Pytorch Neural network library:

We use the prebuilt Pytorch libraries to achieve the simple neural network architecture. Additionally, we have two variations to use SGD optimizer with a drop out value of 0.25 and Adam optimizer with a drop out value of 0.50.

we define Network class which inherits the NN.Module which is base class for all the neural network modules.

```
class Network(nn.Module):
    def __init__(self, input_size,num_classes,dropout):
        super(Network,self).__init__()
        self.fc1 = nn.Linear(input_size, num_classes)
        self.dropout = nn.Dropout(dropout)
        self.relu = nn.ReLU()
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self,x):
        out = self.dropout(x)
        out = self.fc1(out)
        out = self.relu(out)
        out = self.softmax(out)
        return out
```

we use torch.utils.data Module to random sample 50% of training and validation data. Also, we use the Data Loader class from the same module to split the data batchwise with shuffling and process data in each epoch and batch iterations.

```
train = tv.datasets.MNIST('../data',transform = transform, download=True,train=True)
val = tv.datasets.MNIST('../data',transform = transform, download=True,train=False)


My_list = [*range(0, 60000, 1)]
My_list1 = [*range(0, 10000, 1)]

strain = torch.utils.data.Subset(train, random.sample(My_list, train50Len))
sval = torch.utils.data.Subset(val, random.sample(My_list1, val50Len))



netlayers = Network(num_features, num_classes,dropout=0.50)
if torch.cuda.is_available():
    netlayers.to(device)

loss_function = nn.NLLLoss()
optimizer = torch.optim.Adam(netlayers.parameters(), lr=learning_rate)



train_loader = torch.utils.data.DataLoader(strain, batch_size=batch_size,
                                           shuffle=True)
val_loader = torch.utils.data.DataLoader(sval, batch_size=batch_size,
                                           shuffle=True)
```

```
train = tv.datasets.MNIST('../data',transform = transform, download=True,train=True)
val = tv.datasets.MNIST('../data',transform = transform, download=True,train=False)


My_list = [*range(0, 60000, 1)]
My_list1 = [*range(0, 10000, 1)]

strain = torch.utils.data.Subset(train, random.sample(My_list, train50Len))
sval = torch.utils.data.Subset(val, random.sample(My_list1, val50Len))



netlayers = Network(num_features, num_classes,dropout=0.25)
if torch.cuda.is_available():
    netlayers.to(device)

loss_function = nn.NLLLoss()
optimizer = torch.optim.SGD(netlayers.parameters(), lr=learning_rate)



train_loader = torch.utils.data.DataLoader(strain, batch_size=batch_size,
                                           shuffle=True)
val_loader = torch.utils.data.DataLoader(sval, batch_size=batch_size,
                                           shuffle=True)
```

Each epoch iteration contains model.train() and Model.eval() phases. The nn.dropout class will be executed only during the training phase and not in the validation phase.

Each batch iteration contains loss function which is cross entropy loss, backward function for gradient descent process, optimizer step function to update the parameters and Zero Grad function to reset the gradients

```
for epoch in range(num_epochs):
    train_accuracy = 0
    test_accuracy = 0
    epochtrainloss = []
    epochtestloss = []
    netlayers.train()
    for i ,(images,labels) in enumerate(train_loader):
            images = images.view(-1,28*28).to(device)
            labels = labels.to(device)
            optimizer.zero_grad()
            outputs = netlayers(images)
#           print(outputs.shape," ",labels.shape)
            Tloss = loss_function(outputs, labels)
            epochtrainloss.append(Tloss.cpu().item())
            _, predicted = torch.max(outputs,1)
            train_accuracy += np.count_nonzero(predicted.cpu().data.numpy() == labels.cpu().data.numpy())
            Tloss.backward()
            optimizer.step()
    loss.append(np.mean(epochtrainloss))
    trainingacc.append((train_accuracy/train50Len) *100)
    print("Train Accuracy:",(train_accuracy/train50Len) *100)
    netlayers.eval()
    for i ,(images,labels) in enumerate(val_loader):
            images = images.view(-1,28*28).to(device)
            labels = labels.to(device)
            outputs = netlayers(images)
            validloss = loss_function(outputs, labels)
            epochtestloss.append(validloss.cpu().item())
            _, predicted = torch.max(outputs,1)
            test_accuracy += np.count_nonzero(predicted.cpu().data.numpy() == labels.cpu().data.numpy())
    valloss.append(np.mean(epochtestloss))
    validationacc.append((test_accuracy/val50Len) *100)
    print("Test Accuracy:",(test_accuracy/val50Len) *100)
```

1) SGD optimizer with a drop out value of 0.25

In this approach we have used drop out value of 0.25 which means we randomly select 25% of the neurons and set their weights to zero for the forward and backward passes i.e., for one iteration. The drop out is used to generalize the model to reduce the overfitting scenarios.

Final Training and Validation Accuracy reported

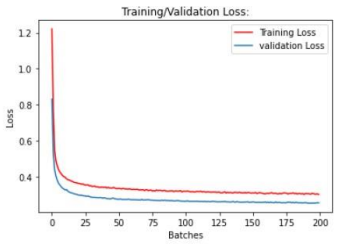
- Training Accuracy: 91.22%
- Validation Accuracy: 92.7%

The Accuracy of Validation phase is more than the Training phase as the model architecture is very simple with only 10 neurons and the number of epochs on which training is performed is less.

Training Loss and Validation Loss Comparison epoch wise

The graph illustrates epoch wise comparison of training and validation loss values. We see that there is no significant improvement of the model after a certain number of epochs(~25). As we are using drop out in this case, we can see that the training loss did not significantly drop when compared to validation loss unlike the overfitting case.

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

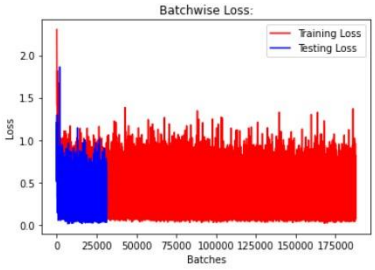


Training Loss and Validation Loss Comparison batch wise

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 gradually 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

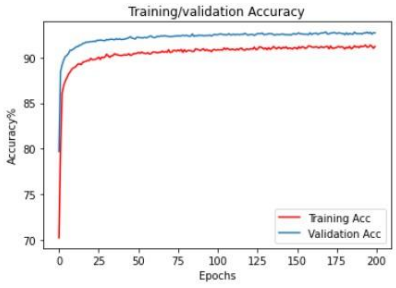
Epochs = 200 and Learning Rate = 0.01 and Batch Size = 32



Training Accuracy and Validation Accuracy Comparison Epoch wise

the model did not overfit during training as the model is generalized and the training and validation accuracy ~flatlined after a certain epoch showing no significant improvement in the accuracy.

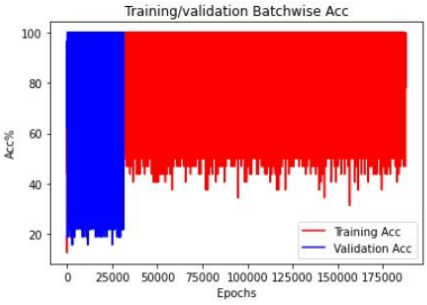
Epochs = 200 and Learning Rate = 0.01 and Batch Size = 32



Training Accuracy and Validation Accuracy Comparison batch wise

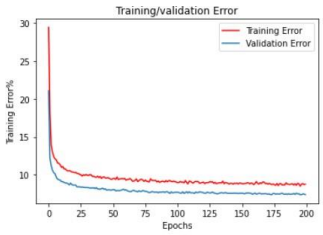
The batchwise training and validation accuracy also conveys same information as above graphs. One trend we can identify is that the batchwise train accuracies fluctuate but overall training accuracy increases. We don't need to train for a greater number of epoch iterations as the model achieves same 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



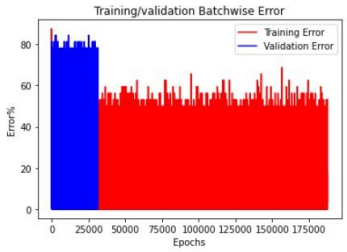
Training Error and Validation Error Epoch wise

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 and validation errors. 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



2) Adam optimizer with a drop out value of 0.50

In this approach we have used drop out value of 0.50 which means we randomly select 50% of the neurons and set their weights to zero for the forward and backward passes i.e., for one iteration. The drop out is used to generalize the model to reduce the overfitting scenarios.

Final Training and Validation Accuracy reported

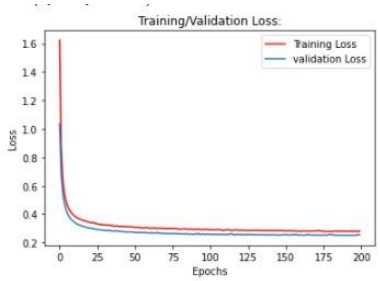
- Training Accuracy: 91.80%
- Validation Accuracy: 92.78%

The Accuracy of Validation phase is more than the Training phase as the model architecture is very simple with only 10 neurons and the number of epochs on which training is performed is less. Also, The Adam algorithm struggles when the learning rate is high (0.01) leading to lower accuracy and eventually affecting convergence. Adams works well for lower learning rate. We have taken 1e-4 as the learning rate.

Training Loss and Validation Loss Comparison epoch wise

The graph illustrates epoch wise comparison of training and validation loss values. We see that there is no significant improvement of the model after a certain number of epochs(~25). As we are using drop out in this case, we can see that the training loss did not significantly drop when compared to validation loss. The Loss values did not decrease as the Adam optimizer works well for low learning rates.

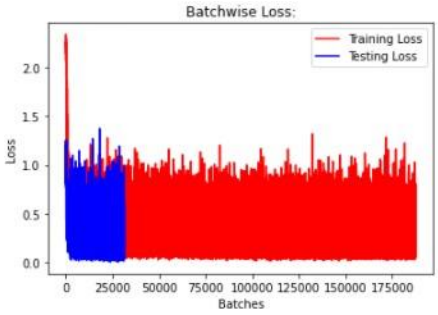
Epochs = 200 and Learning Rate = 1e-4 and Batch Size = 32



Training Loss and Validation Loss 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 gradually 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

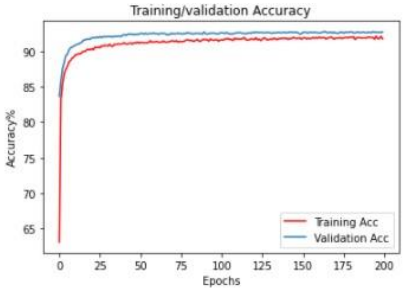
Epochs = 200 and Learning Rate = 1e-4 and Batch Size = 32



Training Accuracy and Validation Accuracy Comparison Epoch wise

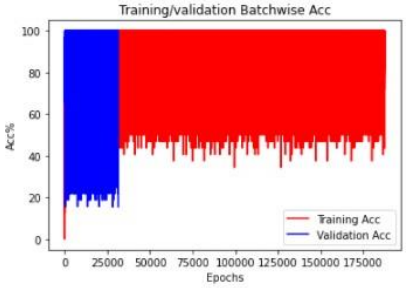
the model did not overfit during training as the model is generalized and the training and validation accuracy after a certain epoch showed no significant improvement.

Epochs = 200 and Learning Rate = 1e-4 and Batch Size = 32



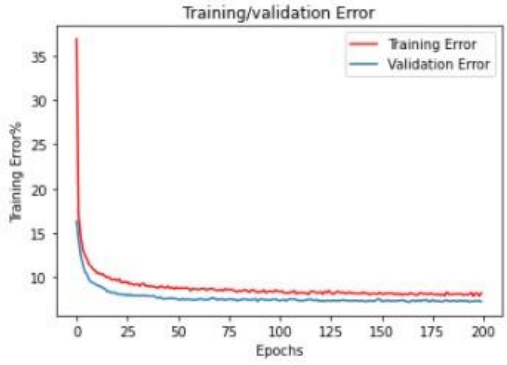
Training Accuracy and Validation Accuracy Comparison batch wise

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 = 1e-4 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 and validation errors ~flatlines. Epochs = 200 and Learning Rate = 1e-4 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. Epochs = 200 and Learning Rate = 1e-4 and Batch Size = 32