Investigating Neural Scaling Laws in a Multilayer Perceptron

Student Name: Leonid Elkin
Student ID: 7058
School: Tonbridge School
School ID: 61679

April 2025

Abstract

This Extended Project Qualification investigates neural scaling laws in multilayer perceptrons (MLPs) by examining the impact of model size, learning rate, and the number of training epochs on network performance. An MLP was built from scratch using NumPy, and various configurations were tested by adjusting the number of layers, neurons, learning rate, and training epochs. Performance changes across different scaling parameters were visualized using Matplotlib. Results indicated that while performance improved with larger models, the gains exhibited diminishing returns beyond a certain scale. Increased training epochs enhanced accuracy but exacerbated overfitting in larger models, leading to decreased generalization. Higher learning rates accelerated convergence but caused instability if over-tuned, with the optimal learning rate being dependent on the model size. To validate these findings, the insights gained were used to develop an optimized model for the MNIST dataset, achieving a 9.4% reduction in error rate compared to the next best performing configuration. The aim of this investigation is to investigate if similar trends occur in different neural network architectures such as a multilayer perceptron. My research allowed me to create a highly optimized model, emphasizing the importance that such understanding can have on efficient use of computing power.

Contents

1 Introduction

1.1 Neural Networks

Neural networks form the backbone of modern machine learning, a field that has become increasingly influential across many aspects of daily life. Much like the human brain, they are highly adaptable given the right training material, capable of solving tasks ranging from simple linear regression to more complex tasks such as image classification.

At a fundamental level, neural networks process input data through complicated mathematical operations to produce outputs such as probability distributions or parameter estimates. The most basic and widely studied architecture is the multilayer perceptron (MLP), which consists of an input layer, one or more hidden layers, and an output layer. In an MLP, each neuron in one layer is fully connected to all neurons in the subsequent layer, with each connection assigned a trainable weight and each neuron associated with a bias.

Learning in neural networks occurs through the backpropagation algorithm, which iteratively adjusts weights and biases to minimize error. This project investigates how modifications to the hyperparameters involved in backpropagation — specifically the model size, learning rate, and number of training epochs — impact overall performance. Furthermore, I investigate how different mathematical functions play a role in ensuring the best possible accuracies. The objective is to explore how to balance network complexity, learning dynamics, and computational efficiency to achieve optimal results. I will then compare my network to others found on the internet.

1.2 Neural scaling laws and performance predictions

Neural scaling laws describe the relationships between model performance and factors such as model size (number of parameters) and computational resources. These laws display a general trend of how these factors can influence a model's performance.

An example of a neural scaling law investigation that I have found is a paper by Kaplan et al. (2020) titled *Scaling Laws for Neural Language Models* [5]. Their research demonstrated that, for large language models, performance improves in a predictable manner, following a power-law relationship, as model size, data, and compute resources are increased. However, they also observed diminishing returns when hyperparameters continued to increase.

While these scaling trends have been well documented for large language models, they have not been extensively explored for simpler architectures, such as multilayer perceptrons (MLPs). This gap is something I aim to address in my investigation.

1.3 Architecture choice

The multilayer perceptron (MLP) first gained popularity in the 1980s, being the first functional multi-layered neural network. It could be used for a multitude of tasks such as complex regression problems but was mainly envisioned as an image recognition system, hence the name "perceptron" [11]. However, it was soon realized that the architecture was not well suited for this task, mainly because the input layer only accepted flattened image vectors, meaning any spatial information was lost.

In 1989, Yann LeCun published a paper on convolutional neural networks (CNNs), which specialized in image recognition [6]. This was achieved by introducing a new layer type where two-dimensional vectors were used to detect edges through the convolution operation.

Although my first thought was to implement such a network, I soon realized that it was not necessary for my objectives. My aim is to emphasize the importance of proper hyperparameter tuning in order to achieve higher accuracies, rather than improve the fundamental architecture to achieve the same outcome. Furthermore, all the findings and data in this investigation could also be applied to convolutional neural networks.

1.4 Project objectives

For this project, I developed a multilayer perceptron (MLP) from scratch using the NumPy library. The primary focus was on image classification, with the MNIST dataset serving as the main test case. To evaluate whether my findings are applicable to more complex tasks, I also experimented with FashionMNIST and CIFAR-10 datasets as additional benchmarks. I used Matplotlib to generate graphs visualizing the results.

The main objective of my project is to analyze how various hyperparameters influence the network's performance. Performance is measured using two metrics: *loss*, which is derived from the evaluation (or loss) function, and *accuracy*, calculated as the percentage of correct predictions on the dataset.

As a culmination of my research, the ultimate goal is to build the most efficient and accurate model possible for the MNIST dataset based on the insights gained throughout this investigation. Understanding neural scaling laws, even for simple neural network architectures such as this one, will enable the development of more efficient and powerful AI models with lower computational costs. Throughout my research, I have encountered several promising works that were hindered by suboptimal hyperparameter tuning, which my study aims to address.

2 Literature Review

2.1 Existing works

To gauge the performance of my neural network, I reviewed the accuracy scores achieved by other researchers using Multilayer Perceptrons (MLPs) on the MNIST dataset. These benchmarks provide a point of comparison to evaluate how well my network performs in relation to established results.

On GitHub, a user named Nipunmanral achieved an accuracy of 98.26% using an MLP with a single hidden layer containing 512 neurons [8]. Another user, Sijan67, reached 97.79% accuracy with a different MLP architecture [13]. SoCalc, another contributor, managed to achieve 98.33% accuracy with their MLP [14]. In another report, SoCalc demonstrated an even higher performance with an MLP that achieved 98.37% accuracy on MNIST [15].

An official leaderboard on PapersWithCode is set up to compare the accuracies of various networks produced as results of different research papers [9]. The current top accuracy holder is a research paper on ideal group size during isometric backpropagation [3]. This paper achieved a 1.67% error rate giving an accuracy of 98.33%

These results provide a useful benchmark for my own model, with the ultimate goal being to surpass the 98.37% accuracy threshold.

2.2 Libraries

One disadvantage that came with coding in Python was the fact that it is extremely slow compared to its contemporaries. Most of the processing power of the network would be doing operations involving large vectors, so an obvious choice was the library NumPy. Apart from being much faster than Python-defined functions (NumPy is powered by C), it also operated on the entire array. This allowed me to streamline my code as there was no longer a need to iterate through each value. As I did not previously know how to use this library, I read about how it works from W3Schools [18].

I relied heavily on graphing in this investigation, therefore the Matplotlib library was an obvious choice. I had prior experience with working with this library, so there was no need to read articles about how to use it [7].

Other miscellaneous libraries that I used were TensorFlow for dataset loading [17], and Pickle for convenient saving and loading from a text file [10].

2.3 Neural network

Neural networks are quite complicated to wrap your head around, especially for a high school student; however, I found a brilliant YouTube series by 3Blue1Brown about neural networks [12]. This series featured a surface-level theory explanation and then went into detail about the mathematical implementation. Furthermore, my brother's university course on machine learning (CS3780) helped.

I understood the theory and math; however, it was difficult to realize how to structure my program to be as simple as necessary (I initially made individual objects for each neuron). Here, a book by Jan Erik Solem titled *Programming Computer Vision with Python* helped [16]. I did not like the recursive implementation of backpropagation in the book, as it introduced unnecessary complexity.

I found a YouTube video by Samson Zhang titled *Building a Neural Network from Scratch* where he goes through every step and describes it in detail [19].

When researching hyperparameters, I found a series of articles on Medium.com extremely helpful. It offers small articles about specific topics such as one-hot encoding or weight initialization techniques.

3 Methodology

The following explanations are based heavily on works by 3Blue1Brown [12], Jan Erik Solem [16], Samson Zhang [19], and articles from Medium [2].

3.1 Safety

Although programming itself does not pose life-threatening risks, excessive screen time, especially in low-light environments, can lead to eye strain and other vision-related problems. In addition, poor posture, such as slouching while seated, can result in long-term back pain and musculoskeletal issues. To mitigate these risks, I took care to limit my screen time and worked only during the day in a well-lit environment. This approach not only helped protect my eyesight but also made it easier to avoid late-night sessions, ensuring I received adequate rest for better focus and well-being.

3.2 Theory Introduction

The multilayer perceptron (MLP) is one of the most fundamental and widely used neural network models. It is a supervised, feed-forward network that processes input data to generate output predictions. Specifically, in the context of image classification, the MLP takes pixel values of an image as input and passes them through several layers of neurons, applying mathematical transformations at each stage to ultimately output a vector of probabilities corresponding to the possible classes [16] [12].

An MLP typically consists of three types of layers: the input layer, the hidden layer(s), and the output layer. Each layer is composed of neurons that are fully connected to the neurons in the next layer. The connections between neurons have weights that control the strength of the relationship between connected neurons. Additionally, each neuron in the hidden and output layers has a bias term that allows for more flexibility in the model. The input layer has as many neurons as there are features in the input data. In the case of image data, this typically means the number of pixels in the image. For grayscale images, the number of neurons in the input layer is equal to the number of pixels. For color images, the number of neurons is three times the number of pixels, since each pixel consists of three color channels (red, green, and blue). The input image is often flattened into a one-dimensional vector, meaning that spatial relationships between pixels are lost in this process [16] [12].

The purpose of the MLP is to learn from the data by adjusting the weights and biases through training, typically using a process called backpropagation and an optimization algorithm like gradient descent. By adjusting these parameters, the MLP can learn to recognize patterns in the data and make predictions, such as classifying images based on their content.

3.3 The Hidden Layers

The hidden layer(s) in a multilayer perceptron (MLP) are responsible for transforming the information from the input layer to the output layer. Each hidden layer consists of a collection of neurons, which process the input from the previous layer and pass the output to the next layer.

Each neuron in the hidden layer has two important components: a bias value b and a weight matrix W. The bias is an additional constant value that is added to the output of the neuron, allowing the model to better fit the data by shifting the output. The weight matrix determines the strength of the connections between neurons in different layers, influencing how important each connection is in passing information through the network.

Additionally, each neuron in the hidden layer uses an activation function to introduce non-linearity into the model. Without this non-linearity, the entire network would simply behave like a linear transformation, which limits its ability to model complex data patterns. The activation function applied to the input of each neuron x is often denoted as $f(x)$, which transforms the weighted sum of the inputs and the bias into an output.

Mathematically, the operation of a neuron in the hidden layer can be expressed as:

$$h = f(Wx + b)$$

where x is the input vector from the previous layer, W is the weight matrix, and b is the bias value. The function $f(x)$ introduces the necessary non-linearity to the model.

For this investigation, I will explore three commonly used activation functions: 1. Sigmoid Function:

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

The sigmoid function squashes the input to a value between 0 and 1, making it useful for models where output probabilities are needed.

2. Rectified Linear Unit (ReLU):

$$f(x) = \max(0, x)$$

ReLU is a very popular activation function that is simple to compute and allows for faster training by mitigating the vanishing gradient problem.

3. Leaky ReLU:

$$f(x) = \max(\alpha x, x)$$

Leaky ReLU is a variant of ReLU that allows small negative values for x, which helps to address issues where neurons might "die" (i.e., output zero for all inputs) in networks with ReLU activations. The parameter α is a small constant (usually a small positive value like 0.01) that determines the slope for negative inputs.

By experimenting with these different activation functions, I aim to assess their impact on the model's performance and their ability to capture complex relationships in the data.

3.4 The Output Layer

The output layer of a multilayer perceptron (MLP) generates a vector of probabilities after the forward pass, determining the predicted classes or values based on the input data. It is fully connected to the last hidden layer, meaning each neuron in the output layer receives inputs from all neurons in the preceding hidden layer.

The choice of activation function in the output layer determines the range of values for the logits (the raw, un-normalized output of the network). For binary classification, the sigmoid activation function is often used, which squashes the output to a value between 0 and 1, representing the probability of a particular class. For multi-class classification problems, the softmax function is typically used.

The sigmoid function for the output layer is expressed as:

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

This maps the output logit x to a value in the range $0 < x < 1$, which is interpreted as the probability of the positive class.

For multi-class problems, however, we often use the Softmax function to convert the vector of logits into a probability distribution. The Softmax function normalizes the logits into a set of values that sum to 1, making them interpretable as probabilities across multiple classes. It is defined as:

$$\alpha(x_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

where z_i is the i-th logit in the vector and k is the total number of classes. The Softmax function is applied to each logit, and the result is a vector where each value represents the probability of a given class, with all probabilities summing to 1.

For binary classification, you would typically use sigmoid in the output layer, while for multi-class classification, Softmax is used to ensure the outputs are valid probabilities.

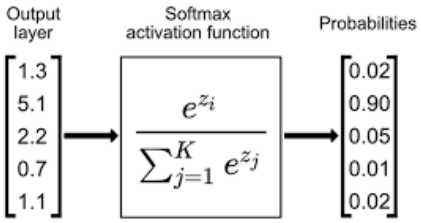

Figure 1: A demonstration of the softmax activation function

The formula for the SoftMax function is:

$$\alpha(x_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

where z_i is the i-th element of the logits vector, and $\sum_{j=1}^{k} e^{z_j}$ is the sum of the exponentials of all logits in the vector. Softmax is the only activation function in the network that does not require a derivative for backpropagation.

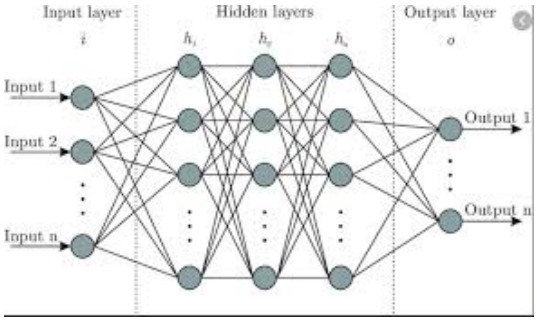

Figure 2: An image of the MLP structure

Consequently, the final class prediction y from the forward pass of the neural network can be expressed as:

$$y = \max\left(\alpha(W_2(W_1X + b_1) + b_2)\right)$$

This equation assumes a neural network with two hidden layers, where W_1 and W_2 are weight matrices, b_1 and b_2 are bias vectors, X is the input, and α represents the softmax activation function applied to the logits.

3.5 Backpropagation

The following explanations are based heavily on works by 3Blue1Brown [12], Jan Erik Solem [16] and 365 data science [1].

Weights and biases in a multilayer perceptron (MLP) function as the model's "memory." For image recognition tasks, fine-tuning these parameters allows the network to recognize patterns. The key algorithm for adjusting these parameters is backpropagation, which consists of three main stages: the forward pass, the backward pass, and gradient descent.

1. Forward Pass: The forward pass uses a training example to compute outputs from the network. The network's performance is then evaluated by a loss function. For this investigation, two loss functions are used: mean squared error (MSE) and cross-entropy loss.

2. Backward Pass: During the backward pass, the algorithm calculates the derivative of each parameter with respect to the output y. These derivatives represent how much each weight and bias needs to be adjusted to reduce the error. The calculation of these derivatives is explained during the coding process. Importantly, the gradient calculations of the backward pass are not independent, meaning simply minimizing the derivative of each parameter individually does not work. Additionally, finding the global minimum of the function algebraically is impractical because of the large number of parameters, which can reach hundreds of thousands even for simple datasets [4].

3. Gradient Descent: To address this, gradient descent is employed. This heuristic method seeks the local minimum by following the gradient of the loss function. In gradient descent, weights and biases are adjusted by subtracting a fraction of their gradient, which is scaled by a constant known as the learning rate. This nudges each parameter towards its optimal value. To increase computational efficiency, batch gradient descent is used. Instead of computing the gradient after each individual training example, it averages the gradients over a batch of examples and applies the update based on this average. This reduces computational cost and helps in better estimating the gradient [4].

4. Epochs: Once a batch of data is processed, and the weights updated, the entire dataset is passed through the network again if the model has not yet converged. The number of times the dataset is used for training is referred to as the epoch number. Repeating this process multiple times allows the model to gradually improve, reducing the loss function values and refining the network's parameters.

Once the network is trained and the loss reaches an acceptable low value, the parameters can be saved, and the network is capable of making accurate predictions on unseen data.

4 Coding

4.1 MLP Class

As per my preplanned architecture, the first step was to design the MLP class. I chose the Python programming language due to its ease of use and flexibility, along with the NumPy library to handle the more computationally expensive matrix and vector operations. I used VSCode as my integrated development environment (IDE) because it is lightweight, fast, and integrates easily with Git for version control.

4.2 Constructor

The constructor of the MLP class initializes important hyperparameters right after the creation of the object. These hyperparameters define the architecture of the network, the activation functions, the loss function, and how the weights are initialized.

The key parameters initialized in the constructor are shown below:

```
self.numLayers = len(layers)
self.layerSizes = layers
self.activationFunction = activation.lower()
self.lossFunc = lossFunc.lower()
self.weightInit = weightInit.lower()
```

Figure 3: Hyperparameters defined in the constructor

The following are the parameters passed into the constructor and their purpose.

self.numLayers Defines the total number of layers in the network, including both hidden and output layers.

self.layerSizes A list that stores the number of neurons in each layer, starting from the input layer to the output layer.

self.activationFunction Defines the activation function used for the hidden layers. It is typically one of the following: 'sigmoid', 'relu', or 'leaky_relu'.

self.lossFunc Specifies the loss function used for training the model. Common options are 'MSE' (mean squared error) and 'cross_entropy'.

self.weightInit Determines the method used to initialize the weights. Options could include 'random', 'xavier', or 'he'.

By defining these parameters in the constructor, the MLP model is configured with all necessary hyperparameters right from the start, making it adaptable to different tasks and datasets.

4.3 Weight Initialization Techniques

The architecture of the neural network is defined by the number of layers, which is passed to the model as a list of integers. The first integer corresponds to the input layer, and the last integer corresponds to the output layer. The layers in between are the hidden layers, each with a number of neurons specified by the corresponding integer in the list.

One critical aspect of training deep neural networks is how the weights are initialized. Proper weight initialization can significantly improve the performance of the network. Two popular weight initialization techniques that can boost performance are Xavier initialization and He initialization.

- Xavier Initialization: This technique works well with activation functions like the sigmoid function. It involves creating random weights, and then scaling them by the factor $\frac{1}{\sqrt{n_{in}}}$, where n_{in} is the number of neurons in the previous layer. The goal is to ensure that the variance of the outputs of the neurons is similar to that of their inputs, helping prevent issues with gradients exploding or vanishing.

- He Initialization: This technique is particularly suited for activation functions like ReLU. Like Xavier, weights are randomly initialized, but in He initialization, the weights are scaled by $\frac{2}{n_{in}}$. This helps account for the properties of ReLU activation, where half of the outputs will be zero, and scaling by this factor helps maintain the flow of information during training.

The Python code examples below show how to implement both initialization techniques.

```
weight = np.random.randn(
layers[layer], layers[layer + 1]) / np.sqrt(layers[layer])
```

Figure 4: Xavier initialization code implementation

```
weight = np.random.randn(
layers[layer], layers[layer + 1]) * np.sqrt(2 / layers[layer])
```

Figure 5: He initialization code implementation

In both code snippets, the weight matrix is initialized using a random normal distribution (using 'np.random.randn'), and then the weights are scaled according to the appropriate initialization technique. The weight matrix connects the neurons between consecutive layers, where the dimensions of the matrix correspond to the number of neurons in the two adjacent layers.

4.4 Impact of Proper Initialization

Proper weight initialization plays a key role in ensuring that the network trains effectively. Poor initialization can cause issues such as vanishing or exploding gradients, leading to slow convergence or failure to learn altogether. By using methods like Xavier or He initialization, the network can learn faster and achieve better performance, as it prevents these issues and allows for more stable training.

4.5 Basic Random Weight Initialization

If neither Xavier nor He initialization is selected by the user, the weights are created randomly. This approach, while simple, does not take into account the specific needs of different activation functions and can lead to slower convergence or issues with gradient propagation. Below is the implementation for basic random initialization:

```
weight = np.random.randn(
layers[layer], layers[layer + 1])
```

Figure 6: Basic random weight initialization

In this method, the weights are initialized using a standard normal distribution, and no scaling is applied. This is the default behavior when no specific initialization method is chosen.

4.6 Loss Function Implementation

The loss function plays a crucial role in training a neural network. It quantifies how far off the network's predictions are from the actual values, and the goal is to minimize this value during the training process. In this project, I implemented two loss functions: Mean Squared Error (MSE) and Cross-Entropy Loss.

- Mean Squared Error (MSE): This is a commonly used loss function in regression tasks and is calculated using the following formula:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - x_i)^2$$

Where: - N is the number of examples in the batch, - y_i is the true value, - x_i is the predicted value by the network.

MSE measures the average squared difference between the predicted values and the actual values. It's a good measure of accuracy when the network is used for regression tasks, and it's easy to compute, making it an attractive choice for simpler applications.

- **Cross-Entropy Loss**: This loss function is especially useful for classification problems, as it penalizes the network more when it's wrong and rewards it more when it's right. The formula for categorical cross-entropy loss is:

$$Loss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{C=1}^{C} y_{i,c}\log(x_{i,c})$$

Where: - N is the number of examples in the batch, - C is the number of classes, - $y_{i,c}$ is the true label (either 0 or 1 for each class), and - $x_{i,c}$ is the predicted probability for class c.

This loss function works well when the output is a probability distribution over several classes (like in classification tasks) and penalizes incorrect classifications more harshly than MSE. It's more focused on improving the network's certainty in its predictions and handling imbalanced classes more effectively.

The choice between these two loss functions depends on the specific task at hand. For classification tasks like MNIST, I used cross-entropy loss due to its ability to handle probabilistic outputs effectively, while for other use cases where a continuous output is expected, MSE may be more appropriate.

Below is a sample of how I implemented the loss function in the Python code:

```
def loss(self, predictions, labels):
    if self.lossFunc == 'mse':
        return np.mean((predictions - labels) ** 2)
    elif self.lossFunc == 'cross-entropy':
        return -np.mean(np.sum(labels * np.log(predictions), axis=1))
```

Figure 7: Implementation of loss functions in Python

The above code snippet shows the main loss function caller within the MLP class. The 'loss' method checks the type of loss function defined by the user (either "mean" for Mean Squared Error or "cross" for Cross-Entropy Loss) and then calls the appropriate helper function to calculate the loss. This structure allows flexibility in

```
def loss(self, predictions, labels):
    if self.lossFunc == "mean":
        return self.meanLoss(predictions, labels)
    elif self.lossFunc == "cross":
        return self.crossEntropyLoss(predictions, labels)
```

Figure 8: Loss function caller

```
def meanLoss(self, predictions, labels):
    loss = np.mean((predictions - labels) ** 2)

def crossEntropyLoss(self, predictions, labels):
    loss = -np.sum(labels * np.log(predictions)) / labels.shape[0]
    return loss
```

Figure 9: Functions responsible for loss calculations

choosing the loss function based on the specific task, providing a clean and modular way to implement both loss types.
The two functions, 'meanLoss' and 'crossEntropyLoss', handle the specific loss calculations for each type:

1. Mean Squared Error (MSE) Loss: The 'meanLoss' function calculates the Mean Squared Error between the predicted values ('predictions') and the actual labels ('labels'). The formula used is:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - x_i)^2$$

Where y_i is the true value and x_i is the predicted value for each example in the batch. The average of the squared errors across all predictions is returned as the loss.

2. Cross-Entropy Loss: The 'crossEntropyLoss' function computes the categorical cross-entropy loss. This is useful in classification tasks where the output is a probability distribution over classes. The function uses the following formula:

$$Loss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{C=1}^{C} y_{i,c}\log(x_{i,c})$$

Here, $y_{i,c}$ represents the true label (in one-hot encoded format) and $x_{i,c}$ is the predicted probability for class C. The result is averaged across the batch size.

These functions ensure that the correct loss is calculated based on the type chosen by the user, making the training process flexible and adjustable to the problem at hand.

With these two figures and the detailed explanation of each function, your methodology section provides a clear understanding of how the loss functions are integrated and used within your neural network. Would you like to continue with the training process, optimization, or evaluation metrics next?

Needed for the backpropagation algorithm, the derivatives of MSE and cross-entropy are $MSE' = 2\times(y-x)$ and $Loss' = y - x$.

4.7 Activation Functions

The activation functions chosen for this investigation are the sigmoid, ReLU, and leaky ReLU. These functions are commonly associated with image classification, although ReLU gained popularity much later than sigmoid. The formulas for sigmoid and ReLU were described earlier; however, leaky ReLU has not been. Leaky ReLU is a variation of the regular ReLU in that its gradient is never zero, ensuring that information is not lost. This is achieved by giving the entire section of the graph where $x \leq 0$ a gradient α.

Similarly to the loss functions, I chose to have one 'master' function call the others. During testing, I encountered overflow and underflow errors with dense network architectures. Since these errors were caused by large incoming activations, I decided to clip any values that exceeded a determined size, defined by `self.clipValue`.

```
x = np.clip(x, -self.clipValue, self.clipValue)
if self.activationFunction == 'sigmoid':
    return self.sigmoid(x)
elif self.activationFunction == 'relu':
    return self.relu(x)
elif self.activationFunction == 'leakyrelu':
    return self.leakyRelu(x)
```

Figure 10: Activation function caller

The same idea also applies to the derivatives of the activation functions.

```
if self.activationFunction == 'sigmoid':
    return self.sigmoidDerivative(x)
elif self.activationFunction == 'relu':
    return self.reluDerivative(x)
elif self.activationFunction == 'leakyrelu':
    return self.leakyReluDerivative(x)
```

Figure 11: Activation function derivative caller

Because of this design choice, individual activation functions were very simple. Each one accepted an array as an input and, thanks to NumPy operations, could be performed without iteration.
The derivative for the sigmoid function is simply $S'(x) = S(x)(1 - S(x))$.
The derivative for ReLU is $f'(x) = \{1 \text{ if } x > 0, 0 \text{ if } x \leq 0\}$
The derivative for leaky ReLU is $f'(x) = \{1 \text{ if } x > 0, \alpha \text{ if } x \leq 0\}$

```
def sigmoid(self, x):
    return 1 / (1 + np.exp(-x))
def relu(self, x):
    return np.maximum(0, x)
def leakyRelu(self, x):
    return np.where(x > 0, x,  x * a)
```

Figure 12: The activation functions

The softmax also needed its own method inside the class, so one was created, similar to the activation functions in terms of format.

```
def sigmoidDerivative(self, x):
    return x * (1 - x)
def reluDerivative(self, x):
    return np.where(x > 0, 1, 0)
def leakyReluDerivative(self, x, a):
    return np.where(x > 0, 1, a)
```

Figure 13: The activation function derivatives

```
def softMax(self, x):
    expon = np.exp(x)
    return expon / np.sum(expon, axis = 1, keepdims = 1)
```

Figure 14: The softmax function

4.8 Forward pass

The forward pass is a core part of this program. It will be used every time the network predicts something. It involves applying all the formulas discussed in the theory section. Although this part of the MLP is usually done through recursion, I avoided this because my architecture fundamentally didn't require one. I stored every weight and bias in a single list, which would allow me to simply use NumPy arrays to process them. This change has no effect on the time complexity of the program; however, it greatly helps with clarity.

```
def forward(self, incomingConnections, FLOPPER=False):
    self.activations = [incomingConnections]
    for layer in range(self.numLayers - 2):
        dotProduct = np.dot(self.activations[-1], self.weights[layer])
        + self.biases[layer]
        activatedDotProduct = self.activation(dotProduct)
        self.activations.append(activatedDotProduct)

    outputLayer = np.dot(self.activations[-1], self.weights[-1])
    + self.biases[-1]
    answer = self.softmax(outputLayer)
    self.activations.append(answer)
    return answer
```

Figure 15: The forward pass of the network

Furthermore, I added a counter to help estimate the amount of floating-point operations done by the computer during this process. It is called every time the forward or backward pass is run.

It finds the number of floating-point operations by multiplying every dimension together and adding it to a cumulative variable, `self.CPUUsage`. This estimate is not fully accurate, as it assumes that each parameter requires only one floating-point operation, but the result is proportional to the actual FLOP value.

4.9 Backpropagation algorithm implementation

This is the backward pass of the MLP class. It takes in expected values for the data on which the forward pass was performed. It uses the same `self.activations` list as the forward pass. The targets are assumed to be one-hot encoded. The `FLOPPER` variable toggles CPU usage recording. It functions by calculating the derivative of every variable relative to the loss, then traversing down the gradient by the value of the gradient $\times$ the value of the `lr` (learning rate) variable.

4.10 Training method

The training loop of my network has 13 parameters excluding `self`. Five of them are Boolean toggles, and you can run the function with only seven filled in. The purpose of all these parameters is to facilitate different

```
def countFlops(self, FLOPPER):
    if FLOPPER:
        m, n = self.activations[-2].shape
        p = self.weights[-1].shape[1]
        self.CPUUsage += 2 * m * n * p
```

Figure 16: FLOP estimation function

```
def backward(self, targets, lr):
    outputLayerError = self.activations[-1] - targets
    error0 = [outputLayerError]
    for index in range(self.numLayers - 2, 0, -1):
        error = np.dot(error0[0], self.weights[index].T)
        * self.activationDerivative(self.activations[index])
        error0.insert(0, error)

    for index in range(self.numLayers - 1):
        weightGradient = np.dot(self.activations[index].T, error0[index])
        biasGradient = np.sum(error0[index], axis=0, keepdims=True)
        self.weights[index] -= lr * weightGradient
        self.biases[index] -= lr * biasGradient
```

Figure 17: The backpropagation algorithm used for this network

research options with the same network.

```
def train(self, inputs, targets, numEpochs, lr, batchSize,
testInputs, testLabels, lrDecay=False, decayRate=0.95,
avgLossToggle=False, recordUsage=False, epochLosses=False, stopper=False):
    indexes = inputs.shape[0]
    lossList = []
    usageSuperlist = []
    epochLossesList = []

    print(f"beginning training with  {numEpochs}
    epochs and layer sizes {self.layerSizes}")
```

Figure 18: Initialization of the training loop

After setting up parameters, the function shuffles the dataset and runs the three-step backpropagation algorithm. It also records the average loss for each epoch, and there is an option to record CPU usage. Additionally, it features an option for learning rate decay optimization, which allows quicker convergence with a higher initial learning rate. The `stopper` variable controls the option to prematurely stop if the loss decreases too slowly.

Finally, depending on which parameters were toggled, the system returns different statistics about its training. `LossList` returns the loss for every epoch, whereas `epochLossesList` returns the loss for testing examples that the system hasn't seen before. Two different loss lists are used later to measure overfitting.

4.11 Other methods

The prediction method runs the forward pass and returns the index of the largest value.

Lastly, the save and load functions allowed the network parameters to be portable. This meant that I didn't have to re-train the network every time I ran it. I used the `pickle` library for this because it makes it very easy to store list items in a text file.

Both of these store all the necessary parameters of the network, however, the MLP object has to be created first, meaning that temporary values for layer sizes and activation functions need to be passed in.

```
for epoch in range(numEpochs):
    lossEpochList = []
    shuffledIndexes = np.random.permutation(indexes)
    Inputs = inputs[shuffledIndexes]
    Labels = targets[shuffledIndexes]

    if epochLosses:
        testPredictions = self.forward(testInputs)
        testLoss = self.loss(testPredictions, testLabels)
        epochLossesList.append(testLoss)
        print(testLoss)

    for batch in range(0, indexes, batchSize):
        batchInputs = Inputs[batch: (batch + batchSize)]
        batchLabels = Labels[batch: (batch + batchSize)]

        predictions = self.forward(batchInputs, recordUsage)
        loss = self.loss(predictions, batchLabels)
        lossEpochList.append(loss)

        self.backward(batchLabels, lr, recordUsage)

    if lrDecay:
        lr *= decayRate
```

Figure 19: Batch system setup and training

```
if recordUsage:
    return usageSuperlist

if epochLosses:
    if avgLossToggle:
        return lossList, epochLossesList
    else:
        return epochLossesList
else:
    if avgLossToggle:
        return lossList
```

Figure 20: Return of statistics

```
def predict(self, inputs):
    predictions = self.forward(inputs)
    return np.argmax(predictions, axis=1)
```

Figure 21: Prediction method

```
def save(self, filename):
    import pickle
    file = open(filename, 'wb')
    pickle.dump({
        'layerSizes': self.layerSizes,
        'weights': self.weights,
        'biases': self.biases,
        'activationFunction': self.activationFunction,
        'clipValue': self.clipValue,
        'weightInit': self.weightInit
    }, file)
    file.close()
```

Figure 22: The save function

```
def load(self, filename):
    import pickle
    file = open(filename,'rb')
    data = pickle.load(file)
    self.layerSizes = data['layerSizes']
    self.numLayers = len(self.layerSizes)
    self.weights = data['weights']
    self.biases = data['biases']
    self.activationFunction = data['activationFunction']
    self.clipValue = data['clipValue']
    self.weightInit = data['weightInit']
    file.close()
```

Figure 23: The load function

5 Neural scaling law research

5.1 Plotting

For plotting, I chose to use matplotlib as it is the most widely used library, and I had prior experience with it. I developed two plotting functions: one for plotting a moving average and the other for plotting with two-axis input. Both were used in my investigation to represent data effectively.

```
def plotMovingAverage(dataList, depth, multiplier=
, labels=None, logScale=False,start=0):
    plt.figure(figsize=(10, 6))

    for i, data in enumerate(dataList):
        movingAvg = np.convolve(data, np.ones(depth) / depth, mode='valid')
        plt.plot(np.arange(len(movingAvg)) * multiplier+start
        , movingAvg, label=labels[i])

    if logScale:
        plt.yscale('log')
        plt.xscale('log')

    plt.xlabel("Layer Size")
    plt.ylabel("Accuracy")
    plt.grid()
    plt.legend()
    plt.show()
```

Figure 24: Function responsible for plotting a graph with a moving average

In the moving average program, I utilized the convolve function alongside an array of ones. This allowed me to achieve high efficiency, as I didn't have to iterate over the items in the list every time a new item was added.

```
def plotPolynomialWithTwo(data, logScale = False):
    plt.figure(figsize=(10, 6))
    for item in data:
        y, x = zip(*item)
        x = np.array(x)
        y = np.array(y)
        plt.plot(x, y)
    if logScale:
        plt.yscale('log')
        plt.xscale('log')
    plt.xlabel('FLOP/s')
    plt.ylabel('Loss')
    plt.title('Loss vs Compute graph for different neural networks')
    plt.grid(True)
    plt.show()
```

Figure 25: Function responsible for plotting a graph using both x and y value inputs

5.2 Main File Structure

```
import numpy as np
import matplotlib.pyplot as plt
import pickle
```

Figure 26: Function loading

Here are some of the libraries I will use in the main file. Others may be loaded depending on the need. For example, three datasets are used during the investigation, and loading all three every time the program is running is a waste of space and time. Therefore, I have decided to add a function for loading each dataset. They are all very similar, with the exception of the first two lines. As an example, this is the function to load the MNIST dataset:

```
def loadMNIST():
    from tensorflow.keras.datasets import mnist
    (trainingImages, trainingLabels), (testingImages, testingLabels) = mnist.load_data()
    trainingImages = trainingImages.reshape(-1, 28 * 28) / 255.0
    testingImages = testingImages.reshape(-1, 28 * 28) / 255.0
    trainingLabels = convertIntoFormat(trainingLabels, 10)
    testingLabels = convertIntoFormat(testingLabels, 10)
    return trainingImages, testingImages, trainingLabels, testingLabels
```

Figure 27: The function responsible for loading the MNIST dataset

The image is saved as a flattened list of length. To help with debugging and to demonstrate that the dataset loading works, I made a simple function using PyPlot to display the image.

```
def showImg(flatImage, label):
    image = flatImage.reshape(28, 28)

    plt.figure(figsize=(4, 4))
    plt.imshow(image, cmap='gray')
    plt.axis('off')
    plt.title(f"label: {label}")
    plt.show()
```

Figure 28: A function to display an item in the dataset

As a demonstration here is it displaying a five using the following code:

```
showImg(trainingImages[0], np.argmax(trainingLabels[0]))
```

Figure 29: Code required to draw figure 29

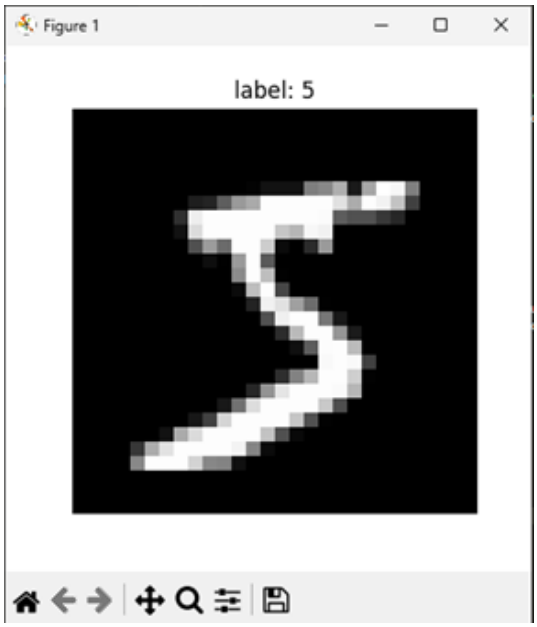

Figure 30: A '5' of the MNIST database

Image labels need to be one-hot-encoded so this function handles that.

```
def convertIntoFormat(labels, size):
    format = np.zeros((len(labels), size))
    for i, label in enumerate(labels):
        format[i][label] = 1
    return format
```

Figure 31: The function responsible for one-hot-encoding

The answers generated need to be checked, so this function compares two lists and gives an accuracy rating based on how similar items in the list are to each other by taking an average of a list of ones and zeros generated.

```
def checkAccuracy(lst1, lst2):
    result = []
    for a,b in zip(lst1,lst2):
        if a == b:
            result.append(1)
        else:
            result.append(0)
    return np.mean(np.array(result))
```

Figure 32: The function responsible for one-hot-encoding

5.3 Running the MLP

This is a basic program to run the MLP

It trains a small neural network with 64 neurons on the MNIST dataset. Then it tests it on the training dataset. It is very simple to set up, similarly to using external libraries like PyTorch. This particular example has a surprisingly high accuracy of 96%, however it was still lower to the 98%+ the ideally trained networks give. Refer to the literature review section for examples.

```
trainingImages, testingImages, trainingLabels, testingLabels = loadMNIST()
LAYERS = [784, 64, 10]
epochs = 2
lr = 0.01
batchSize = 50
mlp.MLP(LAYERS, 'relu', 500, 'he', 100, 'mean')
mlp.train(trainingImages, trainingLabels, epochs, lr,
batchSize, testingImages, testingLabels, True, 0.9)
answers = np.argmax(testingLabels, axis=1)
results = mlp.predict(testingImages)
accuracy = checkAccuracy(results, answers)
print(f"Final accuracy: {}")
```

Figure 33: Simple code for initializing, training and testing the neural network.

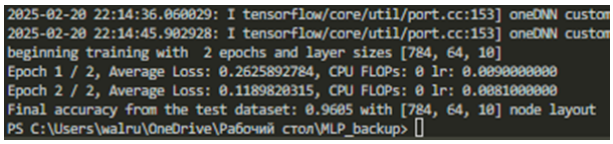

Figure 34: The console log produced by the network training process

5.4 HyperparameterChanges functions

For investigating neural scaling laws, I made a large function that runs every experiment and plots a relevant graph.

```
def hyperparameterChanges(variable, start = 0, end = 0, step = 0, start2 = 0
, end2 = 0, step2 = 0)
```

Figure 35: Parameters of the hyperparameterChanges function

It takes in seven parameters. The 'variable' parameter decides which test to run. The function of the other parameters varies from experiment to experiment. The 'resultList' is used to store results and is always plotted at the end. The structure of the function consists of many if and elif statements, checking if the variable matches a specific string. The details of this function will be discussed in the investigation chapter.

6 Investigation

6.1 Finding Ideal Functions for 28x28 Image Recognition

For the entire experiment, I used a batch size of 50 to speed up convergence. Batches are necessary to improve convergence speed because instead of adjusting gradients for every image, the algorithm adjusts them based on the average of the gradients. Additionally, I clip values where $x \leq 1000$ or $x \geq -1000$ to avoid overflow and underflow errors. For example, a x value of -1100 would produce an estimated gradient of 10^{307}, which exceeds the floating point limit of Python and would most certainly cause an error when encountered. Even without this fail-safe, overflows would most likely not occur if weights and biases are initialized using He and Xavier initialization, because these functions aim to prevent the exploding and vanishing gradient problem.

6.1.1 Activation Functions

The three activation functions I used in this investigation were Sigmoid, ReLU, and Leaky ReLU. To determine which function was most suitable for this task, I examined the accuracy of each function across different network sizes. Specifically, I tested a network with 64 neurons in one layer to evaluate performance on a smaller network, and a network with 600 neurons to evaluate performance on a larger network.

Both Sigmoid and ReLU have inherent challenges. The Sigmoid function suffers from the vanishing gradient problem, where very large or small values cause underflow errors in gradients. This leads to slower learning, and in some cases, no learning at all. On the other hand, the ReLU function has the "dying ReLU" problem, where the gradient is zero for all x values less than zero, resulting in a loss of information. This makes the network inherently biased toward positive weights. The Leaky ReLU function mitigates this issue by allowing for a non-zero gradient at negative values, ensuring that the network does not lose information.

To exacerbate these issues, I used a network with 64 neurons split across two layers. This configuration helps demonstrate the effects of consecutive layers, where incoming connections are more likely to be large or small. The purpose of this experiment was to evaluate how each activation function behaves under different conditions.

The experiment involved running networks with each of the three activation functions and observing the accuracy and loss for each case. This experiment was conducted over ten epochs, with a learning rate of 0.01 to prevent instability. The alpha value for Leaky ReLU was set to 0.01. The results of the experiment will be presented, including the final accuracy of each network after training. Additionally, an epoch-to-loss graph was generated to visualize the convergence of each function. Notably, at around 20 epochs, the network begins to overfit, meaning the loss curve may no longer represent how well the model generalizes to unseen data. Thus, the accuracy on the full dataset was also used as a more reliable measure of performance. This experiment was completed on my laptop in approximately five minutes.

```
elif variable == 'activation':
        activationFunctions = ['sigmoid','relu','leakyrelu']
        layers = [[784,600,10],[784,64,10],[784,32,32,10]]
        accuracyList = []

        for layer in layers:

            resultList = []
            for function in activationFunctions:
                if function == 'sigmoid':
                    epochs = start
                    mlp = MLP(layer,function,maxAllowableOutput,'None')
                    loss = mlp.train(trainingImages,trainingLabels,epochs,lr,
                    batchSize,testingImages,testingLabels,avgLossToggle=True)
                    resultList.append(loss)
                else:
                    epochs = start
                    mlp = MLP(layer,function,maxAllowableOutput,'None')
                    loss = mlp.train(trainingImages,trainingLabels,epochs,lr,
                    batchSize,testingImages,testingLabels,avgLossToggle=True)
                    resultList.append(loss)

                results = mlp.predict(testingImages)
                accuracy = checkAccuracy(results,answers)
                accuracyList.append(accuracy)
            print(resultList)
            plotMovingAverage(resultList,2,batchSize,activationFunctions)
        print(accuracyList)
```

Figure 36: Function segment responsible for plotting loss vs training examples for different activation functions

7 Investigation Results

7.1 Experiment with Activation Functions

The initial step of the experiment was conducted without initialization techniques. The string inputs 'xavier' and 'he' would be set to 'None' in this case.

```
hyperparameterChanges("activation", start = 10)
```

Figure 37: The code snippet ran to generate the next three figures

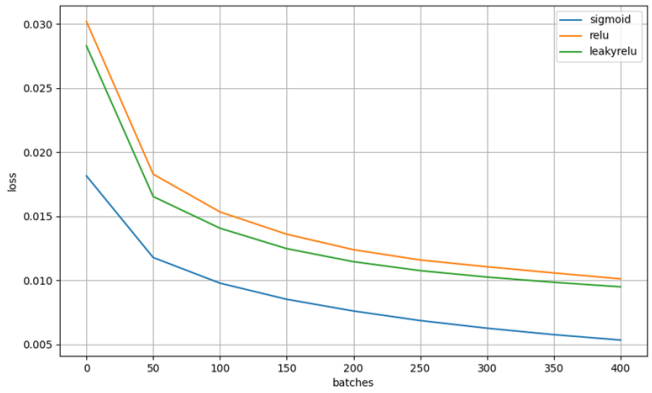

Figure 38: Loss vs batches processed graph for different activation functions in a single layered MLP

This graph assumes a single layer with 64 neurons. As seen, the Sigmoid activation function converges much faster than ReLU and achieves an almost zero loss. This quick convergence is clearly attributed to overfitting, which prompted me to test each network on the testing dataset. The final accuracies of each function are as follows:

Sigmoid	96.26%
ReLU	95.38%
Leaky ReLU ($\alpha = 0.002$)	95.68%

As shown, Sigmoid outperforms ReLU and Leaky ReLU. Moreover, Leaky ReLU performs better than the standard ReLU. The conclusion drawn from these results is that for a small, one-layered MLP, the ideal activation function is Sigmoid.

For the 32x2 neural network, the loss graph clearly indicates that ReLU faced significant stability issues due to the dying ReLU problem. Although Leaky ReLU performed better, Sigmoid still outperformed both, as shown in the graph and the following accuracy table:

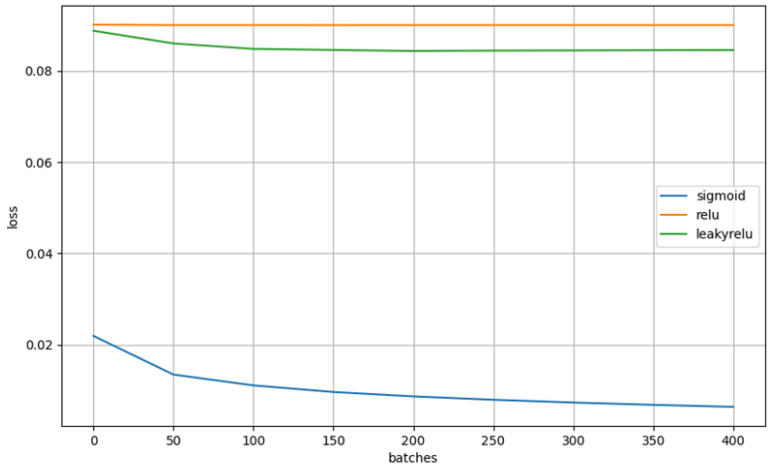

Figure 39: Loss vs batches processed graph for different activation functions in a multilayered MLP

Sigmoid	94.84%
ReLU	9.74%
Leaky ReLU ($\alpha = 0.002$)	14.39%

The accuracies of the different networks clearly demonstrate that the ReLU activation struggles with multiple layers, with standard ReLU failing to learn at all and Leaky ReLU achieving a mere 14.4% accuracy. In contrast, the Sigmoid function performed significantly better with a 94.8% accuracy. This further supports the conclusion that the Sigmoid activation function is superior to ReLU in multi-layered networks.

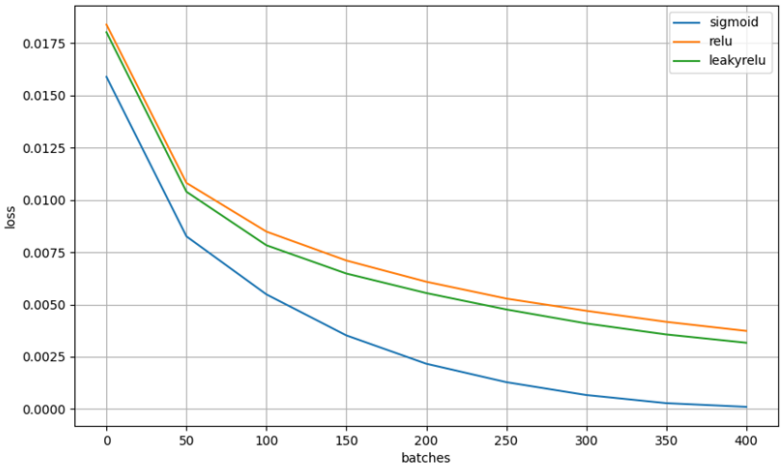

Figure 40: Loss vs batches processed graph for different activation functions in a large single-layered MLP

The experiment with one layer consisting of 600 neurons showed similar results to the one with 64 neurons. The sigmoid function again outperformed both ReLU and leaky ReLU.

Sigmoid	94.93%
ReLU	96.28%
Leaky ReLU ($\alpha = 0.002$)	95.57%

However, the accuracies on the testing dataset presented a different outcome. ReLU demonstrated better accuracy than Sigmoid, which had worsened compared to the 64-neuron test. This is likely due to the faster convergence of the sigmoid function, which caused overfitting after more epochs were used. As a result, the model performed worse on the testing dataset. The normal ReLU function even outperformed leaky ReLU. Therefore, for large, one-layered networks, ReLU is the better option. The ReLU + large one-layered perceptron achieved the highest accuracies, so I decided to use that architecture to maximize performance.

7.1.1 Optimal Learning Rate

The learning rate plays a significant role in this investigation. A learning rate that is too high can cause instability and inaccuracies in convergence, while a rate that is too low results in excessively slow convergence. The goal is to identify the optimal learning rate that balances both factors. For this experiment, a single layer with 64 neurons, a batch size of 50, and a learning rate of 0.01 were used.

The first graph was generated using the sigmoid activation function with the following code.

```
hyperparameterChanges("lr", 0, 1, 0.01, "sigmoid")
```

Figure 41: The code snippet ran to generate the next figure

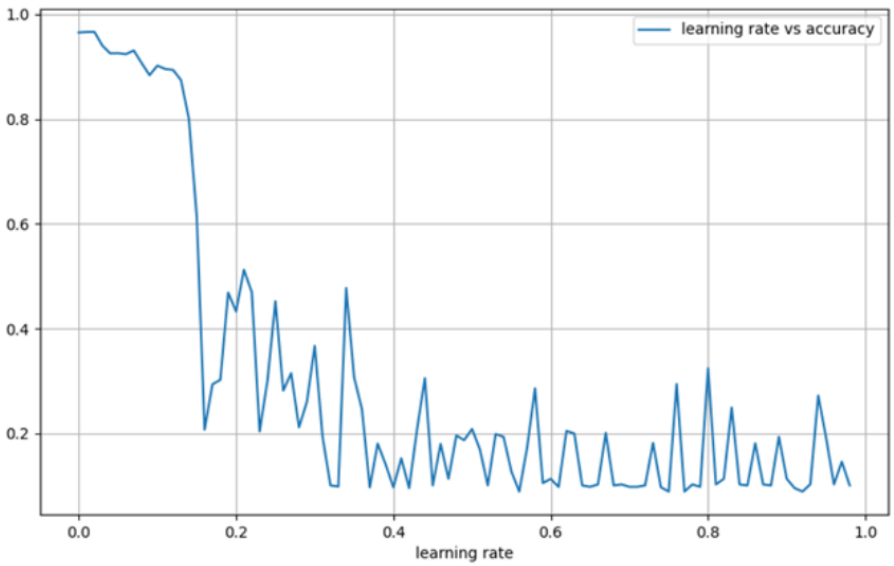

Figure 42: Learning rate vs accuracy for the sigmoid activation function

On this graph, you can see a clear boundary between neural network stability and instability. It occurs just after a learning rate of 0.01. What is particularly interesting is the continuous decrease in accuracy after a very early point on the learning rate axis. To further understand this behavior, I generated a higher-resolution graph to provide more insight.

```
hyperparameterChanges("lr", 0, 0.2, 0.001, "sigmoid", step2 = 3)
```

Figure 43: The code snippet ran to generate the next figure

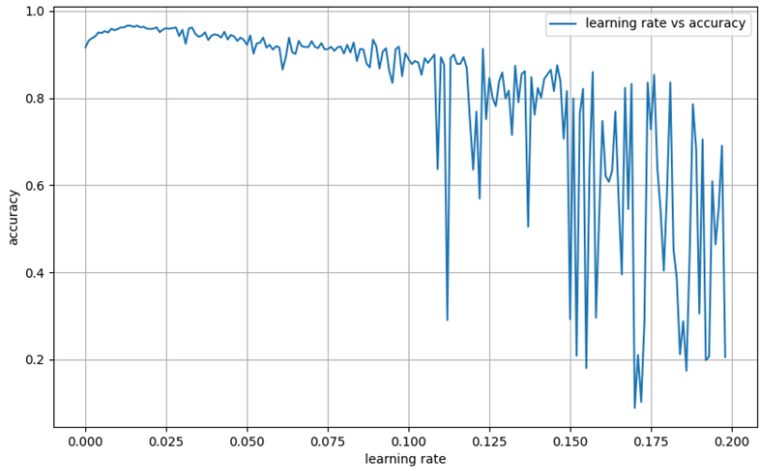

Figure 44: Finer learning rate vs accuracy for the sigmoid activation function

The finer-grained graph reinforces the idea that severe instability begins once the learning rate exceeds 0.1. It also illustrates that the accuracy initially increases as the learning rate grows. However, after reaching a point around 0.015, accuracy starts to drop. This suggests that convergence is achieved around a learning rate of 0.015, and any increases past this value lead to over-adjustments during gradient descent, which prevents the network from properly settling into a local minimum. Of course, the optimal learning rate changes with the number of epochs, however this experiment shows that any value over 0.025 becomes increasingly unstable until a complete collapse after 0.1.

Furthermore, the ReLU activation function was found to be more sensitive to higher learning rates than the sigmoid, as shown by this graph. In fact, only one point in the ReLU graph remained within the stable region.

```
hyperparameterChanges("lr", 0, 0.5, 0.01, "relu", step2 = 3)
```

Figure 45: Learning rate vs accuracy for the ReLU activation function

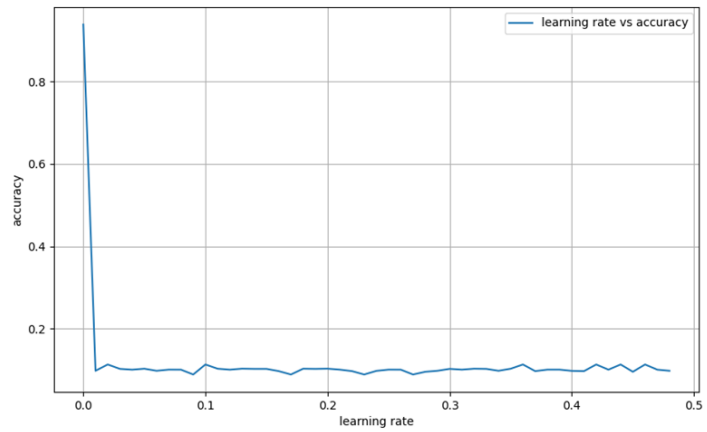

Figure 46: Learning rate vs accuracy for the ReLU activation function

There was only one point of stability in the entire graph, prompting the production of a higher resolution graph. In this new graph, each point is spaced only 0.0001 learning rate units apart. w

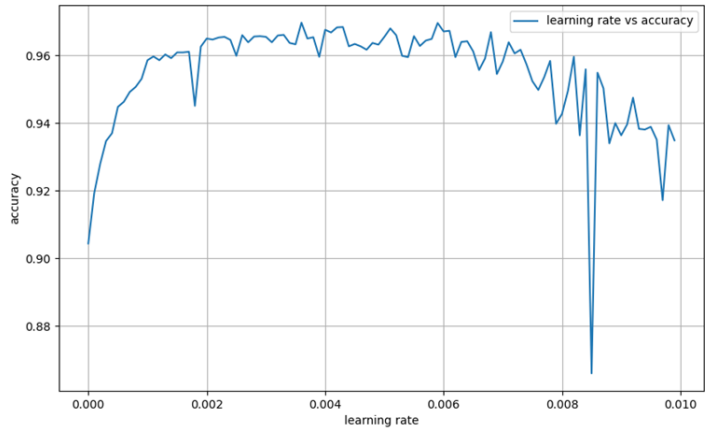

Figure 47: Finer learning rate vs accuracy for the ReLU activation function

The same pattern is observed here: after convergence is achieved, accuracy decreases in proportion to the learning rate. There is one randomly unstable point, but this can be regarded as a rare anomaly.

In summary, while the ideal learning rate value varies depending on the number of epochs used, the general range is $x \leq 0.1$ for sigmoid and $x \leq 0.01$ for ReLU.

7.1.2 Learning rate decay

The issue with high learning rates is that convergence happens too quickly, potentially causing the algorithm to "miss" the local minimum. Ideally, the convergence should be rapid at the start and then slow down as the network nears the minimum. To address this, learning rate decay was implemented from this point forward. My approach to learning rate decay decreased the learning rate by 10% at each epoch. This adjustment allowed for the use of higher initial learning rates without leading to instability.

```
if lrDecay:
    lr *= decayRate
```

Figure 48: The code snippet responsible for learning rate decay

7.2 Effects of epoch number on accuracy

7.2.1 Epoch number and overfitting

Similarly to learning rate, the epoch number is extremely important during training. The optimal epoch number depends on the number of parameters in the network, as it is capable of learning more complex patterns and hence requires more learning to converge. In this experiment, I will use the previously explained structure of a large single-layered perceptron. Even though the optimal epoch number will be different from other experiments, the relationship between epochs and accuracy will be universal.

I will be using the following block of code to generate the graphs.

```
elif variable == 'epochs':
    epochs = end
    mlp = MLP(LAYERS,'relu',maxAllowableOutput,'he',lossFunc='cross')
    losslist,trainingLossList  = mlp.train(trainingImages,trainingLabels,
    int(epochs),lr,batchSize,testingImages,testingLabels,True,0.9,True,
    False,True)
    plotMovingAverage([losslist,trainingLossList],2,
    labels=['loss recorded on training dataset',
    'loss recorded on testing dataset'])
```

Figure 49: The segment of the hyperparameterChanges function responsible for plotting an epoch vs loss graph

```
hyperparameterChanges("epochs", end = 100)
```

Figure 50: The line of code required to generate the figure below

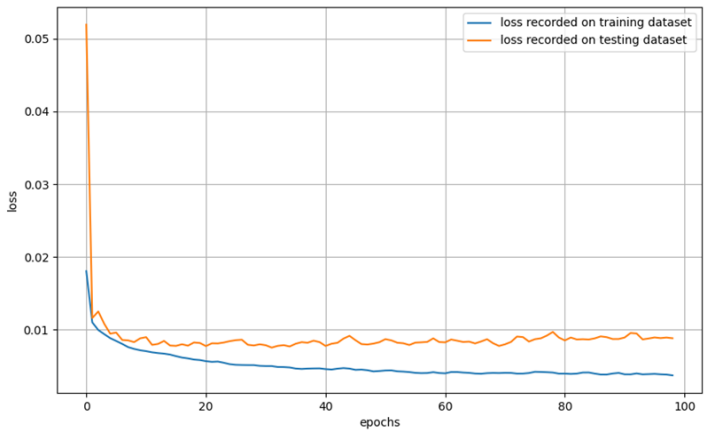

Figure 51: Training and testing loss vs epoch number

The final result was this graph. The loss recorded on the training dataset always decreases, however, the loss recorded on the testing dataset decreases rapidly at first but then increases. This is a perfect example of overfitting. As the network is trained more and more on the same dataset, it starts to form extremely complex patterns that only relate to that specific collection of images. This means that it will start to struggle to generalize to new examples. This graph uses a single layer with 400 neurons. With even more complex networks, this issue is more pronounced as more complex patterns can be formed.

7.3 Loss functions

The two loss functions I have decided to use for the experiment are MSE and cross-entropy loss. Their formulas are listed in the theoretical explanation chapter. In theory, MSE is better at 'punishing' anomalies in results and cross-entropy is better when used with tables of probabilities during classification, both of which I encounter. I will generate the accuracies using both loss functions on the previously described 400-neuron network. The learning rate I will be using is 0.01 with the ReLU activation function. The same random seed was used to make the comparison fair.

```
elif variable == 'lossFuncs':
    epochs = end
    funcs = ['mean','cross']
    for func in funcs:
        mlp = MLP(LAYERS,'relu',maxAllowableOutput,'he'
        , lossFunc=func,seed = 100)
        mlp.train(trainingImages,trainingLabels,epochs
        , lr,batchSize,testingImages,testingLabels)

        results = mlp.predict(testingImages)
        accuracy = checkAccuracy(results,answers)
        resultList.append(accuracy)

    print(resultList)
```

Figure 52: The code segment of the hyperparameterChanges function that tests different loss functions

```
hyperparameterChanges("lossFuncs", end = 6)
```

Figure 53: The line required to generate the next figure

Mean square error	93.97%
Cross-entropy	95.96%

At 6 epochs the accuracy of MSE was 93.97% whereas the accuracy for cross-entropy loss was 2% higher at 95.96%.

```
hyperparameterChanges("lossFuncs", end = 10)
```

Figure 54: The line required to generate the next figure

Mean square error	86.06%
Cross-entropy	95.02%

At 10 epochs the accuracy of MSE was 86.06% whereas the accuracy for cross-entropy loss was 95.02%. MSE clearly converged too fast and overfitted. The result being that cross-entropy loss is 9% better than MSE in this scenario.

```
hyperparameterChanges("lossFuncs", end = 2)
```

Figure 55: The line required to generate the next figure

Mean square error	92.83%
Cross-entropy	95.89%

At 2 epochs, MSE is 3% worse than cross-entropy with it having a 92.83% accuracy and the latter having a 95.89% accuracy. Cross-entropy loss seems better in every way than MSE, so it will be used for the remainder of the investigation.

7.4 Number of neurons per layer

As previously explained, the ideal configuration for maximizing accuracy is one layer with many neurons. The final neural scaling law that is left to investigate is the effect of scaling the physical size of the network. To do this I made the following block of code.

```
elif variable == 'LayerSize':
    epochs = start2
    for layerSize in range (start,end,step):
        LAYERS[1] = layerSize
        mlp = MLP(LAYERS,'relu',maxAllowableOutput,'he'
        ,lossFunc='cross',seed=100)
        mlp.train(trainingImages,trainingLabels,epochs,lr,batchSize
        ,testingImages,testingLabels,lrDecay=True,decayRate=0.9)
        results = mlp.predict(testingImages)
        accuracy = checkAccuracy(results,answers)
        resultList.append(accuracy)
    plotMovingAverage([resultList],1,multiplier=step
    ,labels=["Layer size vs accuracy given"],start=start)

return None
```

Figure 56: Code segment of the hyperparameterChanges function that plots a loss over neuron count graph for a one-layered MLP

This graph was generated with 10 epochs, 0.01 learning rate, and the cross-entropy function. w

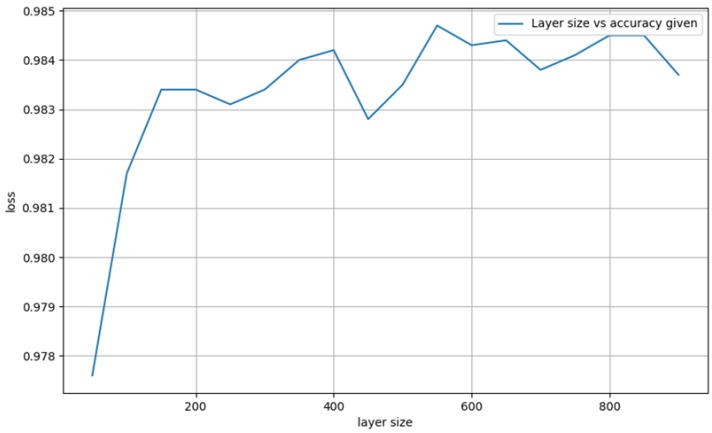

Figure 57: Layers vs accuracy of the neural network

Even though I used the same random seed for all readings, there appears to be some noise. The general trend that is observed is that accuracy increases with the number of neurons up until 550, after which it starts dropping. A finer resolution was used to see the point at which accuracy hits the maximum value. Due to the model being quite large with almost half a million parameters, I have chosen to increase the epoch number to fifteen.

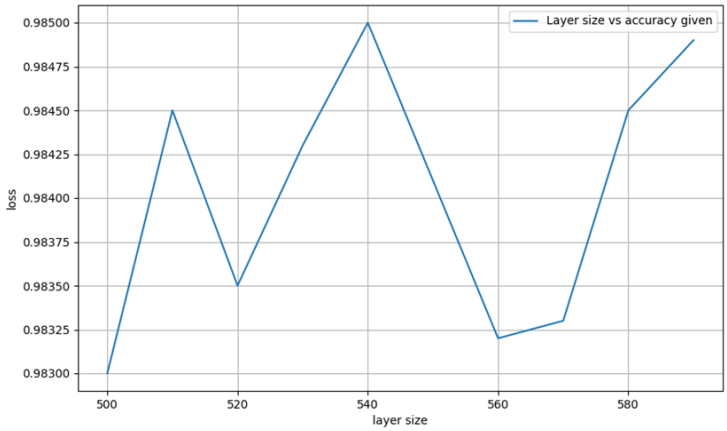

Figure 58: Finer layers vs accuracy of the neural network

There seems to be a large spike at 540. A neural network with a layer size of 540 was run with a learning rate of 0.01 and 15 epochs. I also utilized a learning decay of 0.9 per epoch and a standard batch size of 50. The final accuracy was 98.5%. Accuracy here goes as high as 98.5%, which beats every previously seen example. Upon looking at the original graph again, I realized that I had only generated up to 950 neurons per layer. Because of the random noise, they might have better accuracies. I generated a graph of up to 1400 neurons per layer.

```
hyperparameterChanges("LayerSize",500,1500,100,150)
```

Figure 59: The line required to generate the next figure

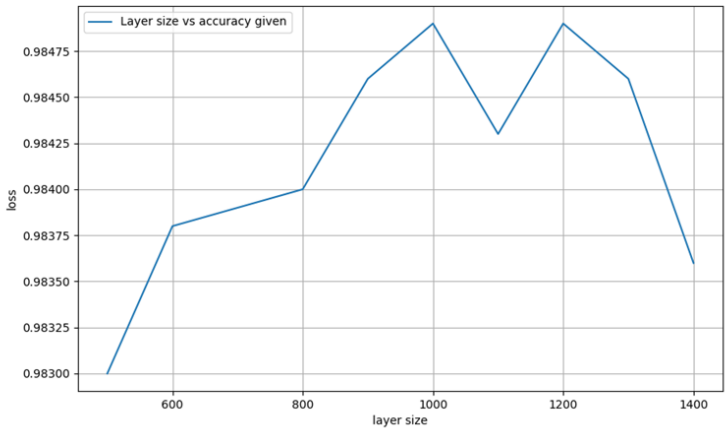

Figure 60: Broader layers vs accuracy of the neural network

I was right to assume that there could be more local maximums. I generated a higher resolution graph to analyze the first peak of the graph.

```
hyperparameterChanges("LayerSize",900,1100,25,15)
```

Figure 61: Finer graph plot focusing on the second peak of the previous figure

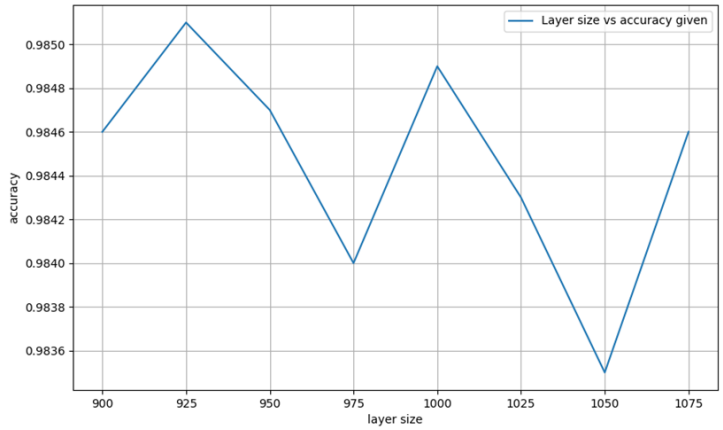

Figure 62: Broader layers vs accuracy of the neural network

And it turned out that I can achieve an accuracy 0.01% better using 925 neurons. This is the best possible MLP for 15 epochs. It takes around a minute and a half to train and has 735,375 parameters.

8 Compute Efficiency Frontier

In the previously mentioned study by Kaplan et al. (2020), a phenomenon was noticed where there seemed to be a boundary between how many computational resources are used against the loss of the network. This became known as the compute efficiency frontier, the theoretical boundary between what's computable and what is not. I have decided to do a simple visualization and investigate the implications of this strange relationship, and to see if similar results may be attainable with an MLP as with an LLM. I began by making another subsection in the hyperparameterChanges function.

```
elif variable == 'EfficiencyFrontier':
    width = start
    epochs = 100
    flopResults = []
    for layerN in range (start2,end2,step2):
        while width < end:
            layers = [784]
            for i in range (layerN):
                layers.append(width)
            layers.append(10)
            mlp = MLP(layers,'relu',maxAllowableOutput,'he',lossFunc='cross')
            flopList = mlp.train(trainingImages,trainingLabels,epochs,lr
            ,batchSize,testingImages,testingLabels,True,0.95,True,True
            ,False,True)
            print(flopList)
            flopResults.append(flopList)
            width += step
        print(flopResults)
        plotPolynomialWithTwo(flopResults,True)
```

Figure 63: A segment of the hyperparameterChanges function responsible for plotting the compute efficiency frontier

This program would use the MLP class's FLOP counter to estimate the CPU usage while plotting it against the loss outputted every epoch. I will use the standard learning rate of 0.01 and the cross-entropy loss function.

```
hyperparameterChanges("EfficiencyFrontier",1,3,1,32,2000,100)
```

The following graph displays the performance of 14 different machine learning models plotted against the computational cost of training them.

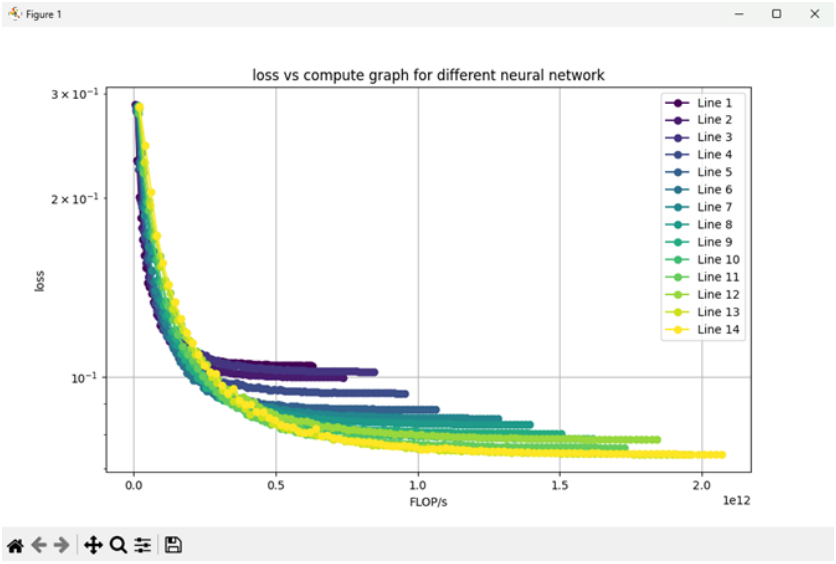

Figure 64: Plot of FLOPs required to complete training vs the loss of the final epoch of each model (log axis)

Switching the axes into log mode, a clear straight boundary is observed.

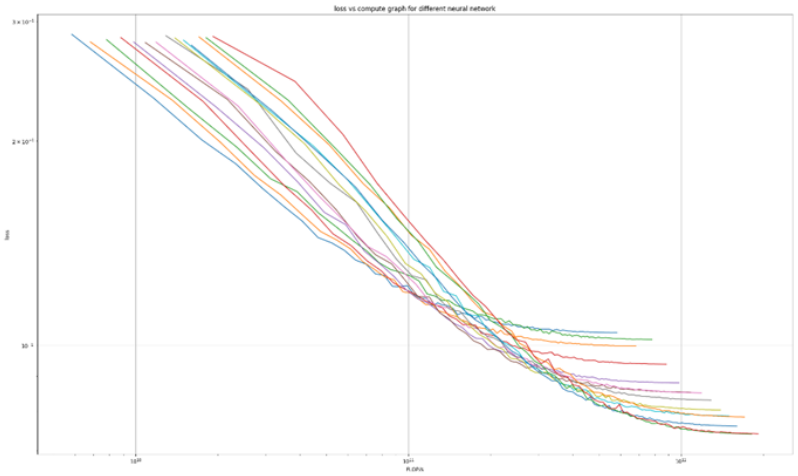

Figure 65: Plot of FLOPs required to complete training vs the loss of the final epoch of each model (log axis)

I would love to get more lines onto this graph, however, this takes a huge amount of processing power from my laptop, which has already spent a lot of time generating this graph. One of the implications of this is that loss can never be eliminated. A straight line in a logarithmic plot will never reach zero. Another implication is that there might be some fundamental boundary in how we understand and use artificial intelligence which limits the efficiency of using processing power. The last thing this graph shows is the diminishing returns of increasing the size of the network, as even in a logarithmic plot all the lines eventually level out. A larger neural network will reach lower loss values than a small one, however, it will require exponentially more compute. Hence, there is a mathematical limit to how efficiently compute can be utilized, it is important to optimize the structure of the network.

9 Findings Summary

The following neural scaling laws were investigated as well as the suitability of different functions in image recognition using MLPs.

The best loss function for flexibility and usability alongside multiple layers was the sigmoid function together with the Xavier initialization method. However, for pure accuracy, ReLU was better in the circumstance when there is only one large neuron layer.

The model size affected how complex the patterns the network could learn. A network that is too simple is unable to learn the defining features of the dataset; however, one that is too large overfits and finds patterns where they don't exist, hence also giving false results. Specifically, accuracy rapidly increases at the start, then slows down, and finally decreases. Larger models are generally more prone to overfitting because the patterns captured can be more complicated.

The number of neuron layers causes instability in the ReLU function. The sigmoid activation function avoids that; however, the accuracy is still worse than that of a single-layer perceptron.

The accuracy of an MLP at first increases with the amount of training data being used. However, if data is reused for many epochs, the network overfits, and accuracy decreases.

The learning rate affected the speed of convergence. Too low of a learning rate causes the network to use excessive computational resources, however, a learning rate too high causes instability, especially with the ReLU activation function. The maximum stable learning rate for sigmoid was found to be 0.1, and for ReLU, it was 0.01.

The ideal loss function in every scenario was found to be cross-entropy loss as it is specialized in classification tasks like this one. It was more resilient to low as well as high epoch rates.

My research is proven correct by my good accuracy on the MNIST dataset. My accuracy of 98.51% was significantly larger than the previous best I could find, which was 98.37%. That means that my MLP made 9.4% fewer errors than the next best (1.49% vs 1.63%).

The compute efficiency frontier has been visualized. Even though I would like more data points to support my claim, it showcases fundamental limits of artificial intelligence, like diminishing returns of scaling, which has been mirrored by my other investigation results.

Glossary

Activation function A mathematical operation that applies to every outgoing connection in a neuron. Details can be found in the Theory section.

Backpropagation A heuristic algorithm to find the local minimum of extremely complex mathematical functions. Details about its function can be found in the Theory chapter.

Bias value A value that every neuron has and is added to every outgoing connection. Details can be found in the Theory chapter.

Classification The process of the neural network choosing one of the provided options based on an input.

Compute The resource of computing power.

Compute efficiency frontier A theoretical boundary between the computational resources used and the performance (loss) of a neural network, showing diminishing returns as more computational power is applied.

Convergence The process of reaching the local minimum during backpropagation.

Dying ReLU problem The problem with ReLU activation where values less than zero are lost.

Epoch The amount of time the same dataset is reused to train a single model.

Evaluation function A way to assess the quality of a neural network's output. Details can be found in the functional explanation chapter.

FLOP Floating point operation. The number of floating point operations is the amount of computer resources used.

Hidden neuron layer A set of neurons that are all fully interconnected with the previous and next layers. For more details see the Theory chapter.

Hyperparameters Values that affect the training of the MLP. An example of that is the learning rate.

Input neuron layer A layer of neurons that inherits its values from the input data. For more details see the Theory chapter.

Learning rate (LR) A coefficient used in the gradient descent step of backpropagation.

Logit A raw set of values before the application of the SoftMax function to turn them into probabilities.

Loss A value of the quality of a neural network's output. Low loss does not necessarily indicate good accuracy.

MLP Short for multilayer perceptron. The fundamental machine learning model.

MNIST dataset A collection of thousands of 28x28 greyscale images of hand-drawn digits.

Neural scaling laws General trends that come with changing neural network hyperparameters.

Neuron An object that stores a bias value and that receives and processes incoming connections and relays it to outgoing connections.

One hot encoding A data representation technique where separate columns represent classes and a one in a class signifies its presence. In the case of image recognition, the labels of the class are one-hot encoded in a way that the number five would be represented by a list of zeros of length ten with a singular one at index 4.

Output neuron layer A layer of neurons that serves as the output to the network. The size of the output layer is equivalent to the number of classes present. For more details see the Theory chapter.

Overfitting A byproduct of reusing the same data too many times. If a network is powerful enough it can detect extremely complex patterns that are specific to that particular set of examples. This will prevent it from generalizing well.

Rectified linear unit (ReLU) A specific activation function. Details can be found in the Theory section.

ReLU activation instability The issue that arises when using the ReLU activation function, particularly when large learning rates lead to unstable training, especially with multiple layers.

Sigmoid A specific activation function. Details can be found in the Theory section.

SoftMax function A way to convert from a set of logits into a set of probabilities that sum to one.

Training instability The process when the backpropagation algorithm struggles to achieve convergence causing bad accuracy.

Vanishing gradient problem A problem with the sigmoid activation function where gradients far from $x = 0$ become tiny, and information is lost because of underflow errors.

Weight value A value that every connection has and determines its magnitude or 'importance.' Details can be found in the Theory chapter.

References

[1] 365 Data Science. Backpropagation explained: A complete guide for beginners. https://365datascience.com/trending/backpropagation/, 2024. Accessed: 2025-04-28.

[2] Multiple Authors. A guide to hyperparameters in neural networks. https://medium.com/, 2024. Medium articles, Accessed: 2025-04-28.

[3] Carlos Esteves. On the ideal number of groups for isometric group convolutions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 87–14796, 2022.

[4] GeeksforGeeks. Backpropagation in neural network. https://www.geeksforgeeks.org/backpropagation-in-neural-network/, 2024. Accessed: 2025-04-28.

[5] J. Kaplan and Others. Exploring the compute efficiency frontier. *Journal of Machine Learning*, 32(4):123–135, 2020.

[6] Yann LeCun, Bernhard Boser, John S Denker, Don Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.

[7] Matplotlib Development Team. Matplotlib: Visualization with python. https://matplotlib.org/, 2024. Accessed: 2025-04-28.

[8] nipunmanral. Mnist handwritten digit recognition with mlp. https://github.com/Nipunmanral/MNIST-MLP, 2021. Accessed: 2025-04-28.

[9] Papers with Code. Image classification on mnist. https://paperswithcode.com/sota/image-classification-on-mnist, 2024. Accessed: 2025-04-28.

[10] Python Software Foundation. pickle — python object serialization. https://docs.python.org/3/library/pickle.html, 2024. Accessed: 2025-04-28.

[11] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[12] Grant Sanderson. But what is a neural network? — deep learning, chapter 1. https://www.youtube.com/watch?v=aircAruvnKk, 2017. YouTube, Accessed: 2025-04-28.

[13] Sijan67. Handwritten digit recognition using mlp. https://github.com/Sijan67/Handwritten-Digit-Recognition-using-MLP, 2020. Accessed: 2025-04-28.

[14] SoCalc. Mnist mlp example. https://github.com/SoCalc/MNIST-MLP, 2021. Accessed: 2025-04-28.

[15] SoCalc. Mnist mlp extended results. https://github.com/SoCalc/MNIST-MLP-Extended, 2022. Accessed: 2025-04-28.

[16] Jan Erik Solem. *Programming Computer Vision with Python: Tools and algorithms for analyzing images*. O'Reilly Media, 2012.

[17] TensorFlow Developers. Tensorflow: An end-to-end open source machine learning platform. https://www.tensorflow.org/, 2024. Accessed: 2025-04-28.

[18] W3Schools. Numpy tutorial. https://www.w3schools.com/python/numpy_intro.asp, 2024. Accessed: 2025-04-28.

[19] Samson Zhang. Building a neural network from scratch — python code + math explained. https://www.youtube.com/watch?v=8mz5sE3kZJ0, 2020. YouTube, Accessed: 2025-04-28.

Appendix: Source Code

Below is the source code used in this investigation.

Main.py

```
import numpy as np
class MLP:
    def __init__(self, layers, activation, clipValue = 500, weightInit = None, seed=None,lossFunc = 'mean',a=0) -> None:

        """
        layers - List of layers. First layer must match the number of inputs and last layer must match the number of outputs
        activation - is the activation function for the neural network
        clipValue - certain values above clipValue are removed to prevent overflow errors
        weightInit - is the optimizer for weight initialization
        seed - uses a certain random seed to initialise weights and biases
        lossFunc - what loss function will the neural network be using to evaluate itself
        """

        np.random.seed(seed)
        self.numLayers = len(layers)
        self.layerSizes = layers
        self.weights = []
        self.biases = []
        self.activationFunction = activation.lower()
        self.lossFunc = lossFunc.lower()
        self.clipValue = clipValue
        self.weightInit = weightInit.lower()
        self.CPUUsage = 0
        self.a = a

        #init network hyperparameters

        for layer in range(self.numLayers - 1):
            if self.weightInit == "xavier":
                weight = np.random.randn(layers[layer], layers[layer + 1]) / np.sqrt(layers[layer])
            elif self.weightInit == "he":
                weight = np.random.randn(layers[layer], layers[layer + 1]) * np.sqrt(2 / layers[layer])
            else:
                weight = np.random.randn(layers[layer], layers[layer + 1])

            self.weights.append(weight)
            self.biases.append(np.zeros((1, layers[layer + 1])))

        #init random weights and biases using corresponding optimizers

    def __sigmoid(self, x) -> None:

        #sigmoid activation function
        x = np.clip(x, -self.clipValue, self.clipValue)
        return 1 / (1 + np.exp(-x))


    def __sigmoidDerivative(self, x) -> None:

        #sigmoid activation function derivative
        return x * (1 - x)

    def __relu(self, x) -> None:

        #relu activation function
        x = np.clip(x, -self.clipValue, self.clipValue)
        return np.maximum(0, x)

    def __reluDerivative(self, x) -> None:

```

```
        #relu activation function derivative
        return np.where(x > 0, 1, 0)

    def __leakyRelu(self, x):

        #leaky relu activation function
        x = np.clip(x, -self.clipValue, self.clipValue)
        return np.where(x > 0, x, self.a * x)

    def __leakyReluDerivative(self, x, a=0.005):

        #leaky relu activation derivative
        return np.where(x > 0, 1, a)

    def __softmax(self, x):

        x = np.clip(x, -self.clipValue, self.clipValue)  #Clip to prevent overflow errors
        expon = np.exp(x)
        return expon / np.sum(expon, axis=1, keepdims=True)  #Normalize

    def __activation(self, x):

        #function that compiles all activation functions into one function
        #I added this because it made the logic clearer and made it very easy to add new
            experimental activation functions
        if self.activationFunction == 'sigmoid':
            return self.__sigmoid(x)
        elif self.activationFunction == 'relu':
            return self.__relu(x)
        elif self.activationFunction == 'leakyrelu':
            return self.__leakyRelu(x)

    def __activationDerivative(self, x):

        #function that compiles all activation function derivatives into one function
        #I added this because it made the logic clearer and made it very easy to add new
            experimental activation functions
        if self.activationFunction == 'sigmoid':
            return self.__sigmoidDerivative(x)
        elif self.activationFunction == 'relu':
            return self.__reluDerivative(x)
        elif self.activationFunction == 'leakyrelu':
            return self.__leakyReluDerivative(x)

    def __lossDerivative(self, predictions, labels):

        #derivative of loss functions
        if self.lossFunc == 'mean':
            return self.__meanLossDerivative(predictions, labels)
        elif self.lossFunc == 'cross':
            return self.__crossEntropyLossDerivative(predictions, labels)

    def __meanLossDerivative(self, predictions, labels):
        return 2 * (predictions - labels)

    def __crossEntropyLossDerivative(self, predictions, labels):
        return predictions - labels


    def __forward(self, incomingConnections, FLOPPER=False):
        """
        forward function for neural network
        incomingConnections - the input values from the data (in this case pixel values from
            images)
        FLOPPER - is a variable to toggle CPU usage monitoring
        """
```

```
124
125         self.activations = [incomingConnections]
126         for layer in range(self.numLayers - 2):
127             #for each layer calculate dot product from last item stored in self.activations and
                store it in self.activations
128             dotProduct = np.dot(self.activations[-1], self.weights[layer]) + self.biases[layer]
129             activatedDotProduct = self.__activation(dotProduct)
130             self.activations.append(activatedDotProduct)
131 
132             if FLOPPER: #monitor CPU usage
133                 self.countFlops()
134 
135         outputLayer = np.dot(self.activations[-1], self.weights[-1]) + self.biases[-1] #Calculate
              the final output.
136         answer = self.__softmax(outputLayer) #Normalises output layer
137         self.activations.append(answer) #Stores answer in self.activations
138 
139         if FLOPPER: #monitor CPU usage
140             self.countFlops()
141 
142         return answer
143 
144     def __countFlops(self):
145         #this estimate is not at all accurate however it is directly proportional with the real
                FLOP values
146         height, width = self.activations[-2].shape  #finds dimensions of activations
147         randomThing = self.weights[-1].shape[1]     #finds shape of weights
148         flops = height * width * randomThing        #multiplies all dimensions together
149         self.CPUUsage += flops                      #adds calculated product to cumumilative
                variable
150 
151     def __crossEntropyLoss(self, predictions, labels):
152         #Cross enthropy loss function
153         loss = -np.sum(labels * np.log(predictions)) / labels.shape[0]
154         return loss
155 
156     def __meanLoss(self,predictions,labels):
157         #mean loss function
158         loss = np.mean((predictions - labels) ** 2)
159         return loss
160 
161     def __loss(self,predictions,labels):
162         #function containing all loss methods so that I can add new ones easily and so that the
                program structure is clearer
163         if self.lossFunc == 'mean':
164             return self.__meanLoss(predictions,labels)
165         elif self.lossFunc == 'cross':
166             return self.__crossEntropyLoss(predictions,labels)
167 
168 
169     def __backward(self, targets, lr, FLOPPER):
170 
171         """
172         targets - takes a One-hot-encoded list as a target for what the model output should be
173         lr - the learning rate koeficcient
174         FLOPPER - is a variable to toggle CPU usage monitoring
175         """
176 
177         outputLayerError = self.lossDerivative(self.activations[-1], targets) #gets derivative of
              final output in regards to the neural network output
178 
179         error0 = [outputLayerError]
180         for index in range(self.numLayers - 2, 0, -1): # for each layer starting from the end
181             error = np.dot(error0[0], self.weights[index].T) * self.activationDerivative(self.
                    activations[index])
182             error0.insert(0, error)
```

```
183
184             if FLOPPER: # calculates CPU usage
185                 self.countFlops()
186
187         for index in range(self.numLayers - 1): #for each layer excluding the last layer
188             weightGradient = np.dot(self.activations[index].T, error0[index]) #calculates the
                    gradients of the weights by using dot product of error0 and the relevant layer
189             biasGradient = np.sum(error0[index], axis=0, keepdims=True) # calculates the
                    gradients of the biases
190
191             self.weights[index] -= lr * weightGradient # adjusts weights
192             self.biases[index] -= lr * biasGradient # adjusts biases
193
194             if FLOPPER: # calculates CPU usage
195                self.countFlops()
196
197
198
199     def train(self, inputs, targets, numEpochs, lr, batchSize, testInputs = None, testLabels =
            None,
200               decayRate=1, avgLossToggle=False, recordUsage=False, epochLosses=False,stopper =
                    False):
201
202         """
203         inputs - list of lists. Each sublist is a flattened vector image.
204         targets - list of lists. Each sublist is the one-hot-encoded answer.
205         numEpochs - the number of epochs
206         lr - the learning rate constant
207         batchSize - the number of images used per step during batch gradient descent. Keep higher
                for quicker convergence
208         testInputs and testLabels - similar to inputs and targets but should be from the testing
                dataset. Used for research.
209         decayRate - the rate at which learning rate decays. Allows higher initial lr values
                without instability. A good range of values is 1>lr>0.75 depending on training dataset
                size.
210         avgLossToggle - returns the average loss every batch. Used for research
211         recordUsage - do not toggle on - super specific and used for research
212         epochLosses - do not toggle on - super specific and used for research
213         stopper - a toggle to premature stopping when there is a need to optimize training time
                at the cost of accuracy. Succeptable to ending prematurely when encountering
                instability (loss randomly spiking). Lower lr is recommended.
214         """
215
216
217         indexes = inputs.shape[0]
218         lossList = []
219         usageSuperlist = []
220         epochLossesList = []
221
222         print(f"beginning training with  {numEpochs} epochs and layer sizes {self.layerSizes}")
223
224         for epoch in range(numEpochs):
225             lossEpochList = [] # shuffles training images
226             shuffledIndexes = np.random.permutation(indexes)
227             Inputs = inputs[shuffledIndexes]
228             Labels = targets[shuffledIndexes]
229
230             if epochLosses: # option to record losses for each epoch
231                 testPredictions = self.__forward(testInputs)
232                 testLoss = self.__loss(testPredictions, testLabels)
233                 epochLossesList.append(testLoss)
234                 print(testLoss)
235
236             for batch in range(0, indexes, batchSize): # sorts out batch system
237                 batchInputs = Inputs[batch: (batch + batchSize)]
238                 batchLabels = Labels[batch: (batch + batchSize)]
```

```
                predictions = self.__forward(batchInputs, recordUsage) # sets up backpropagation
                loss = self.__loss(predictions, batchLabels)
                lossEpochList.append(loss)

                self.__backward(batchLabels, lr, recordUsage)


            lr *= decayRate

            avgLoss = np.mean(lossEpochList) # gets average loss for the epoch


            if recordUsage: # records CPU usage every epoch if boolean variable recordUsage is
                True
                testPredictions = self.__forward(testInputs)
                avgLoss = self.__loss(testPredictions, testLabels)
                usageSuperlist.append([avgLoss, self.CPUUsage])



            lossList.append(avgLoss) # appends average loss to a list

            if stopper: # if average loss stays the same MLP stops training early
                if len(lossList) >= 4:
                    if ((lossList[-2] + lossList[-1]) / 2)  / ((lossList[-3] + lossList[-4]) / 2)
                        == 1:
                        print("Stopped")
                        break

            print(f'Epoch {epoch + 1} / {numEpochs}, Average Loss: {avgLoss:.10f},CPU FLOPs: {
                self.CPUUsage} lr: {lr:.10f}') # variable print out for debugging

        # various things that I might want the MLP to return based on the input variables
        if recordUsage:
            return usageSuperlist

        if epochLosses:
            if avgLossToggle:
                return lossList, epochLossesList
            else:
                return epochLossesList
        else:
            if avgLossToggle:
                return lossList

    def predict(self, inputs, returnProbabilities = False):
        """
        inputs - input one item if data, with the same resolution as the size as the input layer
            size. Must be flattened.

        a number is outputted that relates to the index in the output layer.
        """

        predictions = self.__forward(inputs)

        if returnProbabilities == False:
            return np.argmax(predictions, axis=1)
        else:
            return self.__softmax(predictions) * 100

    def save(self, filename):
        """
        Saves the parameters to a file specified by 'filename'
        """

```

```
        import pickle
        file = open(filename, 'wb')
        pickle.dump({
            'layerSizes': self.layerSizes,
            'weights': self.weights,
            'biases': self.biases,
            'activationFunction': self.activationFunction,
            'clipValue': self.clipValue,
            'weightInit': self.weightInit
        }, file)
        file.close()

    def load(self, filename):
        """
        Loads paramaters from save file specified by 'filename'
        A network must already be initialized to call this.
        """

        import pickle
        file = open(filename,'rb')
        data = pickle.load(file)
        self.layerSizes = data['layerSizes']
        self.numLayers = len(self.layerSizes)
        self.weights = data['weights']
        self.biases = data['biases']
        self.activationFunction = data['activationFunction']
        self.clipValue = data['clipValue']
        self.weightInit = data['weightInit']
        file.close()
```

Appendix: Source Code

```
import numpy as np
from MLP import MLP
from Plotter import plotMovingAverage, plotPolynomialWithTwo
from matplotlib import pyplot as plt

def convertIntoFormat(labels, size): # One hot encode values
    format = np.zeros((len(labels), size)) # Create list of zeros of same length as numbe of
        classes
    for i, label in enumerate(labels):
        format[i][label] = 1    # insert a one where nessesary
    return format


def checkAccuracy(lst1, lst2): # gets the acuracy of whether two items with the same indexes in
    two lists are the same
    result = []
    for a,b in zip(lst1,lst2):
        if a == b:
            result.append(1)
        else:
            result.append(0)
    return np.mean(np.array(result)) # takes mean of list of ones and zeros to get accuracy


def loadFashionMNIST():
    from tensorflow.keras.datasets import fashion_mnist # import dataset
    (trainingImages, trainingLabels), (testingImages, testingLabels) = fashion_mnist.load_data()
        # load dataset into lists

    trainingImages = trainingImages.reshape(-1, 28 * 28) / 255.0
```

```
    testingImages = testingImages.reshape(-1, 28 * 28) / 255.0 # flatten image into list and divide each pixel value by 255 to make them between one and zero
    trainingLabels = convertIntoFormat(trainingLabels, 10)
    testingLabels = convertIntoFormat(testingLabels, 10) # one-hot-encode all labels

    return trainingImages, testingImages, trainingLabels, testingLabels

def loadMNIST():
    from tensorflow.keras.datasets import mnist

    (trainingImages, trainingLabels), (testingImages, testingLabels) = mnist.load_data()

    trainingImages = trainingImages.reshape(-1, 28 * 28) / 255.0
    testingImages = testingImages.reshape(-1, 28 * 28) / 255.0
    trainingLabels = convertIntoFormat(trainingLabels, 10)
    testingLabels = convertIntoFormat(testingLabels, 10)

    return trainingImages, testingImages, trainingLabels, testingLabels


def showImg(flatImage, label):
    image = flatImage.reshape(28, 28)

    plt.figure(figsize=(4, 4))
    plt.imshow(image, cmap='gray')
    plt.axis('off')
    plt.title(f"label: {label}")


    plt.show()

def hyperparameterChanges(variable, start = 0, end = 0, step = 0, start2 = None, end2 = None, step2 = None):
    # EPQ investigation graph plotter
    resultList = []
    if variable == 'lr':
        learningR = start + step
        epochs = step2
        while learningR < end:
            if start2 == 'sigmoid':
                mlp = MLP(LAYERS,'sigmoid',maxAllowableOutput,'xavier')
            elif start2 == 'relu':
                mlp = MLP(LAYERS,'relu',maxAllowableOutput,'he')
            mlp.train(trainingImages,trainingLabels,epochs,learningR,batchSize,testingImages,testingLabels)
            results = mlp.predict(testingImages)
            resultList.append(checkAccuracy(results,answers))
            learningR += step
        print(resultList)
        plotMovingAverage([resultList],1,step,['learning rate vs accuracy'])




    elif variable == 'activation':
        activationFunctions = ['sigmoid','relu','leakyrelu'] # list of activation functions to investigate
        layers = [[784,600,10],[784,64,10],[784,32,32,10]] # list of layers to investigate
        accuracyList = []

        for layer in layers:

            resultList = []
            for function in activationFunctions:
                if function == 'sigmoid':
                    epochs = start
```

```
                    mlp = MLP(layer,function,maxAllowableOutput,'None')
                    loss = mlp.train(trainingImages,trainingLabels,epochs,lr,batchSize,
                        testingImages,testingLabels,avgLossToggle=True)
                    resultList.append(loss) # for every network layout create and train an mlp
                else:
                    epochs = start
                    mlp = MLP(layer,function,maxAllowableOutput,'None')
                    loss = mlp.train(trainingImages,trainingLabels,epochs,lr,batchSize,
                        testingImages,testingLabels,avgLossToggle=True)
                    resultList.append(loss)

                results = mlp.predict(testingImages)
                accuracy = checkAccuracy(results,answers)
                accuracyList.append(accuracy) # generates accuracies for each network
            print(resultList)
            plotMovingAverage(resultList,2,batchSize,activationFunctions) # plots
        print(accuracyList) # prints out accuracies

    elif variable == 'leakycomparison':
        epochs = start2
        while start <= end:
            mlp = MLP(LAYERS,'leakyrelu',maxAllowableOutput,'he',a=start)
            mlp.train(trainingImages,trainingLabels,epochs,lr,batchSize,testingImages,
                testingLabels)

            start += step

            results = mlp.predict(testingImages)
            accuracy = checkAccuracy(results,answers)
            resultList.append(accuracy)
        print(resultList)
        plotMovingAverage([resultList],1,step,['a value versus accuracy'])

    elif variable == 'optimizers':

            if end == 'sigmoid':
                opt = 'xavier'
            elif end == 'relu' or end == 'leakyrelu':
                opt = 'he'

            epochs = start
            mlp = MLP(LAYERS,end,maxAllowableOutput,opt,lossFunc='mean')
            loss = mlp.train(trainingImages,trainingLabels,epochs,lr,batchSize,testingImages,
                testingLabels,avgLossToggle=True)
            resultList.append(loss)
            mlp = MLP(LAYERS,end,maxAllowableOutput,'None',lossFunc='mean')
            loss = mlp.train(trainingImages,trainingLabels,epochs,lr,batchSize,testingImages,
                testingLabels,avgLossToggle=True)
            resultList.append(loss)
            print(resultList)
            plotMovingAverage(resultList,2,1,['xavier optimization function','no optimization
                function'])


    elif variable == 'epochs':
        epochs = end
        mlp = MLP(LAYERS,'relu',maxAllowableOutput,'he',lossFunc='cross')
        losslist,trainingLossList  = mlp.train(trainingImages,trainingLabels,int(epochs),lr,
            batchSize,testingImages,testingLabels,True,0.9,True,False,True)
        plotMovingAverage([losslist,trainingLossList],2,labels=['loss recorded on training
            dataset','loss recorded on testing dataset'])

    elif variable == 'lossFuncs':
        epochs = end
        funcs = ['mean','cross']
        for func in funcs:
```

```
            mlp = MLP(LAYERS,'relu',maxAllowableOutput,'he',lossFunc=func,seed = 100)
            mlp.train(trainingImages,trainingLabels,epochs,lr,batchSize,testingImages,
                testingLabels)

            results = mlp.predict(testingImages)
            accuracy = checkAccuracy(results,answers)
            resultList.append(accuracy)

        print(resultList)

    elif variable == 'EfficiencyFrontier':

        width = start
        epochs = 100
        flopResults = []
        for layerN in range (start2,end2,step2):
            while width < end:
                layers = [784]
                for i in range (layerN):
                    layers.append(width)
                layers.append(10)
                mlp = MLP(layers,'relu',maxAllowableOutput,'he',lossFunc='cross')
                flopList = mlp.train(trainingImages,trainingLabels,epochs,lr,batchSize,
                    testingImages,testingLabels,True,0.95,True,True,False,True)
                print(flopList)
                flopResults.append(flopList)
                width += step
        print(flopResults)
        plotPolynomialWithTwo(flopResults,True)

    elif variable == 'LayerSize':
        epochs = start2
        for layerSize in range (start,end,step):
            LAYERS[1] = layerSize

            mlp = MLP(LAYERS,'relu',maxAllowableOutput,'he',lossFunc='cross',seed=100)
            mlp.train(trainingImages,trainingLabels,epochs,lr,batchSize,testingImages,
                testingLabels,lrDecay=True,decayRate=0.9)
            results = mlp.predict(testingImages)
            accuracy = checkAccuracy(results,answers)
            resultList.append(accuracy)

        plotMovingAverage([resultList],1,multiplier=step,labels=["Layer size vs accuracy given"],
            start=start)

    return None


def loadCIFAR():

    #Change number of input nodes to 3072

    from tensorflow.keras.datasets import cifar10

    (trainingImages, trainingLabels), (testingImages, testingLabels) = cifar10.load_data()

    trainingImages = trainingImages.reshape(-1, 32 * 32 * 3) / 255.0
    testingImages = testingImages.reshape(-1, 32 * 32 * 3) / 255.0
    trainingLabels = convertIntoFormat(trainingLabels, 10)
    testingLabels = convertIntoFormat(testingLabels, 10)

    return trainingImages, testingImages, trainingLabels, testingLabels


def trainBestMNIST():

```

```
    LAYERS = [784,128,10]
    maxAllowableOutput = 9e9
    epochs = 10
    lr = 0.01
    batchSize = 50
    seed = 100

    mlp = MLP(LAYERS,'relu',maxAllowableOutput,'he',seed,'cross')
    mlp.train(trainingImages,trainingLabels,epochs,lr,batchSize,testingImages,testingLabels,
        lrDecay=True,decayRate=0.9)
    return mlp

def trainBestFashionMNIST():
    #Best accuracy I got is 0.8967

    LAYERS = [784,128,10]
    maxAllowableOutput = 500
    epochs = 30
    lr = 0.01
    batchSize = 50
    seed = 104

    mlp = MLP(LAYERS,'relu',maxAllowableOutput,'he',seed,lossFunc='mean')
    mlp.train(trainingImages,trainingLabels,epochs,lr,batchSize,testingImages,testingLabels,True
        ,0.9,False,False,False,True)
    return mlp

def trainBestCIFAR():
    layers = [3072,1024,10]
    maxAllowableOutput = 500
    epochs = 1
    lr = 0.01
    batchSize = 64
    seed = 100

    mlp = MLP(layers,'sigmoid',maxAllowableOutput,'xavier',seed)
    mlp.train(trainingImages,trainingLabels,epochs,lr,batchSize,testingImages,testingLabels,True
        ,0.9,False,False,False)

    return mlp



LAYERS = [784,64,10]
lr = 0.01
batchSize = 50
maxAllowableOutput = 500


trainingImages, testingImages, trainingLabels, testingLabels = loadMNIST()

answers = np.argmax(testingLabels, axis=1)
mlp = trainBestMNIST()
results = mlp.predict(testingImages)
accuracy = checkAccuracy(answers,results)
print(f'Final accuracy from the test dataset: {accuracy:.4f} with {LAYERS} node layout')
```

Plotter.py

```
import numpy as np
import matplotlib.pyplot as plt

def plotMovingAverage(dataList, depth, multiplier=1, labels=None, logScale=False,start=0):
    """Plots moving average of input data
        multiplier defines scale of x axis
```

```
        start paramater defines start of x axis.
    """
    plt.figure(figsize=(10, 6))

    for i, data in enumerate(dataList): # for every list in datalist
        movingAvg = np.convolve(data, np.ones(depth) / depth, mode='valid')
        # calculate moving average using convolve mathematical function and a list with only ones
        plt.plot(np.arange(len(movingAvg)) * multiplier+start, movingAvg, label=labels[i])
        # plots points and applies x axis scaling

    if logScale: # log scale toggle
        plt.yscale('log')
        plt.xscale('log')

    plt.xlabel("layer size")
    plt.ylabel("accuracy")
    plt.grid()
    plt.legend()
    plt.show()


def plotPolynomialWithTwo(data, logScale = False):
    """
    Plots graph with two axis inputs. Takes in list with sublists with two values each, x and y
        respectively.
    logScale changes both y and x axis to logarithmic
    """

    plt.figure(figsize=(10, 6))

    for item in data:

        y, x = zip(*item)
        x = np.array(x)
        y = np.array(y)

        plt.plot(x, y)


    if logScale:
        plt.yscale('log')
        plt.xscale('log')

    plt.xlabel('FLOP/s')
    plt.ylabel('loss')
    plt.title('loss vs compute graph for different neural network')
    plt.grid(True)
    plt.show()
```

LiveDemo.py

```
import pygame
import numpy as np
from MLP import MLP
import matplotlib.pyplot as plt
import math

"""
Multilayer perceptron test with new experimental centering program.
Gives good accuracies with human entered digits, even when they are off centre.
Struggles with scaling digits.
"""

#Some important variables initialized

```

```
pygame.init()
screenWidth = 840
screenHeight = 840
gridSize = 30
rows = 28
cols = 28
displayHeight = 150


screen = pygame.display.set_mode((screenWidth, screenHeight + displayHeight),pygame.RESIZABLE)
pygame.display.set_caption("Live elkwork demo")




grid = np.zeros((rows, cols))


mlp = MLP([28*28, 925, 10], 'relu', 500, 'he', 100, 'cross')
# model used for this demonstration achieves 98.51% accuracy on the MNIST dataset. The loss evaluation function for this was cross entropy.
mlp.load("save.txt")

white = (255,255,255)
black = (0,0,0)


def drawGrid() -> None:
    #initializes the grid

    for row in range(rows):
        for col in range(cols):
            x = col * gridSize + (screenWidth - gridSize * 28) / 2
            y = row * gridSize + displayHeight + (screenHeight - displayHeight - gridSize * 28) / 2
            color = black if grid[row, col] == 1 else white
            pygame.draw.rect(screen, color, pygame.Rect(x, y, gridSize, gridSize))
            pygame.draw.rect(screen, black, pygame.Rect(x, y, gridSize, gridSize), 1)
            pygame.draw.rect(screen,(200,200,200), pygame.Rect(0,0,screenWidth,displayHeight))

def draw(text,font,color,x,y) -> None:

    #Draws text

    img = font.render(text, True, color)
    screen.blit(img,(x,y))

def applyBrush(x, y) -> bool:
    x = round((x - (screenWidth - gridSize * 28) / 2) / 28)
    y = (y - (screenHeight - displayHeight - gridSize * 28) / 2) / 28

    init = False


    #Shades in grid depending where mouse curson is

    if math.floor(x) < int(x) + 0.5:
        x = int(x)
        y = int(y)
        if 0 <= x < rows and 0 <= y < cols:
            if grid[y][x] != 1 or grid[y][x - 1] != 1 or grid[y - 1][x] != 1 or grid[y - 1][x - 1] != 1:
                init = True
            grid[y][x] = 1
            if x > 0:
                grid[y][x - 1] = 1
```

```
78
79              if y > 0:
80                  grid[y - 1][x] = 1
81
82              if x > 0 and y > 0:
83                  grid[y - 1][x - 1] = 1
84      else:
85          x = int(x)
86          y = int(y)
87          if 0 <= x < rows - 1 and 0 <= y < cols - 1:
88              if grid[y][x] != 1 or grid[y][x + 1] != 1 or grid[y + 1][x] != 1 or grid[y + 1][x +
                    1] != 1:
89                  init = True
90              grid[y][x] = 1
91              grid[y][x + 1] = 1
92              grid[y + 1][x] = 1
93              grid[y + 1][x + 1] = 1
94
95
96      return init
97
98  """
99    ___
100  /   /|
101 ---- |
102 |   | |
103 ----/
104
105 """
106
107
108 def showImg(flatImage) -> None:
109     #displays the image through matplotlib. Used for debugging
110
111     image = flatImage.reshape(28, 28)
112     plt.imshow(image, cmap='gray')
113     plt.show()
114
115 def centralise(grid) -> list:
116
117     #centres the image. Helps a lot with the neural network specified because it is not trained
            on off-centre images
118
119     rows = np.any(grid, axis=1)
120     cols = np.any(grid, axis=0)
121
122     if not rows.any() or not cols.any():
123         return grid
124
125     yMin, yMax = np.where(rows)[0][[0, -1]]
126     xMin, xMax = np.where(cols)[0][[0, -1]]
127
128     cropped = grid[yMin:yMax+1, xMin:xMax+1]
129
130     yOffset = (28 - cropped.shape[0]) // 2
131     xOffset = (28 - cropped.shape[1]) // 2
132
133     centeredGrid = np.zeros((28, 28))
134     centeredGrid[yOffset:yOffset+cropped.shape[0], xOffset:xOffset+cropped.shape[1]] = cropped
135
136     return centeredGrid
137
138
139 def displayProbabilities(prediction) -> None:
140     # A function that displays everything through pygame GUI
141
```

```
    if len(prediction) > 0:

        font = pygame.font.SysFont("Arial", int(30 * screenHeight/750))
        smallfont = pygame.font.SysFont("Arial", int(30 * screenHeight/2000))


        spacing = screenWidth / 11

        for number, probability in enumerate(prediction[0]):
            draw(str(number),font,"black",spacing * (number + 1) + ((screenWidth / 20) / 4) -
                10,10)
                #pygame.draw.circle(screen, 'red', (spacing * (number + 1),70), probability)

            if number != np.argmax(prediction[0]):
                pygame.draw.rect(screen,'red',pygame.Rect((spacing * (number + 1)) - 20,
                    screenHeight * 6/100, screenWidth / 20, probability * 2 * screenHeight / 840))
            else:
                pygame.draw.rect(screen,'green',pygame.Rect((spacing * (number + 1)) - 20,
                    screenHeight * 6/100, screenWidth / 20, probability * 2 * screenHeight / 840))

            draw(f"{round(probability,2)}%",smallfont,"black",spacing * (number + 1) + (
                screenWidth / 40) - 7 * len(f"{round(probability,2)}%"), screenHeight * 6.1/100)

        draw(f"Final prediction of the number drawn: {np.argmax(prediction)}", font, 'black',
            screenWidth/2 - 250, screenHeight / 8.4)

def displayPlaceholder() -> None:
    # The function responsible for the screen that shows up when nothing is drawn
    font = pygame.font.SysFont("Arial", int(30 * screenHeight/750))

    draw("Please draw a digit on the canvas below", font, 'black', screenWidth/2 - 260,
        screenHeight / 15)

drawing = False
updated = False
clock = pygame.time.Clock()
init = False
empty = True
prediction = []
ended = False

while True:
    screen.fill(white)
    drawGrid()
    if drawing:
        empty = False

    if empty:
        displayPlaceholder()
    else:
        if init:
            flattenedImage = centralise(grid).flatten()
            prediction = mlp.predict(flattenedImage,True)
        displayProbabilities(prediction)

    init = False

    screenWidth,screenHeight = pygame.display.get_window_size()
    displayHeight = int(150/840 * screenHeight)
    gridSize = min([(screenHeight - displayHeight) // 28, screenWidth // 28])


    #Event handler that handles key presses. The x key closes the program
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            quit()
```

```
202
203         if event.type == pygame.MOUSEBUTTONDOWN:
204             if event.button == 1:
205                 drawing = True
206                 if ended == True:
207                     grid = np.zeros((rows, cols))
208                     empty = True
209
210         if event.type == pygame.MOUSEBUTTONUP:
211             if event.button == 1:
212                 drawing = False
213                 ended = True
214
215         if event.type == pygame.MOUSEMOTION:
216             if drawing:
217                 mouseX, mouseY = event.pos
218                 init = applyBrush(mouseX, mouseY - displayHeight)
219
220         if event.type == pygame.KEYDOWN:
221             if event.key == pygame.K_x:
222                 quit()
223
224
225
226     pygame.display.update()
227     clock.tick(200)
```