Investigating neural scaling laws in a multilayer perceptron

Extended project qualification

Candidate name: Leonid Elkin

Candidate number:

Centre name: Tonbridge School

Centre number:

Date: 24/2/2025

# Abstract

For my Extended project qualification, I have decided to explore *neural scaling laws* in multilayer perceptrons (*MLPs*) by examining the impact of model size, *learning rate*, and number of *epochs* on network performance. The *MLP* was built from scratch using NumPy, and various configurations were tested by adjusting the number of layers, *neurons*, *learning rate*, and training *epochs*. Visualizations were generated to track performance changes across different scaling parameters using the matplotlib library. Results showed that performance improved with larger models, but the gains diminished after a certain point. Additionally, while more *epochs* improved training accuracy, it increased *overfitting* in larger models, and decreased performance. Increasing *the learning rate* significantly increased *convergence*, however, caused *training instability* if over-tuned. The optimal *learning rate* *hyperparamete*r was found to be different for different model sizes. As a demonstration of the quality of my research, I used my findings to create the best possible model for the *MNIST dataset* and compared it to others, and it made 9.4% fewer errors relative to the next best one.

[Abstract 2](#_Toc191277643)

[Introduction 5](#_Toc191277644)

[Neural networks 5](#_Toc191277645)

[Neural scaling laws 5](#_Toc191277646)

[Objective 5](#_Toc191277647)

[Literature Review 6](#_Toc191277648)

[Methodology 6](#_Toc191277649)

[Safety 6](#_Toc191277650)

[Theory 6](#_Toc191277651)

[Introduction 6](#_Toc191277652)

[Hidden layer 6](#_Toc191277653)

[Output layer 7](#_Toc191277654)

[Backpropagation 8](#_Toc191277655)

[Coding 8](#_Toc191277656)

[MLP class 8](#_Toc191277657)

[Constructor 9](#_Toc191277658)

[Loss method implementation 10](#_Toc191277659)

[Activation function implementation 11](#_Toc191277660)

[Forward pass 13](#_Toc191277661)

[Backpropagation algorithm implementation 14](#_Toc191277662)

[Training method 14](#_Toc191277663)

[Miscellaneous methods 16](#_Toc191277664)

[Research files 16](#_Toc191277665)

[Plotting 17](#_Toc191277666)

[Main file 18](#_Toc191277667)

[Running the MLP 19](#_Toc191277668)

[HyperparameterChanges function 20](#_Toc191277669)

[Investigation 20](#_Toc191277670)

[Finding ideal functions for 28x28 image recognition 20](#_Toc191277671)

[Activation functions 20](#_Toc191277672)

[Optimal learning rate 24](#_Toc191277673)

[Learning rate decay 28](#_Toc191277674)

[Effect on epoch number on accuracy 28](#_Toc191277675)

[Epoch number and overfitting 28](#_Toc191277676)

[Loss functions 29](#_Toc191277677)

[Number of neurons per layer 31](#_Toc191277678)

[Compute efficiency frontier 35](#_Toc191277679)

[Findings summary 36](#_Toc191277680)

[Glossary 37](#_Toc191277681)

# Introduction

## Neural networks

Neural networks are the fundamental component of machine learning, a field that is becoming increasingly relevant in our lives today. Like humans, they are extremely adaptable and can perform a variety of tasks from drawing linear regression lines to image *classification*. They achieve their results by passing an input vector of data through an unimaginably complex mathematical function to output a relevant set of probabilities or parameters. The fundamental neural network architecture is the multilayer perceptron (*MLP*), three or more layers of *neurons* each with its *bias* value where each *neuron* in the previous *layer* is connected with all *neurons* in the next with each connection having a *weight* value. It uses the *backpropagation algorithm* to *train*. I will be investigating the effect on the performance of the neural network if the *hyperparameters* for the *backpropagation algorithm* are changed to give a solution to the challenge of balancing model size, *learning rate*, *epoch* number, and the computational resources required.

## Neural scaling laws

Neural scaling laws are theoretical insights into how model performance changes concerning model size (number of *parameters*), data size, and computational resources. These laws can be used to predict model performance with different configurations. One of the most well-known works on neural scaling laws is the study by Kaplan et al. (2020), called ‘Scaling Laws for Neural Language Models.’ This research shows that model performance follows predictable power-law relationships concerning model size (number of parameters), dataset size, and compute resources. This study found that increasing these parameters resulted in a systematic decrease in loss, however with diminishing returns in the case of large language models (LLMs)

## Objective

For my investigation, I created a multilayer perceptron (*MLP*) using the math library NumPy. I chose NumPy because it greatly optimizes mathematical operations, especially operations between large vectors. I have chosen to perform my research on image *classification*, and I have been primarily using the *MNIST dataset*, although as a proof of concept, I have implemented more complex ones like *FashionMNIST* and *CIFAR10*. These datasets are very popular and are supported by the library tensorFlow, meaning that they can be imported directly into the program with only one line. Graphing was done using the plotting library matplotlib. I have used this perticular plotting library before in my other projects, which was very helpful when I was coding the network. My objective is to use these graphs and predict the relationship of these *parameters* with performance. Performance will be measured with two values, *loss* and *accuracy* with *loss* being the result of my *evaluation function* and *accuracy* being the result of correct answers on the dataset as a percentage. As a testament to the accuracy of my research, my final aim is to create an ideal model for the MNIST dataset. Similarly in the real world, a good understanding of neural scaling laws will allow us to make efficient and powerful neural networks with reduced computational cost. Throughout my research I have encountered brilliant pieces of work, which were unfortunately hindered by lack of proper fine-tuning of *hyperparameters*. For example, sijan67’s study featured a final MLP accuracy of 97.79%, with ten neuron layers of 128 neurons each. As seen in the later parts of the study, such a configuration is very inefficient as it causes the *dying ReLU problem.*

# Literature Review

Initially used a variaty of resources to learn about the topic. As an introduction to the field, I found videos by 3Blue1Brown extremely helpful on youtube.

But what is a neural network? | Deep learning chapter 1 – An introduction to neural network architecture. Explains the concepts of neurons, weights, biases and convergence very well.

Gradient descent, how neural networks learn | DL2 – An explanation of gradient descent and the backpropagation algorithm. It also mentions optimisation techneques such as using batch gradient descent instead of stochastic gradient descent.

Backpropagation, step-by-step | DL3 & Backpropagation calculus | DL4 – The mathematecal explanation of backpropagation. This greatly helped me start piecing together the code implementation, however, it was not enough. I could easily implement the forward pass, but not the backwards pass. For large coding projects, I found that it was extremely helpful to plan everything out first and know how each aspect of the project would work, instead of hoping you would underestand it along the way.

An extremely helpful article by medium.com called “how-to-implement-backpropagation-with-numpy by Andres Berejnoi”. It showed in detail how backpropagation should look, and I was confident that I could begin coding.

After completing the coding project, I needed to compare my product with other people’s. On the popular website github, I found some MLP’s other users have created. All of them are similar to mine in the sense that they are created from scratch, only using low level libraries or no libraries at all.

The user nipunmanral on GitHub has achieved an accuracy of 98.26% with a single 512 hidden layer perceptron.

The use of sijan67 on GitHub achieved an accuracy of 97.79% using an alternative MLP implementation.

SoCalc’s MLP achieved a 98.33% accuracy on the MNIST dataset with an MLP.

Another one of SoCalc’s reports stated that an accuracy of 98.37% accuracy was achieved with an MLP.

Hence my target is to surpass the 98.37% accuracy boundary. From a brief review of the code, I have realized that all of these projects suffer from poor hyperparameter tuning. Sijan67 had an extremely eccentric configuration with ten hidden layers. This reenforces the importance of the objective of this research, as there is clearly a lot of wasted potential with how people choose to scale.

# Methodology

## Safety

Although there is no life-threatening danger of programming, excessive screen time, especially in the dark, may cause problems with eyesight. Furthermore, incorrect posture like slumping on your chair causes long-term back pain and other health issues. I made sure to limit my time behind thse computer and only worked during the day in a well-lit room. This organization also made it possible to avoid staying up late working and ensured that I got enough sleep.

## Theory

### Introduction

The multilayer perceptron is the fundamental neural network model. It is a *supervised*, *feed-forward network*. In the case of *image classification*, it feeds forward the pixel values of the image through multiple mathematical functions to output a vector of probabilities for each result. It consists of layers of *neurons* where each layer is fully interconnected with the next with connections that have a variable *weight*. There are three types of *neuron layers*: the input layer, the hidden layer and the output layer. The input layer has the same number of *neurons* as the number of pixels of the image if it is greyscale, and thrice the number if it is full colour. Each neuron of the input layer is assigned the value of its respective pixel value. Noticeably, the image is flattened into a 1D matrix, meaning spacial relation is lost.

### Hidden layer

The hidden layer/s transforms the information from the input layer to the output layers. Similarly, each layer is a collection of *neurons*. In the hidden layers, each *neuron* has a bias value (b) – an abstract constant that is added to each output of the neuron. Each incoming and outgoing connection of neurons has an abstract weight value (W) which influences how important certain connections are. Furthermore, for each neuron, an *activation function* is used to prevent linearity f(x). Mathematically speaking this can be represented as where x in the input vector from the previous layer. The three *activation functions* I will use for this investigation are the *sigmoid* function: , the rectified linear unit (*ReLU*): and *leaky ReLU*, for which a small constant alpha is used when .

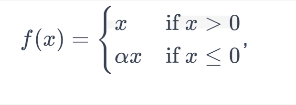


Figure 1: ReLU activation function

Taken from https://www.bragitoff.com/2021/12/efficient-implementation-of-relu-function-and-its-derivative-gradient-in-python/

### Output layer

The output layer is a vector of probabilities that is generated at the end of the forward pass. It is fully connected to the last hidden layer. Depending on the activation function choice, the range of *logit* values may vary, with it being for the *sigmoid activation* function and for *ReLU*. To convert the table of *logits* into a table of probabilities the *SoftMax function* is used. It divides the exponential of each *logit* by the sum of the *logits* to produce a list of probabilities adding to one.

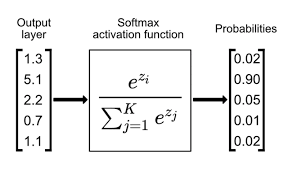


Figure 2: SoftMax function

Taken from https://www.singlestore.com/blog/a-guide-to-softmax-activation-function/

The formula for the SoftMax activation function is where zi is the i'th element of the list and is the sum of the *logits*. It is the only activation function that does not need a derivative.

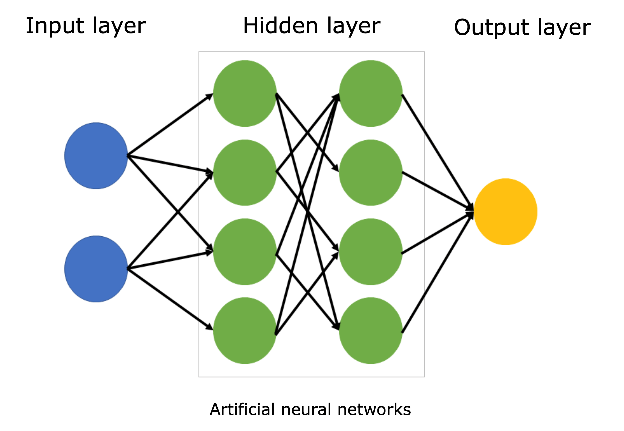


Figure 3: Structure of the multilayer perceptron

<https://www.sciencelearn.org.nz/images/5156-neural-network-diagram>

The forward pass for a multilayer perceptron is essentially a collection of vector operations. The formula for the forward pass of an MLP with one hidden layer is . In this case, y is the neural network’s prediction of what the input is.

### Backpropagation

Weights and biases act as the memory of the MLP. In the case of image recognition fine tuning these parameters would allow the mathematical function above to recognise patterns. The algorithm to tune these parameters is backpropagation. It consists of three key stages, the forward pass, backward pass, and gradient descent.

The forward pass uses a training example to get outputs from the network. The success of the network is then evaluated using the loss function. There are many loss functions but for this investigation, I will be using two: mean squared error (MSE) and cross-entropy loss. MSE is the most basic loss function, and is designed for use in graph regression, while cross-entropy loss is supposed to be specialised to classification tasks such as the one in this investigation. This is because cross-entropy loss penalises wrong decisions much more than MSE. The backward pass includes calculating the derivative of every parameter concerning y. The explanation of how it does so is included during the coding process. The gradient calculations of the backward pass aren't independent, meaning that just minimizing the derivative of each parameter does not work. Working out the global minimum of the function using algebra also doesn't work simply because the number of parameters is in the hundreds of thousands even for the simplest datasets.

 The solution to this is gradient descent. It is the heuristic method of finding the local minimum by incrementing down the gradient of the function. With each weight and bias, each weight gets nudged towards its minimal point by subtracting their respective gradients times by a constant from their original value. That constant is called the learning rate. To increase compute efficiency a technique called batch gradient descent is used. Instead of calculating the gradient after each training example it compiles the average gradient across multiple and uses that for gradient descent. During training after the entire dataset is used up and the model isn't trained yet, it can be reused multiple times. The number of times the same dataset is reused is called the epoch number. After the network is fully trained, which is shown by low-loss function values, the parameters can be saved, and the network can accurately predict the meaning of data it has never seen before.

## Coding

## MLP class

As per my preplanned architecture, I started by making the MLP class. I used the python programming language due to its relative simplicity and the mathematics library NumPy for more efficient vector operations. My *IDE* of choice was VSCode.

## Constructor



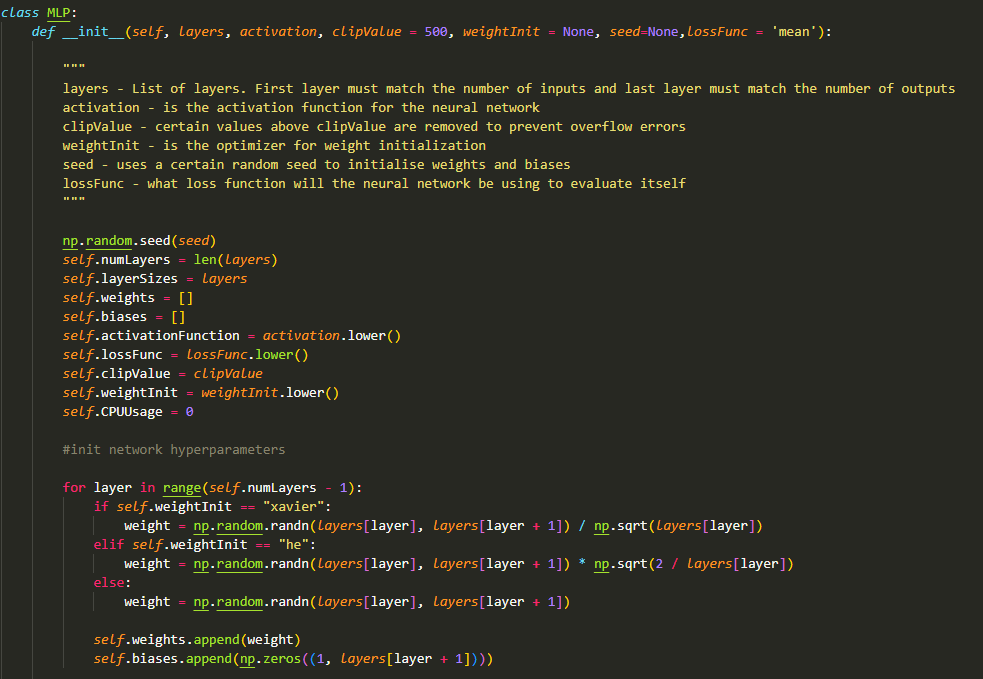
Figure 4: Initial rough layout of the MLP class

Figure 5: The finished constructor method for the MLP class

In my completed constructor method, I take in optional variables to control *what loss functions* and *weight initialization techniques* to use. The clipValue variable is used with np.clip() to avoid overflow errors when using the *sigmoid activation function*.

The *Xavie*r and *He* *weight initialization methods* are used to increase performance. They still utilize the np.random() function however alter the output. *Xavier initialization* is used with the *sigmoid activation function* to keep the variance of the weights in each layer the same hence preventing *exploding and vanishing gradients*. The *weights* of each layer are initialized randomly however they are scaled by where is the number of neurons in the previous layer.



Figure 6: My implementation for the Xavier initialization techneque.

The *He initialization function* is specifically designed for the *ReLU* *activation functio*n. It consists of randomly initializing weights and multiplying them by 2/ where is the number of nodes in the previous layer.



Figure 7: My implementation of the He initialization techneque

If no optimization function is selected in the constructer, the biases are initialized randomly. Theoreticlly this can cause *vanishing and exploding gradients.*



Figure 8: The code for random initialization of biases

### Loss method implementation

The *loss* method contained references to the two other *loss* methods. It calls each one depending on the variable self.lossFunc that was defined in the constructor method.

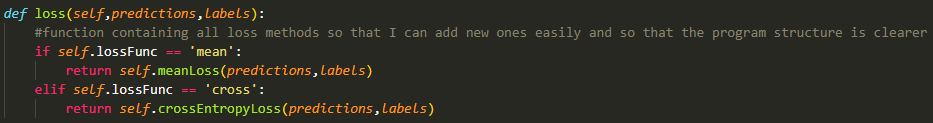


Figure 9: The main loss method that can reach out to other loss methods

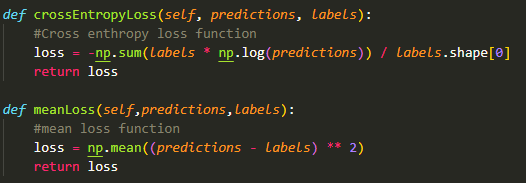


Figure 10: Code for mean square loss and cross-entropy loss

As previously stated, the two *loss functions* used are *mean* and *cross-entropy*. The format of these methods is such that it returns a singular *loss* value used for evaluation. The formula for meanLoss (*Mean squared error loss*) is where N is the batch size, is the expected answer, and is the answer given by the network. It is a straightforward metric of inaccuracy and is useful for identifying models with substantial prediction mistakes.

*Cross Entropy loss* (Categorical Cross-Entropy) is another loss function with the formula where C is the number of classes of the output layer. The result has to be negated because each value of is less than one.

The derivatives of the loss functions that are used in backpropagation are implemented in a similar format.

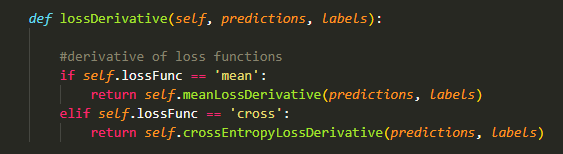


Figure 11: Main derivative method that can reach out to other derivative methods

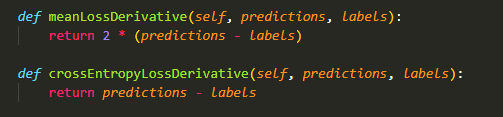


Figure 12: Derivative methods for mean square loss and cross-entropy loss

The derivative for mean squared error is simply and the derivative for cross entropy is

### Activation function implementation

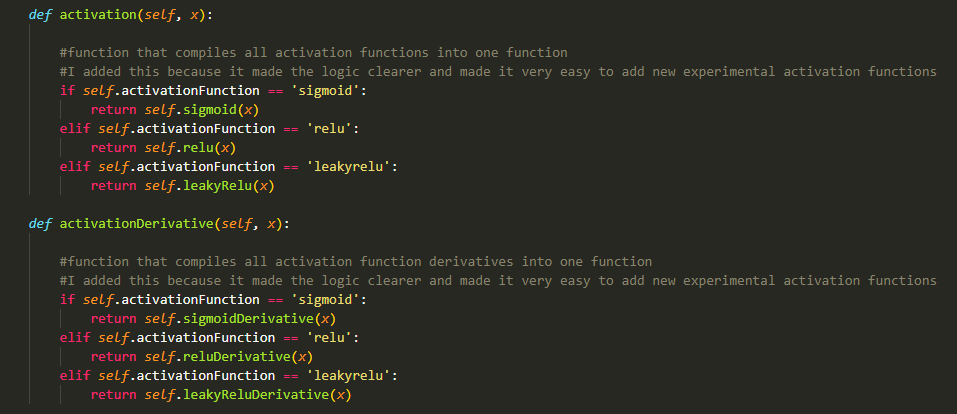


Figure 13: Two methods. One for calling the sigmoid, ReLU, and leaky-ReLU activation functions that are located in separate methods, and the other for calling their derivatives.

Similarly to the *loss functions*, I used a single method to control multiple *activation functions* in a way dependent on the state of the self.activationFunction variable. These are the *activation functions* for:

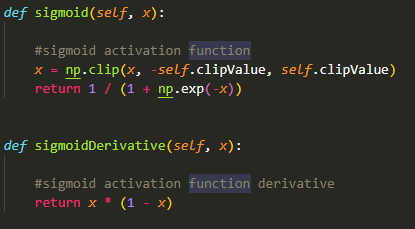


Figure 14: The sigmoid activation function and its derivative

For *sigmoid activation* and its derivative ,

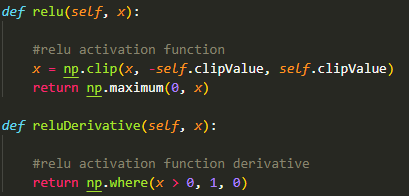


Figure 15: The ReLU activation function and its derivative

For *ReLU activation* and its derivative

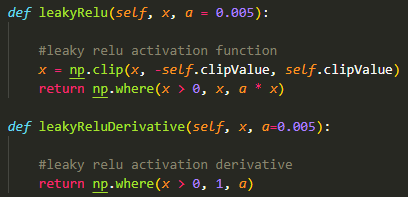


Figure 16: The leaky-ReLU activation function and its derivative

And for *leaky ReLU* and its derivative

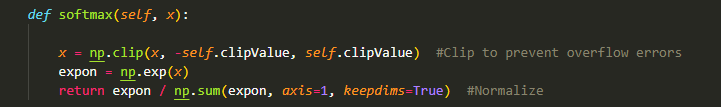


Figure 17: The SoftMax function

This is the *SoftMax function* used for the investigation. I added clipping to prevent overflow errors. The clip boundary may be set during the MLP class initialization.

### Forward pass

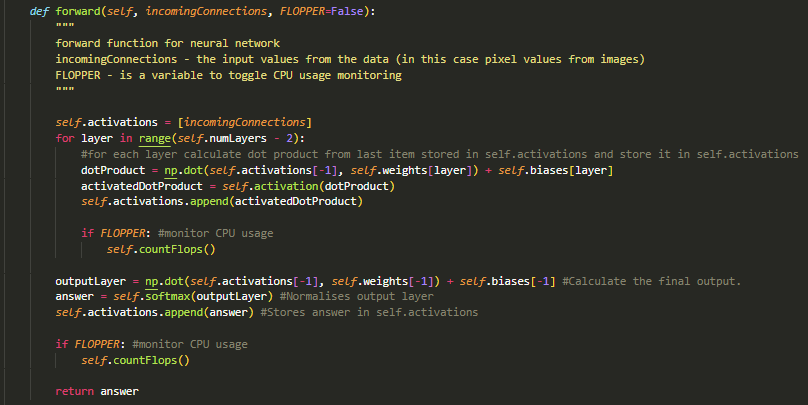


Figure 18: The forward pass of the neural network

This is the *forward pass* of my network. Instead of using recursion which would have been more algorithmically complex I decided to use for loops. The FLOPPER variable toggles CPU usage measurement.

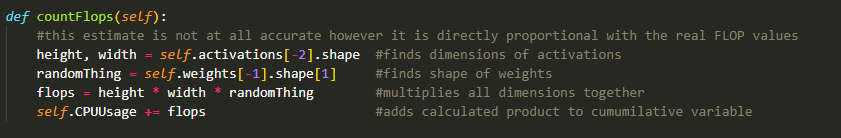


Figure 19: The FLOP estimation program

It finds it by multiplying every dimension together and adds it to a cumulative variable self.CPUUsage. This is by not accurate as it assumes every parameter would take one floating point operation however the answer is proportional to the actual FLOP value.

### Backpropagation algorithm implementation

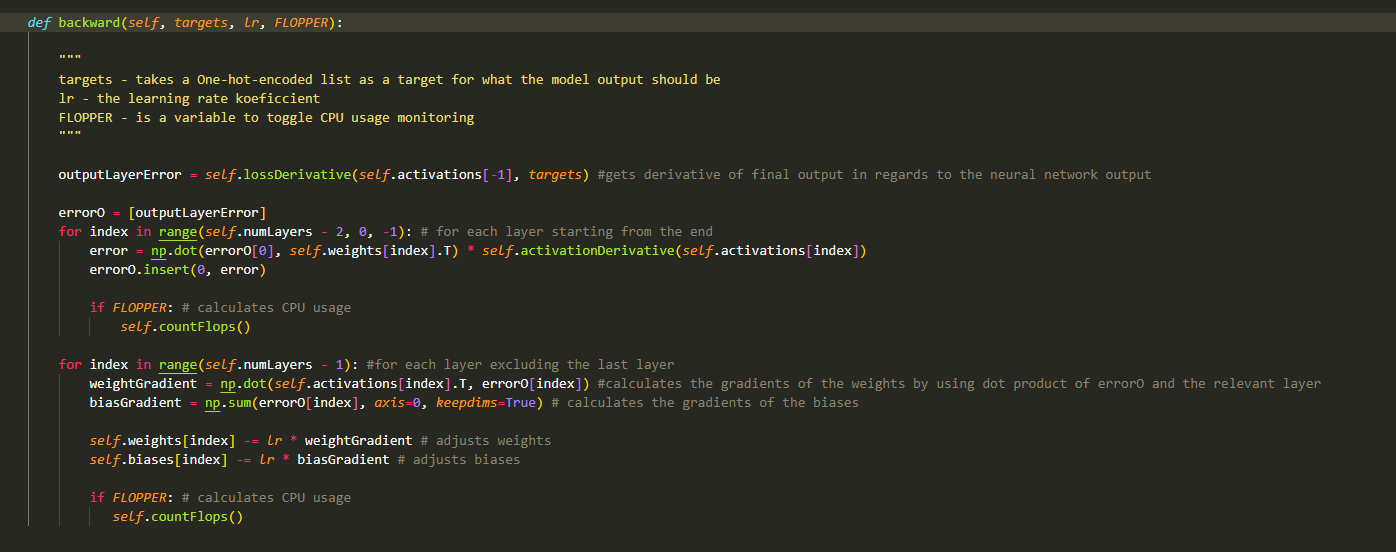


Figure 20: The backpropagation algorithm used to train the neural network

This is the *backward pass* of the MLP class. It takes in expected values for the data on which the *forward pass* was performed on. It uses the same self.activations list as the *forward pass*. The targets are assumed to be *one hot encoded*. The togglable FLOPPER variable measures CPU usage.

### Training method

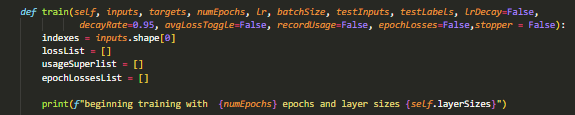


Figure 21: The parameters passed into the training method and the data structures being created in preparation for training

My training function has 13 parameters excluding self. Five of them are Boolean toggles and you can run the function with only seven filled in.



Figure 22: The training procedure of the MLP

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. It also 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 loss decreases too slowly.

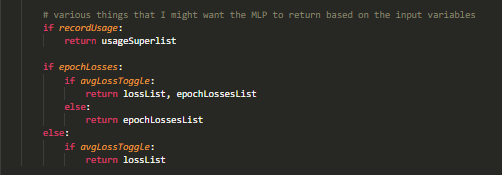


Figure 23: Outputs of the training procedure depending in inputted paramaters

### Miscellaneous methods

At the end, there is an option to output certain data that was collected. LossList returns loss for every epoch whereas epochLossesList returns loss for testing examples that the system hasn't seen before. Two different loss lists are used later to measure overfitting.



Figure 24: Method used to predict the classification of a list of data items

The prediction method runs the *forward pass* and extracts the index of the largest probability.



Figure 25: The saving and loading methods

I have decided to implement a save/load feature for the neural network, which will allow parameters to be portable between machines. For It I used the simple library pickle which allowed me to store each parameter easily. Pickle allows quick saving into a text file without getting bogged down in formatting, hence was chosen. This is important because I would like to save time by not retraining the model every time its run. Also in case I would want to export my model for some reason.

# Research files

### Plotting

For plotting I chose to use matplotlib as it is by far the most popular option, and I had prior experience in using it. I made two plotting functions. One for plotting a moving average and the other for plotting with two-axis input. I used both to represent data in my investigation.

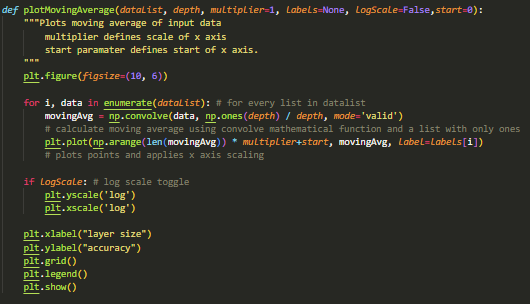


Figure 26: Function to plot graph with a moving average

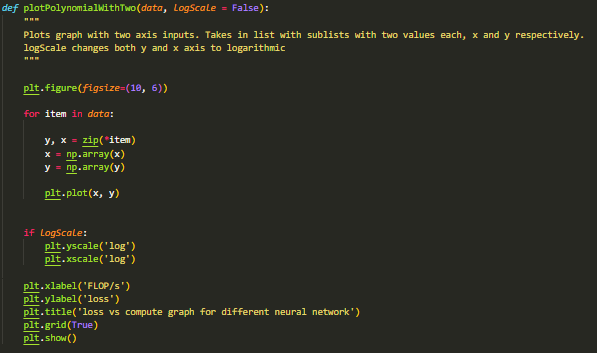


Figure 27: Function to create a graph using two lists which represent x and y coordinates

### Main file



Figure 28: Libraries needed for code to run

These 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 every dataset. They are all very similar with the exception of the first two lines, so as an example, this is the function to load the *MNIST dataset*:

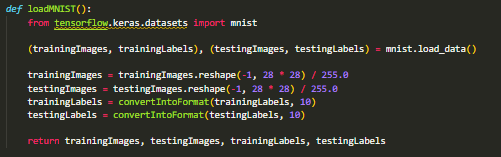


Figure 29: The loading function for the MNIST dataset

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.



Figure 30: A function to display the images from the dataset

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



Figure 31: Code required to show figure 32

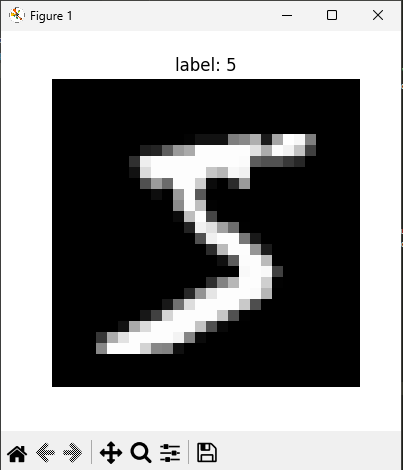


Figure 33: An image from the MNIST dataset

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

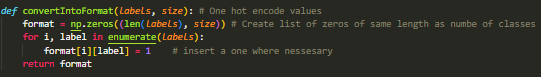


Figure 34: The function used to one-hot-encode numbers

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.

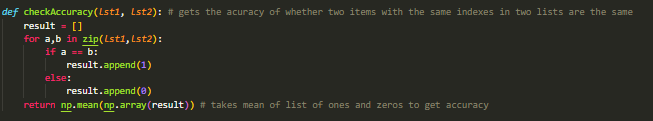


Figure 35: The function used to check accuracy of the answers given by the neural network.

### Running the MLP

This is a basic program to run the MLP.

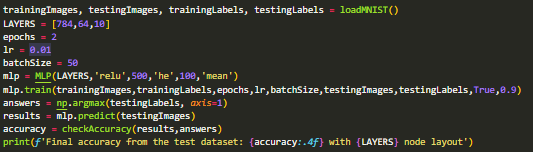


Figure 36: Code to run a simple MLP

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.

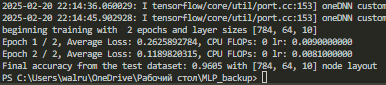


Figure 37: Console log during running figure 36

### HyperparameterChanges function

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



Figure 38: The hyperparamaterChanges function and the parameters being passed into it

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

Investigation

# Finding ideal functions for 28x28 image recognition

For the entire experiment, I will be using the *batch size* of 50 to speed up convergence. Batches are needed to improve convergence speed. Instead of adjusting gradients for every image, the algorithm adjusts them based on the average of the gradients. Clip values where to avoid oveflow and underflow errors. For example a x value of -1100 would produce an estimated gradient of , which exceeds the floating point limit of python, and would most certainly would cause an error when encountered. Even if this extra failsafe wasn’t implemented, overflows would most likely not occur if weights and biases are initialized using He and Xavier initialization, because those functions aim to prevent the exploding and vanishing gradient problem.

### Activation functions

The three *activation functions* I had at my disposal were *sigmoid*, *ReLU*, and *leaky ReLU*. To find out which ones were most suitable for this particular task I decided to inspect the accuracy of each one for different network sizes. I have chosen to use a network with 64 *neurons* in one layer to see *a loss* on a small network and a network with 600 *neurons* to see *a loss* on a large network. Both *Sigmoid* and *ReLU* have their problems that can arise. For example, the *vanishing gradients problem*, where large or small values cause an underflow error in the gradients of the *sigmoid function*. The effect of this is slower learning and in some cases no learning at all. The problem with *ReLU* is fittingly called the ‘*dying ReLU problem*’. Because the gradient at any point of the function below zero is also equal to zero, information is lost, and the network is inherently biased towards positive *weights*. The *leaky ReLU* avoids this issue by having a unique y value for every x value. Consecutive network layers help exacerbate these issues because incoming *connections* are more likely to be large or small. Therefore, a network with 64 neurons split across two layers will be used.

I will be using the following code to conduct this experiment. Aside from plotting an epoch-to-loss graph I also printed out the final accuracies of each network which are judged by the entire dataset. This is because, at 20 epochs, the neural network starts *overfitting*, meaning the *loss value* starts giving a false positive to how well the network performs. The *loss* graph is still important to see what function converges faster and is generally faster to compute. This piece of code took my laptop about five minutes to run. These results used ten *epochs* with a learning rate of 0.01 to prevent instability during training. The *alpha value* for *leaky ReLU* was set at 0.01.

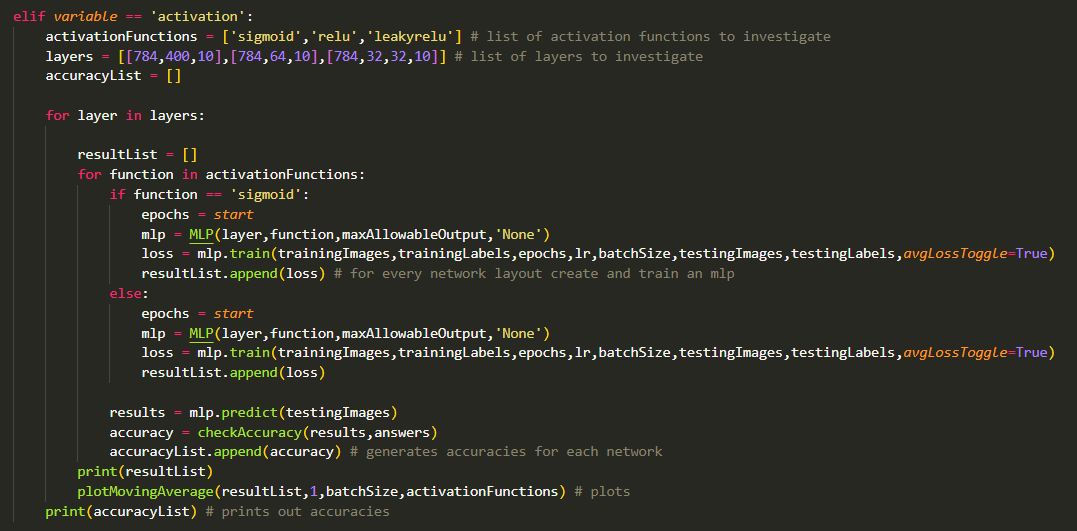


Figure 39: A part of the hyperparamaterChanges function that tests the loss of different activation functions over the amount of data used for training

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



Figure 40: The code snippet ran to generate figure 41, figure 43 and figure 45

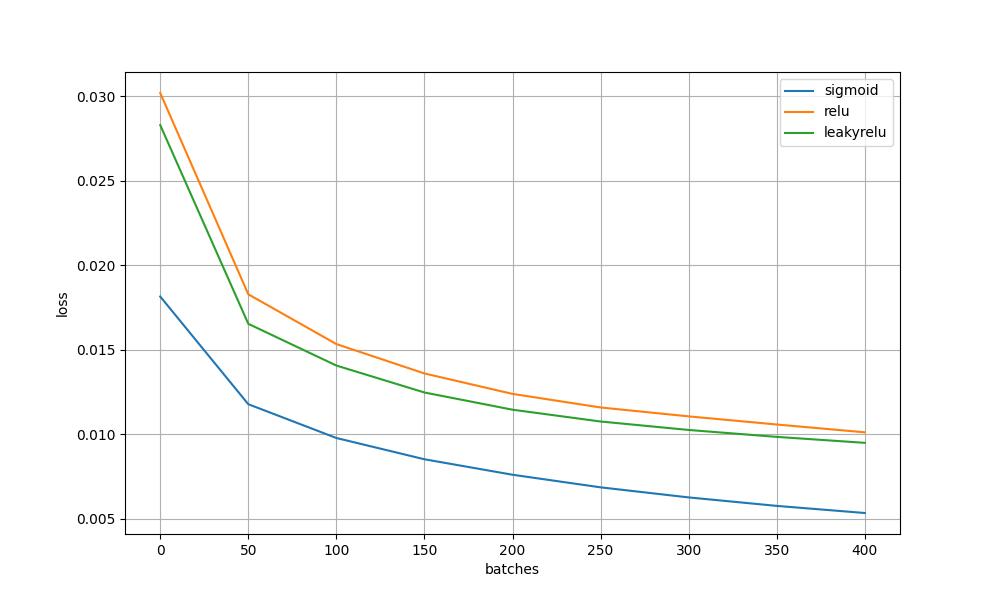


Figure 41: A graph showing the loss of neural networks with one hidden layer of size 64 using different activation functions measured against the amount of data passed into them during training.

As you can see *sigmoid* *converges* much faster than the *ReLU* and achieves an almost zero *loss*. This is clearly because of *overfitting* which incentivised me to test each network on the testing dataset. The final accuracy list is as follows:



Figure 42: Final network accuracies for the networks in figure 41 printed out

As shown here *sigmoid* still outperforms *ReLU* and leaky *ReLU*. In turn, *leaky ReLU* performs better than normal *ReLU*. The conclusion made from this is that a small one-layered *MLP*’s ideal *activation function* is the *sigmoid*.

For the 32x2 neural network, it is clear from the *loss* graph that *ReLU* faced significant stability issues due to the *dying ReLU problem*. Although *leaky ReLU* performed much better *sigmoid* outdid both as shown in this graph and accuracy table:

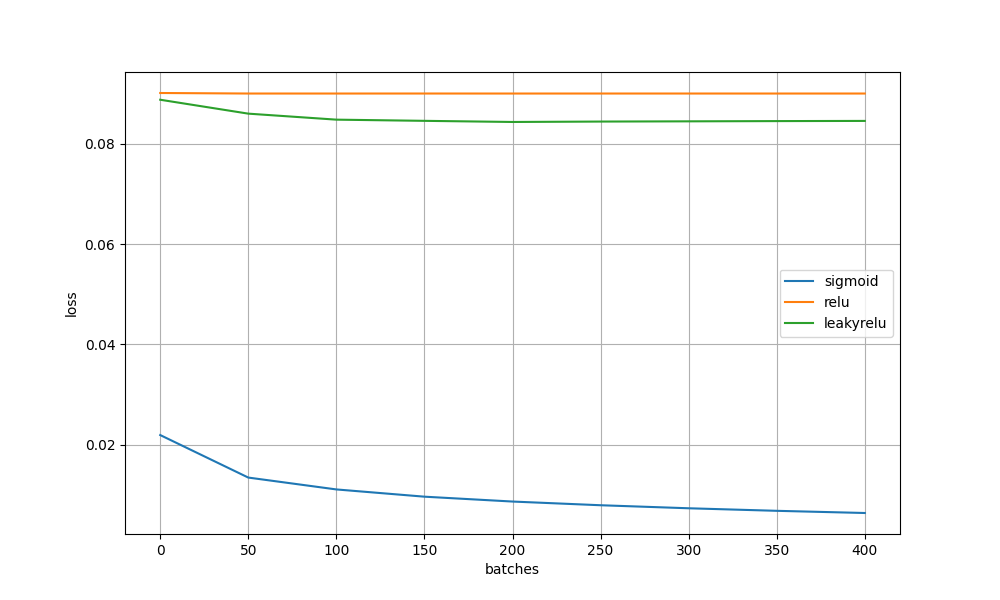


Figure 43: A graph showing the loss of neural networks with two hidden layers each of size 32 using different activation functions measured against the amount of data passed into them during training.



Figure 44: The final network accuracies for the neural networks in figure 43

The accuracies of the different networks clearly show that the ReLU activation struggles with multiple layers, with normal *ReLU* not learning at all and *leaky ReLU* giving an accuracy rating of 14.4% compared to the *sigmoid’s* 94.8%. This shows that the *sigmoid activation function* is again better than the *ReLU* in multi-layered networks.

The experiment with one layer with 600 *neurons* showed similar results to the one with 64. The *sigmoid function* seems to outperform the *ReLU*.

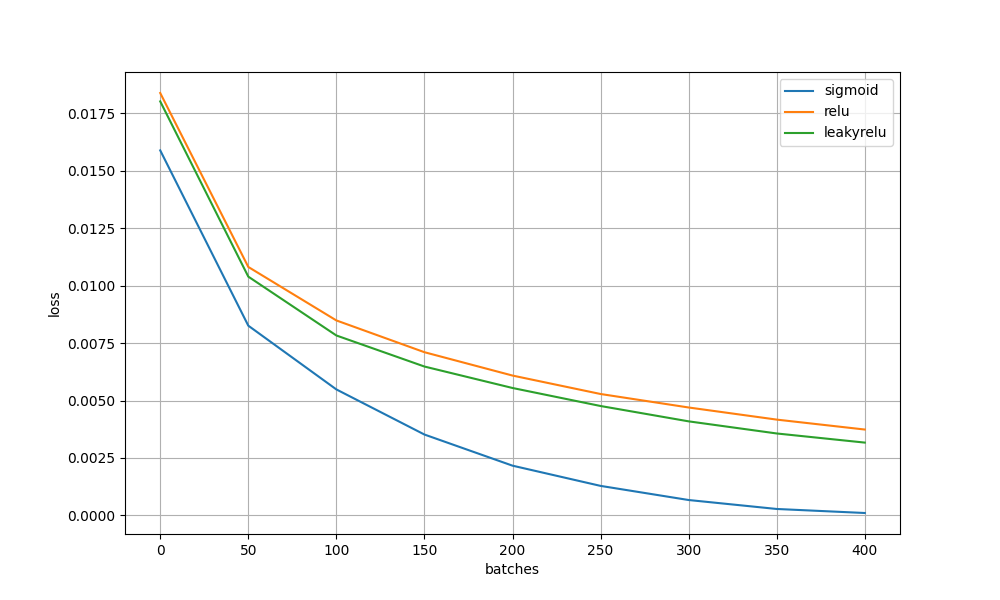


Figure 45: A graph showing the loss of neural networks with one hidden layer of size 400 using different activation functions measured against the amount of data passed into them during training.



Figure 46: The final network accuracies for the neural network shown in figure 45

The accuracies on the testing dataset showed a completely different picture. *ReLU* seemed to have better accuracy than *sigmoid*, which became worse since the 64-neuron test. I attribute this to the faster *convergence* of the *sigmoid function*. Because more *epochs* were used on this network than on the others the model overfitted and gave worse accuracies. The normal *ReLU* function even outperformed *the leaky ReLU*. In conclusion, *ReLU* is better than *sigmoid* for large, one-layered datasets.

The *ReLU* + large one-layered *perceptron* gave the highest accuracies. So, I will be using that architecture to try and achieve the highest accuracy possible.

### Optimal learning rate

The learning rate is an important factor in this investigation. In theory to high of a *learning rate* will cause instability and inaccuracy in *convergence* whereas a low learning rate will cause convergence to be to slow. The goal is hence to find the optimal *learning rate* to balance these factors out. I was using the code below to generate all graphs. For this experiment one layer with 64 neurons was used alongside a batchsize of 50 and a learning rate of 0.01.

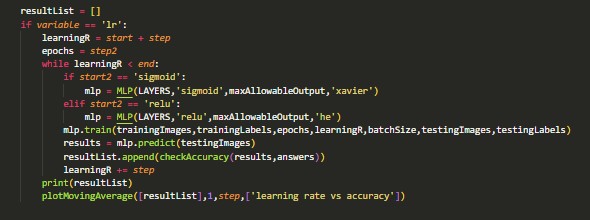


Figure 47: A segment of the hyperparameterChanges function that measures the accuracy of a neural network as the learning rate is changed

The first graph I made was with the *sigmoid function*. I used the following code.



Figure 48: The code snipped used to generate figure 49

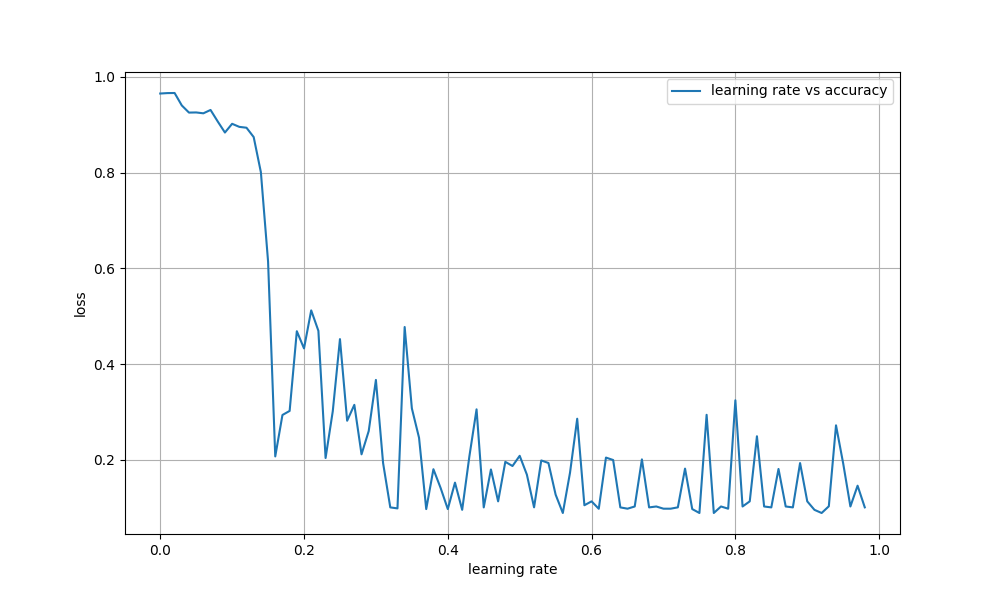


Figure 49: The graph showing the effect of changing the learning rate on the stability and accuracy of the neural network

On this graph, you can see a clear boundary between *neural network stability* and *instability*. It occurs just after 0.01. The curious thing about this graph is the continuous decrease in *learning rate* after a point extremely early on. I generated a higher-resolution graph to help me understand this.



Figure 50: The code snippet used to generate figure 51

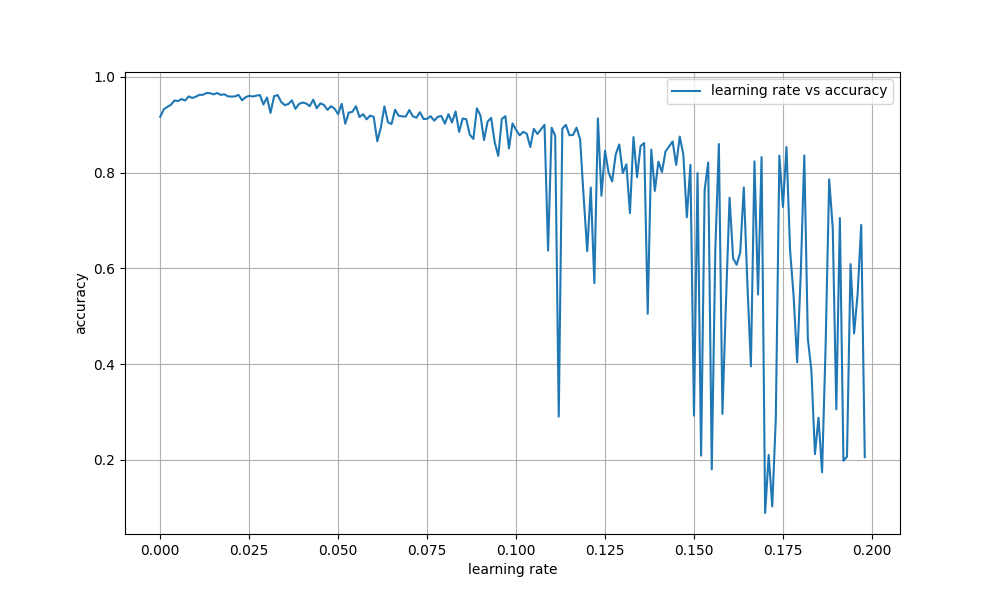


Figure 51: A higher resolution and more precise graph that shows the effect of changing the learning on the stability and accuracy of the neural network

A higher-detail graph reinforces the idea that severe *instability* starts after *the learning rate* surpasses 0.1. It is also shown that at first accuracy increases with *the learning rate*, however after a point around 0.015, it starts dropping. This is explained by *convergence* being achieved at 0.015 and any increases in *learning rate* past that cause overadjustments during *gradient descent* which prevents the network from properly settling into a *local minimum*.

The *ReLU activation function* is proven to be more sensitive to *a higher learning rate* than the *sigmoid* as shown by this graph. Only one point was in the stable region.



Figure 52: The code snippet used to generate figure 53

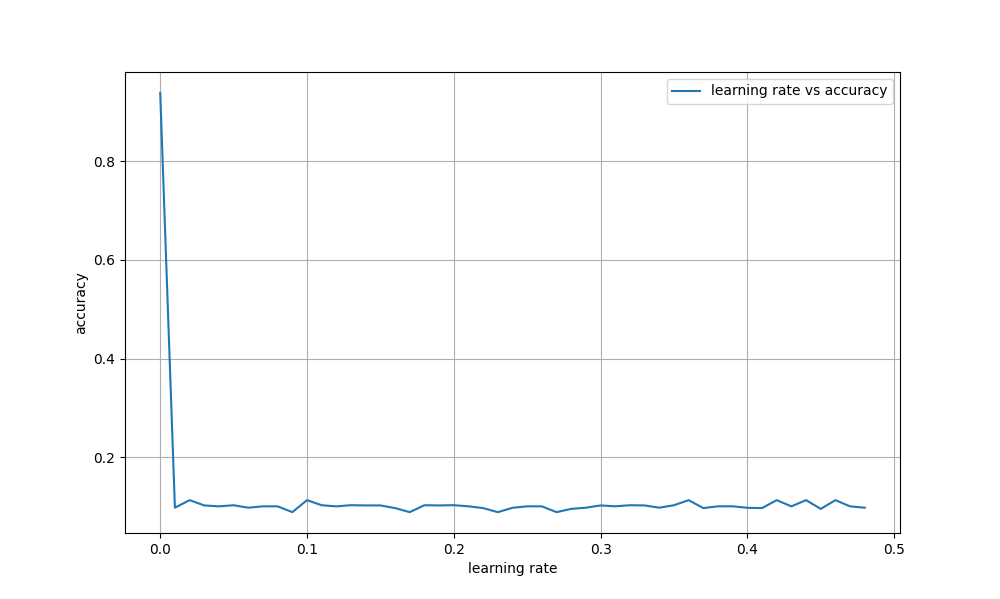


Figure 54: The graph showing the effect of changing the learning rate on a neural network using the ReLU activation function

A higher resolution graph was produced. Each point is only 0.0001 learning rate units apart.



Figure 55: The code snippet used to generate figure 56

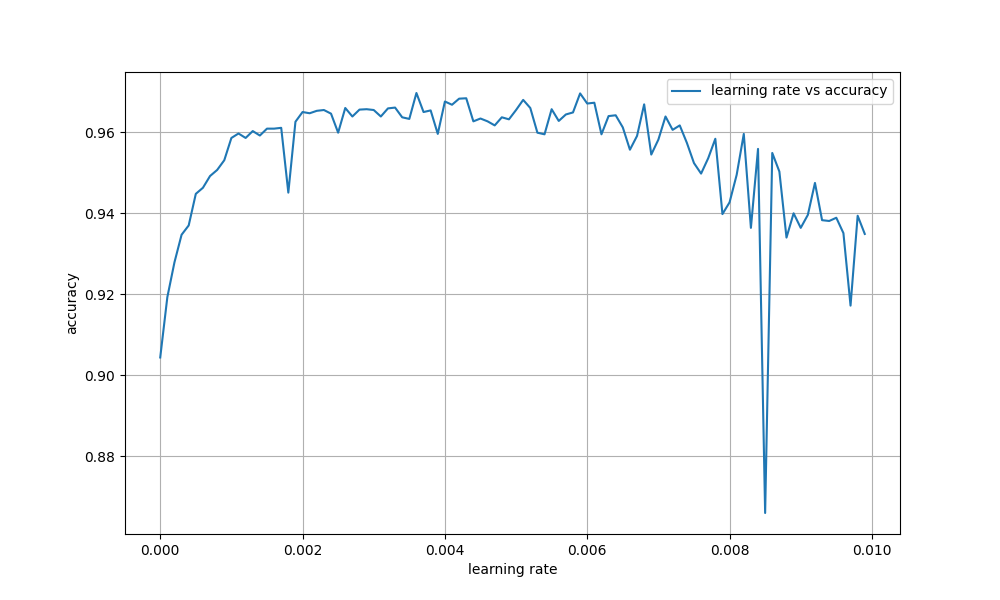


Figure 56: A more precise graph showing the effect different learning rates have on the stability and accuracy of a neural network using the ReLU activation function

The same trend occurs here where after *convergence* is achieved accuracy is decreased proportionally to the *learning rate*. There is one randomly *unstable* point, however this can be ruled out as a rare anomaly.

In summary, even though the *ideal learning rate* value changes with how many epochs are used, the rough range for it is for *sigmoid* and for *ReLU*.

#### Learning rate decay

The issue with high *learning rates* is that *convergence* happens too quickly and hence ‘misses’ the *local minimum*. Ideally, the rate of *convergence* should be quick at the start and decrease as the network nears the minimum. For that *learning rate decay* was used from this point onwards. My implementation of *learning rate decay* decreased the *learning rate* by 10% every epoch. This allowed me to put in *higher learning rates* without causing instability.



Figure 57: The code snippet responsible for decreasing the learning rate every epoch

The code responsible for learning rate decay where decayRate is a parameter to toggle it.

### Effect on epoch number on accuracy

#### 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 because it is capable of learning more complex patterns and hence requires more learning to *converge*. In this experiment, I will be using the previously explained structure of a large single layered *perceptron*. Even though the optimal *epoch* number will be different from others, the relationship between epochs and accuracy will be universal.

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

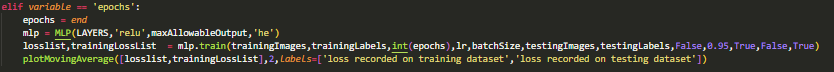


Figure 58: The extract from the hyperparameterChanges function that investigates the effect of epochs on the testing and training loss

It collects the loss of the network with the testing and then the training dataset, which allows me to inspect and investigate *overfitting*.



Figure 59: The code snippet responsible for plotting figure 60

This is the code ran. I have chosen to use 100 epochs because it will cause the network to overfit a lot.

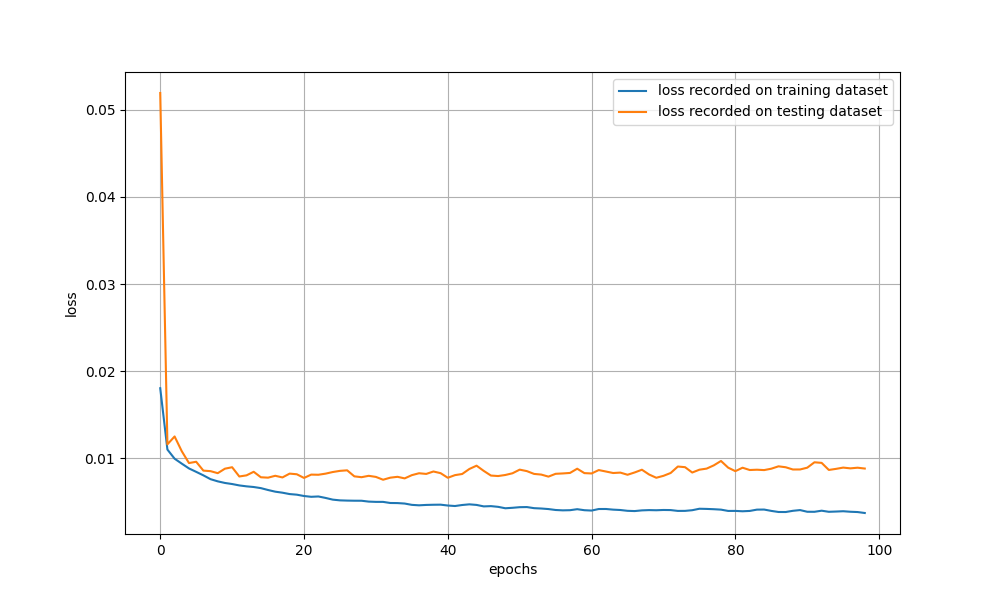


Figure 60: A graph showing the testing and training accuracies of the same neural network over many epochs

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.

### 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.

The code used for this investigation is as follows:

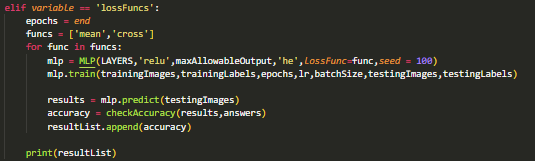


Figure 61: An extract from the hyperparameterChanges function that compares the accuracies of identical neural networks using different loss functions



Figure 62: The code snippet used to generate figure 62



Figure 63: The accuracies generated by figure 61 after figure 62 was run

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



Figure 64: The code snippet used to generate figure 65



Figure 65: The accuracies generated by figure 61 after figure 64 was run

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



Figure 66: The code snippet used to generate figure 67



Figure 67: The accuracies generated by figure 61 when figure 66 was run

At 2 epochs MSE is 3% worse than *cross-entropy* with it having a 92.83% accuracy and the later 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.

### 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.

A computer screen with colorful text

AI-generated content may be incorrect.

Figure 68: An extract from the hyperparameterChanges function that investigates how the size of a hidden layer affects the accuracy of the neural network

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



Figure 69: This is the code snippet used to generate figure 70 using figure 68

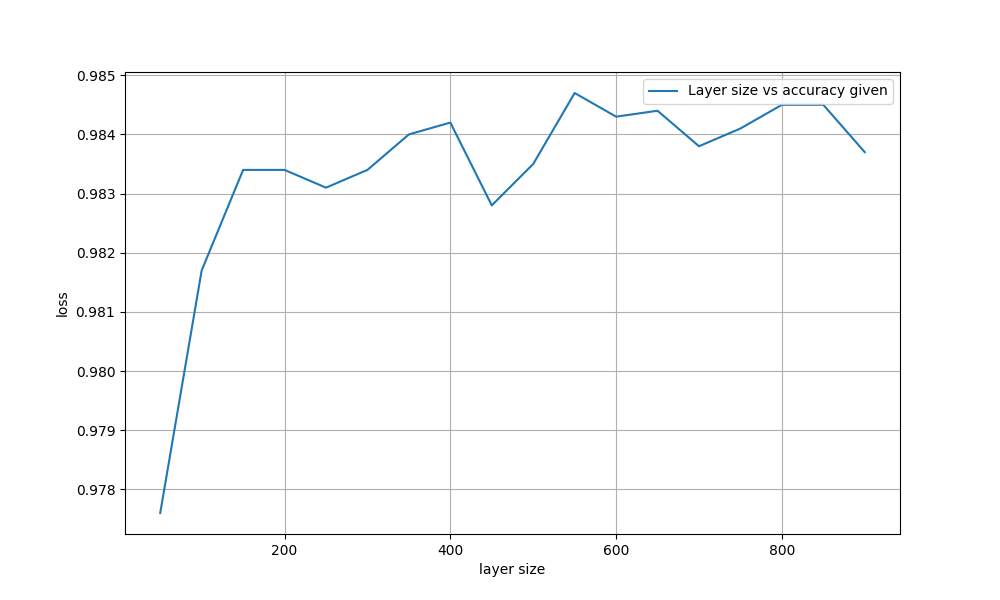


Figure 70: The graph showing the effect of changing the size of a layer on the accuracy of the neural network

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 when 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 71: This code snippet was used to generate figure 72 using figure 68

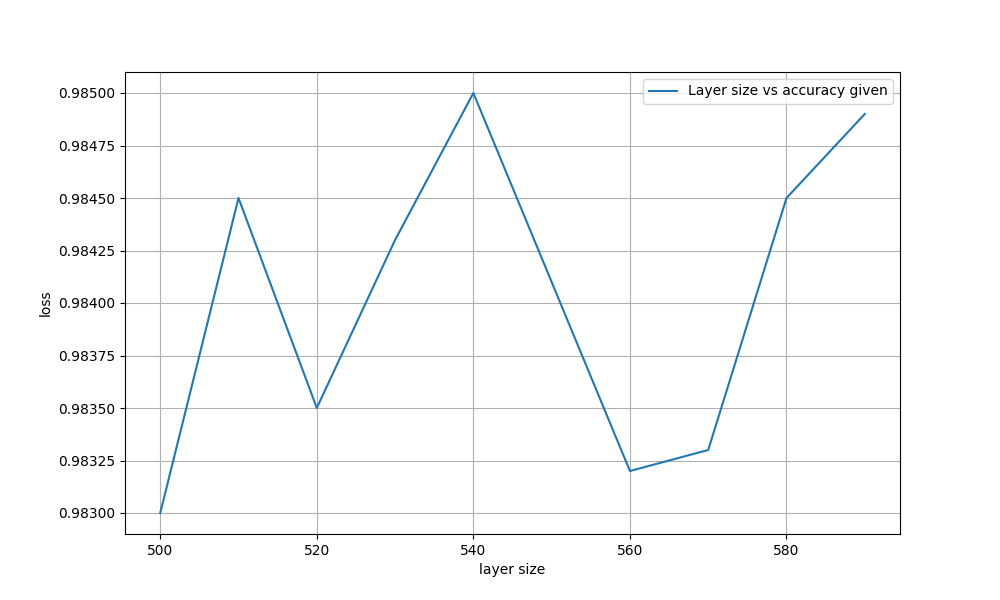


Figure 72: A more precise graph showing the effect increasing the layer size has on accuracy

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.



Figure 73: The console message stating accuracy

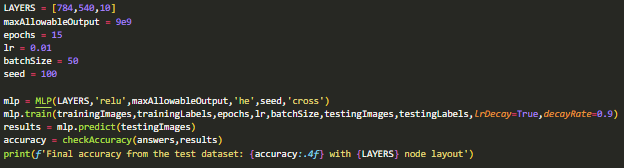


Figure 74: Code that was used to generate figure 73

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.



Figure 75: Code used together with figure 68 to generate figure 75

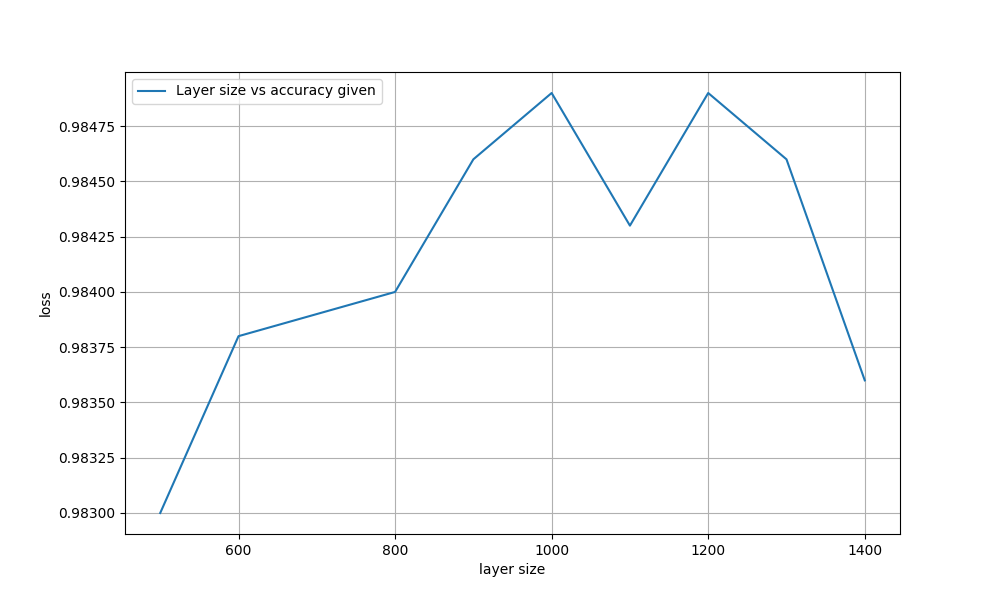


Figure 76: A graph showing the relationship between the layer size and the accuracy, showing two large spikes.

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.



Figure 77: This code was ran together with figure 68 to generate figure 78

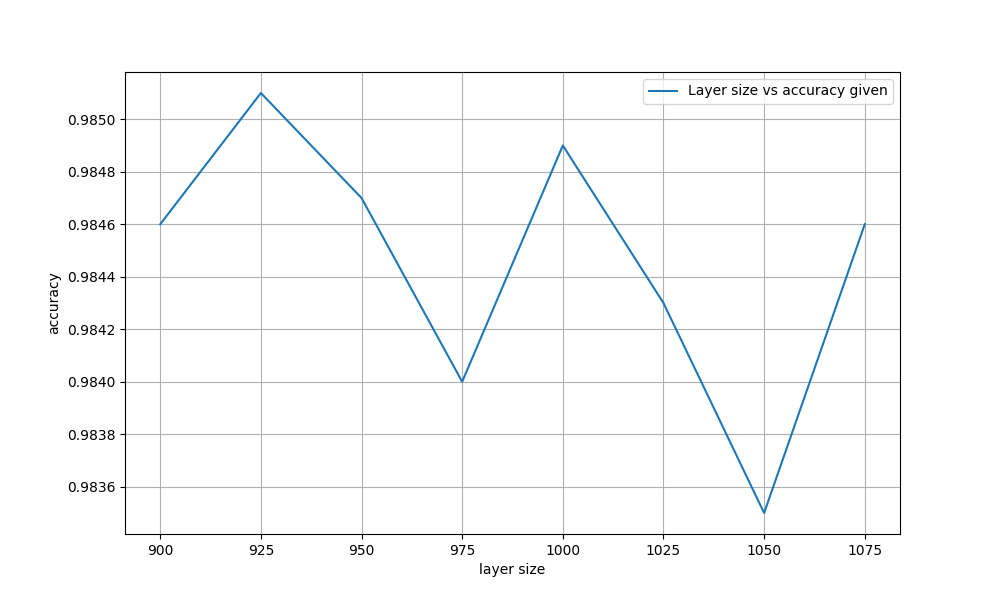


Figure 78: A more precise graph showing an accuracy peak at 925 layer size

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*.

A screen shot of a computer

AI-generated content may be incorrect.

Figure 79: The console log of the neural network being ran

# 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 computer resources were 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. I began by making another subsection in the hyperparameterChanges function.

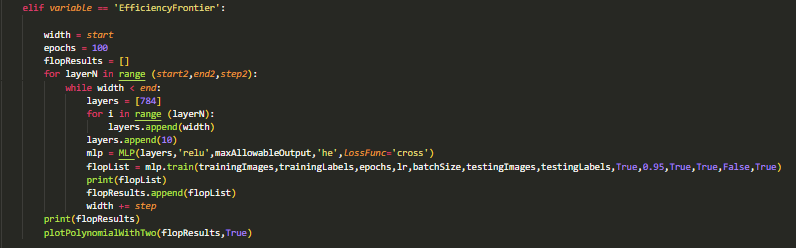


Figure 80: The code block inside hyperparameterChanges to investigate the compute efficiency frontier

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.



Figure 81: The code snippet used to generate figure 82 and 83 using figure 80

The product of this was at first this graph:

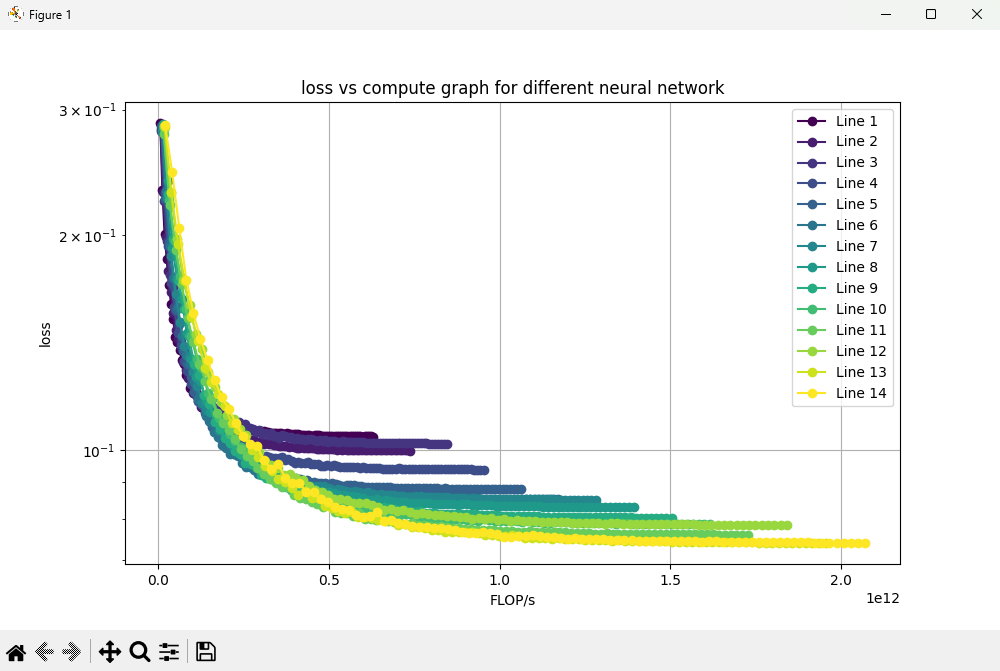


Figure 82: A collection of lines representing different neural network architectures. Their loss is compared to how much computational resources was required to obtain that loss

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

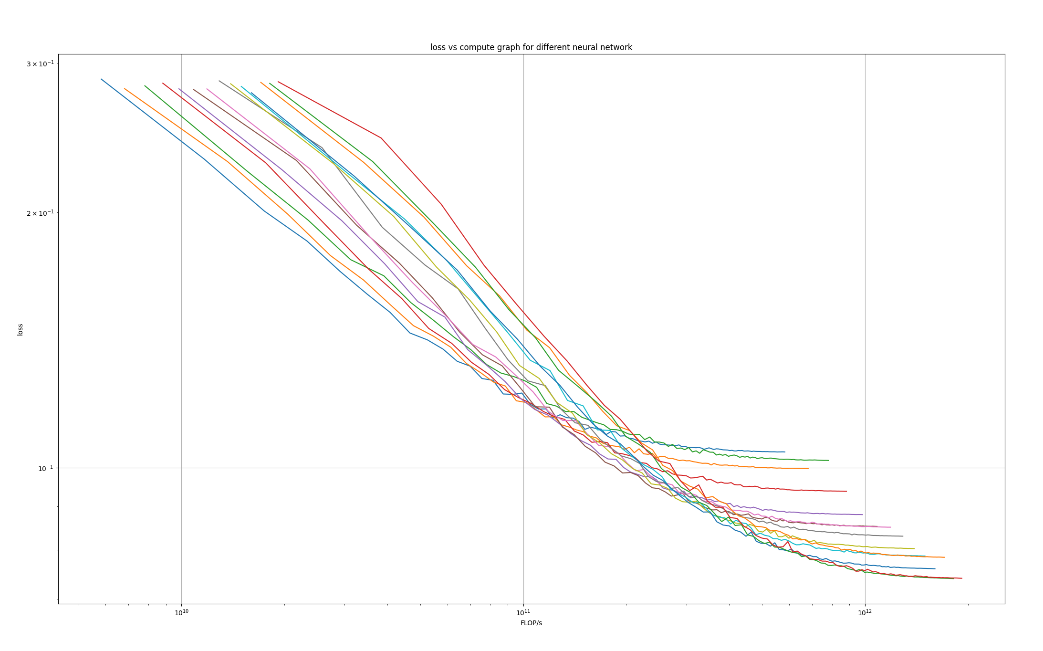


Figure 83: A straight boundary is visible as the axes are turned into log mode

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 *computing*.

# 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 specific circumstances.

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.

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 ideal loss function in every case was found to be cross-entropy loss as it is specialized in classification tasks like this one.

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 issues with 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

*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.

*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 techneque where separate columns represent classes and a one in a class signifies its presense. 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 to 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.

*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.