Modelling Log Loss over Training Iterations of Multi-Class Classification Artificial Neural Networks

Mathematics Higher Level - Internal Assessment

Section Ⅰ - Inspiration, Investigation, Introduction:

Artificial neural networks (ANNs) have been the subject of my curiosity and passion for a number of years now. For school and independent research projects I have programmed and licensed optimisation and multi-class classification ANNs for, among others, predicting temperature-stress conditions in the elderly, small vehicle autonomy and teaching assistants/aids and have been invited to local universities to speak on the topic. Recently, I founded a Code Club in my school to teach interested students the basics of ANNs[1]. By undertaking this investigation I was looking to educate my peers, in an applied sense, the mathematics of training ANNs with pre-university mathematics as well as to deepen my understanding of MCC ANNs for a career as a computer vision engineer.

This investigation aimed to achieve these goals by modelling the function relationship between *log loss* and the *training iterations* of three-layer feed-forward artificial neural networks (ANNs) with the purpose of multi-class classification (MCC). I will conduct this investigation in three sections (sections Ⅱ-Ⅳ). First, I will analyse the mathematics and properties of *forward-propagation*, *backpropagation* and *gradient descent* that make the training of three-layer neural networks possible. I will then apply this understanding by manually calculating one training iteration of a three-layer MCC ANN using the hypothetical example of a 2 by 2 pixel image classifier. Finally, by programming a three-layer handwritten digit classifier with the mathematical basis established in section Ⅲ, for 28 by 28 pixel images, I will model the association between the network's log loss over training iterations in the domain of the first 500 training iterations of the networks as it is trained on a real-world open-source dataset. Conducting the investigation, I have produced the relation of log loss ($L(x)$) and training iterations (x) of: $L(x) = (-8.02 \cdot 10^{-11})x^4 + (7.09 \cdot 10^{-8})x^3 + (-8.55 \cdot 10^{-6})x^2 + (-6.88 \cdot 10^{-3})x + 2.31,\ 0 \leq x \leq 500$.

Artificial neural networks are the cornerstone of machine learning - itself a subfield of artificial intelligence - enthralled by the quest for a 'universal learning machine' as first described by prolific British mathematician Alan Turing in the early 20th century. Feed-forward artificial neural networks are computer programs, inspired by the biological neural networks that constitute organic brains (refer to figure 1.1). They 'learn' to perform tasks by repeatedly considering examples (in supervised learning) without being pre-programmed with task-specific rules ("Artificial Intelligence").

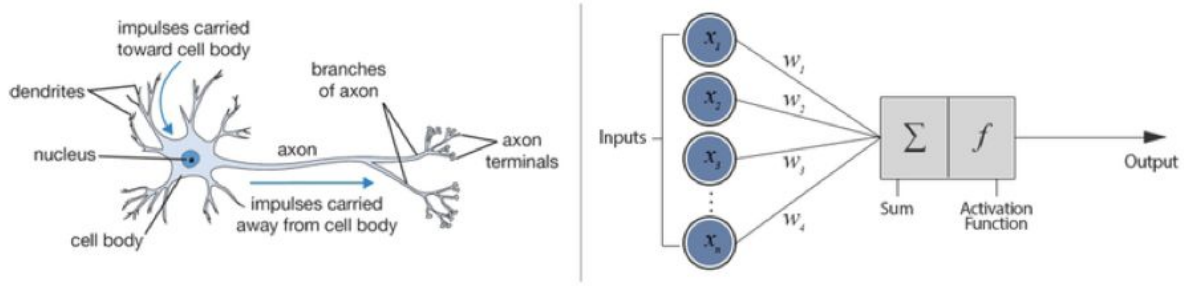

Figure 1.1: Comparison between a biological neuron and an artificial neuron – a *perceptron*
Source: https://towardsdatascience.com/from-fiction-to-reality-a-beginners-guide-to-artificial-neural-networks-d0411777571b

Classification ANNs are a form of ANN wherein the network learns to classify inputs into 'classes' (types/items) based on patterns present in data often indistinguishable by a human. An example of an application of this form of ANN is computer vision - wherein the network learns to identify the item present in an image (i.e. identifying an image of a cat distinct to an image of a dog). Multi-class classification ANNs have been increasingly prominent in this past decade with the property of being able to classify which of greater than two possible classes an image item belongs to (i.e. distinguishing a cat from a dog from a parrot). These ANNs have ever-increasing applications to society across a variety of fields including medical scan diagnoses in healthcare and autonomous vehicles in engineering.

MCC ANNs are based heavily in the mathematics of partial derivatives and matrix operations. They are often likened to universal function approximators as they mathematically achieve an optimal system for item classification by altering matrices of *weights* and *biases* between *nodes*. The log loss function is the key driving calculation of neural network training. It computes the error of the network and represents the difference between the network's current set of weight and bias matrices from the ideal matrices for classification with 100% testing accuracy. The value of log loss it computes thus reduces over the training of a network - as training accuracy increases. This relation between training iterations (cycles of forward and backpropagation) and log loss of a network is the relation whose function I wish to model.

[1] The repository of code provided to students of my Code Club consisted of a mix of internet and self-developed ANN programs linked here: https://github.com/Kumagi360/SCIC_ModellingUnit

Section Ⅱ - MCC ANN Terminology and Mathematical Background

Theory - Network Architecture:

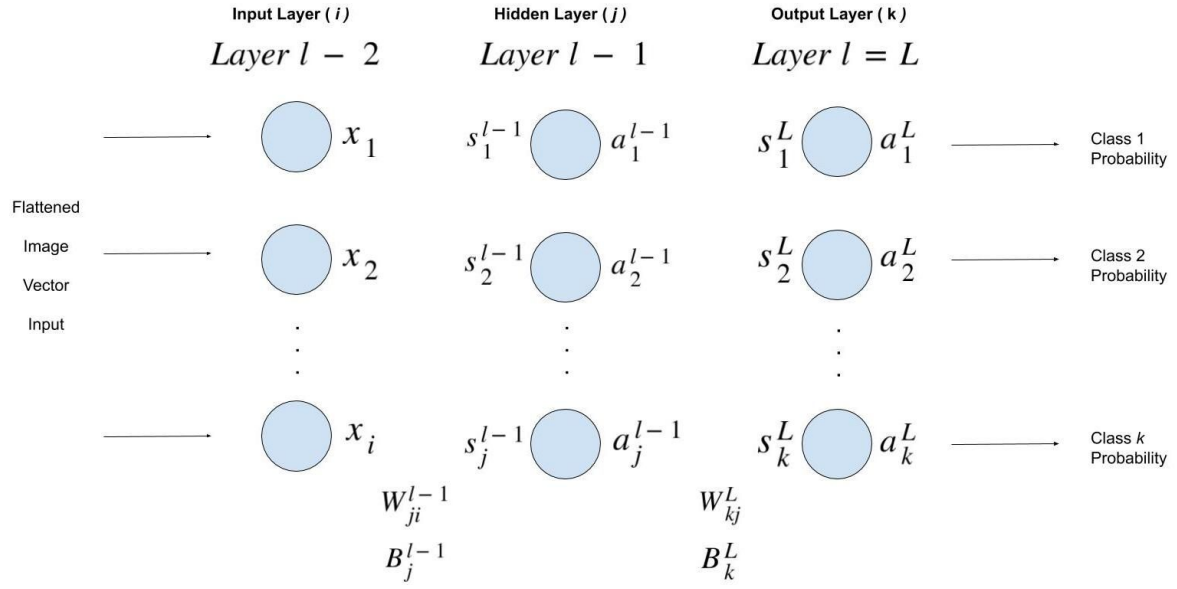

Figure 2.1: General Architecture of an MCC ANN
Source: Self-drawn on https://docs.google.com/drawings

All MCC ANN architectures consist of components that can be visually demonstrated as a network of interconnected parts, similar to the networks of neurons found in biological brains, that dictate the flow and manipulation of data from raw input to processed output of the network (refer to figure 2.1). Firstly, *layers* of nodes represent neurons that receive inputs together in a biological brain. The first layer of inputs is called the *input layer* ($Layer\ l-2$), the last layer is the *output layer* ($Layer\ l=L$) and all layers in between (in figure 2.1, only one) are called *hidden layers* ($Layer\ l-1$). In general convention, all node indexes associated with the input layer have the subscript *n*, all node indexes associated with the first hidden layer the subscript j and the nodes in the output layer have the subscript k. These subscripts are placeholders for the index of the number of nodes being referred to and can be replaced by the number of the node to specify a vector of that node. **The superscript L exclusively represents the output layer of a network, while the superscript l, as used extensively in formulae, can represent any layer in a network.** Each layer holds a *dot-product sum vector*[2] (seen as s_j^{l-1} and s_k^L) and a *post-activation vector* (seen as a_j^{l-1} and a_k^L) that contain processed data. Secondly, *weight matrices* (seen as W_{ji}^{l-1} and W_{kj}^L) connect each node in a layer to each node in the next layer. This is representative of the random strength of connections between neurons in a brain over synapses (gaps between neurons). Finally, *bias matrices* (seen as B_j^{l-1} and B_k^L) contain values, unrelated to the input image, that are used to ensure a network learns optimally[3]. It is these matrices of weights and biases that are tuned over training iterations to allow the MCC ANN to 'learn' patterns within images.

Theory - Forward Propagation:

Forward propagation refers to the forward transfer of data through the network - from the input layer inputs (x_i) to the final outputs (a_k^L). First, the input layer receives a 'flattened' image as a vector, where each row is a data representation of a pixel of the image called a *feature*. To progress to the first of the hidden layers of the network, a dot product multiplication of this input feature vector with the intermediate matrix of weights takes place and is added element-wise to the intermediate bias matrix. This, the dot product sum vector (s_k^l), is demonstrated in figure 2.2 below and acts on every node in a layer. It follows the formula of:

$$s_k^l = B_k^l + \sum_j \left(W_{kj}^l \cdot a_j^{l-1} \right)$$

Where: k represents the number of nodes in layer l
j represents the number of nodes in layer $l-1$
s_k^l represents a dot product sum vector of layer l with the number of rows as nodes in layer l
B_k^l represents a bias vector between layers $l-1$ and l with the number of rows as nodes in layer l

---

[2] A *vector* in ANNs terminology is a single-columned matrix.
[3] The purpose, impact and mathematics of bias matrices on training is complex and out of the scope of this investigation.

W^l_{kj} represents the weights matrix between layers $l-1$ and l of dimensions l by $l-1$

a^{l-1}_j represents the activation vector of layer $l-1$ with the number of rows as nodes in layer $l-1$

The dot product sum vector is passed to an activation function of the layer – wherein based on the function, each vector element has a new value and the vector is now called the post-activation vector. There are a number of commonly used activation functions including sigmoid (logistic) functions and reLU functions used to constrain the range within which the dot product sum vector numbers lay. The choice of function used at each layer is governed by the objective of the network (efficiency, maximal accuracy, etc). This investigation will make use of sigmoid activation for the input and hidden layers. In any case, the post-activation vector has the same dimensions as the dot product sum vector and acts as the output vector of the layer. This is also demonstrated in figure 2.2 below.

Activation functions follow the formula:

$$a^l_k = f\left(s^l_k\right)$$

The sigmoid activation function of the hidden layer follows the formula:

$$a^l_k = \text{sigmoid}\left(s^l_k\right) = \frac{1}{1 + e^{-\left(s^l_k\right)}}$$

Where: a^l_k represents the post-activation vector with k rows - the number of nodes in layer l

s^l_k represents the dot product sum vector to be 'activated' with k rows - the number of nodes in layer l

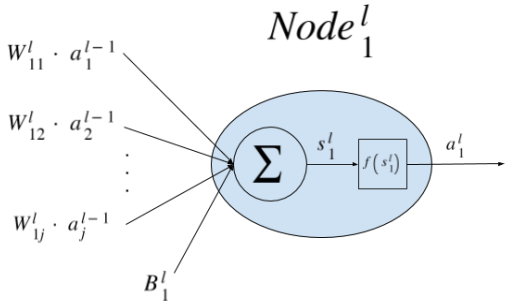

Figure 2.2: Model of the Working of a Single 'Node' (Perceptron) - $Node^l_1$
Source: Self-drawn on https://docs.google.com/drawings

This post-activation vector output then effectively acts as a new input feature vector to the next layer. This dot product and activation process repeats until the output layer - wherein each node's output value represents a possible class of the image. Traditional classification ANNs then predict the class of the original input by selecting the class of the node with the highest value. The concept of a softmax activation function , however, is that the output layer follows instead the unique softmax activation function, which normalises the inputs of the output layer such that its outputs are each between zero and one, and all outputs add up to one, instead of being arbitrarily wild. This serves the unique purpose of providing classification as a probability – where the output value of each node is at a scale such that a node returning 0.4 would have a likelihood for its class twice as high as another node returning 0.2. The softmax function will be used in this investigation to increase the ease of drawing conclusions from changes in the output data across iterations. The softmax activation function of the final layer (L) follows formula:

$$a^L_k = \text{softmax}\left(s^L_k\right) = \frac{e^{\left(s^L_k\right)}}{\sum_c \left(e^{\left(s^L_c\right)}\right)}$$

Where: a^L_k represents the final post-activation vector with k rows - the number of nodes in layer L

s^L_k represents the final dot product sum vector to be 'activated' with k rows - the number of nodes in layer L

$\sum_c \left(e^{\left(s^L_c\right)}\right)$ represents the sum of every element (c) of the s^L_k vector to the power of e

A network gets trained through the process of backpropagation in gradient descent, wherein the weight and bias matrices between each layer get updated incrementally to reduce the *loss* of the network. The loss function that a network using Softmax activation in the last layer uses is called the *log loss function* and the *log loss* it calculates and is given by the function below:

$$E = -\sum_k \left(t_k \cdot \log\left(a^L_k\right)\right)$$

Where: E stands for error (synonymous with loss) and represents the log loss of the network

t_k stands for the target value of each element (k) of the output softmax-activated, post-activation vector (a_k^L)

a_k^L represents the final post-activation vector with k rows - the number of nodes in output layer L

Subsequently, log loss is a value for the total inaccuracy of an MCC ANN and is the negative sum of the elements in the loss vector. This loss vector is the multiplication of the scalar target values (t_k) with the log (base-10) of their corresponding actual output values (a_k^L) for every element in the post-activation vector (every output node). The target values are supplied during 'supervised training', where each image being classified is 'labelled' for the network to see the correct output node values after completing forward propagation of the image - to allow it to calculate log loss. An optimal network would have a log loss value of zero - meaning it can identify the class of any unlabeled input image of a type it has been trained on with 100% accuracy and without requiring further backpropagation to train the network.

Theory - Back Propagation:

Each individual weight and bias in the weight and bias matrices of the network is then tuned in a loss-reducing direction through the process of *backpropagation*. Backpropagation entails the concept of tuning the weights heavily responsible for final log loss more than ones which don't contribute as much to loss. This is based on extensive use of the chain rule and vector math to calculate the partial derivative of the log loss to the preceding layers' weight and bias matrices. Partial derivatives in ANNs are the measure of the change the elements of these matrices induce in the next layer as a result of a small change in itself. This matrix is multiplied by the conventional *learning rate* variable scalar of 0.1 (will be elaborated in section Ⅳ) and the scalar of -1 to prepare the *updating matrix*. Backpropagation itself is the critical element of *gradient descent* - the process by which the network reduces its loss to increase its classification accuracy. Below is the formula for preparing the updating matrix for adjusting the final weight matrix W_{kj}^L. It is the obtaining of the partial derivative of weights and bias matrices with respect to final loss to obtain updating matrices that poses the mathematical challenge of ANNs and is unique for every MCC ANN architecture.

$$\Delta W_{kj}^L = -\left(\varepsilon \cdot \frac{\partial E}{\partial W_{kj}^L}\right)$$

Where: E stands for error (synonymous with loss) and represents the log loss of the network

ε stands for the learning rate variable of the network - almost always given a value of 0.1

$\frac{\partial E}{\partial W_{kj}^L}$ represents the change in loss with respect to a change in the weight matrix (W_{kj}^L)

ΔW_{kj}^L represents the updating matrix for the weight matrix (W_{kj}^L)

By recurring the forward and backpropagation process over several hundred training iterations over several hundred training examples, a network can effectively tune its weight and bias matrices to achieve the highest possible accuracy determining the class of an unlabeled image. The steps of backpropagation change according to the architecture of a network, thus to model backpropagation's impact on log loss of a three-layer MCC ANN over the extent of 500 training iterations - the mathematical steps for backpropagation of a three-layer MCC ANN with conventional properties will be investigated and programmed - from which the model can be derived.

Section Ⅲ - Example MCC ANN Training Iteration and Log Loss Reduction Mathematical Justification:

Background and Architecture:

To model log loss over training iterations of an MCC ANN, the backpropagation of a network can be mathematically demonstrated. For this investigation, with the goal of modelling the log loss over 500 training iterations of a usable real-world 28 by 28 pixel handwritten digit classifier, the working of completing a single iteration would far exceed the page limit of this investigation given (28 x 28 = 784) 784 nodes and a bias in the first layer itself - meaning a minimum dot product of 785 terms even if the hidden layer contained only a single node. Furthermore, each pixel would have a grayscale value of 1 to 256 - meaning hand-written multiplication would be utterly impractical.

Instead, a hypothetical and far simpler (however in the real world, useless) 2 by 2 pixel pattern classifier using only black (1) or white (0) pixel values will be used to demonstrate the steps of backpropagation that is implemented in similar MCC ANNs. This 2 by 2 pixel classifier should be able to distinguish between the pattern of a forward-slash, backslash and a fully blacked-out image (refer to figure 3.1 below). Arbitrarily, the **forward slash (class one)** input will be used to demonstrate this network as the labelled piece of training data.

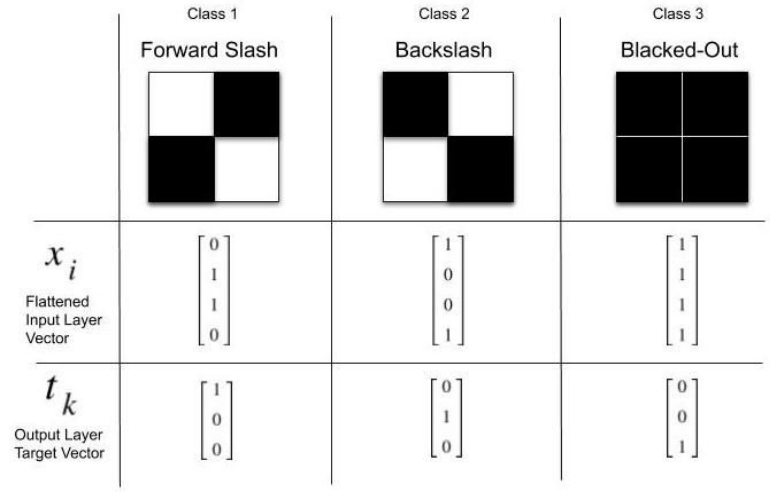

Figure 3.1: Classes of Identification with Input and Output Target Vectors
Source: Self-drawn on https://docs.google.com/drawings

While this identification could easily be hard-coded - the steps of the mathematics for training a network to distinguish these is identical to the steps for training a larger network to classify a larger and more detailed image. The difference is matrices of significantly larger dimensions result in an incredible increase in the number of matrix elements to be tuned individually - an incredible increase in the number and complexity of calculations, viable only for a computer. Hence, investigating the mathematics of the backpropagation of this network is feasible for modelling the backpropagation and subsequent log loss trend of the larger, real-world applicable net to come. Figure 3.2 below is the architecture which will constitute this MCC ANN - 4 input layer nodes (four features - given the four pixels in the input image), 2 hidden layer nodes and 3 output layer identification classes. The variables shown below will be referenced throughout the math to follow.

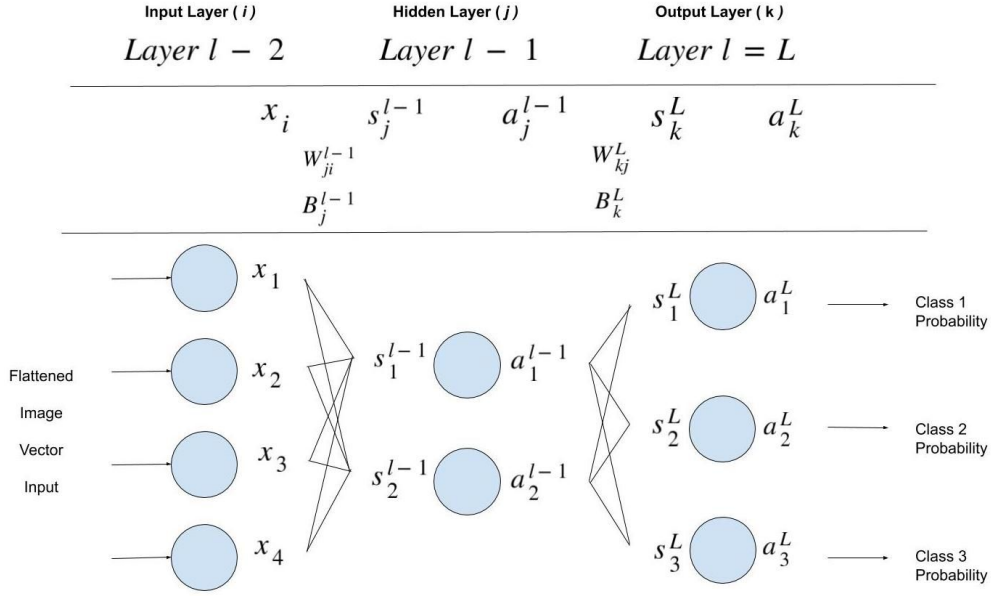

Figure 3.2: Architecture and Mathematical Notation of 2 by 2 pixel classifier MCC ANN to be used
Source: Self-drawn on https://docs.google.com/drawings

To run a training iteration of this architecture, the weight matrices must be initialised with random values and the bias matrices by values of 0 - as per general ANN convention. Below are the starting weight and bias matrices of the 4 weight and bias matrices of this network.

$$W_{ji}^{l-1} = \begin{bmatrix} 0.5 & 0.1 & 1.3 & 2.1 \\ 1.6 & 2.5 & 0.8 & 1.8 \end{bmatrix}, \quad W_{kj}^{L} = \begin{bmatrix} 0.5 & 1.9 \\ 2.2 & 0.4 \\ 1.8 & 1.2 \end{bmatrix}$$

$$B_j^{l-1} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad B_k^{L} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Training Iteration One - Forward Propagation:

The first step of forward propagation is taking the dot product of the input layer vector (x_i) and the weight matrix (W_{ji}^{l-1}) connecting the input layer ($l-2$) to the hidden layer ($l-1$) and adding the resultant vector element-wise to the corresponding bias vector (B_j^{l-1}). The result will be the dot-product sum vector of the hidden layer (s_j^{l-1}) with as many rows as nodes in the hidden layer - therefore a 2*x*1 vector.

$$s_j^{l-1} = \begin{bmatrix} 0.5 & 0.1 & 1.3 & 2.1 \\ 1.6 & 2.5 & 0.8 & 1.8 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} (0.5 \cdot 0) + (0.1 \cdot 1) + (1.3 \cdot 1) + (2.1 \cdot 0) + 0 \\ (1.6 \cdot 0) + (2.5 \cdot 1) + (0.8 \cdot 1) + (1.8 \cdot 0) + 0 \end{bmatrix} = \begin{bmatrix} 1.4 \\ 3.3 \end{bmatrix}$$

This dot-product sum vector (s_j^{l-1}) then must be put through the sigmoid activation function of the hidden layer to become the post-activation vector of the hidden layer (a_j^{l-1}).

$$a_j^{l-1} = \begin{bmatrix} \frac{1}{1+e^{-(1.40)}} \\ \frac{1}{1+e^{-(3.30)}} \end{bmatrix} = \begin{bmatrix} 0.802 \\ 0.964 \end{bmatrix}$$

The dot-product sum vector of the output layer (s_k^L) must now be calculated using the same process as previously, making use of the weight and bias matrices between the hidden and output layer (W_{kj}^L and B_k^L).

$$s_k^L = \begin{bmatrix} 0.5 & 1.9 \\ 2.2 & 0.4 \\ 1.8 & 1.2 \end{bmatrix} \cdot \begin{bmatrix} 0.802 \\ 0.964 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} (0.5 \cdot 0.802) + (1.9 \cdot 0.964) \\ (2.2 \cdot 0.802) + (0.4 \cdot 0.964) \\ (1.8 \cdot 0.802) + (1.2 \cdot 0.964) \end{bmatrix} = \begin{bmatrix} 2.23 \\ 2.15 \\ 2.60 \end{bmatrix}$$

This dot-product sum vector (s_k^L) must then be put through the output layer's softmax activation - normalising the sum of the elements to 1 and allowing each element to act as an individual probability of the input's class.

$$a_k^L = \begin{bmatrix} \frac{e^{2.23}}{e^{2.23}+e^{2.15}+e^{2.60}} \\ \frac{e^{2.15}}{e^{2.23}+e^{2.15}+e^{2.60}} \\ \frac{e^{2.60}}{e^{2.23}+e^{2.15}+e^{2.60}} \end{bmatrix} = \begin{bmatrix} 0.297 \\ 0.274 \\ 0.429 \end{bmatrix}$$

This output can be verified as the elements of the post-activation vector (a_k^L) are each between 0 and 1 and add up to 1.

$$\sum_k \left(a_k^L \right) = 0.297 + 0.274 + 0.429 = 1.00$$

The log loss of this network can now be calculated. Having completely random weight and bias matrices, this first log loss of the network is almost never ideal (0) and often misclassifies the input - as the network has here. The input example is of class one - with a target for the first vector element (a_1^L) of 1 , with the other elements having a probability of 0. However, the network believes it has been shown an input of class three, with the highest probability being .429 on class three - with a much lower probability for class one of 0.297. Computing for log loss (E) will quantify this inaccuracy as a single universal value:

$$E = -((1)\log(0.297) + ((0)\log(0.274)) + ((0)\log(0.429))) = 0.527$$

This value, 0.527, is the starting log loss of the network.

Training Iteration One - Backpropagation:

To tune the weight and bias matrices of the MCC ANN to reduce this log loss, the impact of a change in each weight and bias matrix/vector on the final error must be understood by taking partial derivatives using the chain rule to preceding layers.

The first of these differentiations must be of the impact of a change in the final weight matrix (W_{kj}^L) on log loss (E) to calculate the update of the final weight matrix.

$$\boxed{\frac{\partial E}{\partial W_{kj}^L} = \frac{\partial E}{\partial s_k^L} \cdot \frac{\partial s_k^L}{\partial W_{kj}^L}}$$

The differentiation of the first term ($\frac{\partial E}{\partial s_k^L}$)is far outside of the math HL syllabus. It is given in appendix 1 and was *not* completed by me. The differentiation of this term dictates that the change of log loss with respect to the dot-product sum vector equals the difference between each output layer nodes' post-activation output (a_k^L) and target value (t_k).

$$\delta_k^L = \frac{\partial E}{\partial s_k^L} = a_k^L - t_k$$

This resultant vector of values will be useful in future backpropagation and can be assigned to a variable that, according to convention, will be called the *backpropagating error* and given the symbol delta (δ_k^L).

This differentiation of the second term is trivial given the bias vector (B_k^L) acting as a constant and the multiplication of the weight matrix (W_{kj}^L) and the post-activation vector of the (last layer) hidden layer (a_j^{l-1}) allows the change of the dot-product sum vector ($\frac{\partial E}{\partial s_k^L}$) to be simply the post-activation vector of the hidden layer (a_j^{l-1}).

$$\frac{\partial s_k^L}{\partial W_{kj}^L} = \frac{\partial}{\partial W_{kj}^L}\left(B_k^L + \sum_j \left(W_{kj}^L \cdot a_j^{l-1}\right)\right) = a_j^{l-1}$$

As a result, the updating matrix for the matrix of weights connecting the hidden layer to the output ($\frac{\partial E}{\partial W_{kj}^L}$) can be calculated as on the right.

$$\frac{\partial E}{\partial W_{kj}^L} = \frac{\partial E}{\partial s_k^L} \cdot \frac{\partial s_k^L}{\partial W_{kj}^L} = \delta_k^L \cdot a_j^{l-1}$$

To multiply these vectors, the transpose (reversal of dimensionality) of the post-activation vector will be used. This is as, conventionally, multiple training examples - a *training batch* - are given to the network at once and thus the hidden-layer post-activation matrix (a_j^{l-1}), the hidden-output weights matrix (W_{kj}^L) and the second (final) post-activation matrix (a_k^L) must be transposed to ensure the network trains each example as an individual vector. While this network is being justified here by running a training batch of only one example at a time, the transposing of these matrices to allow for compatible matrix multiplication dimensions is still necessary.

$$\delta_k^L = \begin{bmatrix} 0.297 \\ 0.274 \\ 0.429 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -0.703 \\ 0.274 \\ 0.429 \end{bmatrix}$$

$$a_j^{l-1} = \begin{bmatrix} 0.802 \\ 0.964 \end{bmatrix}$$

$$\frac{\partial E}{\partial W_{kj}^L} = \delta_k^L \cdot a_j^{l-1(.T)} = \begin{bmatrix} -0.703 \\ 0.274 \\ 0.429 \end{bmatrix} \cdot \begin{bmatrix} 0.802 & 0.964 \end{bmatrix} = \begin{bmatrix} -0.564 & -0.678 \\ 0.220 & 0.264 \\ 0.344 & 0.414 \end{bmatrix}$$

Likewise, the update vector for the bias vector connecting the hidden layer to the output ($\frac{\partial E}{\partial B_k^L}$) can be calculated similarly.

The differentiation of the dot-product sum vector from log loss term ($\frac{\partial E}{\partial s_k^L}$) is common to the differentiation conducted previously to find the updating matrix of the hidden-output layer weights - the vector of backpropagating error (δ_k^L). The differentiation of the bias vector from the dot-product sum vector ($\frac{\partial s_k^L}{\partial B_k^L}$) is once again trivial given the constant and exclusive effect of the weight matrix (W_{kj}^L) and previous post-activation vector (a_j^{l-1}) on the dot-product sum vector.

$$\frac{\partial E}{\partial B_k^L} = \frac{\partial E}{\partial s_k^L} \cdot \frac{\partial s_k^L}{\partial B_k^L} = \delta_k^L(1)$$

$$\frac{\partial E}{\partial B_k^L} = \delta_k^L = \begin{bmatrix} -0.703 \\ 0.274 \\ 0.429 \end{bmatrix}$$

Thus, the update of the weight matrix and bias vector between the hidden and output layers have been calculated. They will be used to update the current weight and bias matrices at the conclusion of backpropagation. This brings us to the backpropagation of error to the weight and bias matrices between the input and hidden layers to find their respective update matrices ($\frac{\partial E}{\partial W_j^{l-1}}$ and $\frac{\partial E}{\partial B_j^{l-1}}$).

The chain rule once again allows the differentiation of log loss with respect to the weight matrix ($\frac{\partial E}{\partial W_{ji}^{l-1}}$) to be split into multiple, differentiable expressions.

$$\frac{\partial E}{\partial W_{ji}^{l-1}} = \frac{\partial E}{\partial a_j^{l-1}} \cdot \frac{\partial a_j^{l-1}}{\partial s_j^{l-1}} \cdot \frac{\partial s_j^{l-1}}{\partial W_{ji}^{l-1}}$$

The first term of the differentiation is error with respect to the post-activation vector of the hidden layer ($\frac{\partial E}{\partial a_j^{l-1}}$). Conceptually, this term must be a scalar to represent the total change in log loss as each term in the post-activation vector is used in the next dot product - in turn affecting the final log loss. This is achieved through the differentiation with the overarching sigma that adds each element of the outcome matrix as given in full in appendix 2.

$$\frac{\partial E}{\partial a_j^{l-1}} = \sum_k \left(\delta_k^L \cdot W_{kj}^L \right)$$

The second term ($\frac{\partial a_j^{l-1}}{\partial s_j^{l-1}}$) does not require the chain rule to be differentiated. The post-activation vector (a_j^{l-1}) of the hidden layer, being the output of the activation function, ($sigmoid(x)$) renders this partial derivative to simply be the derivative of the activation function with the dot-product sum vector of the hidden layer (s_j^{l-1}).

$$\frac{\partial a_j^{l-1}}{\partial s_j^{l-1}} = \frac{\partial}{\partial s_j^{l-1}} sigmoid\left(s_j^{l-1} \right) = sigmoid'\left(s_j^{l-1} \right)$$

The final term ($\frac{\partial s_j^{l-1}}{\partial W_{ji}^{l-1}}$) is an identical differentiation to that of the dot-product sum vector with respect to its corresponding weight matrix of the hidden-output layer. The only difference is that it now applies to the dot-product sum vector of the hidden layer (s_j^{l-1}) with respect to the weight matrix of the input-hidden layer (W_{ji}^{l-1}) - resulting in the post-activation vector of the input layer: the input vector (x_i).

$$\frac{\partial s_j^{l-1}}{\partial W_{ji}^{l-1}} = x_i$$

Hence, the input-hidden weight matrix update matrix can be calculated as:

$$\frac{\partial E}{\partial W_{ji}^{l-1}} = \frac{\partial E}{\partial a_j^{l-1}} \cdot \frac{\partial a_j^{l-1}}{\partial s_j^{l-1}} \cdot \frac{\partial s_j^{l-1}}{\partial W_{ji}^{l-1}} = \left(\sum_k \left(\delta_k^L \cdot W_{kj}^L \right) \right) \cdot \left(sigmoid'\left(s_j^{l-1} \right) \right) \cdot \left(x_i \right)$$

$$\frac{\partial E}{\partial a_j^{l-1}} = \sum_k \left(\delta_k^L \cdot W_{kj}^{L\,(T)} \right) = \sum_k \left(\begin{bmatrix} -0.703 \\ 0.274 \\ 0.429 \end{bmatrix} \cdot \begin{bmatrix} 0.5 & 2.2 & 1.8 \\ 1.9 & 0.4 & 1.2 \end{bmatrix} \right)$$

$$= \sum_k \left(\begin{bmatrix} ((0.5 \cdot (-0.703)) + (2.2 \cdot 0.274) + (1.8 \cdot 0.429)) \\ ((1.9 \cdot (-0.703)) + (0.4 \cdot 0.274) + (1.2 \cdot 0.429)) \end{bmatrix} \right) = \sum_k \left(\begin{bmatrix} 1.02 \\ -0.711 \end{bmatrix} \right) = 0.309$$

$$\frac{\partial a_j^{l-1}}{\partial s_j^{l-1}} = sigmoid'\left(s_j^{l-1} \right)$$

$$sigmoid'(x) = \left(\frac{1}{1+e^{-x}} \right)' = \frac{\left(1+e^{-x}\right)(0) - (1)\left(-e^{-x}\right)}{\left(1+e^{-x}\right)^2} = \frac{e^{-x}}{\left(1+e^{-x}\right)^2}$$

$$sigmoid'\left(s_j^{l-1} \right) = \begin{bmatrix} \frac{e^{-1.4}}{\left(1+e^{-1.4}\right)^2} \\ \frac{e^{-3.3}}{\left(1+e^{-3.3}\right)^2} \end{bmatrix} = \begin{bmatrix} 0.159 \\ 0.0343 \end{bmatrix}$$

$$\frac{\partial s_j^{l-1}}{\partial W_{ji}^{l-1}} = x_i = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix}^{(T)}$$

$$\frac{\partial E}{\partial W_{ji}^{l-1}} = (0.309) \cdot \left(\begin{bmatrix} 0.159 \\ 0.0343 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix} \right) = (0.309) \cdot \begin{bmatrix} 0 & 0.159 & 0.159 & 0 \\ 0 & 0.0343 & 0.0343 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0.0491 & 0.0491 & 0 \\ 0 & 0.0106 & 0.0106 & 0 \end{bmatrix}$$

It is useful to define another backpropagating error variable (δ_j^{l-1}) to represent the error with respect to the dot-product sum vector of the hidden layer.

$$\delta_j^{l-1} = \frac{\partial E}{\partial a_j^{l-1}} \cdot \frac{\partial a_j^{l-1}}{\partial s_j^{l-1}} = \left(\sum_k \left(\delta_k^L \cdot W_{kj}^L \right) \right) \cdot \left(sigmoid'\left(s_j^{l-1} \right) \right)$$

$$\delta_j^{l-1} = (0.309) \cdot \left(\begin{bmatrix} 0.159 \\ 0.0343 \end{bmatrix} \right) = \begin{bmatrix} 0.0491 \\ 0.0106 \end{bmatrix}$$

Once again it can be used in finding the update matrix for the input-hidden bias matrix (B_j^{l-1}).

$$\boxed{\frac{\partial E}{\partial B_j^{l-1}} = \frac{\partial E}{\partial a_j^{l-1}} \cdot \frac{\partial a_j^{l-1}}{\partial s_j^{l-1}} \cdot \frac{\partial s_j^{l-1}}{\partial B_j^{l-1}} = \left(\delta_j^{l-1} \right)(1)}$$

$$\frac{\partial E}{\partial B_j^{l-1}} = \delta_j^{l-1} = \begin{bmatrix} 0.0491 \\ 0.0106 \end{bmatrix}$$

Now, with the update matrices for both sets of weights and biases (input-hidden and hidden-output), the network can be trained by updating the weights and biases. The final component is the *learning rate variable*[4], denoted ϵ, which regulates the change in the loss-reducing direction that each matrix can take in a single iteration. Conventionally, it is always 0.1 to prevent overstepping ideal weights and the overfitting of the network to a particular training example. It will be used as 0.1 throughout this investigation. To update a matrix, *gradient descent*, the name of this conventional method of updating weights and biases to minimize loss, calls for the subtraction of the product of the learning rate and updating matrix from the original matrix.

$$W_{kj}^{L} = \begin{bmatrix} 0.5 & 1.9 \\ 2.2 & 0.4 \\ 1.8 & 1.2 \end{bmatrix} - (0.1) \begin{bmatrix} -0.564 & -0.678 \\ 0.220 & 0.264 \\ 0.344 & 0.414 \end{bmatrix} = \begin{bmatrix} 0.556 & 1.97 \\ 2.18 & 0.374 \\ 1.77 & 1.16 \end{bmatrix}$$

$$W_{ji}^{l-1} = \begin{bmatrix} 0.5 & 0.1 & 1.3 & 2.1 \\ 1.6 & 2.5 & 0.8 & 1.8 \end{bmatrix} - (0.1) \begin{bmatrix} 0 & 0.0491 & 0.0491 & 0 \\ 0 & 0.0106 & 0.0106 & 0 \end{bmatrix} = \begin{bmatrix} 0.5 & 0.0951 & 1.30 & 2.1 \\ 1.6 & 2.50 & 0.799 & 1.8 \end{bmatrix}$$

$$B_k^{L} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - (0.1) \begin{bmatrix} -0.703 \\ 0.274 \\ 0.429 \end{bmatrix} = \begin{bmatrix} 0.0703 \\ -0.0274 \\ -0.0429 \end{bmatrix}$$

$$B_j^{l-1} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - (0.1) \begin{bmatrix} 0.0491 \\ 0.0106 \end{bmatrix} = \begin{bmatrix} -0.00491 \\ -0.00106 \end{bmatrix}$$

With the updating of these matrices, one iteration of network training - one forward and backpropagation has been completed. The network would now have moved in a log loss-reducing direction closer to achieving high accuracy classifying the inputs. To verify this statement of ANNs, and the validity of the math of this backpropagation, the next forward propagation of this network can be calculated to compute the new log loss of the network after the single training iteration.

Training Iteration 2 - Forward Propagation:

$$s_j^{l-1} = \begin{bmatrix} 0.500 & 0.0951 & 1.30 & 2.10 \\ 1.60 & 2.50 & 0.799 & 1.80 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} -0.00491 \\ -0.00106 \end{bmatrix} = \begin{bmatrix} 1.39 \\ 3.30 \end{bmatrix}$$

$$a_j^{l-1} = sigmoid\left(\begin{bmatrix} 1.39 \\ 3.30 \end{bmatrix} \right) = \begin{bmatrix} 0.801 \\ 0.964 \end{bmatrix}$$

$$s_k^{L} = \begin{bmatrix} 0.556 & 1.97 \\ 2.18 & 0.374 \\ 1.77 & 1.16 \end{bmatrix} \cdot \begin{bmatrix} 0.801 \\ 0.964 \end{bmatrix} + \begin{bmatrix} 0.0703 \\ -0.0274 \\ -0.0429 \end{bmatrix} = \begin{bmatrix} 2.41 \\ 2.08 \\ 2.49 \end{bmatrix}$$

[4] The purpose, impact on training and mathematics of unconventionally adjusting the learning rate is complex and out of the scope of this investigation.

$$a_k^L = softmax\left(\begin{bmatrix} 2.41 \\ 2.08 \\ 2.49 \end{bmatrix}\right) = \begin{bmatrix} 0.357 \\ 0.257 \\ 0.387 \end{bmatrix}$$

This output can be verified as the elements of the post-activation vector (a_k^L) are each between 0 and 1 and add up to 1.00 in 3 significant figures (although raw 1.001 due to 3sf rounding in final vector).

$$\sum_k \left(a_k^L\right) = 0.297 + 0.274 + 0.429 = 1.00$$

$$E = -\log(0.357) = 0.447$$

Hence, the log loss has reduced significantly over a single training iteration from 0.527 to 0.447. This 15.2% decrease in log loss over a single training iteration has validated that the backpropagation of this network adjusts its weights and biases in a log-loss reducing direction and thus has increased the accuracy of the network. This increase in accuracy can also be seen in the difference between the previous probability for the correct class one output of 0.297 as opposed to the increased 0.357 probability after a training iteration. Although the network still does not identify the class of the image correctly, as class three still has the highest probability of 0.387, the difference between the class one's probability and the incorrect class three probability has greatly reduced. On the untrained network the difference was 0.132 (0.429 - 0.297), while after only a single training iteration the difference reduced 77.3% to 0.0300 (0.387 - 0.357). This being a successful demonstration of the mathematics of a training iteration's impact on log loss, it can be understood that MCC ANNs take hundreds, if not thousands, of training iterations across a multitude of training examples to achieve maximal accuracy on any input image to classify. Hence, the 28 by 28 pixel handwritten digit classifier network that will be used to model the trend of decrease in log loss over the first 500 training iterations can be programmed using the validated math of this small-scale 2 by 2 pixel classifier MCC ANN.

Section Ⅳ - Modelling Log Loss Over 500 Iterations of Training 28 by 28 Pixel Handwritten-Digit Classifier:

Using the mathematics of forward and backpropagation completed and validated above, a computer program can now be written to train an MCC ANN for 500 iterations, keeping track of log loss following each training iteration. To produce results useful in the real world however, the MCC ANN architecture will be changed from a 2 by 2 pixel forwardslash, backslash and blacked-out classifier (4 node input) to a 28 by 28 pixel handwritten digit classifier (784 node input). Although the layers will be of different sizes, and thus all matrices of different dimensions, the flexibility of the mathematics of the previous section carries through and forms the basis of training this, larger, network.

This is a network that can be effectively used in the real world to transcribe student hand-written work to document text for easier exemplar marking or distribution. This is a concept that is becoming ubiquitous in this decade of machine learning augmentation - applying to a wide range of handwritten text to digital text applications. To train this network, an MNIST open-source dataset will be used. This dataset contains thousands of labelled example images of 28 by 28 pixel handwritten digits (digits 0 to 9), and is simple to implement in a Python script.

Hence, having extensive experience in Python programming, a simple-to-embed Python training dataset and the mathematics of ANN MCC training - I will be using Python with a handful of Python add-on software libraries to construct a neural network architecture to be trained to classify 28 by 28 pixel images containing handwritten digits - keeping track of its log loss over the first 500 iterations of its training. Like the 2 by 2 pixel classifier, the network will have a three layer architecture with 784 nodes, 256 nodes and 10 nodes in the layers respectively. The training batch size will, as per convention, be 128 examples instead of only 1 - the transposing of the three key matrices mentioned in section Ⅲ allows this *batch training* to taken place - allowing for 500 iterations of training to have a shorter execution time duration. It also means the log loss value of the network will be the sum of the log losses of the individual training examples in a training batch. Hence, the resultant model will only be applicable to any MCC ANN with a three-layer architecture and 128 training examples per training batch. This is a common architecture of MCC ANNs and hence the resultant model will be applicable to a large range of real-world applications.

This architecture will be the framework for 10 trials of the network (10 independent networks - each starting completely untrained). Each trial will have different randomly-assigned starting weights and biases - and thus a different starting loss. Given the similarity in architecture, however, each trial will likely have similar best-fit equations for log loss over training iterations - whose average will provide an accurate modelling equation for log loss over training iterations.
Figure 4.1 below is a documented excerpt of my 224-line Python program using the mathematics derived in section Ⅲ to construct and train a 28 by 28 pixel MCC ANN to be classify handwritten digits.

```
def runVarNodes(h1_nodes, h2_nodes, h3_nodes):                          # function containing network
    mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
    train_data = mnist.train.images  # Returns np.array                 # assigning datasets and labels
    train_labels = np.asarray(mnist.train.labels, dtype=np.int32)
    eval_data = mnist.test.images  # Returns np.array
    eval_labels = np.asarray(mnist.test.labels, dtype=np.int32)

    tf.logging.set_verbosity(old_v)                                     # for debugging Python script

    n_train = mnist.train.num_examples                                  # 55000 training examples
    n_test = mnist.test.num_examples                                    # 10000 test examples

    n_input = 784                                                       # input layer (28x28 pixels) - 784 nodes
    n_hidden1 = 256                                                     # 1st hidden layer           - 256 nodes
    n_output = 10                                                       # output layer (0-9 digits)  - 10 nodes

    alpha = 0.1                                                         # 0.1 learning rate (also called 'alpha' )
    n_iterations = 500                                                  # assigning 500 training iterations
    batch_size = 128                                                    # assigning 128 batch size
    dropout = 0.5                                                       # prevents malfunction of network

    synapses = {                                                        # intialising weight matrices
        'w1': tf.Variable(tf.truncated_normal([n_input, n_hidden1], stddev=0.1)),
        'out': tf.Variable(tf.truncated_normal([n_hidden1, n_output], stddev=0.1)),
    }

    biases = {                                                          # intialising bias matrices
        'b1': tf.Variable(tf.constant(0.1, shape=[n_hidden1])),
        'out': tf.Variable(tf.constant(0.1, shape=[n_output]))
    }
```

Figure 4.1: Excerpt of Initialisation of Network Architecture and Parameters Using '*Tensorflow*' Python library

Figure 4.2 below is the graph produced by plotting the log loss over training iterations for each training iteration and the subsequent best-fit line of the first trial 28 by 28 pixel MCC ANN.

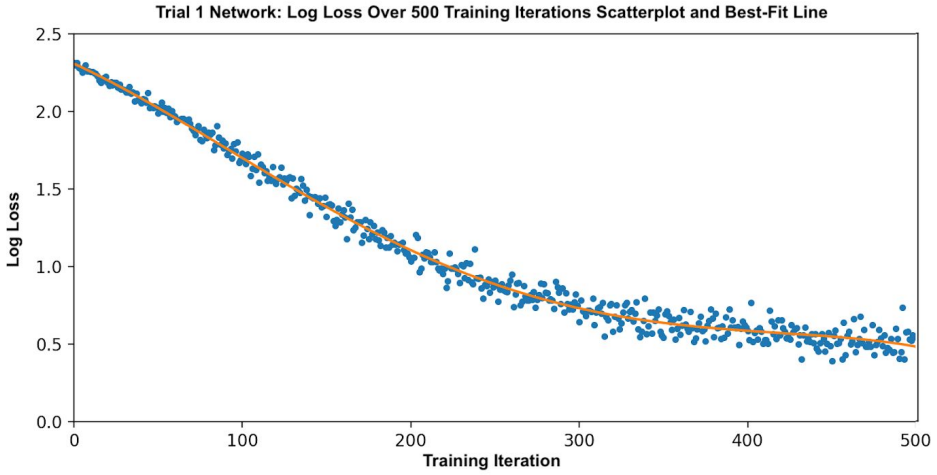

Figure 4.2: Trial 1 Network : Log Loss Over 500 Training Iterations Scatterplot and Best-Fit Line

Figure 4.3 below is the table of 10 trials of this network architectures' best-fit line equations relating log loss over 500 training iterations. These quartic best-fit lines were calculated by a Python library called *'Matplotlib'* having been given the network's log loss value at the end of every training iteration. The difference between the equations is a result of each trial of the network having different randomly-assigned initial weights and biases. The average of each term of the quartics together reliably model the average relation between the first 500 training iterations and log loss of the classifier.

Figure 4.3: Equations of Best-Fit Line per Network Trial and Processed Average - Split by Equation x Degree Term

Trial	x^4	x^3	x^2	x^1	x^0
1	-8.64E-11	6.69E-08	-8.01E-06	-6.55E-03	2.29E+00
2	-7.97E-11	6.89E-08	-8.19E-06	-6.78E-03	2.29E+00
3	-7.95E-11	7.17E-08	-7.96E-06	-6.87E-03	2.34E+00
4	-8.00E-11	7.59E-08	-8.65E-06	-6.59E-03	2.35E+00
5	-7.81E-11	6.72E-08	-8.99E-06	-7.36E-03	2.30E+00
6	-7.66E-11	6.23E-08	-8.48E-06	-7.13E-03	2.30E+00
7	-7.99E-11	7.58E-08	-8.51E-06	-6.99E-03	2.33E+00
8	-7.43E-11	7.48E-08	-8.53E-06	-6.85E-03	2.31E+00
9	-7.95E-11	6.93E-08	-9.09E-06	-6.96E-03	2.32E+00
10	-8.84E-11	7.59E-08	-9.13E-06	-6.73E-03	2.31E+00
AVG	**-8.02E-11**	**7.09E-08**	**-8.55E-06**	**-6.88E-03**	**2.31E+00**

Figure 4.4 below is the graph produced by plotting log loss over training iterations for all 10 trials (each trial in a different colour). It is then superimposed by the average best-fit line of the 10 trials as calculated by figure 4.3 above. This visualises the log loss values over training iterations of each trial network and thus shows that the calculated average best-fit line captures the trend well.

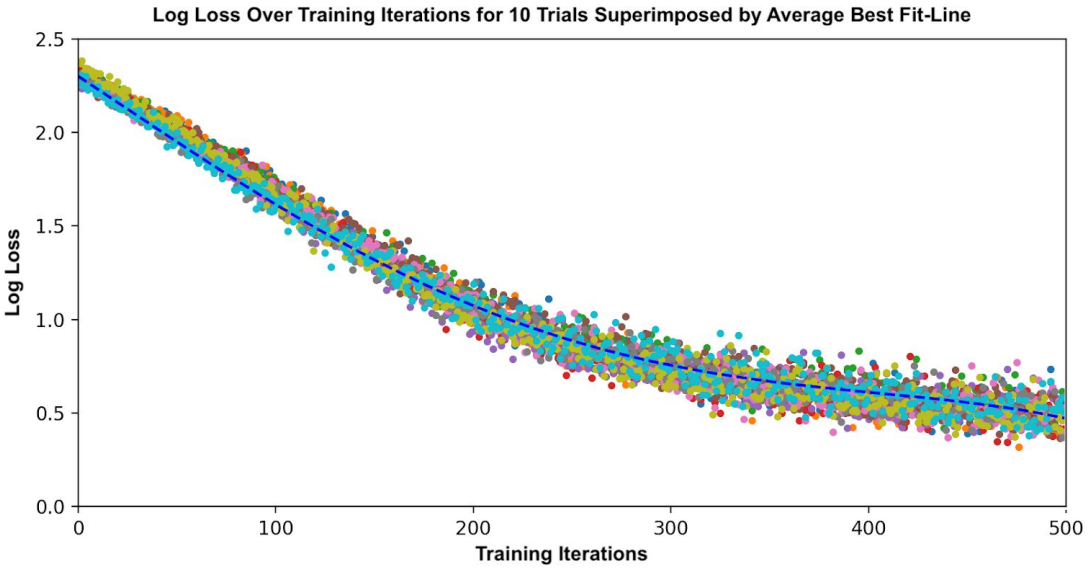

Figure 4.4 : 10 Trials of Log Loss Plots Superimposed by Average Best-Fit Line for Log Loss over 500 Training Iterations

The final outcome of this investigation was the identification of the equation relating log loss to first 500 training iterations of an MCC ANN with conventional parameters and a three-layer architecture. By initially covering the premise of neural networks and deconstructing the concepts and mathematics of a general MCC ANN, I was able to develop and justify by hand, the mathematical training steps for a three-layer 2 by 2 pixel classifier. These steps used pre-university mathematics structured in a manner that my peers can understand the meaning behind the steps. This math holds true for all similar three-layer MCC ANN architectures and hence the function aiming to be modelled by this investigation was chosen to be of a larger MCC ANN with a real world application. This 28 by 28 pixel handwritten digit classifier was programmed using the investigated mathematics and allowed the network's log loss to be recorded over 500 training iterations. By creating a program, 10 trials of the network could easily be conducted to account for random weight and bias matrices affecting starting log loss values.

The average of the 10 independent best-fit lines yields the final model. Applicable to any MCC ANN with three-layers using (the conventional) sigmoid and softmax activations, a 128-example training batch size and a 0.1 learning rate, the final calculated equation relating log loss to the first 500 training iterations is:

$$L(x) = (-8.02 \cdot 10^{-11})x^4 + (7.09 \cdot 10^{-8})x^3 + (-8.55 \cdot 10^{-6})x^2 + (-6.88 \cdot 10^{-3})x + 2.31,\ 0 \leq x \leq 500$$

The implication of this equation is primarily the ability to visualise the training ability and trend of this architecture of MCC ANN. This, in turn, fulfills my goal of demonstrating in an applied sense the training of an MCC ANN to those who are otherwise uninitiated in ANNs. The understanding of the mathematics of forward and backpropagation in a three-layer neural network is the crux of this investigation as it applies to any similar three-layer network. While this investigation has the limitation of having results restricted to the first 500 iterations of a three-layer neural network, it demonstrates in a visual and understandable manner the general training of an MCC ANN. Given the conceptual, mathematical and practical aspects of this investigation, I also now understand these steps to a higher degree. I believe for my future, this investigation is of great significance. As a personal extension, I intend to expand my personal research project on the identification and prevention of temperature stress conditions from a two-layer MCC ANN to a three-layer MCC ANN - whose mathematics and programming completed here will save time and efforts be informed with the understandings gleaned through this successful investigation. Hopefully, upon reading this investigation, others may too be inspired and informed to conduct their own MCC ANN-based projects too.

Works Cited:

"Artificial Intelligence." *Wikipedia*, Wikimedia Foundation, 15 Oct. 2019, en.wikipedia.org/wiki/Artificial_intelligence.

Ashraf, Javed. "Calculating the Backpropagation of a Network." *Medium*, Towards AI, 2 June 2019, medium.com/towards-artificial-intelligence/calculating-back-propagation-of-a-network-1febbcaa2b5d.

Osajima, Jason. "The Math behind Neural Networks - Backpropagation." *The Math behind Neural Networks - Backpropagation | Jason {Osa-Jima}*, www.jasonosajima.com/backprop.

Prin, Bill. "A TensorFlow Glossary/Cheat Sheet." *Medium*, Google Cloud Platform - Community, 29 Aug. 2017, medium.com/google-cloud/a-tensorflow-glossary-cheat-sheet-382583b22932.

Roell, Jason. "From Fiction to Reality: A Beginner's Guide to Artificial Neural Networks." *Medium*, Towards Data Science, 19 June 2017, towardsdatascience.com/from-fiction-to-reality-a-beginners-guide-to-artificial-neural-networks-d0411777571b.

Appendix:

The differentiations of the appendix are in part, or wholly, sourced from https://www.jasonosajima.com/backprop.Understanding the differentiations has no impact on understanding the concepts of MCC ANNs, as they have been explained and covered in simple terms through the investigation. They are however useful in understanding how certain formulae have been arrived at.

Appendix 1:

$$\frac{\partial E}{\partial W_{kj}^L} = \frac{\partial E}{\partial s_k^L} \cdot \frac{\partial s_k^L}{\partial W_{kj}^L}$$

This partial differentiation can be calculated by using the chain rule to break it into the change of log loss with respect to the change of the dot-product sum vector of the final layer ($\frac{\partial E}{\partial s_k^L}$) multiplied by the change of the dot-product sum vector with respect to the change of the final weight matrix ($\frac{\partial s_k^L}{\partial W_{kj}^L}$). The first term is differentiated as:

$$\mathbb{1}_{d=k} = \begin{cases} 1 & \text{if } d=k \\ 0 & \text{else} \end{cases}$$

$$\frac{\partial E}{\partial s_k^L} = -\sum_d \left(t_d \left(\mathbb{1}_{d=k} - \frac{1}{\sum_c \left(e^{s_c^L} \right)} \cdot e^{s_k^L} \right) \right) = -\sum_d \left(t_d \left(\mathbb{1}_{d=k} - a_k^L \right) \right)$$

$$= -\sum_d \left(t_d \left(\mathbb{1}_{d=k} - a_k^L \right) \right) = \sum_d \left(t_d \cdot a_k^L \right) - \sum_d \left(t_d \cdot \mathbb{1}_{d=k} \right)$$

$$= a_k^L \cdot \sum_d \left(t_d - t_k \right) = a_k^L - t_k$$

$$\delta_k^L = \frac{\partial E}{\partial s_k^L} = a_K^L - t_k$$

Appendix 2:

$$\frac{\partial E}{\partial a_j^{l-1}} = \sum_k \left(\frac{\partial E}{\partial s_k^L} \cdot \frac{\partial s_k^L}{\partial a_j^{l-1}} \right) = \sum_k \left(\frac{\partial E}{\partial s_k^L} \cdot \frac{\partial}{\partial a_j^{l-1}} \left(B_k^L + \sum_j \left(W_{kj}^L \cdot a_j^{l-1} \right) \right) \right) = \sum_k \left(\delta_k^L \cdot W_{kj}^L \right)$$

The chain rule used to solve the first term ($\frac{\partial E}{\partial a_j^{l-1}}$) once again has much in common with previous differentiations and can be substituted with the previously assigned backpropagating error (δ_k^L) and trivially differentiated hidden-output weight matrix (W_{kj}^L) - hence meaning the multiplication of these terms.