In [ ]:

```
!nvidia-smi
```

```
Tue Sep 20 14:45:30 2022
+-----------------------------------------------------------------------------
----------+
| NVIDIA-SMI 515.43.04    Driver Version: 515.43.04    CUDA Version:
11.7     |
|-------------------------------+----------------------+------------
----------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Un
corr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  C
ompute M. |
|                               |                      |
MIG M. |
|===============================+======================+============
==========|
|   0  NVIDIA GeForce ...  On   | 00000000:09:00.0 Off |
N/A |
| 36%   50C    P2   115W / 350W |   9275MiB / 24576MiB |      0%
Default |
|                               |                      |
N/A |
+-------------------------------+----------------------+------------
----------+

+-----------------------------------------------------------------------------
----------+
| Processes:
|
|  GPU   GI   CI        PID   Type   Process name                  G
PU Memory |
|        ID   ID                                                   U
sage      |
|=============================================================================
==========|
|    0   N/A  N/A      1206      G   /usr/lib/xorg/Xorg
35MiB |
|    0   N/A  N/A     72979      G   /usr/lib/xorg/Xorg
76MiB |
|    0   N/A  N/A     73105      G   /usr/bin/gnome-shell
15MiB |
|    0   N/A  N/A    714142      C   ...runima/aru_env/bin/python
1767MiB |
|    0   N/A  N/A    984025      C   ...ehasree/DLvenv/bin/python
1791MiB |
|    0   N/A  N/A    984428      C   ...bhumika/DL/dl/bin/python3
3721MiB |
|    0   N/A  N/A    990919      C   ...runima/aru_env/bin/python
1789MiB |
+-----------------------------------------------------------------------------
----------+
```

In [ ]:

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import numpy as np
import matplotlib.pyplot as plt
```

In [ ]:

```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(device)
```

```
cuda:0
```

In [ ]:

```
BATCH_SIZE = 20
NUM_EPOCHS = 5
LEARNING_RATE=0.0001
```

In [ ]:

```
train_data = torchvision.datasets.MNIST(root='./',train=True,transform=transform
s.ToTensor(),download=True)
test_data = torchvision.datasets.MNIST(root='./',train=False,transform=transform
s.ToTensor())

train_loader = torch.utils.data.DataLoader(dataset=train_data,batch_size=BATCH_S
IZE,shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_data,batch_size=len(test_
data),shuffle=False)
```

In [ ]:

```
class CNN(nn.Module):
  def __init__(self):
    super(CNN,self).__init__()
    self.conv_layer1=nn.Sequential(nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, stride=1, padding=1, padding_mode='zeros'),
                                nn.ReLU(),
                                nn.MaxPool2d(kernel_size=2, stride=2))

    self.conv_layer2=nn.Sequential(nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3, stride=1, padding=1, padding_mode='zeros'),
                                nn.ReLU(),
                                nn.MaxPool2d(kernel_size=2, stride=2))

    self.fc1=nn.Sequential(nn.Linear(in_features=32*7*7, out_features=500),
                           nn.ReLU())

    self.fc2=nn.Linear(in_features=500, out_features=10)


  def forward(self,input_):
    out=self.conv_layer1(input_)
    out=self.conv_layer2(out)
    out=out.reshape(out.size(0),-1)
    out=self.fc1(out)
    out=self.fc2(out)
    return out
```

In [ ]:

```
def plot_loss(train_error, test_error, accuracy, steps=200):
    x=int(steps)*np.arange(0, len(train_error))

    plt.rcParams["figure.figsize"] = (18,6)
    fig, (ax1, ax2) = plt.subplots(1, 2)

    ax1.plot(x,train_error, label='Train Error')
    ax1.plot(x,test_error, label='Test Error')
    ax1.set_title('Error Plot')
    ax1.set(xlabel='Iterations', ylabel='Average Error')
    ax1.legend(loc='upper left')
    ax1.grid()



    ax2.plot(x, accuracy, label='Accuracy')
    ax2.set_title('Model Test Accuracy')
    ax2.set(xlabel='Iterations', ylabel='Accuracy')
    ax2.grid()
    plt.show()



def plot_images(show_images, show_labels, model, bn):
    pred=model(show_images, bn=bn)
    predlabels=torch.max(pred,1)[1].cpu().numpy()
    fig=plt.figure(figsize=(10, 10))
    columns = 3
    rows = 1
    for i in range(1, columns*rows +1):
        fig.add_subplot(rows, columns, i)
        trueval=show_labels[i-1].data.cpu().numpy()
        predval=predlabels[i-1]
        a='True: '+str(trueval)+' Pred: '+str(predval)
        plt.title(a)
        plt.axis('off')
        plt.imshow(show_images[i-1].reshape(-1,28), cmap='gray')
    plt.show()
```

# MODEL TRAINING

In [ ]:

```
model = CNN().to(device)
```

In [ ]:

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),lr=LEARNING_RATE)
```

In [ ]:

```
%%time
import time

total_steps = len(train_loader)
cost_train,cost_test,acc = [],[],[]
training_start_time=time.time()

for epoch in range(NUM_EPOCHS):
  for i, (images,labels) in enumerate(train_loader):
    images = images.to(device)
    labels = labels.to(device)

    outputs = model(images)
    loss = criterion(outputs,labels)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (i+1) % 200 == 0:
      print('Epoch:[{}/{}], Step:{}/{}, Loss:{:.6f}'.format(epoch+1,NUM_EPOCHS,i
+1,total_steps,loss.item()))
      model.eval()
      with torch.no_grad():
        correct,total = 0,0
        for images,labels in test_loader:
          images = images.to(device)
          labels = labels.to(device)

          preds = model(images)
          val_loss = criterion(preds,labels)
          _,pred = torch.max(preds.data,1)
          total += labels.size(0)
          correct += (pred == labels).sum().item()

        print('Test Loss:{:4f}, Test Accuracy:{:.4f}'.format(val_loss.item(),cor
rect/total))
        cost_train.append(loss.item())
        acc.append(correct/total)
        cost_test.append(val_loss.item())
print('Training finished, took {:.2f}s'.format(time.time() - training_start_time
))
torch.save(model.state_dict(),'model.ckpt')
```

```
Epoch:[1/5], Step:200/3000, Loss:0.849734
Test Loss:0.695792, Test Accuracy:0.8176
Epoch:[1/5], Step:400/3000, Loss:0.344382
Test Loss:0.393343, Test Accuracy:0.8913
Epoch:[1/5], Step:600/3000, Loss:0.633193
Test Loss:0.312559, Test Accuracy:0.9128
Epoch:[1/5], Step:800/3000, Loss:0.294416
Test Loss:0.277228, Test Accuracy:0.9183
Epoch:[1/5], Step:1000/3000, Loss:0.501083
Test Loss:0.228259, Test Accuracy:0.9344
Epoch:[1/5], Step:1200/3000, Loss:0.279125
Test Loss:0.227319, Test Accuracy:0.9303
Epoch:[1/5], Step:1400/3000, Loss:0.067257
Test Loss:0.199450, Test Accuracy:0.9442
Epoch:[1/5], Step:1600/3000, Loss:0.067029
Test Loss:0.201241, Test Accuracy:0.9401
Epoch:[1/5], Step:1800/3000, Loss:0.299174
Test Loss:0.167326, Test Accuracy:0.9512
Epoch:[1/5], Step:2000/3000, Loss:0.096857
Test Loss:0.156477, Test Accuracy:0.9545
Epoch:[1/5], Step:2200/3000, Loss:0.186756
Test Loss:0.144172, Test Accuracy:0.9574
Epoch:[1/5], Step:2400/3000, Loss:0.076194
Test Loss:0.135223, Test Accuracy:0.9604
Epoch:[1/5], Step:2600/3000, Loss:0.026009
Test Loss:0.131710, Test Accuracy:0.9615
Epoch:[1/5], Step:2800/3000, Loss:0.126760
Test Loss:0.119893, Test Accuracy:0.9636
Epoch:[1/5], Step:3000/3000, Loss:0.009321
Test Loss:0.099981, Test Accuracy:0.9709
Epoch:[2/5], Step:200/3000, Loss:0.050325
Test Loss:0.107192, Test Accuracy:0.9673
Epoch:[2/5], Step:400/3000, Loss:0.073800
Test Loss:0.100694, Test Accuracy:0.9705
Epoch:[2/5], Step:600/3000, Loss:0.032944
Test Loss:0.090223, Test Accuracy:0.9726
Epoch:[2/5], Step:800/3000, Loss:0.187194
Test Loss:0.099886, Test Accuracy:0.9699
Epoch:[2/5], Step:1000/3000, Loss:0.025725
Test Loss:0.085436, Test Accuracy:0.9736
Epoch:[2/5], Step:1200/3000, Loss:0.353789
Test Loss:0.078879, Test Accuracy:0.9767
Epoch:[2/5], Step:1400/3000, Loss:0.101045
Test Loss:0.079942, Test Accuracy:0.9747
Epoch:[2/5], Step:1600/3000, Loss:0.031332
Test Loss:0.071385, Test Accuracy:0.9767
Epoch:[2/5], Step:1800/3000, Loss:0.004944
Test Loss:0.071195, Test Accuracy:0.9765
Epoch:[2/5], Step:2000/3000, Loss:0.043386
Test Loss:0.073078, Test Accuracy:0.9775
Epoch:[2/5], Step:2200/3000, Loss:0.095397
Test Loss:0.075729, Test Accuracy:0.9758
Epoch:[2/5], Step:2400/3000, Loss:0.088071
Test Loss:0.075781, Test Accuracy:0.9764
Epoch:[2/5], Step:2600/3000, Loss:0.019372
Test Loss:0.065836, Test Accuracy:0.9802
Epoch:[2/5], Step:2800/3000, Loss:0.122448
Test Loss:0.069770, Test Accuracy:0.9774
Epoch:[2/5], Step:3000/3000, Loss:0.015931
Test Loss:0.060239, Test Accuracy:0.9811
Epoch:[3/5], Step:200/3000, Loss:0.094740
```

```
Test Loss:0.060504, Test Accuracy:0.9798
Epoch:[3/5], Step:400/3000, Loss:0.014878
Test Loss:0.068209, Test Accuracy:0.9766
Epoch:[3/5], Step:600/3000, Loss:0.127180
Test Loss:0.058092, Test Accuracy:0.9824
Epoch:[3/5], Step:800/3000, Loss:0.015416
Test Loss:0.056459, Test Accuracy:0.9827
Epoch:[3/5], Step:1000/3000, Loss:0.028575
Test Loss:0.058294, Test Accuracy:0.9817
Epoch:[3/5], Step:1200/3000, Loss:0.020101
Test Loss:0.055561, Test Accuracy:0.9827
Epoch:[3/5], Step:1400/3000, Loss:0.074958
Test Loss:0.053316, Test Accuracy:0.9828
Epoch:[3/5], Step:1600/3000, Loss:0.023973
Test Loss:0.055770, Test Accuracy:0.9824
Epoch:[3/5], Step:1800/3000, Loss:0.063499
Test Loss:0.048422, Test Accuracy:0.9857
Epoch:[3/5], Step:2000/3000, Loss:0.049247
Test Loss:0.058669, Test Accuracy:0.9810
Epoch:[3/5], Step:2200/3000, Loss:0.040926
Test Loss:0.050319, Test Accuracy:0.9839
Epoch:[3/5], Step:2400/3000, Loss:0.021810
Test Loss:0.048931, Test Accuracy:0.9839
Epoch:[3/5], Step:2600/3000, Loss:0.085184
Test Loss:0.047539, Test Accuracy:0.9857
Epoch:[3/5], Step:2800/3000, Loss:0.022716
Test Loss:0.046244, Test Accuracy:0.9841
Epoch:[3/5], Step:3000/3000, Loss:0.106513
Test Loss:0.056100, Test Accuracy:0.9817
Epoch:[4/5], Step:200/3000, Loss:0.063561
Test Loss:0.048317, Test Accuracy:0.9842
Epoch:[4/5], Step:400/3000, Loss:0.002909
Test Loss:0.046094, Test Accuracy:0.9847
Epoch:[4/5], Step:600/3000, Loss:0.108059
Test Loss:0.041243, Test Accuracy:0.9864
Epoch:[4/5], Step:800/3000, Loss:0.062989
Test Loss:0.044150, Test Accuracy:0.9854
Epoch:[4/5], Step:1000/3000, Loss:0.070343
Test Loss:0.063219, Test Accuracy:0.9803
Epoch:[4/5], Step:1200/3000, Loss:0.005794
Test Loss:0.045562, Test Accuracy:0.9855
Epoch:[4/5], Step:1400/3000, Loss:0.310674
Test Loss:0.050104, Test Accuracy:0.9843
Epoch:[4/5], Step:1600/3000, Loss:0.003178
Test Loss:0.048343, Test Accuracy:0.9836
Epoch:[4/5], Step:1800/3000, Loss:0.026031
Test Loss:0.044231, Test Accuracy:0.9851
Epoch:[4/5], Step:2000/3000, Loss:0.008884
Test Loss:0.043988, Test Accuracy:0.9855
Epoch:[4/5], Step:2200/3000, Loss:0.032468
Test Loss:0.045397, Test Accuracy:0.9848
Epoch:[4/5], Step:2400/3000, Loss:0.069178
Test Loss:0.039062, Test Accuracy:0.9860
Epoch:[4/5], Step:2600/3000, Loss:0.014446
Test Loss:0.050582, Test Accuracy:0.9833
Epoch:[4/5], Step:2800/3000, Loss:0.002398
Test Loss:0.037827, Test Accuracy:0.9873
Epoch:[4/5], Step:3000/3000, Loss:0.018694
Test Loss:0.040526, Test Accuracy:0.9868
Epoch:[5/5], Step:200/3000, Loss:0.025070
Test Loss:0.036425, Test Accuracy:0.9884
```

```
Epoch:[5/5], Step:400/3000, Loss:0.017337
Test Loss:0.037726, Test Accuracy:0.9873
Epoch:[5/5], Step:600/3000, Loss:0.007466
Test Loss:0.035333, Test Accuracy:0.9890
Epoch:[5/5], Step:800/3000, Loss:0.000711
Test Loss:0.038961, Test Accuracy:0.9883
Epoch:[5/5], Step:1000/3000, Loss:0.048036
Test Loss:0.035580, Test Accuracy:0.9881
Epoch:[5/5], Step:1200/3000, Loss:0.033570
Test Loss:0.035251, Test Accuracy:0.9892
Epoch:[5/5], Step:1400/3000, Loss:0.004029
Test Loss:0.037789, Test Accuracy:0.9867
Epoch:[5/5], Step:1600/3000, Loss:0.009144
Test Loss:0.036323, Test Accuracy:0.9885
Epoch:[5/5], Step:1800/3000, Loss:0.003157
Test Loss:0.034968, Test Accuracy:0.9879
Epoch:[5/5], Step:2000/3000, Loss:0.000849
Test Loss:0.038016, Test Accuracy:0.9875
Epoch:[5/5], Step:2200/3000, Loss:0.019654
Test Loss:0.036931, Test Accuracy:0.9874
Epoch:[5/5], Step:2400/3000, Loss:0.097614
Test Loss:0.033976, Test Accuracy:0.9892
Epoch:[5/5], Step:2600/3000, Loss:0.014395
Test Loss:0.035893, Test Accuracy:0.9878
Epoch:[5/5], Step:2800/3000, Loss:0.054985
Test Loss:0.032896, Test Accuracy:0.9888
Epoch:[5/5], Step:3000/3000, Loss:0.015258
Test Loss:0.037566, Test Accuracy:0.9868
Training finished, took 60.46s
CPU times: user 1min, sys: 106 ms, total: 1min 1s
Wall time: 1min
```

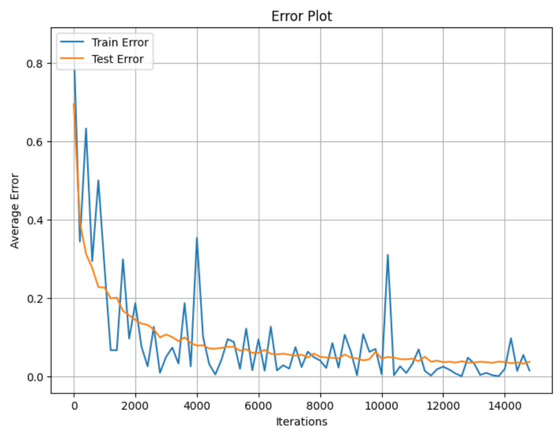
Training finished, took 60.46s// CPU times: user 1min, sys: 106 ms, total: 1min 1s Wall time: 1min

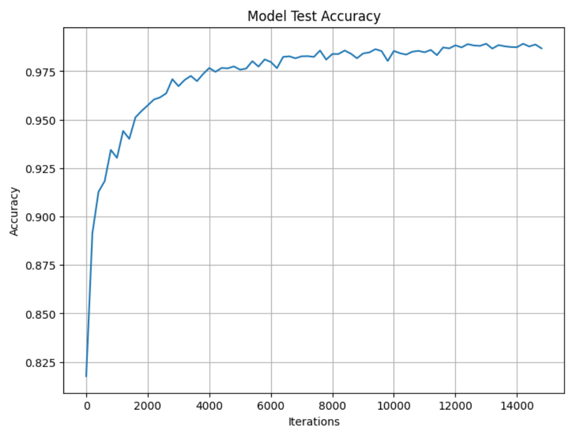
1. Show the plot of training error, validation error and prediction accuracy as the training progresses. At the end of training, report the average prediction accuracy for the whole test set of 10000 images.

Answer: After 5 epochs, the test Accuracy is 98.68%

In [ ]:

```
plot_loss(cost_train, cost_test, acc, 200)
```

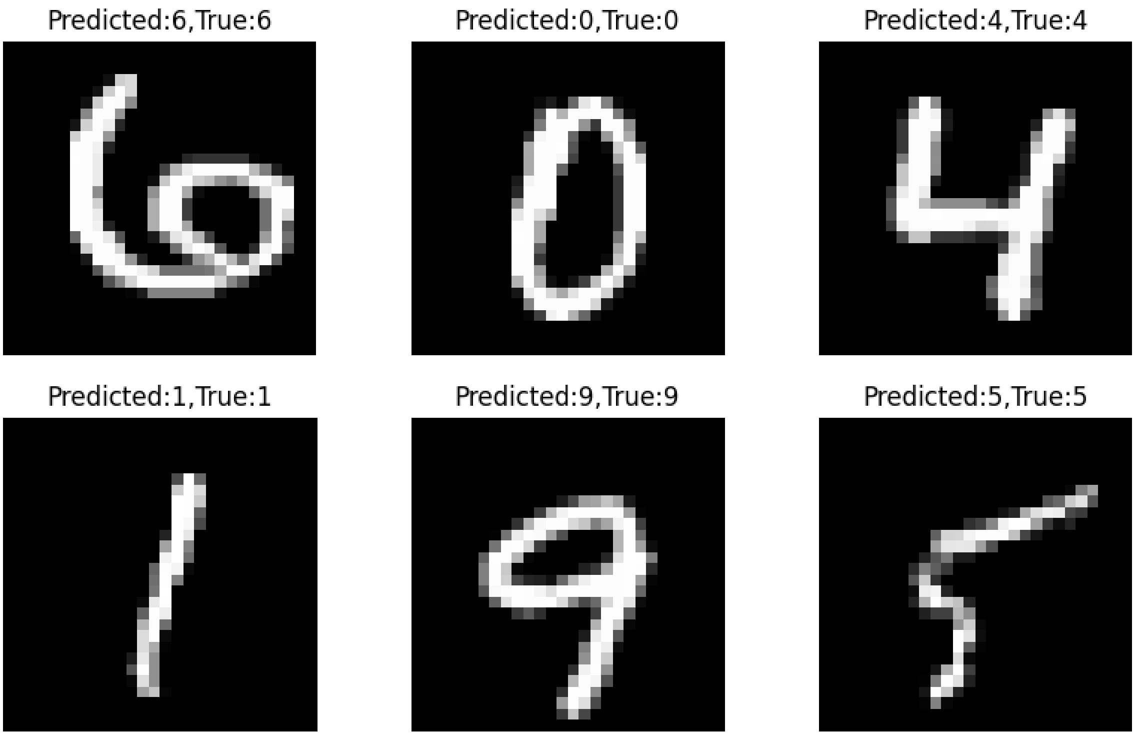


1. Plot randomly selected test images showing the true and predicted class labels.

In [ ]:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
for img,label in test_loader:
  img=img[54:76]
  img = img.to(device)
  label=label[54:76]
  label = label.to(device)
  preds = model(img)
  fig = plt.figure(figsize=(10,6))

  col,row = 3,2
  y_preds = torch.max(preds.data,1)[1].cpu().numpy()
  for i in range(1,col*row+1):
    a = fig.add_subplot(row,col,i)
    a.set_title('Predicted:{},True:{}'.format(y_preds[i-1],label.data[i-1]))
    plt. axis('off')
    plt.imshow(img[i-1,0].cpu().numpy(),cmap='gray')

  plt.show()
```




1. Report the dimensions of the input and output at each layer.
2. How many parameters does your network have? How many of thes are in the fully connected layers and how many are in the convolutional layers?

Answer:

| Layer | Input dimension | Output dimension | Number of Parameters(Weights) | Number of Parameters(biases) | Total Number of Parameters |
|---|---|---|---|---|---|
| layer1.conv1 | 28,28,1 | 28,28,32 | W=3 *3* 32= 288 | b=32 | 288+32 = **320** |
| layer1.maxpool | 28,28,32 | 14,14,32 | 0 | 0 | 0 |
| layer2.conv2 | 14,14,32 | 14,14,32 | W=(3 *3* 32 )* 32= 9216 | b=32 | 9216+32=**9248** |
| layer2.maxpool | 14,14,32 | 7,7,32 | 0 | 0 | 0 |
| fully_connected1 | *7732* = 1568 | 500 (given) | W=1568* 500 = 7.84L | b=500 | 7.84L+500= **7.845L** |
| fully_connected2 | 500 | 10 (num_classes) | W=500* 10= 5000 | b=10 | 5000+10=**5010** |

Total Number of parameters : 7,99,078 ~ 8L

Number of parameters in FC layers : **7,89,510**

Number of parameters in Conv layers : 9568

1. How many neurons does your network have? How many of these are in the fully connected layers and how many are in the convolutional layers?

Answer:

| Layer | Neurons |
|---|---|
| Layer1 | 3 *3* 32= 288 |
| Layer2 | 3 *3* 32* 32= 9216 |
| FC1 | 500 |
| FC2 | 10 |

# MODEL TRAINING USING BATCH NORMALISATION

1. Use batch-normalization. Does it improve the test accuracy? Does it affect training time?

In [ ]:

```
class CNN_BN(nn.Module):
  def __init__(self):
    super(CNN_BN,self).__init__()
    self.conv_layer1=nn.Sequential(nn.Conv2d(in_channels=1, out_channels=32, ker
nel_size=3, stride=1, padding=1, padding_mode='zeros'),
                                   nn.ReLU(),
                                   nn.BatchNorm2d(32),
                                   nn.MaxPool2d(kernel_size=2, stride=2))

    self.conv_layer2=nn.Sequential(nn.Conv2d(in_channels=32, out_channels=32, ke
rnel_size=3, stride=1, padding=1, padding_mode='zeros'),
                                   nn.ReLU(),
                                   nn.BatchNorm2d(32),
                                   nn.MaxPool2d(kernel_size=2, stride=2))

    self.fc1=nn.Sequential(nn.Linear(in_features=32*7*7, out_features=500),
                           nn.ReLU())

    self.fc2=nn.Linear(in_features=500, out_features=10)



  def forward(self,input_):
    out=self.conv_layer1(input_)
    out=self.conv_layer2(out)
    out=out.reshape(out.size(0),-1)
    out=self.fc1(out)
    out=self.fc2(out)
    return out
```

In [ ]:

```
model_bn = CNN_BN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_bn.parameters(),lr=LEARNING_RATE)
```

In [ ]:

```
%%time
import time

total_steps = len(train_loader)
cost_train_bn,cost_test_bn,acc_bn = [],[],[]
training_start_time = time.time()

for epoch in range(NUM_EPOCHS):
  for i, (images,labels) in enumerate(train_loader):
    images = images.to(device)
    labels = labels.to(device)

    outputs = model_bn(images)
    loss = criterion(outputs,labels)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (i+1) % 200 == 0:
      print('Epoch:[{}/{}], Step:{}/{}, Loss:{:.6f}'.format(epoch+1,NUM_EPOCHS,i
+1,total_steps,loss.item()))
      model_bn.eval()
      with torch.no_grad():
        correct,total = 0,0
        for images,labels in test_loader:
          images = images.to(device)
          labels = labels.to(device)

          preds = model_bn(images)
          val_loss = criterion(preds,labels)
          _,pred = torch.max(preds.data,1)
          total += labels.size(0)
          correct += (pred == labels).sum().item()

        print('Test Loss:{:4f}, Test Accuracy:{:.4f}'.format(val_loss.item(),cor
rect/total))
        cost_train_bn.append(loss.item())
        acc_bn.append(correct/total)
        cost_test_bn.append(val_loss.item())

print('Training finished, took {:.2f}s'.format(time.time() - training_start_time
))
torch.save(model_bn.state_dict(),'model_bn.ckpt')
```

```
Epoch:[1/5], Step:200/3000, Loss:0.347887
Test Loss:0.254821, Test Accuracy:0.9329
Epoch:[1/5], Step:400/3000, Loss:0.065046
Test Loss:0.127743, Test Accuracy:0.9611
Epoch:[1/5], Step:600/3000, Loss:0.122614
Test Loss:0.104515, Test Accuracy:0.9694
Epoch:[1/5], Step:800/3000, Loss:0.041508
Test Loss:0.100825, Test Accuracy:0.9690
Epoch:[1/5], Step:1000/3000, Loss:0.142908
Test Loss:0.085262, Test Accuracy:0.9748
Epoch:[1/5], Step:1200/3000, Loss:0.205248
Test Loss:0.082362, Test Accuracy:0.9746
Epoch:[1/5], Step:1400/3000, Loss:0.064580
Test Loss:0.067769, Test Accuracy:0.9791
Epoch:[1/5], Step:1600/3000, Loss:0.385200
Test Loss:0.073187, Test Accuracy:0.9775
Epoch:[1/5], Step:1800/3000, Loss:0.009780
Test Loss:0.059134, Test Accuracy:0.9827
Epoch:[1/5], Step:2000/3000, Loss:0.006281
Test Loss:0.055794, Test Accuracy:0.9820
Epoch:[1/5], Step:2200/3000, Loss:0.052190
Test Loss:0.052710, Test Accuracy:0.9832
Epoch:[1/5], Step:2400/3000, Loss:0.074574
Test Loss:0.048072, Test Accuracy:0.9843
Epoch:[1/5], Step:2600/3000, Loss:0.052867
Test Loss:0.065276, Test Accuracy:0.9779
Epoch:[1/5], Step:2800/3000, Loss:0.010936
Test Loss:0.049556, Test Accuracy:0.9837
Epoch:[1/5], Step:3000/3000, Loss:0.001541
Test Loss:0.053046, Test Accuracy:0.9827
Epoch:[2/5], Step:200/3000, Loss:0.002072
Test Loss:0.044484, Test Accuracy:0.9854
Epoch:[2/5], Step:400/3000, Loss:0.080507
Test Loss:0.042427, Test Accuracy:0.9865
Epoch:[2/5], Step:600/3000, Loss:0.030892
Test Loss:0.048987, Test Accuracy:0.9840
Epoch:[2/5], Step:800/3000, Loss:0.001396
Test Loss:0.043488, Test Accuracy:0.9860
Epoch:[2/5], Step:1000/3000, Loss:0.020935
Test Loss:0.049574, Test Accuracy:0.9846
Epoch:[2/5], Step:1200/3000, Loss:0.079958
Test Loss:0.054626, Test Accuracy:0.9829
Epoch:[2/5], Step:1400/3000, Loss:0.015476
Test Loss:0.040363, Test Accuracy:0.9873
Epoch:[2/5], Step:1600/3000, Loss:0.018883
Test Loss:0.058056, Test Accuracy:0.9800
Epoch:[2/5], Step:1800/3000, Loss:0.003053
Test Loss:0.057497, Test Accuracy:0.9813
Epoch:[2/5], Step:2000/3000, Loss:0.002036
Test Loss:0.048220, Test Accuracy:0.9841
Epoch:[2/5], Step:2200/3000, Loss:0.046650
Test Loss:0.051945, Test Accuracy:0.9830
Epoch:[2/5], Step:2400/3000, Loss:0.004182
Test Loss:0.041244, Test Accuracy:0.9879
Epoch:[2/5], Step:2600/3000, Loss:0.076457
Test Loss:0.043678, Test Accuracy:0.9854
Epoch:[2/5], Step:2800/3000, Loss:0.011926
Test Loss:0.043911, Test Accuracy:0.9860
Epoch:[2/5], Step:3000/3000, Loss:0.018792
Test Loss:0.046393, Test Accuracy:0.9842
Epoch:[3/5], Step:200/3000, Loss:0.000719
```

```
Test Loss:0.031855, Test Accuracy:0.9898
Epoch:[3/5], Step:400/3000, Loss:0.013425
Test Loss:0.033083, Test Accuracy:0.9898
Epoch:[3/5], Step:600/3000, Loss:0.003819
Test Loss:0.042490, Test Accuracy:0.9860
Epoch:[3/5], Step:800/3000, Loss:0.035352
Test Loss:0.036673, Test Accuracy:0.9889
Epoch:[3/5], Step:1000/3000, Loss:0.003020
Test Loss:0.037716, Test Accuracy:0.9882
Epoch:[3/5], Step:1200/3000, Loss:0.001069
Test Loss:0.037715, Test Accuracy:0.9880
Epoch:[3/5], Step:1400/3000, Loss:0.001147
Test Loss:0.037137, Test Accuracy:0.9884
Epoch:[3/5], Step:1600/3000, Loss:0.002795
Test Loss:0.038800, Test Accuracy:0.9869
Epoch:[3/5], Step:1800/3000, Loss:0.109672
Test Loss:0.039806, Test Accuracy:0.9856
Epoch:[3/5], Step:2000/3000, Loss:0.004178
Test Loss:0.032553, Test Accuracy:0.9900
Epoch:[3/5], Step:2200/3000, Loss:0.001432
Test Loss:0.033457, Test Accuracy:0.9895
Epoch:[3/5], Step:2400/3000, Loss:0.014575
Test Loss:0.036116, Test Accuracy:0.9874
Epoch:[3/5], Step:2600/3000, Loss:0.002326
Test Loss:0.034444, Test Accuracy:0.9882
Epoch:[3/5], Step:2800/3000, Loss:0.102650
Test Loss:0.041235, Test Accuracy:0.9863
Epoch:[3/5], Step:3000/3000, Loss:0.184193
Test Loss:0.033099, Test Accuracy:0.9890
Epoch:[4/5], Step:200/3000, Loss:0.003777
Test Loss:0.032021, Test Accuracy:0.9896
Epoch:[4/5], Step:400/3000, Loss:0.000887
Test Loss:0.034631, Test Accuracy:0.9881
Epoch:[4/5], Step:600/3000, Loss:0.013172
Test Loss:0.032698, Test Accuracy:0.9893
Epoch:[4/5], Step:800/3000, Loss:0.000305
Test Loss:0.034085, Test Accuracy:0.9889
Epoch:[4/5], Step:1000/3000, Loss:0.005819
Test Loss:0.037659, Test Accuracy:0.9890
Epoch:[4/5], Step:1200/3000, Loss:0.001987
Test Loss:0.033673, Test Accuracy:0.9892
Epoch:[4/5], Step:1400/3000, Loss:0.001299
Test Loss:0.037654, Test Accuracy:0.9880
Epoch:[4/5], Step:1600/3000, Loss:0.009736
Test Loss:0.034010, Test Accuracy:0.9894
Epoch:[4/5], Step:1800/3000, Loss:0.031054
Test Loss:0.033274, Test Accuracy:0.9897
Epoch:[4/5], Step:2000/3000, Loss:0.008006
Test Loss:0.034075, Test Accuracy:0.9896
Epoch:[4/5], Step:2200/3000, Loss:0.010264
Test Loss:0.042053, Test Accuracy:0.9875
Epoch:[4/5], Step:2400/3000, Loss:0.003170
Test Loss:0.036927, Test Accuracy:0.9884
Epoch:[4/5], Step:2600/3000, Loss:0.000973
Test Loss:0.041767, Test Accuracy:0.9874
Epoch:[4/5], Step:2800/3000, Loss:0.082544
Test Loss:0.037056, Test Accuracy:0.9892
Epoch:[4/5], Step:3000/3000, Loss:0.015091
Test Loss:0.037025, Test Accuracy:0.9899
Epoch:[5/5], Step:200/3000, Loss:0.000572
Test Loss:0.030241, Test Accuracy:0.9908
```

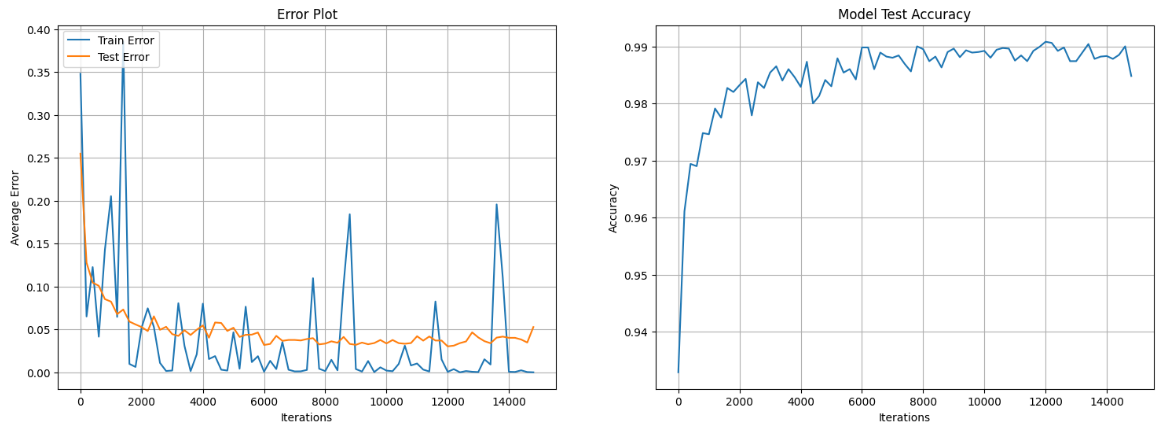
```
Epoch:[5/5], Step:400/3000, Loss:0.003703
Test Loss:0.031092, Test Accuracy:0.9906
Epoch:[5/5], Step:600/3000, Loss:0.000029
Test Loss:0.034035, Test Accuracy:0.9892
Epoch:[5/5], Step:800/3000, Loss:0.001476
Test Loss:0.035985, Test Accuracy:0.9898
Epoch:[5/5], Step:1000/3000, Loss:0.000664
Test Loss:0.046572, Test Accuracy:0.9874
Epoch:[5/5], Step:1200/3000, Loss:0.000317
Test Loss:0.040648, Test Accuracy:0.9874
Epoch:[5/5], Step:1400/3000, Loss:0.015182
Test Loss:0.036315, Test Accuracy:0.9889
Epoch:[5/5], Step:1600/3000, Loss:0.009100
Test Loss:0.033913, Test Accuracy:0.9904
Epoch:[5/5], Step:1800/3000, Loss:0.195644
Test Loss:0.040443, Test Accuracy:0.9878
Epoch:[5/5], Step:2000/3000, Loss:0.110386
Test Loss:0.041613, Test Accuracy:0.9882
Epoch:[5/5], Step:2200/3000, Loss:0.000630
Test Loss:0.040204, Test Accuracy:0.9883
Epoch:[5/5], Step:2400/3000, Loss:0.000420
Test Loss:0.040209, Test Accuracy:0.9878
Epoch:[5/5], Step:2600/3000, Loss:0.002319
Test Loss:0.038257, Test Accuracy:0.9885
Epoch:[5/5], Step:2800/3000, Loss:0.000329
Test Loss:0.034738, Test Accuracy:0.9900
Epoch:[5/5], Step:3000/3000, Loss:0.000032
Test Loss:0.052753, Test Accuracy:0.9848
Training finished, took 66.55s
CPU times: user 1min 7s, sys: 112 ms, total: 1min 7s
Wall time: 1min 6s
```

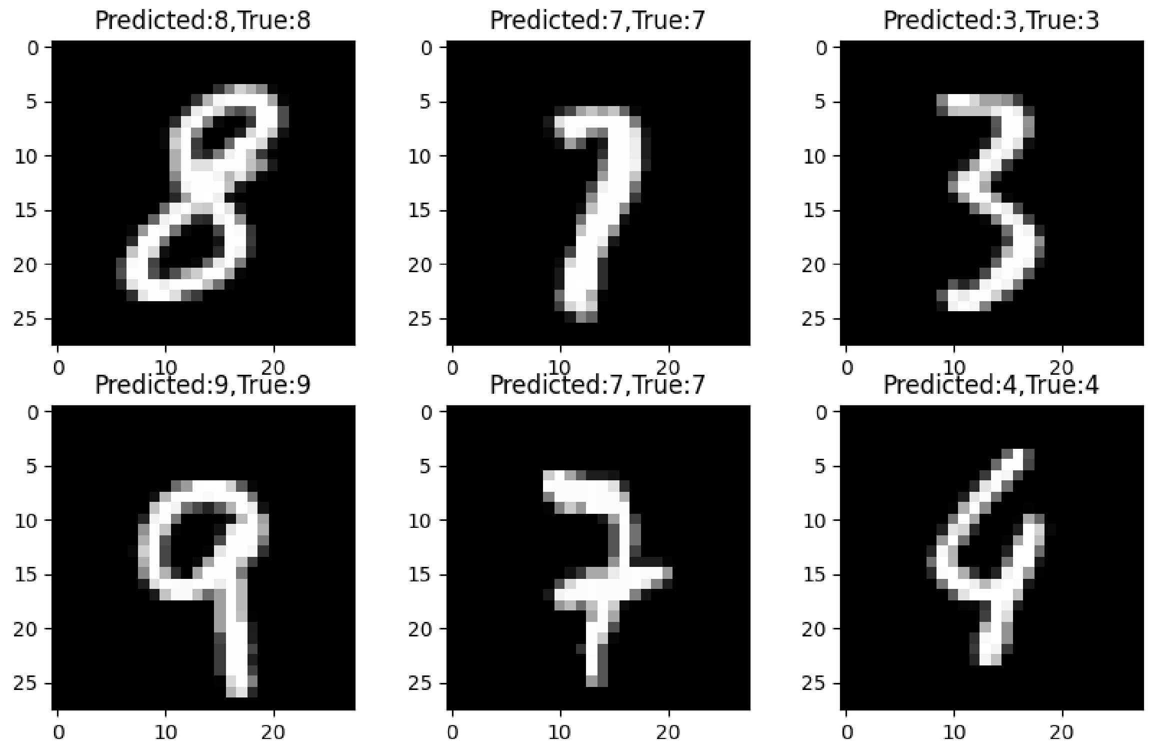
Training finished, took 66.55s// CPU times: user 1min 7s, sys: 112 ms, total: 1min 7s Wall time: 1min 6s

The model **after** batch normalisation takes more time (66.55s) to train and has test accuracy of 98.48%
The model **before** batch normalisation takes lesser time (60.46s) to train and has test accuracy of 98.68% .
</font>

In [ ]:

```
plot_loss(cost_train_bn, cost_test_bn, acc_bn, 200)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
for img,label in test_loader:
  img=img[110:116]
  img = img.to(device)
  label=label[110:116]
  label = label.to(device)
  preds = model_bn(img)
  fig = plt.figure(figsize=(10,6))
  fig.tight_layout()
  col,row = 3,2
  y_preds = torch.max(preds.data,1)[1].cpu().numpy()
  for i in range(1,col*row+1):
    a = fig.add_subplot(row,col,i)
    a.set_title('Predicted:{},True:{}'.format(y_preds[i-1],label.data[i-1]))
    plt.imshow(img[i-1,0].cpu().numpy(),cmap='gray')

  plt.show()
  break
```



# VISUALIZING FILTERS

In [ ]:

```
checkpoint = torch.load('model.ckpt')
model = CNN().to(device)
model.load_state_dict(checkpoint)
model.eval()
for param_tensor in model.state_dict():
    print(param_tensor, "\t", model.state_dict()[param_tensor].size())
```

```
conv_layer1.0.weight     torch.Size([32, 1, 3, 3])
conv_layer1.0.bias       torch.Size([32])
conv_layer2.0.weight     torch.Size([32, 32, 3, 3])
conv_layer2.0.bias       torch.Size([32])
fc1.0.weight     torch.Size([500, 1568])
fc1.0.bias       torch.Size([500])
fc2.weight       torch.Size([10, 500])
fc2.bias         torch.Size([10])
```

In [ ]:

```
l1_weights = model.state_dict()['conv_layer1.0.weight'].cpu().numpy()
l2_weights = model.state_dict()['conv_layer2.0.weight'].cpu().numpy()
```

In [ ]:

```
def normalize(x,eps=1e-8):
  out = np.zeros_like(x)
  for i in range(x.shape[0]):
    high,low = np.amax(x[i]),np.amin(x[i])
    out[i] = (x[i]-low)/(high-low+eps)

  out = out*255
  out = out.astype(np.uint8)
  return out

l1_weights_normalized = normalize(l1_weights)
l2_weights_normalized = normalize(l2_weights)
```

In [ ]:

```
print(l1_weights_normalized.shape)
```

```
(32, 1, 3, 3)
```

In [ ]:

```
l1_weights_normalized = l1_weights_normalized.reshape(32,3,3)
```

In [ ]:

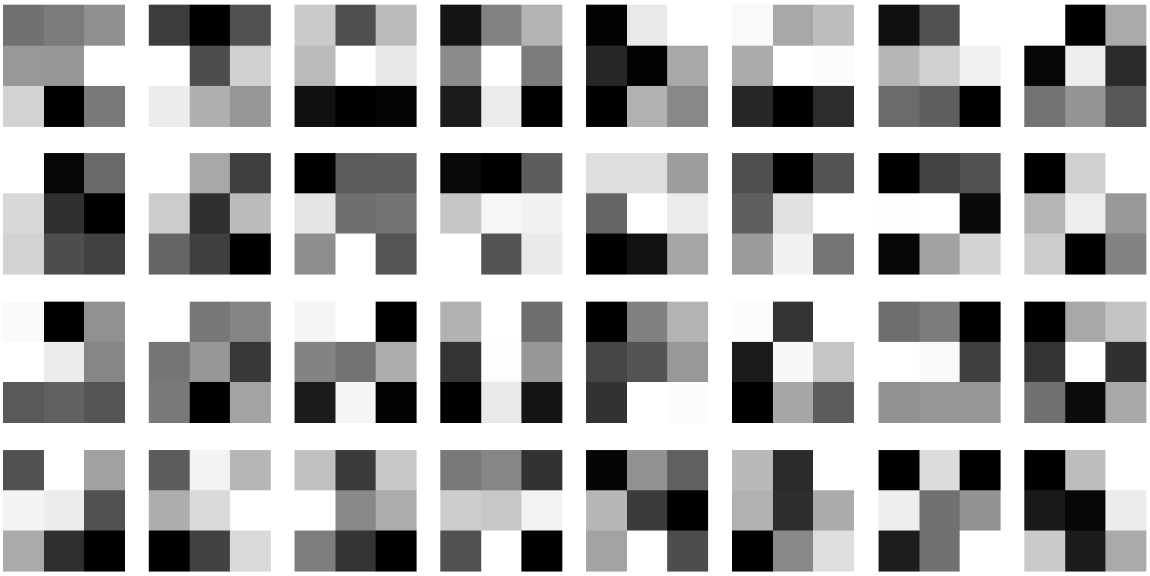
```
print(l2_weights_normalized.shape)
```

```
(32, 32, 3, 3)
```

1. Plot the the conv1 layer filters. Do you observe interesting patterns?

In [ ]:

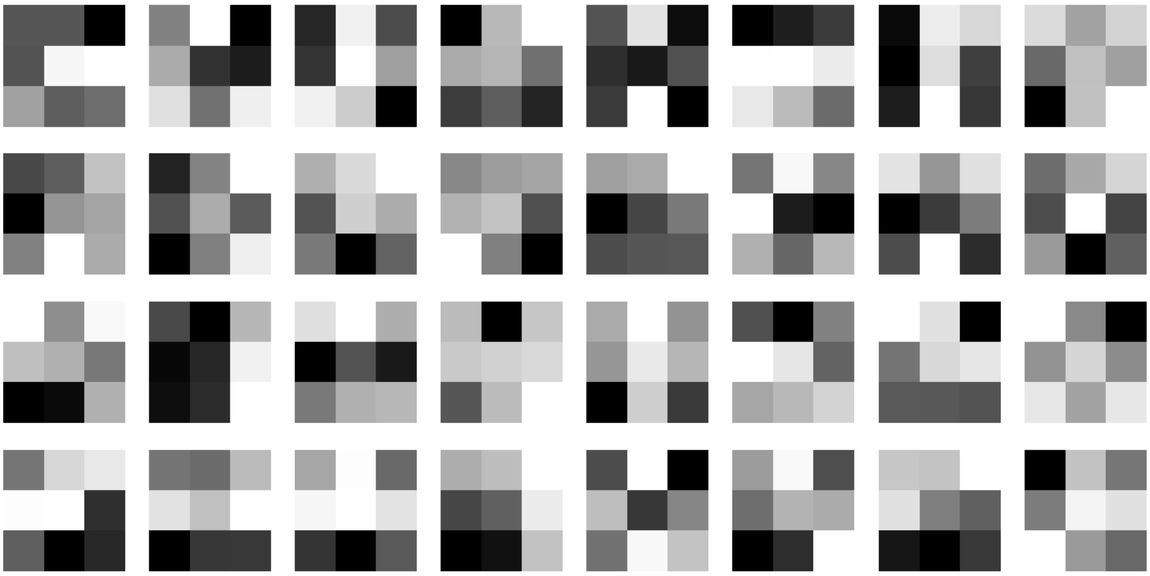
```
fig = plt.figure(figsize=(18,9))
col,row = 8,4
for i in range(1,33):
  fig.add_subplot(row,col,i)
  plt.imshow(l1_weights_normalized[i-1],cmap='gray')
  plt.axis("off")

plt.show()
```



1. Plot filters of a higher layer and compare them with conv1 layer filters.

In [ ]:

```
fig = plt.figure(figsize=(18,9))
col,row = 8,4
for i in range(1,33):
  fig.add_subplot(row,col,i)
  plt.imshow(l2_weights_normalized[i-1,10],cmap='gray')
  plt.axis("off")
plt.show()
```



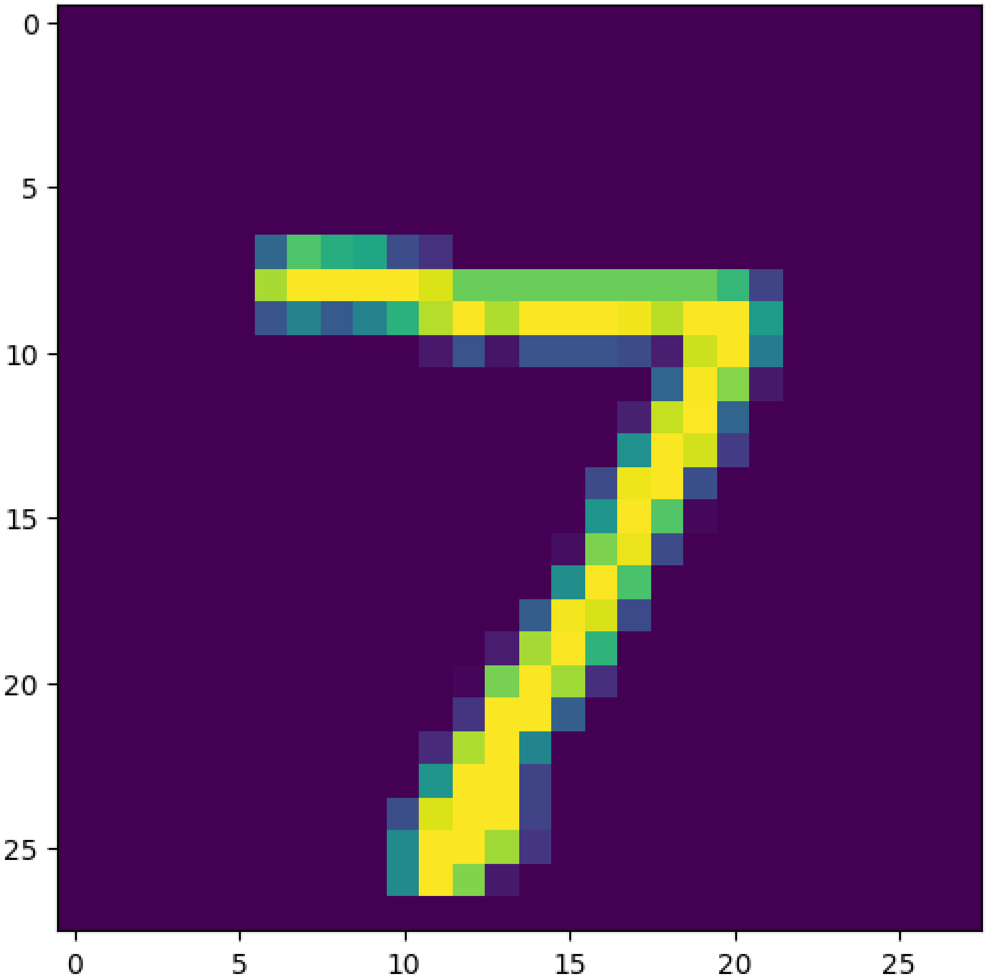
From the plotted filter images, it is very difficult to conclude anything about the patterns but as discussed in the class we know that the initial layer filters look at the lower level features as edges whereas the higher level features are captured by filters in the deeper layers. As our network is not very deep, arriving at any conclusion is difficult.

# VISUALISING ACTIVATIONS

1. Visualize the activations of the convolutional layers. What do you observe as you go deeper?

In [ ]:

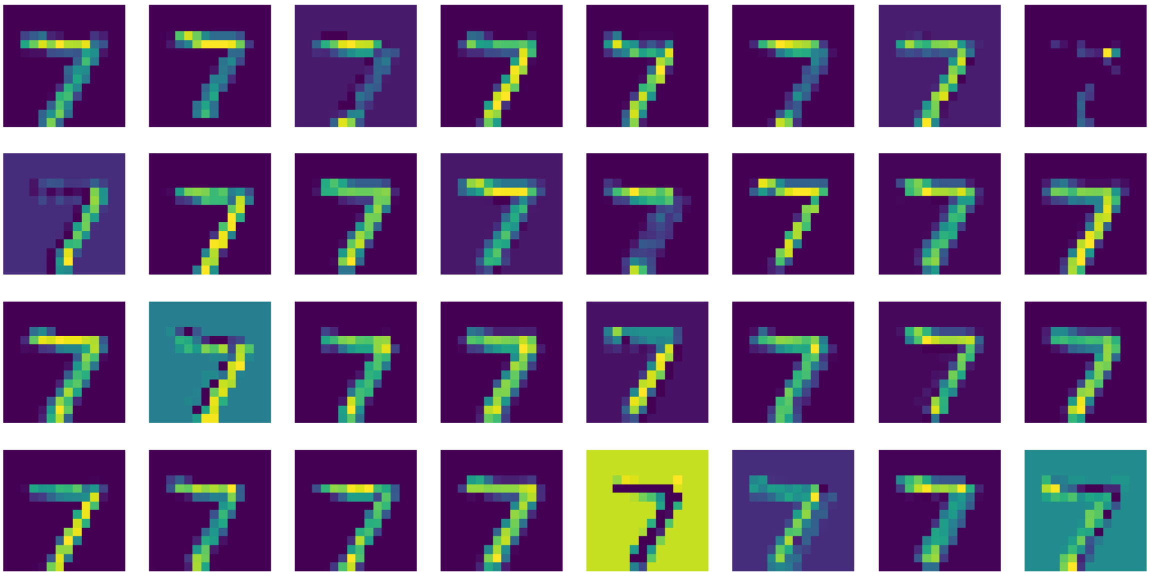
```
#visualising first test image
for img,label in test_loader:
    test_image = img[0].to(device)
    test_label = label[0].to(device)
    img = test_image.cpu().numpy().reshape(28,28)
    plt.imshow(img)
    plt.show()

    layer1_out = model.conv_layer1(torch.reshape(test_image,(1,1,28,28)))
    layer1_out_np = layer1_out.cpu().detach().numpy().reshape(32,14,14)
    print('conv_layer1 output shape:',layer1_out_np.shape)

    layer2_out = model.conv_layer2(layer1_out)
    layer2_out_np = layer2_out.cpu().detach().numpy().reshape(32,7,7)
    print('conv_layer2 output shape:',layer2_out_np.shape)
```




```
conv_layer1 output shape: (32, 14, 14)
conv_layer2 output shape: (32, 7, 7)
```

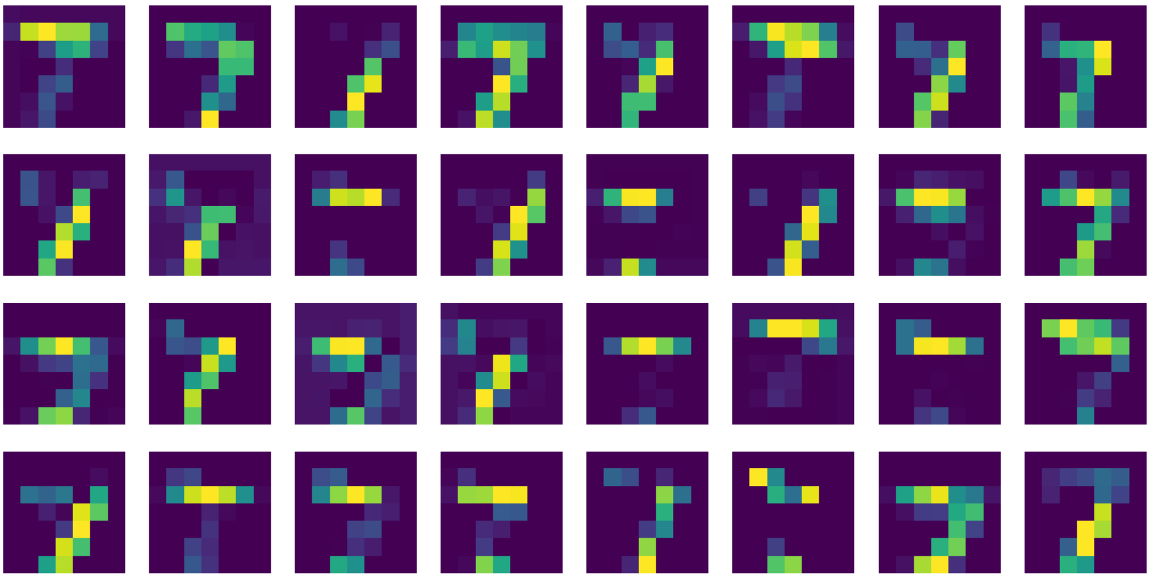
In [ ]:

```
fig = plt.figure(figsize=(18,9))
col,row = 8,4
for i in range(1,33):
  fig.add_subplot(row,col,i)
  plt.imshow(layer1_out_np[i-1])
  plt.axis("off")
plt.show()
```



In [ ]:

```
fig = plt.figure(figsize=(18,9))
col,row = 8,4
for i in range(1,33):
  fig.add_subplot(row,col,i)
  plt.imshow(layer2_out_np[i-1])
  plt.axis("off")
plt.show()
```



From above, We can see that the lower level feature maps are more focussed towards the edges and the higher level feature maps are looking at the overall image(features) of the digit.

# OCCLUDING PARTS OF IMAGE

1. Occluding parts of the image : Suppose that the network classifies an image of a digit successfully. How can we be certain that it is actually observing the main part of the digit in the image as opposed to background or something else?

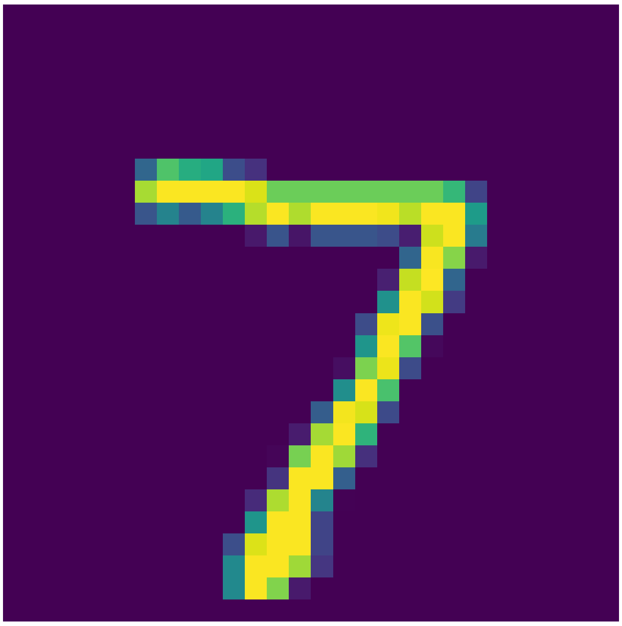
In [ ]:

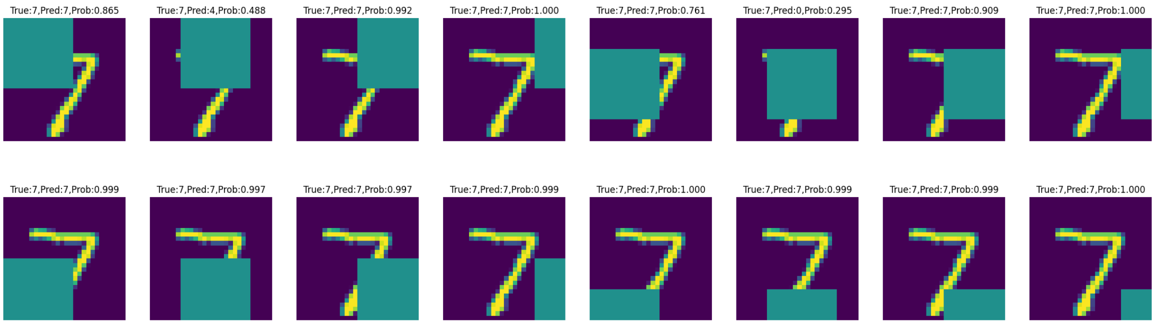
```
def occluded_image(input_image,location,patch_size=7):
  input_image_copy = input_image.copy()
  x=int(location[0])
  y=int(location[1])
  input_image_copy[x:x+patch_size,y:y+patch_size] = 0.5
  return input_image_copy
```

In [ ]:

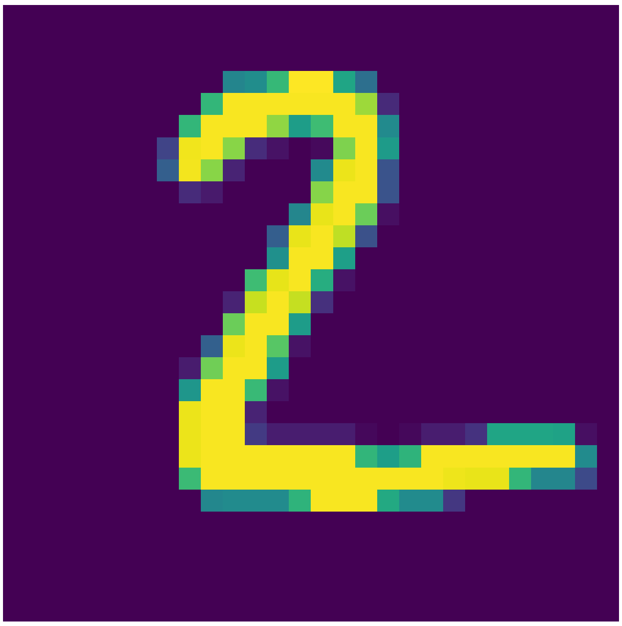
```
for img,label in test_loader:
  img_10_g=img[:10].to(device)
  label_10_g=label[:10].to(device)
  for i in range(10):
    img_1_c=img_10_g[i].cpu().numpy().reshape(28,28)
    fig=plt.figure(figsize=(4,4))
    plt.axis("off")
    plt.title("True Label = "+str(np.squeeze(label.data.cpu().numpy())[i]))
    plt.imshow(img_1_c)
    plt.show()

    fig=plt.figure(figsize=(28,8))

    row,col=2,8
    loc=[]
    for x in range(0,27,7):
      for y in range(0,27,7):
        loc.append([x,y])

    for j in range(16):
      a=fig.add_subplot(row,col,j+1)
      occ_1_img=occluded_image(img_1_c,loc[j],16)
      plt.imshow(occ_1_img)
      plt.axis("off")

      img_1_torch = torch.from_numpy(occ_1_img.reshape(1,1,28,28))
      img_1_gpu = img_1_torch.to(device)
      prediction_vect_gpu = model(img_1_gpu)
      prediction_vect_cpu = prediction_vect_gpu.detach().cpu().numpy()
      prediction_vect_cpu=np.squeeze(prediction_vect_cpu)

      prediction_prob_vect=np.exp(prediction_vect_cpu)/np.sum(np.exp(prediction_
vect_cpu))
      predicted_digit=np.argmax(prediction_prob_vect)
      prediction_probability=prediction_prob_vect[predicted_digit]
      a.set_title('True:{},Pred:{},Prob:{:.3f}'.format(label.data[i],predicted_d
igit,prediction_probability))

    plt.show()
```

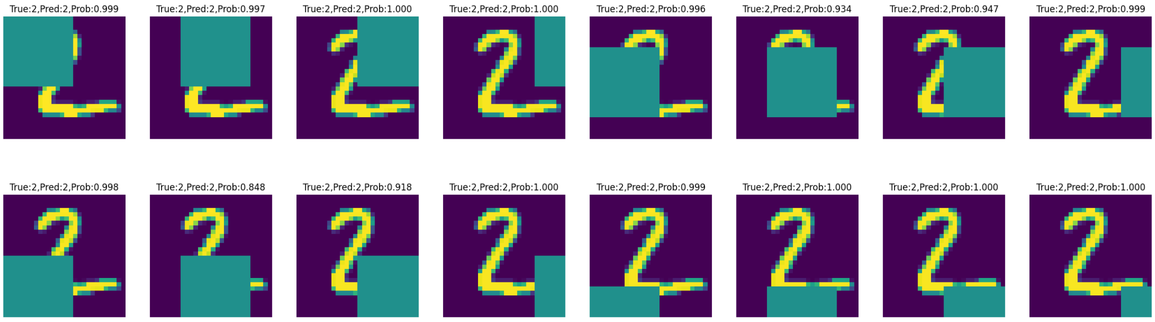
True Label = 7

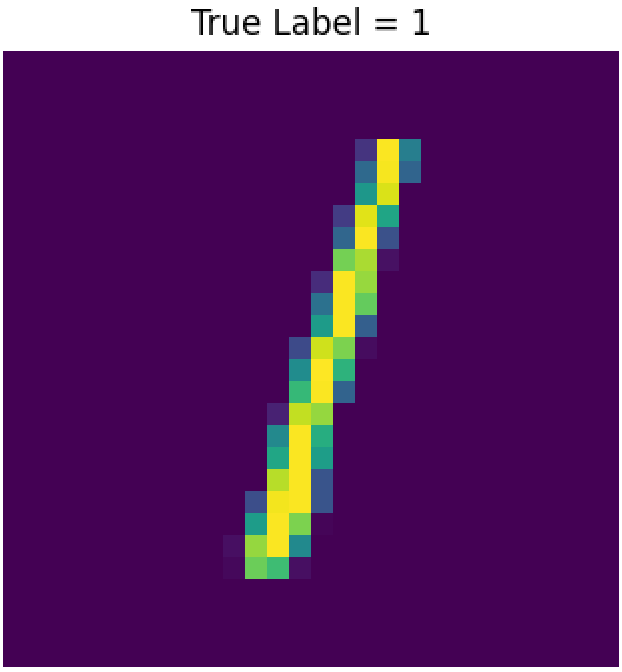


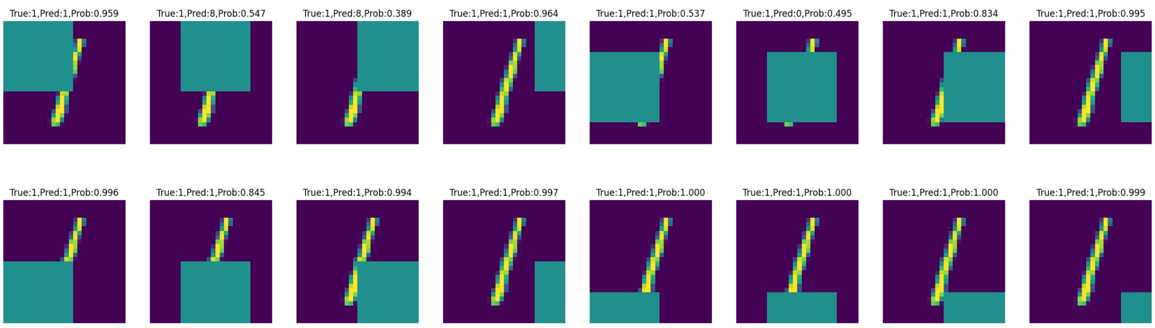


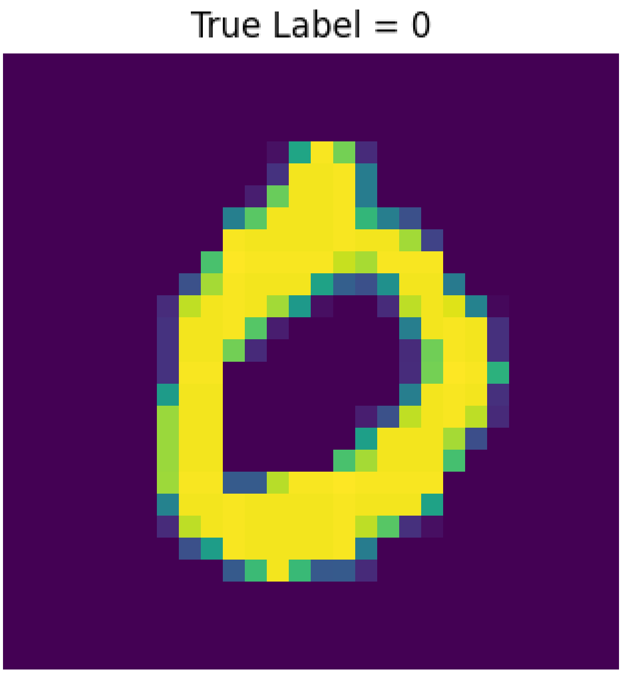

True Label = 2

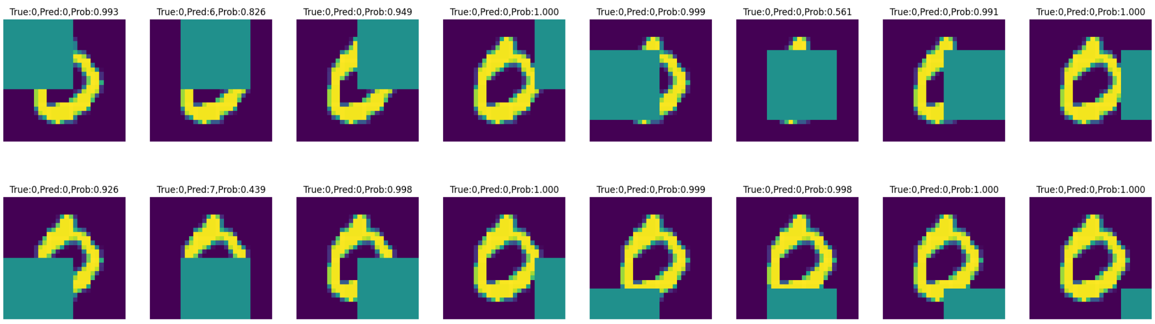


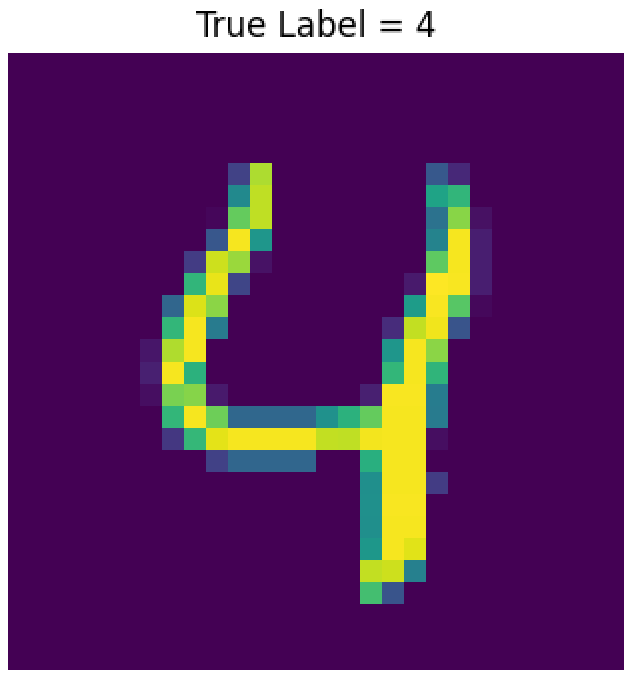

True:2,Pred:2,Prob:0.999
True:2,Pred:2,Prob:0.997
True:2,Pred:2,Prob:1.000
True:2,Pred:2,Prob:1.000
True:2,Pred:2,Prob:0.996
True:2,Pred:2,Prob:0.934
True:2,Pred:2,Prob:0.947
True:2,Pred:2,Prob:0.999
True:2,Pred:2,Prob:0.998
True:2,Pred:2,Prob:0.848
True:2,Pred:2,Prob:0.918
True:2,Pred:2,Prob:1.000
True:2,Pred:2,Prob:0.999
True:2,Pred:2,Prob:1.000
True:2,Pred:2,Prob:1.000
True:2,Pred:2,Prob:1.000

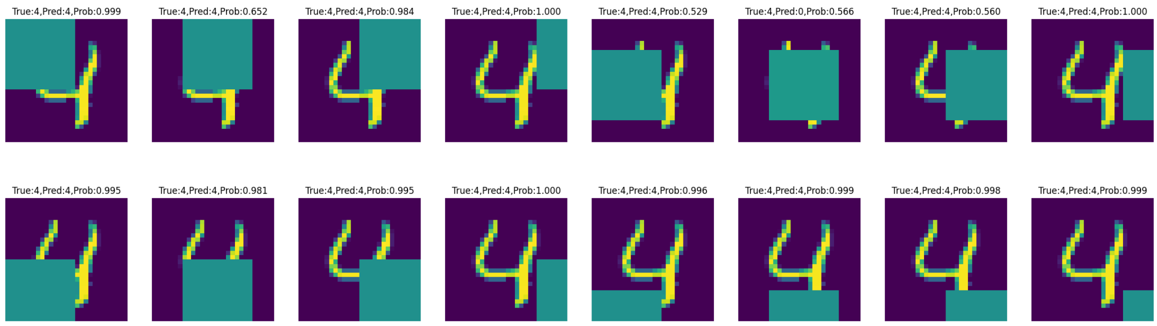

True Label = 1

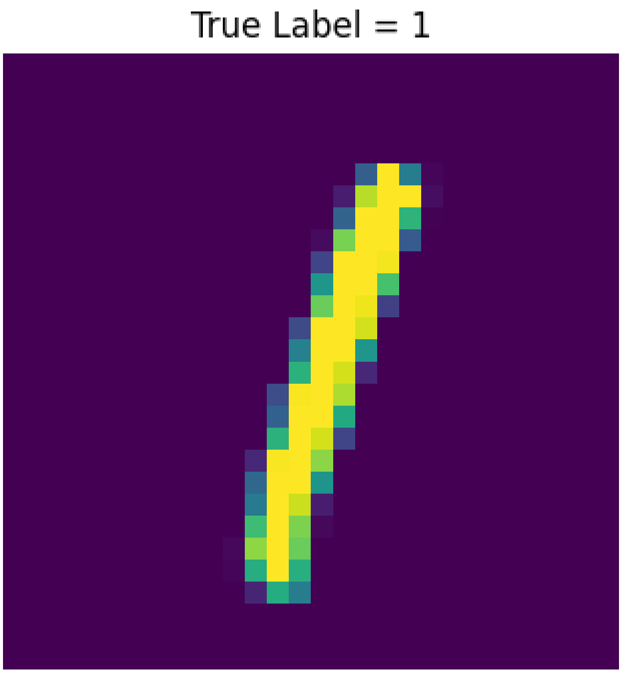

True:1,Pred:1,Prob:0.959
True:1,Pred:8,Prob:0.547
True:1,Pred:8,Prob:0.389
True:1,Pred:1,Prob:0.964
True:1,Pred:1,Prob:0.537
True:1,Pred:0,Prob:0.495
True:1,Pred:1,Prob:0.834
True:1,Pred:1,Prob:0.995
True:1,Pred:1,Prob:0.996
True:1,Pred:1,Prob:0.845
True:1,Pred:1,Prob:0.994
True:1,Pred:1,Prob:0.997
True:1,Pred:1,Prob:1.000
True:1,Pred:1,Prob:1.000
True:1,Pred:1,Prob:1.000
True:1,Pred:1,Prob:0.999

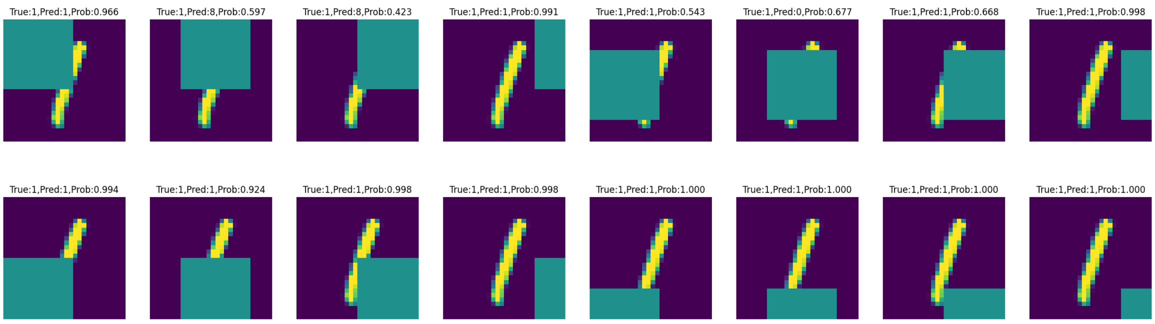

True Label = 0

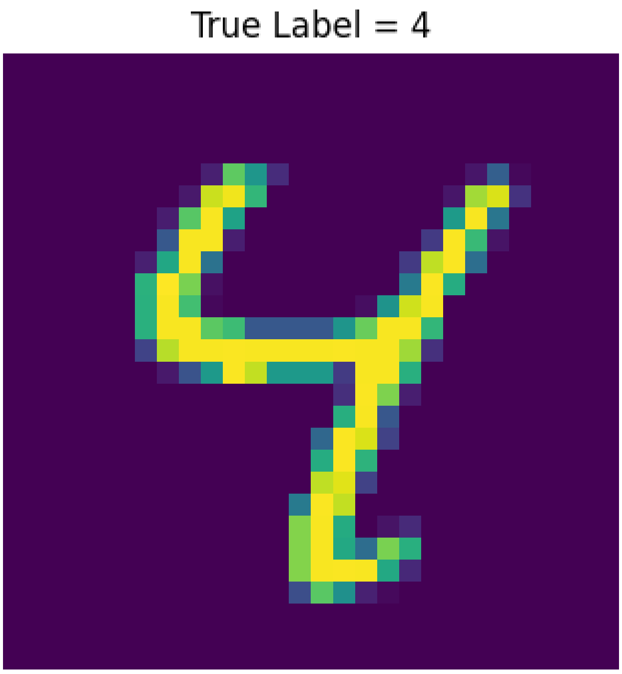

True:0,Pred:0,Prob:0.993
True:0,Pred:6,Prob:0.826
True:0,Pred:0,Prob:0.949
True:0,Pred:0,Prob:1.000
True:0,Pred:0,Prob:0.999
True:0,Pred:0,Prob:0.561
True:0,Pred:0,Prob:0.991
True:0,Pred:0,Prob:1.000
True:0,Pred:0,Prob:0.926
True:0,Pred:7,Prob:0.439
True:0,Pred:0,Prob:0.998
True:0,Pred:0,Prob:1.000
True:0,Pred:0,Prob:0.999
True:0,Pred:0,Prob:0.998
True:0,Pred:0,Prob:1.000
True:0,Pred:0,Prob:1.000

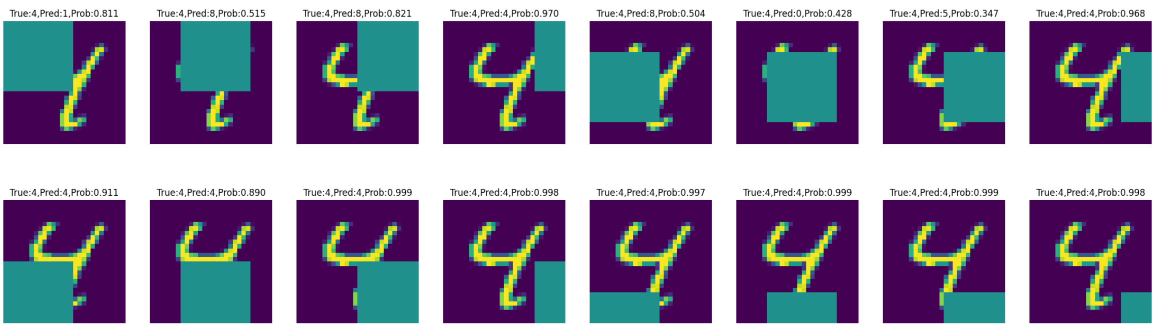

True Label = 4

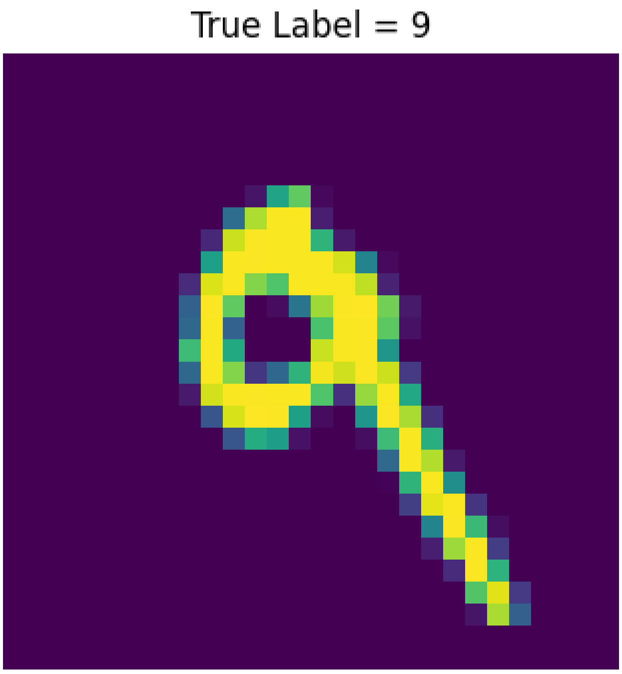

True:4,Pred:4,Prob:0.999
True:4,Pred:4,Prob:0.652
True:4,Pred:4,Prob:0.984
True:4,Pred:4,Prob:1.000
True:4,Pred:4,Prob:0.529
True:4,Pred:0,Prob:0.566
True:4,Pred:4,Prob:0.560
True:4,Pred:4,Prob:1.000
True:4,Pred:4,Prob:0.995
True:4,Pred:4,Prob:0.981
True:4,Pred:4,Prob:0.995
True:4,Pred:4,Prob:1.000
True:4,Pred:4,Prob:0.996
True:4,Pred:4,Prob:0.999
True:4,Pred:4,Prob:0.998
True:4,Pred:4,Prob:0.999

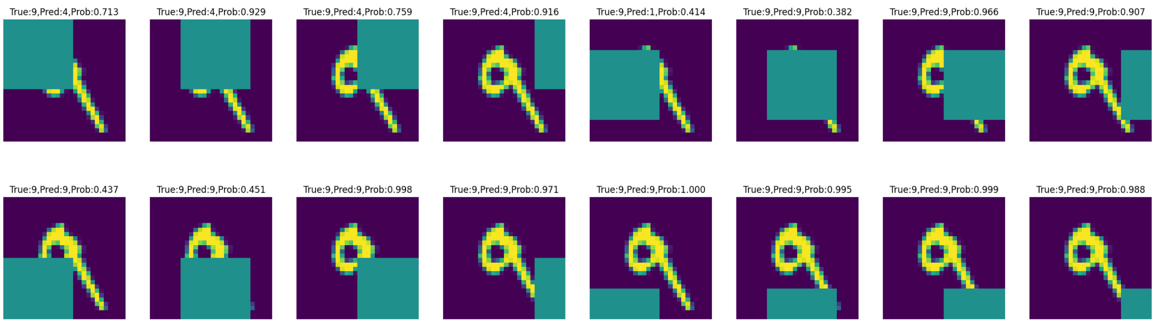

True Label = 1

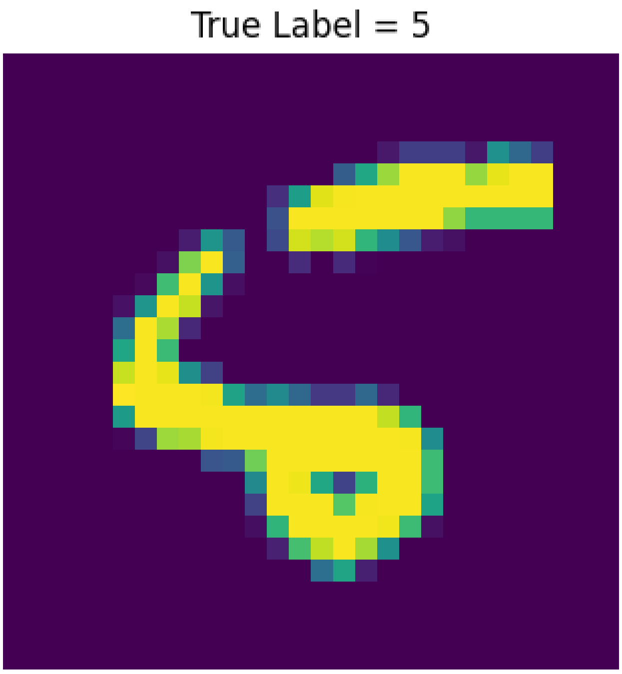

True:1,Pred:1,Prob:0.966
True:1,Pred:8,Prob:0.597
True:1,Pred:8,Prob:0.423
True:1,Pred:1,Prob:0.991
True:1,Pred:1,Prob:0.543
True:1,Pred:0,Prob:0.677
True:1,Pred:1,Prob:0.668
True:1,Pred:1,Prob:0.998
True:1,Pred:1,Prob:0.994
True:1,Pred:1,Prob:0.924
True:1,Pred:1,Prob:0.998
True:1,Pred:1,Prob:0.998
True:1,Pred:1,Prob:1.000
True:1,Pred:1,Prob:1.000
True:1,Pred:1,Prob:1.000
True:1,Pred:1,Prob:1.000

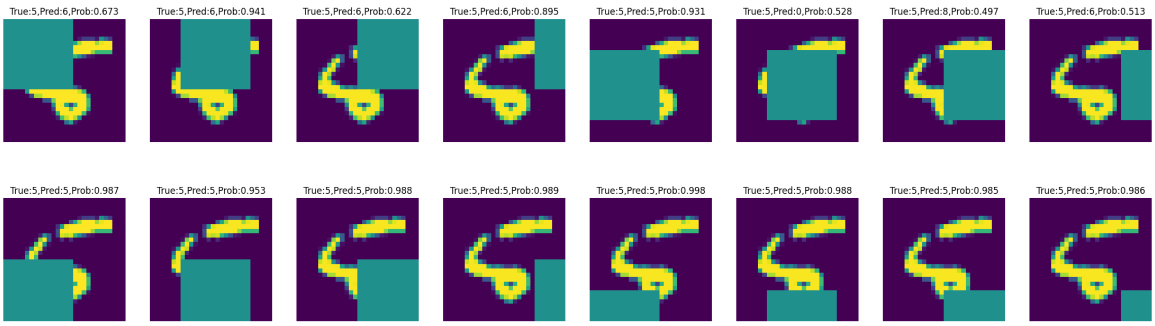

True Label = 4

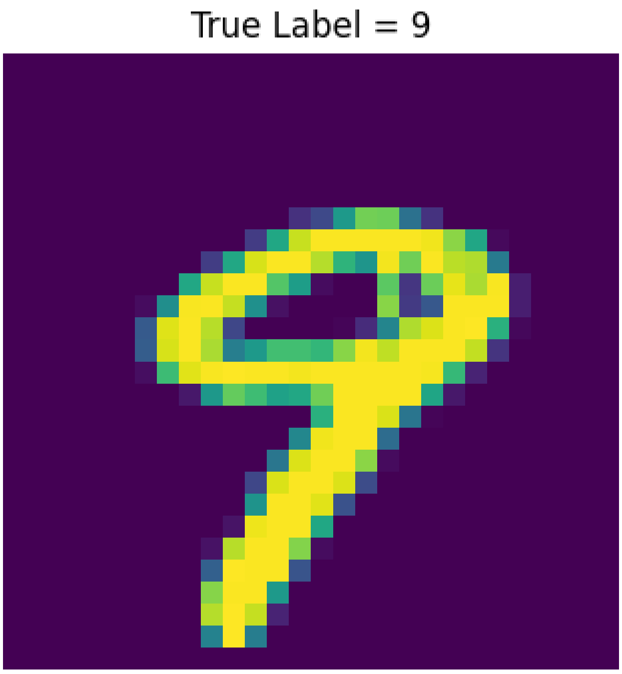

True:4,Pred:1,Prob:0.811
True:4,Pred:8,Prob:0.515
True:4,Pred:8,Prob:0.821
True:4,Pred:4,Prob:0.970
True:4,Pred:8,Prob:0.504
True:4,Pred:0,Prob:0.428
True:4,Pred:5,Prob:0.347
True:4,Pred:4,Prob:0.968
True:4,Pred:4,Prob:0.911
True:4,Pred:4,Prob:0.890
True:4,Pred:4,Prob:0.999
True:4,Pred:4,Prob:0.998
True:4,Pred:4,Prob:0.997
True:4,Pred:4,Prob:0.999
True:4,Pred:4,Prob:0.999
True:4,Pred:4,Prob:0.998

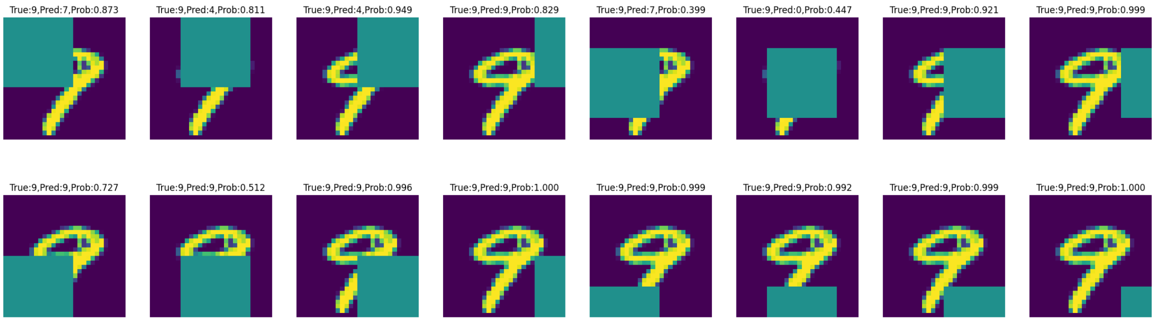

True Label = 9


True:9,Pred:4,Prob:0.713
True:9,Pred:4,Prob:0.929
True:9,Pred:4,Prob:0.759
True:9,Pred:4,Prob:0.916
True:9,Pred:1,Prob:0.414
True:9,Pred:9,Prob:0.382
True:9,Pred:9,Prob:0.966
True:9,Pred:9,Prob:0.907
True:9,Pred:9,Prob:0.437
True:9,Pred:9,Prob:0.451
True:9,Pred:9,Prob:0.998
True:9,Pred:9,Prob:0.971
True:9,Pred:9,Prob:1.000
True:9,Pred:9,Prob:0.995
True:9,Pred:9,Prob:0.999
True:9,Pred:9,Prob:0.988


True Label = 5


True:5,Pred:6,Prob:0.673
True:5,Pred:6,Prob:0.941
True:5,Pred:6,Prob:0.622
True:5,Pred:6,Prob:0.895
True:5,Pred:5,Prob:0.931
True:5,Pred:0,Prob:0.528
True:5,Pred:8,Prob:0.497
True:5,Pred:6,Prob:0.513
True:5,Pred:5,Prob:0.987
True:5,Pred:5,Prob:0.953
True:5,Pred:5,Prob:0.988
True:5,Pred:5,Prob:0.989
True:5,Pred:5,Prob:0.998
True:5,Pred:5,Prob:0.988
True:5,Pred:5,Prob:0.985
True:5,Pred:5,Prob:0.986




As can be seen from above results, As we occlude the digit image, the probability of the actual prediction image drops, and sometimes the model prediction goes wrong, which shows that our model is actually looking at the meaningful features for digit classification.

# NON TARGETED ATTACK

In [ ]:

```
for child in model.children():
  for param in child.parameters():
    param.requires_grad = False
```

In [ ]:

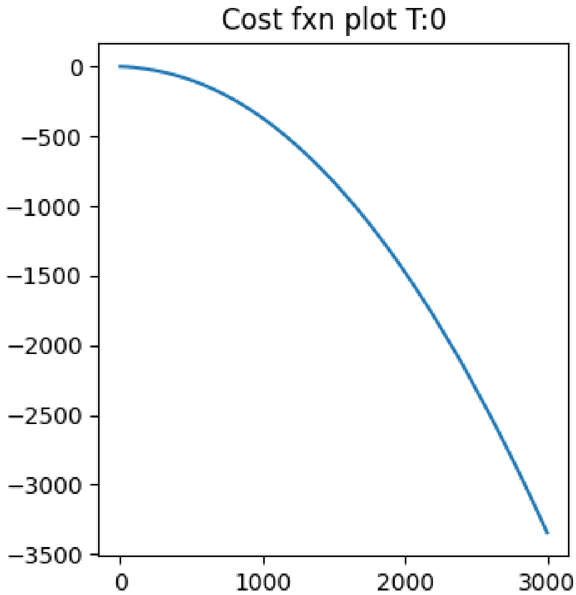
```
img_grid = np.zeros((10,28,28))
cost={}
for i in range(10):
  noise = np.random.normal(loc=0.5,scale=0.05,size=(1,1,28,28)).astype(np.float3
2)
  noise = torch.from_numpy(noise)
  x_var = torch.tensor(noise.type(torch.cuda.FloatTensor),requires_grad=True, de
vice='cuda')
  optimizer = torch.optim.SGD([x_var],lr=0.0001)
  cost[str(i)]=[]
  for j in range(3000):
    model.zero_grad()
    out_var=model(x_var)
    loss=-out_var[0][i]
    loss.backward()
    optimizer.step()


    cost[str(i)].append(loss.item())

  n_img = x_var.cpu().detach().numpy()
  img_grid[i,:,:] = normalize(n_img)
```

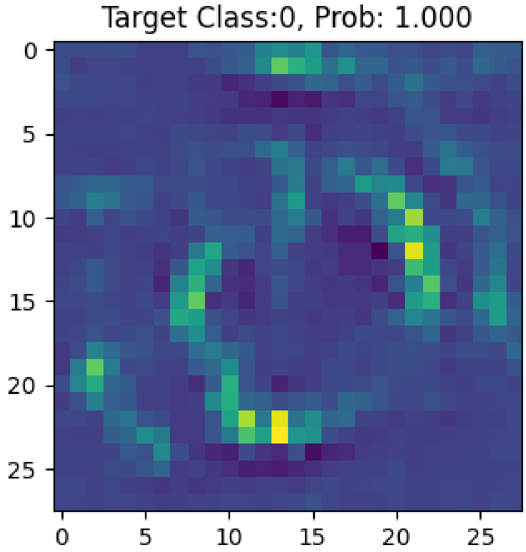
```
/tmp/ipykernel_984428/1069001644.py:6: UserWarning: To copy construc
t from a tensor, it is recommended to use sourceTensor.clone().detac
h() or sourceTensor.clone().detach().requires_grad_(True), rather th
an torch.tensor(sourceTensor).
  x_var = torch.tensor(noise.type(torch.cuda.FloatTensor),requires_g
rad=True, device='cuda')
```

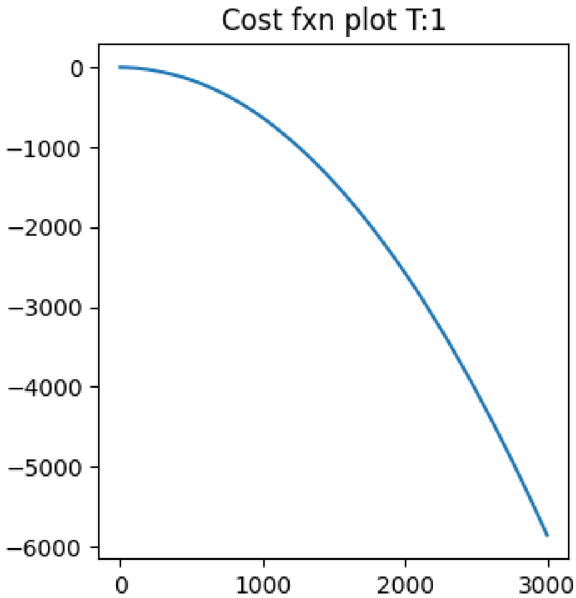
1. Show the generated image for each of the MNIST classes.

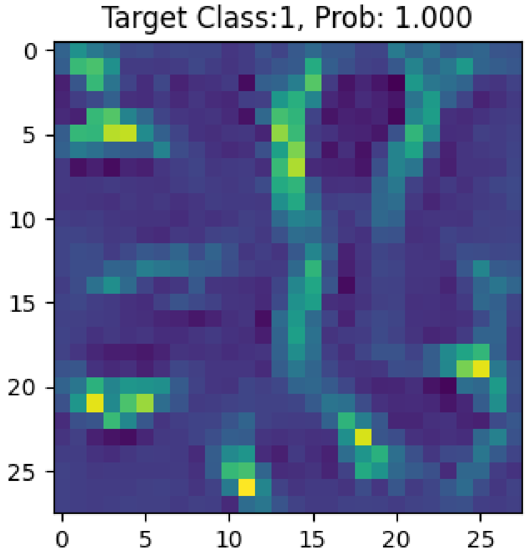
In [ ]:

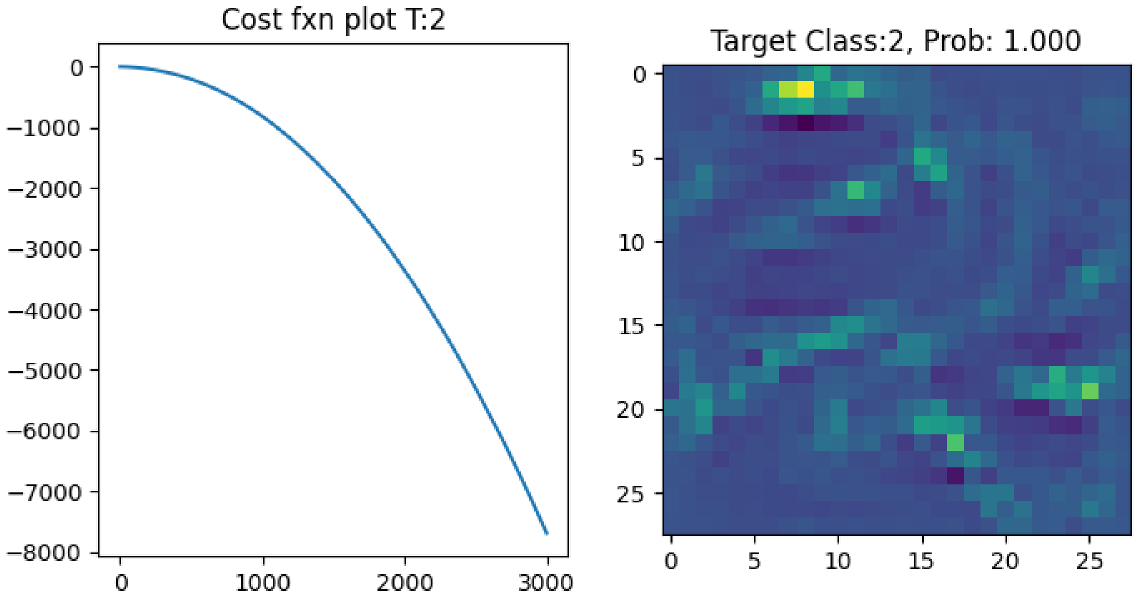
```
for i in range(10):

  fig = plt.figure(figsize=(8,4))
  fig.tight_layout()
  col,row = 2,1

  in_image=torch.reshape(torch.from_numpy(img_grid[i].astype(np.float32)),(1,1,2
8,28)).to(device)
#   print(in_image.size)
  prediction_vect_gpu=model(in_image)
  prediction_vect_cpu = prediction_vect_gpu.detach().cpu().numpy()
  prediction_vect_cpu=np.squeeze(prediction_vect_cpu)-np.max(prediction_vect_cpu
) ##subtracting max value from each entry to avoid overflow in exponential.

  prediction_prob_vect=np.exp(prediction_vect_cpu)/np.sum(np.exp(prediction_vect
_cpu))
  predicted_digit=np.argmax(prediction_prob_vect)
  prediction_probability=prediction_prob_vect[predicted_digit]


  a = fig.add_subplot(row,col,1 )
  a.title.set_text('Cost fxn plot T:{}'.format(i))
  a.plot(np.arange(0,len(cost[str(i)])),np.array(cost[str(i)]))

  b= fig.add_subplot(row,col,2)
  b.title.set_text('Target Class:{}, Prob: {:.3f}'.format(predicted_digit,predic
tion_probability))
  plt.imshow(img_grid[i])

plt.show()
```

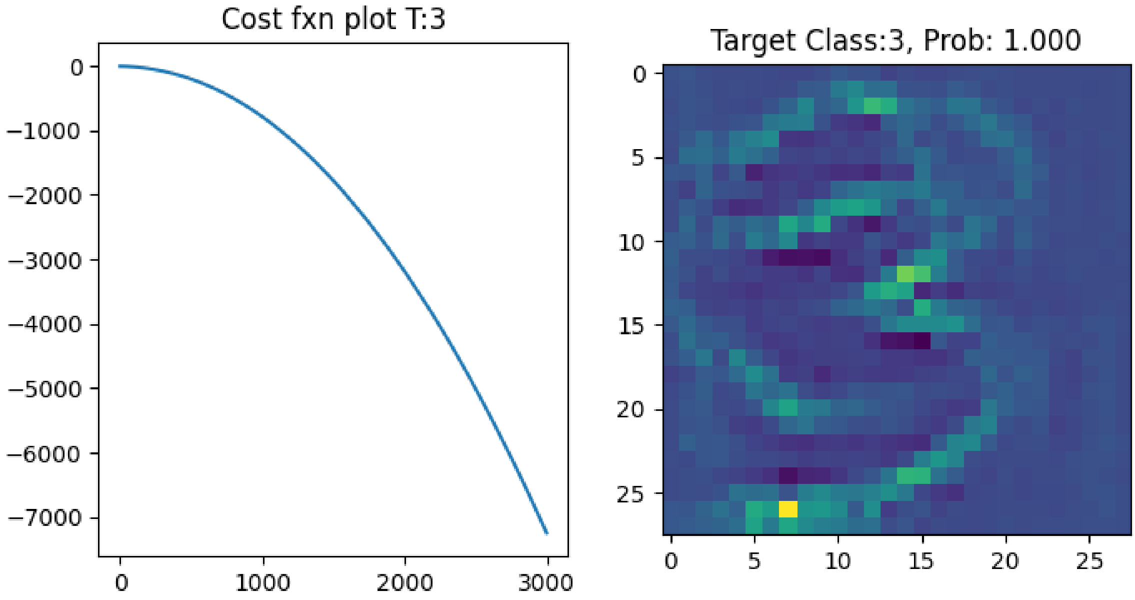

Cost fxn plot T:0
0
−500
−1000
−1500
−2000
−2500
−3000
−3500
0
1000
2000
3000

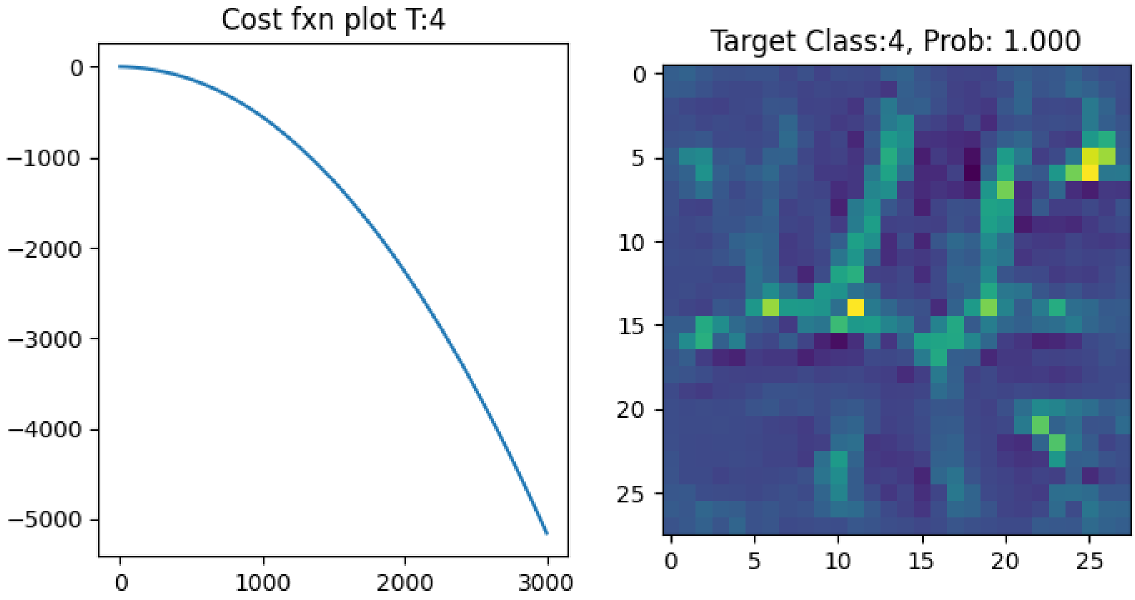

Target Class:0, Prob: 1.000
0
5
10
15
20
25

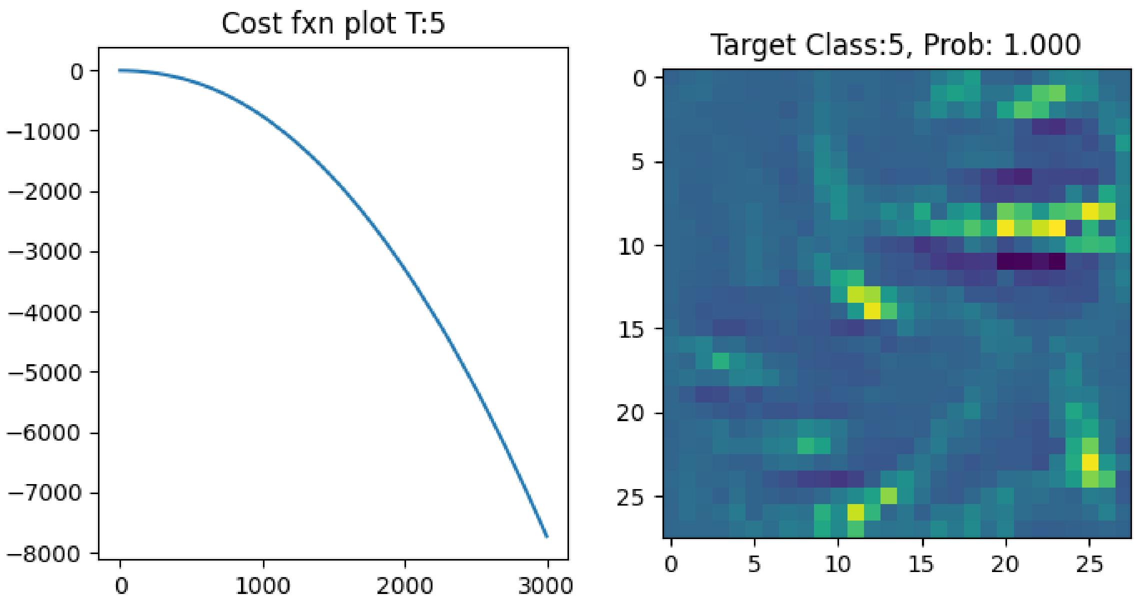

Cost fxn plot T:1
0
−1000
−2000
−3000
−4000
−5000
−6000
0
1000
2000
3000

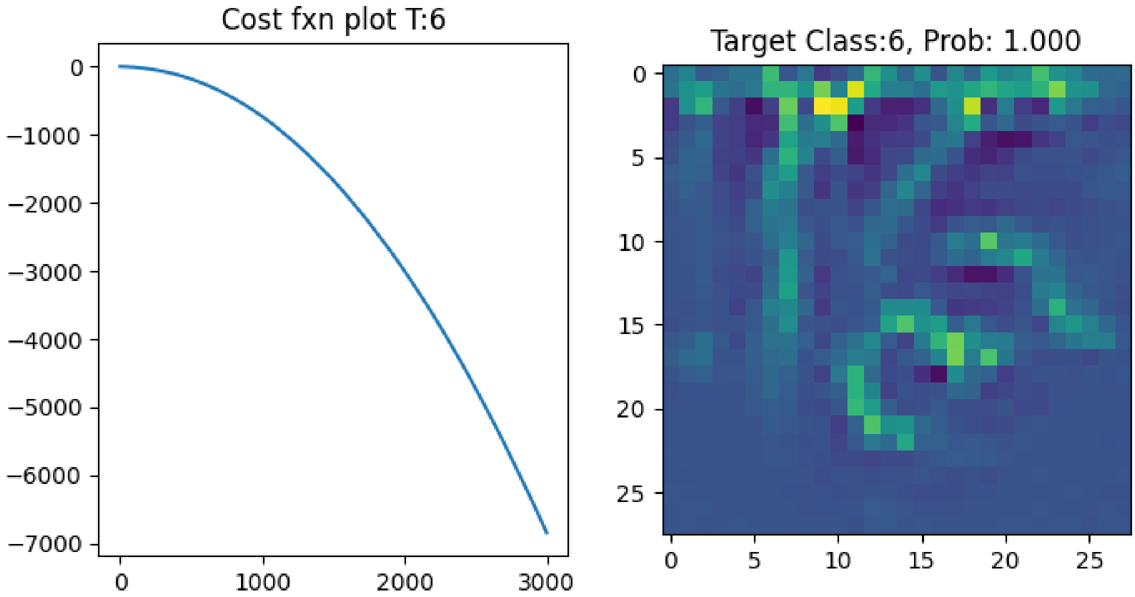

Target Class:1, Prob: 1.000
0
5
10
15
20
25

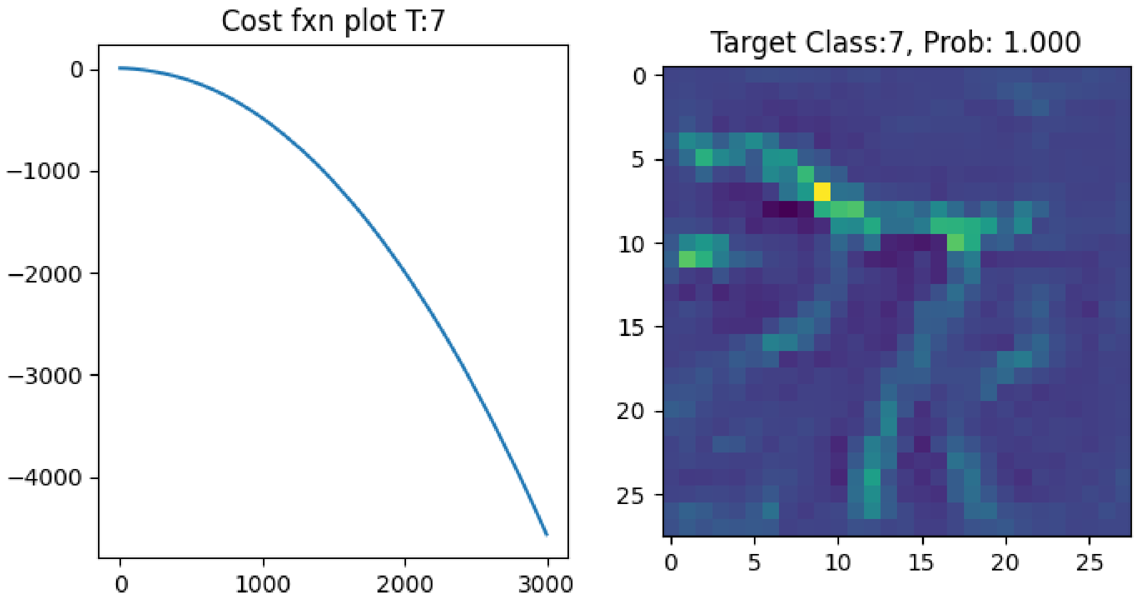

Cost fxn plot T:2
0
−1000
−2000
−3000
−4000
−5000
−6000
−7000
−8000
0
1000
2000
3000
Target Class:2, Prob: 1.000
0
5
10
15
20
25

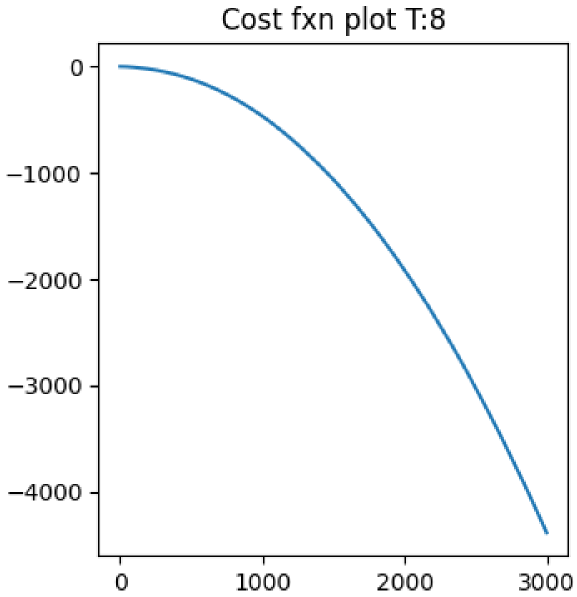

Cost fxn plot T:3
0
−1000
−2000
−3000
−4000
−5000
−6000
−7000
0
1000
2000
3000
Target Class:3, Prob: 1.000
0
5
10
15
20
25

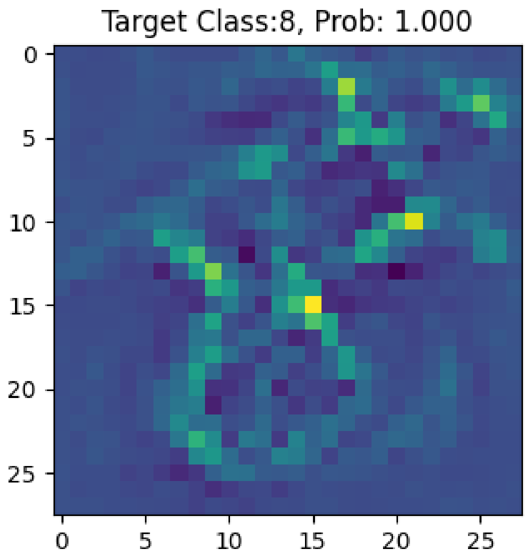

Cost fxn plot T:4
0
−1000
−2000
−3000
−4000
−5000
0
1000
2000
3000
Target Class:4, Prob: 1.000
0
5
10
15
20
25
0
5
10
15
20
25

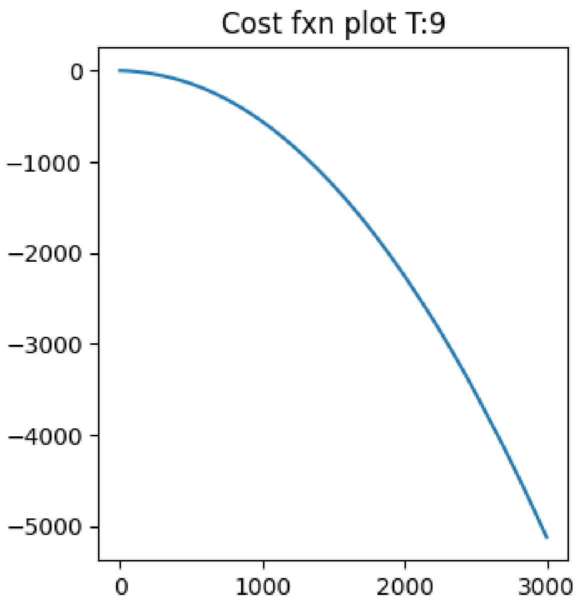

Cost fxn plot T:5
0
−1000
−2000
−3000
−4000
−5000
−6000
−7000
−8000
0
1000
2000
3000
Target Class:5, Prob: 1.000
0
5
10
15
20
25
0
5
10
15
20
25

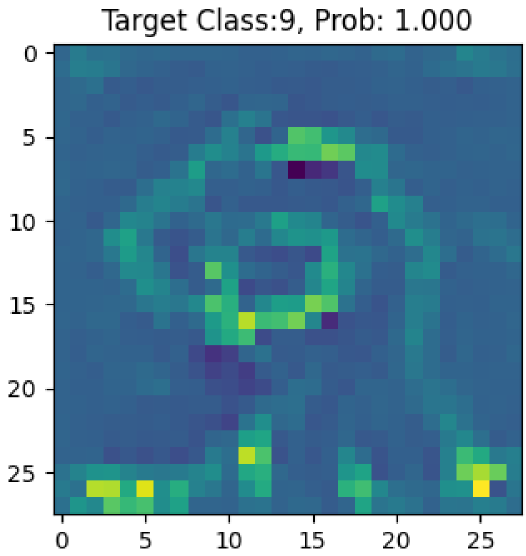

Cost fxn plot T:6
0
−1000
−2000
−3000
−4000
−5000
−6000
−7000
0
1000
2000
3000
Target Class:6, Prob: 1.000
0
5
10
15
20
25


Cost fxn plot T:7
0
−1000
−2000
−3000
−4000
0
1000
2000
3000
Target Class:7, Prob: 1.000
0
5
10
15
20
25


Cost fxn plot T:8
0
−1000
−2000
−3000
−4000
0
1000
2000
3000


Target Class:8, Prob: 1.000
0
5
10
15
20
25
0
5
10
15
20
25


Cost fxn plot T:9
0
−1000
−2000
−3000
−4000
−5000
0
1000
2000
3000


Target Class:9, Prob: 1.000
0
5
10
15
20
25
0
5
10
15
20
25

1. Is the network always predicting targetClass with high confidence for the generated images?

**Yes**

1. Do the generated images look like a number? If not, can you think of some reason?

**No**, Because we are generating images in accordance with how our model looks at images to classify them. So, even though the model classifys it as a digit but the image makes no sense to a human.

1. Plot the cost function. Is it increasing or decreasing?

As I am increasing the probability of the classification of the digit, I am using a loss function to be negative as that given in assignment, Since our optimizers always minimises the cost. So for my case, The cost function is **decreasing**. As per the cost function given in the assignment it would be negative of this plot i.e. 'increasing'

Identifying a Test Image set which contains all classes for proper testing

In [ ]:

```
for images,labels in test_loader:
    print(torch.unique(labels[9705:9715]))
    for k in range(9705,9715):
        print(k, labels[k])
```

```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
9705 tensor(1)
9706 tensor(2)
9707 tensor(3)
9708 tensor(4)
9709 tensor(5)
9710 tensor(6)
9711 tensor(7)
9712 tensor(8)
9713 tensor(9)
9714 tensor(0)
```

# TARGETED ATTACK

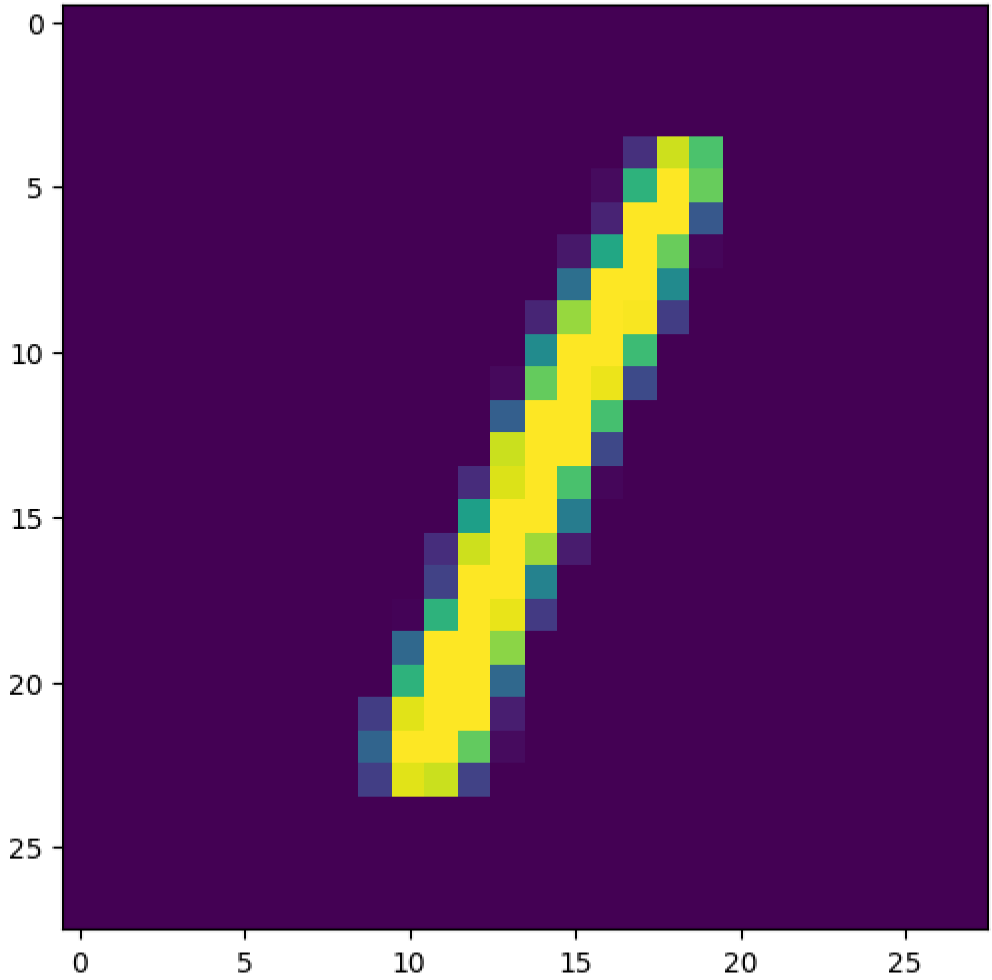
1. Show the generated image for each of the MNIST classes. Do the generated images now look like a number?

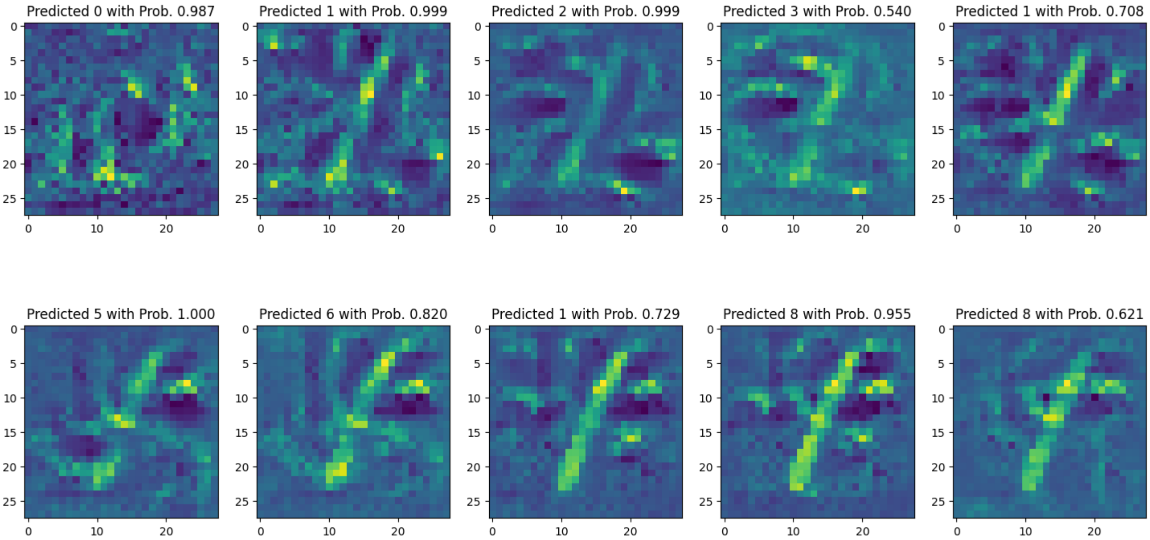
The below cell output shows the generated images for each of classes being falsely classified as all the classes. **Yes**, now the images look like a digit.

In [ ]:

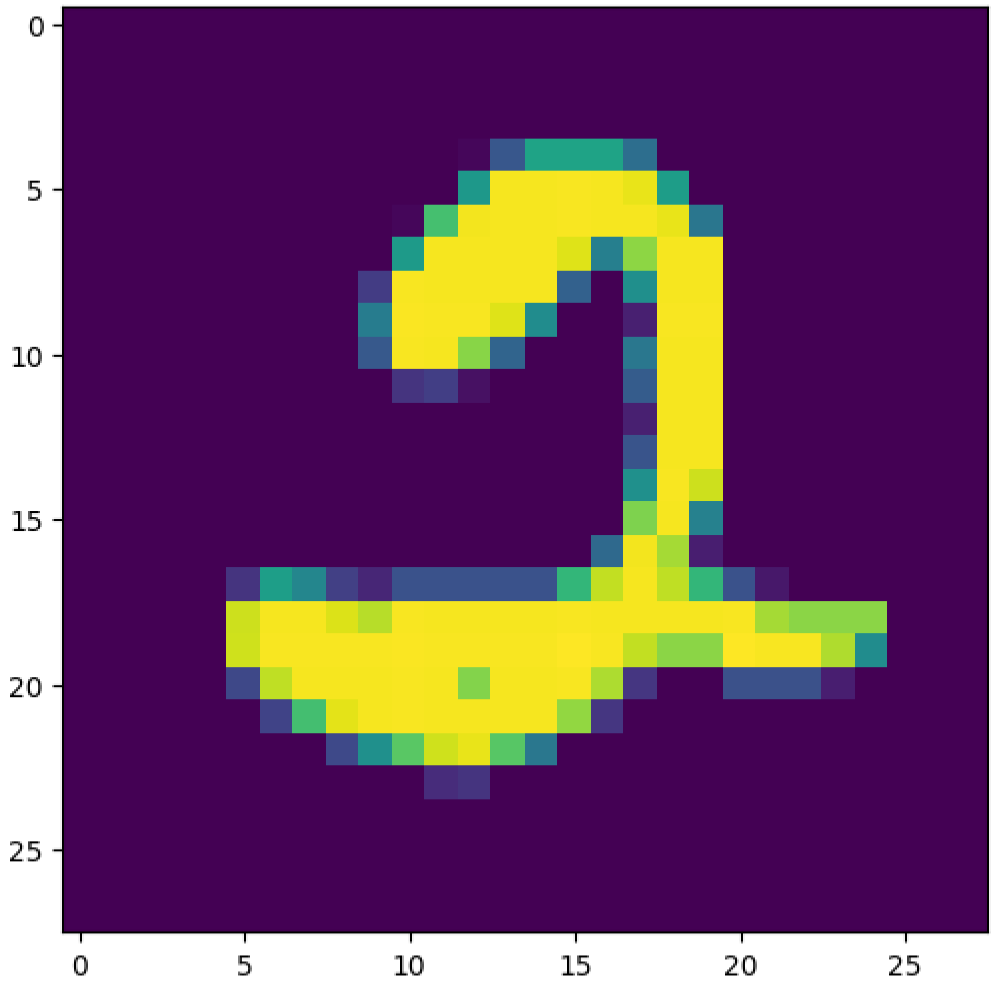
```
for images,labels in test_loader:
    for k in range(9705,9715):

        beta=70

        col,row = 5,2
        target_img=images[k].to(device)
        fig = plt.figure(figsize=(6,6))
        plt.imshow(torch.squeeze(images[k]).cpu().numpy())

        fig = plt.figure(figsize=(18,9))
        fig.tight_layout()

        img_= np.random.normal(loc=0.5,scale=0.05,size=(1,1,28,28)).astype(np.fl
oat32)
        img_=torch.from_numpy(img_)
        img_=img_.to(device)
        img_var=torch.tensor(img_.type(torch.cuda.FloatTensor),requires_grad=Tru
e,device='cuda')
        optimizer=torch.optim.SGD([img_var],lr=0.0001)
        for target_label in range(10):
            for j in range(5000):
                optimizer.zero_grad()
                out_var=model(img_var)
                mse_loss=nn.MSELoss()

                loss=-1*out_var[0][target_label]+beta*mse_loss(img_var,target_im
g)
                loss.backward()
                optimizer.step()

            img_var_cpu=img_var.cpu().detach().numpy()
            img_var_cpu_normalized=normalize(img_var_cpu)


            prediction_vect_gpu=model(img_var)
            prediction_vect_cpu = prediction_vect_gpu.detach().cpu().numpy()
            prediction_vect_cpu=np.squeeze(prediction_vect_cpu)

            prediction_prob_vect=np.exp(prediction_vect_cpu)/np.sum(np.exp(predi
ction_vect_cpu))
            predicted_digit=np.argmax(prediction_prob_vect)
            prediction_probability=prediction_prob_vect[predicted_digit]

            a= fig.add_subplot(row,col,target_label+1)
            a.title.set_text('Predicted {} with Prob. {:.3f}'.format(predicted_d
igit,prediction_probability))
            plt.imshow(np.squeeze(img_var_cpu_normalized))
```

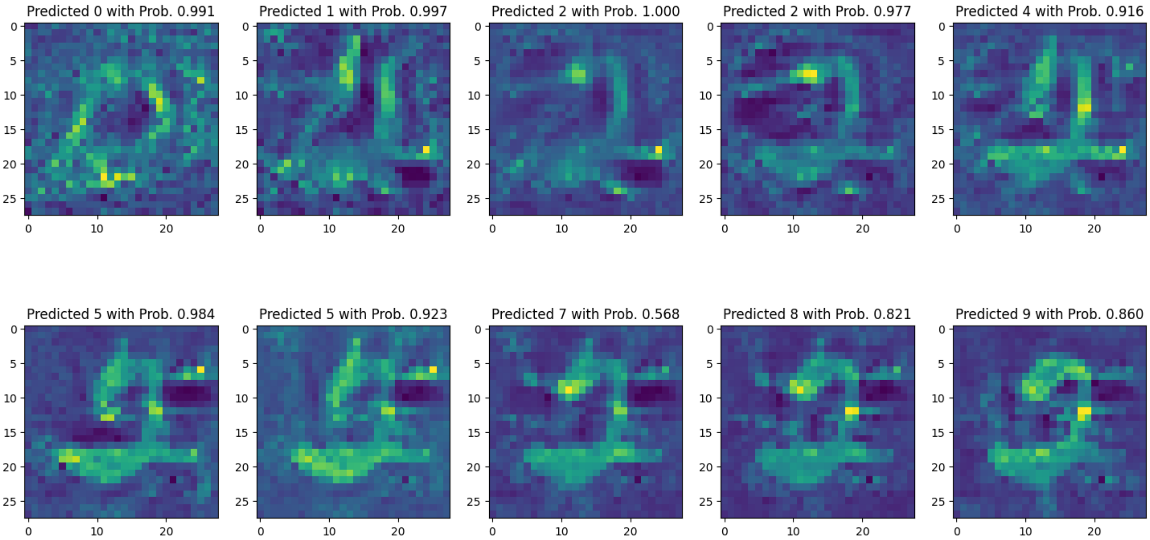
```
/tmp/ipykernel_984428/2746526485.py:17: UserWarning: To copy constru
ct from a tensor, it is recommended to use sourceTensor.clone().deta
ch() or sourceTensor.clone().detach().requires_grad_(True), rather t
han torch.tensor(sourceTensor).
  img_var=torch.tensor(img_.type(torch.cuda.FloatTensor),requires_gr
ad=True,device='cuda')
```

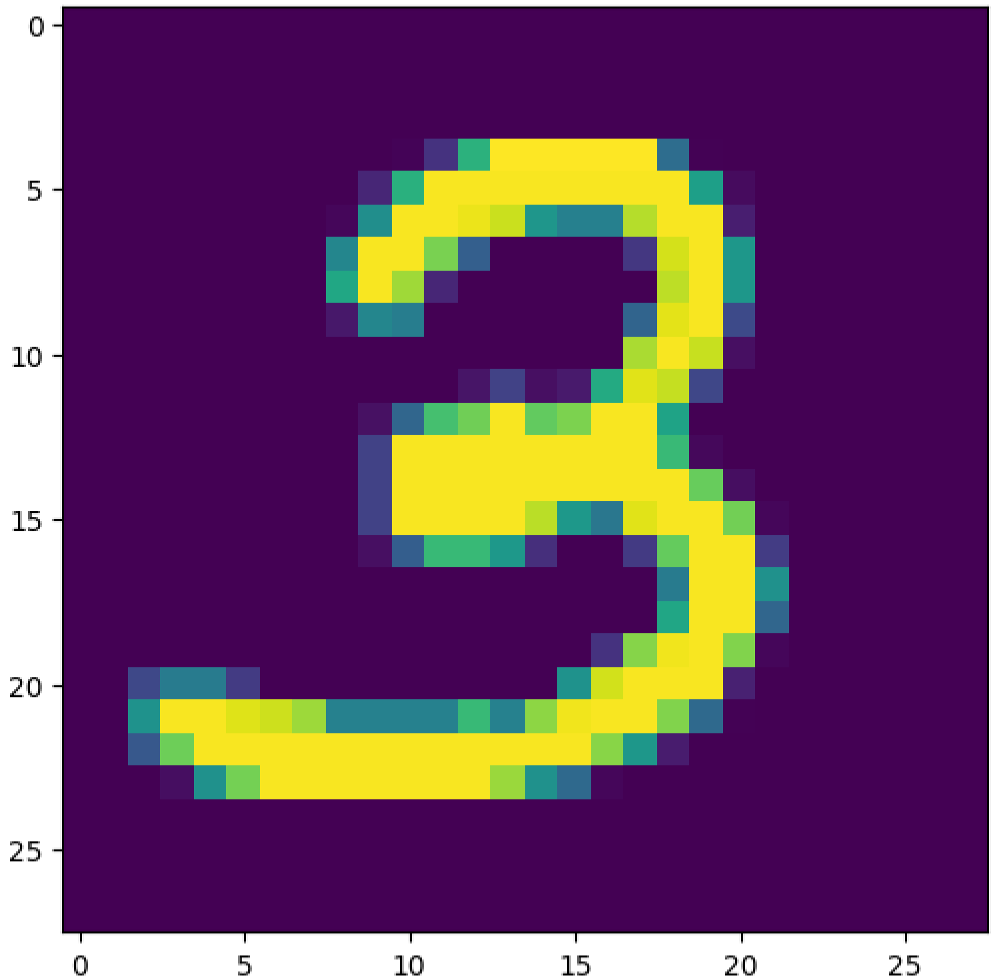


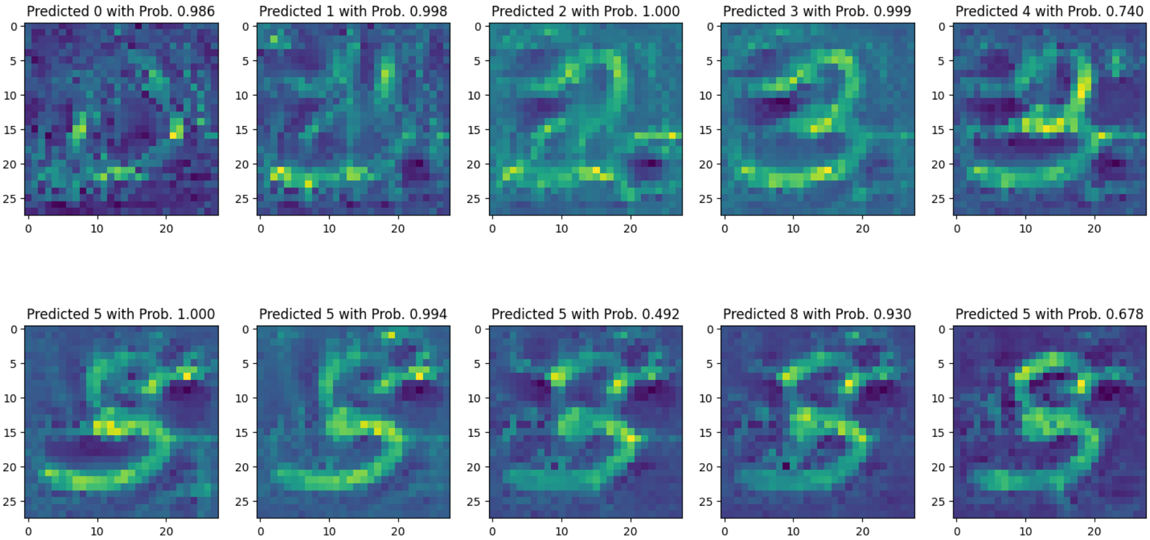


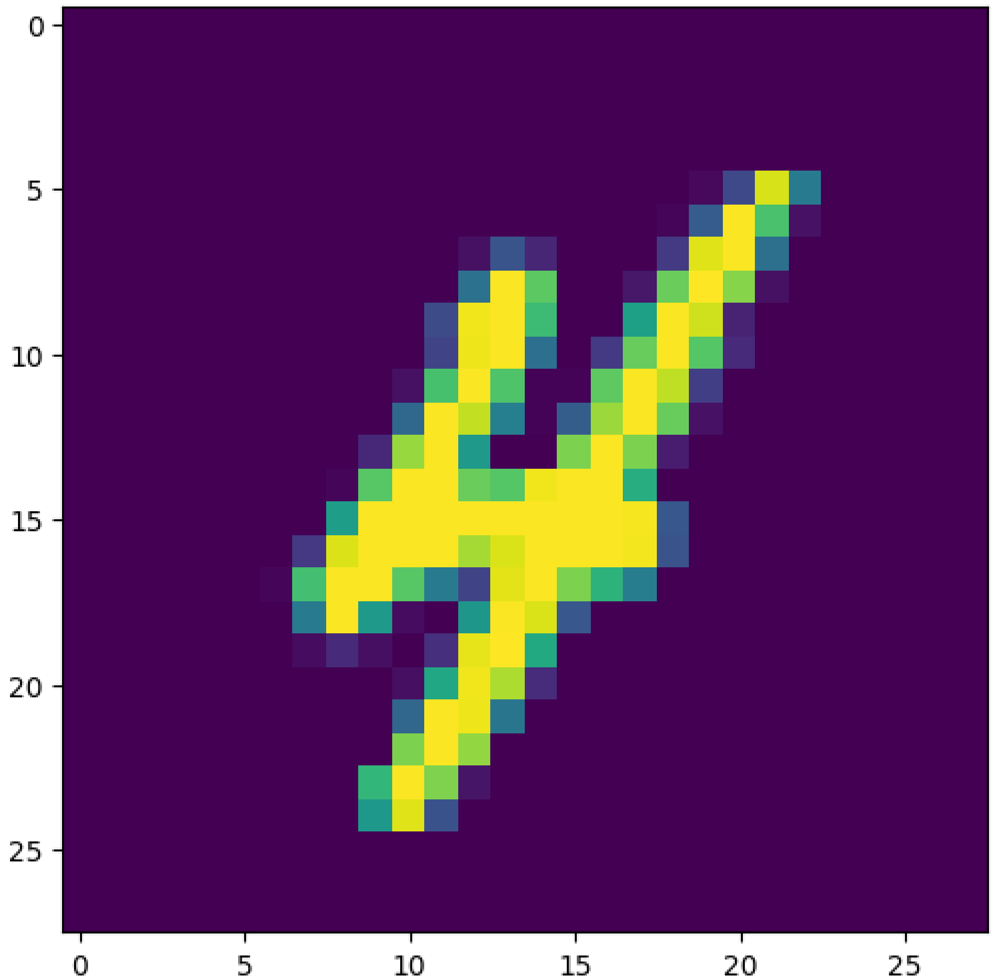
Predicted 0 with Prob. 0.987
Predicted 1 with Prob. 0.999
Predicted 2 with Prob. 0.999
Predicted 3 with Prob. 0.540
Predicted 1 with Prob. 0.708
Predicted 5 with Prob. 1.000
Predicted 6 with Prob. 0.820
Predicted 1 with Prob. 0.729
Predicted 8 with Prob. 0.955
Predicted 8 with Prob. 0.621

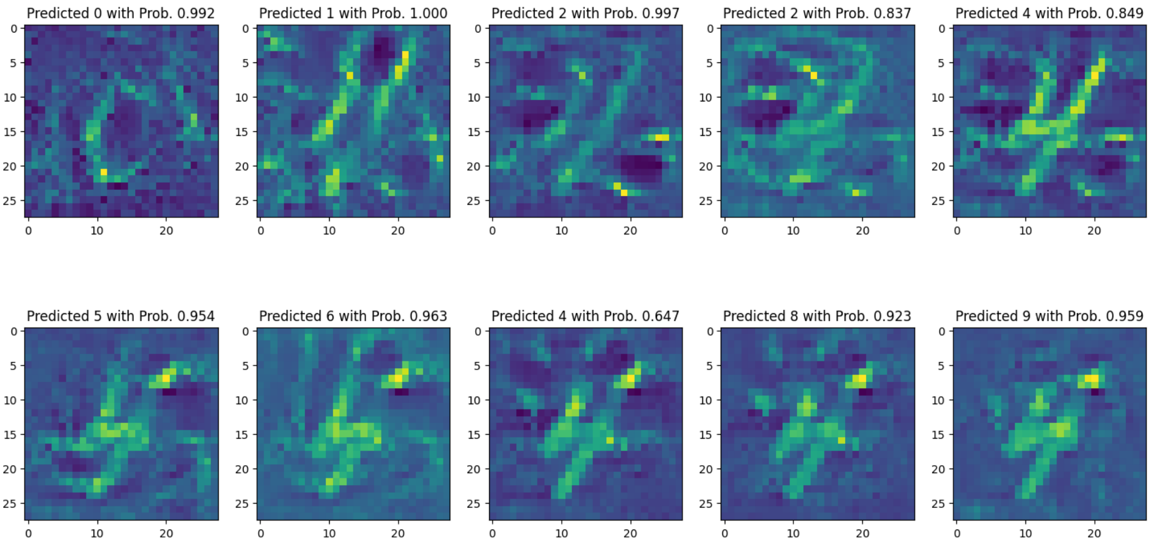



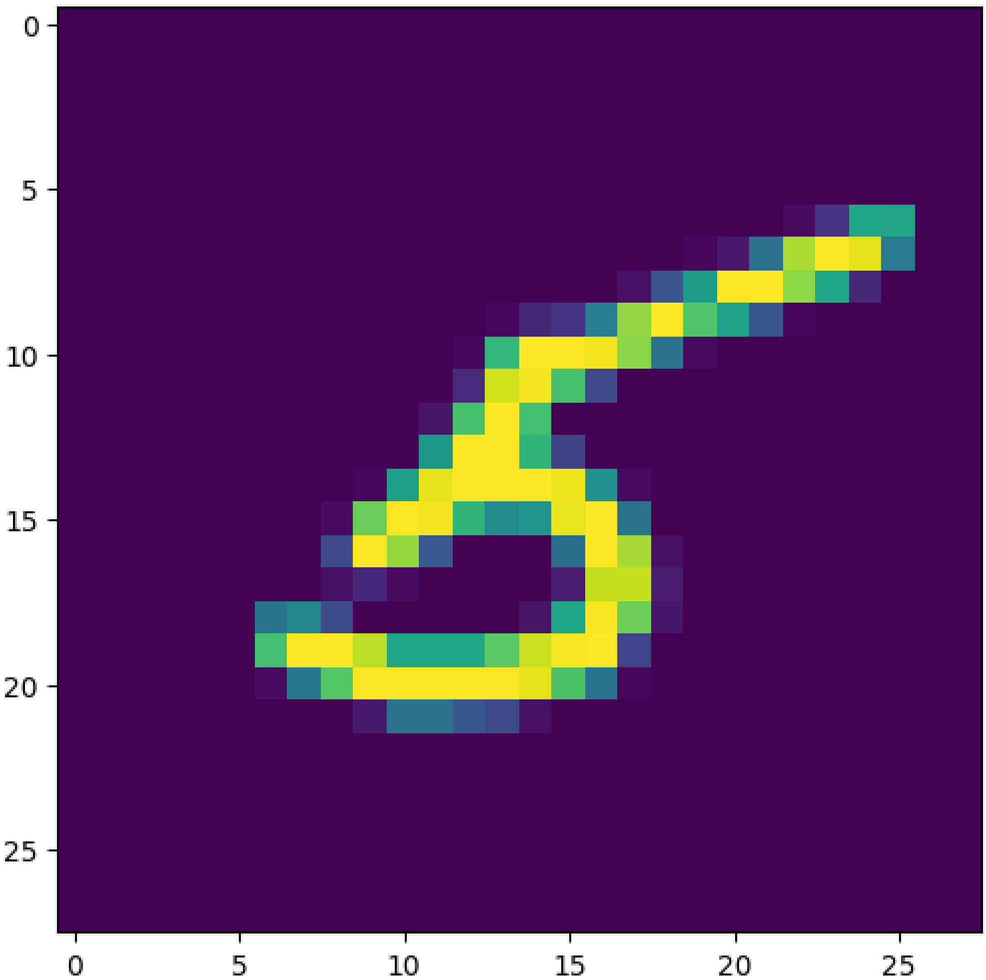

Predicted 0 with Prob. 0.991
Predicted 1 with Prob. 0.997
Predicted 2 with Prob. 1.000
Predicted 2 with Prob. 0.977
Predicted 4 with Prob. 0.916
Predicted 5 with Prob. 0.984
Predicted 5 with Prob. 0.923
Predicted 7 with Prob. 0.568
Predicted 8 with Prob. 0.821
Predicted 9 with Prob. 0.860

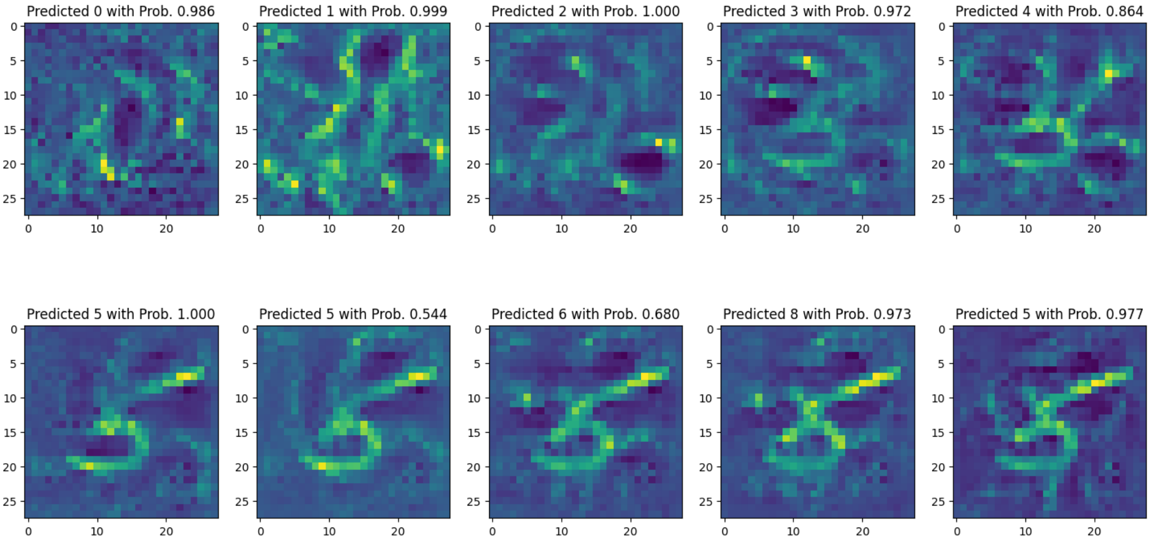


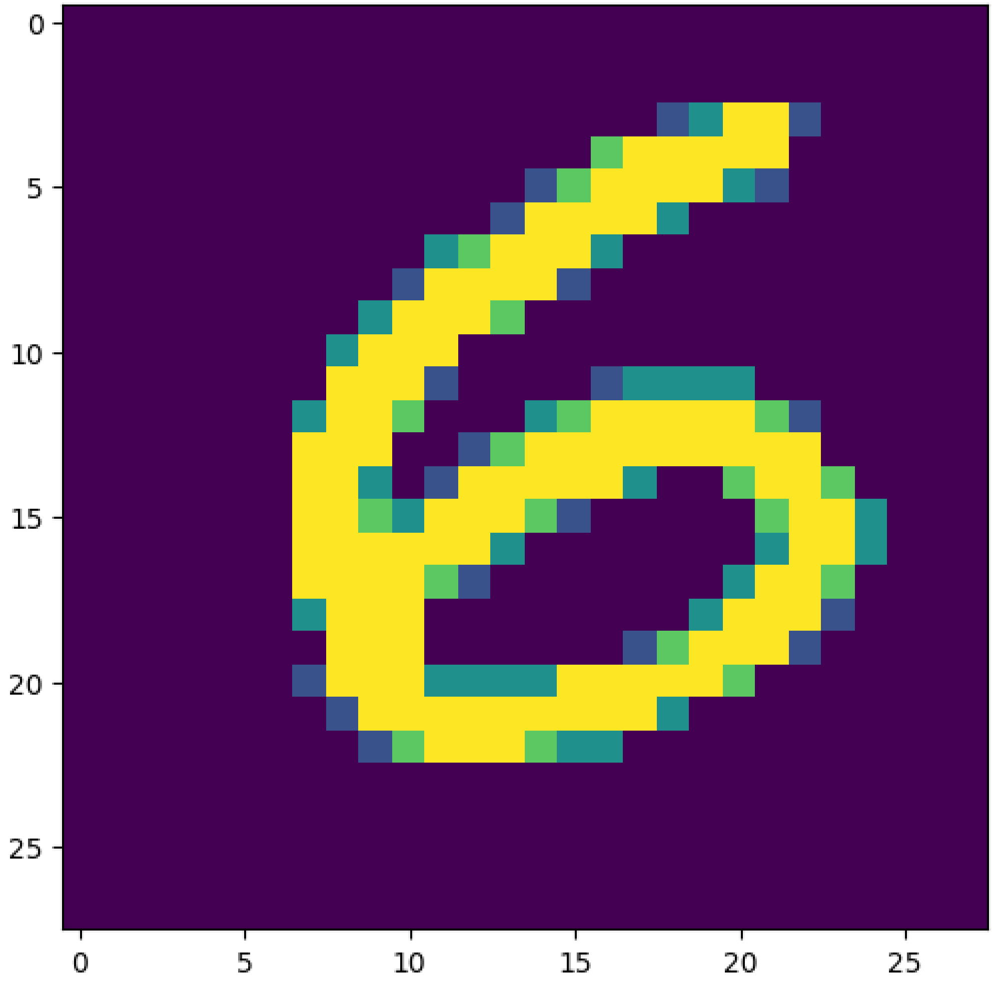

Predicted 0 with Prob. 0.986
Predicted 1 with Prob. 0.998
Predicted 2 with Prob. 1.000
Predicted 3 with Prob. 0.999
Predicted 4 with Prob. 0.740
Predicted 5 with Prob. 1.000
Predicted 5 with Prob. 0.994
Predicted 5 with Prob. 0.492
Predicted 8 with Prob. 0.930
Predicted 5 with Prob. 0.678

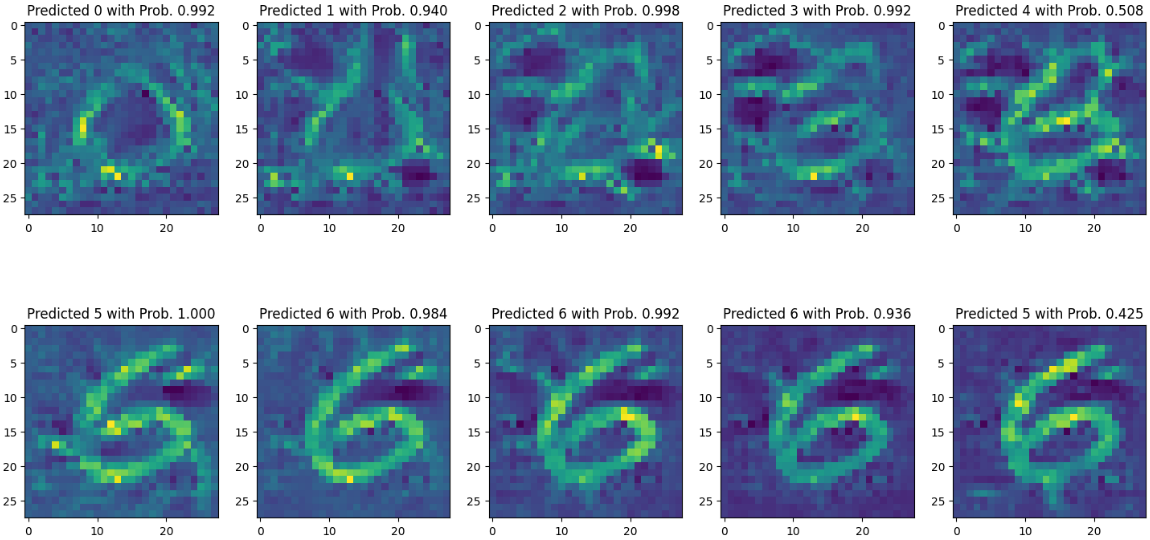


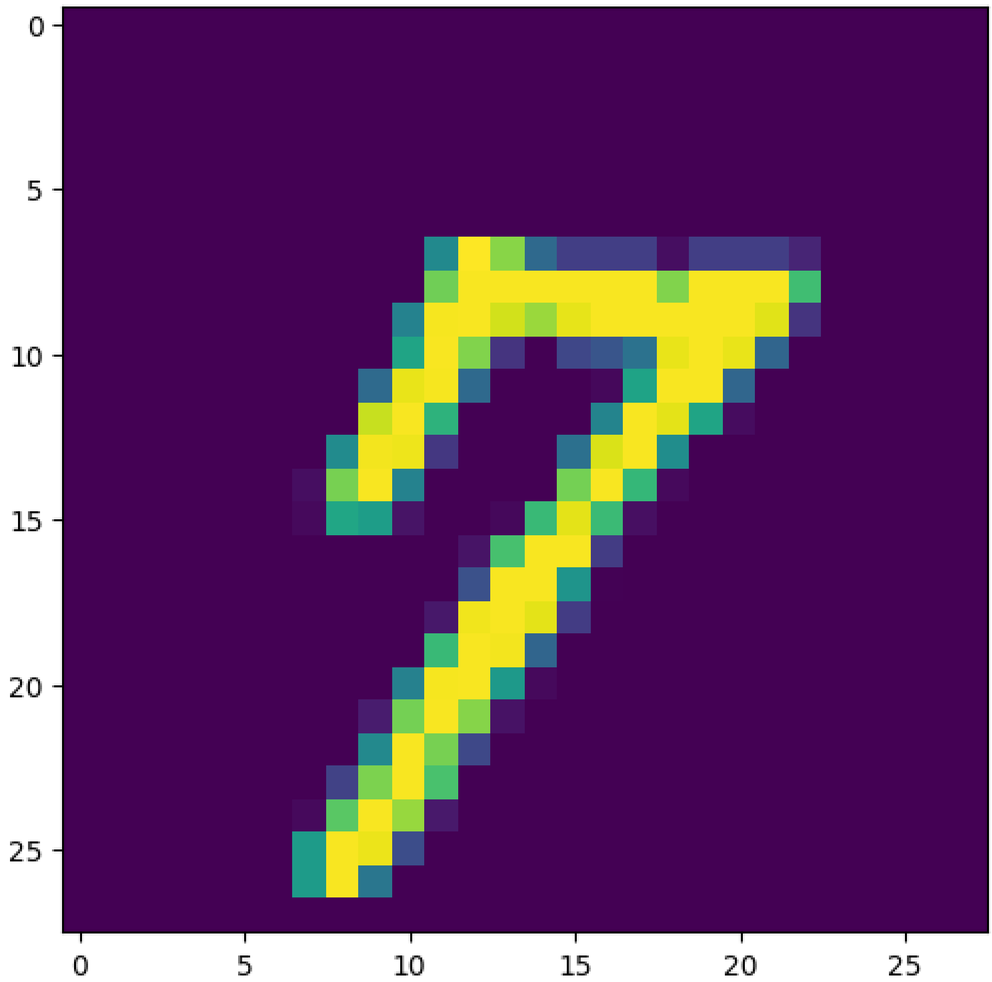

Predicted 0 with Prob. 0.992
Predicted 1 with Prob. 1.000
Predicted 2 with Prob. 0.997
Predicted 2 with Prob. 0.837
Predicted 4 with Prob. 0.849
Predicted 5 with Prob. 0.954
Predicted 6 with Prob. 0.963
Predicted 4 with Prob. 0.647
Predicted 8 with Prob. 0.923
Predicted 9 with Prob. 0.959

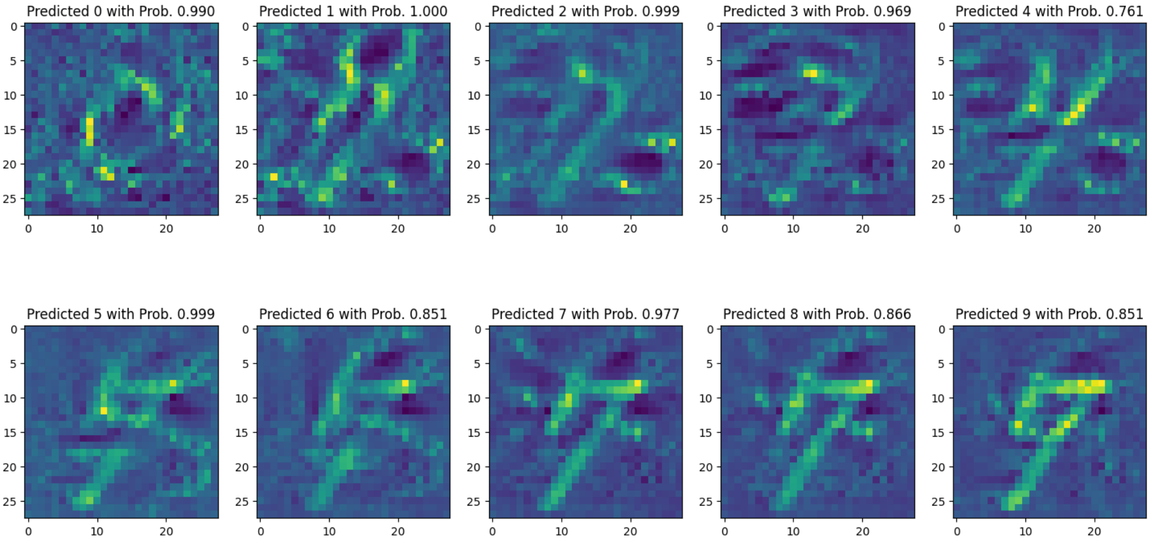


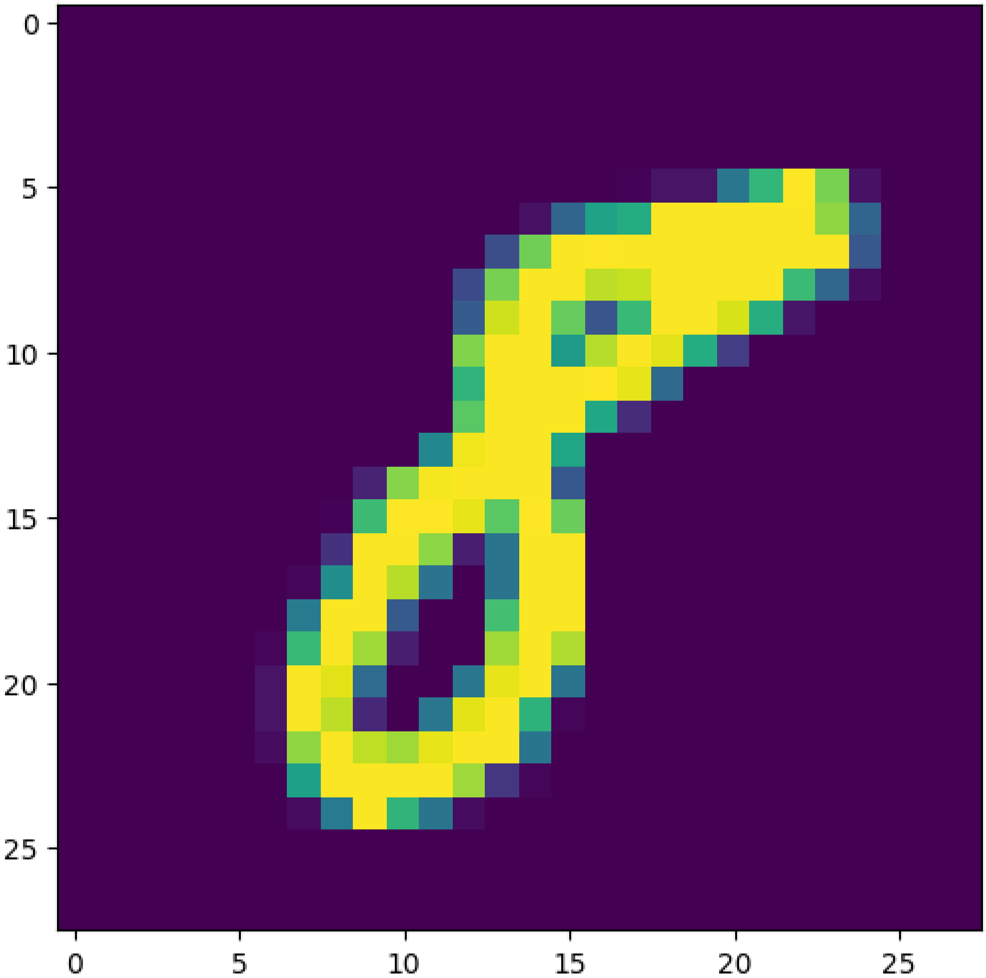

Predicted 0 with Prob. 0.986
Predicted 1 with Prob. 0.999
Predicted 2 with Prob. 1.000
Predicted 3 with Prob. 0.972
Predicted 4 with Prob. 0.864
Predicted 5 with Prob. 1.000
Predicted 5 with Prob. 0.544
Predicted 6 with Prob. 0.680
Predicted 8 with Prob. 0.973
Predicted 5 with Prob. 0.977

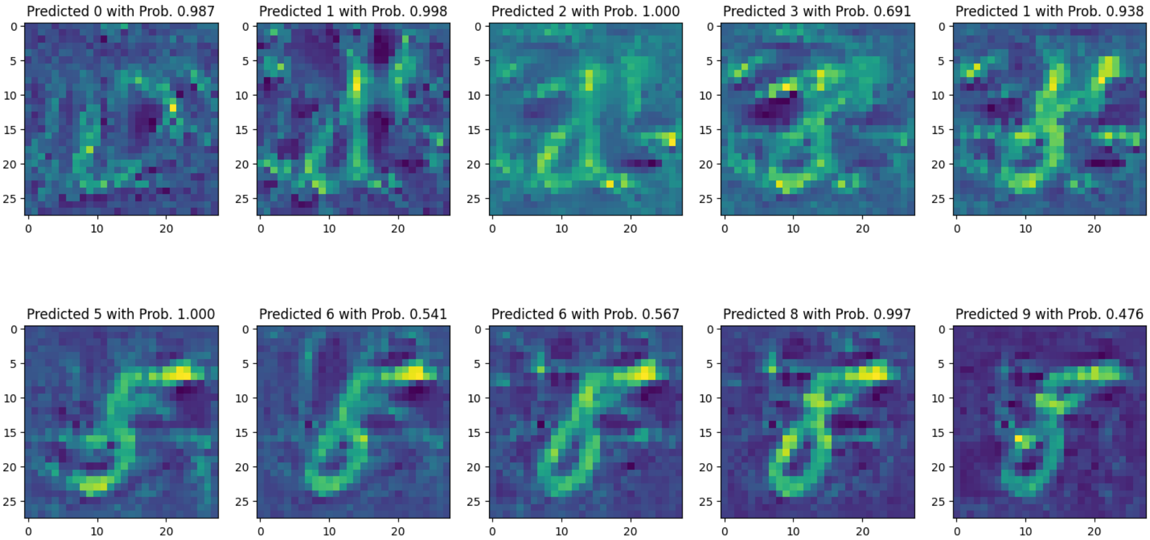


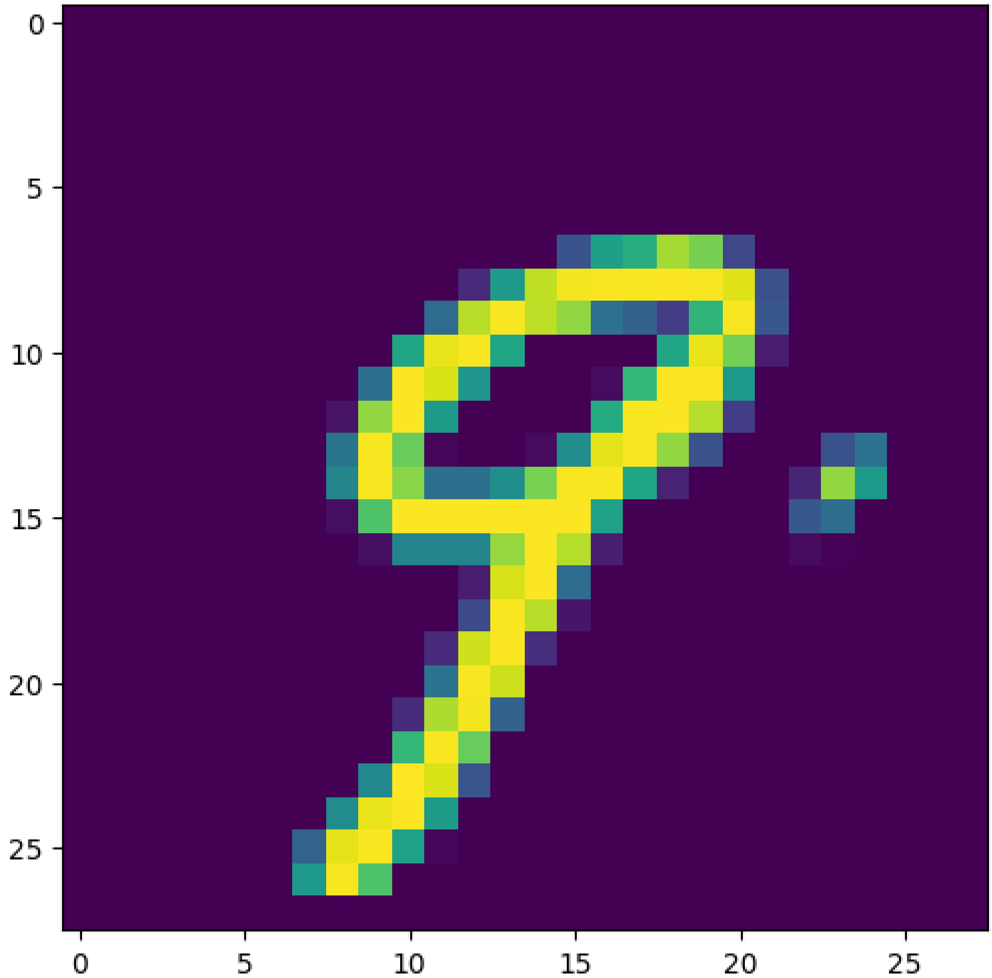

Predicted 0 with Prob. 0.992
Predicted 1 with Prob. 0.940
Predicted 2 with Prob. 0.998
Predicted 3 with Prob. 0.992
Predicted 4 with Prob. 0.508
Predicted 5 with Prob. 1.000
Predicted 6 with Prob. 0.984
Predicted 6 with Prob. 0.992
Predicted 6 with Prob. 0.936
Predicted 5 with Prob. 0.425

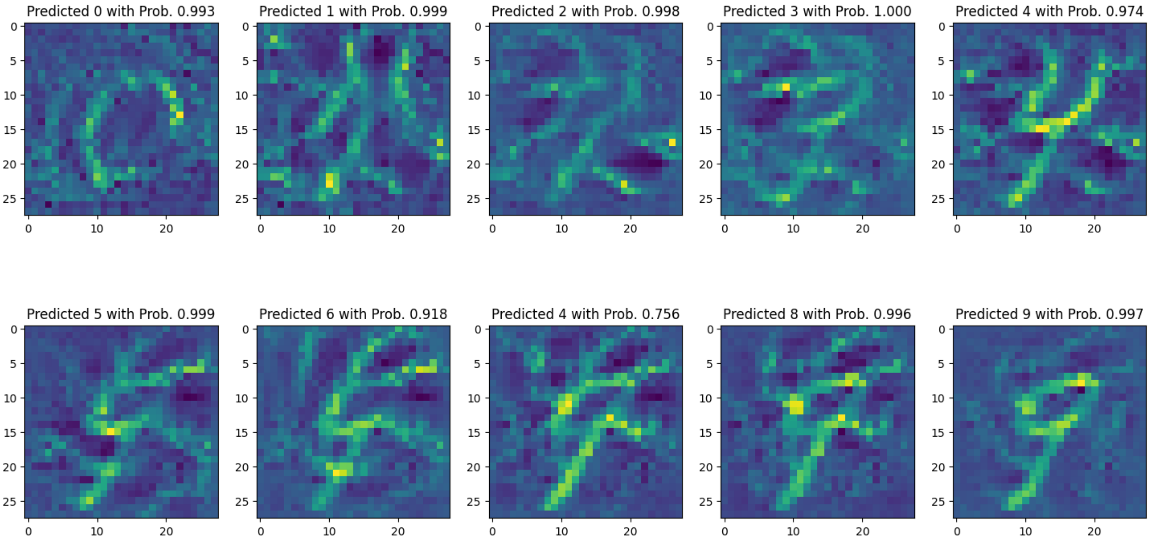


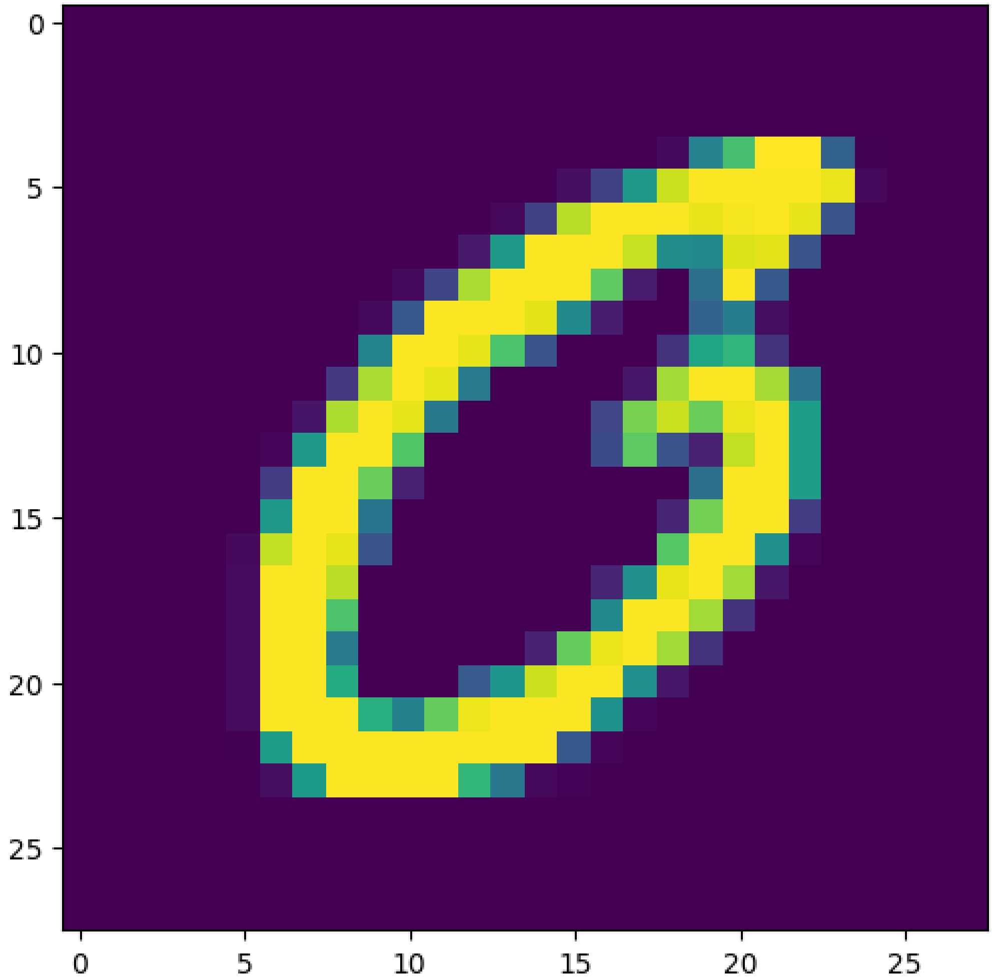

Predicted 0 with Prob. 0.990
Predicted 1 with Prob. 1.000
Predicted 2 with Prob. 0.999
Predicted 3 with Prob. 0.969
Predicted 4 with Prob. 0.761
Predicted 5 with Prob. 0.999
Predicted 6 with Prob. 0.851
Predicted 7 with Prob. 0.977
Predicted 8 with Prob. 0.866
Predicted 9 with Prob. 0.851

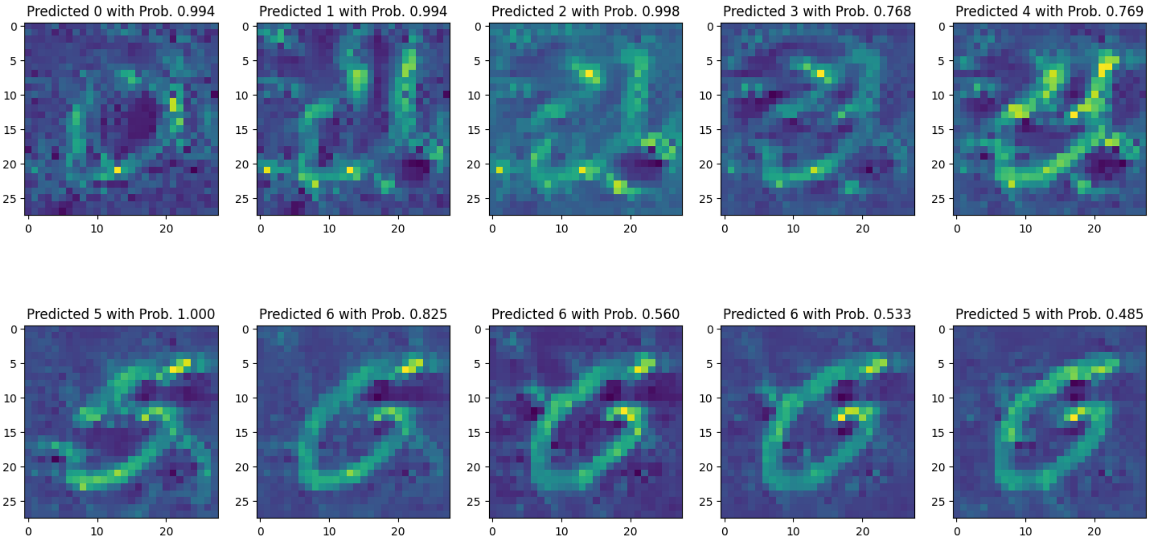



Predicted 0 with Prob. 0.987
Predicted 1 with Prob. 0.998
Predicted 2 with Prob. 1.000
Predicted 3 with Prob. 0.691
Predicted 1 with Prob. 0.938
Predicted 5 with Prob. 1.000
Predicted 6 with Prob. 0.541
Predicted 6 with Prob. 0.567
Predicted 8 with Prob. 0.997
Predicted 9 with Prob. 0.476




Predicted 0 with Prob. 0.993
Predicted 1 with Prob. 0.999
Predicted 2 with Prob. 0.998
Predicted 3 with Prob. 1.000
Predicted 4 with Prob. 0.974
Predicted 5 with Prob. 0.999
Predicted 6 with Prob. 0.918
Predicted 4 with Prob. 0.756
Predicted 8 with Prob. 0.996
Predicted 9 with Prob. 0.997






# ADDITION OF NOISE

1. Show the generated adversarial image, and the noise for each of the classes of MNIST.
2. Sample a fixed set of 10 test examples from the dataset. Add the adversarial noise and classify. Show the true class and the predicted class. Repeat this for all the 10 generated adversarial noise matrices.

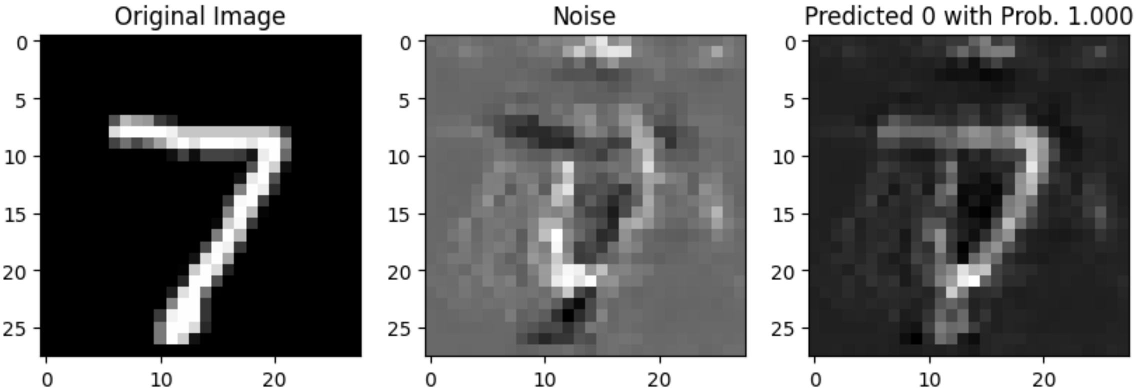
The cell output below contains the noise for each of the classes. Furthermore, a test set of 10 images have been added with these noises and have been misclassified with higer probability as the target classes. Hence the model has been successfully fooled by addition of these noises.

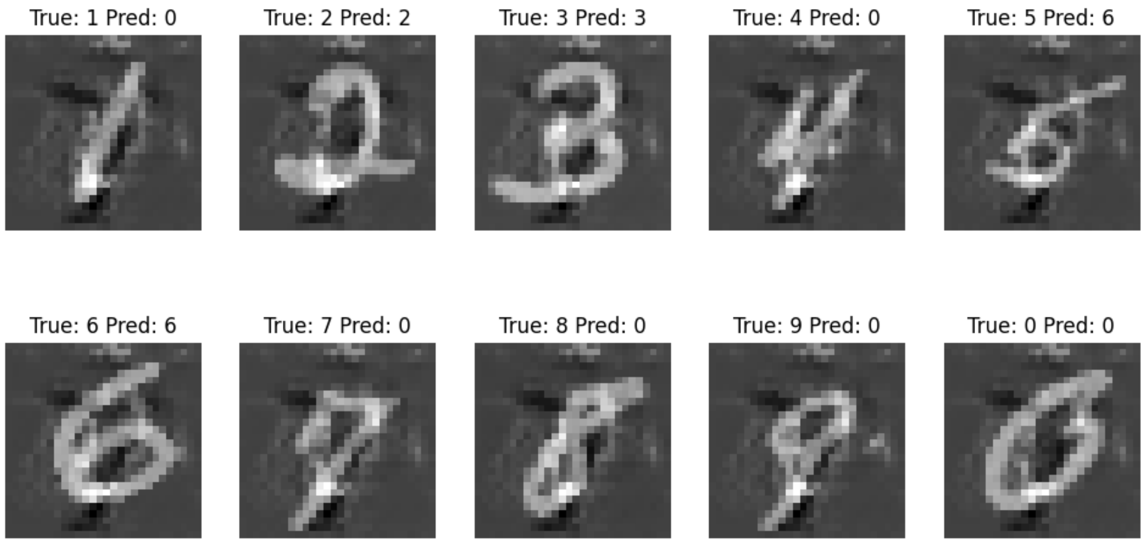
In [ ]:

```
# GENERIC CODE FOR ALL CLASSES

image=images[0].to(device)
label=labels[0].to(device)

#for all target i.e. 0,1,2,3,4,5,6,7,8,9
for target in range(10):
    print("TARGET CLASS: {}".format(target))
    noise1=torch.zeros((1,1,28,28),requires_grad=True,device='cuda')
    target_label=target
    optimizer = torch.optim.SGD([noise1],lr=0.001)
    fig = plt.figure(figsize=(10,6))
    for i in range(2000):
        optimizer.zero_grad()

        out_pred=model(image+noise1)
        loss=-out_pred[0][target_label]
        loss.backward()
        optimizer.step()

    prediction_vect_cpu = out_pred.detach().cpu().numpy()
    prediction_vect_cpu=np.squeeze(prediction_vect_cpu)

    prediction_prob_vect=np.exp(prediction_vect_cpu)/np.sum(np.exp(prediction_ve
ct_cpu))

    predicted_digit=np.argmax(prediction_prob_vect)
    prediction_probability=prediction_prob_vect[predicted_digit]



    a= fig.add_subplot(1,3,1)
    a.title.set_text('Original Image')
    plt.imshow(np.squeeze(image.cpu().detach().numpy()),cmap='gray')



    b= fig.add_subplot(1,3,2)
    b.title.set_text('Noise')
    plt.imshow(np.squeeze(noise1.cpu().detach().numpy()),cmap='gray')

    c= fig.add_subplot(1,3,3)
    c.title.set_text('Predicted {} with Prob. {:.3f}'.format(predicted_digit,pre
diction_probability))
    plt.imshow(np.squeeze((image+noise1).cpu().detach().numpy()),cmap='gray')

    for images, labels in test_loader:
      images=images[9705:9715]
      labels=labels[9705:9715]
      noise1=noise1.cuda()
      images=images.cuda()
      images=images+noise1
      pred=model(images)
      predlabels=torch.max(pred,1)[1].cpu().numpy()
      fig=plt.figure(figsize=(12, 6))
      columns = 5
      rows = 2
      for i in range(1, columns*rows +1):
          fig.add_subplot(rows, columns, i)
```

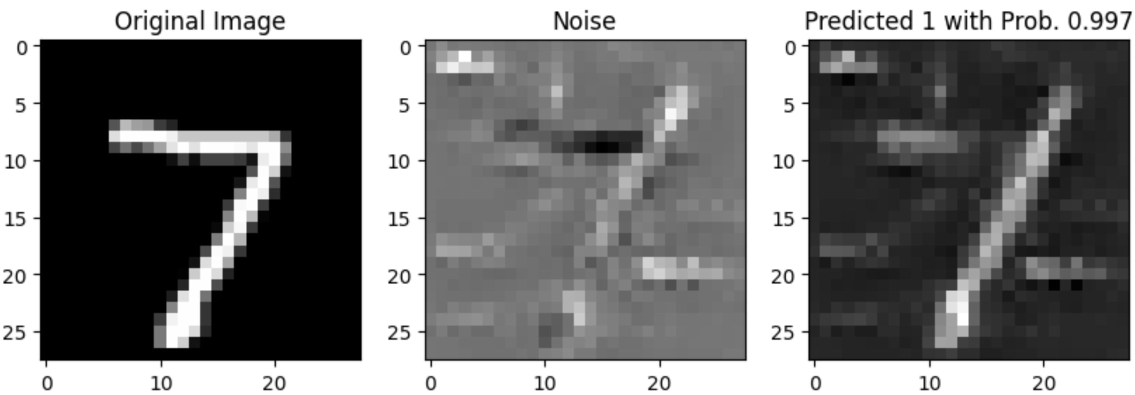
```
            trueval=labels[i-1].data.cpu().numpy()
            predval=predlabels[i-1]
            a='True: '+str(trueval)+' Pred: '+str(predval)
            plt.title(a)
            plt.axis('off')
            plt.imshow(images[i-1].reshape(-1,28).cpu().detach().numpy(), cmap='gr
ay')
        plt.show()
```

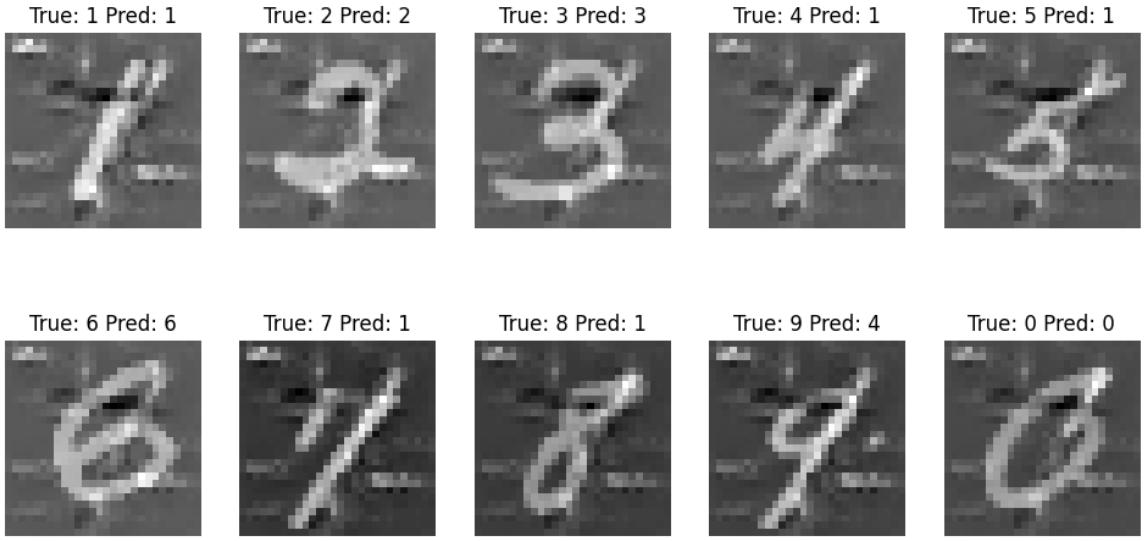
```
TARGET CLASS: 0
```



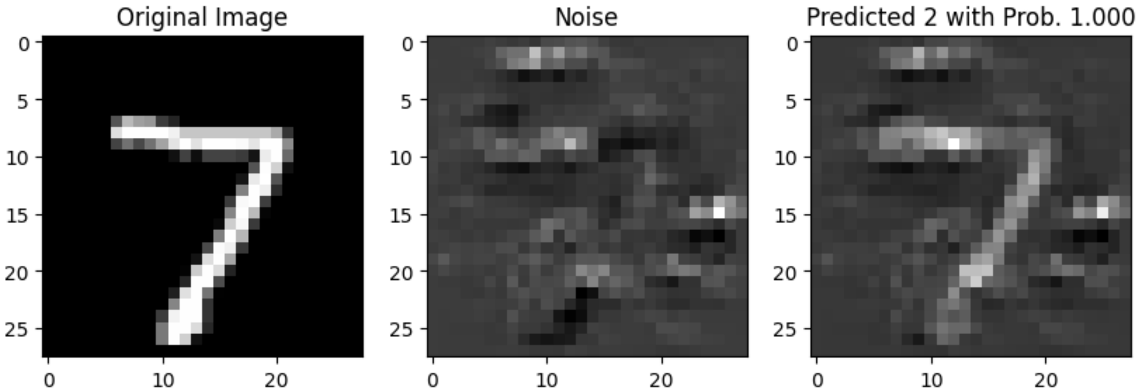



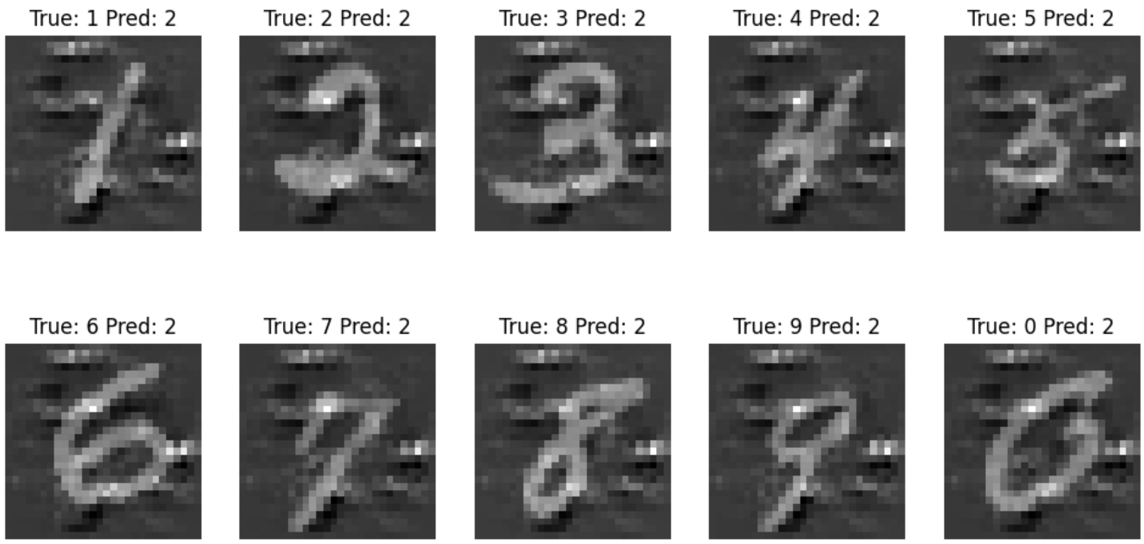

```
TARGET CLASS: 1
```



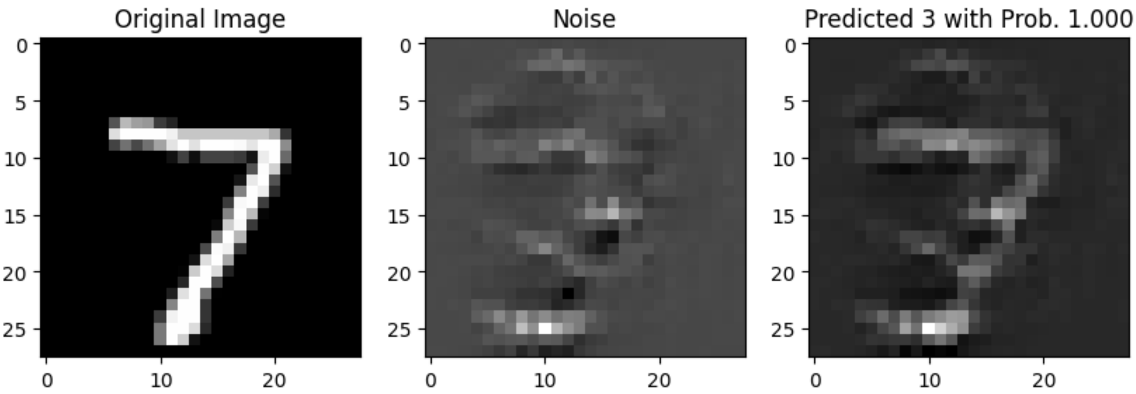


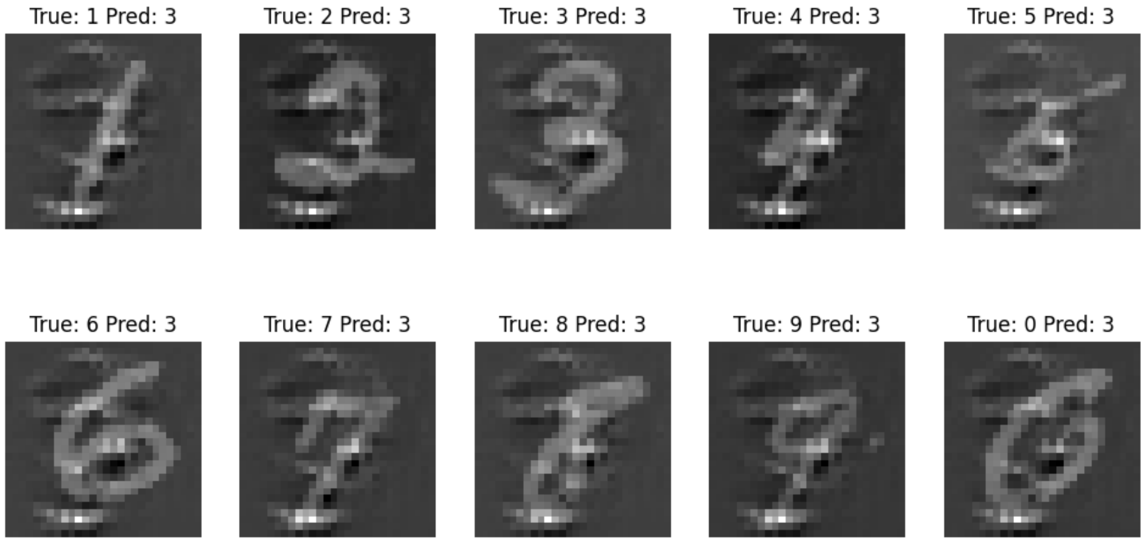

```
TARGET CLASS: 2
```



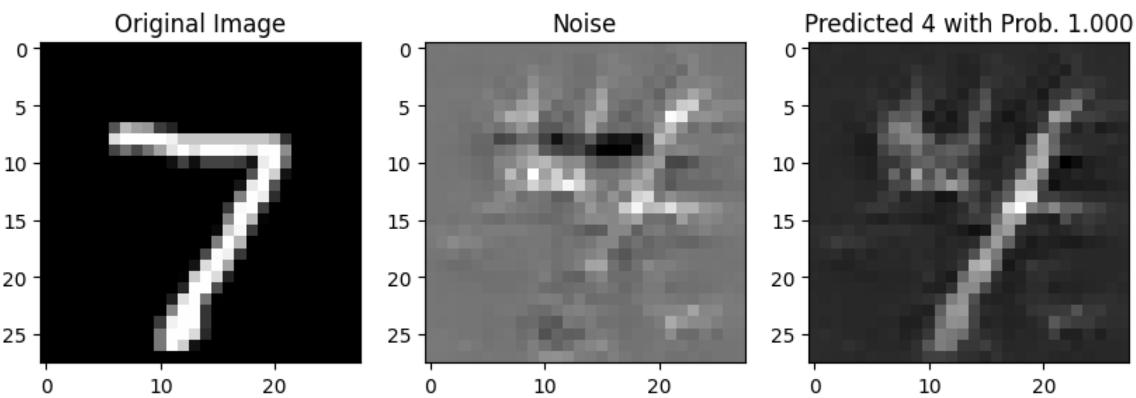



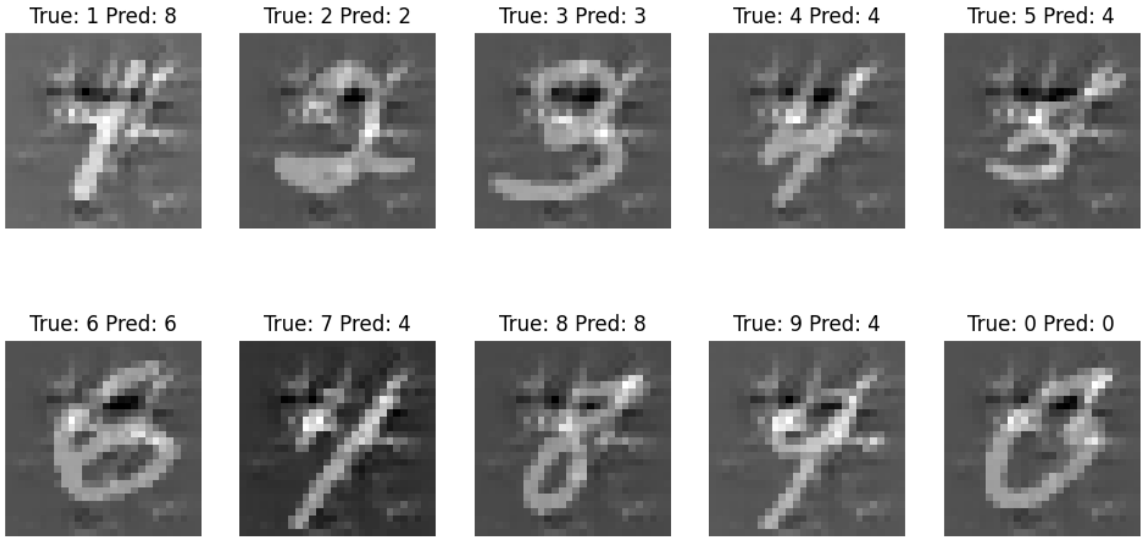

```
TARGET CLASS: 3
```



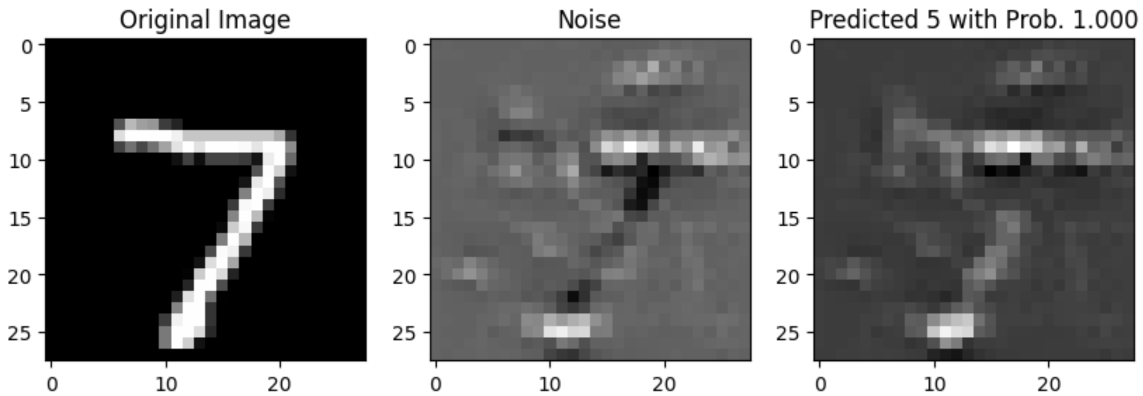



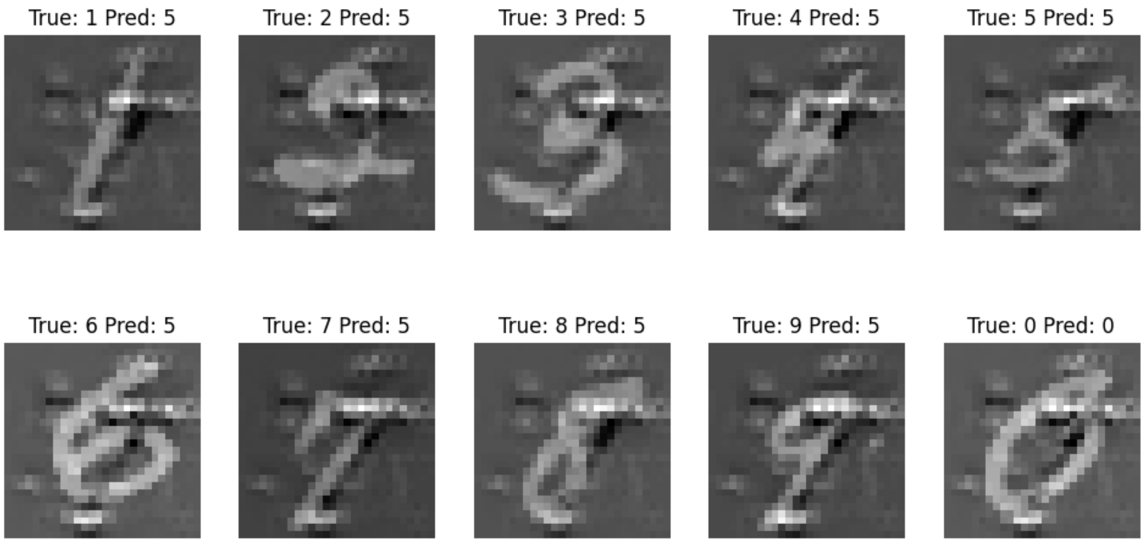

```
TARGET CLASS: 4
```



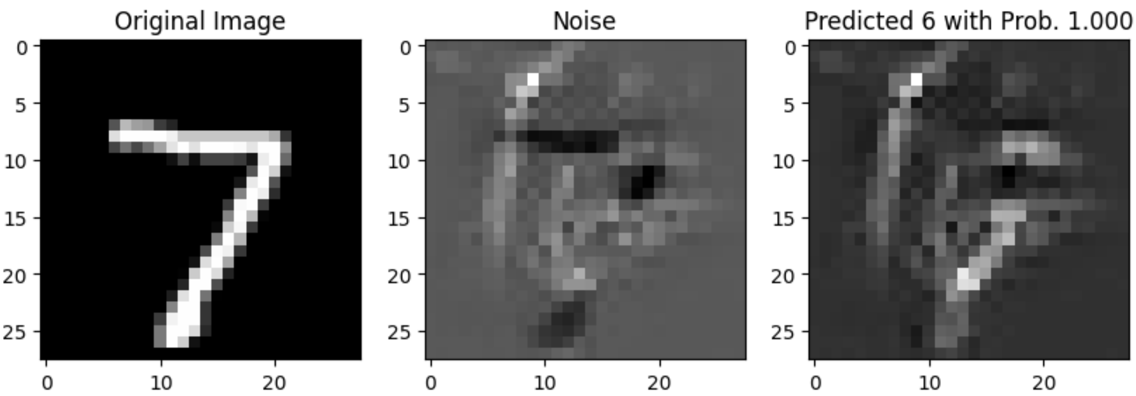


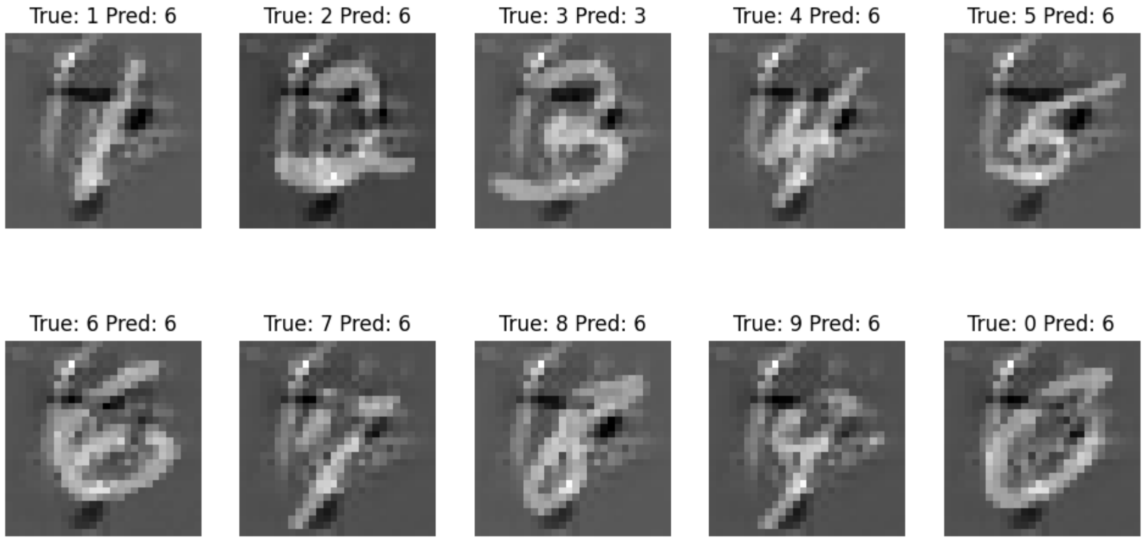

```
TARGET CLASS: 5
```



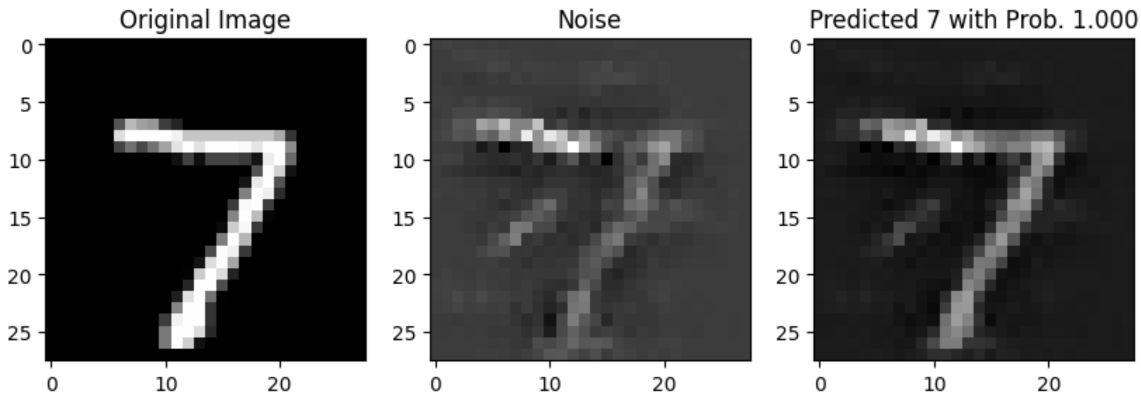



```
TARGET CLASS: 6
```

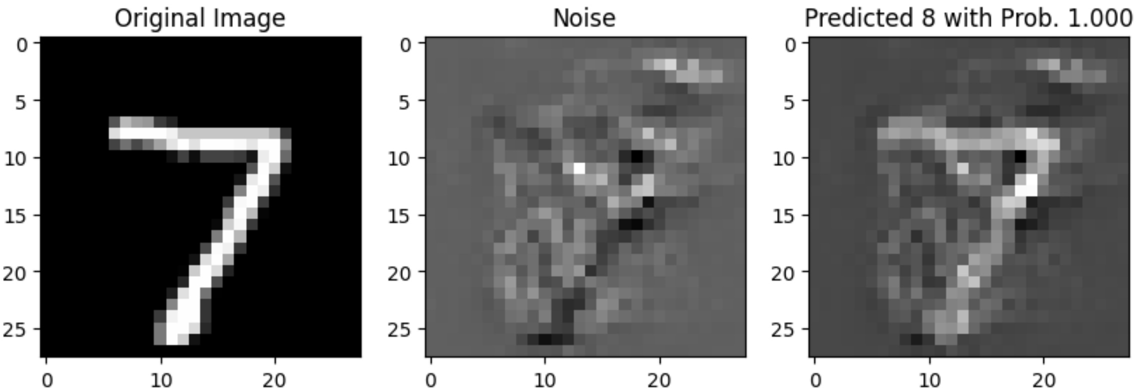


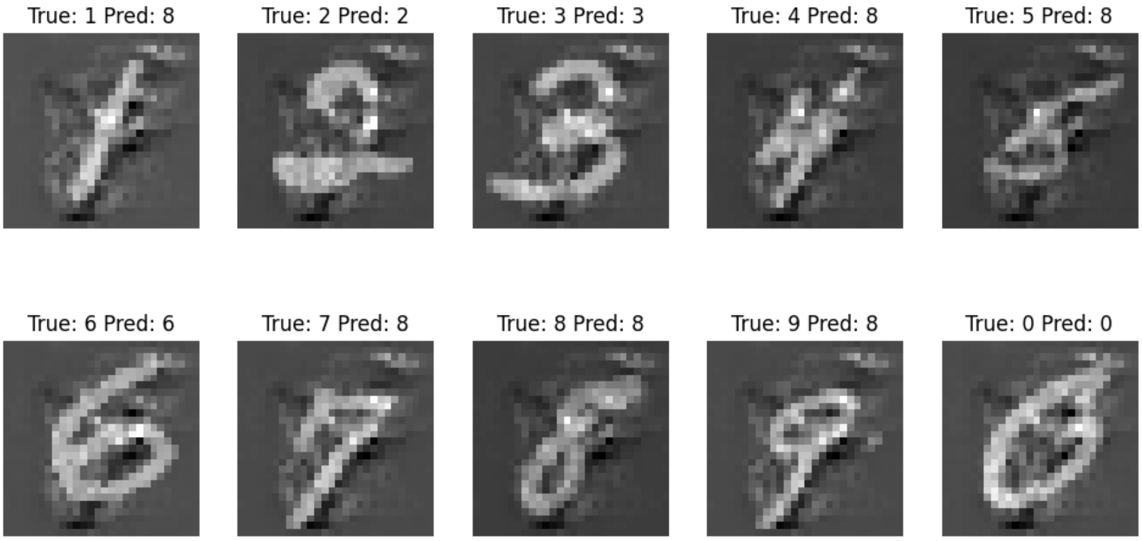



```
TARGET CLASS: 7
```

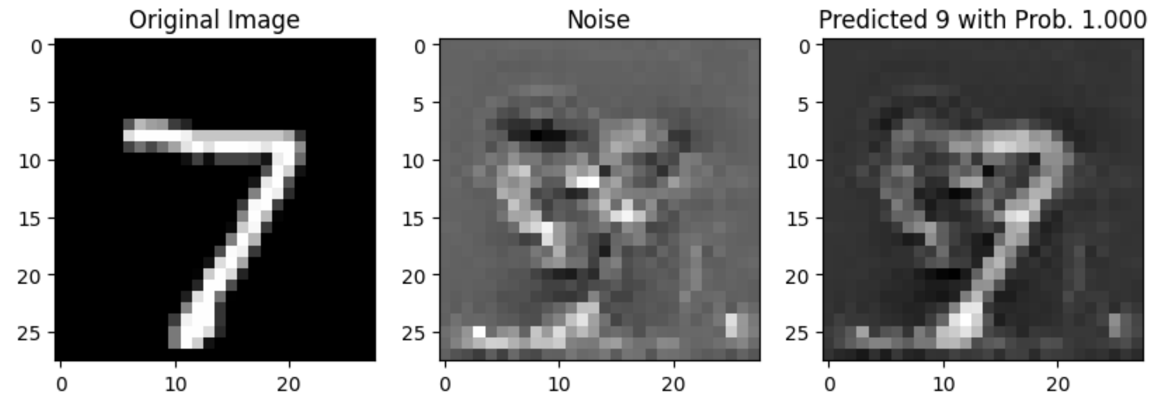


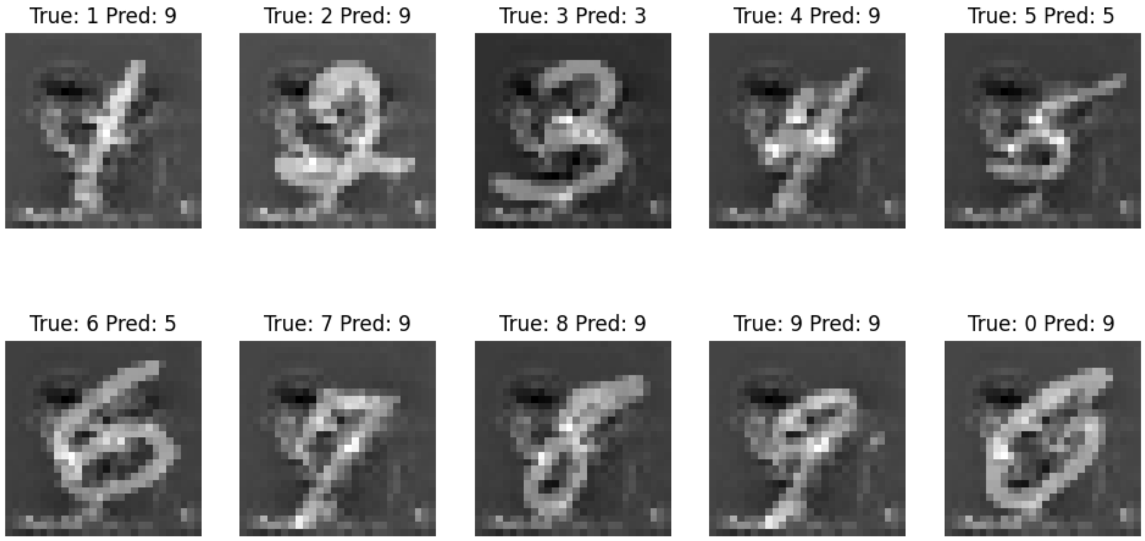

TARGET CLASS: 8






```
TARGET CLASS: 9
```




True: 1 Pred: 9
True: 2 Pred: 9
True: 3 Pred: 3
True: 4 Pred: 9
True: 5 Pred: 5
True: 6 Pred: 5
True: 7 Pred: 9
True: 8 Pred: 9
True: 9 Pred: 9
True: 0 Pred: 9

In [3]:

```
!pip install nbconvert
!sudo apt-get install texlive-xetex texlive-fonts-recommended texlive-plain-gene
ric
!jupyter nbconvert --to html "PA2_EE21S060.ipynb"
```