Project 2: Neural Network and Machine Learning

Student No: 10423513

Abstract

Deep Learning is the most essential technology in Artificial Intelligence, which has been applied in various fields including Face Recognition and Automobiles. Deep Learning is usually referred as deep network using machine learning methodology. In this project we start from a simple Feedforward Neural Network to achieve binary classification. We will also focus on some improvements on the network provided in the project.

1 Introduction

An artificial neural network (ANN) consists of a set of neurons, each neuron is a simple scalar functional unit with a single input and output, modelled by an nonlinear activation function σ. A certain number of these neurons form a layer and an amount of layers then compose the network. In the information flow perspective, there are two types of ANNs:

- **Feed-forward Neural Networks(FNNs)**: The information only flows forward and does not influence any neurons in the current or previous layers, meaning there are no loops in the network.
- **Recurrent Neural Networks(RNNs)** : The information can feed back and keep some states. It is like a network with a function of memorising passed data and use it to impact other data. This feature makes it applicable for some tasks that are thorny for FNNs such as Handwritten Recognition and Speech Recognition [Graves et al, 2009]. The overview of the CNN, see [1]

In the Feed-forward world, we can also distinguish them by two categories:

- **Fully Connected FFNs** : The network we will introduce and apply in this project. In such a network, all nodes in two adjacent layers and the connections between them form a completed bipartite graph. Since all neurons in a layer are linked with neurons in the next layer, it may cause over-fit problems, and it requires massive computation budget to achieve some complicated tasks.
- **Convolutional Neural Networks(CNNs)** : This kind of networks have tackled many disadvantages faced by fully connected cases, they are now widely applied in computer vision problems. Local connectivity is a core feature of CNNs that neurons in convolutional layer do not go through all the neurons in the previous layer, only some spatial arrangements are caught (the edge for example). Usually a filter is established here, and it is used to convolute through neurons in the previous layer in convolutional layer. In such manner, the network can easily handle those tasks requiring millions of weights&biases and reduce them to only hundreds. An overview of how CNNs work, see [2]

When we use ANNs to train some labelled data(supervised learning), **overfitting** is a common problem. It meaning that the trained model performs very well for training data, but faces difficulty when encountering new data hence the predictions are usually unsatisfied. It is because the training process emphasises too heavily on unimportant features, where noise may be produced. A common approach to prevent overfitting is to split the data into 3 different pools:

[1] Figure 9(a), Appendix A.
[2] Figure 9(b), Appendix A.

- **Training data**: Used during the updating process and help train the model.
- **Validation data**: It has no effect on the updating process, and it is used to measure the performance of the current network on unseen data.
- **Test data**: This is similar to Validation data, but it is used for the finally trained network. It sketches how our model behaves on unseen data after training.

★[3] In practical applications, people also implement some improvements during the training process to avoid overfitting. For example, [Srivastava et al, 2014] introduces a **dropout** technique in preventing overfitting, by randomly drop units from the training process. It has been showed to enhance the performance of networks in many supervised learning problems.

In this project I will use a simple feed-forward neural network to achieve binary-classification task. I will start with some basic background about how the network works, then I will train the network and use it to classify three datasets. Comparisons between different models will also be included, in terms of performances in classifying the data and the convergence rate. More on binary-classification can be found on [Higham, 2018].

2 Technical Background

In this section I will demonstrate how the FNN works in terms of algorithms and mathematical tools applied inside the 'Black Box'. Proofs of some mathematical methods will be provided.

2.1 Evaluating the output

Let L denote the number of layers in the network and n_l represent the number of neurons at layer $l, for\ l = 1, 2, ..., L$. The input of the jth neuron at layer l is denoted by $z_j^{[l]}$ and the output of the jth neuron at layer l by $a_j^{[l]}$. The activation function behaves as follow:

$$a_j^{[l]} = \sigma(z_j^{[l]}), \quad for\ l = 2, 3, ..., L; \quad j = 1, ..., n_l. \tag{1}$$

$$a_j^{[l]} = x_j, \quad j = 1, ..., n_l. \tag{2}$$

In a vector form it becomes

$$\mathbf{a}^{[l]} = \sigma(\mathbf{z}^{[l]}), \quad for\ l = 2, 3, ..., L. \tag{3}$$

Basically, the activation σ is a nonlinear monotonically increasing function such as **Sigmoid** function. It can also have other functional forms such as **ReLU** function ($f(x) = max(0, x)$). In this project, one common choice is the tanh function,

$$\sigma(z) = tanh(z) \tag{4}$$

★In practical uses many will apply ReLU instead of a Sigmoid function because it is considered being closer to some biological features. And it can benefit in some optimisation algorithms including back-propagation. See [Prajit, 2017].

Next we proceed to the connections between neurons at different layers. Specifically, a neuron input $z_j^{[l]}$ is a biased linear combination of the outputs of the neurons in layer $l-1$,

$$z_j^{[l]} = \sum_{k=1}^{n_l - 1} (w_{jk}^{[l]} a_k^{[l-1]}) + b_j^{[l]}, \quad for\ l = 2, 3, ..., L, \quad j = 1, ..., n_l. \tag{5}$$

where $w_{jk}^{[l]}$ are the weights and $b_j^{[l]}$ are the biases at layer l. Alternatively, we can also write (5) in a vector form:

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]} \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} \quad for\ l = 2, 3, ..., L. \tag{6}$$

[3] ★ expresses some further discussions on the topic

Combining this with (2) and (3) we find

$$\mathbf{a}^{[l]} = \sigma(\mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}) \quad for\ l = 2, 3, ..., L. \tag{7}$$

$$\mathbf{a}^{[1]} = \mathbf{x} \tag{8}$$

Equations (7) and (8) together define the *feed-forward* algorithm. The network in turn obtains $\mathbf{z}^{[2]}$, $\mathbf{a}^{[2]}$, $\mathbf{z}^{[3]}$, $\mathbf{a}^{[3]}, ..., \mathbf{z}^{[L]}$, $\mathbf{a}^{[L]}$. Weights and biases are adjusted when the network is learning from the data, after the training all of them remain fixed and can be used to evaluate the output given an input vector $\mathbf{x}$.

Exercise 1.3.4(1): Here comes a question, what if the activation function is linear such as $\sigma(z) = z$. Suppose a FNN with L layers, then

$$\mathbf{a}^{[1]} = \mathbf{x}$$

$$\mathbf{a}^{[2]} = \mathbf{W}^{[2]}\mathbf{x} + \mathbf{b}^{[2]}$$

$$\mathbf{a}^{[3]} = \mathbf{W}^{[3]}(\mathbf{W}^{[2]}\mathbf{x} + \mathbf{b}^{[2]}) + \mathbf{b}^{[3]}$$

$$\dots$$

$$\mathbf{a}^{[L]} = \mathbf{W}^{[L]}(\mathbf{W}^{[L-1]}\mathbf{a}^{[L-2]}(\dots) + \mathbf{b}^{[L-1]}) + \mathbf{b}^{[L]}$$

Obviously $\mathbf{a}^{[2]}$ is a linear function dependent on $\mathbf{x}$(with matrix calculation) and $\mathbf{b}^{[2]}$ is a constant vector. And $\mathbf{a}^{[3]}$ is a linear function of $\mathbf{a}^{[2]}$. As we know a linear combination of linear functions keeps the linearity. So $\mathbf{a}^{[L]}$ is then linearly dependent on $\mathbf{x}$,

$$\mathbf{a}^{[L]} = \mathbf{M}\mathbf{x} + \mathbf{c} \tag{9}$$

where the matrix $\mathbf{M}$ is

$$\mathbf{M} = \mathbf{W}^{[L]}\mathbf{W}^{[L-1]} \dots \mathbf{W}^{[2]} \tag{10}$$

and $\mathbf{c}$ is a constant vector. **In this case, the neural network is just a linear regression model, where hidden layers are literally meaningless** because the output layer $\mathbf{a}^{[L]}$ can always be expressed by a linear relationship with input $\mathbf{x}$. Intuitively, to achieve this we just need one output layer.

2.2 Training the network

As I mentioned at previous sections, how the neural network learns from the data finally turns to adjust weights and biases(parameters). Here we introduce the cost to judge whether a network is successful or not. The cost of data point $\mathbf{x}^{\{i\}}$ reflects the closeness between output $\mathbf{a}^{[L]}(\mathbf{x}^{\{i\}})$ produced by the network and the desired output $\mathbf{y}^{\{i\}}$.

$$C_{\mathbf{x}^{\{i\}}} = \frac{1}{2}\|\mathbf{y}^{\{i\}} - \mathbf{a}^{[L]}(\mathbf{x}^{\{i\}})\|_2^2 \tag{11}$$

And the total cost of N inputs defined as:

$$C = \frac{1}{N}\sum_{i=1}^{N} C_{\mathbf{x}^{\{i\}}} \tag{12}$$

$$\|\mathbf{x}\|_2^2 = x_1^2 + x_2^2 + \dots + x_n^2. \tag{13}$$

The process of training is equivalent to minimising the total cost, which a nonlinear function of $\mathbf{p}$, where $\mathbf{p}$ represents all the parameters(weights and biases).

Steepest gradient. In order to minimise C($\mathbf{p}$), here we will firstly introduce the steepest gradient descent method. By using Taylor series and Cauchy-Schwartz inequality[See [Higham, 2018]]. We give a final expression of adjustment of parameter $\mathbf{p}$.

$$\mathbf{p} \leftarrow \mathbf{p} + \Delta\mathbf{p} = \mathbf{p} - \eta\nabla C(\mathbf{p}) \tag{14}$$

This is the steepest gradient descent(usually called gradient descent) to adjust the parameters. The positive constant η represents the learning rate. If conditions are ideal then it can reach the global minimum after a number of steps. However, minimisation of a nonlinear function in high dimensions is fundamentally difficult. First, we need to choose a balanced learning rate η. Then we need massive computation budget to perform the gradient descent algorithm. Thus, instead of targetting on the global minimum of $C(\mathbf{p})$, it is better to look for a value of $\mathbf{p}$ that makes $C(\mathbf{p})$ less than some small threshold.

2.2.1 Stochastic gradient

In (14), every iteration of performing steepest gradient we need to evaluate $\nabla C(\mathbf{p})$, meaning that we need to run over all data points.

$$\nabla C(\mathbf{p}) = \frac{1}{N}\sum_{i=1}^{N} \nabla C_{\mathbf{x}^{\{i\}}}(\mathbf{p}) \tag{15}$$

Obviously evaluation of (15) is computationally expensive if the number of data set N is large. A faster alternative is to randomly choose only one data point at each step,

$$\mathbf{p} \leftarrow \mathbf{p} + \Delta\mathbf{p} = \mathbf{p} - \eta \nabla C_{\mathbf{x}^{\{i\}}}(\mathbf{p}) \tag{16}$$

This method is called stochastic gradient descent. Although in stochastic gradient the reduction in the total cost is smaller than that in steepest gradient-in fact many steps are likely to increase the total cost slightly-but since it can perform many more iterations the convergence is often quicker in a given time. There are different selections of data point i to be used. In this project we will simply randomly choose it independently of previous selections.

2.2.2 Back-propagation

In previous sections we introduced the process of the training by applying stochastic gradient. Now the point is, how can we calculate the differentiation of cost over parameters $\nabla C_{\mathbf{x}^{\{i\}}}(\mathbf{p})$? Here let us define the term **error**,

$$\delta_j^{[l]} = \frac{\partial C_{\mathbf{x}^{\{i\}}}}{\partial z_j^{[l]}} \quad for\ 1 \le j \le n_l \ \ and \ \ 2 \le l \le L. \tag{17}$$

From this we can obtain

$$\boldsymbol{\delta}^{[L]} = \sigma^{'}(\mathbf{z}^{[L]}) \circ (\mathbf{a}^{[L]} - \mathbf{y}) \tag{18}$$

where the operator $\circ$ is the componentwise product,

$$\mathbf{a} \circ \mathbf{b} = (a_1 b_1, a_2 b_2, \ldots, a_n b_n) \quad for\ \mathbf{a}, \mathbf{b}\ \in \Re^n. \tag{19}$$

Further combining with *feed-forward* equation (7) and chain rule, we can obtain the recurrence and the final expressions:

$$\boldsymbol{\delta}^{[l]} = \sigma^{'}(\mathbf{z}^{[l]}) \circ (\mathbf{W}^{[l+1]})^T \boldsymbol{\delta}^{[l+1]} \quad for\ l = 2, \ldots, L-1. \tag{20}$$

$$\frac{\partial C_{\mathbf{x}^{\{i\}}}}{\partial b_j^{[l]}} = \delta_j^{[l]}, \quad for \ \ l = 2, \ldots, L. \tag{21}$$

$$\frac{\partial C_{\mathbf{x}^{\{i\}}}}{\partial W_{jk}^{[l]}} = \delta_j^{[l]} a_k^{[l-1]} \quad for\ l = 2, \ldots, L. \tag{22}$$

Exercise1.3.4(2): Proof of equations (18), (20), (21), (22).

•Since

$$\boldsymbol{\delta}^{[L]} = \begin{pmatrix} \delta_1^{[L]} \\ \vdots \\ \delta_{n_L}^{[L]} \end{pmatrix}$$

Also we have

$$C_{\mathbf{x}^{\{i\}}} = \frac{1}{2}\|\mathbf{y}^{\{i\}} - \mathbf{a}^{[L]}(\mathbf{x}^{\{i\}})\|_2^2 = \frac{1}{2}\sum_{j=1}^{n_L}(y_j - a_j^{[L]})^2.$$

By using chain rule and from (21),

$$\delta_j^{[L]} = \frac{\partial C_{\mathbf{x}^{\{i\}}}}{\partial z_j^{[L]}} = \frac{\partial C_{\mathbf{x}^{\{i\}}}}{\partial a_j^{[L]}}\frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} = (a_j^{[L]} - y_j)\ \sigma'(z_j^{[L]})$$

Aggregate it into a vector then we have (22).

•From what we have obtained above, if we want to go back to evaluate errors in previous layers, such as $\delta_j^{[L-1]}$. Since $\delta_j^{[L-1]}$ matches the neuron of position j at layer $L-1$, and this neuron has connections with all neurons at layer L. So intuitively, if we want to evaluate $\delta_j^{[l]}$, we always need to evaluate $\delta_k^{[l+1]}$ $(k = 1, \ldots, n_{l+1})$ first.

$$\delta_j^{[l]} = \frac{\partial C_{\mathbf{x}^{\{i\}}}}{\partial z_j^{[l]}} = \sum_{k=1}^{n_{l+1}}\frac{\partial C_{\mathbf{x}^{\{i\}}}}{\partial z_k^{[l+1]}}\frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = \sum_{k=1}^{n_{l+1}}\delta_k^{[l+1]}\frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}}$$

Let $\mathbf{w}_k^{[l+1]}$ represent the k-th row of weights matrix $\mathbf{W}^{[l+1]}$. Then from (7)

$$z_k^{[l+1]} = \mathbf{w}_k^{[l+1]}\sigma(\mathbf{z}^{[l]}) + b_k^{[l+1]} = \sum_{t=1}^{n_l} w_{kt}^{[l+1]}\sigma(z_t^{[l]}) + b_k^{[l+1]}$$

$$\frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = w_{kj}^{[l+1]}\sigma'(z_j^{[l]})$$

$$\delta_j^{[l]} = \sum_{k=1}^{n_{l+1}}\delta_k^{[l+1]}w_{kj}^{[l+1]}\sigma'(z_j^{[l]}) = \sigma'(z_j^{[l]})\sum_{k=1}^{n_{l+1}}\delta_k^{[l+1]}w_{kj}^{[l+1]} = \sigma'(z_j^{[l]})\big((\mathbf{W}^{[l+1]})^T\boldsymbol{\delta}^{[l+1]}\big)_j$$

where the last term represents the j-th component of product. And put above into a vector we can get (24).

•From (7) we already know

$$z_j^{[l]} = \big(\mathbf{W}^{[l]}\mathbf{a}^{[l]}\big)_j + \mathbf{b}_j^{[l]} = \sum_{t=1}^{n_{l-1}} w_{jt}^{[l]}\mathbf{a}_t^{[l-1]} + \mathbf{b}_j^{[l]}$$

Since $\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = 1$ and $\frac{\partial z_j^{[l]}}{\partial W_{jk}^{[l]}} = a_k^{[l-1]}$. Through chain rule,

$$\frac{\partial C_{\mathbf{x}^{\{i\}}}}{\partial b_j^{[l]}} = \frac{\partial C_{\mathbf{x}^{\{i\}}}}{\partial z_j^{[l]}}\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = \delta_j^{[l]}$$

$$\frac{\partial C_{\mathbf{x}^{\{i\}}}}{\partial W_{jk}^{[l]}} = \frac{\partial C_{\mathbf{x}^{\{i\}}}}{\partial z_j^{[l]}}\frac{\partial z_j^{[l]}}{\partial W_{jk}^{[l]}} = \delta_j^{[l]}a_k^{[l-1]}$$

Here we get (25) and (26). Hence the proof is completed.

2.3 An overview of of the network

Based on algorithms and methodologies in previous sections, it is now clear that how a neural network trains.

- Initialise weights and biases, choose a learning rate η
- Choose a random data point for implementing Stochastic gradient.

- Evaluate the output $\mathbf{a}^{[l]}$ by using *feed-forward* algorithm.
- Apply *back-propagation* to evaluate errors $\boldsymbol{\delta}^{[l]}$. This makes use of $\mathbf{a}^{[l]}$ and the label $\mathbf{y}$
- Apply Stochastic gradient to calculate $\frac{C_{x\{i\}}}{\mathbf{W}}$ and $\frac{C_{x\{i\}}}{\mathbf{b}}$ by using errors from the last step.
- Check whether the network is trained: by calculating the total cost(if it is below some threshold). This step should be implemented every 1000 iterations (flexible), since evaluation of total cost is computationally expensive when the dataset is large.
- If after a finite number of iterations, the total cost has been reduced below threshold, the training is therefore successful. Otherwise it fails, then we should adjust learning rates or initial weights&biases and start again.

★ Now let us look at the scope of the whole neural network. If we are using a Sigmoid function like tanh(z), then the model is nonlinear. Then what we finally want to produce is actually a target function $\mathbf{F(x)}$, we train the network to approximate this function. In this project we aim to approximate a function where $\mathbf{F(x)}$=0 divides two types of data. We already know from the Approximation Theory that not all functions can be approximated. In this regard, [Cybenko, 1989] had proved that a one-hidden-layer neural network can approximate any continuous function with nonlinear sigmoid activations, to achieve arbitrary level of accuracy. And [Hornik, 1991] broadened Cybenko's results to more classes of nonlinear functions, including ReLU.

3 Training and using the neural network

Here I jumped straight to the final part, assuming all codes are correct. Because the **Test()** function is in fact a process of validation and it is self-contained, I will move it to **section(4)**.

3.1 Train and use the network to classify simple data sets

As described in the ClassifyTestData function. Here I use a {2-3-3-1} neural network to train a 16-pair data set. The learning rate η=0.1 and the threshold is 10^{-4}. Initial parameters $\mathbf{p} \sim N(0, 0.1)$. It takes at average 50000 iterations to successfully converge. The plot is as below, We can observe

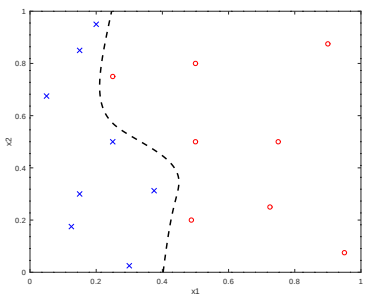


Figure 1: Classification of a simple data set

that the contour divides blue cross and red circles clearly. Hence the training is sufficient to achieve the task of classifying 16 data points.

3.2 How to choose learning rate

Recall that during the updating process of parameters,

$$\mathbf{p} \leftarrow \mathbf{p} + \Delta\mathbf{p} = \mathbf{p} - \eta \nabla C_{\mathbf{x}^{\{i\}}}(\mathbf{p})$$

The gradient decides the direction whereas η decides the size of each step(iteration). Learning rate is an extremely important factor in the neural network and it is still under researching how to choose perfect learning rates.

Here I will try a few various learning rates between 0.001 and 1 to observe how it will impact the convergence of training. In the Figure(2), the network with learning rate η=0.001 takes the

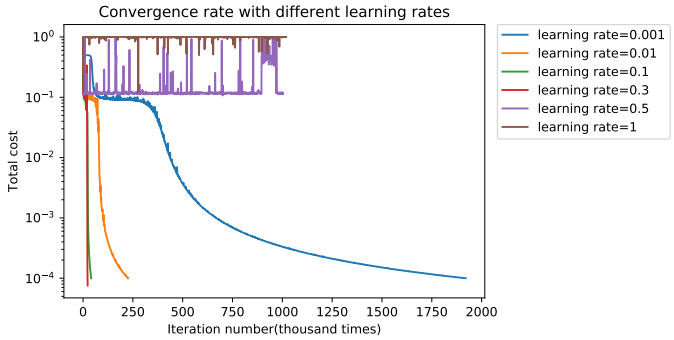
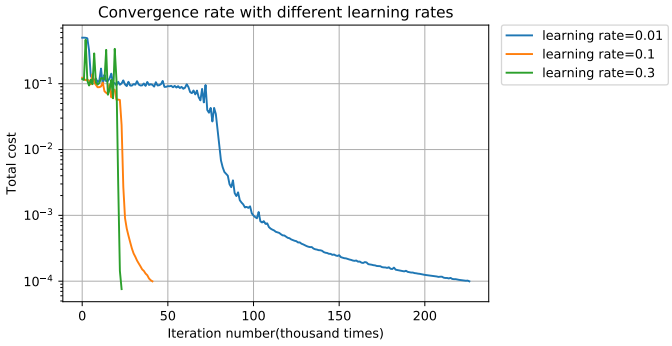
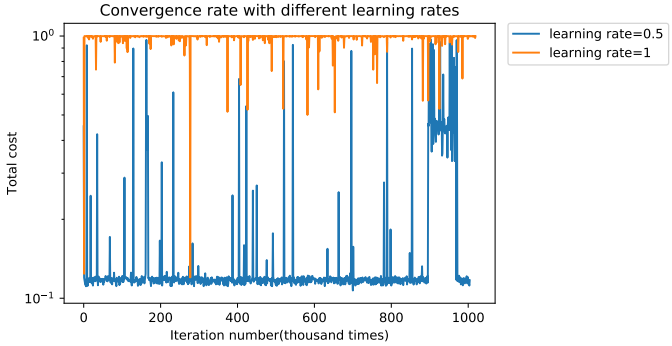


Figure 2: Comparison among different learning rates



(a) Comparison among different learning rates



(b) Comparison among different learning rates

most iterations to converge, while η=0.01,0.1,0.3 takes fewer than 25000 times(It can be more or less) to converge. The network with η=0.5,1 do not converge, the total cost cannot even reach below 10^{-1}. And the Figure (a) shows η=0.1 and η=0.3 are efficient than η=0.01, even if all three can make the network converge after a number of iterations.

To interpret the influence of the learning rate, it is better to start from local minimas and global minima. As we apply stochastic gradient method to update parameters, we want to optimise the parameters to make to total cost reach the global minima(0 or some threshold). However, the optimisation is not a convex problem and it includes many local minimas as described in the Figure (4). **Thus, if our learning rate is set too low, not only the convergence will become very slow (as in previous figures), it may also get stuck in the local minima**. Like a ball rolling up a bit from the local minima then rolls back again. **On the contrary, if learning rate is very large, then every update of parameters is huge, it may then overshoot all minimas including global minima**. For example, the ball rolls very hard to the left peak in the Figure (4), then rolls back with the same magnitude. In this case it will not even reach some good local minimas, let alone convergence(As we see in Figure (b)).

★ Usually learning rate is not fixed at a constant level in neural networks. Because as the training

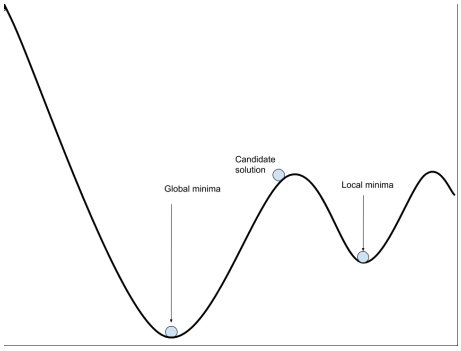


Figure 4: Local minimas and global minima

proceeds, the size of each update should vary due to the optimisation features of non-convex surface. For example, we might want our learning rate to be higher at the beginning and gradually decline at the training approaches toward the global minima.

In [Zulkifli, 2018] it describes three common learning rate schedules are introduced: **Time-based Decay**, **Step-based Decay** and **Exponential Decay**. Further more, learning rates can also be adaptive to gradient descent algorithms, it also gives some brief about them such as **Adagrad**.

3.3 Initialisation of Weights and Biases

When setting up the neural network, we usually first initialise the weights and biases. As what we do in this project, all initial weights and biases follow a normal distribution with mean 0 and standard deviation 0.1. Recall the process of updating weights and biases:

$$w_j \to w_j - \eta \frac{\partial C_{x\{i\}}}{\partial w_j}$$

$$b_j \to b_j - \eta \frac{\partial C_{x\{i\}}}{\partial b_j}$$

And we know that above two partial derivatives both depend on

$$\frac{\partial C_{x\{i\}}}{\partial z_j} \sim \sigma^{'}(z_j) f(x,y), \quad f(x,y) \; represents \; some \; linear \; function$$

Then the main factor here is $\sigma^{'}(z_j)$. Since we are using **tanh** activation, if we look at its graph, We can find its slope goes near 0 when z is positively or negatively large.

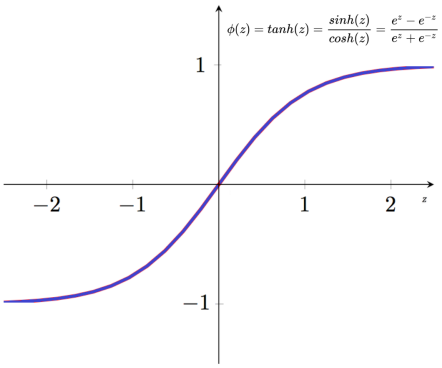


Figure 5: Hyperbolic tangent

And we know that

$$z_j = \sum_k w_k x_k + b_j, \quad w_k, b_k \sim N(0, 0.01)$$

Since all initial parameters in our network are normally distributed. The linear sum up of these parameters also follow a normal distribution. For example, if we let $x_k = 1,\ for\ k = 1, \ldots, n^{[1]}$. Then we have

$$z_j \sim N\big(0, 0.01(1+k)\big)$$

In such a form it is very clear that convergence rate can be heavily impacted by initial normally distributed parameters.

From the Figure(5), large values of z_j will lead to small gradients, so we always want our z_j not to be far from 0. If we initialise parameters with very large standard deviation, according to the Confidence Interval theory, **more values of inputs will stay far away from the centre 0**, since the mean decides the location of the centre. **Thus, the derivatives may be very small as well as each update**. **So it will hinder the convergence rate in such a way**. **Also, very small initial parameters are not desirable**, since they may distinguish our target parameters a lot and it will take a long time to train.

★The philosophy of initialising weights and biases is still under researching. In this project I will apply a better initialisation method to avoid saturation inputs (See ...). Initialise weights with

$$w_k \sim N(0, 1/n^{[l-1]}) \quad l = 2, \ldots, L$$

$$b_k \sim N(0, 1)$$

Under this situation, we have

$$z_j \sim N(0, t) \ \ t \in [1, 2] \quad for\ properly\ valued\ x_k$$

Then it is less likely that nodes at each layer will saturate, inputs z_j can stay at a '**safe**' level where update is proceeding faster than previous cases.

	$N(0, 0.01)$	$N(0, 1/n^{[l-1]})$
Iterations	Around 50000	Around 30000

There are many more approaches to better initialise weights and biases, differentiated by tasks. **For example**, [cao et al, 2017] describes a way in which the weights between the hidden layer and input layer are randomly selected and the weights between the output layer and hidden layer are obtained analytically. Researchers have shown that NNRW has much lower training complexity in comparison with the traditional training of feed-forward neural networks.

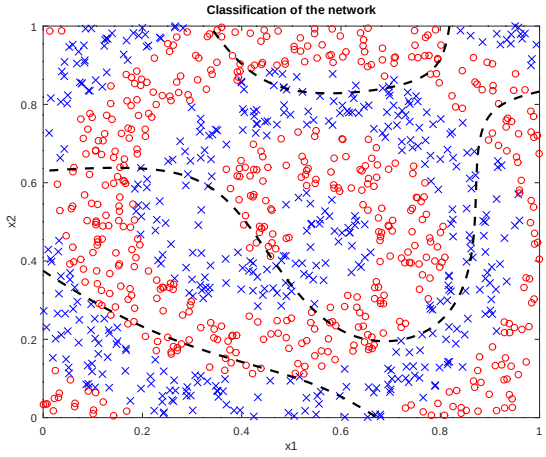
3.4 Classification of more challenging data sets

Instead of training on a simple dataset, now we proceed to a set with 1000 pairs of data from GetSpiralData[4]. I will modify the network in terms of the number of neurons/layers. For other configurations in the network, initial parameters are set as described in the last section, and learning rate falls between 0.005 to 0.1 for different levels of complexity of the network. I will start from 3-layer networks since **too simple networks are not suitable for large data problems**.

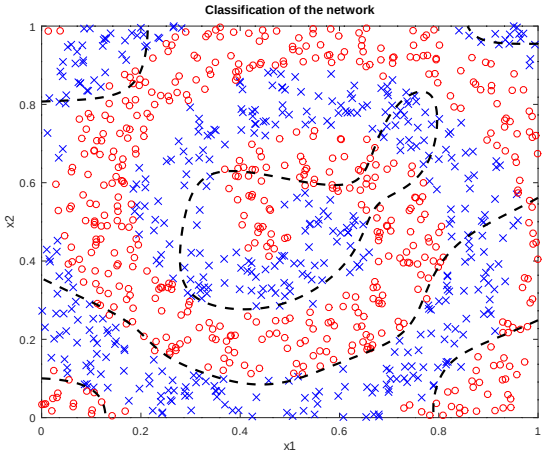
- **Network 6.(a)**: Fails to converge after 2000000 iterations and the final total cost is 0.37. Behaves poorly on classifying the data. (η=0.1)
- **Network 6.(b)**: Fails to converge after 2000000 iterations, with final total cost at 0.26. Performs better than (a) but is not satisfied. (η=0.1)

Now observe the plots below,

[4]CheckerboardData plots will be provided in Appendix A



(a) Classify data with a 2-20-1 neural network

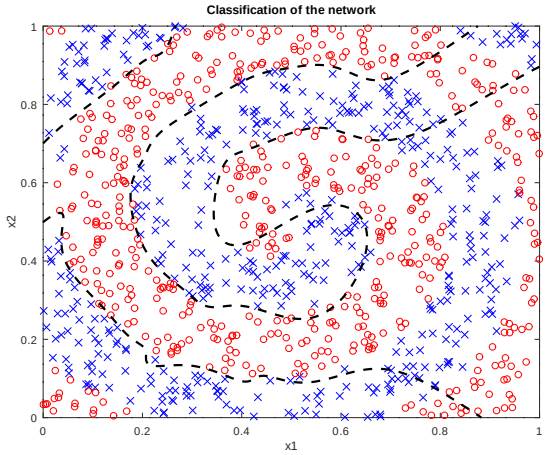


(b) Classify data with a 2-50-1 neural network

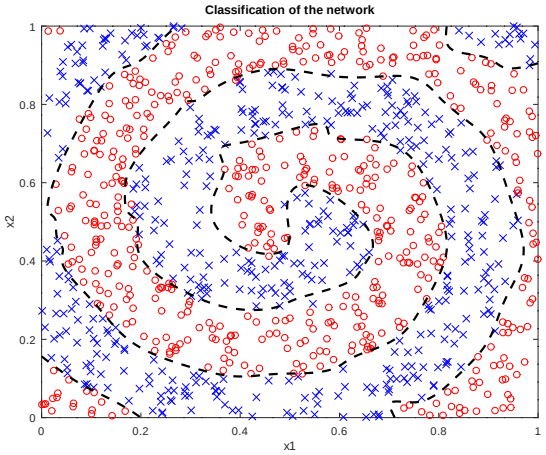
Figure 6: Neural networks with 1 hidden layer

- **Network 7.(a)**: Total cost fails to reach below 0.01 and kept at around 0.17 after 2 million iterations. (η=0.01)
- **Network 7.(b)**: Total cost hits and get stuck at around 0.10. This trained network behaves well enough to classify the data. (η=0.01)

- **Network 8.(a)**: Total cost fails can converge to a rather low level such as 0.05 after around 2500000 iterations. Thus the contour it draws divide the data into two classes very well. (η=0.015)
- **Network 8.(b)**: The only successful network (for threshold 0.01). It takes about 3200000 iterations to hit the required threshold. This model behaves almost perfectly to achieve the classification task. (η=0.01)



(a) Classify data with a 2-5-5-1 neural network



(b) Classify data with a 2-10-10-1 neural network

Figure 7: Neural networks with 2 hidden layers

I also include a plot of convergence of Network 8.(b) in the appendix. In Figure 10(a)[5]. We can see that it is generally difficult to reduce the total cost below 0.01, even if it does, millions of iterations are needed. And there countless fluctuations during the training process, meaning a lot of local minimas appear. On the other hand, it seems easier to train and classify CheckerboardData using the identical neural network. It takes less time and there are also fewer local minimas during the training.

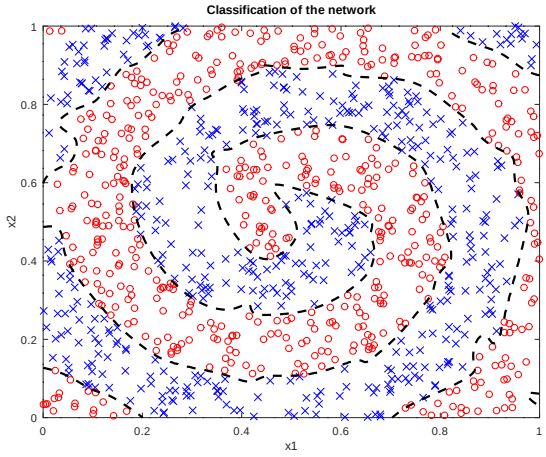
Normally, we need larger networks to solve some complicated problems. But there are still some drawbacks in applying large networks:

• **Computation consuming**: It takes a large amount of time to train a network with lots of neurons&layers.

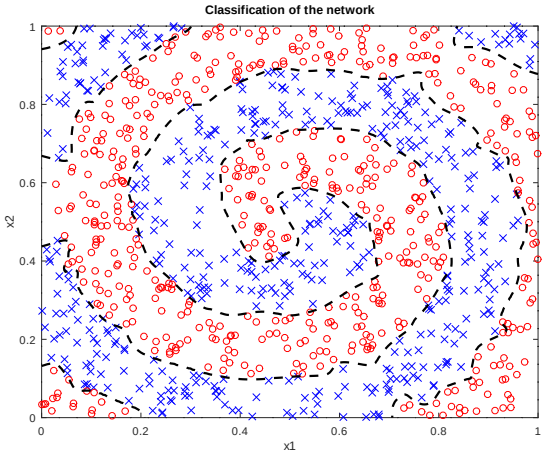
• **Over-fitting**: If we use a rather complicated network to train some simple data, it will over-fits, meaning it may perform poorly on other unseen data.

★ **Here are some extra discussions about benefits of deep neural networks**. [Telgarsky, 2015] applied a deep network show that there exist functions where deep networks are able to exactly classify the data using a finite number of parameters, while it requires exponentially many parameters in a shallow network.

[5]See Appendix A



(a) Classify data with a 2-15-15-1 neural network



(b) Classify data with a 2-20-20-10-1 neural network

Figure 8: Good networks

4 Validation of the code

• To validate $\sigma(z) = tanhz = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ and $\sigma^{'}(z) = \frac{4}{(e^z + e^{-z})^2}$

```
z=0------>sigma(z)=0; sigmaPrime(z)=1.
z=log(2)------>sigma(z)=0.6; sigmaPrime(z)=1.
```

• To validate that initial weights and biases follow a normal distribution as required, **we can look at if all values fall within the 99% confidence interval**. It is not perfect but can check well enough. Here I will use a standard normal distribution, whose 99% CI is $(-2.576, 2.576)$ for every single observation. The output of C^{++} code:

```
Network n1({2,3,3,1 });//set a casual class for testing.
Initial weights and biases for n1:
W[1](    1.8279,   0.133723)
( -0.228932,    1.16543)
(    1.0715,   0.411306)
b[1](-0.0201053, -0.00991875, -0.041893)

W[2]( -0.612341,   0.620902,   0.737878)
( -0.654849,   0.201588,0.00213858)
( -0.471696,   0.967633,   0.530132)
b[2](0.174805, -0.758377, 0.418114)

W[3](  -1.67384,    0.43052,    1.50144)
```

```
b[3](-1.17382)
```

It is obvious that the size of weights matrix and biases vector are correct. And all the values are inside the CI interval.
• To test the **FeedForward** function. Use the example set up: n({ 2, 1 }). **W**=(-0.3,0.2), **b**=(0.5). **x**=({0.3,0.4}) We have

```
tanh(0.5 + (-0.3 * 0.3 + 0.2 * 0.4)) =0.454216432682259.
if (std::abs(n.activations[1][0] - 0.454216432682259) > 1e-10)
{
return false;
}
```

• To validate the **BackPropagateError** function. Use the same three-neuron network, with desired output **y**=(1). We have
$\sigma^{'}(z^{[1]})(a^{[1]} - y) = -0.433181558125802$.

```
if (abs(n.errors[1][0] - (-0.433181558125802)) > 1e-10) { return false; }
```

• To check **Cost** function. Use the same example network with desired output **y**=1. The analytical value of a single cost is,
$C = \frac{1}{2}(y - a^{[1]}) = 0.14893985117704$.

```
if ( abs(n.Cost({ 1 })-0.14893985117704)>1e-10 ) {return false;}
```

•To test the **TotalCost** function. Use the same network, with input data set $\mathbf{x} = \{\{0.3, 0.4\}, \{0.5, 0.6\}, \{0.7, 0.8\}\}$ and output $\mathbf{y} = \{\{1\}, \{-1\}, \{1\}\}$. The output of C^{++} as follow,

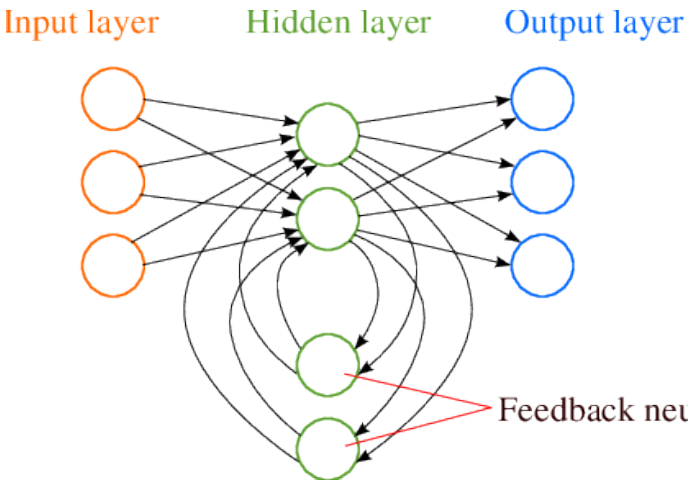
```
Test for total cost function: //here we make use of verified Cost function.
C1:0.14893985117704
C2:1.03420863459272
C3:0.167100380065465
s=C1+C2+C3
if (abs( n.TotalCost(x, y) - s / 3.0) > 1e-10) { return false; }
```

• To validate the **UpdateWeightsAndBiases** function. Use the same network, with **x**=({0.3,0.4}), **y**=({1.0}), η=0.5. Through hands calculations,

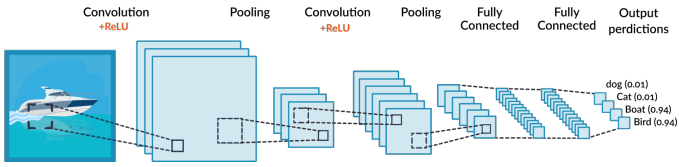
```
The updated weights shoule be (-0.2350227662811297, 0.2866363116251604)
The updated bias should be (0.716590779062901)
if (abs(n.weights[1](0, 0) - (-0.2350227662811297)) > 1e-10
|| abs(n.weights[1](0, 1) - 0.2866363116251604) > 1e-10
|| abs(n.biases[1][0] - 0.716590779062901) > 1e-10)
{
return false;
}
```

My **Network::Test()** function includes all private member functions inside the Network class. Above now completes the validation of them, and finally **Network::Test()** should return 1 if successful. Note that the validation of the **Train()** function depends on the correctness of other private member functions. So I do not validate here again.

Appendix A Some plots or figures

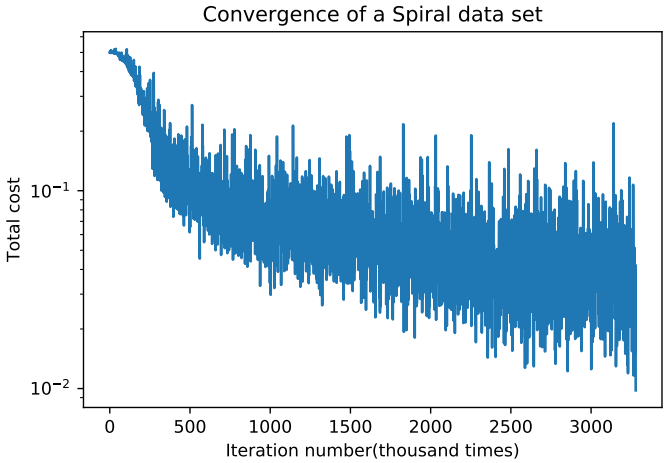


(a) An overview of RNN

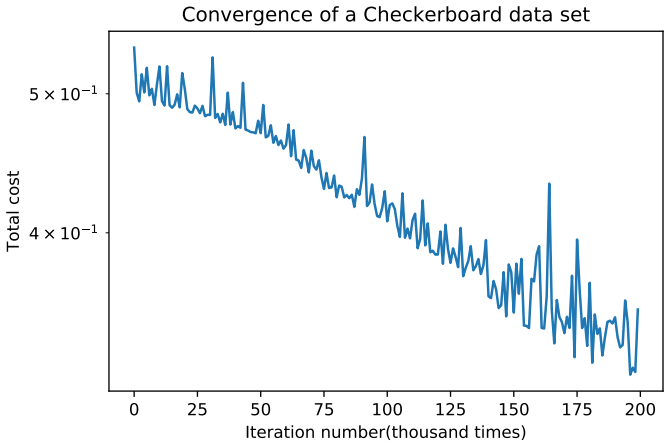


(b) An overview of CNN

Figure 9: From [Missinglink.AI] and [Galetzka, 2014]



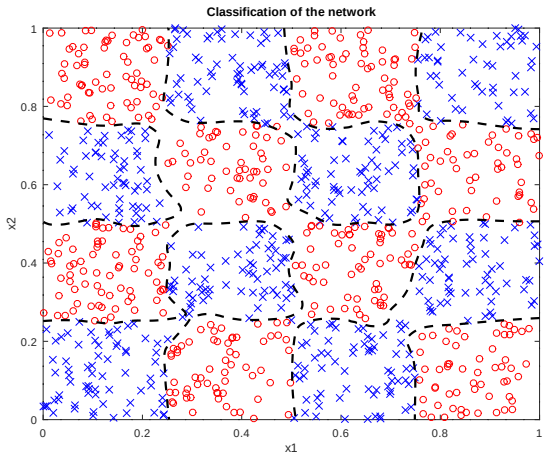
(a) A 2-20-20-10-1 network for SpiralData



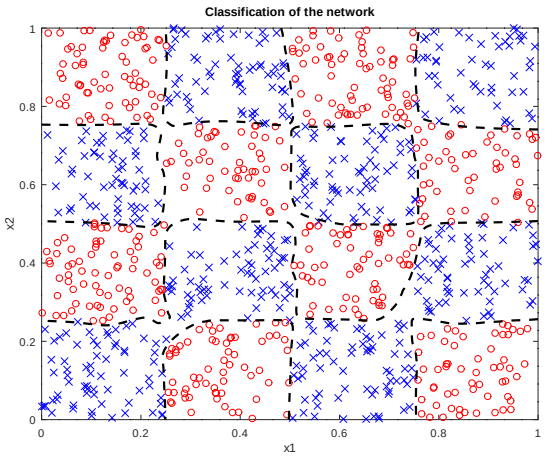
(b) A 2-20-20-10-1 network for CheckerboardData

Figure 10: Convergence of two successful networks

-



(a) A 2-20-20-1 network for classifying CheckerboardData



(b) A 2-20-20-10-1 network for classifying CheckerboardData

Figure 11: Classifications of CheckerboardData

Appendix B Header file code

B.1 VecMat.h

```
#pragma once
#include <vector>
#include<iostream>
#include<ostream>
using namespace std;
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)
```

```
22      double operator[](int index) const {
23          return v[index];
24      }
25
26      int size() const { return v.size(); } // number of elements
27
28  private:
29      std::vector<double> v;
30  };
31
32  class MMatrix
33  {
34  public:
35      // constructors
36      MMatrix() : nRows(0), nCols(0) {} //default constructor.
37      MMatrix(int n, int m, double x = 0) : nRows(n), nCols(m), A(n* m, x) {}
38
39      // set all matrix entries equal to a double
40      MMatrix& operator=(double x)
41      {
42          for (unsigned i = 0; i < nRows * nCols; i++) A[i] = x;
43          return *this;
44      }
45
46      // access element, indexed by (row, column) [rvalue]
47      double operator()(int i, int j) const
48      {
49          return A[ j + i * nCols];
50      }
51
52      // access element, indexed by (row, column) [lvalue]
53      double& operator()(int i, int j)
54      {
55          return A[j + i * nCols];
56      }
57
58      // size of matrix
59      int Rows() const { return nRows; }
60      int Cols() const { return nCols; }
61      int size() const { return A.size(); }
62  private:
63      unsigned int nRows, nCols;
64      std::vector<double> A;
65  };
```

Appendix C Implementation

C.1 neural.cpp

```
1   #include"VecMat.h"
2   #include <cmath>
3   #include <random>
4   #include <iostream>
5   #include <fstream>
6   #include <cassert>
7   #include <typeinfo>
8   using namespace std;
9   //
        ////////////////////////////////////////////////////////////////////////////
10  // Set up random number generation
```

```
11
12 // Set up a "random device" that generates a new random number each time
       the program is run
13 std::random_device rand_dev;
14
15 // Set up a pseudo-random number generater "rnd", seeded with a random
       number
16 std::mt19937 rnd(rand_dev());
17
18 // Alternative: set up the generator with an arbitrary constant integer.
       This can be useful for
19 // debugging because the program produces the same sequence of random
       numbers each time it runs.
20 // To get this behaviour, uncomment the line below and comment the
       declaration of "rnd" above.
21 //std::mt19937 rnd(12345);
22
23
24 //
       ////////////////////////////////////////////////////////////////////////////

25 // Some operator overloads to allow arithmetic with MMatrix and MVector.
26 // These may be useful in helping write the equations for the neural
       network in
27 // vector form without having to loop over components manually.
28 //
29 // You may not need to use all of these; conversely, you may wish to add
       some
30 // more overloads.
31
32 // MMatrix * MVector
33 MVector operator*(const MMatrix& m, const MVector& v)
34 {
35     assert(m.Cols() == v.size());//if the condition is false the program is
        terminated.
36
37     MVector r(m.Rows());
38
39     for (int i = 0; i < m.Rows(); i++)
40     {
41         for (int j = 0; j < m.Cols(); j++)
42         {
43             r[i] += m(i, j) * v[j];
44         }
45     }
46     return r;
47 }
48
49 // transpose(MMatrix) * MVector
50 MVector TransposeTimes(const MMatrix& m, const MVector& v)
51 {
52     assert(m.Rows() == v.size());
53
54     MVector r(m.Cols());
55
56     for (int i = 0; i < m.Cols(); i++)
57     {
58         for (int j = 0; j < m.Rows(); j++)
59         {
60             r[i] += m(j, i) * v[j];
61         }
```

```
62      }
63      return r;
64  }
65
66  // MVector + MVector
67  MVector operator+(const MVector& lhs, const MVector& rhs)
68  {
69      assert(lhs.size() == rhs.size());
70
71      MVector r(lhs);
72      for (int i = 0; i < lhs.size(); i++)
73          r[i] += rhs[i];
74
75      return r;
76  }
77
78  // MVector - MVector
79  MVector operator-(const MVector& lhs, const MVector& rhs)
80  {
81      assert(lhs.size() == rhs.size());
82
83      MVector r(lhs);
84      for (int i = 0; i < lhs.size(); i++)
85          r[i] -= rhs[i];
86
87      return r;
88  }
89
90  // MMatrix = MVector <outer product> MVector
91  // M = a <outer product> b
92  MMatrix OuterProduct(const MVector& a, const MVector& b)
93  {
94      MMatrix m(a.size(), b.size());
95      for (int i = 0; i < a.size(); i++)
96      {
97          for (int j = 0; j < b.size(); j++)
98          {
99              m(i, j) = a[i] * b[j];
100         }
101     }
102     return m;
103 }
104
105 // Hadamard product
106 MVector operator*(const MVector& a, const MVector& b)
107 {
108     assert(a.size() == b.size());
109
110     MVector r(a.size());
111     for (int i = 0; i < a.size(); i++)
112         r[i] = a[i] * b[i]; //a1*b1+a2*b2...
113     return r;
114 }
115
116 // double * MMatrix
117 MMatrix operator*(double d, const MMatrix& m)
118 {
119     MMatrix r(m);
120     for (int i = 0; i < m.Rows(); i++)
121         for (int j = 0; j < m.Cols(); j++)
122             r(i, j) *= d;
```

```
123
124     return r;
125 }
126
127 // double * MVector
128 MVector operator*(double d, const MVector& v)
129 {
130     MVector r(v);
131     for (int i = 0; i < v.size(); i++)
132         r[i] *= d;
133
134     return r;
135 }
136
137 // MVector -= MVector
138 MVector operator-=(MVector& v1, const MVector& v)
139 {
140     assert(v1.size() == v.size());
141
142     MVector r(v1);
143     for (int i = 0; i < v1.size(); i++)
144         v1[i] -= v[i];
145
146     return r;
147 }
148
149 // MMatrix -= MMatrix
150 MMatrix operator-=(MMatrix& m1, const MMatrix& m2)
151 {
152     assert(m1.Rows() == m2.Rows() && m1.Cols() == m2.Cols());
153
154     for (int i = 0; i < m1.Rows(); i++)
155         for (int j = 0; j < m1.Cols(); j++)
156             m1(i, j) -= m2(i, j);
157
158     return m1;
159 }
160
161 // Output function for MVector
162 inline std::ostream& operator<<(std::ostream& os, const MVector& rhs)
163 {
164     std::size_t n = rhs.size();
165     os << "(";
166     for (std::size_t i = 0; i < n; i++)
167     {
168         os << rhs[i];
169         if (i != (n - 1)) os << ", ";
170     }
171     os << ")";
172     return os;
173 }
174
175 // Output function for MMatrix
176 inline std::ostream& operator<<(std::ostream& os, const MMatrix& a)
177 {
178     int c = a.Cols(), r = a.Rows();
179     for (int i = 0; i < r; i++)
180     {
181         os << "(";
182         for (int j = 0; j < c; j++)
183         {
```

```
184             os.width(10);
185             os << a(i, j);
186             os << ((j == c - 1) ? ')' : ',');
187         }
188         os << "\n";
189     }
190     return os;
191 }
192
193 //
    ////////////////////////////////////////////////////////////////////////////
194 // Functions that provide sets of training data
195
196 // Generate 16 points of training data in the pattern illustrated in the
    project description
197 void GetTestData(std::vector<MVector>& x, std::vector<MVector>& y)
198 {
199     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},
200          {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} };
201
202     y = { {1},{1},{1},{1},{1},{1},{1},{1},
203          {-1},{-1},{-1},{-1},{-1},{-1},{-1},{-1} };
204 }
205
206 // Generate 1000 points of test data in a checkerboard pattern
207 void GetCheckerboardData(std::vector<MVector>& x, std::vector<MVector>& y)
208 {
209     std::mt19937 lr;
210     x = std::vector<MVector>(1000, MVector(2));
211     y = std::vector<MVector>(1000, MVector(1));
212
213     for (int i = 0; i < 1000; i++)
214     {
215         x[i] = { lr() / static_cast<double>(lr.max()),lr() / static_cast<
    double>(lr.max()) };
216         double r = sin(x[i][0] * 12.5) * sin(x[i][1] * 12.5);
217         y[i][0] = (r > 0) ? 1 : -1;//if (r>0) y=1 else y=-1.
218     }
219 }
220
221
222 // Generate 1000 points of test data in a spiral pattern
223 void GetSpiralData(std::vector<MVector>& x, std::vector<MVector>& y)
224 {
225     std::mt19937 lr; //generate a seudo-random number.
226     x = std::vector<MVector>(1000, MVector(2));
227     y = std::vector<MVector>(1000, MVector(1));
228
229     double twopi = 8.0 * atan(1.0);
230     for (int i = 0; i < 1000; i++)
231     {
232         x[i] = { lr() / static_cast<double>(lr.max()),lr() / static_cast<
    double>(lr.max()) };
233         double xv = x[i][0] - 0.5, yv = x[i][1] - 0.5;
234         double ang = atan2(yv, xv) + twopi; //atan2(y,x)=acrtangent(y/x)
235         double rad = sqrt(xv * xv + yv * yv);//radius
236
237         double r = fmod(ang + rad * 20, twopi); //take the reminder.
```

```
238             y[i][0] = (r < 0.5 * twopi) ? 1 : -1;
239         }
240     }
241
242     // Save the the training data in x and y to a new file, with the filename
        given by "filename"
243     // Returns true if the file was saved succesfully
244     bool ExportTrainingData(const std::vector<MVector>& x, const std::vector<
        MVector>& y,
245         std::string filename)
246     {
247         // Check that the training vectors are the same size
248         assert(x.size() == y.size());
249
250         // Open a file with the specified name.
251         std::ofstream f(filename);
252
253         // Return false, indicating failure, if file did not open
254         if (!f)
255         {
256             return false;
257         }
258
259         // Loop over each training datum
260         for (unsigned i = 0; i < x.size(); i++)
261         {
262             // Check that the output for this point is a scalar
263             assert(y[i].size() == 1);
264
265             // Output components of x[i]
266             for (int j = 0; j < x[i].size(); j++)
267             {
268                 f << x[i][j] << " ";
269             }
270
271             // Output the only component of y[i]
272             f << y[i][0] << " " << std::endl;
273         }
274         f.close();
275
276         if (f) return true;
277         else return false;
278     }
279
280
281
282
283     //
        ////////////////////////////////////////////////////////////////////////////////
284     // Neural network class
285
286     class Network
287     {
288     public:
289
290         // Constructor: sets up vectors of MVectors and MMatrices for
291         // weights, biases, weighted inputs, activations and errors
292         // The parameter nneurons_ is a vector defining the number of neurons
            at each layer.
293         // For example:
```

```
294     //   Network({2,1}) has two input neurons, no hidden layers, one output
        neuron
295     //
296     //   Network({2,3,3,1}) has two input neurons, two hidden layers of
        three neurons
297     //                      each, and one output neuron
298     Network(std::vector<unsigned> nneurons_)
299     {
300         nneurons = nneurons_;
301         nLayers = nneurons.size();
302         weights = std::vector<MMatrix>(nLayers);
303         biases = std::vector<MVector>(nLayers);
304         errors = std::vector<MVector>(nLayers);
305         activations = std::vector<MVector>(nLayers);
306         inputs = std::vector<MVector>(nLayers);
307         // Create activations vector for input layer 0
308         activations[0] = MVector(nneurons[0]);
309
310         // Other vectors initialised for second and subsequent layers
311         for (unsigned i = 1; i < nLayers; i++)
312         {
313             weights[i] = MMatrix(nneurons[i], nneurons[i - 1]);
314             biases[i] = MVector(nneurons[i]);
315             inputs[i] = MVector(nneurons[i]);
316             errors[i] = MVector(nneurons[i]);
317             activations[i] = MVector(nneurons[i]);
318         }
319
320         // The correspondence between these member variables and
321         // the LaTeX notation used in the project description is:
322         //
323         // C++                         LaTeX
324         // ------------------------------------
325         // inputs[l-1][j-1]      =  z_j^{[l]}
326         // activations[l-1][j-1] =  a_j^{[l]}
327         // weights[l-1](j-1,k-1) =  W_{jk}^{[l]}
328         // biases[l-1][j-1]      =  b_j^{[l]}
329         // errors[l-1][j-1]      =  \delta_j^{[l]}
330         // nneurons[l-1]         =  n_l
331         // nLayers               =  L
332         //
333         // Note that, since C++ vector indices run from 0 to N-1, all the
        indices in C++
334         // code are one less than the indices used in the mathematics (
        which run from 1 to N)
335     }
336
337     // Return the number of input neurons
338     unsigned NInputNeurons() const
339     {
340         return nneurons[0];
341     }
342
343     // Return the number of output neurons
344     unsigned NOutputNeurons() const
345     {
346         return nneurons[nLayers - 1];
347     }
348     // Evaluate the network for an input x and return the activations of
        the output layer
349     MVector Evaluate(const MVector& x)
```

```
350     {
351         // Call FeedForward(x) to evaluate the network for an input vector
        x
352         FeedForward(x);
353 
354         // Return the activations of the output layer
355         return activations[nLayers - 1];
356     }
357 
358 
359     // Implement the training algorithm outlined in section 1.3.3
360     // This should be implemented by calling the appropriate private member
         functions, below
361     bool Train(const std::vector<MVector> x, const std::vector<MVector> y,
362         double initsd, double learningRate, double costThreshold, int
        maxIterations)
363     {
364         // Check that there are the same number of training data inputs as
        outputs
365         assert(x.size() == y.size());
366         ofstream fl("costdata.txt"); //write the cost into a file.
367         //InitialiseWeightsAndBiases(initsd);
368         InitialiseWeightsAndBiases1(initsd);
369         // TODO: Step 2 - initialise the weights and biases with the
        standard deviation "initsd"
370         cout << TotalCost(x, y) << endl;
371         for (int iter = 1; iter <= maxIterations; iter++)
372         {
373             // Step 3: Choose a random training data point i in {0, 1, 2,
        ..., N}
374             int i = rnd() % x.size();
375             FeedForward(x[i]);
376             // TODO: Step 4 - run the feed-forward algorithm
377             BackPropagateError(y[i]);
378             // TODO: Step 5 - run the back-propagation algorithm
379             //if (TC >= 1e-3) { UpdateWeightsAndBiases(0.1);}
380             //if (TC > 1e-3 / 8.0&&TC<1e-3) { UpdateWeightsAndBiases(0.05);
         }
381             //if (TC > 1e-4 &&TC< 1e-3 / 8.0) { UpdateWeightsAndBiases
        (0.01); }
382             UpdateWeightsAndBiases(learningRate);
383             //UpdateWeightsAndBiases1(learningRate,iter);
384             // TODO: Step 6 - update the weights and biases using
        stochastic gradient
385             //                 with learning rate "learningRate"
386 
387             // Every so often, perform step 7 and show an update on how the
         cost function has decreased
388             // Here, "every so often" means once every 1000 iterations, and
         also at the last iteration
389             if ((!(iter % 1000)) || iter == maxIterations)
390             {
391                 // TODO: Step 7(a) - calculate the total cost
392                 double TC=TotalCost(x, y);
393                 fl << TC << endl;//write every 1000-th cost into the file.
394                 // TODO: display the iteration number and total cost to the
         screen
395                 cout << "This is the " << iter << "-th iteration" << endl;
396                 cout << "The total cost is : " << TC << endl;
397                 // TODO: Step 7(b) - return from this method with a value
        of true,
```

```
398                 //                    indicating success, if this cost is
    less than "costThreshold".
399                 if (TC < costThreshold) { return true; }
400             }
401
402         } // Step 8: go back to step 3, until we have taken "maxIterations"
     steps
403         fl.close();
404         // Step 9: return "false", indicating that the training did not
    succeed.
405         return false;
406     }
407
408
409     // For a neural network with two inputs x=(x1, x2) and one output y,
410     // loop over (x1, x2) for a grid of points in [0, 1]x[0, 1]
411     // and save the value of the network output y evaluated at these points
412     // to a file. Returns true if the file was saved successfully.
413     bool ExportOutput(std::string filename) //to be used in classfify
    function.
414     {
415         // Check that the network has the right number of inputs and
    outputs
416         assert(NInputNeurons() == 2 && NOutputNeurons() == 1);
417
418         // Open a file with the specified name.
419         std::ofstream f(filename);
420
421         // Return false, indicating failure, if file did not open
422         if (!f)
423         {
424             return false;
425         }
426
427         // generate a matrix of 250x250 output data points
428         for (int i = 0; i <= 250; i++)
429         {
430             for (int j = 0; j <= 250; j++)
431             {
432                 MVector out = Evaluate({ i / 250.0, j / 250.0 });
433                 f << out[0] << " ";
434             }
435             f << endl;
436         }
437         f.close();
438
439         if (f) return true;
440         else return false;
441     }
442
443     static bool Test();
444
445 private:
446     // Return the activation function sigma
447     double Sigma(double z)
448     {
449         return ((exp(z) - exp(-z)) / (exp(z) + exp(-z)));
450         // TODO: Return sigma(z), as defined in equation (1.4)
451     }
452
453     // Return the derivative of the activation function
```

```
454     double SigmaPrime(double z)
455     {
456         return 4.0 / pow(exp(z) + exp(-z), 2);
457         // TODO: Return d/dz(sigma(z))
458     }
459     MVector Sigmav(MVector m)
460     {
461         MVector v(m.size());
462         for (int i = 0;i < m.size();i++)
463         {
464             v[i] = Sigma(m[i]);
465         }
466         return v;
467     }
468     MVector SigmaPrimev(MVector m)
469     {
470         MVector v(m.size());
471         for (int i = 0;i < m.size();i++)
472         {
473             v[i] = SigmaPrime(m[i]);
474         }
475         return v;
476     }
477
478     // Loop over all weights and biases in the network and set each
479     // term to a random number normally distributed with mean 0 and
480     // standard deviation "initsd"
481     void InitialiseWeightsAndBiases(double initsd)
482     {
483         // Make sure the standard deviation supplied is non-negative
484         assert(initsd >= 0);
485
486         // Set up a normal distribution with mean zero, standard deviation
    "initsd"
487         // Calling "dist(rnd)" returns a random number drawn from this
    distribution
488         std::normal_distribution<> dist(0, initsd);
489         for (unsigned i = 1;i < nLayers;i++)
490         {
491             for (unsigned j = 0;j < nneurons[i];j++)
492             {
493                 for (unsigned k = 0;k < nneurons[i - 1];k++)
494                 {
495                     weights[i](j, k) = dist(rnd);
496                 }
497                 biases[i][j] = dist(rnd);
498             }
499         }
500         // TODO: Loop over all components of all the weight matrices
501         //       and bias vectors at each relevant layer of the network.
502     }
503
504     void InitialiseWeightsAndBiases1(double initsd)
505     {
506             //Another approach to initialise parameters, based on the
    number of neurons
507             //in the previous layer. Aiming to squash down the standard
    deviation.
508             normal_distribution<> distb(0, initsd);
509             for (unsigned i = 1;i < nLayers;i++)
510             {
```

```
511                 for (unsigned j = 0;j < nneurons[i];j++)
512                 {
513                     normal_distribution<> dist(0, 1.0/sqrt(nneurons[i - 1])
    );
514                     for (unsigned k = 0;k < nneurons[i - 1];k++)
515                     {
516                         weights[i](j, k) = dist(rnd);
517                     }
518                     biases[i][j] = distb(rnd);
519                 }
520             }
521     }
522
523     // Evaluate the feed-forward algorithm, setting weighted inputs and
    activations
524     // at each layer, given an input vector x
525     void FeedForward(const MVector& x)
526     {
527         // Check that the input vector has the same number of elements as
    the input layer
528         assert(x.size() == nneurons[0]);
529         inputs[0]=activations[0] = x;
530         for (unsigned i = 1;i < nLayers;i++)
531         {
532             inputs[i] = weights[i] * activations[i - 1] + biases[i];

533             activations[i] = Sigmav(inputs[i]);
534         }
535         // TODO: Implement the feed-forward algorithm, equations (1.7),
    (1.8)
536     }
537
538     // Evaluate the back-propagation algorithm, setting errors for each
    layer
539     void BackPropagateError(const MVector& y)
540     {
541         // Check that the output vector y has the same number of elements
    as the output layer
542         assert(y.size() == nneurons[nLayers - 1]);
543         unsigned L = nLayers - 1;
544         errors[L] = SigmaPrimev(inputs[L]) * (activations[L] - y);
545         for (unsigned i = L-1;i>=1;i--)
546         {
547             errors[i] = SigmaPrimev(inputs[i]) * TransposeTimes(weights[i +
     1], errors[i + 1]);
548         }
549         // TODO: Implement the back-propagation algorithm, equations (1.22)
     and (1.24)
550     }
551
552
553     // Apply one iteration of the stochastic gradient iteration with
    learning rate eta.
554     void UpdateWeightsAndBiases(double eta)
555     {
556         // Check that the learning rate is positive
557         assert(eta > 0);
558         for (unsigned i = 1;i < nLayers;i++)
559         {
560             weights[i] -= eta * OuterProduct(errors[i], activations[i - 1])
    ;
```

```
561             biases[i] -= eta * errors[i];
562         }
563     }
564     // Return the cost function of the network with respect to a single the
        desired output y
565     // Note: call FeedForward(x) first to evaluate the network output for
        an input x,
566     //       then call this method Cost(y) with the corresponding desired
        output y
567     double Cost(const MVector& y)
568     {
569         // Check that y has the same number of elements as the network has
        outputs
570         assert(y.size() == nneurons[nLayers - 1]);
571         double initial = 0;
572         for (int i = 0;i < y.size();i++)
573         {
574             initial += pow((y - activations[nLayers - 1])[i],2);
575         }
576         return 0.5 * initial;
577         // TODO: Return the cost associated with this output
578     }
579
580     // Return the total cost C for a set of training data x and desired
        outputs y
581     double TotalCost(const std::vector<MVector> x, const std::vector<
        MVector> y)
582     {
583         // Check that there are the same number of inputs as outputs
584         assert(x.size() == y.size());
585         double a = 0;
586         for (unsigned i = 0;i < x.size();i++)
587         {
588             FeedForward(x[i]);
589             a += Cost(y[i]);
590         }
591         return a / x.size();
592         // TODO: Implement the cost function, equation (1.9), using
593         //       the FeedForward(x) and Cost(y) methods
594     }
595
596     // Private member data
597
598     std::vector<unsigned> nneurons;
599     std::vector<MMatrix> weights;
600     std::vector<MVector> biases, errors, activations, inputs;
601     unsigned nLayers;
602
603 };
604
605
606
607 bool Network::Test()
608 {
609     // This function is a static member function of the Network class:
610     // it acts like a normal stand-alone function, but has access to
        private
611     // members of the Network class. This is useful for testing, since we
        can
612     // examine and change internal class data.
613     //
```

```
614     // This function should return true if all tests pass, or false
        otherwise
615
616     // Make some example networks to test different functions.
617     Network n({ 2, 1 });
618     Network n1({2,3,3,1 });
619     //A test for Sigma and SigmaPrime function.
620     {
621         double t1 = n.Sigma(0); // set z=log(2) by hand.
622         double t2 = n.SigmaPrime(0); //set z=0 .
623         cout << "Test for Sigma (z) : " << t1 << endl;
624         cout << "Test for SigmaPrime (z) : " << t2 << endl;
625         if ((t1 - 0.6) > 1e-10 || (t2 - 1) > 1e-10) { return false; }
626     }
627
628     // An example test of FeedForward
629     {
630         //set weights and biases by hand.
631         n.biases[1][0] = 0.5;
632         n.weights[1](0, 0) = -0.3;
633         n.weights[1](0, 1) = 0.2;
634         // Call function to be tested with x = (0.3, 0.4)
635         n.FeedForward({ 0.3, 0.4 });
636         // Display the output value calculated
637         cout <<"Test for FeedForward(x): " <<n.activations[1][0] << endl;
638         cout.precision(15);cout << tanh(0.5 + (-0.3 * 0.3 + 0.2 * 0.4)) <<
        endl;
639         // Correct value is = tanh(0.5 + (-0.3*0.3 + 0.2*0.4))
640         //                  = 0.454216432682259...
641         // Fail if error in answer is greater than 10^-10:
642         if (std::abs(n.activations[1][0] - 0.454216432682259) > 1e-10)
643         {
644             return false;
645         }
646     }
647
648     // A test for InitialiseWerightAndBiases function
649     {
650         n1.InitialiseWeightsAndBiases(1);
651         cout << "Initial weights and biases for n1: " << endl;
652         for (unsigned i = 1;i < n1.nLayers;i++)
653         {
654             cout << "W[" << i << "]" << n1.weights[i];
655             cout << "b[" << i << "]" << n1.biases[i] << endl;
656             cout << endl;
657         }
658         if (n1.weights[1].size() != 6 || n1.weights[2].size() != 9 || n1.
        weights[3].size() != 3
659             || n1.biases[1].size()!=3 || n1.biases[3].size() != 1)
660         {
661             return false;// if the number of weights at each layer does not
         coincide then return a false.
662         }
663
664     }
665
666     //test for BackPropagateError function
667     {
668         //set weights and biases by hand.
669         n.biases[1][0] = 0.5;
670         n.weights[1](0, 0) = -0.3;
```

```
671         n.weights[1](0, 1) = 0.2;
672         n.FeedForward({ 0.3, 0.4 });
673         n.BackPropagateError({ 1.0 });
674         cout << "Test for BackPropagateError : " << n.errors[1][0] << endl;
675         cout.precision(15);cout << n.inputs[1][0] << endl;
676         if (abs(n.errors[1][0] - (-0.433181558125802)) > 1e-10) { return
        false; }
677         cout << endl;
678     }
679
680     //test for UpdateWeightsAndBiases function.
681     {
682         //set weights and biases by hand.
683         n.biases[1][0] = 0.5;
684         n.weights[1](0, 0) = -0.3;
685         n.weights[1](0, 1) = 0.2;
686         n.FeedForward({ 0.3, 0.4 });
687         n.BackPropagateError({ 1.0 });
688         n.UpdateWeightsAndBiases(0.5);
689         cout <<"Test for UpdateWeightsAndBiases function" << endl;
690         cout << "updated W[" << 1 << "]" << n.weights[1];
691         cout << "updated b[" << 1 << "]" << n.biases[1] << endl;
692         cout << endl;
693         //the updated weights shoule be (-0.2350227662811297,
        0.2866363116251604)
694         //updated bias should be (0.716590779062901)
695         if (abs(n.weights[1](0, 0) - (-0.2350227662811297)) > 1e-10
696             || abs(n.weights[1](0, 1) - 0.2866363116251604) > 1e-10
697             || abs(n.biases[1][0] - 0.716590779062901) > 1e-10)
698         {
699             return false;
700         }
701     }
702
703     //test for a single cost function
704     {
705         //set weights and biases by hand.
706         n.biases[1][0] = 0.5;
707         n.weights[1](0, 0) = -0.3;
708         n.weights[1](0, 1) = 0.2;
709         n.FeedForward({ 0.3, 0.4 });
710         //In this case the Cost function should return 0.14893985117704.
711         cout <<"Test for Cost function : "<< n.Cost({ 1 }) << endl;
712         if ( abs(n.Cost({ 1 })-0.14893985117704)>1e-10 ) {return false;}
713         cout << endl;
714     }
715
716     //test for total cost function.
717     {
718         //set weights and biases by hand.
719         n.biases[1][0] = 0.5;
720         n.weights[1](0, 0) = -0.3;
721         n.weights[1](0, 1) = 0.2;
722         vector<MVector> x = { {0.3,0.4},{0.5,0.6},{0.7,0.8} };
723         vector<MVector> y = { {1},{-1},{1} };
724         double s = 0;
725         cout << "Test for total cost function" << endl;
726         for (int i = 0;i < 3;i++)
727         {
728             n.FeedForward(x[i]);
729             s += n.Cost(y[i]);
```

```
730             cout << "C" << i + 1 << ":" << n.Cost(y[i]) << endl;
731         }
732         cout<<"Total cost : "<<n.TotalCost(x, y)<<endl;
733         if (abs( n.TotalCost(x, y) - s / 3.0) > 1e-10) { return false; }
734
735     }
736     // TODO: for each part of the Network class that you implement,
737     //       write some more tests (for other algorithms) here to run that
        code and verify that
738     //       its output is as you expect.
739     //       I recommend putting each test in an empty scope { ... }, as
740     //       in the example given above.
741
742     return true;
743 }
744
745 //
        ////////////////////////////////////////////////////////////////////////////////

746 // Main function and example use of the Network class
747
748 // Create, train and use a neural network to classify the data in
749 // figures 1.1 and 1.2 of the project description.
750 //
751 // You should make your own copies of this function and change the network
        parameters
752 // to solve the other problems outlined in the project description.
753 void ClassifyTestData()
754 {
755     // Create a network with two input neurons, two hidden layers of three
        neurons, and one output neuron
756     Network n({ 2,20,20,10,1 });
757
758     // Get some data to train the network
759     std::vector<MVector> x, y;
760     //GetTestData(x, y);
761     GetSpiralData(x, y);
762     //GetCheckerboardData(x, y);
763     // Train network on training inputs x and outputs y
764     // Numerical parameters are:
765     //  initial weight and bias standard deviation = 0.1
766     //  learning rate = 0.1
767     //  cost threshold = 1e-4
768     //  maximum number of iterations = 10000
769     bool trainingSucceeded = n.Train(x, y,0.5,0.01,0.01,5000000);
770
771     // If training failed, report this
772     if (!trainingSucceeded)
773     {
774         std::cout << "Failed to converge to desired tolerance." << std::
        endl;
775     }
776
777     // Generate some output files for plotting
778     ExportTrainingData(x, y, "test_points.txt");
779     n.ExportOutput("test_contour.txt");
780     //write data into a file.
781 }
782
783
784 int main()
```

```
785 {
786     //Call the test function
787     bool testsPassed = Network::Test();
788 
789     // If tests did not pass, something is wrong; end program now
790 
791     if (!testsPassed)
792     {
793         std::cout << "A test failed." << std::endl;
794         return 1;
795     }
796     //Tests passed, so run our example program.
797     ClassifyTestData();
798     return 0;
799 }
```

References

[Graves et al, 2009] Graves, A.; Liwicki, M.; Fernandez, S.; Bertolami, R.; Bunke, H.; Schmidhuber, J. "A Novel Connectionist System for Improved Unconstrained Handwriting Recognition", *IEEE Transactions on Pattern Analysis and Machine Intelligence. 31 (5): 855–868.* URL:http://www.idsia.ch/~juergen/tpami_2008.pdf

[Missinglink.AI] URL:https://missinglink.ai/guides/convolutional-neural-networks/graph-convolutional-networks/

[Galetzka, 2014] Michael Galetzka, "Intelligent Predictions: an empirical study of the Cortical Learning Algorithm", 2014.
URL:https://www.researchgate.net/publication/299748192_Intelligent_Predictions_an_empirical_study_of_the_Cortical_Learning_Algorithm

[Srivastava et al, 2014] Nitish Srivastava , Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever and Ruslan Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting ". *Journal of Machine Learning Research 15 (2014) 1929-1958*
URL: http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf?utm_content=buffer79b43&utm_medium=social&utm_source=twitter.com&utm_campaign=buffer

[Prajit, 2017] Ramachandran, Prajit; Barret, Zoph; Quoc, V. Le. "Searching for Activation Functions", t*arXiv:1710.05941 [cs.NE].* 2017
URL:https://arxiv.org/abs/1710.05941

[Zulkifli, 2018] Hafidz Zulkifli, "Understanding Learning Rates and How It Improves Performance in Deep Learning". *Towards Data Science. Retrieved 15 February 2019.*
URL:https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performar

[cao et al, 2017] Weipeng Cao, Xizhao Wang, Zhong Ming, Jinzhu Gao. "A review on neural networks with random weights", *Neurocomputing 275 (2018) 278–287*
URL:https://www.sciencedirect.com/science/article/abs/pii/S0925231217314613

[Telgarsky, 2015] Matus Telgarsky, "Representation Bene

ts of Deep Feedforward Networks", *Machine Learning (cs.LG); Neural and Evolutionary Computing (cs.NE)*
URL:https://arxiv.org/abs/1509.08101

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

[Cybenko, 1989] G. Cybenko, "Approximation by superpositions of a sigmoidal function",1989. *Mathematics of Control, Signals and Systems December 1989, Volume 2, Issue 4, pp 303–314*
URL:https://link.springer.com/article/10.1007/BF02551274

[Hornik, 1991] Kurt Hornik, "Approximation capabilities of multilayer feedforward networks", *Neural Networks Volume 4, Issue 2, 1991, Pages 251-257*
URL:https://www.sciencedirect.com/science/article/abs/pii/089360809190009T