CS 5814 - Introduction to Deep Learning: HW1 Report

General Introduction to Dataset, Model Architecture, Code overview

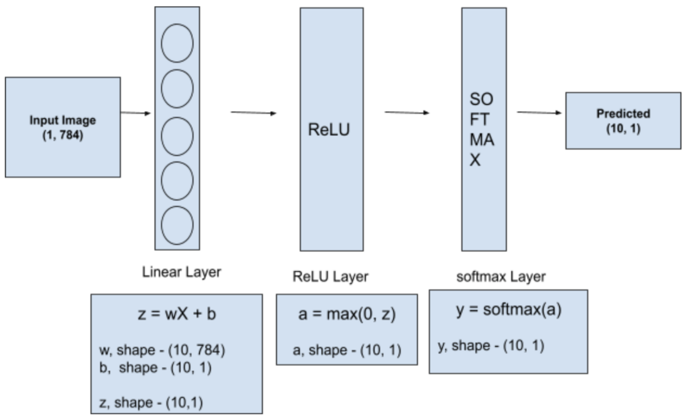
MNIST Dataset

The MNIST dataset is a database of handwritten digits. The database contains 60,000 training images and 10,000 test images. The size of the images is 28x28 and the label is the digit value represented by the image.

Model Architecture

The model has a three-layer architecture defined below:

- Linear Layer (contains 10 neurons)
- ReLU Layer
- Softmax Layer



The output of the softmax layer is the predicted label represented in 1 hot vector style where the index of max value in that vector would represent the digit. If we consider one image input to the model then first the input would be flattened to a vector of dimension (1, 784).

The Linear layer would transform the input as described in the image above and the output of the linear layer would be of dimension (10, 1) and this output would be input to the ReLU layer which would introduce non-linearity in the transformation. Further, this output would go into the softmax layer and that would produce an output of (10,1) i.e. 1-hot representation of the label.

Forward propagation and Backward propagation are discussed in detail in the Appendix section at the end of the report.

Code Overview:

- The solution has 3 .ipynb files, corresponding .py files, and one util .py file
- utils.py file contains the additional methods that are being used by the jupyter notebook
- A rough layout of the code is as follows:
    - Download the dataset, if not downloaded
    - Load the dataset in the correct format
    - Design your architecture
    - Define your model, optimizer, hyperparameter settings
    - Train your model and test the model on validation set in each epoch after we have trained it on all the mini-batches

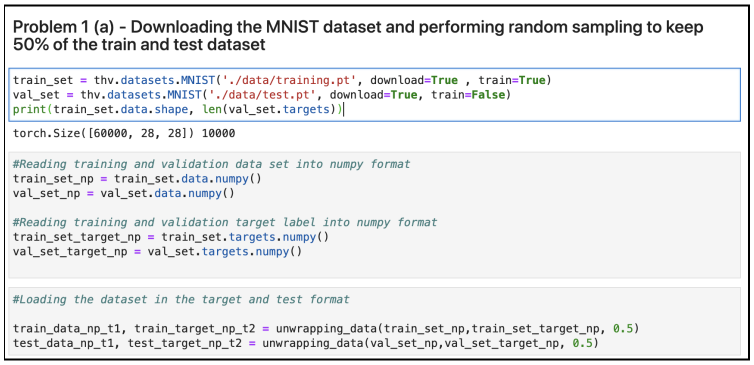
Problem 1 - Create Neural Network from scratch using NumPy for MNIST dataset

(a) Downloading the dataset using torchvision and sampling 50% dataset for train and validation set

Downloading the MNIST data in the local machine and then performing minor processing to randomly sampling 50% of the data in the train and the validation set.

A custom function name - ‘unwrapping_data’ has been written for the following tasks
- Randomly sample the train and test image 50% and their respective labels
- Convert the labels in the 1 hot encoded vector dim

The sample in the test and train sets are chosen based on the indices. For instance, we have 60,000 images in the training set therefore we randomly choose 30,000 numbers from 0 to 59,999 using numpy.random.choice method. Once the indices are chosen then we crop the data for those indices and store them in a numpy array.



Problem 1 (a) - Downloading the MNIST dataset and performing random sampling to keep 50% of the train and test dataset

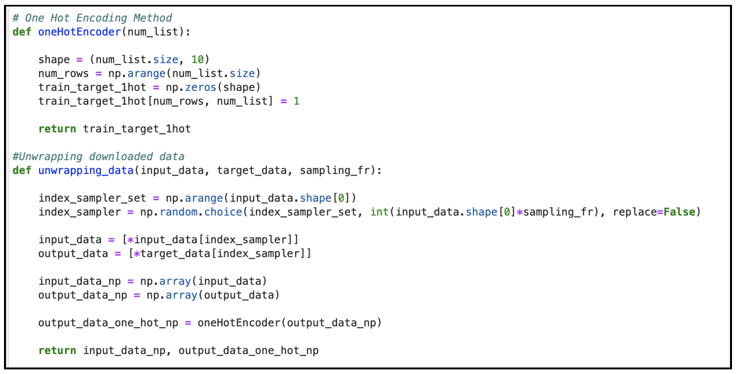
```
train_set = thv.datasets.MNIST('./data/training.pt', download=True , train=True)
val_set = thv.datasets.MNIST('./data/test.pt', download=True, train=False)
print(train_set.data.shape, len(val_set.targets))
```

```
torch.Size([60000, 28, 28]) 10000
```

```
#Reading training and validation data set into numpy format
train_set_np = train_set.data.numpy()
val_set_np = val_set.data.numpy()

#Reading training and validation target label into numpy format
train_set_target_np = train_set.targets.numpy()
val_set_target_np = val_set.targets.numpy()
```

```
#Loading the dataset in the target and test format

train_data_np_t1, train_target_np_t2 = unwrapping_data(train_set_np,train_set_target_np, 0.5)
test_data_np_t1, test_target_np_t2 = unwrapping_data(val_set_np,val_set_target_np, 0.5)
```

Code snippet for downloading data and performing random sampling

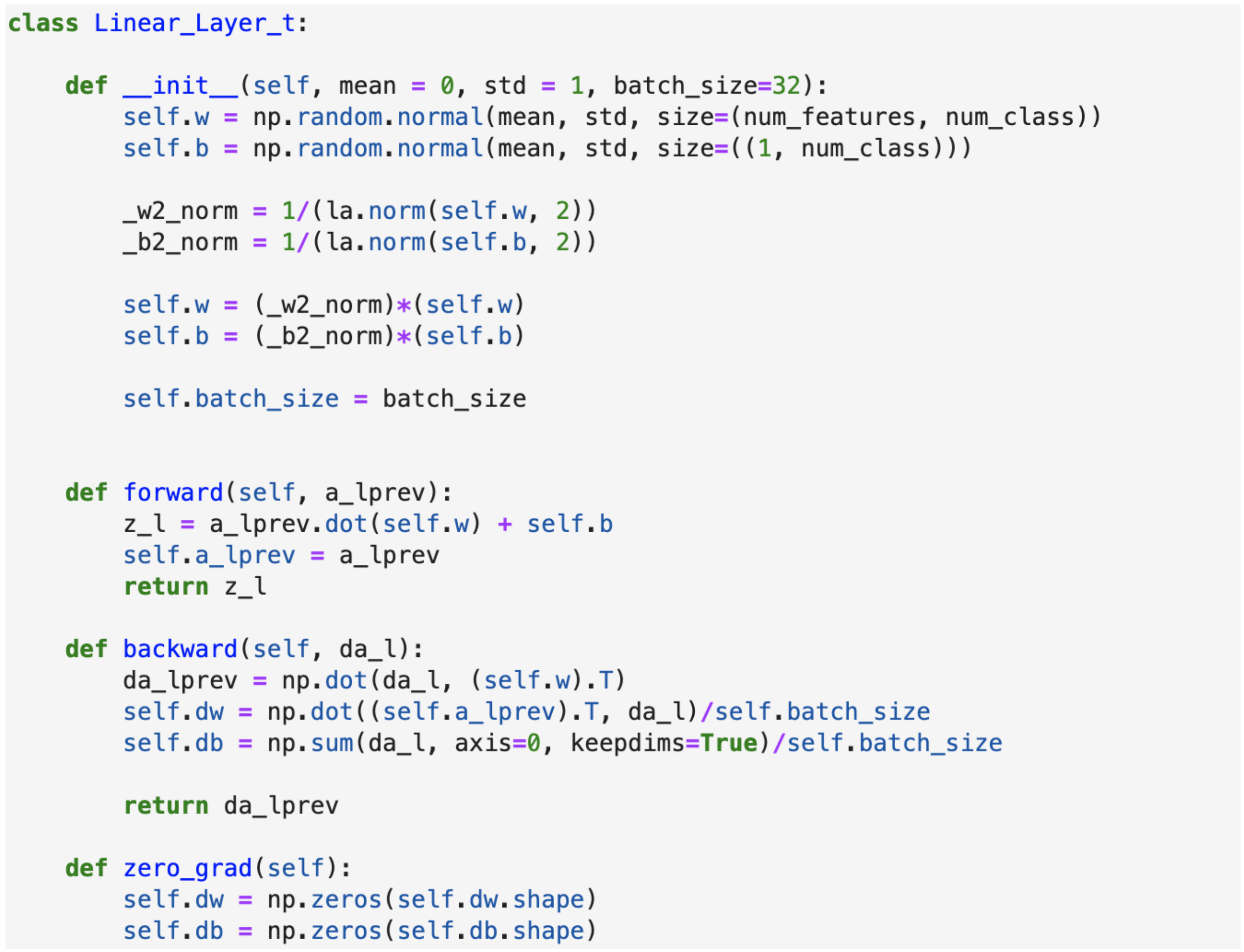


```
# One Hot Encoding Method
def oneHotEncoder(num_list):

    shape = (num_list.size, 10)
    num_rows = np.arange(num_list.size)
    train_target_1hot = np.zeros(shape)
    train_target_1hot[num_rows, num_list] = 1

    return train_target_1hot

#Unwrapping downloaded data
def unwrapping_data(input_data, target_data, sampling_fr):

    index_sampler_set = np.arange(input_data.shape[0])
    index_sampler = np.random.choice(index_sampler_set, int(input_data.shape[0]*sampling_fr), replace=False)

    input_data = [*input_data[index_sampler]]
    output_data = [*target_data[index_sampler]]

    input_data_np = np.array(input_data)
    output_data_np = np.array(output_data)

    output_data_one_hot_np = oneHotEncoder(output_data_np)

    return input_data_np, output_data_one_hot_np
```

Code snippet for processing the numpy data and 1-hot encoding

(b) Implementing Linear Layer

Since the MNIST dataset has 10 classes so the linear layer has 10 neurons doing the linear transformation of the input data. Weights and Biases are initialized with the Gaussian entries (mean - 0 and standard deviation -1) with their L2norm equals 1.



```python
class Linear_Layer_t:

    def __init__(self, mean = 0, std = 1, batch_size=32):
        self.w = np.random.normal(mean, std, size=(num_features, num_class))
        self.b = np.random.normal(mean, std, size=((1, num_class)))

        _w2_norm = 1/(la.norm(self.w, 2))
        _b2_norm = 1/(la.norm(self.b, 2))

        self.w = (_w2_norm)*(self.w)
        self.b = (_b2_norm)*(self.b)

        self.batch_size = batch_size


    def forward(self, a_lprev):
        z_l = a_lprev.dot(self.w) + self.b
        self.a_lprev = a_lprev
        return z_l

    def backward(self, da_l):
        da_lprev = np.dot(da_l, (self.w).T)
        self.dw = np.dot((self.a_lprev).T, da_l)/self.batch_size
        self.db = np.sum(da_l, axis=0, keepdims=True)/self.batch_size

        return da_lprev

    def zero_grad(self):
        self.dw = np.zeros(self.dw.shape)
        self.db = np.zeros(self.db.shape)
```

Forward Propagation

For batch size 32, the input will be flattened so the input would be a matrix (X) of dimension - (32, 784).

Weight matrix dimension (w) - (10, 784)
Bias dimension (b) - (10, 1)
Output dimension (z) - (10, 1): z = X(w.T) + b

Numpy vectorization technique has been used to perform the computation.

Backward Propagation

dz/dx = w
dz/dw = X
dz/db = 1

(c) Implementing ReLU Layer

ReLU is a non-linear activation function.

```python
class Relu_t:

    def __init_(self):
        pass

    def forward(self, z_l):
        self.a_l = np.maximum(0, z_l)
        return self.a_l

    def backward(self, dL):
        self.a_l[self.a_l <= 0] = 0
        self.a_l[self.a_l > 0] = 1
        self.dz_da = dL*self.a_l
        return self.dz_da

    def zero_grad(self):
        self.dz_da = np.zeros(self.dz_da.shape)
```

Forward Propagation

a = max(0, z) when z>0 otherwise a = 0

Backward Propagation

da/dz = 1 when z>0 otherwise 0

(d) Implementing combined softmax and cross-entropy layer

The softmax activation function is the last layer in the architecture and its output is used for predicting the class label.

```python
class Softmax_Cross_Entropy_t ():

    def __init__(self):
        pass

    def forward(self, z_lprev, yhat=None):
        y_out = np.exp(z_lprev)/(np.sum(np.exp(z_lprev), axis=1, keepdims=True))
        self.pred = y_out
        if yhat is not None:
            loss = - np.sum((yhat*np.log(y_out)), axis = 1, keepdims=True)
            avg_loss = (1/(loss.shape[0]))*np.sum(loss, axis=0, keepdims=True)
            return self.pred, avg_loss
        return self.pred


    def backward(self, yhat):
        self.dL_dz = self.pred - yhat
        return self.dL_dz

    def zero_grad(self):
        self.dL_dz = np.zeros(self.dL_dz.shape)
```

<u>Forward Propagation</u>:

y is a matrix of dimension (32, 10)

$$y_i = \frac{np.exp(a_i)}{\sum_{i=1}^{10} np.exp(a_i)}$$

$$C = -\sum_{i=1}^{10} \hat{y} * (log(y))$$

$$L_{avg} = \sum_{i=1}^{32} C_i$$

<u>Backward Propagation</u>:

$$\frac{dL}{da} = y - \hat{y}$$

(e) Implementing forward and backward propagation for n=1 batch size

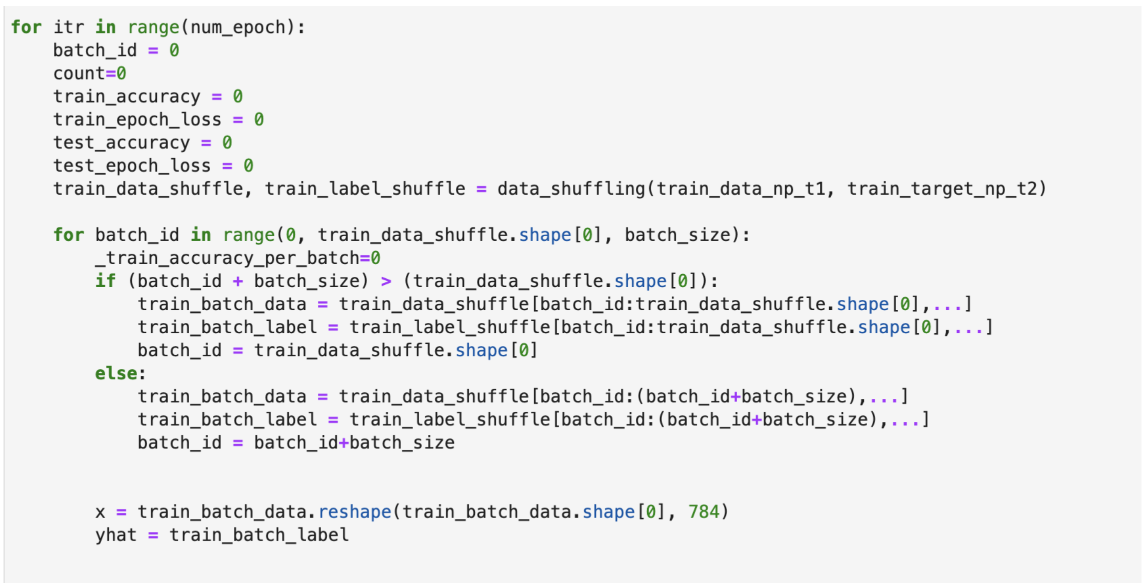
After implementing the Linear, ReLU, and Softmax, I tested the code for 1 iteration for 1 image to test the implementation of the forward and backward propagation. I randomly chose training sample 4 from the dataset and showed the computation in the Jupyter Notebook.

(f) & (g) Train neural network and test on validation dataset

Hyperparameter setting:

Parameters	**Values**
Learning Rate	0.01
Epochs	120
Batch size	32

I have first flattened the input matrix of dim 28x28 to a vector of size (1, 784). For batch size 32, the input would be a matrix of size (32, 784). Since there are 30,000 images in the training batch there would be 938 batches of input. In every epoch, the dataset is shuffled before starting the batch-wise training.

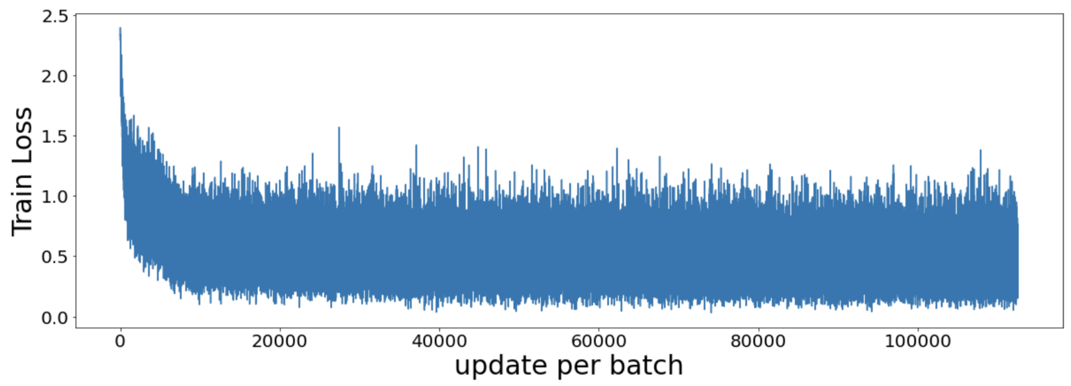


```
for itr in range(num_epoch):
    batch_id = 0
    count=0
    train_accuracy = 0
    train_epoch_loss = 0
    test_accuracy = 0
    test_epoch_loss = 0
    train_data_shuffle, train_label_shuffle = data_shuffling(train_data_np_t1, train_target_np_t2)

    for batch_id in range(0, train_data_shuffle.shape[0], batch_size):
        _train_accuracy_per_batch=0
        if (batch_id + batch_size) > (train_data_shuffle.shape[0]):
            train_batch_data = train_data_shuffle[batch_id:train_data_shuffle.shape[0],...]
            train_batch_label = train_label_shuffle[batch_id:train_data_shuffle.shape[0],...]
            batch_id = train_data_shuffle.shape[0]
        else:
            train_batch_data = train_data_shuffle[batch_id:(batch_id+batch_size),...]
            train_batch_label = train_label_shuffle[batch_id:(batch_id+batch_size),...]
            batch_id = batch_id+batch_size


        x = train_batch_data.reshape(train_batch_data.shape[0], 784)
        yhat = train_batch_label
```

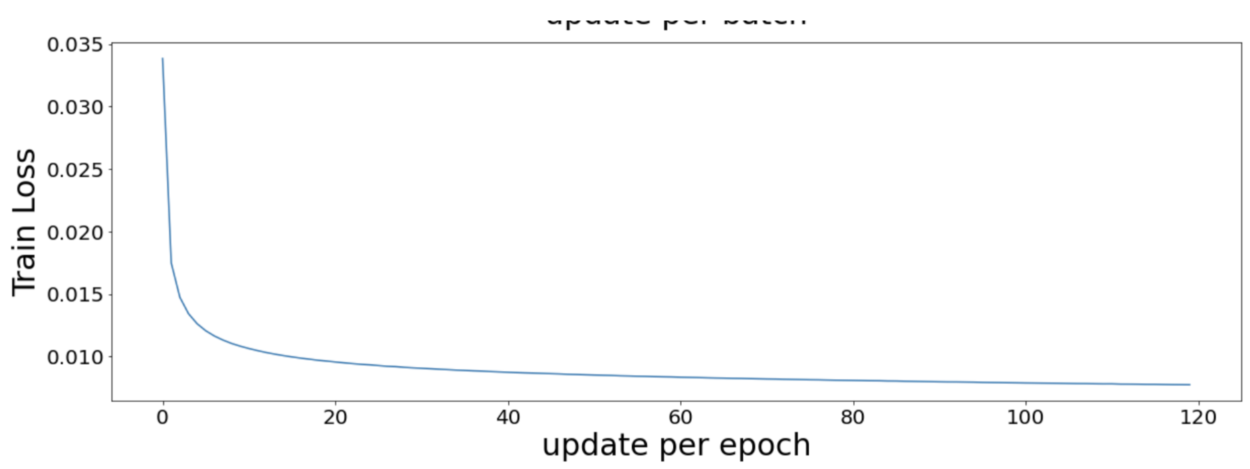
In the above code, for every epoch, ‘data_shuffling’ function is called which takes input and label as input and returns shuffled input and label sets.

Every batch is visited 120 times with the given learning rate and this has helped the network learn the dataset in an optimal manner. The below code describes the forward propagation and backward propagation calculation with weight and biases being updated at the end of training. After parameter updates, zero_grad() method is called to reset the previously calculated gradients.

```
    a1 = layer1.forward(x)
    a2 = layer2.forward(a1)
    pred, loss = layer3.forward(a2, yhat)

    pred_digit = np.argmax(pred, axis=1)
    true_digit = np.argmax(yhat, axis=1)

    train_accuracy += np.count_nonzero(pred_digit==true_digit)

    _train_accuracy_per_batch = np.count_nonzero(pred_digit==true_digit)
    train_accuracy_per_batch.append(_train_accuracy_per_batch/batch_size)

    train_epoch_loss += loss[0]
    train_loss_per_batch.append(loss[0])

    da2 = layer3.backward(yhat)
    da1 = layer2.backward(da2)
    dx1 = layer1.backward(da1)

    dw = layer1.dw
    db = layer1.db

    layer1.w = layer1.w - lr*dw
    layer1.b = layer1.b - lr*db

    layer1.zero_grad()
    layer2.zero_grad()
    layer3.zero_grad()


train_epoch_loss = train_epoch_loss/train_data_shuffle.shape[0]
train_accuracy = train_accuracy/(train_data_shuffle.shape[0])

train_loss_per_epoch.append(train_epoch_loss)
train_accuracy_per_epoch.append(train_accuracy)
```

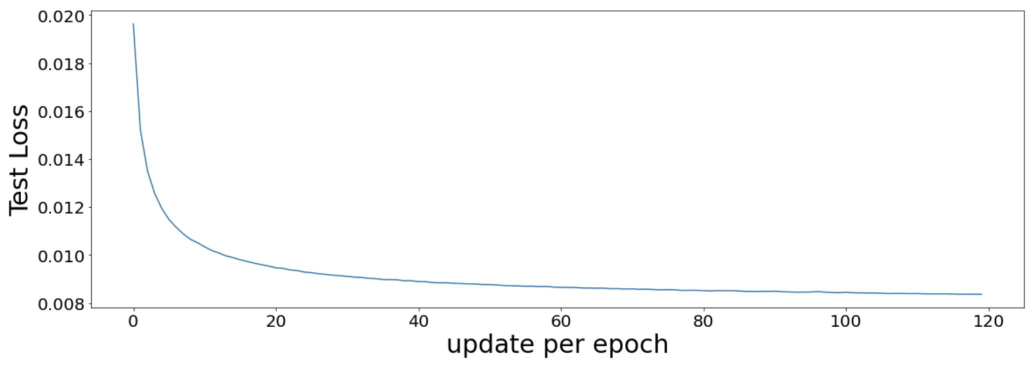


Since we are using the mini-batch gradient descent method, loss optimization is noisy here because the loss calculation and parameter updates are carried out for every batch. If we look at the loss curve and parameter updates on epoch level the curve looks relatively smoother.

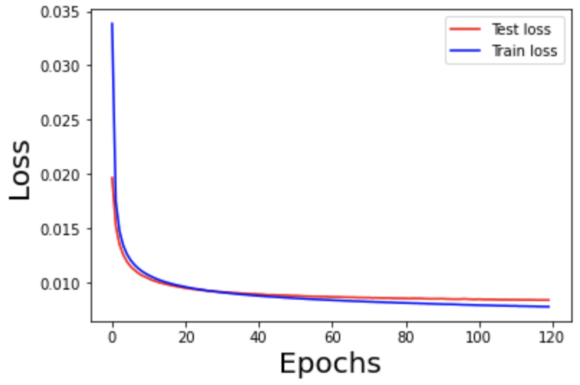


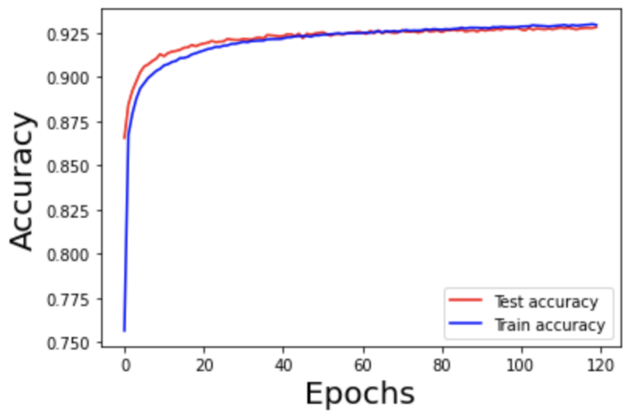
Model is tested on validation dataset in every epoch after weight and biases are updated for every batch.

```
for batch_id in range(0, test_data_np_t1.shape[0], batch_size):
    _test_accuracy_per_batch=0
    if (batch_id + batch_size) > (test_data_np_t1.shape[0]):
        test_batch_data = test_data_np_t1[batch_id:test_data_np_t1.shape[0],...]
        test_batch_label = test_target_np_t2[batch_id:test_data_np_t1.shape[0],...]
        batch_id = test_data_np_t1.shape[0]
    else:
        test_batch_data = test_data_np_t1[batch_id:(batch_id+batch_size),...]
        test_batch_label = test_target_np_t2[batch_id:(batch_id+batch_size),...]
        batch_id = batch_id+batch_size

    xtest = test_batch_data.reshape(test_batch_data.shape[0], 784)
    yhat_test = test_batch_label

    a1out = layer1.forward(xtest)
    a2out = layer2.forward(a1out)
    test_pred, test_loss = layer3.forward(a2out, yhat_test)

    test_pred_digit = np.argmax(test_pred, axis=1)
    test_true_digit = np.argmax(yhat_test, axis=1)


    test_accuracy += np.count_nonzero(test_pred_digit==test_true_digit)

    _test_accuracy_per_batch = np.count_nonzero(test_pred_digit==test_true_digit)


    test_accuracy_per_batch.append(_test_accuracy_per_batch/batch_size)

    test_epoch_loss += test_loss[0]
    test_loss_per_batch.append(test_loss[0])

test_epoch_loss = test_epoch_loss/test_data_np_t1.shape[0]
test_accuracy = test_accuracy/test_data_np_t1.shape[0]

test_loss_per_epoch.append(test_epoch_loss)
test_accuracy_per_epoch.append(test_accuracy)
```

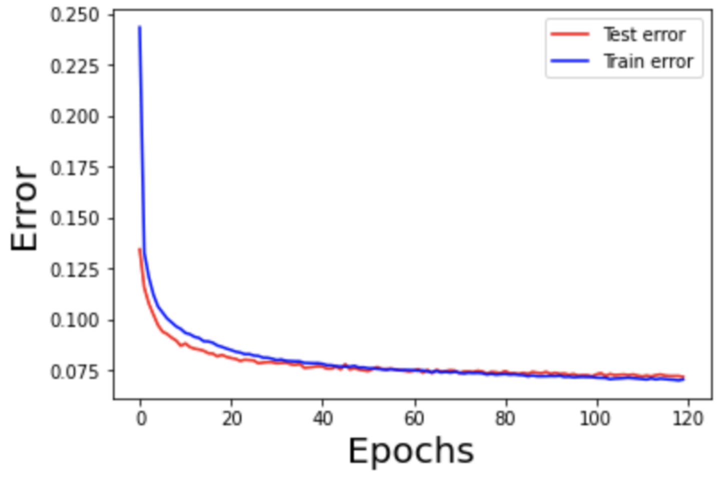


After training the model, I plotted the train and test loss per epoch and train and test accuracy per epoch.





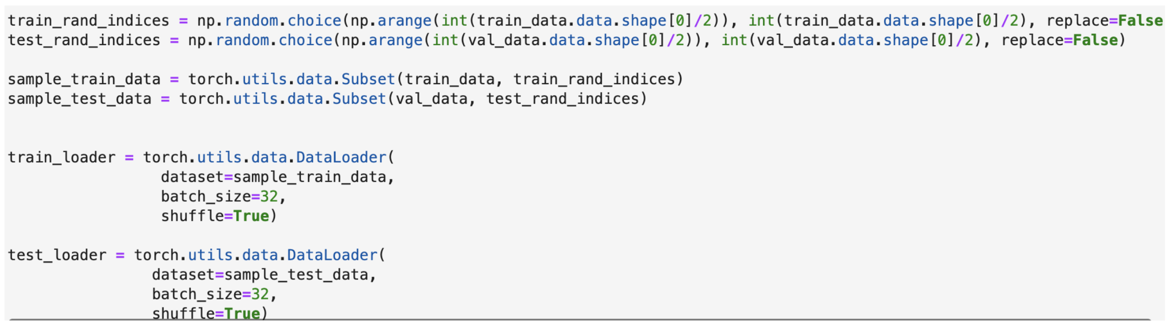
Both train and test loss converge after 120 epochs and we can see that our model performance/accuracy on test instances is also quite good. The final results for the train and validation set are tabulated below in the table.



	Train set	Validation set
Loss	0.0146	0.0152
Accuracy	0.92	0.91
Error	0.08	0.09

Problem 2 - Training Neural Network using PyTorch for MNIST dataset

(a) Using SGD Optimizer

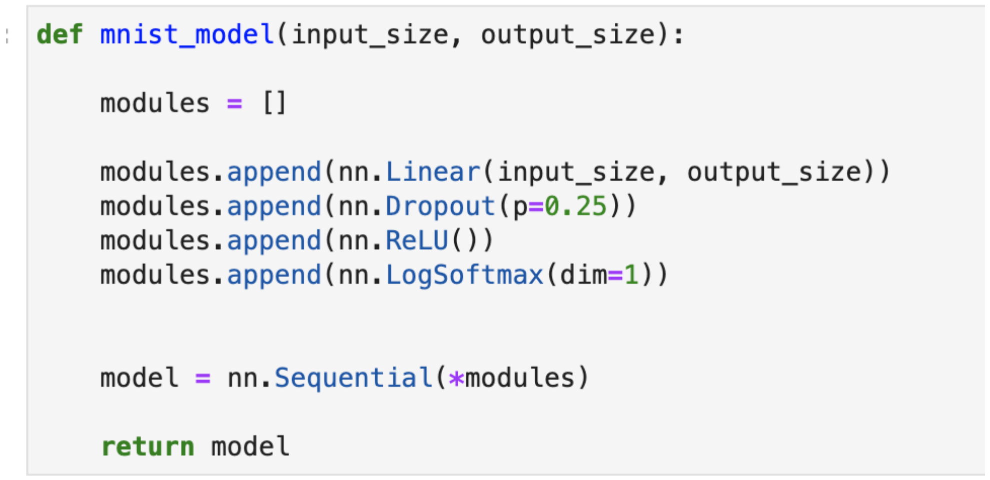


```
train_rand_indices = np.random.choice(np.arange(int(train_data.data.shape[0]/2)), int(train_data.data.shape[0]/2), replace=False
test_rand_indices = np.random.choice(np.arange(int(val_data.data.shape[0]/2)), int(val_data.data.shape[0]/2), replace=False)

sample_train_data = torch.utils.data.Subset(train_data, train_rand_indices)
sample_test_data = torch.utils.data.Subset(val_data, test_rand_indices)


train_loader = torch.utils.data.DataLoader(
                 dataset=sample_train_data,
                 batch_size=32,
                 shuffle=True)

test_loader = torch.utils.data.DataLoader(
                dataset=sample_test_data,
                batch_size=32,
                shuffle=True)
```

Data loading, sampling, and shuffling

Model is defined using nn.Sequential container using PyTorch

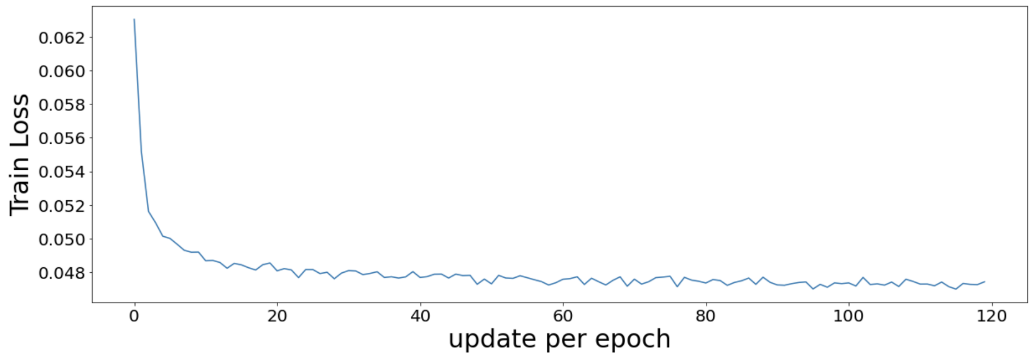
Defining Model

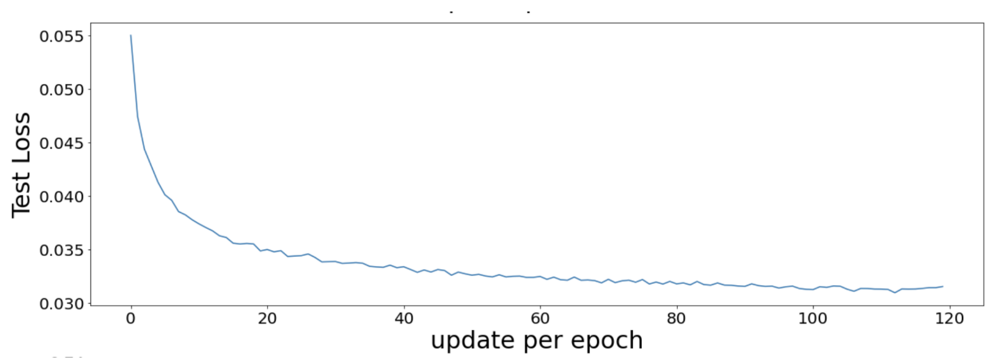


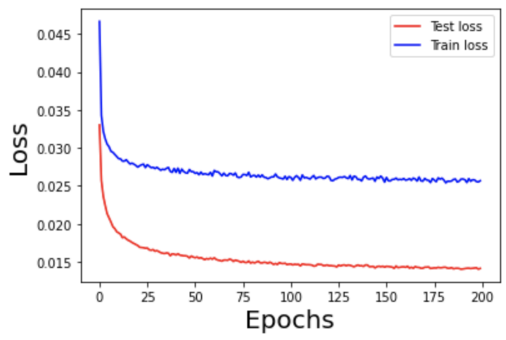
```
def mnist_model(input_size, output_size):

    modules = []

    modules.append(nn.Linear(input_size, output_size))
    modules.append(nn.Dropout(p=0.25))
    modules.append(nn.ReLU())
    modules.append(nn.LogSoftmax(dim=1))


    model = nn.Sequential(*modules)

    return model
```

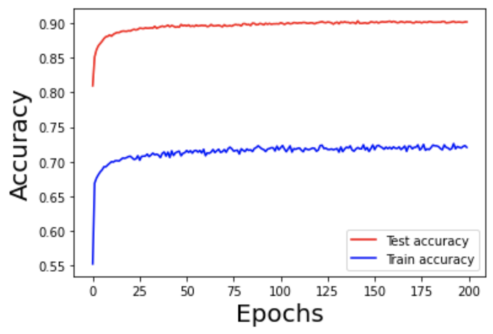
Hyperparameter setting:

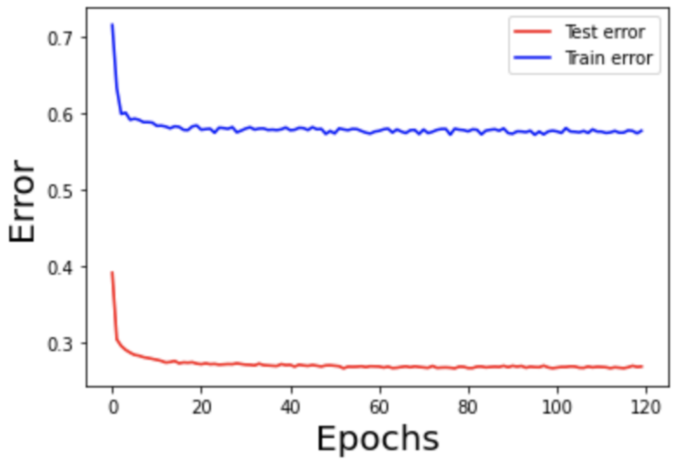
Parameters	Values
Learning Rate	0.01
Epochs	200
Batch size	32











Using the SGD optimizer and drop-out with a rate of 0.25, our train loss and accuracy have suffered because there are only 10 neurons and a single linear layer so deactivating neurons would hurt the network performance during that time. Nonetheless, in the eval mode, model performance is optimal as all the neurons are activated and with optimal parameters, the model should perform better. Hence, we see the weird trend where the test accuracy is higher than train accuracy.

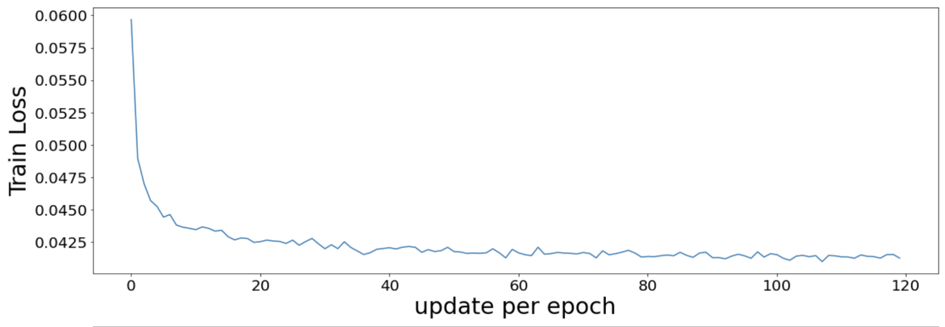
	Train set	**Validation set**
Loss	0.031	0.021
Accuracy	0.635	0.791
Error	0.364	0.208

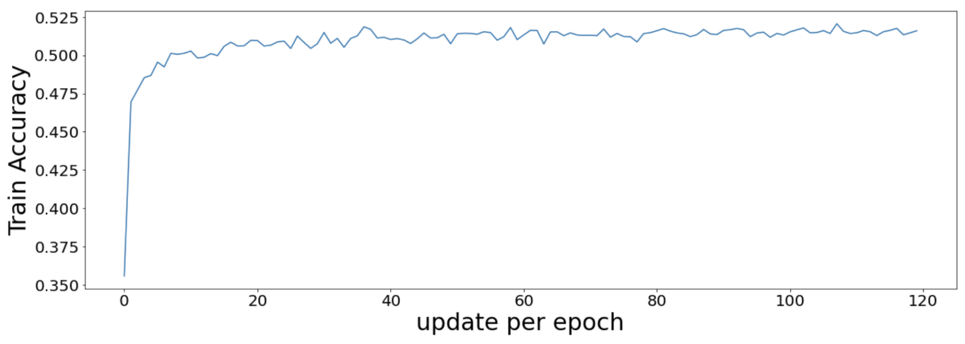
(b) Using Adam Optimizer

In this part, most of the part of the code is the same as the SGD optimization with the PyTorch part except I have changed the optimizer to Adam and tweaked the parameter.

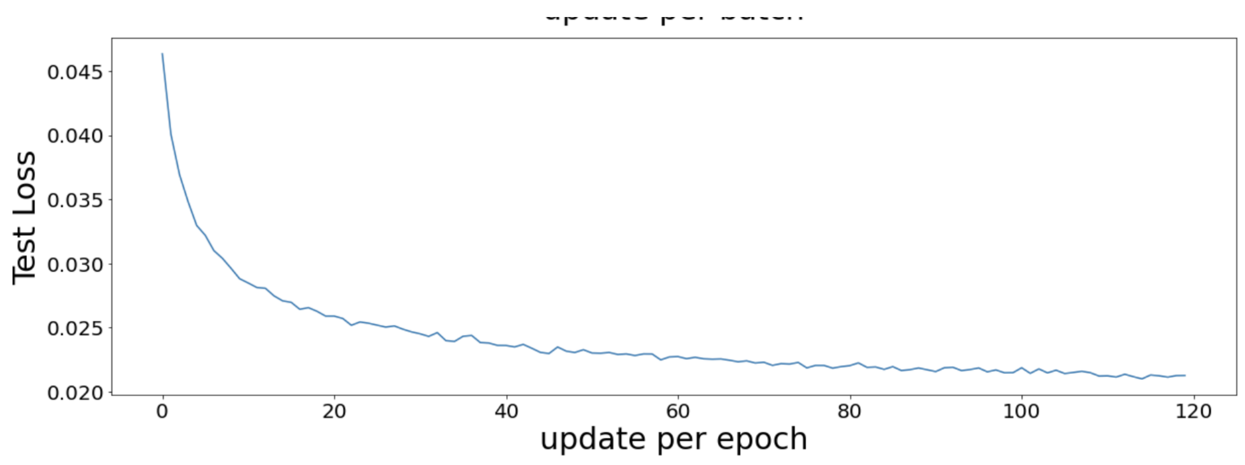
Hyperparameter setting:

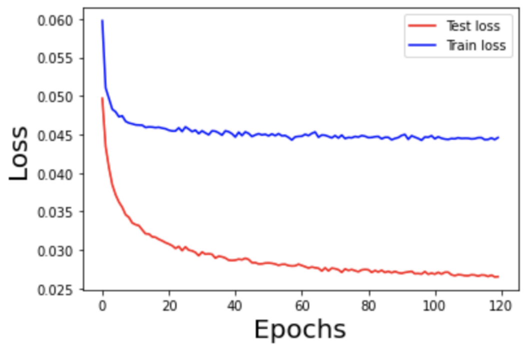
Parameters	**Values**
Learning Rate	0.0001
Epochs	120
Batch size	32

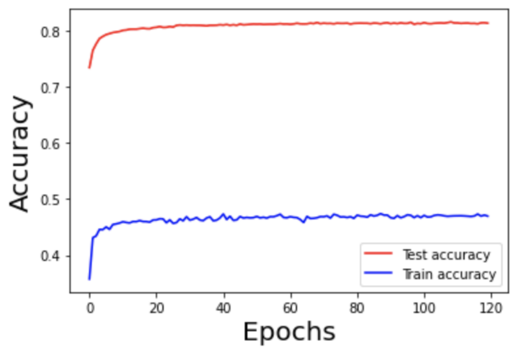


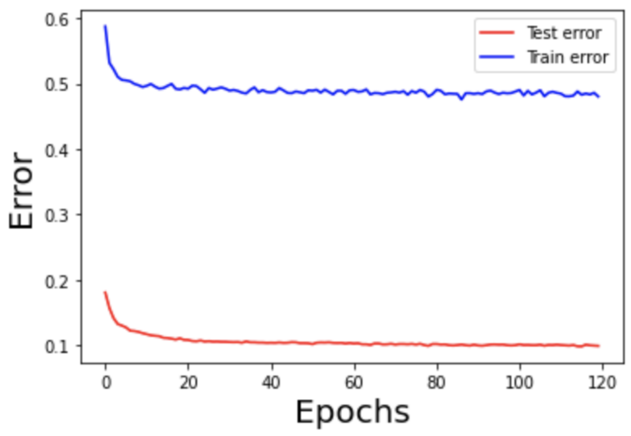


The training accuracy is very low and it seems reasonable because we are using a dropout rate of 50% with just one linear layer architecture which means half of the time a neuron is not working therefore overall its accuracy on the training set would reduce roughly by half. If we do not use a drop out then our accuracy would be boosted by a factor of 2 approximately.







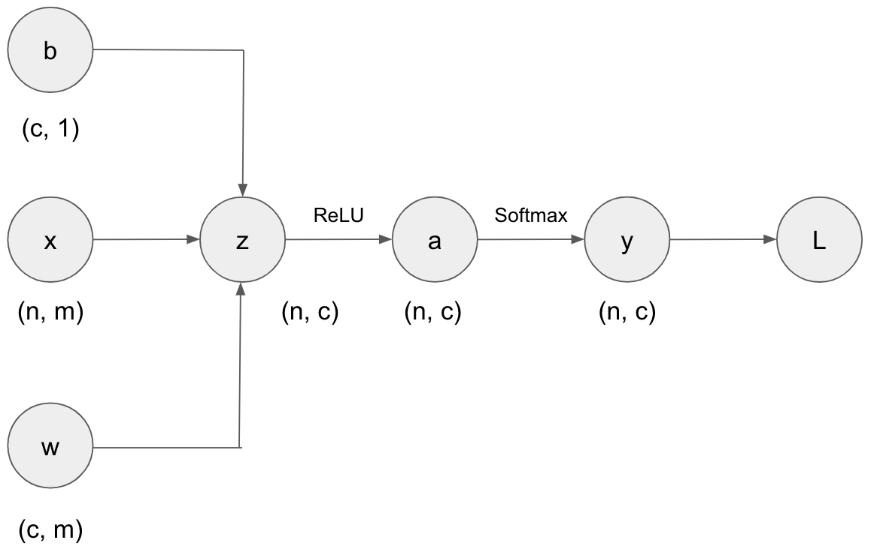


When applying the trained model on the training set, we see that test accuracy is always better than the train and the reason behind this owes to the drop-out layer. When the model has drop-out enabled then our architecture complexity reduces and that's why it underfits on the training set.

	Train set	Validation set
Loss	0.047	0.031
Accuracy	0.42	0.73
Error	0.57	0.26

<u>Appendix: Deriving the Backward Propagation for this architecture</u>

Computation Graph of the architecture for one neuron



Summary of Forward Propagation

$$z = X(w.T) + b$$

$$a = max(0, z)$$

$$y_i = \frac{np.exp(a_i)}{\sum_{i=1}^{10} np.exp(a_i)}$$

$$C = -\sum_{i=1}^{10} \hat{y} * (log(y))$$

Backward Propagation:

$$\frac{dL}{dw} = (\frac{dL}{dy})(\frac{dy}{ds})(\frac{ds}{da})(\frac{da}{dz})X$$

$$\frac{dL}{db} = \left(\frac{dL}{dy}\right)\left(\frac{dy}{ds}\right)\left(\frac{ds}{da}\right)\left(\frac{da}{dz}\right)1$$

For batch size = 1,

$\frac{dL}{dy}$: $[0,\ 0, \ldots \frac{1}{y_r}, \ldots\ 0]$ - derivative of loss with predicted output. Its dimension would be (1,10)

$\frac{dy}{ds} = [\ \ ]$ - derivative of the predicted output with softmax. Its dimension would be (10, 10)

The backpropagation from the softmax and cross-entropy layer is simplified after taking the product between $\frac{dL}{dy}$ and $\frac{dy}{ds}$

$\frac{dL}{ds} = [\,]$ derivative of the predicted output with softmax. Its dimension would be (1, 10)

$\frac{ds}{da} = [\,]$ - derivative of the softmax with relu. Its dimension would be (1, 10)

$\frac{da}{dz} = [\,]$ - derivative of relu with linear layer. Its dimension would be (1, 10)

$\frac{dz}{dw} = X$ - derivative of the linear layer with weights. Its dimension would be (1, 784)

$\frac{dz}{db} = 1$ derivative of the linear layer with weights. Its dimension would be (1, 10)