How can mathematics be applied to improve the performance of neural networks?

Word Count: 3995
IB Mathematics Extended Essay

Contents

1 Introduction

In recent years, there have been many incredible leaps in the capabilities of computers, from self-driving cars to facial recognition. One of the many powerful tools used by computer scientists to allow computers to master such complex tasks is the neural network. This essay will explore how mathematics can be applied to improve the performance of these neural networks. This is done first by exploring the structure of neural networks, determining a measure of the performance of a neural network, and then developing a method for strategically altering the properties of a neural network to improve this performance measure. This will involve the use of concepts from calculus and linear algebra. As neural networks and machine learning become increasingly prevalent in society, it is important to understand the mathematics underlying these tools as it provides insight into why they are so effective as well as where their limitations may lie.

2 Basics of a Neural Network

At its core, neural network is a function which takes several values as input, performs a calculation using a large number of adjustable parameters, and then gives several values as output. A neural network can be visualized as a multilayered structure of interconnected nodes (Seth). There are many different types of neural networks, however this essay will focus on the simplest kind, a fully connected neural network. In this type of neural network, every node in one layer is connected to every other node in the next layer by a weighted connection (Isaksson). The weight of this connection indicates the degree to which the node is affected by the value of the node which comes before it. Finally, each node has the property of a bias, a constant which changes the value of the node independently of any other node (Seth). It is from these weights and biases that the value of each node can be determined. The following figure gives a visual example of a neural network with three layers, using circles to represent the nodes and lines of varying thicknesses to represent the weighted connections.

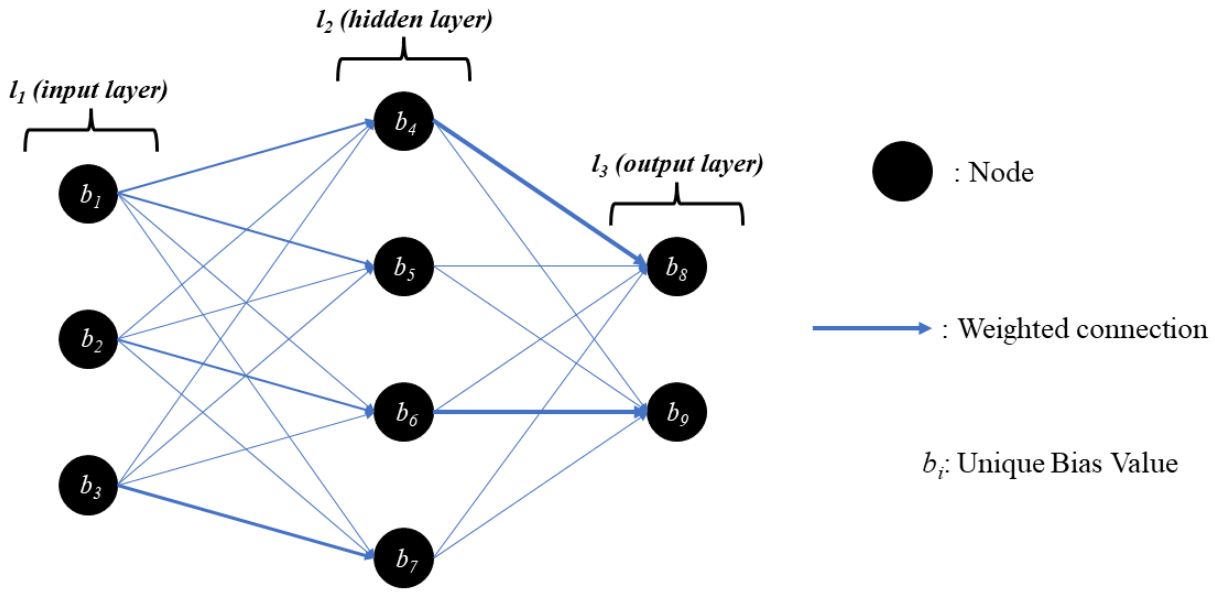

Figure 1: An example of a visual representation of a neural network

As indicated by the figure, the first layer is called the "input layer", the last layer is called the "output layer", and any intermediate layers are called hidden layers. To start the calculation, the nodes in the input layer have their values initialized to whatever values we want the neural network to take as input. Then, the connecting weights and relevant biases are used to determine the values of every node in the next layer. This allows for the values in the next layer to be determined, and so on until the values of the nodes in the output layer are found. These values are the output of the neural network.

More specifically, the value of each non-input node is given by the sum of each previous node's value multiplied by the weight of the connection plus the node's bias (Seth). For example, consider some layer l_x with n nodes. These n nodes have the known values $\{u_1, u_2, \ldots, u_n\}$, where u_i is the value of the ith node in layer l_x. Each of these nodes are connected to some node a in the layer l_{x+1}, by connections with weights $\{w_1, w_2, \ldots, w_n\}$, where w_i is the weight of the connection between node a and the ith node in layer l_x. As well, let node a's bias be given by b_a. As such, the unknown value of node a, v_a, is given by the following expression:

$$v_a = w_1u_1 + w_2u_2 + \ldots + w_nu_n + b_a$$

If we define a vector containing all the connection weights and another containing all the values of the nodes in layer l_x, the expression for v_a can be simplified as the dot product of these two vectors. As such, if $\vec{w} = (w_1, w_2, \ldots, w_n)$ and $\vec{u} = (u_1, u_2, \ldots, u_n)$, then by the properties of the dot product,

$$\vec{w} \cdot \vec{u} = w_1u_1 + w_2u_2 + \ldots + w_nu_n$$
$$\therefore v_a = w_1u_1 + w_2u_2 + \ldots + w_nu_n + b_a = \vec{w} \cdot \vec{u} + b_a$$

However, as v_a could be any real number, this value is typically put through an activation function which alters the value to give it a well-defined range. While many are used, a very common one is the sigmoid function (Sharma). The sigmoid function, denoted by $\sigma(x)$, is defined as follows:

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

As is shown by the following graph of the sigmoid function, it compresses any input value into its range of $0 < \sigma(x) < 1$.

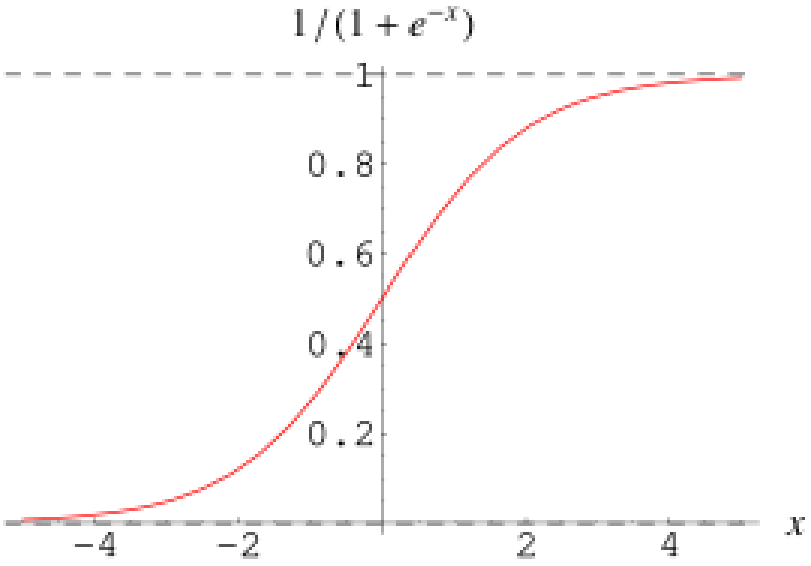

Figure 2: Graph of the sigmoid function (Weisstein).

As such, ultimately the value of node a is given by the equation $v_a = \sigma(\vec{w} \cdot \vec{u} + b_a)$, restricting it to a value between 0 and 1. This calculation is done for every node in the layer l_{x+1} to determine all of their values.

To demonstrate the process and provide a sample of the mathematics that have been explored thus far, we will calculate the output values for the simple segment of a neural network shown in the figure below.

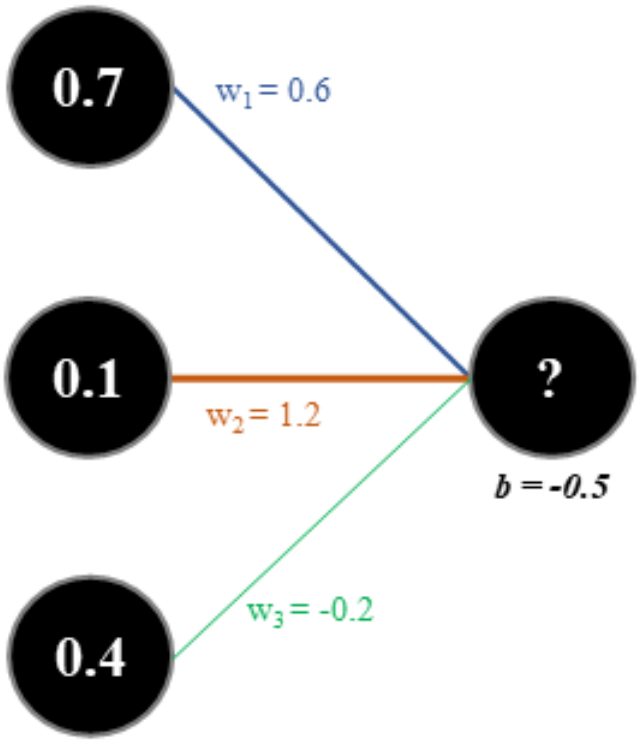

Figure 3: Diagram of a sample neural network showing its structure and the nodes, weights, and biases relevant to the first calculation.

First, the known input values are used to construct the vector $\vec{u} = (0.7, 0.1, 0.4)$. We then set b as the bias of the node in the second layer, giving $b = -0.5$. Finally, the connecting weights are used to construct the vector $\vec{w} = (0.6, 1.2, -0.2)$. We now have all the necessary components to calculate the values of the node in the second layer, which we will represent with v. Using the equation from before,

$$
\begin{aligned}
v &= \sigma(\vec{w} \cdot \vec{u} + b) \\
v &= \sigma\left(\begin{bmatrix} 0.6 \\ 1.2 \\ -0.2 \end{bmatrix} \cdot \begin{bmatrix} 0.7 \\ 0.1 \\ 0.4 \end{bmatrix} + (-0.5)\right) \\
v &= \sigma\Big(\big(0.6 \cdot 0.7 + 1.2 \cdot 0.1 + (-0.2) \cdot 0.4\big) - 0.5\Big) \\
v &= \sigma(-0.04) \\
v &\approx 0.490
\end{aligned}
$$

Having found the value of the unknown node, the same process could be used to calculate the values of all the other nodes in that second layer. These values would then allow us to calculate the values of all the nodes in the next layer. This process could be repeated until the output layer values are found. For this example, weights and biases were simply chosen arbitrarily. However, in the following sections, we will explore the mathematics that allow us to find weights and biases which make the neural network produce useful output.

3 The Error Function

To optimize a neural network to give the desired output for a given input, the quantity which we are trying to optimize must first be defined. To do this, we must choose a measure of how accurate the current neural network is at producing the output which best suits the given task. This can be done by comparing its current performance to an ideal performance using training cases. A training case is defined as an input with a known desired output. For example, suppose we are trying to create a neural network to differentiate between images of cats and dogs. A possible training case for this task would be an image of a cat labelled "cat". Here the image is the input and the label of "cat" is the desired output. Suppose the neural network had two output nodes, one for the degree to which it thinks the image is a cat and another for the degree to which it thinks the image is a dog. For our training case, the ideal output would be a "1" as the cat output value and a "0" as the dog output value, as these are the maximum and minimum node values as a result of the sigmoid function. However, initially the neural network would output two very different values, as its parameters have yet to be optimized for the task.

There are many ways to compare the desired output of the neural network and its actual output. However, this essay will only consider the sum of squares method, which is the most common. This method involves taking the sum of the squares of the differences between each output node's ideal output and its actual output as the error measurement for a given training case (Christiansen). For example, suppose that for a particular training case a neural network returns the output values $(0.3, 0.5, 0.6, 0.1)$, while the desired output values are $(1, 0, 0, 0)$. In this example, the error for that training case, E, would be given by the following:

$$\begin{aligned} E &= (0.3-1)^2 + (0.5-0)^2 + (0.6-0)^2 + (0.1-0)^2 \\ E &= 0.49 + 0.25 + 0.36 + 0.01 \\ E &= 1.11 \end{aligned}$$

To include the most data in its calculation, the error function for a neural network is given by the mean of the error for all training cases. The worse the neural network is at performing the task, the greater the difference between the actual output values and the desired output values will be. This in turn results in a greater error function value, making it a measure of how poor the performance of a neural network is.

It is important to recognize that for a given input, the output of a neural network is determined solely by its weights and biases. As such, for a given set of training cases, the error function can be treated as a function which takes as input the weights and biases of the neural network and outputs a measure of how poor the neural network is currently performing. Thus, to improve the performance of a neural network, we can minimize the output of the error function for a given set of training data by adjusting the neural network's weights and biases. The following section discusses the method by which we can accomplish this: gradient descent.

4 Gradient Descent

Gradient descent is a method of approximating a local minimum of a complicated function such as the error function. It begins by choosing an arbitrary point on the function (IBM). For a neural network, this corresponds to initially assigning random values to all the weights and biases (Hardesty). The next step is to determine the gradient vector of the function at that particular point. The gradient vector has the direction of the fastest increase of a function at that point and a magnitude which is given by the rate of fastest increase at that point (Paolucci). For example, the gradient vector of a hill would point up the hill.

Since the gradient vector points in the direction of fastest increase, the negative of the gradient vector points in the direction of fastest decrease for the function at that point (Paolucci). So during gradient descent, we adjust the inputs of our function in the direction of the negative gradient vector, thus taking the quickest step toward some local minimum of the function. The process can then be repeated until the local minimum is sufficiently well approximated (IBM).

It is important to note that gradient descent comes with limitations. Importantly, it only allows us to find an approximation of a local minimum of our function, not the global minimum. As such, using this method it is possible to find a local minimum which is much higher than the true minimum of the function, which would result in poor optimization (Paolucci). Additionally, the gradient vector requires much more calculation the more inputs are considered. As such, typically a variation of this method is used which involves only using a subset of the training data for each step in the gradient descent process, making it less accurate to the true gradient of the whole data set (IBM). These limitations provide insight both into the importance of computing power in developing a neural network, as well as the possibility of poor performance if the initial weights and biases are chosen carelessly or happen to be close to a relatively large local minimum.

To perform gradient descent, we first take the gradient of this error function. We then adjust the weights and biases according to this gradient. Repeating these steps sufficiently many times causes the weights and biases to adjust such that the error function approaches some local minimum. This improves the performance of the neural network, as the lower this error function is, the better the network is at giving the desired output for a given input. The next section will explore how the gradient vector of the error function can be calculated.

5 Finding the Gradient Vector Using Backpropagation

Backpropagation is an algorithm which is used to calculate the gradient of an error function (Johnson). As such, the following section will explore the mathematics used in the backpropagation algorithm for our established model of a neural network.

First, we must define the gradient of a function. For some function $f(x_1, x_2, \ldots, x_n)$, the gradient of that function is represented by $\nabla f(x_1, x_2, \ldots, x_n)$, and is typically defined as

follows:

$$\nabla f(x_1, x_2, \ldots x_n) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right)$$

The notation $\frac{\partial y}{\partial x}$ denotes the partial derivative of y with respect to x. The partial derivative is very similar to the standard derivative for single-variable functions, only defined for multivariable functions. The process of computing a partial derivative with respect to a particular variable simply involves treating all other input variables like constants and then finding the derivative as one normally would with a single-variable function.

The gradient vector for the error function can be defined in the same way. From before, the error function is a function of all the weights and biases in a neural network. For a neural network with α weights $\{w_1, w_2, \ldots, w_\alpha\}$ and β biases $\{b_1, b_2, \ldots, b_\beta\}$, the gradient vector of the neural network's error function is given by,

$$\nabla E(w_1, w_2, \ldots, w_\alpha, b_1, b_2, \ldots, b_\beta) = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_\alpha}, \frac{\partial E}{\partial b_1}, \frac{\partial E}{\partial b_2}, \ldots, \frac{\partial E}{\partial b_\beta}\right)$$

So now the problem becomes calculating each of these partial derivatives of the error function with respect to each weight and bias. First, we recall how the error function was defined as the average of all the individual training case error values, or $E(w_1, w_2, \ldots, w_\alpha, b_1, b_2, \ldots b_\beta) = \frac{1}{n}(E_1 + E_2 + \ldots + E_n)$, where n is the number of training cases and E_i is the error value of the ith training case. We will begin by considering finding the partial derivative of the error function with respect to some weight/bias p. Initially, the process of computing $\frac{\partial E}{\partial p}$ is the same regardless of whether p represents a weight or a bias. As such, the two cases will not be distinguished until later in the essay. We begin by taking the partial derivative of the error function with respect to p as follows,

$$\begin{aligned} E(w_1, w_2, \ldots, w_\alpha, b_1, b_2, \ldots b_\beta) &= \frac{1}{n}(E_1 + E_2 + \ldots + E_n) \\ \frac{\partial E}{\partial p} &= \frac{\partial}{\partial p}\Big(\frac{1}{n}(E_1 + E_2 + \ldots + E_n)\Big) \\ \frac{\partial E}{\partial p} &= \frac{1}{n}\left(\frac{\partial E_1}{\partial p} + \frac{\partial E_2}{\partial p} + \ldots + \frac{\partial E_n}{\partial p}\right), \text{ since } \frac{1}{n} \text{ is a constant} \end{aligned}$$

Thus, if we could find a method of calculating the derivative of an error value with respect to some weight/bias for a single training case, we would be able to compute $\frac{\partial E}{\partial p}$ by summing these values up for each training case and dividing by the number of cases. Therefore, we will consider the training case error E_1, knowing the following process could be applied to any training case error. From before, the error for a given training case is given by $E_1 = (o_1 - d_1)^2 + (o_2 - d_2)^2 + \ldots + (o_\gamma - d_\gamma)^2$, where $(o_1, o_2, \ldots, o_\gamma)$ is the list of the neural network's output values for the training case's input and $(d_1, d_2, \ldots, d_\gamma)$ is the corresponding list of desired output values. Using this relation, we can take the partial derivative of E_1

with respect to p.

$$\frac{\partial E_1}{\partial p} = \frac{\partial}{\partial p}\Big((o_1 - d_1)^2 + (o_2 - d_2)^2 + \ldots + (o_\gamma - d_\gamma)^2\Big)$$
$$\frac{\partial E_1}{\partial p} = \frac{\partial}{\partial p}\Big((o_1 - d_1)^2\Big) + \frac{\partial}{\partial p}\Big((o_2 - d_2)^2\Big) + \ldots + \frac{\partial}{\partial p}\Big((o_\gamma - d_\gamma)^2\Big)$$

We will begin by considering only the partial derivative involving o_1 and d_1. We can apply the chain rule, which gives that $\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial x}$. Thus, if we define $y = u^2$, $u = o_1 - d_1$, and $x = p$, we can apply the chain rule as follows.

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial x}$$
$$\frac{\partial}{\partial p}(u^2) = \frac{\partial}{\partial u}(u^2) \cdot \frac{\partial}{\partial p}(u)$$
$$\frac{\partial}{\partial p}(u^2) = 2u \cdot \frac{\partial}{\partial p}(u), \text{ using the power rule}$$
$$\frac{\partial}{\partial p}\Big((o_1 - d_1)^2\Big) = 2(o_1 - d_1) \cdot \frac{\partial}{\partial p}(o_1 - d_1), \text{ by substituting } u = o_1 - d_1$$
$$\frac{\partial}{\partial p}\Big((o_1 - d_1)^2\Big) = 2(o_1 - d_1)\left(\frac{\partial o_1}{\partial p} - \frac{\partial d_1}{\partial p}\right)$$

As the desired output, d_1, is independent of any weight or bias in the neural network, $\frac{\partial d_1}{\partial p} = 0$. Therefore, the expression simplifies to the following,

$$\frac{\partial}{\partial p}\Big((o_1 - d_1)^2\Big) = 2(o_1 - d_1)\left(\frac{\partial o_1}{\partial p}\right)$$

This can of course be applied to every actual output and desired output pair. As such we can substitute this expression into our equation for $\frac{\partial E_1}{\partial p}$.

$$\frac{\partial E_1}{\partial p} = \frac{\partial}{\partial p}\Big((o_1 - d_1)^2\Big) + \frac{\partial}{\partial p}\Big((o_2 - d_2)^2\Big) + \ldots + \frac{\partial}{\partial p}\Big((o_\gamma - d_\gamma)^2\Big)$$
$$\frac{\partial E_1}{\partial p} = 2(o_1 - d_1)\frac{\partial o_1}{\partial p} + 2(o_2 - d_2)\frac{\partial o_2}{\partial p} + \ldots + 2(o_\gamma - d_\gamma)\frac{\partial o_\gamma}{\partial p}$$

Since all output values and desired output values are known, the only unknown values in the above equation for $\frac{\partial E_1}{\partial p}$ are the partial derivatives of each output with respect to p, which is either a weight or a bias. This means that if we can find $\frac{\partial o_1}{\partial p}$ for all possible values of p, we will have all the components we need to find the gradient vector of the error function. Finding this gradient vector ultimately allows us to minimize the error function and thus improve the performance of the neural network.

We will first consider the case where the relevant parameter p is a weight represented by w_z. For this case, we must consider three subcases, which are described as follows. Note that the "relevant output node" refers to the node with value o_1.

- **Case 1**: w_z is directly connected to an output node other than the relevant output node.
- **Case 2**: w_z is directly connected to the relevant output node.
- **Case 3**: w_z is not directly connected to any node in the output layer.

These three cases cover all possible values of w_z. We must be able to find the derivative $\frac{\partial o_1}{\partial w_z}$ for each of these cases. Case 1 is the simplest. Since w_z is connected to the output layer but not the relevant node, the output node's value is independent of w_z. As such, $\frac{\partial o_1}{\partial w_z} = 0$ for Case 1. The next two cases will require more work. We will begin by recalling the equation for the relevant output node's value, o_1, in terms of the connecting weights and its bias. Let $(w_1, w_2, \ldots, w_\delta)$ be the list of δ weights directly connected to the output node. As well, let $(v_1, v_2, \ldots, v_\delta)$ be the list of values of each node connected to the output node, where the node with value v_i is connected to the output node by weight w_i. Finally, let the bias of the output node be given by b. From before, o_1 is given by the following equation,

$$o_1 = \sigma(w_1 v_1 + w_2 v_2 + \cdots + w_\delta v_\delta + b)$$

We can then use this equation to find the partial derivative of the output node's value with respect to w_z as follows:

$$\frac{\partial o_1}{\partial w_z} = \frac{\partial}{\partial w_z}\Big(\sigma(w_1 v_1 + w_2 v_2 + \cdots + w_\delta v_\delta + b)\Big)$$

We once again apply the chain rule by setting $y = \sigma(u)$, $u = w_1 v_1 + w_2 v_2 + \cdots + w_\delta v_\delta + b$, and $x = w_z$.

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial x}$$
$$\frac{\partial}{\partial w_z}\big(\sigma(u)\big) = \frac{\partial}{\partial u}\big(\sigma(u)\big) \cdot \frac{\partial u}{\partial w_z}$$

We will separately calculate $\frac{\partial}{\partial u}\big(\sigma(u)\big)$, using the definition of the sigmoid function.

$$\begin{aligned}
\sigma(u) &= \frac{1}{1 + \sigma^{-u}} \\
\sigma(u) &= (1 + \sigma^{-u})^{-1} \\
\frac{\partial}{\partial u}\big(\sigma(u)\big) &= -\big(1 + e^{-u}\big)^{-2}\big)(-e^{-u}), \text{ by the chain rule} \\
\frac{\partial}{\partial u}\big(\sigma(u)\big) &= \frac{e^{-u}}{(1 + e^{-u})^2}
\end{aligned}$$

Substituting this back into the equation for $\frac{\partial}{\partial w_z}\big(\sigma(u)\big)$ and using the relationship

$u = w_1v_1 + w_2v_2 + \cdots + w_\delta v_\delta + b$ we obtain the following,

$$\frac{\partial}{\partial w_z}\big(\sigma(u)\big) = \frac{e^{-u}}{(1+e^{-u})^2} \cdot \frac{\partial u}{\partial w_z}$$

$$\frac{\partial}{\partial w_z}\big(\sigma(w_1v_1 + w_2v_2 + \cdots + w_\delta v_\delta + b)\big) = \frac{e^{-u}}{(1+e^{-u})^2} \cdot \frac{\partial}{\partial w_z}(w_1v_1 + w_2v_2 + \cdots + w_\delta v_\delta + b)$$

$$\frac{\partial o_1}{\partial w_z} = \frac{e^{-u}}{(1+e^{-u})^2} \cdot \left(\frac{\partial}{\partial w_z}(w_1v_1) + \frac{\partial}{\partial w_z}(w_2v_2) + \cdots + \frac{\partial}{\partial w_z}(w_\delta v_\delta) + \frac{\partial b}{\partial w_z}\right)$$

Since the bias is independent of the weights, $\frac{\partial b}{\partial w_z} = 0$, and thus can be removed from the equation above. We will now begin to consider how this equation applies to Case 2 and Case 3. In Case 2, w_z is directly connected to the relevant output node, meaning $w_z \in \{w_1, w_2, \ldots, w_\delta\}$. As such, the values of all the nodes in the previous layer are independent of w_z. As well, all weights are independent of all other weights, meaning that $\frac{\partial}{\partial w_z}(w_iv_i) = 0$, for all $1 \leq i \leq \delta, i \neq z$. As such, in Case 2, the equation for $\frac{\partial o_1}{\partial w_z}$ can be simplified as follows.

$$\frac{\partial o_1}{\partial w_z} = \frac{e^{-u}}{(1+e^{-u})^2} \cdot \left(\frac{\partial}{\partial w_z}(w_1v_1) + \frac{\partial}{\partial w_z}(w_2v_2) + \cdots + \frac{\partial}{\partial w_z}(w_\delta v_\delta)\right)$$

$$\frac{\partial o_1}{\partial w_z} = \frac{e^{-u}}{(1+e^{-u})^2} \cdot \left(0 + 0 + \cdots + \frac{\partial}{\partial w_z}(w_zv_z) + \cdots + 0\right)$$

$$\frac{\partial o_1}{\partial w_z} = \frac{e^{-u}}{(1+e^{-u})^2} \cdot \frac{\partial}{\partial w_z}(w_zv_z)$$

We recognize that v_z can be treated as a constant since it is independent of w_z. As such, $\frac{\partial}{\partial w_z}(w_zv_z) = v_z \cdot \frac{\partial w_z}{\partial w_z} = v_z$, giving the final equation

$$\frac{\partial o_1}{\partial w_z} = \frac{e^{-u}}{(1+e^{-u})^2} \cdot v_z, \text{ where } u = w_1v_1 + w_2v_2 + \ldots + w_\delta v_\delta + b$$

Since the values of the nodes and the weights are all known, this equation allows us to determine the value of $\frac{\partial o_1}{\partial w_z}$ for Case 2, where w_z is directly connected to the relevant output node. More generally, we have shown how to determine the partial derivative of a node's value with respect to w_z when w_z is directly connected to the node. Since Case 1 allows us to solve for $\frac{\partial o_1}{\partial w_z}$ when w_z connects to a node other than the relevant node in the same layer, we have shown it is always possible to find $\frac{\partial o_1}{\partial w_z}$ when w_z connects to the same layer as the relevant node.

The final case, Case 3, is where w_z is not connected to the output layer. For this case, we must use the equation derived earlier for the partial derivative of an output node value, o_1, with respect to some w_z. Importantly, recall that this equation was derived without making assumptions about the position of w_z.

$$\frac{\partial o_1}{\partial w_z} = \frac{e^{-u}}{(1+e^{-u})^2} \cdot \left(\frac{\partial}{\partial w_z}(w_1v_1) + \frac{\partial}{\partial w_z}(w_2v_2) + \cdots + \frac{\partial}{\partial w_z}(w_\delta v_\delta)\right)$$

In Case 3, $w_z \notin \{w_1, w_2, \ldots, w_\delta\}$. As such, all the weights are independent of w_z, and therefore can be treated as constants, giving the following,

$$\frac{\partial o_1}{\partial w_z} = \frac{e^{-u}}{(1+e^{-u})^2} \cdot \left(w_1 \frac{\partial v_1}{\partial w_z} + w_2 \frac{\partial v_2}{\partial w_z} + \cdots + w_\delta \frac{\partial v_\delta}{\partial w_z} \right)$$

Since the value of u and the values of the weights are all known, we now must only find the values of the partial derivatives of the other nodes' values with respect to w_z. However, we can recognize that finding these partial derivatives is the same problem as finding $\frac{\partial o_1}{\partial w_z}$, except instead of the output node we are considering a node in the previous layer. As such, Case 3 can be resolved through recursion. The above equation shows that the partial derivative of every non-input node's value with respect to w_z can be expressed in terms of the same partial derivative for the nodes in the previous layer. These partial derivatives could then be expressed in terms of the partial derivatives with respect to w_z for the nodes in the next previous layer. This process can then be repeated until the layer of nodes being considered is the layer which w_z connects to. Once this point is reached, all the partial derivatives of these node values can be determined as they will fall under either Case 1 or Case 2, which we have shown are solvable. By allowing these know values to propagate back up to the output layer, we can find $\frac{\partial o_1}{\partial w_z}$ for Case 3. This recursive technique is the essence of backpropagation (Johnson), and it allows us to find the value of $\frac{\partial o_1}{\partial w_z}$, regardless of where w_z sits in the neural network.

Having found $\frac{\partial o_1}{\partial p}$ for when p is a weight, to solve for the gradient vector of the error function we must now only find $\frac{\partial o_1}{\partial p}$ for when p is some bias b_z. The method we employ is very similar to the one used when p is a weight. In the first case, b_z is the bias of an output node other than the relevant output node, making o_1 independent of b_z and thus $\frac{\partial o_1}{\partial b_z} = 0$. In the second case, b_z is the bias of the output node, and in the third case b_z is the bias of some non-output node. Here the equation for the output node value can be used to determine $\frac{\partial o_1}{\partial b_z}$.

$$\begin{aligned} o_1 &= \sigma(w_1 v_1 + w_2 + v_2 + \ldots + w_\delta v_\delta + b) \\ \frac{\partial o_1}{\partial b_z} &= \frac{\partial}{\partial b_z}\big(\sigma(w_1 v_1 + w_2 + v_2 + \ldots + w_\delta v_\delta + b\big) \end{aligned}$$

This equation can be solved by once again using the chain rule and defining $u = w_1 v_1 + w_2 v_2 + \ldots + w_\delta v_\delta + b$.

$$\begin{aligned} \frac{\partial o_1}{\partial b_z} &= \frac{e^{-u}}{(1+e^{-u})^2} \cdot \frac{\partial}{\partial b_z}(w_1 v_1 + w_2 v_2 + \ldots + w_\delta v_\delta + b\big) \\ \frac{\partial o_1}{\partial b_z} &= \frac{e^{-u}}{(1+e^{-u})^2} \cdot \left(\frac{\partial}{\partial b_z}(w_1 v_1) + \frac{\partial}{\partial b_z}(w_2 v_2) + \ldots + \frac{\partial}{\partial b_z}(w_\delta v_\delta) + \frac{\partial b}{\partial b_z} \right) \end{aligned}$$

In the second case where $b = b_z$, all the node values are independent of b_z. Since the weights are also independent of the biases, $\frac{\partial}{\partial b_z}(w_i v_i) = 0$ for all $1 \leq i \leq \delta$. As well, since

$b = b_z$, $\frac{\partial b}{\partial b_z} = \frac{\partial b_z}{\partial b_z} = 1$. As such, our equation for the second becomes the following,

$$\frac{\partial o_1}{\partial b_z} = \frac{e^{-u}}{(1+e^{-u})^2} \cdot (0 + 0 + \ldots + 0 + 1)$$
$$\frac{\partial o_1}{\partial b_z} = \frac{e^{-u}}{(1+e^{-u})^2}$$

For the final case where the bias is for a node not contained in the output layer, the same recursive argument used for the weights can be applied. First, we recall that we can express $\frac{\partial o_1}{\partial b_z}$ in terms of the values and weights of connected nodes through the following equation,

$$\frac{\partial o_1}{\partial b_z} = \frac{e^{-u}}{(1+e^{-u})^2} \cdot \left(\frac{\partial}{\partial b_z}(w_1 v_1) + \frac{\partial}{\partial b_z}(w_2 v_2) + \ldots + \frac{\partial}{\partial b_z}(w_\delta v_\delta) + \frac{\partial b}{\partial b_z} \right)$$

Since all the weights are independent of b_z, they can be treated as constants. As well, since $b \neq b_z$, $\frac{\partial b}{\partial b_z} = 0$. Using these we can obtain the following equation.

$$\frac{\partial o_1}{\partial b_z} = \frac{e^{-u}}{(1+e^{-u})^2} \cdot \left(w_1 \frac{\partial v_1}{\partial b_z} + w_2 \frac{\partial v_2}{\partial b_z} + \ldots + w_\delta \frac{\partial v_\delta}{\partial b_z} \right)$$

Since the values of the weights, the bias b, and u are known, the only necessary values are the partial derivatives of the values of the nodes in the previous layer with respect to b_z. Once again, the same recursive argument can be used to find $\frac{\partial o_1}{\partial b_z}$, as we have shown that each partial derivative can be expressed in terms of partial derivatives of the values of nodes in the previous layer. The layer that contains the node with the bias b_z will have every node fall under the first or second case, meaning it can be solved. This then propagates back up through the layers, allowing for the value of $\frac{\partial o_1}{\partial b_z}$ to be found.

As such, we have shown it possible to find both $\frac{\partial o_1}{\partial w_z}$ and $\frac{\partial o_1}{\partial b_z}$ for all values of w_z and b_z. This means that $\frac{\partial o_1}{\partial p}$ can be found regardless of whether the parameter p is a weight or a bias. Recall the following equation for the partial derivative of the error for a training case, E_1, with respect to p,

$$\frac{\partial E_1}{\partial p} = 2(o_1 - d_1)\frac{\partial o_1}{\partial p} + 2(o_2 - d_2)\frac{\partial o_2}{\partial p} + \ldots + 2(o_\gamma - d_\gamma)\frac{\partial o_\gamma}{\partial p}$$

We now know it is possible to solve for all of the $\frac{\partial o_i}{\partial p}$ terms. Since all the actual output values, $(o_1, o_2, \ldots, o_\gamma)$, and desired output values, $(d_1, d_2, \ldots, d_\gamma)$, are known, we can now find the value of $\frac{\partial E_1}{\partial p}$ for every training case. These values are then used in the equation below to calculate the partial derivative of the error function with respect to p.

$$\frac{\partial E}{\partial p} = \frac{1}{n}\left(\frac{\partial E_1}{\partial p} + \frac{\partial E_2}{\partial p} + \ldots + \frac{\partial E_n}{\partial p} \right)$$

Finally, the values of $\frac{\partial E}{\partial p}$ for every weight and bias p are what compose the gradient vector of the error function, as the gradient vector is given as follows,

$$\nabla E(w_1, w_2, \ldots, w_\alpha, b_1, b_2, \ldots b_\beta) = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_\alpha}, \frac{\partial E}{\partial b_1}, \frac{\partial E}{\partial b_2}, \ldots, \frac{\partial E}{\partial b_\beta}\right)$$

As such, we have shown that using back-propagation it is possible to calculate the gradient vector of the error function.

6 Applying Gradient Descent

Having found the gradient vector of the error function, it can now be used to optimize the weights and biases such that the error function comes closer to the local minimum. This is done by adjusting each weight or bias p by its corresponding component in the gradient vector (Christiansen). Let p' be the new value of p after gradient descent has been performed and E be the error function. The value of p' will then be given by the following equation:

$$p' = p - \frac{\partial E}{\partial p}$$

Note that we subtract $\frac{\partial E}{\partial p}$ from p as the negative gradient vector gives the direction of fastest decrease, and we are trying to find a local *minimum* of the error function.

7 Conclusion

In conclusion, this essay has shown that mathematics can be applied to improve the performance of neural networks through the use of gradient descent and backpropagation to adjust the neural network's parameters such that it produces an improved output. To accomplish this, the structure of a neural network and its parameters was laid out. Then, the error function was developed as a way to measure the ability of a neural network to produce the desired outputs. Next, gradient descent was explored as a method to minimize the error function. Finally, it was demonstrated how backpropagation could be used to calculate the gradient vector of a neural network's error function. This then allowed for gradient descent to be applied, minimizing the error function and ultimately improving the performance of the neural network. However, this essay also discussed the limitations of this approach. Most notably, gradient descent only allows us to find a local minimum of the error function, which ultimately limits our ability to optimize the performance of the neural network.

8 Works Cited

Christiansen, Niels H, et al. "Comparison of Neural Network Error Measures for Simulation of Slender Marine Structures." *Journal of Applied Mathematics,* 2 March 2014, https://doi.org/10.1155/2014/759834.

Hardesty, Larry. "Explained: Neural networks." *Explained: Neural networks | MIT News | Massachusetts Institute of Technology,* Massachusetts Institute of Technology, 14 April 2017, https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414.

IBM. "What is gradient descent?" *What is gradient descent? | IBM,* IBM, 2022, https://www.ibm.com/topics/gradient-descent.

Isaksson, Martin. "Four Common Types of Neural Network Layers." *Four Common Types of Neural Network Layers | by Martin Isaksson | Towrds Data Science,* Towards Data Science, 6 June 2020, https://towardsdatascience.com/four-common-types-of-neural-network-layers-c0d3bb2a966c.

Johnson, Daniel. "Back Propagation in Neural Network: Machine Learning Algorithm." *Back Propagation in Neural Network: Machine Learning Algorithm,* GURU99, 29 October 2022, https://www.guru99.com/backpropogation-neural-network.html.

Paolucci, Roman. "The Gradient Vector." *The Gradient Vector. What is it and how do we compute it? | by Roman Paolucci | Towards Data Science,* Towards Data Science, 4 June 2020, https://towardsdatascience.com/the-gradient-vector-66ad563ab55a.

Seth, Naha. "Estimation of Neurons and Forward Propagation in Neural Net." *Estimation of Neurons and Forward Propagation in Neural Net,* Analytics Vidhya, 26 April 2021, https://www.analyticsvidhya.com/blog/2021/04/estimation-of-neurons-and-forward-propagation-in-neural-net.

Sharma, Sagar. "Activation Functions in Neural Networks." *Activation Functions in Neural Networks | by SAGAR SHARMA | Towards Data Science,* Towards Data Science, 6 Sept 2017, https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

Weisstein, Eric W. "Sigmoid Function." *Sigmoid Function – from Wolfram MathWorld,* Wolfram MathWorld, https://mathworld.wolfram.com/SigmoidFunction.html.