Artifical Neural Networks

by Edward Small

Submission date: 15-Jan-2021 01:34AM (UTC+0000)
Submission ID: 142175125
File name: Neural_Networks_1.pdf (3.35M)
Word count: 12000
Character count: 57987

Excellent work, see comments below.

Neural Networks and Machine Learning

Edward Alexander Small (10786391)

November 2020

Full code, including results from computations, unit tests, vector classes, matrix classes, function classes, and operator overloads, is available at *https://https://github.com/Teddyzander/NeuralNetwork*. Contact edward.small-2@postgrad.manchester.ac.uk for access.

Contents

1 Introduction to Machine Learning

Neural Networks (NNs) are a subset of *Machine learning* (ML), so before we discuss NNs (the main motivation for this project), it is important to understand the basic concept of ML.

1.0.1 Machine Learning

The concept of modern ML comes as an answer to a very simple question:

What if my code could learn?

Whilst the true inspiration for creating dynamic code that learns could be attributed to Alan Turing from his famous paper *Computing Machinery and Intelligence*[2], the father of ML is Arthur Samuel, a computer scientist at IBM, who defined ML as *"[A] Field of study that gives computers the ability to learn without being explicitly programmed."*[1].

To put it simply, ML is the science (and, in some ways, art) of allowing a computer to refine and alter a model or algorithm based on previous experience. How this happens is very dependent on the problem that is being solved and how the system is designed, but the fundamental idea remains the same.

ML can, if utilised correctly, solve a vast range of problems. Sometimes we may need to find patterns in data sets so large with so many dimensions, that for a human to do so would be impossible. Sometimes we may want to make predictions for outcomes of an event, but have no idea how input data effect the probability of a certain outcome[3].

1.1 Types of Machine Learning

ML comes in a huge variety of types, and as the field has grown a number of sub-types and combinations have emerged to solve unique problems. Despite this, most ML techniques can be split into one of three categories.

1.1.1 Supervised Learning

Supervised learning is used when a labeled data set is available. That is to say, we have a data set X which contains inputs $x \in \mathbb{R}^n$, and we have a corresponding data set Y which contains outputs $y \in \mathbb{R}^m$. In a labeled data set, we have knowledge of inputs into a system and the corresponding, correct outputs, such that

$$y = f(x), \text{ where } y \in Y \text{ and } x \in X \tag{1}$$

However, what is missing here is an understanding of how each x relates to each y, or $f : X \to Y$. This is where supervised learning steps in. Supervised learning

- Begins with some function g that attempts to replicate the behaviour of f
- Takes an input $x_i \in X$ and applies $g(x_i)$

- Creates an output $\hat{y}_i = g(x_i)$
- Uses the correct output $y_i \in Y$ and the estimated output $\hat{y}_i$ to create some error value
- Uses this error value to make adaptions to the function g
- Repeat this process until either g is replicating f (to some acceptable tolerance), or too many iterations have passed

An example of this type of learning will make up the main body of this paper.

1.1.2 Unsupervised Learning

Whilst for *supervised learning* we have access to a labeled data set, in unsupervised learning we only possess the inputs for the system, and not the corresponding outputs. The goal here, therefore, is to find some kind of structure, grouping, or regularity to the input data. From there, a probability density function can be created using probability density estimation[4] .

A good application of unsupervised learning is *dimension reduction*, or complexity reduction. As an example, let's say we are trying to figure out the probability that a person will be diagnosed with a certain cancer later in life. The data set X could contain information that is relevant (eg family history, smoker, certain biomarkers), information that is irrelevant (eg education, marital status), and information that could be relevant (eg location, diet). Not all this data is useful, and so unsupervised learning can sift through all this data and pick out the parts that are the most useful for determining a solution. This is particularly well utilised in areas such as signal processing and bioinformatics where X can be colossal and the data in X isn't always entirely well understood.

1.1.3 Reinforcement Learning

Reinforcement learning learns through trial and error using a penalty/reward system. So, given a certain goal, can a system learn what actions to take to maximise rewards. Performing actions that bring the system closer to the goal earn more rewards, and actions that take the system further away from the goal earn penalties.

A fun example of this is a computer learning to play Super Mario World for the SNES[5]. The reward for this system was progressing through the level (getting as far right as possible). The computer could interact with it's environment by moving left or right, jumping, and ducking (there are more controls available, but this is all that is needed to beat the first level).

2 Artificial Neural Networks

Artificial Neural Networks (ANN) are loosely modelled on biological neural systems. Specifically, they look to simulate the electrical activity of the brain by creating a system of processing elements (neurons) that all connect with each other and activate each other, depending on the output of a neuron[6].

2.1 Neuron (Perceptron)

To build an understanding of how an ANN works, it is first important to understand how a single neuron, also known as a perceptron, works. A neuron's job is quite simple - it has to decide, from inputs, whether it is active or not. It does this by "studying" the inputs, applying a function to the collection of inputs, and giving out a scalar output that tells the system whether or not it was activated.

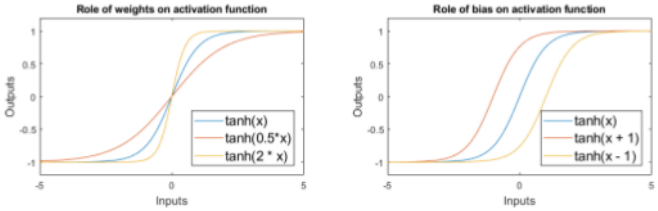
Take a set of inputs $x \in \mathbb{R}^n$. Each input x_i has it's own "weight" w_i, which is a value that justifies how useful each input is in determining whether or not to activate the neuron. The more extreme

In LaTeX, use backtick ` for open quotes, ' for close quotes.

the weight, the more important this input is. A set of weights $w \in \mathbb{R}^n$ is applied to each value in x, and then summed. A bias b is then applied to the summation, and the result is put through an *activation function* $\sigma : x \to [-1, 1]$ (or $[0, 1]$, depending on which activation function you choose) giving an output, or activation, a[8].

$$a = \sigma(\sum_{i=1}^{n}(w_i x_i) + b) \tag{2}$$

It can be difficult, from the equation, to see what role weights and biases truly play here, but graphing some simple examples can give a good explanation.



Good

Figure 1: How different weights and biases affect the activation function

As can be seen in figure 1, weights alter the steepness of the activation function, and biases shift the activation function.

2.2 The Network

A neural network, therefore, is just a system where the output of each neuron feeds into other neurons, eventually giving a final output. The Neurons are typically arranged in L number of layers, with each layer being represented by $l = 1, 2, ..., L$. $l = 1$ represents the input layer, and $l = L$ represents the output layer. The layers between the input and output layer are often called *hidden layers* because they are hidden from the user. Each layer contains $n^{[l]}$ number of neurons.

2

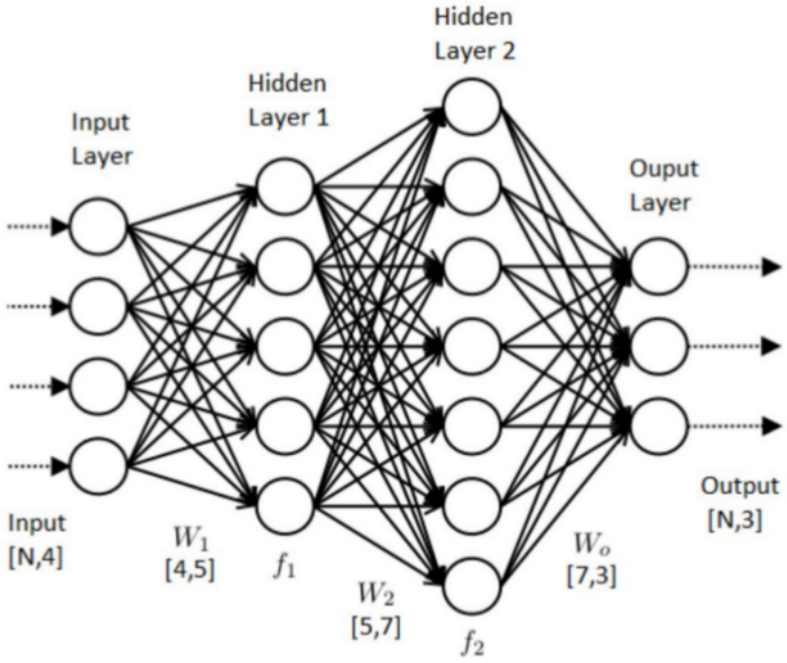


Figure 2: An visual representation of a neural network[7]

In figure 2 each circle in the hidden layer represents a neuron. Each hidden layer l has $n^{[l]}$ number of neurons and $n^{[l]}$ number of biases (a bias for each neuron). Every single arrow has a weight $w_{ij}^{[l]}$ associated with it, so each layer l has a weight matrix $W^{[l]} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}$. This is because for every neuron in one layer $l-1$ to connect to every neuron in the next layer l there must be $n^{[l]}n^{[l-1]}$ connections. It's important here to recognise the linear algebra, which builds from the simple perceptron mentioned earlier, as it means we can tackle things layer by layer, rather than neuron by neuron

$$\begin{bmatrix} a_1^{[l]} \\ a_2^{[l]} \\ \vdots \\ a_{n^{[l]}}^{[l]} \end{bmatrix} = \sigma\Bigg(\begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,n^{[l-1]}} \\ w_{2,1} & w_{2,2} & \dots & w_{2,n^{[l-1]}} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n^{[l]},1} & w_{n^{[l]},2} & \dots & w_{n^{[l]},n^{[l-1]}} \end{bmatrix} \begin{bmatrix} a_1^{[l-1]} \\ a_2^{[l-1]} \\ \vdots \\ a_{n^{[l-1]}}^{[l-1]} \end{bmatrix} + \begin{bmatrix} b_1^{[l]} \\ b_2^{[l]} \\ \vdots \\ b_{n^{[l]}}^{[l]} \end{bmatrix} \Bigg) \tag{3}$$

which can be written as

$$a^{[l]} = \sigma(W^{[l]}a^{[l-1]} + b^{[l]}) = \sigma(z^{[l]}) \tag{4}$$

This is exactly the same as equation 2, except now we are applying it to all neurons in layer $l-1$ to find the activations for the next layer. So, activations in one layer $a^{[l-1]}$ determine the activations in the next layer $a^{[l]}$. And this effect dominoes through the network, all the way to the final layer where we receive the output $\hat{y} = a^{[L]}$.

2.2.1 Over-fitting

Over-fitting occurs when the trained neural network performs really well on the training data set, but doesn't truly capture the nature of the function it is trying to replicate. A validation data set is

another, separate labeled data set that isn't used to train the neural network, but rather to check how well it performs. We would expect a trained neural network to perform very well on data it hasn't seen before, but sometimes this isn't the case - slight deviations from the training data cause incorrect outputs, so the network is over-fitted.

This is quite a common issue, and there are a few ways one could go about reducing the chance of over-fitting:

- Add more data to the training data set, if available
- Reduce the complexity of the network (removing neurons or even entire layers)
- Manipulate your training data to artificially increase the training data set. This obviously has to be done carefully, but one example could be that if your input is images, you can rotate/mirror the image without changing with the result should be

2.3 Types of Neural Network

Feedforward networks are fundamentally what has been described above, and is the type of network that will be used for this report. However, these are not the only types of neural networks. In fact, there are many types, and each have their own somewhat special use cases. That is not to say one type is necessarily better than another - and often more than one type of network can solve a particular problem - but they do offer their own advantages and disadvantages.

2.3.1 Feedforward Neural Network

The term *feedforward* described how the data moves through the network. In a feedforward neural network, data passes through a neuron in one layer to the neurons in another, but is not passed back. Data can only be passed from input neurons, to the hidden layers, to the output, with no loops or cycles. These kinds of ANNs are useful for when historical data/past outputs have no impact or influence over future outputs of the ANN.

Over the last 30 or so years, a lot of work has been done to prove that every continuous function that maps a set of real numbers to another set of real numbers ($f : \mathbb{R}^n \to \mathbb{R}^m$) can be approximated arbitrarily closely using ANN. This is called *universal approximation theorem*[9][11][12].

2.3.2 Recurrent Neural Network

Recurrent Neural Network (RNN) differ from feedforward networks in that they allow previous outputs from nodes to be used as inputs. RNNs, therefore, have some knowledge of the state of the system and time, containing previous outputs to assist in predicting/classifying the next output, unlike in feedforward where previous results have no impact on future results[13].

This makes them particularly useful for problems such as speech or handwriting recognition. As an example, when recognising a handwritten word on a page we can increase the probability of recognising the word correctly not only by considering each letter individually, but also the letters that have come before the current letter being analysed. Eg, two consecutive U's very rarely occur in the English language. So, if the previous letter in an English word was detected to be a U then it is highly unlikely that the current letter being analysed will also be a U.

2.3.3 Fully Connected Neural Network

Again, the ANN studied in section 2.2 is a fully connected neural network, as every neuron in a layer connects to every neuron in the previous and next layer. These types of networks are a sort of "one size fits all" approach, and are architecturally quite simple as they assume no knowledge of the function they are trying to simulate going in.

2.3.4 Convolutional Neural Network

In a *convolutional neural network* (CNN) there is a sense of "locality" about the network. What is meant by that is that not every neuron in one layer will necessarily be connected to every neuron in the next layer because there is some understanding when designing the network that not every neuron should influence every neuron in the next layer. An example of where this could be applicable is feature or object recognition in an image. If an image consists of 2 objects, the network does not need knowledge of the pixels contained in and near object A to determine what object B is (and vice versa).

But why bother designing removing connects and designing a network that is structurally more complex? Well, because of the design and iterative nature of ANNs, they are computationally expensive. Every connection removed is a computation removed, many removals over many iterations can have a significant impact on performance. To design a CNN well requires good knowledge of the input data and how it may relate to itself.

3 The Learning Process

The previous section mainly talked about how to use an ANN, but only touched briefly on how one may learn. As mentioned in section 1.1.1, ANNs learn using training data sets which are labeled data sets contain inputs X and matching outputs Y. A data $x \in X$ is put through the ANN and an estimate $\hat{y}$ is given out. We can then compare $\hat{y}$ to the corresponding $y \in Y$ and find how wrong the network is. Tweaks can then be made to the network, which can now be seen as an adjustment to the weights and biases (which are initialised somewhat arbitrarily), and then the network can try again. But how do we know what to change, and by how much?

3.1 Cost Function

During the training phase, a cost function is used to determine how bad a network is, or how different $\hat{y}$ is from y. The cost function for an input x is defined as

$$C(x) = \frac{1}{2}\sum_{i=1}^{n^{[L]}}((\hat{y}_i - y_i)^2) \tag{5}$$

or the sum of the square of the differences between the actual value and the ANN output value. The sum is expected to be small when the ANN output is very close to the true value, and large when the ANN's answer is far from the true value.

However, this does not tell the whole story. The ANN could perform very well for a single set of inputs, giving a small value for $C(x)$. What we are really interested in is the *total cost* of the network, $C_{tot}(X)$, or the average cost of the entire training data set. For this, we require

$$C_{tot}(X) = \frac{1}{N}\sum_{i=1}^{N} C(x_i) \tag{6}$$

This gives a sense of how well the ANN is performing for the whole training set. These algorithms are implemented on line 594 (Cost) and line 614 (TotalCost) in ANN code in the appendix. Because it is run over the entire training set, it can be computationally expensive (depending on the size of the training set). The aim of the ANN, therefore, is to minimise the value of $C_{tot}(X)$. It is worth noting here that your starting point for C_{tot} will affect your ending point, as only a local minimum can be found. It should become clear why in the next section.

3.2 Stochastic Gradient Descent

The goal of the ANN is to minimise $C_{tot}(X)$ as quickly as possible. The easiest way to do this is find the derivative, $\nabla C_{tot}(X)$, or the gradient of the cost function for the current network. When $\nabla C_{tot}(X) = 0$ a local minimum has been found (bearing in mind that $C_{tot}(X)$ is dependent on all weights and biases (since that is how each $\hat{y}_i$ is calculated) of the network). A network with improved weights and biases p_i is made by taking the gradient of the cost function for the current network, multiplied by a training parameter η.

$$p_i = p_{i-1} - \eta \nabla C_{tot}(p_{i-1}) \tag{7}$$

Where ∇ is the gradient operator acting over ALL the weights and biases over the network. The reason for this is because what we are fundamentally trying to find out is how sensitive the cost is to each weight and bias in the network (look at the *chain rule*). The training parameter is usually very small, and is used to prevent over shooting (though making it too small means many steps are needed, so a balance needs to be struck). The "stochastic" part of the gradient descent comes from the fact that, instead of analysing $C_{tot}(p_{i-1})$ over the entire training set, we simple pick a random data subset. This reduces the computational cost massively, meaning that this step can be run far more times, but still gives a result that is probably close to the true value.

This algorithm is applied in line 571 (UpdateWeightsAndBiases) in ANN code in the appendix.

4 Applied Example of an ANN

The main function (line 1238) in the code supplied in the appendix creates a simple ANN. It contains 2 input neurons ($x = (x_1, x_2)$, where $x_i \in [0, 1]$), 2 hidden layers (each with 3 neurons) and single neuron in the output layer. This can be defined as a *binary classifier*, as it defines the output $a^{[L]} = \hat{y}$ as 1 or -1 (perhaps can be seen as success or failure).

Using the training data supplied in GetTestData (line 210), the following function $F(x)$ is generated using the ANN.

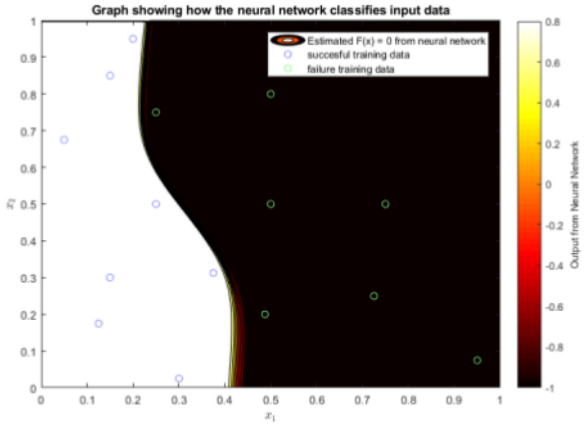


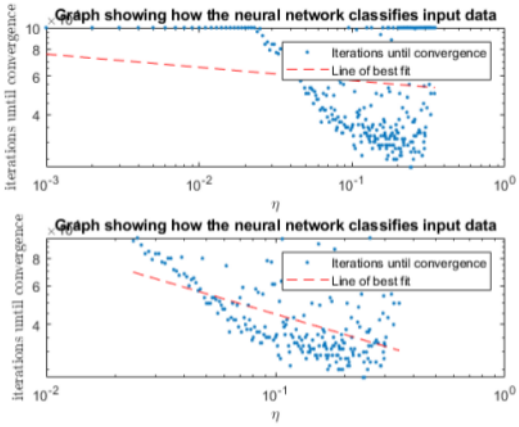
Figure 3: The ANN divides the plane into two sections, with the white area representing $F(x) \approx 1$ and the black area representing $F(x) \approx -1$

For this ANN, any input data (x_1, x_2) that falls to the left of the black line will be classified as a success, and any that falls to the right will be classified as a failure.

4.1 The role of η on convergence

The role of η or the *learning rate* is crucial in the learning stages for a neural network. Since it is applied to the gradient of the cost function (as shown in equation (7)), it has a huge impact on the speed of convergence as it dictates how large of a change can be made to the network in a single iteration.

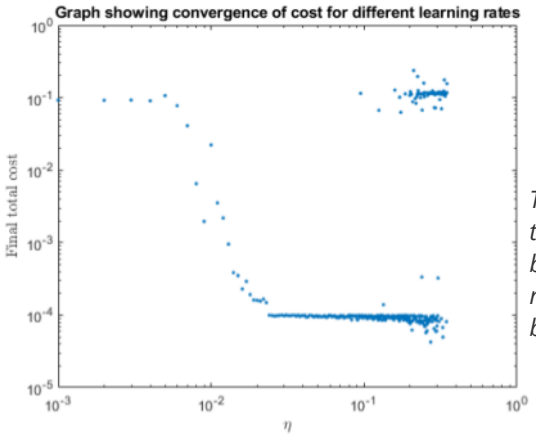
If η is too small, the changes made to the network each step will be tiny, meaning it could take a huge number of iterations to meet the required tolerance (which is a huge problem for very complex networks, where one iteration in the training period is already computationally taxing). However, if it is too large, the step will continually over shoot the minimum that it is trying to be find, and never converge at all. This can be shown analytically by playing around with η for the network described above.



Good. I like the log axes, but do you think a line of best fit is appropriate for these data?

Figure 4: Top graph has all η sampled from $[0, 0.35]$. The bottom graph has failed iterations removed, to better show the line of best fit

As figure 4 shows, for $\eta < 0.02$ the network fails to converge in the number of iterations we have capped it at - 100000. These probably would have converged eventually, but clearly this inefficient. Likewise, for $\eta > 0.35$ the network also always failed to converge (which is why this data is excluded). In fact, as η grows beyond around 0.2, the likelihood of it failing to converge begins to increase, as shown by the grouping in the top graph. So there is a sweet spot somewhere around $0.02 < \eta < 0.2$ for this ANN specifically. This shows that our choice of η is not arbitrary, and is actually an important step for creating a well trained network. This can also be shown by analysing the final cost of the network.



The results here are rather dominated by the threshold of 10^-4 -- it might have been better to show cost after some fixed number of iterations, with no threshold-based exit from the loop.

Figure 5: loglog graph showing final total cost against different values for η

As show in figure 5, the C_{tot} tends to stay around 0.1 for the bad values of η mentioned above. The "flat line" grouping is caused by the fact that the tolerance (meaning acceptable C_{tot}) is set to 0.001. In fact, convergence was almost guaranteed for $0.011 < \eta < 0.11$.

4.2 The role of weights and biases on convergence

The learning rate isn't the only thing that effects how quickly an ANN might learn, the initialised values of weights and biases does too. To analyse this, we need to examine the role that the derivative of the activation function plays in the learning process, and how this works with back-propogation.

For a the j^{th} neuron in layer l, the error $\delta_j^{[l]}$ is defined as

$$\delta_j^{[l]} = \frac{\partial C_{tot}(x)}{\partial z_j^{[l]}} \tag{8}$$

To shift the weights and biases, we must find the error in the final layer L and then use this error to calculate errors in previous layers, excluding $l = 1$.

$$\delta^{[L]} = \sigma'(z^{[L]}) \circ (a^{[L]} - y) \tag{9}$$

And then propagate this error back through the system with

$$\delta^{[l]} = \sigma'(z^{[l]}) \circ (W^{[l+1]})^T \delta^{[l+1]} \tag{10}$$

This is then used to update the weights and biases for the next iteration i of the network, such that

$$b_{i+1}^{[l]} = b_i^l - \eta \delta^{[l]} \tag{11}$$

$$W_{i+1}^{[l]} = W_i^{[l]} - \eta(\delta^{[l]} \otimes a_i^{[l-1]}) \tag{12}$$

Since $z^{[l]}$ is already reliant on weights and biases (from equation (4)), it's pretty clear that errors for each layer are massively reliant on W and b. To understand how the error effects the learning rate, we can look at the graph for $\sigma'(x)$.

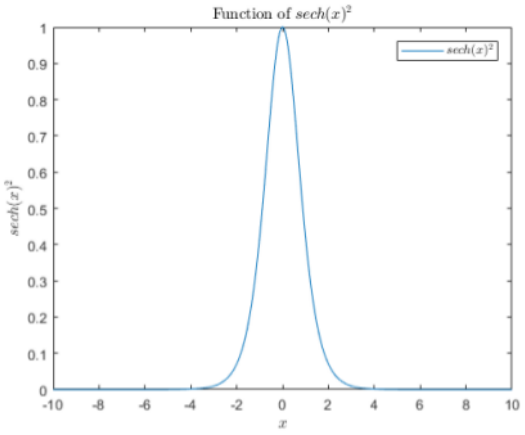


Figure 6: Figure showing that extreme inputs for the derivative of the activation function yields small values

Extreme values for $\sigma'(x)$ yield very small values. In fact, if $x \notin [-4, 4]$, $\sigma'(x) \approx 0$. Very small values from $\sigma'(x)$ means very small values for δ which means that each iteration yields an incredibly small training effect, which will, in turn, increase the amount of time it takes to get to convergence. Not only that, but small *delta* also implies large W and/or b, meaning that the ratio for change in the system is massively imbalanced. This means that a large standard deviation for values in W and b is not desirable.

A large standard deviation also means a large spread of values for the weights and biases, meaning that some are initialised as much more dominant than others in the system. Trying some different standard deviations for the example ANN shows the likelihood of convergence occurring shrinks as it grows beyond around 2.

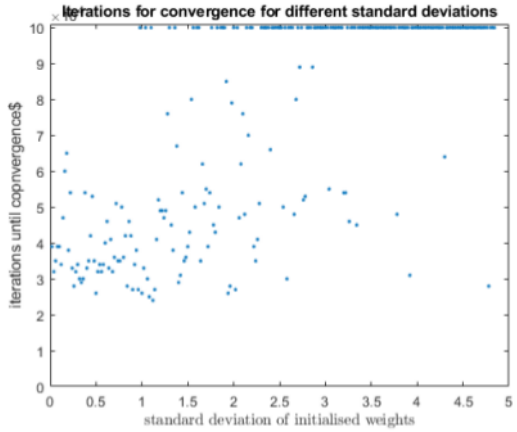


Figure 7: As standard deviation of the initialised weights grows beyond a certain threshold, probability of the total cost being below the tolerance decreases

4.3 The role of ANN structure on function shape

Over-fitting is a term that has already been mentioned, and it is the structure of the ANN, namely the number of layers and the number if neurons in each layer, that can cause this. The optimal number of neurons in an ANN is dependent on the complexity of the function it is trying to replicate. More neurons are often needed for more complex problems.

Too few neurons in the ANN will cause under-fitting (imagine using a linear function to fit data from a polynomial), and too many neurons will cause over-fitting, where the network maps itself onto the training data too well, instead of creating a general function to classify new data[10].

This therefore means that there is a balance to be struck. Replicating complex functions requires more neurons, but not only does this risk over-fitting the training data, this also increases computation time for both the learning stage and utilisation stage.

Is over-fitting relevant to this problem in this report, where there is no distinction between training and validation data?

4.3.1 Example

To show this in action, two complex training datasets were used to train a variety of neural networks, ranging from no hidden layers, all the way to 3 hidden layers, each with 20 neurons. Each was initialised with $\eta = 0.005$, standard deviation of 0.1 for weights and biases, an acceptable tolerance of 0.01, and each was allowed 3 million iterations. One data set created a "checkered" pattern (success and failures were grouped like a chess board), and the other created two outward spiraling sets of success and failures. Clearly, creating a function to split the success and failures up is somewhat complicated for a network to learn, at least more so than the simpler data set in the example before.

The results for various ANN designs are in appendix A, split up by data set, layer number, and number of neurons in each layer. By looking at each figure, you can see how the number of neurons and layers effects how accurately the ANN can create the mapping. Increasing the number of neurons and layers does increase the networks ability to learn. There is, however, a large caveat, and that is computation time. 3 million iterations for one set of 5 trainable neurons takes minutes to complete. For 3 layers of 20 neurons, it can take half an hour, and there is no guarantee that the network it creates is at all usable.

Some machine learning experts simply go by intuition, experience, and experimentation when designing a network. However, one school of thought is to study your data, and look at "lines" and "connections" needed to separate the data correctly. The first hidden layer should contain n neurons equal to the minimum number of straight line segments needed to separate the data well. Hidden layer 2 should contain n number of connections between this lines, etc. Whilst this may create a network more complex than the minimum complexity, it seems to at least guarantee a usable network after the learning phase, and it converges in few iterations.

Good, but would benefit from a citation

5 Testing

Each function inside the ANN code was tested and validated in the *Test* section (line 654). The general mantra of testing was

- Split the neural network into separate functions
- Test each function in isolation with a very simple network
- Test each function in isolation with a more complex network, checking (to a tolerance) results against hand calculated values
- If a function fails, tell the user exactly which test failed

Testing is good - but a little more text here would have been beneficial.

Conclusion?

References

[1] Jason Bell *Machine Learning: Hands-On for Developers and Technical Professionals*. Pages 1-16 John Wiley and Sons, 3 November 2014.

[2] A M Turing *Computing Machinery and Intelligence*, Mind: Volume LIX, Pages 433-460. Oxford Academic, 01 October 1950.

[3] Ethem Alpaydin. Francis Bach *Introduction to Machine Learning*. MIT Press, 22 August 2014.

[4] Ilya Narsky. Frank C. Porter *Statistical Analysis Techniques in Particle Physics : Fits, Density Estimation and Supervised Learning*. John Wiley and Sons, 23 December 2013.

[5] Seth Bling *MarI/O - Machine Learning for Video Games*.

shorturl.at/egmDF 13 June 2015

[6] Ilya Narsky. Frank C. Porter *Encyclopedia of Physical Science and Technology (Third Edition)*. Pages 631-645 Science Direct, 2003.

[7] Jayesh Bapu Ahire *The Artificial Neural Networks handbook: Part 1* Medium, 24 August 2018.

[8] M. N. Murty. Rashmi Raghava *Support Vector Machines and Perceptrons* . Pages 27-40 Springer Link, 17 August 2016.

[9] G. Cybenko *Approximation by Superpositions of a Sigmoidal Function*. Springer-Verlag New York, 1989.

[10] Ian Goodfellow. Yoshua Bengio. Aaron Courville *Deep Learning*. Section 5.2 Goodfellow, 2016.

[11] Kurt Hornik *Neural Networks: Approximation capabilities of multilayer feedforward networks*. Pages 251-257 Science Direct, 1991.

[12] Anastasis Kratsios *The Universal Approximation Property*. Cornell University, 28 November 2020.

[13] Alex Graves *Supervised Sequence Labelling with Recurrent Neural Networks*. Introduction Springer Link, 2012.

A Results for ANN Structure for Complex Data

A.1 Checkerboard Data

A.1.1 Data set

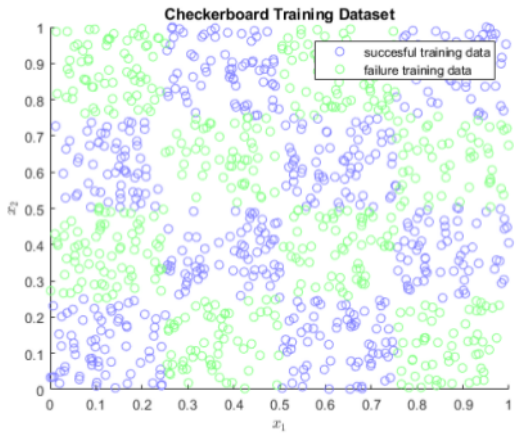


Figure 8: Training Checkerboard data set

A.1.2 No Hidden Layers

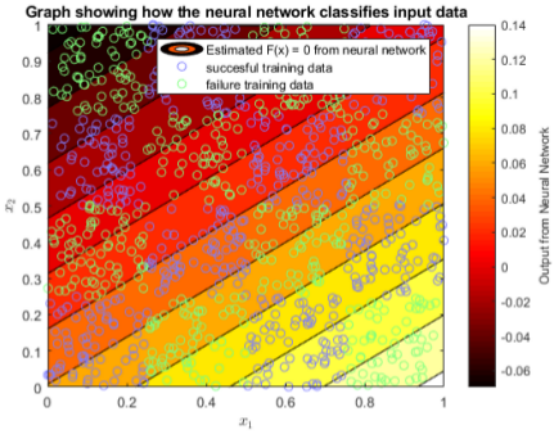


Figure 9: 2 1 structure fails to capture the likeness of the data (did not converge, total cost stayed around 0.5)

A.1.3 1 Hidden Layer

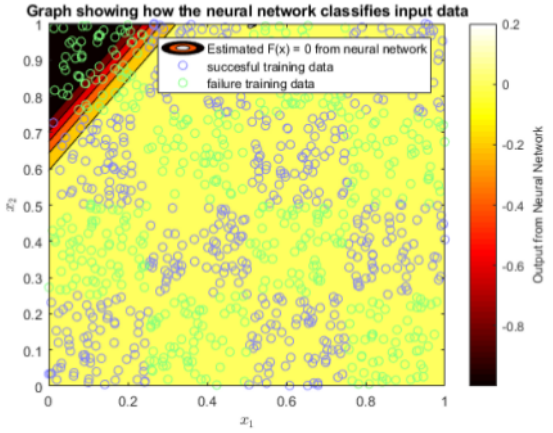


Figure 10: 2-5-1 structure fails to capture the likeness of the data (did not converge, total cost stayed around 0.5)

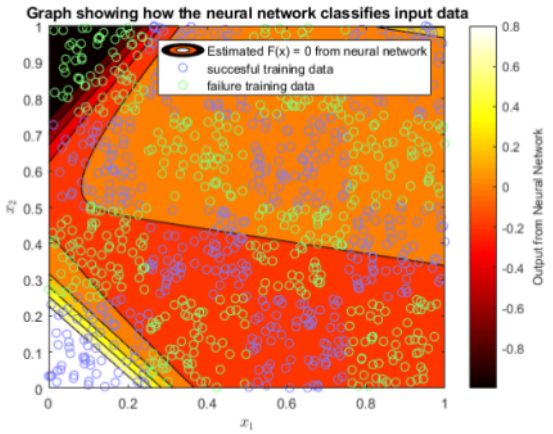


Figure 11: 2-10-1 structure fails to capture the likeness of the data, but did begin to recognise some areas eg bottom left corner and top left. (did not converge, total cost stayed around 0.4)

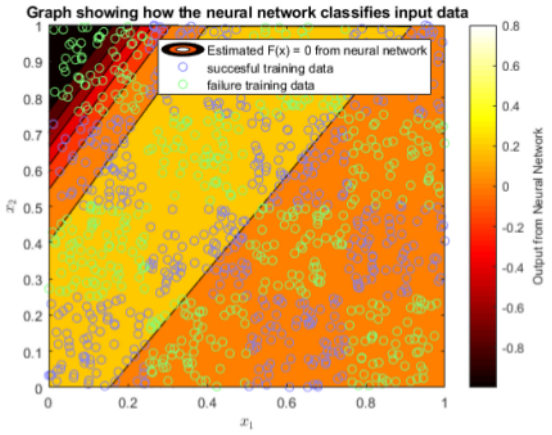


Figure 12: 2-15-1 structure fails to capture the likeness of the data (did not converge, total cost stayed around 0.5)

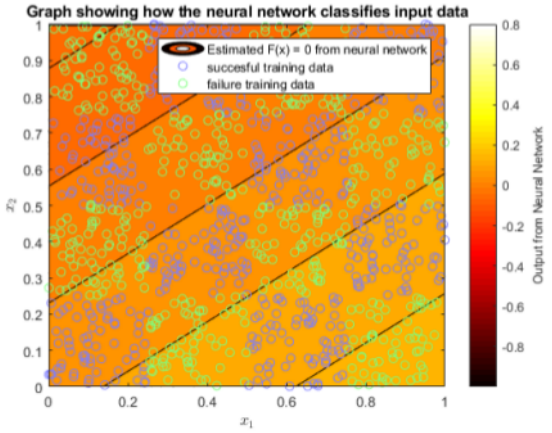


Figure 13: 2-20-1 structure fails to capture the likeness of the data (did not converge, total cost stayed around 0.5)

A.1.4 2 Hidden Layers

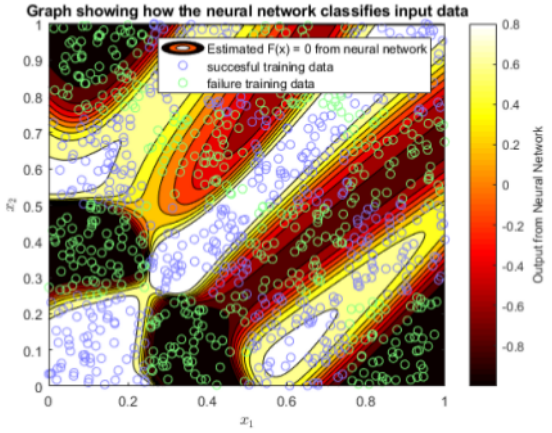


Figure 14: 2-5-5-1 shows some promise after max iterations, and is quite certain about some areas correctly (the white and black areas) (did not converge, total cost stayed around 0.2)

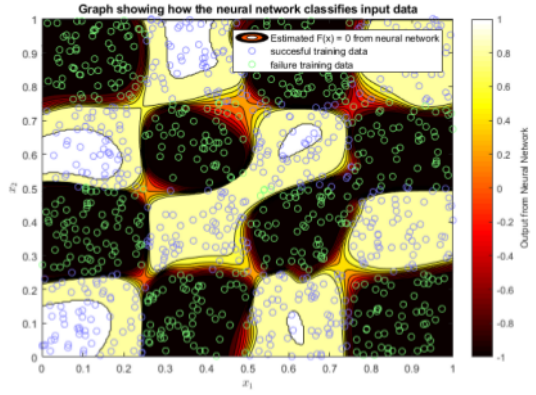


Figure 15: 2-10-10-1 shows an even better performance than the previous network. The different areas are starting to look a little more "boxy", as we'd expect. (did not converge, total cost stayed around 0.08)

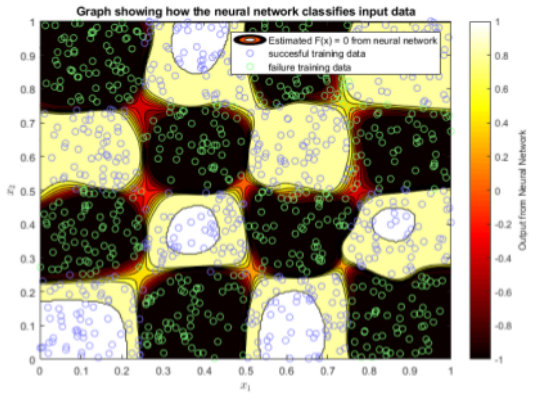


Figure 16: 2-15-15-1 shows some improvement over 2-10-10-1. The white and black areas are far more separate (did not converge, total cost stayed around 0.05)

A.1.5 3 Hidden Layers

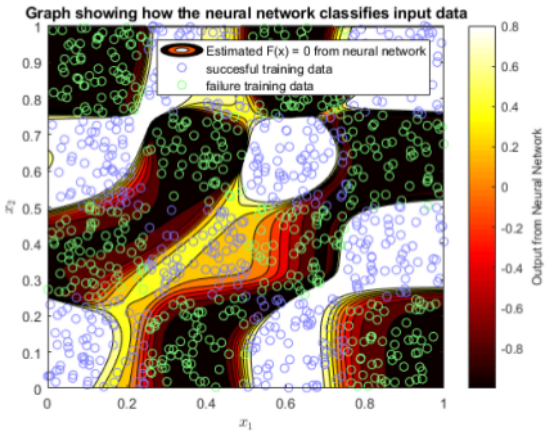


Figure 17: 2-5-5-5-1 showed some ability to learn the checker patter, but ultimately didn't capture the true function (did not converge, total cost stayed around 0.2)

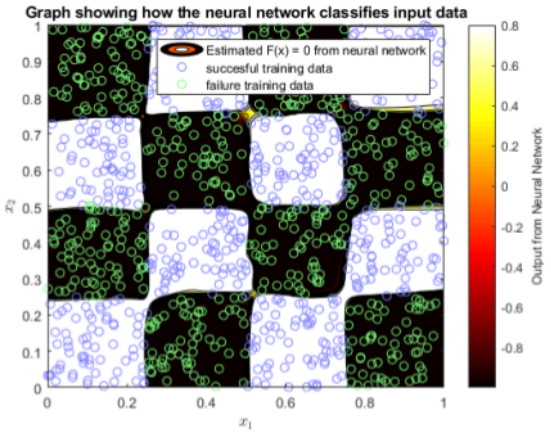


Figure 18: 2-10-10-10-1 displayed a great ability to replicate the checkerboard pattern, with near certainty in most areas (did not converge, total cost stayed around 0.02)

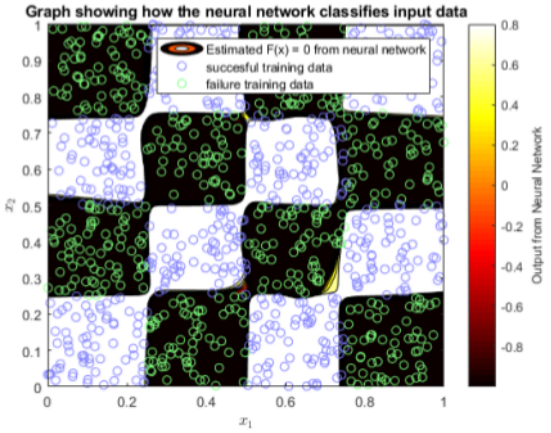


Figure 19: 2-20-20-20-1 also captured the function likeness well. The cost was lower than for 2-10-10-10-1, though it is difficult to see how much better it truly is. It took significantly longer to learn, due to the extra neurons (did not converge, total cost stayed around 0.013)

A.2 Spiral Data

A.2.1 Data set

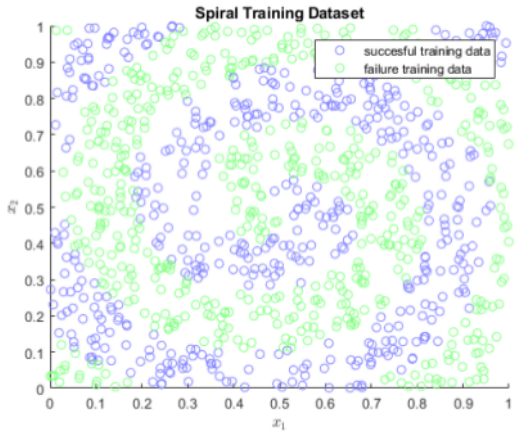


Figure 20: Training Spiral data set

A.2.2 No Hidden Layers

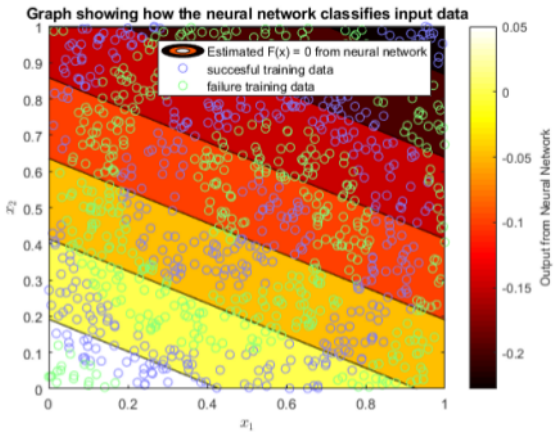


Figure 21: 2-1 Showed no ability to learn the spiral data set (total cost stayed around 0.5)

A.2.3 1 Hidden Layer

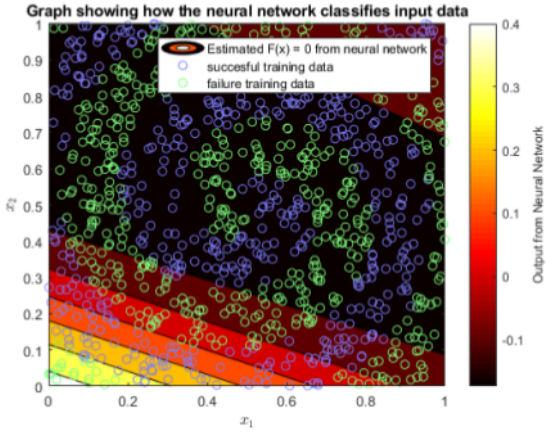


Figure 22: 2-5-1 Showed no ability to learn the spiral data set (total cost stayed around 0.5)

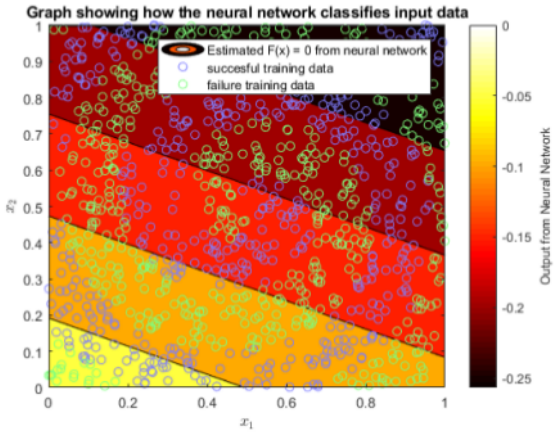


Figure 23: 2-10-1 Showed no ability to learn the spiral data set (total cost stayed around 0.5)

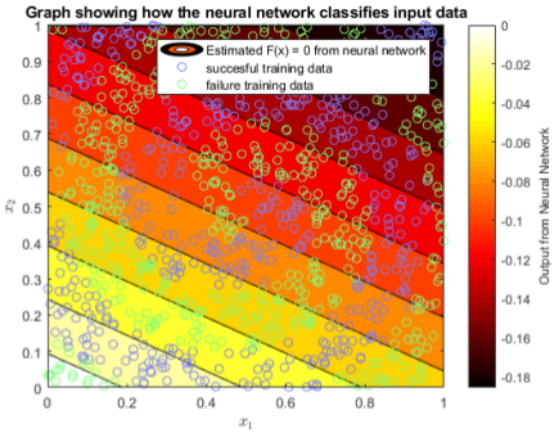


Figure 24: 2-15-1 Showed no ability to learn the spiral data set (total cost stayed around 0.5)

A.2.4 2 Hidden Layers

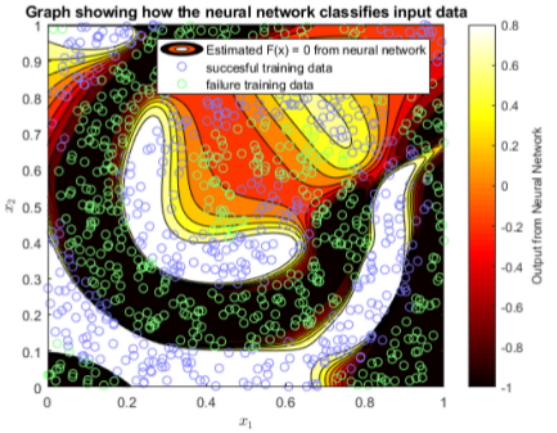


Figure 25: 2-5-5-1 Showed some progress. Surprising for such a simple network to see some areas of high certainty being correct (total cost stayed around 0.25)

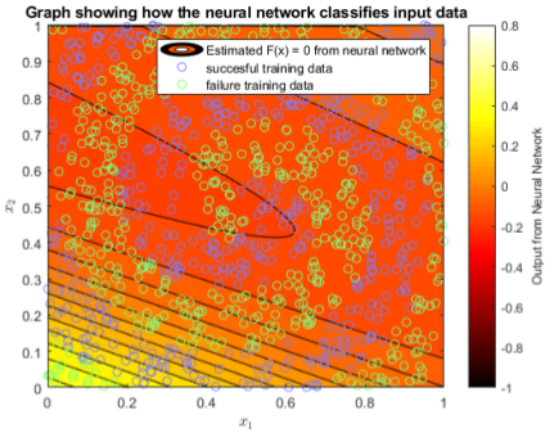


Figure 26: 2-10-10-1 Showed no ability to learn the spiral data set (total cost stayed around 0.5)

A.2.5 3 Hidden Layers

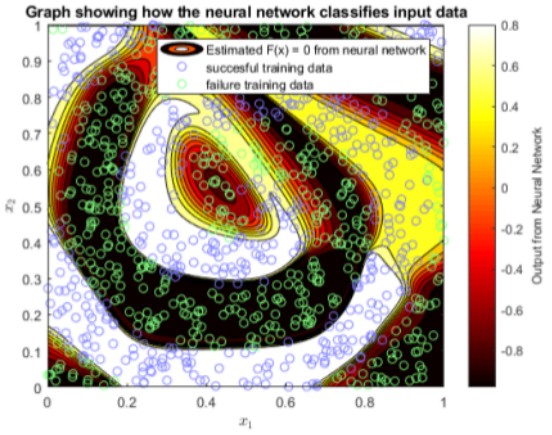


Figure 27: 2-5-5-5-1 improved again on 2-5-5-1 (total cost stayed around 0.19)

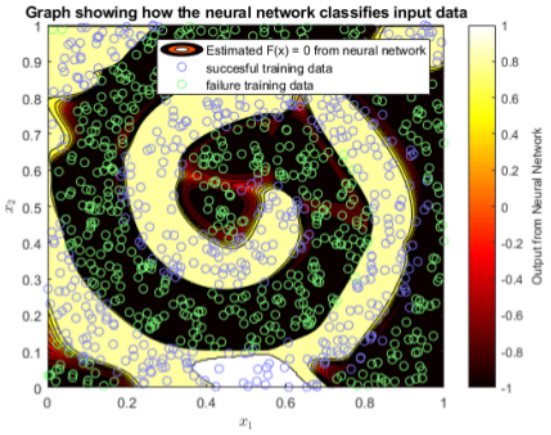


Figure 28: 2-10-10-10-1 also showed a very good fit, clearly capturing the general shape with good precision (total cost stayed around 0.09)

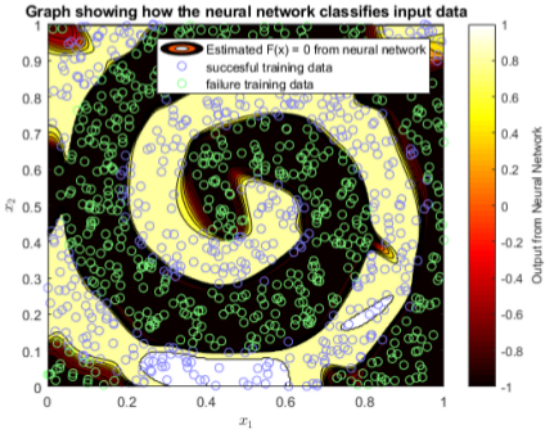


Figure 29: 2-15-15-15-1 had the best fit out of all the network structures I tried, even getting all the corners correct (total cost stayed around 0.06)

Good checkerboard and spiral data graphs, but I would like to have seen these discussed more in the body of the report.

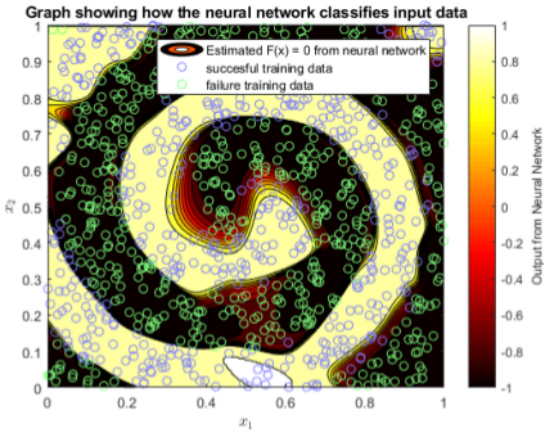


Figure 30: 2-20-20-20-1 was marginally worse than the previous network, but still had a very good fit (total cost stayed around 0.08)

B Artificial Neural Network Code

```
1
2  #include "mvector.h"
3  #include "mmatrix.h"
4
5  #include <cmath>
6  #include <random>
7  #include <iostream>
8  #include <fstream>
9  #include <cassert>
10 #include <string>
11
12
13 ////////////////////////////////////////////////////////////////////////////////
14 // Set up random number generation
15
16 // Set up a "random device" that generates a new random number each time the program
       is run
17 std::random_device rand_dev;
18
19 // Set up a pseudo-random number generater "rnd", seeded with a random number
20 std::mt19937 rnd(rand_dev());
21
22 // Alternative: set up the generator with an arbitrary constant integer. This can be
       useful for
23 // debugging because the program produces the same sequence of random numbers each
       time it runs.
24 // To get this behaviour, uncomment the line below and comment the declaration of "rnd
       " above.
25 //std::mt19937 rnd(12345);
26
27
28 ////////////////////////////////////////////////////////////////////////////////
29 // Some operator overloads to allow arithmetic with MMatrix and MVector.
30 // These may be useful in helping write the equations for the neural network in
31 // vector form without having to loop over components manually.
32 //
33 // You may not need to use all of these; conversely, you may wish to add some
```

```
34 // more overloads.
35
36 // MMatrix * MVector
37 MVector operator*(const MMatrix &m, const MVector &v)
38 {
39   assert(m.Cols() == v.size());
40
41   MVector r(m.Rows());
42
43   for (int i=0; i<m.Rows(); i++)
44   {
45     for (int j=0; j<m.Cols(); j++)
46     {
47       r[i]+=m(i,j)*v[j];
48     }
49   }
50   return r;
51 }
52
53 // transpose(MMatrix) * MVector
54 MVector TransposeTimes(const MMatrix &m, const MVector &v)
55 {
56   assert(m.Rows() == v.size());
57
58   MVector r(m.Cols());
59
60   for (int i=0; i<m.Cols(); i++)
61   {
62     for (int j=0; j<m.Rows(); j++)
63     {
64       r[i]+=m(j,i)*v[j];
65     }
66   }
67   return r;
68 }
69
70 // MVector + MVector
71 MVector operator+(const MVector &lhs, const MVector &rhs)
72 {
73   assert(lhs.size() == rhs.size());
74
75   MVector r(lhs);
76   for (int i=0; i<lhs.size(); i++)
77     r[i] += rhs[i];
78
79   return r;
80 }
81
82 // MVector - MVector
83 MVector operator-(const MVector &lhs, const MVector &rhs)
84 {
85   assert(lhs.size() == rhs.size());
86
87   MVector r(lhs);
88   for (int i=0; i<lhs.size(); i++)
89     r[i] -= rhs[i];
90
91   return r;
92 }
93
94 // MMatrix = MVector <outer product> MVector
95 // M = a <outer product> b
96 MMatrix OuterProduct(const MVector &a, const MVector &b)
97 {
98   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;
}
```

```
164 // Output function for MVector
165 inline std::ostream &operator<<(std::ostream &os, const MVector &rhs)
166 {
167   std::size_t n = rhs.size();
168   os << "(";
169   for (std::size_t i=0; i<n; i++)
170   {
171     os << rhs[i];
172     if (i!=(n-1)) os << ", ";
173   }
174   os << ")";
175   return os;
176 }
177
178 // Output function for MMatrix
179 inline std::ostream &operator<<(std::ostream &os, const MMatrix &a)
180 {
181   int c = a.Cols(), r = a.Rows();
182   for (int i=0; i<r; i++)
183   {
184     os<<"(";
185     for (int j=0; j<c; j++)
186     {
187       os.width(10);
188       os << a(i,j);
189       os << ((j==c-1)?')':',');
190     }
191     os << "\n";
192   }
193   return os;
194 }
195
196 //struct for returning stuff needed for plots
197 struct NetworkData
198 {
199   bool success;
200   double eta;
201   int iterations;
202   double cost;
203 };
204
205
206 ////////////////////////////////////////////////////////////////////////////////
207 // Functions that provide sets of training data
208
209 // Generate 16 points of training data in the pattern illustrated in the project
      description
210 void GetTestData(std::vector<MVector> &x, std::vector<MVector> &y)
211 {
212   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},
213      {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}};
214
215   y = {{1},{1},{1},{1},{1},{1},{1},{1},
216      {-1},{-1},{-1},{-1},{-1},{-1},{-1},{-1}};
217 }
218
219 // Generate 1000 points of test data in a checkerboard pattern
220 void GetCheckerboardData(std::vector<MVector> &x, std::vector<MVector> &y)
221 {
222   std::mt19937 lr;
223   x = std::vector<MVector>(1000, MVector(2));
224   y = std::vector<MVector>(1000, MVector(1));
225
```

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


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

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

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

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

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

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

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

    // Output components of x[i]
    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 layer.
  // For example:
  //   Network({2,1}) has two input neurons, no hidden layers, one output neuron
  //
  //   Network({2,3,3,1}) has two input neurons, two hidden layers of three neurons
  //                      each, and one output neuron
  Network(std::vector<unsigned> nneurons_)
  {
    nneurons = nneurons_;
    nLayers = nneurons.size();
    weights = std::vector<MMatrix>(nLayers);
    biases = std::vector<MVector>(nLayers);
    errors = std::vector<MVector>(nLayers);
    activations = std::vector<MVector>(nLayers);
    inputs = std::vector<MVector>(nLayers);
    // Create activations vector for input layer 0
    activations[0] = MVector(nneurons[0]);

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

    // The correspondence between these member variables and
    // the LaTeX notation used in the 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 indices in C++
    // code are one less than the indices used in the mathematics (which run from 1 to
     N)
  }

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

```
354   }
355
356   // Return the number of output neurons
357   unsigned NOutputNeurons() const
358   {
359     return nneurons[nLayers-1];
360   }
361
362   // Evaluate the network for an input x and return the activations of the output
        layer
363   MVector Evaluate(const MVector &x)
364   {
365     // Call FeedForward(x) to evaluate the network for an input vector x
366     FeedForward(x);
367
368     // Return the activations of the output layer
369     return activations[nLayers-1];
370   }
371
372
373   // Implement the training algorithm outlined in section 1.3.3
374   // This should be implemented by calling the appropriate private member functions,
        below
375   NetworkData Train(const std::vector<MVector> x, const std::vector<MVector> y,
376         double initsd, double learningRate, double costThreshold, int maxIterations)
377   {
378     // Check that there are the same number of training data inputs as outputs
379     assert(x.size() == y.size());
380
381     //initialise the weights and biases with the standard deviation "initsd"
382     InitialiseWeightsAndBiases(initsd);
383
384     for (int iter=1; iter<=maxIterations; iter++)
385     {
386       // Step 3: Choose a random training data point i in {0, 1, 2, ..., N}
387       int i = rnd()%x.size();
388
389       // Step 4: run the feed-forward algorithm
390       FeedForward(x[i]);
391
392       //Step 5: run the back-propagation algorithm
393       BackPropagateError(y[i]);
394
395       // Step 6: update the weights and biases using stochastic gradient
396       //                 with learning rate "learningRate"
397       UpdateWeightsAndBiases(learningRate);
398
399       // Every so often, perform step 7 and show an update on how the cost function
      has decreased
400       // Here, "every so often" means once every 1000 iterations, and also at the last
       iteration
401       if ((!(iter%1000)) || iter==maxIterations)
402       {
403         // Step 7(a): calculate the total cost
404         double total_cost = TotalCost(x, y);
405
406
407         // display the iteration number and total cost to the screen
408         std::cout << "Iteration: " << iter << "\tTotal Cost:" << total_cost << std::
      endl;
409
410         // Step 7(b): return from this method with a value of true,
411         //                    indicating success, if this cost is less than "
      costThreshold".
412
```

```
413         if (total_cost < costThreshold)
414         {
415           return { true, learningRate, iter, total_cost };
416         }
417       }
418
419     } // Step 8: go back to step 3, until we have taken "maxIterations" steps
420
421     // Step 9: return "false", indicating that the training did not succeed.
422     return { false, learningRate, maxIterations, TotalCost(x, y) };
423   }
424
425
426   // For a neural network with two inputs x=(x1, x2) and one output y,
427   // loop over (x1, x2) for a grid of points in [0, 1]x[0, 1]
428   // and save the value of the network output y evaluated at these points
429   // to a file. Returns true if the file was saved successfully.
430   bool ExportOutput(std::string filename)
431   {
432     // Check that the network has the right number of inputs and outputs
433     assert(NInputNeurons()==2 && NOutputNeurons()==1);
434
435     // Open a file with the specified name.
436     std::ofstream f(filename);
437
438     // Return false, indicating failure, if file did not open
439     if (!f)
440     {
441       return false;
442     }
443
444     // generate a matrix of 250x250 output data points
445     for (int i=0; i<=250; i++)
446     {
447       for (int j=0; j<=250; j++)
448       {
449         MVector out = Evaluate({i/250.0, j/250.0});
450         f << out[0] << " ";
451       }
452       f << std::endl;
453     }
454     f.close();
455
456     if (f) return true;
457     else return false;
458   }
459
460
461   static bool Test(int test);
462
463 private:
464   // Return the activation function sigma
465   /*
466   Activation function that returns value between [-1, 1]. The more extreme the input,
467   the closer the returned value will be to 1, with extreme negatives ~ -1 and extreme
468   positives ~ 1
469   */
470   double Sigma(double z)
471   {
472     return tanh(z);
473   }
474
475   // Return the derivative of the activation function
476   /*
477   Derivative of tanh(x) = sech^2(x) = 1 / cosh^2(x)
```

```
478   */
479   double SigmaPrime(double z)
480   {
481     double temp = cosh(z);
482     return (1.0 / (temp * temp));
483   }
484
485   // Loop over all weights and biases in the network and set each
486   // term to a random number normally distributed with mean 0 and
487   // standard deviation "initsd"
488   void InitialiseWeightsAndBiases(double initsd)
489   {
490     // Make sure the standard deviation supplied is non-negative
491     assert(initsd>=0);
492
493     // Set up a normal distribution with mean zero, standard deviation "initsd"
494     // Calling "dist(rnd)" returns a random number drawn from this distribution
495     std::normal_distribution<> dist(0, initsd);
496
497     // go through each layer (except the first)
498     for (int i = 1; i < biases.size(); i++)
499     {
500       // at each layer, the biases are a vector of values equal to the number of
      neurons
501       // at each layer, the weights are a matrix of values equal to the
502       // number of neurons x number of neurons in previous layer, so we can go through
503       // these data structures simulatenously
504
505       // set all weights in this layer
506       weights[i] = dist(rnd);
507
508       // go through all biases in this layer and set them
509       for (int j = 0; j < biases[i].size(); j++)
510       {
511         biases[i][j] = dist(rnd);
512       }
513     }
514
515   }
516
517   // Evaluate the feed-forward algorithm, setting weighted inputs and activations
518   // at each layer, given an input vector x
519   void FeedForward(const MVector &x)
520   {
521     // Check that the input vector has the same number of elements as the input layer
522     assert(x.size() == nneurons[0]);
523
524     // TODO: Implement the feed-forward algorithm, equations (1.7), (1.8)
525
526     // use the set values in the activation paramater
527     // first layer is equal to the input
528     activations[0] = x;
529
530     // go through each layer, excluding the first layer
531     for (int l = 1; l < nLayers; l++)
532     {
533       int prev_l = l - 1;
534
535       // get our inputs (z) for the activation function (sigma)
536       inputs[l] = weights[l] * activations[prev_l] + biases[l];
537
538       // for each neuron in the current layer put the inputs through sigma
539       for (int n = 0; n < inputs[l].size(); n++)
540       {
541         activations[l][n] = Sigma(inputs[l][n]);
```

```
542       }
543     }
544   }
545
546   // Evaluate the back-propagation algorithm, setting errors for each layer
547   void BackPropagateError(const MVector &y)
548   {
549     // Check that the output vector y has the same number of elements as the output
      layer
550     assert(y.size() == nneurons[nLayers - 1]);
551
552     // TODO: Implement the back-propagation algorithm, equations (1.22) and (1.24)
553
554     // The error is  vector of vectors, where each layer has an error for each nueron.
555     // We must evaluate back to front, neurons to find errors for the previous neuron
556
557     // evaluate final layer
558
559     MVector diff = activations[nLayers - 1] - y;
560     std::vector<MVector> temp = inputs; // temp vector for signma prime
561
562     for (int n = 0; n < inputs[nLayers - 1].size(); n++)
563     {
564       temp[nLayers - 1][n] = SigmaPrime(inputs[nLayers - 1][n]);
565     }
566
567     errors[nLayers - 1] = temp[nLayers - 1] * diff;
568
569     // now we can use the final layer to propogate the errors back through the network
570     // No error associated with the input layer, so end after l == 1.
571     for (int l = nLayers - 2; l > 0; l--)
572     {
573
574       for (int n = 0; n < inputs[l].size(); n++)
575       {
576         temp[l][n] = SigmaPrime(inputs[l][n]);
577       }
578       MVector temp2 = TransposeTimes(weights[l + 1], errors[l + 1]);
579       errors[l] = temp[l] * temp2;
580     }
581   }
582
583
584   // Apply one iteration of the stochastic gradient iteration with learning rate eta.
585   void UpdateWeightsAndBiases(double eta)
586   {
587     // Check that the learning rate is positive
588     assert(eta>0);
589
590     // TODO: update the weights and biases according to the stochastic gradient
591     //       iteration, using equations (1.25) and (1.26) to evaluate
592     //       the components of grad C.
593
594     // go through each layer and update weights and biases, excluding the first layer
595
596     for (int l = 1; l < nLayers; l++)
597     {
598       biases[l] -= eta * errors[l];
599       weights[l] -= eta * OuterProduct(errors[l], activations[l - 1]);
600     }
601   }
602
603
604   // Return the cost function of the network with respect to a single the desired
      output y
```

```
  // Note: call FeedForward(x) first to evaluate the network output for an input x,
  //       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 outputs
    assert(y.size() == nneurons[nLayers-1]);

    // TODO: Return the cost associated with this output
    double c = 0;

    // find L_2 of y - a^{L} x where y is the output, x is the input,
    //a^{L} are the activations in the final layer of neurons

    for (int i = 0; i < y.size(); i++)
    {
      c += std::pow(y[i] - activations[nLayers - 1][i], 2);
    }

    return 0.5 * c;
  }

  // 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());

    // TODO: Implement the cost function, equation (1.9), using
    //       the FeedForward(x) and Cost(y) methods

    double total_cost = 0;

    for (int i = 0; i < x.size(); i++)
    {
      FeedForward(x[i]);
      total_cost += Cost(y[i]);
    }

    return (1.0 / x.size()) * total_cost;
  }

  // Private member data

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

};



bool Network::Test(int test = 0)
{
  // 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

  double tol = 1e-10;

  // A example test of FeedForward
  if (test == 1 || test == 0)
```

```
670   {
671     // Make a simple network with two weights and one bias
672     Network n({2, 1});
673
674     // Set the values of these by hand
675     n.biases[1][0] = 0.5;
676     n.weights[1](0,0) = -0.3;
677     n.weights[1](0,1) = 0.2;
678
679     // Call function to be tested with x = (0.3, 0.4)
680     n.FeedForward({0.3, 0.4});
681
682     // Display the output value calculated
683     std::cout << n.activations[1][0] << std::endl;
684
685     // Correct value is = tanh(0.5 + (-0.3*0.3 + 0.2*0.4))
686     //                  = 0.454216432682259...
687     // Fail if error in answer is greater than 10^-10:
688     if (std::abs(n.activations[1][0] - 0.454216432682259) > tol)
689     {
690       std::cout << "FAILED: FeedForward" << std::endl;
691       return false;
692     }
693   }
694
695
696   // TODO: for each part of the Network class that you implement,
697   //       write some more tests here to run that code and verify that
698   //       its output is as you expect.
699   //       I recommend putting each test in an empty scope { ... }, as
700     //       in the example given above.
701
702   // test activation function
703   if (test == 2 || test == 0)
704   {
705     Network n({ 2, 1 });
706
707     double result = n.Sigma(1000);
708     double expect = 1;
709
710     if (std::abs(result - expect) > tol)
711     {
712       std::cout << "FAILED: activation function, extreme positive" << std::endl;
713       return false;
714     }
715
716     result = n.Sigma(-1000);
717     expect = -1;
718
719     if (std::abs(result - expect) > tol)
720     {
721       std::cout << "FAILED: activation function, extreme negative" << std::endl;
722       return false;
723     }
724
725     result = n.Sigma(0);
726     expect = 0;
727
728     if (std::abs(result - expect) > tol)
729     {
730       std::cout << "FAILED: activation function, zero input" << std::endl;
731       return false;
732     }
733
734     result = n.Sigma(1);
```

```
735     expect = 0.7615941559557;
736
737     if (std::abs(result - expect) > tol)
738     {
739       std::cout << "FAILED: activation function, realistic positive input" << std::
    endl;
740       return false;
741     }
742
743     result = n.Sigma(-0.3);
744     expect = -0.29131261245159;
745
746     if (std::abs(result - expect) > tol)
747     {
748       std::cout << "FAILED: activation function, realistic negative input" << std::
    endl;
749       return false;
750     }
751
752   }
753
754   // test derivative of activation function
755   if (test == 3 || test == 0)
756   {
757     Network n({ 2, 1 });
758
759     double result = n.SigmaPrime(0);
760     double expect = 1;
761
762     if (std::abs(result - expect) > tol)
763     {
764       std::cout << "FAILED: activation function derivative, 0 input" << std::endl;
765       return false;
766     }
767
768     result = n.SigmaPrime(1000);
769     expect = 0;
770
771     if (std::abs(result - expect) > tol)
772     {
773       std::cout << "FAILED: activation function derivative, extreme positive" << std::
    endl;
774       return false;
775     }
776
777     result = n.SigmaPrime(-1000);
778     expect = 0;
779
780     if (std::abs(result - expect) > tol)
781     {
782       std::cout << "FAILED: activation function derivative, extreme negative" << std::
    endl;
783       return false;
784     }
785
786     result = n.SigmaPrime(-0.8);
787     expect = 0.55905516773;
788
789     if (std::abs(result - expect) > tol)
790     {
791       std::cout << "FAILED: activation function derivative, extreme negative" << std::
    endl;
792       return false;
793     }
794
```

```
795     result = n.SigmaPrime(0.4);
796     expect = 0.85563878608;
797 
798     if (std::abs(result - expect) > tol)
799     {
800       std::cout << "FAILED: activation function derivative, extreme negative" << std::
    endl;
801       return false;
802     }
803   }
804 
805   // test initialisation if weights and biases
806   if (test == 4 || test == 0)
807   {
808     // create a simple network
809     Network n({ 2, 3, 3, 1 });
810     double expect = 0;
811 
812     // check that weights and biases are all 0 before initialisation. Display them
813     for (int i = 1; i < n.biases.size(); i++)
814     {
815       for (int j = 0; j < n.biases[i].size(); j++)
816       {
817         if (n.biases[i][j] != expect)
818         {
819           std::cout << "FAILED: weights and biases pre-initilasation, biases" << std::
    endl;
820           return false;
821         }
822         for (int k = 0; k < n.weights[i].Cols(); k++)
823         {
824           if (n.weights[i](j, k) != expect)
825           {
826             std::cout << "FAILED: weights and biases pre-initilasation, weights" <<
    std::endl;
827             return false;
828           }
829         }
830       }
831     }
832 
833     // initalise weights and biases to a SD of 10 and check none are 0
834     n.InitialiseWeightsAndBiases(10);
835 
836     for (int i = 1; i < n.biases.size(); i++)
837     {
838       std::cout << "For layer " << i + 1 << std::endl;
839       for (int j = 0; j < n.biases[i].size(); j++)
840       {
841         std::cout << "For neuron " << j + 1 << std::endl;
842         std::cout << "Biase: " << n.biases[i][j] << std::endl;
843         if (n.biases[i][j] == expect)
844         {
845           std::cout << "FAILED: weights and biases initilasation, biases" << std::endl
    ;
846           return false;
847         }
848         for (int k = 0; k < n.weights[i].Cols(); k++)
849         {
850           std::cout << "Weights: " << n.weights[i](j, k) << std::endl;
851           if (n.weights[i](j, k) == expect)
852           {
853             std::cout << "FAILED: weights and biases initilasation, weights" << std::
    endl;
854             return false;
```

```
855           }
856         }
857       }
858     }
859
860     // create a new, larger, more complex network, test size of each step is correct
861     unsigned int input_size = 2;
862     unsigned int layer_size = 10;
863     unsigned int output_size = 1;
864     Network n2({ input_size, layer_size, layer_size - 1, layer_size + 1, output_size
    });
865     n2.InitialiseWeightsAndBiases(5);
866     int loops = 0;
867     int expected_loops = (input_size * layer_size) +
868       (layer_size * (layer_size - 1)) +
869       ((layer_size - 1) * (layer_size + 1)) +
870       ((layer_size + 1) * output_size);
871
872     for (int i = 1; i < n2.biases.size(); i++)
873     {
874       for (int j = 0; j < n2.biases[i].size(); j++)
875       {
876         if (n2.biases[i][j] == expect)
877         {
878           std::cout << "FAILED: weights and biases large network initilasation, biases
    " << std::endl;
879           return false;
880         }
881         for (int k = 0; k < n2.weights[i].Cols(); k++)
882         {
883           if (n2.weights[i](j, k) == expect)
884           {
885             std::cout << "FAILED: weights and biases large network initilasation,
    weights" << std::endl;
886             return false;
887           }
888           loops += 1;
889         }
890       }
891     }
892
893     // check we have checked every value
894     if (loops != expected_loops)
895     {
896       std::cout << "FAILED: weights and biases large network initilasation, check all
    values" << std::endl;
897     }
898
899   }
900
901   // test back propogation
902   if (test == 5 || test == 0)
903   {
904     // Make a simple network with two weights and one bias
905     Network n({ 2, 1 });
906
907     // Set the values of these by hand
908     n.biases[1][0] = 0.5;
909     n.weights[1](0, 0) = -0.3;
910     n.weights[1](0, 1) = 0.2;
911
912     // applying this feed forward gives an output of
913     // approx 0.454216432682259
914     n.FeedForward({ 0.3, 0.4 });
915
```

```
    // check that giving 0.454216432682259 as the true value
    // gives a VERY small error
    n.BackPropagateError({ 0.454216432682259 });

    if (std::abs(n.errors[1][0]) > tol)
    {
      std::cout << "FAILED: back propogation, final layer" << std::endl;
      return false;
    }

    // check that giving |y| >> 0.454216432682259 as the true value
    // gives a large error

    n.BackPropagateError({ -99999999 });

    if (std::abs(n.errors[1][0]) < 1000)
    {
      std::cout << "FAILED: back propogation, final layer" << std::endl;
      return false;
    }

    // check that giving |y| = 1 as the true value
    // gives an error of approx -0.43318

    n.BackPropagateError({ 1 });

    if (std::abs(n.errors[1][0] - -0.43318) > 0.00001)
    {
      std::cout << "FAILED: back propogation, final layer" << std::endl;
      return false;
    }
  }

  // test updating of weights and biases
  if (test == 6 || test == 0)
  {
    // 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;

    n.UpdateWeightsAndBiases(0.5);

    // since no data has beem run, errors should be all 0, so biases and weights
    // should be unchanged

    if (std::abs(n.biases[1][0] - 0.5) > tol)
    {
      std::cout << "FAILED: updating weights and biases, 0 error for biase" << std::
    endl;
      return false;
    }
    if (std::abs(n.weights[1](0, 0) - -0.3) > tol)
    {
      std::cout << "FAILED: updating weights and biases, 0 error for weights" << std::
    endl;
      return false;
    }
    if (std::abs(n.weights[1](0, 1) - 0.2) > tol)
    {
      std::cout << "FAILED: updating weights and biases, 0 error for weights" << std::
    endl;
```

```
978       return false;
979     }
980
981     // run the network with the exact answer in the back propogation
982     // should expect no change to weights and biases
983     // applying this feed forward gives an output of
984     // approx 0.454216432682259
985
986     n.FeedForward({ 0.3, 0.4 });
987     n.BackPropagateError({ 0.454216432682259 });
988
989     n.UpdateWeightsAndBiases(0.1);
990
991     if (std::abs(n.biases[1][0] - 0.5) > tol)
992     {
993       std::cout << "FAILED: updating weights and biases, exact answer for biase" <<
      std::endl;
994       return false;
995     }
996     if (std::abs(n.weights[1](0, 0) - -0.3) > tol)
997     {
998       std::cout << "FAILED: updating weights and biases, exact answer for weights" <<
      std::endl;
999       return false;
1000    }
1001    if (std::abs(n.weights[1](0, 1) - 0.2) > tol)
1002    {
1003      std::cout << "FAILED: updating weights and biases, exact answer for weights" <<
      std::endl;
1004      return false;
1005    }
1006
1007    // set some values for activations and errors
1008
1009    n.activations[0][0] = 0.1;
1010    n.activations[0][1] = 0.1;
1011    n.errors[1][0] = 0.1;
1012    n.UpdateWeightsAndBiases(0.1);
1013
1014    // should epect biases to change by 0.01
1015    // should expect weights to change by 0.001
1016
1017    if (std::abs(n.biases[1][0] - 0.49) > tol)
1018    {
1019      std::cout << "FAILED: updating weights and biases, small error for biase" << std
      ::endl;
1020      return false;
1021    }
1022    if (std::abs(n.weights[1](0, 0) - -0.301) > tol)
1023    {
1024      std::cout << "FAILED: updating weights and biases, small error for weights" <<
      std::endl;
1025      return false;
1026    }
1027    if (std::abs(n.weights[1](0, 1) - 0.199) > tol)
1028    {
1029      std::cout << "FAILED: updating weights and biases, small error for weights" <<
      std::endl;
1030      return false;
1031    }
1032  }
1033
1034  // test Cost
1035  if (test == 7 || test == 0)
1036  {
```

```
    // 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;

    n.FeedForward({ 0.3, 0.4 });

    // cost should be 0

    double cost = n.Cost({ 0.454216432682259 });
    if (std::abs(cost - 0) > tol)
    {
      std::cout << "FAILED: Cost, simple network; exact answer" << std::endl;
      return false;
    }

    // Make a more complex network
    Network n2({ 2, 3, 4, 3 });

    n2.FeedForward({ 0.1, 0.2 });
    n2.activations[3][0] = 1;
    n2.activations[3][1] = 0.5;
    n2.activations[3][2] = 0.1;

    cost = n2.Cost({ 1, 0.5, 0.1 });

    // check that if output is equal to activations, cost is 0
    if (std::abs(cost - 0) > tol)
    {
      std::cout << "FAILED: Cost, complex network; exact answer" << std::endl;
      return false;
    }

    // check that if output is wrong for one neuron we get the right cost
    //a^L[0] is out by 0.5, so we expect cost to equal 0.5 * (1-0.5)^2

    cost = n2.Cost({ 0.5, 0.5, 0.1 });
    if (std::abs(cost - 0.125) > tol)
    {
      std::cout << "FAILED: Cost, complex network; one wrong neuron" << std::endl;
      return false;
    }

    // check that the costs add up correctly if >1 are wrong
    // first neuron cost is still 0.125
    // second is 0.5 * (0.5 - (0.8))^2 = 0.045
    // third is 0.5 * (0.1 - (0.9))^2 = 0.32
    // expect cost to be 0.49
    cost = n2.Cost({ 0.5, 0.8, 0.9 });
    if (std::abs(cost - 0.49) > tol)
    {
      std::cout << "FAILED: Cost, complex network; many wrong neurons" << std::endl;
      return false;
    }
  }
  // test total cost
  if (test == 8 || test == 0)
  {
    // Make a simple network with two weights and one bias
    Network n({ 2, 1 });

    // set them to correct values
```

```
    n.biases[1][0] = 0.5;
    n.weights[1](0, 0) = -0.3;
    n.weights[1](0, 1) = 0.2;
    n.activations[1][0] = 0.454216432682259;

    // Make a simple training data set which holds correct answers
    // expect total cost to be almost 0

    std::vector<MVector> x, y;
    x = { { 0.3, 0.4 } };
    y = { {0.454216432682259} };

    double total_cost = n.TotalCost(x, y);
    if (std::abs(total_cost) > tol)
    {
      std::cout << "FAILED: TotalCost, right input/output" << std::endl;
      return false;
    }

    //expect total cost to be specifc value

    y = { {0.40421643268225} }; //incorrect by 0.05
    double expect = 0.00125;

    total_cost = n.TotalCost(x, y);

    if (std::abs(total_cost - expect) > tol)
    {
      std::cout << "FAILED: TotalCost, wrong input/output" << std::endl;
      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(std::vector<unsigned> nneurons, std::string filename = "", int
    dataset = 0)
{
  // Create a network with two input neurons, two hidden layers of three neurons, and
    one output neuron
  Network n(nneurons);

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

  if (dataset == 1)
  {
    GetCheckerboardData(x, y);
  }
  else if (dataset == 2)
  {
    GetSpiralData(x, y);
  }
  else
  {
    GetTestData(x, y);
```

```
1165  }
1166
1167  // Train network on training inputs x and outputs y
1168  // Numerical parameters are:
1169  //  initial weight and bias standard deviation = 0.1
1170  //  learning rate = 0.1
1171  //  cost threshold = 1e-4
1172  //  maximum number of iterations = 10000
1173
1174  /*
1175  // try different learning rates and see what results we get
1176  double eta = 0.001;
1177  std::ofstream myfile;
1178  myfile.open("eta_conergence_small.txt");
1179
1180  while (eta <= 0.35)
1181  {
1182    Network n1(nneurons);
1183    NetworkData trainingSucceeded = n1.Train(x, y, 0.1, eta, 1e-4, 100000);
1184
1185    // If training failed, report this
1186    if (!trainingSucceeded.success)
1187    {
1188      std::cout << "Failed to converge to desired tolerance." << std::endl;
1189    }
1190
1191    myfile << trainingSucceeded.eta << "\t" << trainingSucceeded.cost << "\t" <<
1192      trainingSucceeded.iterations << std::endl;
1193
1194    eta += 0.001;
1195  }
1196  myfile.close();
1197  */
1198
1199  /*
1200  // try different standard deviations
1201  double sd = 5;
1202  std::ofstream myfile;
1203  myfile.open("sd_conergence_large.txt");
1204
1205  while (sd <= 10)
1206  {
1207    std::cout << "standard deviation: " << sd << std::endl;
1208    Network n1(nneurons);
1209    NetworkData trainingSucceeded = n1.Train(x, y, sd, 0.1, 1e-4, 100000);
1210
1211    // If training failed, report this
1212    if (!trainingSucceeded.success)
1213    {
1214      std::cout << "Failed to converge to desired tolerance." << std::endl;
1215    }
1216
1217    myfile << sd << "\t" << trainingSucceeded.cost << "\t" <<
1218      trainingSucceeded.iterations << std::endl;
1219
1220    sd += 0.1;
1221  }
1222  myfile.close();
1223  */
1224  NetworkData trainingSucceeded = n.Train(x, y, 0.1, 0.0005, 0.05, 5000000);
1225
1226  // If training failed, report this
1227  if (!trainingSucceeded.success)
1228  {
1229    std::cout << "Failed to converge to desired tolerance." << std::endl;
```

```
  }

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

  std::cout << "Tests passed, procede to example program...\n" << std::endl;

  // Tests passed, so run our example program.

  //std::vector<unsigned> nneurons = {2, 3, 3, 1 };
  //ClassifyTestData(nneurons);

  //test complex data
  // no hidden layers

  std::vector<unsigned> nneurons = {2, 1 };
  ClassifyTestData(nneurons, "Fcheckers21", 1);
  ClassifyTestData(nneurons, "Fspiral21", 2);

  // one hidden layer

  for (unsigned int i = 5; i < 21; i += 5)
  {
    std::vector<unsigned> nneurons1 = { 2, i, 1 };
    ClassifyTestData(nneurons1, "Fcheckers2" + std::to_string(i) + "1", 1);
    ClassifyTestData(nneurons1, "Fspiral2" + std::to_string(i) + "1", 2);
  }

  // two hidden layer

  for (unsigned int i = 10; i < 21; i += 5)
  {
    std::vector<unsigned> nneurons1 = { 2, i, i, 1 };
    ClassifyTestData(nneurons1, "Fcheckers2" + std::to_string(i) + std::to_string(i) +
     "1", 1);
    ClassifyTestData(nneurons1, "Fspiral2" + std::to_string(i) + std::to_string(i) + "
    1", 2);
  }

  // three hidden layer
  for (unsigned int i = 5; i < 21; i += 5)
  {
    std::vector<unsigned> nneurons1 = { 2, i, i, i, 1 };
    ClassifyTestData(nneurons1, "Fcheckers2" + std::to_string(i) +
      std::to_string(i) + std::to_string(i) + "1", 1);
    ClassifyTestData(nneurons1, "Fspiral2" + std::to_string(i) +
      std::to_string(i) + std::to_string(i) + "1", 2);
  }


```

```
1293   return 0;
1294 }
```

Artifical Neural Networks

GRADEMARK REPORT

FINAL GRADE

/60

GENERAL COMMENTS

Instructor

PAGE 1

Text Comment. Excellent work, see comments below.

PAGE 2

PAGE 3

Comment 1

One of the topics I considered when writing this project was a neural network to play 'Flappy Bird', which was very big at the time (not so much now). It's a similar classification problem in R^3 (inputs are x[0]=distance to next gate, x[1]=height offset from next gate, x[2]=time since last flap. Output y[0] >0 means flap, otherwise is flap or don't flap)

Text Comment. In LaTeX, use backtick ` for open quotes, ' for close quotes.

PAGE 4

QM

Good

Comment 2

in what sense? I think this is only unambiguous if you already know what a hidden layer is...

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

Text Comment. Good. I like the log axes, but do you think a line of best fit is appropriate for these data?

PAGE 10

Text Comment. The results here are rather dominated by the threshold of 10^-4 -- it might have been better to show cost after some fixed number of iterations, with no threshold-based exit from the loop.

PAGE 11

PAGE 12

Text Comment. Is over-fitting relevant to this problem in this report, where there is no distinction between training and validation data?

Text Comment. Good, but would benefit from a citation

Text Comment. Testing is good - but a little more text here would have been beneficial.

Text Comment. Conclusion?

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

Text Comment. Good checkerboard and spiral data graphs, but I would like to have seen these discussed more in the body of the report.

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

PAGE 34

PAGE 35

PAGE 36

PAGE 37

PAGE 38

PAGE 39

PAGE 40

PAGE 41

PAGE 42

PAGE 43

PAGE 44

PAGE 45

PAGE 46