Mathematics Behind Loss Functions in Neural Networks

Research Question: To what extent does the mathematical choice of loss functions affect the efficiency and accuracy of training neural networks to perform regression tasks?

Subject: Mathematics

Word Count: 3977

Table of Contents

1. Introduction

In recent years, artificial intelligence, a technology that stimulates the processing that occurs in the human brain, has become prevalent in our daily lives (Marc Peter Deisenroth et al., 2020). First developed in the 1950s by Alan Turing (Kaul et al., 2020), artificial intelligence has and continues to aid us in many tasks including predicting stock markets to even classifying DNA (Das et al., 2015). Most of these artificial systems stem from a structure known as neural networks used to detect patterns in data based on a task that the developer is trying to achieve. While neural networks and artificial intelligence may be solely related to computer science, I was intrigued to notice the immense amounts of mathematics involved in the creation of these networks. This made me question the extent to which mathematics is necessary when deciding specific features of the network, such as the loss function. This thought led me to my research question, "To what extent does the mathematical choice of loss functions affect the efficiency and accuracy of neural networks in regression tasks?" Therefore, this essay will explore the mathematical operations that occur within neural networks and the prominence of the loss functions in a network. To best answer the research question, I will be exploring four different loss functions, mean average error, mean squared error, Huber loss and log-cosh error; by examining their mathematical principles, I hope to identify how they influence the efficiency, how fast the network runs, and the accuracy. Through such exploration using visual and mathematical equations as well as an application of such loss functions, this essay concludes that the efficiency and accuracy of neural networks are influenced by the choice of these loss functions to a large extent.

2. Background Knowledge

For this exploration, the scope is set to the task of regressions, which is when the user inputs data into the network creating an output that predicts the variable of interest.

According to Bishop (1998), each of the circles in *Figure 1* are referred to as nodes. Nodes process and perform operations on data from the preceding nodes, creating new data for the following nodes. A neural network is divided into three components known as layers, consisting of the input layer, the hidden layer(s) and the output layer. The input layer (the column of green circles) is where the data first enters the network. The hidden layer(s) (the two columns of blue circles) transform the data in a way that allows the computer to recognize certain patterns. The output layer then uses the process to formulate a final output as shown by the purple circle at the end of the network (Bishop, 1998).

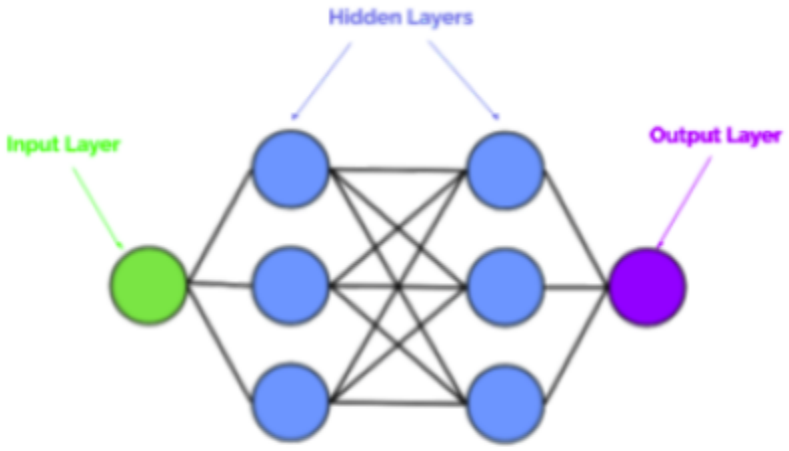

Figure 1 Structure of a Neural Networks

When data is passed through connections between nodes, they are multiplied by a weight. A weight is a numerical value that determines the strength or influence a connection has on the output. The data then arrives at the following node, where a bias is added to the data, a constant term that is added to the product of the weight for a better approximation of the data. If biases are not present, when an input is 0, the output would also be 0, as there is no constant term added,

resulting in the data being omitted when it might be useful (Caterini, 2017). Afterwards, a summation formula occurs condensing all the data that arrives from the preceding nodes. The summation algorithm is depicted by the formula below (Bishop, 1998):

$$s_j = \sum_{i=0}^{i} (w_{ij} x_i + b_j)$$

where s is the resulting data, j is the node that has received all the data, i as the number of nodes preceding the node, w is the weight between node i and j and x is the input that is communicated from node i, and b is the bias at node j. Note that the summation only occurs when there is more than one connection from its preceding nodes.

The dataset is then passed through an activation function which decides whether a node should be activated or not based on its significance to its end calculation. The activation function introduces non-linearity into the neural network, meaning that each node does not need to be confined to adapting a linear model, $y = mx + b$, where m is the weight and b is the bias, from the summation formula.. This linearization reduces the practicality of the model, hence activation functions are essential for the understanding of complex patterns (Sharma et al., 2020). In mathematical notation, the passing of data through the non-linear activation function is denoted as the following:

$$z_J = a(s_j)$$

Where a is the activation function and z_j is the final data that will be used as the input for the next node. in the neural network.

Figure 2 showcases a summary of the data processing process for a neural network with one input node, two hidden nodes and two output nodes.

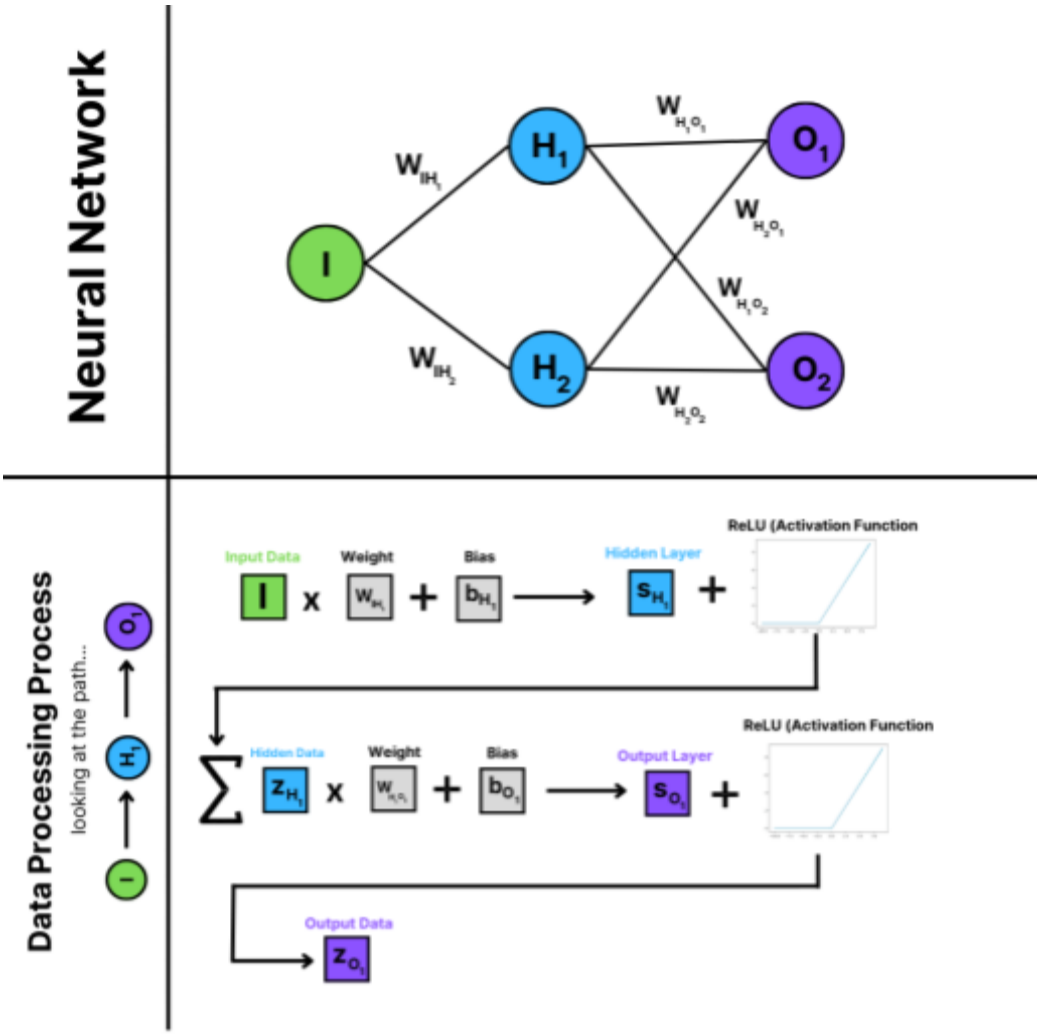

Figure 2. How Neural Networks Process Information

3. Training Neural Networks & The Significance of Loss Functions

At first, neural networks will not know the task that they are assigned to do; they need to be trained. The training process for neural networks starts off by comparing the output that the model gets to the actual outputs that the network should reproduce. This comparison occurs through the use of a loss function, which quantifies the deviation in the output of the neural network from the desired target (Ciampiconi et al., 2023). An example of a loss function is the mean squared error (MSE), given by the equation:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \widehat{Y}_i)^2$$

The mean squared error finds the difference in value between the target, $\widehat{Y}_i$ and the value that was produced in the network, Y_i. It then takes the sum of all n data points that is outputted and averages it out.

As shown in *Figure 3*, assuming that $n = 2$, computers attempt to minimize the loss within a neural network. In order to do this, they will use differentiation to find the local minimum. In the context of the MSE function, it is when the difference between the target and the output is 0.

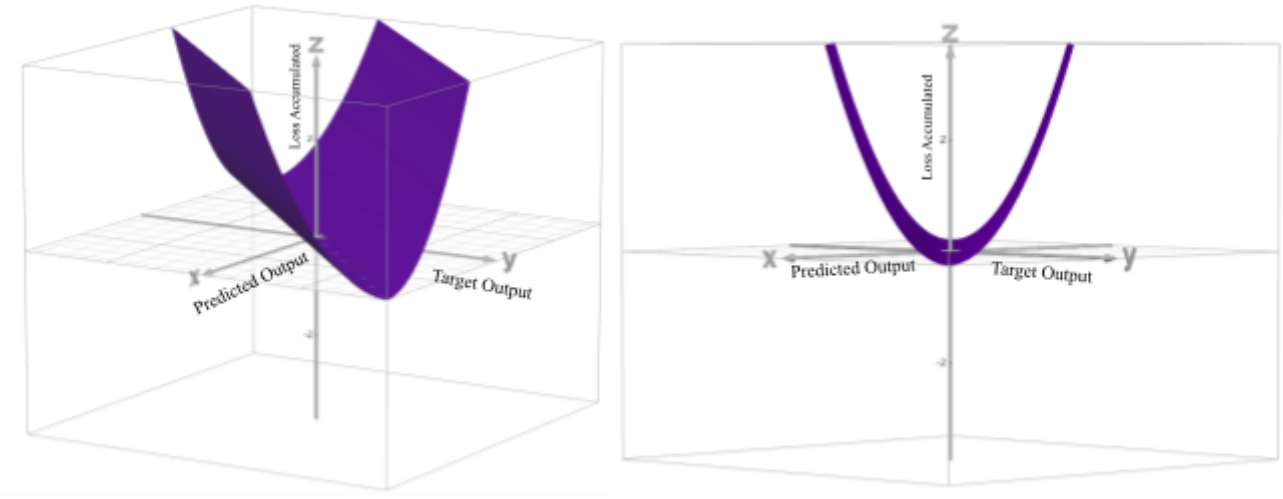

Figure 3. Graphical Representations of the MSE Loss Function, $z = 0.5\ (x - y)^2$

After identifying the value of the loss, the network must change the weights and biases to minimize loss, which is done through an optimizer. One popular optimizer is called gradient descent, which looks at the partial derivative of the loss function with respect to the weight or bias that is being looked at. It applies the mathematical formula below to change the weight in order to reach a local minimum (Doshi, 2020).

$$W_{new} = W_{old} - \alpha \times \frac{\partial L}{\partial W_{old}}$$

Where W is the weight that is being changed, α refers to the learning rate, which is the rate at which the weight is changed, $\frac{\partial L}{\partial W_{old}}$ is how much the loss changes with respect to changing a certain weight. It is important to note that the derivative of the loss is not represented by $\frac{dL}{dW_{old}}$ but $\frac{\partial L}{\partial W_{old}}$ as this is a partial derivative. Multivariable differentiation or partial differentiation, refers to how much the output of a function changes when the value of only one variable involved is changed. The key component is that there are other variables, not only one. Partial differentiation is denoted by ∂ instead of d but uses the same differentiation techniques, with other variables in the equation being treated as constant.

However, calculating the partial derivative, $\frac{\partial L}{\partial W_{old}}$ is not intuitive as there exists no direct relationship between the weight and the loss itself and the weight influences the data going forth. Hence, we need to consider the following nodes and how it affects the loss accumulated over as well.

In *Figure 4,* let's consider changing the weight W_{IH_1}.

As you can see there is a large influence that the weight of W_{IH_1} on the entirety of the network and the loss through two paths, one through the first output neuron and one through the second output neuron depicted by the red lines. Since we know that W_{IH_1}, has an influence on s_{H_1}, which has an influence on z_{H_1}, and so on and so forth, we can consider each layer as a function, and the entire network as a composite function. Thus, we can use chain rule to calculate the partial derivative of $\frac{\partial L}{\partial W_{IH_1}}$. Using the first tree diagram in *Figure 4* and working backwards to find the weight that is looking to be changed, the partial derivative can be written as:

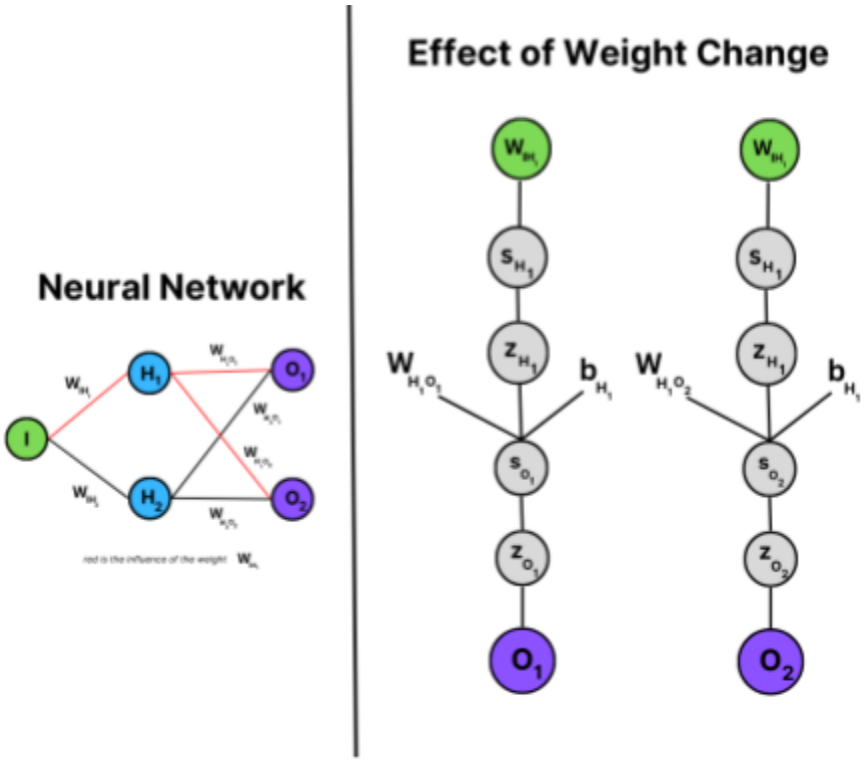

Figure 4. Tree Diagram For The Effect Of The Weight Change

$$\frac{\partial L}{\partial W_{IH_1}} = \frac{\partial L}{\partial z_{O_1}} \frac{\partial z_{O_1}}{\partial s_{O_1}} \frac{\partial s_{O_1}}{\partial z_{H_1}} \frac{\partial z_{H_1}}{\partial s_{H_1}} \frac{\partial s_{H_1}}{\partial W_{IH_1}}$$

We can simplify by looking at each individual derivative in the chain rule. Looking first at $\frac{\partial L}{\partial z_{O_1}}$, we can use the *MSE* as our loss function assuming $n = 1$. Y_i within the loss function can be

substituted with z_{O_1} as that is the output that the neural network will produce. This leaves us with the following:

$$L = (Y_i - \widehat{Y}_i)^2$$

$$L = (z_{O_1} - \widehat{Y}_i)^2$$

$$\frac{\partial L}{\partial z_{O_1}} = 2(z_{O_1} - \widehat{Y}_i) \times (z_{O_1})^0$$

$$\frac{\partial L}{\partial z_{O_1}} = 2(z_{O_1} - \widehat{Y}_i)$$

Looking at the second partial derivative of $\frac{\partial z_{O_1}}{\partial s_{O_1}}$, we note that $z_{O_1} = a(s_{O_1})$, where a is the activation function. This means that $\frac{\partial z_{O_1}}{\partial s_{O_1}} = a'(s_{O_1})$.

For the next partial derivative, $\frac{\partial s_{O_1}}{\partial z_{H_1}}$, we can solve it as shown below:

$$s_{O_1} = W_{H_1O_1} \times z_{H_1} + b_{O_1}$$

$$\frac{\partial s_{O_1}}{\partial z_{H_1}} = W_{H_1O_1} \times (z_{H_1})^0$$

$$\frac{\partial L}{\partial z_{O_1}} = W_{H_1O_1}$$

The partial derivative calculations for the fourth and fifth derivative are similar to the ones above (see *Appendix A*). This leaves us with the new partial derivative of $\frac{\partial L}{\partial W_{IH_1}}$ when substituting all of the partial derivatives in the chain rule:

$$\frac{\partial L}{\partial W_{IH_1}} = \frac{\partial L}{\partial z_{O_1}} \frac{\partial z_{O_1}}{\partial s_{O_1}} \frac{\partial s_{O_1}}{\partial z_{H_1}} \frac{\partial z_{H_1}}{\partial s_{H_1}} \frac{\partial s_{H_1}}{\partial W_{IH_1}}$$

$$\frac{\partial L}{\partial W_{IH_1}} = 2(z_O - \widehat{Y}_i) \cdot a'(s_O) \cdot W_{H_1O_1} \cdot a'(s_{H_1}) \cdot z_I$$

This is not the end of the solution for this partial derivative. Notice how in *Figure 4,* the W_{IH_1} also has an effect on the O_2 through the connection between H_1 and O_2. Hence, summing the chain rule of the second output node with the chain rule of the first output node (see *Appendix B*) gives us:

$$\frac{\partial L}{\partial W_{IH_1}} = \frac{\partial L}{\partial z_{O_1}} \frac{\partial z_{O_1}}{\partial s_{O_1}} \frac{\partial s_{O_1}}{\partial z_{H_1}} \frac{\partial z_{H_1}}{\partial s_{H_1}} \frac{\partial s_{H_1}}{\partial W_{IH_1}} + \frac{\partial L}{\partial z_{O_2}} \frac{\partial z_{O_2}}{\partial s_{O_2}} \frac{\partial s_{O_2}}{\partial z_{H_1}} \frac{\partial z_{H_1}}{\partial s_{H_1}} \frac{\partial s_{H_1}}{\partial W_{IH_1}}$$

$$\frac{\partial L}{\partial W_{IH_1}} = 2(z_{O_1} - \widehat{Y}_i) \cdot a'(s_{O_1}) \cdot W_{H_1O_1} \cdot a'(s_{H_1}) \cdot z_I + 2(z_{O_2} - \widehat{Y}_i) \cdot a'(s_{O_2}) \cdot W_{H_1O_2} \cdot a'(s_{H_1}) \cdot z_I$$

This partial derivative was proven by numerous individuals including Sanderson (2017). In the $\frac{\partial L}{\partial W_{IH_1}}$ partial derivative, it is important to note that the $2\ (z_{O_1} - \widehat{Y}_i)$ and $2\ (z_{O_2} - \widehat{Y}_i)$ is the partial derivative of the loss function. Hence, the loss function has a major role from a

mathematical standpoint within the computation of derivatives and how much the weight or bias changes from the optimizer.

4. Types of Loss Functions and their Mathematical Role

According to Ciampiconi et al (2023), there are two important characteristics that are involved in looking into neural networks and their mathematical properties to determine the accuracy and efficiency of such.

The first factor is continuity, which is when a function $f(x)$ has no breaks or jumps, it is a single unbroken curve belonging to the real domain. This is necessary within a neural network as there may be two gradients if the function reaches a certain point where the function breaks resulting in a failure in the training process. Mathematically, a function $f(x)$ is continuous at the real number of a, if $\lim_{x \to a} f(x) = f(a)$.

The second factor is differentiability across the entire function which is particularly important such that if there exists a vertical line where the slope is undefined, the learning process, specifically the optimizer, will not work, as the gradient is considered infinity. Moreover, if there is not a smooth transition, as in the gradient gradually decreases to 0, the derivative will be undefined, which will also result in the optimizer not working (Ciampiconi et al, 2023).

4.1 Mean Absolute Error

The mean absolute error (MAE) is defined as the average absolute difference between the targets and the actual output. Mathematically, it is given by the formula:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}\left| y_i - \widehat{y}_i \right|$$

Where $\widehat{Y}_i$ is the target value, Y_i is the value that was produced in the network and *n* is the number of data points that is outputted or that is aimed to be targeted. One of the major benefits involved in the understanding of the mean average error is the lack of complexity involved in the calculation. As the function only involves the difference between two values and an average, a computer can easily calculate such values. This reduces the time complexity dramatically, improving its efficiency.

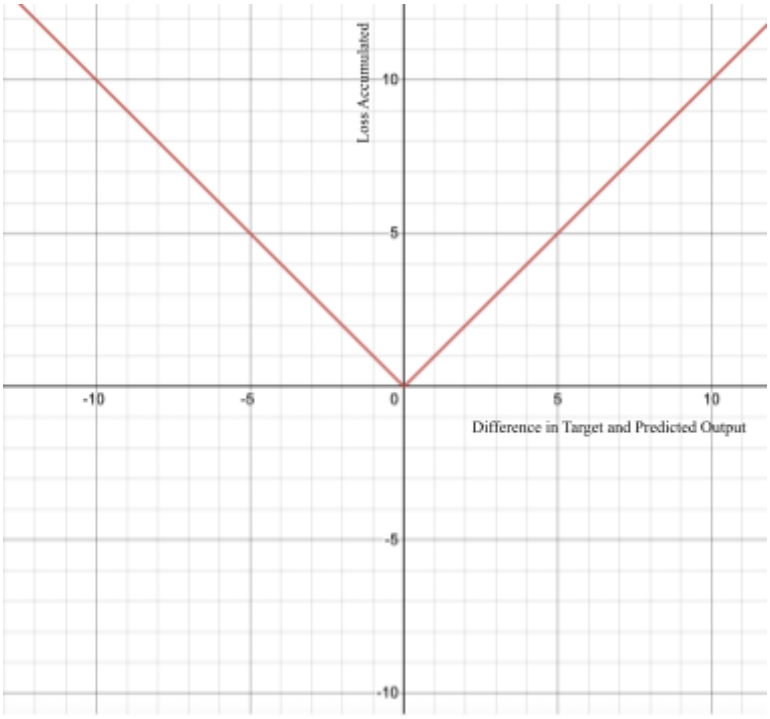

Figure 5. Graph of the Mean Average Error Function, y = | x |

However, looking at *Figure 5*, we notice that there exists one major downfall involved within the loss function. The derivative of the absolute function is given by the formula $\frac{dy}{dx} = \frac{x}{|x|}$. When x = 0, it is considered undefined. Hence, this loss function violates the second factor, that the function is differentiable. This especially becomes problematic when the gradient of the loss function with respect to a weight is undefined, which may result in the failure of the training (Yathish, 2022).

Another important feature of the function is its linear behaviour. The derivative of the MAE loss function has only three values, a negative constant derivative to the left of x = 0, an undefined

derivative at x = 0 or a positive constant derivative to the right of x = 0. Referring back to the chain rule, the $\frac{\delta L}{\delta z_{O_1}}$, will simply be a constant value, only telling the network whether the weight should increase or decrease. This implies that the gradient magnitude is not dependent upon the loss function, which may result in less accuracy, as the weight updates from the optimizer will take place on a smaller scale (Balawejder, 2022).

4.2 Mean Squared Error

To solve the major problems listed above, the mean squared error (MSE) was created. This loss function is quite similar to the MAE loss function, except in replacement of the absolute function, a quadratic is involved. The formula is depicted below:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \widehat{Y_i}\right)^2$$

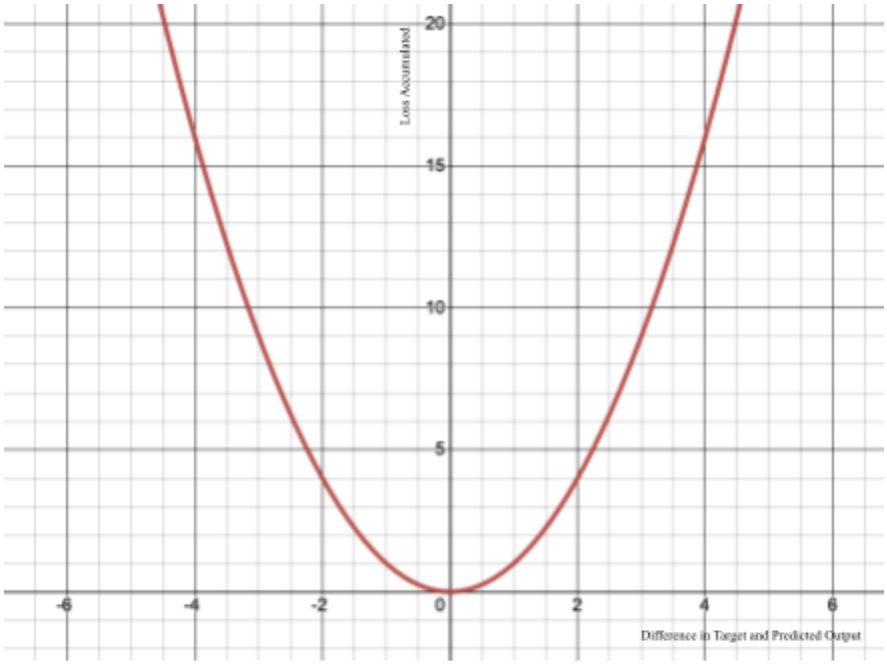

Figure 6. Graph of the Mean Squared Error, $y = x^2$

Looking at the mathematical formula and *Figure 6*, the function is differentiable at all real values, including the vertex point. Furthermore, each point in the quadratic function is different, which means that the optimizer will be more flexible in changing the weight values, as the derivative of the loss with respect to the output is no longer constant, which results

in higher accuracy. Another significant feature about this loss function is the fact there exists one global minimum, meaning that the lowest loss will be achieved (Ciampiconi et al, 2023).

The major issue with this loss function comes into play when there are large outliers in the predictions. Due to the squared term within the MSE formula, large differences between the target and the prediction will be substantial, affecting the loss value, leading to a model that is overly sensitive to its outliers. This can potentially result in a decrease in the accuracy. Furthermore, comparative to the MAE loss, there exists more complexity due to the squared which in effect increases the time (Yathish, 2022).

4.3 Huber Loss

Huber loss takes into consideration both of the benefits and drawbacks that are associated with the MAE and MSE. This is done through a specific condition based on a constant represented by δ that looks into transitioning between the MSE and MAE. Using this idea, this should be represented by the formula below:

$$L_{Huber\ loss} = \begin{cases} x^2 & for\ |x| \leq \delta \\ |x| & for\ |x| > \delta \end{cases}$$

Where x is the difference between the target value and the predicted value. However, there is an issue for this function, as it is non-differentiable and when $\delta \neq 1$, the function is not continuous. This means that this loss function would not work effectively in the training of a neural network.

In order for a function to be differentiable, the transition from the absolute function to the quadratic must be smooth, in that the slope between the transition point must be the same between both functions. This means that the second function, the $y = |x|$ must be changed and transformed to effectively fit the criteria of smooth functionality. Let's consider x > 0, meaning that absolute function that is being transformed will now be y = x.When x = δ, two conditions must be true: the gradient of the y = x^2 function must be the same as the gradient of the transformed y = x and the value of both functions must be the same. Using the first condition, we start off by finding the gradient that is needed to be the same.

$$y = x^2$$

$$\frac{dy}{dx}\Big|_{\delta} = 2x$$

$$\frac{dy}{dx}\Big|_{\delta} = 2\delta$$

Now we need to find a linear function to extend the x^2 function that has the same gradient of 2δ. We can use integration to figure such lines out.

$$\int 2\delta \, dx = 2\delta x + c$$

Using the second condition that both values needed to be the same, we can find the c value.

$$x^2 = 2\delta x + c$$

$$(\delta)^2 = 2\delta(\delta) + c$$

$$-c = 2\delta^2 - \delta^2$$

$$c = -\delta^2$$

Since we assumed that x > 0, the second linear equation turns to be $y = 2\delta|x| - \delta^2$. This results in the new huber loss function that accounts for continuity and differentiation, no matter the value of the constant δ.

$$L_{Huber\ Loss} = \begin{cases} x^2 & for\ |x| \leq \delta \\ 2\delta|x| - \delta^2 & for\ |x| > \delta \end{cases}$$

This loss function has been proven by numerous individuals including Theekshana (2021). Looking at *Figure 7* , where δ = 1, the blue section is related to the equation of x^2 whereas the red curve represents $y = 2\delta|x| - \delta^2$. The graph showcases the benefits of the combination between MSE and MAE; not only is the entire function differentiable including at x = 0 but the outliers will not overly affect the total loss generation in comparison to MSE (Yathish, 2022).

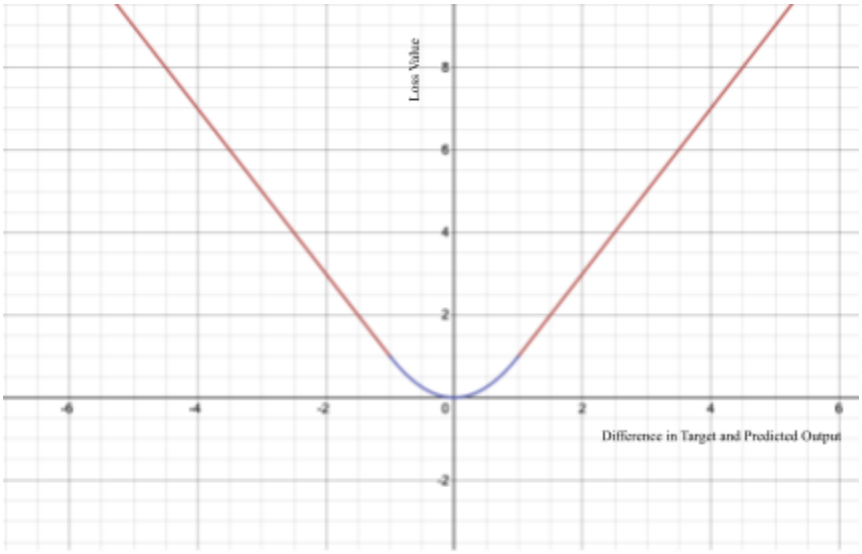

Figure 7. Graph of Huber Loss

The only major downfall is the time complexity as it has to operate two functions and make a comparison between such. Furthermore, when $|x| \leq \delta$, the gradient is constant, operating similar to the MAE, which may result in less accuracy as the change in weight is not dependent upon the value of the loss as it is considered constant (Jadon et al., 2022).

4.4 Log-Cosh Loss

Another loss function that could be used is the log-cosh loss function. Before explaining such, it is important to understand hyperbolic functions. Many of the most common trigonometric functions are deviated from the unit circle, whereas hyperbolic functions are from what is known as a hyperbola. A hyperbola is a pair of open curves that face outward from the center, defined by the equation of $x^2 - y^2 = 1$. This is shown in *Figure 8*.

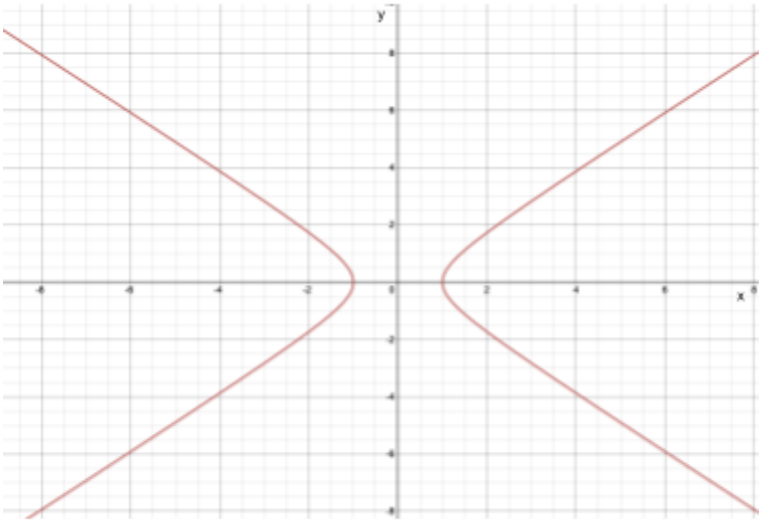

Figure 8. Graph of Hyperbola Function, $x^2 - y^2 = 1$

In the unit circle, the inputs to trigonometric functions are the value of the angle, typically in radians, whereas within hyperbolic functions, the input is with reference areas in calculations, as shown in figure 9. As proven by Máté (2016), when a line from the origin to a point on the hyperbolic function, the area is half the value that is imputed into the hyperbolic functions, as depicted by *Figure 9*. Furthermore, similar to unit circle, the x-axis is representative of cosh, and the y-axis as sinh, which are two hyperbolic functions.

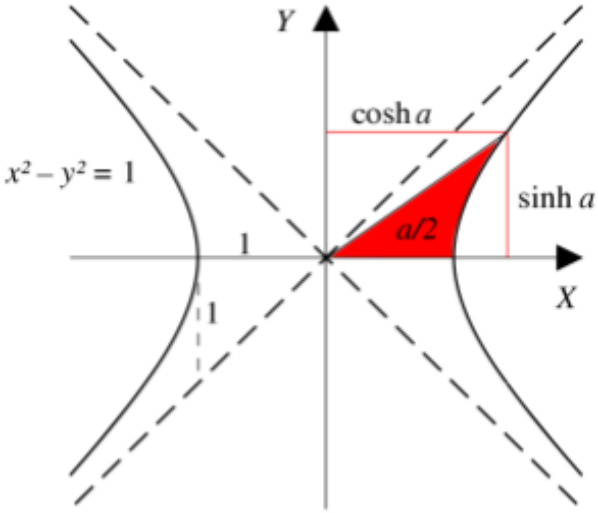

Figure 9. Hyperbolic Connection Between Area and Input

Our focus will be on the cosh function, applied with a logarithmic function which will give an effective loss function that can be used as depicted by *Figure 10*. Notice how this is quite similar to the Huber Loss, except there are less computations, which means that time complexity may be significantly less. Moreover, the less steep gradients will result in the function being less oversensitive to any outliers, benefiting the overall training process of the neural network (Jadon et al., 2022).

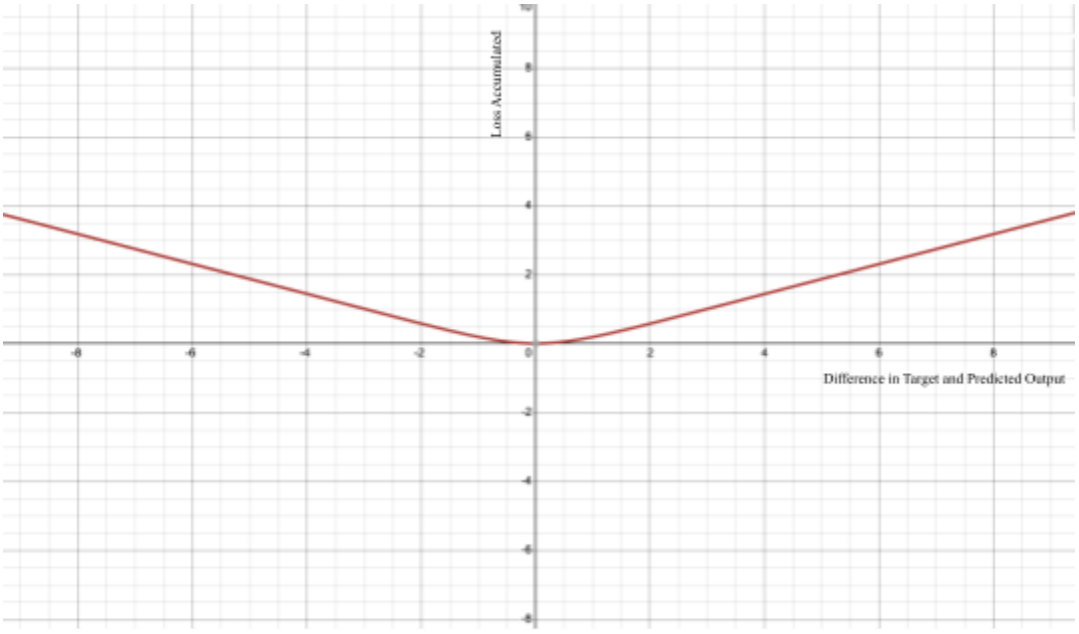

Figure 10. Graph of Log-Cosh Loss Function, log(cosh(x))

However, as the gradients are less steep in comparison to other models, the optimizer will change its weights on a lower scale than usual. Furthermore, even though the calculation of the loss is less complex, finding the derivative of this function with respect to the output is computationally expensive which could reduce its efficiency. This is shown through the derivation below (Jadon et al., 2022).

Below are some key derivatives and hyperbolic identities that should be known is differentiation of the log-cosh function:

(1) $\frac{dy}{dx} log(x) = \frac{1}{x\, ln10}$

(2) $\frac{dy}{dx} cosh(x) = sinh(x)$.

(3) $\frac{sinh(x)}{cosh(x)} = tanh(x)$

Using chain rule to simplify the derivative:

$$y = log(cosh(x))$$

$$\frac{dy}{dx} = \frac{1}{cosh(x)\ ln\ 10} \times sinh(x)$$

$$\frac{dy}{dx} = \frac{sinh(x)}{cosh(x)\ ln\ 10}$$

$$\frac{dy}{dx} = \frac{tanh(x)}{ln\ 10}$$

It has also been proven by Tabrizian (2019) that $tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$. Due to its complex nature, the proof has been omitted. Hence, we could simplify this derivative even further:

$$\frac{dy}{dx} = \frac{\frac{1-e^{-2x}}{1+e^{-2x}}}{ln\ 10}$$

$$\frac{dy}{dx} = \frac{1-e^{-2x}}{(1+e^{-2x})(\ ln\ 10)}$$

This is only one of the derivatives that needed to be found to change the weights and biases, which makes this model not efficient due to its extensive calculations involved. Hence, time wise, this may not be an efficient loss function.

4.5 Comparison

In terms of accuracy, assuming that there exists no outliers, the log-cosh function appears to be the most accurate due to its low sensitivity to outliers, which is then followed by Huber loss, MSE & MAE. However, in terms of time complexity, due to its simplistic calculations, MAE prevails with MSE, Huber loss and log-cosh following.

5. Testing

5.1 Methodology

To test the mathematical analysis out, I have created a neural network that will be trained on the Boston Housing dataset. The neural network will look into 13 numerical values and create a regression to predict the median value of owner-occupied homes in $1000s (Harrison & Rubinfeld, 1978). The neural network will consist of an input layer with 13 nodes (matching with the number of input values) with two hidden layers of 64 nodes and an output layer of 1 node. The optimizer will remain as gradient descent with a learning rate of 0.01, and only the loss function will change. Each loss function will have three trials averaged in terms of loss and efficiency. Each trial will have a different order of the data that is fed into the network. Furthermore, there will be 50 epochs that are run; an epoch is how many times the neural network will go through the entire dataset to train. Accordingly, there will be a calculation of the average loss generated from the data at each epoch and how much time it took over the 50 epochs. To create the code for testing, I used a tutorial made by Eduardo (2021), to aid me in my exploration (*see Appendix C*).

5.2 Results

As expected from the mathematical analysis, the log-cosh loss function was the most effective, whereas the MSE was by far the worst in terms of accuracy.

Looking at *Figure 11,* where the blue dots are representative of the MSE loss function, the black the log-cosh loss function, red the huber loss function and green the MAE loss function (*see Appendix D, E, F, G, H, I, J, K* for data & processing). The loss that was generated by the MSE was much more substantial in comparison to the other three loss functions, which suggests that the Boston Housing dataset had enormous outliers as the mean squared error is overly sensitive to such data. Hence, in this instance it is the least accurate. It is also worth mentioning that the other three loss functions have losses that are close together, this is due to their low sensitivity to outliers.

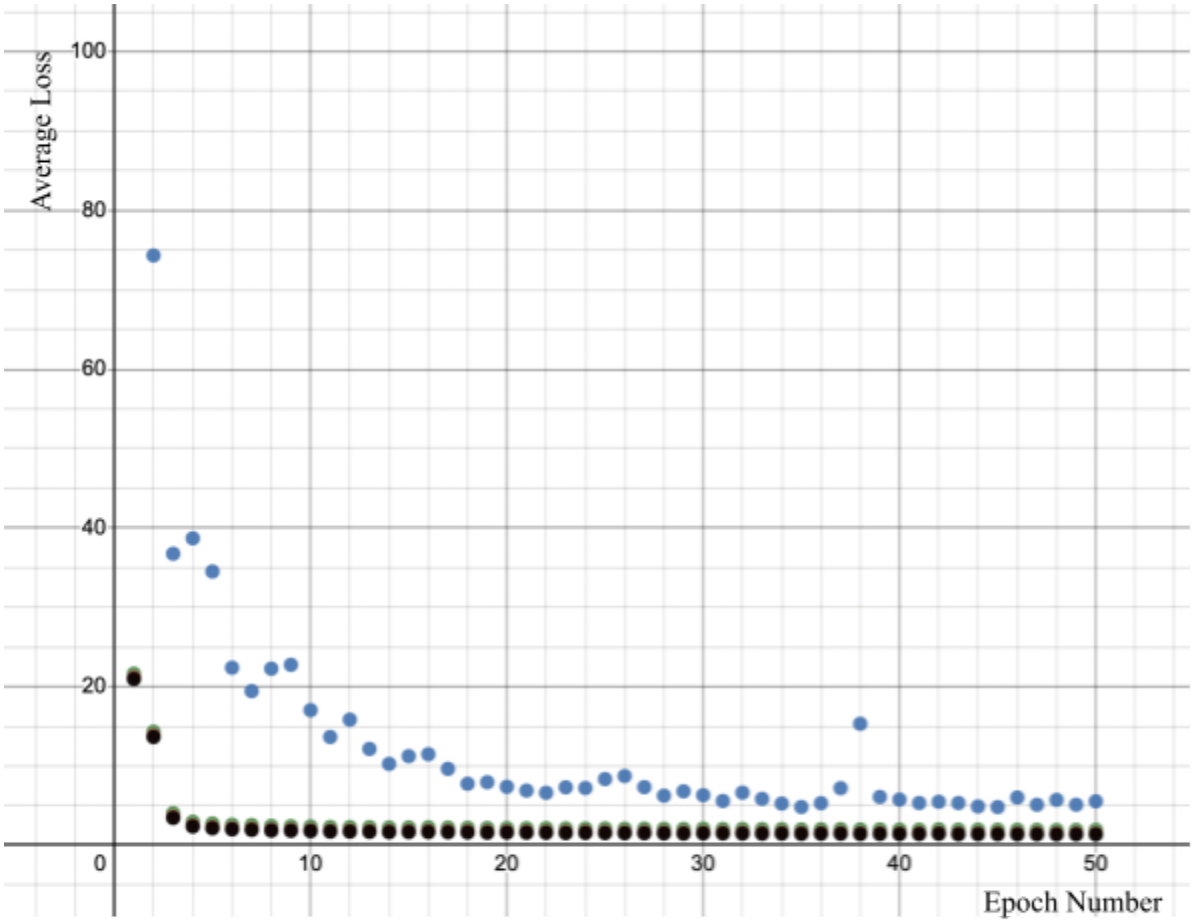

Figure 11. Comparison of Loss Functions in Application

Looking more closely into *Figure 12*, as predicted, the log-cosh loss is much more accurate, as it generates a much smaller loss overtime. It reaches the minimum point of the loss function more easily and is less sensitive to the outliers present in comparison to the other three functions.

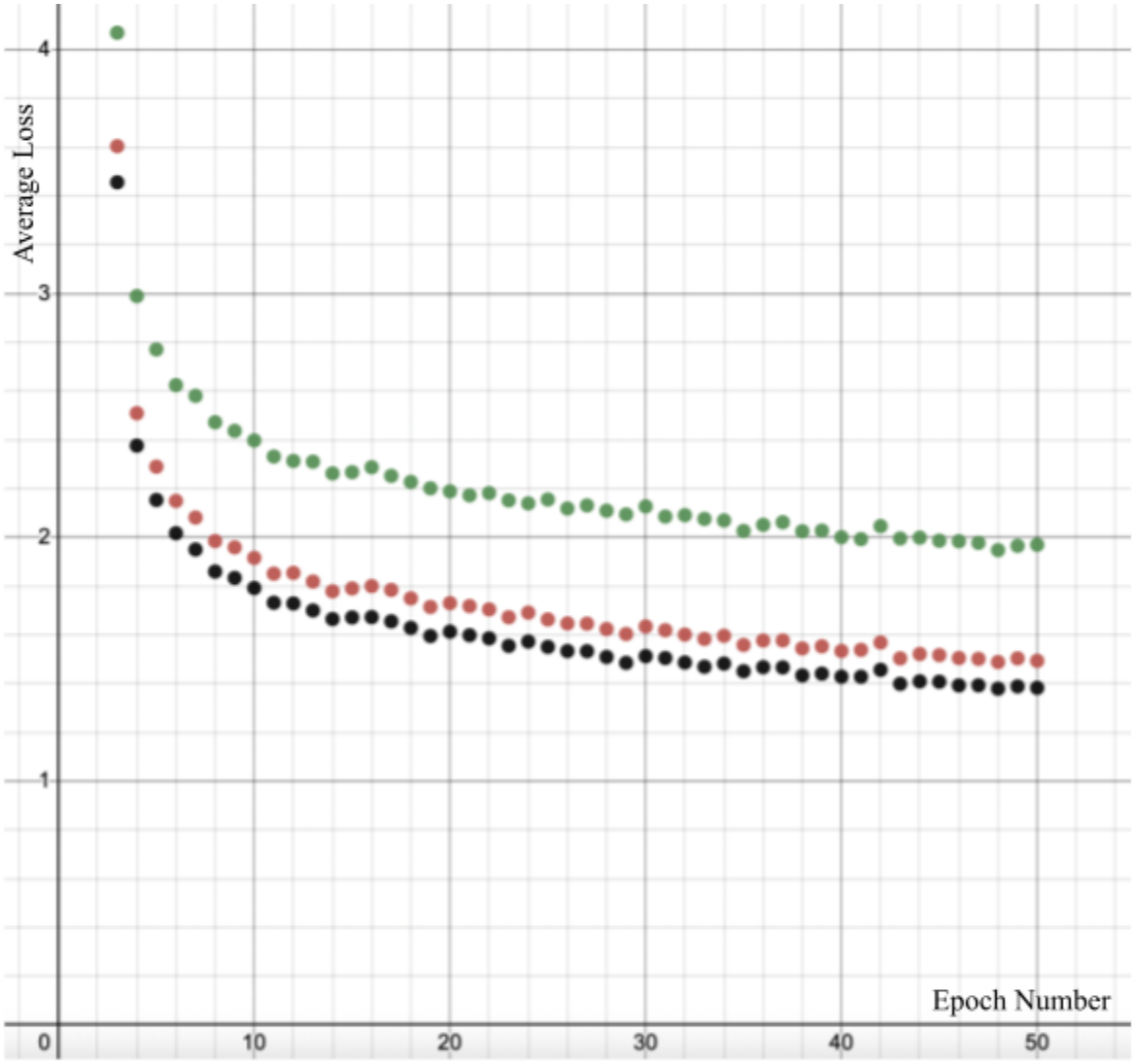

Figure 12. Close-Up Comparison of Loss Functions in Application

It is important to note that as the data did contain major outliers which resulted in the MSE loss function being very sensitive. If there were no outliers, this may differ.

As far as time complexity is concerned, as shown by *Table 1*, the mathematical theory aligned with the resulting experiment as MAE was the quickest loss function due to its simple calculations, followed by MSE, Huber loss and log-cosh.

Table 1. Neural Network Loss Function Training Timing

Loss Function	**Trial 1 (s)**	**Trial 2 (s)**	**Trial 3 (s)**	**Average**
MAE	11.87742829322815	12.358670234680176	12.698989152908325	12.311695893606
MSE	12.010592222213745	12.720363855361938	12.633536577224731	12.454830884933
Huber	12.891549825668335	13.05600118637085	13.015963315963745	12.987838109334
Log-Cosh	14.051464796066284	13.965463876724243	15.46820616722107	14.495044946671

It is important to note that these timings may potentially be influenced by the processing speed of the computer, which is a limitation.

It is significant how the experimental values align with the mathematical theory. In effect, this implies that there exists a large extent to which the mathematical choice of loss functions that influences the efficiency and accuracy of the neural network.

6. Conclusion

In conclusion, through the exploration of four different loss functions, including mean average error, mean squared Error, Huber loss, log-cosh loss, there exists a large extent to which the mathematical choice of the loss function influences the efficiency and accuracy of the neural network. While the difference within the timing and accuracy was not as substantial, as developers try to deal with more complex neural networks that have more hidden layers or huge datasets, the difference in terms of the timing and accuracy becomes significant. In these instances, with the large resources involved, the network must be trained in order to effectively adapt to a model, which can take hours or even days. If different loss functions are tested experimentally, the process will take months. However, by doing a mathematical analysis, programmers can effectively identify which loss function best fits their task, finding a balance between accuracy and efficiency prior to the training process. Overall, the nature of mathematics within neural networks in its training process becomes quintessential in order to understand efficiency and accuracy, which can help how we train such forms of artificial intelligence.

References

Balawejder, M. (2022, March 30). *Loss functions in Machine Learning*. Nerd for Tech. https://medium.com/nerd-for-tech/what-loss-function-to-use-for-machine-learning-project-b5c5bd4a151e

Bishop, C. M. (1998). *Neural networks and machine learning*. Springer.

Caterini, A. (2017). *A Novel Mathematical Framework for the Analysis of Neural Networks*. University of Waterloo. https://uwspace.uwaterloo.ca/bitstream/handle/10012/12173/caterini_anthony.pdf?sequence=3

Ciampiconi, L., Elwood, A., Leonardi, M., Mohamed, A., & Rozza, A. (2023). *A survey and taxonomy of loss functions in machine learning*. https://arxiv.org/pdf/2301.05579.pdf

Das, S., Dey, A., Pal, A., & Roy, N. (2015). Applications of Artificial Intelligence in Machine Learning: Review and Prospect. *International Journal of Computer Applications*, *115*(9), 31–41. https://doi.org/10.5120/20182-2402

Doshi, S. (2020, August 3). *Various Optimization Algorithms For Training Neural Network*. Medium. https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6

Eduardo, L. (2021, July 20). *How to create a neural network for regression with PyTorch*. GitHub. https://github.com/christianversloot/machine-learning-articles/blob/main/how-to-create-a-neural-network-for-regression-with-pytorch.md

Grant Sanderson. (2017). Backpropagation calculus | Deep learning, chapter 4. In *YouTube*. https://www.youtube.com/watch?v=tIeHLnjs5U8

Harrison, D., & Rubinfeld, D. L. (1978). Hedonic housing prices and the demand for clean air. *Journal of Environmental Economics and Management*, *5*(1), 81–102. https://doi.org/10.1016/0095-0696(78)90006-2

Jadon, A., Patil, A., & Jadon, S. (2022). *A Comprehensive Survey of Regression Based Loss Functions for Time Series Forecasting*. https://arxiv.org/pdf/2211.02989.pdf

Kaul, V., Enslin, S., & Gross, S. A. (2020). History of artificial intelligence in medicine. *Gastrointestinal Endoscopy*, *92*(4), 807–812. https://doi.org/10.1016/j.gie.2020.06.040

Marc Peter Deisenroth, A Aldo Faisal, & Cheng Soon Ong. (2020). *Mathematics for machine learning*. Cambridge University Press.

Máté, A. (2016). *Hyperbolic functions **. http://www.sci.brooklyn.cuny.edu/~mate/misc/hyperbolic_functions.pdf

Sharma, S., Sharma, S., & Athaiya, A. (2020). Activation Functions In Neural Networks. *International Journal of Engineering Applied Sciences and Technology*, *4*(12), 310–316. https://www.ijeast.com/papers/310-316,Tesma412,IJEAST.pdf

Tabrizian, P. (2019, January 11). *Derivation of cosh and sinh*. Www.youtube.com. https://www.youtube.com/watch?v=NVC1w4_ulzI

Theekshana, T. (2021, January 15). *Huber loss: Why is it, like how it is?* Medium. https://www.cantorsparadise.com/huber-loss-why-is-it-like-how-it-is-dcbe47936473

Yathish, V. (2022, August 4). *Loss Functions and Their Use In Neural Networks*. Medium. https://towardsdatascience.com/loss-functions-and-their-use-in-neural-networks-a470e703f1e9

8. Appendix

Appendix A: Derivative Calculations for Output Node 1

Solving for $\frac{\partial z_{H_1}}{\partial s_{H_1}}$:

$$z_{H_1} = a(s_{H_1})$$

$$\frac{\partial z_{H_1}}{\partial s_{H_1}} = a'(s_{H_1})$$

Solving for $\frac{\partial s_{H_1}}{\partial w_{IH_1}}$

$$s_{H_1} = W_{IH_1} \times z_I + b_{H_1}$$

$$\frac{\partial s_{H_1}}{\partial w_{IH_1}} = (W_{IH_1})^0 \times z_I$$

$$\frac{\partial s_{H_1}}{\partial w_{IH_1}} = z_I$$

Appendix B: Derivative Calculations for Output Node 2 Pathway

Solving for $\frac{\partial z_{O_2}}{\partial s_{O_2}}$:

$$L = (Y_i - \widehat{Y}_i)^2$$

$$L = (z_{O_2} - \widehat{Y}_i)^2$$

$$\frac{\partial L}{\partial z_{O_2}} = 2(z_{O_2} - \widehat{Y}_i) \times (z_{O_2})^0$$

$$\frac{\partial L}{\partial z_{O_2}} = 2(z_{O_2} - \widehat{Y_i})$$

Solving for $\frac{\partial z_{O_2}}{\partial s_{O_2}}$:

$$z_{O_2} = a(s_{O_2})$$

$$\frac{\partial z_{O_2}}{\partial s_{O_2}} = a'(s_{O_2})$$

Solving for $\frac{\partial s_{O_2}}{\partial z_{H_1}}$:

$$s_{O_2} = W_{H_1O_2} \times z_{H_1} + b_{O_2}$$

$$\frac{\partial s_{O_2}}{\partial z_{H_1}} = W_{H_1O_2} \times (z_{H_1})^0$$

$$\frac{\partial L}{\partial z_{O_2}} = W_{H_1O_2}$$

$\frac{\partial z_{H_1}}{\partial s_{H_1}}$ and $\frac{\partial s_{H_1}}{\partial w_{IH_1}}$ remains the same, hence:

$$\frac{\partial L}{\partial z_{O_2}} \frac{\partial z_{O_2}}{\partial s_{O_2}} \frac{\partial s_{O_2}}{\partial z_{H_1}} \frac{\partial z_{H_1}}{\partial s_{H_1}} \frac{\partial s_{H_1}}{\partial W_{IH_1}} = 2(z_{O_2} - \widehat{Y_i}) \cdot a'(s_{O_2}) \cdot W_{H_1O_2} \cdot a'(s_{H_1}) \cdot z_I$$

Appendix C: Code Used For Testing

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from sklearn.preprocessing import StandardScaler
from torchmetrics.regression import LogCoshError
import time
import pandas as pd
import numpy as np

data_url = "http://lib.stat.cmu.edu/datasets/boston"
raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
target = raw_df.values[1::2, 2]


class BostonDataset(torch.utils.data.Dataset):
 '''
 Prepare the Boston dataset for regression
 '''

 def __init__(self, X, y, scale_data=True):
   if not torch.is_tensor(X) and not torch.is_tensor(y):
     # Apply scaling if necessary
     if scale_data:
         X = StandardScaler().fit_transform(X)
     self.X = torch.from_numpy(X)
     self.y = torch.from_numpy(y)

 def __len__(self):
     return len(self.X)

 def __getitem__(self, i):
     return self.X[i], self.y[i]


class NeuralNetwork(nn.Module):
 '''
   Multilayer Perceptron for regression.
 '''
 def __init__(self):
   super().__init__()
   self.layers = nn.Sequential(
     nn.Linear(13, 64),
     nn.ReLU(),
     nn.Linear(64, 64),
     nn.ReLU(),
     nn.Linear(64, 1)
   )

 def forward(self, x):
   '''
     Forward pass
   '''
   return self.layers(x)
if __name__ == '__main__':
 torch.manual_seed(84) #change order of data through changing value
 dataset = BostonDataset(data, target)
 trainloader = torch.utils.data.DataLoader(dataset, batch_size=10, shuffle=True, num_workers=1)
 neuralnetwork = NeuralNetwork()
 loss_function = LogCoshError() #change loss function here
 optimizer = torch.optim.SGD(neuralnetwork.parameters(), lr=0.01)
 training_loss_values = []
 start_time = time.time()
 for epoch in range(0, 50):
   current_loss = 0.0
   for i, data in enumerate(trainloader, 0):
     inputs, targets = data
     inputs, targets = inputs.float(), targets.float()
     targets = targets.reshape((targets.shape[0], 1))
     optimizer.zero_grad()
     outputs = neuralnetwork(inputs)
     loss = loss_function(outputs, targets)
     loss.backward()
     optimizer.step()
     current_loss += loss.item()

   epoch_loss = current_loss / len(trainloader)
   training_loss_values.append(epoch_loss)
```

```
  print(f'{epoch+1}, {epoch_loss:.3f}')

end_time = time.time()
training_time = end_time - start_time

print(f'Training process has finished. Total training time: {training_time} seconds.')
```

Appendix D: Trials of the Mean Average Error Loss Function:

Trial 1 MAE		**Trial 2 MAE**		**Trial 3 MAE**	
Epoch	**Loss**	**Epoch**	**Loss**	**Epoch**	**Loss**
1	21.702	1	21.565	1	21.661
2	14.599	2	14.201	2	14.198
3	3.793	3	4.205	3	4.214
4	2.887	4	3.125	4	2.958
5	2.737	5	2.807	5	2.769
6	2.574	6	2.639	6	2.661
7	2.543	7	2.586	7	2.613
8	2.446	8	2.488	8	2.483
9	2.48	9	2.414	9	2.417
10	2.334	10	2.428	10	2.431
11	2.28	11	2.362	11	2.353
12	2.295	12	2.31	12	2.336
13	2.299	13	2.334	13	2.298
14	2.252	14	2.261	14	2.274
15	2.248	15	2.285	15	2.269
16	2.181	16	2.37	16	2.312
17	2.19	17	2.299	17	2.268
18	2.219	18	2.239	18	2.224
19	2.191	19	2.224	19	2.189
20	2.114	20	2.272	20	2.18
21	2.152	21	2.185	21	2.18
22	2.178	22	2.179	22	2.187
23	2.143	23	2.156	23	2.156
24	2.104	24	2.226	24	2.088
25	2.142	25	2.15	25	2.174

26	2.121	26	2.115	26	2.119
27	2.079	27	2.144	27	2.17
28	2.084	28	2.102	28	2.142
29	2.049	29	2.127	29	2.106
30	2.122	30	2.147	30	2.112
31	2.077	31	2.07	31	2.109
32	2.149	32	2.097	32	2.027
33	2.065	33	2.063	33	2.099
34	2.079	34	2.073	34	2.056
35	2.028	35	1.979	35	2.072
36	2.069	36	1.987	36	2.098
37	2.056	37	2.093	37	2.037
38	2.049	38	1.981	38	2.045
39	2.083	39	1.979	39	2.021
40	1.983	40	1.956	40	2.063
41	1.972	41	1.999	41	2.007
42	2.039	42	2.015	42	2.084
43	1.967	43	1.978	43	2.041
44	1.962	44	1.956	44	2.078
45	1.99	45	1.965	45	2.004
46	1.963	46	1.963	46	2.026
47	1.966	47	1.96	47	2.006
48	1.957	48	1.925	48	1.961
49	1.983	49	1.923	49	1.99
50	1.982	50	1.932	50	1.995

Appendix E: Average of Mean Average Error Loss Function:

Average MAE	
Epoch	**Loss**
1	21.64266667
2	14.33266667

3	4.070666667
4	2.99
5	2.771
6	2.624666667
7	2.580666667
8	2.472333333
9	2.437
10	2.397666667
11	2.331666667
12	2.313666667
13	2.310333333
14	2.262333333
15	2.267333333
16	2.287666667
17	2.252333333
18	2.227333333
19	2.201333333
20	2.188666667
21	2.172333333
22	2.181333333
23	2.151666667
24	2.139333333
25	2.155333333
26	2.118333333
27	2.131
28	2.109333333
29	2.094

30	2.127
31	2.085333333
32	2.091
33	2.075666667
34	2.069333333
35	2.026333333
36	2.051333333
37	2.062
38	2.025
39	2.027666667
40	2.000666667
41	1.992666667
42	2.046
43	1.995333333
44	1.998666667
45	1.986333333
46	1.984
47	1.977333333
48	1.947666667
49	1.965333333
50	1.969666667

Appendix F: Trials of the Mean Squared Error Loss Function:

Trial 1 MSE		**Trial 2 MSE**		**Trial 3 MSE**	
Epoch	**Loss**	**Epoch**	**Loss**	**Epoch**	**Loss**
1	237.488	1	270.978	1	330.043
2	75.823	2	50.477	2	96.646

3	43.438	3	29.318	3	37.484
4	42.932	4	31.854	4	41.281
5	24.57	5	54.012	5	24.937
6	23.482	6	23.464	6	20.216
7	20.153	7	21.239	7	16.925
8	30.426	8	17.652	8	18.725
9	28.921	9	17.514	9	21.824
10	19.838	10	19.247	10	12.036
11	13.383	11	16.301	11	11.291
12	15.912	12	13.055	12	18.563
13	12.401	13	11.698	13	12.373
14	10.147	14	10.665	14	9.997
15	14.65	15	10.466	15	8.632
16	12.072	16	9.93	16	12.452
17	8.85	17	9.634	17	10.461
18	7.831	18	7.782	18	7.693
19	8.475	19	6.95	19	8.478
20	7.083	20	7.689	20	7.403
21	6.707	21	6.948	21	7.092
22	7.297	22	6.623	22	5.985
23	6.738	23	7.128	23	8.09
24	6.482	24	7.036	24	8.171
25	7.548	25	6.706	25	10.811
26	8.043	26	7.584	26	10.595
27	6.835	27	6.737	27	8.459
28	5.629	28	7.166	28	6.007
29	6.435	29	7.599	29	6.425
30	6.515	30	6.648	30	5.835
31	5.553	31	5.716	31	5.534
32	6.248	32	6.2	32	7.503

33	5.095	33	5.18	33	7.334
34	5.164	34	5.546	34	5.118
35	4.399	35	5.449	35	4.694
36	5.059	36	5.432	36	5.483
37	6.57	37	8.538	37	6.535
38	6.033	38	34.215	38	5.73
39	5.149	39	8.859	39	4.318
40	4.826	40	7.892	40	4.589
41	4.437	41	6.609	41	4.991
42	4.802	42	6.57	42	5.127
43	4.244	43	7.218	43	4.576
44	4.183	44	5.778	44	4.776
45	4.176	45	5.551	45	4.782
46	4.928	46	7.408	46	5.774
47	4.436	47	5.652	47	5.284
48	3.884	48	8.754	48	4.571
49	4.027	49	6.915	49	4.45
50	4.226	50	7.447	50	5.004

Appendix G: Average of Mean Squared Error Loss Function:

Average MSE	
Epoch	**Loss**
1	279.503
2	74.31533333
3	36.74666667
4	38.689
5	34.50633333
6	22.38733333

7	19.439
8	22.26766667
9	22.753
10	17.04033333
11	13.65833333
12	15.84333333
13	12.15733333
14	10.26966667
15	11.24933333
16	11.48466667
17	9.648333333
18	7.768666667
19	7.967666667
20	7.391666667
21	6.915666667
22	6.635
23	7.318666667
24	7.229666667
25	8.355
26	8.740666667
27	7.343666667
28	6.267333333
29	6.819666667
30	6.332666667
31	5.601
32	6.650333333
33	5.869666667

34	5.276
35	4.847333333
36	5.324666667
37	7.214333333
38	15.326
39	6.108666667
40	5.769
41	5.345666667
42	5.499666667
43	5.346
44	4.912333333
45	4.836333333
46	6.036666667
47	5.124
48	5.736333333
49	5.130666667
50	5.559

Appendix H: Trials of the Huber Loss Function

Trial 1 Huber Loss		**Trial 2 Huber Loss**		**Trial 3 Huber Loss**	
Epoch	**Loss**	**Epoch**	**Loss**	**Epoch**	**Loss**
1	21.202	1	21.065	1	21.161
2	14.113	2	13.7	2	13.683
3	3.323	3	3.734	3	3.759
4	2.425	4	2.625	4	2.478
5	2.246	5	2.326	5	2.298

6	2.096	6	2.189	6	2.163
7	2.027	7	2.105	7	2.112
8	1.944	8	2.002	8	2.008
9	1.966	9	1.944	9	1.967
10	1.87	10	1.938	10	1.939
11	1.786	11	1.878	11	1.889
12	1.824	12	1.869	12	1.869
13	1.809	13	1.831	13	1.816
14	1.77	14	1.791	14	1.775
15	1.769	15	1.822	15	1.78
16	1.707	16	1.876	16	1.817
17	1.699	17	1.833	17	1.821
18	1.712	18	1.779	18	1.758
19	1.71	19	1.721	19	1.712
20	1.674	20	1.793	20	1.723
21	1.691	21	1.714	21	1.749
22	1.699	22	1.708	22	1.706
23	1.658	23	1.675	23	1.683
24	1.655	24	1.755	24	1.663
25	1.654	25	1.649	25	1.686
26	1.618	26	1.649	26	1.673
27	1.594	27	1.642	27	1.701
28	1.605	28	1.611	28	1.655
29	1.573	29	1.617	29	1.62

30	1.63	30	1.637	30	1.636
31	1.591	31	1.618	31	1.648
32	1.628	32	1.619	32	1.556
33	1.56	33	1.571	33	1.618
34	1.599	34	1.597	34	1.592
35	1.573	35	1.526	35	1.576
36	1.584	36	1.541	36	1.605
37	1.566	37	1.617	37	1.548
38	1.536	38	1.512	38	1.585
39	1.574	39	1.532	39	1.553
40	1.53	40	1.49	40	1.581
41	1.522	41	1.542	41	1.551
42	1.545	42	1.554	42	1.606
43	1.472	43	1.482	43	1.556
44	1.516	44	1.445	44	1.603
45	1.522	45	1.454	45	1.574
46	1.461	46	1.493	46	1.56
47	1.474	47	1.506	47	1.524
48	1.471	48	1.474	48	1.52
49	1.512	49	1.446	49	1.552
50	1.479	50	1.453	50	1.548

Appendix I: Average of Huber Loss Function

Average Huber Loss	
Epoch	**Loss**
1	21.14266667
2	13.832
3	3.605333333
4	2.509333333
5	2.29
6	2.149333333
7	2.081333333
8	1.984666667
9	1.959
10	1.915666667
11	1.851
12	1.854
13	1.818666667
14	1.778666667
15	1.790333333
16	1.8
17	1.784333333
18	1.749666667
19	1.714333333
20	1.73
21	1.718
22	1.704333333
23	1.672
24	1.691

25	1.663
26	1.646666667
27	1.645666667
28	1.623666667
29	1.603333333
30	1.634333333
31	1.619
32	1.601
33	1.583
34	1.596
35	1.558333333
36	1.576666667
37	1.577
38	1.544333333
39	1.553
40	1.533666667
41	1.538333333
42	1.568333333
43	1.503333333
44	1.521333333
45	1.516666667
46	1.504666667
47	1.501333333
48	1.488333333
49	1.503333333
50	1.493333333

Appendix J: Trials of the Log-Cosh Loss Function

Trial 1 Log-Cosh Loss		**Trial 2 Log-Cosh Loss**		**Trial 3 Log-Cosh Loss**	
Epoch	**Loss**	**Epoch**	**Loss**	**Epoch**	**Loss**
1	21.009	1	20.872	1	20.968
2	13.929	2	13.518	2	13.506
3	3.178	3	3.581	3	3.612
4	2.294	4	2.488	4	2.347
5	2.109	5	2.187	5	2.164
6	1.965	6	2.055	6	2.031
7	1.894	7	1.971	7	1.985
8	1.821	8	1.877	8	1.881
9	1.838	9	1.823	9	1.839
10	1.745	10	1.812	10	1.819
11	1.672	11	1.76	11	1.763
12	1.704	12	1.74	12	1.742
13	1.69	13	1.711	13	1.699
14	1.656	14	1.667	14	1.671
15	1.653	15	1.698	15	1.663
16	1.592	16	1.742	16	1.684
17	1.571	17	1.7	17	1.695
18	1.591	18	1.653	18	1.642
19	1.591	19	1.601	19	1.593
20	1.56	20	1.666	20	1.613
21	1.573	21	1.598	21	1.624
22	1.583	22	1.588	22	1.586
23	1.54	23	1.559	23	1.565
24	1.533	24	1.636	24	1.548

25	1.541	25	1.536	25	1.574
26	1.499	26	1.538	26	1.563
27	1.482	27	1.534	27	1.582
28	1.488	28	1.5	28	1.539
29	1.456	29	1.499	29	1.5
30	1.506	30	1.514	30	1.517
31	1.473	31	1.505	31	1.536
32	1.516	32	1.497	32	1.446
33	1.449	33	1.459	33	1.499
34	1.481	34	1.482	34	1.483
35	1.456	35	1.426	35	1.47
36	1.46	36	1.444	36	1.498
37	1.457	37	1.494	37	1.448
38	1.424	38	1.401	38	1.475
39	1.457	39	1.419	39	1.446
40	1.42	40	1.397	40	1.468
41	1.413	41	1.428	41	1.443
42	1.439	42	1.436	42	1.494
43	1.368	43	1.385	43	1.443
44	1.407	44	1.34	44	1.48
45	1.407	45	1.353	45	1.462
46	1.353	46	1.376	46	1.448
47	1.363	47	1.401	47	1.414
48	1.365	48	1.362	48	1.41
49	1.398	49	1.336	49	1.431
50	1.362	50	1.346	50	1.439

Appendix K: Average of Log-Cosh Loss Function

Average Log-Cosh Loss	
Epoch	**Loss**
1	20.94966667
2	13.651
3	3.457
4	2.376333333
5	2.153333333
6	2.017
7	1.95
8	1.859666667
9	1.833333333
10	1.792
11	1.731666667
12	1.728666667
13	1.7
14	1.664666667
15	1.671333333
16	1.672666667
17	1.655333333
18	1.628666667
19	1.595
20	1.613
21	1.598333333

22	1.585666667
23	1.554666667
24	1.572333333
25	1.550333333
26	1.533333333
27	1.532666667
28	1.509
29	1.485
30	1.512333333
31	1.504666667
32	1.486333333
33	1.469
34	1.482
35	1.450666667
36	1.467333333
37	1.466333333
38	1.433333333
39	1.440666667
40	1.428333333
41	1.428
42	1.456333333
43	1.398666667
44	1.409
45	1.407333333
46	1.392333333
47	1.392666667
48	1.379

49	1.388333333
50	1.382333333