C++ Project 2

November 2022
Student I.D. 10459970

1 An Overview of Neural Networks

In the Summer of 1966, computer scientist Marvin Minsky enlisted the help of undergrad student Gerald Sussman to help develop part of "visual system." His primary goal of the summer was to "construct a system of programs that will divide a [picture] into regions such as: likely objects, likely background areas, and chaos," with a final goal of "object identification."[1] This project was allocated one summer for completion. Yet, this problem is still being worked on to this day, with significant headway only being made within the last 15 years. What makes this problem, identifying what an object is, so different from other problems in computing, and why has it taken so long for progress to finally be made?

An algorithm is almost always comprised of three parts: inputs, a step-by-step process involving those inputs, and outputs. An algorithm with no inputs is a hard-coded task with a determined output (which can be practically useful, such as calculating π from an infinite series), and one with no outputs is the empty algorithm, that does nothing with any given inputs. However, these are the exception. Generally, the intermediate steps in a given algorithm have been specified by a human, with a precise correct answer to be determined. But, what if there is no precisely-defined correct answer, where the boundaries between different possible outputs are blurry, and poorly defined?

This is where neural networks come into play. Identifying objects in a picture isn't precisely defined. For example, a handwritten 4 and 9 are often difficult to distinguish between, and there are people trained specifically to decipher poor handwriting [2]. So, how are humans able to "learn" how to distinguish between the two? The human brain is a huge, interconnected web of neurons, in which there is both a process of learning and a process of recollection. The learning stage involves neurons in

the brain tweaking the function of each neuron and their connections in a way that can "recognise" patterns, then subsequently tweaking and refining both the neurons and the connections to better identify those patterns. The recollection stage involves seeing a new object, and running the inputs through these networks of neurons to see which ones identify the object. We now have an input, an unknown process with a known structure, and an output. Ever since the 1960s[3], there have been attempts to use series of logic gates (and later bounded functions) to try to recreate this process of learning and recollection using networks of connected nodes, but only recently has there been enough computing power available to properly train these networks, with this area of research being very active currently. But, how do they work?

2 Training a Neural Network

Before continuing, here is a diagram of a simple, fully connected, forward-feeding neural network (more on what that specifically means later).

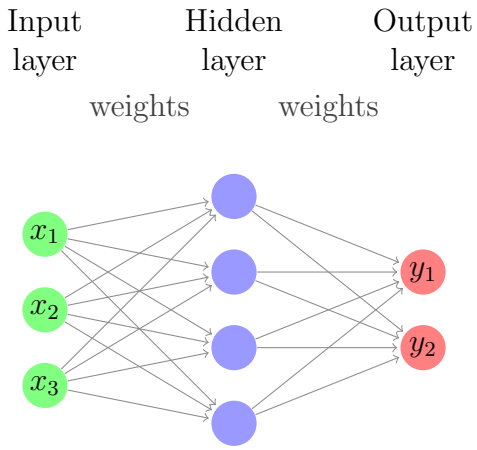


Figure 1: A diagram of a neural network.

This is mathematically known as a directed graph, with the vertices representing neurons and the edges representing connections between neurons. The neurons themselves take an input, perform a biased function on that input (such as $y = \tanh(x + bias)$), and then pass on their output to the connections. The connections have a certain weight, only passing on a certain proportion of the input to the next layer of nodes. There are three types of neurons: the input neurons (which, unlike the rest, don't have a function), the hidden neurons, and the output neurons.

Now, we can more easily define the training and recollection processes.

The initial network will be given random biases and weights on every neuron and connection, allowing us to begin the learning process. The learning process involves two stages: testing the network and subsequently tweaking it.

Testing the network the input neurons receiving some data with an associated expected output, which is then passed through the network of hidden neurons, repeatedly being weighted, biased and put through a function, until it reaches the output neurons. The output of the network is then compared to the expected output, and the error (cost) is determined. Since the outputs are determined by the inputs and the functions between, we can recursively feed back the cost backwards through the network, and determine the cost associated with every input and weight.

From that, we can determine by how much we should tweak each weight and bias (using a *stochastic gradient*), which tweaks the network with the desire of the network output being closer to the expected output. This is repeated many times, with occasional breaks to calculate the *true cost,* which is the mean squared error of all data we are training the network with. Eventually, the true cost will approach an acceptably small number, which means that our network is outputting similar results to what we are expecting. This network is now trained.

The aim of a trained network is not to be able to sort already existing data into the correct categories (this could be done easily without a neural network). Our aim is to take new data of the same form, and use our neural network to generate an output, with the expectation that the network is now reasonably competent at finding the correct output. But, will this expectation be true?

3 Testing and Optimising Neural Networks

After a network has been trained with *training data*, it is important to then test the network with new *validation data*. Although the true cost of our training data will have decreased to an acceptably small value, this may not hold true for other samples of data. For example, a neural network may have so many parameters that noise in the training data or empty sample spaces may be wrongly accounted for in the training stage. This is called over-fitting, which will result in a low overall cost in training data but a higher overall cost in a different sample of the same dataset. This is where validation data is used: to estimate how accurate the network is for the data you're using. It will be able to identify your network's mean numerical error for

numerical data, classification error for data belonging to classes, or whatever other error measure your dataset can be measured in. Then, if the network has shown to be over-fit, you can attempt to re-run your network with fewer parameters to deter it finding more complicated patterns than exist within the data, or feed the network *augmented data*, which is a method of altering the inputs of a data point in some form whilst keeping the same desired output. This allows for more diverse data without having to collect new data. But, so far we have only discussed one type of neural network.

There are many variations of the standard model of neural network. Fig. 1 has connections that only feed in one direction. But, networks can have connections where a neurons connects within its own layer, or even backwards or to itself! These are called *recurrent* networks. Another property of Fig. 1 is that any two layers have weights leading from every neuron in the first layer to every neuron in the second, making the network *fully connected.* In reality, most networks begin to "drop" less-useful connections as they train, and those can be ignored entirely if they have a negligible effect on the outcome. This can even help reduce over-fitting, as fewer parameters exist within the network. Dropping these weights speeds up training and recollection, and make the network *convolutional.* Both these types of network are beyond the scope of the project.

A real-life example of a more complex neural network would be giving an appropriately-trained neural network a picture of either a cat or dog, and the network correctly determining which one the picture is of. Of course, the idea that a carefully tweaked, huge neural network can do that seems abstract and confusing. This is absolutely true! Changing the number of layers of hidden neurons in the network, how many neurons are in each hidden layer, and how far the network should attempt to change each iteration all affect the network in difficult-to-determine ways. Many networks fail to realise underlying patterns in data, and most of the rest only begin to converge after a computationally prohibitive amount of time. Neural networks are fickle, mystical boxes with indecipherable insides that can seemingly magically do what we can't program ourselves, yet also are hugely scaled-down versions of what our own brains do with ease.

4 Programming my own Neural Network

We are given a vector and matrix class in *C++*, with given operations allowing for easy implementation of the equations used in the neural network algorithm. From this, I have been tasked with implementing functions on the class *Network*, which

when called together can train a neural network of any given size using input training data. The program then must run new data through the trained network and output the results to a file.

To understand the functions necessary to create a neural network algorithm, we first need to define the algorithm itself. Here is the algorithm of a standard neural network:

1. Choose a learning rate η, the amount we attempt to modify the network by each step. Too small a rate results is slow progress, and too high will skip right past where the learning direction is useful.
2. Choose initial bias and weight values in the network. In our network, these will be normally distributed random variables, making up our parameter vector $\mathbf{p}$.
3. Choose a random training data point $i \in \{1, 2, \ldots, N\}$.
4. Run the data point through our network (called a pass of the feed-forward algorithm) to evaluate the neural network output $\mathbf{a}^{[L]}$ (the output values of the network at output layer L) from the training input $\mathbf{x}^{\{i\}}$:
   (a) Evaluate the input data $\mathbf{a}^{\{1\}}$. Set layer $l = 2$.
   (b) Evaluate $\mathbf{a}^{\{l\}}$ according to weight and biases of the layer of the network.
   (c) If $l < L$, increase l by one and go back to step (b).
5. Run the error value back through the network (a pass of the back-propagation algorithm) to evaluate the error values $\boldsymbol{\delta}^{[l]}$ of all weights and biases:
   (a) Evaluate the cost of the chosen data point at the output layer (error $\boldsymbol{\delta}^{[L]}$). Set layer $l = L$.
   (b) Perform one step of an iterative equation that tracks backwards through our network, that uses the derivative of our neuron activation functions to calculate the error associated with the weights and biases at layer ($\boldsymbol{\delta}^{[l-1]}$).
   (c) If $l > 2$, decrease l by 1 and go back to step (b).
   Otherwise, all errors $\boldsymbol{\delta}$ have been found (since no errors can be associated with the input layer).
6. Apply a stochastic gradient iteration, which calculates from our error values $\boldsymbol{\delta}^{[l]}$ the amounts each parameter in the network should change by ($\nabla C_{x^{\{i\}}}$).

7. Check whether the network has been sufficiently trained:

   (a) Calculate the total cost C. Each term in the mean squared error sum is evaluated by looping over every data point $\mathbf{x}^{\{i\}}$: for $i = 1, 2, \ldots N$ and evaluating $\mathbf{a}^{[L]}$ from step 4's feed-forward algorithm.

   (b) If the total cost is less than some chosen small threshold, the network has been deemed sufficiently trained and the algorithm terminates. Since this has to be done for a whole dataset, this step is relatively computationally expensive and should not be done every step. Skip over except when an iterator hits a multiple of a large number (I used 1,000).

8. Repeat steps 3 through 6 a large number of times (between 10^6 and 10^7 is about right).

9. If the algorithm hasn't terminated, the network has failed to converge in a reasonable time. Repeat the algorithm with a modified network structure, learning rate, or convergence threshold to increase chance of convergence.

Now we have a well-defined algorithm, we can start implementing it in *C++*. The full code is available in the appendix, which along with the files `mmatrix.h` and `mvector.h` will run a network. The functions that actually run the neural network algorithm are explained below:

Step 1 is simply calling the over-arching `train` function with a specific parameter, so doesn't require a function.

Step 2 does require a function, that assigns a normally distributed random variable to all weights and biases. Note the vector `biases` is a vector of vectors, while the vector `weights` is a vector of matrices.

```
void InitialiseWeightsAndBiases(double initsd){ //set up initial network
    assert(initsd>=0); //ensure standard deviation >= 0
    //set up normal distribution with sd = initsd, sampled by dist(rnd)
    std::normal_distribution<> dist(0, initsd);
    for(int i = 1; i < nLayers; i++){ //for each non-input layer
        for(int j = 0; j < nneurons[i]; j++){ //weight leading to j
            for(int k = 0; k < nneurons[i - 1]; k++){ //from k
                weights[i](j, k) = dist(rnd); //set up random weights}}
        for(int j = 0; j < nneurons[i]; j++){
            biases[i][j] = dist(rnd); //set up random biases}}}
```

To test my functions, I used a `test` function which could test my individual steps with a specific case to see whether my code was working, either from inspection or

through a boolean test. For `InitialiseWeightsAndBiases`, I set up a network, and user `RStudio` to test the distribution of weights and biases:

```
Network n({1, 200}); //200 weights and biases in network
n.InitialiseWeightsAndBiases(5); //mean should be 0, sd should be 5
std::cout << n.biases[1] << std::endl;
std::cout << n.weights[1] << std::endl;
```

```
> vector <- c(n.biases[1]);
> mean(vector)
[1] -0.1241582
> sd(vector)
[1] 5.246925
> vector <- c(n.weights[1]);
> mean(vector)
[1] 0.0417329
> sd(vector)
[1] 5.0916224
```

From inspection, the weights and biases appear to be correctly distributed.

Step 3 requires choosing a random variable from our training dataset `x`:

```
int i = rnd()%x.size(); //random integer modulo size of dataset
```

Step 4 requires running data through the network, needing two functions; one that passes data through the network, and one that performs the activation function.

```
double Sigma(double z){ //perform activation function
                return tanh(z);}

void FeedForward(const MVector &x){ //pass data through neural network
    //check input vector same size as input layer
    assert(x.size() == nneurons[0]);
    activations[0] = x; //match data to input layer
    for(int i = 1; i < nLayers; i++){ //pass through entire network
        //feed data forward one layer
        inputs[i] = (weights[i] * activations[i - 1]) + (biases[i]);
        for(int j = 0; j < nneurons[i]; j++){
            //run activation function for each neuron in current layer
            activations[i][j] = Sigma(inputs[i][j]);}}}
```

Step 5 requires calculating the error associated with each parameter within the network, requiring two functions; one that propagates error backwards through the network, and one that performs the derivative of the activation function, which is used in the calculation.

```
double SigmaPrime(double z){ //return derivative of activation function
    return 1 - pow(tanh(z), 2);}

//work backwards through network, evaluating errors for each parameter
void BackPropagateError(const MVector &y){
    //check training data has same number of elements as output layer
    assert(y.size() == nneurons[nLayers - 1]);
    MVector z(y.size()); //set up error calculation vector
    for(int i = 0; i < y.size(); i++){ //for each output neuron
        //calculate activation derivative for neurons in output layer
        z[i] = SigmaPrime(inputs[nLayers - 1][i]);}
    //calculate error values for output layer
    errors[nLayers - 1] = z * (activations[nLayers - 1] - y);
    for(int i = nLayers - 2; i > 0; i--){ //go backwards through network
        MVector z(nneurons[i]); //set up error calculation vector
        for(int j = 0; j < nneurons[i]; j++){ //for each neuron in layer
            z[j] = SigmaPrime(inputs[i][j]);} //activation derivative
        //calculate error valuesfor current layer
        errors[i] = z * TransposeTimes(weights[i + 1], errors[i + 1]);}}
```

Step 6 calculates the errors associated with each parameter and updates the weight and biases accordingly, which can be done in one function:

```
void UpdateWeightsAndBiases(double eta){
    assert(eta>0); //check that the learning rate is positive
    for(int i = 1; i < nLayers; i++){ //for each layer in network
        //calculate error for weights and biases in current layer and
        //update their values
        biases[i] -= eta * errors[i];
        weights[i] -= eta * OuterProduct(errors[i], activations[i-1]);}}
```

Step 7 is the last step requiring its own functions, with steps 8 and 9 being part of the `train` function. These functions calculate the total cost of all data points when all run through the current version of the network:

```
double Cost(const MVector &y){ //find cost of one data point
    //check output layer has same number of elements as data
    assert(y.size() == nneurons[nLayers-1]);
    MVector z(y.size()); //cost calculation vector
    for(int i = 0; i < y.size(); i++){ //for each output node
        //find difference between expected and actual output values
        z[i] = (y[i] - (activations[nLayers - 1][i]));
        z[i] = 0.5 * pow(z[i], 2);} //mean square error
    double cost = 0;
    for(int i = 0; i < y.size(); i++){
        cost += z[i];} //sum up all error components in z
    return cost;}

//return total cost of all training data within current network
```

```
double TotalCost(const std::vector<MVector> x,
const std::vector<MVector> y){
    assert(x.size() == y.size()); //check training data is matched up
    double totalCost = 0;
    for(int i = 0; i < x.size(); i++){ //for all training data
        FeedForward(x[i]); //run through network
        totalCost += Cost(y[i]);} //calculate error and add to sum
    return totalCost / y.size();}
```

To test steps 3 through 7, I set up a simple neural network and calculated the expected results of training a specified training data by hand, then compared the values:

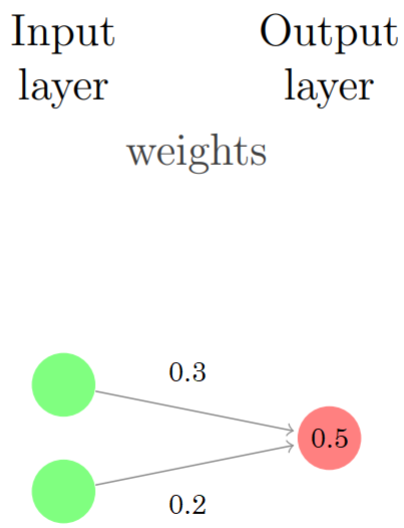


Figure 2: My neural network, with fixed initial values.

```
training data: input = {(0.3, 0.4), (0.6, -0.5)}; output = {(1), (-1)}
network feedforward training data 1 output = tanh(0.5+0.3*0.3+0.2*0.4)
= 0.584979883
network feedforward training data 2 output = tanh(0.5+0.3*0.6+0.2*-0.5)
= 0.52266543
costs = 0.5 * ((1-0.584979883)^2,(-1-0.52266543)^2)
= (0.0861208488, 1.15925501)
total cost = (0.0861208488 + 1.15925501) / 2 = 0.622687929
network backpropogation error for data point 1 =
(1-tanh(0.67)^2)*(0.584979883 - 1)
= -0.272999626
```

```
network updated weights (eta = 0.1) = [0.3, 0.2] -
(0.1 * -0.272999626 * [0.3, 0.4]) = [0.308189989, 0.210919985]
network updated biases (eta = 0.1) = 0.5 - (0.1 * -0.272999626)
= 0.527299963
```

I then compared my calculated values with the following *C++* code:

```
Network n({2, 1});
std::cout << n.TotalCost({{0.3, 0.4}, {0.6, -0.5}},
{{1}, {-1}}) << std::endl;
n.biases[1][0] = 0.5; n.weights[1](0,0) = 0.3; n.weights[1](0,1) = 0.2;
n.FeedForward({0.3, 0.4});
std::cout << n.activations[1] << std::endl;
n.BackPropagateError({1});
std::cout << n.errors[1] << std::endl;
n.UpdateWeightsAndBiases(0.1);
std::cout << n.weights[1] << std::endl;
std::cout << n.biases[1] << std::endl;
```

```
>> 0.622688
>> (0.58498)
>> (-0.273)
>> (   0.30819,   0.21092)
>> (0.5273)
```

The calculated values matched the function outputs of my neural network, showing the functions work as planned.

Finally, steps 8 and 9 are implemented by connecting all our functions so far into one big `train` function:

```
bool Train(const std::vector<MVector> x,
const std::vector<MVector> y, double initsd, double learningRate,
double costThreshold, int maxIterations){ //train neural network
    assert(x.size() == y.size()); //check training data is matched
    InitialiseWeightsAndBiases(initsd); //step 2
    for (int iter=1; iter<=maxIterations; iter++){ step 8 repeater
        int i = rnd()%x.size(); //step 3
        FeedForward(x[i]); //step 4
        BackPropagateError(y[i]); //step 5
        UpdateWeightsAndBiases(learningRate); //step 6
        if ((!(iter%1000)) || iter==maxIterations){ //every 1000 passes
            double cost = TotalCost(x, y); //step 7
            //print cost
            std::cout << iter << "␣␣␣␣" << cost << std::endl;
            if(cost < costThreshold){
```

```
                return true;}}}
    return false; //step 9 (if network fails to converge)}
```

Now, we have a set up a training algorithm for a neural network!

5 Testing the Network

We are given test data. The input is 16 points on a $2-D$ Cartesian plane between $(x, y) \in [0, 1]$, and the outputs are either -1 or 1. Running these test data through a network of $[2, 3, 3, 1]$ neurons, with given $\eta = 0.1, initsd = 0.1, costThreshold = 1e{-}4$ (hyperparameter values), we find a convergence in 32,000 iterations. I then ran a grid of new data points through the trained network (251 points spaced evenly between 0 and 1), and plot a contour diagram of our results, overlayed on our training data:

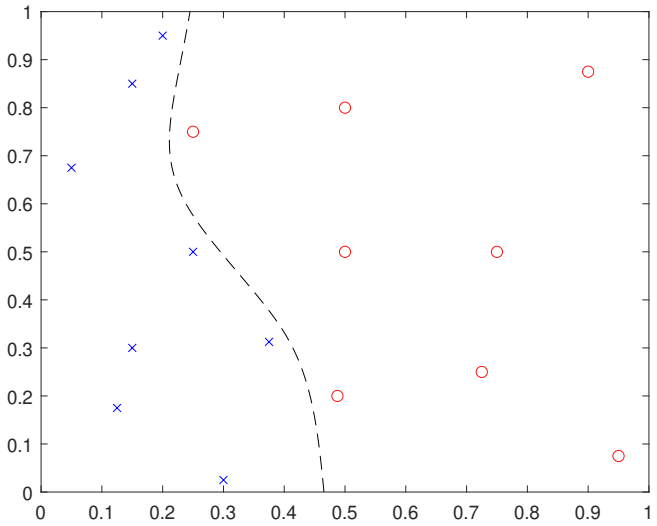
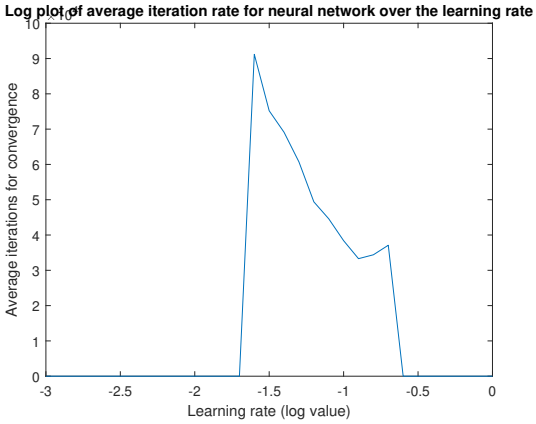


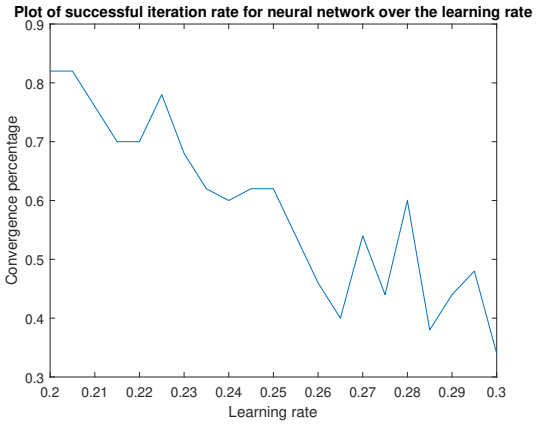
Figure 3: A contour plot of our new data points, with the test data overlayed.

The neural network worked! These training data have been correctly classified, and the contour line is roughly what would be expected. Importantly, the contour doesn't look like a simple mathematical function of x and y; it resembles the line a human would've drawn with a pencil given the diagram without a contour. The network has organically learned from data shown to it, without using regression or another predefined algorithm. But, this has been done using specified hyperparameters. Can we get the network to converge more quickly if we alter the hyperparameters?

Let's start with η. Our learning rate decides how far our network adapts to the stochastic gradient in each training iteration, and is a crucial factor in determining if and how quickly our network will converge. Too low a learning rate will slow the rate of convergence and may cause the network to fall into a sub-optimal local minimum, and too large a learning rate will fail to account for gradient changes, causing the network to jump over local minima completely, causing it to jump around erratically and only converge by happenstance. To test the learning rate on my neural network, I tested the average convergence rate for my training for values spaced logarithmically between 0.001 and 1:



(a) A contour plot of our new data points, with the test data overlayed. A value of zero is shown when the network failed to converge within 100,000 iterations over 50% of the time.



(b) A plot of the convergence success rate over the learning rate.

The learning rate hugely impacts the convergence time. For this simple dataset, a learning rate below $10^{-1.7}$ usually converges in over 100,000 iterations (the maximum convergence time I could use whilst collecting a meaningful amount of data), and decreases with an increased learning rate until roughly $10^{-0.9}$. After this, the convergence time increases again, and quickly crosses back over the 100,000 iteration cutoff. The decrease in convergence time shows that a faster learning rate does indeed learn more quickly, but the sudden increase is only explained in the context of these data. Although the unsuitably high learning rate continued to converge quickly when a local minima was found, it more often skipped over any local minima and instead bounced around the solution space, taking far longer to begin to converge than any time saved once a convergence was stumbled upon. This is made clear when inspecting the percentage of the time that the network converges between $\eta = 0.2$ and

$\eta = 0.3$, which sharply decreases towards the latter end. In the case of this network, $\eta = 0.1$ appears to be the happy medium.

Another hyperparameter that can alter the convergence of our network is the variance of the initial weights and biases. As shown by Fig. 5, there is a "Goldilocks zone" of quick convergence between 10^{-2} and $10^{0.5}$. No variance at all means the network struggles to find a meaningful direction to travel in since the inputs will all have the same output (zero), which means that the network may fail to recognise patterns within the data, and too large a variance starts the network with a huge total cost, far away from the space where lower-cost pockets occur and convergence will happen. Unlike the learning rate, there doesn't appear to be a specific best initial variance, but instead a reasonable range where the initial values are within a sensible enough range for the network to learn.

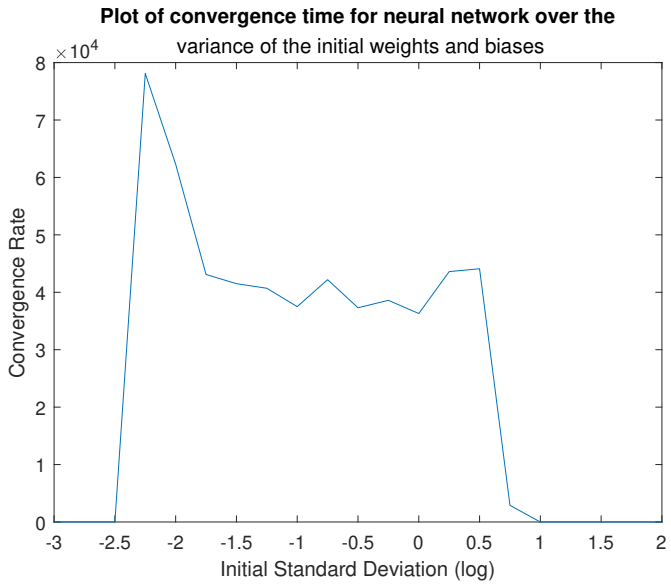
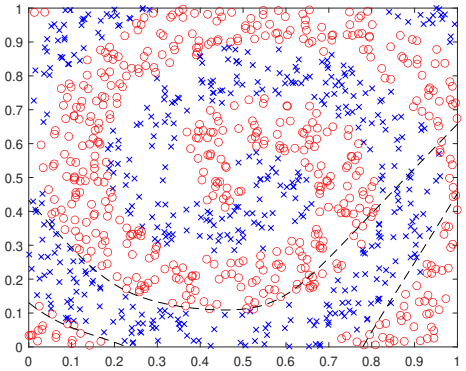


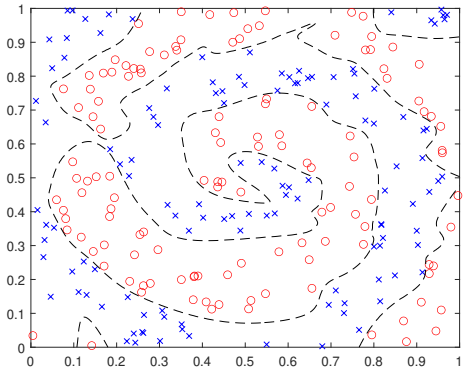
Figure 5: A contour plot of our new data points, with the test data overlayed.

We are also given other, more extensive training data in the form of `GetCheckerboardData` and `GetSprialData`. Each generate 1,000 random points between 0 and 1 on the XY-plane, split into two categories as before. These data have much more complicated underlying patterns to learn than the training data we've been using so far, and as a result require more neurons and layers to be able to learn them. However, over-fitting can become a problem in our model of neural network when too many layers and neurons are added. The idea of altering the network's composition to account for

over-fitting beings up more questions. Does under-fitting exist? What do the number of neurons and layers in your network control? In general, the more neurons in your data and the more layers of data you have, the more complicated the patterns your network can uncover. Too few, and your network won't be able to perceive complicated patterns in your data (for example, without a hidden layer a neural network can only decide a linear split in data), which is called under-fitting (see Fig. 6). Too many, and the network will account for patterns which are artefacts of the data you have collected instead of true properties of the pattern attempting to be found (see Fig. 7).

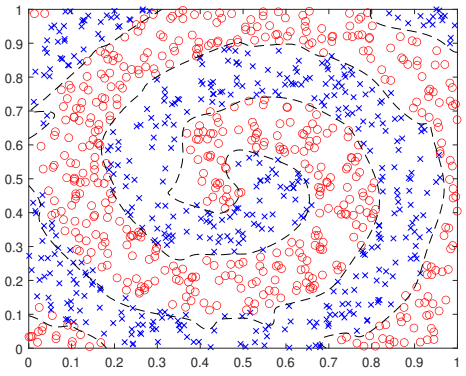


(a) Figure 6: An example of under-fitting in a neural network. The network identified a couple of boundaries but the network wasn't complicated enough to be able to draw a spiral. Network $\in \{2, 3, 3, 1\}$, cost of training data = 0.371691, cost of 1,000,000 randomly-generated validation data = 0.401652.

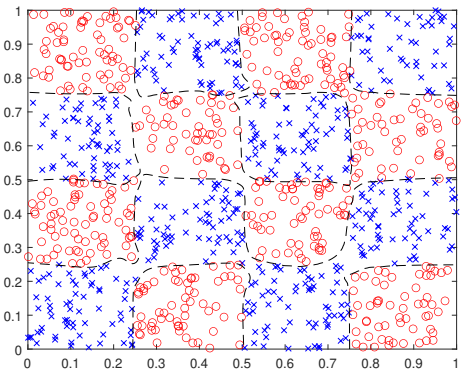


(b) Figure 7: An example of over-fitting in a neural network. The spiral shape is generally followed, but the line has waves and breaks which wouldn't be drawn by a human, who would notice the spiral pattern and would know any further detail is due to sparse data (250 points instead of the usual 1,000). Network $\in \{2, 20, 20, 20, 1\}$, cost of training data = 0.00824296, cost of 1,000,000 randomly-generated validation data = 0.228673.

Although there are two ways of increasing the complexity of a neural network (number of layers and number of neurons per layer), it is difficult to determine how exactly they change the network's complexity. In general, finding a good network structure goes down to a initial educated guess, followed by trial and error.



(a) Figure 8: An example of a well-fit neural network. The spiral pattern is identified well, and generally doesn't veer off-path. Network $\in \{2, 13, 13, 1\}$, cost of training data = 0.0707537, cost of 1,000,000 randomly-generated validation data = 0.143889.



(b) Figure 9: Another example of a well-fit neural network, instead using a checkerboard pattern. Network $\in \{2, 13, 13, 1\}$, cost of training data = 0.00973521, cost of 1,000,000 randomly-generated validation data = 0.0510734.

However, since neural networks are still a very recent and poorly understood innovation, the rules for deciding the right structure of a network are vague and require trial and error. Moreover, even without over-fitting, a large network requires huge amount of computing to train so is undesirable. So how is a middle-ground decided, and can anything be changed about the training process to help guide a network?

At this point, the fact that neural networks are a new area of research becomes apparent. There only exists rules of thumb in deciding how big or small your network should be, depending on your data's form and complexity. We are still far off of understanding exactly how neural networks can reveal patterns, especially considering that some networks can find properties of data that we can't as humans. Furthermore, since neural network algorithms have already garnered widespread commercial use, it is often in the interest of large companies to keep secret any development made in the area. Over the last few years, neural networks have come a long way, and without doubt the field will continue to develop at a fast pace. Even Alan Turing's famous Turing test has arguably been broken by some recent neural networks, and the goalposts for networks to achieve have been moved far beyond what some previously thought possible. This has both amazing and terrifying implications on technology as a whole (which is unfortunately beyond the scope of this report), and goes to show just how truly fascinating and novel neural networks are.

6 Appendix

```
#include "mvector.h"
#include "mmatrix.h"
#include <cmath>
#include <random>
#include <iostream>
#include<iomanip>
#include <fstream>
#include <cassert>


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

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

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

// Alternative: set up the generator with an arbitrary constant integer. This can be u
// debugging because the program produces the same sequence of random numbers each tim
// To get this behaviour, uncomment the line below and comment the declaration of "rnd
//std::mt19937 rnd(12345);


////////////////////////////////////////////////////////////////////////////
// Some operator overloads to allow arithmetic with MMatrix and MVector.
// These may be useful in helping write the equations for the neural network in
// vector form without having to loop over components manually.
//
// You may not need to use all of these; conversely, you may wish to add some
// more overloads.

// 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 training data in the pattern illustrated in the project descr
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}, {0.2
                {0.75, 0.5}, {0.95, 0.075}, {0.4875, 0.2}, {0.725,0.25}, {0.9,0.875},

        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));

        for (int i=0; i<1000; i++)
        {
                x[i]={lr()/static_cast<double>(lr.max()),lr()/static_cast<double>(lr.m
                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.m
                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 "fi
// 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]
                for (int j=0; j<x[i].size(); j++)
                {
                        f << x[i][j] << " ";
                }

                // 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
        // 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 neur
        //                      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 project description 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
            //
            // Note that, since C++ vector indices run from 0 to N-1, all the indi
            // code are one less than the indices used in the mathematics (which r
    }

    // 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 outpu
    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 outlined in section 1.3.3
    // This should be implemented by calling the appropriate private member functi
    bool Train(const std::vector<MVector> x, const std::vector<MVector> y,
                    double initsd, double learningRate, double costThreshold, i
    {
            // Check that there are the same number of training data inputs as out
            assert(x.size() == y.size());
            // TODO: Step 2 - initialise the weights and biases with the standard
            InitialiseWeightsAndBiases(initsd);
            for (int iter=1; iter<=maxIterations; iter++)
            {
                    // Step 3: Choose a random training data point i in {0, 1, 2,
                    int i = rnd()%x.size();

                    // TODO: Step 4 - run the feed-forward algorithm
                    FeedForward(x[i]);
                    // TODO: Step 5 - run the back-propagation algorithm
                    BackPropagateError(y[i]);
                    // TODO: Step 6 - update the weights and biases using stochast
```

```
                        //                  with learning rate "learningRate"
                        UpdateWeightsAndBiases(learningRate);
                        // Every so often, perform step 7 and show an update on how th
                        // Here, "every so often" means once every 1000 iterations, an
                        if ((!(iter%1000)) || iter==maxIterations)
                        {
                                // TODO: Step 7(a) - calculate the total cost
                                double cost = TotalCost(x, y);
                                // TODO: display the iteration number and total cost t
                                std::cout << iter << "    " << cost << std::endl;
                                // TODO: Step 7(b) - return from this method with a va
                                //                   indicating success, if this cost
                                if(cost < costThreshold){
                                        return true;
                                }
                        }

                } // Step 8: go back to step 3, until we have taken "maxIterations" st

                // Step 9: return "false", indicating that the training did not succee
                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 output data points
                for (int i=0; i<=250; i++)
                {
                        for (int j=0; j<=250; j++)
                        {
                                MVector out = Evaluate({i/250.0, j/250.0});
```

```
                                f << out[0] << " ";
                        }
                        f << std::endl;
                }
                f.close();

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


        static bool Test();

private:
        // Return the activation function sigma
        double Sigma(double z)
        {
                return tanh(z);
        }

        // Return the derivative of the activation function
        double SigmaPrime(double z)
        {
                return 1 - pow(tanh(z), 2);
        }

        // Loop over all weights and biases in the network and set each
        // term to a random number normally distributed with mean 0 and
        // standard deviation "initsd"
        void InitialiseWeightsAndBiases(double initsd){
                // Make sure the standard deviation supplied is non-negative
                assert(initsd>=0);

                // Set up a normal distribution with mean zero, standard deviation "in
                // Calling "dist(rnd)" returns a random number drawn from this distrib
                std::normal_distribution<> dist(0, initsd);

                // TODO: Loop over all components of all the weight matrices
                //       and bias vectors at each relevant layer of the network.
                for(int i = 1; i < nLayers; i++){
                        for(int j = 0; j < nneurons[i]; j++){
                                for(int k = 0; k < nneurons[i - 1]; k++){
                                        weights[i](j, k) = dist(rnd);}}
                        for(int j = 0; j < nneurons[i]; j++){
                                biases[i][j] = dist(rnd);}}}

        // Evaluate the feed-forward algorithm, setting weighted inputs and activation
        // 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
                assert(x.size() == nneurons[0]);
                activations[0] = x;
                for(int i = 1; i < nLayers; i++){
                        inputs[i] = (weights[i] * activations[i - 1]) + (biases[i]);
                        for(int j = 0; j < nneurons[i]; j++){
                                activations[i][j] = Sigma(inputs[i][j]);
                        }
                }
                // TODO: Implement the feed-forward algorithm, equations (1.7), (1.8)
        }

        // 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 t
                assert(y.size() == nneurons[nLayers - 1]);
                MVector z(y.size());
                for(int i = 0; i < y.size(); i++){
                        z[i] = SigmaPrime(inputs[nLayers - 1][i]);
                }
                errors[nLayers - 1] = z * (activations[nLayers - 1] - y);
                for(int i = nLayers - 2; i > 0; i--){
                        MVector z(nneurons[i]);
                        for(int j = 0; j < nneurons[i]; j++){
                                z[j] = SigmaPrime(inputs[i][j]);
                        }
                        errors[i] = z * TransposeTimes(weights[i + 1], errors[i + 1]);
                }
                // TODO: Implement the back-propagation algorithm, equations (1.22) an
        }


        // Apply one iteration of the stochastic gradient iteration with learning rate
        void UpdateWeightsAndBiases(double eta)
        {
                // Check that the learning rate is positive
                assert(eta>0);
                for(int i = 1; i < nLayers; i++){
                        biases[i] -= eta * errors[i];
                        weights[i] -= eta * OuterProduct(errors[i], activations[i-1]);
                }
                // TODO: update the weights and biases according to the stochastic gra
                //       iteration, using equations (1.25) and (1.26) to evaluate
                //       the components of grad C.
        }
```

```
        // Return the cost function of the network with respect to a single the desire
        // Note: call FeedForward(x) first to evaluate the network output for an input
        //       then call this method Cost(y) with the corresponding desired output y
        double Cost(const MVector &y)
        {
                // Check that y has the same number of elements as the network has out
                assert(y.size() == nneurons[nLayers-1]);
                MVector z(y.size());
                for(int i = 0; i < y.size(); i++){
                        z[i] = (y[i] - (activations[nLayers - 1][i]));
                        z[i] = 0.5 * pow(z[i], 2);
                }
                double cost = 0;
                for(int i = 0; i < y.size(); i++){
                        cost += z[i];
                }
                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 totalCost = 0;
                for(int i = 0; i < x.size(); i++){
                        FeedForward(x[i]);
                        totalCost += Cost(y[i]);
                }
                return totalCost / y.size();
                // TODO: Implement the cost function, equation (1.9), using
                //       the FeedForward(x) and Cost(y) methods
        }

        // Private member data

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

};



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
        /*
        {
                // Make a simple network with two weights and one bias
                Network n({2, 2, 2});

                // Set the values of these by hand
                n.biases[1][0] = 0.6;
                n.biases[1][1] = 0.8;
                n.biases[2][0] = 0.1;
                n.biases[2][1] = 0.2;

                n.weights[1](0,0) = 0.2;
                n.weights[1](0,1) = 0.7;

                n.weights[1](1,0) = 0.4;
                n.weights[1](1,1) = 0.8;

                n.weights[2](0,0) = 0.1;
                n.weights[2](0,1) = 0.5;

                n.weights[2](1,0) = 0.1;
                n.weights[2](1,1) = 0.2;
                // Call function to be tested with x = (0.3, 0.4)
                n.FeedForward({0.1, 0.1});
                // Display the output value calculated
                std::cout << n.errors[1] << "     " << n.errors[2] << std::endl;

                // Correct value is = tanh(0.5 + (-0.3*0.3 + 0.2*0.4))
                //                  = 0.4542164326822259...
                // Fail if error in answer is greater than 10^-10:
                if (std::abs(n.activations[1][0] - 0.4542164326822259) > 1e-10)
                {
                        return false;
                }
        }

        {
                // Make a simple network with two weights and one bias
                Network n({2, 1});
```

```
        // Call function to be tested with z = (0.3)
        double z = n.Sigma({0.3});

        // Display the output value calculated
        std::cout << std::setprecision(11) << z << std::endl;

        // Correct value is = tanh(0.3)
        //                  = 0.291312612245...
        // Fail if error in answer is greater than 10^-10:
        if (std::abs(z - 0.291312612245) > 1e-10)
        {
            return false;
        }
    }
    {
        // Make a simple network with two weights and one bias
        Network n({2, 1});

        // Call function to be tested with z = (0.3)
        double z = n.SigmaPrime({0.3});

        // Display the output value calculated
        std::cout << std::setprecision(11) << z << std::endl;

        // Correct value is = 1 - tanh(0.3) ^ 2
        //                  = 0.915136961820...
        // Fail if error in answer is greater than 10^-10:
        if (std::abs(z - 0.915136961820) > 1e-10)
        {
            return false;
        }
    }
    {
        // Make a simple network with two weights and one bias
        Network n({2, 3, 3, 1});

        // Call function to be tested with x = (0.3, 0.4)
        n.InitialiseWeightsAndBiases(1);

        // Display the output value calculated
        std::cout << n.weights[0] << "e" << n.weights[1] << "e" << n.weights[2
        std::cout << "f" << n.biases[0] << "e" << n.biases[1] << "e" << n.bias
    }
    {
        // Make a simple network with two weights and one bias
        Network n({1, 2});
```

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

            // Call function to be tested with x = (0.3, 0.4)
            n.FeedForward({0.3});
            MVector y = {1, 1};
            // Display the output value calculated
            std::cout << n.Cost(y) << std::endl;

            // Correct value is = tanh(0.5 + (-0.3*0.3 + 0.2*0.4))
            //                  = 0.4542164326822259...
            // Fail if error in answer is greater than 10^-10:
            if (std::abs(n.activations[1][0] - 0.4542164326822259) > 1e-10)
            {
                    return false;
            }
    }
    {
            // Make a simple network with two weights and one bias
            Network n({1, 2});

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

            // Call function to be tested with x = (0.3, 0.4)
            // Display the output value calculated
            std::cout << n.TotalCost({{0.3}, {0.6}}, {{1, 1}, {-1, -1}}) << std::e

            // Correct value is = tanh(0.5 + (-0.3*0.3 + 0.2*0.4))
            //                  = 0.4542164326822259...
            // Fail if error in answer is greater than 10^-10:
            if (std::abs(n.activations[1][0] - 0.4542164326822259) > 1e-10)
            {
                    return false;
            }
    }
    {
            // 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});
                n.BackPropagateError({1});
                std::cout << n.weights[1] << std::endl;
                n.UpdateWeightsAndBiases(0.1);
                // Display the output value calculated
                std::cout << n.weights[1] << std::endl;

                // Correct value is = tanh(0.5 + (-0.3*0.3 + 0.2*0.4))
                //                  = 0.4542164326822259...
                // Fail if error in answer is greater than 10^-10:
                if (std::abs(n.activations[1][0] - 0.4542164326822259) > 1e-10)
                {
                        return false;
                }
        }
        */
        return true;
}

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

// Create, train and use a neural network to classify the data in
// figures 1.1 and 1.2 of the project description.
//
// You should make your own copies of this function and change the network parameters
// to solve the other problems outlined in the project description.
void ClassifyTestData()
{
        // Create a network with two input neurons, two hidden layers of three neurons
        Network n({2, 20, 20, 20, 1});

        // Get some data to train the network
        std::vector<MVector> x, y;
        GetCheckerboardData(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, 0.01, 0.01, 10000000);
```

```
        // 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.txt");
}

int main()
{
        // 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;
        }

        // Tests passed, so run our example program.
        ClassifyTestData();

        return 0;
}
```

References

1: Papert, S (1966). *The Summer Vision Project.* http://dspace.mit.edu/bitstream/handle/1721.1/6125/AIM-100.pdf

2: Brumble, S (2015). *The USPS Facility That Deciphers Your Illegible Handwriting.* http://www.slate.com/blogs/atlas_obscura/2015/05/19/unreadable_mail_ends_up_at_the_usps_remote_encoding_facility_in_salt_lake.html?via=gdpr-consent

3: Jaspreet (2016). *A Concise History of Neural Networks.* https://towardsdatascience.com/a-concise-history-of-neural-networks-2070655d3fec