EE 550 - ARTIFICIAL NEURAL NETWORKS

Project 3 - Implementation of Multilayer Perceptron (MLP) Algorithm

Ahmet Pala
2020802018

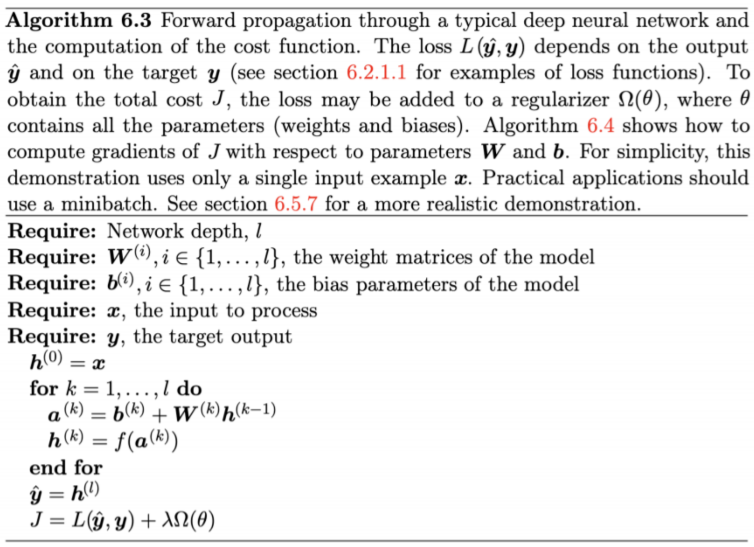
Boğaziçi University

20.05.2021

1 INTRODUCTION

Within the scope of this project, 3 different multilayer perceptron models are created, trained with train datasets and tested on the test datasets. For each model, different learning rates and initialization techniques for weight and bias parameters are tried. For each model, different threshold values for the loss function are applied for stopping criteria. Different numbers of minimum and maximum epoches are also tried to get rid of some local minima points. All the Python code for this project can be found here.

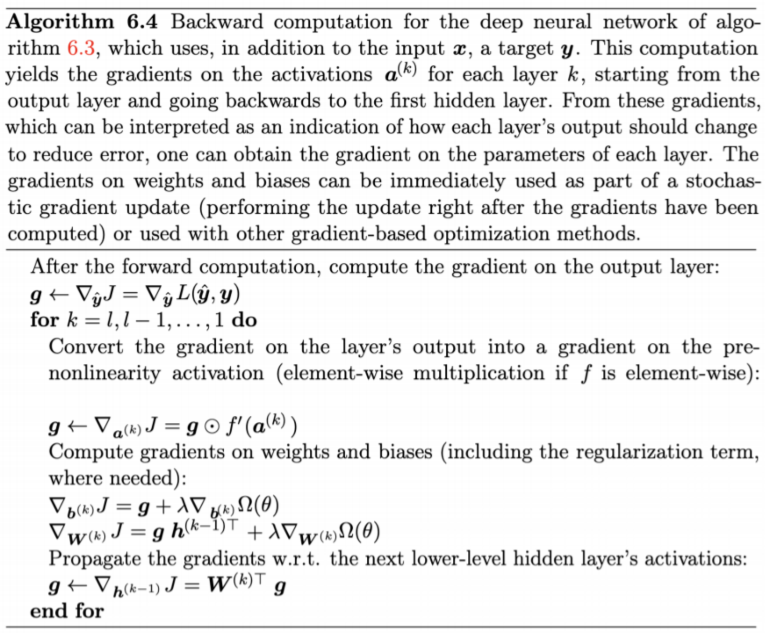
For the network architecture, forward and backward propagation rules are applied as stated by Goodfellow et. al. [1]. The forward and back-propagation algorithms used in this project are given in the Figure 1 and Figure 2, respectively. For each part of the project, Stochastic Gradient Descent (SGD) Algorithm is applied.



Algorithm 6.3 Forward propagation through a typical deep neural network and the computation of the cost function. The loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ depends on the output $\hat{\boldsymbol{y}}$ and on the target $\boldsymbol{y}$ (see section 6.2.1.1 for examples of loss functions). To obtain the total cost J, the loss may be added to a regularizer $\Omega(\theta)$, where θ contains all the parameters (weights and biases). Algorithm 6.4 shows how to compute gradients of J with respect to parameters $\boldsymbol{W}$ and $\boldsymbol{b}$. For simplicity, this demonstration uses only a single input example $\boldsymbol{x}$. Practical applications should use a minibatch. See section 6.5.7 for a more realistic demonstration.

```
Require: Network depth, $l$
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output
  $\boldsymbol{h}^{(0)} = \boldsymbol{x}$
  for $k = 1, \ldots, l$ do
    $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)}\boldsymbol{h}^{(k-1)}$
    $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
  end for
  $\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
  $J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda\Omega(\theta)$
```

Figure 1: Forward propagation algorithm [1]



Algorithm 6.4 Backward computation for the deep neural network of algorithm 6.3, which uses, in addition to the input $\boldsymbol{x}$, a target $\boldsymbol{y}$. This computation yields the gradients on the activations $\boldsymbol{a}^{(k)}$ for each layer k, starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer's output should change to reduce error, one can obtain the gradient on the parameters of each layer. The gradients on weights and biases can be immediately used as part of a stochastic gradient update (performing the update right after the gradients have been computed) or used with other gradient-based optimization methods.

```
  After the forward computation, compute the gradient on the output layer:
  $\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
  for $k = l, l-1, \ldots, 1$ do
    Convert the gradient on the layer's output into a gradient on the pre-
    nonlinearity activation (element-wise multiplication if $f$ is element-wise):

    $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
    Compute gradients on weights and biases (including the regularization term,
    where needed):
    $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$
    $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$
    Propagate the gradients w.r.t. the next lower-level hidden layer's activations:
    $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top}\, \boldsymbol{g}$
  end for
```

Figure 2: Back-propagation algorithm [1]

2 BINARY XOR MODEL

For this part of the project, a multilayer perceptron model consisting of 3 layers; 1 input, 1 hidden and 1 output layer is created. The sigmoid activation function is used for both hidden and output layers. The main aim of this model is to predict XOR function outputs. Note that the outputs of the XOR function are not linearly seperable as it can be seen from the Figure 3.

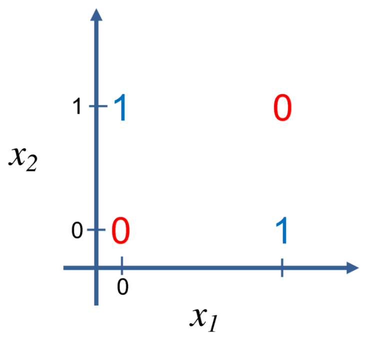


Figure 3: Visual representation of XOR outputs [2]

The visual representation of the network architecture can be seen from the Figure 4

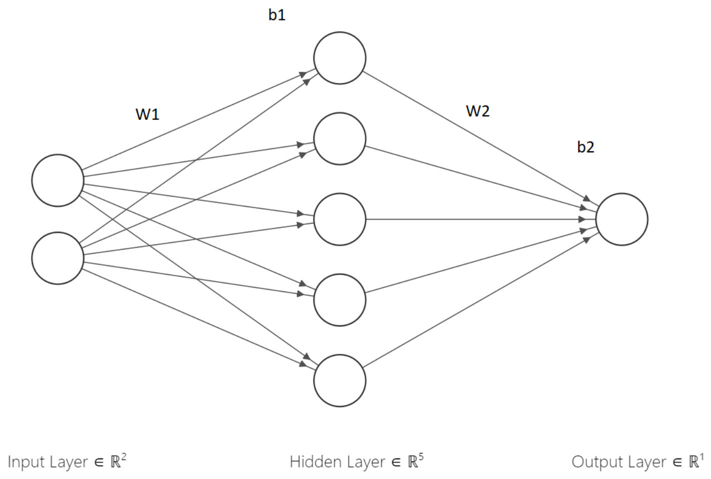


Figure 4: Network architecture for the binary XOR problem

2.1 Generating Dataset & Network Structure

For training the model, 4 sample patterns are produced and fed to the XOR function. Finally, the input and ground truth target values are obtained.

$$inputs = \begin{pmatrix} 0,0 \\ 0,1 \\ 1,0 \\ 1,1 \end{pmatrix} \qquad targets = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

These generated 4 sample patterns are fed to the multilayer perceptron model consisting of 3 layers; 1 input, 1 hidden and 1 output layer. As it can be seen from the generated patterns, the dimensions of inputs are 2 and the output is 1. Therefore, the input layer has 2 neurons while the output layer has 1. The number of hidden neurons in the hidden layer is accepted as 5. However, the Python code is arranged to adapt to each number of neurons in the hidden layer. Therefore, different numbers of hidden neurons can be tried using the Python code given in Section 1.

As discussed before, the outputs of the XOR function are not linearly seperable. Therefore, nonlinear activation functions should be used in at least 1 layer to overcome this issue. For this reason, sigmoid activation function is used both for hidden and output layer. Note that the output layer gives the number between 0 and 1. Therefore, the final output value should be rounded so that 0 or 1 values can be obtained which are the exact outputs of the XOR function. Finally, mean squared error (MSE) is used for the loss function.

2.2 Training the MLP Model

After defining the necessary functions in Python, the MLP model is trained. All the weights and biases are initialized randomly from the interval (0,+1). For the learning rate to use, 0.09 is decided after some trials. The maximum number of epoch is decided as 20000. Moreover, the training loop is designed to stop when the convergence is reached. For the convergence criterion, the model calculates the change in the loss function. If the loss function does not change at least 0.01 %, the algorithm stops and assume that the convergence is achieved. Also note that the minimum number of epoch is set as 2500 in order to get rid of local minimum points and some underfitting issues. After the completition of training phase, the loss function vs each epoch is plot as it can be seen from the Figure 5.

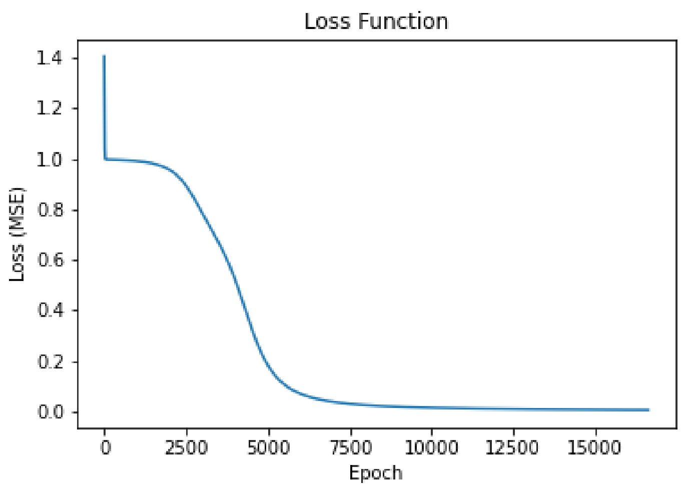


Figure 5: Loss function change for each epoch

The training algorithm is ended automatically after 16616 epoches since the convergence criteron for the threshold value is reached. As it can be seen from the figure, the loss function drops drastically first then smoothly decrease until the convergence.

2.3 Testing the Model

After the training phase is completed, the final parameters (weights and biases) are obtained. These final parameter set is used for estimating the original pattern results. The generated 4 sample patterns given in Section 2.2 is fed to the model containing the final parameter sets. The estimations for these sample patterns are given in the Table 1.

x_1	x_2	**Ground Truth**	**Prediction**	**Rounded**
0	0	0	0.03531275	0
0	1	1	0.96057946	1
1	0	1	0.96942986	1
1	1	0	0.03719592	0

Table 1: Comparison of MLP results and ground truth labels

As it can be seen from the comparison table, all the samples are classified correctly. The original predictions are very close to their ground truth values. After rounding the predictions, all the samples are predicted correctly.

3 FUNCTION APPROXIMATION

For this part of the project, a multilayer perceptron model consisting of 4 layers; 1 input, 2 hidden and 1 output layer is created for approximation to a nonlinear function given in the Equation 1. There is 1 neuron in both input and output layers. For the hidden layers, there are 4 and 5 hidden neurons, respectively. The Python code is arranged to adapt to each number of neurons in the hidden layers. Therefore, different numbers of hidden neurons can be tried using the Python code given in Section 1. The *tanh(x)* activation function is used for hidden layers. For the output layer, the linear activation function is preferred. Finally, the mean squared error is used for the loss function.

$$f(x) = sin(x) + 2cos(x) \tag{1}$$

where x is given in radians. A 200 dimensional dataset for x is generated between 0 an 2π. Each generated data point is fed to the Equation 1 and the ground truth values are obtained. The visual representation of the generated data points is given in the Figure 6.

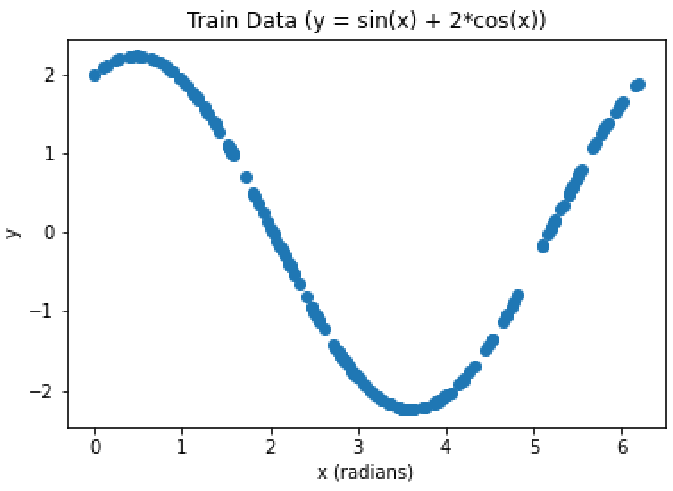


Figure 6: Generated data points

The visual representation of the network architecture can be seen from the Figure 7

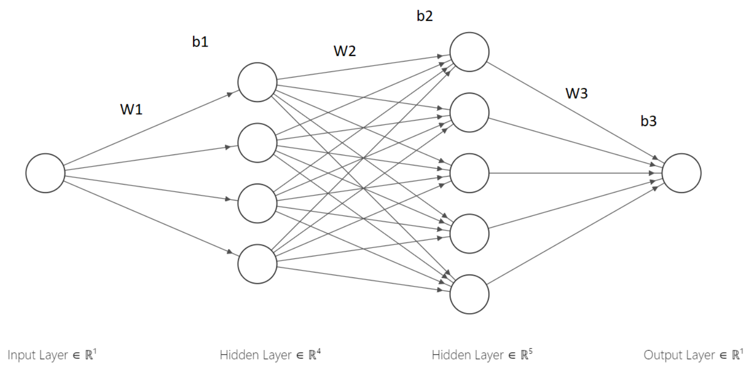


Figure 7: Network architecture for the function approximation problem

3.1 Training the MLP Model

All the weights and biases are initialized randomly from the interval (0,+1). During the training phase, the learning rate to use is decided as 0.01 after some trials. The maximum number of epoch is decided as 2000. Moreover, the training loop is designed to stop when the convergence is reached. For the convergence criterion, the model calculates the change in the loss function. If the (current) loss function does not change at least 0.5 %, the algorithm stops and assume that the convergence is achieved. Also note that the minimum number of epoch is set as 200 in order to get rid of local minimum points and some underfitting issues. After the completition of training phase, the loss function vs each epoch is plot as it can be seen from the Figure 8.

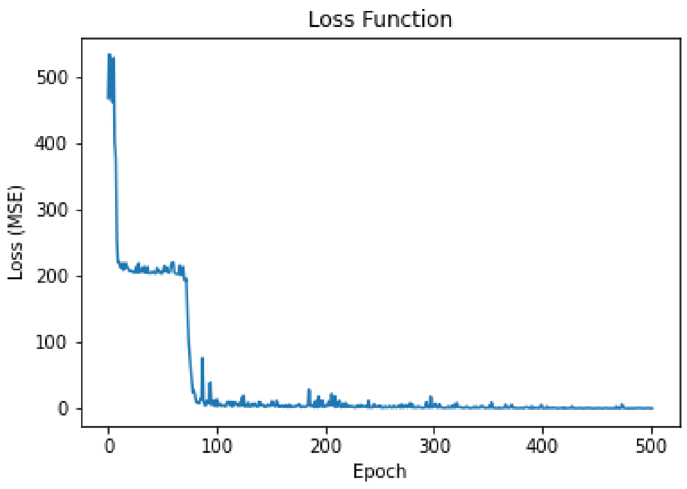


Figure 8: Loss function change for each epoch

The training algorithm is ended automatically after 502 epoches since the convergence criteron for the threshold value is reached. As it can be seen from the figure, the loss function drops drastically first then converged to a local optima point which means that the loss function does not change obviously for a while. At this point, the minimum number of epoch hepls us to get rid of this issue. After training more the model, it converges to a better point.

In addition to the loss function change for each epoch, the estimations from the created MLP model with the original data points are given in the Figure 9

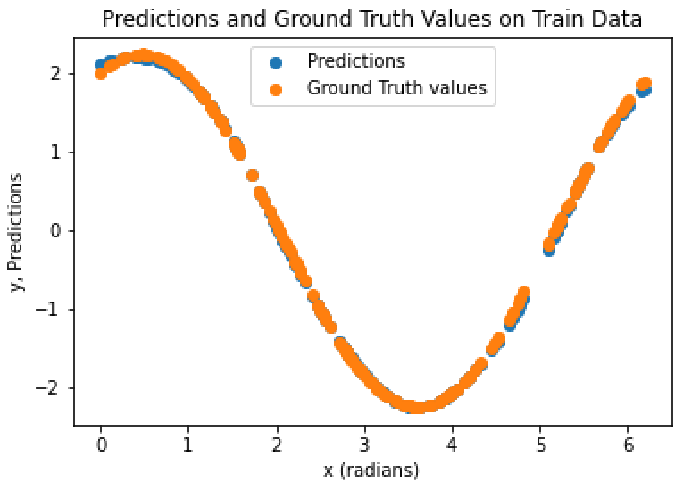


Figure 9: Predictions and ground truth values on the training data

3.2 Testing the MLP Model

After the training phase is completed, the final parameters (weights and biases) are obtained. These final parameter set is used for testing the model performance on other generated 25 test data points. These test data points and the predictions are given in the Table 2.

x	Ground Truth	Prediction
2.0531	-0.0416	-0.0670
2.4379	-0.8780	-0.8896
2.1696	-0.3013	-0.3270
5.9144	1.5050	1.4703
0.7210	2.1624	2.1154
3.8294	-2.1801	-2.1861
3.3032	-2.1349	-2.1326
2.8016	-1.5520	-1.5380
1.7878	0.5459	0.5357
0.6497	2.1975	2.1467
4.9916	-0.4100	-0.4975
3.8278	-2.1809	-2.1870
5.6817	1.0831	1.0820
1.7046	0.7243	0.7210
1.0506	1.8618	1.8529
5.5886	0.8967	0.8976
5.4764	0.6615	0.6552
4.2608	-1.7725	-1.7795
5.6149	0.9502	0.9513
3.1959	-2.0513	-2.0426
1.6153	0.9100	0.9138
2.3705	-0.7374	-0.7539
1.5066	1.1263	1.1368
3.7503	-2.2126	-2.2210
4.3768	-1.6028	-1.6230

Table 2: Test data points and predictions from the MLP model

3.3 Visualization of Predictions on Test Data

In this part, the visual representation of the generated test data points and the predictions from the trained MLP model is given in the Figure 10

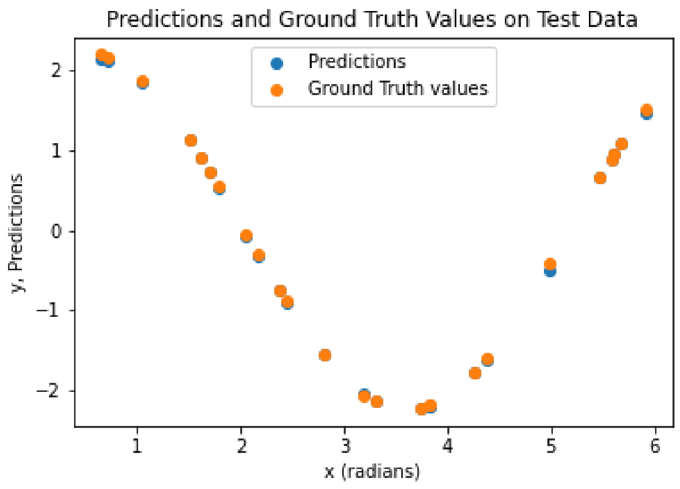


Figure 10: Predictions and ground truth values on the test data

4 RECOGNITION OF HANDWRITTEN DIGITS

For this part of the project, a multilayer perceptron model is created for the recognition of handwritten digits. At the first part, the Optical Recognition of Handwritten Digits data set (tes) is downloaded from here. This dataset has 1797 samples and each sample has 64 attributes which generate an input matrix of 8x8 where each element is an integer in the range from 0 to 16. As an illustration, example patterns from each class is plot as it can be seen from the Figure 11.

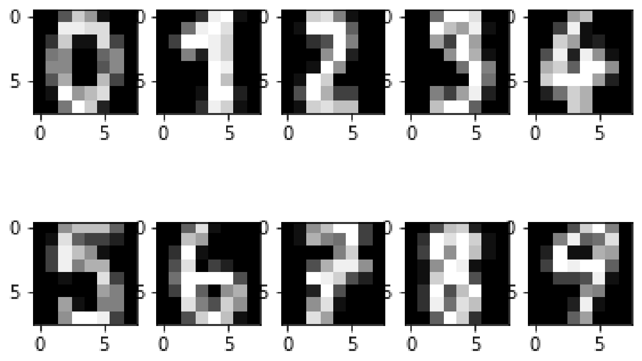


Figure 11: Example patterns from each class

For creating the training and test datasets, stratified sampling is applied. Firstly, 100 samples from each class are choosen randomly for the training dataset which finally consists of 1000 samples. Then, 20 samples from each class are choosen randomly for the test dataset which finally consists of 200 samples. The training dataset is used for train the MLP model, and the test dataset is used for measuring the created model performance. Since the input dataset vary from 0 to 16, it is normalized by dividing each element by 16 in order to make the network learn the optimal parameters more quickly [3].

Since the inputs are 64-dimensional, the input layer has 64 neurons. The model is created with 2 hidden layers having 32 and 24 hidden neurons, respectively. For the output layer, is is assumed that there are 10 neurons, one for each class since there are 10 different classes. The Python code is arranged to adapt to each number of neurons in the hidden layers. Therefore, different numbers of hidden neurons can be tried using the Python code given in Section 1. For the hidden layers, *tanh(x)* activation function is preferred. for the output layer, there is *softmax* function in order to convert each estimation to the probability for the corresponding class in each layer. Therefore, the sum of predictions in each output neuron for a data point is equal to 1.

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{2}$$

Finally, Cross-entropy loss function given in the Equation 3 is used as the loss function.

$$\text{Cross-Entropy} = \sum_{i=1}^{m} -y_i * log(\hat{y}_i) \tag{3}$$

where y_i stands for the one-hot representation of the ground truth labels while $\hat{y}_i$ indicates the predictions and m is the mini-batch size (for this project, it is taken as 1 since the Stochastic Gradient Descent algorithm is performed). Note that both these terms are 10-dimensional vectors for each individual sample. For the final estimation of the corresponding data, the class having the maximum probability value is choosen.

4.1 Training the MLP Model

All the weights and biases are initialized randomly from the interval (-1,+1) since the input size is relatively high (64). If they are initialized from the interval (0,+1), the output of the *tanh(x)* function will be close to zero. Therefore, the initialization is performed using the interval (-1,+1). During the training phase, the learning rate to use is decided as 0.001 after some trials. The maximum number of epoch is decided as 200. Moreover, the training loop is designed to stop when the convergence is reached. For the convergence criterion, the model calculates the change in the loss function. If the (current) loss function does not change at least 1 %, the algorithm stops and assumes that the convergence is achieved. Also note that the minimum number of epoch is set as 75 in order to get rid of local minimum points and some underfitting issues. After the completition of training phase, the loss function vs each epoch is plot as it can be seen from the Figure 12.

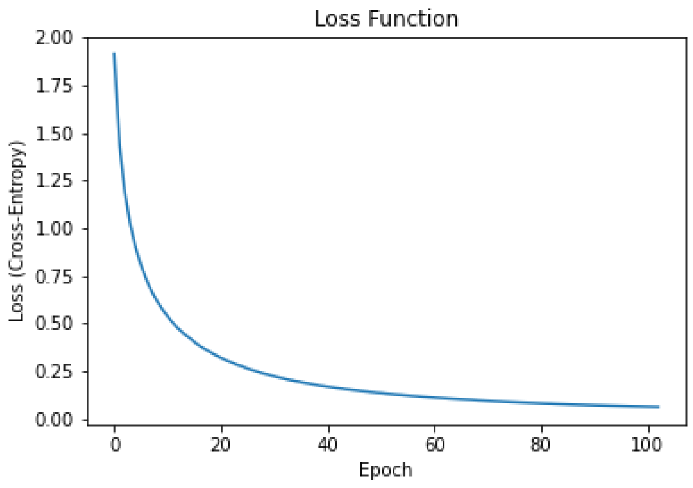


Figure 12: Loss function change for each epoch

The training algorithm is ended automatically after 103 epoches since the convergence criteron for the threshold value is reached. As it can be seen from the figure, the loss function decreases smoothly and converged. In addition to the loss function change for each epoc, the training accuracy is also calculated for each epoch during the training as it can be seen from the Figure 13. The final training accuracy is calculated as 0.995%.

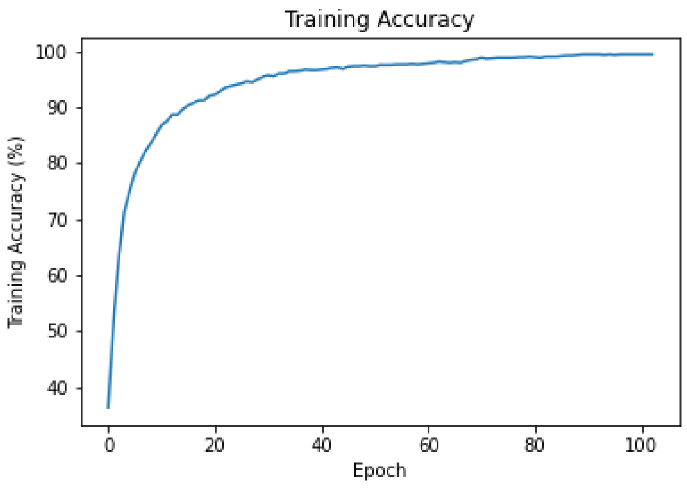
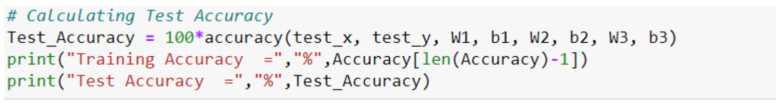


Figure 13: Training accuracy change for each epoch

4.2 Testing the MLP Model

After the training phase is completed, the final parameters (weights and biases) are obtained. These final parameter set is used for testing the model performance on other 200 test data points using the code given in the Figure 14.



```
# Calculating Test Accuracy
Test_Accuracy = 100*accuracy(test_x, test_y, W1, b1, W2, b2, W3, b3)
print("Training Accuracy  =","%",Accuracy[len(Accuracy)-1])
print("Test Accuracy  =","%",Test_Accuracy)
```

Figure 14: Training accuracy change for each epoch

Applying this code block, test accuracy is calculated as 94%.

4.3 Sample Inputs and Estimations

In this part, sample inputs from the test data are plotted for each class.

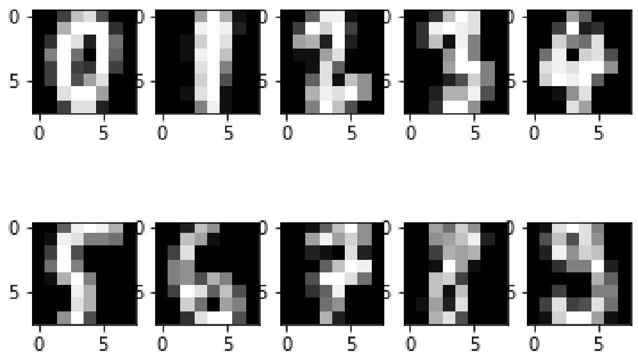


Figure 15: Samples for estimation

These sample inputs are fed to the trained network and the estimations are obtained as given in the Table 3.

Ground Truth	**Prediction**
0	0
1	1
2	3
3	3
4	4
5	5
6	6
7	7
8	8
9	9

Table 3: Predictions and ground truth labels for 10 sample patterns

As it can be seen from the table, all classes are estimated correctly except the class 2.

References

[1] Goodfellow, I., Y. Bengio and A. Courville, *Deep Learning*, MIT Press, 2016, `http://www.deeplearningbook.org`.

[2] Spears, B., “Contemporary machine learning: a guide for practitioners in the physical sciences”, , 2017.

[3] Ogasawara, E., L. C. Martinez, D. De Oliveira, G. Zimbrão, G. L. Pappa and M. Mattoso, “Adaptive normalization: A novel data normalization approach for non-stationary time series”, *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, IEEE, 2010.