Multilayer Neural Network from scratch

How to run the code:

The code is also on google colab and can be found here:

https://colab.research.google.com/drive/1HDThZ30IR3iE86YRzeoxsklIPNrak6so

The parameters used for the model are summarised below:

- **PATH_TO_CODE_FOLDER** - Path of the code folder in which Algorithm, input and output folders are present.
- **Layers**:

  Each layer needs to be added in the below format for hidden layers variables.

  [Number_of_nodes,activation,dropout_rate]

  Example: [50,'LReLU',0] for hidden layer with 50 neurons and Leaky ReLu activation and with no drop out

  Input and output layers are not required to be provided in the inputs, they are automatically initialized depending on the dataset with softmax for output activation.

- **batch_size** - refer to size of the batch for Mini batch SGD
- **epochs** - Number of epochs to be used for training
- **learning_rate** - Rate of learning to be used while updating weights
- **momentum_term** - Value of gamma for "SGD with momentum". Please provide 0 for "SGD without momentum"
- **epoch_for_log_display** - The value of epoch for which loss and accuracy needs to be displayed during training
- **weight_decay_const** - Value of lambda for L2 weight regularisation. Please provide 0 for no regularisation.
- **NUMBER_OF_FOLDS** - Number of cross folds to be used while training the data for hyper parameter validation
- **NORMALIZE_DATA** - Boolean value i.e. True or False to enable of disable data normalization prior to training
- **batch_normalization** - Boolean value i.e. True or False to enable of batch normalization during training and testing
- **c_eps** - Values of epsilon to be used during batch normalization

There are 2 types of execution pathways:

1. Running with optimal parameters
2. Running with new parameters

Please follow the steps carefully.

---- RUNNING WITH OPTIMAL PARAMETERS ----

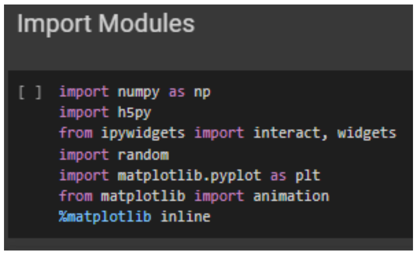
Optimal parameters:

- batch_size = 128
- epochs = 500
- learning_rate=0.5
- momentum_term = 0.9
- epoch_for_log_display = 10
- weight_decay_const = 0.0001
- NUMBER_OF_FOLDS = 5
- NORMALIZE_DATA = True
- batch_normalization = False #True
- Hidden layers:

3. [240, ‘LReLU’,0.05]
4. [220, ‘ReLU’, 0]

- c_eps = 1e-5

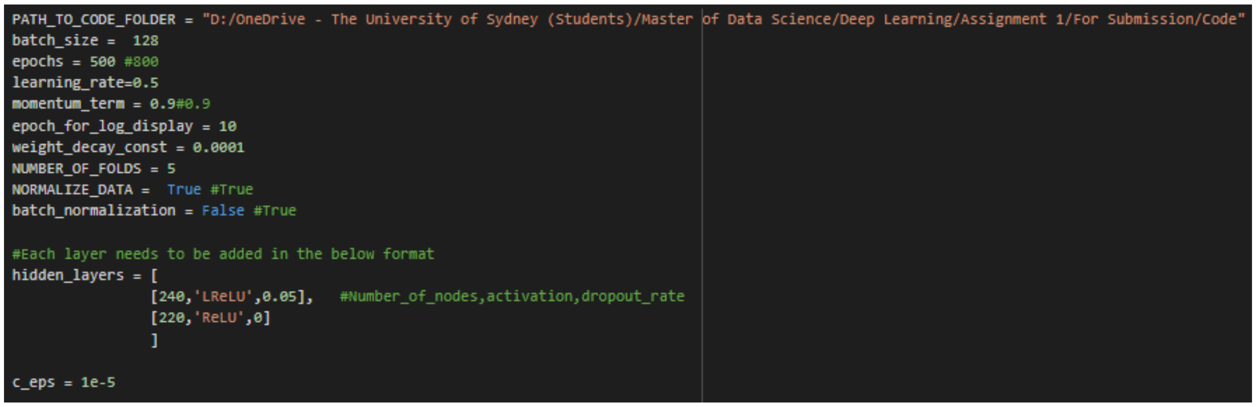
1. ***Load*** the packages with the first cell



Import Modules

```
[ ] import numpy as np
    import h5py
    from ipywidgets import interact, widgets
    import random
    import matplotlib.pyplot as plt
    from matplotlib import animation
    %matplotlib inline
```

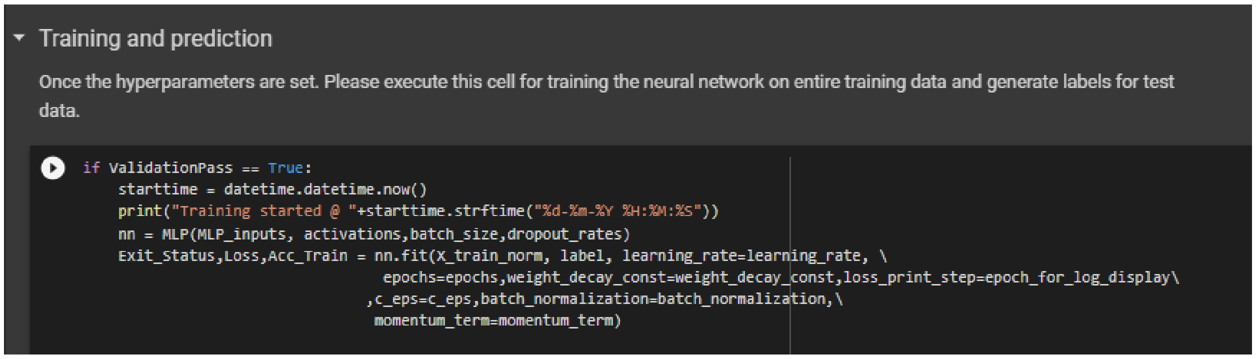
2. ***Run*** the next cell and ensure that the **PATH_TO_CODE_FOLDER** is changed and the optimal parameters are entered correctly



```
PATH_TO_CODE_FOLDER = "D:/OneDrive - The University of Sydney (Students)/Master of Data Science/Deep Learning/Assignment 1/For Submission/Code"
batch_size =  128
epochs = 500 #800
learning_rate=0.5
momentum_term = 0.9#0.9
epoch_for_log_display = 10
weight_decay_const = 0.0001
NUMBER_OF_FOLDS = 5
NORMALIZE_DATA =  True #True
batch_normalization = False #True

#Each layer needs to be added in the below format
hidden_layers = [
                [240,'LReLU',0.05],   #Number_of_nodes,activation,dropout_rate
                [220,'ReLU',0]
                ]

c_eps = 1e-5
```

3. ***Select*** the next cell labelled “**Load Data**” and select Runtime → Run After

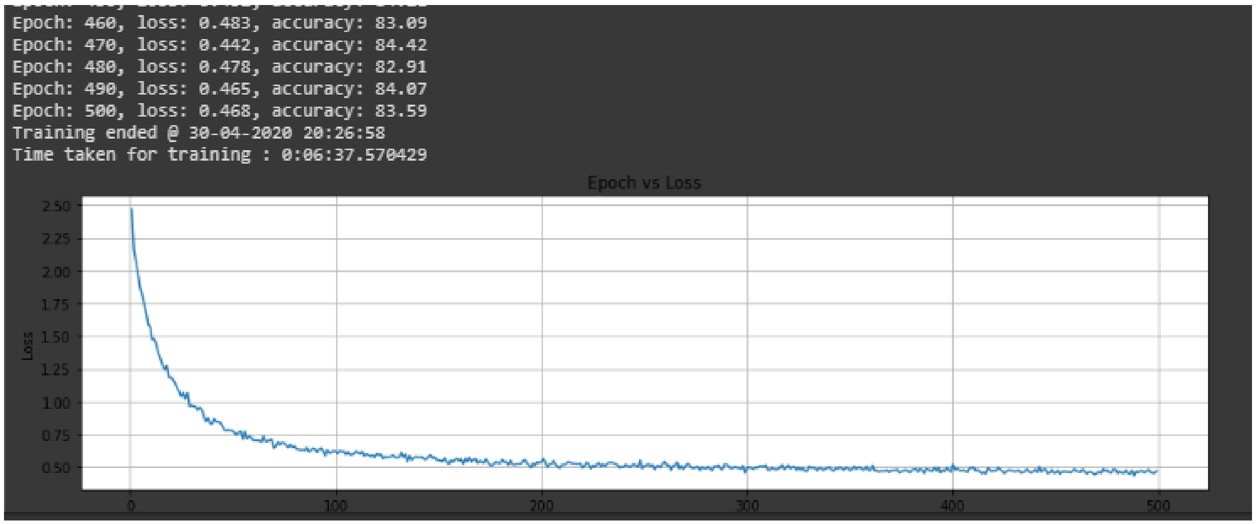
This allows automatic running of subsequent cells. The 2nd last cell labelled "**Training and Prediction**" is to be evaluated for the training accuracy of the model. The last cell called "**Evaluate hyper parameters via cross validation**" is <u>not</u> necessary to run for this step.



Training and prediction

Once the hyperparameters are set. Please execute this cell for training the neural network on entire training data and generate labels for test data.

```
if ValidationPass == True:
    starttime = datetime.datetime.now()
    print("Training started @ "+starttime.strftime("%d-%m-%Y %H:%M:%S"))
    nn = MLP(MLP_inputs, activations,batch_size,dropout_rates)
    Exit_Status,Loss,Acc_Train = nn.fit(X_train_norm, label, learning_rate=learning_rate, \
                                  epochs=epochs,weight_decay_const=weight_decay_const,loss_print_step=epoch_for_log_display\
                                 ,c_eps=c_eps,batch_normalization=batch_normalization,\
                                 momentum_term=momentum_term)
```

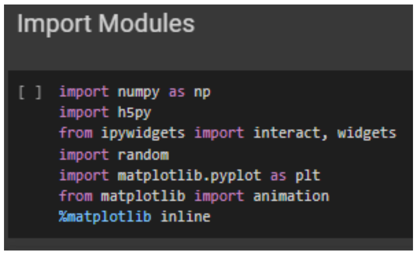


```
Epoch: 460, loss: 0.483, accuracy: 83.09
Epoch: 470, loss: 0.442, accuracy: 84.42
Epoch: 480, loss: 0.478, accuracy: 82.91
Epoch: 490, loss: 0.465, accuracy: 84.07
Epoch: 500, loss: 0.468, accuracy: 83.59
Training ended @ 30-04-2020 20:26:58
Time taken for training : 0:06:37.570429
```

---- RUNNING WITH NEW PARAMETERS ----

This is similar to running with the optimal parameters HOWEVER, every cell is run in sequential order <u>EXCEPT</u> for the "**Training and Prediction"** cell

1. ***Load*** the packages with the first cell

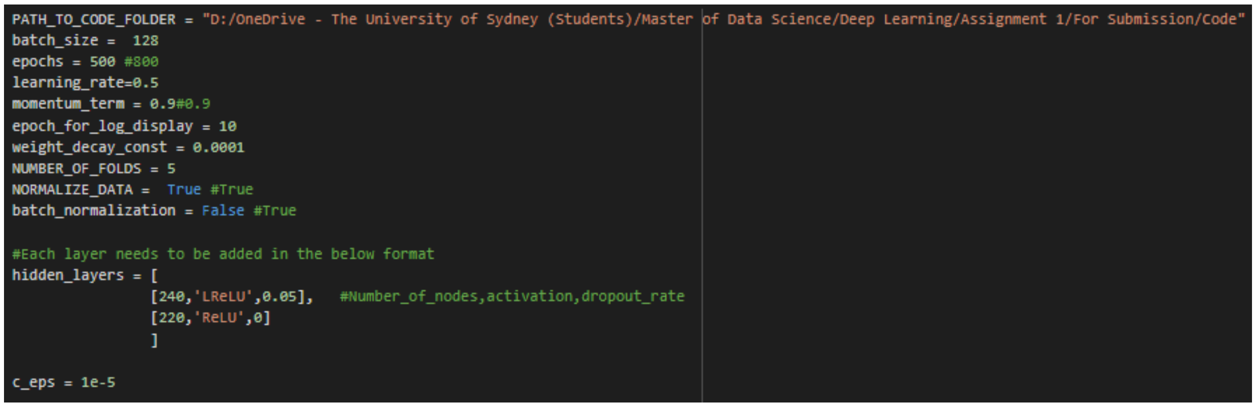


Import Modules

```
[ ] import numpy as np
    import h5py
    from ipywidgets import interact, widgets
    import random
    import matplotlib.pyplot as plt
    from matplotlib import animation
    %matplotlib inline
```

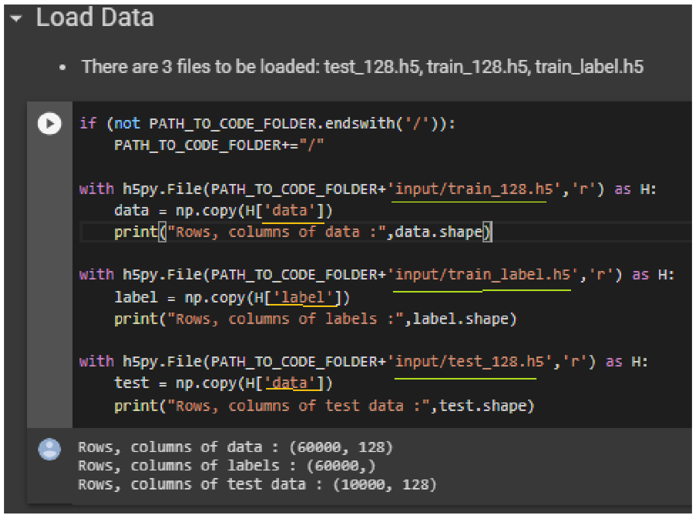
2. ***Change*** the parameters in the next cell if necessary then ***RUN*** it

This is summarised in the screenshot below.



```
PATH_TO_CODE_FOLDER = "D:/OneDrive - The University of Sydney (Students)/Master of Data Science/Deep Learning/Assignment 1/For Submission/Code"
batch_size =  128
epochs = 500 #800
learning_rate=0.5
momentum_term = 0.9#0.9
epoch_for_log_display = 10
weight_decay_const = 0.0001
NUMBER_OF_FOLDS = 5
NORMALIZE_DATA =  True #True
batch_normalization = False #True

#Each layer needs to be added in the below format
hidden_layers = [
                [240,'LReLU',0.05],   #Number_of_nodes,activation,dropout_rate
                [220,'ReLU',0]
                ]

c_eps = 1e-5
```

3. ***Load*** the data set in the next cell (run the cell).



Load Data

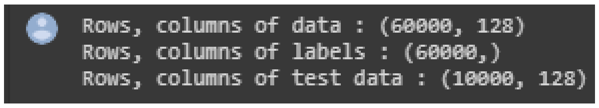
- There are 3 files to be loaded: test_128.h5, train_128.h5, train_label.h5

```
if (not PATH_TO_CODE_FOLDER.endswith('/')):
    PATH_TO_CODE_FOLDER+="/"

with h5py.File(PATH_TO_CODE_FOLDER+'input/train_128.h5','r') as H:
    data = np.copy(H['data'])
    print("Rows, columns of data :",data.shape)

with h5py.File(PATH_TO_CODE_FOLDER+'input/train_label.h5','r') as H:
    label = np.copy(H['label'])
    print("Rows, columns of labels :",label.shape)

with h5py.File(PATH_TO_CODE_FOLDER+'input/test_128.h5','r') as H:
    test = np.copy(H['data'])
    print("Rows, columns of test data :",test.shape)
```

```
Rows, columns of data : (60000, 128)
Rows, columns of labels : (60000,)
Rows, columns of test data : (10000, 128)
```

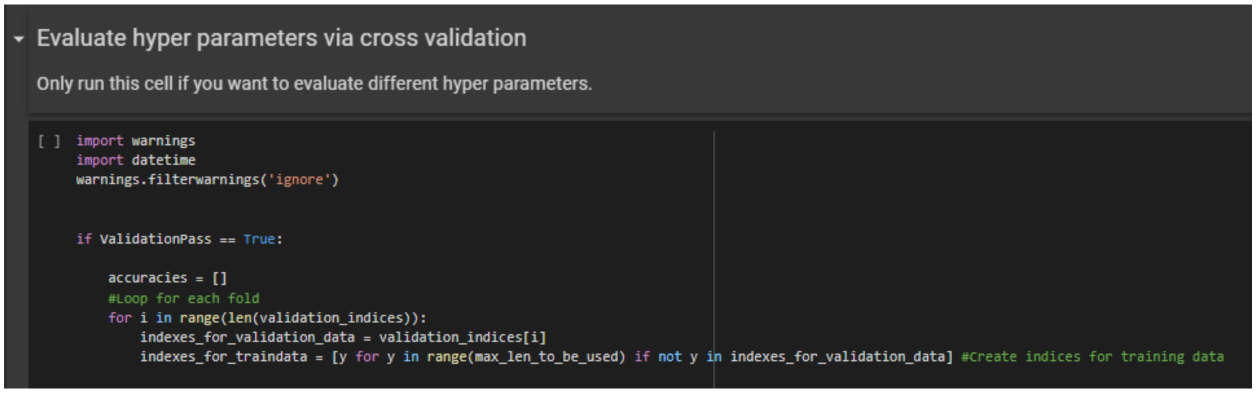
Please ensure that these file name correspond to the ones written in the code wrapped in the h5py brackets and the appropriate H[“ “] is used

Once successful, the dimensions of each folder should be printed.



```
Rows, columns of data : (60000, 128)
Rows, columns of labels : (60000,)
Rows, columns of test data : (10000, 128)
```

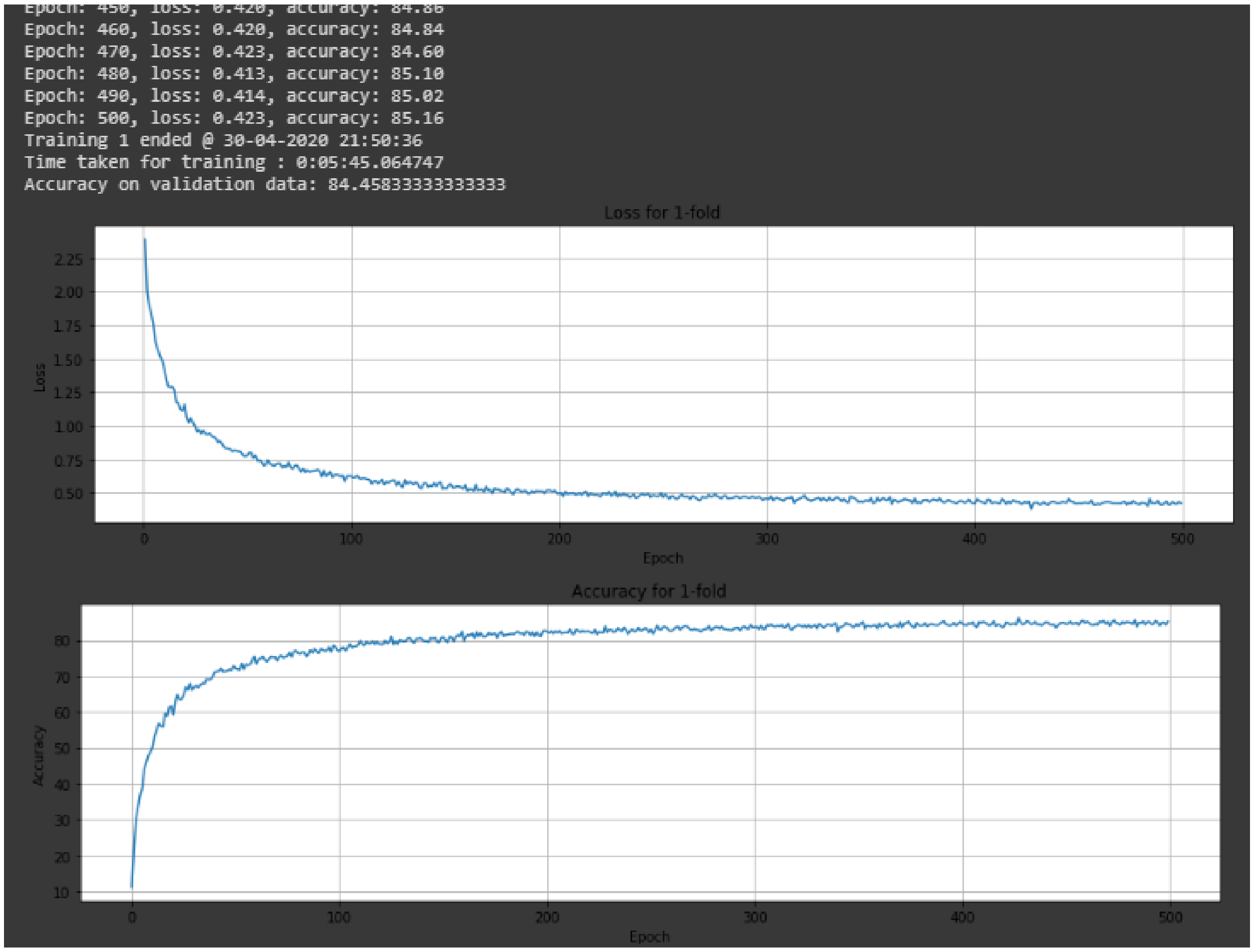
3. ***Run*** all cells up until “**Training and Prediction**” (DO NOT RUN TRAINING AND PREDICTION). Instead, run “***Evaluate hyper parameters via cross validation***”.



Evaluate hyper parameters via cross validation

Only run this cell if you want to evaluate different hyper parameters.

```
[ ]  import warnings
     import datetime
     warnings.filterwarnings('ignore')


     if ValidationPass == True:

         accuracies = []
         #Loop for each fold
         for i in range(len(validation_indices)):
             indexes_for_validation_data = validation_indices[i]
             indexes_for_traindata = [y for y in range(max_len_to_be_used) if not y in indexes_for_validation_data] #Create indices for training data
```



```
Epoch: 450, loss: 0.420, accuracy: 84.86
Epoch: 460, loss: 0.420, accuracy: 84.84
Epoch: 470, loss: 0.423, accuracy: 84.60
Epoch: 480, loss: 0.413, accuracy: 85.10
Epoch: 490, loss: 0.414, accuracy: 85.02
Epoch: 500, loss: 0.423, accuracy: 85.16
Training 1 ended @ 30-04-2020 21:50:36
Time taken for training : 0:05:45.064747
Accuracy on validation data: 84.45833333333333
```

Table of Contents

Introduction

Aim of the study

The aim of this study is to create a multi-class classifier which utilizes various optimization techniques and analyzing their performance on the given data set.

In this particular study, the classifier is a multilayer perceptron. It uses supervised learning to classify linearly separable datasets. This report will explore various optimization and regularization techniques and examine how they affect the model's performance.

Why the study is important

Deep learning is a crucial aspect of the machine learning world whereby complex inputs are synthesised layer by layer in order to label the outcome. It is an extremely powerful tool which studies large data-sets and extracts patterns to make predictions which can be applied in future applications (Murnane, 2016)

Regarding this study, regularisation and optimisation are key concepts to boost the efficiency and accuracy of the model. Regularisation is utilised to reduce overfitting, a phenomenon where the model appears to memorise from the input examples rather than generalising the patterns from it. This makes the model too specific to the initial training example and will not be suitable for new examples.

Optimisation strategies are also used for Deep Learning in order to minimise errors between the actual and predicted results in an efficient manner. This is done through changing or adding parameters to the existing algorithms to achieve certain effects.

Overall, there are many procedures available to optimise and regularize our model and it is crucial to determine their performances and effects so we can have a clearer understanding of what should be used for certain situations in the future.

Methods

Specifications

The neural network was run on Google Colab with the following specs:

- GPU: Tesla P100-PCIE-16GB (UUID: GPU-c89d4de5-1ec9-fce2-4e0d-729ff8b74129)
- CPU: Intel(R) Xeon(R) CPU @ 2.00GHz
- CPU count: 2
- RAM: 13G
- Hard disk space: 34G

Pre-processing - normalisation

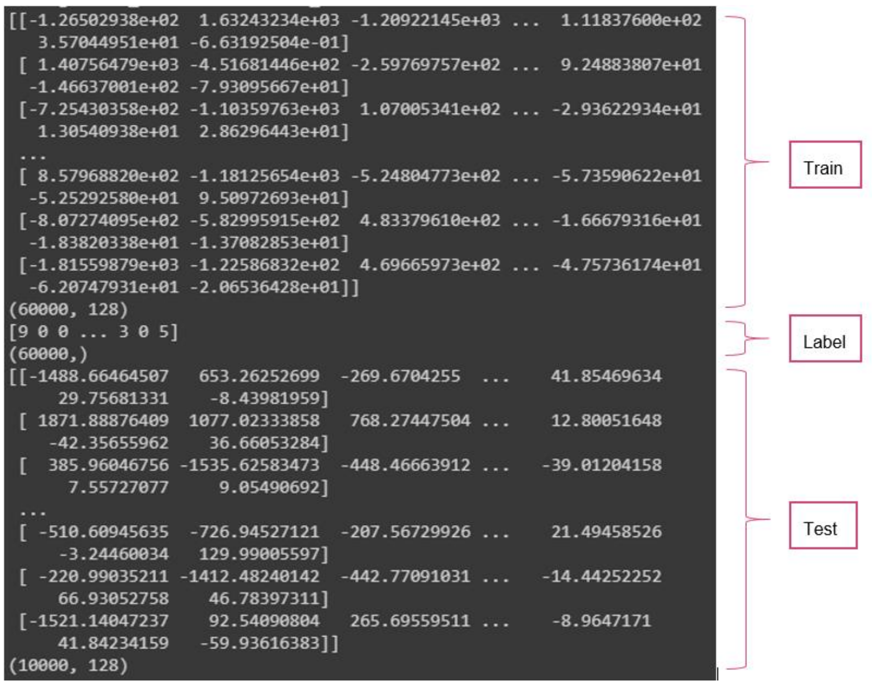

```
[[-1.26502938e+02  1.63243234e+03 -1.20922145e+03 ...  1.11837600e+02
   3.57044951e+01 -6.63192504e-01]
 [ 1.40756479e+03 -4.51681446e+02 -2.59769757e+02 ...  9.24883807e+01
  -1.46637001e+02 -7.93095667e+01]
 [-7.25430358e+02 -1.10359763e+03  1.07005341e+02 ... -2.93622934e+01
   1.30540938e+01  2.86296443e+01]
 ...
 [ 8.57968820e+02 -1.18125654e+03 -5.24804773e+02 ... -5.73590622e+01
  -5.25292580e+01  9.50972693e+01]
 [-8.07274095e+02 -5.82995915e+02  4.83379610e+02 ... -1.66679316e+01
  -1.83820338e+01 -1.37082853e+01]
 [-1.81559879e+03 -1.22586832e+02  4.69665973e+02 ... -4.75736174e+01
  -6.20747931e+01 -2.06536428e+01]]
(60000, 128)
[9 0 0 ... 3 0 5]
(60000,)
[[-1488.66464507   653.26252699  -269.6704255  ...    41.85469634
     29.75681331    -8.43981959]
 [ 1871.88876409  1077.02333858   768.27447504 ...    12.80051648
    -42.35655962    36.66053284]
 [  385.96046756 -1535.62583473  -448.46663912 ...   -39.01204158
      7.55727077     9.05490692]
 ...
 [ -510.60945635  -726.94527121  -207.56729926 ...    21.49458526
     -3.24460034   129.99005597]
 [ -220.99035211 -1412.48240142  -442.77091031 ...   -14.44252252
     66.93052758    46.78397311]
 [-1521.14047237    92.54090804   265.69559511 ...    -8.9647171
     41.84234159   -59.93616383]]
(10000, 128)
```

Fig 1. Structure of original data

The current data consists of 60,000 examples with 128 attributes. Example values of the data and their shape can be seen in ***Fig 1***. Normalisation is required before feeding it into the perceptron model because it is difficult to extract patterns as some values may overshadow the importance of others. Not only does normalisation improve numerical stability, it often reduces training time as the values are simpler, which is crucial when working with large data sets (Zhang, 2019). Below in ***Fig 2.*** is the output of the dataset after normalisation which utilises the min-max normalisation formula.

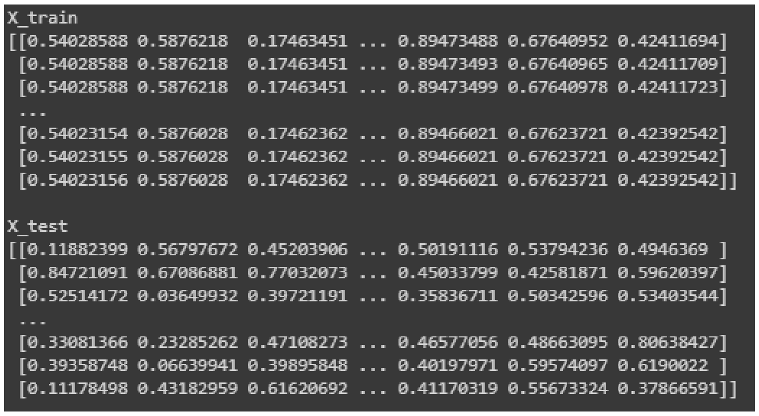

```
X_train
[[0.54028588 0.5876218  0.17463451 ... 0.89473488 0.67640952 0.42411694]
 [0.54028588 0.5876218  0.17463451 ... 0.89473493 0.67640965 0.42411709]
 [0.54028588 0.5876218  0.17463451 ... 0.89473499 0.67640978 0.42411723]
 ...
 [0.54023154 0.5876028  0.17462362 ... 0.89466021 0.67623721 0.42392542]
 [0.54023155 0.5876028  0.17462362 ... 0.89466021 0.67623721 0.42392542]
 [0.54023156 0.5876028  0.17462362 ... 0.89466021 0.67623721 0.42392542]]

X_test
[[0.11882399 0.56797672 0.45203906 ... 0.50191116 0.53794236 0.4946369 ]
 [0.84721091 0.67086881 0.77032073 ... 0.45033799 0.42581871 0.59620397]
 [0.52514172 0.03649932 0.39721191 ... 0.35836711 0.50342596 0.53403544]
 ...
 [0.33081366 0.23285262 0.47108273 ... 0.46577056 0.48663095 0.80638427]
 [0.39358748 0.06639941 0.39895848 ... 0.40197971 0.59574097 0.6190022 ]
 [0.11178498 0.43182959 0.61620692 ... 0.41170319 0.55673324 0.37866591]]
```

Fig 2. Training and testing data after normalisation

Structure of the model

The shapes of each file are depicted in the table below.

File	**Shape**
Train	(60 000,128)
Label	(60 000,)
Test	(10 000, 128)

Fig 3. Table summarising the dimensions of each file

The number of rows correspond to the number of examples in the data set and the columns coincide with the number of attributes. The target are the classes belonging to the training data set examples. There are 60,000 examples in the training set which have 128 attributes and 10 000 examples are used as the test set. With the knowledge of the data-set dimensions, the input, hidden and output layer sizes can be determined.

Input Layer

For the input layer, each neuron corresponds to one feature of the model. Since there are 128 attributes in the dataset, there are 128 input neurons.

2 Hidden Layers

The number of hidden layers was decided through experimentation and the most optimal was 240 neurons in the first hidden layer and 220 in the second hidden layer.

Output layer

The output layer should result in the differing classifications of the data-set. In this case, there are 10 different classes, hence, 10 neurons are required.

In order to improve the accuracy and efficiency of the model, various modules should be implemented. These modules will be discussed and analysed in the Principles of Different Modules section.

Principles of different modules

Hidden layer (more than 1)

Hidden layers allow for the function of a neural network to be broken down into specific transformations of the data. Each hidden layer function is specialized to produce a defined output based on specified activation functions. This is crucial so that the hidden layer transforms the input into something suitable for the next layer to use to introduce more complexity in the learning process.

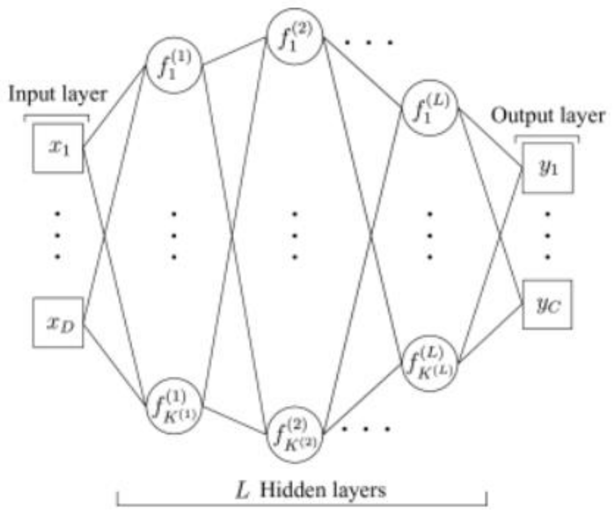


Fig 4. Hidden layers in neural network

ReLu Activation

It is very important to add a non linear activation to learn complex relationships, otherwise the relationship between input and output will always be linear, even after introducing multiple hidden layers.

This is when ReLu activation plays an important role by introducing the below to the neural network.

1. Non-linearity at the required layers
2. Being faster and efficient than other activation functions (as it does not involve much computations).
3. No vanishing gradients

ReLu works by outputting 0 if the input is less than or equal to 0. Otherwise the output is the same as the input. This is summarised in the formula and visualisation below in ***Fig 5***.

$$a_j^i = f(x_j^i) = \max(0, x_j^i)$$

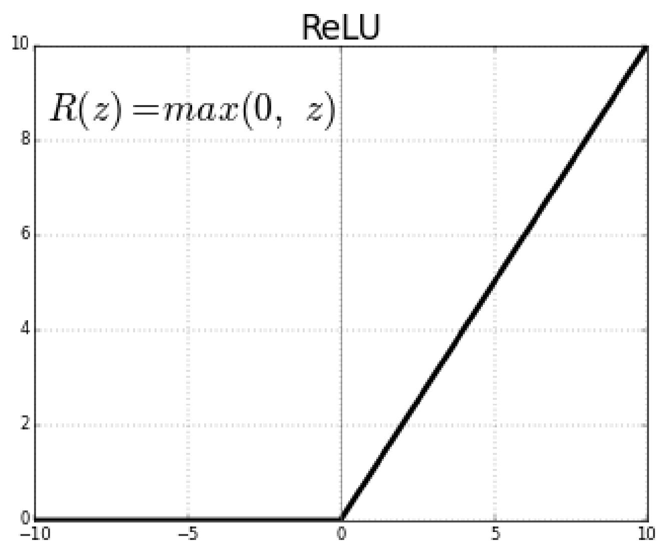


Fig 5. ReLu activation

Leaky ReLu Activation

Unfortunately, ReLU units can be fragile during training and can “die”. For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again. If this happens, then the gradient flowing through the unit will forever be zero from that point on and the model does not learn further.

Leaky ReLUs are one attempt to fix the “dying ReLU” problem. Instead of the function being zero when x < 0, a leaky ReLU will instead have a small negative slope (of 0.01, or so). In ***Fig 6.***, there is only one point in which the output will be 0 and this is when x = 0. Otherwise, negative values have a small result.

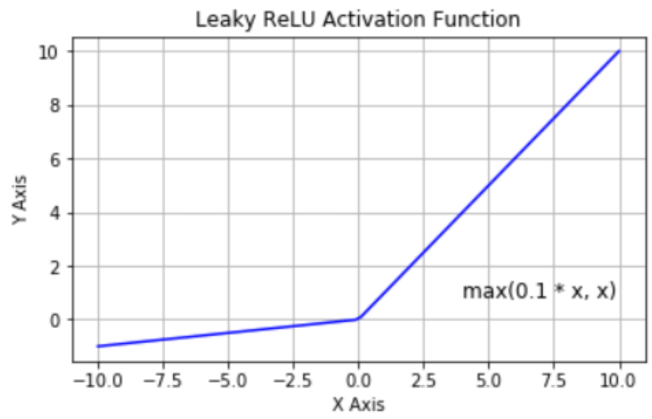


Fig 6. Leaky ReLU activation

Weight decay

An increase in hidden layers and activations may make the model too complex and hence may overfit. However, weight decay is a method used to penalise the complexity and not compromise the important parameters in the model.

During training of the neural network, after each update, a weight decay term is added onto the loss function to penalize large weights, thereby limiting how much the weight grows in a network. The formula for the L2 regularization as a penalty to the loss function is as below:

$$L = L_{\text{data}} + \frac{\lambda}{2}||W||_2^2$$

Fig 7. Weight decay formula

If this is differentiated further, the decay of weight at each gradient step is depicted to ***Fig 8.***

$$w_{t+1} = w_t - \alpha \Delta_w L_{\text{data}} - \lambda w$$

Fig 8. Gradient step formula

Momentum in SGD

Stochastic gradient descent

In Vanilla Batch Gradient Descent, gradients are computed iteratively for the entire data set to update the weights during backpropagation. This is time consuming and computationally intensive, especially for large data sets. Stochastic Gradient Descent (SGD) is an algorithm which aims to optimise the function as well as reduce computation. Instead of updating after running through the whole training set, it uses one random training example to compute the gradient in every iteration. This is much more efficient in terms of computational power and time. This formula is shown in ***Fig. 9*** whilst the Vanilla Gradient Descent is in ***Fig. 10.***

$\theta = \theta - \eta \cdot \nabla_\theta J(\theta, \mathcal{X}^{(i)})$ each examp	$\theta = \theta - \eta \cdot \nabla_\theta J(\theta, \mathcal{X}^{(1:end)})$ all
Fig 9. Stochastic Gradient Descent Formula	***Fig 10. Gradient Descent Formula***

One DISADVANTAGE would be if the randomly chosen samples are outliers then the path to reach the local minimum can be noisier and might produce high error. While **SGD is more efficient** than standard gradient descent because there are less samples to focus on, the error function is not as minimised as it would be in gradient descent because of the convergence as shown in the below figure.

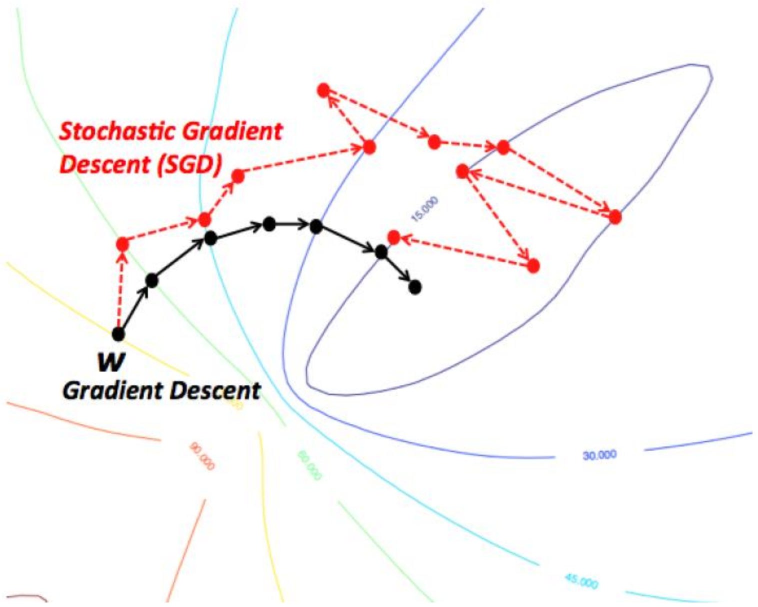


Fig 11. SGD vs Gradient descent

Momentum

In order to deal with the problem of convergence to reach the local minima introduced by SGD, we can use **Momentum** along with SGD. Instead of using only the gradient of the current step to guide the search, momentum is an additional term which also accumulates the gradient of the past steps. It helps determine the direction to go and benefits by speeding up the learning process and dampens oscillations. In ***Fig 12***. the oscillations are much smoother with momentum in comparison to without momentum.

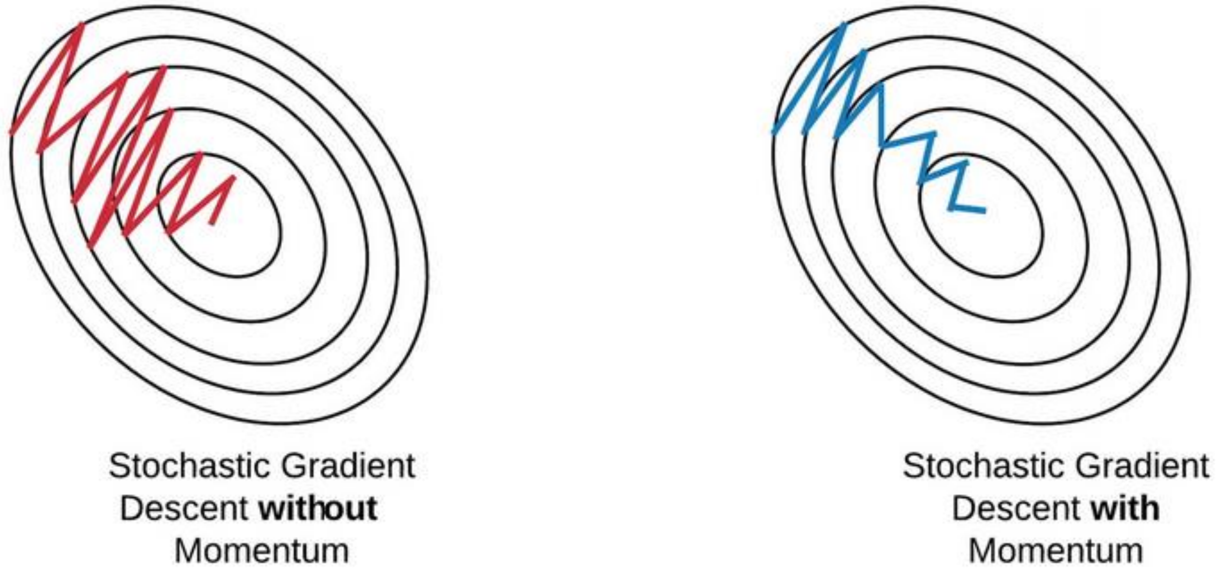

Stochastic Gradient Descent **without** Momentum
Stochastic Gradient Descent **with** Momentum

Fig 12. SGD without Momentum vs SGD with Momentum

The Momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions.

$$
\begin{aligned}
&v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \\
&\theta_t = \theta_{t-1} - v_t
\end{aligned}
$$

Fig 13. SGD with Momentum formula

From the equation above, there are 2 hyperparameters to calculate the velocity (vt). Once this velocity is determined, this is subtracted from the parameter Θ to adjust it in the correct direction.

Dropout

In a fully connected layer neurons develop co-dependency amongst each other during training which curbs the individual power of each neuron leading to over-fitting of training data. We can use dropout to deal with this problem.

Dropout refers to ignoring randomly chosen neurons i.e. units are not considered during a particular forward or backward pass during the training phase.

At each training stage, individual nodes are either dropped out of the net with probability 1-p or kept with probability p, so that a reduced network is left; incoming and outgoing edges to a dropped-out node are also removed.

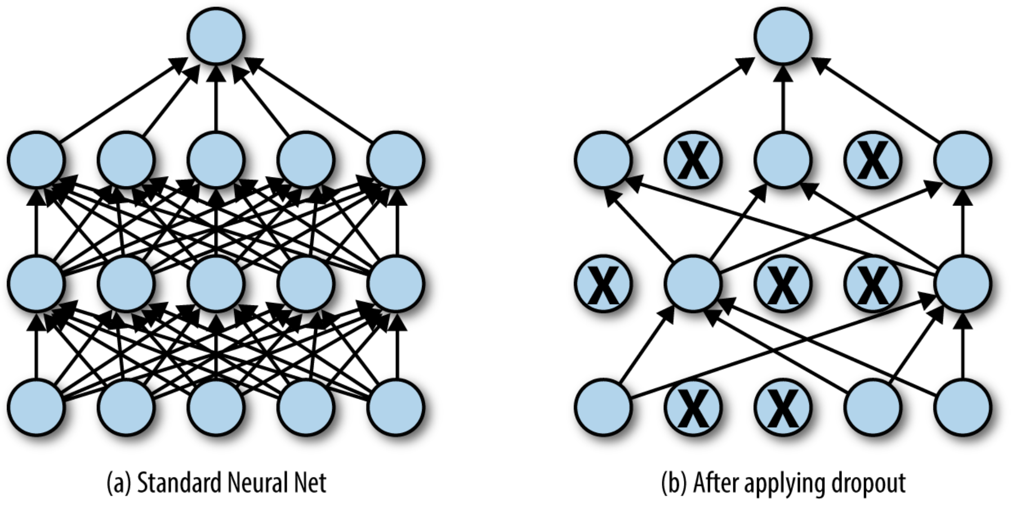


Fig 14. Neural network before and after dropout

Dropout forces a neural network to learn more robust features that are useful in conjunction with many different random subsets of the other neurons. Dropout does not penalize the use of large weights and is also invariant to parameter scaling.

Softmax and entropy loss

Softmax

Softmax function is implemented on the output layer's input to normalize the value into a vector of values that follows a probability distribution whose total sums up to 1. This function is useful in determining which class the given inputs belong to.

The output values are between the range [0,1] which is nice because we are able to avoid binary classification and accommodate as many classes or dimensions in our neural network model.

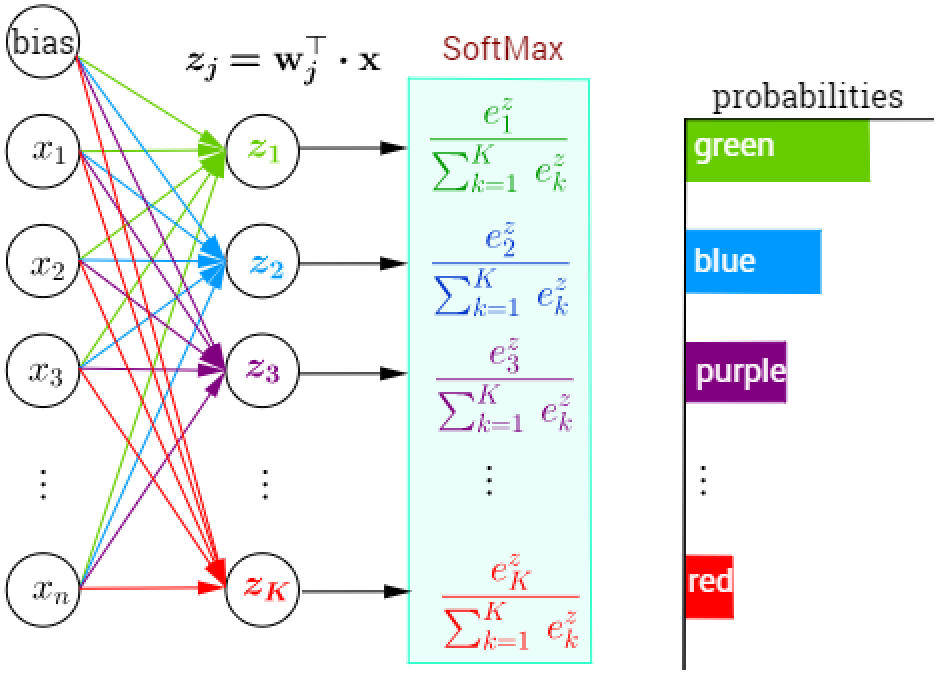

bias
$z_j = \mathbf{w}_j^\top \cdot \mathbf{x}$
SoftMax
probabilities
x_1
x_2
x_3
x_n
z_1
z_2
z_3
z_K
$\frac{e_1^z}{\sum_{k=1}^K e_k^z}$
$\frac{e_2^z}{\sum_{k=1}^K e_k^z}$
$\frac{e_3^z}{\sum_{k=1}^K e_k^z}$
$\frac{e_K^z}{\sum_{k=1}^K e_k^z}$
green
blue
purple
red

Fig 15. Demonstrating SoftMax application and SoftMax formula

Cross Entropy Loss

Cross entropy loss is a technique used after applying softmax to measure the performance of the model based on the probability of each class. If cross entropy loss increases, that means the model's predicted value is diverging from the actual label.

This module ensures that the difference between the two probability distributions of the predicted and actual values are minimised.

$$L_{\text{cross-entropy}}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_i y_i \log(\hat{y}_i)$$

Fig 16. Cross Entropy loss formula

Mini Batch training

With SGD we introduce a problem of convergence to reach the local minima due to noisy samples, in order to overcome this we can choose a group of samples i.e. mini-batch rather than one sample while calculating gradient descent. In this way the gradient is not susceptible to outliers and the computation is also less as compared with gradient descent. So our mini-batch will have different n samples (randomly chosen) for every iteration and will reach local optima in less number of iterations as shown below.

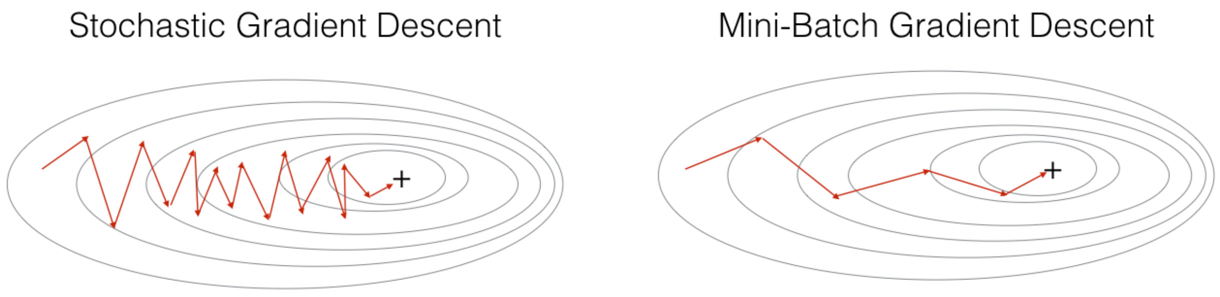


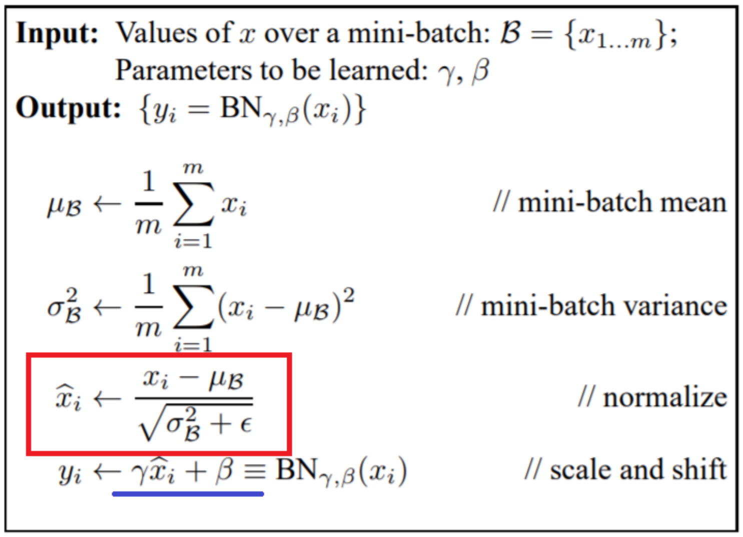
Fig 17. Convergence with SGD vs Mini-Batch

Mini Batch gradient descent brings the best of Batch Gradient Descent and Stochastic Gradient Descent.

Batch Normalization

Some data sets may require normalization in order to regulate the effects of different features in a model and ensure that the data is within a reasonable range so they aren't overshadowed by others.

The output of one layer directly affects the input of the next and may lead to exploding or vanishing gradients due to this change in distribution. This phenomenon is called covariate shift. By normalising the distribution at each layer, we can avoid this so that the layers don't affect the next layers much.



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\boxed{\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}} \qquad \text{// normalize}$$

$$y_i \leftarrow \underline{\gamma\widehat{x}_i + \beta} \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Fig 18. Batch Normalisation equations over a mini-batch

Batch normalization reduces the amount by which the hidden unit values shift and allows each layer of a network to learn by itself a little bit more independently of other layers. During Batch normalization inputs to neural networks are normalized to either the range of [0, 1] or [-1, 1] or to mean=0 and variance=1. We then scale the data as $y_i \leftarrow \gamma\hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i)$ to ensure the networks recovers from any inconsistencies we have introduced

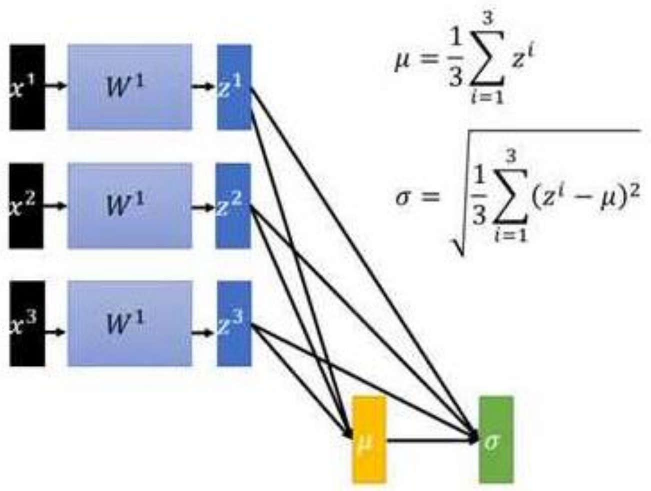

x^1
x^2
x^3
W^1
W^1
W^1
z^1
z^2
z^3
μ
σ
$\mu = \frac{1}{3}\sum_{i=1}^{3} z^i$
$\sigma = \sqrt{\frac{1}{3}\sum_{i=1}^{3}(z^i - \mu)^2}$

Fig 19. Batch normalization example of mean = 0, variance =1 at one layer

We also need to pass through the normalized layer during back propagation. We calculate the gradient for the same using the below formula.

$$\frac{\partial l}{\partial \mu_B^{(k)}} = \sum_{i=1}^{m} \frac{\partial l}{\partial y_i^{(k)}} \frac{-\gamma}{\sqrt{\sigma_B^{(k)^2} + \epsilon}} + \frac{\partial l}{\partial \sigma_B^{(k)^2}} \frac{1}{m} \sum_{i=1}^{m} (-2) \cdot (x_i^{(k)} - \mu_B^{(k)})$$

Cross Validation

Cross validation is a resampling procedure used for deep learning models to validate its performance under limited data. The experiments and results shown below all use k-fold cross validation where k is set to 10. This value was considered to estimate the model well under low bias and modest variance. Cross validation works by first randomizing the order of the entire dataset. Then, based on the given k value, separate the data into k subsamples. Each subsample is treated as validation data to test the model while the rest are training data, used to train the model. Thus, all observations are used for both training and testing.
The output accuracy for 10 fold will be averaged to get the final accuracy. An example diagram of the procedure is depicted in Fig 20.

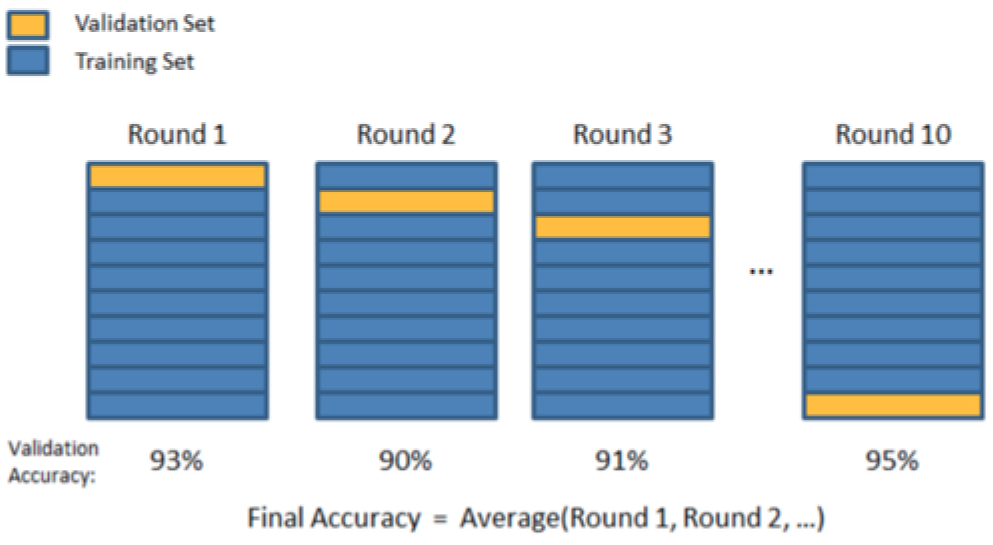


Fig 20. Cross validation procedure using k=10

Experiments and results

Data Normalization

Normalizing the data is an essential part of the neural network as it helps the model learn more accurately. To summarise the plot below:

- The normalised data set had an accuracy of 70% compared to 10% for the original
- The normalised data also caused an exploding gradient problem.

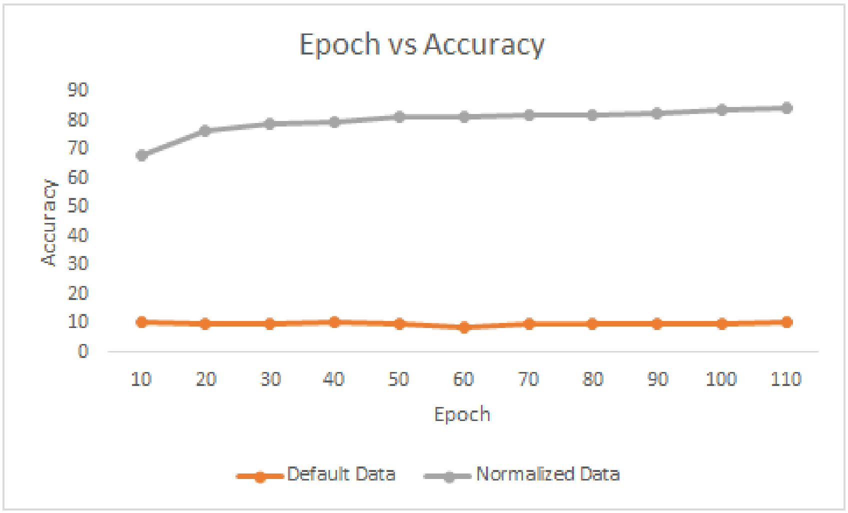


Fig 21. Comparison of accuracies between the original and normalised versions of the dataset

The reason for this increase in accuracy is that the inputs are easier to work with, as mentioned previously in the Principles of Different Modules section. The plot shows almost a 60% increase in accuracy. Therefore, it was much more beneficial to use the normalised version.

Epochs

Epochs refers to the amount of runs the model goes through the entire data set.

To summarise from the plots captured during experiments:

1. Increase in epochs leads to a linear increase in execution time
2. After 400-500 epochs the loss flattens
3. After 400-500 accuracy also does not increase high
4. Our model might overfit on the training data after 800 epochs
5. Limiting the epochs after 500-800 would be computationally efficient

An increase in epochs tend to give higher accuracy due to having more chances to learn. However, there is a danger of using too many epochs which may lead to overfitting of the dataset. This is due to the model attempting to memorise the data set rather than generalising it.

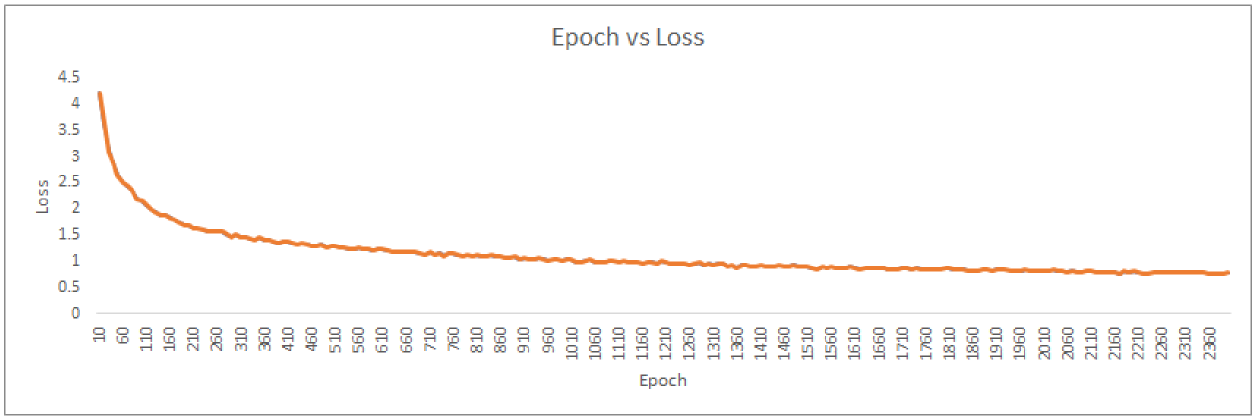


Fig 22. Loss value over epochs

It can be observed in ***Fig 22.*** that there is an exponential decrease in the loss in the initial sets of epochs. Over various epochs, the loss curve becomes shallower as the model is reaching more suitable parameters. The accuracy has a similar pattern in ***Fig 23***.

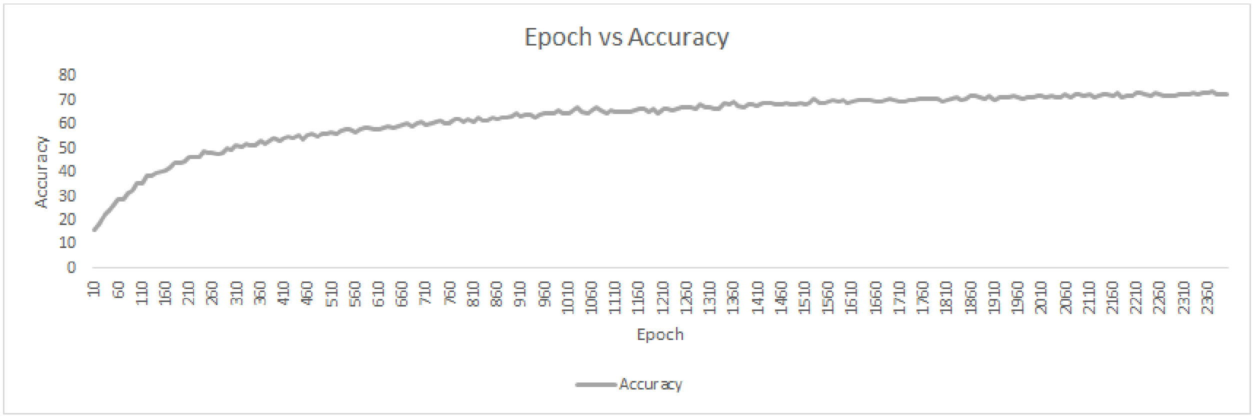


Fig 23. Accuracy over epochs

At around 500 epochs, the increase in accuracy is minimal from then onwards, but there is still an increasing relationship. As mentioned before, there is a chance that the model could be overfitting. To help determine what epoch value is most suitable, the execution time is also analysed in ***Fig 24.***

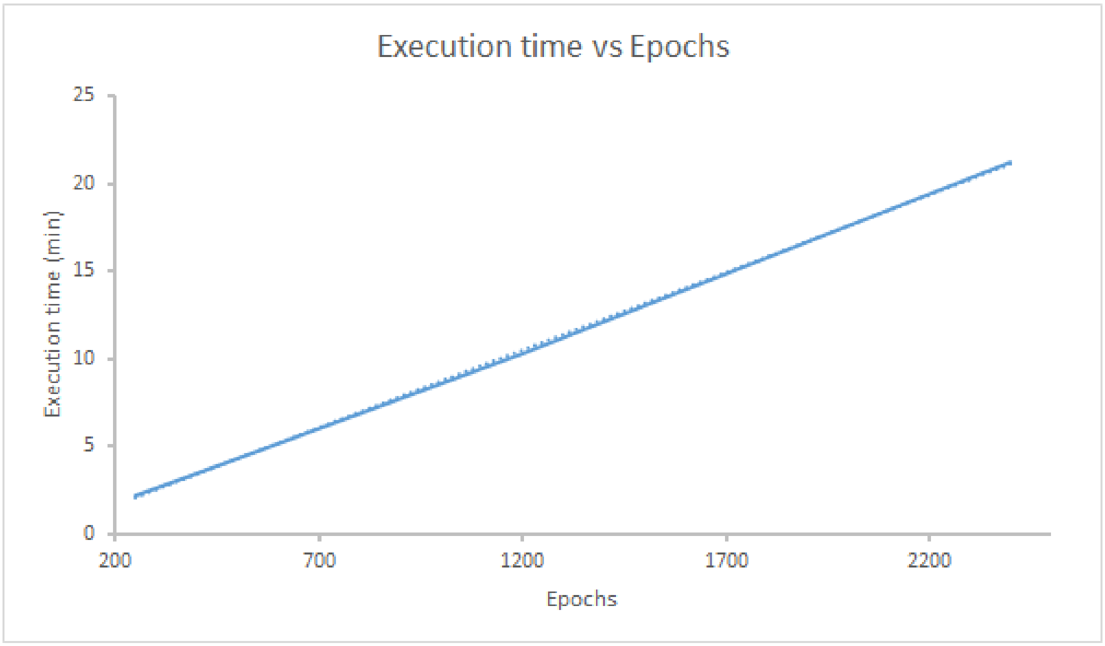


Fig 24 .Execution time for increasing epochs

An increase in epochs will subsequently increase the execution time of the code. It will require the model to run the data set for longer loops and hence take longer amounts of time, which is an issue for large data sets. There is a linear relationship between these 2 variables which is depicted in the graph below.

It was decided that the training shouldn’t take more than 10 minutes but the accuracy should still bre reasonably high, therefore around 500-800 epochs was considered optimal.

Learning rate

Learning rate is a crucial element that determines the step size at each iteration. This hyperparameter varies from different data sets but in most instances, 0.1 is used because it is more likely to reach a minima given enough iterations. Larger learning rates have the tendency to jump back and forth around the minima.

Some observation from our experiments:

- Accuracy from 0.01 to 0.1 substantially increased
- Accuracy from learning rate of 0.5 showed decreased
- Learning rates that are too large dramatically decreased accuracy

These can be generalised as below

- **Small Learning Rate -** demand more epochs for the model to learn leading to computation and time inefficiency.
- **Large Learning Rate** - might not reach local minima
- **Optimum Learning Rate** - determined through trial and error to find a computation that would reach a local minima

Time Taken for 0.5 learning rate : 6 minutes, 45 seconds for 800 epochs.

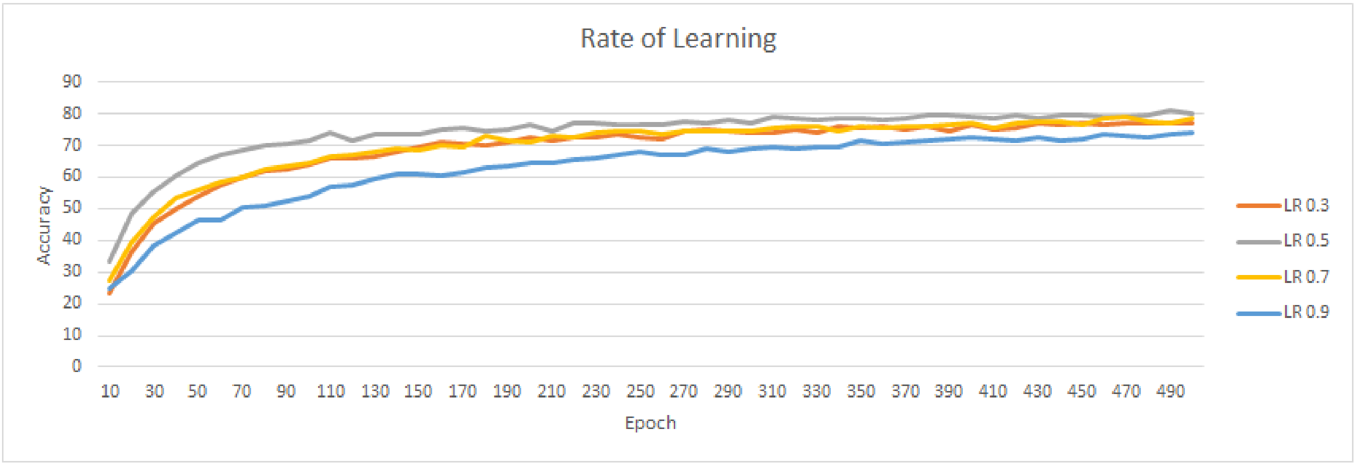


Fig 25. Accuracy over Epochs for different learning rates

It is clear from this diagram there is no real trend amongst the different learning rates and is very much dependent on the model's parameters and the data set given. 0.5 showed a consistent accuracy after 300 as the curve plateaus whilst other learning rates are still linearly increasing.

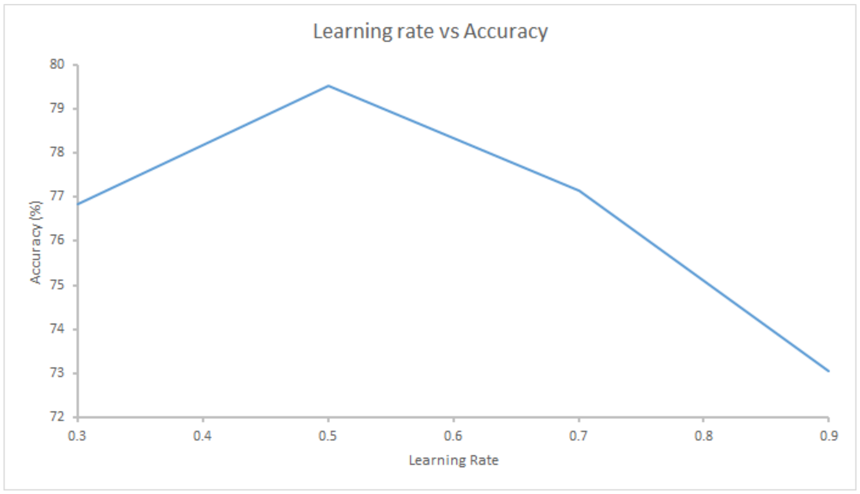


Fig 26. Learning rate vs Accuracy

It is clear that there is a learning rate that is suitable among the other parameters used which is 0.5. It shows the highest point in accuracy whilst a very high learning rate of 0.9 showed a sharp decrease in accuracy. The reason for these is dependent on how the model navigates towards a minima using the correct step. Too small of a learning rate may lead to very slow convergence whilst too high may overshoot the minima.

Weight decay

Weight decay is a regularisation technique which penalizes very high weights. L2 regularisation was utilised in this network.

The findings are summarised in the below points:

- Some values of weight decay like 0.001 help the network in achieving better accuracy
- Weight decay coupled with differing learning rates can cause unexpected behaviour
- In a neural network with only ReLU activations which cause dead ReLu after a while, weight decay has a very small impact.
- For large weights especially when sigmoid and tanh are not used, L2 regularisation can have a drastic impact on the network.

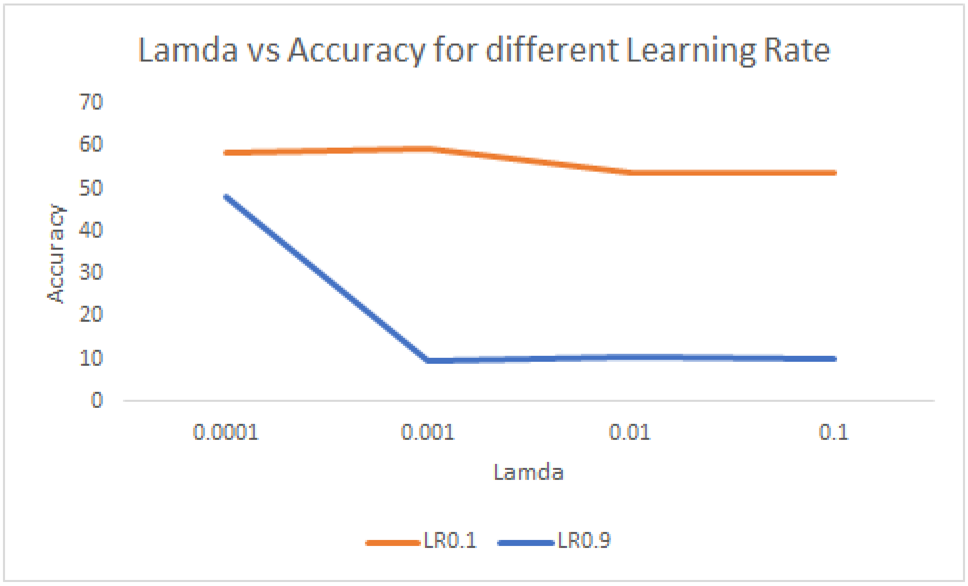


Fig 27. Weight decay accuracy with different learning rates

Momentum in SGD

There is an exponential relationship among epochs and accuracy with the momentum implemented in comparison with the linear relationship without it. This ties in with the property of momentum accelerating gradients in the correct direction which allows for faster convergence of the loss function.

Our observations:

- Good increase in accuracy with momentum.
- Quick decrease in loss with momentum.
- SGD without momentum has a steadier decline in loss.

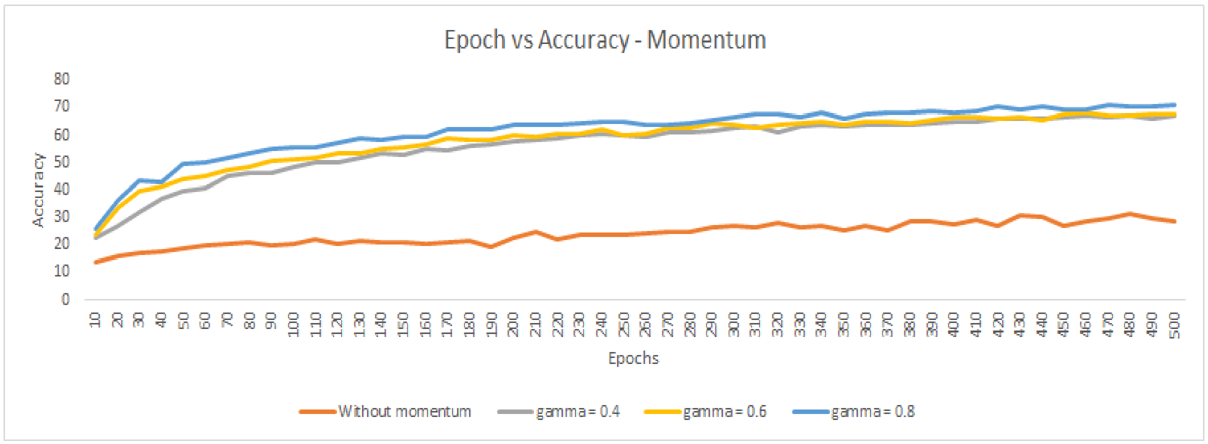


Fig 28. Accuracy of different momentum values

As shown on the graph, the accuracy has significant increase while the momentum term is applied. Different values of momentum terms have small differences, 0.8 is slightly better than the other two. Same goes the loss, applying momentum significantly reduces the loss, and different values gives little difference in loss change.

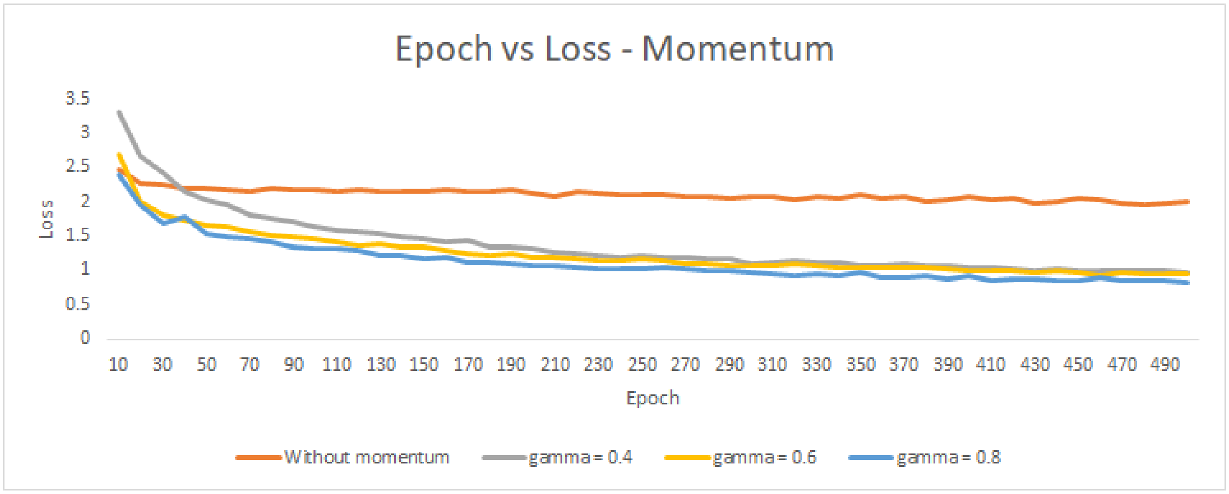


Fig 29. Loss over Epochs for different momentum values

Hidden layers

Number of layers

The number of hidden layers is dependent on how complex the data set is so the optimal number of layers is determined through trial and error.

Our experiments with 1,2,3 hidden layers are as suggested below:

1. Loss decreased quickly with 2 Hidden layers
2. Accuracy is similar with 1, 2 layers
3. 3 layers may be adding too much complexity to the problem

As shown on the graph, all three networks can reduce the loss to small values, but the network with 3 layers has the worst accuracy compared to other two networks. Therefore, we choose to use a neural network with 2 layers since it is more compatible with large dataset.

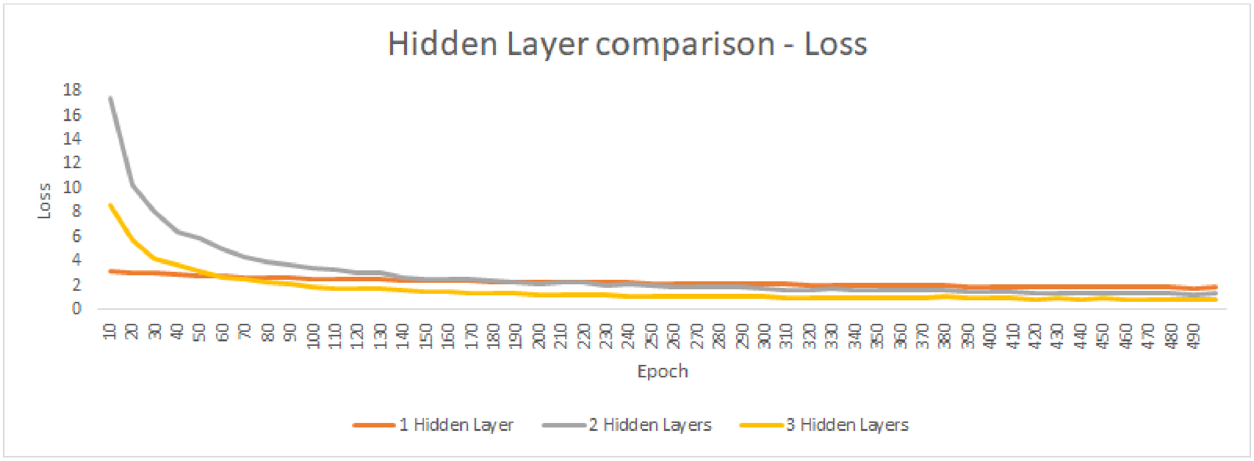


Fig 30. Loss over Epochs for different number of hidden layers

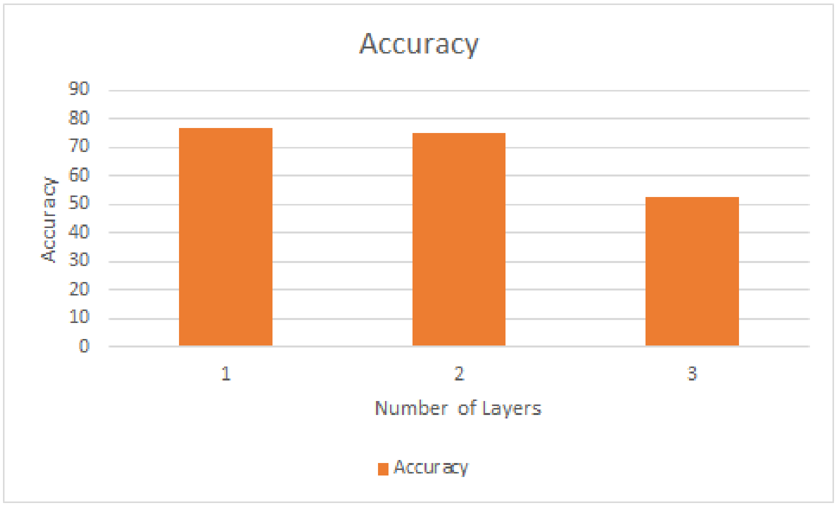


Fig 31. Overall accuracy of using different number of layers

Activation Functions

Tanh and sigmoid activation functions were not used as they result in the vanishing gradient problem due to their very shallow gradients at the extremes. ReLUs on the other hand do not suffer from this issue. They are also easy to compute and converge much faster. A downside is that sparsely activated due to the property of dead neurons since they output zero for all negative inputs. Leaky ReLU was included for comparison to accommodate this.

1 layer

Similar accuracies are achieved when using 1 hidden layer

Activation	**Accuracy (%)**

ReLU	75.44999999999999
Leaky ReLU	75.2

2 layer

We tried a combination of ReLu and Leaky ReLu for 2 hidden layers and found the below.

Activation on first layer	**Activation on second layer**	**Accuracy (%)**
ReLU	LReLU	75.33
ReLU	ReLU	74.78
LReLU	ReLU	76.13
LReLU	LReLU	74.95

We realised the below.

1. It is better to use a combination of ReLu and LReLu as it has a slight improvement over the regular ReLUs which coincides with their ability to prevent dead neurons.
2. Using only ReLU causes dead units and the network cannot learn anymore.

3 layer

We tried a combination of ReLu and Leaky ReLu for 2 hidden layers and found the below.

Activation on first layer	**Activation on second layer**	**Activation on third layer**	**Accuracy(%)**
LReLU	ReLU	ReLU	71.58333333
LReLU	LReLU	ReLU	72.33333333
ReLU	ReLU	LReLU	73.31666666

We realised the below.

1. There is not much difference in using different orders of LReLU and ReLU, the accuracy varies around 2 percent.

Size of layers

Size of 2 hidden layers	**Accuracy(%)**
10+7	14.38333333333
100+70	66.03333333333
200+170	75.66666666667
300+270	77.63333333333

- As seen on the tables, the neural network can barely learn anything while the size of the hidden layer is too small.
- From 100-200, the increasing size results in significantly improved accuracy.
- From 200-300, the increase is much less and more time consuming. Thus the size of hidden layers should be around 200.

Size of 3 hidden layers	**Accuracy(%)**
10+7+5	18.16666666
100+70+50	64.21666666
240+220+200	76.63333333
340+320+300	82.38333333

As shown on the graph, compared to using 2 hidden layers, 3 hidden layers have improvement on accuracy when the size of hidden layers is large, less size has no difference between 2 hidden layers. Therefore, for simplicity and less computational cost, 2 layers were used.

Batch Size

The batch sizes used have 32, 64, 128, 256, 512 and 1048 examples respectively. After computing on these sizes, the following relationships were found:

- Increase in batch size after a point corresponds to decreased accuracy
- Linear correlation between batch size and accuracy
- Increase in batch size decreases computation time by a significant factor/

It aims to minimise the amount of computations required and hence decrease execution time which is useful for large data sets. Less updates are required because there is only one update per mini-batch. Having less batches and hence more examples per batch, this decreases how accurate the estimate will be because of the overgeneralization of the data set.

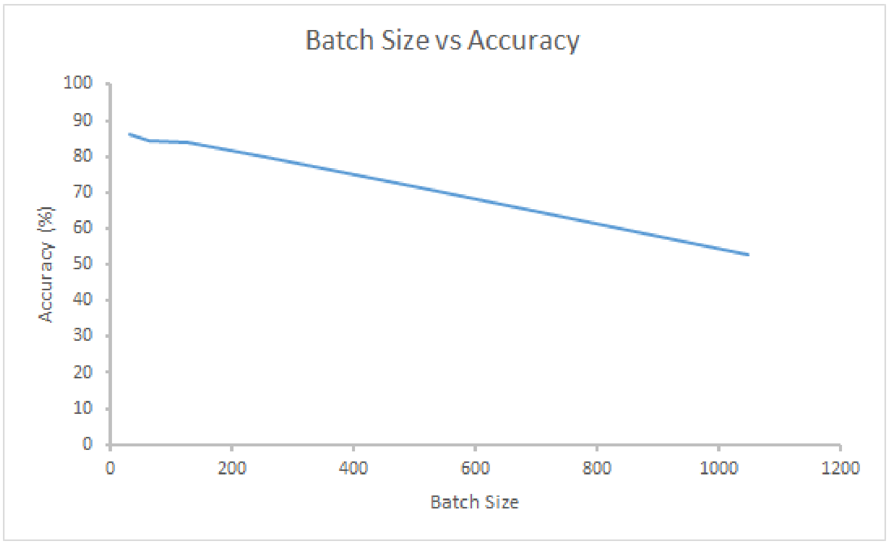


Fig 32. Batch sizes affecting Accuracy

It can be shown in the above figure, that there is a linear relationship between the batch size and the accuracy which shows the large impact this parameter has on the model's learning process.

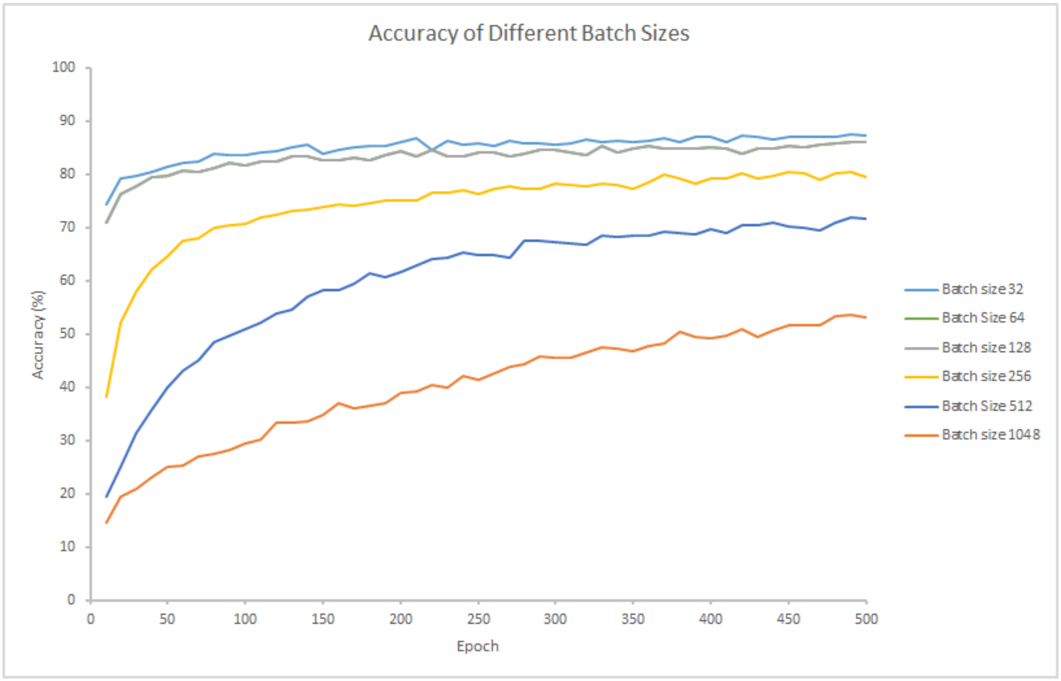


Fig 33. Batch size accuracy change over epochs

It is clear that an increase in batch size causes a decrease in accuracy. This is because batch sizes are used as estimations on partitions of the dataset. If the batch sizes are too big, there are more examples being averaged and hence lost. Additionally, there are less updates with large batch sizes. There is however, a trade off between the accuracy and speed of the learning process.

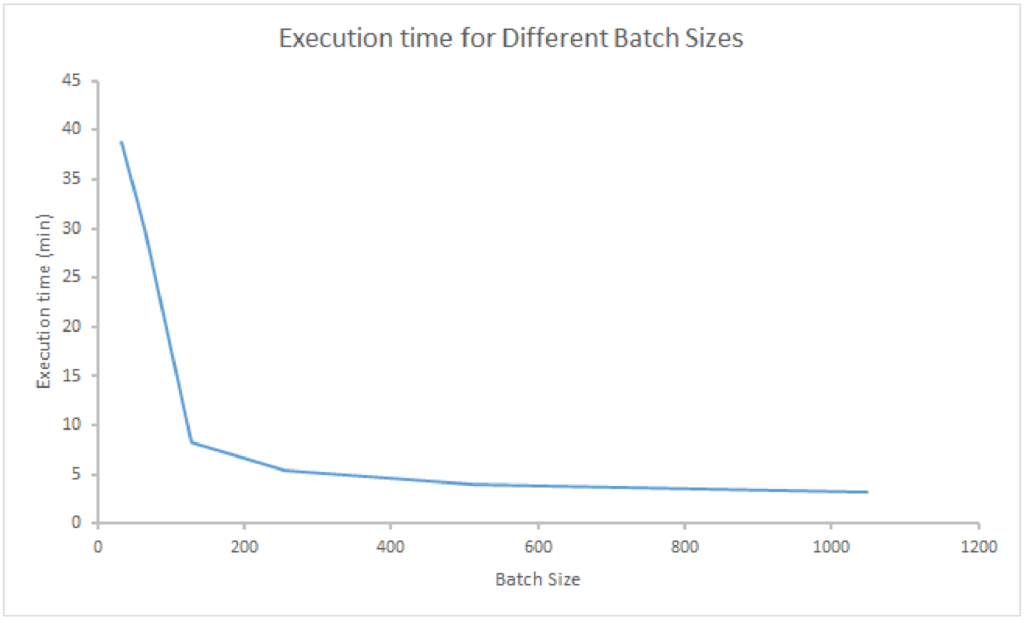


Fig 34. Batch size and their execution time

There is a large decrease in execution time as the batch size increases. This is because less updates are required. So overall, the smaller the batch sizes, the higher the accuracy. The downside of having these high accuracies is the high computational power required.

Dropout

We have tried with drop out rates of 0, 0.1, 0.2.
After computing on these rates, the following relationships were found:

- Current network are not suitable for using dropout
- When dropout rate set to 0, it achieves the best accuracy and loss
- Increasing dropout rate hurt the performance of the models.

The effect of different dropout rates on **accuracy** is shown below.

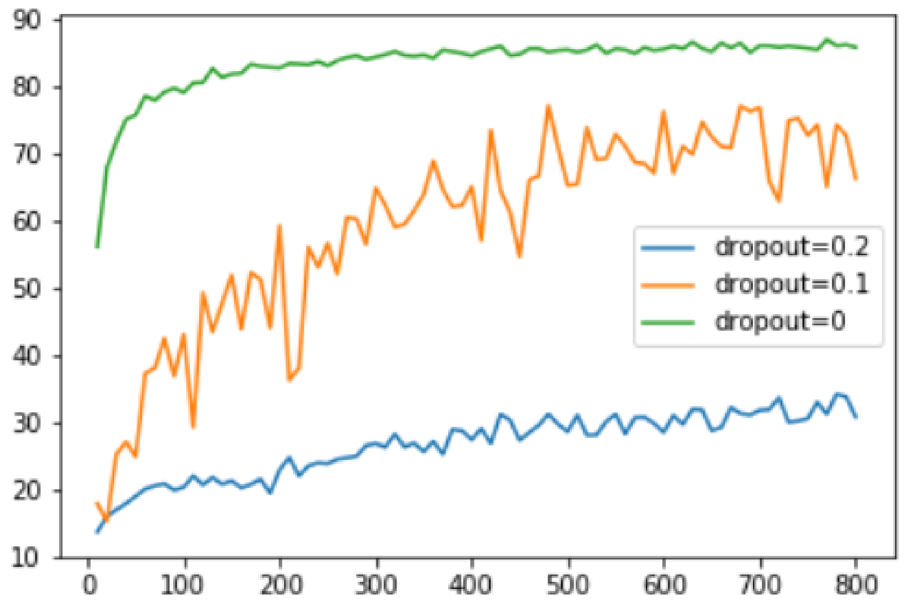


Fig 35. Accuracy for different drop out values

The effect of different dropout rate on **loss** is shown below.

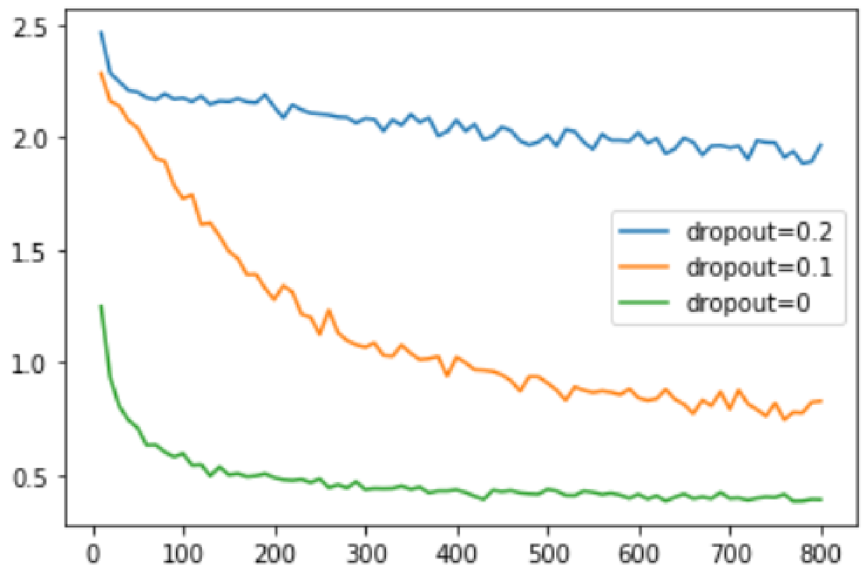


Fig 36. Loss for different drop out values

As shown on the graph, dropout doesn't perform any better in our experiment, actually, it hurts the performance. When dropout rate is set to 0, the accuracy is high and smooth, while dropout rate set to 0.1 gives an unstable accuracy over epoches. Higher dropout rate like 0.2 makes it worse.
As a dropout regularization technique in deep learning, it is often used in large networks consisting of millions of neurons. The neural network we use only contains around 600 neurons, the network size is already small. Using dropout makes it even smaller that it cannot properly function. Thus, in our network, setting the dropout rate to a very small value i.e. like 0.05 would be suitable.

We have also noticed that introducing drop out in most of the layers (or all hidden layers) is making the network inactive.

Batch Normalization

We have experimented by applying Batch Normalization at layer 1 and realised as below.

1. Loss and accuracy after initial epochs remain the same.
2. Probably batch normalization does not have much role in our scenario as we have already normalized the data before training.

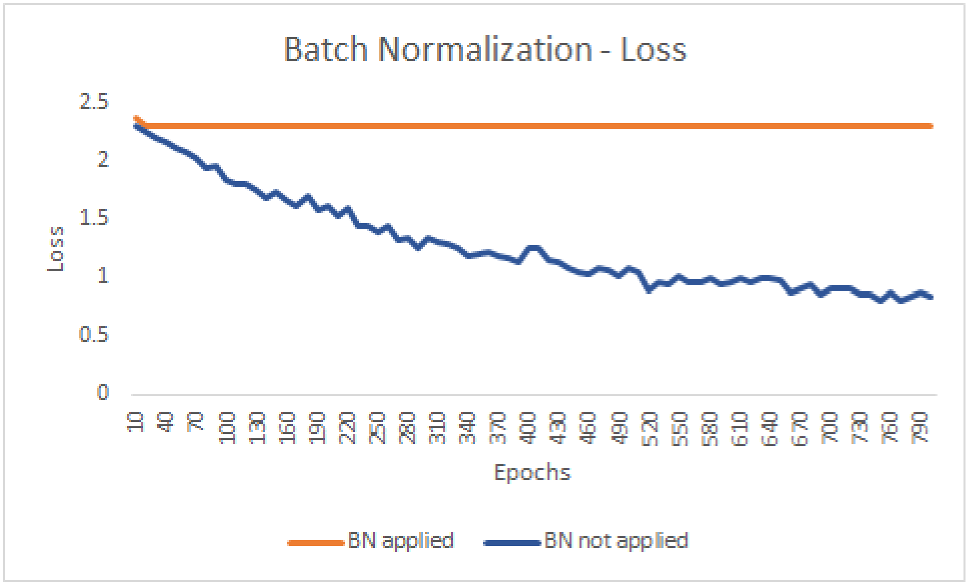


Fig 37. Loss over Epochs for a Batch Normalised model vs Non-batch normalised model

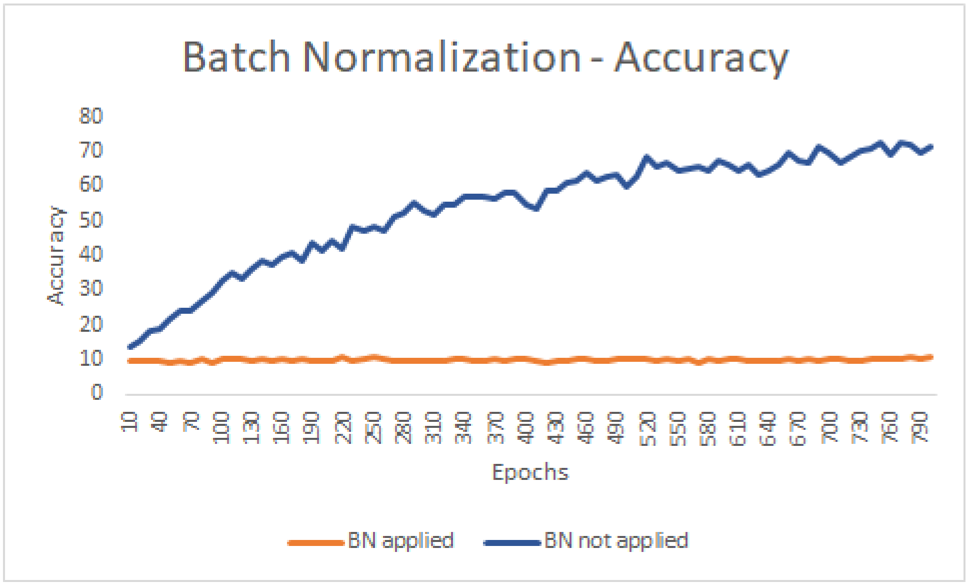


Fig 38. Accuracy over Epochs for a Batch Normalised model vs Non-batch normalised model

Discussion

Upon analysis of the results in the experimentation section, it is clear that changing certain modules does indeed have an influence on the results. These all relate to their formulas and the theory behind the modules, whether it's for optimisation or regularisation purposes.

We agree on the below things based on our experiments.

1. Normalizing the data before training is a really good approach.
2. More epochs increase the accuracy but it should be justified with a reasonable computation time
3. We should try a couple of learning rates to understand how the loss and accuracy are affected and then choose one which leads to a reasonable increase of accuracy.
4. Weight decay plays an important role if your neural network has large weights. We should also exercise caution by testing it with the appropriate learning rate.
5. Using momentum based SGD helps the neural network to learn faster and reach local minima faster and also saves computation and time.
6. We should not choose a high value for momentum term gamma as it might not converge to local minima.
7. Adding hidden layers and neurons in it might slightly add time for performing calculations.
8. Adding too many hidden layers might over complicate the network.
9. Activation functions are essential to introduce non linearity in data.
10. ReLu causes dead neurons and hence your network might stop to learn after a while. Every activation function has its upsides and downsides and hence the right combination has to be chosen for the problem at hand.
11. Overloading a neural network with all optimization and regularization techniques is not a good idea. The answer is often a simple network.

12. Mini batch SGD saves a lot of time as the network does not perform back propagation for every data point, but this also means the network might not converge to local minima, hence our batch size should not be too large that it might miss local minima and not too small that it takes a lot of time performing back propagations.
13. Drop out introduces regularization into the neural network by forcing other neurons to be independent. Inverted dropout is easier to implement and maintain.
14. High drop out value is not a good idea as all it might make most of the network inactive.
15. Introducing drop out in most or all hidden layers will also make most of the network inactive.
16. Batch Normalization does not always guarantee good results and hence it has to be chosen wisely.

Through the analysis of each module in detail, it is discovered how they contribute significantly to improvements of the model. However, applying these methods to this dataset does not mean it is suitable for all datasets. For example, softmax will not work if examples belong to multiple classes since it is assumed that each record has a probability of belonging to the one class.

Conclusion

Deep learning has various algorithms in place to regulate the optimization and regularisation of perceptrons. It can be shown how these procedures not only help to improve the efficiency of the model but also the accuracy. It is vital to understand the key concepts behind using these techniques to maximise the accuracy of the model being used. This will also help understand what methods to use for future models to better guide decisions of utilising certain modules and tuning specific parameters.

Final neural network architecture

We find the below neural network architecture reasonable for the problem data set.

Data Normalisation
Layers:

Layer 1 - 128 neurons (i.e. features of the dataset)
Layer 2 - 240 neurons - LReLu activation - 0.05 drop out rate
Layer 3 - 220 neurons - ReLu activation
Layer 4 (output) - 10 neurons (i.e. unique labels in data set) - softmax activation

Loss function - cross entropy
Optimization - momentum based mini batch SGD
Batch Size - 128
Gamma (momentum_term) - 0.9

Regularisation - L2 regularisation

Lamda (weight_decay_const) - 0.0001

Other Hyperparameters

Epochs - 500

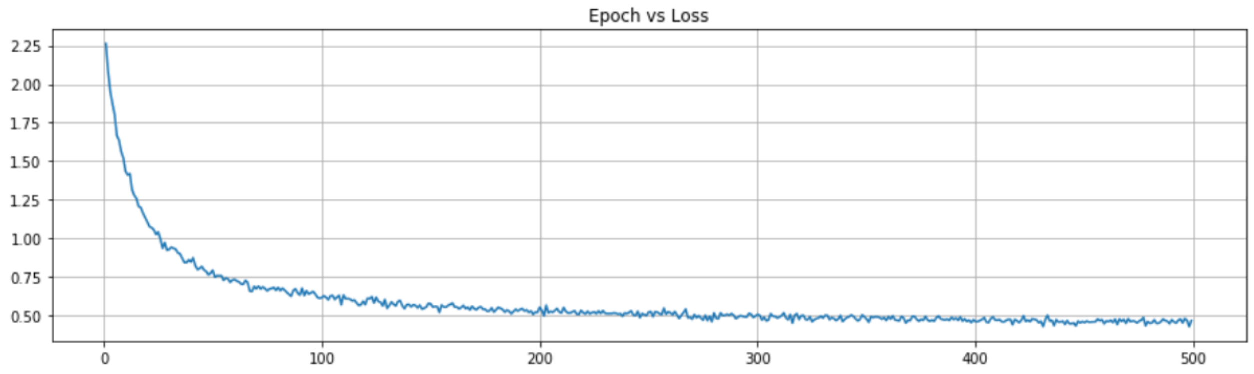
Learning Rate - 0.5

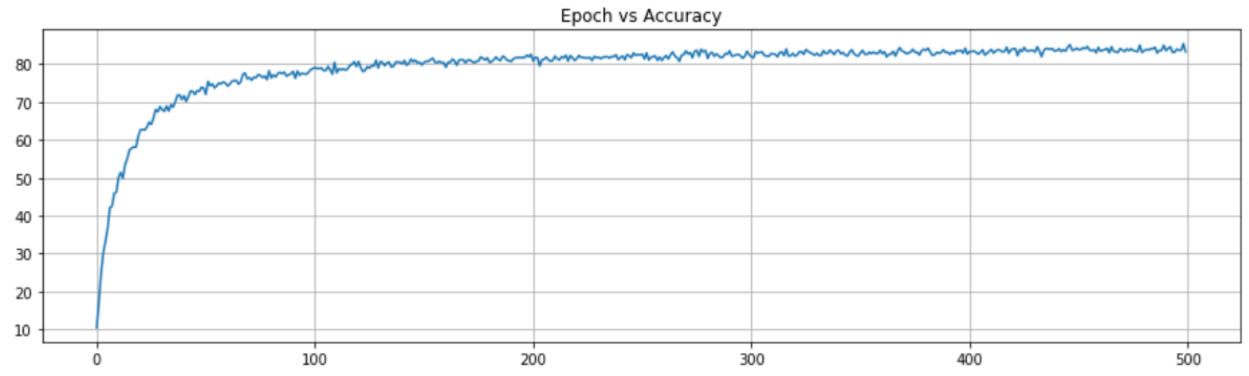
Results:

Final loss (on the last batch of data): 0.464

Final accuracy (on the last batch of data): 83.29%

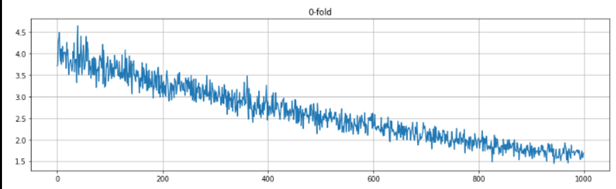
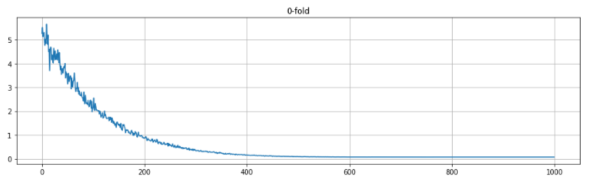
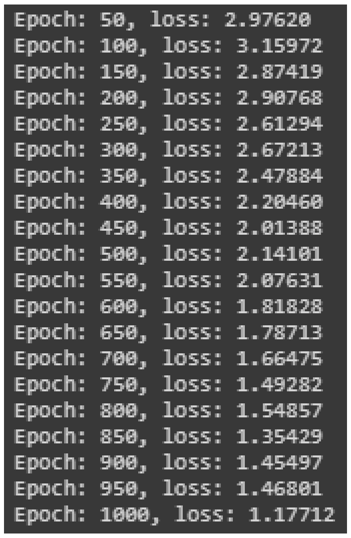
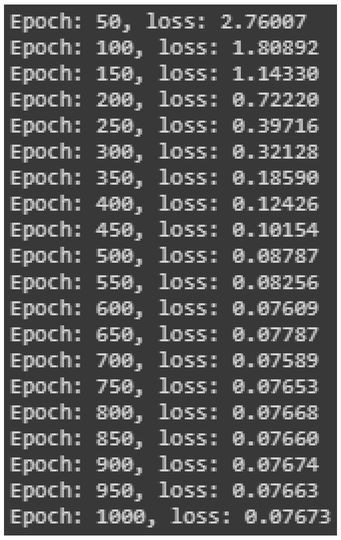
Time taken (on the entire data): 06 minutes 37 seconds





Appendix

Loss before and after momentum

Before Momentum	After Momentum
	
	

```
Epoch: 50, loss: 2.97620
Epoch: 100, loss: 3.15972
Epoch: 150, loss: 2.87419
Epoch: 200, loss: 2.90768
Epoch: 250, loss: 2.61294
Epoch: 300, loss: 2.67213
Epoch: 350, loss: 2.47884
Epoch: 400, loss: 2.20460
Epoch: 450, loss: 2.01388
Epoch: 500, loss: 2.14101
Epoch: 550, loss: 2.07631
Epoch: 600, loss: 1.81828
Epoch: 650, loss: 1.78713
Epoch: 700, loss: 1.66475
Epoch: 750, loss: 1.49282
Epoch: 800, loss: 1.54857
Epoch: 850, loss: 1.35429
Epoch: 900, loss: 1.45497
Epoch: 950, loss: 1.46801
Epoch: 1000, loss: 1.17712
```

```
Epoch: 50, loss: 2.76007
Epoch: 100, loss: 1.80892
Epoch: 150, loss: 1.14330
Epoch: 200, loss: 0.72220
Epoch: 250, loss: 0.39716
Epoch: 300, loss: 0.32128
Epoch: 350, loss: 0.18590
Epoch: 400, loss: 0.12426
Epoch: 450, loss: 0.10154
Epoch: 500, loss: 0.08787
Epoch: 550, loss: 0.08256
Epoch: 600, loss: 0.07609
Epoch: 650, loss: 0.07787
Epoch: 700, loss: 0.07589
Epoch: 750, loss: 0.07653
Epoch: 800, loss: 0.07668
Epoch: 850, loss: 0.07660
Epoch: 900, loss: 0.07674
Epoch: 950, loss: 0.07663
Epoch: 1000, loss: 0.07673
```

ReLU

Accuracy with only ReLUs

Number of hidden layers	**Accuracy (%)**
1 (ReLU)	76.85
2 (ReLU, ReLU)	74.78333333333333
3 (ReLU, ReLU, ReLU)	52.25