Project 4: Introduction to Deep Learning

Group Members - Sruti Munukutla and Mahvash Maghrabi

Github - https://github.com/mahvashmaghrabi/CS6140_Project4

Working Pattern -

Task 1, 2 - Mahvash Maghrabi
Task 3, 4 - Sruti Munukutla

MNIST dataset

This project is based on the MNIST dataset. The MNIST dataset is a collection of 70,000 grayscale images of handwritten digits from 0 to 9, with 60,000 images used for training and 10,000 for testing. This dataset is used for image classification tasks. Each image in the dataset is 28x28 pixels in size, and the goal of the classification task is to correctly classify the digit represented in each image.

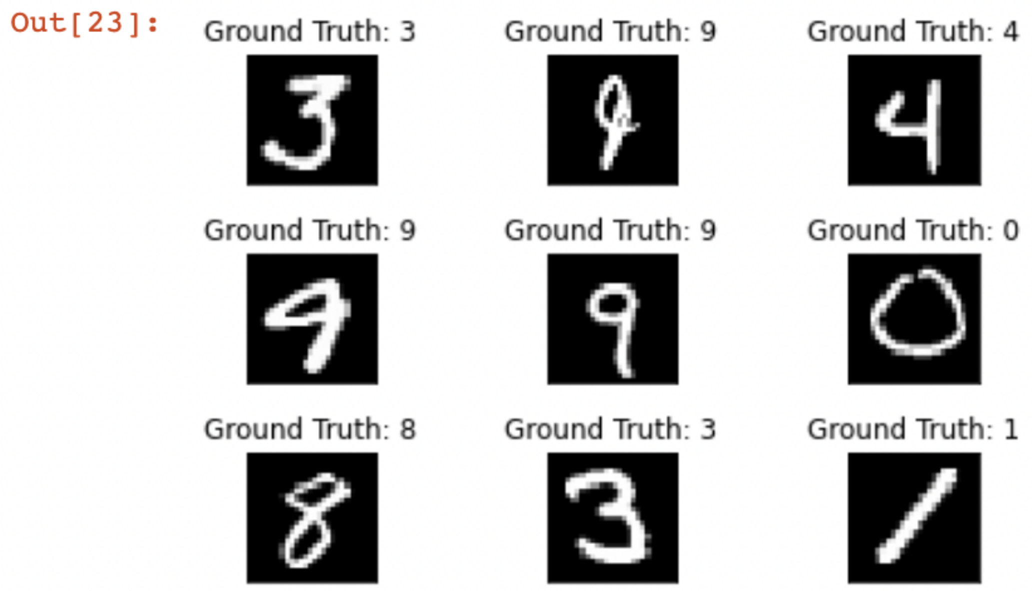
A Convolutional Neural Network (CNN) is a Deep Learning algorithm that can take in an input image, assign learnable weights and biases to various objects in the image, and be able to differentiate one from the other. The working of CNN is done by processing the input image through a series of convolutional layers, pooling layers, and fully connected layers to learn a set of features that can be used for classifying the image.
For the MNIST digit recognition task that we are working on for Task 1, a network architecture can be designed to take in a 28x28 pixel grayscale image as input and output a probability distribution over the 10 possible digits.
The first layer of the network would be a convolutional layer, which would apply a set of learnable filters to the input image to extract important features such as edges and corners. The output of the convolutional layer would then be passed through a non-linear activation function, such as a ReLU, to introduce non-linearity into the model. Then a pooling layer can be added to the network to reduce the spatial dimensionality of the feature maps while retaining important features. This can help to reduce the number of parameters in the model and prevent overfitting. Additional convolutional layers and pooling layers can be added as necessary to further learn more about the features. Finally, the output of the convolutional layers can be passed through one or more fully connected layers to learn a set of weights that can be used to classify the input image. The output layer of the network would typically be a softmax layer, which would output a probability distribution over the 10 possible digits. This project is done using PyTorch. It is an open-source machine learning framework that is primarily used for developing deep learning models. PyTorch was

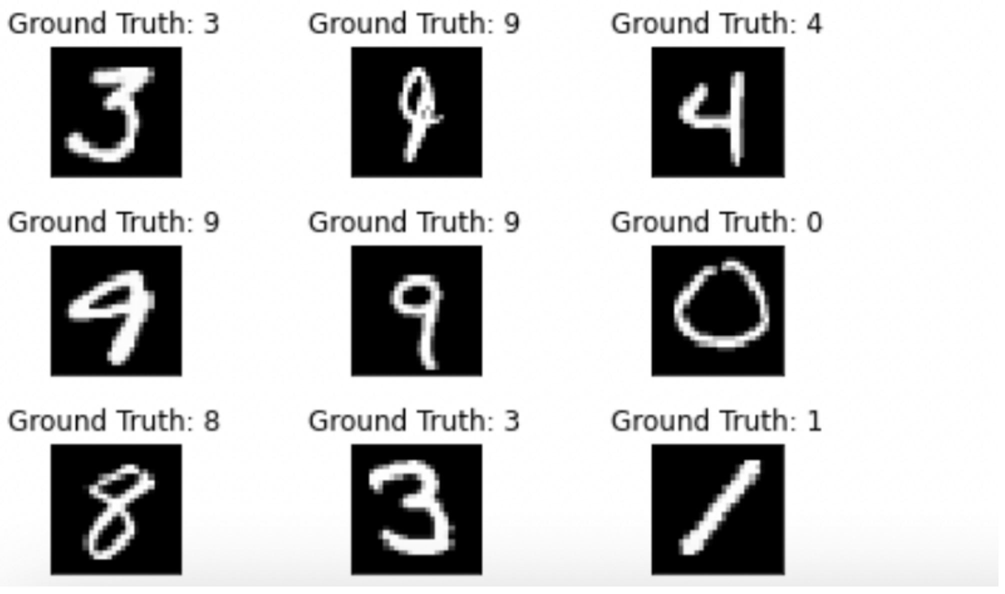
developed by Facebook's AI research team. It has dynamic execution graphs which means the computation graph is created on the fly.

Task 1 - MNIST Tutorial

The first step towards a ML project is to import the necessary libraries. The hyperparameters like number of epochs show the number of times we will loop over the training dataset and the learning rate and momentum are for optimization. We will need dataloaders. DataLoaders are used to load and preprocess data from a dataset to use while training. It allows for efficient loading of large datasets that may not fit into memory.

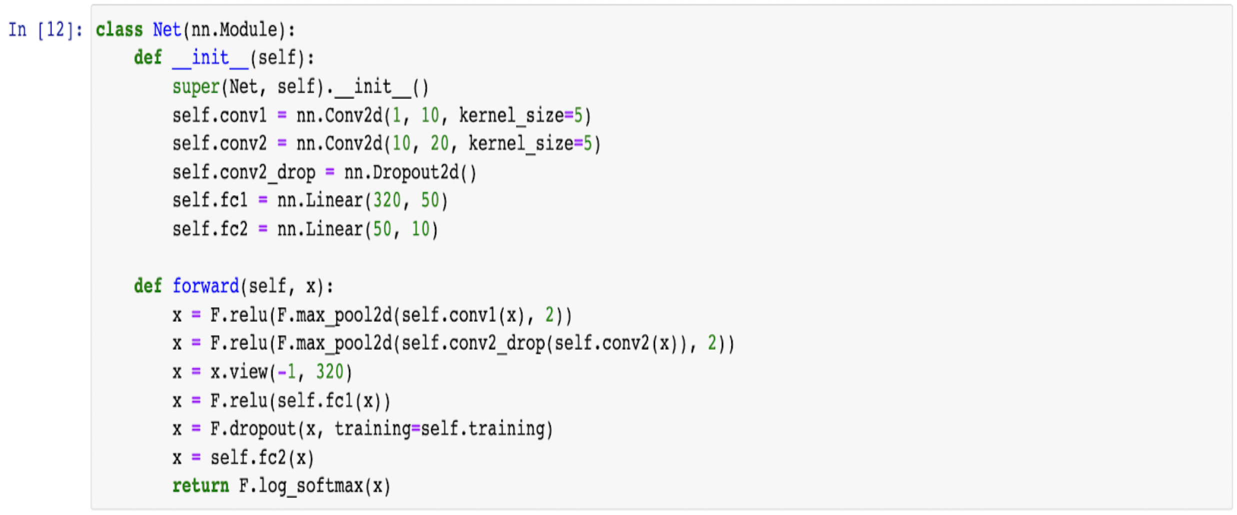
Plotting the test data batch using matplotlib.





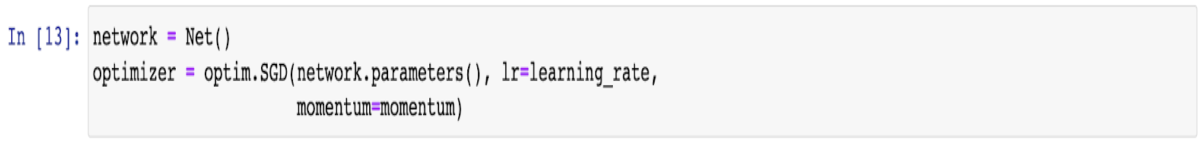
Convolutional Neural Network

The convolution network we are building here has 2-D convolutional layers followed by two linear layers. For activation Rectified Linear Units (ReLUs) is used and for regularization two dropout layers will be used.



```
In [12]: class Net(nn.Module):
             def __init__(self):
                 super(Net, self).__init__()
                 self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
                 self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
                 self.conv2_drop = nn.Dropout2d()
                 self.fc1 = nn.Linear(320, 50)
                 self.fc2 = nn.Linear(50, 10)

             def forward(self, x):
                 x = F.relu(F.max_pool2d(self.conv1(x), 2))
                 x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
                 x = x.view(-1, 320)
                 x = F.relu(self.fc1(x))
                 x = F.dropout(x, training=self.training)
                 x = self.fc2(x)
                 return F.log_softmax(x)
```

The next step would be to initialize the network and the optimizer.



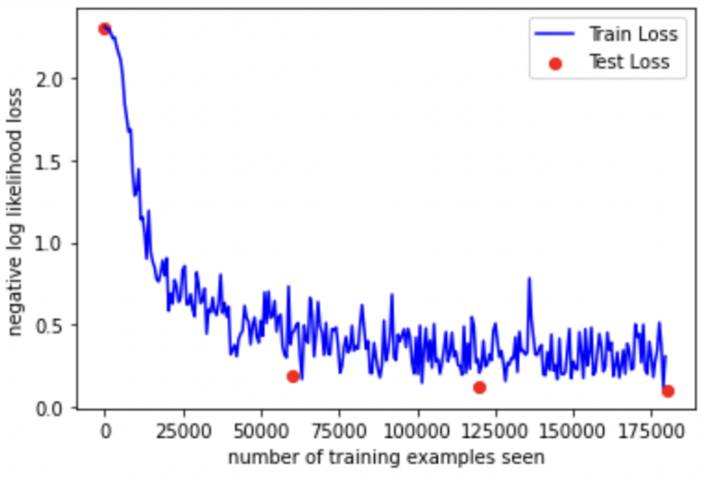
```
In [13]: network = Net()
         optimizer = optim.SGD(network.parameters(), lr=learning_rate,
                               momentum=momentum)
```

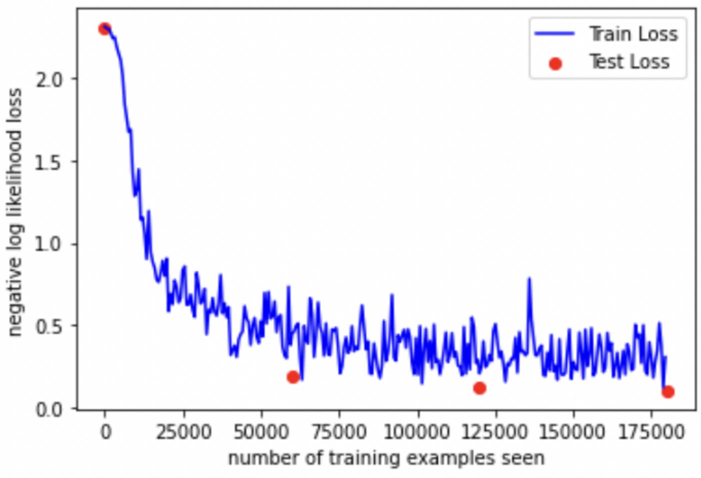
Training and Evaluating Model

The next step will be training the model. For creating a training curve later we also create two lists for saving training and testing losses. X-axis will display the number of training examples the network has seen during training. Running the test loop before starting will help us see the accuracy/loss we achieve just with randomly initialized network parameters.

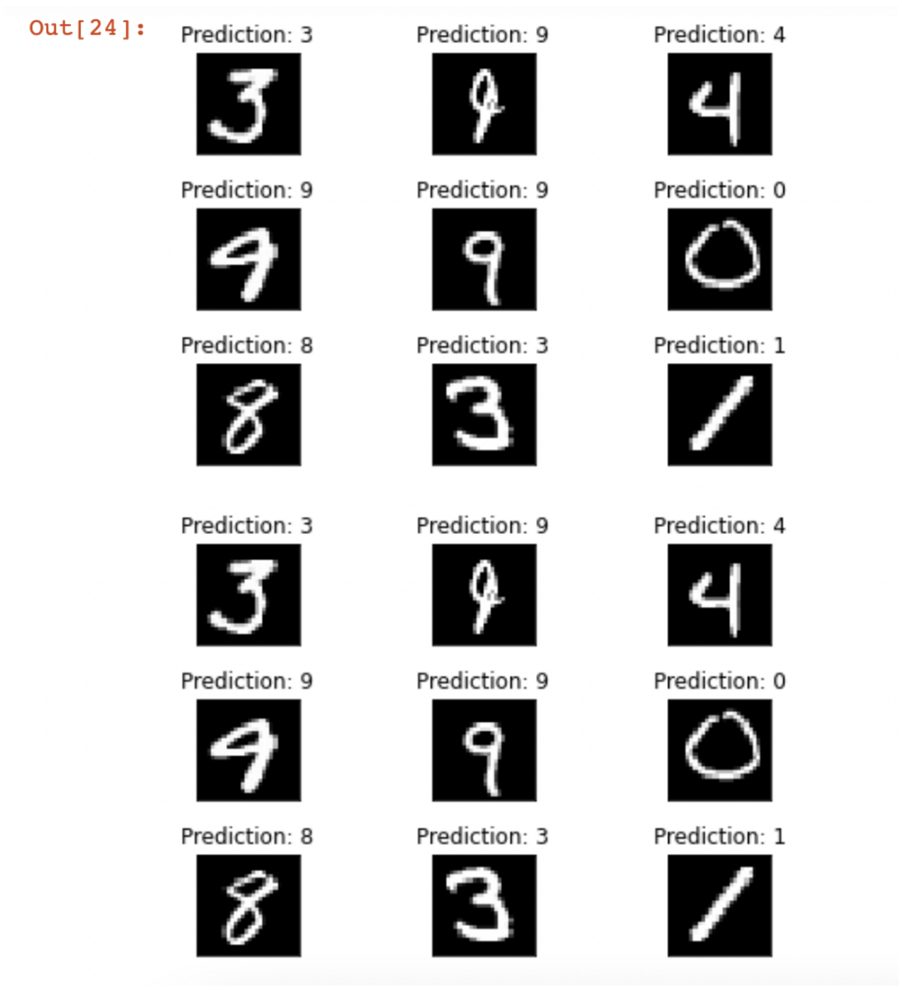
Plotting the Training Curve (Train Loss and Test Loss)

```
Out[18]:
```



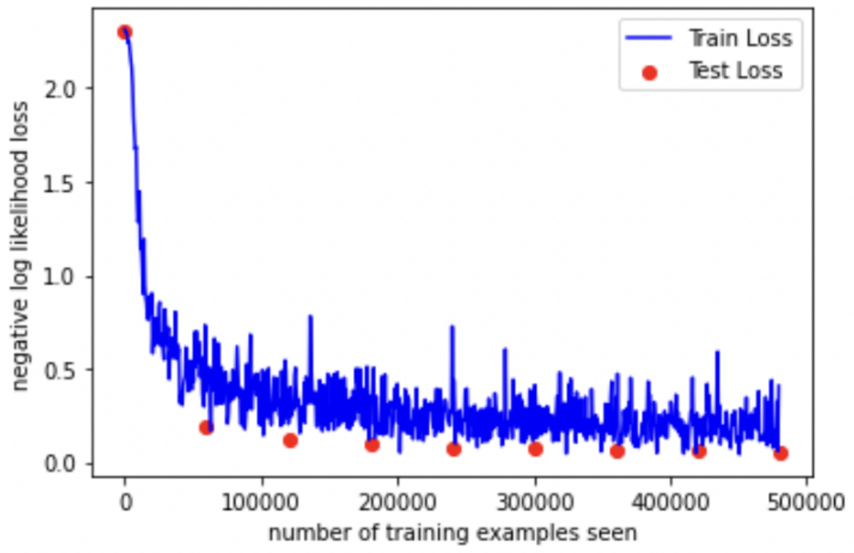


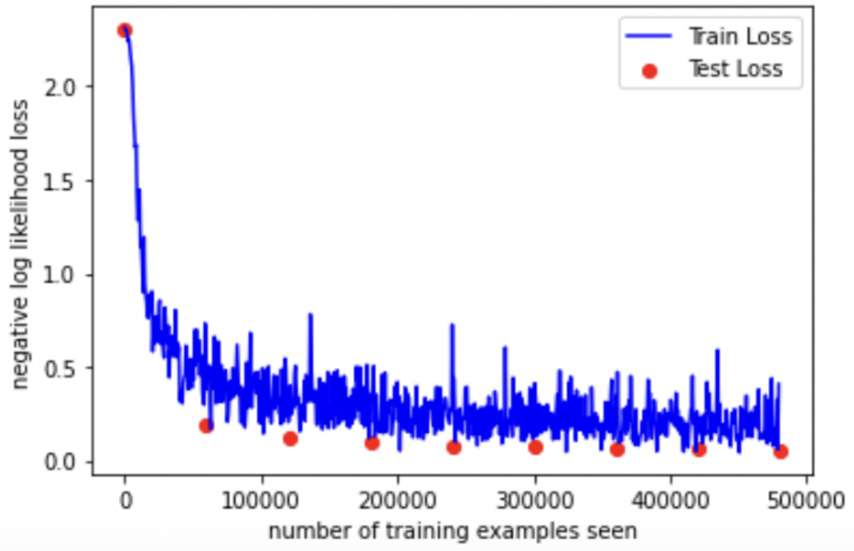
Comparing the Model’s output


Out[24]:

Continuing training the network and visualizing this to further inspect the training progress.

```
Out[57]:
```





This gives a smooth learning curve and there are no overfitting issues which shows that the dropout layers did a good job regularizing the model.

Task 2 - Experiment with Network Variations

For this task we are using the MNIST Fashion data set. This dataset is a collection of 70,000 grayscale images of 28x28 pixels and each image represents a different fashion item. Ths is used for image classification tasks in machine learning and computer vision.The dataset is divided into two sets: a training set of 60,000 images and a test set of 10,000 images. There are images in the dataset that correspond to one of 10 possible classes, which include: T Shirt/tops, Trousers, Pullovers, Dresses, Coats, Sandals, Shirts, Sneakers, Bag, Ankle boots. These images were collected by the researchers at Zalando, a popular online fashion retailer, and were labelled by human annotators.

The goal of this task was to evaluate the effect of changing different aspects of the network and how it would optimize the network performance and/or training time along those dimensions.

Importing and Loading

The first step towards any machine learning project is importing the necessary libraries and loading the dataset.

Building the Network

The code for building the network has a NetEstimator class that extends Net which is a neural network model and it implements the BaseEstimator interface from Scikit-learn so that we can perform Grid Search CV later during the task.

The __init__ method initializes the class with hyperparameters such as the number of convolutional layers (n_conv_layers), filter size (conv_filter_size), number of filters (n_conv_filters), number of nodes in the dense layer (dense_nodes), dropout rate (dropout_rate), pool filter size (pool_filter_size), activation function (activation_func), batch size (batch_size), number of epochs (epochs), and learning rate (learning_rate).

The fit method trains the neural network on the input data using the Adam optimizer and cross-entropy loss function and it iterates through the training data for epochs number of epochs, and for each epoch, it loops through the batches in the train_loader.

The predict method makes predictions on the input data using the trained model. It iterates through the test data in batches using the test_loader.

The score method calculates the accuracy of the model predictions on the input data with ground truth labels.

The get_params method returns a dictionary of the hyperparameters of the model that will be used for Grid Search CV.

Training and Evaluating the Model

The code for training and testing has two functions to train and test a PyTorch model.
The train function takes as input the model to be trained, train_loader and val_loader containing training and validation set, criterion which is the loss function, optimizer for the model's parameters, and the number of epochs to train the model for.
The test function takes as input the model to be tested and the test_loader containing the test data. The test function returns the average test accuracy. After training the model the Test Accuracy obtained is 89.81%.

GridSearchCV

I have used GridSearchCV to automate the process. GridSearchCV is a hyperparameter tuning technique used in machine learning to find the optimal set of hyperparameters for a given model.It works by taking a set of hyperparameters, usually defined as a dictionary, and creating all possible combinations of hyperparameters from that set. Then the model is then trained and evaluated for each combination of hyperparameters and the best combination is selected.

The hyperparameters considered for the dictionary were :

1. The number of convolution layers
2. The size of the convolution filters
3. The number of convolution filters in a layer
4. The number of hidden nodes in the Dense layer
5. The dropout rates of the Dropout layer
6. The size of the pooling layer filters
7. The activation function for each layer

The hypothesis as to how it is expected of the network to behave along each dimension is as follows :

1. **The number of convolution layers:** When the number of convolution layers increases it also increases the network's ability to extract higher-level features from the input data.

2. **The size of the convolution filters:** Larger filters are expected to capture larger spatial features in the input data, whereas smaller filters are expected to capture finer details. Using larger filters may lead to higher accuracy.

3. **The number of convolution filters in a layer:** When the number of filters is increased the network's ability to learn more complex and diverse features from the input data also increases.

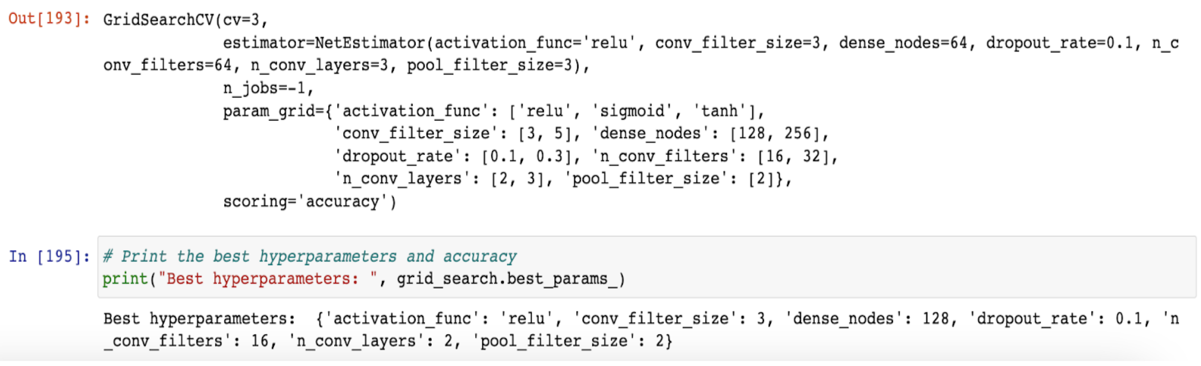
4. **The number of hidden nodes in the Dense layer:** When the number of nodes in the dense layer increases it also increases the model's capacity to learn complex non-linear relationships between the features extracted by the convolution layers, but it might lead to overfitting.

5. **The dropout rates of the Dropout layer:** As the dropout rate increases the network's ability to generalize to new data while reducing overfitting also increases whereas a very high dropout rate may result in underfitting.

6. **The size of the pooling layer filters:** Increasing the size of the pooling layer filters is expected to increase the amount of downsampling of the feature maps which leads to a decrease in the spatial resolution of the data but it increases the network's ability to generalize to new data.

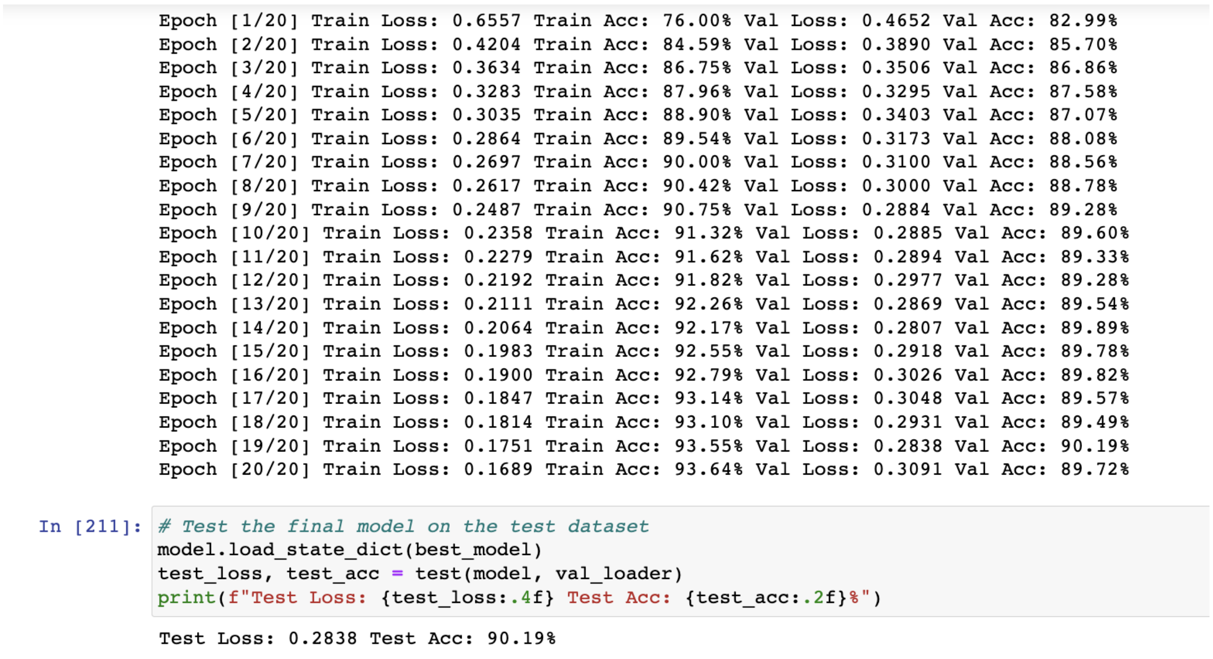
7. **The activation function for each layer:** Activation functions have varied properties, like being more or less sensitive to saturation and being more or less suited to handle different types of input data. Choosing the right activation function for each layer is important to get good performance on the task.

After performing GridSearchCV the results obtained for the best hyperparameters were :



```
Out[193]: GridSearchCV(cv=3,
                       estimator=NetEstimator(activation_func='relu', conv_filter_size=3, dense_nodes=64, dropout_rate=0.1, n_c
          onv_filters=64, n_conv_layers=3, pool_filter_size=3),
                       n_jobs=-1,
                       param_grid={'activation_func': ['relu', 'sigmoid', 'tanh'],
                                   'conv_filter_size': [3, 5], 'dense_nodes': [128, 256],
                                   'dropout_rate': [0.1, 0.3], 'n_conv_filters': [16, 32],
                                   'n_conv_layers': [2, 3], 'pool_filter_size': [2]},
                       scoring='accuracy')
```

```
In [195]: # Print the best hyperparameters and accuracy
          print("Best hyperparameters: ", grid_search.best_params_)

          Best hyperparameters:  {'activation_func': 'relu', 'conv_filter_size': 3, 'dense_nodes': 128, 'dropout_rate': 0.1, 'n
          _conv_filters': 16, 'n_conv_layers': 2, 'pool_filter_size': 2}
```



```
Epoch [1/20] Train Loss: 0.6557 Train Acc: 76.00% Val Loss: 0.4652 Val Acc: 82.99%
Epoch [2/20] Train Loss: 0.4204 Train Acc: 84.59% Val Loss: 0.3890 Val Acc: 85.70%
Epoch [3/20] Train Loss: 0.3634 Train Acc: 86.75% Val Loss: 0.3506 Val Acc: 86.86%
Epoch [4/20] Train Loss: 0.3283 Train Acc: 87.96% Val Loss: 0.3295 Val Acc: 87.58%
Epoch [5/20] Train Loss: 0.3035 Train Acc: 88.90% Val Loss: 0.3403 Val Acc: 87.07%
Epoch [6/20] Train Loss: 0.2864 Train Acc: 89.54% Val Loss: 0.3173 Val Acc: 88.08%
Epoch [7/20] Train Loss: 0.2697 Train Acc: 90.00% Val Loss: 0.3100 Val Acc: 88.56%
Epoch [8/20] Train Loss: 0.2617 Train Acc: 90.42% Val Loss: 0.3000 Val Acc: 88.78%
Epoch [9/20] Train Loss: 0.2487 Train Acc: 90.75% Val Loss: 0.2884 Val Acc: 89.28%
Epoch [10/20] Train Loss: 0.2358 Train Acc: 91.32% Val Loss: 0.2885 Val Acc: 89.60%
Epoch [11/20] Train Loss: 0.2279 Train Acc: 91.62% Val Loss: 0.2894 Val Acc: 89.33%
Epoch [12/20] Train Loss: 0.2192 Train Acc: 91.82% Val Loss: 0.2977 Val Acc: 89.28%
Epoch [13/20] Train Loss: 0.2111 Train Acc: 92.26% Val Loss: 0.2869 Val Acc: 89.54%
Epoch [14/20] Train Loss: 0.2064 Train Acc: 92.17% Val Loss: 0.2807 Val Acc: 89.89%
Epoch [15/20] Train Loss: 0.1983 Train Acc: 92.55% Val Loss: 0.2918 Val Acc: 89.78%
Epoch [16/20] Train Loss: 0.1900 Train Acc: 92.79% Val Loss: 0.3026 Val Acc: 89.82%
Epoch [17/20] Train Loss: 0.1847 Train Acc: 93.14% Val Loss: 0.3048 Val Acc: 89.57%
Epoch [18/20] Train Loss: 0.1814 Train Acc: 93.10% Val Loss: 0.2931 Val Acc: 89.49%
Epoch [19/20] Train Loss: 0.1751 Train Acc: 93.55% Val Loss: 0.2838 Val Acc: 90.19%
Epoch [20/20] Train Loss: 0.1689 Train Acc: 93.64% Val Loss: 0.3091 Val Acc: 89.72%
```

```
In [211]: # Test the final model on the test dataset
          model.load_state_dict(best_model)
          test_loss, test_acc = test(model, val_loader)
          print(f"Test Loss: {test_loss:.4f} Test Acc: {test_acc:.2f}%")

          Test Loss: 0.2838 Test Acc: 90.19%
```

Finally, the model was trained using the best hyperparameter values and number of epochs = 20 and after training the model accuracy increased to 90.19% and the test loss was 0.2838.

TASK 3:

The main aim of this task is to implement Transfer learning on a pre-trained model for the MNIST digits. In Transfer learning, the knowledge gained from training one model is stored and applied to another similar/related problem. Part3.ipynb implements the code from Part1 and the model and optimizer are saved as model.pth and optimizer.pth. While saving the trained model's parameters, it is only necessary to save the learned parameters(weights, biases). The learnable parameters are available in the model's state_dict which is a Python dictionary object which maps each layer to its parameter tensor.

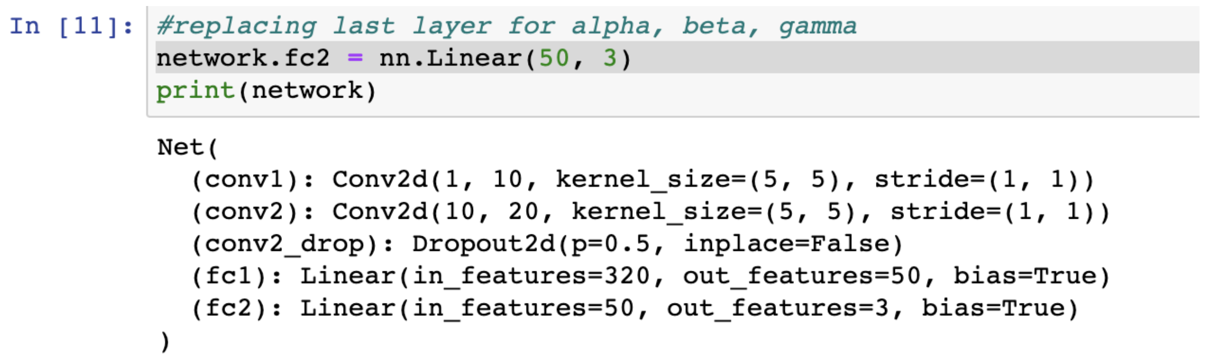
The dataset of Greek letters contains 27 images of alpha, beta and gamma which are coloured and and 133 x 133 in resolution. Since the MNIST digit recognition model was trained on grayscale images of 28 x 28 size, the greek letter images are scaled, cropped and transformed into 28 x 28 grayscale images to match the format of MNIST digit dataset.

IMPLEMENTATION:

Firstly the model trained on MNIST digits is loaded and the network is built. The weights are freezed so that to use the knowledge from the MNIST digit dataset for greek letter dataset. Since the digit dataset's output layer has 10 nodes (0-9 digits and hence 10 labels) and the Greek Letter dataset has 3 labels in the output layer (alpha, beta and gamma), the last Linear layer of the learned model needs to be removed and updated for 3 nodes implemented by the line:

network.fc2 = nn.Linear(50, 3)

After which the network as below:



```
In [11]: #replacing last layer for alpha, beta, gamma
         network.fc2 = nn.Linear(50, 3)
         print(network)

         Net(
           (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
           (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
           (conv2_drop): Dropout2d(p=0.5, inplace=False)
           (fc1): Linear(in_features=320, out_features=50, bias=True)
           (fc2): Linear(in_features=50, out_features=3, bias=True)
         )
```

When n_epochs = 3,

Test set: Avg. loss: 0.6685, Accuracy: 26/27 (96%)

N_epochs = 10, Test set: Avg. loss: 0.4776, Accuracy: 26/27 (96%)

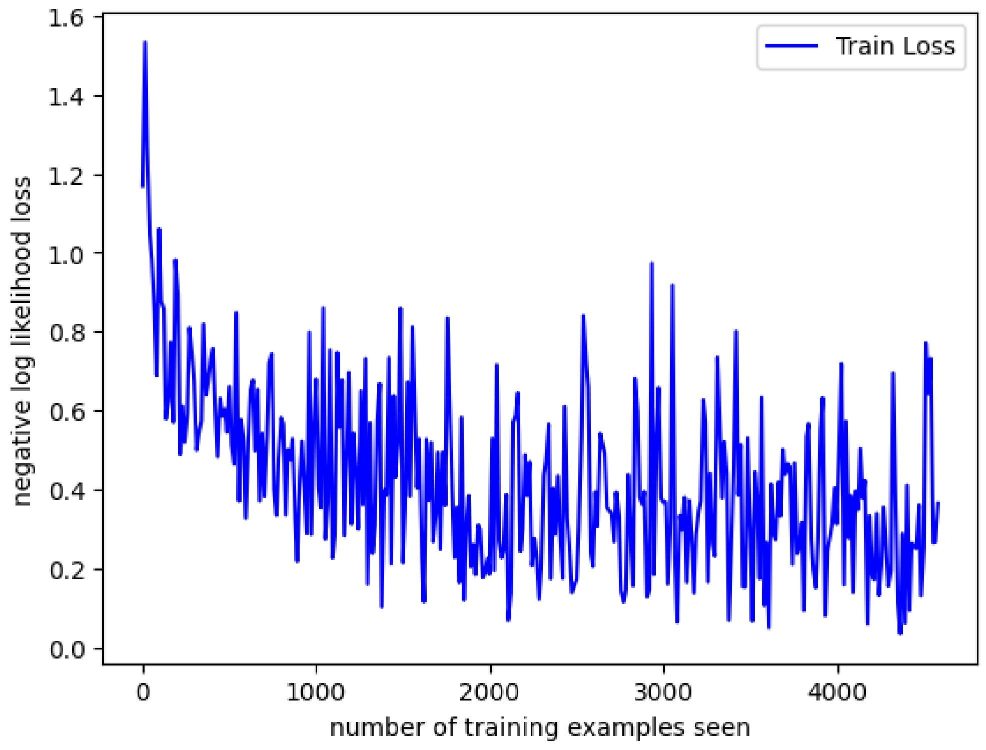
For n_epoch = 169, the accuracy was stably showing 100% accuracy.

Train Epoch: 169 [0/27 (0%)]	Loss: 0.731707
Train Epoch: 169 [15/27 (50%)]		Loss: 0.265068

Test set: Avg. loss: 0.1337, Accuracy: 27/27 (100%)

```
Train Epoch: 168 [0/27 (0%)]    Loss: 0.770938
Train Epoch: 168 [15/27 (50%)]  Loss: 0.642796

Test set: Avg. loss: 0.1340, Accuracy: 26/27 (96%)

Train Epoch: 169 [0/27 (0%)]    Loss: 0.731707
Train Epoch: 169 [15/27 (50%)]  Loss: 0.265068

Test set: Avg. loss: 0.1337, Accuracy: 27/27 (100%)

Train Epoch: 170 [0/27 (0%)]    Loss: 0.266834
Train Epoch: 170 [15/27 (50%)]  Loss: 0.365134

Test set: Avg. loss: 0.1336, Accuracy: 27/27 (100%)
```

The figure below shows training losses when total epoch size = 170, which means there were 170 * 27 = 4,590 samples.



TASK 4:

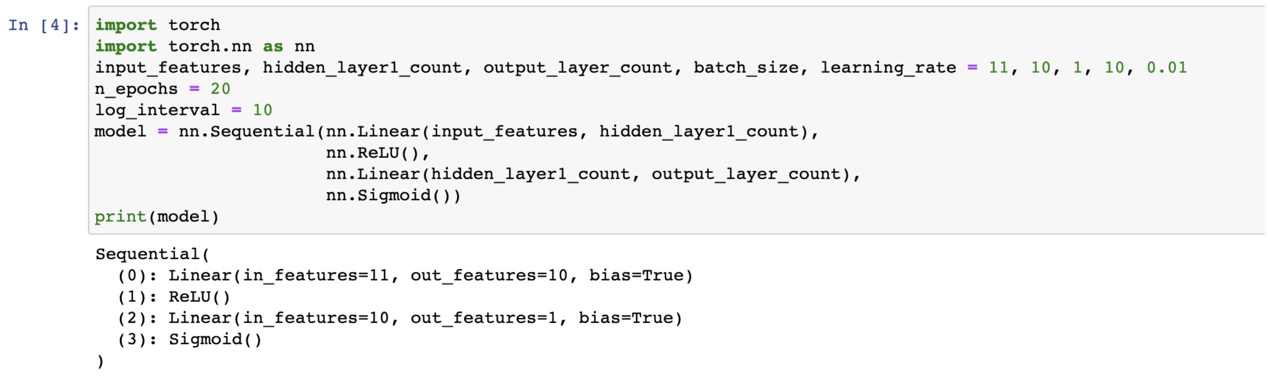
The aim of this task is to build an artificial neural network to train and classify the Heart Dataset used in Project 3. Since data is loaded from a CSV file, a custom DataSet class is built around which the DataLoader can be wrapped to load data from a Pandas Dataframe.

For Loss Function, Mean Squared Error loss function provided by PyTorch is used.

The parameters used below are as follows:

input_features, hidden_layer1_count, output_layer_count, batch_size, learning_rate = 11, 10, 1, 10, 0.01

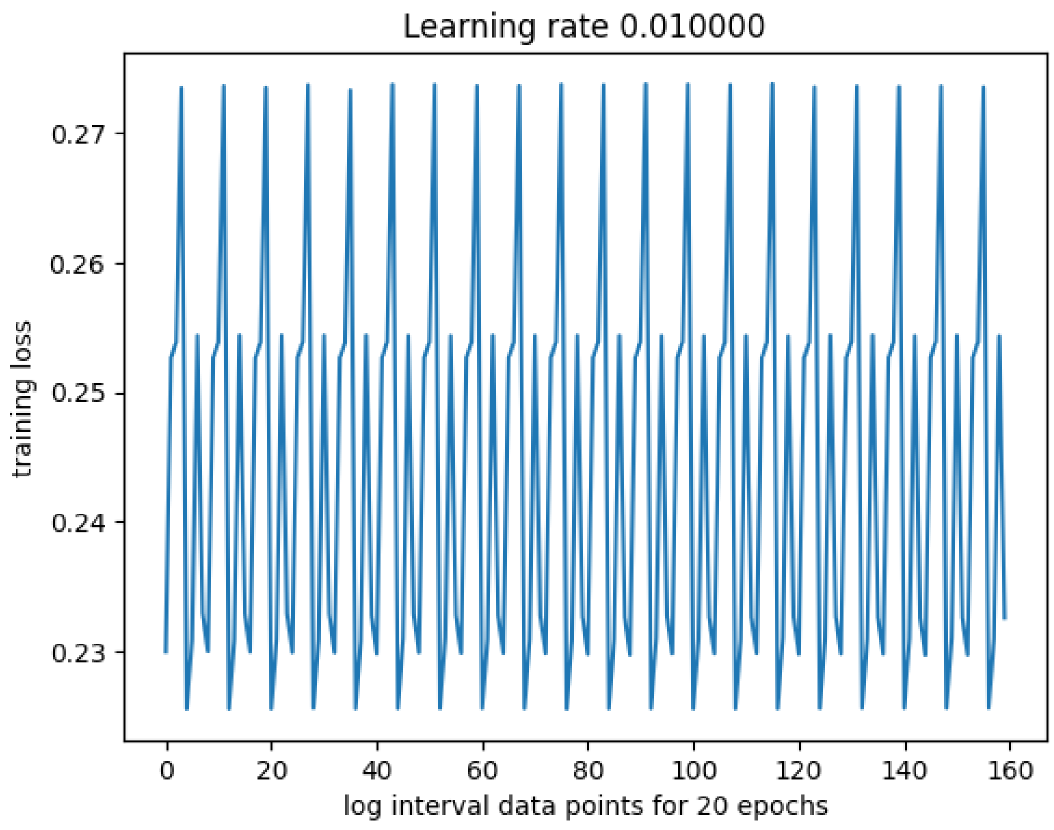
The first architecture of the neural network has an input layer with 11 nodes (representing all the 11 input features of the csv file), one hidden layer with 10 nodes and an output layer with one node representing the binary classification. The network which is a simple sequential layer with sigmoid output layer is printed as below:



```
In [4]: import torch
        import torch.nn as nn
        input_features, hidden_layer1_count, output_layer_count, batch_size, learning_rate = 11, 10, 1, 10, 0.01
        n_epochs = 20
        log_interval = 10
        model = nn.Sequential(nn.Linear(input_features, hidden_layer1_count),
                              nn.ReLU(),
                              nn.Linear(hidden_layer1_count, output_layer_count),
                              nn.Sigmoid())
        print(model)
```

```
Sequential(
  (0): Linear(in_features=11, out_features=10, bias=True)
  (1): ReLU()
  (2): Linear(in_features=10, out_features=1, bias=True)
  (3): Sigmoid()
)
```

The stochastic gradient descent optimizer provided by PyTorch is used. The results of the network losses for each epoch are shown below.



The smallest loss for 20 epochs was 0.23

```
Train Epoch: 20 [700/718 (97%)] Loss: 0.232522
```

Network architecture : 2

For this model, an increase in hidden layers with back-propagation was done to see if it leads to better performance.

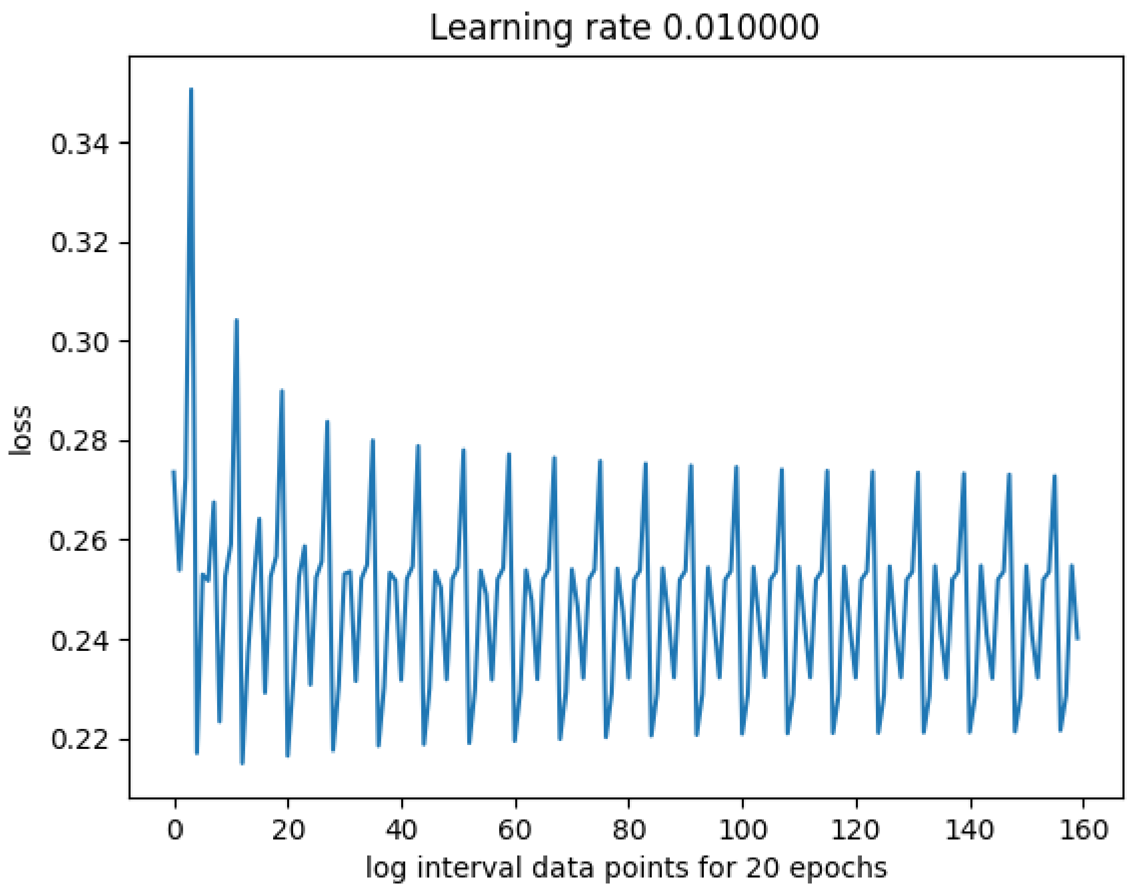
The network is printed as below:

```
Sequential(
  (0): Linear(in_features=11, out_features=15, bias=True)
  (1): ReLU()
  (2): Linear(in_features=15, out_features=20, bias=True)
  (3): ReLU()
  (4): Linear(in_features=20, out_features=30, bias=True)
  (5): ReLU()
  (6): Linear(in_features=30, out_features=40, bias=True)
  (7): ReLU()
  (8): Linear(in_features=40, out_features=1, bias=True)
  (9): Sigmoid()
)
```

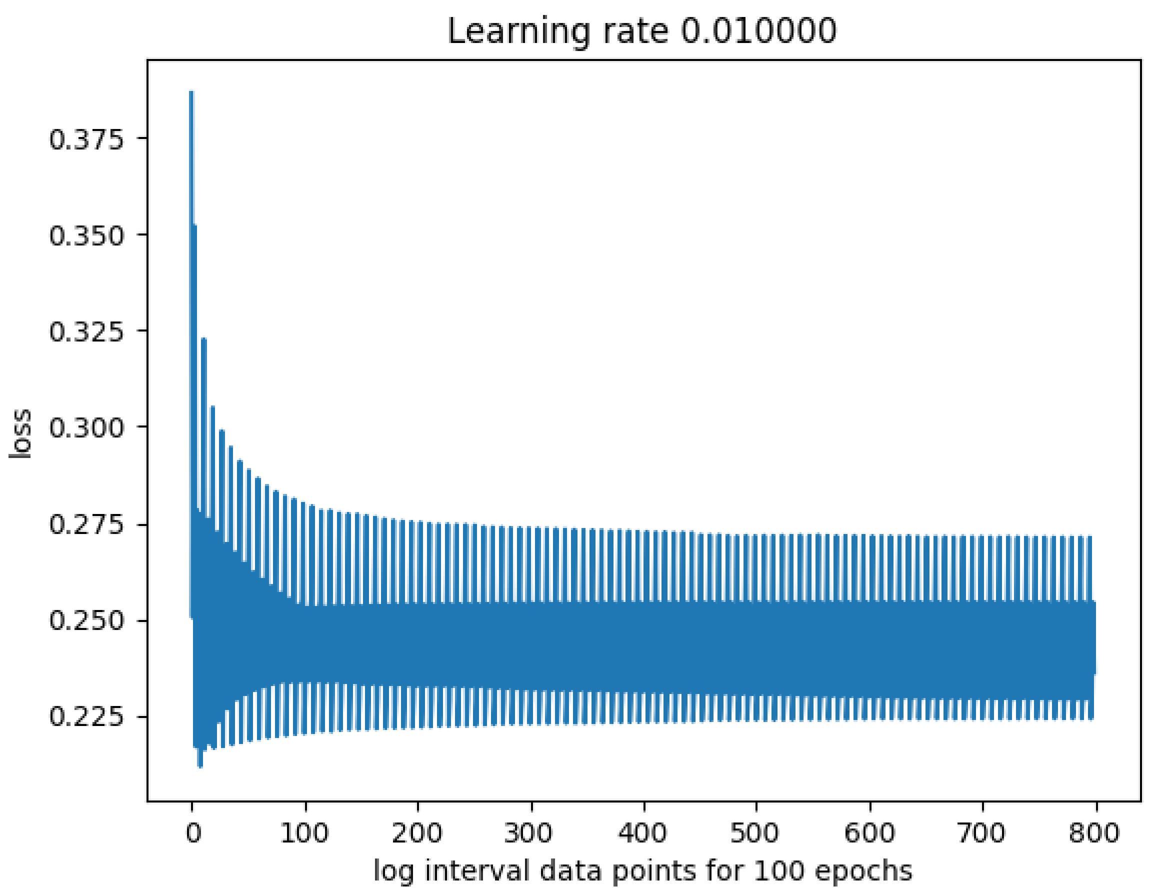
In total, there are 4 hidden layers where the first hidden layer has 15 nodes, second - 20 nodes, third 30 nodes and fourth hidden layer - 40 nodes. The final output layer is same as in the first model.

Observation:

By increasing the number of hidden layers, a significant improvement in the losses was not observed. The epoch size, learning rate and all the other hyper-parameters were maintained the same.



The training loss observed over the log intervals of 10 for epoch size - 20 is shown above. The loss gradually shows a descent, it does not reduce much in comparison to architecture 1. Results for increasing epoch size to 100 for architecture 2 are shown below:



The loss stagnates after a certain point and shows no change with respect to epoch size.

In comparison to Decision Tree results, it has been observed that the performance on test data was not good. The accuracy for both models is 48%. We felt that with better feature extraction using PCA and regularisation with Dropout function can help improve the performance.

SUMMARY

This project was a very good learning experience. It helped a lot to understand the working of Deep Learning. Starting with the MNIST digit recognition dataset which is regarded as the hello world of image recognition and further moving to the MNIST fashion dataset which was a little more challenging but similar in size and it also provided more room to see the effects of changes in the network. A lot of things were covered in detail and cleared the concept of image recognition using PyTorch in depth. It was also very helpful in understanding how transfer learning can be used on pre-trained models to predict output of similar problems. The importance of deciding the correct epoch size and other hyper-parameters is crucial in helping achieve high accuracy for a chosen model. Also, from our analysis on the Heart Disease dataset, we realised that feature extraction is crucial even for complex models and increasing the number of layers and epoch size may not necessarily always yield best results.

REFERENCES

1. https://nextjournal.com/gkoehler/pytorch-mnist
2. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53
3. https://pytorch.org/vision/main/generated/torchvision.datasets.FashionMNIST.html
4. https://www.kaggle.com/datasets/zalando-research/fashionmnist/code
5. https://www.youtube.com/watch?v=8z2oLfK2sIc
6. https://machinelearningmastery.com/how-to-develop-a-cnn-from-scratch-for-fashion-mnist-clothing-classification/
7. https://medium.com/@aaysbt/fashion-mnist-data-training-using-pytorch-7f6ad71e96f4
8. https://towardsdatascience.com/build-a-fashion-mnist-cnn-pytorch-style-efb297e22582
9. https://towardsdatascience.com/build-a-simple-neural-network-using-pytorch-38c55158028d
10. https://androidkt.com/load-pandas-dataframe-using-dataset-and-dataloader-in-pytorch/
11. https://www.datascienceweekly.org/tutorials/pytorch-mnist-load-mnist-dataset-from-pytorch-torchvision
12. https://pytorch.org/tutorials/beginner/saving_loading_models.html#what-is-a-state-dict

ACKNOWLEDGEMENT

I, Mahvash Maghrabi, and my group partner, Sruti Munukutla, would like to express our gratitude and appreciation to all those who gave us the opportunity to complete this project and the report. We extend our special thanks to Professor Bruce Maxwell who provided us the opportunity to work on this project and always gave us stimulating suggestions and encouragement during the lectures. We would also like to express our gratitude to our Teaching Assistants, Srishti Hedge and Yiming Ge, who were always available to resolve our doubts and queries. Lastly, we would like to thank each other as we both worked together and supported each other throughout the project.