**Multilayer Neural Network**

**Donggyu Kim**

**Introduction**

This report delves into the intricate processes of forward and back propagation, with a specific focus on architectures comprising two hidden layers.

Forward propagation is the process through which neural networks make predictions. Inputs are passed through the network's layers, each comprising nodes or neurons, which apply weights, biases, and activation functions to transform the data. The presence of two hidden layers allows the network to learn more abstract representations of the input data, thereby enhancing its ability to model complex problems.

Conversely, backpropagation is the essence of learning in neural networks. It is an algorithm that calculates the gradient of the loss function with respect to each weight by the chain rule, effectively propagating the error backward through the network. This allows for the iterative adjustment of weights and biases in a manner that minimizes the difference between the actual and predicted outputs.

This report aims to elucidate the theoretical underpinnings and actual programming of employing a neural network architecture with two hidden layers for forward and back propagation.

**Experimental setup**

I used the spyder IED to run the pytorch and python libraries. The version of pytorch was 2.1.2 with python 3.10.8 and the version of cuda was 11.8 for this programming assignment. To run the simulation of the learning, I used the laptop with CPU i7-8750 and GPU RTX 2070 max\_Q(for laptop).

**Hyper Parameters and Model description**

There are Three types of hyper parameters in this model.

1. Learning rate: For this model, I adjusted the learning rate by observing and analyzing the graph of inaccuracy over epochs. to prevent the overfitting, Learning rate was setted to 0.03
2. Weight and Bias size and initialization:
   1. The weight of pytorch in linear initialized with the learnable weights of the module of shape(out\_features,in\_features). The values are initialized from -root(k) to root(k) where k = 1/in\_feature
   2. The bias of of pytorch in linear initialized with the learnable weights of the module of shape(out\_features).The values are initialized from -root(k) to root(k) where k = 1/in\_feature
3. Mini Batch Size:
4. The mini Batch size of this model is 50 which implies there are 1000 iteration for each epoch.

**Model description**

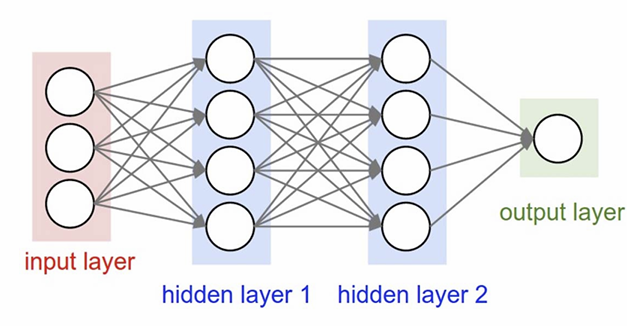
There are 4 fully connected layer in this model with input size of 784 features input and 10 possible class output. and Each layers is initialized with pytorch linear layer.

1. Input Layer X: There are 784 feature value
2. Hidden Layer H1: There are 100 nodes with weight W1(784\*100), bias W1\_0(100\*1)
   1. H1 = Relu((W1^t)\*X + W1\_0)
3. Hidden Layer H2: There are 100 nodes with weight W2(100\*100), bias W2\_0(100\*1)
4. H2 = Relu((W2^t)\*H1 + W2\_0)
5. Output Layer Y: There are 10 nodes with weight W3(100\*10), bias W3\_0(10\*1)
6. Y = Softmax((W3^t)\*H2 + W3\_0)

**Multilayer Neural Network with Backward Propagation**

**Multilayer Neural Network**

A multilayer neural network is a type of artificial neural network that consists of more than one layer of neurons between the input and output layers. These intermediate layers are called hidden layers. The basic structure of a multilayer neural network includes:



Input Layer: This layer receives the input features. It acts as the initial data layer through which patterns are introduced to the network.

Hidden Layers: One or more layers of neurons that process the inputs received from the previous layer. Each neuron in a hidden layer transforms the values from the previous layer with a weighted linear summation followed by a non-linear activation function. These layers allow the network to learn complex patterns and relationships in the data.

Output Layer: The final layer that produces the output of the network. The function of this layer and the number of neurons depend on the specific task.

The connections between neurons across layers are associated with weights that are adjusted during the training process.

**Forward Propagation**

There are input layer X, two hidden layer H, and output layer Y in this neural network.

1. Input layer to First hidden layer H1:

The input layer (X) receives the input features. These features are then multiplied by a set of weights (W1) and added to a bias term (W1\_0) to compute the pre-activation values for the first hidden layer (H1).

The formula for the H1 = Relu((W1^t)\*X + W1\_0)

1. H1 to second hidden layer H2:

The output of the first hidden layer (after applying ReLU) is then passed to the econd hidden layer. This involves multiplying by the second set of weights (W2) and adding a second bias term (W2\_0) to compute the pre-activation values for H2.

The formula for the H2 = Relu((W2^t)\*H1 + W2\_0)

1. H2 to Output layer Y:

The activated output of the second hidden layer is used to compute the pre-activation values of the output layer. This is done by multiplying H2 by the output layer weights (W3) and adding an output bias term (W3\_0)

.The formula for the Y = Softmax((W3^t)\*H2 + W3\_0)

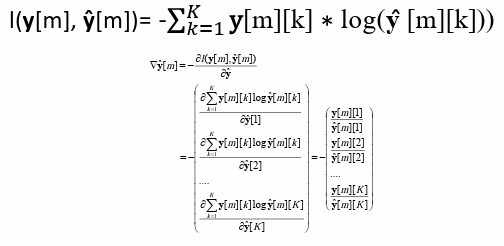
Where Relu(x) = max(0,x), softmax(Yi) = e^Yi/sum e^Yj

**Backward Propagation**

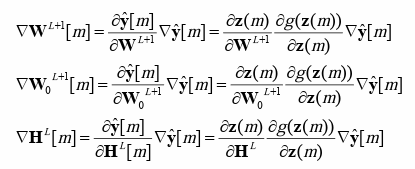
The gradient of the loss function is calculated with respect to each weight in the network. This process starts from the output layer and moves backward through the network. This is achieved by applying the chain rule of calculus, which allows the computation of the gradients of the loss function with respect to each weight by propagating the error gradient backward through the network.

**Computing gradient of Y**

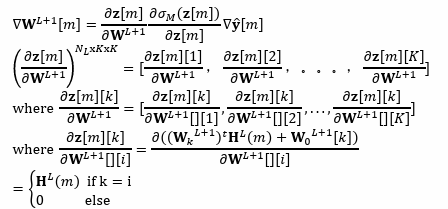
To calculate the gradient of Y with cross entropy from given data D= {X(m), Y(m)}. We could use

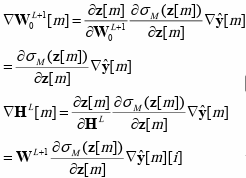


By using gradient of Y, We could calculate the gradient of W3, W3\_0 and H2.This is achieved by applying the chain rule of calculus. Where Z(m) = (W3^t)\*H2 + W3\_0. Computing process of gradient of each layer have same equation except difference in activation functions.

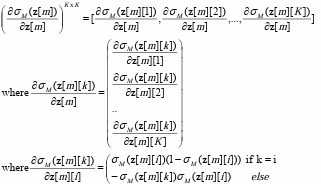
****

For our case, L = 2 in this model. We know that sigma function of Y is softmax. Therefore,

****

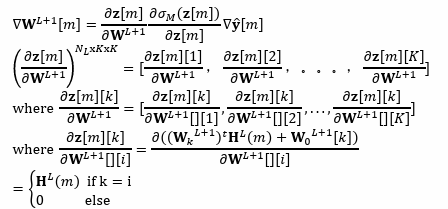
****

where:

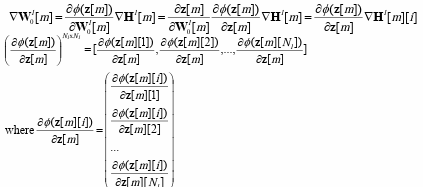
****

**Computing gradient of Hidden layers**

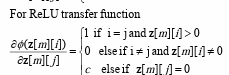
By using gradient of H2 and so on, We could calculate the gradient of W2, W2\_0,W1,W1\_0 and H1. This is also achieved by applying the chain rule of calculus. Where Z(m) = (W2^t)\*H1 + W2\_0 to get H1 and gradient of second hidden layer. where Where Z(m) = (W1^t)\*X + W1\_0 to get gradient of first hidden layer. In this process only one difference from the previous equation is activation function which is relu.

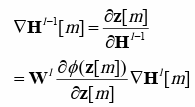
****

To get the gradients of second hidden layer to we could set L = 1. and To get the gradient of first hidden layer we could set L= 0, and H^0 = X. There are some difference in sigma function(relu) therefore,



where:



****

Every equation and picture in this section is from the class slide(Chapter3 Deep Neural Network)

**Mini batch**

There are 50000 training data sets. To get more accurate training results, mini batch selection was used in this model. the mini batch size was 50, and for each epoch, there are 1000 iterations. To get the average gradient of the mini batch. I just sum up all 50 data and get the average by dividing it by 50.

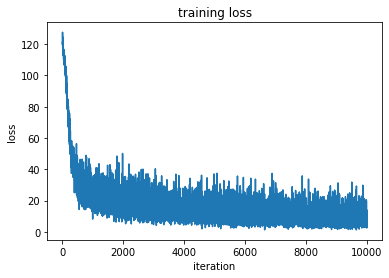
**Wl\_avg = (sum of 50 Wl elements in mini-batch)/50**

After calculating the gradient matrix, we could update the Theta for the next epoch. We could repeat gradient descent until it reaches the number of maximum epochs. For my implementation, the learning rate is 0.03 and the maximum epoch is 10. The R(theta) on the equation below is regularization and It was ignored for this assignment(ECSE 4850).



**Training Error, Testing Error and Training loss**

**Training loss**

****

**Training loss over iteration(each iteration was performed with 50 size minibatch)**

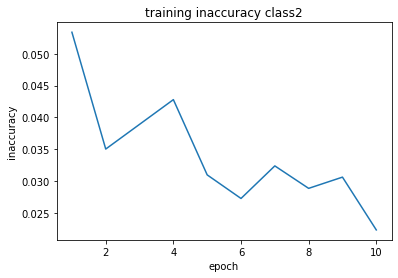
Initial Learning Phase: The model experiences a steep decline in training loss during the initial iterations, indicating that it is rapidly learning from the training data. This phase is critical as the model parameters are adjusted significantly to reduce the loss, suggesting that the initial parameter values were quite far from optimal.

Intermediate Learning Phase: Following the sharp decline, the curve begins to flatten, with a gradual decrease in loss. This is typical of the learning process as the model starts to fine-tune the parameters on more nuanced patterns within the training data.The fluctuating nature of the curve within this phase reflects the stochastic gradient descent process employed with mini-batches, which can cause the loss to bounce around due to the variance in each mini-batch.

Convergence Phase: Beyond approximately 8,000 iterations, the loss appears to plateau with minor fluctuations around a mean value. This behavior suggests the model is approaching convergence and finding a local minimum.The lack of a sustained downward trend in the latter iterations implies that additional training is yielding marginal or no significant improvements.

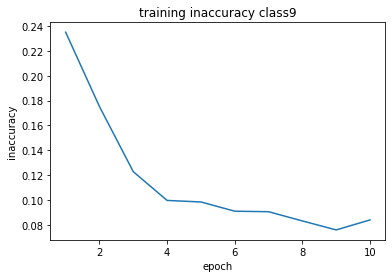
The training loss graph indicates a typical learning trajectory, with rapid initial improvement followed by a gradual approach to a loss minimum. The model shows signs of convergence after around 8,000 iterations, with minor fluctuations.

**Training Error of each class**

****

****

****

****

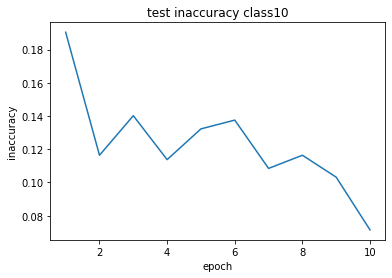
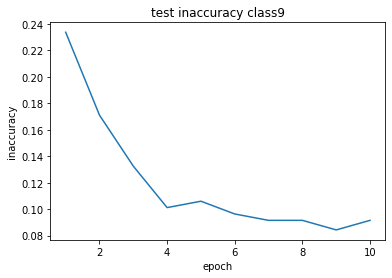
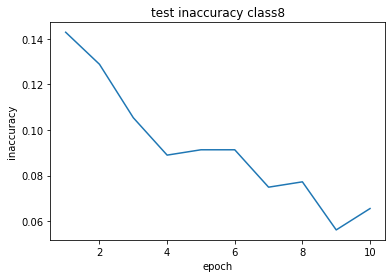
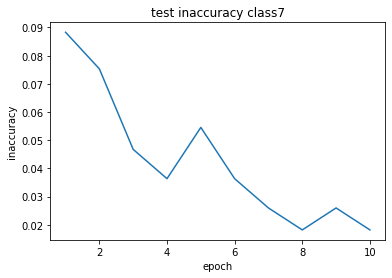
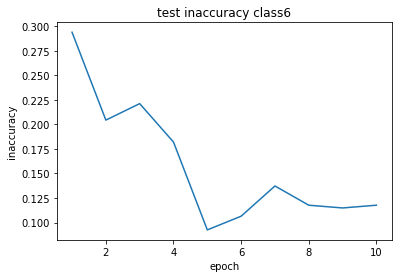
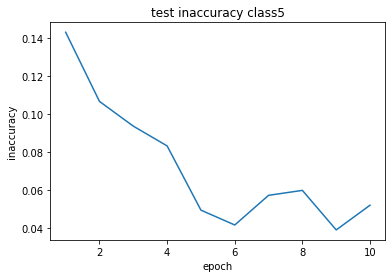
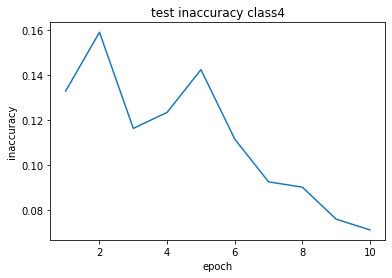
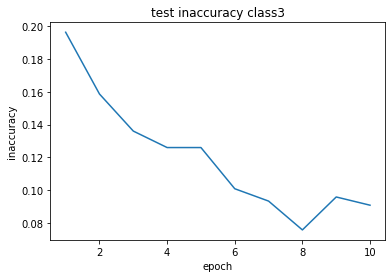
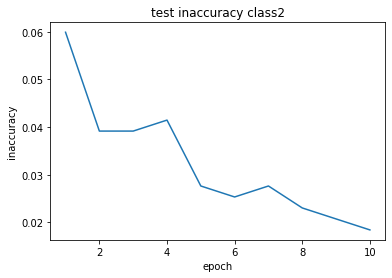
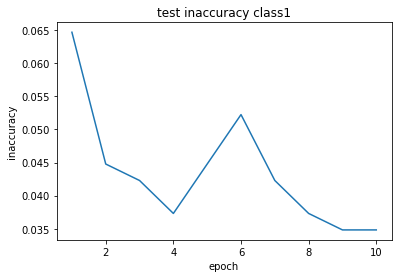
Classes 3, 5, 6, and 9 show a consistent decrease in training inaccuracy, suggesting that the model is steadily learning to classify these classes better over time.

Class 6 has the highest initial inaccuracy but shows significant improvement, suggesting it may have more complex features that the model is eventually learning to decipher.

Classes 1, 2, 4, 7, 8, and 10 show some degree of volatility in training inaccuracy, with rises and dips across epochs. Such behavior might be a sign of the model struggling to fit to the more nuanced aspects of these classes, or it could reflect a learning rate that is too high, causing the model to overshoot the optimal parameters.

The training inaccuracy trends across different classes suggest that while the model is learning and improving, there is room for improvement in its training regimen. Particularly, the implementation of early stopping, learning rate adjustments, and potential data augmentation could lead to better, more stable generalization across all classes.

**Testing Error of each class**

****

Each test error graphs are similar to the same class train error graphs.

Classes 3, 5 , 6, and 9 show a consistent decrease in training inaccuracy, suggesting that the model is steadily learning to classify these classes better over time.

Class 6 has the highest initial inaccuracy but shows significant improvement, suggesting it may have more complex features that the model is eventually learning to decipher.

Classes 1, 2, 4, 7, 8, and 10 show some degree of volatility in testing inaccuracy, with rises and dips across epochs. Such behavior might be a sign of the model struggling to fit to the more nuanced aspects of these classes, or it could reflect a learning rate that is too high, causing the model to overshoot the optimal parameters.

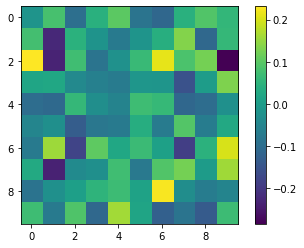
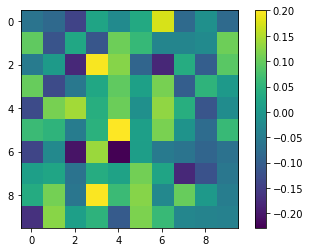
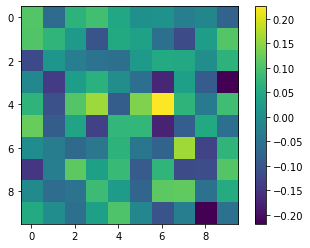
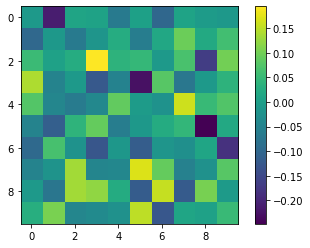
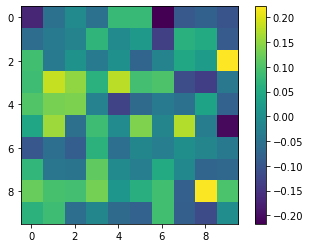
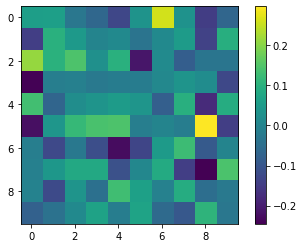
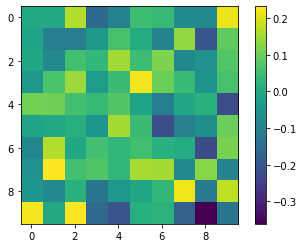
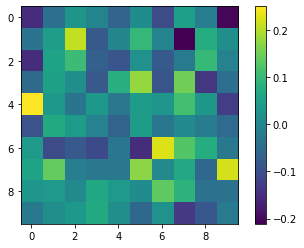
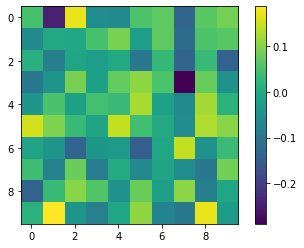
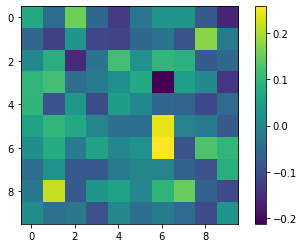
.The gap between the training and testing inaccuracy is small and consistent throughout the training process, which is a good sign that the model is not overfitting to the training data and is likely to perform well on unseen data.

**Initial weight and Final weight of W3**

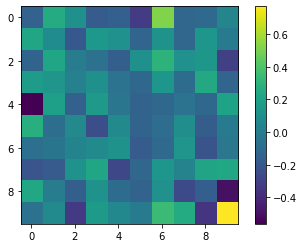
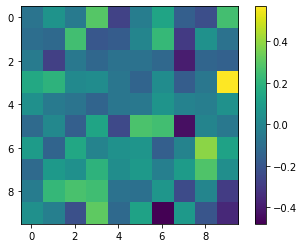
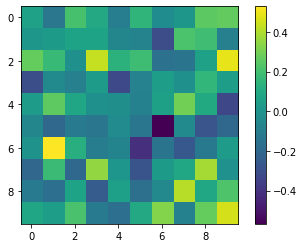
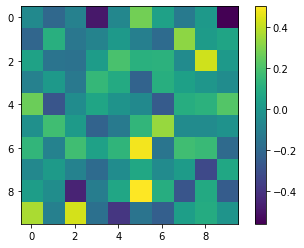
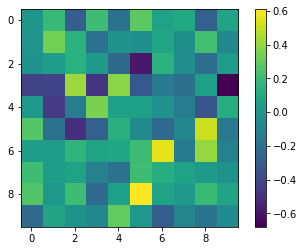
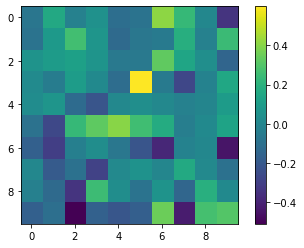
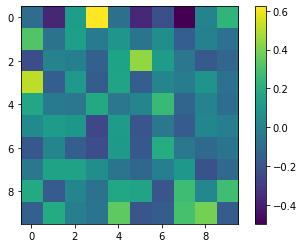
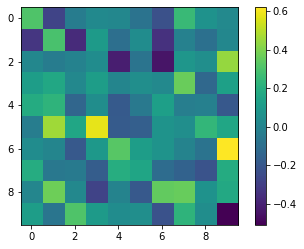
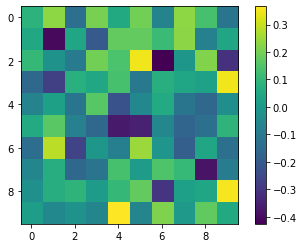
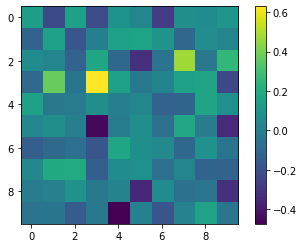
The weight of pytorch in linear initialized with the learnable weights of the module of shape(out\_features,in\_features). The values are initialized from -root(k) to root(k) where k = 1/in\_feature

The bias of of pytorch in linear initialized with the learnable weights of the module of shape(out\_features).The values are initialized from -root(k) to root(k) where k = 1/in\_feature

​

****

**Randomly Initialized Weight of W3 for each class (Starting from 1 to 10)**

****

**Final Weight of W3 for each class (Starting from 1 to 10)**

**Inaccuracy of testing data for each class**

| **Class** | **Inaccuracy** |
| --- | --- |
| 1 | 0.01990049751243781 |
| 2 | 0.013824884792626729 |
| 3 | 0.07052896725440806 |
| 4 | 0.07600950118764846 |
| 5 | 0.057291666666666664 |
| 6 | 0.0784313725490196 |
| 7 | 0.025974025974025976 |
| 8 | 0.053864168618266976 |
| 9 | 0.06265060240963856 |
| 10 | 0.06084656084656084 |
| AVG | 0.051932224781129965 |

**Inaccuracy of testing data for each class**

The table shows a table listing the inaccuracy of testing data for ten different classes and the average inaccuracy across all classes.

* The neural network performed best on Class 2 and Class 1, with inaccuracies of approximately 1.38% and 1.99%, respectively.
* The most challenging classes for the network were Class 6 and Class 4, with the highest inaccuracies at roughly 7.84% and 7.60%.
* The average inaccuracy across all classes is about 5.19%, suggesting that on average, the network gets about 5 out of every 100 predictions wrong across all classes.

The variation in inaccuracy rates across different classes could be due to several factors, including the inherent difficulty of distinguishing certain classes, the amount and quality of training data available for each class, or the features that the model is using to make predictions. The relatively low average inaccuracy suggests that the model, while having varying performance across different classes, is overall making accurate predictions on the testing dataset.

**conclusion**

In conclusion, the presented table indicates the performance of a multi-layer neural network across ten different classes in a classification task, with the average inaccuracy rate at approximately 5.19%. Notably, the model exhibits variable performance across the classes, suggesting an opportunity for optimization. While Classes 1 and 2 show the best performance with the lowest inaccuracies, Class 6 demonstrates a higher susceptibility to misclassification, which may be a focal point for improvement.

To enhance the model's performance, several strategies could be employed. This includes augmenting the training data for underperforming classes, and error analysis to identify common misclassification trends. Additionally, adjusting model complexity and applying regularization techniques could address potential overfitting issues.

Moreover, incorporating an early stopping mechanism into the training process could prevent overfitting by terminating the training when the model's performance on a validation set ceases to improve. Early stopping serves as a form of regularization, ensuring that the model maintains generalization capabilities and does not continue to learn noise and unnecessary details from the training dataset.

By implementing these strategies, especially early stopping, there is a promising potential to improve the neural network's accuracy and achieve more balanced performance across all classes. This will lead to a more robust model capable of better generalization when exposed to new, unseen data.