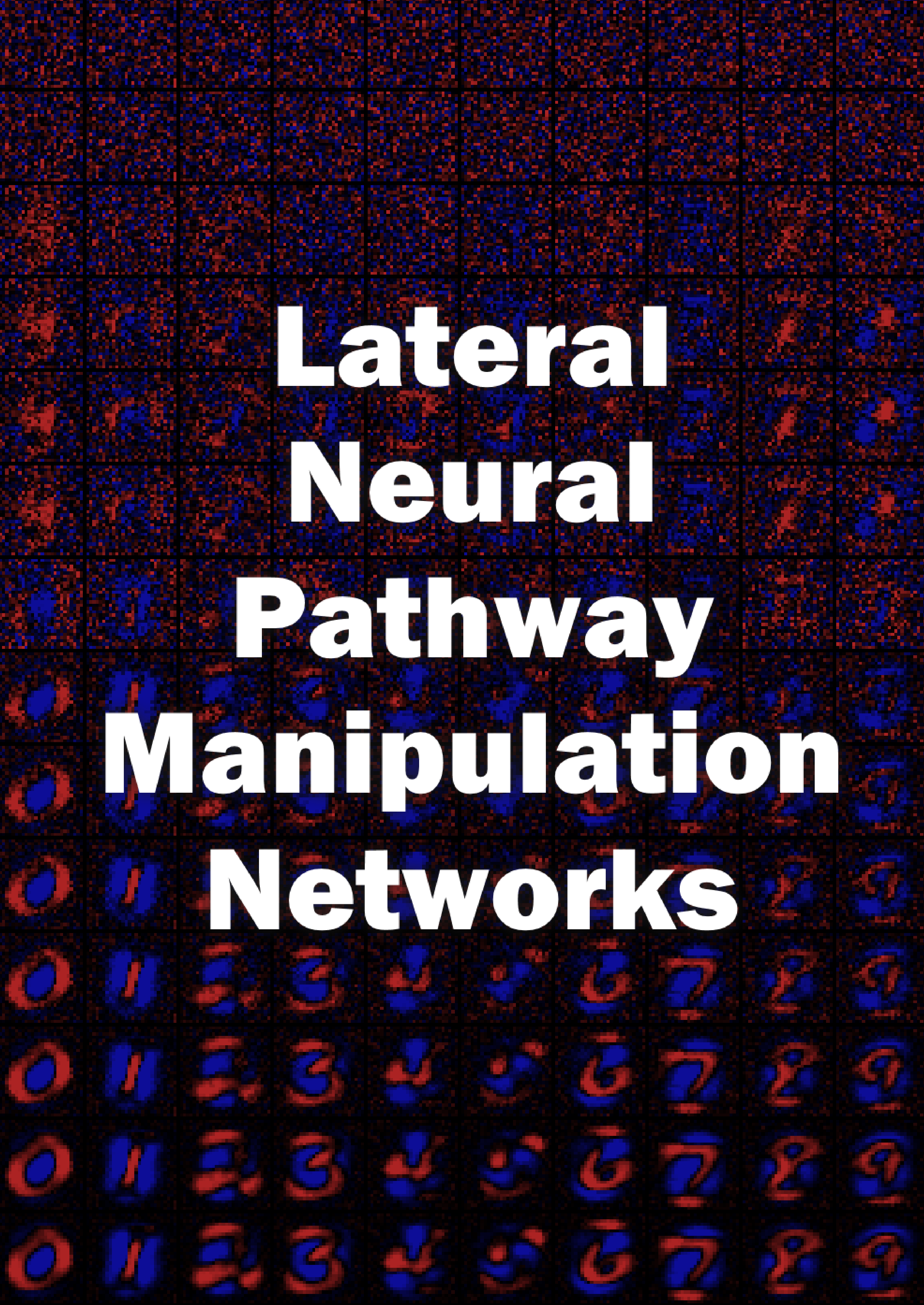


Lateral Neural Pathway Manipulation Networks

Contents

Lateral Neural Pathway Manipulation Networks - A New Type Of Artificial Neural Network Which Is Adaptive To New Situations.

George Bird - May 19, 2018
Candidate Number: 7024

Abstract

Artificial Neural Networks have been trained to solve a plethora of problems given to them. However, they often fail to generalise when given an unfamiliar, yet similar, problem to those which they have been trained on. They lack the ability of lateral thought to resolve these problems and reach a sensible conclusion. For example an artificial neural network would struggle recognising a image of a letter as a distinct character when they have only ever been trained on number images.

For this reason a new approach to generalising neural networks is needed. In this paper I aim to describe a new type of neural network I have called 'Lateral Neural Pathway Manipulation Network'. It aims to alter the neural network's structure, as it learns, to allow the neural network to efficiently learn new skills without impeding the functionality of the original network.

Notes.

It is important to note at this stage, that the views on the definitions of the phrase 'Artificial Intelligence' (AI) is wide ranging, and no consensus has been agreed as yet. A.I is most often means a general purpose intelligence which can master many different styles of problems. However, the term is often used for specific purpose intelligence which is widely available today, such as Speech Recognition, Siri and Driverless Cars. Ourselves and a selection of other animals are the only general purpose intelligence known currently.

For the purposes of this study, I will define artificial intelligence, as any algorithm which utilises supervised or unsupervised machine learning to solve a problem.

Another central aspect of Lateral Neural Pathway Manipulation is varying the number of output neurons. This could potentially present a problem to applications reliant on the number of outputs from a neural networks being constant, such as in robotics when one output is assigned to one physical motor. Otherwise applications could be designed to work with this new functionality, allowing networks to extend their learning capacity.

In addition 'Lateral Neural Pathway Manipulation Networks' is simply an idea proposed to advance the functionality of Neural Networks. Due to the time constraints for this project, and my lack of suitable computing hardware, I will not be able to test the algorithms to determine their effectiveness. This project is just explaining the idea, mathematics and inspiration for this new style of Neural Networks.

Introduction

In this section I will first outline the history of Artificial Intelligence and Neural Networks, as well as explore a little on the biology on which it is based. Following this I will explain the back-propagation algorithm and the maths of a basic Artificial Neural Network.

A brief history of Artificial Intelligence.

Artificial Intelligence (AI) has long been a feature of Sci-fi books, films and TV shows. However over the last few decades this idea is becoming a reality. With ever more complex AI's which can read, write, construct sentences, understand speech, play chess, this list goes on, being created. This field of computer science has been exponentially growing for many years now.

In 1943, two computer scientists interested in the workings of the brain, Walter Pitts and Warren McCullock, developed a computer model for how biological neural networks could be simulated. This was called Threshold logic (Gefter, 2015). It used binary inputs which were passed through layers of logic gates to obtain binary outputs. This system could approximately model a biological neural network, and inspired numerous people to investigate this field further.

In subsequent years, these ideas were greatly expanded upon by countless scientists who began to investigate the electrical processes in our brains, as well as the applications of artificial neural networks to solve problems. Many innovations followed such as backpropagation (which will be explained later in this project), Hebbian Learning, Convolution Networks alongside a myriad of other scientific breakthroughs.

Biological Basis

Artificial Neural Network development has always been inspired from the way our mammalian brains operate. Drawing on concepts from psychological and biological research. Our brains are a vast network of neurons, numbering above 100 billion (Unknown, 2016), with each neuron having around ten-thousand connections to other neurons, which send and receive electrical impulses to communicate. This design has been refined over millions of years of evolution to produce the most complicated structure known in the universe (Kaku, 2014), and has resulted in some of the best problem solvers across the animal kingdom.

For clear reasons, this immense biological computer has captivated the interest of many scientists, who still work tirelessly to understand the brain's intricacies. Artificial Neural Networks are an attempt of replicating this biological structure to achieve better performance on tasks than any other computer programs of the past. Currently this technology is still a work in progress and the networks are less sophisticated than even small mammals. However if this technology progresses to super-human intelligence, it could revolutionise the world we live in. It is predicted Artificial Intelligence will cause similar increase in development as the Industrial Revolution caused.

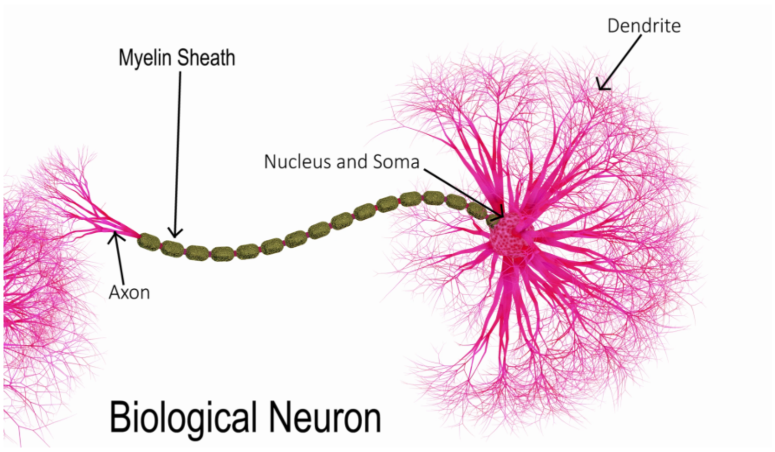
The seeds of this technological revolution have already begun to germinate, with brand new software's such as self-driving cars, stock market predictions, medical diagnostics, fraud detection, and speech recognition, to name but a few. All of which utilising these artificial neural networks, and other machine learning, to match or outperform humans on these tasks.

A Biological Neuron

As well as the basic layout of Artificial Neural Networks mimicking that of biological networks, the structure of the individual neurons in these networks is also comparable.

The structure of a biological neuron is shown to the upper-right. The unique sections of a neuron consist of an axon, axon hillock, myelin sheath, dendrite, nucleus, soma, and synapses.

The dendrites are a branching structure which receive information from neighbouring neurons and transmits it to the soma. At the end of each dendrite is a synapse which detects neuro-transmitting chemicals from the adjacent neuron such as acetylcholine, dopamine and serotonin. This triggers an electrical impulse which travels to the soma. The incoming signal from the dendrites can either be a excitatory or inhibitory signal, where excitatory increases the likelihood of the neuron firing and inhibitory signal reduces this likelihood.



The soma is joined to all the dendrites, and the electrical signals from the dendrites culminate here and add together. It is this summed signal which is then passed to the axon hillock. The axon hillock controls the firing of the neuron. The neuron will only fire when the total strength of the incoming electrical impulse surpasses a threshold. If it does surpass the threshold an electrical impulse, known as the action potential, is transmitted down the axon. The axon hillock either fires or does not, resulting in a boolean output. In fig 1: a graph of an example activation function of the hillock is shown where the neuron either fires or does not.

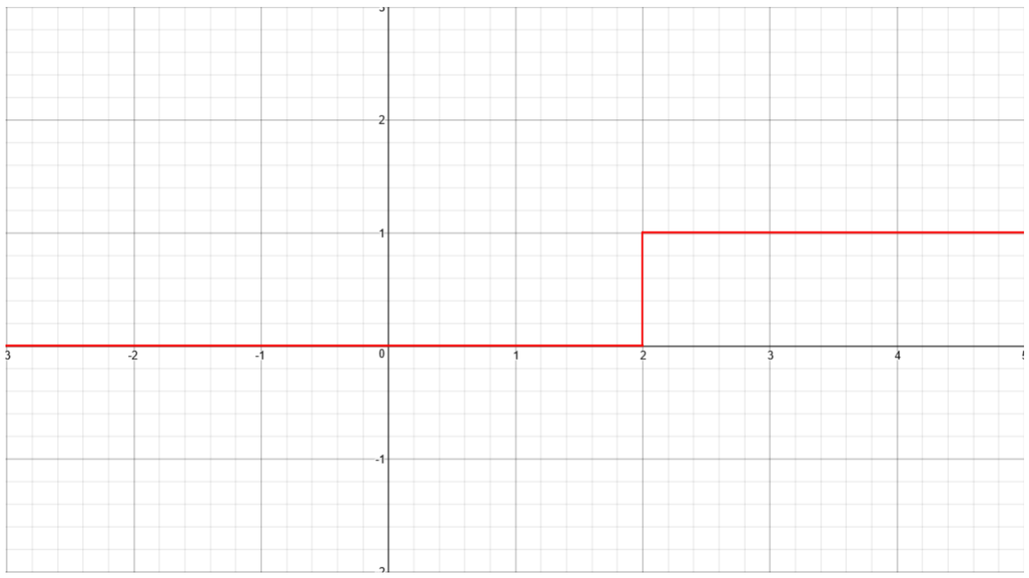


Figure 1. For this example the neuron would fire if the summed result of the inhibitory and excitatory signal surpasses this arbitrary value 2 on the x-axis.

As mentioned if signal in the axon hillock surpasses the threshold an impulse is sent down the axon. This axon is connected to another adjacent neuron and the impulse can be transmitted through more synapses and neurotransmitters (Soso, 2016). Neurons mostly have one axon, and this axon is surrounded by the myelin sheath. The myelin sheath speeds up the rate of impulse transmission, and the more frequently a neuron fires; the greater the myelin sheath is built up (Khan, 2010).

An Artificial Neuron

Artificial neurons are incredibly similar to biological neurons. They have comparable sections with analogous functions which produce similar outputs. However artificial neurons are very fast at learning as they utilise CPUs and GPUs which can operate in the region of four-billion calculation per second and this can be easily exceeded by super computers. This means a network of these artificial neurons can learn to read, from scratch, in minutes as opposed to years. As a result they pose an exciting new technology.

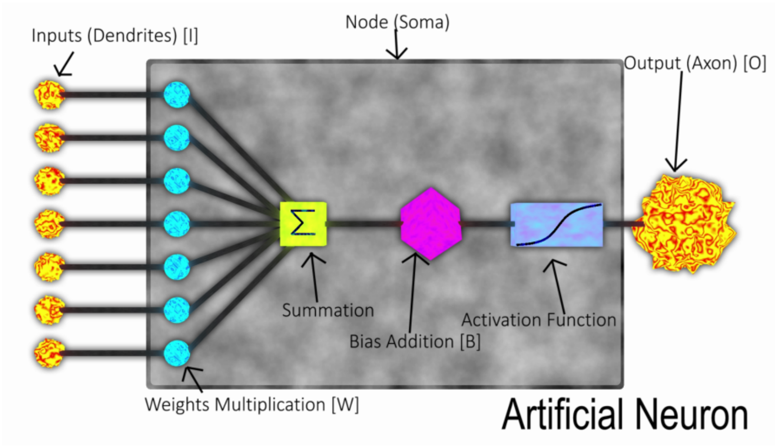


Figure . A graphical interpretation of a artificial neuron "perceptron", outlining the basic order of calculations per neuron in an Artificial Neural Network.

A graphical interpretation of an artificial neuron shown above-right (Fig 2). There are several different types of artificial neurons but the one I will describe in this project is a "perceptron" using the sigmoid activation function.

To begin the artificial neuron has inputs, just like how the biological neuron has the dendrites. These inputs connect to previous layers of neurons in the network; each neuron receives an activation signal varying between 0 and 1. Activation describes the strength of a signal. An activation of one is comparable to an excitatory signal in a biological neuron, whilst zero is comparable to an inhibitory signal. Each of these inputs is then multiplied by their differing values known as weights.

Weights are unique to artificial neural networks and represents how much importance is given to a particular input signal. It is somewhat comparable to the sensitivity of each dendrite's synapse to a particular neuro-transmitter. Following all the input-weight multiplications the resulting activation is then summed into one activation value. The following formula describes the mathematical calculations so far.

$$activation = \sum_{i=0}^{n} (input_i \times weight_i)$$

These calculations are commonly computed as matrices. As shown by the following equation: (The superscript denotes the layer of the neural network, e.g. L+1 is the layer after L.)

$$a^{L+1} = \left[a_1^L \, a_2^L \, \dots \, a_n^L\right] \times \left[w_1^L \, w_2^L \, w_n^L\right]$$

The summation function is similar to the soma in the way it sums input signals into one signal which is transmitted further along the neuron. Following the summation a bias value is added to the signal. The value for the bias is unique to every neuron in the network and describes how likely the neuron is to fire, regardless of the input activation. Like the weights, the bias is distinctive to the artificial neuron and cannot be found within the biological neuron. The closest biological interpretation of the bias is the myelin sheath, allowing neurons with a high amount of bias or myelin sheath, to transmit signals more easily to the next neuron in the network. This following formula appends the new step to the calculation:

$$activation = \sum_{i=0}^{n} (input_i \times weight_i) + bias$$

Again it is computed with matrices.

$$a^{L+1} = \left(\begin{bmatrix} a_1^L\ a_2^L \dots\ a_n^L \end{bmatrix} \times \begin{bmatrix} w_1^L\ w_2^L\ w_n^L \end{bmatrix}\right) + b^L$$

Finally the artificial neuron has an activation function which determines the strength of the transmitted signal based on the received signal. This is incredibly close to the biological axon hillock which transmits signals passed a certain threshold. There is a multitude of different activation functions used in Artificial Neural Networks including sigmoid, tanh, step-function (like the biological activation function), Rectified Linear Unit, softmax and many others, but each produces an output zero and one (except tanh from negative one to one). This is slightly different from the biological equivalent which must be either zero or one, no in between, as it is an electrical impulse. Below (Fig 3) shows the sigmoid function and how it limits a number to between zero and one.

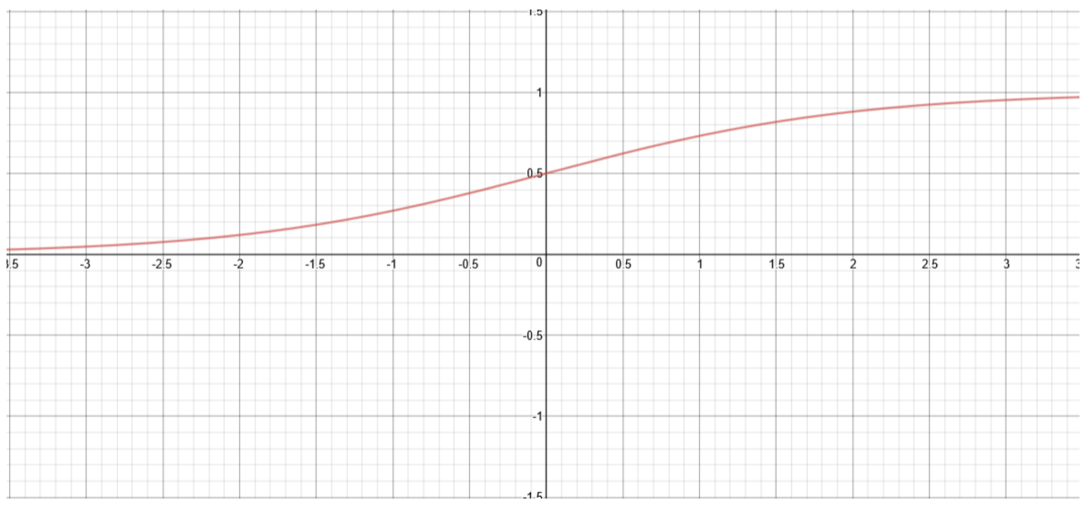


Figure 3. Sigmoid Function

$$sigmoid(x) = \frac{1}{1+e^{-x}}$$

This activation function completes all the necessary calculations for an artificial neuron, allowing the resulting signal to forward propagate through the network. The entire formula for the artificial neuron is shown below, assuming sigmoid is used.

$$a = \frac{1}{1+e^{-(\sum_{i=0}^{n}(input_i \times weight_i)+bias)}}$$

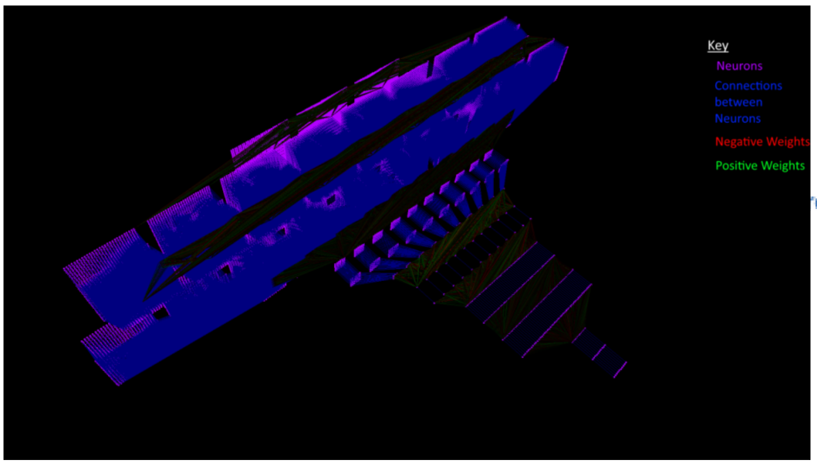
As before, there is an equivalent matrices calculation:

$$a^{L+1} = activationfunction\left(\begin{bmatrix} a_1^L & a_2^L & \dots & a_n^L \end{bmatrix} \times \begin{bmatrix} w_1^L & w_2^L & w_n^L \end{bmatrix} + b^L\right)$$

Numerous Neurons make a Network

So far I have described a single artificial and biological neuron, but as I previously mentioned these neurons connect together to form a vast web of neurons called a artificial neural network. One neuron alone can only compute a very basic function comparable to an electronic logic gate, however combining hundreds, or thousands, if not millions into a network makes a very powerful intelligent problem solver. Fig 4 demonstrates how 36476 artificial neurons chain together through 317172 neuron connections, to form a basic convolution network used for recognising images.

The organisation of artificial neurons into a network can take different structures, for different situations. The most commonly used structures include Convolution Networks used for image recognition (LeCun, Convolutional Neural Networks, 1998), Long-Short-Term-Memory (LSTM) Networks used frequently for speech recognition (Hochreiter & Schmidhuber, 1997), and Fully Connected Networks used in almost all ANN applications. The above image shows a combination of neurons in both fully connected and convolution structures, often used for image classification as it

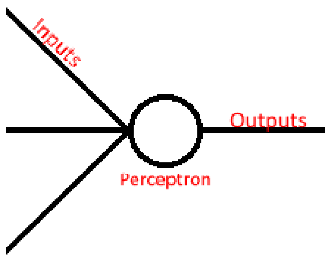


re.

mimics the human visual cortex structure and it performs exceptionally well.

An incredibly popular neural network structure is NEAT, short for Neuro-Evolution by Augmented Topologies (Stanely & Miikkulainen, 2002). NEAT is an adaptation of the widely used fully connected network, where the network begins as a random scattering of individual neurons. Subsequently, each iteration NEAT is used the most successful networks are allowed to 'reproduce' and slight variations are added to the network's structure per generation. After many cycles of NEAT all the randomly structured networks are whittled down to a few of the most successful and efficient network architectures. As suggested by the Neuro-Evolution in NEAT, this closely resembles Charles Darwin's survival of the fittest, where the most successful networks are identified by mathematical natural selection.

As mentioned NEAT works very well across many neural network applications, by using experimental methods to finding the best network structure. However, there is one major downside with NEAT: The majority of the network's learned parameters are lost by the change in network



architecture. The reason why this occurs is rather intuitive, you're removing or adding neurons to a network which is finely tuned to its own current structure. As previously mentioned, each neuron has a weight and bias. The bias is at a constant activation strength, so a neuron will overall have an average strength output. The subsequent layers in the network will almost expect and rely on this average output, so removing a neuron randomly with NEAT almost leaves the network in shock when this expected output is removed. NEAT does not compensate the network for this loss in activation, so the network's learning progress is lost.

My idea for Lateral Neural Pathway Manipulation (LNPM) has a similar result as NEAT where pathways of artificial neurons are added and removed. However, in theory, learning should not be lost after each architecture change as the network is compensated for the addition or removal of neurons. LNPM also uses an algorithmic approach, instead of experimental neuro-evolution, to decide when and how to change a network architecture based on whether the input data falls under certain criteria. This criteria will be discussed later in this project. The algorithm for LNPM I derived to work alongside the backpropagation algorithm, so that the network compensation preserves the function of learned network parameters, whilst still changing the networks architecture, ensuring little of previous learning is lost.

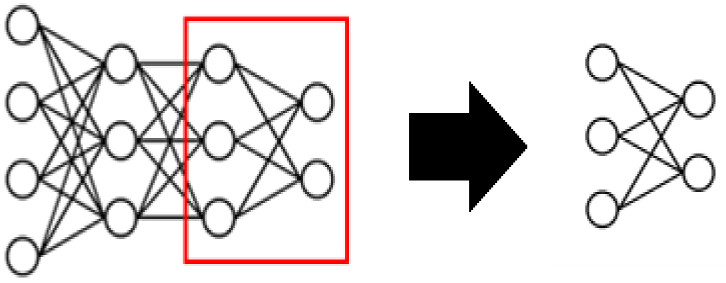
I will return to the details of LNPM later in this project, but before then I will explain a little on the mathematics of fully connected neural networks, so that the algorithms discussed later are clearer in their function. The following sections involves a lot of maths, which I certainly found daunting when I began learning about Artificial Neural Networks. I found that the best way, for me, I could understand this maths was through YouTube videos graphically describing the formulas. In my opinion the YouTube channel 3blue1brown does an exceptional job of graphically displaying how these formulas operate in an clear and concise way, which I cannot recreate in this project. Therefore, I would highly recommend his series on artificial neural networks to clarify the mathematics found in this section of my project. His series on Neural Networks can be found at the link in the bibliography (3Blue1Brown, 2017).

To begin to explain forward propagation through a fully connected network, I am going to use a lot of diagrammatic representations of networks and a lot of matrices to express the calculations. Below (Fig 5) shows a single perceptron neuron encountered in the previous section.

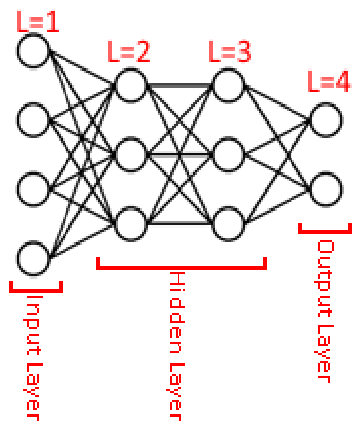
These single perceptrons are then chained together into layers with various quantities of neurons. In fully connected networks each single perceptron neuron is connected to all the neurons in the previous layer. Note that they usually are connected to all the neurons in the following layer too, as long as the next layer is also a fully connected layer. There is some important terminology used frequently in neural network papers, which describe the different layers in neural networks. The first layer is imaginatively named the "Input Layer" (analogous to sensory neurons), and the last layer the "Output Layer" (analogous to motor neurons). However, everything in between are named the "Hidden Layers" (similar to the interneurons) . It is 'hidden' because these layers are rarely interacted with by the user of the network, therefore, they are hidden. Figure 6 demonstrates these different network layers.

This example network is very small compared to many deep-learning networks. However, even for this small network there are 199 calculations to perform in order to forward propagate a single input and then backpropagate the network. Therefore, I will be using layers 3 and 4 to explain forward propagation, but the same principles learned can also be used for forward propagation through all layers of varying sizes, as long as they are fully connected.

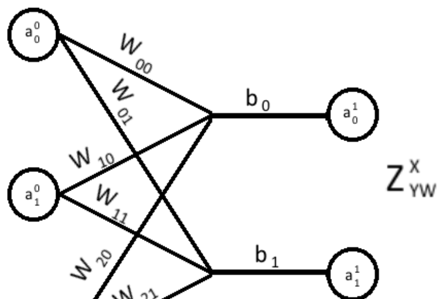
Fi
co



fully
3,3,2].



In this small network there are five neurons, seperated into two layers, with six neuron to neuron connections, interconnecting the layers. With these interconnections there are six weights and two biases to compute. As labelled below in Figure 8. Each weight, bias and activation can then be represented by a unique element in a mathematical matrix, this allows for fast scaleable calculations in many computing languages.



Z^X_{YW}

Where:
Z is the type of value, a activation, b bias, w weight.
X is the layer.
Y is the neuron in the current layer.
W is the neuron in the next layer.

This matrix calculations performed are multiplying the activations in layer zero by the weights then add the bias. The activations are represented by coordinate-vectors (horizontal vectors) with length equal to the number of neurons in that layer. The weights are represented with matrix of dimensions (3, 2) where the matrix height is equal to the number of neurons in the current layer and the width is equal to the number of neurons in the next layer. Finally the bias is also a coordinate-vector with length equal to the number of neurons in the next layer. This calculation is then put through an activation function, such as sigmoid, to obtain the coordinate-vector activations of the next layer. This can be written as follows:

$$\begin{bmatrix} a_0^1 \; a_1^1 \end{bmatrix} = \; activation\, function\left(\left(\begin{bmatrix} a_0^0 \; a_1^0 \; a_2^0 \end{bmatrix} \times \begin{bmatrix} w_{00} \; w_{01} \; w_{10} \; w_{11} \; w_{20} \; w_{21} \end{bmatrix}\right) + \begin{bmatrix} b_0 \; b_1 \end{bmatrix}\right)$$

Without the use of matrices the previous calculations looks like:

$$a_0^1 = \; activation\, function(a_0^0 \times w_{00} + a_1^0 \times w_{10} + a_2^0 \times w_{20} + b_0)$$

$$a_1^1 = \; activation\, function(a_0^0 \times w_{01} + a_1^0 \times w_{11} + a_2^0 \times w_{21} + b_1)$$

This calculation can be used for all fully connected layers in neural networks, with 'n' neurons in the current layer and 'm' neurons in the next layer. This works for every subsequent fully connected layer in the neural network: take the previous layer's activations, multiply by that layer's weight and add that layer's bias, followed by the activation function.

$$\begin{bmatrix} a_0^1 & \dots & a_m^1 \end{bmatrix} = \; activation\, function\left(\left(\begin{bmatrix} a_0^0 & \dots & a_n^0 \end{bmatrix} \times \begin{bmatrix} w_{00} & \dots & w_{0m} & \vdots & \ddots & \vdots & w_{n0} & \cdots & w_{nm} \end{bmatrix}\right) + \begin{bmatrix} b_0 & \dots & b_m \end{bmatrix}\right)$$

The final extension of this formula covers the forward propagation for batched inputs. This is for when more than one input is passed to a artificial neural network at the same time and all the inputs passed at the same time are collated into a batch matrix. Arguably this is one of the biggest advantages AI have over biological neural networks, where they can process masses of information at the same time and in a very short time period via mathematical matrices, accelerating the rate which AI can learn in comparison to ourselves.

The way this is computed is by extending the coordinate vectors for the activations in to 2D matrices, where the length represents the number of neurons, like the previous formula, but the height is equal to the number of inputs passed to the network in the batch. For the following formula 'n' is the number of neurons in the current layer, 'm' is the number of neurons in the next layer of the network, and 'p' is the number of input data in the batch.

$$\left[a[0]_0^1 \;\ldots\; a[0]_m^1 \;\vdots\; \ddots \;\vdots\; a[p]_0^1 \;\cdots\; a[p]_m^1\right] = activation\ function\left(\left(\left[a[0]_0^0 \;\ldots\; a[0]_n^0 \;\vdots\; \ddots \;\vdots\; a[p]_0^0 \;\cdots\; a[p]_n^0\right] \times\right.\right.$$

As previously mentioned, this same formula can be used for any layer in a fully connected network, just adjust the matrix dimensions to be suitable for that layer and the batch size suitable to the input, and the calculation will work. The following formula concludes the fully connected forward propagation maths by replacing the 0th layer and the 1st layer with 'L' and 'L+1' respectively, representing any 'L' layer in the network.

$$\left[a[0]_0^{L+1} \;\ldots\; a[0]_m^{L+1} \;\vdots\; \ddots \;\vdots\; a[p]_0^{L+1} \;\cdots\; a[p]_m^{L+1}\right] = activation\ function\left(\left(\left[a[0]_0^L \;\ldots\; a[0]_n^L \;\vdots\; \ddots \;\vdots\; a[p]_0^L \;\cdots\; a\right.\right.\right.$$

This is the final formula needed to begin computing multiple inputs to a fully connected artificial neural network. One thing to note at this stage is that all of these calculations are matrices. In maths, matrix multiplications are used to calculate spatial transformations such as rotations, reflections, skews, ect. and matrix additions are used for spatial translations. Therefore, abstractly, this is what a neural network is calculating in potentially many hundreds of spatial dimensions. This is not commonly mentioned in neural network descriptions, but mathematically they are certainly spatially transforming the input data you provide into more useful, less dimensional, spatial data.

In addition as previously mentioned the input data is provided in coordinate-vectors where each unique input represents a unique spatial point for the Neural Network much like how we use coordinates on graphs. Due to this Neural Networks can take in such a wealth of data, organise its structure in many dimensions, and therefore understand the patterns in it and this is something which the human brain cannot even begin to comprehend! This has given Artificial Neural Networks an advantage when dealing with 'big data', and has allowed it to surpass human performance already in many complex tasks, such as the AlphaGo-Zero neural network (Silver, et al., 2017) and (Hassabis, 2017). Below shows two images taken from a neural network's activation matrices where the higher dimensional data has been approximated into three dimensions using t-Stochastic Neighbour Embedding (Maaten, 2008).

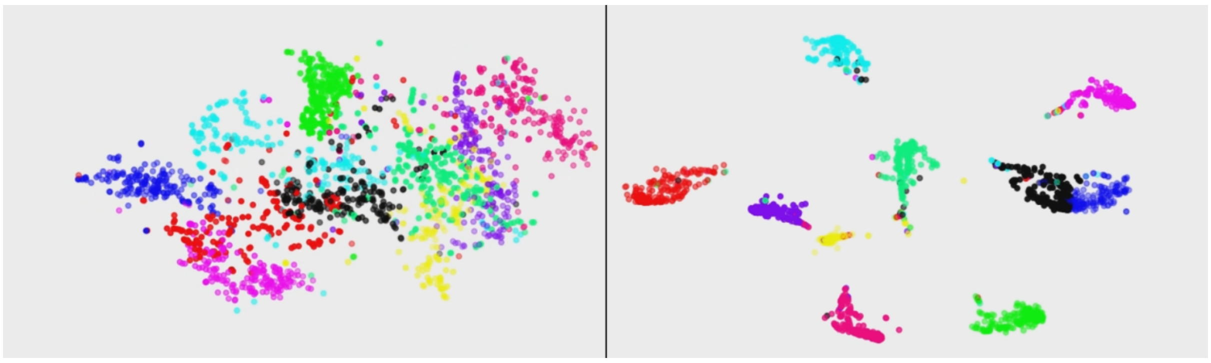


Figure 9: The image on the far left shows the 784D to 3D rendering of the input activations presented to the neural network, whilst the image on the right shows the resulting transformation of the input activations by the neural network into 10D data approximated into 3D coordinates by t-Stochastic Neighbour Embedding. Each colour signifies a different classification of the input data and it is apparent that the neural network has understood these categorisations and separated them using the matrix multiplications and additions into distinct spatial areas, seen in the right image.

This may seem like a bit of a tangent from the maths of forward propagation, however, I think it is important to bear in mind this idea throughout learning Artificial Neural Networks especially in this section and the next few on backpropagation. In other sources I have read there has been scarce mention of this dimension idea, but it is what the actual calculations inside a Artificial Neural Network are doing. I believe this insight will help a lot in developing Artificial Neural Network into the future and advancing their functionality, and is important to remember when creating any new neural network algorithm that it must be compatible with this dimensions idea otherwise the new algorithm will not be working optimally. It is this concept which the algorithm in Lateral Neural Pathway Manipulation utilises to preserve network learning when changing a network's structure.

Later in this project I will refer back to this dimension idea and expand upon it including how Lateral Neural Pathway Manipulation is derived from this understanding of a neural network's mathematical function.

How Neural Networks Learn.

Potentially thousands of weights and biases comprise an Artificial Neural Network's structure, all beginning with entirely random values when the network is created. As a result the network's initial performance on any task is very poor and the outputs are useless, semi-random noise. Thus the network must be trained through example to make adjustments to its parameters until an adequate output is reached. Many Artificial Neural Networks learn much like humans do, by attempting a problem and then being told how to improve by a more experienced overseer. This is called "Supervised Learning" and is what this section will cover.

As previously mentioned, a Neural Network learns through example and one of the most commonly used tasks to train your first neural network is recognising handwritten numbers. Specifically the MNIST dataset (LeCun, Cortes, & Burges, MNIST, 2013) is used to demonstrate how networks learn by presenting them with 60000 handwritten, 28 by 28 pixel, images of numbers zero through nine. The image below, figure 10, shows a selection of the handwritten digits provided in the MNIST dataset.

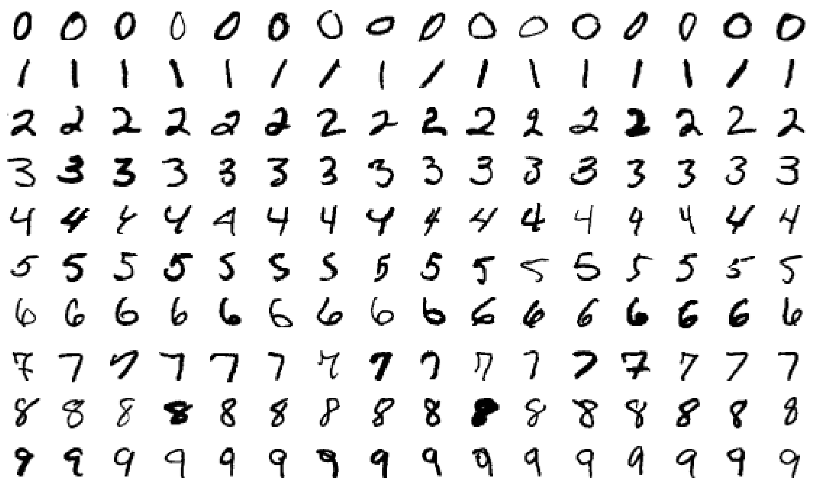


Figure 10: The MNIST Dataset (Source Wikipedia).

This number recognition task may seem trivial, however it is an amazingly difficult task for the human brain! Requiring communication between the optic nerve, visual cortices, left occipito-temporal cortex, superior temporal lobe and a number of other different brain regions, all working together to classify a distinct but highly variable pattern of pixels in an image. To program a computer, without the use of neural networks, to recognise these specific patterns is incredibly difficult due to the variations in each person's unique handwriting style. To accomplish tasks like MNIST requires a function which has thousands of parameters which can model complex inputs into simple data, much like our brains. Of course Neural Networks are perfect for this task.

However standard image data is not suitable for fully connected Neural Networks as they accept vector inputs only. Consequently, the MNIST images must be converted into vector form. To do this the 28 by 28 pixels images are divided up into its rows of 28 pixels and then these rows are stacked end to end to form a 784 element coordinate vector. Remember this vector represents a unique point in space for the neural network. This vector is then supplied to the network's inputs with activations such that the brightest pixels encode to numbers closer to one, whilst the darker pixels are closer to zero. The example below, figure 11, shows the conversion of a simple 3 by 3 pixel image to a vector.

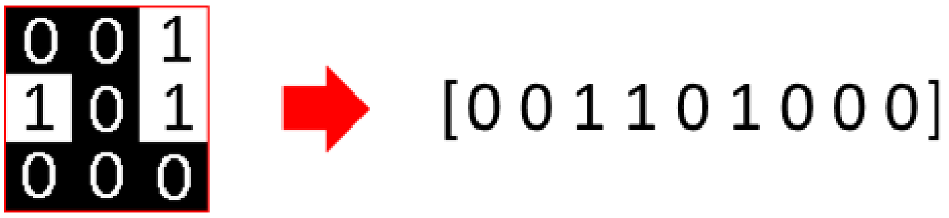


Figure 11: A simple image converted into a 9 element vector.

Once this image is 'vectorised' the data is passed to the untrained neural network, then forward propagation begins calculating each layer's activation based on the relevant weights and biases until the output activation is calculated. For an untrained network the output activation will be seemingly random noise and not give any indication as to what it thinks is in the image. Subsequently the network is most likely told it is wrong and the correct classification of the image is provided to the network by the supervisor. But how is the neural network meant to learn from this correct data? For this a function is needed called 'Cost' or sometimes called "Loss".

Cost is a measure of the neural networks performance across the entirety of a dataset. Cost describes the accuracy and uncertainty of the network in mathematical terms, by comparing the network's output with the desired output from the supervisor. There are a variety of different cost functions used in neural networks from 'mean-squared error' to 'cross-entropy', all of which quantify the network's success in a given task. Humans also have somewhat of a cost function, where our learned behaviours seem to be motivated by our surrounding environments. The cost function I will be referring to in this chapter is the mean-squared error. Below shows the formulas for both Cross-Entropy and Mean Squared Error. Where X is the output from the neural network and Y is the desired output from the supervisor.

$$Mean\ Squared\ Error\ Cost\ =\ \frac{1}{2n}\sum_{i=1}^{n}\left(X_i - Y_i\right)^2$$

$$Cross\ Entropy\ Cost\ =\ -\frac{1}{n}\sum_{i=1}^{n}\left(Y_i ln\left(X_i\right) +\ (1\ - Y_i)ln(1\ - X_i)\right)$$

Therefore, by minimising the value of the cost function the network's output more closely matches the desired output and the error is reduced. Each individual weight and bias can have a value between negative to positive infinity (theoretically), and there can be millions of neurons in hundreds of layers in a deep neural network. Therefore to find perfect values for every parameter to minimise cost the most for the entire dataset would be near impossible and if possible take a very long time to search through every permutation of parameters. Consequently an algorithm named 'Gradient Descent' is used to repeatedly change parameter values slightly moving them closer to the values required for lowest cost.

Gradient Descent ties in with the idea of neural network dimensions from earlier. Now this is where it gets a little intangible, every parameter and every input represents an independent spatial dimension to the backpropagation algorithm, and as the word 'Gradient Descent' implies these dimensions have an independent height dimension which can be descended. The cost function is dependent on every input and parameter, and each unique position along every spatial dimension has a corresponding cost value. Across every position in all dimensions these cost values represent a height and it forms a 'surface' through these multitude of spatial dimensions where the highest cost values (highest error) represent the highest points and the lowest cost values (lowest error) the lowest point.

This forms the cost surface, and like in real life if you place an object at any point on a sloping surface and apply gravity the object will move to the lowest point. If this is applied to the hyper-dimensional cost surface, gradient descent is the gravity, and the neural network is the object on the surface, the neural network will move to the point of lowest cost, or a local minimum in cost, and reduce the network's error. Fig 12 shows how the neural network moves down the cost surface gradient to reach the local cost minimum.

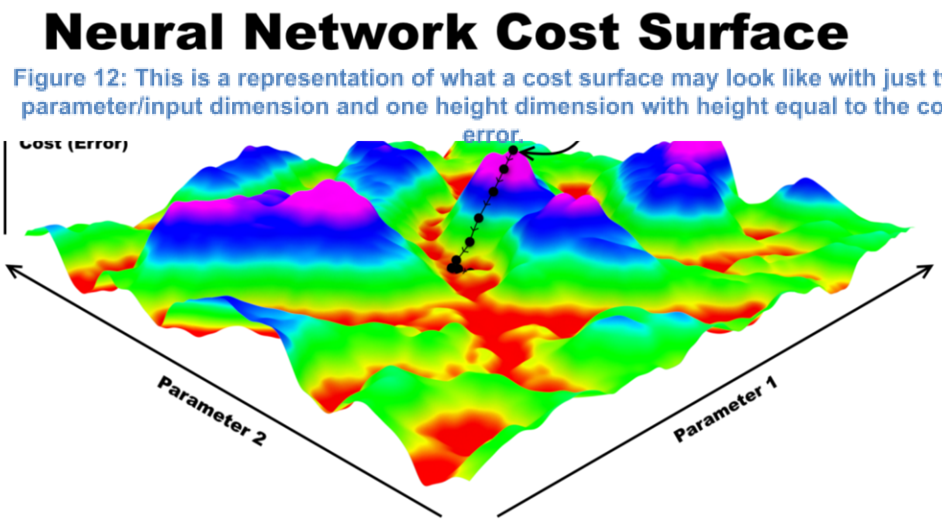


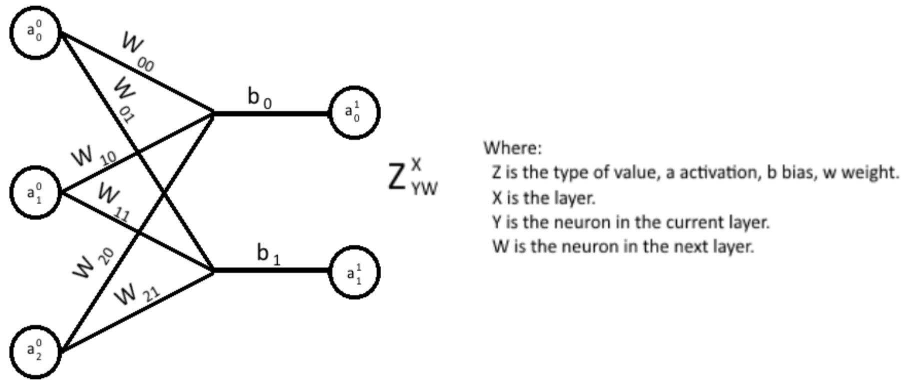
Figure 12: This is a representation of what a cost surface may look like with just two parameter/input dimension and one height dimension with height equal to the cost error.

Hence it most likely has now solved the dataset classification and is a 'trained' neural network. This is how gradient descent and backpropagation effectively work, but instead of three dimensions you have potentially millions, and the maths for making gradient descent work through all these dimensions separated into abstraction layers is rather difficult!

The difficulty arises when calculating the gradient of the cost function through each layer of the neural network. Effectively the network is ran in reverse, but instead using derivatives of the functions. Backpropagation is short for 'The backward propagation of error gradients' and this is almost opposite to the forward propagation described earlier. Backpropagation is arguably the biggest improvement ever in neural network history as it made training the networks feasible, and therefore neural networks viable. It uses the derivative chain rule to calculate the derivatives (gradients) of weights, biases and activations through each stage of the network, and then changes the parameters to move the network down the cost surface slope. It begins at the cost function, where the gradient of the cost function in respect to the network output is calculated. The gradient can be positive or negative now as the squared is removed, therefore the 'uphill' gradient is positive, and the 'downhill' negative. The larger the magnitude of a parameter's error gradient, the more it contributes to the final network's error. Therefore it indicates the parameter's importance in the network.

$$C(X,\ Y) = \frac{1}{2n}\sum_{i=1}^{n}\left(X_i - Y_i\right)^2 \quad \rightarrow \quad \frac{\delta C(X,Y)}{\delta X} = \frac{1}{n}\sum_{i=1}^{n}(X_i - Y_i)$$

As mentioned to move the network along the cost surface the weights and biases must be changed to move down the gradient to reduce the error. Therefore the gradient of error in respect to these parameters must be calculated using the chain rule. Once again, I will be using the smaller 3 to 2 neuron network used to describe forward propagation shown below (Fig 13).


a^0_0
a^0_1
a^0_2
W_{00}
W_{01}
W_{10}
W_{11}
W_{20}
W_{21}
b_0
b_1
a^1_0
a^1_1
Z^X_{YW}
Where:
Z is the type of value, a activation, b bias, w weight.
X is the layer.
Y is the neuron in the current layer.
W is the neuron in the next layer.

Before calculating the gradients for backpropagation, it is important to unpick the steps used in forward propagation up to where the cost is calculated. For this stage I am going to refrain from using matrix calculations as it can overcomplicate the derivatives at this stage. Instead I am going to use 'a' for an activation vector, 'L' for the layer, 't' the desired output or target output, 'w' for the weight matrix, 'b' for the bias vector, and 'z' for a temporary vector used to store data before the activation function. Thus the forward propagation process is simplified to the following set of equations. I am going use sigmoid for the activation function, and mean-squared error for the cost function.

$$z^L = \left(a^{L-1} \times w^L\right) + b^L$$

$$a^{L+1} = \frac{1}{1+e^{-z^L}}$$

$$C\left(a^{L+1},\ t\right) = \frac{1}{2n}\sum_{i=1}^{n}\left(a_i^{L+1} - t_i\right)^2$$

This process now must be differentiated in respect to the activations, weights and biases. As previously mentioned the differentiation chain rule is needed. In this section I am going to presume prior knowledge of the chain rule, as explaining the chain rule will lengthen tphis extended project considerably. Below shows the gradients needed in backpropagation and the respective chain rule derivatives needed for the above example.

$$\frac{\delta C\left(a^{L+1}, t\right)}{\delta w^L} = \frac{\delta C\left(a^{L+1}, t\right)}{\delta a^{L+1}} \times \frac{\delta a^{L+1}}{\delta z^L} \times \frac{\delta z^L}{\delta w^L}$$

$$\frac{\delta C\left(a^{L+1}, t\right)}{\delta b^L} = \frac{\delta C\left(a^{L+1}, t\right)}{\delta a^{L+1}} \times \frac{\delta a^{L+1}}{\delta z^L} \times \frac{\delta z^L}{\delta b^L}$$

$$\frac{\delta C\left(a^{L+1}, t\right)}{\delta a^L} = \frac{\delta C\left(a^{L+1}, t\right)}{\delta a^{L+1}} \times \frac{\delta a^{L+1}}{\delta z^L} \times \frac{\delta z^L}{\delta a^L}$$

Note that for all these differentials, the first two products in the chain rule are identical, this reduces the amount of calculations required massively, as they only have to be calculated once per cycle of backpropagation. This method of backpropagation was one of the biggest developments in neural network technology, and made the use of neural networks on large and complicated datasets viable, outperforming all other Artificial Intelligence methods. It also functions similarly to the Gauss-Newton Algorithm (Wikipedia, Gauss-Newton Algorithm).

By calculating these derivatives it indicates how each parameter affects the final error of the network. The greater the magnitude of a specific parameter the more it is responsible for the current high cost, and its sign indicates in which direction gradient descent should occur to reduce this cost. The two common differentials across all parameters in a neural network layer have been expanded below to show the formula needed to calculate them.

$$C\left(a^{L+1},\ t\right) = \frac{1}{2n}\sum_{i=1}^{n}\left(a_i^{L+1} - t_i\right)^2 \quad \rightarrow \quad \frac{\delta C\left(a^{L+1}, t\right)}{\delta a^{L+1}} = \frac{1}{n}\sum_{i=1}^{n}\left(a_i^{L+1} - t_i\right)$$

$$a^{L+1} = \frac{1}{1+e^{-z^L}} \quad \rightarrow \quad \frac{\delta a^{L+1}}{\delta z^L} = \frac{1}{1+e^{-z^L}} \times \left(1 - \frac{1}{1+e^{-z^L}}\right) = \frac{e^{-z^L}}{\left(1+e^{-z^L}\right)^2}$$

Despite this common derivative in all parameters, each weight and bias gradient for every neuron in each layer of the network must still be derived requiring large amounts of computation time. Luckily matrices reduce this issue with fast matrix libraries such as Python's numpy, allowing all the weight gradients to be calculated simultaneously, along with the biases simultaneously, and activations too. For now though I will stick to using the letter representations for the matrices to make the formulae more intuitive.

Below shows the final equation which must be differentiated, and the results of differentiating this equation with respect to each of the variables.

$$z^{L} = \left(a^{L-1} \times w^{L}\right) + b^{L}$$

$$\frac{\delta z^{L}}{\delta a^{L-1}} = w^{L} \qquad \frac{\delta z^{L}}{\delta b^{L}} = 1 \qquad \frac{\delta z^{L}}{\delta w^{L}} = a^{L}$$

You may be wondering why the activation gradient is calculated, if only the weight and bias parameters are adjusted in backpropagation. In the 3 to 2 neuron network there is only the input layer and output layer therefore there is no need to calculate the 3 neuron layer activation gradients. However if this network is expanded with more neuron layers then this activation gradient needs to be calculated to allow the backpropagation of errors to be extended further back in the network and this allows parameters in all layers of the network to be adjusted.

The below equations show the same calculations previously mentioned but with the use of matrices instead. This is useful for when coding a neural network software from scratch. Note that some of the matrices are transposed (indicated by the superscript 'T') which allows them to be multiplied with the gradients matrices.

$$\frac{\delta C\left(a^{L+1}, t\right)}{\delta a^{L+1}} = \frac{1}{n}\sum_{i=1}^{n}\left(\left[a[0]_{0}^{L+1} \ \dots \ a[0]_{m}^{L+1} \ \vdots \ \ddots \ \vdots \ a[p]_{0}^{L+1} \ \cdots \ a[p]_{m}^{L+1}\right] - \left[t[0]_{0}^{L+1} \ \dots \ t[0]_{m}^{L+1} \ \vdots \ \ddots \ \vdots \ t[p]_{0}^{L+1} \ \cdots\right.\right.$$

$$\frac{\delta a^{L+1}}{\delta z^{L}} = sigmoid\left(z^{L}\right) \times \left(1 - sigmoid\left(z^{L}\right)\right)$$

$$\frac{\delta a^{L+1}}{\delta a^{L}} = \frac{\delta a^{L+1}}{\delta z^{L}} \times \left[w_{00}^{L} \ \dots \ w_{0m}^{L} \ \vdots \ \ddots \ \vdots \ w_{n0}^{L} \ \cdots \ w_{nm}^{L}\right]^{T}$$

$$\frac{\delta a^{L+1}}{\delta w^{L}} = \left[a[0]_{0}^{L} \ \dots \ a[0]_{n}^{L} \ \vdots \ \ddots \ \vdots \ a[p]_{0}^{L} \ \cdots \ a[p]_{n}^{L}\right]^{T} \times \frac{\delta a^{L+1}}{\delta z^{L}}$$

$$\frac{\delta a^{L+1}}{\delta b^{L}} = \frac{\delta a^{L+1}}{\delta z^{L}}$$

Now this algorithm for calculating gradients can be repeated for each layer in the network, backpropagating the error through the network, and finding how parameters must be changed to reduce the network's error and increase its performance. From the gravity analogy, these calculations effectively calculate how strong the 'gravity' on the neural network object is and which direction the force should be applied to move object down the slope. However the algorithm for actually moving this object is not described in the above calculations. This final step concludes the gradient descent algorithm.

To move the neural network along the cost surface, each parameter must be slightly adjusted in proportion to its error gradient. This is simply done by subtracting a proportion of the error gradient from the parameter value as shown below.

$$w^{L} \rightarrow w^{L'} = w^{L} - \eta\frac{\delta C\left(a^{L+1}, t\right)}{\delta w^{L}}$$

$$b^L \rightarrow b^{L'} = b^L - \eta \frac{\delta C(a^{L+1}, t)}{\delta b^L}$$

The variable eta 'η' defines the proportion of the error gradient to subtract. It is also known as the learning rate and defines how much the parameters should be adjusted per gradient descent cycle. When the learning rate is too high the network reduces general error quickly but will struggle to distinguish data in the dataset, if it is too low it will take excessive time for the neural network to learn. Therefore this parameter must be optimised for each new network and it is somewhat guess work when training a network what value to use. Often η=0.01 seems to be fairly successful in networks I have made.

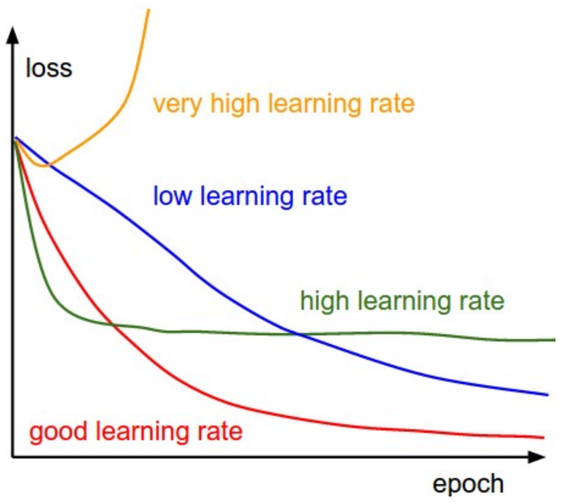


Figure 14: A diagram indicating the general learning patterns for a neural network with varying learning rates, against time (epoch).

This concludes the algorithms used in gradient descent. Note that gradient descent must be used repeatedly on a network to train it, it may requires thousands of cycles (epochs) of gradient descent before a network begins to solve its specified task. Each epoch moves the network slightly along the cost surface eventually nearing the local minimum, if it moves too fast it may overshoot this minima and end with higher error. However it is important to reduce the amount of epochs needed to train a network without overshooting the local minima, primarily for economic reasons. The faster a network is operational; the sooner it can be of benefit to a company. There are many techniques of reducing the training cycles, such as the use of 'momentum' or 'dropout' in gradient descent.

Problems When Training Neural Networks And Their Solutions

There are a multitude of problems which a network can encounter when training it to solve a dataset problem. These problems include slow learning, overfitting of data, misclassified data, or a network which does not have the correct classification available for the data. To begin with, momentum tackles the problem of slow learning in a neural network.

Momentum (Rumelhart, Hinton, & Williams, 1986) as the name implies, again takes the concept of real world physics and applies it to the neural network on the cost surface. Usually when gradient descent is used, the network makes a single 'step' along the cost surface then stops and waits for the following cycle before moving proportionally to the error gradient again. However, the momentum algorithm gives the neural network object a model of velocity. The neural network initially starts at rest with zero velocity, then the error gradients are added to this velocity to start the network moving. Each epoch of training the error gradients are again added to this total velocity, and the parameters are adjusted proportionally to the velocity not the error gradient. This allows the neural network to gain 'momentum' moving down a slope and keep it this speed across a flat part of the cost surface, which would otherwise leave the neural network stranded for a long while without the use of momentum. This idea also allows a neural network to roll out of a higher local error minima and continue down the slope to reach a lower error, as shown below (Fig 15).

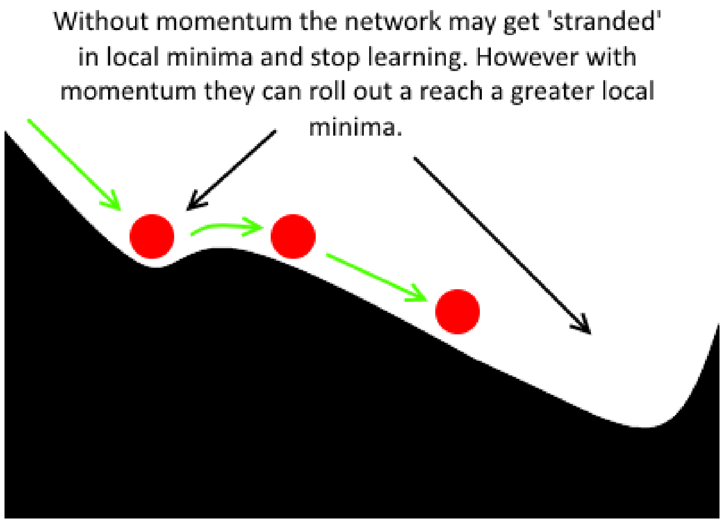

Without momentum the network may get 'stranded' in local minima and stop learning. However with momentum they can roll out a reach a greater local minima.

Figure 15: This shows how momentum prevents a neural network from being 'stuck' in a local minima.

To prevent the velocity of the neural network exponentially increasing and overshooting the minimum error points, friction is applied to the neural network. This is a very basic model of friction, where the velocity is reduced by a certain percentage each epoch. This momentum feature is incredibly useful in gradient descent and speeds up network training considerably so is widely used.

Dropout (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov) is another method of improving Neural Network learning. Instead of increasing the rate of learning like momentum, it increases the generalisation of the learning by reducing overfitting of data. Overfitting is a problem with regression algorithms, where the regression curve too closely matches the data rather than finding the general trend of the data. This is a problem in many subjects and is commonly referred to as Occam's razor (Wikipedia, Occam's razor). This makes the regression algorithm superb at classifying known examples of the data, but it has very low accuracy at classifying new data. This is easiest to see diagrammatically as shown below (Fig 16).

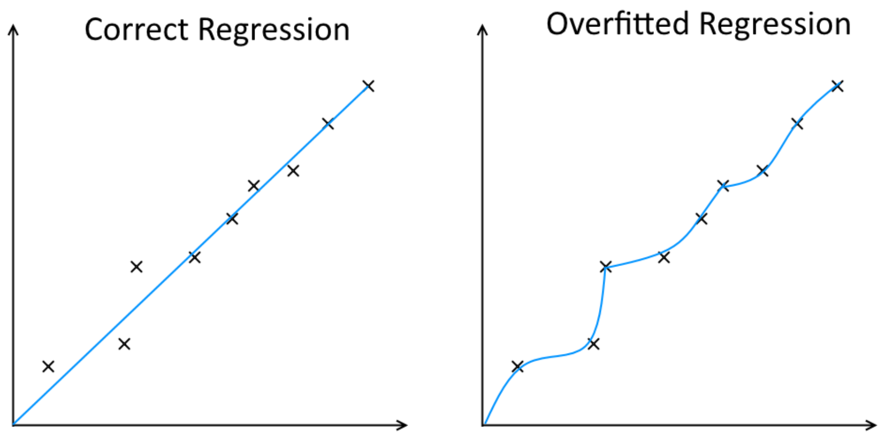
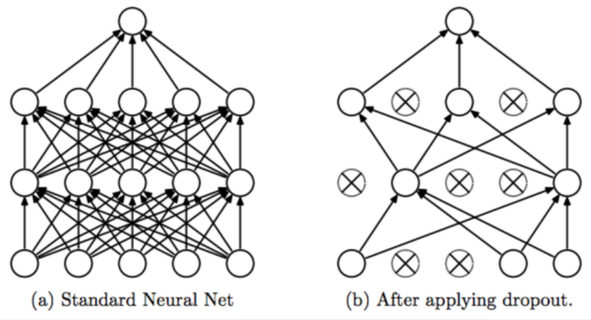


Figure 16: Shows the difference between overfitted and correct regression of data.

Although the data is very variable, it is clear that the regression line on the left is more likely to be accurate at predicting new data rather than the overfitted regression line on the right. The overfitted data is excellent at classifying known data points, but has very little accuracy when interpolating or extrapolating the trend. Neural Networks tend to suffer from this overfitting issue on datasets and this is why dropout is used.



Dropout tackles this issue by temporarily disabling a random proportion of the neurons in a neural network, each epoch. Normally neural networks tend to reinforce a certain pathway through the network, and subsequently becomes too reliant on this pathway to solve the dataset problem. By randomly removing certain neurons in the network, this can disrupt the reinforced neuron pathway and force the network to 'reroute' the activation signals around adjacent neurons to solve the task, as shown right in figure 17. This increases the learning down alternative pathways and prevents the network becoming too reliant on specific neurons to function. After many epochs of temporarily disabling random proportion of the network, the network should have completely stopped relying on reinforced pathways and instead can use any combination of its neurons to solve the dataset problem. Once training the network has finished all neurons are re-enabled and their weights are reduced in magnitude in proportion to the percentage of neurons disabled each epoch. Reducing the weights prevents the network from being 'over stimulated' by the activation signals. This greatly reduces the amount of overfitting problem in the network, and makes the network much better at generalising its learning to identify new data.

As mentioned previously, another issue with neural network training is the misclassification of data, or the lack of a correct classification available for the data. One of the simplest examples for this can be imagined for the MNIST handwritten number dataset. The MNIST dataset contains handwritten numbers from zero to nine, a misclassification error would be when a handwritten four is misclassified as a nine for example. To my knowledge this is not a problem for the MNIST dataset as it has been rigorously reviewed for these types of errors, but it could potentially be a big problem when training a neural network on less accurate datasets. This is because the neural network backpropagation assumes the training data is always correctly classified, therefore it will still try and learn from the misclassified data. As a result the learning quality for the misclassified category and the actual category is damaged and should be avoided to ensure optimum network performance.

Another similar issue is the combination of misclassified data and the lack of a correct classification available. Returning to the MNIST idea, an example of this error would be the handwritten letter 'A' being classified as a three. This becomes an issue as the theoretical network would only expect to receive handwritten numbers so would not have the capability to begin classifying letters too, especially due to the limited output neurons. This dataset error would also damage the learning quality of the misclassified category, so again it needs to be avoided.

For the MNIST example, the misclassification of data is not too important for the network. However for other type of datasets it could yield devastating effects for the neural network learning. Also some benefits could occur from finding a solution to these problems. For example neural networks are often used to identify the species of animals and plants. If a new species were to be scanned by a neural network, instead of producing a misclassified result, it would be an advantage for the network to realise it has a new distinct classification which is previously unseen. Therefore, finding a solution could allow the emergence of new neural network technology and further advancements.

An approach to addressing this problem needs to be able to reduce the backpropagation algorithm's dependence on the classifications of data. Learning and backpropagation must also be unaffected by changing the organisation and quantity of neurons in the network to allow new classification outputs to be generated. Currently learned behaviour in the neural network is incredibly sensitive to this changing of network topology. Finally the network must be able to detect misclassified data and new distinct data in the dataset. From these principles I have designed Lateral Neural Pathway Manipulation Networks.

Introduction Conclusion

With ever more frequent breakthroughs and growing interest from the scientific community, I hope the potential and prospects of Artificial Neural Networks is apparent, and the great benefit it could bring to the world. Evidence for this approach for Artificial Intelligence already supported by the highly successful biological networks, devised from millions of years of evolution, from which ANNs have been inspired. With new neural network structures, optimisations, and applications are quickly solving many of the remaining problems the networks face.

The intention is for LNPM to be primarily used for restructuring a neural network without losing learned data, however I believe it may also be a good approach at tackling the aforementioned classification issues. LNPM networks are somewhat inspired by how the mammalian brain restructures neural connections to allow new learning to occur. The mammalian brain processes have been refined over millions of years of evolution, so I believe taking interpretations of these processes into artificial neural networks should yield good results.

The next chapter of this extended project will explain the details of the Lateral Neural Pathway Manipulation Networks in more depth, clarifying its purpose, the maths underpinning it, and the problems it is designed to address.

Lateral Neural Pathway Manipulation

The Inspiration and Aim for LNPM

The initial issue with Neural Networks which stimulated me to design LNPM, is the difference between how human and artificial neural networks learn new skills. Humans are born with the capacity to acquire any new skill needed throughout their lifetime, whereas artificial neural networks have neural architecture predesigned to solve a known required task, a so called 'specific-purpose' network.

If a completely new manner of problem is presented to this 'specific-purpose' network, it will most likely be unable to solve it. Not least because the existing network has a constant neuron architecture which is inflexible to change, unlike our biological network which easily generates new neurons and connections to accommodate this new skill without impeding the rest of the network. LNPM will not be a solution to creating a general-purpose artificial intelligence, but the algorithms may be useful for allowing a neural network to change topology easily without impeding the existing network's function.

The idea stemmed from how I imagined early human cavemen would have been analogous to these 'specific-purpose' network, where the brain's structure was predisposed to be excellent at a few lifesaving skills, such as gathering food, sourcing water ect. However over millions of years numerous new expertise were acquired, and the adaptable structure of the brain allowed different brain regions to grow. Hence enabling the development of language, music, art, mathematics, and many more avenues of skills.

Consequently modern day humans are now analogous to general-purpose intelligence where any multitude of new skills can be learned without the brain's structure being predetermined to learn it. Our brains can add new neurons, strengthen existing connections (through myelination), and prune unused connections (constantly!) to enable us to learn, without ever effecting the functionality of our brain to solve other tasks.

Flexible neural architecture allows for the possibility for a network to learn from vast and different datasets which it wasn't initially intended to learn from. The network can extend its capacity to acquire these new skills, unused regions of the network can also be pruned to make the network more efficient. Overall I believe the ability for a network to have vary its architecture should be incredibly beneficial, and allow the development of much more advanced networks.

As previously mentioned it should also aid in the problem of misclassified data, as the algorithms for LNPM should pick up this distinct anomalous data and be able to generate new neural pathways and output neurons to correctly create its own category for future use. Again if this new category is unused, it will simply be pruned from the network.

Requirements To Make The Algorithm Successful

To make the Lateral Neural Pathway Manipulation algorithm to work as desired, it will need several smaller functions to make adjustments to the whole network. These adjustments include 'Pathway Addition, 'Neural Pruning', 'Frequency Layers' and a few changes to the backpropagation algorithm.

'Pathway Addition' will be a very simple algorithm which can append a parallel chain of neurons onto an existing network. The new pathway will need to be especially sensitive to data which closely matches the input data which triggered the pathway addition. Hence it must produce a strong positive activation when similar data is presented again.

However if a new neural pathway was simply appended onto the existing network, stray activations could transfer from the pathway to this original network. This would compromise the learning and functionality of the existing network because the learned parameters may be highly tuned to existing activations and disrupted by any extra activations. Therefore, this pathway must be comprised of two differently initiated neurons I have named 'Pathway Neurons' and 'Scaffold connections.

'Pathway neurons' will connect from all neurons in the previous layer to the single neuron in pathway of the next layer. These pathway neurons should have its parameters initiated to be extra responsive to the data resembling the input data which triggered the pathway addition. Therefore allowing high activation down the pathway if similar data is presented again, and otherwise low activation to prevent the network misclassifying data.

'Scaffold connections will connect from the pathway to all neurons in the next layer belonging to the original network. These neurons will be initialised with parameters so that no activations will transfer from the pathway to the existing network. In affect this isolates the new pathway from the original network, as activation signals can only forward propagate. You may question why these scaffold connections are created if not to be used. The intention is that they will not be inactive when the pathway is initially generated, however as the network continues to learn, backpropagation should theoretically slowly activate these connections allowing activations to travel through them. This reactivation will be a slow process though and require many epochs until the scaffold function as a normal connection. This gradual reactivation should allow the pathway and original network to slowly merge into one network over time, but at a rate which allows the original network to adapt and not lose functionality and learned parameters.

Below shows a diagram (fig 18) of the stages of pathway addition:

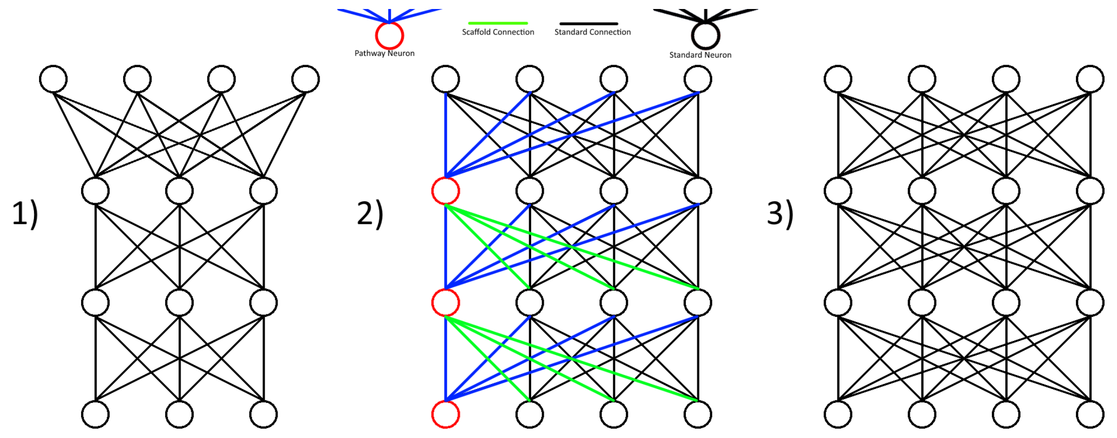


Figure 18: Stages of pathway addition. Stage 1 shows the original network before pathway addition is triggered. Stage 2 shows how the pathway is generated and where the different types of connections are formed. Stage 3 shows how this new pathways slowly merges with the original network and the scaffold neurons activate to form one complete network.

The next function required is 'Neuron Pruning', this is intended to prevent excessive growth of the network. Neural Pruning will remove specific neurons which have a consistently low magnitude of activation, as they are not contributing to the network, so are unneeded. Pathway addition could potentially add a multitude of new useless pathways. Also the larger the network is; the longer it takes to learn, so minimising the number of neurons is important. Therefore neural pruning will remove unused segments of the network to make it more efficient and prevent too much growth.

However just removing the neuron without compensating the network for the loss in activation will cause errors throughout the network. Therefore an approximation for the neurons average activation must be accounted for in the following neuron layer. Every neuron in the network supplies some average activation, however small it may be, to all its forward propagating neurons. Simply removing a neuron will consequently remove this average activation and will lower the activation in the next layer of neurons. This will have a cascade effect and cause damage to the learned parameters in the network. Therefore it is essential to compensate the network for this loss.

Artificial neuron pruning is somewhat analogous to the synaptic pruning in mammals (America, 2017). Like LNPM the biological brain creates an over abundance of neurons to aid in learning, however this quantity is not needed. To make the brain processes more efficient synaptic pruning decays specific unused axons and dendrites until they completely die off. This mechanism occurs mostly in young infants and adolescents. It simply removes unused connections, to streamline and speed up the important connections. Below (Fig 19) shows the stages of neuron pruning for the artificial neural network.

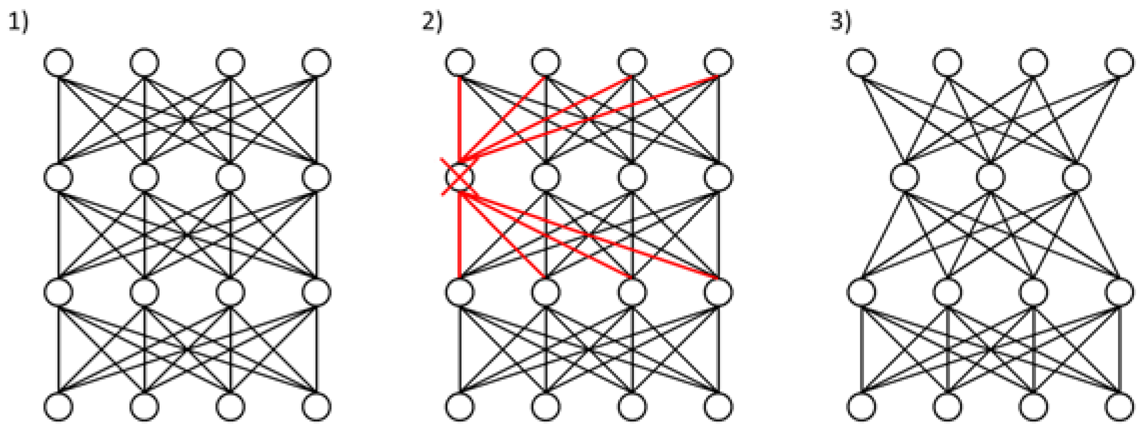


Figure 19: Stages of Neuron Pruning. Stage 1 simply shows the network before neuron pruning. Stage 2 shows the unused neuron and the corresponding connections which must be compensated for when the neuron is removed. Stage 3 shows post neuron pruning, and how the network's parameters have been compensated.

To determine when a neuron needs removing, an importance value must be assigned to every neuron in the network. This takes the form of the 'Frequency Layers'. The frequency layer is intended to express how often a neuron is active. This is done by taking the sum of the magnitudes of each neuron's activation. Therefore, each neuron will have a specific value which represents how often important the neuron is in network decisions. Neuron pruning will occur when the frequency value for a specific neuron becomes a statistical outlier, well below the frequency values of all other neurons. This will indicate the neuron is abnormally inactive, so can be pruned from the network.

The frequency values will also have a simple decay factor, which slightly reduces the frequency each epoch. This prevents the values getting too large, and also makes the frequency vector more representative of current neuron activation trends.

Another use for the frequency layers could be to accelerate learning down important pathways. This could potentially be unneeded in Lateral Neuron Pathway Manipulation, but maybe beneficial to the network. This could take the form of making the activation of the neuron proportional to the frequency value. For neurons with a high activation frequency, the activation would be boosted stronger, making it learn faster as a consequence. Whilst neurons with low frequency values have their activations reduced, lowering their importance in the network, potentially initiating pruning. This has the effect of reinforcing the most important neuron pathways, whilst diminishing the less important neurons.

In effect this acts like the saying: "If you don't use it, you lose it." Unused pathways simply won't learn as fast, and important pathways are prioritised when learning and producing network outputs. This again, somewhat, mimics myelination (Salzer, 2016). Myelination occurs in all hinged-jaw vertebrates, and its function is to increase the speed and efficiency of nerve impulses down the axons. The myelin helps to preserve the electrical signals down neuron, similar to boosting the activation of artificial neurons with the frequency values. Using frequency values to increase activation will need to occur after the bias is added to the activation, but before the activation function is used. This ensures the greater activation will still be between zero and one.

The average proportion by which the activation is changed must also have a mean multiplying effect of 1. Hence the proportion of neurons whose activations is boosted will be roughly equal to the proportion which is deteriorated. This prevents a neuron layer from being excessively active which could generate errors. Therefore it must be balanced to keep the network working at optimum performance.

Finally a few changes to the backpropagation algorithm must be made to allow variable number of output neurons. This will require a change in the cost surface function. This change will need to serve two purposes, the first to detect when pathway addition should occur, and the second to determine how to train neurons in the new pathway.

Detecting when lateral pathway addition should occur, should be a fairly simple task. When a neural network outputs a stochastic batch of data, whilst it is training, the cost function will be computed for each data example in the batch. If one of these cost functions is so large that it is a statistical outlier compared to the rest of the batch, then lateral pathway addition will be triggered. When the cost is very high, it indicates the network is inaccurate, and uncertain. If the cost is also a statistical outlier then it indicates that the input example could potentially be misclassified data, requiring a new output to correctly classify it. So a pathway is generated.

The second function is to attempt to train new lateral pathways correctly. This problem becomes clear when thinking about an example dataset such as MNIST. MNIST contains 60000 images of handwritten numbers, and a corresponding classification for each one of them ranging from zero to nine. Therefore a network with ten outputs can easily represent all these possibilities, and can be trained using the ten available classifications from the dataset. Now for example a new distinct character could occur in the dataset requiring its own new output, resulting in pathway addition. For this network it has eleven outputs from the network, but only ten classification options provided by the dataset. So how would this eleventh output be trained when only information for ten outputs is supplied? This requires a change to the 'target vector' supplied from the dataset.

The target vector is usually a 'one hot array' (Wikipedia, One-Hot), where the one represents the value the network should try to produce, this is how the neural network learns and generates a cost surface. The below equations shows the target vector represented as 't' in the mean squared error cost function.

$$C\left(a^{L+1},\ t\right) = \frac{1}{2n}\sum_{i=1}^{n}\left(a_i^{L+1} - t_i\right)^2$$

However with the new eleventh neuron, for the MNIST example, the 't' value must be influenced by the networks result, to generate a target value for this new output. One way I thought of doing this, is by taking a weighted average of the network's output vector and the target vector supplied by the dataset. Where the extra values needed for the target vector are set at zero unless the network's highest output is from one of the new neurons generated by lateral pathway addition. If the most certain classification by the network is one of the new neurons, then instead a one is the value set in its corresponding index of the target vector instead of a zero.

In the weighted average, the weights assigned to both the target vector and network's output activation are proportional to their certainty. So the closer the values are to one or zero, and furthest from 0.5, increase the weight value in the weighted average. In affect this means the more certain the network is of its outputs, then the stronger the average will be in favour of the network's output rather than the one supplied from the dataset. This weighted averaged target is then used to replace the original target, provided from the dataset, in the cost function. As a result the network has the ability to somewhat train itself, if it becomes very certain of its classifications, and takes little notice of the targets supplied from the dataset. The majority of the time the network will learn from the dataset like normal as its certainty will be low, but when its certainty becomes higher it will trust itself more with the correct classifications to learn from.

As a result the new classifications generated can be trained just like the original neurons linked to the dataset. These amendments will also allow a good way of determining when pathway addition should be activated.

Overall these outlined ideas encompasses all the functions desired for Lateral Neural Pathway Manipulation. The next section will describe some potential problems to consider when constructing the mathematical algorithms from these summarised functions. Primarily how they must be compatible with the dimensions idea, explained earlier, which arises from the backpropagation algorithm.

Considering the Cost Surface

Throughout designing the algorithm for Lateral Neural Pathway Manipulation, I have put particular emphasis on how it interacts with the dimensions concept in backpropagation. I have a strong opinion that algorithms which do not disrupt the gradients in backpropagation, too much, should preserve learning and consequently yield better results. When referring to the dimensions 'concept' or 'idea' it is important to stress that gradient descent really does work using these dimensions and it is entirely what the maths describes, not just an idea to make it backpropagation tangible.

As previously mentioned backpropagation, more specifically gradient descent, takes all the parameters of every neuron in the network and treats them as a dimension of space. Where every position in these spatial dimensions correlates to a specific, unique variation of the network. Inside this space every possible configuration of the network's parameters exists extended to positive and negative infinity in every parameter dimension. Then the gradient descent function is used to navigate these dimensions and find the optimum network configuration.

It does this using the cost surface. Ideally the cost surface is created from every example in the dataset and averaged to find a height corresponding to every point in the parameter dimensions, then the minimum point is found. However the cost surface is instead estimated usually using a small sample of the data, and only calculated at the neural network's current position inside the space. This saves an infinite amount of computation time.

Once the local cost surface is estimated, the gradient of the surface is calculated at the position of the network in the dimension space, it is key to think of the cost surface as an abstract height in the space. Once the gradient is calculated, gradient descent acts like gravity and moves the neural network down the cost surface slope. The neural network can be thought of as a point object in this space sitting on the cost surface. Moving the neural network point in this space corresponds to a change in the parameter values for the neural network.

By definition of the cost function, the highest points of the surface represent when the network has the highest error and consequently hasn't learnt anything. Logically moving the network to a lower point, using gradient descent, will result in it learning. Below (Fig 20) shows five neural networks traversing these parameter dimension.

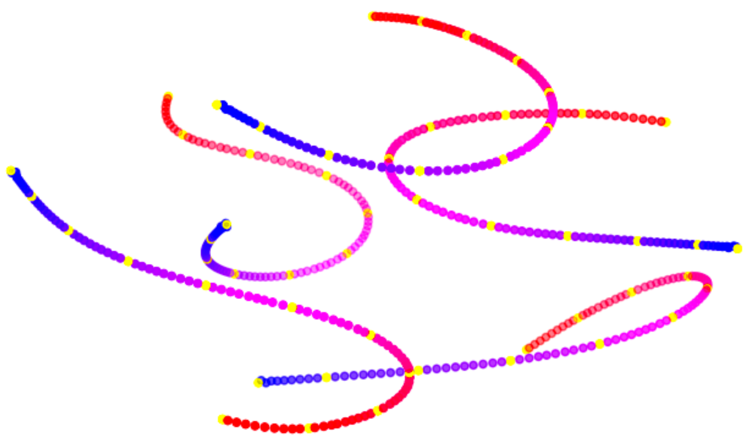


Figure 20: This shows the tracks of five different neural networks moving through parameter space as they learn. Red dots indicate the position of the neural network closer to when it was created, blue dots show it closer to when it had finished learning and yellow dots indicate the position of that specific network every 200 epochs as a time reference. This figure was created using all 16330 parameters of a specific network type, meaning it represents a 3 dimensional approximation of 16330 dimensional space using t-Stochastic neighbour Embedding. The cost surface could not be

represented in the diagram due to its high complexity, but the shapes and curvature of the paths indicate the cost surface shape: red areas being high cost, and blue lower cost.

I hope it is evident from the diagram that this dimension concept is truly how neural networks learn skills. As a neural network's sole purpose is to learn how to solve problems, then it is essential that any new neural network algorithm must be derived so that it minimises change to the shape and dimensions of the cost surface whilst the network is learning. Any disruption to the cost surface has the potential to reduce network's functionality.

LNPM functions whilst the network is running and learning, so it is even more critical when creating the algorithms that best effort be made for it to be compatible with backpropagation to prevent learning loss. When neurons are added and removed in LNPM, parameters are created and lost. As a result the number of dimensions for backpropagation changes. This will undoubtedly change the shape of the cost surface no matter which algorithm is used. However steps such as the 'bias correction' and isolating new neuron pathways from the original network should reduce change to the cost surface.

When a neuron is removed from a trained network it will result in the cost surface becoming higher and steeper as learned parameters are lost. By using the 'bias correction' it compensates the remaining network for some of the activation lost, by changing bias parameters in the following neuron layer. The aim of this is to reduce how the height change of the cost surface and attempt to keep the neural network near to the local minimum of the gradient. Therefore preserving the functionality of the network. Some functionality will be undoubtedly lost as there are now fewer neurons in the network and although the removed neuron was deemed insignificant it will have still contributed as small amount to the network's performance.

When adding a new neuron pathway two steps are taken to minimise disruption to the cost surface, these are the 'scaffold connections' and 'pathway neurons'. Scaffold connections as previously described prevent the propagation of activation signals from the pathway to neurons of the original network. This isolates the new pathway and also isolates it in terms of the backpropagation dimensions. By creating any new parameters it forms new dimensions for gradient descent, this generates a new shape for the cost surface. However scaffold neurons ensure that the this new shape of the surface preserves the gradient in all of the original dimensions. Hence keeping the network in the local minimum for all the original parameters even though the height and shape of the cost surface has changed. This preserves learning for the original network.

When pathway neurons are initialised they have specific weight parameters based on the inputs. Again the new dimensions for the cost surface are created, the network would usually be put at position 0 for each of these new dimensions. However the specific values for the weight parameters attempt to shift the position of the network in the new dimensions to near a local minimum point, hence attempting to accelerate the learning process for the new neurons.

All of the other functions in LNPM should not influence the cost surface significantly as they do not directly affect the parameter gradients. It is also important to note that LNPM is only designed to work on fully connected network layers as other network layers, such as convolution, have different backpropagation gradients and hence more complex cost surfaces, preventing the algorithms from working. This means that pathways should only be added and neurons removed from fully connected layers. LNPM should still work on networks with mixed structures, however can only operate on the fully connected layers.

In conclusion it is important to minimise the amount the cost surface is changed whilst the network is learning, to preserve network functionality and increase learning rate. The functions in LNPM have been created so that the minimise changes to the parameter dimensions and have extra features such as bias correction which only functions to preserve the cost surface shape. Overall I feel that when designing new neural network features it is important to consider the effects on the cost surface to create a more successful algorithm.

The Lateral Neural Pathway Manipulation Algorithm

This section is intended to outline all the formulas and algorithms behind LNPM and describe how each function operates in the system. I will overview each part of the algorithm individually, but when implementing LNPM all functions must be used simultaneously. It is important to stress that I cannot test the effectiveness of LNPM due to the lack of time and hardware available for this project. As a result I do not know whether each algorithm will truly work as intended, the maths used in formulas is designed to have the desired function but will have only been tested independently from the other functions in LNPM.

Consequently the descriptions of the maths in this section are only theoretical, but even if the maths does not work when tested, I feel in principle LNPM could be a very successful step for neural networks. Allowing a network to adapt it structure similarly to the human brain. This could allow the possibility for much more advanced networks to solve a multitude of new problems.

In this section, algorithms will be displayed similarly to Python and Python's numpy notation, as I feel python is a logical way of communicating these algorithms.

To begin the cost function must be adapted slightly to allow a target vector to be generated for extra output neurons, not featured in the dataset. Below shows the original formula for the mean-squared error cost:

$$C\left(a^{L+1},\ t\right) = \frac{1}{2n}\sum_{i=1}^{n}\left(a_i^{L+1} - t_i\right)^2$$

It is the 't_i' in this formula which LNPM will adapt. Usually 't_i' will be a one-hot vector supplied by the dataset to cover all the output neurons for that dataset. However as LNPM is creating new outputs, t_i must be enlarged to accommodate this. It is important to define some variables at this stage which are used throughout the algorithm:

$$x = Number\ of\ Original\ Output\ Neurons$$

$$y = Number\ of\ New\ LNPM\ Output\ Neurons$$

So if the t_i vector supplied by the dataset is 'x' elements long, then 'y' number of elements must be appended to the end. These extra elements are initiated at the value 0, to preserve the one-hot array. Despite this, if the maximum output activation from the network is on a newly generated output then t_i is initiated differently. The algorithm is shown below:

```
for i in range(n):
    if argmax($a_i^{L+1}$) >= x:
        $t_i$ = numpy.append(0.5*$t_i$, numpy.zeros((1,y)))
        $t_i$[argmax($a_i^{L+1}$)] = 1
    else:
        $t_i$ = numpy.append($t_i$, numpy.zeros((1,y)))
```

This simply appends a zero vector of length y onto the original target vector supplied by the dataset. Although if the maximum output has an index greater than x (it is a newly generated neuron), then the dataset target vector is halved and has a zero vector appended onto it, like before, however a value of one is set at the index of maximum activation.

As a result the network usually relies on the dataset one-hot target vector. However if the most certain classification is from a newly generated output then it will positively train that output and place less emphasis on the dataset classification. As a result it allows the network to train classifications which aren't present in the dataset as long as the network is certain it's the correct classification for the data.

The next function is a non-linear weighted average formula, which I have designed to modify the t_i vector.

$$t_i' = \frac{\left(1-2a_i\right)^2 y a_i + \left(1-2t_i\right)^2 x t_i}{\left(1-2a_i\right)^2 y + \left(1-2t_i\right)^2 x}$$

$$\in\{a\neq 0.5 \cup t\neq 0.5 \cup 0\leq a\leq 1 \cup 0\leq t\leq 1\}$$

This function creates a new target vector which is averaged between the original target vector and the output activations from the network. It predominantly weights the average towards the original target vector when the original neurons outnumber the newly generated neurons. It also weights the average towards the output activations when they have very high positive or negative certainty. This is when the activation is nearing zero (i.e. 100% certain it is false) or one (i.e. 100% certain the classification is true.). This is a little unintuitive but 0.5 represents a 0% certainty that the classification is true or false, and 0.75 is 50% certain the output is true.

As a result, the more certain the network is; the greater the target vector will be weighted in favour of the network's classification as opposed to the dataset's classification. This, in a sense, gives the network the ability to train and learn itself independently from the dataset. it does this increasingly as its classification certainty becomes higher. This allows it to ignore the dataset if it appears, statistically to the network, that some data is misclassified. Combining this new adapted cost with the mean-squared error yields the formula below:

$$C\left(a^{L+1},\ t\right) = \frac{1}{2n}\sum_{i=1}^{n}\left(a_i^{L+1} - \frac{\left(1-2a_i\right)^2 y a_i + \left(1-2t_i\right)^2 x t_i}{\left(1-2a_i\right)^2 y + \left(1-2t_i\right)^2 x}\right)^2$$

$$\in\{a\neq 0.5 \cup t\neq 0.5 \cup 0\leq a\leq 1 \cup 0\leq t\leq 1\}$$

And its corresponding differentiated form required for gradient descent:

$$\frac{\delta C}{\delta a^{L+1}}\left(a^{L+1},\ t\right) = \frac{1}{n}\sum_{i=1}^{n}\left(a_i^{L+1} - \frac{\left(1-2a_i^{L+1}\right)^2 y a_i^{L+1} + \left(1-2t_i\right)^2 x t_i}{\left(1-2a_i^{L+1}\right)^2 y + \left(1-2t_i\right)^2 x}\right) \times \left(1 - \frac{\left(\left(1-2a_i^{L+1}\right)^2 y + \left(1-2t_i\right)^2 x\right)\left(\left(1-2a_i^{L+1}\right)^2 y - 4ya_i^{L+1}\left(1-2a_i^{L+1}\right)\right) + 4\left(\left(1\right.\right.}{\left(\left(1-2a_i^{L+1}\right)^2 y + \left(1-2t_i\right)^2 x\right)^2}\right.$$

$$\epsilon\{a \neq 0.5 \cup t \neq 0.5 \cup 0 \leq a \leq 1 \cup 0 \leq t \leq 1\}$$

This adapted target vector can also be implemented into many other cost functions. Its purpose just to average the target and activation vectors weighted based on the network's classification certainty. As a result the dataset may classify one data example as a handwritten one, however the network is 99% sure the input is in fact not a one but a handwritten eight. In this case the value for one represented in target value will be reduced, and the eighth element's value increased. This will consequently put more emphasis on the network to learn that data example as an eight not a one, as suggested by the dataset. This is useful for misclassification errors.

The next amendment for the neural network is the 'frequency vectors'. Frequency vectors store the sum of the magnitudes of each neurons activation per epoch. Then this is multiplied with that activation, before the activation function, to increase or decrease the resulting activation. In effect the more a neuron fires, the stronger its activations become in the network. This acts like the saying "if you don't use it; you lose it." Where inactive neurons slowly have their function diminished, and other very active neurons reinforce their activation signals. It is important to note that frequency vectors do not need to be implemented on every layer of neurons. I imagine it to work a little like the activation layer, where it is optional to use the frequency vector in addition to the usual feed forward layer. Although a frequency vector must be used on the output layer for LNPM to work effectively.

Below shows the algorithm describing frequency vectors. The algorithm has been broken down into stages which will be explained individually.

First stage is calculating the usual activation found from multiplying the previous layer's activation and adding the bias.

Stage 1:

$$z^L = \left(a^{L-1} \times w^L\right) + b^L$$

$$f^{L'} = 2\left|z^L - 0.5\right| + \alpha f^L$$

The previous equations show how the frequency vector '$f^{L'}$' is updated. It has the activation 'z^L' added to 'αf^L'. Alpha in this equation is the constant decay factor and is used to reduce the previous value for the frequency vector by a certain proportion each iteration. This reduced frequency vector and the magnitude of the current activation 'z' are then summed to find the new frequency vector. Before 'z^L' is added to the frequency vector 0.5 is subtracted from it, this is because 0.5 represents 0% certainty. Frequency vectors sums the 'activeness' or certainty of the neuron and nearing 0 or 1 is when the neuron is considered most certain therefore most active .In this equation the frequency vector has equal length to the activation vector, yet alpha is a constant scalar chosen when creating the network. The frequency vector is also initiated as a vector of ones.

Stage 2:

$$N^{L} = \frac{f^{L}}{\Sigma f^{L}} \times \dim dim\left(f^{L}\right) \times z^{L}$$

$$a^{L} = \frac{1}{1+e^{-N^{L}}}$$

In this step the frequency vector is effectively multiplied by the original activation to create a temporary vector 'N' which is then passed through an activation function, in this case sigmoid, to generate the output 'a' for the layer. The equation below normalises the frequency vector to have a mean of one across all its elements. As a result the frequency layer does not increase or decrease the activation overall, just increase/decreases individual values for activation. As a result the neuron layer does not become too active or inactive. In the below equation '$dim(f^{L})$' finds the length of the vector as an integer, and $\frac{f^{L}}{\Sigma f^{L}}$ finds the frequency vector divided by the sum of all elements in that vector.

$$\frac{f^{L}}{\Sigma f^{L}} \times dim(f^{L})$$

The resulting differentiation of the frequency layer produces the below equations:

$$\frac{\delta a^{L}}{\delta a^{L-1}} = \left(\frac{1}{1+e^{-N^{L}}}\right) \times \left(1 - \frac{1}{1+e^{-N^{L}}}\right) \times \frac{f^{L}}{\Sigma f^{L}} \times dim(f^{L}) \times w^{L}$$

$$\frac{\delta a^{L}}{\delta w^{L}} = \left(\frac{1}{1+e^{-N^{L}}}\right) \times \left(1 - \frac{1}{1+e^{-N^{L}}}\right) \times \frac{f^{L}}{\Sigma f^{L}} \times dim(f^{L}) \times a^{L-1}$$

$$\frac{\delta a^{L}}{\delta b^{L}} = \left(\frac{1}{1+e^{-N^{L}}}\right) \times \left(1 - \frac{1}{1+e^{-N^{L}}}\right) \times \frac{f^{L}}{\Sigma f^{L}} \times dim(f^{L})$$

Next in the LNPM algorithm neurons must be added in 'pathway addition'. To begin a condition must be met by the network to result in pathway addition being activated. This condition is abnormally high cost. This is using the aforementioned modified cost from earlier in this chapter.

Anomalously high cost describes when the network is either very unsure of the correct classification for the data compared to its usual classification accuracy, or it strongly disagrees with the dataset's provided classification. The latter example describes when there may be a misclassification error in the dataset, and the initial example might describe when there is no correct category available for the data. Both these conditions result in high cost, but potentially not abnormally high cost. I define abnormally high cost, in this context, when the cost value for a specific data example is significantly higher than the cost for all the other data examples in the stochastic training batch. This can be found statistically when the cost is above the mean plus five standard deviations for the rest of the training batch. Below shows the conditions to activate pathway addition.

```
length = cost.shape
#length is the number of training batches in the cost vector.
for i in range(length):
        if cost[i] > np.mean(cost) + 5*np.std(cost):
                #Activate Lateral Pathway Addition
```

This is assuming the is in the form of a vector before it is summed into a single value. Hence each element represents the cost for a single data example in the stochastic batch. If Lateral Pathway is then triggered, new neurons must be added to each fully connected layer, as shown in figure 18. In this section I will not describe how each type of neuron, 'pathway' and 'scaffold' neuron, as this is previously explained. However the algorithms for initiating their parameters are shown below:

Scaffold Connection:

$$w_{ij}^{L} = 0$$

Scaffold connections are initialised to prevent forward propagation of activation signals from the pathway to the original network, therefore isolating the pathway. This can be achieved by setting the weight on the connection to zero. This works as $a_j^{L+1} = w_{ij}^{L} a_i^{L} + b_j^{L}$,so if $w_{ij}^{L} = 0$ then $a_j^{L+1} = 0 \times a_i^{L} + b_j^{L} = b_j^{L}$ therefore preventing propagation of the activation signal.

As a result it also has a gradient of zero in respect to the activation, but a non zero gradient in respect for the weight allowing learning to occur and slowly allow propagation of signals to occur.

Pathway Neuron:

$$b_j^{L} =- \frac{1}{2n} \sum_{i=1}^{n} \left(a_i^{L} \frac{1-e^{-5a_i^{L}}}{1+e^{-5a_i^{L}}} \right)$$

$$w_{ij}^{L} = \frac{1-e^{-5a_i^{L}}}{1+e^{-5a_i^{L}}}$$

Pathway neurons are initialised so that they produce a strong activation when similar activations to a_i^{L} are presented to the network again, and a low activation otherwise. The variable a_i^{L} is the activation from the previous layer, corresponding to the weight, from the batch which triggered pathway addition in the network. The bias is calculated using the averages of the function: $a_i^{L} \frac{1-e^{-5a_i^{L}}}{1+e^{-5a_i^{L}}}$ from all the activations in the previous layer. This initialisation of the pathway neuron parameters should result in a high activation when similar values to a_i^{L} are presented to the neuron again. Pathway neurons connect from any of the newly generated neuron to all neurons in the previous layer.

Finally some network variables must be updated such as x, y and the length of the frequency vector:

```
$x' = x$
$y = y+1$
$f_j^L = \frac{1}{n}\sum_{i=0}^{n} f_i^L$
```

To conclude the LNPM algorithms neuron pruning must be implemented. Neural pruning occurs when a neuron generated by LNPM is particularly inactive. This means it's activation is consistently near 0.5, i.e. it is 0% certain that a value is true for false. Neural pruning is triggered when it is determined that a neuron is has a anomalously low certainty. This can be found using the frequency vector if a particular element, corresponding to a certain neuron, is significantly, statistically, lower than the other frequency values for that layer. This would indicate the neuron is not contributing much to the network so can be removed. Also neuron pruning should only occur in neurons added to the network by pathway addition. Below shows an algorithm describing this criteria:

```
#Only test this criteria for fully connected layers with a frequency vector implemented, as only these layers can be pruned.
for i in range(y):
    if f[x+i] < np.mean(f) - 5*np.std(f):
        x' = x
        y = y-1
        #Activate neuron pruning
```

If the criteria for the neuron pruning is fulfilled then the neuron is removed and adjustments made to surrounding connections. The aforementioned algorithm must be then restarted from i=0 to test for any other underperforming neurons which can be removed in the same epoch.

The neuron is removed by deleting all forward and back propagating connections from the inactive neuron as shown in figure 19. However first bias correction must be made to all the neurons in the following layer. This involves updating the biases corresponding to each neuron in the following layer, by adding an estimate of the average activation provided to the following neurons from the deleted neuron. This is described by the below formula:

$$b_j^{L+1'} = b_j^{L+1} + 0.5w_{ij}^L$$

Where w_{ij}^L is the weight value going from the deleted neuron to the neuron in the next layer with bias b_j^{L+1}. This weight value is multiplied by 0.5 as an estimate of the average activation for the deleted neuron, as the activation ranges from zero to one, so 0.5 is a midway point. It is an estimate of the average activation of the neuron as the neuron is very unlikely to have a true average activation of 0.5.

This concludes all the algorithms needed to make Lateral Neural Pathway Manipulation functional. It is important to note that the algorithms for lateral pathway addition and neural pruning should occur at the end of the epoch after the modified cost has been calculated, and backpropagation and gradient descent have occurred.

The algorithm is designed so that any input which produces an anomalous output, has a new neuron pathway generated for the input. This neuron pathway is isolated from the main network and is initiated so that it is likely to fire if a similar input is presented again. This pathway also has an output neuron so that a new classification can be expressed. Over time this isolated pathway will merge with the original network, via gradient descent and backpropagation, so that it is no longer isolated and can communicate with the original network. If any of these newly generated neurons, including the output classification, are deemed statistically useless; this redundant neuron is then removed from the network, and surrounding neurons compensated for the loss in activation.

Theoretically this pathway addition and neural pruning should somewhat compete over time and reach an equilibrium where the network does not grow unnecessarily resulting in longer learning times. Instead a stable network structure arises with all the necessary classifications to correctly categorise the dataset's data. This flexible network architecture may allow a network to learn new problems, from extra datasets it wasn't initially designed to learn from.

It is important to note that LNPM is designed to run when the network is in its training phase. Also many applications for neural networks may struggle with the variable number of output neurons/classifications, therefore software may need to be changed to accommodate LNPM changing the number of outputs.

Conclusion

Networks In Society

I hope that from this project it is clear how much potential neural networks have to benefit society. Currently neural networks are moderately simple imitations of our own brains, yet have already yielded such amazing advancements in many disciplines from medicine to self driving cars. The field of radiology has had rapid innovation during recent years in using convolution neural networks to identify malignant cells in numerous forms of medical imaging. In one report a neural network has managed to reach a 97.5% accuracy when identifying breast cancers from biopsies, and this technology is already more than six years old (Wenger, 2013)!

This medicinal application of neural networks has undoubtedly led to early recognition of disease allowing faster treatment and potential to extend lifespan. Neural networks have now got their own field of medicine named bioinformatics, where they are used to scan through genomes to identify how genes are expressed. This has led to greater understanding of the human genome allowing steps to prevent genetic diseases.

Besides these medicinal uses, neural networks have played a key part in developing other new technologies. Self driving cars rely on convolution networks to identify obstacles and road markings, as well as this large companies, such as Amazon, use neural network technologies to predict customer spending patterns. Currently neural network use is widespread throughout industry and its applications will only continue to advance bringing many benefits to the world.

Problems With Current Networks And Upcoming Solutions

One of the main benefits in using neural networks is that they can understand patterns involving hundreds of variables easily, whereas due to the limits of human perception we can only plot graphs using at most three dimensions, allowing two independent variables and a dependant output to visualise patterns. However the more complex the problem; the more time it takes to successfully train a neural network to solve it.

Despite the fact that networks learn considerably faster than humans, they can still be considered as a rather slow method to solving certain problems. Some advancements such as backpropagation, momentum, improvements in processor speeds, and GPU integration have unquestionably reduced this time significantly, there needs to be further significant learning improvements made to make them viable for prevalent use, and to develop larger more complex networks. Some promise comes in the form of Neuromorphic chips being developed currently. These chips are specifically designed to run deep neural networks by utilising hardware which closely mimics the brains processing system. These chips hold great potential to further reduce network learning times allowing more complex neural networks to become practical.

Another prominent problem with neural networks is when the produce an incorrect output. This could potentially have disastrous effects, especially in the medical field, if the produce a false-positive or false-negative result. As a consequence they will need to go through extensive medical trials before being properly implemented into modern health care. Several methods of improving performance has been mentioned in this project such as dropout and momentum. In addition it appears one of the best solutions to improve network performance is to train them on larger datasets. There is a direct correlation between the accuracy of a neural network and the size of its training dataset. As a result neural networks have been one of the large driving forces behind 'big data'.

Big data is the idea of collecting and storing an abundance of data from the physical world. This trend began many years ago, and now this large mass of data can finally be utilised to train ever more sophisticated neural networks. This will be one of the best ways of combating neural networks poor performance on certain tasks, however it does also increase computation time to train these networks. Despite this networks will always have some inaccuracy, especially if a dataset is not accurate in itself.

Finally I believe one of the biggest problems with current neural networks is that they are currently designed to solve specific task, they are not yet general purpose networks. When networks are created, they have a specified constant number of neurons available to them throughout training which limits them to only solving the desired task. Certain algorithms such as NEAT help find this optimum amount of neurons, however once the network has begun learning the structure is fixed, and if changed large learning loss will occur. This leads onto the aim of this project: LNPM. In this conclusion I will evaluate whether LNPM is a successful method to allowing a network to vary its structure without losing learning progress. I will also evaluate whether the concept of varying a network structure may be useful in allowing a neural network to specialise in more than one task, edging the technology closer to general purpose networks.

Lateral Neural Pathway Manipulation Networks Evaluation And Development

Are the algorithms described in this extended project effective in changing a networks structure to learn new skills? The answer is: I'm not sure due to lack of testing. Despite this, I believe the idea of allowing a network to change its structure could be hugely beneficial to creating larger more complex neural networks with the ability to solve an array of new problems. I also hope that allowing a network to learn multiple specialities may be an effective way of bringing specific purpose networks closer to more human like general purpose networks.

To begin, I will overview development and benefits LNPM. If LNPM works as intended it allows a neural network to change the number and organisation of neurons in its structure to create new available space for a network to encode learning. If these newly created neurons aren't used by the network, they are carefully removed to optimise the learning rate of the network.

When developing the idea for adding neurons to an existing network I did some preliminary investigations which including outlining difficulties when creating neurons. It became apparent that the disparity between the original trained network and freshly created untrained neurons would be an issue. As the new neurons would chaotically fire interrupting the precision of the previous network. Due to this I formed the idea of using two different neuron to neuron connections. These developed into the so called 'pathway neuron' and 'scaffold connection'. The scaffold connection was the first development in the LNPM theory.

The idea of scaffold neurons was to create an inactive neuron barrier which prevented the chaotic signals from the new neurons from transmitting to the original network. However the original network still needed a connection to these neurons for them to eventually merge into one unified network. I feel the concept of these scaffold neurons is a successful part of LNPM, theoretically its parameters are set so that they prevent transmission of signals but can be eventually reactivated by gradient descent to allow useful signals to propagate down them. This allows for the successful merger of new neurons to the original network, allowing the network to grow its capacity for learning.

Another method to combat this disparity between the original network and new neurons was the 'pathway neuron' idea. The idea is to set the parameters of the new neurons so that they are predisposed to fire when a similar input is presented again. My intention was by setting the parameters the pathway is more likely to propagate signals which closely match the data which triggered pathway addition. Usually this would take time for a network to learn, but by creating the neurons in this way it somewhat accelerates the learning process for the pathway and determines how it should learn. This allows the pathway to catch up quicker to the original networks functionality, reducing the time needed to train it and allowing faster merger of the pathway and initial network.

After creating these new pathways of neurons, a new problem arose. This is how to train the newly created neurons so they can learn in the network. For the original neurons it was easy, just used the classifications provided by the dataset to teach the network, however for the extra neurons there was no data provided from the given dataset on how to teach them. Therefore I modified the cost function by changing how the target vector taught the network. The modified target vector is designed to find a target vector which is capable of teaching all the output neurons. This was done by merging the activations from the network and the dataset classifications so that the target vector reflected the classification proposed by the network if the network was very certain it was correct. By doing this it allowed the network to compare its classification 'opinion' with the dataset and learn from both available training methods.

Teaching the network in this way may mean it struggles to learn further once it has become too confident on its own classifications. As a result this customized cost function may need additional improvement for it to work effectively within the LNPM algorithm. However, again, testing is the only way to determine this.

Quickly it became apparent through a few simple tests that the criteria for generating a new lateral neuron pathway is very sensitive in many datasets, despite using five sigma deviation from the mean. As a result potentially many redundant neurons would be generated in the network. The more neurons a network has; the more capability it has to learn; but the longer it will take to learn. This balance needed to be managed by LNPM as many new neurons would be constantly generated. As a result 'neuron pruning' is needed, which is somewhat comparable to synaptic pruning in the brain.

Neuron pruning is designed to reduce the amount of connection, therefore teachable parameters, in the network by removing these useless neurons. I determined that a useless neuron would be any neuron which has a very low magnitude of activation below five-sigma standard deviation from the mean magnitude of activation of other neurons in the same layer. This activation data would be stored using the frequency vectors, which summed the magnitude of activations over every epoch. The values in this frequency vector would become very large after thousands of epochs, therefore a decay factor was introduced to reduce the values at a constant rate per epoch.

The next challenge arose when determining how to remove a neuron and its connections. Unlike biological neurons, artificial neurons will have a mean baseline activation they provide to the surrounding neurons, this is like their average activation. By simply removing this baseline the other neurons it is connected to will have their input activations reduced. Neural networks performance are very sensitive to small changes in activations, therefore removing this baseline activation would damage the networks performance. Therefore I decided to modify the neurons, in the following layer, bias values to account for this loss in activation. Theoretically this should keep the network's learned behaviour preserved as the increased bias matches this baseline activation loss; resulting in minimal loss in activation for the network.

I later decided that it may be beneficial to multiply the new frequency vector by a neuron layer's activation. Therefore neurons which fire more would produce a stronger than those which fire infrequently. In addition to this, neurons which have a high magnitude of activation also learn faster. As a result this creates a difference in learning rates for neurons in a single layer, allowing more important neurons to learn faster and reinforce whereas redundant neurons learn even slower so can be more easily identified and removed. This function may not necessarily be needed though, as a neural network tends to do this on its own accord, where certain neurons will learn faster than others so become more important in the network. Consequently it may be found in testing that the frequency vector should not be multiplied with the activation, as this could disrupt the networks normal learning process. Potentially this could be a problem with LNPM, which only testing will identify. However it is not essential to LNPM so this function can be removed from the LNPM algorithm.

Problems For The LNPM Algorithm

There are potentially many problems with LNPM which will be uncovered through testing, which could result in LNPM being defunct for its intended purpose. However there are some problems which can potentially be predicted before the algorithm is tested, in this section I will describe some of these outstanding issues with LNPM and the likelihood of each one occurring.

As previously mentioned, multiplying the activations from a neuron layer by the frequency vector may result in the network not learning as intended. The objective of doing this is to try to accelerate learning down important neuron pathways, however neural networks tend to do this by themselves, unless using dropout. Therefore by incorporating frequency vectors too, it may result in neurons becoming too overactive, producing incorrect results for the network. Consequently this function in LNPM may have to be removed if it proves, in testing, to negatively affect the network's performance and learning.

Another prospective problem with LNPM is the modified cost function. In this the target vector is changed so it represents an weighted average between the network's and dataset's classifications. However this function has not been tested with a neural network so may potentially weight the average too strongly in favour of the network. If this overrepresentation of the network's classifications occurs, it could slow down or even reverse learning progress for the network. This develop because the target values for the network become too closely aligned with the network, therefore the network always believes it is correct, as it closely resembles the target values, so will not learn from its errors. Learning may even reverse if a network becomes over confident and effectively ignores the dataset classifications, this could result in it following an incorrect learning path. This problem is moderately likely to occur but could potentially be reduced by changing the priority of the weighting system for the target vector to weight more in the dataset's favour.

This modified cost has too further possible problems. The first is that it may not work with other cost functions except mean-squared error (MSE). There are many cost functions, which sometimes outperform mean-squared error, and there is no evidence that the modified target vector is compatible with these other cost functions. Therefore, if is not compatible, LNPM would be limited to only working with networks utilising MSE cost, which would greatly reduce LNPM's applications. However, I do not feel this is a likely problem to occur as the modified cost is simply changing the targets to be reflective of the network's activations too. This should not interfere too much with how other cost functions work, as they should still produce high cost when the network does not match the target vector, and low cost otherwise. Hence the network should still learn effectively.

The final problem is with the modified target vector, is that the function is undefined when both the activation and original target vectors are 0.5. This results in a divide-by-zero error, as a result the function does not produce a valid target vector, nor gradient at this point. As a result the neural network code may crash, and cease learning. Despite this, this error is extremely unlikely to occur, as the likelihood of both the target vector and activation function being exactly 0.5 is negligible. This is becomes apparent as even if the activation and target vector differed from 0.5 by a one trillionth the function would be defined so would still work. Therefore this function is usable as likelihood of error is infinitesimally small.

Theoretically another issue with the LNPM code is if pathway addition criteria is too sensitive whereas neuron pruning not sensitive enough. As a result the rate of neurons added by pathway addition would greatly exceed the rate at which neurons are removed and the network would grow exponentially. If the network grew to large, there would be an overabundance of parameters, meaning the gradient of cost would reduce and learning slow down hugely. This is a fairly likely problem, but can easily be solved by changing the criteria for pathway addition and neuron pruning. By changing the multiples of deviation from the mean for both functions, can result increased neuron pruning and decreased pathway addition. In this project I used five standard deviations from the mean to determine the criteria for both functions, however this value is somewhat arbitrary and can easily be changed throughout testing of LNPM.

Bias correction is intended to replace the baseline activation lost when a neuron is removed from the network. However the corrected bias represents only an estimate of what this average baseline may be. Consequently the network may still lose some baseline activation, resulting in the network's learning progress being compromised. This is potentially a problem for LNPM, despite this it is still an improvement on simply removing the neuron without any correction. Throughout testing it may be found that bias correction needs further revision to bring it closer to compensate the true baseline activation lost, as a result the algorithms behind LNPM may need to be tweaked during the testing phase.

One very unique and large consequence of LNPM is that the number of output neurons in the network can vary. I am not aware of this occurring before in common neural network algorithms and could represent an issue when developing software to utilise LNPM. The number output neurons are allowed to change in LNPM so a neural network can express more options for the provided data than represented in a given dataset, however almost all applications of neural networks rely on a constant number of neurons. Particularly in fields like robotics where each output may be assigned to a certain motor, therefore increasing outputs would be futile as there are insufficient physical motions to represent the network's new outputs. The only way to solve this issue is to specifically design software to be able to utilise this changing output topology. The output neurons are allowed to alter so a network can learn new skills from datasets it wasn't initially designed to learn from. These new outputs allow a network to express its newly acquired knowledge on these other datasets. I feel that making software to accept these adaptive outputs could bring about new applications for neural networks, allowing a neural network to slowly become more general purpose and less specialised on a few minimal datasets.

However, currently the criteria for LNPM to create new pathways is more sensitive to misclassification errors in a single dataset, rather than allowing it to learn from new unforeseen datasets entirely. This is certainly an issue with LNPM, as it is likely to be able to effectively distinguish a letter as a unique character when learning from a handwritten number dataset and develop a new output to convey this. But it will not be as successful when given an entirely new dataset to learn from. This will definitely require much more development on the criteria for pathway addition to occur and I feel the best way to do this is only through extensive testing. However LNPM in its current form, should be effective when creating neural networks used to identify visual data.

A clear example of this would be if a neural network is designed to receive data from a real world camera like in driverless cars. This neural network would be trained to recognise road markings, trees, pedestrians etc. LNPM could be used if a new real world object appeared on this camera which the neural network did not recognise such as a dog. Using LNPM the neural network should then be able to distinguish it as a distinct new object, create a new output neuron to represent it, and consequently the software utilising the neural network could take the further required action.

Finally I cannot foresee any problem with implementing LNPM in networks utilising other algorithms such as dropout or momentum. Momentum might need slight revision to accommodate the changing amount of neurons in the network and be able to apply the momentum to these new neurons. However this should not be a large issue, and should simply be solved by modifying the size of the momentum arrays when a new neuron is generated, or removing a specific element in the array when a neuron is removed.

Despite all these potential problems with LNPM, I feel the most likely and disastrous problems have been averted by the LNPM algorithms. Furthermore any remaining or unforeseen problems can hopefully be resolved through testing and amending the functions. Therefore I feel the problems outlined should not drawback from the benefits of lateral neural pathway manipulation networks, and any outstanding problems may possibly be fixable.

Is Lateral Neural Pathway Manipulation An Improvement For The Technology?

I hope it is clear that throughout development of LNPM's algorithms, the benefits and pitfalls of each function have been carefully considered. If a particular function could cause a detrimental decrease in the network's performance then steps have been taken to minimise this, such as the bias correction. However it us undoubtable that further problems with LNPM may arise through testing, requiring further modification to the program.

Despite this I feel the ability for a neural network to change its neuron structure, whilst learning, but without compromising its performance will be hugely advantageous when designing new neural networks. After all it has clearly been beneficial enough for natural selection to develop similar systems in biology. Even if the algorithms behind LNPM are ineffective, I feel further development of algorithms with the common aim of altering network's neuron architecture should be taken as it will allow many new applications of neural networks to arise. Moving this technology a little more towards the desired general intelligence which would benefit the world greatly.

In my opinion connecting many different specific purpose artificial neural networks together into one more generalised network will open the door to many new improved applications of neural networks, and the algorithms in LNPM might be a good stepping stone to develop a tool to achieve this in the future.

Final Notes:

All the mathematics detailed in this final chapter of my EPQ: "Lateral Neural Pathway Manipulation" is entirely new, and I have created the mathematics for the purpose of this EPQ. They have not originated from any other outside sources, and the maths I have developed myself. Each function has individually been tested. However the algorithm as a collective, as aforementioned, has not been tested. This is due to the time constraints for project, I have already spent a significant amount of time on writing neural network codes from scratch, and all my code would need to be entirely repurposed to include LNPM, therefore is out of scope for this project.

Bibliography

3Blue1Brown. (2017, October 5). *But what *is* a Neural Network? | Deep learning, chapter 1.* Retrieved November 24, 2018, from YouTube: https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

America, S. (2017, May). *Why Is Synaptic Pruning Important for the Developing Brain?* Retrieved March 13, 2019, from Scientific America: https://www.scientificamerican.com/article/why-is-synaptic-pruning-important-for-the-developing-brain/

Hassabis, D. (2017, October 18). *DeepMind.* Retrieved November 24, 2018, from AlphaGo-Zero Learning from scratch.: https://deepmind.com/blog/alphago-zero-learning-scratch/

Hochreiter, S., & Schmidhuber, J. (1997, November 15). *Long Short-Term Memory.* Retrieved November 13, 2018, from ACM DL: https://dl.acm.org/citation.cfm?id=1246450

K. A. (2010, February 11). *Anatomy of a neuron.* Retrieved September 05, 2018, from YouTube: https://www.youtube.com/watch?v=ob5U8zPbAX4

Kaku, D. M. (2014, February 25). *Michio Kaku explores human brain.* Retrieved September 2018, 2018, from wnyc: https://www.wnyc.org/story/michio-kaku-explores-human-brain/

LeCun, Y. (1998, November). *Convolutional Neural Networks.* Retrieved 11 13, 2018, from Yann LeCun: http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf

LeCun, Y., Cortes, C., & Burges, C. J. (2013). *MNIST.* Retrieved November 26, 2018, from Yann LeCun: http://yann.lecun.com/exdb/mnist/

Maaten, L. v. (2008, November 9). *t-Stochastic Neighbour Embedding.* Retrieved November 11, 2018, from Laurens van der Maaten: https://lvdmaaten.github.io/tsne/

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986, October 09). *Learning Representations by Backpropagation Errors.* Retrieved March 03, 2019, from Nature: https://www.nature.com/articles/323533a0

Salzer, J. (2016). *Myelination.* Retrieved March 03, 2019, from ScienceDirect: https://www.sciencedirect.com/science/article/pii/S0960982216308685

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., et al. (2017). *Nature.* Retrieved November 24, 2018, from Mastering the game of Go without : https://www.nature.com/articles/nature24270.epdf?author_access_token=VJXbVjaSHxFoctQQ4p2k4tRgN0jAjWel9jnR3ZoTv0PVW4gB86EEpGqTRDtpIz-2rmo8-KG06gqVobU5NSCFeHILHcVFUeMsbvwS-lxjqQGg98faovwjxeTUgZAUMnRQ

Soso, M. (2016, June 10). *What are the parts of the neuron and their function?* Retrieved September 05, 2018, from Quora: https://www.quora.com/What-are-the-parts-of-the-neuron-and-their-function

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (n.d.). *Dropout: A Simple Way to Prevent Neural Networks from Overfitting.* Retrieved March 03, 2019, from JMLR: http://jmlr.org/papers/v15/srivastava14a.html

Stanely, K. O., & Miikkulainen, R. (2002). *Evolving Neural Networks through Augmented Topologies.* Retrieved November 13, 2018, from Evolving Neural Networks through Augmented Topologies: http://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=2ahUKEwiw2e-untLeAhVrLcAKHU29DZwQFjAAegQIBxAC&url=http%3A%2F%2Fnn.cs.utexas.edu%2Fdownloads%2Fpapers%2Fstanley.ec02.pdf&usg=AOvVaw0LTSukexmVEUSoZCpjZ9MW

Unknown. (2016, November 06). *The Human Brain: Statistics & Research News.* Retrieved September 04, 2018, from Disabled World: https://www.disabled-world.com/health/neurology/brain/

Wenger, B. (2013, May 23). *How to make a neural network in your bedroom .* Retrieved April 4, 2019, from YouTube: https://youtu.be/n-YbJi4EPxc

Wikipedia. (n.d.). *Gauss-Newton Algorithm.* Retrieved 01 2019, 13, from Wikipedia: https://en.wikipedia.org/wiki/Gauss%E2%80%93Newton_algorithm

Wikipedia. (n.d.). *Occam's razor.* Retrieved January 19, 2019, from Wikipedia: https://en.wikipedia.org/wiki/Occam%27s_razor

Wikipedia. (n.d.). *One-Hot.* Retrieved March 16, 2019, from Wikipedia: https://en.wikipedia.org/wiki/One-hot