Mathematics IB Internal Assessment

An Investigation into the Mathematics of Neural Networks

AI (Artificial Intelligence) IA (Internal Assessment)

Name: ……………………………………………………Vikram Procter

IB Candidate Number: ………………………………………..JSV607

Date: ……………………………………………………………..2023

Introduction:

If someone was asked to identify what figure 1 is, they would say it's a "3" (a trivial task). In a person's head the 28x28 pixel that is figure 1, creates the response "3". Now if this same 28x28 pixel array were given to a computer, what mathematical model can complete the task of identifying the image as a three. And beyond that also be able to recognize that every image in figure 2 is a three. To people this seems trivial but mathematically very complex. The best solution mathematics and computation has to offer is neural networks or artificial intelligence.

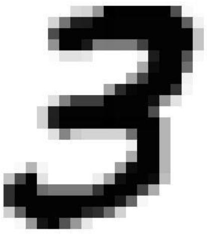
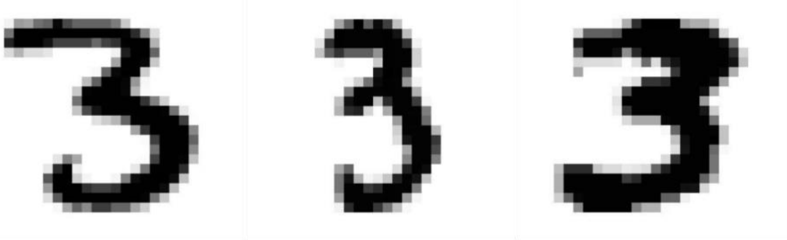


Figure 1 - Image of Handwritten '3'

Figure 2 - Three Images of Handwritten '3's



I began taking an interest in programming in 2019, at that time the mathematical and programming concepts utilized within neural networks were far too advanced for the novice I was at the time. Now equipped with the mathematical knowledge of the past three years of high school, along with extensive independent research, I have the motivation and knowledge to fully understand how these complex mathematical masterpieces, we call neural networks, function. My aim is to display the inner workings of neural networks and apply them by providing an example of how to solve the problem of recognizing handwritten numbers mathematically.

Architecture of a Neural Network:

To understand how a neural network works, it is important to understand its architecture. You can think of a neural network as a machine where one end accepts inputs and the other that outputs a decision.

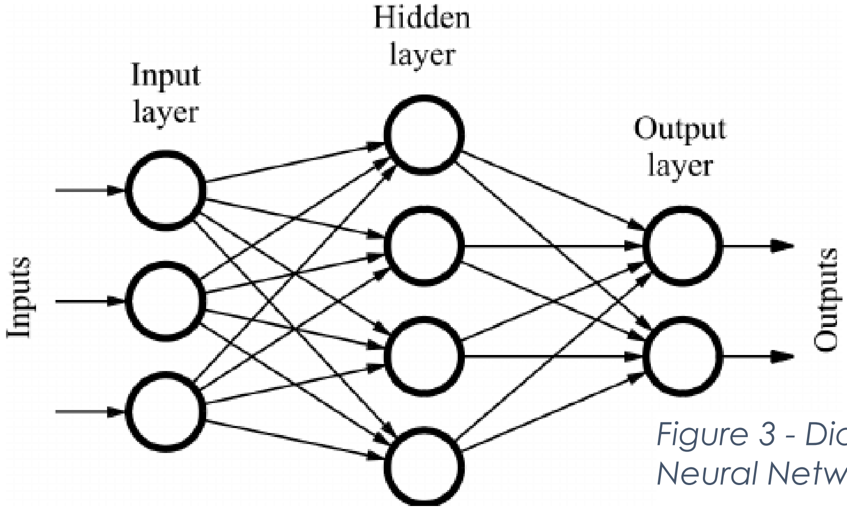


Figure 3 - Diagram of Neural Network[1]

Neural networks are organized into **layers**. All Neural networks require at least three layers: input layer, hidden layer, and output layer. More complex neural networks use more hidden layers to capture the complexity of the problem (Figure 3 there is one hidden layer).

Each layer within a neural network is organized into **neurons** (figure 3 shows each neuron as a circle). Each neuron holds a value called an "**activation**", the activation ranges from 0 (not activated) to 1 (activated). The first layer, the input layer, represents some situation. Ex. in the handwritten number identifier, each pixel would be an input and the value of the pixel would be equal to the activation of the first layer's neurons. Mathematically, inputs are arranged as a one column matrix:

$$x_i = input\ at\ index\ 'i'\ (or\ neuron\ 'i'\ in\ the\ first\ layer)$$

The layers after the first layer but before the last layer are called hidden layers. These values are also arranged in a single column matrix, written out like this:

$$a_i^l = hidden\ neuron\ i\ in\ layer\ l$$

Where the superscripts are the layer index and **not** exponents (as seen in figure 4). This will be superscript layer notation will be used going forward.

The final layer in the neural network is the output layer, it functions much like the hidden layers before it. Notationally:

$$a_i^L = output\ at\ index\ i\ in\ the\ last\ layer\ L$$

To use the input activations to create an output there is a web of connections between each layer, these connections are called weights. In Figure 4 these are depicted as arrows between the layers. The weight determines the importance of a neuron on the next layer. How these weights work will be explained in the next section. Notationally:

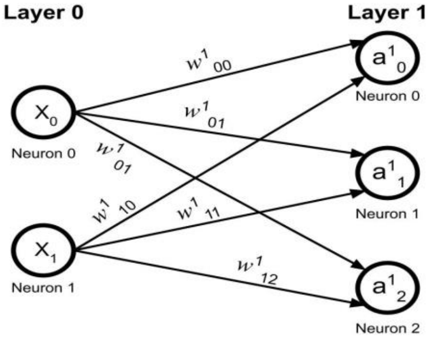


Figure 4 - First Two Layers of Example Neural Network with Weights Labeled

$$w_{j,k}^l = weight\ stating\ at\ neuron\ k\ in\ layer\ l-1, connecting\ to\ neuron\ j\ in\ layer\ l$$

Not shown on these simplified diagrams (figure 3 or 4) is another value connected to every neuron (except for the inputs) called a bias. It is like the weights however it is independent from the preceding neurons. Notationally:

$$b_i^l = bias\ connecting\ to\ neuron\ i\ in\ layer\ l$$

In summary, there are three classifications of layers in a neural network: input, hidden and output. The input layer takes in values and is connected to the next hidden layer with a web of weights. There can be any number of these hidden layers each connected to the next with a web of weights as well as a bias. The last layer is the output layer holding the values of the decision the neural network made.

Neural Network Function – How does it Work?

Mathematically neural networks create outputs through a series of function composition and matrix multiplication[2]. The process of taking the inputs and mathematically calculating the outputs is called the "Feed Forward" process[3]. The value of a hidden layer neuron is calculated using a bias added to a weighted sum of the neurons in the previous layer and then normalized using an activation function. A very popular activation function[4] used is called the "sigmoid function[5] or $\sigma(x)$", normalizing values to between 0 and 1. (Seen in figure 5).

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

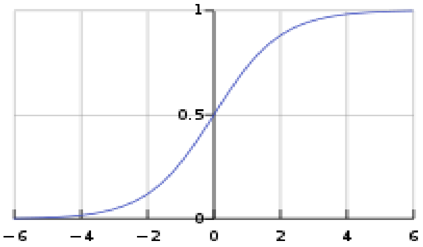


Figure 5 - Graph of Sigmoid Function[5]

To calculate the activation of a neuron each in neuron the previous layer is multiplied by the weight connecting the two neurons together and all of these values are added together. Once this weighted sum has been calculated a bias is added. This final value is then run through the sigmoid function calculating the normalized activation for that neuron. Notationally – For any hidden layer feeding forward to the next:

$$a_i^l = \sigma\left(\left(w_{i,0}^l a_0^{l-1} + b_0^l\right) + \left(w_{i,1}^l a_1^{l-1} + b_1^l\right) + (w_{i,2}^l a_2^{l-1} + b_2^l) + \cdots + (w_{i,n}^l a_n^{l-1} + b_i^l)\right)$$

Where n is the last or total neurons in the previous layer

This formula can be applied to all layers:

First hidden layer (layer 1):

$$a_i^1 = \sigma(w_{i,0}^1 x_0 + w_{i,1}^1 x_1 + w_{i,2}^1 x_2 + \cdots + w_{i,n}^1 x_n + b_i^1)$$

Output layer (layer l):

$$a_i^L = \sigma(w_{i,0}^L a_0^{L-1} + w_{i,1}^L a_1^{L-1} + w_{i,2}^L a_2^{L-1} + \cdots + w_{i,n}^L a_n^{L-1} + b_i^L)$$

Where L is the last layer index

To manage all this data, and intelligently preform these calculations, matrices can be used.

Each layer can be arranged into a single dimensional matrix with one column and as many rows as neurons. The weights can be placed in a 2-dimensional matrix with as many columns as neurons in the layer it begins in and as many rows as neurons in the layer it ends in. A bias matrix can also be represented as 1D matrix with one column and as many rows as neurons. If the matrices are arranged in this fashion the dot product can be taken between the weight matrix and the input matrix and then added to the bias matrix before being mapped into the sigmoid function to produce the next layer neuron values.

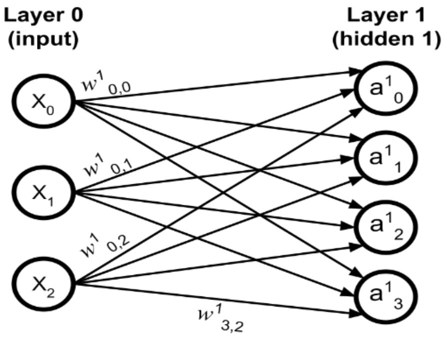


Figure 6 - Example Neural Network Diagram (First Two Layers)

For example, the neural network shown in figure 6 would look like this:

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \qquad w^1 = \begin{bmatrix} w_{0,0}^1 & w_{0,1}^1 & w_{0,2}^1 \\ w_{1,0}^1 & w_{1,1}^1 & w_{1,2}^1 \\ w_{2,0}^1 & w_{2,1}^1 & w_{2,2}^1 \\ w_{3,0}^1 & w_{3,2}^1 & w_{3,2}^1 \end{bmatrix} \qquad b^1 = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

$$a^1 = \sigma((w^1 \bullet x) + b^1)$$

$$a^1 = \sigma\left(\left(\begin{bmatrix} w_{0,0}^1 & w_{0,1}^1 & w_{0,2}^1 \\ w_{1,0}^1 & w_{1,1}^1 & w_{1,2}^1 \\ w_{2,0}^1 & w_{2,1}^1 & w_{2,2}^1 \\ w_{3,0}^1 & w_{3,2}^1 & w_{3,2}^1 \end{bmatrix} \bullet \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}\right) + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}\right) = \begin{bmatrix} \sigma((x_0 \cdot w_{0,0}^1 + x_1 \cdot w_{0,1}^1 + x_2 \cdot w_{0,2}^1) + b_0^1) \\ \sigma((x_0 \cdot w_{1,0}^1 + x_1 \cdot w_{1,1}^1 + x_2 \cdot w_{1,2}^1) + b_1^1) \\ \sigma((x_0 \cdot w_{2,0}^1 + x_1 \cdot w_{2,1}^1 + x_2 \cdot w_{2,2}^1) + b_2^1) \\ \sigma((x_0 \cdot w_{3,0}^1 + x_1 \cdot w_{3,1}^1 + x_2 \cdot w_{3,2}^1) + b_3^1) \end{bmatrix} = \begin{bmatrix} a_0^1 \\ a_1^1 \\ a_2^1 \\ a_3^1 \end{bmatrix}$$

$$a^l = \sigma((w^l \bullet a^{l-1}) + b^l)$$

Where $M_1 \bullet M_2$ is the matrix multiplication between the two matrices $M_1 \bullet M_2$ (more info in appendix).

Neural Network Function – Why does it work?

Answering how a neural network produces an output is only half the battle, understanding "why", is the other half. An artificial neural network is loosely based on the biological neural networks we call brains. The idea is that when one neuron is fired (activation > 0.5) it stimulates other neurons connected to it, a certain amount (the weight). The bias we add to each neuron also approximates biology. Biological nerves only fire when a threshold is reached, the bias we add acts like this threshold, only allowing a neuron to be active if the weighted sum reaches a certain level controlled by the bias. All neuron values are run through the sigmoid normalization function to allow each neuron to be proportional with ever other one.

A neural network produces an output by calculating biased and weighted sums, the only way to get the correct outputs is by having the correct biases and weights. This proposes the question: How are the "correct" biases and weights calculated? The answer is learning, the neural network must learn how to tune all the weights and biases to calculate the right decision.

How do you “teach” a neural network?

Neural networks learn by example, huge data sets of inputs and the desired or “correct” outputs are used to teach how to change the weights and biases. The overall process is called **Back Propagation**. To start this process all the weights and biases are set randomly. At this point an input is given to the neural network, and a completely random (very likely wrong) output is calculated through the feed forward algorithm. The square of the difference between this wrong output (a_i^L) and the expected outputs (y_i), are averaged calculating the called the “cost” of a single training example. Where y_i is the expected value.

$$C_t(a^L, y) = \frac{\sum_i^n (a_i^L - y_i)^2}{n} = Cost\ of\ a\ single\ training\ example$$

For t is the given training example.

The full cost function is an average of each of the training examples:

$$C(a^L, y) = \frac{\sum_{k=0}^K C_k(a^L, y)}{K}$$

This cost value will be large when the neural network decisions do not match the expected values and small when the neural network is producing outputs close to the expected outputs. This cost function is taken for each of the many training examples, and then averaged to produce a measure for how inaccurate the neural network is behaving.

This average cost value only reveals how bad the neural network is doing and not how to improve it. To improve the neural network, the cost function must be minimized. To find the local minimum or maximum of a function the derivative at a specific point must be zero. For simpler functions this can be performed, however this is not feasible for very complicated such as this cost function (seeing how a typical neural network trying to tackle the “handwritten number problem” could take in over 13000 parameters into this function).

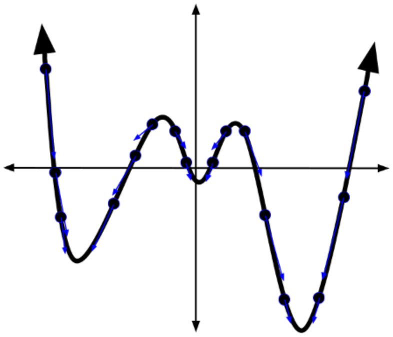


Figure 7 - 2D Gradient Decent Graph

Instead, random weights are assigned, and the gradient[6] of the cost function is calculated using partial derivatives. The gradient is which “direction” (change) to the weights produces the steepest increase in cost. The negative of this gradient the steepest decrease in the cost function can be found. This is the quickest way to change the weights to produce the least cost (or error). This gradient will find which direction each of the weights should be changed and by how much based on the steepness of the curve. This idea can be seen in the simplified to a 2D diagram (figure 7), where the x axis shows a one-dimensional representation of the weights and y representing the cost function. Each point on this graph is an example of a random place to define the weights as and the arrow showing the negative gradient.

In summery the gradient defines how changes in the weights and biases changes cost function changes and therefore can be used to change the weights biases toward decreasing the cost function.

To understand the gradient calculations, a simplified model will be used initially:

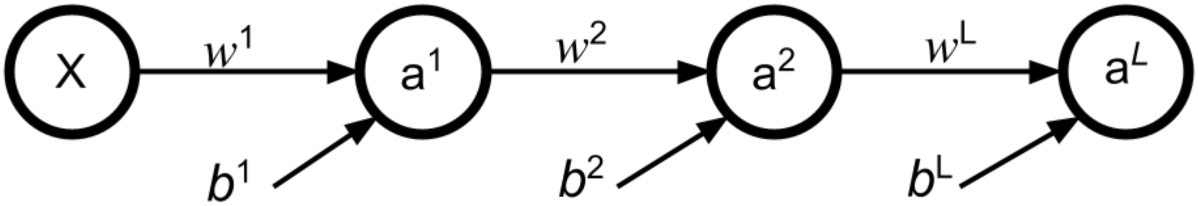


Figure 8 - Simplified Neural Network with Only One Neuron in Each Layer

This model is composed of three weights and three biases, each of these weights and biases impacts the cost function in some way:

$$C_t(a^L, y) = (a^L - y)^2$$

Using the feed forward algorithm a^L is defined as this:

$$a^L = \sigma(w^L a^{L-1} + b^L)$$

To make apply partial derivative methods, z will be used to represent an intermediate value:

$$z^L = w^L a^{L-1} + b^L$$

$$a^L = \sigma(z^L)$$

Calculating this gradient vector between the cost function and the weights using partial derivatives:

Focusing on **only** the connections between the last two neurons:

$$\frac{\partial C_t}{\partial w^L} = \frac{\partial z^L}{\partial w^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial C_t}{\partial a^L}$$

This is essentially chain rule for the cost function's, function composition relative to one training example, t.

The derivative of *cost* function with respect to the *weights* can be represented as the derivatives of each of the intermediate steps. How changes in w^L impact z^L, and how changes in z^L impact a^L, and then finally how changes in a^L impact C_t.

Dissecting each component:

$$\frac{\partial z^L}{\partial w^L} = a^{L-1}$$

This means that changes in the weight value will change the z^L proportionally to the value of the neuron it was connected to before it.

$$\frac{\partial a^L}{\partial z^L} = \sigma'(z^L)$$

This means that changes in z^L impact a^L, based on the derivative of the sigmoid function.

The derivative of the sigmoid function (σ'):

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

Note: Work for derivative can be found in appendix

$$\frac{\partial C_t}{\partial a^L} = 2(a^L - y)$$

This means that changes in the a^L, change the cost function proportionally to the difference between what is expected and the value the neural network found.

Therefore, the derivative of the cost function (of one training example) looks like this:

$$\frac{\partial C_t}{\partial w^L} = (a^{L-1})\sigma'(z^L)2(a^L - y) = 2\sigma'(z^L)(a^{L-1})2(a^L - y)$$

This determines how the weights should be changed to increase the cost function.

The **full cost function** regarding one layer's weights through all training examples:

$$\frac{\partial C}{\partial w^L} = \frac{1}{n}\sum_{k=0}^{n}\frac{\partial C_k}{\partial w^L}$$

$\frac{\partial C}{\partial w^L}$ is only one part of the cost *gradient vector* (only representing the w^L) portion. The bias must also be accounted for.

The portion of the gradient that is the *bias*, is very similar to the weights:

$$\frac{\partial C_t}{\partial b^L} = \frac{\partial z^L}{\partial b^L}\cdot\frac{\partial a^L}{\partial z^L}\cdot\frac{\partial C_t}{\partial a^L} = (1)\sigma'(z^L)2(a^L - y) = \sigma'(z^L)2(a^L - y)$$

This determines how the biases should be changed to increase the cost function.

Because:

$$z^L = w^L a^{L-1} + b^L$$

$$\frac{\partial z^L}{\partial b^L} = 1$$

And therefore, the **full cost function** regarding one layer's bias:

$$\frac{\partial C}{\partial b^L} = \frac{1}{n}\sum_{k=0}^{n}\frac{\partial C_k}{\partial b^L}$$

To find how layers before the last impact the cost function (and subsequently how to change them), the idea of **Back Propagation** comes into play. To find the gradient of a pervious layer a similar method is applied. Looking at one layer before the last:

$$z^{L-1} = w^{L-1}a^{L-2} + b^{L-1} \qquad a^{L-1} = \sigma(z^{L-1})$$

$$\frac{\partial C_t}{\partial w^{L-1}} = \frac{\partial z^{L-1}}{\partial w^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial C_t}{\partial a^{L-1}}$$

$$\frac{\partial z^{L-1}}{\partial w^{L-1}} = a^{L-2} \qquad \frac{\partial a^{L-1}}{\partial z^{L-1}} = \sigma'(z^{L-1}) \qquad \frac{\partial C_t}{\partial a^{L-1}} = a^{L-1}\ impact\ on\ C_t$$

$$\frac{\partial C_t}{\partial w^{L-1}} = (a^{L-2})\sigma'(z^{L-1})\frac{\partial C_t}{\partial a^{L-1}}$$

This process can be repeated for the biases.

$$\frac{\partial C_t}{\partial b^{L-1}} = \frac{\partial z^{L-1}}{\partial b^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial C_t}{\partial a^{L-1}} = (1) \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial C_t}{\partial a^{L-1}} = \sigma'(z^{L-1})\frac{\partial C_t}{\partial a^{L-1}}$$

Because:

$$z^{L-1} = w^{L-1}a^{L-2} + b^{L-1} \qquad \frac{\partial z^{L-1}}{\partial b^{L-1}} = 1$$

Both the weights and biases of a previous layer are proportional to the gradient of this previous layer's activation. This is the idea behind back propagation, propagating the gradient backwards throughout each layer. By applying the same partial derivative method:

$$z^L = w^L a^{L-1} + b^L \qquad a^L = \sigma(z^L) \qquad C_t(a^L, y) = (a^L - y)^2$$

$$\frac{\partial C_t}{\partial a^{L-1}} = \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial C_t}{\partial a^L} = w^L \sigma'(z^L)2(a^L - y)$$

Because:

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

Applying this logic to any layer j^{th} previous from the last layer:

$$\frac{\partial C_t}{\partial w^{L-\mathrm{j}}} = (a^{L-\mathrm{j}-1})\sigma'(z^{L-\mathrm{j}})\frac{\partial C_t}{\partial a^{L-\mathrm{j}}}$$

$$\frac{\partial C_t}{\partial b^{L-\mathrm{j}}} = \sigma'(z^{L-\mathrm{j}})\frac{\partial C_t}{\partial a^{L-\mathrm{j}}}$$

$$\frac{\partial C_t}{\partial a^{L-\mathrm{j}}} = w^{L-j}\sigma'\left(z^{L-j}\right)\frac{\partial C_t}{\partial a^{L-j+1}}$$

Each equation defined recursively until the last layer, which is calculated using the expected outcomes. This recursive definition allows the last layers gradient to propagate throughout the previous layer's gradient calculations.

The previous gradient calculations used a simplified one neuron on each level, to move to an actual neural network with multiple neurons on each level. Although this moves up several dimensions it stays fairly similar, just a few more indices to keep track of.

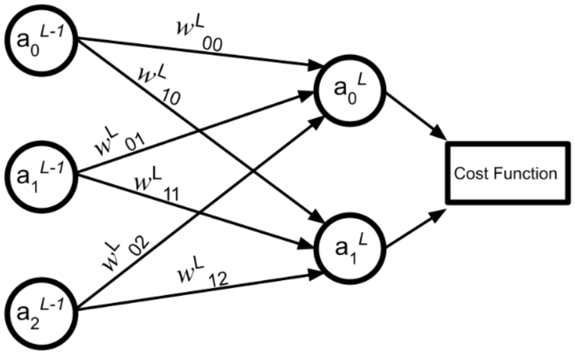


Figure 9 - Last Two layers of Example Neural Network

$z_i^L = w_{i,0}^L a_0^{L-1} + w_{i,1}^L a_1^{l-1} + w_{i,2}^L a_2^{L-1} + \cdots + w_{i,n}^L a_n^{L-1} + b_i^L \qquad a_i^L = \sigma(z_i^L)$

$$C_t(a^L, y) = \sum_{i=0}^{n} \left(a_i^L - y_i\right)^2$$

Now the gradient refers to one of the weights ($w_{i,k}^L$) connected to the last layer (L):

$$\frac{\partial C_t}{\partial w_{i,k}^L} = \frac{\partial z_i^L}{\partial w_{i,k}^L} \cdot \frac{\partial a_i^L}{\partial z_i^L} \cdot \frac{\partial C_t}{\partial a_i^L}$$

$$\frac{\partial z_i^L}{\partial w_{i,k}^L} = a_k^{L-1} \qquad \frac{\partial a_i^L}{\partial z_i^L} = \sigma'(z_i^L) \qquad \frac{\partial C_t}{\partial a_i^L} = 2\left(a_i^L - y_i\right)$$

$$\frac{\partial C_t}{\partial w_{i,k}^L} = (a_k^{L-1})\sigma'\left(z_i^L\right)2\left(a_i^L - y_i\right) = 2\sigma'\left(z_i^L\right)(a_k^{L-1})\left(a_i^L - y_i\right)$$

Expanding this gradient to all the weights in the partial gradient weight matrix connected to the last layer.

$$\frac{\partial C_t}{\partial w^L} = \begin{bmatrix} 2\sigma'(z_0^L)(a_0^{L-1})(a_0^L - y_0) & 2\sigma'(z_0^L)(a_1^{L-1})(a_0^L - y_0) & \cdots & 2\sigma'(z_0^L)(a_k^{L-1})(a_0^L - y_0) \\ 2\sigma'(z_1^L)(a_0^{L-1})(a_1^L - y_0) & 2\sigma'(z_1^L)(a_1^{L-1})(a_1^L - y_1) & \dots & 2\sigma'(z_1^L)(a_k^{L-1})(a_1^L - y_1) \\ \vdots & \vdots & \ddots & \vdots \\ 2\sigma'\left(z_i^L\right)(a_0^{L-1})\left(a_i^L - y_i\right) & 2\sigma'\left(z_i^L\right)(a_1^{L-1})\left(a_i^L - y_i\right) & \dots & 2\sigma'\left(z_i^L\right)(a_k^{L-1})\left(a_i^L - y_i\right) \end{bmatrix}$$

Comparing this matrix to the weights matrix: a_k^{L-1} the k corresponds to the k in $w_{i,k}^L$ and the column index in both matrices; z_i^L as well as $(a_i^L - y_i)$ the i corresponds to the i in $w_{i,k}^L$ and the row index in both matrices. This resultant weight gradient matrix can be mathematically looked at as matrix product the matrices $z^L, (a^l - y)$ and a^{L-1} matrix transpose (appendix 3):

$$\frac{\partial C_t}{\partial w^L} = \left(2 \begin{bmatrix} a_0^L - y_0 \\ a_1^L - y_1 \\ \vdots \\ a_i^L - y_i \end{bmatrix} \sigma' \begin{bmatrix} z_0^L \\ z_0^L \\ \vdots \\ z_\mathrm{i}^L \end{bmatrix} \right) \bullet [(a_0^{L-1}) \quad (a_1^{L-1}) \quad \dots \quad (a_k^{L-1})]$$

$$\frac{\partial C_t}{\partial w^L} = (2\big(a^l - y\big)\sigma'(z^L)) \bullet T((a^{L-1}))$$

Where $T(M)$ is the transposition of matrix M, refer to appendix for more info about $T(M)$.

Same would apply to the biases:

$$\frac{\partial C_t}{\partial b_i^L} = \frac{\partial z_i^L}{\partial b_i^L} \cdot \frac{\partial a_i^L}{\partial z_i^L} \cdot \frac{\partial C_t}{\partial a_i^L} = (1)\sigma'\big(z_i^L\big)2\big(a_i^L - y_i\big) = 2\sigma'\big(z_i^L\big)\big(a_i^L - y_i\big)$$

$$\frac{\partial C_t}{\partial b^L} = \begin{bmatrix} 2\sigma'(z_0^L)(a_0^L - y_0) \\ 2\sigma'(z_1^L)(a_1^L - y_1) \\ \vdots \\ 2\sigma'(z_n^L)(a_n^L - y_n) \end{bmatrix} \qquad \frac{\partial C_t}{\partial b^L} = 2\sigma'(z^L)(a^l - y)$$

To <u>propagate backwards</u> – The derivative of the cost function with respect to ***one*** weight in some layer j^{th} before the last layer:

$$\frac{\partial C_t}{\partial w_{i,k}^{L-j}} = \frac{\partial z_i^{L-j}}{\partial w_{i,k}^{L-j}} \cdot \frac{\partial a_i^{L-j}}{\partial z_i^{L-j}} \cdot \frac{\partial C_t}{\partial a_i^{L-j}}$$

Similar to the last layer:

$$\frac{\partial z_i^{L-j}}{\partial w_{i,k}^{L-j}} = a_k^{L-j-1} \qquad \frac{\partial a_i^{L-j}}{\partial z_i^{L-j}} = \sigma'(z_i^{L-j})$$

$$\frac{\partial C_t}{\partial w_{i,k}^{L-j}} = \big(a_k^{L-j-1}\big) \cdot \sigma'(z_i^{L-j}) \cdot \frac{\partial C_t}{\partial a_i^{L-j}}$$

Expanding this from the single weight gradient to the matrix of weight gradients:

$$\frac{\partial C_t}{\partial w^{L-j}} = \begin{bmatrix} (a_0^{L-j-1})\sigma'\big(z_0^{L-j}\big)\frac{\partial C_t}{\partial a_0^{L-j}} & (a_1^{L-j-1})\sigma'\big(z_0^{L-j}\big)\frac{\partial C_t}{\partial a_0^{L-j}} & \cdots & (a_k^{L-j-1})\sigma'\big(z_0^{L-j}\big)\frac{\partial C_t}{\partial a_0^{L-j}} \\ (a_0^{L-j-1})\sigma'\big(z_1^{L-j}\big)\frac{\partial C_t}{\partial a_1^{L-j}} & (a_1^{L-j-1})\sigma'\big(z_1^{L-j}\big)\frac{\partial C_t}{\partial a_1^{L-j}} & \dots & (a_k^{L-j-1})\sigma'\big(z_1^{L-j}\big)\frac{\partial C_t}{\partial a_1^{L-j}} \\ \vdots & \vdots & \ddots & \vdots \\ (a_0^{L-j-1})\sigma'\big(z_i^{L-j}\big)\frac{\partial C_t}{\partial a_i^{L-j}} & (a_1^{L-j-1})\sigma'\big(z_i^{L-j}\big)\frac{\partial C_t}{\partial a_i^{L-j}} & \dots & (a_k^{L-j-1})\sigma'\big(z_i^{L-j}\big)\frac{\partial C_t}{\partial a_i^{L-j}} \end{bmatrix}$$

Where i is the number of neurons in the layer $(L - j)$, and k is the number of neurons in the layer $(L - j - 1)$

$$\frac{\partial C_t}{\partial w^{L-j}} = \left(\sigma'\big(z^{L-j}\big)\big(\frac{\partial C_t}{\partial a^{L-j}}\big)\right) \bullet T(a^{L-j-1})$$

The biases would follow suite:

$$\frac{\partial C_t}{\partial b_i^{L-j}} = \frac{\partial z_i^{L-j}}{\partial b_i^{L-j}} \cdot \frac{\partial a_i^{L-j}}{\partial z_i^{L-j}} \cdot \frac{\partial C_t}{\partial a_i^{L-j}} = \sigma'\left(z_i^{L-j}\right)\frac{\partial C_t}{\partial a_i^{L-j}}$$

$$\frac{\partial C_t}{\partial b^{L-j}} = \begin{bmatrix} \sigma'\left(z_0^{L-j}\right)\frac{\partial C_t}{\partial a_0^{L-j}} \\ \sigma'\left(z_1^{L-j}\right)\frac{\partial C_t}{\partial a_1^{L-j}} \\ \vdots \\ \sigma'\left(z_n^{L-j}\right)\frac{\partial C_t}{\partial a_n^{L-j}} \end{bmatrix} = \sigma'\left(z^{L-j}\right)\frac{\partial C_t}{\partial a^{L-j}}$$

Again, both gradients have the $\frac{\partial C_t}{\partial a_i^{L-j}}$ term which can be thought of as how the neuron activations in a previous layer impact the cost:

$$\frac{\partial C_t}{\partial a_k^{L-j}} = \left(\sum_{i=0}^{n} \frac{\partial z_i^{L-j}}{\partial a_k^{L-j-1}} \cdot \frac{\partial a_i^{L-j}}{\partial z_i^{L-j}}\right)\left(\sum_{i=0}^{n} \frac{\partial C_t}{\partial a_i^{L-j+1}}\right) = \sum_{i=0}^{n} w_{i,k}^{L-j}\sigma'\left(z_i^{L-h}\right)\left(\sum_{i=0}^{n} \frac{\partial C_t}{\partial a_i^{L-j+1}}\right)$$

Where n is the total number of neurons in layer $(L-j)$ or $(L-j+1)$.

Which is recursively defined until the last layer (much like the simplified model):

$$\frac{\partial C_t}{\partial a_k^{L-1}} = \sum_{i=0}^{n} \frac{\partial z_i^{L}}{\partial a_k^{L-1}} \cdot \frac{\partial a_i^{L}}{\partial z_i^{L}} \cdot \frac{\partial C_t}{\partial a_i^{L}} = \sum_{i=0}^{n} w_{i,k}^{L}\sigma'\left(z_i^{L}\right)2\left(a_i^{L} - y_i\right)$$

Where n is the total number of neurons in the last layer, L.

The summation in the formulas would be used because the a_k^{L-1} value impacts the cost function through each of the weights and each of the neurons they are connected to.. This can be seen in figure ten, neuron a_1^{L-1} is connected to two weights that connect to the output neurons which are then used to calculate the cost. a_1^{L-1} impact on the cost is proportional to the outputs and the weights that connected them, and therefore must be summed to be accounted for.

Example of this is:

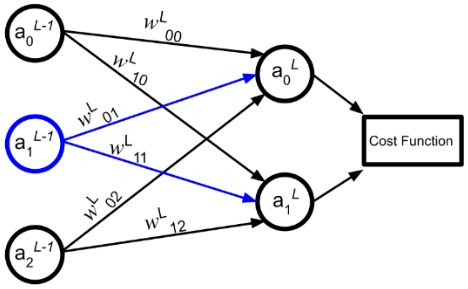


Figure 10 - Last Two Layers of Example Neural Network (Showing how previous layers impact the last)

In summary: The formula for the partial gradient matrices for the last weights and biases matrices:

$$\frac{\partial C_t}{\partial w^L} = (2\left(a^l - y\right)\sigma'(z^L)) \bullet T((a^{L-1}))$$

$$\frac{\partial C_t}{\partial b^L} = 2\sigma'(z^L)(a^l - y)$$

And using Backwards Propagation the formulas to the partial gradient matrices for weights and biases matrices for previous layers:

$$\frac{\partial C_t}{\partial w^{L-j}} = \left(\sigma'\left(z^{L-j}\right)\left(\frac{\partial C_t}{\partial a^{L-j}}\right)\right) \bullet T(a^{L-j-1})$$

$$\frac{\partial C_t}{\partial b^{L-j}} = \sigma'\left(z^{L-j}\right)\frac{\partial C_t}{\partial a^{L-j}}$$

Where $\frac{\partial C_t}{\partial a_k^{L-j}}$ is recursively defined until the last layer:

$$\frac{\partial C_t}{\partial a_k^{L-j}} = \sum_{i=0}^{n} w_{i,k}^{L-j}\sigma'\left(z_i^{L-h}\right)\left(\sum_{i=0}^{n}\frac{\partial C_t}{\partial a_i^{L-j+1}}\right)$$

$$\frac{\partial C_t}{\partial a_k^{L-1}} = \sum_{i=0}^{n} w_{i,k}^{L}\sigma'\left(z_i^L\right)2\left(a_i^L - y_i\right)$$

Each of the previous equations were in reference to the one training example (t). To find the overall cost of the network every gradient for each training example must be averaged.

How to Use the Gradients to Teach the Neural Network:

To bring back to the initial problem at hand, "how to tune each of the weights and bias to make the neural network produce the correct answer?", the solution is using the gradients. Mathematically, the complete gradient (containing the average of all training examples) should be used to change the weights and biases however it is not practical to code and often requires multiple training iterations to perfect. To get around this issue, the neural network is trained in batches, a subgroup of the training examples is selected and the gradient for it, is calculated. To ensure no subgroup causes to drastic of a change each gradient is multiplied by a **learning rate** which scales how fast the neural network learns (how much the weights and biases are changed). This final subgroup gradient matrix is then scaled by the learning rate and subtracted (because gradients point toward steepest increase in cost) to each of the weights and biases to tune them toward the desired output. The training process is often repeated for all subgroups until the accuracy of the neural network is sufficient.

Tackling the Handwritten Number Problem:

Using my programing abilities, I wrote a java library to perform the neural network math and utilized this library to tackle the handwritten number problem.

The inputs to this neural network will be a matrix with one column and 784 rows, one for each pixel in the 28x28 pixel handwritten number images. There will be two hidden layers the first with 32 neurons the second with 16 neurons. The output layer will consist of ten neurons one for each possible result.

The value of the inputs will be scaled between 0 and 1, where 0 is a white pixel and darker pixels scale until black is a 1. The outputs will each represent how sure the neural network is for each possible answer. For example, if the neural network was completely sure that handwritten number was a 5, its output would be (0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,) indexed from 0.

To train this neural network the dataset used is called the MNIST Data Base[7], it's a collection of 70 000 labeled hand images of handwritten numbers. 60 000 of which will be used for training the neural network and the other 10 000 will be reserved for testing the neural networks accuracy when given new data.

There are a variety of configurations for the neural network that could cause it to function better or worse. Using different learning rates, batch sizes and number of neurons at each hidden layer all can impact the effectiveness of the neural network. To explore the different possibilities an array of testing was preformed, comparing how the learning rate and batch size impact the trained neural network given a constant 60 000 training images, and neural network architecture (two hidden layer 32, 16).

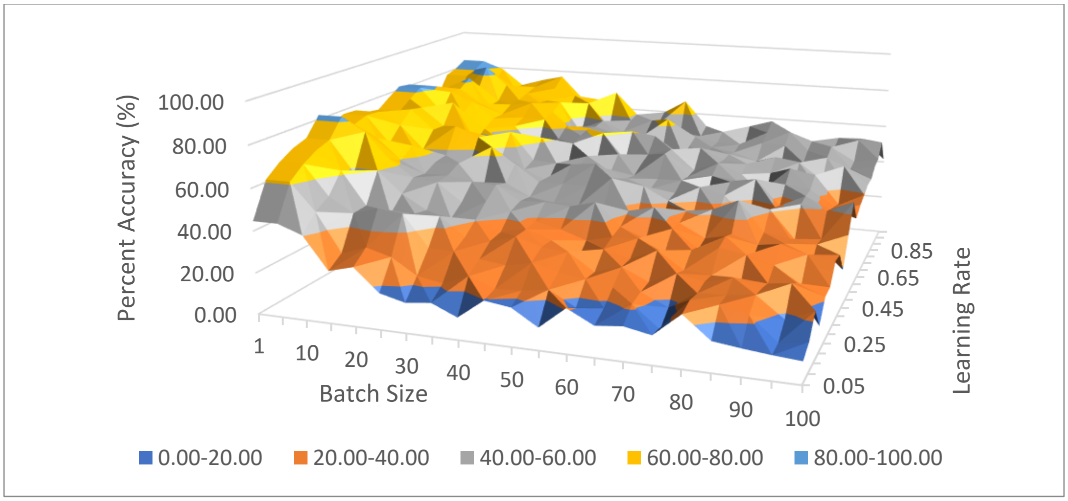


Figure 11 - Percent Accuracy of a Set Neural Network as a Function of Applied Learning Rate and a Variety of Batch Sizes | Small Learning Rates (See Appendix 4 for Data)

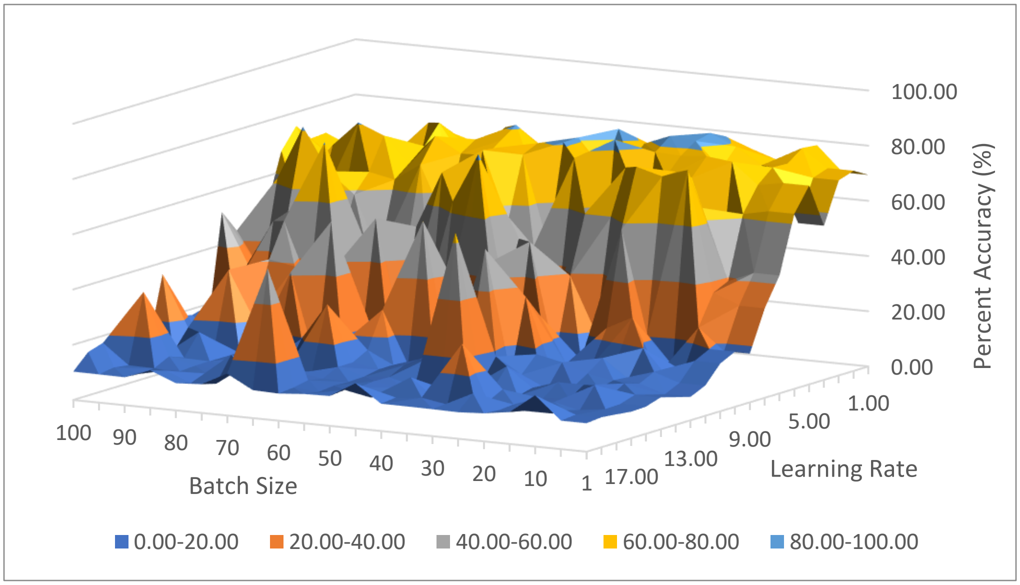


Figure 12 - Percent Accuracy of a Set Neural Network as a Function of Applied Learning Rate and a Variety of Batch Sizes | Large Learning Rates (See Appendix 4 for Data)

After all the training tests had been completed, it can be determined that there is a trend between these variables. It seems that when both learning rate and batch size are relatively low, the trained neural network is generally more accurate (left side of figure 11, batch size: 1, learning rate 0.15). Also, when the batch size and learning rate were both generally large, again improved accuracy was found, until the extremes (as seen in figure 12).

The best result found was using batches of 40 images, with a learning rate of 4, reaching an accuracy of 83.50% when trained with 60 000 images. However, when giving it more training by iterating through the 60 000 multiple times, it was found that a smaller learning rate created greater accuracy. Batch size of 20 learning rate of 0.9 reached an accuracy of 89.45% with 600 000 training examples. There is an endless quantity of experiments to preform: manipulating the number of layers and neurons in each hidden layer, far more extreme rates and batches, multiple iterations, analyzing learning curves while learning. With more computational power and time, the effect each of these manipulations on accuracy would be revealed. I am excited to continue testing independent of my IA (as it doesn't fit into scope) to continue my understanding neural networks.

Conclusion:

The complex mess of matrix multiplication and gradients calculations that is neural networks, is now, still very complex, but no longer a mess of abstract math but an intuitive system to tackle complex problems. Exploring this topic has both developed my appreciation for mathematics but also for computer science. As neural networks are used in many modern software solutions, they are increasingly becoming a part of every one's lives. To deeply understand how google predicts what your google search will be, or why a social media app gave you this specific post, or countless other examples, allows me to value mathematics through every digital interaction in my life.

To summarize: Attempting to calculate the solution to problems that at the surface seem un-solvable, neural networks can step in and tackle the problem. With a complex web of connection, loosely mimicking how neurons in biological brains function, tasks that used to seem impossible for computers can now be attempted. By training a neural network to best utilize this web of connections, the correct solution can be calculated. For the handwritten number problem, enough training, allows a neural network to compute the surprisingly complex world that is recognizing numbers that we take for granted.

The world of neural networks is vast and ever expanding, with endless new problems to be solved and equally endless ways to use neural networks to find these solutions. The neural network explored in this IA is fairly rudimentary, as this is relatively new technology, modifications and experiments for improvements are currently underway. This exploration can now act as a springboard for my curiosity to further examine different ways neural networks work.

Appendix:

1 - Work for finding the derivative of the sigmoid function:

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

$$\sigma'(x) = \frac{d}{dx}(1 + e^{-x})^{-1}$$

$$\sigma'(x) = -1(1 + e^{-x})^{-2}(-e^{-x})$$

$$\sigma'(x) = \frac{e^{-x}}{(1 + e^{-x})^2}$$

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

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

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

2 - Matrix Multiplication:

$$M^1 = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \qquad M^2 = \begin{bmatrix} g & h \\ i & j \\ k & l \end{bmatrix}$$

Where the rows of the first one must equal the columns of the second, and the resultant matrix has the number rows of the first matrix and the number columns of the second[9]. The commutative property is not held ($M_1 \bullet M_2 \neq M_2 \bullet M_1$).

$$M^1 \bullet M^2 = \begin{bmatrix} ag + bi + kc & ah + bj + lc \\ gd + ie + kf & dh + ej + fl \end{bmatrix}$$

Where the value at the index along the row of the first is multiplied by the value at the same index along the column of the second is found then added up. This value is placed in the resultant matrix in the row number, being multiplied of the first matrix and in the column number, being multiplied in the second matrix.

3 - Matrix Transposition:

$$M^1 = \begin{bmatrix} M^1_{0,0} & M^1_{0,1} & M^1_{0,2} \\ M^1_{1,0} & M^1_{1,1} & M^1_{1,2} \end{bmatrix}$$

The transposed version of a matrix switches the number of rows and columns in that matrix. Each value within that matrix is moved from its original spot at row, column to the new spot row# = column# and column# = row#.

$$T(M^1) = \begin{bmatrix} M^1_{0,0} & M^1_{1,0} \\ M^1_{0,1} & M^1_{1,1} \\ M^1_{0,2} & M^1_{1,2} \end{bmatrix}$$

4 – Figure 11 Data Tables | Percent Accuracy Across Batch Size and Learning Rate (%):

		Learning Rate									
		0.05	0.10	0.15	0.20	0.25	0.30	0.35	0.40	0.45	0.50
Batch Size	5	44.62	61.17	66.76	70.79	73.15	82.10	75.95	78.03	77.54	74.42
	10	44.62	61.17	66.76	70.79	73.15	82.10	75.95	78.03	77.54	74.42
	15	40.41	48.18	52.32	61.85	57.90	75.50	69.92	64.72	71.64	77.47
	20	24.97	29.79	45.42	45.91	60.23	62.91	59.60	64.41	60.05	67.68
	25	28.11	33.76	48.87	43.03	49.74	54.88	53.80	56.97	57.76	70.29
	30	17.31	30.50	47.54	44.37	42.79	53.53	54.15	57.32	52.50	65.48
	35	14.31	23.02	32.60	44.34	39.85	44.68	51.83	43.34	63.93	55.64
	40	15.73	20.90	32.22	35.40	40.55	46.29	51.40	49.39	56.23	50.67
	45	10.46	25.69	29.88	34.88	35.74	42.51	44.61	43.48	48.08	45.33
	50	19.72	21.19	28.45	38.29	38.28	38.77	45.16	41.78	48.23	55.23
	55	18.06	21.77	30.91	34.70	34.28	36.15	44.10	41.48	48.58	56.65
	60	10.47	23.62	27.70	33.63	25.60	31.96	47.25	40.83	53.10	50.14
	65	20.77	16.93	25.98	25.40	32.62	35.07	36.51	41.05	30.17	47.28
	70	14.46	18.05	23.56	29.18	29.22	35.62	31.70	43.98	30.29	41.00
	75	15.75	21.51	22.90	23.56	34.62	29.83	38.03	37.41	40.25	33.48
	80	13.38	11.50	17.31	28.95	26.69	32.55	29.40	31.13	41.97	36.22
	85	24.13	26.32	25.20	26.56	25.67	34.86	36.81	34.49	47.64	28.83
	90	14.04	25.66	25.69	29.90	19.22	32.84	26.00	30.45	33.01	41.15
	95	12.47	22.61	27.40	26.42	24.65	29.35	35.68	32.31	31.21	27.37
	100	11.20	12.04	18.03	29.99	26.32	21.52	31.98	24.16	29.72	28.82

		Learning Rate									
		0.55	0.60	0.65	0.70	0.75	0.80	0.85	0.90	0.95	1.00
Batch Size	5	73.42	77.79	81.73	80.77	75.83	74.85	68.62	78.28	80.15	83.39
	10	73.42	77.79	81.73	80.77	75.83	74.85	68.62	78.28	80.15	83.39
	15	76.05	70.57	69.21	78.02	77.73	81.70	73.08	84.08	73.55	79.22
	20	67.20	77.31	69.12	73.51	70.08	66.52	67.52	74.34	70.54	70.34
	25	65.80	57.18	61.67	64.97	64.90	68.29	68.64	70.12	71.87	74.00
	30	61.91	59.58	56.07	61.59	64.10	69.80	67.80	66.86	71.88	76.83
	35	66.64	57.11	51.32	47.69	63.77	65.70	68.84	61.64	68.36	64.76
	40	52.90	62.01	51.94	56.23	60.71	55.83	52.14	66.91	63.81	64.35
	45	52.10	40.24	57.29	59.04	60.28	59.04	59.81	54.11	71.66	61.40
	50	51.61	54.13	44.98	61.31	58.48	55.77	54.78	58.67	59.73	50.27
	55	51.66	46.71	53.36	47.44	60.59	54.25	53.96	59.46	54.84	59.53
	60	47.22	45.78	49.04	51.51	58.98	43.82	63.39	54.31	46.10	66.72
	65	42.93	45.08	55.64	55.29	49.18	45.82	49.28	51.87	49.68	53.14
	70	41.84	45.46	44.44	45.03	54.35	47.87	49.43	49.52	53.14	45.00
	75	45.94	33.60	45.90	49.80	48.79	49.18	47.56	49.52	45.36	54.80
	80	42.82	46.78	44.44	44.99	47.65	39.41	43.65	51.17	54.49	58.59
	85	30.10	49.21	39.65	31.23	43.77	49.62	37.15	50.60	55.80	52.17
	90	39.03	36.64	44.54	50.01	38.30	47.64	44.77	48.89	51.47	38.18
	95	41.79	42.74	42.79	47.67	37.42	39.01	48.61	51.73	49.13	51.77
	100	39.41	37.20	35.47	33.42	50.47	36.31	43.61	44.83	53.81	52.27

4 – Figure 12 Data Tables | Percent Accuracy Across Batch Size and Learning Rate (%):

		Learning Rate									
		1.00	2.00	3.00	4.00	5.00	6.00	7.00	8.00	9.00	10.00
Batch Size	5	69.40	72.53	70.66	55.84	71.96	63.37	42.70	32.38	17.61	19.95
	10	78.32	78.93	74.88	75.12	70.34	61.42	38.78	36.18	25.21	11.06
	15	69.85	73.70	73.34	54.95	77.87	28.02	64.32	79.69	10.43	8.92
	20	71.62	76.87	79.18	72.73	78.89	72.80	76.51	76.45	20.94	28.63
	25	65.67	80.38	83.16	81.87	78.15	72.25	71.59	79.02	58.00	23.87
	30	60.05	74.38	73.31	82.79	78.54	39.20	76.18	59.78	19.64	11.36
	35	64.30	75.69	75.66	79.08	73.96	83.72	19.30	60.22	39.30	11.38
	40	60.40	74.36	74.56	83.50	70.27	76.86	77.74	30.92	19.56	52.29
	45	59.45	70.69	73.18	72.41	81.05	80.93	44.85	46.43	50.85	32.69
	50	46.25	71.24	73.79	60.93	66.79	79.68	77.14	54.67	64.36	82.18
	55	55.05	65.04	68.47	73.80	72.66	63.75	78.72	79.50	36.20	64.14
	60	56.74	65.04	72.81	81.21	77.39	73.42	43.01	40.00	72.10	10.32
	65	42.56	67.79	77.84	72.90	76.66	73.13	49.45	21.99	51.52	15.26
	70	48.99	67.09	67.19	65.98	59.62	78.51	69.73	38.19	25.66	55.17
	75	56.01	64.62	73.72	79.22	72.36	67.83	63.45	38.23	44.38	38.17
	80	49.16	59.05	76.74	55.31	56.75	76.87	65.34	58.73	42.34	80.87
	85	47.49	65.92	58.17	64.04	72.65	80.64	53.82	66.24	26.66	56.12
	90	55.50	62.47	67.47	73.69	67.35	71.54	49.77	81.41	52.37	50.45
	95	45.18	60.98	59.26	61.35	57.36	49.91	56.13	61.80	29.19	34.13
	100	36.97	57.39	69.32	68.87	75.00	67.53	48.78	18.69	43.59	51.88

		Learning Rate									
		11.00	12.00	13.00	14.00	15.00	16.00	17.00	18.00	19.00	20.00
Batch Size	5	17.25	10.96	8.45	9.97	11.54	9.80	9.58	10.23	10.83	10.28
	10	17.25	10.96	8.45	9.97	11.54	9.80	9.58	10.23	10.83	10.28
	15	10.10	14.04	11.35	13.05	9.74	15.74	8.21	9.81	10.36	18.53
	20	20.06	10.43	11.66	10.28	13.26	9.74	10.31	10.10	10.32	11.50
	25	15.32	17.18	13.25	9.74	13.32	8.96	9.80	10.09	9.58	10.13
	30	12.03	10.29	16.28	8.33	10.32	8.92	13.01	10.18	14.81	9.58
	35	11.60	11.08	18.27	12.52	17.18	10.10	9.59	30.31	9.80	9.74
	40	27.93	36.57	9.58	18.15	9.80	9.55	10.32	8.92	11.37	9.82
	45	10.04	12.71	54.57	22.16	63.75	9.74	9.58	16.13	10.47	9.85
	50	45.50	19.92	17.44	9.74	17.53	8.93	9.80	8.92	9.58	16.33
	55	30.40	10.96	9.93	14.38	13.82	11.13	11.33	10.88	11.35	10.78
	60	15.77	60.32	9.82	11.35	11.35	10.28	10.08	10.32	9.74	10.57
	65	10.32	22.49	28.86	15.84	12.94	12.93	10.28	11.35	10.28	9.80
	70	20.09	19.54	18.28	17.21	33.12	10.32	11.35	10.10	52.57	10.10
	75	10.17	10.18	58.97	8.92	19.64	10.09	10.10	9.74	15.51	15.12
	80	38.32	12.28	36.94	41.30	10.09	10.28	9.74	17.79	8.93	10.37
	85	30.47	39.07	10.50	14.11	11.29	9.75	9.82	10.11	8.18	9.80
	90	53.06	40.76	20.42	10.11	10.29	10.35	10.28	9.91	11.79	13.51
	95	12.78	9.82	37.66	9.80	18.35	10.73	12.76	10.10	9.68	10.09
	100	25.70	10.09	10.90	17.46	10.14	9.64	35.19	10.28	11.04	10.19

Bibliography:

[1]: Data Bricks. 2022. Neural1. https://www.databricks.com/wp-content/uploads/2019/02/neural1.jpg (image). retrieved October 3, 2022

[2]: Wikipedia Authors. October 2, 2022. Backpropagation. https://en.wikipedia.org/wiki/Backpropagation. retrieved October 3, 2022

[3]: Wikipedia Authors. September 8, 2022. Feedforward neural network. https://en.wikipedia.org/wiki/Feedforward_neural_network. retrieved October 3, 2022

[4]: Wikipedia Authors. September 12, 2022. Activation Function. https://en.wikipedia.org/wiki/Activation_function. retrieved October 3, 2022

[5]: Wikipedia Authors. September 13, 2022. Sigmoid Function. https://en.wikipedia.org/wiki/Sigmoid_function. retrieved October 3, 2022

[6]: Wikipedia Authors. September 8, 2022. Gradient. https://en.wikipedia.org/wiki/Gradient. retrieved October 3, 2022

[7]: Yann LeCun, Corinna Cortes, Christopher J.C. Burges. 1998. THE MNIST DATABASE. http://yann.lecun.com/exdb/mnist/. retrieved August 10, 2020

[8]: Jason Brownlee. July 27, 2018. How to Configure the Number of Layers and Nodes in a Neural Network. https://machinelearningmastery.com/how-to-configure-the-number-of-layers-and-nodes-in-a-neural-network/. retrieved October 3, 2022

[9]: Andre Staltz. March 14, 2019. Matrix Multiplication. http://matrixmultiplication.xyz/

[10]: Wikipedia Authors. September 22, 2022. Transpose. https://en.wikipedia.org/wiki/Transpose. retrieved October 3, 2022

[11]: 3Blue1Brown. August 1, 2018. Neural Networks. https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi. Accessed October 3, 2022