Classification of MNIST image data using Backpropagation algorithm on Multi Layer Perceptron

1st Priyankha Bhalasubbramanian
Unity Id: pbhalas

2nd Shashank Shekhar
Unity Id: sshekha4

3rd Shraddha Dhyade
Unity Id: sddhyade

***Abstract*—The purpose of this project is to train a neural network using backpropagation to classify MNIST dataset images and achieve a competitive accuracy on the validation and test data. There are two phases involved. The first phase is to tune the hyperparameters and the second phase is to train the model and then use it to predict the class labels of images in the test dataset.**

I. Introduction

This project trains a neural network based on multi-layer (3 layer) perceptron model following the back propagation algorithm. The task is to classify the images provided in the MNIST dataset. The procedure used for the classification trains the model using one simple rule: The weight change is proportional to the difference of the target value and the predicted value by computing the activation of the node. In this process of training the neural network, we have used stochastic gradient descent. It offers several advantages like high throughput and faster convergence. The high throughput helps in utilizing a large number of cores in the GPU. Two hyperparameters are taken and tuned in the process:

1) Learning Rate (l)
2) Batch Size (b)

Learning Rate is taken from the set: $\{0.0001, 0.001, 0.01\}$
Batch Size is taken from the set: $\{128, 256, 384\}$

II. Network Structure

The Neural Network architecture that we used consists of 3 layers

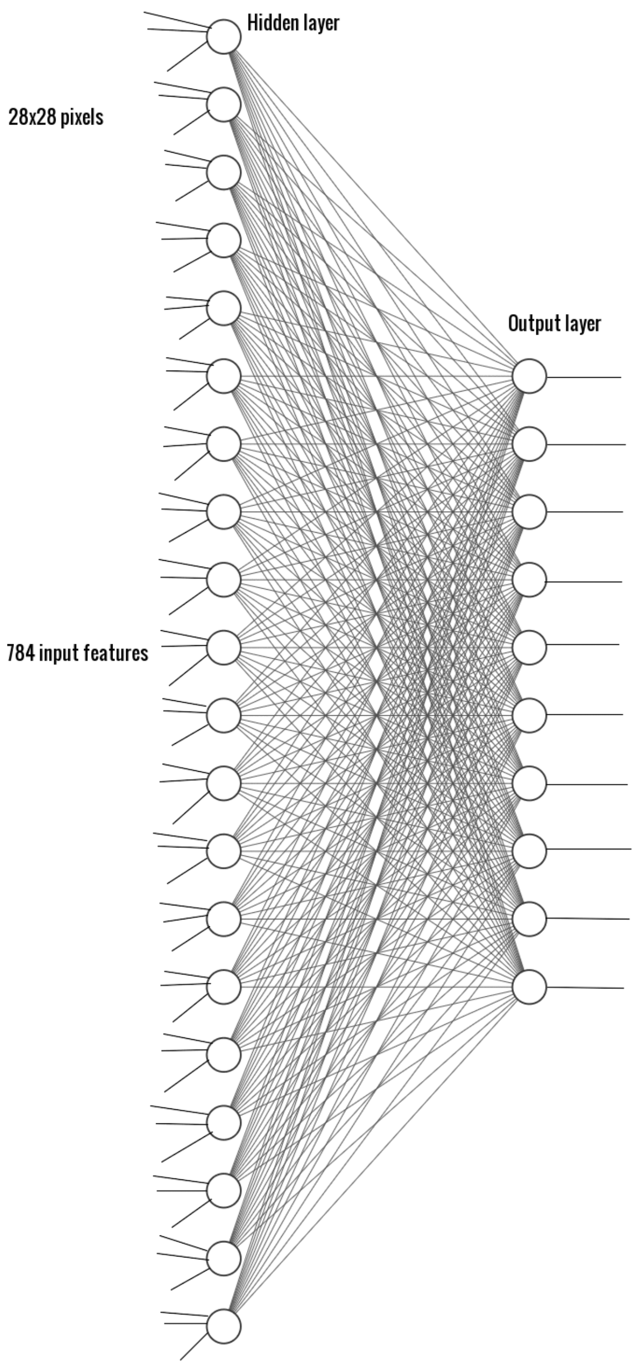
1) Input layer
2) Hidden layer
3) Output layer

The input layer consists of the set of input features. Each input is a 28x28 pixel image, so it has in total 784 pixels as input features. These are then fed to every neuron in the hidden layer. The hidden layer consists of 20 neurons. Each neuron applies a linear transformation of the input vector (in the hidden layer the input vector is of size 784) that it receives i.e., by doing a dot product (scalar product) with the weight vector and adds a bias associated with the neuron. Then the activation function (in this case sigmoid) is applied on the linear transformation to get the corresponding output of the neuron. The activation functions add non-linearity to the model. This process happens at every neuron in the hidden layer and the output activations of the hidden layer now become the input to the output layer. The output layer has 10 neurons (corresponding to the 10 class labels). And at each neuron we apply linear transformation of the input it receives from hidden layer using the weights and bias and finally apply the activation function (sigmoid) to get the output activations. Every output activation from the output layer corresponds to the probability that the input image belongs to the corresponding class label. Loss function like cross entropy loss (cross entropy loss is used here because the output is a categorical output) is used as a measure of variation in the expected and predicted class label. This is the forward phase. In the backward phase we compute the gradient of the loss with respect to bias and weights by finding the gradients with respect to previous layers and applying chain rule, in order to update the weights and biases such that the loss is minimized. This process is repeated for every input image in mini-batches- all of which corresponds to one epoch. The algorithm is repeated for as many epochs as decided initially until the model learns the best parameters (one that yields maximum validation/test accuracy)
The hyper-parameters learning rate and batch size was tuned such that the best validation accuracy is achieved. A grid search over values $\{0.0001, 0.001, 0.01\}$ for learning rate and $\{128, 256, 384\}$ over batch size was performed to get the best value of 0.001 and 128 respectively.

III. Algorithm

A. Forward Propagation

In this algorithm, the inputs are propagated through hidden layers of the feedforward multi layer perceptron network. At each layer, input to the layer is transformed by weights and biases and an activation function g (sigmoid in our case) is applied. Output from the last layer is final output. Then we compute cost using loss function, regularization function and weight. In our case, we compute the cost using cross entropy loss function.



Algorithm - Forward Propagation

```
$h^{(0)} = x$
**for** $k = 1, ..., K$ **do**
    $a^{(k)} = b^{(k)} + W^{(k)\top} \cdot h^{(k-1)}$
    $h^{(k)} = g(a^{(k)}$
**end for**
$\hat{y} = h^{(k)})$
**return** $L_y(\hat{y}) + \lambda \cdot \Omega(\theta)$
```

B. Backward Propagation

In this algorithm, we compute gradient of the Loss function at each layer with respect to biases and weights of the layer. We keep propagating backward starting from the last layer till the first hidden layer. This helps us to adjust the weights and biases such that the cost is minimized.

Algorithm - Backward Propagation

```
$Run\ Forward\ Propagation$
$G \leftarrow \nabla_{\hat{y}} L_y(\hat{y})$
**for** $k = K \rightarrow 1$ **do**
    $G \leftarrow \nabla_{a^{(k)}} L = G \bigodot g'(a^{(k)})$
    $\nabla_{b^{(k)}} L = G + \lambda \cdot \nabla_{b^{(k)}} \Omega(\theta)$
    $\nabla_{W^{(k)}} L = h^{(k-1)} \cdot G^\top + \lambda \cdot \nabla_{W^{(k)}} \Omega(\theta)$
    $G \leftarrow \nabla_{h^{(k-1)}} L = W^{(k)\top} \cdot G$
**end for**
```

IV. Derivation

Given: $L_y(\hat{y}) = \sum_k -(1-y_k)ln(1-\hat{y_k}) - y_k ln(\hat{y_k})$
where $y, \hat{y}, a \in \mathbb{R}^n$ and L represents the cross-entropy loss
To prove: $\nabla_a L_y(\hat{y}) = \hat{y} - y$

Proof: Taking partial derivative of L with respect to $\hat{y}$ we get

$$\frac{\partial L_y(\hat{y})}{\partial \hat{y_k}} = \frac{(1-y_k)}{1-\hat{y_k}} - \frac{y_k}{\hat{y_k}} = \frac{\hat{y_k} - y_k}{(1-\hat{y_k})(\hat{y_k})}$$

$\hat{y_k} = \sigma(a_k)$, where σ is the sigmoid function

$$\frac{d\hat{y_k}}{da_k} = \sigma'(a_k) = (\sigma(a_k))(1-\sigma(a_k)) = (1-\hat{y_k})(\hat{y_k})$$

So, $\frac{\partial L_y(\hat{y})}{\partial a_k} = \frac{\partial L_y(\hat{y})}{\partial \hat{y_k}} \frac{d\hat{y_k}}{da_k}$ (By applying Chain Rule)

Hence, $\frac{\partial L_y(\hat{y})}{\partial a_k} = \frac{\hat{y_k} - y_k}{(1-\hat{y_k})(\hat{y_k})}(1-\hat{y_k})(\hat{y_k}) = \hat{y_k} - y_k$

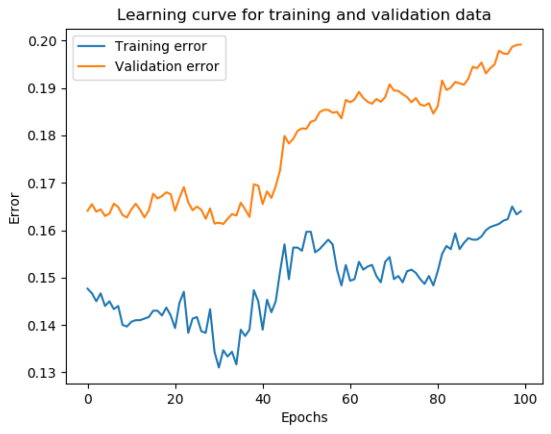
Therefore, $\nabla_a L_y(\hat{y}) = \hat{y} - y$

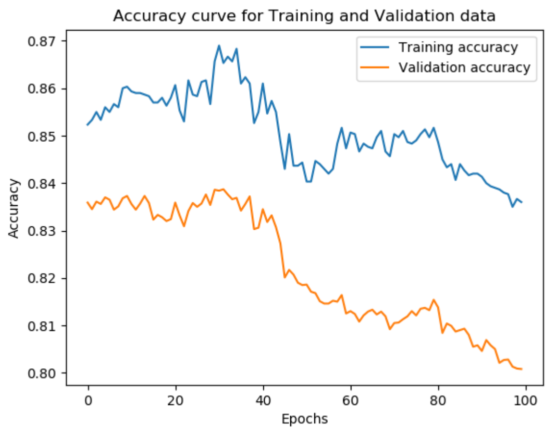
V. Methodology

MNIST dataset is used and we use 3000 images as training data, 10000 images as validation data and 10000 images for test data. The input for the train data received is a 784 pixel input and the output is a one hot encoded output corresponding to every class label (from 0 to 9). The first phase of project is to find the best hyper-parameters namely learning rate and batch size based on validation accuracy after training the model on the training data. Learning Rate is taken from the set: {0.0001, 0.001, 0.01}. Batch Size is taken from the set: {128, 256, 384}. A grid search was performed to find the model which gives the best validation accuracy by exhaustively trying out all combinations of learning rate and batch size. In the process we also save the parameters - weights and biases of the model that gives the best validation accuracy as a JSON file so that the same model weights can be utilised at the beginning of the actual training phase instead of giving random initial values of weights and biases. After running 100 epochs for every combination of learning rate and batch size, we recorded that the learn rate of 0.01 and batch size of 128 gives the best validation accuracy. Consequently the model parameters (weights and biases) that gave the highest validation accuracy was saved in the bestmodel.json file. In the second phase we combine the training and validation data as updated train data to train the model and use test data to benchmark the model by computing the accuracy and error. As a pre-processing step the labels of the validation data had to be one hot encoded to make it compatible with the train data. The best hyper-parameters obtained in the first phase- (learning rate of 0.001 and batch size 0f 128) was used. The model weights and biases are imported from the best model saved in bestmodel.json (from phase 1) instead of assigning arbitrary initial values for weights and biases. This way the model will converge to best accuracy in lesser number of epochs.

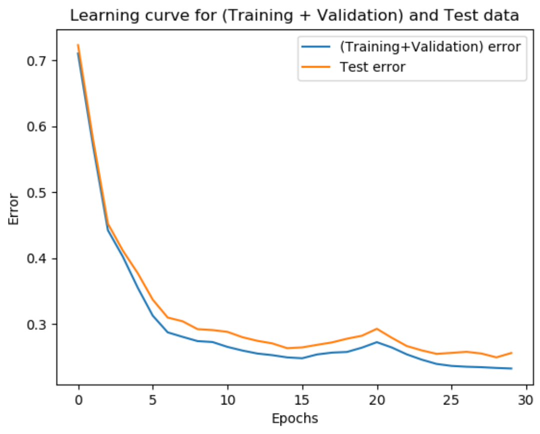
VI. Results

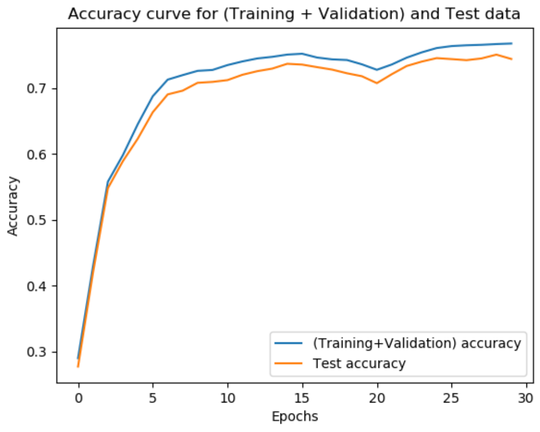
In the first phase we implemented a grid search over the learning rate values $\{0.0001, 0.001, 0.01\}$ and batch size values $\{128, 256, 384\}$, to get the best value of hyper parameters. For this purpose the validation accuracy was used as a measure to determine the best values of hyper-parameters. After applying all 9 combinations over 100 epochs each, we found that the hyper-parameters: learning rate, $\eta = 0.001$ and and batch size = 128 gave the best validation accuracy of around **84%**. Hence, the same values of hyper-parameters and parameters (weights and biases) was used in the initial model construction in the second phase. The plots for the learning curve and accuracy curve for train data and validation data is depicted as follows:





In the second phase, we combine the train and validation data as updated train data to train our model and used the standalone test data to benchmark the accuracy and loss. The graphs showing the learning curve and accuracy curve for the updated train and tested data is plotted. Finally we observe that the best test accuracy obtained using the hyper-parameters (learn rate $\eta = 0.001$) and batch size = 128 is **75.08%**





VII. Some Common Mistakes

1) The matrix products must be taken appropriately so that they are compatible.
2) During the second phase it is important to one-hot encode the output label of validation data before combining it to the train data.
3) The best model obtained in terms of validation accuracy has to be saved in a json file so that it can be loaded directly for training the model in the second phase.

Acknowledgment

We would like to express our deepest appreciation to all our colleagues who helped us in completing this report. We would like to give a special gratitude to our professor Dr. Edgar LobatonI who provided us with the sufficient background knowledge to complete this project and the report.

References

[1] https://www.cs.umd.edu/class/fall2012/cmsc828d/reportfiles/doan3.pdf
[2] https://page.mi.fu-berlin.de/rojas/neural/chapter/K7.pdf