

An Investigation Into Artificial Neural Networks

Max Hicks

April 27, 2020



Department of Physics, Astronomy and Mathematics
University of Hertfordshire
Semester B 2020

Contents

1	Introduction	2
2	Artificial neural networks: A brief history	3
3	Mathematical theory	4
3.1	Properties of activation functions	4
3.1.1	Monotonicity	5
3.1.2	Order of continuity	5
3.2	Activation functions and their purposes	6
3.2.1	Sigmoid logistic function	6
3.2.2	Hyperbolic tangent	7
3.2.3	ReLU	9
3.2.4	Softmax	11
3.3	Loss functions	11
3.3.1	MAE	11
3.3.2	MSE	12
3.3.3	Cross Entropy Loss	13
3.3.4	Hinge Loss	14
3.4	Single layer perceptron	17
3.4.1	Backpropagation and feed-forward data flow	17
3.4.2	The XOR gate problem	20
3.5	Multi layer perceptron	22
3.5.1	Learning rate optimisers	24
3.5.2	Initialisation	27
3.5.3	Architectural composition	27
3.6	Convolutional Neural Networks	28
4	Implementation	33
4.1	SLP in Numpy	33
4.2	MLP in TensorFlow/Keras	39
4.3	CNN in TF/Keras	44
5	Conclusion	53
5.1	Conclusion of Results: SLP & MLP	53
5.2	Conclusion of Results: CNN	53
6	Lay Summary	54

Acknowledgements

I would like to thank all of the people that provided me help and support for my project. Dr Oleg Blyuss, Dr Vidas Regelskis and Dr Charles Strickland-Constable for taking the time to provide constructive and insightful feedback that has greatly benefitted my project and my understanding of the topic at hand.

To my family and friends who have supported me through out university. Your support encouraged me to keep working hard on this project and to see how far I have come from only just a few year ago could not have been done if it were not for all of you.

Thank you all.

1 Introduction

Neural networks are a part of everyday life now, and in this investigation we will try and establish what those networks are and what purpose they serve in our everyday lives as it stands to date.

But what is a neural network really? Well in this investigation our neural networks will be used in a way which is not too dissimilar to how our brains perceive information to help us classify things we see in the real world. For example, when you see the number 2 in person how do you know it is in fact the number 2?

The answer lies in the human brain's capacity to learn and retain information, so in the case of the number 2, you may have seen the number 2 to know the general shape of it. The curve at the top going clockwise, then the straight diagonal bit that transitions into a flat line toward the bottom.

Our brains are very adaptable you see, so we can write the number 2 slightly differently and we are still able to tell that it is still the same number. We can do this as we see the rough pattern that signals to our brain that what you are seeing, what you are perceiving is in fact the number 2.

This can also be done in a very similar way with a neural network, aptly named a *neural* network as it is modelled after the synapses and neurons inside our brains. The goal of a neural network within our investigation is to be able to classify certain types of information, be that images or points on a graph. A neural network can be trained like our brains, through processes that will be discussed further in this investigation. To be able to mimic our brain's ability to distinguish between different images and classify them as we would do.

But neural networks aren't just limited to just classifying images, they can aid us in our everyday lives by recognising patterns, just as any human can. For example if you tell me what genre of music you like I could perhaps recommend a few artists. A neural network could recommend you far more artists by just knowing your listening habits with regard to your favourite music. This can be seen on YouTube for example, where machine learning is used to recommend different songs to you based on what you usually listen to with far more depth than anyone could ever suggest.

Neural networks can aid our lives greatly by recognising patterns in our behaviour in order to help us discover things we may be interested in. It can learn things about us that we bury in our subconscious and then present us with these interesting things we may want to try. Neural networks can greatly improve our lives, and hopefully in this investigation you will see a glimpse of how this can be achieved.

2 Artificial neural networks: A brief history

The history of neural networks finds its origins in the late 1940's with Donald Olding Hebb, frequently cited as the father of neural network with his book, published in 1949 titled: "The Organization of Behavior: A Neuropsychological Theory". This book led to what is called "Hebbian learning" which serves as the basis from which most neural networks have been built upon.

Hebb's theory was eventually put to the test and in 1954 Farley and Clark simulated a Hebbian network and shortly thereafter was complimented and developed upon by Rosenblatt, who developed the perceptron in 1958 [15]. The perceptron was the initial breakthrough in pattern recognition, it was to demonstrate that clusters of data could be separated by a decision boundary.

The explosion in research wasn't to last for two important reasons, The first was that computers at the time were simply not powerful enough to process large amounts of data, so in accordance with Moore's law we would have to wait a little while for the processing power of computers to catch up with the huge amounts of data we would want to throw at them. The second reason however was much more important: The single layer perceptron was incapable of processing the XOR gate, or any non linear dataset for that matter as Minsky and Papert demonstrated in 1969.

However in 1975 there was a renewed interest in machine learning research as Paul John Werbos came up with a backpropagation algorithm to help train feed forward multi layered neural networks; subsequently, this solved the XOR gate problem raised in 1969.

This backpropagation from the output layer of a feed forward network essentially laid the ground work for neural networks for years to come, in the field of classification for example, multi layered perceptron (MLP) networks were popular up until the mid 2000's and early 2010's until real development of convolutional neural networks (CNN's) became more mainstream and has numerous advantages over an MLP network mainly that for large resolution images and also temporal data such as; motion pictures, sound signature identification and sentence/word identification, an MLP drops off in performance due to the enormous scale of the data and the sheer size of the network and does not pose the CNN's ability to reduce the amount of calculation that it has to perform by introducing convolutional layers that reduce the amount of data flowing through the network.

3 Mathematical theory

At the heart of a simple artificial neural network (ANN), specifically a feed forward ANN which is what this investigation is mainly focused on, however we will touch a little on other architectures of ANN's to show the continued development of ANN's and to show how simple the mathematics is for simple SLP systems to a deep CNN network with tens or even well over a thousand layers [8].

First we must briefly mention the different types of learning that can be undertaken by an ANN, these fall into two main categories. Supervised learning. Where in which the data being inputted to the network has some label associated with it e.g a dataset consisting of cats and dogs will have the labels "cat" and "dog" associated with them respectively, this can be encoded as a number in this case 0 or 1 for "cat" and "dog" respectively.

This type of learning is useful for training a network to recognise and categorise things in the same way as we humans do, this can only be achieved through supervised learning as the label of the data serves as a reference to whether or not the network has achieved the desired result. For example if the network is shown a picture of a dog and categorises it as a cat we can tell the network it has achieved the incorrect answer and as a result we can adjust the network so that in the future it can achieve the desired result. Therefore the label essentially serves a purpose to compute the error term of the network.

the next type of learning is unsupervised learning, where the data has no label associated with it. This type of learning is used to abstract patterns in data that perhaps we may not be able to see, so for instance we could take some dataset and parse it into our neural network, and have it try and detect outliers or anomalies within our dataset. A sub-set of this type of learning is in fact called anomaly detection where a neural network is trained to detect irregular behaviour. An example of this is detecting bank fraud and or card theft of someones credit or debit card.

In this section we will talk about all of theory that needs to be know for an image classifier network. This will comprise of activation functions and the properties they exhibit. Loss functions, when and where to use certain types of loss functions is very important given that we are mainly focusing on classification and not regression. We will also talk about the three types of networks that we will later implement those being the single layer perceptron (SLP), the multi layer perceptron (MLP) and finally a convolutional neural network (CNN).

3.1 Properties of activation functions

Before we analyse what a given activation functions pros and cons are, we should talk about what are nice properties in an activation function that we would want to look for.

3.1.1 Monotonicity

A monotonic function is defined as "A function between ordered sets that preserves or reverses the given order." [18] A monotonic function does not necessarily have to be used in a neural network [4], however it is desirable as computationally a monotonic function is typically much cheaper to compute, also if the function is not monotonic, an increase in the weight on the neuron may cause the output to have less influence which can lead to either chaotic or divergent behaviour.

usually for activation function we want the function to be monotonically increasing functions, a simple example is the "identity" function otherwise known as the linear activation function $y = x$ where as x increases y also increases.

3.1.2 Order of continuity

The order of continuity, usually noted by $C^n, (\forall n \in \mathbb{Z} - [-\infty, -2])$ denotes what order derivative can be taken and have the function still be continuous, for example C^{-1} describes a function that is never continuous, these functions are discontinuous functions, such as the binary step function as seen here:

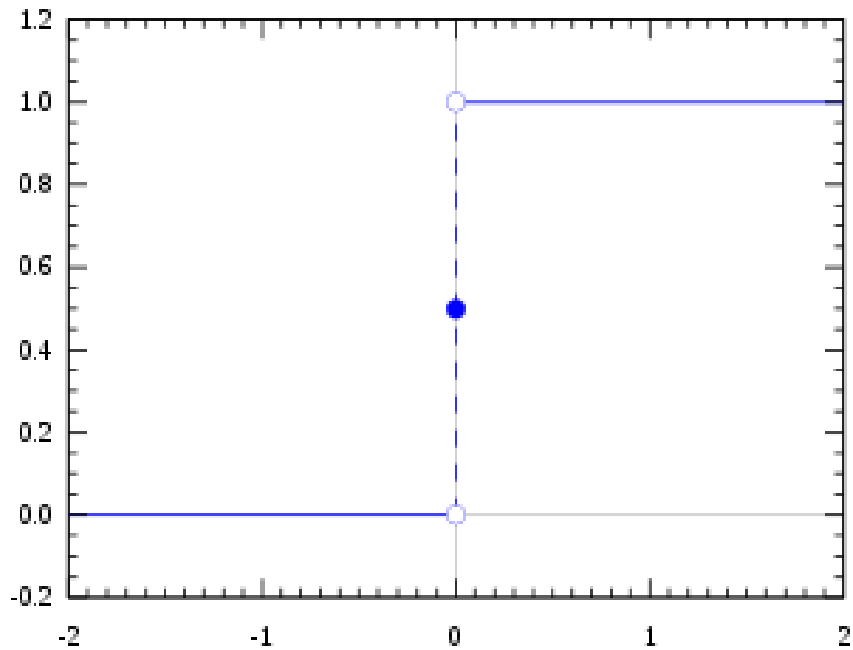


Figure 1: Binary Step Functions

Other examples of this notation would be C^0 where the function $f(x)$ is continuous but its derivatives are not. And on the other side of the spectrum we have C^∞ where a function is continuous and all of its derivatives are also continuous examples of this would be the sigmoid function $\sigma(x)$, note that this does not mean that the derivative is monotonic however there are activation functions

where in which all of its derivatives are continuous and has a monotonic first derivative.

3.2 Activation functions and their purposes

Activation functions are a crucial part of any neural network, this function is what lead to our output and subsequent calculations that tell us how we should update our weights and biases accordingly, that is to say that the activation function is the predictive function within in a neuron which tell us what the machine "thinks" the result should be.

In this section we will go over a few different activation function, and their typical use cases, some activation functions for example are more suited to regression problems and others to more suited to classification problems.

3.2.1 Sigmoid logistic function

First we have the Sigmoid logistic function, this function is defined as follows:

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

and it's derrivative as:

$$\sigma'(x) := \sigma(x)(1 - \sigma(x))$$

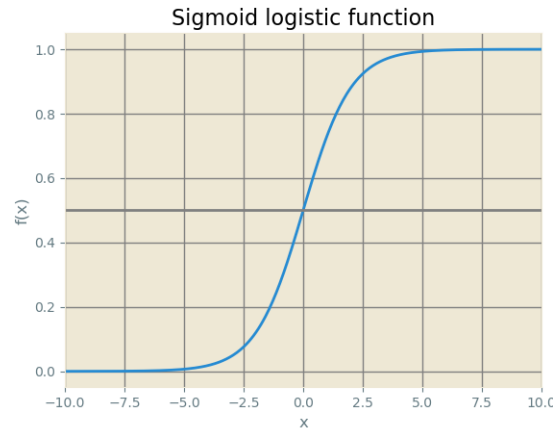


Figure 2

Some pros of the sigmoid function is that it is really very useful for the output layers of binary classification networks as it ranges from $[0,1]$, this is useful for when you want to get the probability of an output is a certain label (sometimes called class).

it is also a non-linear function which means for classifying complex datasets this is a fantastic function, and also since its range only goes from $[0,1]$ it cannot

output a number that will "blow up" the output to subsequent layers (if used in any middle layers) and will therefore not make our network diverge or behave chaotically.

However it has some fairly large cons associated when used in anything but the output layer of a network. This is because that as x gets very large $f(x)$ changes by a very small amount, this subsequently gives rise to a problem called the "vanishing gradient" problem where effectively the use of a sigmoid function in a fully connected layer can saturate the network with outputs that are either all 0 or all 1 because at the tail end of the first derivative of the sigmoid function the gradient is essentially 0.

3.2.2 Hyperbolic tangent

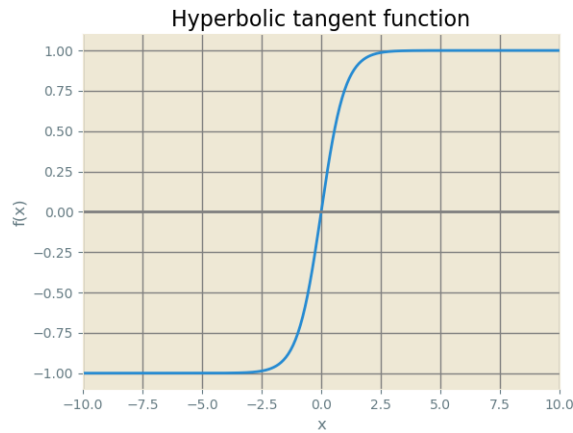


Figure 3

tanh defined by:

$$f(x) := \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$f'(x) := 1 - f(x)^2$$

Tanh is another sigmoidal function meaning it has essentially all the same benefits as $\sigma(x)$ however the tanh function's range is large $[-1,1]$, and it also has larger derivatives than $\sigma(x)$, this can allow for quicker convergence for finding the global minimum of a given loss function. However it should be noted for small datasets this isn't really an observable benefit, however for deep networks it can provide an observable benefit.

LeCun's tanh LeCun's function is defined as:

$$f(x) := 1.7159 \tanh\left(\frac{2x}{3}\right)$$

and its derivative:

$$f'(x) = 1.14393 \operatorname{sech}^2\left(\frac{2x}{3}\right)$$

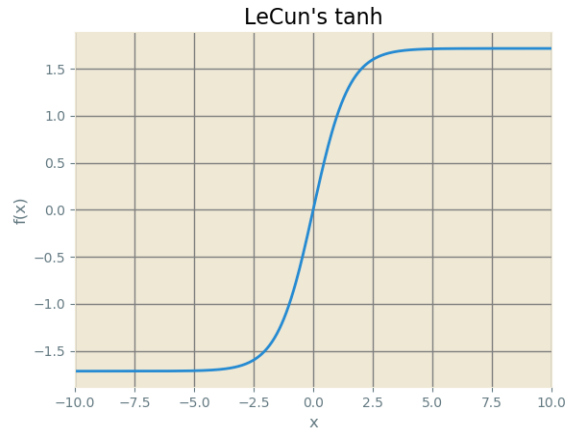


Figure 4

This is a modification of the hyperbolic tangent developed by Yan LeCun [13]. The whole point of this augmentation to the hyperbolic tan function is to speed up convergence, ideally when you normalise your inputs i.e have a mean of 0 and standard deviation of 1, LeCun shows that this keeps the variance of the output as pretty close to a value of 1 due to the gain of the function being roughly 1.

3.2.3 ReLU

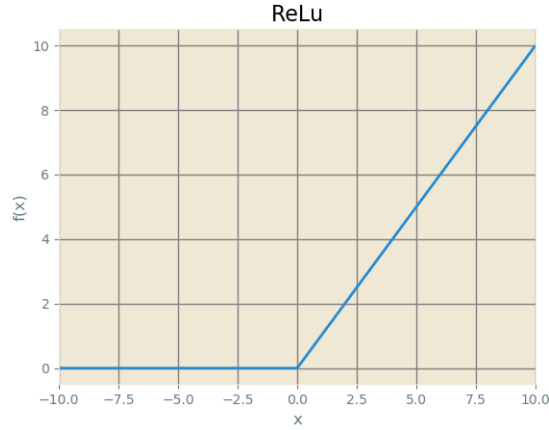


Figure 5

The ReLU function is defined as:

$$f(x) := \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$$

and its first derivative as:

$$f'(x) := \begin{cases} 0 & \text{for } x \leq 0 \\ 1 & \text{for } x > 0 \end{cases}$$

The main use of a ReLU (rectified linear unit) function is for use in the middle (hidden) layers of a network, this is because of a few key reasons. The first reason is because the gradient of a ReLU function will not vanish, or is far less likely to vanish compared to its sigmoidal counterparts. This is because when the function is greater than 0 it just returns the input undisturbed.

The next advantage a ReLU has is it can create a more sparse data flow in your network, where as a sigmoidal function will have a far denser data flow, this is because the ReLU function is 0 when the inputs is negative, where the same cannot be said for sigmoidal functions, where they will likely always be some output from a given neuron, this can lead to faster convergence in a given network.

It is also a very easy to compute function compared to a sigmoidal type function, as it only has to choose between two options, that is if x is negative return 0, if x greater than or equal to 0 return x, no complex computations are needed so for large datasets this is fantastic, as it can speed up convergence for densely connected layers.

However, the ReLU function can have quite large drawbacks. One of these is that because ReLU is unbounded as it can output any value in the interval $[0, \infty]$

it can blow up outputs for a given layer. This can lead to numerical instability in the loss function sometimes, and if you were to graph the loss function it may start oscillating and never get to a low enough level to be considered satisfactory.

The next downside, is perhaps the most well know, the so called "Dying ReLU" problem. Because the ReLU function returns a value of 0 for all negative values of the input, a neuron can effectively die because either a large negative bias or weight is affecting the input, this leads to the neuron outputting nothing other than 0 and will never change until the networks state is reset, however this always has a chance of happening.

The next thing to mention, is that functions like the ReLU function should not be used on the output layer of a classification network. The reason being is that you can never get a good probability for your label. For example let us take a network that performs binary classification, where in which the outputs are labeled 0 or 1.

A ReLU function can't give us a sensible interpretation because it is an unbounded function, where if x gets very large than y also gets very large this can lead to some answers which disagree with the logic of classification, where a number greater than 1.0 does not make sense. Also for a negative value, ReLU returns 0, however a negative value in for example, the sigmoid logistic function does not provide 0 for all negative values, instead it can map negative values to low probabilities.

Leaky ReLU A further modification of the ReLU function, it should be noted that the figure below has a value of $0.1x$ for $x \leq 0$ to show the pronounced effect of a decreasing gradient.

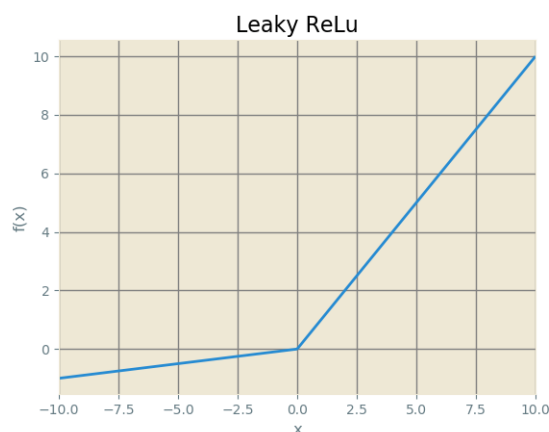


Figure 6

a leaky ReLU function is defined as:

$$f(x) := \begin{cases} 0.01x & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$$

and its first derivative as:

$$f'(x) := \begin{cases} 0.01 & \text{for } x \leq 0 \\ 1 & \text{for } x > 0 \end{cases}$$

This modification to the ReLU function fixes the biggest most fundamental problem, the "dying ReLU" problem. as for negative values of the inputs, it can return that value divided by a constant, usually 100.

This prevents neurons from outputting a zero constantly for negative inputs, and therefore the network will not behave chaotically and can achieve convergence.

3.2.4 Softmax

The Softmax function is defined as:

$$\frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \text{ for } i = 1, \dots, K$$

Which is to say take some vector say \vec{z} , we take this vector and divide it by all elements in \vec{z}

The Softmax function is a multi class extension of the sigmoid function, where for classification purposes it can distribute probabilities over a given n dimensional label space. It can also be used for binary classification just like the sigmoid function. However it will achieve the same result and is a bit more expensive to compute, therefore it is only ever used on the output layer for a multi label classification type network where as a regular sigmoid function cant do the job nearly as well because we need a vectorised output for our label space as it consists of more than usual number of labels for just binary classification.

3.3 Loss functions

Loss functions are perhaps the most important part of a neural network as this is where our comparison between our actual value and our predicted label are essentially compared. Let's talk about the different types of loss functions and compare their pros and cons.

3.3.1 MAE

MAE or mean absolute error, sometimes referred to as L1 loss is defined as follows:

$$MAE = \frac{\sum_{i=1}^n |y_a^i - y_p^i|}{n}$$

Where y_a^i is our actual output value and y_p^i is our predicted output from our network.

The calculation is quite simple, we simply take the difference of our actual and predicted value and then take its absolute value. This is a fairly inexpensive computation, which is good for large numbers of iterations, and it also is fairly robust in the sense that it is more resistant to outliers in a given dataset. This is because, unlike MSE, it does not rely on squaring the error term, so an outlier does not get squared and will not become a large initial error outputted from the loss function.

MAE is not treated in the same way as MSE when it comes to gradient descent. Mainly because the MAE function has a point of discontinuity at $x = 0$, instead we have to use a method called subgradients, which is actually a little more expensive to calculate, but ultimately achieves the same goal. however this can lead to multiple solutions and can therefore be unstable sometimes, In conclusion this loss function is great for when your data has outliers as it will minimise their effect on your model

3.3.2 MSE

MSE or mean squared error, sometimes referred to as L2 loss is defined as follows:

$$MSE = \frac{\sum_{i=1}^n (y_a^i - y_p^i)^2}{n}$$

Where y_a^i is our actual output value and y_p^i is our predicted output from our network.

MSE relies on squaring the error terms, now squaring a number can be more computationally expensive, the upsides are pretty big, as the calculations for backpropagation are not only elegant but easy to calculate also, it also is a very stable loss function as it is convex, it will have only one unique solution as opposed to MAE which can have multiple solutions. Or a region of zero gradient.

The downsides of MSE are quite punishing however. If you are working with data where you need to include outliers, you should not use MSE.

The reason for this is simple as the MSE tries to basically adjust for the outliers in your data which can impact your final result substantially if performing a regression task

It should be noted that MSE and MAE are typically used for regression networks and not for classification. Just as an aside, we can actually derive MSE from the MLE of a normal distribution here.

Let us start by assuming our data is normally distributed:

$$P(y|x) = \mathcal{N}(y_a; y_p, \sigma^2)$$

$$\mathcal{L}(y|x) = \prod_{i=1}^m p(y^i|x^i)$$

$$\mathcal{L}(y|x) = \prod_{i=1}^m \sqrt{\frac{1}{2\pi\sigma^2}} \cdot e^{\frac{-1}{2\sigma^2}(y_a^i - y_p^i)^2}$$

Now, we take logs:

$$\log(\mathcal{L}(y|x)) = \log\left(\prod_{i=1}^m \sqrt{\frac{1}{2\pi\sigma^2}} \cdot e^{\frac{-1}{2\sigma^2}(y_a^i - y_p^i)^2}\right)$$

$$\log(\mathcal{L}(y|x)) = \sum_{i=1}^m \log\left(\sqrt{\frac{1}{2\pi\sigma^2}}\right) + \log\left(e^{\frac{-1}{2\sigma^2}(y_a^i - y_p^i)^2}\right)$$

$$\log(\mathcal{L}(y|x)) = \sum_{i=1}^m \log\left(\sqrt{\frac{1}{2\pi\sigma^2}}\right) - \frac{1}{2\sigma^2}(y_a^i - y_p^i)^2$$

Since the first term has no y_p^i term we can ignore this term, this then becomes:

$$-\log(\mathcal{L}(y|x)) = \sum_{i=1}^m \frac{1}{2\sigma^2}(y_a^i - y_p^i)^2$$

Again σ^2 also does not depend on y_p^i we can also ignore that, we could also ignore the $\frac{1}{2}$ however it is nice to include as it cancels thing out when taking the derivative.

Now, all we have to do is take the mean of this sum which means dividing by m which leaves us with:

$$MSE = \frac{1}{2m} \sum_{i=1}^m (y_a^i - y_p^i)^2$$

However for Binary classification i.e our data can be sorted in to two classes, this data is actually from a Bernoulli distribution instead, and thus we cannot use this for classification in most scenarios. However in [3] and explained in [6] it can be the case that a classification problem can also be a regression problem in disguise and can be used in such circumstances.

3.3.3 Cross Entropy Loss

Cross entropy loss comes in a couple of different flavours, Binary cross entropy and multi class cross entropy is the go to loss function for classification problems. And is defined as follows:

$$L := -(y \log(p) + (1 - y) \log(1 - p))$$

Where y is a binary indicator either 0 or 1 which indicates if the class is the correct class for a given observation and p is the probability calculated for said observation, so if $y = 1$ the function reduces to $-\log(p)$ and for $y = 0$ this reduces to $-\log(1 - p)$.

The multi class version is defined somewhat similarly, however for simplicity I will choose to write this slightly differently as it can be seen here but with slightly different variable names [17].

$$L(p, q) = - \sum_{\forall x} p(x) \log(q(x))$$

In [17] we can a simple example of why this is a very good multi class classifier, we have an example of an actual observation value vector \vec{y}_a and a vector of predicted results \vec{y}_p where $y_a = [1, 0, 0, 0, 0]$ and $y_p = [0.1, 0.5, 0.1, 0.1, 0.2]$ so we can almost immediately see that the example network has made a bad prediction and the loss will be $-1 \cdot \log(0.1)$ which is ≈ 2.303 as all other terms in y_p are multiplied 0 as the true value vector has a 1 in the first slot of the vector and zeros in the remaining 4 slots.

As seen in [17] we can clearly see how cross entropy punishes bad results as we get a result of $L \approx 2.303$, which is a pretty bad result. But the formula in question does not depend on the values of the incorrect probability predictions in y_p as cross-entropy loss looks at the value of a prediction for the first element of the prediction vector, or in other words the element of the prediction vector which is a numerical 1 or the true category.

This is probably the most common loss function in most neural networks, as when combined with the softmax activation function on the output layer of a given classifier network can yield very good results in loss reduction.

3.3.4 Hinge Loss

This is perhaps the most complicated loss function in this paper, the hinge loss or multi class SVM classification, SVM standing for support vector machine, is an advanced loss function that not only penalises miss-classifications but also non-confident correct classifications. This is because essentially what the hinge loss does is it draws a decision boundary or decision plane between data points of different classes and calculates the maximum boundary between them.[20]. As is demonstrated in [20] we see that the actual computation for the loss function is actually quite simple for binary classification:

$$\sum_i \max(0, 1 - y_i * h_{\theta}(x_i))$$

A binary classification is outlined in [20] as follows: "Our labels y_i are either -1 or 1, so the loss is only zero when the signs match and $|(h_{\theta}(x_i))| \leq 1$ with h_{θ} being our prediction. For example, if our score for a particular training example was 0.2 but the label was 1, we'd incur a penalty of 1.2, if our score was 0.7 (meaning that this instance was predicted to have label 1) we'd still incur a penalty of 0.3, but if we predicted 1.1 then we would incur no penalty"

so we can see here that above a certain threshold value we actually incur 0 penalty, and similarly when we have a value that is close to our true label value of -1 but it still does not exceed -1 then we will still incur some sort of penalty

as it may still be on the boundary. This is on top of the fact that an incorrect result is still heavily punished just as a cross entropy loss function would.

or more generally formulated for multi class classification in [14]:

$$L = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

An example in [14] can be seen as we have a 3 class problem with various predictions made. Our classes being three photos of animals, that of a dog, cat and horse respectively. These labels have predictions made and we can see just how simple the calculation can be. The function performs a summation of max functions, as discussed in the section under ReLU activation functions the max function is very in expensive as it just has to choose the bigger number, and in this case sum all of them together to then get the total loss. We can see in particular the "threshold" for the cat label is 3.28 so any values less than 3.28 will be chosen to a 0. This results in an overall loss of zero. In reality we are checking the values of the other labels, if they are negative and the predicted label threshold is positive we can immediately say without any calculation that the result will be 0, similarly so for any value for less than a given threshold.

Hinge loss also gets rid of our dependence on the softmax function, or for that matter any non linear function off of our output layer and just use the hinge loss function for backpropagation, and interestingly if we modify the hinge loss function some smooth approximation of the hinge loss function it behaves similarly to logistic regression almost like a regular sigmoid-cross entropy model, this can be seen in this amazing book by Christopher M Bishop in section 7.1.2. [5]

Overall the aim of the hinge loss is to just correctly classify classes within your data, even if it does include some miss-classifications, this is by design, as if we did not allow miss-classification to occur with in the margins of our boundary, we could actually end up with the loss function being too sensitive to outliers in our data.

this creates a maximum margin classifier with a hard margin, where points cannot enter in the margin in between two classes of data. We can see this in the figure below, that we have a decision boundary with a margin. If this is a soft margin and allows miss-classification this may very well yield better classification for future examples.

As opposed to a margin which does not allow this. This boundary can effectively become squished up against the red points we see here in our figure if we then get a blue point right next to the red point for example.

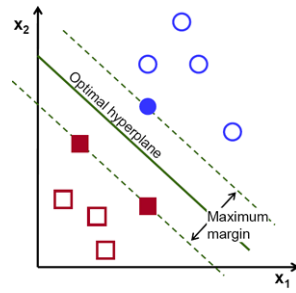


Figure 7: Maximum margin classifier

3.4 Single layer perceptron

Our simplest form of network is the single layer perceptron network or SLP. The main use of the SLP network is usually in classification type problems, where the goal is to classify between different types of data. However as previously mentioned, an SLP can only separate linear datasets i.e we can draw a boundary between the two types of data, any non-linear dataset must be separated via a MLP network.

The structure of this network consists of our example network one input neuron with a bias neuron. Typically by convention the bias is initialised with a value of 1, this is to help simplify the backpropagation algorithm, the bias will have a weight usually denoted as b . Weights on the other hand can be initialised as random numbers and are usually denoted as w_i for $I \in \mathbb{Z}_0^+$.

However it should be noted that poor initialisation can lead to very bad results within a network, this is however not a large concern within our example of an SLP network, but when we discuss MLP networks it will be an important topic.

The network is essentially an algorithm that takes inputs transforms them by summing the product of a given input and its weight and the bias of the neuron which gets fed into an activation function, this activation function then gives us the output we needed to calculate the changes we need to make to our weights and biases through observations in the calculated output compared to the actual output.

But how are the weight and bias values altered? For this we will need to explain the back propagation algorithm.

3.4.1 Backpropagation and feed-forward data flow

A key feature of the SLP (and the MLP for that matter) architecture is that the data only flows one way, namely forward which means data that is transformed by one layer gets directly outputted to the next layer and is not fed back to the same layer or previous layer(s).

This is done via a process called gradient descent, the reason it is called gradient descent is because we are trying to minimise our loss function.

Why minimise the loss function of the network instead of simply minimising the error?

Here is why: let us say from our output we have two sets of four values ranging from 0 to 1 that represent our error.

$$E_1 = 0.2 + 0.2 + 0.2 + 0.2 = 0.8$$

$$E_2 = 0 + 0 + 0 + 0.7 = 0.7$$

We can see from here why we don't want to strictly minimise sum of the errors, if we do that our network will look at the second set and determine that this is a more suitably set to train rather than the above set, after all if we give the

computer an instruction to minimise the error it will choose the lowest total error.

This is quite clearly a problem as we can see E_2 has a value of 0.7 this is a very very large error as opposed to the errors in E_1 which are all very small errors.

we can account for this by choosing an appropriate loss function that is based on a little statistical intuition, that is. If we square the errors and then sum them we get a better representation of how impactful each individual error truly is. This is discussed in section 3.4.2 (MSE) So E_1 and E_2 now become:

$$E_1 = 0.2^2 + 0.2^2 + 0.2^2 + 0.2^2 = 0.16$$

$$E_2 = 0^2 + 0^2 + 0^2 + 0.7^2 = 0.49$$

We can now see the total error has massively decreased for E_1 and now the total error for E_2 is much higher and a truer representation of the error of the that set of values.

To make the maths a bit simpler instead of our loss function being $L = (y_a - y_p)^2$ we can instead choose $L = \frac{(y_a - y_p)^2}{2}$ where y_p is the predicted value from our network and y_a is our actual output. This is the mean squared error from our section on loss functions.

We can finally talk about the backpropagation algorithm itself, this can be thought of as the rate of change of the loss function with respect to a given weight, and therefore a partial derivative.

Now, for a simple SLP network the following calculation sufficiently explains how the weights will be updated. For this example we will use the Sigmoid activation function which is define by $f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$, with an input say z , we have the following equations.

Our loss function is defined as:

$$L := \frac{(y_a - y_p)^2}{2}$$

This Loss function also has the advantage of being a convex function which means we will find the global minimum when we apply gradient descent as opposed to a non convex function which can have multiple local minima.

Our input:

$$z_1 = w_1 x_1 + 1 \cdot b$$

The predicted outcome which is our input that goes into our activation function:

$$y_p = r_1 = \sigma(z_1)$$

We have to start by applying the chain rule, we first have to find the derivative of C with respect to the predicted outcome: r_1 and our derivative of our activation function to eventually find our end goal of $\frac{\partial C}{\partial w_1}$ which tells us how much to update our weight by.

Now, for the intermediate steps:

$$\begin{aligned}\frac{\partial r_1}{\partial z_1} &= \sigma(z_1)(1 - \sigma(z_1)) \\ \frac{\partial L}{\partial r_1} &= \frac{\partial \left(\frac{(y_a - r_1)^2}{2} \right)}{\partial r_1} = \frac{\partial \left(\frac{y_a^2 - 2y_r r_1 + r_1^2}{2} \right)}{\partial r_1} = r_1 - y_a \\ \frac{\partial L}{\partial z_1} &= \frac{\partial L}{\partial r_1} \frac{\partial r_1}{\partial z_1} \\ (r_1 - y_a)\sigma(z_1)(1 - \sigma(z_1)) &= (\sigma(z_1) - y_a) \cdot \sigma(z_1)(1 - \sigma(z_1))\end{aligned}$$

Now, that we have all the intermediate steps out of the way we can now begin to find $\frac{\partial C}{\partial w_1}$.

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z_1} \frac{\partial z_1}{\partial w_1}$$

where:

$$\begin{aligned}\frac{\partial z_1}{\partial w_1} &= x_1 \\ \therefore \frac{\partial L}{\partial w_1} &= (\sigma(z_1) - y_a) \cdot \sigma(z_1)(1 - \sigma(z_1)) \cdot x_1\end{aligned}$$

The next part is crucial, we have to introduce a learning rate η . This learning rate determines how much of the gradient we should adjust our weight by. The reason for this is because if the learning rate is too high the network may never converge to the optimal weights and biases, if the learning rate is too low, it will just take a very long time to converge, typically this rate is set to 0.01, so our final formula for how to update our weights is as follows.

$$w_1^{new} = w_1 - \eta \frac{\partial L}{\partial w_1}$$

It is also similar for our bias:

$$b^{new} = b - \eta \frac{\partial L}{\partial b}$$

It should also be noted, that if we say had another input node say w_2 then the we would just have $w_2^{new} = w_2 - \eta \frac{\partial L}{\partial w_2}$.

We can keep on applying this algorithm, however when we start to get into very large datasets this method of gradient descent starts to become very very slow. This is because for right now this would just be for 1 point in a simple SLP with one input layer however when we start using a more robust SLP like one with just two inputs that are say x and y coordinates on a graph and we are trying to optimise for the slope and intercept of the line in between our data points this get very expensive very fast and as a result our process takes an extremely

long time.

That is because we will be summing and computing our loss for all inputs from our dataset which, provided the dataset is sufficiently large, which it definitely will be. This can be combated with stochastic gradient descent where we take a random subset of our training dataset and use that to then update our weights and biases accordingly, instead of calculating the loss for every input we have then adjusting. so instead of $L = \frac{1}{2m} \sum_{i=1}^m (y_a^i - y_p^i)$ we have $L = \frac{1}{2n} \sum_{i=1}^n (y_a^i - y_p^i)$

where $n < m$ which leads to $\frac{1}{2n} \sum_{i=1}^n (y_a^i - y_p^i) \approx \frac{1}{2m} \sum_{i=1}^m (y_a^i - y_p^i)$

3.4.2 The XOR gate problem

The XOR gate problem is a well known problem in machine learning, where in which it can be demonstrated that an SLP cannot converge for the inputs and outputs of the XOR gate truth table and for any set of data that is non linear for that nature.

Here is why an SLP cannot solve the XOR problem, if we think of an SLP as summing a set of linear combinations we can reason through 4 equations that it is impossible for us to solve the XOR problem through a set of linear combinations. first the truth table for the XOR gate is as follows:

p	q	$p \oplus q$
0	0	0
0	1	1
1	0	1
1	1	0

Equations of this table can be represented as follows:

$$(0 \times w_1) + (0 \times w_2) + w_0 \leq 0 \iff w_0 \leq 0$$

$$(0 \times w_1) + (1 \times w_2) + w_0 > 0 \iff w_0 > -w_2$$

$$(1 \times w_1) + (0 \times w_2) + w_0 > 0 \iff w_0 > -w_1$$

$$(1 \times w_1) + (1 \times w_2) + w_0 \leq 0 \iff w_0 \leq -w_1 - w_2$$

Where w_1 and w_2 are our weights and w_0 is our bias weight sometimes called a threshold.

We can clearly see equations 2 and 3 contradict equation 4 as w_0 is both bigger than $-w_1$ and $-w_2$ but somehow less than the difference of them, this is a contradiction, and clearly impossible.

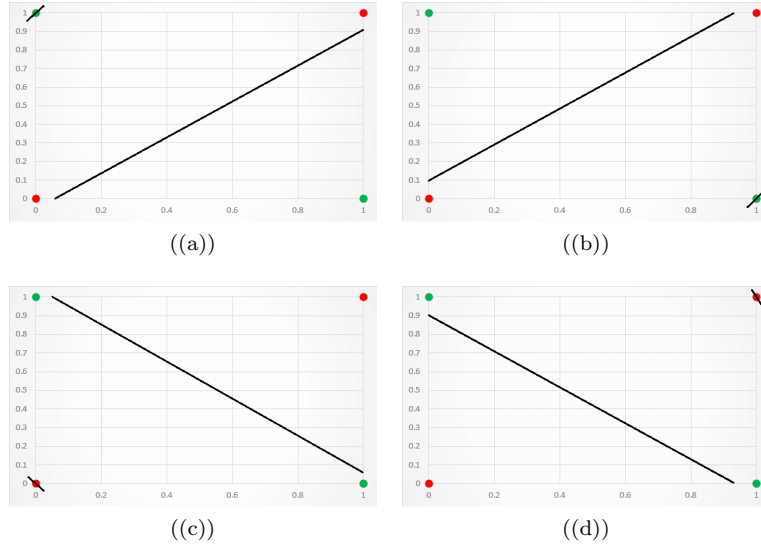


Figure 8: XOR visualisation.

As we can see in the above figure as provided by [2]. It is impossible to separate this set with just one line, this makes sense as an SLP is really just finding a solution for $\sum_{i=1}^n w_i x_i + b$, this is the equation of a straight line with w as the gradient and b as the intercept and as we can see, one line is not enough to separate this set, we in fact need two lines.

however we will see in the next section this can be solved when introducing something new, a middle layer in between our input layer and output layer, the hidden layer.

3.5 Multi layer perceptron

A MLP or multi layer perceptron is a network architecture which generally has 1 input layer, n number of hidden layers and a single output layer, where each layer can have any number of neurons and all layer are said to be densely connected. This means each neuron in a given layer is connected as an output to every other neuron in the layer in front and behind it.

As we saw in our earlier section we had a problem when it came to the XOR gate inputs feeding into a single layer perceptron. This is because of the way SLP's deal with equations as if they are linearly separable, and as we showed the XOR gate is not linearly separable. We need the introduction of a new layer to transform our data.

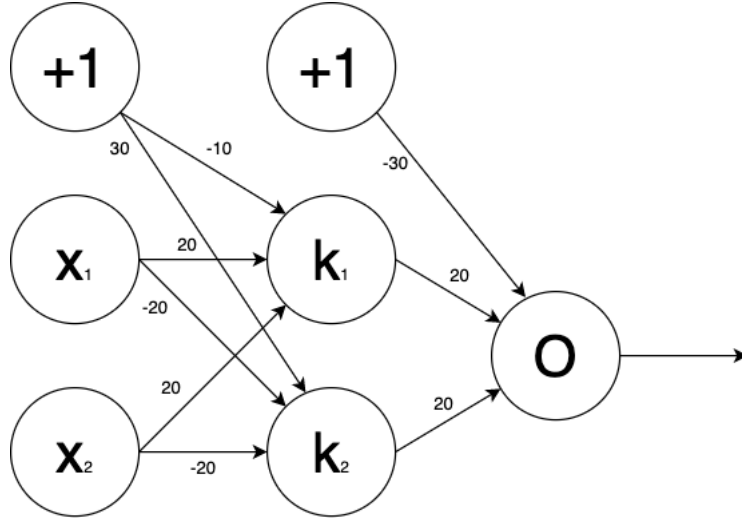


Figure 9: MLP for solving XOR

Now, if we go through our network for all the truth table values of the XOR gate we will see that it does indeed solve the XOR problem.

For our diagram the activation functions shall be $\sigma(x)$.

Our equations for the output of the neurons in this layer will be:

$$\sigma(20x_1 + 20x_2 - 10)$$

$$\sigma(-20x_1 - 20x_2 + 30)$$

And for our output layer will be:

$$\sigma(20k_1 + 20k_2 - 30)$$

Lets substitute the values from top to bottom of the truth table in section 3.5.2.

For $x_1 = 0, x_2 = 0$ the hidden layer:

$$\sigma((20 \times 0) + (20 \times 0) - 10) \approx 0$$

$$\sigma((-20 \times 0) + (-20 \times 0) + 30) \approx 1$$

And the output layer:

$$\sigma(20 \times 0 + 20 \times 1 - 30) \approx 0$$

For $x_1 = 1, x_2 = 0$ the hidden layer:

$$\sigma((20 \times 1) + (20 \times 0) - 10) \approx 1$$

$$\sigma((-20 \times 1) + (-20 \times 0) + 30) \approx 1$$

And the output layer:

$$\sigma(20 \times 0 + 20 \times 1 - 30) \approx 1$$

For $x_1 = 0, x_2 = 1$ the hidden layer:

$$\sigma((20 \times 0) + (20 \times 1) - 10) \approx 1$$

$$\sigma((-20 \times 0) + (-20 \times 1) + 30) \approx 1$$

And the output layer:

$$\sigma(20 \times 1 + 20 \times 1 - 30) \approx 1$$

For $x_1 = 1, x_2 = 1$ the hidden layer:

$$\sigma((20 \times 1) + (20 \times 1) - 10) \approx 1$$

$$\sigma((-20 \times 1) + (-20 \times 1) + 30) \approx 0$$

And the output layer:

$$\sigma(20 \times 1 + 20 \times 0 - 30) \approx 0$$

As we can see from our equations we have solved the XOR problem with the use of our new hidden layer. This is an idealised system meaning we have chosen the weights and biases before hand to solve the XOR problem, in real world practice these values would be updated via the backpropagation algorithm we mentioned earlier

3.5.1 Learning rate optimisers

Now, that we are going into deeper networks a primary concern is of network convergence.

We briefly mentioned learning rates before in our backpropagation section. But choosing a value for this learning rate is almost certainly non-workable as choosing a static learning rate.

Another name for an optimiser of this nature is an adaptive learning rate. The goal is to vary the learning rate according to the loss value of the network per iteration.

By far the most common learning rate optimiser is the ADAM optimiser. ADAM stands for adaptive moment estimation and borrows ideas from two other learning rate optimisers; RMSProp and Momentum.

It is also worth noting that it is worth while to vary this learning rate over time, by which I mean scheduling the optimiser. Such scheduling is often referred to as a cyclical learning rate schedules. This prevents our network from finishing local minima instead of a global minima. This is discussed in great detail in the following paper titled Train 1 get M for free [9]

First we will start with the definitions of the two latter optimisers then move onto the ADAM optimiser.

Momentum The idea of momentum is to take into account past gradients to speed up the standard gradient descent process.

There are two ways to think of momentum intuitively. That is one being a physical concept, while the other being abstract. Let's go over what these two intuitions are.

The first will be a physical intuition. Where the idea of this algorithm is similar to the idea of a ball rolling down a curve. Like a horse saddle or bowl if you will, we are trying to find the minimum point of this surface.

We are trying to minimise the change in the vertical direction in this case, as we want the vertical change to be close to zero. but have this horizontal change be able to find this minimum point.

The second would be a slightly more abstract concept, which I personally prefer for this case. Where if you picture various contours that make a contour plot with the minimum at the centre of this plot. We want to slow down the learning process in this vertical axis, let's call this b and a change in the horizontal axis, let's call this W . We want to minimise the change in, that is $db = 0$ whilst speeding up the horizontal learning that being dW .

We can define momentum as:

$$v_{dW} = \beta v_{dW} + (1 - \beta) \frac{\partial L}{\partial W}$$

$$v_{db} = \beta v_{db} + (1 - \beta) \frac{\partial L}{\partial b}$$

$$W = W - \alpha v_{dW}$$

As we can see here there is no db present. This is because when we use this in conjunction with our SGD (stochastic gradient descent) we will end up seeing v_{db} go to 0

Now, let's define just what all these symbols mean. v_{dW} & v_{db} are our changes in the "horizontal" and "vertical" directions respectively, β is some hyper parameter which is chosen, the partial derivative of the loss function is part of the gradient descent. W is our weight tensor and α is our learning rate.

RMSProp RMSProp is similar to that of momentum where the second analogy (abstract analogy) stays the same, RMSProp stands for root mean square proportional, where by RMSProp, unlike momentum, does not look at previous gradients to speed up convergence.

Instead it calculates a weighted average and then divides the learning rate by this average.

Let's define RMSProp:

$$s_{dW} = \beta s_{dW} + (1 - \beta) \left(\frac{\partial L}{\partial W} \right)^2$$

$$s_{db} = \beta s_{db} + (1 - \beta) \left(\frac{\partial L}{\partial b} \right)^2$$

$$W = W - \alpha \frac{\frac{\partial L}{\partial W}}{\sqrt{s_{dW} + \varepsilon}}$$

$$b = b - \alpha \frac{\frac{\partial L}{\partial b}}{\sqrt{s_{db} + \varepsilon}}$$

Now, it should be noted that even though we have used β again as our hyper parameter it is not the same as the one as in our momentum description.

Now, the symbols mean the same things with W being our "horizontal" and b being the "vertical" of our contour surface where W is our weight tensor. There is also a small value in the form of ε to make the denominator nonzero. As if s_{dW} gets close to 0 then we will start having convergence issues. This constant balances this out so we don't encounter such a problem. A typical value is something in the range of 10^{-8} .

ADAM Adaptive momentum estimation or otherwise know as ADAM is a combination of the two previous ideas of RMSProp and momentum. Once again the goal of the ADAM algorithm is to speed up convergence in our network.

ADAM is therefore defines as such:

$$\begin{aligned}
v_{dW} &= \beta_1 v_{dW} + (1 - \beta_1) \frac{\partial L}{\partial W} \\
v_{db} &= \beta_1 v_{db} + (1 - \beta_1) \frac{\partial L}{\partial b} \\
s_{dW} &= \beta_2 s_{dW} + (1 - \beta_2) \left(\frac{\partial L}{\partial W} \right)^2 \\
s_{db} &= \beta_2 s_{db} + (1 - \beta_2) \left(\frac{\partial L}{\partial W} \right)^2 \\
v_{dW}^{corrected} &= \frac{v_{dW}}{(1 - \beta_1^t)} \\
v_{db}^{corrected} &= \frac{v_{db}}{(1 - \beta_1^t)} \\
s_{dW}^{corrected} &= \frac{s_{dW}}{(1 - \beta_2^t)} \\
s_{db}^{corrected} &= \frac{s_{db}}{(1 - \beta_2^t)} \\
W &= W - \alpha \frac{v_{dW}^{corrected}}{\sqrt{s_{dW}^{corrected} + \varepsilon}} \\
b &= b - \alpha \frac{v_{db}^{corrected}}{\sqrt{s_{db}^{corrected} + \varepsilon}}
\end{aligned}$$

We can now move on to talk about what these symbols mean and what is going on here in these equations. The v and s variables we have seen before in momentum and RMSProp and remain the same. We have 3 hyper parameters this time which need tuning. For α we still have to pick a value for in the normal sort of way. But for β_1 and β_2 it is slightly different. The original authors of the ADAM paper recommend a value of 0.9 for β_1 , 0.999 for β_2 and 10^{-8} for ε [12].

Next we have these corrected terms popping up because in our calculation we have a bias in our uncorrected calculation, that being $(1 - \beta_1)$ and $(1 - \beta_2)$ respectively. We do this as the first to remove our initial guess of our first iteration, as this will be dominating the other terms in our iterations.

Lastly, we have our updates for W our weight tensor. And b our bias tensor, or the "vertical distance", noting that db again should average to 0.

3.5.2 Initialisation

Initialisation or weight-initialisation is very important in any neural network. This is because if weights are poorly initialised, that is they have large negative values the network may not even function at all as a portion of the neurons will be inactive and just produce zeros. To solve this problem we have to come up with a way to make sure that our initialised weight matrix has sensible values which won't result in our network being non functional.

We could not continue this discussion without mentioning the landmark paper by Xavier et al 2010 [7]. This paper is noteworthy as it outlines a robust experimentation process when initialising a weight matrix W_{ij} where I is the number of weights in the j th layer. Where W_{ij} is defined as $W_{ij} \sim U\left[\frac{-1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$ where U is a uniform distribution $U[-a, a]$ and n is the size of the previous layer.

However an alternative can be used to this, that is instead of using a uniform distribution, a normal distribution may be more preferential as we can control the mean and standard deviation of the distribution for the weight matrix (tensor). This initialisation method can be thought of as a random set of weights drawn from a normal distribution $N(\mu, \sigma)$ where in our case $\mu = 0$ and $\sigma = \sqrt{\frac{1}{inputs+outputs}}$ where the inputs are the neurons of layer n and the outputs will be the number of neurons into the $(n+1)$ th layer.

This type of Xavier normal initialisation shows a good performance increase in terms of the saturation with respect to a given neuron in a given layer. This helps to speed up the learning process and thus our network can converge to an accurate prediction more quickly as well as becoming more stable during the epoch iteration process i.e not having a ton of dead neurons everywhere. We can later use this type of initialisation in our implementation of the theory by using a high level API such as Keras which has been implemented into TensorFlow 2.

3.5.3 Architectural composition

The architectural composition of a neural network is designed for a specific task at hand. This means the actual overall shape of the network diagram is the more important part, the number of neurons in a layer is less important.

An example of this is the architecture of a very common network composition, that of the auto encoder. An auto encoder's primary function is to reduce noisy data from our inputs and return us the important bits of the data that we put in. For example if we put in an image of an object, the job of the auto encoder is to reduce the noise in said image. This has the added benefit of reducing the total amount of data in our image.

From this description it becomes clear we want as many layer 0 neurons as we do for layer n neurons to preserve the overall image. But we want to take steps to reduce this data, in turn reducing the number of neurons smaller and smaller, usually to two neurons. and then "mirror" this layer composition for the output.

As we can see we have used our intuition to design a network architecture that is both suitable and appropriate for our task of dimension reduction within an image. This serves as an example of why network architecture composition is very important when considering the overall goal of what our task may be.

3.6 Convolutional Neural Networks

Convolutional neural networks are a type of classification network that is most commonly used and well known for image classification, however these types of networks can be used for much more than just this.

Let's talk about image classification then. How a convolutional network can be adapted from MLP image classifier? Well that comes to a form of adding a Convolutional layer to the networks composition.

The idea of this layer is to reduce the amount of data flowing through the network, essentially reducing the number of calculations on how to update the weights and biases as the number of neurons in subsequent layers can be reduced.

But what is going on inside this convolutional layer? Well let us assume we are going to use our CNN for image classification, and for the sake of brevity we will say the image consists only of grey scale pixels, meaning they have a value between 0 and 1 where 0 corresponds to black and 1 corresponds to white.

We feed our image in through a flattened input layer. That is to say we take the dimensions of the layer and find out its total number of pixels and use this as the number for our inputs. Next after passing through perhaps a hidden layer or two, which allows the network to identify features about the image which may contribute to predicting the label of said image. It is then passed into our convolutional layer.

Here is where the real work gets done. A convolutional layer consists of a kernel, picture a grid. Which strides over our image, sums and then averages these values in this cell. The size and stride length of this grid can be varied of course. A kernel can also have what is called a dilation value, which is basically space between elements of a grid. This dilation factor is usually a constant, though it can be a tuple describing how to expand the grid in any dimension. For example a 3x3 kernel with dilation factor 1 is just a regular old 3x3 grid, a 3x3 grid with a dilation factor of 2 is a 3x3 grid that has each element separated by one pixel resulting in a 7x7 kernel. However later on we will probably keep this at a value of 1, though it doesn't hurt to experiment. A dilated kernel can result in faster run times as it can cover a larger global view with the same amount of parameters as an un-dilated version of the same kernel.

The effects of dilation can really be seen in [21] where you can use a dilated kernel to skip the whole sub sampling process using an un-dilated kernel.

Now, all is fine and well in averaging the averages grey scale value but then what do we do? As after all that hasn't reduced our data flow, it has just

transformed it. Well next what we are going to do is then pool the maximum value of our transformed image. That is to say put a max pooling layer after our convolutional one.

The max pooling layer is basically a region which takes the maximum value and shrinks the image down, thus reducing our data. Picture a square striding across our transformed image and then taking the maximum value and then creating a new image out of these maximum values. These new values are placed from the motion of this area striding over our image. If the kernel strides from top left to bottom right. The new and reduced image is created in the same motion.

We can repeat this process several times to reduce the data further. However reducing our data to much can sometimes result in nothing useful being learned from our network.

CNN's can be used for a wide variety of applications, far beyond image processing. You can even use them to identify sound signatures. This type of process is already used in the music identification app Shazam for example.

The deeper these networks get, the more powerful they can become, even by re-colouring old movie footage from the days of the silver screen, such a network is a temporal CNN as it deals with data that is time dependant. Such an example can be found here in [10].

One way of also explaining CNN's is through that of dimensional i.e 1 2 or 3 dimensions for a given kernel. The most typical would be a 2 dimensional kernel, that being of the type of image classification where our kernel grid slides across a given image. This 2d image can also have depth in the form of RGB values so even though the kernel is technically a 3 dimensional cube. Its motion is confined to 2 dimensions across an image.

Here are some figures to help visualise this dimensional kernel talk, these images come courtesy of one Shiva Verma and can be found in his blog [16].

First lets start at a 1 dimensional kernel:

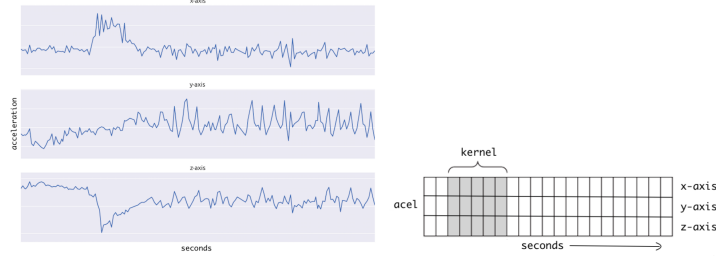


Figure 10: 1D convolution example

We can see in figure 9 an example of 1 dimensional data on the right and a 1 dimensional A 1 dimensional kernel is that in use for data where a kernel only has to move in 1 dimension. Something like sound or text on a line, not to dissimilar to that of the work done at Shazam.

Next let's talk about the most common case for classification, that of 2D kernel convolution. 2D kernel convolution as briefly discussed before can technically be thought of as a volume if using a standard RGB image where there is a red, green and blue layer respectively. However in black and white images this can be thought of as having a depth of 1, where the only thing that vary is how white or black the pixel is, Usually from 0 to 255 for 8 colour bits. In practice however we would squish these values into a range from 0 to 1 however to then pass into our network.

When we talk about RGB images however we need some way to detect features on all three colour channels. How we do this is actually treating the 2D images as a 3 dimensional structure say $n_h \times n_w \times n_c$ where these three terms correspond to the height width and number of channels respectively. We then apply what is called a filter to our image, before this filter term was just a single kernel "window" but in the case of an RGB image this needs to also have a depth parameter to account for the value of n_c being greater than 1. We can visualise this in the following figure:

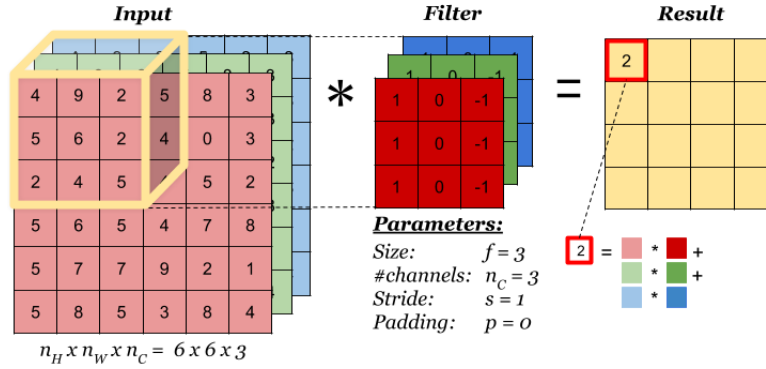


Figure 11: RGB kernel filter

It is worth noting that after we have applied this filter to our image we have reduced the number of dimensions of this input from $6 \times 6 \times 3$ to just 4×4 where the number of channels, or the depth of our input to just 1. However that would be for just one feature, so we could end up after the filtering process with a smaller over all image but the number of channels or the depth may be different depending on what features we look for. For example if we had a filter that only detects vertical edges, we would indeed end up with a $4 \times 4 \times 1$ but if we wanted to detect horizontal edges as well as vertical edges we may in some cases, depending on the filter we use have to sue tow filters as opposed to just the one. this results with us having two 4×4 outputs which we can then stack to create a $4 \times 4 \times 2$ stack.

Filters can also take on their own internal values and can have a different effect applied to the image as a result. Before in the case of grey scale we just passed a kernel over the image and then pooled the layers. However we can do more with these filters. For example we can apply an edge detection filter. This as you might imagine can detecting edges of of objects, we can even apply a Gaussian blur to our image. The reason I mention these two in particular is because the edge detection filter is very sensitive to sharp and noisy images, where the filter will detect these sharp lines as definite edges when it just could be an artifact in the image. To counter act this we can try and use a Gaussian blur filter as to smooth out the image and allow for better edge detection. We can use two main types of edge detection filters either Sobel or Laplacian, their effects can be demonstrated here courtesy of [1]:

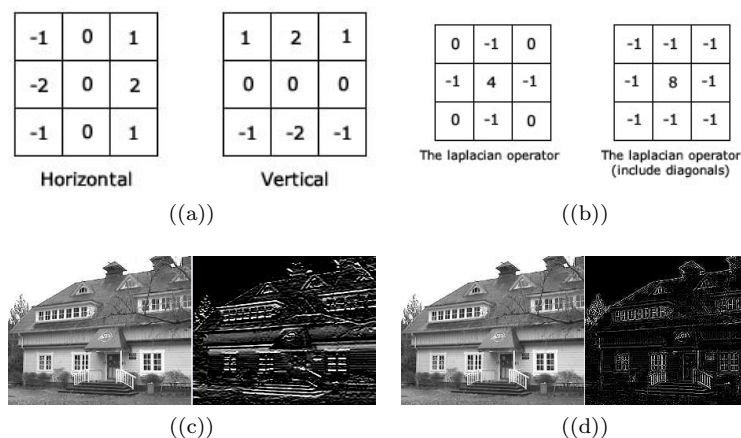


Figure 12: Edge detection kernels.

What we have going on here is the Sobel edge detection is a first order derivative approximation versus the Laplacian filter which is a second order derivative approximation. Noting that both kernels are only 3x3 and you may get a slightly better result using a slightly larger kernel say 5x5.

The final type we will discuss would be 3 dimensional kernel convolution. This type of convolution I find particularly interesting as it deals with volumes and not areas. This type of kernel can be used for volumetric data such as MRI scans of someones brain for detecting abnormalities which can potentially aid as an assist tool for MRI technicians and radiologists when looking for tumors for example. Here is an example of what this kernel convolution looks like:

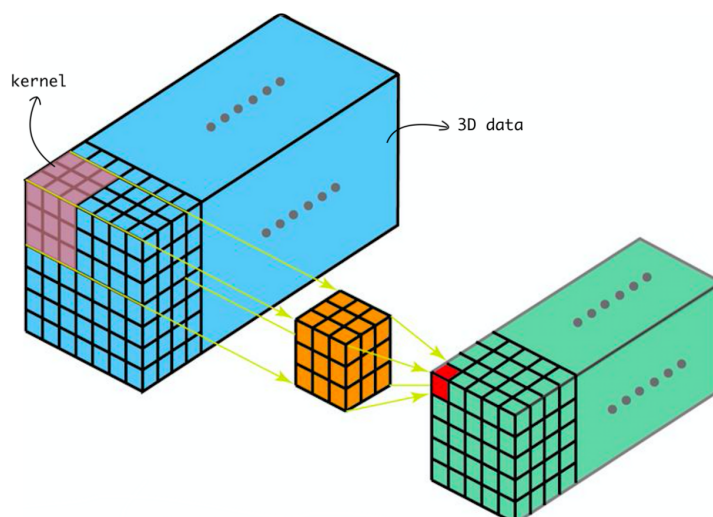


Figure 13: 3D convolution example

Another important aspect about CNN's is to vary and or transform elements of the dataset to account for irregularities and variations from potential test images, this could be noise or perhaps even rotation of the images. We can build this into our network but rotating images and even adding noise if we so wish to help our network account for these irregularities. These are called data augmentations and greatly aid in reducing over fitting in our network. Figures 10, 11 and 13 courtesey of [19]

4 Implementation

For my implementation I will be using the python programming language, we will look and explain the goings on of the 3 types of ANN's, one from first principles being just an SLP to see a practice of basic theory.

The next two involve implementation of a high level API in the for of Keras which has been implemented into TensorFlow since version 2.0 released. It is also possible to write these networks from first principles. But in the interest of time we will use these powerful python modules to really show how powerful and efficient a neural network can be.

4.1 SLP in Numpy

In this section I would like to show you a network that is capable of drawing a decision boundary between two distinct classes. We will implement previous bits of mathematical theory we have talked about. Those being the sigmoid activation function, MSE loss function and our backpropagation algorithm. We will also touch briefly on initialisation as well.

The first step is to first get or in our case just create some example data. Here is what our example data will look like:

```

%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np

#Data is a list of lists in the format of x coordinate,y coordinate,class

data = [[3.2, 1.5, 1],
        [2, 1, 0],
        [4.1, 1.5, 1],
        [2.75, 1, 0],
        [3.56, .5, 1],
        [2, .5, 0],
        [5.5, 1, 1],
        [1, 1, 0],
        [4, 1, 1],
        [0, 0, 0],
        [1, 0.5, 1]]

mystery_point = [4.5, 1]

```

Figure 14: Example data structure

As we can see here we have a lists of lists situation with our data, we will plot the first two element of each sub list to provide our x and y coordinates and we will colour them according to the value of the class parameter in the third entry in the sub list, where 0 is a yellow point and 1 is a magenta point. we can see this in the next figure.

```

# scatter plot them
def data_grid():
    plt.grid()

    for i in range(len(data)):
        c = 'm'
        if data[i][2] == 0:
            c = 'y'
        plt.scatter([data[i][0]], [data[i][1]], c=c)

    plt.scatter([mystery_point[0]], [mystery_point[1]], c='k')

plt.title("Scatter plot")
plt.xlabel("x")
plt.ylabel("y")
data_grid()

```

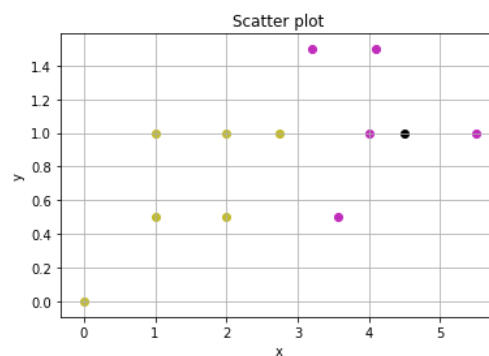


Figure 15: Scatter plot of our data

What we are doing here is creating a function to return a plot of our data. We do this by starting a loop that goes over all entries in our dataset, then we set the variable `c` as our colour parameter, by default this has a string value of "m" which corresponds to magenta. Next we say if the 3rd entry of the `i`-th element is equal to zero set `c` to "y" which stands for yellow. Then we can start plotting these points. Next we have a statement outside of the loop which say to plot the point of our mystery point. The goal of this is to remember we want to train our network to decide if our point is magenta or if it is yellow just given by its `x` and `y` coordinates.

Next we should define our activation function. For this example we will just use the sigmoid function. Then we can start to actually implement the training process for our network. I will break all of the elements down line by line so we can really understand what is going on here.

```
def sigmoid(x):
    return 1/(1+np.exp(-x))

def sigmoid_dx(x):
    return sigmoid(x) * (1-sigmoid(x))

def train():
    #random init of weights
    w1 = np.random.randn()
    w2 = np.random.randn()
    b = np.random.randn()

    iterations = 20000
    learning_rate = 0.1
    loss_values = [] # keep loss values during training, to see if they go down
```

Figure 16: Activation function and initialisation.

The first two function definitions are just that of the sigmoid function and its derivative. We will use these later on in our training process. In the next cell we start by defining our function to train our network, hence the name `train`. We can use information from our section on initialisation to help us remember that if we initialise our network with inappropriate values, meaning very large or extremely negative values. That our network will not be able to train or learn due to the gradients vanishing completely. We initialise our values for our weights and our bias node from the standard normal distribution that being $\mathcal{N}(\mu, \sigma^2)$, $\mu = 0, \sigma = 1$. In drawing random numbers in this fashion we can be more certain that our initial activation outputs will be dead and not output 0. Let's move on to the next step of our code.

```
iterations = 20000
learning_rate = 0.1
loss_values = [] # keep loss values during training, to see if they go down

for i in range(iterations):
    # get a random point
    ri = np.random.randint(len(data))
    point = data[ri]

    z = point[0] * w1 + point[1] * w2 + b
    pred = sigmoid(z) # predicted value

    target = point[2] # actual value of label
```

Figure 17: Defining variables, activation and loss

Here we have defined two variables that being the number of iterations and the learning rate. And we have declared an empty array to store our loss values in which we will later use for plotting our loss curve. Next we define our for loop. The purpose of this for loop is it contains our entire backpropagation algorithm and therefore has a iteration value equal to the number of iterations

we want to perform our algorithm for. Inside the beginning of our for loop we have the variable "ri" which stands for random integer. This random integer will be used to at random elect a point from our dataset which we can then feed the x and y coordinates into our backpropagation algorithm. We see just below the variable "z" this will be our input into our activation function. It is defined in the usual way, that is to say $\sum_{i=1}^n x_i w_i + b$ and in our case our network will have two input nodes for the x and y coordinates so we have to grab them as their array index from our randomly selected point which would be the zero'th and first index. Our predicted value is just the variable z used as the input for our sigmoid activation function which we will compare with the real label which is defines as "target".

Next we want to define and calculate the loss of our network.

```
# MSE for point
loss = 0.5*np.square(pred - target)

# Append the loss over all data points every 1000 iterations
if i % 100 == 0:
    c = 0
    for j in range(len(data)):
        p = data[j]
        p_pred = sigmoid(w1 * p[0] + w2 * p[1] + b)
        c += 0.5*np.square(p_pred - p[2])
    loss_values.append(c)
```

Figure 18: Loss calculation and appending

We have a conditional if statement here which has a job of calculating and appending our loss value to our empty array, where the appending happens as the temporary variable c is defined as zero to start and then the loss for this given iteration and is then re-assigned its value and then later appended.

Next is where the real process starts in so far as we have really only calculated a few numbers so far. But now we want to start back-propagating our loss values.

```

dloss_dpred = (pred - target) # differentiate 0.5 * (y_p - y_a)^2 -> y_p - y_a
dpred_dz = sigmoid_dx(z)

dz_dw1 = point[0]
dz_dw2 = point[1]
dz_db = 1

dloss_dz = dloss_dpred * dpred_dz

dloss_dw1 = dloss_dz * dz_dw1
dloss_dw2 = dloss_dz * dz_dw2
dloss_db = dloss_dz * dz_db

w1 = w1 - learning_rate * dloss_dw1
w2 = w2 - learning_rate * dloss_dw2
b = b - learning_rate * dloss_db

return loss_values, w1, w2, b

```

Figure 19: Back-propagation

We can see our implementation of our back propagation algorithm here in the code. We have all of our partial derivatives set out which we then are going to put into our variable updates that we can see at the bottom near the return statement. We are going to return our loss values array along with our weights and biases for inspection and visualisation. Speaking of which let's get on to plotting our loss curve and inspecting the numeric values we have.



Figure 20: Loss of network and values of parameters

Let's start with the numerical outputs, first we have 4 print statements that output the values of our networks weights and bias which in this particular case are 2.714, 0.218 and -8.434 for w_1 , w_2 and b respectively. We then can see our

loss function reaches a minimum loss at the 196th index of our loss values array with a value of 0.135. Next we can see a plot of our loss curve. This is an ideal loss curve as we start off very high but quickly start approaching zero with some minimal fluctuations along the way. However this network can suffer from the vanishing gradients problem and may be in a situation where it might not converge.

Next up let us draw our decision plane.

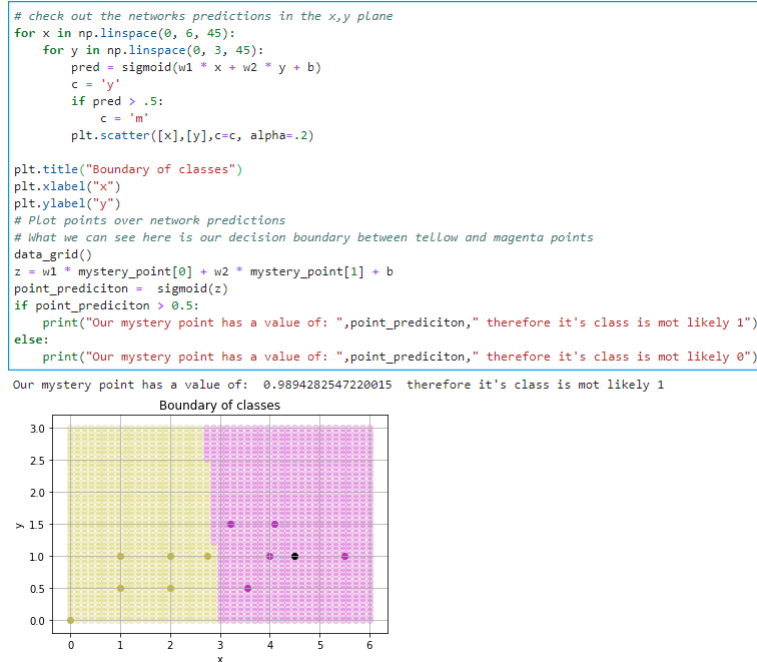


Figure 21: Descision Plane

This for loop is really only doing one thing which is plotting yellow points from 0 to 6 on the x axis and 0 to 3 on the y axis, that is unless the prediction is greater than 0.5 then it will plot a magenta point. then we plot our data_grid on top of our grid and we can visualise where our guess is classified. As we can see here. it has been classified as a magenta type point and therefore has a class of 1. We can also see this as we can calculate the output and see that in this case our mystery point has a class value of 1 and therefore is a magenta point.

4.2 MLP in TensorFlow/Keras

For this implementation I have opted to use the MNIST fashion dataset which is a slightly newer version of the older MNIST dataset which is of handwritten digits. We will be using TensorFlow version 2.1.0 which has implemented the Keras sequential model. This makes building a neural network much easier by allowing for easier composition of complex architectures and use of more complex loss, activation functions and optimisation algorithms to perform harder

classification tasks.

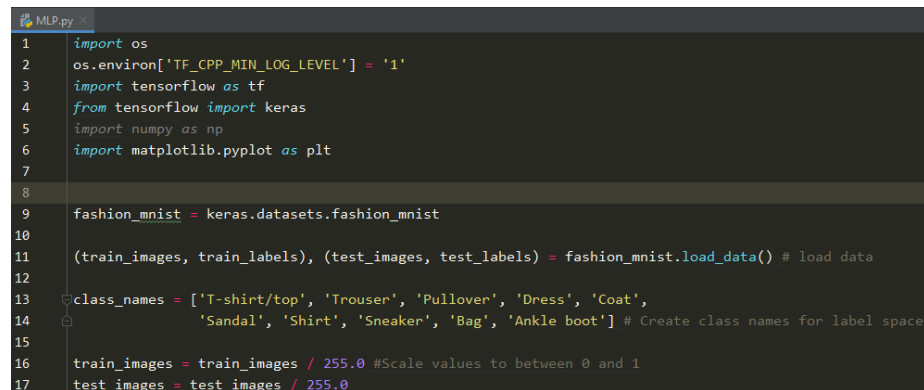
What we are going to do is build a network that can classify each of the elements of this dataset into their correct labels. This dataset has 60000 training images and 10000 for validation. Each image is a 28x28 pixel image where each image has a grey scale value from 0 to 255. Only a minor amount of preparation is needed in order to begin the training process. But just before we start we should note that we should not the initial structure of our network. Our structure will consist of an input layer of 784 nodes meaning 1 node for each pixel, 128 nodes for our intermediate layer, this will be sued for our feature maps. And finally 10 outputs nodes which output a value between 0 and 1 which correspond to probabilities of an object for that given item of clothing.

Here is an example of our dataset:



Figure 22: MNIST Fashion.

Now, that we know what kind of data we are dealing with we can begin to start writing code to prepare our data and the subsequently write code for our network architecture. You will notice a substantial decrease in the amount of lines needed to write a much more powerful network than we did before for our SLP network.



```
1 import os
2 os.environ['TF_CPP_MIN_LOG_LEVEL'] = '1'
3 import tensorflow as tf
4 from tensorflow import keras
5 import numpy as np
6 import matplotlib.pyplot as plt
7
8
9 fashion_mnist = keras.datasets.fashion_mnist
10
11 (train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data() # load data
12
13 class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
14               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot'] # Create class names for label space
15
16 train_images = train_images / 255.0 #Scale values to between 0 and 1
17 test_images = test_images / 255.0
```

Figure 23: Imports and Data Preparation

Let's break down line by line what is happening here. What we have on the first two lines of import statements is just to stop tensor flow from filling our console with GPU detection statements as it is quite annoying to see a wall of red text when running the program. It should be noted that I am using the GPU version of TensorFlow to reduce the workload of my CPU whilst potentially improving performance. From the next couple of lines we just import our machine learning libraries of choice and a couple of helper libraries for plotting etc. Next on line 9 we declare a variable which will be our MNIST fashion dataset. On line 11 we define the first tuple containing variables where the training image is the image itself and the training label will be the associated label with this given image. For example the first entry will be the image of an ankle boot with a label value of 9. The same thing is then done for our validation set which will be called our test images with their corresponding test labels. Next on line 13 we just give these numerical values of our class labels names. Line 16 and 17 are our actual preparatory steps. We can't use values from 0 to 255. The reason being is that if we pass in values to our activation function we will have such a huge range of activation values that our weights will change too often and therefore will never converge or behave chaotically due to a non linear function exhibiting vanishing gradients or a ReLU type function exploding the outputs.

Next we will build our model:

```
19  @#This is the shape of our model.
20  # An input layer that is 784 nodes wide
21  # Next a 128 node wide layer for our feature maps
22  @#Then we have 10 nodes in our label space for catgorisation
23  @model = keras.Sequential([
24      keras.layers.Flatten(input_shape=(28, 28)),
25      keras.layers.Dense(128, activation='relu'),
26      keras.layers.Dense(10)
27  ])
28
29  model.compile(optimizer='adam',
30               loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
31               metrics=['accuracy'])
32
33  model.summary()
34
35  history = model.fit(train_images, train_labels, verbose=2, epochs=10, validation_data=(test_images, test_labels))
36
```

Figure 24: Model building and metric outputs

Here we can see the power of the Keras sequential model. There are various ways to construct architectures with the sequential model provided by Keras but I personally find this way by far the neatest as you can actually visualise the layers in the model just from the code alone. We can define our model using this sequential model and inside this sequential model we start with a flatten layer.

A flatten layer is not explicitly the same as just a regular old dense layer that is 784 neurons big. It takes our 2 dimensional array of image pixel values ranging from 0 to 1 and flattens this to a 1 dimensional array. This can also be done before hand by reshaping the data to an array but this is just a shortcut to achieve the same thing. Next up we have our dense layer for our feature maps. A dense layer just means densely connected meaning each neuron is has an input and output to every neuron available to it in the previous and next layer. The reason for this intermediate layer is for as I said feature maps. What that means is the network will look at certain parts of the image to help it predict the outcome label correctly. If we could look inside one of these neurons it may look like part of a dress or it could look like nothing intelligible to us. But the network has taken on values which it thinks are important.

The value of the number of neurons does not have much significance however. Choosing the number of neurons in a given intermediate layer is a bit of trial and error and a slight bit of intuition. But as a general rule of thumb should be about less than half of the input layer. I have opted to use a ReLU function here to aid in speeding up convergence, however a sigmoidal function can also work here. Next we output to another dense layer of just 10 neurons. This is our label space, named as such because these are the available labels to us in our data going from 0 to 9. You may notice there is no activation function for our output value. How do we obtain the values of the output then? We can actually get these from the model.compile statement which we will now talk about.

In this model.compile statement we can choose our learning rate optimiser, which I personally have opted to use the ADAM optimiser as it yielded the best end results for accuracy. The next argument is how we obtain the loss values

from our label space neurons. We use Keras's "SparseCategoricalCrossentropy" loss function, which expects integer values for labels, which is what we have. It is otherwise the same exact thing as categorical cross entropy. We can see that we have a value of "from logits = true". This is saying that the outputs from our neurons is the logistic function but for a multi class label space which means it is the softmax function. Model.summary() just returns the structure of the model but in a nice format in the console. Next we have the fitting process which is on line 35. This is where the training process starts, we feed our network our 60000 training images and validate each epoch with our test data of 10000 images at each epoch.

Now, we can begin plotting our loss and accuracy of our data:

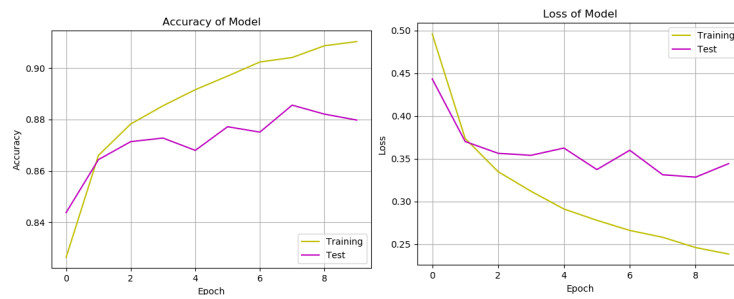


Figure 25: Accuracy and Loss of network

What we can see here from the accuracy and loss graphs is that our network is slightly over-fitting our data. We can see this as the accuracy is a significant amount higher than our validation data test result. The same can also be said for our loss graph where we see in both graphs the deviation starts at around the third epoch and by the fourth and fifth epoch the deviation is rather significant.

What we can conclude from this model is that we have been able to correctly predict the label of any given image to a reasonable degree $\approx 89\%$ and while our model does over fit our data, it isn't to such a degree where it would be considered a major problem.

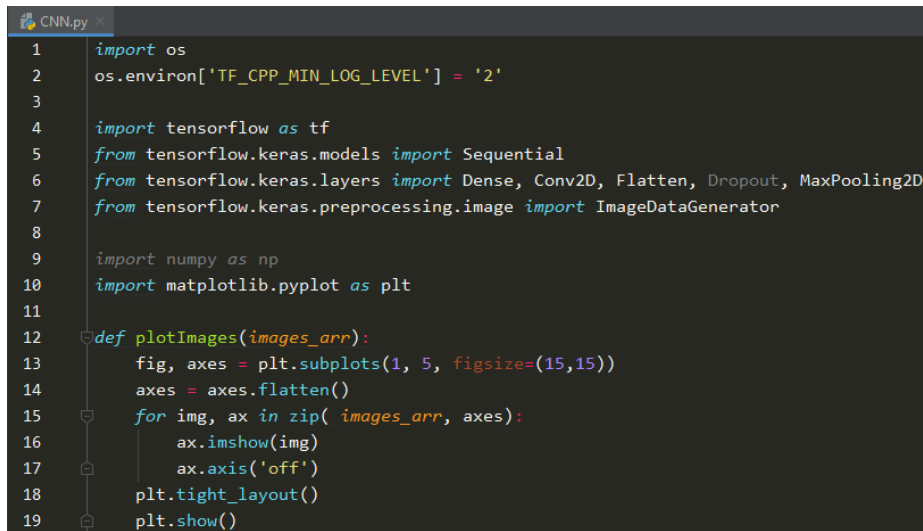
4.3 CNN in TF/Keras

Now, we are going to get slightly more advanced in this section by replacing our MLP model with a convolutional architecture. We will still be using the same method of constructing our model. However our data requires slightly more preparation than our data before. This is because we will be using a new dataset, that being a trimmed down version of the cats vs dogs dataset from Kaggle and provided by Microsoft's research team[11].

The reason I want to use this dataset as opposed to the MNIST fashion dataset is because it isn't really necessary to use a CNN for such small images that are 28x28 and have nothing else in the background of the image. As explained in [22] by one Zeeshan Zia he explains very well that a reason for this is because of spatial invariance. As he correctly writes: "The reason for choosing this special structure, is to exploit spatial or temporal invariance in recognition. For instance, a "dog" or a "car" may appear anywhere in the image. If we were to learn independent weights at each spatial or temporal location, it would take orders of magnitude more training data to train such an MLP. In over-simplified terms, for an MLP which didn't repeat weights across space, the group of neurons receiving inputs from the lower left corner of the image will have to learn to represent "dog" independently from the group of neurons connected to upper left corner, and correspondingly we will need enough images of dogs such that the network had seen several examples of dogs at each possible image location separately."

What he is saying here is the fundamental difference of the MLP and CNN structures. And further to my reason of why it is not appropriate to use an MLP for classifying the images in this dataset as it really would take orders of magnitude longer for the network to correctly classify these images as not all of the dogs are in the same location and thus it would be extremely difficult for the network to tell what pixels truly represent the dog in the case of multiple dog images.

Now, we can get started with our code. We will start with importing our required modules and defining a function to show examples of our data:



```

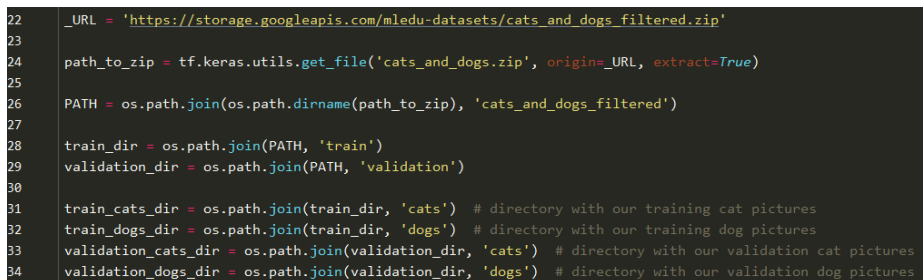
1  import os
2  os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
3
4  import tensorflow as tf
5  from tensorflow.keras.models import Sequential
6  from tensorflow.keras.layers import Dense, Conv2D, Flatten, Dropout, MaxPooling2D
7  from tensorflow.keras.preprocessing.image import ImageDataGenerator
8
9  import numpy as np
10 import matplotlib.pyplot as plt
11
12 def plotImages(images_arr):
13     fig, axes = plt.subplots(1, 5, figsize=(15,15))
14     axes = axes.flatten()
15     for img, ax in zip( images_arr, axes):
16         ax.imshow(img)
17         ax.axis('off')
18     plt.tight_layout()
19     plt.show()

```

Figure 26: Imports

In the same way as last time we have the same import of the `os` module followed by a line which stops the annoying log messages. Unlike last time we will use the `os` module for more than just log message suppression. We will use it for create a file path that we can read our images from. The rest of the modules are the imports for the required parts of TensorFlow/Keras that we need to build our model and prepare our data. And as per usual we have `numpy` and `matplotlib` in case we need them. This time we have a function statement called `plotImages`. This function reads an array of images and plots them in a 1x5 subplot. This will help us see what our data look like.

We can start by downloading our dataset and creating the file paths necessary to read from so we can prepare our data:



```

22 _URL = 'https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip'
23
24 path_to_zip = tf.keras.utils.get_file('cats_and_dogs.zip', origin=_URL, extract=True)
25
26 PATH = os.path.join(os.path.dirname(path_to_zip), 'cats_and_dogs_filtered')
27
28 train_dir = os.path.join(PATH, 'train')
29 validation_dir = os.path.join(PATH, 'validation')
30
31 train_cats_dir = os.path.join(train_dir, 'cats') # directory with our training cat pictures
32 train_dogs_dir = os.path.join(train_dir, 'dogs') # directory with our training dog pictures
33 validation_cats_dir = os.path.join(validation_dir, 'cats') # directory with our validation cat pictures
34 validation_dogs_dir = os.path.join(validation_dir, 'dogs') # directory with our validation dog pictures

```

Figure 27: File Path Creation

What we have here here on line 22 is the statement where we download our dataset. The next lines, 24 and 26 are where we extract our zip file and set our file path. On lines 28 and 29 we have our directory for our training data and validation data. We can then further separate this by the labels `cats` and `dogs` in lines 31 through 34.

Next we will set up some vector for later that we will use in our fitting stage.

```
36 #Total amount of training and validation images
37 num_cats_tr = len(os.listdir(train_cats_dir))
38 num_dogs_tr = len(os.listdir(train_dogs_dir))
39
40 num_cats_val = len(os.listdir(validation_cats_dir))
41 num_dogs_val = len(os.listdir(validation_dogs_dir))
42
43 total_train = num_cats_tr + num_dogs_tr
44 total_val = num_cats_val + num_dogs_val
```

Figure 28: Step Vectors

Here we just define some simple vectors. We have 2000 training images and 1000 validation images. The reason for defining these is we will be using these values for our epoch steps and our validation steps later on.

```
46 #Setting global variables so we dont have to keep changing them in functions
47 batch_size = 128
48 epochs = 15
49 IMG_HEIGHT = 150
50 IMG_WIDTH = 150
```

Figure 29: Global Variables

We have here some global variables for our input shape, the number of epochs as well as our batch size. Next we will go over how to read our images into our network.

```
52 #Rescaling tensor values to values between 0 and 1
53 train_image_generator = ImageDataGenerator(rescale=1./255)
54 validation_image_generator = ImageDataGenerator(rescale=1./255)
55
56 train_data_gen = train_image_generator.flow_from_directory(batch_size=batch_size,
57                                                         directory=train_dir,
58                                                         shuffle=True,
59                                                         target_size=(IMG_HEIGHT, IMG_WIDTH),
60                                                         class_mode='binary')
61
62 val_data_gen = validation_image_generator.flow_from_directory(batch_size=batch_size,
63                                                            directory=validation_dir,
64                                                            target_size=(IMG_HEIGHT, IMG_WIDTH),
65                                                            class_mode='binary')
```

Figure 30: Iage reading and tesnor transformation

What we have here are the crucial lines of code that will allow us to read images from our disk, decode the images and convert the format of the image to a grid shape from their RGB content. Convert these tensors into floating point tensors and finally as before we will reshape our data to values between 0 and 1. But let's go line by line now to understand just what's going on.

The lines 53 and 54 are the lines responsible for reading the images from our disk and converting these into batches of tensor. Then on lines 56 to 62 do the same thing as each other but on the training set and validation set. What these class of objects do, specifically the "flow_from_directory" class is responsible for re scaling and re sizing to our images. Of course we are specifying the class mode as binary as we only have a label space of 2 outcomes either it is a picture of a cat or a picture of a dog. Next we are going to plot our data using the plotImages function from earlier.



Figure 31: Data plotting

This is just a little bit of code to help us inspect our data. But now we can move on to the far more substantial portion of our code where we begin to actually compile and fit our model.

```

72 model = Sequential([
73     Conv2D(16, 3, padding='same', activation='relu', input_shape=(IMG_HEIGHT, IMG_WIDTH, 3), dilation_rate=1),
74     MaxPooling2D(),
75     Conv2D(32, 3, padding='same', activation='relu'),
76     MaxPooling2D(),
77     Conv2D(64, 3, padding='same', activation='relu'),
78     MaxPooling2D(),
79     Flatten(),
80     Dense(512, activation='relu'),
81     Dense(1)
82 ])

```

Figure 32: CNN Architecture

This will be the structure of our convolutional model, as per usual it consists of 2D convolutional layers followed by max pooling layers to reduce the size of our data going through our network. I have chosen to use a dilation rate of 1 for the initial input layer as it yielded slightly better results. With each pooling layer we increase the number of channels by a factor of 2 each time. We start with just 16 channels but eventually we end up at 64 channels. Typically the deeper you go into a network you reduce the dimension by a factor of 2 whilst increasing the number of channels by a factor of 2. However there is a slight limit to this as reducing the dimensions to far can lead to nothing of use being learned. After our final pooling layer we flatten the dimension of the tensor to a straight 1 dimensional array then we create feature maps from this flattened layer as a densely connected layer. Then we output to just a single node that will, the same as before output a soft max in the compile section. The reason for just 1 node is because we are doing binary classification here as opposed to multi class classification. It is also worth noting we are using the ReLU function here as we have a deeper network from last time and ReLU activation functions speed up the learning process in deeper networks.

Next let's summarise compile and create a history of our model for plotting

purposes.

```
84 model.compile(optimizer='adam',
85               loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
86               metrics=['accuracy'])
87
88 model.summary()
89
90 history = model.fit(
91     train_data_gen,
92     steps_per_epoch=total_train // batch_size,
93     epochs=epochs,
94     validation_data=val_data_gen,
95     validation_steps=total_val // batch_size,
96     verbose=2
97 )
```

Figure 33: Compilation and history of the CNN

In line 84 we have a similar thing going on here as to before with our MLP where we have our compile statement. The main difference being that we are using binary cross entropy now instead of the multi class version. One key difference is we have set a training and validation schedule for our model. This means every epoch we will train for 15 steps and validate for 7 steps this is shown by lines 92 and 95 where we perform integer division on our numbers for training images and testing images with our batch size.

The next step is to output our metrics of loss and accuracy for training and testing set and plot them.

```
99 acc = history.history['accuracy']
100 val_acc = history.history['val_accuracy']
101
102 loss=history.history['loss']
103 val_loss=history.history['val_loss']
104
105 epochs_range = range(epochs)
106
107 plt.figure(figsize=(8, 8))
108 plt.subplot(1, 2, 1)
109 plt.plot(epochs_range, acc, label='Training Accuracy',color='tab:orange')
110 plt.plot(epochs_range, val_acc, label='Validation Accuracy',color='m')
111 plt.legend(loc='lower right')
112 plt.title('Training and Validation Accuracy')
113
114 plt.subplot(1, 2, 2)
115 plt.plot(epochs_range, loss, label='Training Loss',color='tab:orange')
116 plt.plot(epochs_range, val_loss, label='Validation Loss',color='m')
117 plt.legend(loc='upper right')
118 plt.title('Training and Validation Loss')
119 plt.show()
```

Figure 34: CNN plot results.

Here is code which is very similar to the plotting we did in the previous section where we will analyse our result and see if our network is any good at classifying our images. Let's take a look at the results.



Figure 35: Accuracy and Loss

What we can see here is a very interesting phenomenon. Not only are we just seeing over fitting in our model. But we are also seeing the accuracy for our test data oscillate, but as accuracy marginally increases for our test data we see the loss actually go up. What could very well be happening here is the more borderline cases are being classified correctly more frequently. However objects that are being classified incorrectly are just being classified more incorrectly. This has the end result of accuracy going up as more borderline cases are correctly classified, however the loss is increasing as incorrect classifications are getting more incorrect. this behaviour is exhibited with or without kernel dilation interestingly enough.

However there are things we can do to address these issues. We mentioned in the section on CNN's that we can actually augment and transform our data with various techniques, things like zooming stretching rotating etc. This could

help us in addressing our over fitting problem as well as potentially decrease the odd behaviour of loss and accuracy increasing. As we only want accuracy to increase inversely to loss. We will also try to come up with a new model to further reduce over fitting. Let's take a look at some new pieces of code for data augmentation and our new model.

```
#Augment data
image_gen_train = ImageDataGenerator(
    rotation_range=15,
    rescale=1./255,
    shear_range=0.1,
    zoom_range=0.2,
    horizontal_flip=True,
    width_shift_range=0.1,
    height_shift_range=0.1
)
```

Figure 36: Data Augmentation

Here we can see all of the arguments necessary to augment our data. These include zoom, rotation, image flipping, image shift and image sheering along with our regular re scaling. Now we can take a look at our new model.

```

67 model_new = Sequential([
68     Conv2D(16, 3, padding='same', activation='relu',
69         input_shape=(IMG_HEIGHT, IMG_WIDTH, 3)),
70     MaxPooling2D(),
71     Dropout(0.2),
72     Conv2D(32, 3, padding='same', activation='relu'),
73     MaxPooling2D(),
74     Conv2D(64, 3, padding='same', activation='relu'),
75     MaxPooling2D(),
76     Dropout(0.2),
77     Flatten(),
78     Dense(512, activation='relu'),
79     Dense(1)
80 ])

```

Figure 37: New CNN model

We have a new CNN model that looks very similar but this time we have included a new layer called a dropout layer. What a dropout layer does is it randomly "kills" a specified proportion of neurons in order to reduce the effect of over fitting. These nodes being "killed" means they will not participate in the feed forward algorithm. They will not output any value at all i.e null.

Now, we should take a look at the result this has had on our model.

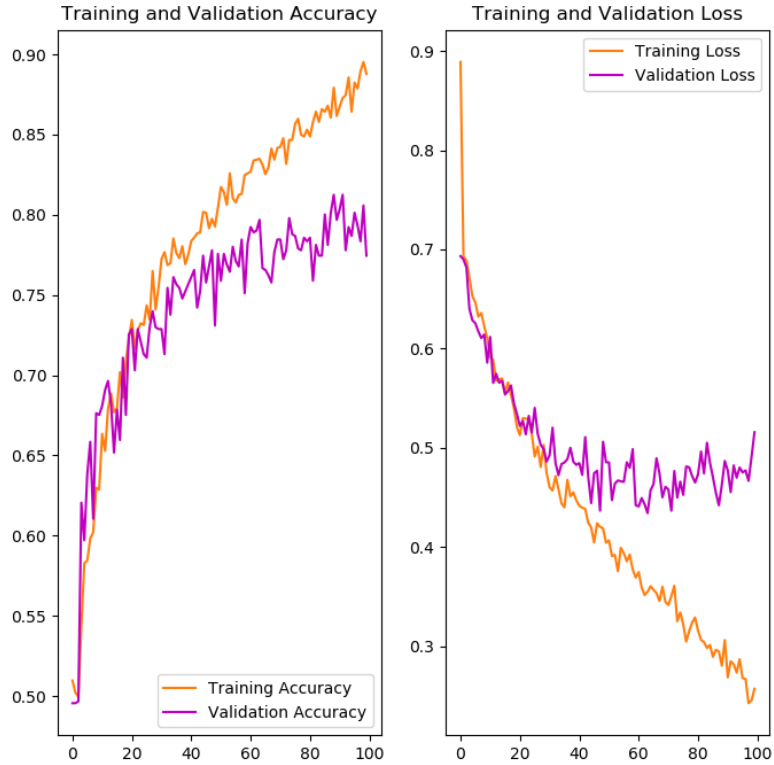


Figure 38: Augmented data CNN results.

We can see convergence on the network was fairly slow as we have applied a lot of transformations/augmentations to our images, so we have to run our model a bit longer to come up with a decent accuracy value. Our previous model will not progress past the seventy percent mark as the loss value will continue to be above 1 or 2. We can see that it will still over fit our data but the point when this starts to become apparent is much much later. This test was ran for 100 epochs at about 14 to 15 seconds each and was able to achieve a much higher accuracy than the model before. We have been able to achieve a model with $\approx 77\%$ accuracy which is a significant increase from before and we have curtailed at least a decent amount of over fitting that we inherited from our previous example.

5 Conclusion

In our conclusion of results I would like to touch on each implementation separately as the SLP and MLP networks operate on a completely different principle from our convolutional network.

5.1 Conclusion of Results: SLP & MLP

What we have seen in our SLP results is we were able to demonstrate the ability of our SLP to draw a decision plane between a linearly separable data set. We were able to implement parts of our discussed theory into code also. We implemented the sigmoid activation function along with our mean squared error loss function. I would consider the results of this simple network reliable as it will work for any linearly separable dataset as an SLP will not be able to classify non-linear datasets as proven in our XOR problem section.

I believe the results of our MLP network were fairly good, albeit the data was slightly over fitted. The test results were very promising at $\approx 90\%$. I enjoyed working with the Keras sequential model and implementing more complex architectures to perform a more complex classification task involving multi class classification. We were able to break down line-by-line the working of the code to explain what was going on. Overall I find these results to be of a good quality and the network need little improvements to achieve higher accuracy's.

5.2 Conclusion of Results: CNN

This is the network I was most excited to work on. It took the most time and even though the results were at best satisfactory at $\approx 77\%$ test accuracy with a maximum of just over 80%. Improvements can be made to this network however. As I mentioned at the start of the CNN section we are using a trimmed down dataset of the cats vs dogs dataset from Microsoft's research team. The full dataset would have provided better results as there are more examples to look analyse with the network. The next would be to possibly introduce some sort of learning rate schedule using the callbacks sub-module in Keras. Next we could probably use "BatchNormalisation" so that the activation functions are more normally distributed to aid in performance as well as maybe increase the number of output channels from our filters by a factor of 2 at all levels. We could also perform multi class categorisation meaning having two nodes on the output instead of just the one to help with class separation.

But in closing I would consider the investigation a success. I have achieved my goal as to understanding how this network works and be able to interpret the results.

6 Lay Summary

In this investigation we will be investigating neural networks, how they work, how to implement one in the python programming language and finally explain are results and findings. But this topic should be of interest to many people as it is at an intersection of computer science and statistics.

But first we must define what a neural network is as to continue talking any further. A neural network is a compute program that can take data type inputs, be it in the form of images text or just a simple matrix of numbers. It takes this information and learns patterns from our data we provided. We then test new data inside our network to see how it performs to images it has not seen. This is to validate that our network can actually classify data. Neural networks learns via the process of backpropagation. Backpropagation is an algorithm that relies on just simple calculus. We will look at this in detail and even derive backpropagation in a line by line fashion so we can get a better understanding of how this algorithm works.

In this work we investigate the theory that governs neural networks, that being all the components that are involved in this backpropagation algorithm. The theory sections includes explanations of activation functions, these are functions which are the output of an individual neuron in our network and investigate what properties we might want in an activation function as to help define the behaviour of our network. We will also talk about and investigate the theory of loss functions which is how we can measure how well our network is classifying our data.

There will also be more advanced theory on the types of networks we will program to solve our classification problem, those being the single layer perceptron network (SLP), a multi layer perceptron network (MLP) and finally a convolutional neural network (CNN). We will implement them in order of complexity and explain why the more complex networks can perform harder classification tasks. This will further our understanding of why networks like the SLP and MLP have fallen slightly out of favour for image classification and why CNN's are much more effective in their role as a classifier.

Now that we know what a neural network is we can start to introduce our problem that we want to solve. We want to solve classification problems. These problems can involve a dataset containing objects that may have 2 or more categories assigned to them. In our investigation we investigate 3 different datasets. The first is a simple array of arrays that contain both x and y coordinates as well as a class parameter. We will then classify a mystery point i.e a point that we give to our network that has only an x and y coordinate but no class. Our goal is for our network to predict this point.

Next, we implement a network to classify the MNIST fashion dataset. This dataset is comprised of 10 different items of clothing and we will build a network that is able to classify these items of clothing correctly. And then finally we implemented a CNN for classifying a more complex dataset. This dataset is called "Cats vs Dogs" and contains RGB images of cats and dogs. We have

built a network to classify these images based on the output of our network.

Finally, we concluded our results where we found that we were able to classify our data for our SLP, MLP and CNN respectively. We found zero issues with our classification of our simple SLP network. But both our MLP and CNN exhibited a phenomenon known as "over fitting". This is where our network is classifying images it has seen already to a better degree to images that it has not. In the case of our MLP this was minimal. In our CNN however we saw a larger over fitting effect. We then sought out a way to rectify this issue with a relative amount of success, improving our accuracy by $\approx 10\%$ in the process. We also made a list of recommendations for potential improvements that could be made to our network in order to achieve a better result.

References

- [1] Shack. Ai. *The Sobel and Laplacian Edge Detectors*. <https://www.aishack.in/>. Answer on. URL: <https://www.aishack.in/tutorials/sobel-laplacian-edge-detectors/>.
- [2] Bapu-Ahire.Jayesh. *The XOR Problem in Neural Networks*. <https://medium.com/>. Graphs for XOR. URL: <https://medium.com/@jayeshbahire/the-xor-problem-in-neural-networks-50006411840b/>.
- [3] Claudia Beleites, Reiner Salzer, and Valter Sergo. “Validation of soft classification models using partial class memberships: An extended concept of sensitivity & co. applied to grading of astrocytoma tissues”. In: *Chemo-metrics and Intelligent Laboratory Systems* 122 (2013), pp. 12–22.
- [4] Toni. Bellamo. *Non-monotonic activation funcitons in neural networks*. [quora.com](https://www.quora.com/Can-non-monotonic-activation-function-neural-networks-be-trained-using-the-same-backpropagation-algorithm-as-monotonic-activation-function-neural-networks). URL: <https://www.quora.com/Can-non-monotonic-activation-function-neural-networks-be-trained-using-the-same-backpropagation-algorithm-as-monotonic-activation-function-neural-networks>.
- [5] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.
- [6] Beleites. C. *Can the mean squared error be used for classification?*. <https://stats.stackexchange.com/>. Answer on. URL: <https://stats.stackexchange.com/questions/46413/can-the-mean-squared-error-be-used-for-classification>.
- [7] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.
- [8] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.
- [9] G Huang et al. “Snapshot ensembles: Train 1, get M for free. arXiv 2017”. In: *arXiv preprint arXiv:1704.00109* ().
- [10] Satoshi Iizuka and Edgar Simo-Serra. “DeepRemaster: Temporal Source-Reference Attention Networks for Comprehensive Video Enhancement”. In: *ACM Trans. Graph.* 38.6 (Nov. 2019). ISSN: 0730-0301. DOI: 10.1145/3355089.3356570. URL: <https://doi.org/10.1145/3355089.3356570>.
- [11] Kaggle. *Dogs vs. Cats*. <https://www.kaggle.com/>. Answer on. URL: <https://www.kaggle.com/c/dogs-vs-cats/overview/>.
- [12] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).
- [13] Yann A LeCun et al. “Efficient backprop”. In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.
- [14] Parmar. Ravindra. *Common Loss functions in machine learning*. <https://towardsdatascience.com/>. Section on hinge loss. URL: <https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23>.

- [15] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6 (1958), p. 386.
- [16] Verma. S. *1d and 3d convnets*. <https://towardsdatascience.com/>. Answer on. URL: <https://towardsdatascience.com/understanding-1d-and-3d-convolution-neural-network-keras-9d8f76e29610>.
- [17] Neil. Slater. *Multi class cross entropy*. datascience.stackexchange.com. Section on cross entropy. URL: <https://datascience.stackexchange.com/questions/20296/cross-entropy-loss-explanation>.
- [18] Christopher. Stover. *Monotonic Function*. From *MathWorld—A Wolfram Web Resource*. Last visited on 13/4/2012. URL: <http://mathworld.wolfram.com/MonotonicFunction.html>.
- [19] Thomas.Christopher. *An introduction to Convolutional Neural Networks*. <https://towardsdatascience.com/>. Graphs for XOR. URL: <https://towardsdatascience.com/an-introduction-to-convolutional-neural-networks-eb0b60b58fd7/>.
- [20] Rohan. Varma. *Picking Loss Functions - A comparison between MSE, Cross Entropy, and Hinge Loss*. <https://rohanvarma.me/Loss-Functions/>. Section on hinge loss. URL: <https://rohanvarma.me/Loss-Functions/>.
- [21] Zhengyang Wang and Shuiwang Ji. “Smoothed dilated convolutions for improved dense prediction”. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2018, pp. 2486–2495.
- [22] Zia.Zeeshan. *What is the difference between a convolutional neural network and a multilayer perceptron?* <https://www.quora.com/>. Top explanation by Zeeshan Zia. URL: <https://www.quora.com/What-is-the-difference-between-a-convolutional-neural-network-and-a-multilayer-perceptron/>.