Introduction to Neural Networks and the Impact of Hyperparameters

Andrzej Socha

December 2023

Contents

Chapter 1

Introduction

Artificial neural networks are an ubiquitous part of the modern world. They have applications in every discipline, ranging from business[5], medicine[4] and information technology[6]. Part of why they are so common is their accessibility - everyone with a digital device can potentially utilise them and the learning curve is not too steep and almost entirely bypassed by some of the popular packages that implement them. They are the natural option for classifying and evaluating data in the age when data is available at arm's length in abundant quantities. Nevertheless, in spite of their popularity, not much concrete theory exists for predicting their output - they are often treated as a black box and relying on intuition for parameter tweaking is a common occurrence, even for large models. This is a huge part of their appeal - anyone who has understanding of the basics can use them to classify and make predictions based on the data of users, patients, weather, sales, etc. without having to become an expert in a particular field. It is also their shortcoming, as diagnosing any issues during training can be extremely difficult and blindly shooting for a good result is in no way a reliable strategy. In this paper we will attempt to shed some light on the properties of those models. In sections 2 and 3 we will explain the structure and mechanism of neural networks, highlighting the key user-set parameters that influence the performance of such models. In chapter 4 we will then train networks on toy datasets and expose any patterns that emerge, attempting to mathematically explain them and to make predictions that generalise well to larger and more complicated networks and datasets. Chapter 5 and the appendix contain information on error testing and the source code respectively.

Chapter 2

Structure of a Neural Network

2.1 General Structure

The concept of an artificial intelligence is as old as computing itself and the first steps towards the nebulous goal of creating it usually lead t o construction of algorithms that simulate the biological processes of the brain. In his 1950 paper [2], Alan Turing put forward an idea that a mind could be entirely mechanical in nature, with the consciousness and intelligence arising due to the sheer amount of interactions between the simple structures in such a model. Though far from reaching the goal of AGI (artificial general intelligence), NNs (neural networks), whose structure matches that description, are as ubiquitous as they are common in the digital world and they can be used for almost any task provided there exists a high quality dataset to train them on. In general, NNs are deterministic models that follow a linear structure. The input data, encoded in a vector of features, is sequentially transformed by the model, and the output can be leveraged to make predictions about the data. The exact way of interpreting the output depends on the specific network and the way it was trained as it is easy to modify the network for a specific use case.

While NNs consist of a large amount of simple neurons, it is easiest to conceptualise them by grouping the neurons into layers and treating the network as a series of interactions between subsequent layers. Fig. 1 depicts an example network.

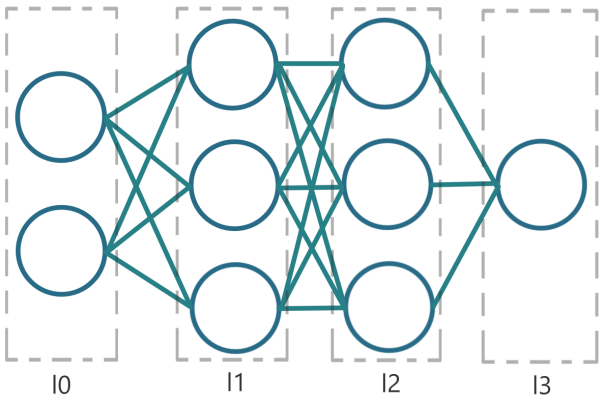


Figure 2.1: Diagram of a simple neural network. The circles represent neurons and the lines between them show what other neurons they are connected to. The dashed-line boxes segment the network into layers labeled $l_0, \cdots, l_3$.

The flow of data through such a net is left to right, where l_0 is the input layer that takes the value of whatever input vector we wish to make predictions for, and the last layer, l_3, is the output of the network. We can see that the neurons in each of the layers only connect to the neurons in neighbouring layers and so we can evaluate each layer sequentially to get the result.

2.2 Interaction Formula

In order to pass data through the network, we need to know the patterns of interactions between the neurons and between the layers. In reality, neurons interact with each other by sending and receiving electric impulses and there exist systems of equations that model that behaviour, but solving them numerically is slow, not proven to be any more accurate than simpler methods and vastly less efficient in practice. the formula that such networks use is

$$\hat{n} = \sigma\Big[\sum_i (w_i * n_i) + b\Big], \tag{2.1}$$

where $\hat{n}$ is the output, n_i are the input neurons connected to $\hat{n}$, σ is the activation function, w_i are the weights of the connections between the neurons and b is the bias. Since the input is determined by a particular data vector, it is not inherent to the network and so the output given some input depends only on the weights and biases of the network as well as the activation function used. Every connection between the neurons has a weight associated with it and every neuron (besides the input layer) has a bias. This simple model drastically cuts down the computational complexity while retaining the aspects of real neural interactions that are threshold for activation and loss of impulse due to connection strength. Thinking of the network in terms of layers instead, we can write the equation in matrix-vector form

$$\mathbf{x}^{(l+1)} = \sigma\big[W^{(l)}\mathbf{x}^{(l)} + b^{(l)}\big], \tag{2.2}$$

where $\mathbf{x}^{(l)}$ is the vector representing the values of neurons in layer l, $W^{(l)}$ is a matrix representing the weights of connections between layers l and $l+1$, and $b^{(l)}$ is the vector of biases associated with layer l.

2.3 Activation Functions

Activation functions are a crucial part of the model, allowing the network to be more than a simple linear regression model. To see why they are so important we can consider the network in the case that $\sigma(x) = x$. Then the output of the network is a linear function of the previous layer, which itself is a linear function of the layer before. Composition of linear functions results in another linear function and so the entire network becomes equivalent to one iteration of (2.2) with different weights and biases - exactly a linear regression model. Including a nonlinear activation function between each of the layers solves that issue with the trade-off of making analytical optimisation of the network much more complicated - a trade off we can accept since, as we will see in the later sections, we will use numerical optimisation instead. There are many commonly used activation functions, here we will list the most common ones and provide some descriptions for their use cases.

Tanh
Hyperbolic tangent $\sigma(x) = \tanh x$. The domain of this function is the entire real line while the range is the interval (-1, 1). This means that the the values of the neuron activations in each layer are bounded which prevents us from encountering the exploding gradient problem during optimisation, where the activations diverge causing the training to fail. Since the expected value of tanh for a random input is 0, the mean of the activations in a layer will be close to 0 for all layers. Such centering can make the training process more stable. The downside, and the reason it's usage is decreasing, is that it's gradient becomes incredibly small even for small input magnitude, which often leads to the vanishing gradient problem, where the gradients, which we will later see are necessary for optimisation, get so small that the training stops almost entirely.

Sigmoid
The logistic function $\sigma(x) = 1/(1+e^{-x})$. Similar to tanh, the function maps the real numbers to a set (0, 1). It comes with the same ups and downs as tanh, with the added benefit of turning the output into a probability when used on the final layer. It can also worsen the vaninshing gradient problem since it's gradient is even smaller than that of tanh.

ReLU
Rectified linear unit $\sigma(x) = \max(x, 0)$. The most popular choice for neural networks due to it's computational simplicity as well as effectiveness. It allows us to discount the non-active neurons since they do not contribute to the result. The linear part of the function is non-saturating, meaning that so long as the training is stable it will converge much faster than the saturating alternatives. It's shortcoming is that in some cases neurons will be permanently inactive, leading to redundancy, which can be particularly impactfull on small networks where dead neurons are a significant percentage of the total. In general the positives far outweigh the negatives and ReLU is considered the go-to activation function for most networks.

Chapter 3

Fitting the Model to Data

3.1 Cost Function

Fitting the model to the data means that we assign labels to the data points and want the network to return values close to the labels for the data points. In order to do that we need a way of measuring how wrong the model is when making a prediction. The cost function (also known as a loss function) is a real-valued function that takes in the output of the network x and the target labels for a data point y and returns a scalar that quantifies the total error. There are many cost functions that are used in different scenarios, here we list the most common ones.

Mean Square Error (least squares, L2 loss)
$\text{MSE}(x) = \frac{1}{2\dim(x)} \sum_i (x_i - y_i)^2$. Commonly used in analytical models for convex optimisation due to it having only one, global minimum, it is still common in machine learning where that's not a guarantee. Can be used both for regression and classification purposes, although it is mainly used for regression since there are better alternatives for classification. It is not robust against outliers due to the square.

Mean Absolute Error
$\text{MAE}(x) = \frac{1}{\dim(x)} \sum_i (|x_i - y_i|)$. Used for regression, has been shown to achieve lower cost tan MSE in certain situations[3]. It's more robust against outliers than MSE.

Cross-Entropy (log loss)
$\text{CE}(x) = -\sum_i \left[y_i * \ln x_i\right]$. The most common cost function used for classification, this function measures the difference between two probability distributions - in this case the model and the underlying distribution we are approximating. It demands that the output values of the network are probabilities, so we need to have a transformation function that converts the output into values that sum up to 1. In the case of 2-class classification this function is commonly sigmoid (binary cross entropy), while for multiclass models it is usually softmax (categorical cross entropy).

3.2 Optimisation

Now that we have a cost function to quantify the error, we have to solve an optimisation problem - minimise the total cost of the dataset with respect to the model parameters (weights and biases). In order to do that we use gradient descent, an iterative algorithm that reduces the cost by perturbing the parameters in the direction of steepest descent.

$$p_{n+1} = p_n - \alpha * \bigtriangledown_p C(p), \tag{3.1}$$

where p_n are the parameters at step n, $C(p)$ is the cost function and α is a hyperparameter (parameter set by hand instead of being inferred from data) called learning rate, which is positively correlated to the rate of convergence but also the error made at every step. With the learning rate too small the training will take too long while if it's too large the parameters will diverge. While this algorithm will work for small data sets, having to calculate the cost over all data points at every iteration makes it infeasible for datasets with more than a few thousand examples. In general, NNs use stochastic gradient descent (SGD), which instead of using the entire dataset, uses only a randomly chosen entry or a batch of entries. While that makes every step less accurate and can make some steps increase the cost instead of decreasing it, SGD allows us to take many more total steps which leads to faster convergence in most practical situations.

3.3 Backpropagation

Since the learning rate α is set by hand and the parameters are known, the only value we have yet to determine in (3.1) is the gradient of the cost function with respect to the parameters. Let $z^{(l)}$ denote the pre-activation inputs to the neurons in the layer l and $a^{(l)}$ denote the activated values of the neurons in that layer. We again use the sequential nature of the network and use the chain rule to evaluate the gradients in subsequent layers. We define $\delta^{(l)} = \frac{\partial C}{\partial z^{(l)}}$ as the error in layer l. Then the error for the last layer N is given by

$$\delta^{(N)} = \frac{\partial C}{\partial a^{(N)}} \frac{da^{(N)}}{dz^{(N)}} = (a^{(N)} - y) \circ \sigma'(z^{(N)}), \tag{3.2}$$

where $\circ$ is the component-wise multiplication. Knowing the last error we can find all the other errors and, subsequently, the gradient of the cost with respect to weights and biases as follows:

$$\delta^{(l)} = \sigma'(z^{(l)}) \circ (W^{(l+1)})^T \delta^{(l+1)}, \tag{3.3}$$

$$\frac{\partial C}{\partial b^{(l)}} = \delta^{(l)}, \tag{3.4}$$

$$\frac{\partial C}{\partial W_{ij}^{(l)}} = \delta_i^{(l)} a_k^{(l-1)}. \tag{3.5}$$

This way we propagate the errors back through the layers and evaluate the gradients layer by layer.

3.4 Convergence

Training of a neural network consists of repeatedly applying the SGD algorithm with backpropagation until an appropriate level of performance is reached. In order to evaluate the performance, we need a way to quantify it. Fortunately, we can use the cost function here again and stop the training when the cost becomes smaller than some threshold. Another way of evaluating the performance is via the accuracy of the model. For small datasets this is not very informative as the accuracy often quickly reaches 100% after which we can not infer if there is progress being made just by looking at the accuracy, but it becomes helpful for larger datasets, for whose it is uncommon for a network to ever reach maximum accuracy. After enough iterations the cost will saturate, effectively meaning that no further progress can be made, since the perturbations introduced in gradient descent become too large relative to the changes necessary for improvement. We can assuage this by implementing a variable learning rate scheme that decreases the learning rate if training at it's current value has plateaued.

3.5 Data Split

Reaching the lowest possible cost for a given dataset is only a viable goal if we can prevent the model from overfitting - learning the noise of the dataset and losing the ability to generalise to data from outside of the trainng set. This is why in every machine learning project we split the available data into training and test sets (usually 4:1 split), so that we can train the network on the training set and then evaluate it's ability to generalise on the test set that it is unfamiliar with. In most cases we will see that the cost with respect to the test set initially decreases, until reaching a turning point and starting to increase. This is the moment when the model starts to overfit to the training data.

Chapter 4

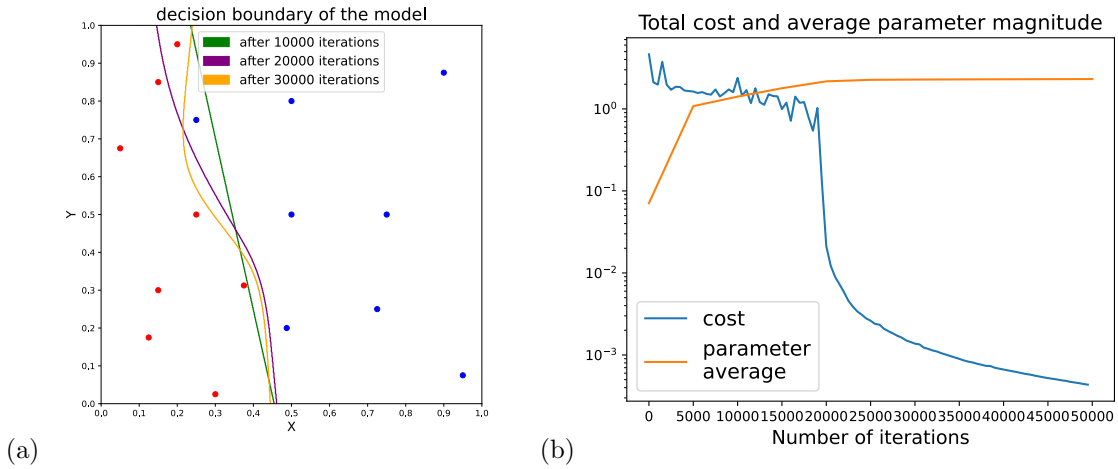
Examining the Convergence

In this chapter we will examine the behaviour of the network during training and compare rates of convergence for different hyperparameters. We focus on networks used for binary classification, although some of the results could be extended to regression and multiclass classification. In order to make the review reproducible, there are 3 functions that generate datasets for us to classify included in the code in the appendix: "GetTestData", "GetCheckerboardData" and "GetSpiralData". Due to the nonlinearity of the NNs as well as the large number of possible parameters, not all mechanisms of such models are well understood. Here we focus on examining trends in the numerical results and provide possible explanations for some of it's behaviour.

4.1 Decision Boundary

In the chapter 2 we have already discussed the importance of the activation functions - without them the model would be linear. In that case we would expect the input space to be divided into two half-spaces by the model, where all points on either side are grouped into one class. If we introduce one activation function, then we would expect it to transform the boundary into a different curve. If the activation function is continuous, then, since composition of continuous functions is also continuous, applying it to the linear boundary would also yield a continuous curve which means the network still divides the space into 2 halves. We could then expect the network to struggle when the dataset for the classification task involves multiple disjoint sets of one class surrounded by the other as there would be no good way of separating all of them with one continuous curve.

Let's consider a simple 2x3x3x1 network (where the numbers represent the number of neurons in a layer) with tanh as the activation function after every layer and MSE as the cost function. We initialise the parameters by drawing from a normal distribution with mean 0 and standard deviation 0.1, train it on the "GetTestData" dataset (including points in $(0,1)^2 \subset \mathbb{R}^2$) and examine the evolution of the decision boundary. We can predict the boundary to initially resemble a straight line inside of the domain of the data. That is because for tanh, the linear approximation around the origin is $\tanh(x) \approx x$ and since all our parameters are initially small in magnitude, we can expect expect it to approximate the real curve well. Fig.2 a) shows the results (only up to $3 * 10^4$ iterations because the visual change of the boundary afterwards becomes minimal). We can see that initially the boundary is an almost straight line (green curve) which then starts to distort to separate the data points. At that point maximum accuracy has been achieved and any further adjustments are minor and only reduce the cost. Fig. 4.1 (b) blue curve shows that there is an initial period when the cost remains at the initial order of magnitude, followed by a sharp decrease of the cost that then tapers off as the network parameters begin to converge. We can see that the sharp decrease starts at around 18000 iterations, which corresponds to the transition of the boundary away from an almost straight line, depicted in (a), that happens between 10000 and 20000 iterations. The orange curve from (b) shows that the initial period of relatively constant total cost coincides with the period during which the average parameter magnitude is changing from it's initial value towards the final value. As we discussed before, for small parameters the boundary will resemble a straight line, which explains why the boundary transitions into a curve at the end of that initial period. Overall, we can see that the network manages to properly separate the datasets.



(a) (b)

Figure 4.1: (a) Decision boundary of a neural network with respect to the number of training iterations. The red and blue dots represent the data points in both classes and the colored curves represent the boundary. (b) The evolution of total cost and the average magnitude of the parameters during training.

4.2 Impact of Learning Rate

Learning rate is one of the hyperparameters of the network and choosing it's value will have an impact on the rate of convergence. Setting a value that is to high can even cause the training to fail entirely. To understand why we look at Taylor expansion which is a basis of the gradient descent algorithm. Let f be a real-valued function of vector variables, $\mathbf{x}, \mathbf{p}$ be real-valued vectors, then

$$f(\mathbf{x}+\mathbf{p}) = f(\mathbf{x}) + < \mathbf{p}, \bigtriangledown f(\mathbf{x}) > + O(||\mathbf{p}||^2). \tag{4.1}$$

We can see that disregarding the higher order terms is only a possibility if the norm of $\mathbf{p}$ is small enough. Since the goal in gradient descent is to minimise the function, we are looking for $\mathbf{p}$ that causes the inner product term to be as negative as possible. Normalising the vectors gives

$$< \mathbf{p}, \bigtriangledown f(\mathbf{x}) >= ||\mathbf{p}|| \;\; || \bigtriangledown f(\mathbf{x})|| < \frac{\mathbf{p}}{||\mathbf{p}||}, \frac{\bigtriangledown f(\mathbf{x})}{|| \bigtriangledown f(\mathbf{x})||} > . \tag{4.2}$$

Since we are dealing with an euclidean vector space, we recall that

$$< \mathbf{a}, \mathbf{b} >= ||\mathbf{a}|| \; ||\mathbf{b}|| \cos(\theta), \tag{4.3}$$

where θ is the angle between the vectors $\mathbf{a}$ and $\mathbf{b}$. We can then see that the minimum value that an inner product of two unit vectors can take is -1, corresponding to the case when one vctor is a negative of the other, i.e. $\theta = \pi$. Then the lowest possible value that the inner product term in (4.1) can take corresponds to the case when $\mathbf{p}$ is perpendicular to the gradient of f, $\mathbf{p} = -\alpha \bigtriangledown f(\mathbf{x})$, where α is some scalar. This leads us to the iterative algorithm

$$\mathbf{x} \rightarrow \mathbf{x} + \mathbf{p} = \mathbf{x} - \alpha \bigtriangledown f(\mathbf{x}), \tag{4.4}$$

that decreases the objective function as much as possible at ever step. It then becomes apparent why a learning rate too large will cause the algorithm to diverge - the higher order terms become too large to disregard.

We now wish to know what happens for smaller learning rates. Under the assumption that the h.o.t.s evaluate to 0 and with a constant gradient, one step with a learning rate α is equivalent to taking n steps with learning rate α/n. Since in reality the contribution due to h.o.t.s will not be 0, every step of the algorithm will not be taken in the optimal direction and so we can expect no more than n steps to be necessary to match one step with the larger learning rate as more steps allow for more adjustments to the direction which means that no more than n times the

iterations will be necessary to reach the same total cost.

Going back to the example network we used in the previous section, the objective function $C(\mathbf{x})$ to minimise is the MSE loss. If we use the gradient descent on just the cost function the (4.1) becomes

$$C(\mathbf{x}_{n+1}) \approx C(\mathbf{x}_n) - \alpha < \bigtriangledown C(\mathbf{x}_n), \bigtriangledown C(\mathbf{x}_n) > . \tag{4.5}$$

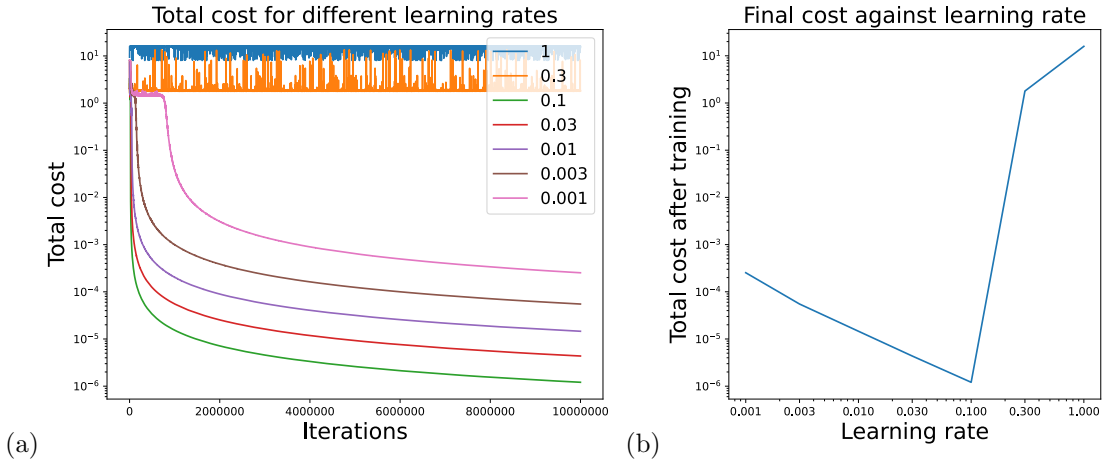
From the definition of MSE we have

$$\alpha \ < \bigtriangledown C(\mathbf{x}_n), \bigtriangledown C(\mathbf{x}_n) >= \alpha \sum_i (x_{n_i} - y_i)^2 = 2\alpha C(\mathbf{x}_n). \tag{4.6}$$

Substituting that back into (4.5) gives

$$C(\mathbf{x}_{n+1}) \approx (1 - 2\alpha) C(\mathbf{x}_n). \tag{4.7}$$

This would imply exponential decay and the optimal choice of learning rate being $\alpha \approx 0.5$ with extremely fast convergence. Since the network is more complicated than just the cost function and we are using SGD instead of gradient descent the convergence will be slower that that. Fig. 4.2 (a) shows how the learning rate impacts the total cost during training for the same network as the one we used in the section above.



(a) (b)

Figure 4.2: (a) Every curve shows the total cost during training for the learning rate stated in the legend. (b) shows the total cost after training based on the learning rate.

We can see that when the learning rate is too large, the total cost oscillates around the initial value and never decreases further. The learning rate for just the cost function of $\alpha = 0.5$, is too large for SGD on the entire network. For the smaller learning rates, the lower the rate the slower the convergence and every other curve is separated by just less than 1 order of magnitude. This is consistent with the prediction that a learning rate n times lower will cause the algorithm to converge no more than n times slower. Graph (b) also reflects this, by showing that larger learning rates lead to faster convergence, up to a point at which they are too large and cause learning to fail. Furthermore, for the learning rates that lead to convergence the log-log plot is an almost straight line which suggests a power relation between the learning rate and total cost. We can calculate the power by finding the slope between two points.

$$y = x^m \implies \ln y = m \ln x. \tag{4.8}$$

Then for two points $(x_0, y_0), (x_1, y_1)$ we have

$$m = \frac{\ln (x_1/x_0)}{\ln (y_1/y_0)}. \tag{4.9}$$

Calculating the slope between each of the successively smaller learning rates gives the 4 resulting powers: $-1.27, -1.21, -1.00, -1.17$. While they are not all the same, the small differences between them seem to suggest that the relationship between cost and learning rate resembles a power function with the average power of -1.16 within the examined range.

4.3 Impact of Initial Parameter Distribution

In section 4.1 we have seen that a significant progress in reducing the total cost only seems to be made after the average parameter magnitude has settled close to it's limiting value. We can then expect that distributions leading to large differences between initial and final parameter magnitudes would cause the convergence to be slower. While we can not know the final parameters beforehand, we can attempt to draw conclusions from experimental data. We once again use the same network as before with the training data from the "GetTestData" function. Fig. 4.3 shows the evolution of total cost during training for different values of mean and standard deviation of the initial parameter distribution.

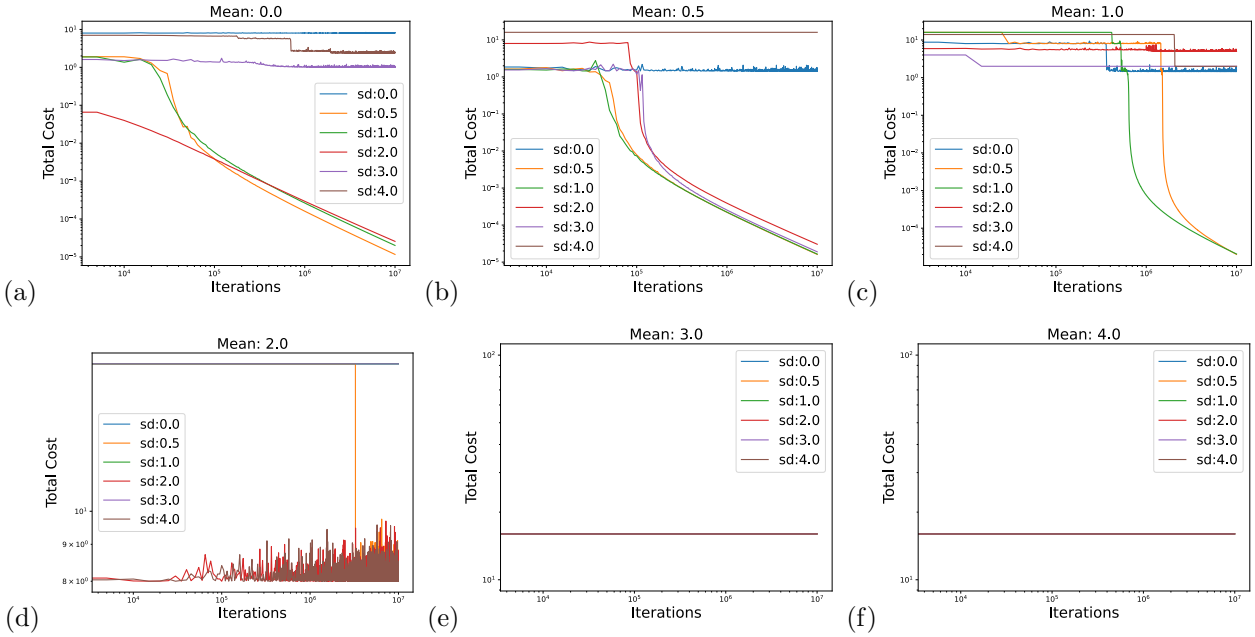


Figure 4.3: Log-log plots of total cost against number of iterations during training. Each of the plots corresponds to different mean of the distribution and each of the curves is annotated with it's respective standard deviation. Each of the results is an average over multiple training processes. Training failed for layer counts larger than 6.

Throughout the plots we can see that the cost corresponding to standard deviations, s, of 0, 3 and 4 never converges. There are two mechanisms responsible for that. In the case of $s = 0$, every parameter will be initialised to the value of the mean, which means that all the parameters are equal. Since our network has one output neuron, the error associated with the final layer, $\delta^{(N)}$ given by (3.2), will be a scalar. We can see from the equations (3.3)-(3.5) that every other error depends on the parameters of the network and the activation values, but since all the parameters are equal, the activation values will be the same within each layer, which means that the gradients with respect to the weights and biases will be the same within each of the layers. This means that the parameters within each layer will always be equal and the model is effectively constrained to be equivalent to a network with one neuron in each of the layers, which is not capable of representing the dataset. In the case of $s >= 3$, the issue is not that the cost will never converge, but rather that the rate of convergence is extremely slow. We can see in (a) that the cost associated with those values does decrease, although much slower than for the rest. The initial period that we have seen in section 4.1, during which the cost remains mostly unchanged and the parameters approach their limiting values, seems to last longer for the larger standard deviations, most likely because the average distance between the initial and final parameters is larger. In the cases that the cost does begin to steadily decrease, after the initial large adjustment, all the curves seem to tend to straight lines with the same slopes, which on a log-log plot suggests a power relation. This is consistent with our observation from the section above and the average of the power from the plots gives $m \approx -1.16$ which is the same estimate as in the previous section. We can see that for means larger or equal to 2 none of the costs begin to converge. This can by explained by the vanishing gradient problem which we have mentioned in section 2. Since the derivative of our activation function, tanh, is rapidly approaching zero for any values with magnitude larger than 0, we can expect the gradients for large weights and biases to be almost 0 which means that the iterations necessary for gradient descent to converge grow unboundedly, and we quickly reach machine precision limit after which

all the gradients get rounded to 0 and any training becomes impossible. The reason why we only consider positive means is that the derivative of tanh is symmetric around origin and so the gradients would have the same magnitude even if the signs of parameters were flipped, which means that gradient descent would converge at the same rate.

4.4 Larger Datasets and Larger Networks

For a single dataset we could expect a network with more total parameters to outperform a smaller network, but there are a few important things to consider. The way parameters are distributed can impact the performance - less layers with more neurons could behave differently to more layers with less neurons. We have brought up the vanishing gradient problem before with respect to the magnitude of the parameters, but large amount of layers can lead to a similar situation, where each successive vector of errors gets smaller and asmaller in magnitude, since they are multiplicatively related to the previous ones, leading to smaller and smaller gradients. Even though we are not considering the test-train split in this report, it is important to note that a network which is too large for a dataset will have the capacity to "remember" the data instead of reflecting it's trends, leading to overfitting and could result in lower performance on the test dataset as compared to a smaller network. Here we will examine the performance of networks on different datasets based on the number of layers and the amount of neurons in each layer. First we look at the performance on the "GetTestData" dataset. Fig 4.4 shows the experimentallly gathered data and Fig. 4.5 shows the decision boundaries of the different networks.

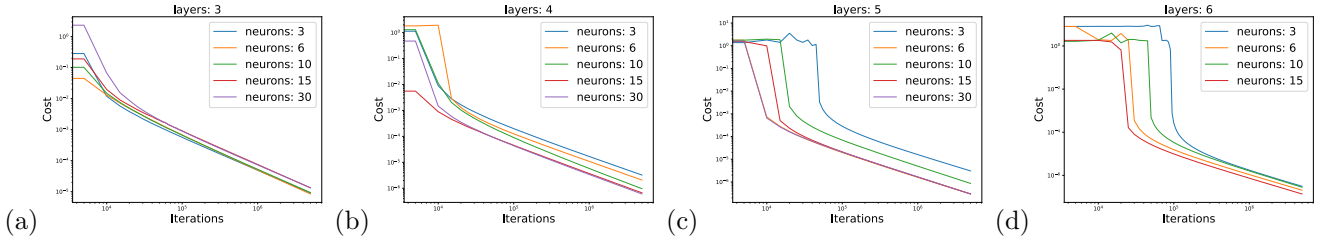


Figure 4.4: Log-log plots of total cost against number of iterations during training. Each of the plots corresponds to different number of layers and each of the curves is annotated with it's respective neuron per hidden layer count. Each of the results is an average over multiple training processes.

For 3 layers, every neuron count seems to perform at a very similar level, but for networks with more layers it appears that higher neuron counts lead to the initial parameter adjustment phase being shorter, which means the network converges faster. The power relation that we have observed before can be seen.

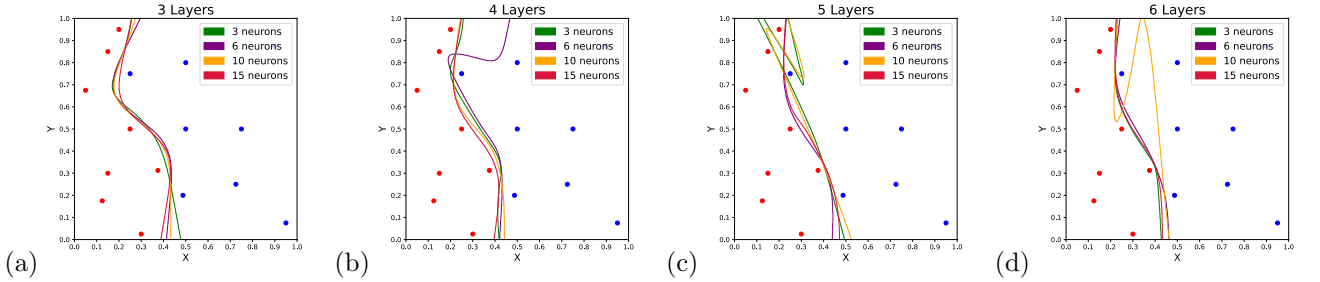


Figure 4.5: Decision boundaries for the different models. Each plot corresponds to a network with different number of layers and each curve within a plot corresponds to the boundary for different neuron count.

We can see that all the models manage to achieve 100% accuracy on the training data. For some neuron counts the boundaries do not follow the trends of the data and look like they would not generalise well. This highlights the importance of splitting the training data and cross-validating the results. For the highest neuron counts, the boundaries seem to converge to the desired shape, which suggests that networks with higher amount of neurons could

generalise better than smaller ones.

We now move on to classify the data provided by "GetCheckerboardData" function. This dataset contains 1000 entries and the pattern is much more challenging to reproduce, so we can expect the performance of the network to be much worse than in the simpler case given the same network structure. We have discussed before that a learning rate too large would cause the training to fail. In this case our previous learning rate of 0.1 was too large and the training failed. This is understandable since we are using SGD with one data point at a time and so with a large learning rate the network will adjust too much to fit that one data point at the cost of failing to fit to the other points. With 1000 points and higer overall difficulty this is much more pronounced than in the previous case. Fig 4.6 shows the results of the training.

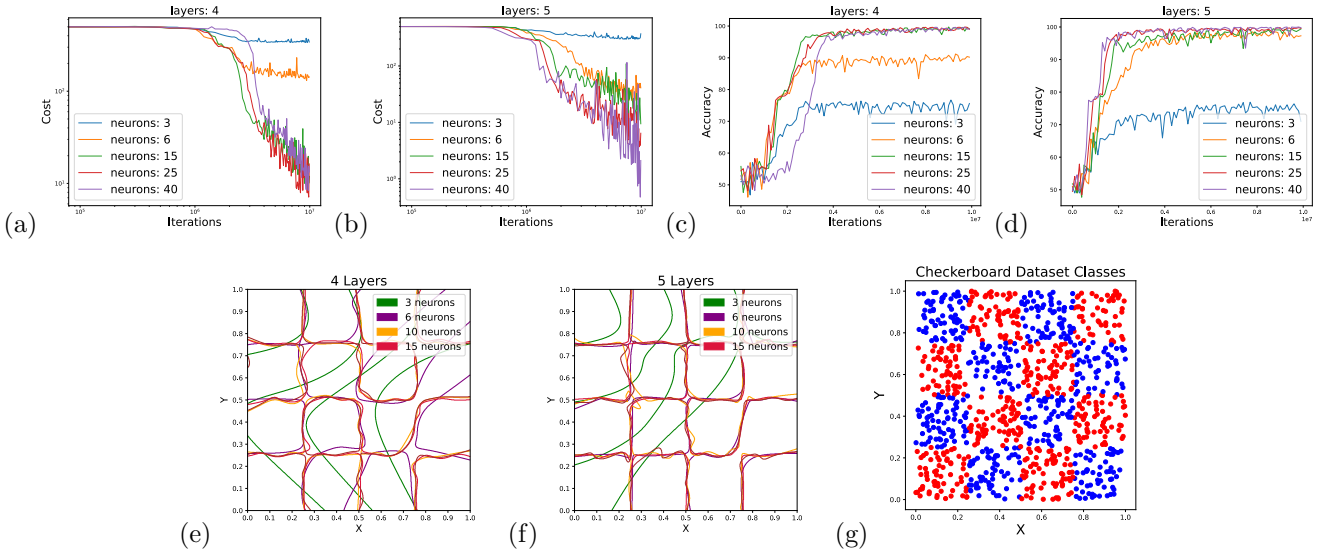


Figure 4.6: (a) & (b) Total cost during training. (c) & (d) Accuracy of the model during training. (e) & (f) Decision boundary after training. (g) Class distribution of the dataset. Each plot corresponds to different layer count and each curve corresponds to different neuron count within the layers.

We can see that the 3-neuron layers version of the network is inadequate for the task and fails to get above 75% accuracy. It is also clear that the larger the neuron count per layer the better the network does at classifying the test data. The graphs only show results for 4 and 5 layers since the network failed to converge for 6 and more layers within 1e+7 iterations. The total cost curves are noisy, (which is only clearly visible for lower values, but that's due to the nature of log-log plots), but they still retain the generally linearly decreasing trend after the initial period of parameter changes.

Finally, we attempt to classify the data from "GetSpiralData" Dataset. This set also contains 1000 entries that are not linearly separable. We continue to use the same network structure as above with learning rate of 0.03. Fig. 4.7 shows the results of the training.

We can observe a general trend that higher neuron counts lead to lower cost and higher accuracy, with the same being true for number of layers. It is notable that this relationship seems to be reversed for networks with only 3 layers, but their results are much worse compared to larger models and so they would not be useful in practice. The accuracy plots show that the results for 25 and 40 neurons are very similar, although the cost is not, suggesting that the networks might be overfitting and that there might be differences in performance on separate test datasets. Although after the accuracy saturates near the maximum it begins to fluctuate, only the 5-layer-25-neuron and 6-layer-40-neuron networks have been able to achieve 100% accuracy on the dataset, supporting the assumption that larger networks perform better. The linear aspect of the linear decrease trend we have identified in earlier experiments is not clearly visible here and would require averaging over many more training sessions to verify, which was not done here due to time constraints.

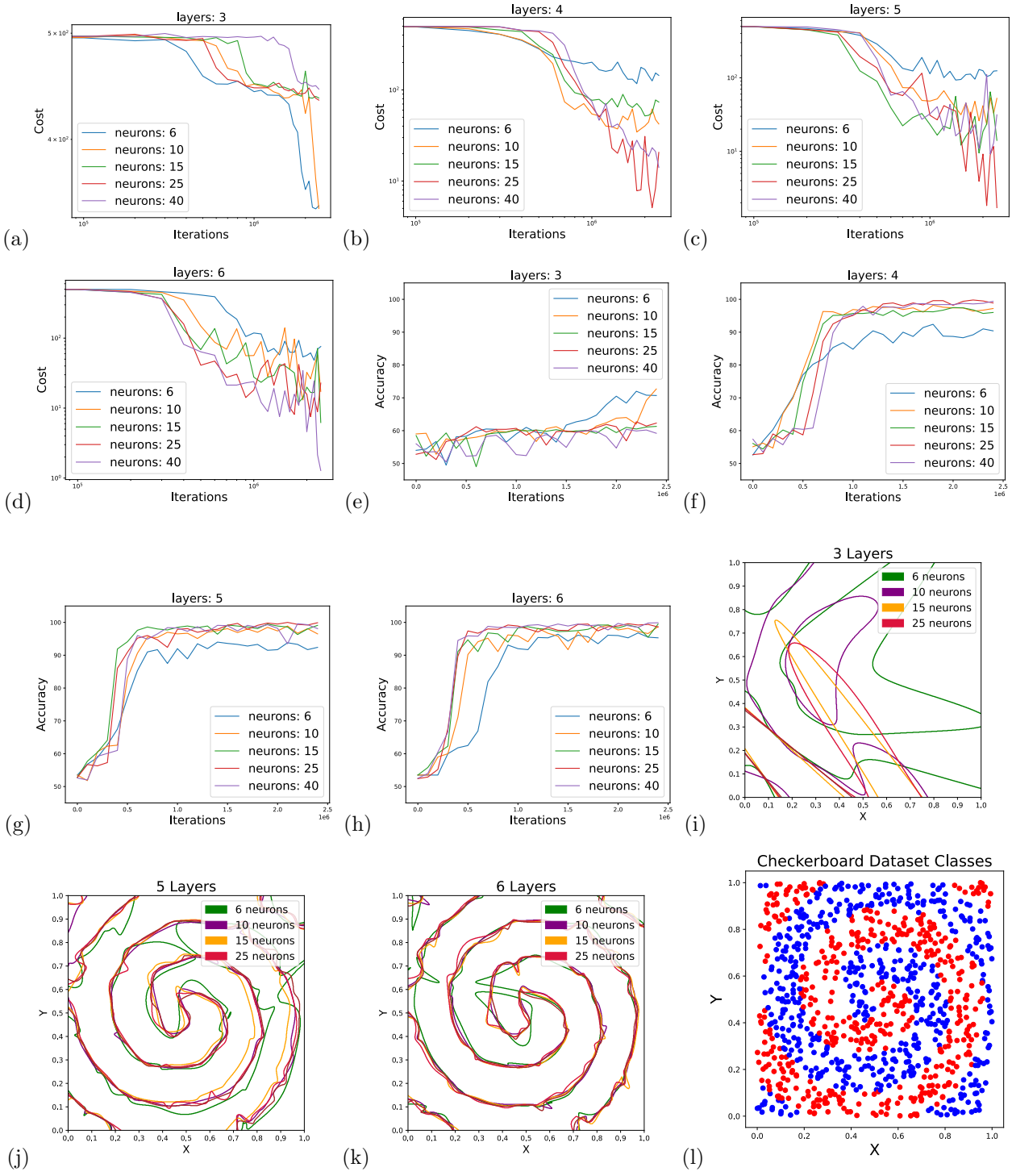


Figure 4.7: (a) - (d) Total cost during training, displayed on log-log plots. (e) - (h) Accuracy of the model during training, displayed on linear-linear plots. (i) - (k) Decision boundary after training. (l) Class distribution of the dataset. Each plot corresponds to different layer count and each curve corresponds to different neuron count within the layers.

Chapter 5

Code Validation

In this chapter we show the methods and test cases used to validate the code. The entire code for this report is included in the appendix, here we will only highlight the testing process.

5.1 Network Structure

In this section we examine the tests made on all the function members of the Network class. All the code is written in the Network::Test() function and different sections are written in separate scopes. Every test here is based on checking whether the code outputs the pre-calculated values for a given test case. The desired values were calculated by hand and using Wolfram Alpha. Every test passes quietly and in the case of failure prints an error message explaining which function failed. The first test checks the FeedForward, BackPropagateError and UpdateWeightsAndBiases functions. The second test checks whether the parameters have been initialised properly by comparing the parameter vectors and matrices to ones created by the basic constructor, effectively checking if they have been altered after being created. This can produce a false positive if all the values drawn from the distribution happen to be 0, but this is extremely unlikely. The third test checks whether the activation function, it's derivative and their vector generalisations are working properly. Fourth test checks the FeedForward function again on a different test case. The fifth test checks the cost functions. The last test runs the FeedForward, BackPropagateError and UpdateWeightsAndBiases functions again on a different test case, but outputs values to be examined instead of automatically determining a failure.

5.2 Additional Tests

Network::Test2() is a function can be useful in case the training process stagnates. It attempts to train the model on a dataset and writes the average parameter values to a file after every training iteration. This can be later examined to see if any progress is being made even if the total cost remains on the same level, which may suggest that the issue is too small number of training iterations.
Network::Test3() was used to verify that the behaviour of the network when the initial parameters are drawn from an uniform distribution was correct after that case has shown to lead to low performance. This has now been explained in section 4.3.

After the initial tests of all the member functions have been passed, all that remained was to test the network's capability yo learn. This has been shown throughout the report, with the figures being generated from the data output by the network. The fact that 100% accuracy on difficult datasets has been achieved suggests that the code is working properly, as any bugs in the implementation of the relatively straightforward member functions would likely lead to drastic losses in performance. Additionally, in order to guarantee that the data displayed throughout this report is representative of the true performance of such networks, every test has been run multiple times and the results displayed are averaged over all the runs.

Chapter 6

Conclusion

In this report we have discussed the general structure of neural networks, explaining every mechanism that is important for their functionality. We have described the algorithm responsible for the learning and mentioned the commonly varied parts of the model and the cases in which they are useful. In particular we have discussed the importance of choosing the right activation function and demonstrated that tanh leads to extremely slow progress for increasing layer counts. This observation is backed up by the fact that saturating activation functions have been being phased out in favor of ReLU and it's derivatives, which are now the most commonly used ones. Additionally, we examined the performance of the neural networks and how hyperparameters like learning rate, number of hidden layers and number of neurons per layer affect the rate and degree of convergence. It has been shown that simple networks trained on simple datasets have clear inverse power relation between total cost and number of iterations and that this trend seems to continue for more complex problems, even though it is significantly more noisy. This is consistent with the known fact that convex optimisation problems are guaranteed to converge to 0 cost, since the simple dataset used was almost linearly separable, and suggests that more complex problems still retain some degree of convexity. Finally, we have shown that larger structures lead to better results, but we also explained the importance of cross validation, highlighted by some of the decision boundaries overfitting to the training data.

References

[1]Catherine F. Higham and Desmond J. Higham. Deep learning: An introduction for applied mathematicians. 2018. URL https://arxiv.org/abs/1801.05894.

[2]Alan Turing. Computing Machinary and Intelligence. Mind, 59(236), 1950.
URL https://www. jstor.org/stable/2251299.

[3]J. Qi, J. Du, S. M. Siniscalchi, X. Ma and C. -H. Lee, "On Mean Absolute Error for Deep Neural Network Based Vector-to-Vector Regression," in IEEE Signal Processing Letters, vol. 27, pp. 1485-1489, 2020,
URL https://doi: 10.1109/LSP.2020.3016837.

[4] M. Sordo, "Introduction to neural networks in healthcare." Open clinical: Knowledge management for medical care (2002).

[5]A. Vellido, P.J.G. Lisboa, J. Vaughan, Neural networks in business: a survey of applications (1992–1998), Expert Systems with Applications, Volume 17, Issue 1, July 1999, Pages 51-70

[6]W. Serrano, Neural Networks in Big Data and Web Search. Data. 2019; 4(1):7. URL https://doi.org/10.3390/data4010007

Appendix - Code

Code for the neural network:

```
#include "mvector.h" //header file describing a vector class
#include "mmatrix.h" //header file describing a matrix class

#include <cmath> //for access ot basic mathematical functions
#include <random> //for access to random numbers
#include <iostream> //for ability to write to terminal
#include <fstream> //for ability to write to files
#include <cassert> //for acccess to assert functionality
#include <iomanip> //for ability to control precision of the output
#include <string> //for access to string variables


////////////////////////////////////////////////////////////////////////////
// Set up random number generation

// Set up a "random device" that generates a new random number each time the program is run
std::random_device rand_dev;

// Set up a pseudo-random number generater "rnd", seeded with a random number
std::mt19937 rnd(rand_dev());

////////////////////////////////////////////////////////////////////////////
// Some operator overloads to allow arithmetic with MMatrix and MVector.

// MMatrix * MVector
MVector operator*(const MMatrix &m, const MVector &v)
{
   assert(m.Cols() == v.size());

   MVector r(m.Rows());

   for (int i=0; i<m.Rows(); i++)
   {
      for (int j=0; j<m.Cols(); j++)
      {
         r[i]+=m(i,j)*v[j];
      }
   }
   return r;
}

// transpose(MMatrix) * MVector
MVector TransposeTimes(const MMatrix &m, const MVector &v)
{
   assert(m.Rows() == v.size());

   MVector r(m.Cols());
```

```
   for (int i=0; i<m.Cols(); i++)
   {
      for (int j=0; j<m.Rows(); j++)
      {
         r[i]+=m(j,i)*v[j];
      }
   }
   return r;
}

// MVector + MVector
MVector operator+(const MVector &lhs, const MVector &rhs)
{
   assert(lhs.size() == rhs.size());

   MVector r(lhs);
   for (int i=0; i<lhs.size(); i++)
      r[i] += rhs[i];

   return r;
}

// MVector - MVector
MVector operator-(const MVector &lhs, const MVector &rhs)
{
   assert(lhs.size() == rhs.size());

   MVector r(lhs);
   for (int i=0; i<lhs.size(); i++)
      r[i] -= rhs[i];

   return r;
}

// MMatrix = MVector <outer product> MVector
// M = a <outer product> b
MMatrix OuterProduct(const MVector &a, const MVector &b)
{
   MMatrix m(a.size(), b.size());
   for (int i=0; i<a.size(); i++)
   {
      for (int j=0; j<b.size(); j++)
      {
         m(i,j) = a[i]*b[j];
      }
   }
   return m;
}

// Hadamard product
MVector operator*(const MVector &a, const MVector &b)
{
   assert(a.size() == b.size());

   MVector r(a.size());
   for (int i=0; i<a.size(); i++)
      r[i]=a[i]*b[i];
   return r;
}

// double * MMatrix
MMatrix operator*(double d, const MMatrix &m)
{
```

```
   MMatrix r(m);
   for (int i=0; i<m.Rows(); i++)
      for (int j=0; j<m.Cols(); j++)
         r(i,j)*=d;

   return r;
}

// double * MVector
MVector operator*(double d, const MVector &v)
{
   MVector r(v);
   for (int i=0; i<v.size(); i++)
      r[i]*=d;

   return r;
}

// MVector -= MVector
MVector operator-=(MVector &v1, const MVector &v)
{
   assert(v1.size()==v.size());

   for (int i=0; i<v1.size(); i++)
      v1[i]-=v[i];

   return v1;
}

// MMatrix -= MMatrix
MMatrix operator-=(MMatrix &m1, const MMatrix &m2)
{
   assert (m1.Rows() == m2.Rows() && m1.Cols() == m2.Cols());

   for (int i=0; i<m1.Rows(); i++)
      for (int j=0; j<m1.Cols(); j++)
         m1(i,j)-=m2(i,j);

   return m1;
}

// Output function for MVector
inline std::ostream &operator<<(std::ostream &os, const MVector &rhs)
{
   std::size_t n = rhs.size();
   os << "(";
   for (std::size_t i=0; i<n; i++)
   {
      os << rhs[i];
      if (i!=(n-1)) os << ", ";
   }
   os << ")";
   return os;
}

// Output function for MMatrix
inline std::ostream &operator<<(std::ostream &os, const MMatrix &a)
{
   int c = a.Cols(), r = a.Rows();
   for (int i=0; i<r; i++)
   {
      os<<"(";
      for (int j=0; j<c; j++)
```

```
      {
         os.width(10);
         os << a(i,j);
         os << ((j==c-1)?')':',');
      }
      os << "\n";
   }
   return os;
}


////////////////////////////////////////////////////////////////////////////
// Functions that provide sets of training data

// Generate 16 points of almost linearly separable training data
void GetTestData(std::vector<MVector> &x, std::vector<MVector> &y)
{
   x = {{0.125,.175}, {0.375,0.3125}, {0.05,0.675}, {0.3,0.025}, {0.15,0.3}, {0.25,0.5}, {0.2,0.95}, {0.15,
       0.85},
       {0.75, 0.5}, {0.95, 0.075}, {0.4875, 0.2}, {0.725,0.25}, {0.9,0.875}, {0.5,0.8}, {0.25,0.75},
           {0.5,0.5}};

   y = {{1},{1},{1},{1},{1},{1},{1},{1},
       {-1},{-1},{-1},{-1},{-1},{-1},{-1},{-1}};
}

// Generate 1000 points of test data in a checkerboard pattern
void GetCheckerboardData(std::vector<MVector> &x, std::vector<MVector> &y)
{
   std::mt19937 lr;
   x = std::vector<MVector>(1000, MVector(2));
   y = std::vector<MVector>(1000, MVector(1));
    std::cout << x[0] << std::endl;

   for (int i=0; i<1000; i++)
   {
      x[i]={lr()/static_cast<double>(lr.max()),lr()/static_cast<double>(lr.max())};
      double r = sin(x[i][0]*12.5)*sin(x[i][1]*12.5);
      y[i][0] = (r>0)?1:-1;
   }
}


// Generate 1000 points of test data in a spiral pattern
void GetSpiralData(std::vector<MVector> &x, std::vector<MVector> &y)
{
   std::mt19937 lr;
   x = std::vector<MVector>(1000, MVector(2));
   y = std::vector<MVector>(1000, MVector(1));

   double twopi = 8.0*atan(1.0);
   for (int i=0; i<1000; i++)
   {
      x[i]={lr()/static_cast<double>(lr.max()),lr()/static_cast<double>(lr.max())};
      double xv=x[i][0]-0.5, yv=x[i][1]-0.5;
      double ang = atan2(yv,xv)+twopi;
      double rad = sqrt(xv*xv+yv*yv);

      double r=fmod(ang+rad*20, twopi);
      y[i][0] = (r<0.5*twopi)?1:-1;
   }
}
```

```
// Save the the training data in x and y to a new file, with the filename given by "filename"
// Returns true if the file was saved succesfully
bool ExportTrainingData(const std::vector<MVector> &x, const std::vector<MVector> &y,
                std::string filename)
{
   // Check that the training vectors are the same size
   assert(x.size()==y.size());

   // Open a file with the specified name.
   std::ofstream f(filename);

   // Return false, indicating failure, if file did not open
   if (!f)
   {
      return false;
   }

   // Loop over each training datum
   for (unsigned i=0; i<x.size(); i++)
   {
      // Check that the output for this point is a scalar
      assert(y[i].size() == 1);

      // Output components of x[i]
      f << "[[ ";
      for (int j=0; j<x[i].size()-1; j++)
      {
         f << x[i][j] << ", ";
      }
      f << x[i][x[i].size()-1];
      f << "], ";

      // Output only component of y[i]
      f << y[i][0] << "]" << std::endl;
   }
   f.close();

   if (f) return true;
   else return false;
}




////////////////////////////////////////////////////////////////////////////////
// Neural network class

class Network
{
public:

   // Constructor: sets up vectors of MVectors and MMatrices for
   // weights, biases, weighted inputs, activations and errors
   // The parameter nneurons_ is a vector defining the number of neurons at each layer.
   // For example:
   //  Network({2,1}) has two input neurons, no hidden layers, one output neuron
   //
   //  Network({2,3,3,1}) has two input neurons, two hidden layers of three neurons
   //                  each, and one output neuron
   Network(std::vector<unsigned> nneurons_)
   {
      nneurons = nneurons_;
      nLayers = nneurons.size();
```

```
   weights = std::vector<MMatrix>(nLayers);
   biases = std::vector<MVector>(nLayers);
   errors = std::vector<MVector>(nLayers);
   activations = std::vector<MVector>(nLayers);
   inputs = std::vector<MVector>(nLayers);
   // Create activations vector for input layer 0
   activations[0] = MVector(nneurons[0]);

   // Other vectors initialised for second and subsequent layers
   for (unsigned i=1; i<nLayers; i++)
   {
      weights[i] = MMatrix(nneurons[i], nneurons[i-1]);
      biases[i] = MVector(nneurons[i]);
      inputs[i] = MVector(nneurons[i]);
      errors[i] = MVector(nneurons[i]);
      activations[i] = MVector(nneurons[i]);
   }

   // The correspondence between these member variables and
   // the LaTeX notation used in the paper is:
   //
   // C++                    LaTeX
   // ------------------------------------
   // inputs[l-1][j-1]    =  z_j^{[l]}
   // activations[l-1][j-1] = a_j^{[l]}
   // weights[l-1](j-1,k-1) = W_{jk}^{[l]}
   // biases[l-1][j-1]    =  b_j^{[l]}
   // errors[l-1][j-1]    =  \delta_j^{[l]}
   // nneurons[l-1]       =  n_l
   // nLayers             =  L

}

// Return the number of input neurons
unsigned NInputNeurons() const
{
   return nneurons[0];
}

// Return the number of output neurons
unsigned NOutputNeurons() const
{
   return nneurons[nLayers-1];
}

// Evaluate the network for an input x and return the activations of the output layer
MVector Evaluate(const MVector &x)
{
   // Call FeedForward(x) to evaluate the network for an input vector x
   FeedForward(x);

   // Return the activations of the output layer
   return activations[nLayers-1];
}


// Implement the training algorithm
bool Train(const std::vector<MVector> x, const std::vector<MVector> y,
        double initsd, double learningRate, double costThreshold, int maxIterations, int checkpoint =
            100000, bool write = false, std::string path = "./train.txt", double mean = 0)
{
   // Check that there are the same number of training data inputs as outputs
   assert(x.size() == y.size());
```

```
   std::ofstream f(path, std::ios_base::app);

   // Initialise the weights and biases with the standard deviation "initsd"
   InitialiseWeightsAndBiases(initsd, mean);

   for (int iter=1; iter<=maxIterations; iter++)
   {
      // Choose a random training data point i in {0, 1, 2, ..., N}
      int i = rnd() % x.size();

      // Run the feed-forward algorithm
      FeedForward(x[i]);

      // Run the back-propagation algorithm
      BackPropagateError(y[i]);

      // Update the weights and biases using stochastic gradient with learning rate "learningRate"
      UpdateWeightsAndBiases(learningRate);

      // Every so often, show an update on how the cost function has decreased
      if ((!(iter%checkpoint)) || iter==maxIterations)
      {
         // Calculate the total cost
         double tc = TotalCost(x, y), ac = Accuracy(x, y);

         // Display the iteration number and total cost to the screen
         if (write) {f << "[" << iter << "," << tc << "," << ac << "]" << std::endl;}
         else
         {std::cout << "Iteration: " << iter << "\n" << "Total cost: " << tc << "\n" << "Accuracy: " <<
             ac << std::endl;}

         // Return from this method with a value of true,
         //                 indicating success, if this cost is less than "costThreshold".
         if (tc < costThreshold) return true;
      }

   } // Continue the loop, until we have taken "maxIterations" steps

   // Return "false", indicating that the training did not succeed.
   return false;
}


// For a neural network with two inputs x=(x1, x2) and one output y,
// loop over (x1, x2) for a grid of points in [0, 1]x[0, 1]
// and save the value of the network output y evaluated at these points
// to a file. Returns true if the file was saved successfully.
bool ExportOutput(std::string filename)
{
   // Check that the network has the right number of inputs and outputs
   assert(NInputNeurons()==2 && NOutputNeurons()==1);

   // Open a file with the specified name.
   std::ofstream f(filename);

   // Return false, indicating failure, if file did not open
   if (!f)
   {
      return false;
   }

   // generate a matrix of (250x250) 750x750 output data points
```

```
      for (int i=0; i<=750; i++)
      {
         f << "[";
         for (int j=0; j<750; j++)
         {
            MVector out = Evaluate({i/750.0, j/750.0});
            if (out[0] < 0) out[0] = 0;
            else out[0] = 1;
            f << out[0] << ", ";
         }
         MVector out = Evaluate({i/750.0, 1});
         if (out[0] < 0) out[0] = 0;
         else out[0] = 1;
         f << out[0] << "";
         f << "],";
      }
      f.close();

      if (f) return true;
      else return false;
   }
   //tests of the member functions
   static void Test2();

   static bool Test3();

   static bool Test();

private:
   // Return the activation function sigma
   double Sigma(double z)
   {
      return tanh(z);
      //if (z>0) {return z;} return 0.1*z; // ReLU
   }

   // Return the derivative of the activation function
   double SigmaPrime(double z)
   {
      return 1.0 - tanh(z)*tanh(z);
      //if (z>0) {return 1;} return 0.1; //ReLU derivative
   }

   MVector Sigma(const MVector &z) //sigma definition for a vector
   {
      MVector temp(z.size());
      for (int i = 0; i < z.size(); i++)
      {  temp[i] = Sigma(z[i]); }
      return temp;
   }

   MVector SigmaPrime(const MVector &z)// sigma prime definition for a vector
   {
      MVector temp(z.size());
      for (int i = 0; i < z.size(); i++)
      {  temp[i] = SigmaPrime(z[i]); }
      return temp;
   }

   // Looping over all weights and biases in the network and setting each
   // term to a random number normally distributed with mean 0 and
   // standard deviation "initsd"
   void InitialiseWeightsAndBiases(double initsd, double mean = 0)
```

```
{
   // Make sure the standard deviation supplied is non-negative
   assert(initsd>=0);

   // Set up a normal distribution with mean zero, standard deviation "initsd"
   // Calling "dist(rnd)" returns a random number drawn from this distribution
   std::normal_distribution<> dist(mean, initsd);

   // Loop over all components of all the weight matrices
   //      and bias vectors at each relevant layer of the network.

   for (int l = 1; l < nLayers; l++)//looping through the layers
   {
      for (unsigned b = 0; b < nneurons[l]; b++)//looping through bias vector entries
      {
         biases[l][b] = dist(rnd); //updating the bias
      }

      for (unsigned i = 0; i < nneurons[l]; i++)//looping through the weight rows
      {
         for (unsigned j = 0; j < nneurons[l-1]; j++)//looping through weight columns
         {
            weights[l](i,j) = dist(rnd); //updating the weight
         }
      }
   }
}

// Evaluate the feed-forward algorithm, setting weighted inputs and activations
// at each layer, given an input vector x
void FeedForward(const MVector &x)
{
   // Check that the input vector has the same number of elements as the input layer
   assert(x.size() == nneurons[0]);

   // The feed-forward algorithm
   inputs[0] = x;
   activations[0] = x;

   for (unsigned l = 1; l < nLayers; l++)
   {
      inputs[l] = weights[l] * activations[l-1] + biases[l];
      activations[l] = Sigma(inputs[l]);
   }
}

// Evaluate the back-propagation algorithm, setting errors for each layer
void BackPropagateError(const MVector &y)
{
   // Check that the output vector y has the same number of elements as the output layer
   assert(y.size() == nneurons[nLayers - 1]);

   // The back-propagation algorithm
   //errors for the last layer
   errors[nLayers - 1] = SigmaPrime(inputs[nLayers - 1]) * (activations[nLayers - 1] - y);

   //looping through the remaining layers in reverse order
   for (unsigned l = nLayers - 2; l > 0; l--)
   {
      errors[l] = SigmaPrime(inputs[l]) * TransposeTimes(weights[l+1], errors[l+1]);
   }
}
```

```
// Apply one iteration of the stochastic gradient iteration with learning rate eta.
void UpdateWeightsAndBiases(double eta)
{
   // Check that the learning rate is positive
   assert(eta>0);

   // Update the weights and biases according to the stochastic gradient iteration
   for (int l = 1; l < nLayers; l++)//looping through the layers
   {
      for (unsigned b = 0; b < nneurons[l]; b++)//looping through bias vector entries
      {
         biases[l][b] -= eta * errors[l][b]; //updating the bias
      }

      for (unsigned i = 0; i < nneurons[l]; i++)//looping through the weight rows
      {
         for (unsigned j = 0; j < nneurons[l-1]; j++)//looping through weight columns
         {
            weights[l](i,j) -= eta * errors[l][i] * activations[l-1][j]; //updating the weight
         }
      }
   }
}


// Return the cost function of the network with respect to the desired output y
double Cost(const MVector &y)
{
   // Check that y has the same number of elements as the network has outputs
   assert(y.size() == nneurons[nLayers-1]);

   // TODO: Return the cost associated with this output
   double cost = 0, difference; //intialising the variables
   for (unsigned i = 0; i < y.size(); i++) //looping through y components
   {
   difference = y[i] - activations[nLayers - 1][i];
   cost += difference * difference; //summing the cost over components
   }
   cost /= (y.size() * 2);//normalising the cost
   return cost;
}

// Return the total cost C for a set of training data x and desired outputs y
double TotalCost(const std::vector<MVector> x, const std::vector<MVector> y)
{
   // Check that there are the same number of inputs as outputs
   assert(x.size() == y.size());

   double cost = 0;
   for (unsigned i = 0; i < y.size(); i++)//looping through target vectors
   {
      FeedForward(x[i]); //evaluating the network for a given data point
      cost += Cost(y[i]);//summing the total cost of all targets
   }
   return cost;
}

//accuracy for 2 class classification
double Accuracy(const std::vector<MVector> x, const std::vector<MVector> y)
{
   // Check that there are the same number of inputs as outputs
   assert(x.size() == y.size());
```

```
      double acc = 0;
      for (unsigned i = 0; i < y.size(); i++)
      {
         //counting total true positives
         FeedForward(x[i]);
         if ((activations[nLayers-1][0] > 0 && y[i][0] > 0) || (activations[nLayers-1][0] < 0 && y[i][0] <
             0)) acc += 1;
      }
      acc /= x.size();
      acc *= 100; //displaying as a percentage
      return acc;
   }

   // Private member data

   std::vector<unsigned> nneurons;
   std::vector<MMatrix> weights;
   std::vector<MVector> biases, errors, activations, inputs;
   unsigned nLayers;

};

bool Network::Test3() //outputs values to be seen and examined
{
   Network n({1,1,1});
   n.InitialiseWeightsAndBiases(1, 0);
   n.FeedForward({1});
   std::cout << n.activations[2] << std::endl;
   n.BackPropagateError({1});
   std::cout << n.errors[2] << std::endl;
   std::cout << n.weights[2](0,0) << std::endl;
   n.UpdateWeightsAndBiases(0.1);
   std::cout << n.weights[2](0,0) << std::endl;

   return true;
}

//test that outputs average absolute parameter values during training to a file
void Network::Test2() //comment out parameter initialisation in train for this to work as intended
{
   Network n({2, 3, 3, 1}); //initialising the network
   n.InitialiseWeightsAndBiases(0.1);
   std::vector<MVector> x, y;
   GetTestData(x, y); //getting training data
   std::string path, costs = "./1/contour_costs.txt", params = "./1/average_params.txt";

   //summing up the absolute values of the parameters
   double total = 0;
      for (unsigned u = 1; u < n.nLayers; u++) //looping through layers
      {
         for (int i = 0; i < n.weights[u].Rows(); i++) //looping through neurons
         {
            total += std::abs(n.biases[u][i]);
            for (int j = 0; j < n.weights[u].Cols(); j++)
            {
               total += std::abs(n.weights[u](i,j));
            }
         }
      }
      std::ofstream f(params, std::ios_base::app);
      f << (total / 25.0) << std::endl; //writing the average value to a file
```

```
   for (int i = 0; i < 10; i++) //looping through training iterations
   {
      std::cout << i << std::endl;
      path = "./1/contour_" + std::to_string(i) + ".txt";
      n.Train(x, y, 0.1, 0.1, 1e-12, 5000, 500, true, costs); //trianing
      n.ExportOutput(path);

      double total = 0;
      for (unsigned u = 1; u < n.nLayers; u++)
      {
         for (int i = 0; i < n.weights[u].Rows(); i++)
         {
            total += std::abs(n.biases[u][i]);
            for (int j = 0; j < n.weights[u].Cols(); j++)
            {
               total += std::abs(n.weights[u](i,j));
            }
         }
      }
      std::ofstream f(params, std::ios_base::app);
      f << (total / 25.0) << std::endl;
   }
}

bool Network::Test()
{
   // This function is a static member function of the Network class:
   // it acts like a normal stand-alone function, but has access to private
   // members of the Network class. This is useful for testing, since we can
   // examine and change internal class data.
   //
   // This function should return true if all tests pass, or false otherwise

   // A example test of FeedForward, BackPropagateError and UpdateWeightsAndiases
   {
      // Make a simple network with two weights and one bias
      Network n({2, 1});

      // Set the values of these by hand
      n.biases[1][0] = 0.5;
      n.weights[1](0,0) = -0.3;
      n.weights[1](0,1) = 0.2;

      // Call function to be tested with x = (0.3, 0.4)
      n.FeedForward({0.3, 0.4});

      // Display the output value calculated
      std::cout << n.activations[1][0] << std::endl;

      // Correct value is = tanh(0.5 + (-0.3*0.3 + 0.2*0.4))
      //                  = 0.454216432682259...
      // Fail if error in answer is greater than 10^-10:
      if (std::abs(n.activations[1][0] - 0.454216432682259) > 1e-10)
      {
         return false;
      }
      std::cout << "1st feed forward test passed" << std::endl;

      n.BackPropagateError({1}); //backpropagation with target y=1
      //fail if error is too large
      if (std::abs(n.errors[1][0] + 0.43318) >1e-5) return false;
      std::cout << "Bacpropagation test passsed" << std::endl;
```

```
   n.UpdateWeightsAndBiases(1e-2); //updating the parameters
   if (std::abs(n.weights[1](0,0) + 0.2987) > 1e-4) return false;
   std::cout << "Updating parameters test passed" << std::endl;
}

{  //test whether all the parameters have been initialised properly
   Network n({2,2,1});
   n.InitialiseWeightsAndBiases(5, 5);
   MVector b;
   MMatrix w;
   for (unsigned i =1; i< n.nLayers; i++)
   {
      b = n.biases[i];
      MVector b1(b.size()); //empty vector
      w = n.weights[i];
      MMatrix w1(w.Rows(), w.Cols()); //empty matrix
      std::cout << n.biases[i] << std::endl;
      if ((w == w1) || b == b1) {return false;} //comparing the weights and biases to the empty structures
      //std::cout << "Bias " << i << ": " << b << std::endl; //printing the biases
      //std::cout << "Weights " << i << ": " << w << std::endl; //printing the weights
   }
   std::cout << "parameter initalisation test passed" << std::endl;
}

{  //testing the activation functions and their derivatives
   Network n({1,1}); //initialising a network for access to the activation functions
   double a = 0.5, sa, spa; //value, sigma(value), sigma'(value)
   MVector b(2, 0.5), sb, spb;
   sa = n.Sigma(a);
   spa = n.SigmaPrime(a);
   sb = n.Sigma(b);
   spb = n.SigmaPrime(b);
   if (std::abs(sa - 0.46211715726) > 1e-10) return false;
   if (std::abs(sb[0] - 0.46211715726) > 1e-10) return false;
   if (std::abs(spa - 0.786447732966) > 1e-10) return false;
   if (std::abs(spb[0] - 0.786447732966) > 1e-10) return false;
   std::cout << "activation functions test passed" << std::endl;
}

{  //testing the feed forward function
   Network n({1,1,1,1});
   for (int i = 1; i < 4; i++)
   {
      n.biases[i][0] = 0; //initialising biases to 0
      n.weights[i](0,0) = 1; //initialising weights to 1
   }
   MVector x(1, 1); //input vector
   n.FeedForward(x);
   if (std::abs(n.activations[1][0] - 0.761594155956) > 1e-10) return false;
   if (std::abs(n.activations[2][0] - 0.642014992012) > 1e-10) return false;
   if (std::abs(n.activations[3][0] - 0.566269975961) > 1e-10) return false;
   std::cout << "feed forward test passed" << std::endl;

}

{
   //testing the cost functions
   Network n({1,1});
   n.weights[1](0,0) = 1;
   n.biases[1][0] = 0;

   n.FeedForward({1});
```

```
      if (std::abs(n.Cost({0}) - 0.290012829193) > 1e-10) {return false;}
      std::cout << "Cost test passed" << std::endl;

      std::vector<MVector> x, y;
      MVector x1(1,1), y1(1,0);

      for (int i = 0; i<5; i++)
      {x.push_back(x1); y.push_back(y1);}

      double total = n.TotalCost(x, y);
      if (std::abs(total - 5*0.290012829193) > 1e-10) return false;
      std::cout << "Total cost test passed" << std::endl;
   }

   {  //testing behaviour with parameters being 0
      std::normal_distribution<> dist(0, 0);
      Network n({1,1});
      n.weights[1](0,0) = dist(rnd);
      n.biases[1][0] = dist(rnd);
      n.FeedForward({1});
      std::cout << n.activations[1] << std::endl;

      n.BackPropagateError({1});
      std::cout << n.errors[1] << std::endl;

      n.UpdateWeightsAndBiases(1e-2);
      std:: cout << n.weights[1](0,0) << ", " << n.biases[1][0] << std::endl;
   }

   return true;
}

////////////////////////////////////////////////////////////////////////////
// Main function and example use of the Network class

// Create, train and use a neural network to classify the data from GetTestData()
void ClassifyTestData()
{
   // Create a network with two input neurons, two hidden layers of three neurons, and one output neuron
   Network n({2, 3, 3, 1});

   // Get some data to train the network
   std::vector<MVector> x, y;
   GetTestData(x, y);

   // Train network on training inputs x and outputs y
   // Numerical parameters are:
   //  initial weight and bias standard deviation = 0.1
   //  learning rate = 0.1
   //  cost threshold = 1e-4
   //  maximum number of iterations = 10000
   bool trainingSucceeded = n.Train(x, y, 0.1, 0.1, 1e-4, 1000000);

   // If training failed, report this
   if (!trainingSucceeded)
   {
      std::cout << "Failed to converge to desired tolerance." << std::endl;
   }

   // Generate some output files for plotting
   ExportTrainingData(x, y, "./test_points.txt");
   n.ExportOutput("./test_contour_1.txt");
}
```

```
int main_average_parameters()
{
   Network::Test2(); //in order for this to work as intended comment out the InitialiseWeightsAndBiases()
       in Network.Train() so that params don't get reset
   //ClassifyTestData();
}

int main__1()
{
   // Call the test function
   bool testsPassed = Network::Test();

   // If tests did not pass, something is wrong; end program now
   if (!testsPassed)
   {
      std::cout << "A test failed." << std::endl;
      return 1;
   }
   std::cout << "Passed";
   // Tests passed, so run our example program.
   //ClassifyTestData();

   return 0;
}

int main_2() //examining convergence for different learning rates
{
   std::vector<std::string> fs({"./convergence_eta=0.001.txt", "./convergence_eta=0.003.txt",
       "./convergence_eta=0.01.txt", "./convergence_eta=0.03.txt", "./convergence_eta=0.1.txt",
       "./convergence_eta=0.3.txt", "./convergence_eta=1.txt"});
   MVector etas(7, 0.001);
   etas[1] = 0.003; etas[2] = 0.01; etas[3] = 0.03; etas[4] = 0.1; etas[5] = 0.3; etas[6] = 1;
   std::vector<MVector> x, y;
   GetTestData(x, y);

   for (unsigned i = 0; i < etas.size(); i++)
   {
      Network n({2,3,3,1});
      n.Train(x, y, 0.1, etas[i], 1e-12, 10000000, 1000, true, fs[i]);
   }
   return 0;
}

int main_3() //examining the effect of initial distribution on convergence
{
   //using eta = 0.01;
   std::string base = "./distributions/1/_test_distribution_", path;
   std::vector<double> means({0.0}); //{0, 0.5, 1, 2, 3, 4}
   std::vector<double> sds({3.0}); //{0, 0.5, 1, 2, 3, 4}
   std::vector<MVector> x, y;
   GetTestData(x, y);


   for (unsigned i=0; i<means.size(); i++)
   {
      for (unsigned j=0; j<sds.size(); j++)
      {
      path = base + "mean_" + std::to_string(means[i]) + "sd_" + std::to_string(sds[j]) + ".txt";
      std::cout << path << std::endl;
      Network n({2,3,3,1});
      n.Train(x, y, sds[j], 0.01, 1e-12, 1e+8, 1e+5, true, path, means[i]);
      }
```

```
   }
   return 0;
}

int main_export_data()
{
    std::vector<MVector> x, y;
   GetCheckerboardData(x, y);
    ExportTrainingData(x, y, "./task_4/real/checkerboard_data.txt");
    return 0;
}


int main_4() //task 4
{
   std::vector<unsigned> layers({3,4,5,6}), neurons({6, 10, 15, 25, 40});
   std::string base = "./task_4/real/spiral_network_size_", path, path1;
   for (unsigned i =0 ; i < layers.size(); i++) //number of layers
   {
      for (unsigned j = 0; j < neurons.size(); j++)
      {
         std::vector<unsigned> nlayers(layers[i], neurons[j]); //vector to initialise the network
         nlayers[0] = 2; nlayers[layers[i]-1]=1; //numbers of neurons in first and last layers are
             predetermined

         Network n(nlayers);
         std::vector<MVector> x, y;
         GetSpiralData(x, y);

         path = base + "layers_" + std::to_string(layers[i]) + "_neurons_" + std::to_string(neurons[j]) +
             ".txt"; //file path to save the results to
            path1 = base + "contour_" + "layers_" + std::to_string(layers[i]) + "_neurons_" +
                std::to_string(neurons[j]) + ".txt"; //save the contour
         std::cout << path << std::endl;
         n.Train(x, y, 0.1, 0.003, 1e-12, 1e+7, 1e+5, true, path, 0); //write the convergence results to a
             file
            n.ExportOutput(path1);
      }
   }

   return 0;
}
```

Code for the MVector class:

```
    #ifndef MVECTOR_H // the 'include guard'
#define MVECTOR_H // see C++ Primer Sec. 2.9.2

#include <vector>

// Class that represents a mathematical vector
class MVector
{
public:
   // constructors
   MVector() {}
   explicit MVector(int n) : v(n) {}
   MVector(int n, double x) : v(n, x) {}
   MVector(std::initializer_list<double> l) : v(l) {}

   // access element (lvalue) (see example sheet 5, q5.6)
   double &operator[](int index)
   {
```

```
      return v[index];
   }

   // access element (rvalue) (see example sheet 5, q5.7)
   double operator[](int index) const {
      return v[index];
   }

   bool operator==(const MVector &r) //comparison operator
   {
      for (unsigned u = 0; u<v.size(); u++)
      {
         if (v[u] != r[u]) {return false;} //false if any element doesn't match
      }
      return true;
   }

   int size() const { return v.size(); } // number of elements

private:
   std::vector<double> v;
};

#endif
```

Code for the MMatrix class:

```
#ifndef MMATRIX_H // the 'include guard'
#define MMATRIX_H

#include <vector>
#include <iostream>

// Class that represents a mathematical matrix
class MMatrix
{
public:
   // constructors
   MMatrix() : nRows(0), nCols(0) {}
   MMatrix(int n, int m, double x = 0) : nRows(n), nCols(m), A(n * m, x) {}

   // set all matrix entries equal to a double
   MMatrix &operator=(double x)
   {
      for (unsigned i = 0; i < nRows * nCols; i++) A[i] = x;
      return *this;
   }

   // access element, indexed by (row, column) [rvalue]
   double operator()(int i, int j) const
   {
      return A[j + i * nCols];
   }

   // access element, indexed by (row, column) [lvalue]
   double &operator()(int i, int j)
   {
      return A[j + i * nCols];
   }

   // size of matrix
   int Rows() const { return nRows; }
   int Cols() const { return nCols; }
```

```
   bool operator==(const MMatrix &r) //comparison operator
   {
      int row, col;
      for (unsigned u=0; u<A.size(); u++)
      {
         col = u % r.Cols();
         row = (u - col) / r.Cols();
         if (A[u] != r(row ,col)) {return false;} //false if any element doesn't match
      }
      return true;
   }

private:
   unsigned int nRows, nCols;
   std::vector<double> A;
};

#endif
```