Introduction:

A Neural Network library has been built which achieves a similar functionality of the Pytorch library using the python NumPy module. NumPy is an efficient Python library used for working with arrays/matrix operations.

We are solving a classification problem using a very basic neural network architecture. The Neural Network architecture is traversed in two stages- 1) forward pass and 2) Back propagation. The classification task is performed by solving a Minimization optimization problem where the goal is to minimize the loss function which is calculated during the forward pass. The loss function value which is obtained in the forward pass is minimized in the backward pass/ back propagation which adjusts the weights and bias values in the connected neural network layers therefore achieving better classification results.

we solve the classification problem using two approaches 1) using a custom neural network library built 2) Pytorch predefined neural network library.

Using a custom neural network library built:

Inputs and parameters:

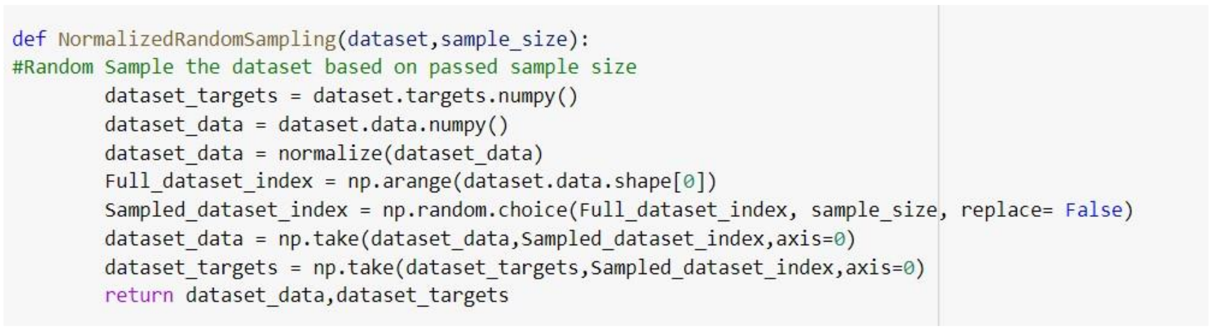
MNIST Input dataset – MNIST Dataset is a benchmark dataset used in image recognition tasks. The MINST training dataset consists of 60,000 28×28-pixel grayscale images of handwritten single digits between 0 and 9. We are basically solving a 10-class problem. We are going to random sample 50% of the training data for simpler complexity of the problem.

Similarly for the validation dataset we are going to take MNIST Testing dataset consisting of 10000 images of 28×28-pixel grayscale images of handwritten single digits between 0 and 9. We are going to random sample 50% of the testing data for simpler complexity of the problem.

The 28×28-pixel values of each image are flattened as 784 features and sent as input to the Neural Network architecture to classify as one of the 10 classes (0 to 9).

The inputs images in the sampled training and testing datasets are taken batch wise and processed in this architecture design. We called the batches as mini batches.

The class labels are one hot encoded where each class label has a unique one hot encoded value.

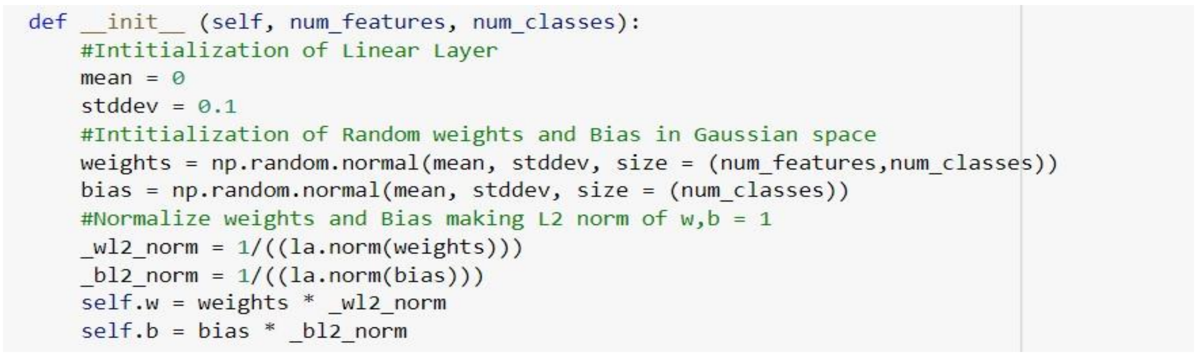


```
def NormalizedRandomSampling(dataset,sample_size):
#Random Sample the dataset based on passed sample size
        dataset_targets = dataset.targets.numpy()
        dataset_data = dataset.data.numpy()
        dataset_data = normalize(dataset_data)
        Full_dataset_index = np.arange(dataset.data.shape[0])
        Sampled_dataset_index = np.random.choice(Full_dataset_index, sample_size, replace= False)
        dataset_data = np.take(dataset_data,Sampled_dataset_index,axis=0)
        dataset_targets = np.take(dataset_targets,Sampled_dataset_index,axis=0)
        return dataset_data,dataset_targets
```

Parameter Initializations:

Weight / Bias Initialization:

The Weights and bias values are initialized based on the Gaussian space entries 0 to 1 and the entries of w and b are normalized using L2- Norm so the squared Euclidean of these values would be equal to 1.



```
def __init__ (self, num_features, num_classes):
    #Intitialization of Linear Layer
    mean = 0
    stddev = 0.1
    #Intitialization of Random weights and Bias in Gaussian space
    weights = np.random.normal(mean, stddev, size = (num_features,num_classes))
    bias = np.random.normal(mean, stddev, size = (num_classes))
    #Normalize weights and Bias making L2 norm of w,b = 1
    _wl2_norm = 1/((la.norm(weights)))
    _bl2_norm = 1/((la.norm(bias)))
    self.w = weights * _wl2_norm
    self.b = bias * _bl2_norm
```

Epochs: 200

We will run 200 epoch iterations, where we will train the model with all the training images 200 times. Total forward passes we will have through the network would be 30000 * 200 .

Batch size : 32

We have taken batch size as the 32 and processed 32 images in each batch iteration. For each epoch iteration we will have 30000/32 = 938 batch iterations. Total batch iterations will be 938 * 200 iterations.

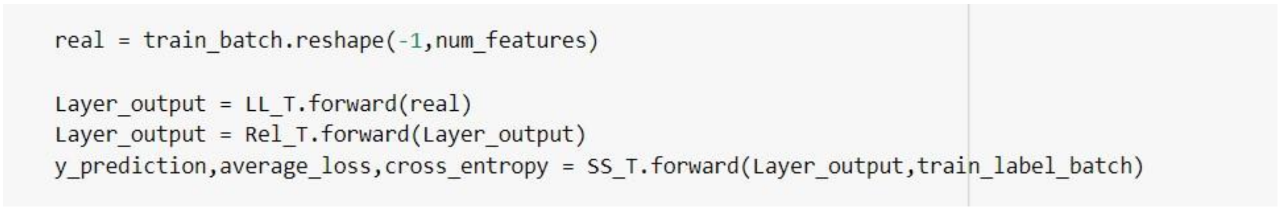
Learning Rate: 0.01

We are using Mini Batch gradient descent to update weights and bias.

Model architecture:

The Model consists of an input layer, Linear Fully connected Layer, Relu activation layer, and finally SoftMax layer. A Cross entropy loss is calculated using the SoftMax Layer and Ground truth label. We minimize the Cross-entropy loss using back propagation that is mini-batch stochastic gradient descent and parameter updates for a specific learning rate which will be discussed in the next section.

The matrix orders discussed below will be specific to one image but however, the code will be executed for a batch of images.



```
real = train_batch.reshape(-1,num_features)

Layer_output = LL_T.forward(real)
Layer_output = Rel_T.forward(Layer_output)
y_prediction,average_loss,cross_entropy = SS_T.forward(Layer_output,train_label_batch)
```

Input Layer:

First Layer is input layer which contains 784 feature inputs of each image. The grayscale image of resolution 28*28 is flattened to be a feature array of size 784 and this feature array is the input layer and it is passed as input to next layers.

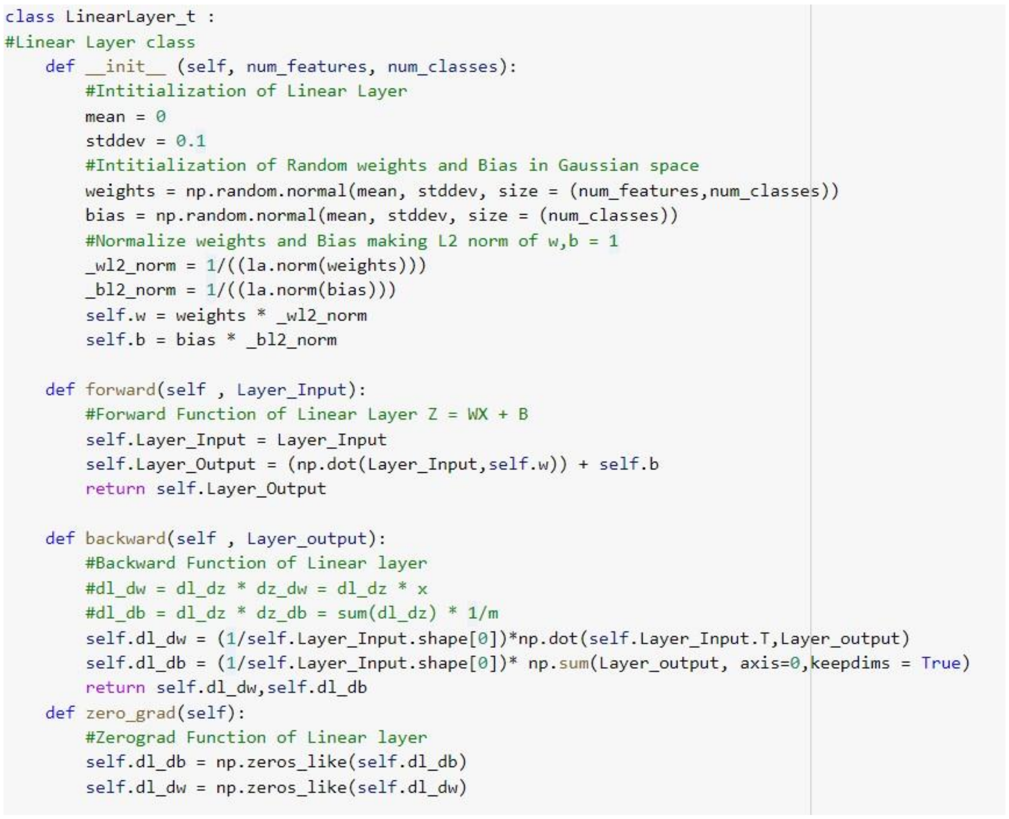
The 784 input layers are fully connected a linear layer of size 10.

Fully connected Linear Layer:

The number of connections between the input layer and the 10 neurons is 784 * 10. The number of weight parameters will same 784 * 10. Additionally, the 10 neurons will have 10 bias values added.

The final output of the Linear layer will be calculated as $Z' = W^T. X + B$ The

matrix order of Z' is 1 * 10 for one image.



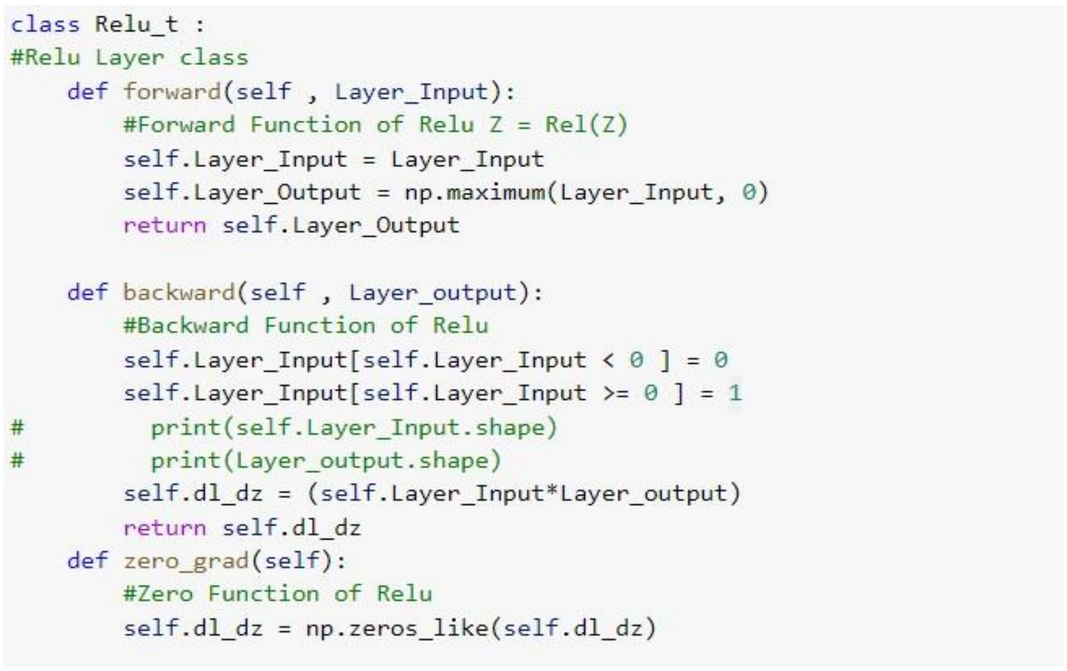
```
class LinearLayer_t :
#Linear Layer class
    def __init__ (self, num_features, num_classes):
        #Intitialization of Linear Layer
        mean = 0
        stddev = 0.1
        #Intitialization of Random weights and Bias in Gaussian space
        weights = np.random.normal(mean, stddev, size = (num_features,num_classes))
        bias = np.random.normal(mean, stddev, size = (num_classes))
        #Normalize weights and Bias making L2 norm of w,b = 1
        _wl2_norm = 1/((la.norm(weights)))
        _bl2_norm = 1/((la.norm(bias)))
        self.w = weights * _wl2_norm
        self.b = bias * _bl2_norm

    def forward(self , Layer_Input):
        #Forward Function of Linear Layer Z = WX + B
        self.Layer_Input = Layer_Input
        self.Layer_Output = (np.dot(Layer_Input,self.w)) + self.b
        return self.Layer_Output

    def backward(self , Layer_output):
        #Backward Function of Linear layer
        #dl_dw = dl_dz * dz_dw = dl_dz * x
        #dl_db = dl_dz * dz_db = sum(dl_dz) * 1/m
        self.dl_dw = (1/self.Layer_Input.shape[0])*np.dot(self.Layer_Input.T,Layer_output)
        self.dl_db = (1/self.Layer_Input.shape[0])* np.sum(Layer_output, axis=0,keepdims = True)
        return self.dl_dw,self.dl_db
    def zero_grad(self):
        #Zerograd Function of Linear layer
        self.dl_db = np.zeros_like(self.dl_db)
        self.dl_dw = np.zeros_like(self.dl_dw)
```

Relu Layer:

Relu layer is the activation function Z = max(Z',0). This activation function supresses all the negative values and initializes zero and retains the positive values. The matrix order of the Relu function will be 1*10 for one image.



```
class Relu_t :
#Relu Layer class
    def forward(self , Layer_Input):
        #Forward Function of Relu Z = Rel(Z)
        self.Layer_Input = Layer_Input
        self.Layer_Output = np.maximum(Layer_Input, 0)
        return self.Layer_Output

    def backward(self , Layer_output):
        #Backward Function of Relu
        self.Layer_Input[self.Layer_Input < 0 ] = 0
        self.Layer_Input[self.Layer_Input >= 0 ] = 1
#          print(self.Layer_Input.shape)
#          print(Layer_output.shape)
        self.dl_dz = (self.Layer_Input*Layer_output)
        return self.dl_dz
    def zero_grad(self):
        #Zero Function of Relu
        self.dl_dz = np.zeros_like(self.dl_dz)
```

SoftMax Layer:

The SoftMax layer is the final layer which computes the probability of membership for each class. This function converts a vector of numbers into a vector of probabilities, where the probabilities of each value are proportional to the relative scale of each value in the vector. The matrix order of the SoftMax function will be 1*10 for one image.

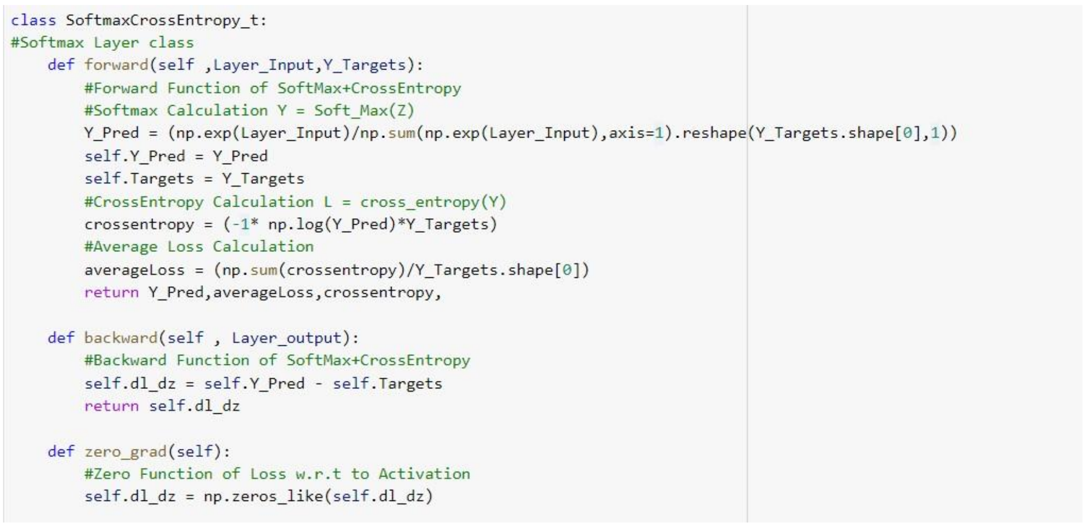
The index which contains the maximum probability value will be the class label of the input image. Each image will have 10 prediction values.

$$y_{prediction} = \frac{ez}{\sum e^{Z}}$$

Cross Entropy Loss:

Cross-entropy is a measure of the difference between two probability distributions for a given random variable or set of events. In this model we calculate this cross entropy as the difference between the SoftMax which is intuitively the probability distribution of the prediction of the input image and the ground truth label of the input image. The matrix order of the Relu function will be 1*10 for one image.

$$CE\ Loss = -\sum_{i=1}^{n} y_{ground\_truth} * Log(y_{prediction})$$



```
class SoftmaxCrossEntropy_t:
#Softmax Layer class
    def forward(self ,Layer_Input,Y_Targets):
        #Forward Function of SoftMax+CrossEntropy
        #Softmax Calculation Y = Soft_Max(Z)
        Y_Pred = (np.exp(Layer_Input)/np.sum(np.exp(Layer_Input),axis=1).reshape(Y_Targets.shape[0],1))
        self.Y_Pred = Y_Pred
        self.Targets = Y_Targets
        #CrossEntropy Calculation L = cross_entropy(Y)
        crossentropy = (-1* np.log(Y_Pred)*Y_Targets)
        #Average Loss Calculation
        averageLoss = (np.sum(crossentropy)/Y_Targets.shape[0])
        return Y_Pred,averageLoss,crossentropy,

    def backward(self , Layer_output):
        #Backward Function of SoftMax+CrossEntropy
        self.dl_dz = self.Y_Pred - self.Targets
        return self.dl_dz

    def zero_grad(self):
        #Zero Function of Loss w.r.t to Activation
        self.dl_dz = np.zeros_like(self.dl_dz)
```

The Goal is to minimize the cross-entropy loss which is achieved using the back propagation.

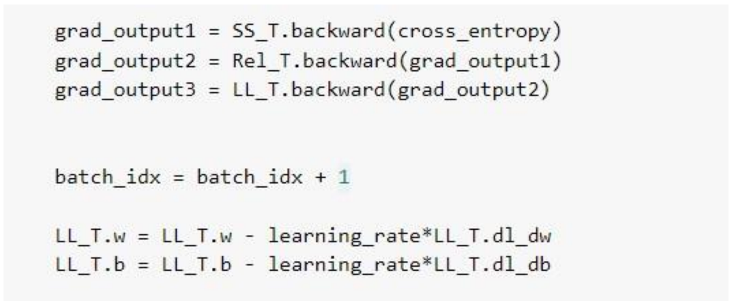
Backward Pass/ Backpropagation:

To optimize loss value and improve the classification accuracy, we need to compute the change in loss with respect to change in weights and bias of the Network. This is also called as gradients.

To compute the gradient of Loss with respect to weights and bias, we can use the chain rule.

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y} * \frac{\partial y}{\partial Z} * \frac{\partial Z}{\partial Z}{}' * \frac{\partial Z'}{\partial W}$$

$$\frac{\partial L}{\partial B} = \frac{\partial L}{\partial Y} * \frac{\partial y}{\partial Z} * \frac{\partial Z}{\partial Z}{}' * \frac{\partial Z'}{\partial B}$$



```
grad_output1 = SS_T.backward(cross_entropy)
grad_output2 = Rel_T.backward(grad_output1)
grad_output3 = LL_T.backward(grad_output2)


batch_idx = batch_idx + 1

LL_T.w = LL_T.w - learning_rate*LL_T.dl_dw
LL_T.b = LL_T.b - learning_rate*LL_T.dl_db
```

$$\mathbf{W} = \mathbf{W} - \textbf{learning rate} * \frac{\partial L}{\partial W}$$

$$\mathbf{B} = \mathbf{B} - \textbf{learning rate} * \frac{\partial L}{\partial B}$$

Model Output:

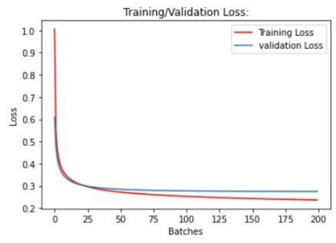
The Approach for training and validation is to Train all the batches and then validate all the Test Batches in each epoch iteration.

Final Training and Validation Accuracy reported

- Training Accuracy: 93.33%
- Validation Accuracy: 92.02%

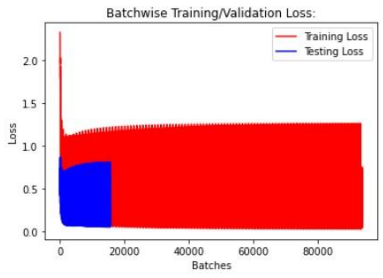
Training Loss and Validation Loss Comparison epoch wise

The Graph illustrates the comparison of the Training Loss and Validation Loss for each epoch. The training Loss and validation loss are calculated as the average of the batch wise loss values of the epoch. The Training loss is less compared to validation loss. Epochs = 200 and Learning Rate = 0.01 and Batch Size = 32



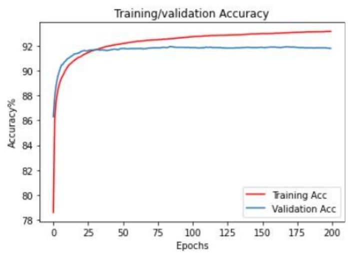
Training Loss and Validation Loss Comparison batch wise

The graph illustrates the comparison of batchwise training and validation accuracy. This indicates that the model has decrease in loss up to certain number of batches(~19000) and then flatlining. The training can be stopped after processing 20 epochs(18750 batches) to avoid overfitting. Epochs = 100 and Learning Rate = 0.01 and Batch Size = 32



Training Accuracy and Validation Accuracy Comparison Epoch wise

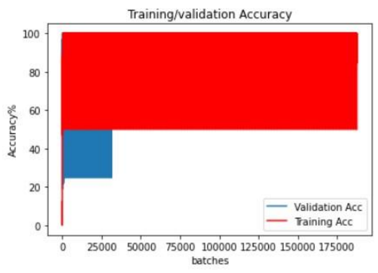
The graph illustrates Accuracy comparison between Training and Validation Epoch wise. Training accuracy is more than validation accuracy. However, after ~20 epochs the validation accuracy ~flatlined which indicates that model overfitting started after ~20 epochs. Even though the training accuracy is increasing, the validation accuracy did not improve. Epochs = 200 and Learning Rate = 0.01 and Batch Size = 32



Training Accuracy and Validation Accuracy Comparison batch wise

The batchwise training and validation accuracy also conveys same information as above graphs. One trend we can identify is that the batchwise train accuracies fluctuate but overall training accuracy increases. We don't need to train for a greater number of epoch iterations as the model achieves same accuracy for a smaller number of epochs. No valuable information can be identified from this graph

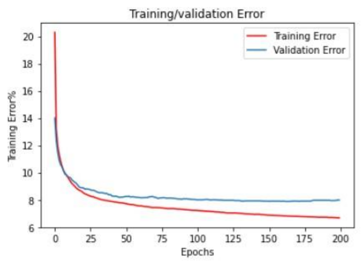
Epochs = 200 and Learning Rate = 0.01 and Batch Size = 32



Training Error and Validation Error Epoch wise

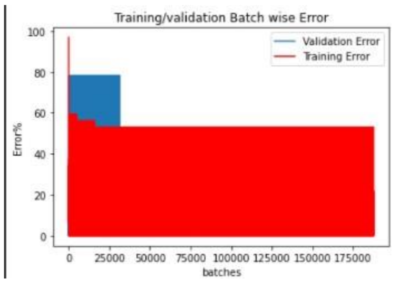
The Graph illustrates the Training vs Validation error epoch wise. As we see in the graph after a certain number of epochs the training error decreases but the validation error flatlines indicating overfitting.

Epochs = 200 and Learning Rate = 0.01 and Batch Size = 32



Training Error and Validation Error batch wise

The batchwise training and validation error also conveys same information as above graphs. One trend we can identify is that the batchwise train errors fluctuate but overall training error decreases. We don't need to train for a greater number of epoch iterations as the model achieves same accuracy for a smaller number of epochs. No valuable information can be identified from this graph Epochs = 200 and Learning Rate = 0.01 and Batch Size = 32


Training/validation Batch wise Error
Error%
batches
Validation Error
Training Error