CS 5814 - Introduction to Deep Learning: Assignment 1

Problem -1: Write a library completely from scratch to train a Neural Network

Input dataset - MNIST

MNIST dataset is a well-known dataset of hand written digits. The training set consist of 60,000 grayscale images of digits each having 28 X 28 pixels. Similarly, the test set of MNIST dataset consist of 10,000 images. The goal here is to predict whether the digit given a handwritten image of a digit between 0-9.

Sampling the data and encoding the class label

Since the training set of the MINST dataset consist of 60,000 records, we are going randomly sample half of the training set for simplicity. We perform this by first converting the MNIST data into numpy format using numpy(), then use np.random.choice() to randomly select indices for sampling, and finally use np.take() to fetch value of those randomly generated indices from the dataset. We also ensure that the sampling is performed without replacement by setting the replace parameter as false.

We perform sampling on both train and test set and then pass these sets to the neural network in a batchwise fashion i.e., mini batches.

```
# to read the data into numpy format
train_np = train.data.numpy()
val_np = val.data.numpy()

# read the corresponding train and target label into numpy format
t_target = train.targets.numpy()
v_target = val.targets.numpy()

# sampling the indices randomly in the range of 0-59999 for the train set and 0-9999 for the test set.
train_sample = np.arange(60000)
test_sample = np.arange(10000)

train_sample_index = np.random.choice(train_sample, 30000, replace=False)
test_sample_index = np.random.choice(test_sample, 5000, replace=False)

sample_train_target = [np.take(train_np,train_sample_index,axis=0), np.take(t_target,
                                                                            train_sample_index,axis=0)]
sample_test_target = [np.take(val_np,test_sample_index,axis=0), np.take(v_target,
                                                                        test_sample_index,axis=0)]
print(len(sample_train_target[1]))
print(len(sample_test_target[0][0]))
```

We perform one hot encoding on the train and test labels to split the categorical variable into multiple binary variables.

```
# performing One hot encoding on the train lables
one_hot=np.zeros((train_target.size, 10))
one_hot[np.arange(train_target.size), train_target]=1

# performing One hot encoding on the test lables
one_hot_test=np.zeros((test_target.size, 10))
one_hot_test[np.arange(test_target.size), test_target]=1
```

Parameters

Learning rate (Eta): 0.01

The learning rate is an hyperparameter which controls how much the model should change in response to the error each time when the model weights are updated. In this problem, the learning rate is set to 0.01

Weights and Bias:

Weights are biases are the learnable parameter of a neural network. Weights w and bias b of the model is initialized based on the gaussian space with their values initialized using L2 norm. Initially, the weight metrics will be 784 X 10 (number of features and number of classes)

```
def __init__ ( self ) :
    self.w=np.random.normal(0,1, size=(784,10)) # number of classes=784,class lables=10
    self.b=np.random.normal(0,1, size=(1,10)) # bias for each neuron so it's deminesion is 1x10

    # performing L2 norm on w and b using linalg
    norm_w2=1/(np.linalg.norm(self.w,2))
    norm_b2=1/(np.linalg.norm(self.b,2))

    self.w = (norm_w2)*(self.w)
    self.b = (norm_b2)*(self.b)
```

Batch Size: 32

Since the batch size is 32, there will 32 training examples in one forward/backward pass of the neural network. In each epoch iteration, there will be 937 batch iteration and the total number of batch iteration will be 937*number of epochs.

Epoch: 240

The number of epochs determine how many times should the model run through the entire dataset. Since we have taken number of epochs as 240, we'll train all the images 240 times and the total number of forward passes will be 240 * 30000.

Model Architecture:

As the model architecture has a total of only three layers, the model is quite simple and relatively straightforward. The first layer after the input layer is the fully connected Linear Layer which has 10 neurons, a ReLU activation layer, and lastly a SoftMax layer.

Linear Layer:
The input, Z, to the Linear layer would be dot product of weights and the inputs combined with each neuron's bias. Since the dimensions of the input after flattening will be N X 784 (where N is the Batch size, which is 32 in this case) and the dimensions of the weights are 784 X 10, the result of their dot product will a matrix of dimension of 32 X 10. The final input to the Linear Layer would be, $Z = W.X + B$.
As discussed, the dimensions of *Z* would be 32 X 10 for the batch size of 32.

```
class LinearLayer_t :
    def __init__ ( self ) :
        self.w=np.random.normal(0,1, size=(784,10)) # number of classes=784,class lables=10
        self.b=np.random.normal(0,1, size=(1,10)) # bias for each neuron so it's deminesion is 1x10

        # performing L2 norm on w and b using linalg
        norm_w2=1/(np.linalg.norm(self.w,2))
        norm_b2=1/(np.linalg.norm(self.b,2))

        self.w = (norm_w2)*(self.w)
        self.b = (norm_b2)*(self.b)

    def forward(self , a_prev):
        # performing linear trasformation i.e z=wx+b
        a_curr = np.dot(a_prev,self.w)+self.b
        self.a_prev = a_prev
        return a_curr

    def backward(self , da):
        da_prev=da.dot((self.w).T)
        self.dw=((self.a_prev).T).dot(da)
        self.db=np.sum(da,axis=0,keepdims=True)/batch_size
        return da_prev

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

ReLU Layer:
The ReLU layer is in charge of converting the weighted sum of inputs into an output across all nodes. Because the ReLU function is max(0,Z), it suppresses all negative input values by mapping them to zero and only deals with positive input values.. Since we are not performing any dot products here, the dimensions of the output values would be same as its input values, i.e it will be 32 X 10 for batch size of 32.

```
class Relu_t:
    def __init__(self):
        pass

    def forward(self, a):
        # forward function of ReLU= max(0,a)
        b=np.maximum(0,a)
        self.a=b
        return b

    def backward(self,dz):
        self.a[self.a<=0]=0
        self.a[self.a>0]=1
        self.dz=dz*self.a
        return self.dz

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

SoftMax Layer:

The SoftMax layer, which is the output layer, is the architecture's final layer. This layer transforms a vector of K real values into a vector of K real values that add to 1. It compresses the input, which contains values ranging from less than zero to larger than zero, into values between 0 and 1 in order to understand them as probabilities. Each image will have 10 predicted values and the index which contains maximum probability indicates the class label of that input image. The SoftMax function is described below

$$y_{pred} = \frac{e^z}{\sum e^z}$$

The order of the output of this layer would be same as that of ReLU i.e., 32 X 10 per batch.

```python
class SoftmaxCrossEntropy_t:
    def __init__(self):
        pass

    def forward(self,z,yhat):
        self.yhat=yhat
        sum_exp=np.sum(np.exp(z),axis=1,keepdims=True)
        # calculating SoftMax(Z)
        y_pred = np.exp(z)/ sum_exp
        # calculating the cross entropy loss
        loss = -np.sum(yhat*np.log(y_pred),axis=1,keepdims=True)
        self.y_pred = y_pred
        n=np.shape(loss)[0]
        # calculating average loss
        avg_loss=np.sum(loss,axis=0,keepdims=True)/(n)
        return y_pred, avg_loss

    def backward(self):
        self.dz = self.y_pred - self.yhat
        return self.dz

    def zero_grad(self):
        self.dz=np.zeros_like(self.dz)
```

Cross Entropy Loss:

The measure of differences between two probability distributions for a given random variable or sequence of events is known as cross entropy loss. Since our SoftMax layer gives probability distribution of the prediction of the input digits, we can use cross entropy loss to calculate the difference in the probabilities of y_{pred} and the ground truth. The cross-entropy loss is calculated using the below formula

$$Loss = -\sum_{i=1}^{n} y_{hat} * log(y_{pred})$$

Backpropagation:

Backpropagation is used to understand the deviation of the predicted values from the ground truth, i.e it is about understanding how changing the weights and biases would change the cost function. Here we calculate gradient of the loss with respect to all the weights and biases in the neural network using the chain rule.

$$\frac{\partial C}{\partial W} = \frac{\partial C}{\partial y} * \frac{\partial y}{\partial z} * \frac{\partial z}{\partial z'} * \frac{\partial z'}{\partial W}$$

$$\frac{\partial C}{\partial B} = \frac{\partial C}{\partial y} * \frac{\partial y}{\partial z} * \frac{\partial z}{\partial z'} * \frac{\partial z'}{\partial B}$$

After identifying the error in the weights and bias, we update both the weights and biases for each batch. We perform update these by multiplying the learning rate with $\frac{\partial C}{\partial W}$ and then subtracting it from the wight W and bias B.

$$W = W - Eta * \frac{\partial C}{\partial W}$$

$$B = B - Eta * \frac{\partial C}{\partial B}$$

Model Input and Output:

Input:

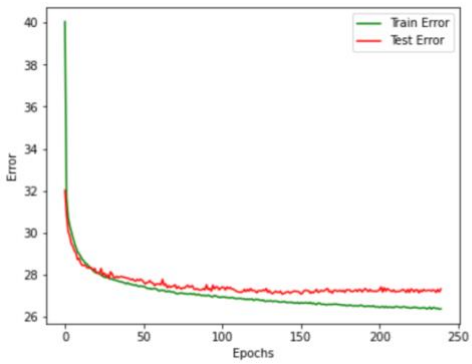
As mentioned previously, the input to the model would be grayscale handwritten images whose resolution would be 28 X 28. These images are flattened into 1 X 784 vector which is then passed as in input to the linear layer.

Output:

The maximum accuracy achieved on the training set is **93.44%** and the final training accuracy was **93.37%.** Similarly, for the validation set, maximum accuracy was 92.66% and the final accuracy was 92.5%.

Plots and Analysis:

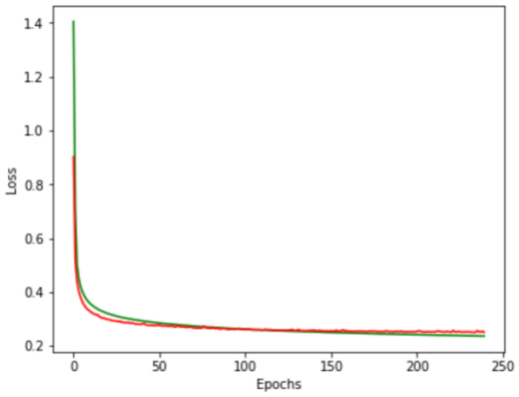
Comparison between Training Error and Validation Error



As seen in the above plot, the both training and the validation error decreases as we iterate through the epochs until a certain point. We can see that after few epoch iterations, the validation error and training error does not decrease much as compared to error in the initial epochs. This is because as the number of epochs increases, the models learns more and trains better, thus reduces the error.

The above plot is plotted with the parameter setting as follows:
Epochs=240, Batch size=32, and Learning rate=0.01

Comparison between Training loss and Validation loss



When we compare the training loss with the validation loss, we observe that in the very initial iterations, the training loss is high but after a few epoch iterations, it decreases drastically as the model tries to learn and tries to reduce the loss. The validation loss starts from the half way as compared to training loss, this is because our models has learned and has generalized well. We see that the training loss is slightly better than validation loss. The above plot is plotted with the parameter setting as follows:
Epochs=240, Batch size=32, and Learning rate=0.01

Problem -2: Implement Neural Network using Pytorch

Loading the dataset:

We first load the module with the help of torchvision's transforms which helps in image transformations. These image transformations can be chained using Compose method. The code snippet through for loading the data is given below.

```
transform=transforms.Compose([transforms.ToTensor()])

train_data = thv.datasets.MNIST('./data' , transform=transform, download=True , train=True)
val_data = thv.datasets.MNIST('./data', transform=transform, download=True, train=False)
```

Sampling the dataset:

We first pull up random 30000 indices from the train set and 5000 indices from the validation set as we need 50% of train and test data. We this with the help of python's random.sample() method which takes the returns a length of items (indexes in our case) chose from the given sequence. We then take the values represented by those indices from the respective training and validation sets using data.Subset method.
After creating the subset for train and validation set, we shuffle the data within those sets using data.DataLoader. The code the sampling of the dataset is given below.

```
eta=0.01
batch_size=32

train_index=[]
val_index=[]

for i in range(0,60000):
    train_index.append(i)
for i in range(0,10000):
    val_index.append(i)

train_subset=torch.utils.data.Subset(train_data, random.sample(train_index,30000))
val_subset=torch.utils.data.Subset(val_data, random.sample(val_index,5000))

tloader=torch.utils.data.DataLoader(train_subset,batch_size=batch_size,shuffle=True)
vloader=torch.utils.data.DataLoader(val_subset,batch_size=batch_size,shuffle=True)
```

Model training using Pytorch:

As the scope of problem 2 is similar to that of problem 1, we will using be the same number of layers namely Linear, ReLU, and SoftMax. However, dispute the layers being the same, their

implementation differs from that of problem 1. The first linear layer, performs linear transformation on the input data: $Z = W.X + B$. This is performed using nn.Linear() function which takes two parameters: number of feature and number of classes.

Similarly, the next two layers are ReLU and SoftMax layer which are implemented using nn.ReLU() and nn.LogSoftmax(). The nn.LogSoftmax() applies the log of SoftMax(X) on the input to calculate the probabilities.

```
class NeuralNetwork(nn.Module):
    def __init__(self, num_features, num_classes, dropout):
        super(NeuralNetwork,self).__init__()
        self.linear = nn.Linear(num_features, num_classes)
        self.dropout = nn.Dropout(dropout)
        self.relu = nn.ReLU()
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self,x):
        x = self.linear(x)
        x = self.relu(x)
        x = self.softmax(x)
        return x
```

We also use the negative log likelihood loss function i.e nn.NLLLoss() which is usually used with SoftMax.

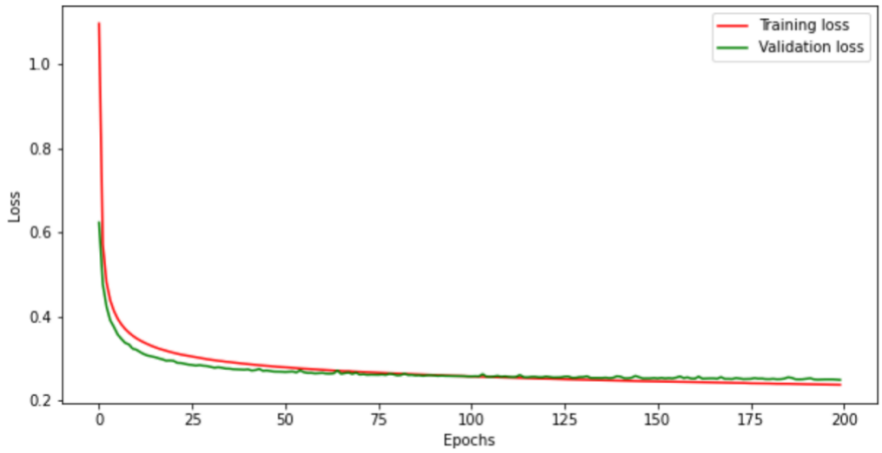
1) SGD optimizer with dropout=0.25

Unlike batch gradient descent, stochastic gradient descent updates the parameters for each training sample. By executing one update at a time, SGD eliminates redundancy. As a result, it is typically significantly faster. SGD makes frequent, high-variance updates, causing the objective function to vary a lot.

Dropout is a regularization approach for approximating concurrent training of a large number of neural networks with varied architectures. We perform dropout using nn.Dropout() which uses samples from a Bernoulli distribution, zeroes some of the elements of the input tensor at random with probability p during training. Here our probability is 0.25.

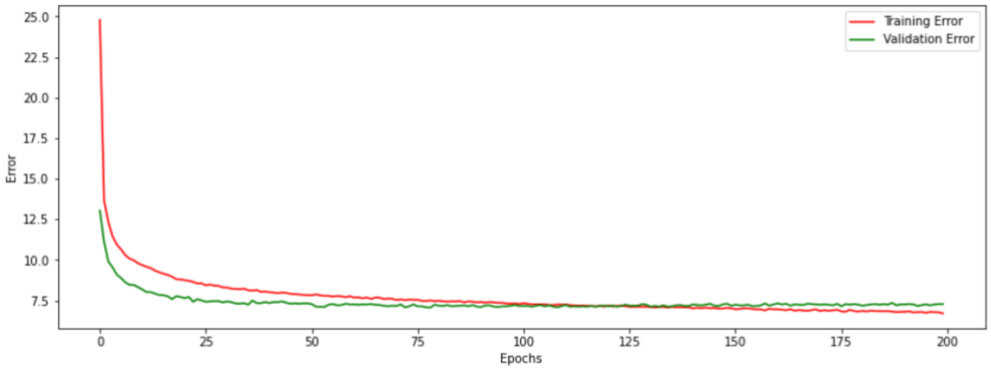
Accuracy of the model:
Maximum Training accuracy: **93.30%**
Maximum Validation accuracy: **92.94%**
Final Training accuracy: **93.30%**
Final Validation accuracy: **92.72%**

Training loss and Validation loss using SGD optimizer:



When we compare the training loss to the validation loss, we see that the training loss is significant in the early iterations but drops dramatically after a few epoch iterations as the model tries to learn and lower the loss. When compared to training loss, validation loss begins halfway through. This is because our models have learned and generalized well. The validation loss gets slightly better during the initial epoch iterations. In the final epoch iteration, the training loss is slightly better than the validation loss, as can be shown. The above plot is plotted with the parameter setting as follows:
Epochs=200, Batch size=32, and Learning rate=0.01

Training error and Validation error using SGD optimizer:



Similar to that of problem 1, as we progress through the epochs, both the training and validation errors reduce until we reach a particular threshold. We can see that the validation and training error do not diminish significantly after a few epoch iterations when compared to the error in the initial epochs. This is due to the fact that as the number of epochs grows, the models learn more and train better, lowering the error. The validation error is noticeably

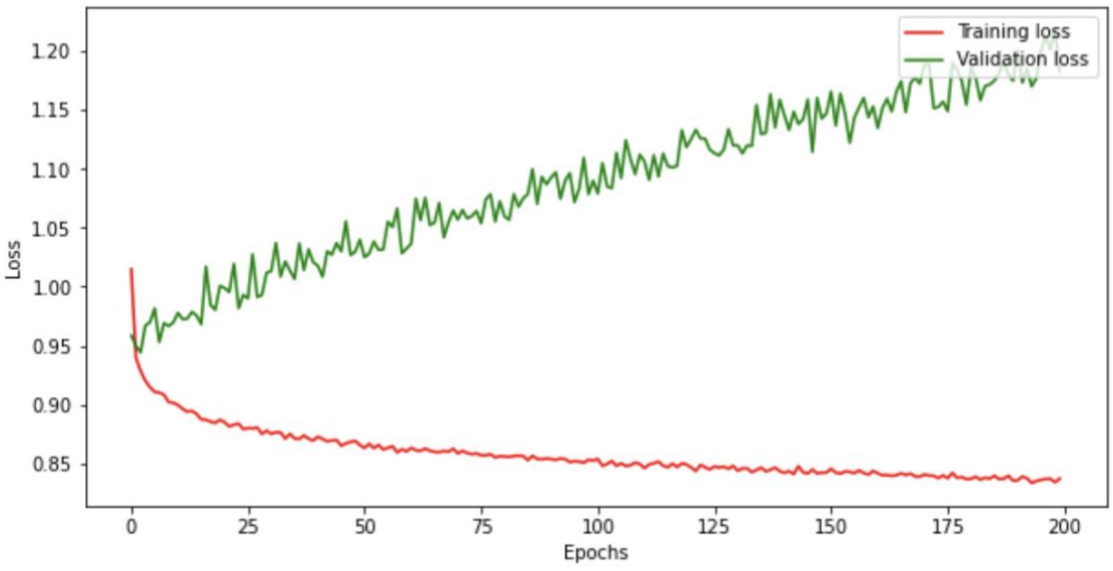
better during the initial epoch iterations but in the final epoch iteration, the training error is slightly better than the validation error. The above plot is plotted with the parameter setting as follows:
Epochs=200, Batch size=32, and Learning rate=0.01

2) Adam optimizer with dropout=0.50

Adam is a stochastic gradient descent extension that combines the benefits of two earlier extensions, Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp). AdaGrad has a high per-parameter learning rate, which helps it perform better on problems with sparse gradients. RMSProp also keeps per-parameter learning rates that are adjusted based on the average of recent gradient magnitudes for the weight.

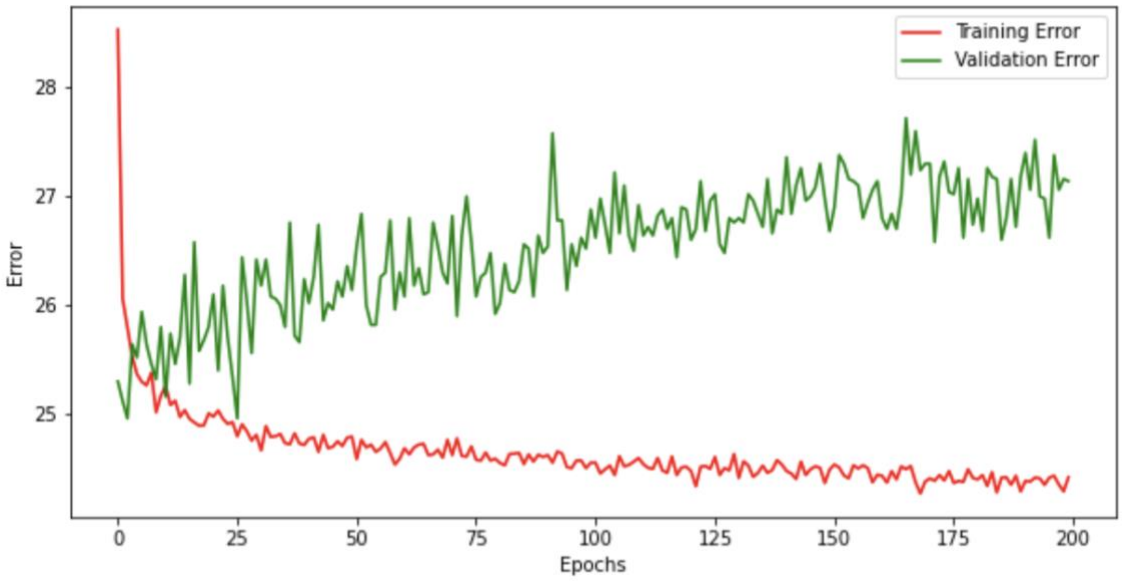
Accuracy of the model:
Maximum Training accuracy: **75.73%**
Maximum Validation accuracy: **75.03%**
Final Training accuracy: **75.58%**
Final Validation accuracy: **75.03%**

Training loss and Validation loss using Adam optimizer:



Validation loss of Adam is poor compared to its training loss because the model is getting stuck in the local minima and also because it has higher dropout value. The performance of Adam's optimization could have been better if the dropout value would have been less. Oscillating values of validation loss indicates that the models are getting stuck in the local minima

Training error and Validation error using Adam optimizer:



Adam's validation error is lower than its training error because the model gets stuck in local minima and has a greater dropout value. Adam's optimization could have performed better if the dropout value had been lower. Oscillating values of validation loss indicates that the models are getting stuck in the local minima