets/OutputListener.java

```
public interface OutputListener {
        /** Called when the layer produces an output
         * @param output The values of the output from
         * each neuron in the layer
         */
        public void onOutput(float[] output);
}
```

Next, we need a way to connect these layers. This is where the `Synapse` class comes in. The `Synpase` class holds all of the weights between each of the neurons in the "from layer" (the layer from which the data flows as ouput) and each of the neurons in the "to layer" (the layer to which the data flows as input). It will register an `OutputListener` with the from layer, and when it receives an output, will calculate the net inputs for each of the neurons in the to layer as a `double[]` array, including any bias values. This forms the input of the next layer, and the process will continue.

/src/com/henrythompson/neuralnets/Synapse.java

```
public class Synapse implements OutputListener {
        /** The weights between the previous layer
         * and the next layer. Each weight represents
         * a connection.
         */
        private Weights mWeights;

        /** The layer from which the connection is made */
        private Layer mFromLayer;
```

```
/** The layer to which the connection is made */
private Layer mToLayer;

/** The number of neurons in the from layer */
private int mFromLayerSize = 0;

/** The number of neurons in the to layer */
private int mToLayerSize = 0;
/**
 * Instantiates a new Synapse connecting the neurons between
 * the from layer and the to layer
 * @param fromLayer The layer from which inputs should flow
 * @param toLayer The layer to which outputs should flow
 */
public Synapse(Layer fromLayer, Layer toLayer) {
	this(fromLayer, toLayer, new Weights(fromLayer.size(),
		toLayer.size()));
}

/** Instantiates a new Synapse connecting the neurons between
 * the from layer to the to layer
 * @param fromLayer The layer from which inputs should flow
 * @param toLayer The layer to which outputs should flow
 * @param weights The weights for each connection between neurons.
 * The size of the weights be consistent with the size of the fromLayer
 * and the toLayer.
 */
public Synapse(Layer fromLayer, Layer toLayer, double[][] weights) {
	this(fromLayer, toLayer, new Weights(weights));
}

/** Instantiates a new Synapse connecting the neurons between
 * the from layer to the to layer
 * @param fromLayer The layer from which inputs should flow
 * @param toLayer The layer to which outputs should flow
 * @param weights The weights for each connection between neurons.
 * The size of the weights be consistent with the size of the fromLayer
 * and the toLayer.
 */
public Synapse(Layer fromLayer, Layer toLayer, Weights weights) {
	if (weights.getFromLayerSize() != fromLayer.size() &&
			weights.getToLayerSize() != toLayer.size()) {
		throw new IllegalArgumentException("The size of the weights "
				+ "matrix should be consistent with the sizes of "
				+ "the input and output layers");
	}

	mFromLayer = fromLayer;
	mToLayer = toLayer;
	mWeights = weights;

	mFromLayerSize = mFromLayer.size();
	mToLayerSize = mToLayer.size();

	mFromLayer.addOutputListener(this);
}

/**
 * Sets each weight in the synape to be a random value
 * within a range of -amplitude to amplitude. For
 * example, if {@code amplitude == 0.4} then each weight
 * will be set to a value within the range of -0.4 and
 * 0.4
 * @param amplitude Each weight will be set to a random
 * weight whose value lies between amplitude and
 * -amplitude
 */
```

```
	public void randomiseWeights(double amplitude) {
		mWeights.randomize(amplitude);
	}


	/** Called when the from layer outputs a set of values to
	 * be passed onto the to layer
	 * @param output The output of the from layer to be passed
	 * onto the to layer
	 */
	@Override
	public void onOutput(double[] output) {
		double[] netInputs = calculateNetInputs(output);
		mToLayer.processInput(netInputs);
	}

	/** @return The layer to which data flows from the synapse */
	public Layer getToLayer() {
		return mToLayer;
	}

	/** @return The layer from which data flows into the synapse */
	public Layer getFromLayer() {
		return mFromLayer;
	}

	/** For each neuron in the to layer, this function
	 * will calculate the net input to each neuron in
	 * the to layer, by weighting every value and then
	 * summing the results.
	 *
	 * @param output The output of the from layer to be
	 * weighted and summed
	 * @return The weighted, biased, summed input to each
	 * individual neuron in the to layer
	 */
	private double[] calculateNetInputs(double[] output) {
		double[] result = new double[mToLayerSize];

		for (int i = 0; i < mToLayerSize; i++) {
			for (int j = 0; j < mFromLayerSize; j++) {
				result[i] += output[j] * mWeights.getWeight(j, i);
			}

			result[i] += mWeights.getBias(i);
		}

		return result;
	}

	/** @return The weights used in this synapse */
	public Weights getWeights() {
		return mWeights;
	}
}
```

The weights are held in the field `mWeights`, an instance of the `Weight` class which stores the values in a 2-dimensional `double[][]` array (which can be treated as a matrix). Weight also allows includes a method to randomise the weights between a specified range (whose signature is `randomize(double amplitude)`), as well as many other operations, and also holds the biases. The biases are treated as synapses to an "invisible" neuron (the "bias neuron") which always outputs 1. This way, the biases can be trained with minimum adjustment required to the code for it. The code for `Weights` is not included here as it is relatively straightforward – it can be found in `/src/com/henrythompson/neuralnets/Weights.java`.

Training is performed by Trainer classes. There is an abstract `Trainer` class which can be extended, so as to provide the opportunity to use different training algorithms in the future if needed. Here is the abstract `Trainer`:

/com/henrythompson/neuralnets/src/Trainer.java

```
public abstract class Trainer {
	/** The network this Trainer object is training */
	private AbstractNetwork mNetwork;

	/** The learning rate */
	private double mLearningRate;

	/** The layers contained in the network */
	private ArrayList<Layer> mLayers;

	/** The synapses contained in the network */
	private ArrayList<Synapse> mSynapses;

	/** Set to {@code true} if the current training run
	 * is aborted */
	private boolean mAborted = false;

	/** @param network The {@code Network} to be trained */
	public Trainer(AbstractNetwork network) {
		this(network, 0.1);
	}

	/** @param network The {@code Network} to be trained
	 * @param learningRate The learning rate to use */
	public Trainer(AbstractNetwork network, double learningRate) {
		mNetwork = network;
		mLearningRate = learningRate;

		mNetwork.registerTrainer(this);
	}

	/** Sets the learning rate
	 * @param rate The desired learning rate
	 */
	public void setLearningRate(double rate) {
		mLearningRate = rate;
	}

	/** @return The learning rate being used in training */
	public double getLearningRate() {
		return mLearningRate;
	}

	/** Prepares the {@code Trainer} object for training by
	 * receiving references to the layers and synapses to
	 * train
	 * @param layers The layers constituting the network to
	 * be trained
	 * @param synapses The synapses constituting the network
	 * to be trained
	 */
	public void registerNetwork(ArrayList<Layer> layers,
			ArrayList<Synapse> synapses) {
		mLayers = layers;
		mSynapses = synapses;
	}

	/** @return The final layer in the network to be trained */
	private Layer getOutputLayer() {
		return mLayers.get(mLayers.size() - 1);
```

```
}

/** This method will perform online training for a given set of training
 * samples
 * @param trainingSet The sample set to train the network
 * @param condition The criteria necessary for training to stop
 * @param listener Interface which is notified when certain training
 * events occur
 */
public void trainOnline(ArrayList<TrainingSample> trainingSet,
              StoppingCondition condition) {
       trainOnline(trainingSet, condition, null);
}

/**
 * This method will perform online training for a given set of training
 * samples. Any listeners will be notified when the relevant event occurs.
 * @param trainingSet The sample set to train the network
 * @param condition The criteria necessary for training to stop
 * @param listener Interface which is notified when certain training
 * events occur
 */
public void trainOnline(ArrayList<TrainingSample> trainingSet,
            StoppingCondition condition, TrainingProgressListener listener) {
       long start = System.nanoTime();

       sendTrainingStart(listener);
       condition.onTrainingStart(trainingSet, mNetwork);
       int epoch = 1;

       Layer outputLayer = getOutputLayer();

       while (true && !mAborted) {
              Collections.shuffle(trainingSet);

              for (TrainingSample sample: trainingSet) {
                     double[] output =
                            mNetwork.processInput(sample.getInput());
                     condition.onSampleTested(sample, output);
              }

              if (condition.shouldStop()) {
                     break;
              }

              for (TrainingSample sample: trainingSet) {
                     double[] output =
                            mNetwork.processInput(sample.getInput());
                     double[] difference = sample.getDifference(output);
                     outputLayer.setErrorGradients(difference);

                     performTraining();
                     sendSampleTrained(listener, sample);
              }

              sendEpochComplete(listener, epoch);
              condition.onEpochFinished(epoch);

              epoch++;
       }

       TrainingStatistics stats = new TrainingStatistics(epoch,
              System.nanoTime() - start);
       stats.setAborted(mAborted);
       sendTrainingComplete(listener, stats);
       mAborted = false;
}
```

```
/** Notifies a listener that training is starting
 * @param listener The listener to notify
 */
private void sendTrainingStart(TrainingProgressListener listener) {
	if (listener != null) {
		listener.onTrainingStart();
	}
}

/** Notifies a listener that a sample is trained
 * @param listener The listener to notify
 * @param sample The sample which has been trained
 */
private void sendSampleTrained(TrainingProgressListener listener,
		TrainingSample sample) {
	if (listener != null) {
		listener.onSampleTrained(sample);
	}
}

/** Notifies a listener that an epoch is completed
 * @param listener The listener to notify
 * @param epoch The current epoch
 */
private void sendEpochComplete(TrainingProgressListener listener,
		int epoch) {
	if (listener != null) {
		listener.onEpochComplete(epoch);
	}
}

/** Notifies a listener that training is finished
 * @param listener The listener to notify
 */
private void sendTrainingComplete(TrainingProgressListener listener,
		TrainingStatistics stats) {
	if (listener != null) {
		listener.onTrainingComplete(stats);
	}
}

/** Trains each synapse, starting at the final synapse
 * and moving its way forwards. The actual implementation
 * of the training algorithm is left to subclasses.
 */
private void performTraining() {
	mAborted = false;

	int n = mSynapses.size() - 1;

	for (int i = n; i >= 0 && !mAborted; i--) {
		Synapse synapse = mSynapses.get(i);
		train(synapse);
	}

	mAborted = false;
}

/** Aborts any currently running training */
public void abortTraining() {
	mAborted = true;
}

/** Returns an appropriate error gradient for each output
 * neuron and this training algorithm
 * @param actualOutput The actual output of the network
 * when the sample input was fed into it
 * @param sample The actual sample itself
```

```
         * @return The appropriate error gradients
         */
        public abstract double[] getOutputErrorGradients(double[] actualOutput,
                TrainingSample sample);

        /** The actual implementation of the training algorithm
         * @param synapse The {@code Synapse} to be trained
         */
        public abstract void train(Synapse synapse);
}
```

Different training algorithms are implemented as different subclasses, each of which must provide an implementation of the `train(Synapse synapse)` function. Here is the `GradientDescentTrainer`'s train function:

/com/henrythompson/neuralnets/src/GradientDescentTrainer.java

```
@Override
public void train(Synapse synapse) {
        Weights weights = synapse.getWeights();

        Layer fromLayer = synapse.getFromLayer();
        int fromLayerSize = fromLayer.size();

        Layer toLayer = synapse.getToLayer();
        int toLayerSize = toLayer.size();

        double[] lastInput = fromLayer.getLastOutput();

        // Learning rate may be varied over time, but it must be
        // constant over an epoch
        double learningRate = getLearningRate();

        for (int i = 0; i <= fromLayerSize; i++) {
                // If i == fromLayerSize then we want the bias neuron,
                // which always outputs 1
                double z_i = (i != fromLayerSize) ? lastInput[i] : 1;

                for (int j = 0; j < toLayerSize; j++) {
                        double error = toLayer.getErrorGradient(j);
                        double deltaw_ij = learningRate * error * z_i;

                        weights.adjustWeight(i, j, deltaw_ij);
                }
        }

        // Now set the error for the previous layer
        double[] delta_k = toLayer.getErrorGradients();
        double[] lastOut = fromLayer.getLastOutput();
        double[] derivatives = fromLayer.getActivationDerivative(lastOut);

        for (int i = 0; i < fromLayerSize; i++) {
                double delta_j = 0;

                for (int j = 0; j < toLayerSize; j++) {
                        double w_jk = weights.getWeight(i, j);
                        delta_j += derivatives[i] * delta_k[j] * w_jk;
                }

                fromLayer.setErrorGradient(i, delta_j);
        }
}

@Override
public double[] getOutputErrorGradients(double[] actualOutput, TrainingSample
                sample) {
```

```
        return sample.getDifference(actualOutput);
}
```

Several other classes were also created, but are not included here. These include the `StoppingCondition` class, which is an abstract class which encapsulates the conditions required for a training run to stop – it can be overridden to provide many different stopping criteria. The `TrainingSample` class encapsulates a single training sample, with its input values and expected outputs as well as functions to calculate various error fuctions, such as SSE, CEE and absolute difference. These can be found in the source code.

Perceptron

A perceptron is just a neural network with two layers of MPNs, which must of course be connected by a Synapse. Since the MPNs use a threshold activation function, the `activationFunction(double[] netInput)` method has been overridden in a subclass of `Layer`, called `ThresholdLayer` to reflect this:

`/src/com/henrythompson/neuralnets/ThresholdLayer.java`

```
@Override
protected double[] activationFunction(double[] netInput) {
        int n = size();
        double[] output = new double[n];

        for (int k = 0; k < n; k++) {
                output[k] = netInput[k] > 0 ? 1.0 : 0.0;
        }

        return output;
}
```

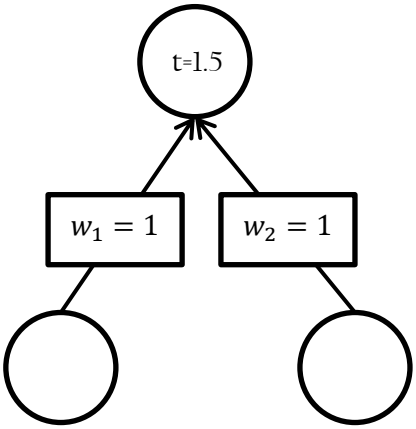
I have also written a class `Perceptron` which uses a Builder pattern (`Perceptron.Builder`) to easily build and train a perceptron for the sake of convenience. The code for this has not been included here, but can be found at `/src/com/henrythompson/neuralnets/Perceptron.java`.

Solving AND and OR

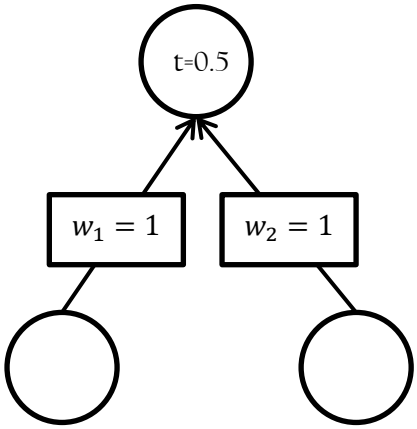
We begin with a simple problem: produce perceptrons capable of logical AND and OR operations, the truth tables for which are as follows -

i_1	i_2	$i_1 AND\ i_2$	$i_1 OR\ i_2$
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	1

In order to do this we need the following perceptrons:



AND



OR

Using the `Perceptron` class and builder mentioned above, the code for this extremely simple:

```
public static void andOrTest() {
	double[][] weights = new double[][] {{1}, {1}};

	// Set up the perceptron for AND
	Perceptron and = new Perceptron.Builder(2, 1)
		.setWeights(weights)
		.setBiases(new double[] {1.5})
		.create();

	// Set up the perceptron for OR
	Perceptron or = new Perceptron.Builder(2, 1)
		.setWeights(weights)
		.setBiases(new double[] {0.5})
		.create();

	// Test their outputs
	for (int i = 0; i < 4; i++) {
		double[] input = new double[]{i % 2, i < 2 ? 0 : 1};
		double[] outputAnd = and.processInput(input);
		System.out.println(input[0] + " AND " + input[1]
			+ " = " + outputAnd[0]);

		double[] outputOr = or.processInput(input);
		System.out.println(input[0] + " OR " + input[1]
			+ " = " + outputOr[0] + "\n");
	}
}
```

As expected, the output of the program to the console was:

```
0.0 AND 0.0 = 0.0
0.0 OR 0.0 = 0.0

1.0 AND 0.0 = 0.0
1.0 OR 0.0 = 1.0

0.0 AND 1.0 = 0.0
0.0 OR 1.0 = 1.0

1.0 AND 1.0 = 1.0
1.0 OR 1.0 = 1.0
```

Training the Perceptron to Classify Points On A Graph

Using a perceptron, it is possible to classify points as existing in the region on one side of a linear graph, or the other. To keep things simple, we will begin by trying to produce a neural network to classify whether some points lie above or below the line $y = x$. In order to do this, we will generate 1,000 random sample points to use as our test set, and then use online training on a Perceptron with a signum output layer. The samples, pseudorandomly generated using the `Math.random()` method, should be well spread as they follow a normal distribution, according to the Java documentation, and so should be a reliable indicator of the performance of the network when the results of multiple trials are averaged out. This test can be run by starting the program with the command-line argument `perceptron_test [rate]`, where `[rate]` should be replaced with the learning rate and is optional (default rate is 0.1).

/src/com/henrythompson/neuralnets/NeuralNetworks.java

```
public static void perceptronTest(double rate) {
	Perceptron perceptron = new Perceptron.Builder(2, 1)
	.useSigmoidOutputLayer(false)
	.setLearningRate(rate)
	.create();

	ArrayList<TrainingSample> samples = generateTrainingSamples(1000);
	perceptron.trainOnline(samples, 5);
}

private static ArrayList<TrainingSample> generateTrainingSamples(
	int number) {
	ArrayList<TrainingSample> samples = new ArrayList<TrainingSample>();

	for (int i = 0; i < number; i++) {
		double x = Math.random() * 10;
		double y = Math.random() * 10;

		double output = x > y ? 0 : 1;
		samples.add(new TrainingSample(
			new double[]{x, y}, new double[]{output}));
	}

	return samples;
}
```

The "perfect" set of weights are when $w_{11} = -x$, then $w_{21} = x$. Using the stopping condition that training stops only when all 1000 training samples are outputted with a SSE of 0, the program learnt remarkably quickly and accurately. For example, using a learning rate of 0.1 and random weight amplitude of 0.2, outputs were as follows (on a laptop running an Intel Core i3-330M CPU and 8GB of RAM):

Run 1:

```
Samples: 1000
...
Training finished after 5 epochs. SSE = 0.0
Weights: -4.818898651406216, 4.829165680267481
Time taken: 00.156s
```

Run 2:

```
Samples: 1000
...
Training finished after 2 epochs. SSE = 0.0
Weights: -4.03035694179174, 4.02012585424517
Time taken: 00.105s
```

Run 3:

```
Samples: 1000
...
Training finished after 11 epochs. SSE = 0.0
Weights: -6.886635633002313, 6.891685009229402
Time taken: 00.240s
```

Run 4:

```
Samples: 1000
...
Training finished after 8 epochs. SSE = 0.0
Weights: -6.970751044261585, 6.961580480363131
Time taken: 00.145s
```

Run 5:

```
Samples: 1000
...
Training finished after 1 epochs. SSE = 0.0
Weights: -1.9726365546274467, 1.9734236637823528
Time taken: 00.074s
```

As can be seen, the values of the weights calculated are very accurate, always to within at least 2 significant figures of the ideal weights. They are also calculated quickly – it took, on average, approximately 45ms per epoch. Since this required training over 1000 samples each time, this is reasonably fast, especially considering how Java is relatively slow due to the overhead of its Virtual Machine. However, the number of epochs the neural network takes to meet the stopping condition can vary largely – in this simple example alone, the number of epochs required varied from 1 to 11. Whilst this is a matter of milliseconds in this example, for more complex networks which require millions or even billions of epochs to train, the time difference may be hours or even days: since we are using random weights and samples, it is not possible to predict with any accuracy how long the training will take. In any given run, it may be that the chosen weights so happen to make training extremely slow, and the only way to know is the rerun the entire training algorithm. Furthermore, it stands to reason that using far fewer samples will require more epochs to train using this method, though clearly an appropriate spread of samples will be required improve the training.

Multi-Layer Feed-Forward Neural Network

We can arrange and manipulate the Layer and Synapse classes differently to produce a multi-layer feed-forward neural network (FFNN). First of all, we need a subclass of `Layer` for Sigmoid layers:

/src/com/henrythompson/neuralnets/SigmoidLayer.java:

```
@Override
protected double[] activationFunction(double[] netInput) {
	int n = size();
	double[] output = new double[n];
```

```
        for (int k = 0; k < n; k++) {
                output[k] = 1 / (1 + Math.exp(-netInput[k]));
        }

        return output;
}
```

We then need one for Softmax layers:

/src/com/henrythompson/neuralnets/SigmoidLayer.java

```
@Override
protected double[] activationFunction(double[] netInput) {
        int n = size();

        double sum = 0;
        double[] output = new double[n];

        for (int k = 0; k < n; k++) {
                output[k] = Math.exp(netInput[k]);
                sum += output[k];
        }

        for (int k = 0; k < n; k++) {
                output[k] = output[k] / sum;
        }

        return output;
}
```

It is also convenient to have a class which will build and manage the layers of FFNN. The class `FeedForwardNeuralNetwork.Builder` is used to easily construct an FFNN and is used to produce a new instance of `FeedForwardNeuralNetwork`, which manages the layers and the training. The code for the builder is not included as it is not strictly relevant to the actual neural network, though to demonstrate its use here is an example of constructing an FFNN:

```
// Construct a FFNN with input layer with 5 neurons, output
// layer with 6 neurons, and 3 hidden layers with 7, 5 and 4
// neurons respectively
FeedForwardNeuralNetwork.Builder builder =
        new FeedForwardNeuralNetwork.Builder(5, 6, new int[]{7,5,4});

FeedForwardNeuralNetwork ffnn = builder.create();
```

Solving XOR

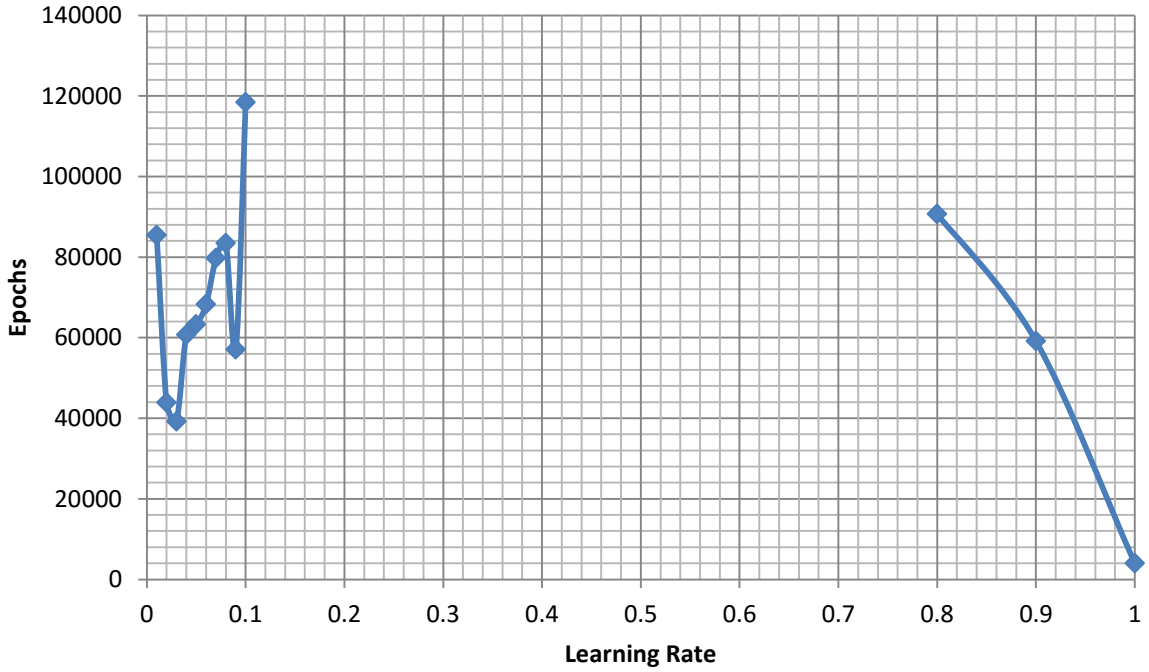
We now have everything required to produce an ANN which can learn XOR. We will have one `LinearLayer` for the input, and two `SigmoidLayers` for the hidden and output layers, set up in a biased 2-2-1 configuration. In order to make the code cleaner, the class `XORNetwork` is responsible for creating and maintaining the network used. The XOR test can be run by running the program with the command line argument `xor_test`.

Using a learning rate of 0.5 and randomised weights of amplitude 0.2, the network learnt XOR in 580 epochs as a mean over 100 runs to an accuracy of $RMSE = 0.05$. The mean time was 0.035s. Again, considering that this is running on Java on a laptop-grade processor, this is a reasonable performance.

For further analysis, I have shown how the mean number of epochs it took to solve XOR to within $RMSE = 0.05$ with different learning rate and initial weight parameters, as a mean over 100 runs for each learning rate:

Randomised Weights Amplitude: 0.001

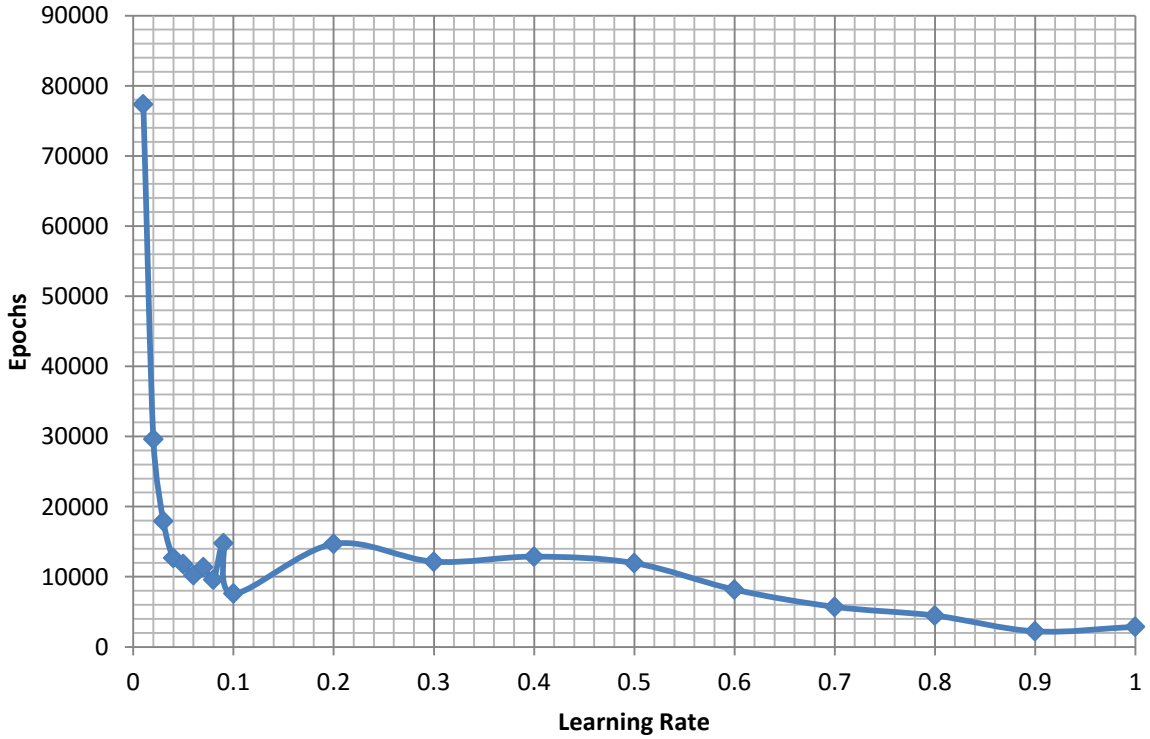
Learning Rate	Epochs
0.01	85,415.12
0.02	43,892.88
0.03	39,143.59
0.04	60,726.00
0.05	63,212.54
0.06	68,226.55
0.07	79,710.67
0.08	83,457.17
0.09	57,077.00
0.1	118,424.00
0.2	>200,000
0.3	>200,000
0.4	>200,000
0.5	>200,000
0.6	>200,000
0.7	>200,000
0.8	90,637.00
0.9	59,142.56
1	3,986.88



Interestingly, between $0.1 < \varepsilon < 0.8$, the program never managed to complete within 200,000 epochs (beyond this, the program terminated that training run and started on the next one).

Randomised Weights Amplitude: 0.1

Learning Rate	Epochs
0.01	70955.86
0.02	26441.56
0.03	14100.19
0.04	9398.2
0.05	8876.56
0.06	5539.48
0.07	4530.65
0.08	4200.72
0.09	3560.719
0.1	2916.04
0.2	1973.115
0.3	2138.557
0.4	995.7347
0.5	1279.354
0.6	580.3053
0.7	574.202
0.8	656.0417
0.9	528.72
1	1225.1

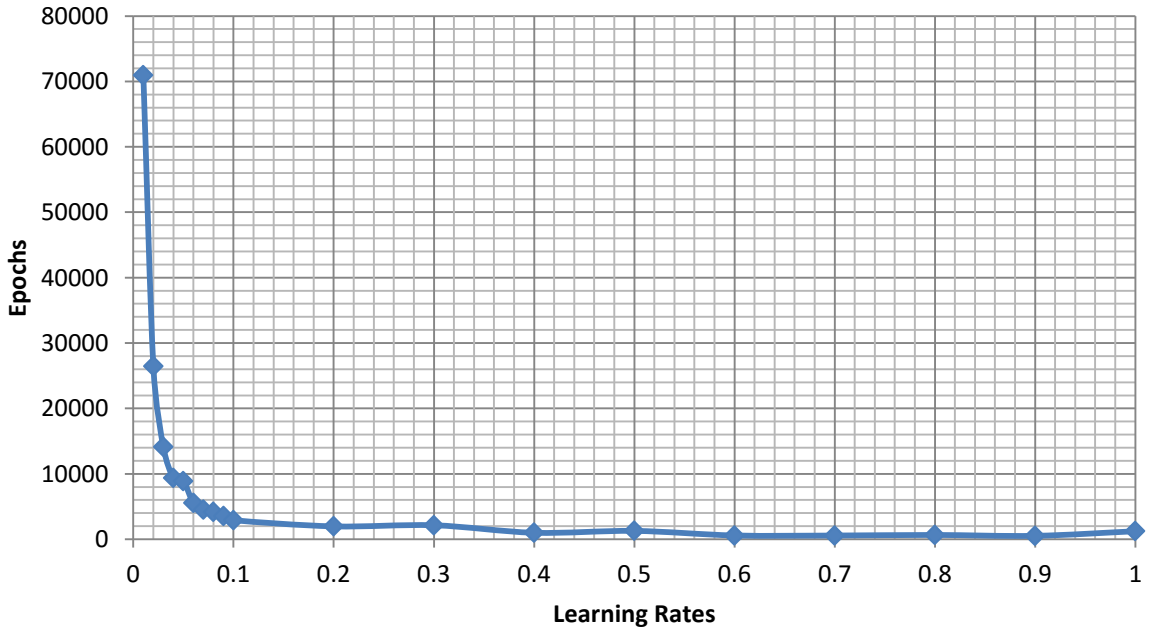


There is a clear trend here that, in general, the greater the learning rate, the faster it learns on average. This is, of course, to be expected. There are, however, fluctuations in the data: sometimes, it takes

significantly longer to train with a higher learning rate. The could well be simply random variations, though it is not yet clear what the underlying cause of them is.

Randomised Weights Amplitude: 0.1

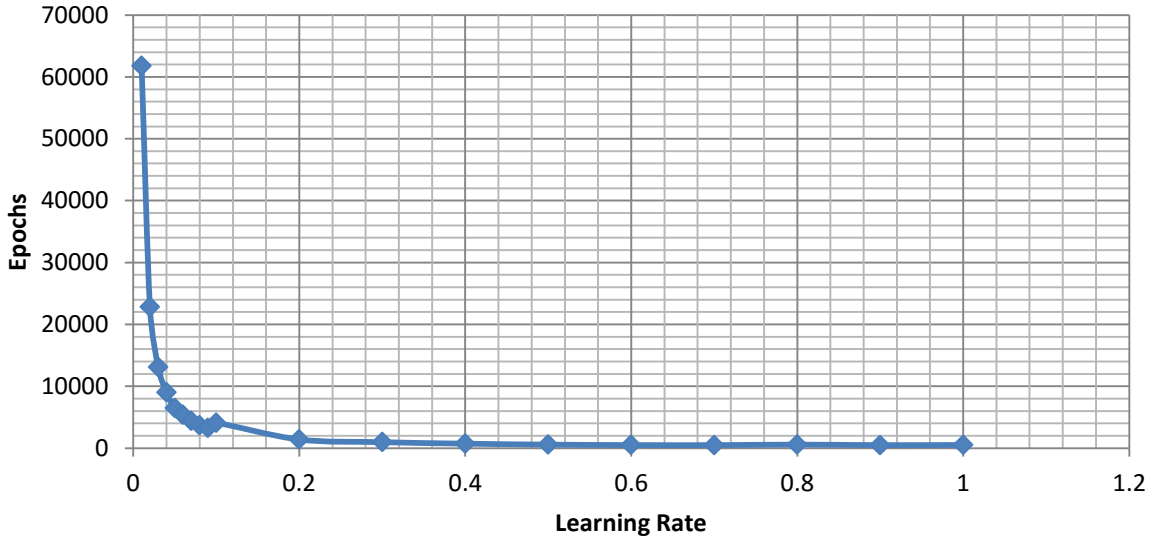
Learning Rate	Epochs
0.01	70955.86
0.02	26441.56
0.03	14100.19
0.04	9398.2
0.05	8876.56
0.06	5539.48
0.07	4530.65
0.08	4200.72
0.09	3560.719
0.1	2916.04
0.2	1973.115
0.3	2138.557
0.4	995.7347
0.5	1279.354
0.6	580.3053
0.7	574.202
0.8	656.0417
0.9	528.72
1	1225.1



This time, there is a far smoother graph. Furthermore, the average number of epochs required is often extremely low, reaching a minimum of 528 (for $\eta = 0.9$).

Randomised Weight Amplitude: 0.2

Learning Rate	Epoch
0.01	61789.36
0.02	22848.63
0.03	13111.67
0.04	8996.367
0.05	6463.643
0.06	5400.241
0.07	4407.16
0.08	3676.95
0.09	3277.244
0.1	4091.329
0.2	1405.696
0.3	973.9271
0.4	735.0612
0.5	580.7172
0.6	509.0103
0.7	498.1616
0.8	582.9381
0.9	497.1717
1	521.03

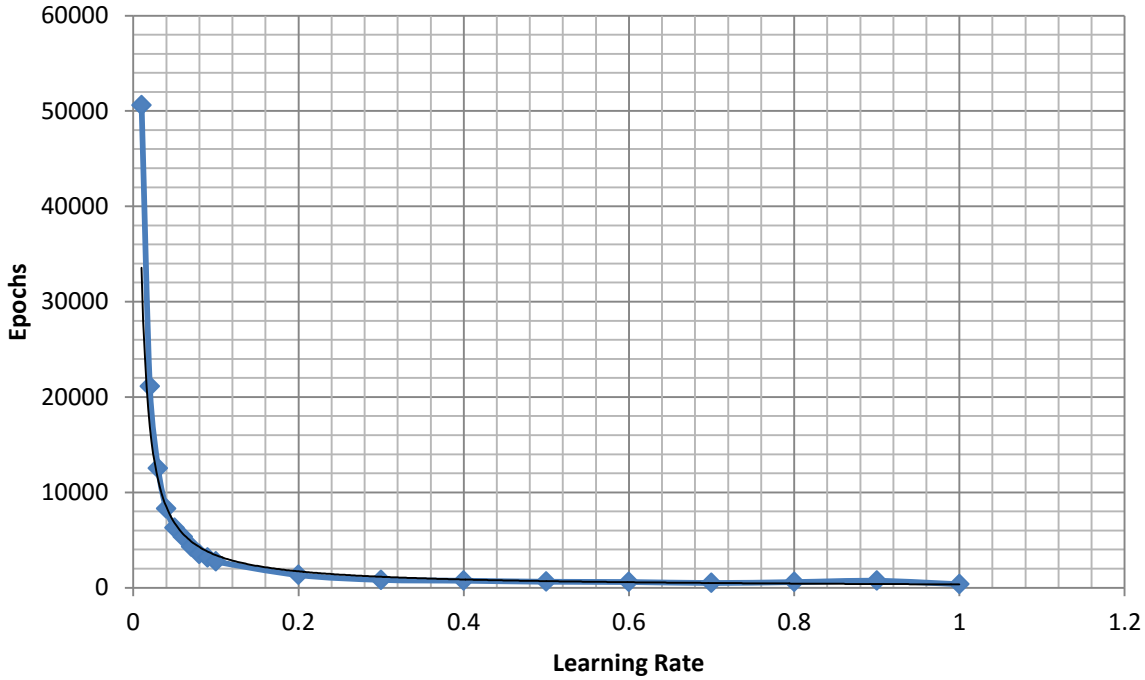


The graph is smoother yet, except the result at $\eta = 0.9$. The minimum number of epochs required has again dropped, this time to 497 (for $\eta = 0.9$).

Randomised Weight Amplitude: 0.3

Learning Rate	Epoch
0.01	50593.67
0.02	21113.81
0.03	12521.37
0.04	8286.121
0.05	6276.444

0.06	5337
0.07	4285.772
0.08	3539.972
0.09	3162.412
0.1	2744.948
0.2	1330.662
0.3	820.7595
0.4	733.2593
0.5	623.9405
0.6	598.232
0.7	493.967
0.8	592.1047
0.9	726.1348
1	358.39

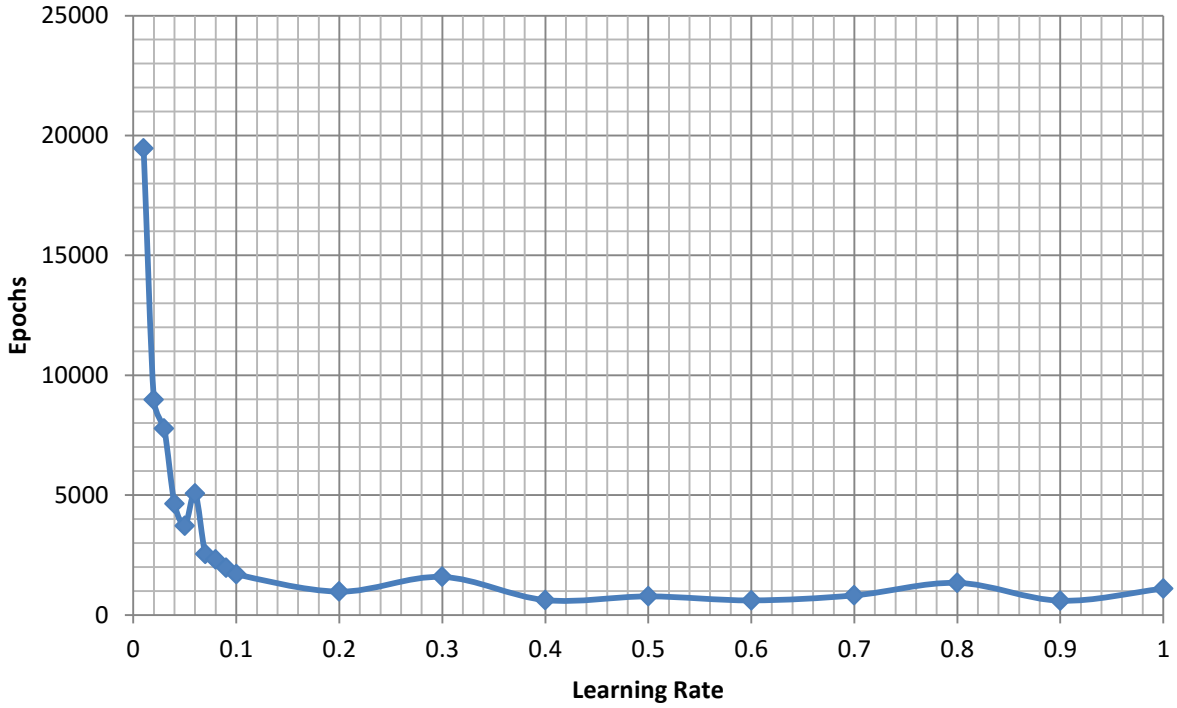


The graph is even smoother again and very closely follows $y = \frac{348}{x}$ (shown in black). The minimum number of average epochs required was 358.39 when $\eta = 1.0$. Each epoch took 0.013s on average.

Randomised Weight Amplitude: 1.0

Learning Rate	Epochs
0.01	19464.06
0.02	8972.03
0.03	7774.96
0.04	4630.07
0.05	3714.83

0.06	5065.87
0.07	2541.22
0.08	2304.94
0.09	1968.35
0.1	1695.88
0.2	970.28
0.3	1586.13
0.4	619.03
0.5	773.53
0.6	606.2
0.7	815.88
0.8	1337.97
0.9	592.99
1	1098.07



Interestingly, the graph has now become less smooth, with unexpected peaks at $\eta = 0.06, 0.3, 0.8$. The number of epochs required reached a minimum of 523 when $\eta = 0.9$, twice as great as when the weights were 358; however, the maximum was 19,464 which is a significant decrease from the other results.

Overall Analysis

There is a clear general trend within each of the graphs, except when the randomised weights amplitude was 0.001, that as learning rate increases, the number of epochs required to train it decreases. This is expected, as greater learning rates means that the stochastic gradient descent takes greater steps in each epoch.

However, there are many apparently random fluctuations in the number of epochs required in each graph. Curiously, they are greatest when the random weights amplitude is extremely small, then smooth out at around $\eta = 0.3$ to an approximately $\frac{1}{x}$ relationship before, beyond some point, increasing again.

This experiment does highlight the importance of choosing appropriate randomised weights and learning rates, and the huge effects that choosing the incorrect one may have on learning performance. Unfortunately, it is almost impossible to predict these without a trial and error approach as above.

Whilst it is hard to explain all of the observations, there are a number of possible explanations for some of the observed results.

- When the random variations in the weights are so small, training takes a long time to get going, since the original outputs from each neuron will be small and the weight update equations rely on those values. After several epochs, they become large enough that the weight updates become significant; however, until then, many epochs will pass without much change in the error.
- Sometimes, however, it took longer to train when the initial weights grew ever larger. This can be accounted for by the fact that the 2-2-1 XOR error gets flatter and flatter the further from the global minimum you travel landscape (Boers E.J.W., Sprinkhuizen-Kuyper Ida G, 1998); as a result, if the weights started out such that they were far from the global minimum, the training may become extremely slow in certain cases.
- Since the results were averaged over 100 different runs, any random fluctuations should have been minimised. The pseudorandom weights generated using the `java.lang.Math.random()` function, however, produces number which are distributed along an approximately normal distribution, according to the Java documentation. Since this does not provide an even distribution, this may have skewed the results slightly, as the weights were likely to be in the middle of the range between zero and the amplitude. However, the general trend would still have been achieved: as the amplitude increases, so do the weights (on average).
- As a result, when seeing such strong fluctuations when the random weights had amplitude of 1.0, it suggests that this is a reflection of the error landscape of the 2-2-1 XOR network. Since there are no local minima in the XOR landscape landscape (Boers E.J.W., Sprinkhuizen-Kuyper Ida G, 1998), this cannot account for the fluctuations. However, at certain learning rates, the results may have tended to oscillate around the global minimum due to the step sizes coinciding with the diameter of the minimum.
- That said, the fluctuations occur for completely different learning rates depending on the initial weights used. It can only be assumed therefore that either the starting positions affect which learning rates are most likely to cause this bouncing effect, or there is some other unknown factor affecting these results.

However, it is extremely hard to judge the reliability of these results. Although the general trends are clear, it appears to be that they are extremely affected by random variations. Suspecting this to be the case, I reran the test for when the random weights amplitude is 1.0 and was surprised to find that the minimum average number of epochs dropped to 258 – half of what it was according to the initial test. As a result, although it seems safe to draw conclusions on the general trends, it is extremely hard to analyse the network's specific behaviours.

This is, of course, one of the well-known problems with neural networks. They are entirely unpredictable and their performance at any instance is highly dependent on uncontrollable, random factors. Nonetheless, considering how these results were the mean of 100 attempts, it was surprising to see just how significant this was.

Analysis of Implementation

In terms of speed, the implementation seems quite good. The fastest training took just 281 epochs, completing in 2,789,842ns (approximately $2.79 \times 10^{-3}s$). Averaging not just this run but the figures for 100 runs suggests that each epoch takes approximately 10μs to process. Again, using just a single core of the "mobile"-grade 2011 quad-core 2.3GHz Intel Core i3-330M (though, according to the task manager it was running at only 900MHz during the processing), this is very good as it would suggest that training should not take unreasonably long, even when large numbers of epochs (i.e. several tens of millions) are required, and especially considering that this is not native code.

Of course, there are a number of obvious ways to speed up the processing of the neural network. It could be rewritten so as to compile to native code using a language such as C or C++. Furthermore, the modularised structure of the code introduces the overhead of extra method calls, inheritances, listener objects and various other interfaces. That said, the usefulness of the flexibility achieved through this structure far outweighs the loss in performance, which is slight at most.

Perhaps a more significant performance boost would be running calculations in parallel: since each layer's calculations run independently of each other, each layer could run in a separate thread. It would receive its input from the thread for the previous layer, process it, and then pass its output onto the next thread before receiving its next input from the previous thread again. Running on a quad-core laptop, where calculations could genuinely occur simultaneously, this could see some significant improvements in speed. Nonetheless, it would have to be extremely carefully coded to minimize locking shared objects and to avoid concurrency errors. Furthermore, as different layers of different sizes may take different times to run, it is conceivable that some threads may spend most of their time waiting for the previous layer to notify them of completion, holding back threads further down as well. As such, a single thread could prove to be a bottleneck to the entire performance of the multithreaded network. If not properly implemented, the overhead and complications of object locking, threads waiting and thread notifications could considerably impact performance, and even lead to an overall performance drop. A potential way to minimize this would be to have only as many threads as there are cores in the processor and to give each thread multiple adjacent layers to process. It would be sensible to balance the load of each thread by taking into account the relative sizes of each layer. This would be an instance of the "Partition Problem" and could be solved using a dynamically programmed algorithm by minimizing the cost function of each partition. In any case, it would be more complex (though certainly possible) to code – however, the current neural network has not be designed to run in parallel, and converting it would require significant restructuring.

In terms of design, the implementation is working very well. Having been deliberately designed to be flexible and modularised, the interface of each component has been separated completely from their implementations, and different functionality has been completely separated out. Heavy utilisation of "listener" interfaces allows external classes to monitor the progress and behaviour of each network as it runs. This is proving to be very useful, taking no time at all to swap in and out different networks, layers, training methods and stopping conditions by changing less than a single line of code each time.

That said, it has been hard to anticipate every scenario, and significant changes to the code were required so that it would allow outputting the results to files. Although having different components separated out is great for flexibility, without the appropriate listeners (and it can be very messy if there are too many of them), outputting to CSV files using a single `PrintWriter` when all the results were being produced by different, entirely separate components proved to be difficult. The lazy solution of passing around the `PrintWriter` object would easily become very messy.

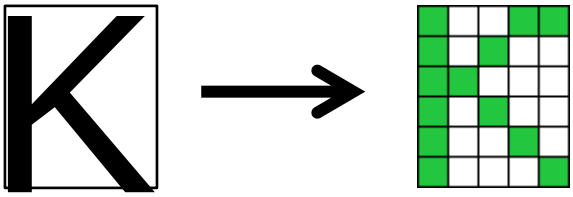
In the end, the best solution turned out to be to create a new `StoppingCondition` object, the `stoppingconditions.FileOutputRMSECondition` class, to output the error information as training occurred. Although this required modifying some of the events it was notified about from the `Trainer` class, this was a far cleaner solution. Additional information, such as time taken to train, was saved to a separate file by listening in on the gradient descent's events by a separate interface. This meant that the introduction of a `TrainingStatistics` class was also required, so that the `Trainer` classes would return statistics regarding the training which could then be saved by the pre-registered listeners. This proved to be an effective and clean way to save information from various classes whilst minimizing refactoring of the code required.

Multi-Class Classification: Learning to Recognise Letters of the Alphabet

One of the areas in which neural networks have been especially successful is pattern recognition, and within this field they have been shown to be especially adept at tasks such as OCR (optical character recognition). Even simple neural networks have been shown to be sufficient to accomplish this, and as a result I felt I would like to attempt to do this. I also felt that the implementation was robust and now well-designed enough for this to potentially work.

There have been three main methods of converting images into inputs for neural networks:

1. **Each pixel forms an input**: This technique, however, often requires huge inputs. An image as small as 100x200 will have 20,000 inputs. Furthermore, it has often been shown that pixels are too fine grained – we are more interested in the overall shape of letters, and since pixels are so small, it is rare that even the presence or absence of a single pixel will always correspond to a single letter or subset of letters. As a result, it is extremely difficult for neural networks to be able to properly classify letters this way.
2. **Divide the image up into a large grid**: This is a commonly employed strategy and has been shown to work very well, as it captures the general shape of the letter. The grids are often only of tiny resolution, such as 5x6, and can only take on values of either 1 (pixel) or 0 (no pixel). However, the bipolar representation (0.5 and -0.5; or 1 and -1) is often preferred as it leads to faster and more accurate training. For example:



   Whilst this does work relatively well, there are potentially issues regarding how to scale images of different sizes, especially when they contain anti-aliasing. Nonetheless, edge-

detection and noise-reduction algorithms, such as the Canny algorithm and Non-Local Means algorithm, can overcome these problems.

3. **Use Receptors:** This avoids the issue of rescaling images, but its accuracy is uncertain. It involves using arbitrarily sized and positioned vectors across the image, which are assigned a value of "0.5" if they cross any part of the letters, or a value of "-0.5" if they do not.

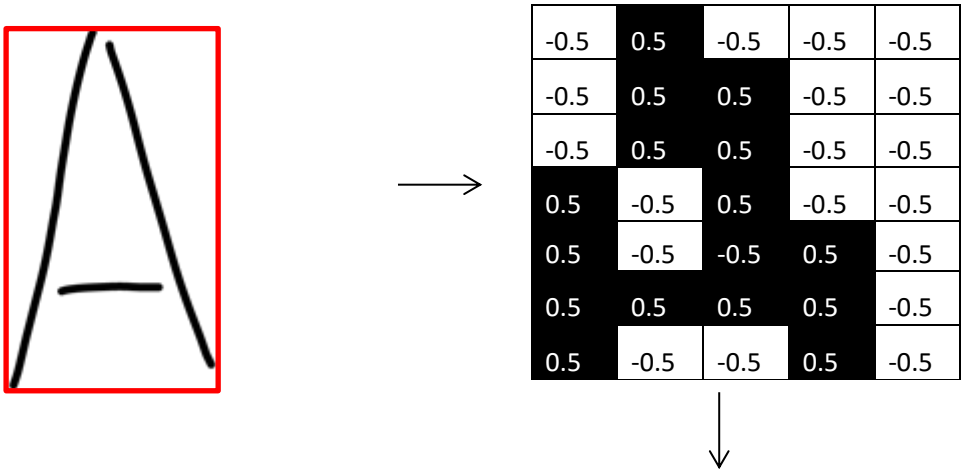
The vectors used are specifically chosen from a large set of randomly generated ones, as they have the greatest data entropy across all letter samples – in other words, the ones which show greatest consistency of being assigned either 0.5 or -0.5 for any given letter, but the least consistency of being assigned either 0.5 or -0.5 across all letters, are used. This makes sense, as these lines are therefore always characteristic of certain letters but not of others, and therefore the idea is that this approach picks up the most characteristic features of each letter.

It is unclear from the literature whether the receptors or grid-based method is better. However, the grid based method is far more common and has been used in professional applications (such as the US postal system's automatic ZIP code reader). As a result, this is the approach I will take. For the sake of simplicity, I will begin by only attempting to get the network to recognise capital letters.

1. Reducing the letter to a grid

To begin with, I will use a sample of different letters. Since I would like the network to recognise handwritten letters, the samples will be drawn onto a smartphone screen – in order to do this, I have written a small Android smartphone application to do this, and convert the images into an array of 0.5's and -0.5's. (Source at `/LetterGrid/src`)

This is done by cropping the drawn image to remove surrounding whitespace, scaling it up/down whilst maintaining the aspect ratio so it fits the entire height or width of a 500x700px box, then dividing it into a 5x7 grid. Each cell has a value of 0.5 if more than 8% of the pixels in it are black; otherwise it has a value of -0.5. (The 5x7 grid was chosen as this seemed to be the most common successfully implemented grid size in the literature; 8% was chosen through experimentation, as it seemed to remove just the sufficient amount of unnecessary detail). The values are then arranged in a vector, read off going from left to right, top to bottom.



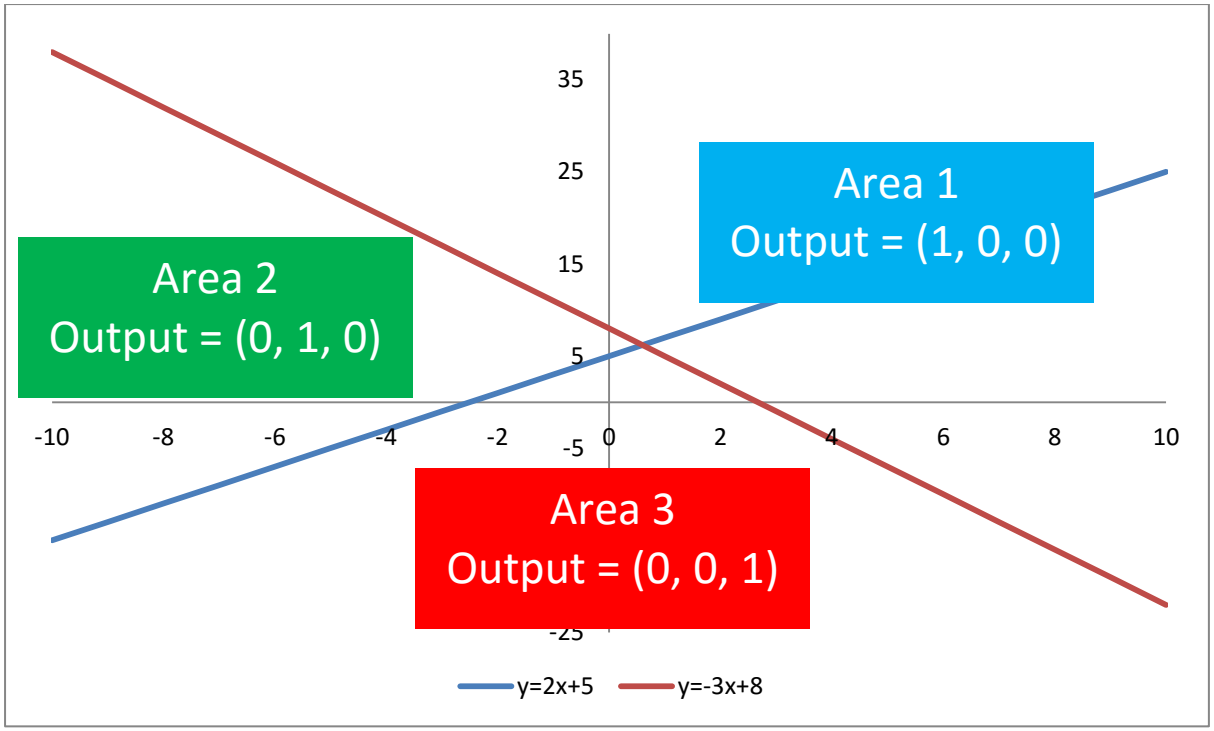
-0.5,0.5,-0.5,-0.5,-0.5,-0.5,0.5,0.5,-0.5,-0.5,-0.5,0.5,0.5,-0.5,-0.5,0.5,-0.5,0.5,-0.5,-0.5,0.5,-0.5,-0.5,0.5,0.5,-0.5,0.5,0.5,0.5,0.5,-0.5,0.5,-0.5,-0.5,0.5,-0.5,

13 samples of each letter were produced this way. 10 of these samples would be used to train the neural network; the 3 remaining ones would be used as unseen samples to test the relative performances of the different networks produced.

2. Testing the MultiClass ANN

Before running the learning algorithm, a test was done to confirm that the multiclass network class worked correctly. It was found that the weights grew increasingly large, until the computer could no longer handle the size of the numbers involve due to the exponentiation involved in the Softmax layer (after $e^{\sim 700}$ Java would assign this a value of `Infinity` and the CEE error became `NaN`, beyond which training was not possible). As a result, every output is now multiplied by a very small number, (e.g. 0.001) before training – as long as the weights remain approximately proportionally the same, training can still occur correctly, though this was to a limit. Sometimes, when this method was used, training would not be able to get below a CEE as high as 0.3 for each sample. That said, the smaller the number, the slower the training, so this was kept to the largest number possible. In this case, that was 0.1.

After this was resolved, the algorithm worked flawlessly. Using 100 random training points, a 2-5-3 neural network composed of a Linear layer, Sigmoid layer and Softmax layer was able to classify them into the following three areas:



It was able to do so on to within a total SSE error of 0.05 in just 2 epochs, taking an average time of 0.0248s to do so. This was a very pleasing result, as it meant that the training required extremely few epochs and was fast: despite having as many as 100 training samples and the extra computation required for the hidden Sigmoid layer, and more computationally complex Softmax layer, each epoch took an average of just 0.0124s. Also, after running the algorithm 20 times, it not once got caught in a local minimum.

3. Teaching a Network to Recognise Letters

Now that the multiple class classification is known to work, it is possible to begin training the network to recognise characters. Unfortunately, we don't know the optimal parameters to minimise training time, avoid local minima, and get the best generalisation.

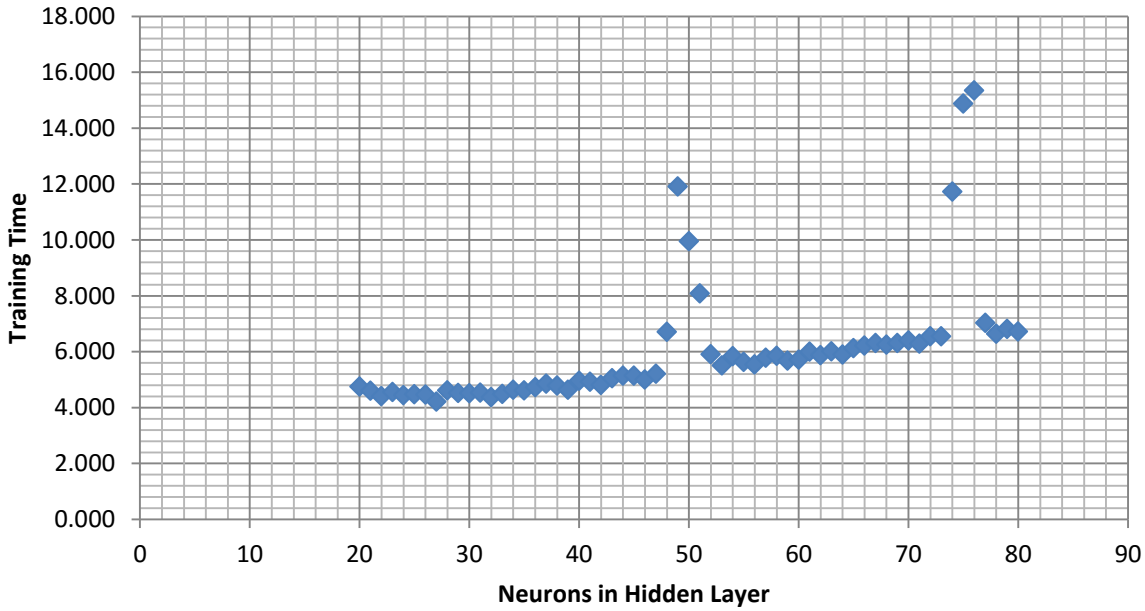
As a result, the OCR test class (accessible using `ocr_test` argument from the command line) automatically tries out different combinations of networks, parameters and early stopping points to find the optimum. The network will always consist of 35 inputs, one for each cell, and 26 outputs, one for each layer. However, the network will be automatically varied as follows:

- The program will attempt different layers of hidden neurons, beginning with just 20 neurons and increasing it until 80. Preliminary tests showed that the algorithm never converged with 2 or 3 hidden layers.
- Each time, random weights will be set to 0.3. The learning rate will be set to a relatively small 0.1 so as to minimize "bouncing" about the minima.
- When the average CEE error for each training sample reaches a certain set of points (3.25, 3.0, 2.75, 2.5,...,0.005, 0.0025, 0.001) during the training, then the weights at that point will be saved to disk in an XML format, such that the ANN at that point can be reconstructed. In order to do this, I have written the classes `NetworkExporter` and `NetworkImporter`.
- If a network makes it below an average CEE of 0.001, it is finished and we move onto the next configuration.

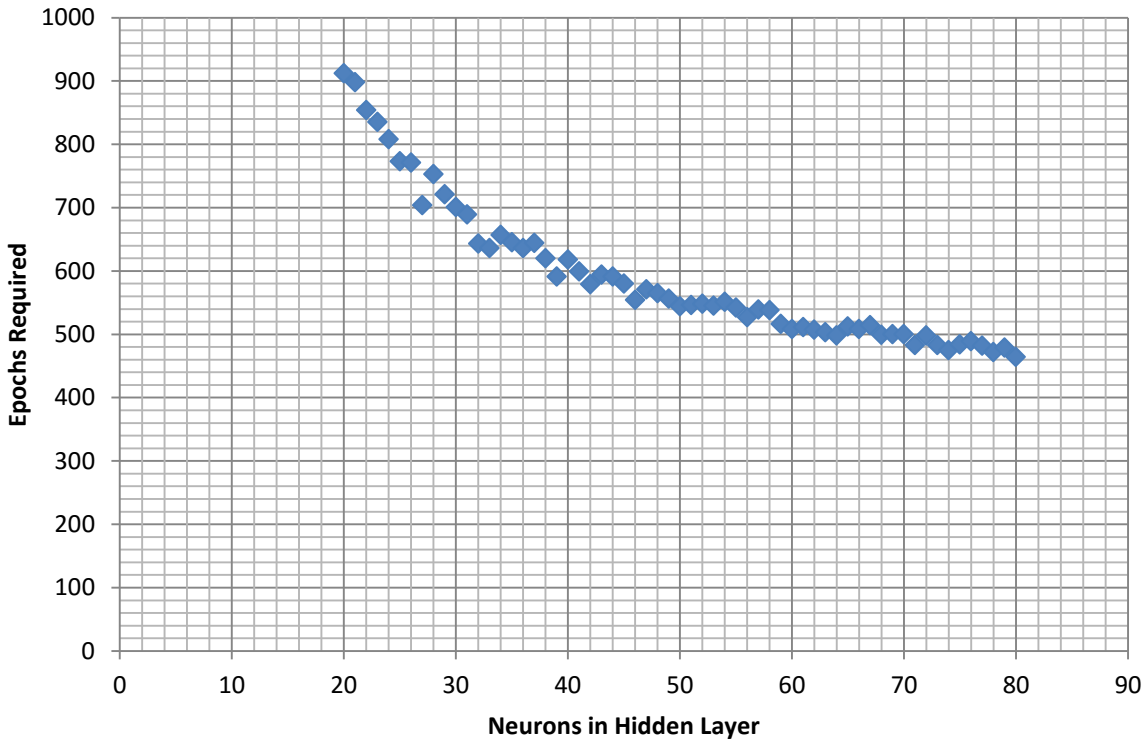
Following this, we will have generated several possible neural networks with varying accuracy and generalisation. Each network will then be tested by presenting each one with the same set of unseen letter images, and then testing the accuracy with which it correctly determines the letter (this is the letter given the highest value in the output Softmax layer). The one which achieves the greatest accuracy in recognising the letters wins.

Training was remarkably fast despite having 260 training samples, 35 inputs, 26 outputs and up to 141 neurons at any one time. In fact, with a training rate of 0.2, a network with hidden layer size 76 trained in 1.21 seconds to an accuracy of average CEE = 0.001!

However, a learning rate of 0.1 gave better results overall, with an average learning time of 6.055s, with the fastest being 4.22s for a hidden layer of 27 neurons. In general, the larger the network, the slower the training, save for some anomalous results:



Clearly, this was to be expected. However, in any case, learning was extremely fast and this should not be an issue, even with the anomalous results, none of which are greater than 16s. Most interesting about the anomalous results was that they were clustered around two particular regions of the graph, suggesting that there may be some underlying reason in the structure of those networks which meant that training was slower. However, analysing the number of epochs required offered a different explanation – and it was also perhaps unexpected that the number of epochs required to train decreased as the number of neurons in the hidden layer increased:



Since the anomalous increases in time do not correspond to increased numbers of epochs, it seems likely, therefore, that the increased time required was in fact just because some other process on the computer interrupted the training during those two points.

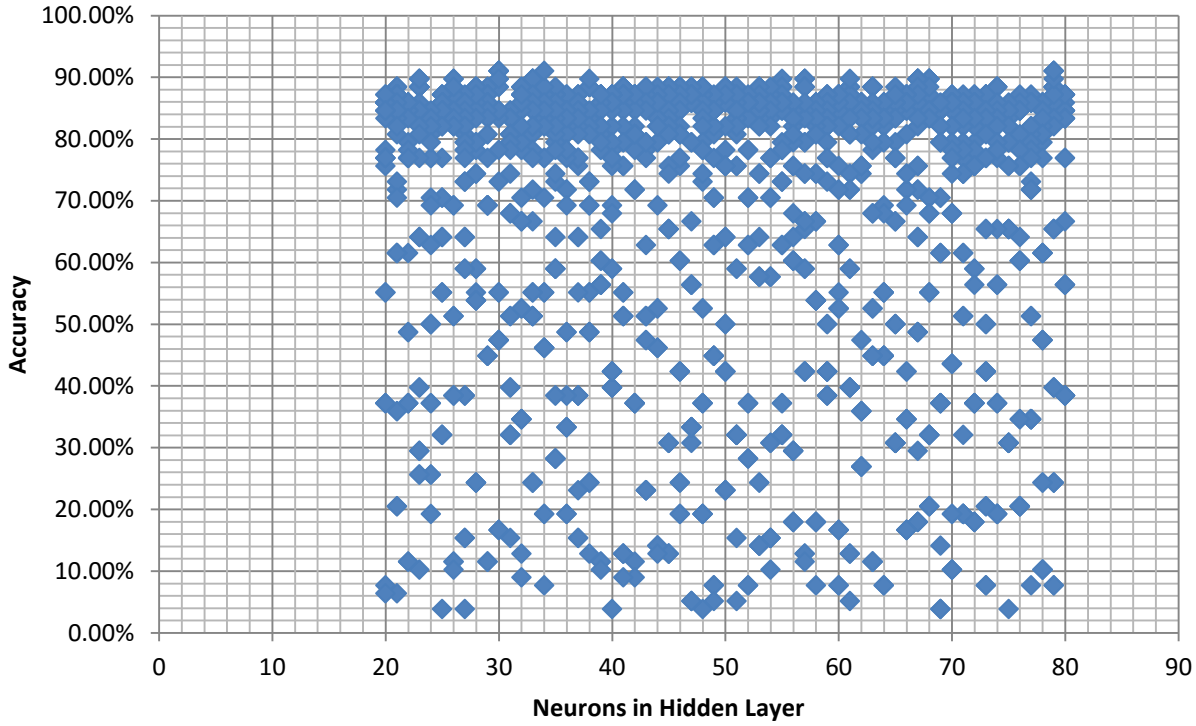
Results of the OCR System

The accuracy with which this system is able to perform OCR is also extremely good. As a perhaps rather rudimentary technique, it was expected that the accuracy of the system would be poor. The program outputted 1562 different neural network configurations, and astonishingly some of them performed extremely well, correctly identifying over 90% of the characters in the unseen sample set! These following networks produced the best results:

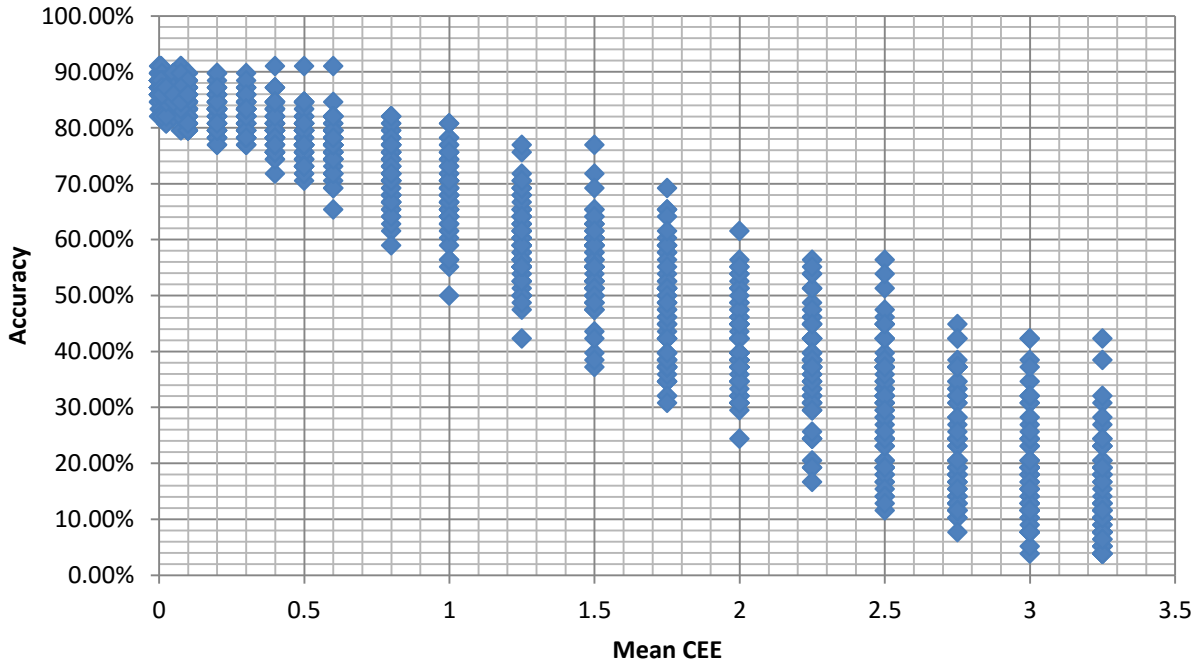
Hidden Layer Size	Mean CEE When Training Stopped	Accuracy
30	0.001	91.03%
30	0.0025	91.03%
30	0.005	91.03%
30	0.0075	91.03%
30	0.075	91.03%
34	0.075	91.03%
79	0.4	91.03%
79	0.5	91.03%
79	0.6	91.03%

It is interesting to note that the all but one of the best performances was by neural networks with a hidden layer of size either 30 or 79. Since there is much randomness involved in the process thanks to the random initial weights, it is expected that some neural networks produce extremely good results,

not as a reflection of some feature of their structure but as a result of, by chance, starting off with the random weights which so happen to produce optimal networks. This appears to be the case here: running the program 5 times shows that, on different occasions, different networks produce the best results. This graphs illustrates this point clearer:

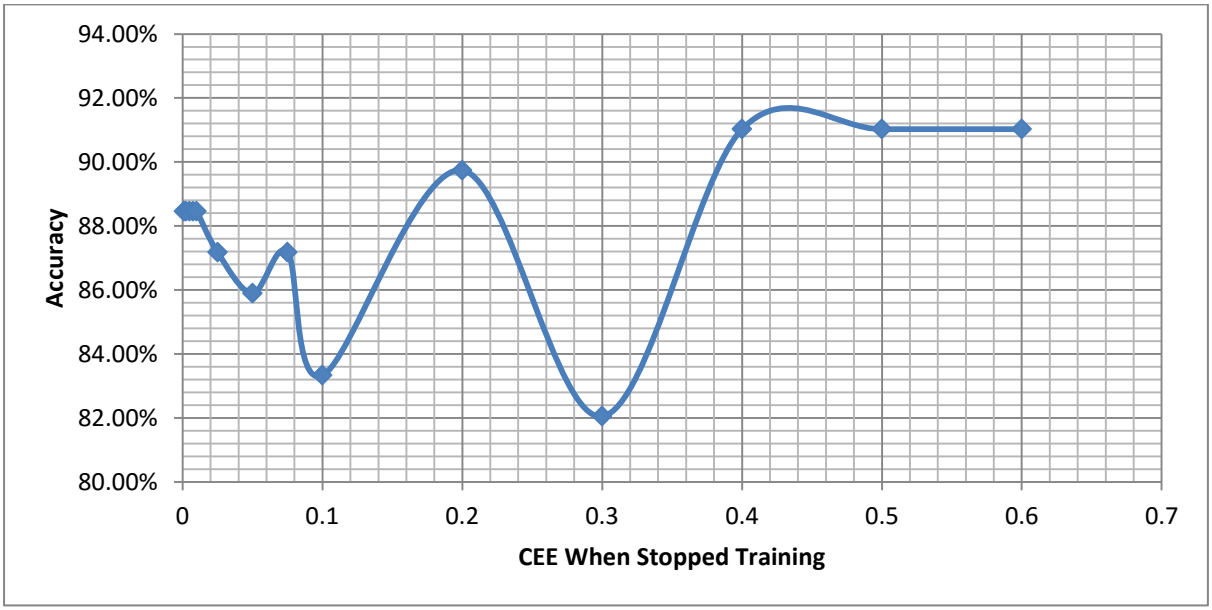


There is, however, as expected, a close link between how early the training is stopped and how accurate the network is at identifying unseen samples correctly:



Although this is perhaps one of the weaknesses of neural networks – that they are unpredictable and inherently are affected by randomness – the fact that they can be set up and trained quickly certainly does reduce the effects of this weakness, certainly for smaller networks like the one just used.

The results above also do illustrate how stopping early can sometimes be helpful, though it is hard to know when. With the network of size 79, the network was accurate to over 90% when stopped at CEE=0.6,0.5,0.4. But as CEE decreased, the accuracy did not necessarily improve:



This is an example of where generalisation is lost as training increases. However, a larger issue is that, again, it is completely impossible to predict when to stop training, other than to do it at intervals and test each one (as I have done above).

ANNs for OCR: Conclusions

Certainly for OCR, ANNs seem to be more than adequate. If such rudimentary handwriting recognition systems are able to achieve accuracies of over 90%, then more sophisticated networks paired with more sophisticated feature-extraction techniques on the images of letters should be able to perform very well, and it is therefore clear why they have been used in applications such as automatic address reading by postal services, or signature forging detection software. Of course, the networks produced are only able to recognise individual capital letters. Although extending it to read lowercase letters, numerals and punctuation should not be any issue at all, extending it to read joined-up handwriting would very extremely difficult. It would, of course, be far more useful for it be able to read pages of words – this, again, is in theory not too hard provided it is done in non-joined up handwriting or printed letters. The image of the page would first be processed to remove noise, convert to grayscale, and change the colour range to produce pure whites and blacks (which are easier for computers to read). Graph-theory based algorithms are able to detect where lines of words exist, which could be split up, and again these could be split up into individual words and letters. In short, this system could be extended to read pages of information.

Other considerations include that currently, the network is trained only on my handwriting. Of course, provided I can get hold of samples of other people's handwriting, this can easily be changed to

cover everyone's handwriting. For now, I have been able to produce a smartphone app which, after you draw a letter, can tell you what it is likely to be: I have compiled the code for the ANNs into a JAR library, imported this into an Android application project and, by using the XML format described above, I am able to reconstruct the ANN with 30 hidden neurons and a mean CEE of 0.001 produced by the above experiment. By modifying the app which originally produced samples to train with to instead produce the samples and feed them through the network rather than saving them, the application is able to guess which letter I have written.

Conclusion

Overall, ANNs appear to be excellent ways for machines to learn, and be able to perform pattern matching. They are relatively easy to set up, understand, and are very flexible. Their performance in recognising images of handwritten capital letters of the alphabet was certainly impressive, recognising over 90% of an unseen sample set correctly, and the speed with which they learn to do so is perfectly reasonable. Other types of ANNs such as Hopfield's nets and Convoluted Networks, which are not covered here, are able to do even more sophisticated learning, recalling and pattern matching. Whilst ANNs are unpredictable, due to the speed of modern computers it is possible to get an optimal result through repeated trial and error extremely quickly. That said, it was very fortunate that none of the examples above contained local minima (or if they did, I did not encounter them), which may otherwise be extremely difficult to overcome if there are many of them.

The ANNs which I have written have worked sufficiently well, although they may occasionally be slow, largely due to the overhead of the JVM. Of course, my code could certainly be improved for performance, and the actual architecture of the code, as described above, most likely does not help. Several micro-optimisations are certainly possible such as removing method calls, though at the cost of code readability and maintainability. Nonetheless, all ANNs trained fast enough even on a mid-range laptop – in the case of the OCR network, unexpectedly so – and as a result, I am overall happy with their performance. Similarly, the design of the code has been extremely flexible due to its modularised nature, which more than makes up for any performance hit. However, if I were to write this again, I would attempt to make the code multithreaded for increased performance, and it would certainly be better to write it using a natively-compiled language such as C or C++.

Sources

Boers E.J.W., Sprinkhuizen-Kuyper Ida G., "The Error Surface of the simplest XOR Network has no local Minima", Leiden University, Department of Computer Science, 1994

Boers E.J.W., Sprinkhuizen-Kuyper Ida G., "The error surface of the 2-2-1 XOR network: The finite stationary points", Neural Networks Volume 11, Issue 4, June 1998, Pages 683–690

Bullinaria, John A., Lecture Notes from "Neural Computation", Lectures 1-7, Birmingham University, Department of Computer Science, 2013

Heaton, Jeff, "Introduction to Neural Networks with Java"

Ni, Dong Xiao, "Application of Neural Networks to Character Recognition", Seidenberg School of CSIS, Pace University, White Plains, NY, 2007

Sangalli, Arturo, "The Importance of Being Fuzzy and Other Insights from the Border Between Math and Computers", 1998, Princeton University Press, New Jersey

Shiffmann, Daniel, "The Nature of Code"

Winston, Patrick H, Recording of "Artificial Intelligence, Lecture 12 – Learning: Neural Nets, Back Propagation, 6.034," Lecture by Department of Computer Science, MIT, Autumn 2010

"Derivation of Backpropagation Algorithm", Lecture Notes from CS81 — Adaptive Robotics, Swathmore University, Spring 2010

JOONE Open Source Neural Network Engine for Java

Encog Open Source Neural Network Engine for Java

Neuroph Open Source Neural Network Engine for Java