Investigating the effects of differing activation functions on reducing the vanishing gradient problem

To what extent does the use of the rectified linear unit activation function, compared to the sigmoid activation function, limit the vanishing gradient problem and improve the performance of deep neural networks across the CIFAR-100 dataset?

Evan Schwartz

Subject: Computer science

Word count: 3964

CS EE World
https://cseeworld.wixsite.com/home
30/34 (A)
May 2025

Submitter info:
I didn't start this until after winter break and I still did ok. Don't stress if you're procrastinating. If you have any questions, feel free to reach out.
eschwartz813 [at] gmail [dot] com

Abstract

This paper investigates the effectiveness of the rectified linear unit (ReLU) activation function compared to the sigmoid activation function in limiting the vanishing gradient problem and improving the performance of deep neural networks. Two networks with either activation function and identical architectures were trained over 20 epochs using the CIFAR-100 dataset, which consists of 60,000 32x32 color images across 100 classes. The results demonstrated that the ReLU activation function significantly outperforms the sigmoid activation function regarding both accuracy and gradient growth. These results were supported by a paired t-test comparing the gradients of the networks (t=-8.07, p<0.05). The ReLU network exhibited consistent and substantial gradient increases across all layers, while the sigmoid network showed minimal and fluctuating gradient changes. These gradient changes led to correlating results in accuracy across the epochs. These findings suggest that ReLU is more effective than sigmoid at preventing the vanishing gradient problem, leading to better learning rates and higher accuracy in deep neural networks.

Table of Contents

Introduction

Machine learning (ML) is a field of artificial intelligence (AI) that focuses on creating systems that can improve without clear programming instructions. These systems have revolutionized the field of computer science by automating sophisticated tasks once thought to be reserved solely for human ingenuity. In particular, conversational AI like Chat GPT and Gemini have been widely praised for their human-like language processing skills, becoming the face of the AI revolution in recent years. While these models are daunting in both their size and complexity, their underlying systems are easily replicable and testable at a smaller scale.

These models rely on neural networks (NNs). Neural networks utilize supervised learning, where they are given labeled datasets and adjust their underlying factors based on the training's results to better predict the patterns in a dataset. ("What is a Neural Network? - Artificial Neural Network Explained"). A common underlying factor that is predetermined in training is the activation function, a function that is applied to neurons to introduce non-linearity in an NN's learning (SHARMA). The sigmoid function was one of the first activation functions used in NNs, but it was phased out due to a limitation known as the vanishing gradient problem where networks stop learning due to the flattened end behavior of the function (Topper). The Rectified Linear Unit (ReLU) function was introduced as a function meant to solve this problem due to its linear end behavior ("Rectified Linear Units (ReLU) in Deep Learning"). Sources in this essay were carefully chosen to objectively support an investigation of the differences in activation functions. This essay will investigate the effectiveness of these functions through the research question "To what extent does the use of the rectified linear unit (ReLU) activation

function, compared to the sigmoid activation function, limit the vanishing gradient problem and improve the performance of deep neural networks across the CIFAR-100 dataset?"

Key terms

- **Machine Learning (ML)**: A field of artificial intelligence focused on the creation of systems that can improve without explicit programming instructions.
- **Neural Network (NN)**: A system of neurons that uses weights, biases, activation functions, and training data to learn and identify patterns.
- **Feed-forward neural network:** The simplest type of neural network that has one-way data flow: the network makes predictions when training (known as a forward pass) by processing data. The NN then adjusts its parameters in a backward pass after loss is calculated.
- **Weight:** A parameter of a neural network that determines the strength/importance of a specific connection between 2 neurons.
- **Bias:** A neural network parameter that allows a neuron to shift its output value to learn patterns that are not centered around 0.
- **Activation Function:** A function applied to the weighted sum of neurons in a neural network that enables it to learn non-linear relationships.
- **Sigmoid Function:** An activation function $f(x)= \frac{1}{1+e^{-x}}$ used to compress output values between 0 and 1. It is commonly associated with the vanishing gradient problem.
- **Rectified Linear Unit (ReLU):** An activation function f(x) = max(0, x), designed to prevent the vanishing gradient problem.

- **Loss Function**: A function that calculates the difference between predicted and actual output utilized to guide the learning process.
- **Gradient:** A vector used to modify the weights and bias of a neural network to decrease loss. A crucial element of computing gradients is the derivative of the activation function, and gradients are updated by applying the chain rule through a network.
- **Backpropagation:** A method used in the training of neural networks that calculates how every individual parameter affects the loss function, allowing for efficient optimization of an NN's performance.
- **Gradient descent:** An optimization algorithm that uses the calculations from backpropagation to limit the loss function of a neural network to improve its performance.
- **Vanishing Gradient Problem:** A problem in deep learning where the gradients become extremely small and the network stops learning.
- **Epoch:** A single pass through the entire dataset during neural network training.
- **Batch Size:** The number of samples after which the model's parameters are updated.
- **CIFAR-100 Dataset:** A test set of 60,000 32x32 color images of 100 everyday objects used for neural network training.

Background

Neural networks are the most popular models used in machine learning (Nielsen). They are a system of connected neurons that use weights, biases, activation functions, and sets of training data to recognize patterns and "learn" from their sample data sets. Neural networks are

inspired by the interconnected nature of the human brain as they contain interconnected layers of neurons that communicate and produce an output ("What is a Neural Network? - Artificial Neural Network Explained"). As depicted in figure 1, a typical feed-forward neural network consists of an input layer, several hidden layers of neurons, and one output layer. The input layer receives the data to be processed, the hidden layers process the data and find patterns in a manner that is not easily evident to programmers, and the output layer takes this processed data and outputs it as the network's best guess for the given input.

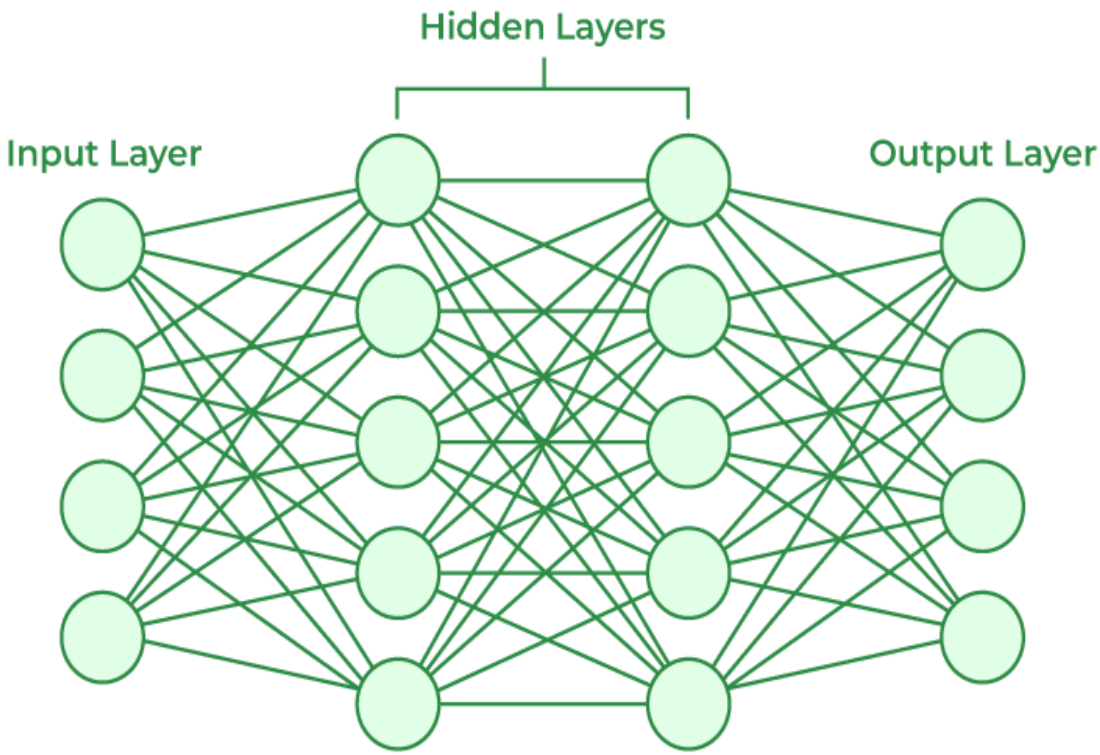

Figure 1. The general structure of a neural network

However, neural networks cannot directly analyze any data it is given if it is not numerical, so data must be converted into a processible form. (ex. RGB values of pixels in an image).

Different neurons are given “weights”, which multiply the output value of a neuron by a constant, determining the “weight” or relative significance of the neuron’s output relative to other neurons. Neurons also contain a “bias”, another adjustable parameter that controls the input required for a neuron to activate. This bias effectively adjusts the threshold at which a neuron activates. Combining all of these values results in the equation $z = \sum_{i=1}^{n} w_i x_i + b$, where x_i =input value of a neuron, w_i = weight of a neuron, b = the bias of the neuron, n = the total number of input neurons, and z= the neuron’s weighted sum (“What is a Neural Network? - Artificial Neural Network Explained”). However, this weighted sum is not the final value of the neuron, as it is then processed through an activation function to aid in the network’s learning. Activation functions are an essential part of an NN’s learning process, as they allow it to adjust to and represent complex relationships.

Activation Functions

Activation functions are used to introduce non-linearity into networks, allowing neural networks to represent non-proportional relationships (SHARMA). Common examples of these functions are the sigmoid curve, defined by the function $\frac{1}{1+e^{-x}}$ (figure 2), tanh function, rectified linear unit (ReLU) (figure 3), and leaky ReLU. As shown in figure 4, some functions compress all possible values between a range of y values, while others have no defined maximum or minimum. While a compressed function like sigmoid or tanh can make calculations and categorization easier, this compression can result in issues at its extremes. Functions with a

predetermined range like sigmoid are often used in the final layer of an NN to produce output values polarized toward 1 or 0 to help a NN make a final prediction. Activation functions are also useful in the process of backpropagation, as the derivatives of the activation functions are used to adjust the accuracy of the neural network.

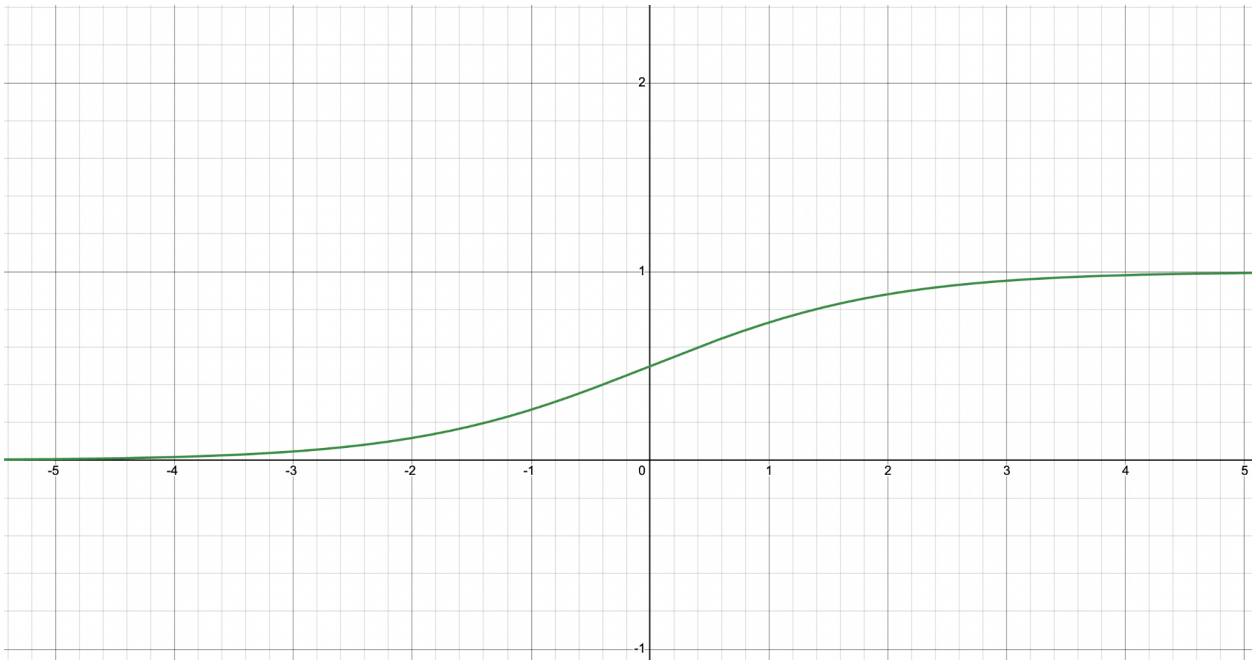

Figure 2. Sigmoid activation function

Rectified linear unit is an activation function that has been widely popularized in recent years due to its non-linearity. This helps it avoid the vanishing gradient problem, a common issue where gradients in NNs become so small the network stops learning. The function of ReLU is max (0, x), shown in Figure 3. This function means any positive input's slope is 1 and output is the inputted value, and any negative input's slope is 0 and output is 0. While a slope of 1 for all positive values may appear to not affect a network's non-linearity, ReLU's handling of negative values provides a network with the necessary non-linearity to learn and adapt to complex patterns (Brownlee). However, this same handling of negative numbers can lead to the dying

neuron problem, where neurons stop learning if they receive continuous negative inputs (Krishnamurthy and Whitfield). However, this problem occurs more frequently in datasets with a majority of negative inputs, and thus will not be as significant a concern in this testing, as the inputted RGB values are all positive.

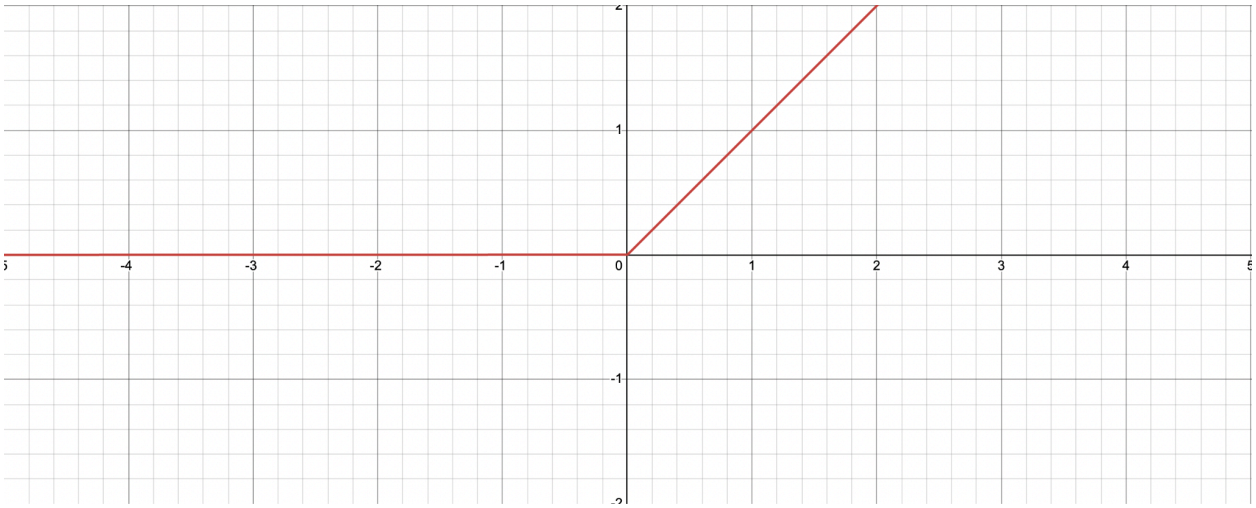

Figure 3. ReLu activation function

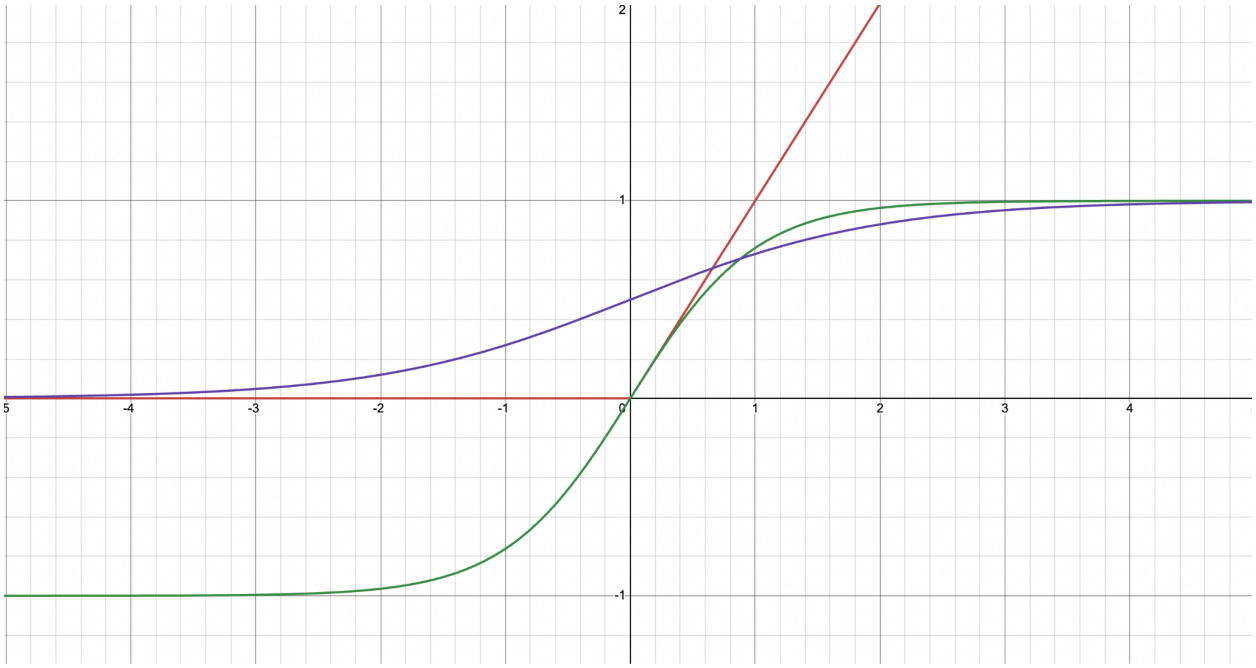

Figure 4: ReLU (red) vs. sigmoid (purple) vs. tanh (green)

Learning in Neural Networks

The process of learning in machine learning occurs through the use of large sample sets that give a neural network an input with a predetermined output. As the neural network receives more marked examples, it adjusts the weights and biases of the network and records its accuracy with these adjusted weights (Nielsen). This process will be repeated in epochs until the network reaches a certain benchmark of accuracy. The network is then tested on new examples it has never seen before to determine if the "learning" process was effective or if more refinement is needed (SabrePC). This process may need to be repeated several times for it to be effective.

After a neural network has performed calculations on a set of training examples with known values, a loss function is calculated by determining the difference between the value predicted by the neural network and the actual value of the example ("What is Loss Function?"). In short, this function quantifies the error made by the NN. A common method to calculate the loss function is Mean Squared Error (MSE), represented by the equation: $\frac{1}{N} * \sum_{i=0}^{N} (y_i - \widehat{y_i})^2$. This function takes the predicted value y_i, subtracts it from the real value $\widehat{y_i}$, and squares this difference for N number of data points. This method is used for AI tasks related to regression, or predicting a specific numerical value based on other values. Other methods can be used to calculate loss functions in neural networks for different purposes, but all neural networks share a common goal of finding the minimum of this loss function, as this means they will be the most accurate.

Gradients are a neural network's way of reducing this function. These vectors are calculated through backpropagation and are used to adjust a neural network's weights and biases. By repeatedly adjusting these weights and biases, the network begins to recognize and account for patterns in a dataset that allow it to eventually perform its task very accurately (Punnen). As long as an NN has gradients of a substantial size, it will continue to adjust its parameters and alter its performance (Nielsen). While gradients apply to both a neural network's weights and biases, for the purpose of this experiment, only the gradients of the weights will be analyzed, as they better demonstrate relationships between a network's layers and are also more susceptible to the vanishing gradient problem.

Backpropagation is a commonly used method that attempts to determine how each weight and bias individually affect the loss function. After a complete forward pass, the loss function is used to determine the error of the network. This error is then propagated through the network in a backward pass using the chain rule of calculus (Nielsen). The chain rule allows for a calculation of how each individual weight and bias of a layer affect the loss function, so they can be updated individually. For a neural network with multiple layers, the gradient of the loss L with respect to a weight w in layer l is calculated as:

$$\frac{\partial L}{\partial w^{l}} = \frac{\partial L}{\partial w^{l+1}} \cdot \frac{\partial w^{l+1}}{\partial a^{l}} \cdot \frac{\partial a^{l}}{\partial z^{l}} \cdot \frac{\partial z^{l}}{\partial w^{l}}$$

Where $z^{l} = w^{l}a^{l-1} + b^{l}$, or the weighted sum at layer l, $a^{l} = \sigma(z^{l})$ is the activation output of layer l, $\frac{\partial L}{\partial w^{l+1}}$ is the gradient of the loss with respect to the weighted sum at the next layer l+1,

$\frac{\partial w^{l+1}}{\partial a^{l}}$ is the derivative of the weighted sum at layer l+1 with respect to the activation output at layer l, $\frac{\partial a^{l}}{\partial z^{l}}$ is the derivative of the activation function at layer l, and $\frac{\partial z^{l}}{\partial w^{l}}$ is the weighted sum of layer l with respect to its weight (Venugopal).

Gradient descent is how an NN uses these calculated gradients to update its weights and biases. The general formula is $\widehat{\theta} = \theta - \alpha * \frac{\partial L}{\partial \theta^{l}}$, where $\frac{\partial L}{\partial \theta^{l}}$ is the gradient of the loss function with respect to the parameter that will be updated, α is the learning rate, θ is the original parameter to be updated, and $\widehat{\theta}$ is the updated parameter. Essentially, gradient descent determines the strength of the weight adjustment by multiplying the gradient of the loss function by the learning rate. Overall, the goal of gradient descent is to minimize the loss function by reducing its slope to find its minimum. This allows a neural network to optimize its predictions and limit loss, improving its overall performance.

The vanishing gradient problem occurs during a NN's process of backpropagation. Since the derivative of the activation function is used in the chain rule to adjust the weights of an NN, problems can arise in certain activation functions like sigmoid when this number is small. For example, the derivative of a sigmoid function at an x-value as small as 12 is .00001. When an extremely small derivative is multiplied by other factors, it can result in neurons having almost no change in weights, hindering or entirely stopping the learning of a neural network (Guide). In contrast, the derivative of ReLU for all positive values is 1, allowing for a network to make significant adjustments regardless of the size of its weighted sum.

Epochs and Batch size

When training a neural network, there are several variables to consider to optimize the network's performance. The two most important of these are batch size and number of epochs. Batch size refers to the number of samples that are calculated before the network adjusts its weights (SabrePC). A small batch size means the network will frequently change its weights and biases, while a large batch size means fewer changes will occur. While a smaller batch size may seem favorable, smaller batch sizes can make drastic changes due to limited data and hinder the progress of the NN's learning process across large datasets.

An epoch represents an NN's complete pass through the entire dataset (SabrePC). Generally, more epochs are favorable, as they allow the NN to continue its process of learning, although iteration through too many epochs can be memory intensive and result in overtraining of models that have reached very high degrees of accuracy. For the purposes of this experiment, the number of epochs will be kept constant at 20 and the batch size will remain constant at 64 so the models have the same base learning rate. This will allow the differences in activation functions to be evident.

Experimental Procedure

Two neural networks will be tasked with evaluating the CIFAR-100 dataset, a collection of 60,000 32x32 color images of common objects (Krizhevsky et al.). There are 100 different objects represented in total, ranging from common animals to automobiles, and each object has

600 accompanying photos: 500 sample photos and 100 testing photos (Krizhevsky et al.). This dataset was chosen due to the complexity of the dataset. When the data is compressed into the processing layer, a total of 3072 features are processed. This number comes from multiplying the height and width (32x32) by the three primary color channels.

This complexity is important because the vanishing gradient problem occurs more frequently and with greater consequence in large neural networks with complex inputs. An analysis of a simpler dataset like the MNIST (Modified National Institute of Standards and Technology database) dataset, a similarly sized dataset of 60,000 hand-drawn numbers 0 through 9 would not likely be sufficiently complex to produce the vanishing gradient error in a neural network with 4 hidden layers containing over 1500 total neurons (Krizhevsky et al.). The MNIST database has only 784 input features (28x28, grayscale) and 10 possible outputs, meaning the network would become extremely accurate across the 20-epoch training time, not allowing time for any noticeable gradient decay to occur. In contrast, the CIFAR-100 dataset used for this experiment allows the neural network to both incrementally learn over time and presents enough complexity to potentially produce noticeable vanishing gradients. Although both models will likely not reach high degrees of accuracy across 20 epochs due to the intricate nature of the dataset, this same complexity makes it ideal to test the relationship between activation functions and the vanishing gradient problem.

The neural network's construction and evaluation will be done using TensorFlow, an open-source Python-based machine learning platform. This platform will allow for the creation of neural networks with customizable layers and activation functions for these layers.

TensorFlow's GradientTape tool records the gradients with respect to both weights and biases for the hidden layers for each epoch, meaning the progression of the gradients with respect to weights can be easily recorded and later visualized to determine if vanishing gradients are present. Tensorflow also contains a .train function for its neural networks, which records the network's accuracy (on a scale of 0 to 1) and loss, allowing for a comparison of a network's efficacy and its gradients. This comparison will help to demonstrate the vanishing gradient's effects on a neural network's accuracy.

This experiment will utilize two identical neural networks to investigate the differences between ReLU and sigmoid. The networks will share the same structure: one input layer to flatten the 2D images into processible vectors, 4 hidden layers of 512 neurons with either activation function, and one output layer with the softmax activation function, a function designed to polarize values to 0 or 1 to ensure the neural network gives a clear answer. The only difference between the two networks will be the activation function used, allowing for a direct comparison of the efficacy of ReLU vs. sigmoid.

This experiment will investigate the differences between the ReLU and sigmoid activation functions using the CIFAR-100 dataset to answer the question "To what extent does the use of the rectified linear unit (ReLU) activation function, compared to the sigmoid activation function, limit the vanishing gradient problem and improve the performance of deep neural networks across the CIFAR-100 dataset?"

Experimental Results

After performing the experiment, the following results were obtained for both neural networks' accuracy and gradients:

	Sigmoid Layer gradient (10^{-3})					
Epochs	Layer 1	Layer 2	Layer 3	Layer 4	Average Gradient	Sigmoid accuracy
1	3.45	8.03	3.32	9.76	6.14	0.0174
2	5.60	25.06	6.25	11.07	11.99	0.0544
3	6.52	32.00	6.54	10.24	13.83	0.0725
4	6.84	33.22	7.41	10.74	14.55	0.0818
5	8.05	28.05	4.77	9.17	12.51	0.0895
6	6.07	20.28	4.44	9.41	10.05	0.099
7	6.57	24.32	5.06	9.48	11.35	0.1044
8	8.16	29.70	5.98	10.47	13.58	0.114
9	7.75	23.29	4.78	9.19	11.25	0.1155
10	9.18	23.28	5.02	9.50	11.74	0.1215
11	7.26	13.44	3.58	9.44	8.43	0.1264
12	9.30	24.02	5.87	10.04	12.31	0.1335
13	8.72	19.59	5.08	9.71	10.78	0.1392
14	8.74	17.16	4.92	9.86	10.17	0.146
15	9.74	25.54	6.17	10.73	13.04	0.1522
16	9.08	19.21	5.47	10.81	11.14	0.1563
17	9.23	23.64	6.17	11.92	12.74	0.1611
18	8.01	20.15	6.04	12.07	11.57	0.1636
19	8.22	21.45	6.57	12.32	12.14	0.1644
20	11.06	22.15	6.47	12.79	13.12	0.1703

Table 1. Sigmoid training results

	ReLU Layer gradient (10^{-3})					
Epoch	ReLU layer 1 gradient	ReLU layer 2 gradient	ReLU layer 3 gradient	ReLU layer 4 gradient	ReLU Average gradient	ReLU Accuracy
1	17.93	21.87	23.15	25.41	22.09	0.0464
2	10.72	18.67	19.23	23.16	17.94	0.1199
3	13.02	22.30	20.98	22.98	19.82	0.152
4	10.39	23.61	23.37	26.94	21.07	0.174
5	10.75	26.40	22.86	26.69	21.68	0.1984
6	12.47	27.88	25.71	29.67	23.93	0.2157
7	13.08	28.45	27.69	32.17	25.35	0.2314
8	13.92	33.36	31.96	35.88	28.78	0.2461
9	15.56	32.84	31.90	37.36	29.41	0.2565
10	17.14	34.30	33.98	40.19	31.40	0.2773
11	17.03	41.63	41.87	46.63	36.79	0.29
12	20.10	41.70	40.72	46.63	37.29	0.3121
13	17.81	43.50	44.04	50.64	39.00	0.3252
14	21.06	49.32	46.08	51.87	42.08	0.3503
15	24.26	54.11	47.48	55.86	45.43	0.367
16	25.26	50.80	50.88	54.88	45.46	0.3958
17	23.30	52.39	52.48	59.77	46.98	0.4123
18	25.87	58.14	57.32	64.52	51.46	0.4339
19	48.61	72.43	67.98	71.33	65.08	0.4594
20	18.08	57.87	58.74	65.02	49.93	0.4808

Table 2. ReLU training results

Graphing the gradients of both NNs shown in table 1 and 2 by layer, a clear trend emerges. The gradients of the ReLU NN layers show a clear trend of general growth relative to the gradients of the sigmoid NN layers, demonstrated by figure 5. Although the first layer of the ReLU function lags behind the other ReLU layers in terms of gradients, this is expected in the training of an NN due to the chain rule, and this layer still shows a pattern of overall growth.

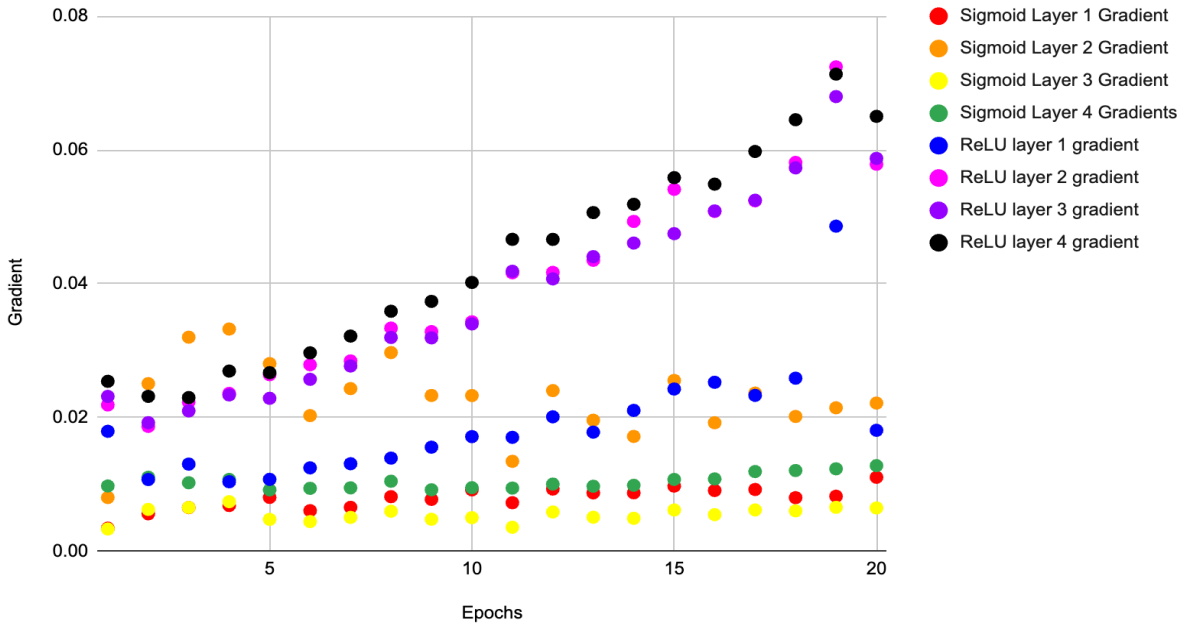

Figure 5. Gradients of ReLU NN layers vs. gradients of sigmoid NN layers

This can be more clearly seen when the NNs are divided into 2 graphs and trendlines are added to show growth. All of the ReLU layers show significant growth across the 20 epochs, and while the first layer's relatively lower growth is more clearly shown here, again, the math of backpropagation makes this an expected outcome. Overall, the gradients of the ReLU NN layers grew across the 20-epoch training period, as shown in figure 6.

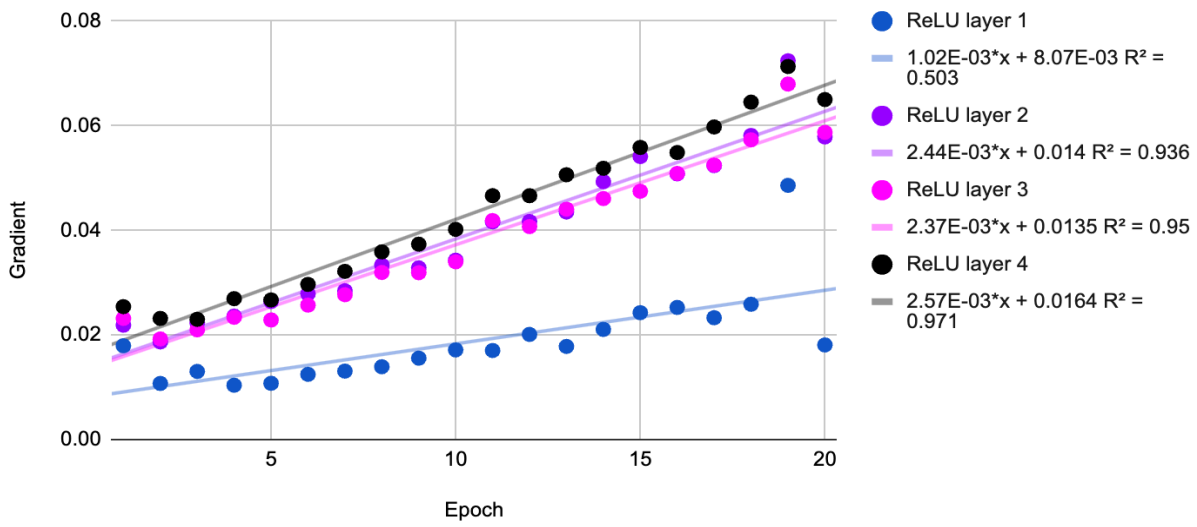

Figure 6. Gradients of ReLU NN layers

In contrast, the sigmoid NN layers demonstrated fluctuating gradients with no clear trend. As figure 7 shows, these gradients were also much smaller on average than the ReLU network's gradients, indicating this network made much smaller adjustments to its weights. Interestingly, the individual sigmoid layers did not display the behavior of the ReLU network with ascending gradients correlating with ascending layers, as the layer with the consistently highest gradient was the second layer, and the 3rd layer consistently had the smallest gradients.

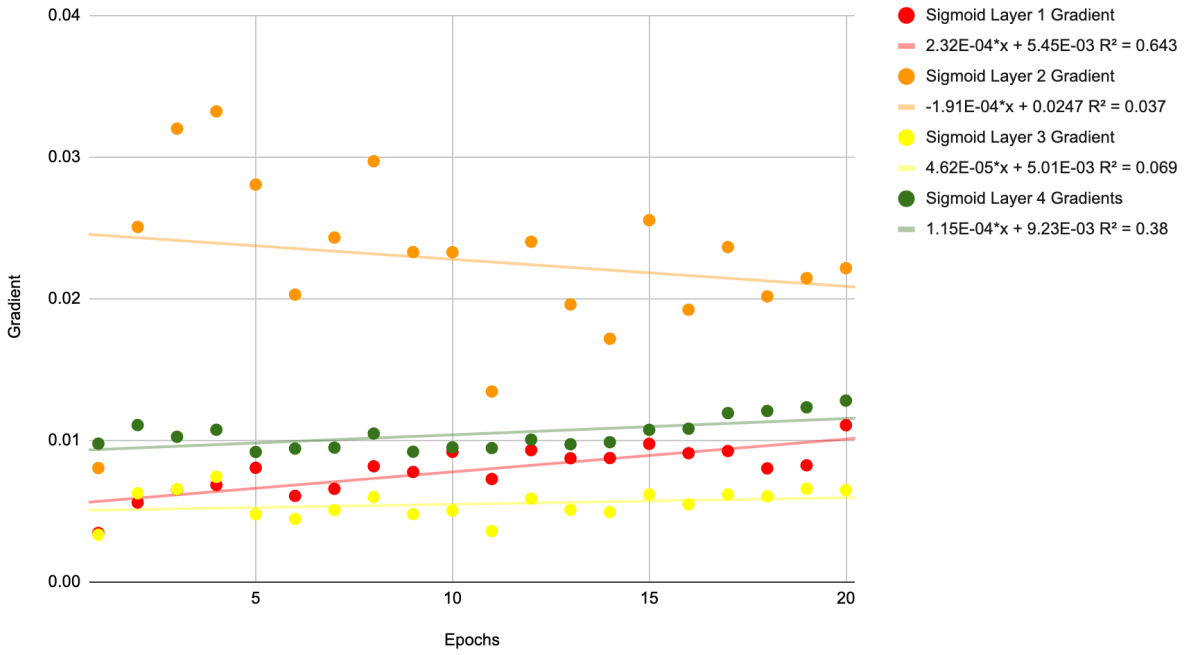

Figure 7. Gradients of sigmoid NN layers

Finally, averaging the layer's gradients for each epoch clearly shows the trends illustrated by the other graphs. Figure 8 shows that the ReLU NN's graph has a clear upward trend regarding its average gradient as epochs continue while the sigmoid NN's average gradient has no clear trend, appearing closer to a sinusoidal function than a linear function.

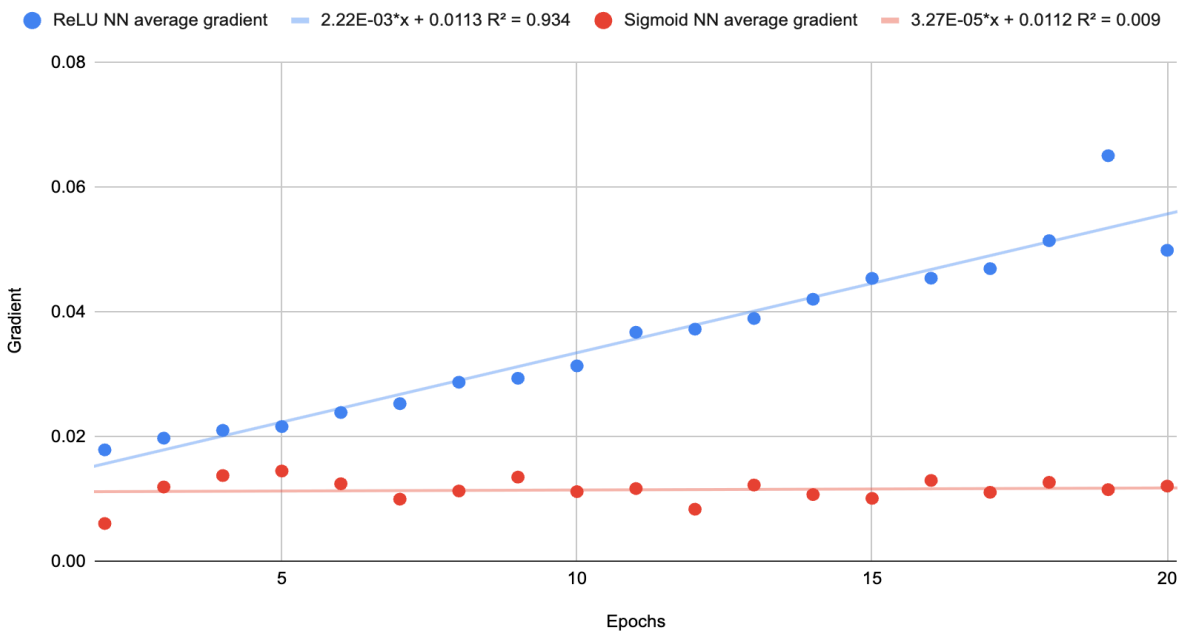

Figure 8. ReLU NN average gradient vs. sigmoid NN average gradient

A paired T-test compares the means of 2 paired data sets to determine if a significant difference between the two datasets is present. This test generates 2 values, a t-statistic, and a p-value. The t statistic represents the difference between the two datasets, and is calculated by the formula $\frac{\bar{d}\sqrt{n}}{s_d}$, where $\bar{d}$ is the mean of the differences, is n the number of pairs, and s_d is the standard deviation of the differences. A p-value is calculated by determining the odds of the difference between the datasets occurring by chance if no significant difference was present, where any value less than .05 means that a difference was likely not due to chance. A paired t-test on the average gradients resulted in a t-statistic of -8.07 and a p-value of 1.47×10^{-7}. The p-value is significantly below the significance threshold of .05, meaning the results are significant. The t-statistic of -8.07 means the gradients of the sigmoid neural network are

consistently lower than the gradients of the ReLU neural network, supporting the claim that ReLU helps to avoid the vanishing gradient problem.

Figures 5-8 establish the ReLU function made clear, increasing changes to its weights compared to the sigmoid function's minimal changes, and the results of these differing trends are clearly shown when evaluating the neural networks' accuracies.

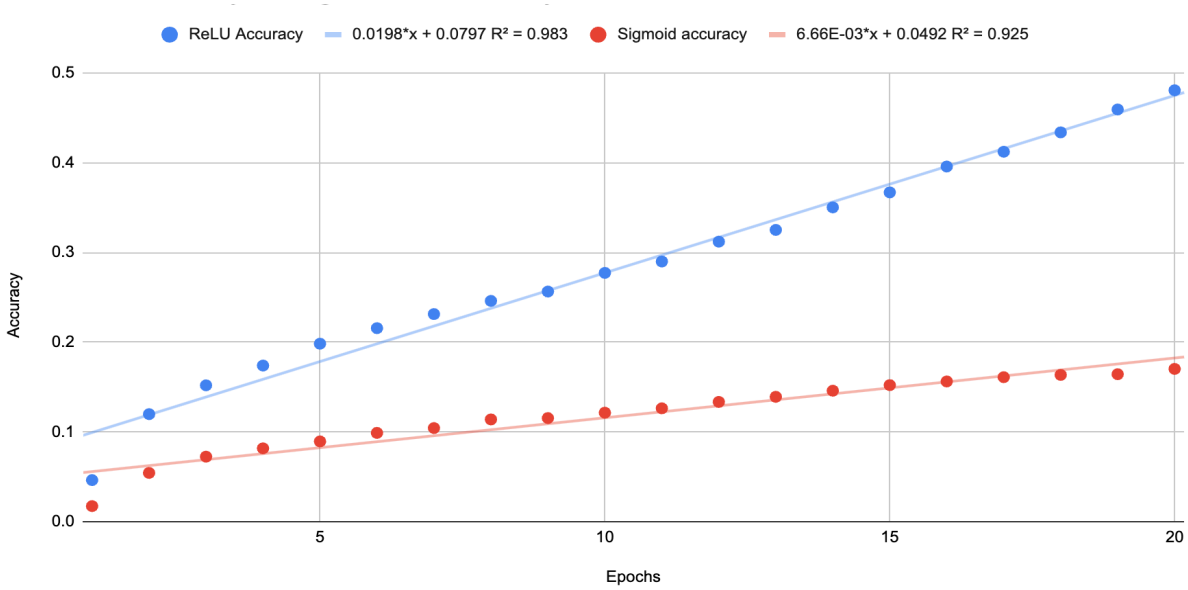

Figure 9. ReLU NN accuracy vs. sigmoid NN accuracy

Figure 9 demonstrates that the ReLU function continued to learn and improve at a constant, predictable rate, while the sigmoid function's accuracy quickly began to flatline. The networks' identical structures are evident in the first 2 epochs of each network. Both started at a very low accuracy and nearly doubled their accuracy in the second epoch. This is because, in earlier epochs, the activation function of neurons is less impactful as the network has not learned how to make useful changes. However, as epochs progressed, the learning rates of the two

networks diverged as the vanishing gradient problem appeared in the sigmoid network, while the ReLU network continued to learn at a significant rate.

Figures 8 and 9 also demonstrate a correlation between a neural network's gradients and its accuracy. This makes sense, as in earlier epochs the network is very inaccurate and needs to make consistent changes to improve its performance. This trend would likely shift as a network became more accurate. Gradients would begin to shrink as the network became very accurate, as the weights and biases of the NN would not require significant adjusting. Neither network reached these levels of accuracy, as even the more effective ReLU network was still incorrect over half of the time, although with more epochs, this level of accuracy likely would have been reached. Processing power limitations led to this experiment only investigating early epochs. However, future experiments could analyze the correlation between accuracy and gradients in the two neural networks after they have both been trained, from 20-40 epochs. Overall, these results show why the vanishing gradient problem is significant, as it results in a network almost completely stopping learning.

Conclusion

After analyzing the results, the ReLU activation function clearly mitigates the vanishing gradient problem and its associated decreased learning rate. This problem was evident in the neural network with the sigmoid activation function as a result of its extremely small slopes as it approaches its limits. The results of this experiment were similar to those found in *Algorithms for Intelligent Systems* by Akhilesh A. Waoo and Brijesh K. Soni. This article summarized the results of a comparison of the sigmoid and ReLU activation functions in evaluating the MNIST

dataset. Although this experiment had a different dataset and different neural network architecture of three hidden layers with 600, 300, and 100 nodes respectfully, this experiment generated similar results, supporting the findings of this experiment. The simpler dataset resulted in the ReLU network having an accuracy of .92 compared to the sigmoid network having an accuracy of .11 after only 10 epochs (Sheth et al. 48-49). This experiment did not record the gradients of the layers, but the significant difference in accuracy is likely due to the vanishing gradient problem.

While the experimental neural network was relatively small, containing only four hidden layers, these results demonstrate that the conditions for the vanishing gradient problem are activation function dependent. For computer scientists creating larger models, like conversational AI or large classifying models, the activation function chosen is extremely important to ensure constant learning and effective outputs. For example, Google's Gemini chatbot uses ReLU as its activation function, and ChatGPT uses GeLU, a ReLU variant for its activation function (ChatGPT guide). While sigmoid was considered an effective activation function in the early days of machine learning, the increased complexity of neural networks has rendered it obsolete in most cases due to its shortcomings, as demonstrated in this experiment. Sigmoid activation functions are still used in output layers for binary classification (Topper). Still, sigmoid clearly cannot rival ReLU in terms of results as a primary activation function, as shown by the experiment.

Works Cited

Brownlee, Jason. “A Gentle Introduction to the Rectified Linear Unit (ReLU) - MachineLearningMastery.com.” *Machine Learning Mastery*, 20 August 2020, https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/. Accessed 25 January 2025.

ChatGPT guide. “Activation functions explained.” *Chat GPT guide*, https://www.chatgptguide.ai/2024/03/03/what-is-activation-function-llms-explained/.

DataRobot. “Introduction to Loss Functions.” *DataRobot*, 2018, https://www.datarobot.com/blog/introduction-to-loss-functions/.

Guide, Step. “Vanishing Gradient Problem in Deep Learning: Understanding, Intuition, and Solutions.” *Medium*, 12 June 2023, https://medium.com/@amanatulla1606/vanishing-gradient-problem-in-deep-learning-understanding-intuition-and-solutions-da90ef4ecb54. Accessed 25 January 2025.

Jacob, Tina. “Vanishing Gradient Problem: Causes, Consequences, and Solutions.” *KDnuggets*, https://www.kdnuggets.com/2022/02/vanishing-gradient-problem.html. Accessed 25 January 2025.

Krishnamurthy, Bharath, and Brennan Whitfield. “ReLU Activation Function Explained.” *Built In*, https://builtin.com/machine-learning/relu-activation-function. Accessed 25 January 2025.

Krizhevsky, Alex, et al. “CIFAR-10 and CIFAR-100 datasets.” *Department of Computer Science, University of Toronto*, https://www.cs.toronto.edu/~kriz/cifar.html. Accessed 25 January 2025.

"MNIST Dataset." *Kaggle*, https://www.kaggle.com/datasets/hojjatk/mnist-dataset. Accessed 25 January 2025.

Nielsen, Michael. "Neural networks and deep learning." *Neural networks and deep learning*, http://neuralnetworksanddeeplearning.com/chap2.html. Accessed 25 January 2025.

Punnen, Alex. "Explaining Gradient Descent." *Medium*, https://medium.com/data-science-engineering/explaining-neural-network-as-simple-as-possible-gradient-descent-00b213cba5a9#:~:text=The%20gradient%20is%20the%20first,like%20the%20x%C2%B2%2D16%20function.&text=All%20the%20above%20are%20examples,inputs%20and%2.

"Rectified Linear Units (ReLU) in Deep Learning." *Kaggle*, https://www.kaggle.com/code/dansbecker/rectified-linear-units-relu-in-deep-learning. Accessed 25 January 2025.

SabrePC. "Epochs, Batch Size, Iterations - How are They Important to Training AI and Deep Learning Models." *SabrePC*.

SHARMA, SAGAR. "Activation Functions in Neural Networks | by SAGAR SHARMA." *Towards Data Science*, https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6. Accessed 25 January 2025.

Sheth, Amit, et al., editors. *Intelligent Systems: Proceedings of SCIS 2021*. Springer Nature Singapore, 2021. *Springer Nature*, https://link.springer.com/chapter/10.1007/978-981-16-2248-9_5. Accessed 22 February 2025.

Topper, Noah. “Sigmoid Activation Function: An Introduction.” *BuiltIn*, https://builtin.com/machine-learning/sigmoid-activation-function.

Venugopal, Puneeth. “The Chain Rule of Calculus: The Backbone of Deep Learning Backpropagation.” *Medium*, 14 October 2023, https://medium.com/@ppuneeth73/the-chain-rule-of-calculus-the-backbone-of-deep-learning-backpropagation-9d35affc05e7. Accessed 25 February 2025.

“What is a Neural Network? - Artificial Neural Network Explained.” *AWS*, https://aws.amazon.com/what-is/neural-network/. Accessed 25 January 2025.

“What is Backpropagation?” *IBM*, 2 July 2024, https://www.ibm.com/think/topics/backpropagation. Accessed 25 January 2025.

“What is Loss Function?” *IBM*, 12 July 2024, https://www.ibm.com/think/topics/loss-function. Accessed 25 January 2025.

Appendix

Code for training NNs

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Input
from tensorflow.keras.optimizers import Adam
import numpy as np

# Loads CIFAR-100 dataset
(x_train, y_train), (x_test, y_test) =
tf.keras.datasets.cifar100.load_data()

# Preprocesses the data
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
y_train = tf.keras.utils.to_categorical(y_train, 100)
y_test = tf.keras.utils.to_categorical(y_test, 100)

# Defines the 4-layer neural network
#neural network design code goes here

# Compiles the model
model.compile(optimizer=Adam(), loss='categorical_crossentropy',
metrics=['accuracy'])

# Defines a function to evaluate gradients
def evaluate_gradients(model, x_batch, y_batch):
   """
   Compute the gradient magnitudes for each trainable layer of the model.
   """
   with tf.GradientTape() as tape:
       predictions = model(x_batch, training=True)
       loss = tf.keras.losses.categorical_crossentropy(y_batch,
predictions)
   gradients = tape.gradient(loss, model.trainable_weights)
```

```
    gradient_magnitudes = [tf.reduce_mean(tf.abs(grad)).numpy() for grad in
gradients if grad is not None]
    return gradient_magnitudes

# Defines a function to train the model and record gradients
def train_with_gradient_check(model, x_train, y_train, epochs=20,
batch_size=64):
    """
    Train the model while recording gradient magnitudes after each epoch.
    """
    epoch_gradient_magnitudes = []

    for epoch in range(epochs):
        print(f"Epoch {epoch + 1}/{epochs}")

        # Trains the model for one epoch
        model.fit(x_train, y_train, epochs=1, batch_size=batch_size,
verbose=1)

        # Evaluates gradients after completing the epoch
             gradients = evaluate_gradients(model, x_train[:64],
y_train[:64])           epoch_gradient_magnitudes.append(gradients)
     print(f"Gradient magnitudes after epoch {epoch + 1}: {gradients}")

    return epoch_gradient_magnitudes

# Trains the model and records gradients
epoch_gradient_magnitudes = train_with_gradient_check(model, x_train,
y_train, epochs=20, batch_size=64)

Inspects recorded gradients
print("Recorded gradient magnitudes:")
for epoch, gradients in enumerate(epoch_gradient_magnitudes):
    print(f"Epoch {epoch + 1}: {gradients}")
```

Code for Sigmoid Neural Network

```
model = Sequential([
    Input(shape=(32, 32, 3)),
    Flatten(),
    Dense(512, activation='sigmoid'),  # First hidden layer
    Dense(512, activation='sigmoid'),  # Second hidden layer
    Dense(512, activation='sigmoid'),  # Third hidden layer
    Dense(512, activation='sigmoid'),  # Fourth hidden layer
    Dense(100, activation='softmax')  # Output layer
])
```

Code for ReLU Neural Network

```
model = Sequential([
    Input(shape=(32, 32, 3)),
    Flatten(),
    Dense(512, activation='relu'),  # First hidden layer
    Dense(512, activation='relu'),  # Second hidden layer
    Dense(512, activation='relu'),  # Third hidden layer
    Dense(512, activation='relu'),  # Fourth hidden layer
    Dense(100, activation='softmax')  # Output layer
])
```

Full training data

Epochs	Sigmoid Layer 1 Gradient	Sigmoid Layer 2 Gradient	Sigmoid Layer 3 Gradient	Sigmoid Layer 4 Gradients	Average Gradient	Sigmoid accuracy
1	0.0034481778	0.008031591	0.003320401	0.009757116	0.00613932145	0.0174
2	0.0055964724	0.025058512	0.006248479	0.011065295	0.0119921896	0.0544
3	0.0065179653	0.032004066	0.0065387823	0.010239814	0.0138251569	0.0725
4	0.0068362076	0.033220656	0.0074102306	0.010736961	0.0145510138	0.0818
5	0.008048997	0.028053336	0.0047673	0.00916826	0.01250947325	0.0895
6	0.006065887	0.020280644	0.0044352226	0.009405926	0.0100469199	0.099
7	0.0065652863	0.024315005	0.005056105	0.009475186	0.01135289558	0.1044
8	0.008157232	0.029701516	0.005975545	0.010468818	0.01357577775	0.114
9	0.007752421	0.02328613	0.0047823666	0.009185635	0.01125163815	0.1155
10	0.009177495	0.023276586	0.005019965	0.009495141	0.01174229675	0.1215
11	0.00725699	0.0134403035	0.003582324	0.009441788	0.008430351375	0.1264
12	0.009295784	0.024017021	0.005868911	0.010038979	0.01230517375	0.1335
13	0.00872263	0.019586634	0.005082843	0.009713437	0.010776386	0.1392
14	0.008737952	0.01716214	0.004921603	0.009857565	0.010169815	0.146
15	0.009739841	0.02553906	0.006165982	0.010731785	0.013044167	0.1522
16	0.009081188	0.019212274	0.0054695504	0.0108105	0.0111433781	0.1563
17	0.009233472	0.023643069	0.006173875	0.011915107	0.01274138075	0.1611
18	0.0080068745	0.020152912	0.006036827	0.012065731	0.01156558613	0.1636
19	0.00822435	0.02144588	0.006566832	0.012321877	0.01213973475	0.1644
20	0.011057269	0.022148866	0.006473418	0.012794057	0.0131184025	0.1703

	ReLU layer 1 gradient	ReLU layer 2 gradient	ReLU layer 3 gradient	ReLU layer 4 gradient	ReLU Average gradient	ReLU Accuracy
1	0.017933792	0.021872688	0.023151863	0.025407236	0.02209139475	0.0464
2	0.010723221	0.018665068	0.019226272	0.023157345	0.0179429765	0.1199
3	0.013024189	0.022301883	0.020981163	0.022980515	0.0198219375	0.152
4	0.010385009	0.023605708	0.023373744	0.02693515	0.02107490275	0.174
5	0.0107493475	0.026396556	0.02286484	0.026694983	0.02167643163	0.1984
6	0.012468219	0.02787556	0.025705632	0.029667739	0.0239292875	0.2157
7	0.013083474	0.028453883	0.027694993	0.032167673	0.02535000575	0.2314
8	0.013921377	0.033363793	0.03195653	0.035884343	0.02878151075	0.2461
9	0.015560125	0.032838702	0.031904034	0.037355546	0.02941460175	0.2565
10	0.01713644	0.034302842	0.033983856	0.040190108	0.0314033115	0.2773
11	0.017026862	0.041626073	0.041865334	0.046634305	0.0367881435	0.29
12	0.020096676	0.04169789	0.040724363	0.046626493	0.0372863555	0.3121
13	0.017805213	0.04350257	0.04404075	0.050635517	0.0389960125	0.3252
14	0.021055253	0.04931528	0.04607749	0.05187367	0.04208042325	0.3503
15	0.024257937	0.054109	0.04747878	0.05585838	0.04542602425	0.367
16	0.025260739	0.05080317	0.050879404	0.054884613	0.0454569815	0.3958
17	0.023297703	0.052390337	0.052480035	0.059771206	0.04698482025	0.4123
18	0.025873287	0.0581401	0.0573193	0.064516775	0.0514623655	0.4339
19	0.04860537	0.072425544	0.06797542	0.07133283	0.065084791	0.4594
20	0.018079668	0.057866104	0.05873999	0.06502014	0.0499264755	0.4808

Code for paired T-test

```
from scipy.stats import ttest_rel



gradients_sigmoid = [0.00613932145, 0.0119921896, 0.0138251569,
0.0145510138, 0.01250947325, 0.0100469199, 0.01135289558, 0.01357577775,
0.01125163815, 0.01174229675,
                     0.008430351375, .01230517375, 0.010776386,
0.010169815, 0.013044167, 0.0111433781, 0.01274138075, 0.01156558613,
0.01213973475, 0.0131184025]
gradients_Relu= [0.02209139475, 0.0179429765, 0.0198219375, 0.02107490275,
0.02167643163, 0.0239292875, 0.02535000575, 0.02878151075, 0.02941460175
                     ,0.0314033115, 0.0367881435, 0.0372863555,
0.0389960125, 0.04208042325, 0.04542602425, 0.0454569815, 0.04698482025,
0.0514623655, 0.065084791, 0.0499264755]

# Performs the paired t-test
t_statistic, p_value = ttest_rel(gradients_sigmoid, gradients_Relu)

# Outputs the results
print(f"T-statistic: {t_statistic}")
print(f"P-value: {p_value}")

if p_value < 0.05:
   print("Gradients are significantly different.")
else:
   print("No significant difference in gradients.")
```