EE5179 : Deep Learning for Imaging

Programming Assignment 2: Convolutional Neural Networks

In [1]:

```
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision.utils import make_grid
from torchvision import datasets, transforms
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix
%matplotlib inline
```

```
/home/jannie/.conda/envs/DL/lib/python3.8/site-packages/tqdm/auto.py:22: T
qdmWarning: IProgress not found. Please update jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

In [2]:

```
!nvidia-smi
```

```
Fri Sep 23 15:18:42 2022
+-----------------------------------------------------------------------------
----+
| NVIDIA-SMI 515.43.04    Driver Version: 515.43.04    CUDA Version: 11.7
|
|-------------------------------+----------------------+----------------------
----+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr.
ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute
M. |
|                               |                      |               MIG
M. |
|===============================+======================+======================
====|
|   0  NVIDIA GeForce ...  On   | 00000000:05:00.0 Off |
N/A |
|  0%   40C    P8    18W / 350W |  15198MiB / 24576MiB |      0%      Defa
ult |
|                               |                      |
N/A |
+-------------------------------+----------------------+----------------------
----+

+-----------------------------------------------------------------------------
----+
| Processes:
|
|  GPU   GI   CI        PID   Type   Process name                  GPU Mem
ory |
|        ID   ID                                                   Usage
|
|=============================================================================
====|
|    0   N/A  N/A      1717      G   /usr/lib/xorg/Xorg                     9
MiB |
|    0   N/A  N/A      1859      G   /usr/bin/gnome-shell                   8
MiB |
|    0   N/A  N/A   1856332      C   ....conda/envs/DL/bin/python        1691
MiB |
|    0   N/A  N/A   2723630      C   ...da/envs/hbpenv/bin/python       11203
MiB |
|    0   N/A  N/A   2758739      C   ...da/envs/hbpenv/bin/python        2031
MiB |
|    0   N/A  N/A   3196462      C   ...da/envs/hbpenv/bin/python         251
MiB |
+-----------------------------------------------------------------------------
----+
```

In [3]:

```
# cpu-gpu
a = torch.randn((3, 4))
print(a.device)

device = torch.device("cuda")
a = a.to(device)
print(a.device)

# a more generic code
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

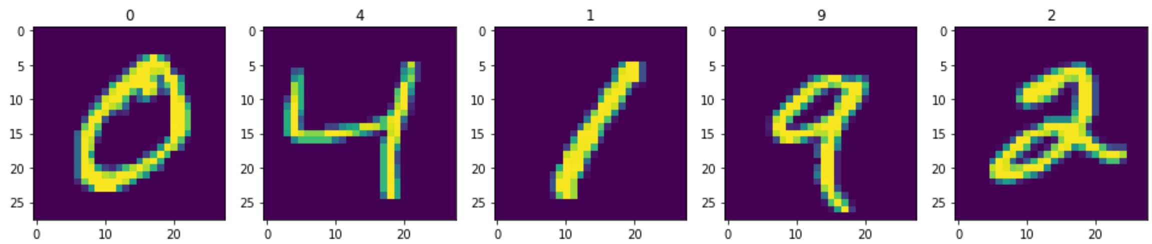
```
cpu
cuda:0
```

In [4]:

```
mnist_trainset = datasets.MNIST(root='./data', train=True, download=True, transform=transforms.Compose([transforms.ToTensor()]))
mnist_testset = datasets.MNIST(root='./data', train=False, download=True, transform=transforms.Compose([transforms.ToTensor()]))

mnist_valset, mnist_testset = torch.utils.data.random_split(mnist_testset, [int(0.9 * len(mnist_testset)), int(0.1 * len(mnist_testset))])
```

In [5]:

```
# visualize data
fig=plt.figure(figsize=(20, 10))
for i in range(1, 6):
    img = transforms.ToPILImage(mode='L')(mnist_trainset[i][0])
    fig.add_subplot(1, 6, i)
    plt.title(mnist_trainset[i][1])
    plt.imshow(img)
plt.show()
```



1. MNIST Classification using CNN

Create model

In [6]:

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.layer1 = nn.Sequential(nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1
), nn.ReLU(
        ), nn.MaxPool2d(kernel_size=2, stride=2))
        self.layer2 = nn.Sequential(nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=
1), nn.ReLU(
        ), nn.MaxPool2d(kernel_size=2, stride=2))
        self.layer3 = nn.Sequential(nn.Linear(7*7*32, 500), nn.ReLU())
        self.layer4 = nn.Linear(500, 10)

    def forward(self, x):
        out = self.layer1(x.float())
        out = self.layer2(out)
        out = out.reshape(out.size(0), -1)
        out = self.layer3(out)
        out = self.layer4(out)
        return F.log_softmax(out, dim=1)
```

In [7]:

```
model = CNN()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
print(model)
```

```
CNN(
  (layer1): Sequential(
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mo
de=False)
  )
  (layer2): Sequential(
    (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mo
de=False)
  )
  (layer3): Sequential(
    (0): Linear(in_features=1568, out_features=500, bias=True)
    (1): ReLU()
  )
  (layer4): Linear(in_features=500, out_features=10, bias=True)
)
```

Training

In [8]:

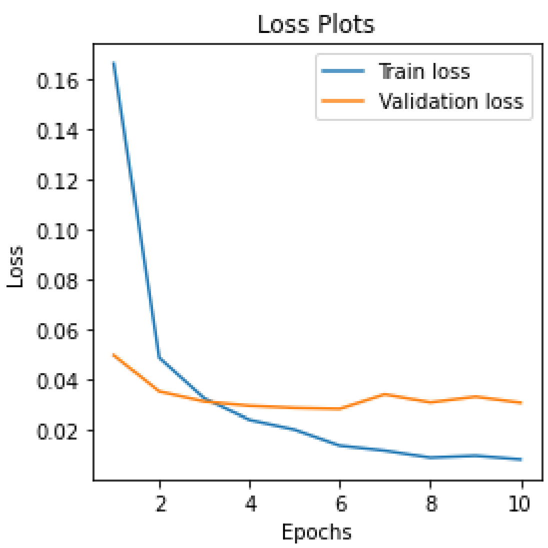
```
train_dataloader = torch.utils.data.DataLoader(mnist_trainset, batch_size=64, shuffle=True)
val_dataloader = torch.utils.data.DataLoader(mnist_valset, batch_size=32, shuffle=False)
test_dataloader = torch.utils.data.DataLoader(mnist_testset, batch_size=32, shuffle=False)
```

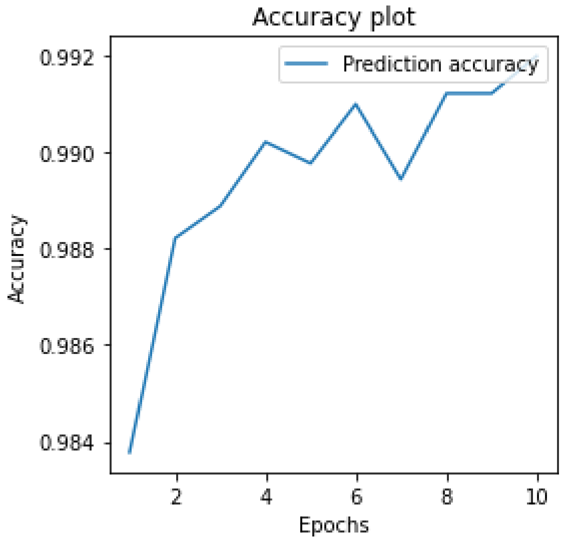
In [9]:

```
no_epochs = 10
train_loss = list()
val_loss = list()
pred_accuracy = list()
best_val_loss = 1
for epoch in range(no_epochs):
    total_train_loss = 0
    total_val_loss = 0

    model.train()
    # training
    for itr, (image, label) in enumerate(train_dataloader):
        optimizer.zero_grad()

        pred = model(image)

        loss = criterion(pred, label)
        total_train_loss += loss.item()

        loss.backward()
        optimizer.step()

    total_train_loss = total_train_loss / (itr + 1)
    train_loss.append(total_train_loss)

    # validation
    model.eval()
    total = 0
    for itr, (image, label) in enumerate(val_dataloader):
        pred = model(image)

        loss = criterion(pred, label)
        total_val_loss += loss.item()

        pred = torch.nn.functional.softmax(pred, dim=1)
        for i, p in enumerate(pred):
            if label[i] == torch.max(p.data, 0)[1]:
                total = total + 1

    accuracy = total / len(mnist_valset)
    pred_accuracy.append(accuracy)

    total_val_loss = total_val_loss / (itr + 1)
    val_loss.append(total_val_loss)

    print('\nEpoch: {}/{}, Train Loss: {:.8f}, Val Loss: {:.8f}, Val Accuracy: {:.8f}'.
format(epoch + 1, no_epochs, total_train_loss, total_val_loss, accuracy))

    if total_val_loss < best_val_loss:
        best_val_loss = total_val_loss
        print("Saving the model state dictionary for Epoch: {} with Validation loss:
{:.8f}".format(epoch + 1, total_val_loss))
        torch.save(model.state_dict(), "model.dth")

fig=plt.figure(figsize=(4, 4))
plt.plot(np.arange(1, no_epochs+1), train_loss, label="Train loss")
plt.plot(np.arange(1, no_epochs+1), val_loss, label="Validation loss")
plt.xlabel('Epochs')
plt.ylabel('Loss')
```

```
plt.title("Loss Plots")
plt.legend(loc='upper right')
plt.savefig('loss.png')


fig=plt.figure(figsize=(4, 4))
plt.plot(np.arange(1, no_epochs+1), pred_accuracy, label="Prediction accuracy")
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title("Accuracy plot")
plt.legend(loc='upper right')
plt.savefig('accuracy.png')
```

```
Epoch: 1/10, Train Loss: 0.16626306, Val Loss: 0.04987140, Val Accuracy:
0.98377778
Saving the model state dictionary for Epoch: 1 with Validation loss: 0.049
87140

Epoch: 2/10, Train Loss: 0.04889413, Val Loss: 0.03545206, Val Accuracy:
0.98822222
Saving the model state dictionary for Epoch: 2 with Validation loss: 0.035
45206

Epoch: 3/10, Train Loss: 0.03267100, Val Loss: 0.03142209, Val Accuracy:
0.98888889
Saving the model state dictionary for Epoch: 3 with Validation loss: 0.031
42209

Epoch: 4/10, Train Loss: 0.02421143, Val Loss: 0.02971109, Val Accuracy:
0.99022222
Saving the model state dictionary for Epoch: 4 with Validation loss: 0.029
71109

Epoch: 5/10, Train Loss: 0.02030896, Val Loss: 0.02886758, Val Accuracy:
0.98977778
Saving the model state dictionary for Epoch: 5 with Validation loss: 0.028
86758

Epoch: 6/10, Train Loss: 0.01396488, Val Loss: 0.02845046, Val Accuracy:
0.99100000
Saving the model state dictionary for Epoch: 6 with Validation loss: 0.028
45046

Epoch: 7/10, Train Loss: 0.01195322, Val Loss: 0.03427150, Val Accuracy:
0.98944444

Epoch: 8/10, Train Loss: 0.00916488, Val Loss: 0.03103979, Val Accuracy:
0.99122222

Epoch: 9/10, Train Loss: 0.00991746, Val Loss: 0.03331340, Val Accuracy:
0.99122222

Epoch: 10/10, Train Loss: 0.00845043, Val Loss: 0.03090557, Val Accuracy:
0.99200000
```



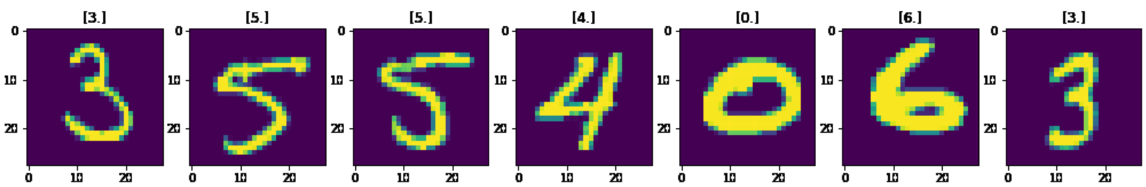


In [10]:

```
for itr, (image, label) in enumerate(test_dataloader):
        pred = model(image)
        pred_label = np.zeros((6000,1))
        pred = torch.nn.functional.softmax(pred, dim=1)
        for i, p in enumerate(pred):
            pred_label[i] = int(torch.max(p.data, 0)[1])
```

In [11]:

```
fig=plt.figure(figsize=(20, 10))
for test_images, test_labels in test_dataloader:
    for i in range(1,8):
        sample_image = test_images[i]
        sample_label = pred_label[i]
        img = transforms.ToPILImage(mode='L')(sample_image)
        fig.add_subplot(1, 8, i)
        plt.title(str(sample_label))
        plt.imshow(img)
plt.show()
```



Batch training

In [12]:

```
class CNN_batchnorm(nn.Module):
    def __init__(self):
        super(CNN_batchnorm, self).__init__()
        self.layer1 = nn.Sequential(nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1
), nn.ReLU(
        ), nn.MaxPool2d(kernel_size=2, stride=2))
        self.layer2 = nn.Sequential(nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=
1), nn.ReLU(
        ), nn.MaxPool2d(kernel_size=2, stride=2))
        self.layer3 = nn.Sequential(nn.Linear(7*7*32, 500), nn.ReLU(), nn.BatchNorm1d(5
00))
        self.layer4 = nn.Linear(500, 10)

    def forward(self, x):
        out = self.layer1(x.float())
        out = self.layer2(out)
        out = out.reshape(out.size(0), -1)
        out = self.layer3(out)
        out = self.layer4(out)
        return F.log_softmax(out, dim=1)
```

In [13]:

```
model = CNN_batchnorm()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
print(model)
```

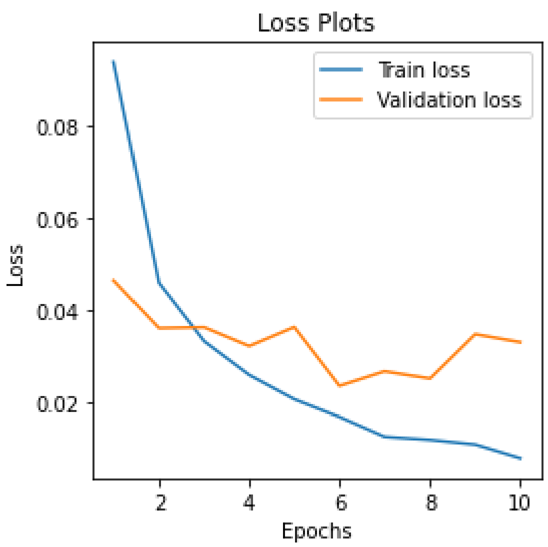
```
CNN_batchnorm(
  (layer1): Sequential(
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mo
de=False)
  )
  (layer2): Sequential(
    (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mo
de=False)
  )
  (layer3): Sequential(
    (0): Linear(in_features=1568, out_features=500, bias=True)
    (1): ReLU()
    (2): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
  )
  (layer4): Linear(in_features=500, out_features=10, bias=True)
)
```

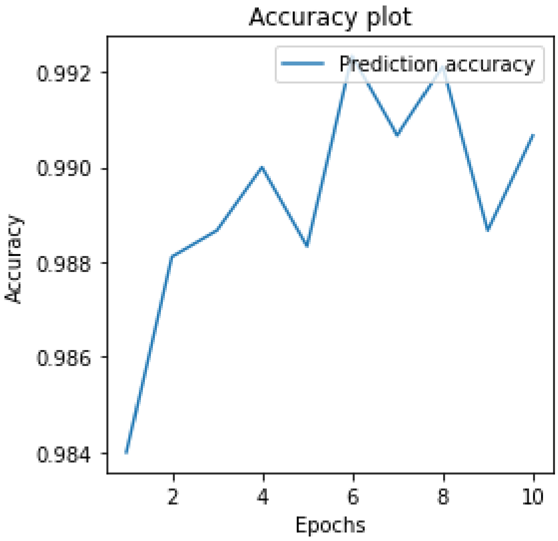
In [14]:

```
no_epochs = 10
train_loss = list()
val_loss = list()
pred_accuracy = list()
best_val_loss = 1
for epoch in range(no_epochs):
    total_train_loss = 0
    total_val_loss = 0

    model.train()
    # training
    for itr, (image, label) in enumerate(train_dataloader):
        optimizer.zero_grad()

        pred = model(image)

        loss = criterion(pred, label)
        total_train_loss += loss.item()

        loss.backward()
        optimizer.step()

    total_train_loss = total_train_loss / (itr + 1)
    train_loss.append(total_train_loss)

    # validation
    model.eval()
    total = 0
    for itr, (image, label) in enumerate(val_dataloader):
        pred = model(image)

        loss = criterion(pred, label)
        total_val_loss += loss.item()

        pred = torch.nn.functional.softmax(pred, dim=1)
        for i, p in enumerate(pred):
            if label[i] == torch.max(p.data, 0)[1]:
                total = total + 1

    accuracy = total / len(mnist_valset)
    pred_accuracy.append(accuracy)

    total_val_loss = total_val_loss / (itr + 1)
    val_loss.append(total_val_loss)

    print('\nEpoch: {}/{}, Train Loss: {:.8f}, Val Loss: {:.8f}, Val Accuracy: {:.8f}'.
format(epoch + 1, no_epochs, total_train_loss, total_val_loss, accuracy))

    if total_val_loss < best_val_loss:
        best_val_loss = total_val_loss
        print("Saving the model state dictionary for Epoch: {} with Validation loss:
{:.8f}".format(epoch + 1, total_val_loss))
        torch.save(model.state_dict(), "model.dth")

fig=plt.figure(figsize=(4, 4))
plt.plot(np.arange(1, no_epochs+1), train_loss, label="Train loss")
plt.plot(np.arange(1, no_epochs+1), val_loss, label="Validation loss")
plt.xlabel('Epochs')
plt.ylabel('Loss')
```

```
plt.title("Loss Plots")
plt.legend(loc='upper right')
plt.savefig('loss.png')


fig=plt.figure(figsize=(4, 4))
plt.plot(np.arange(1, no_epochs+1), pred_accuracy, label="Prediction accuracy")
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title("Accuracy plot")
plt.legend(loc='upper right')
plt.savefig('accuracy.png')
```

```
Epoch: 1/10, Train Loss: 0.09373666, Val Loss: 0.04637084, Val Accuracy:
0.98400000
Saving the model state dictionary for Epoch: 1 with Validation loss: 0.046
37084

Epoch: 2/10, Train Loss: 0.04584939, Val Loss: 0.03603472, Val Accuracy:
0.98811111
Saving the model state dictionary for Epoch: 2 with Validation loss: 0.036
03472

Epoch: 3/10, Train Loss: 0.03319882, Val Loss: 0.03621881, Val Accuracy:
0.98866667

Epoch: 4/10, Train Loss: 0.02587089, Val Loss: 0.03214301, Val Accuracy:
0.99000000
Saving the model state dictionary for Epoch: 4 with Validation loss: 0.032
14301

Epoch: 5/10, Train Loss: 0.02062521, Val Loss: 0.03626806, Val Accuracy:
0.98833333

Epoch: 6/10, Train Loss: 0.01678899, Val Loss: 0.02350841, Val Accuracy:
0.99233333
Saving the model state dictionary for Epoch: 6 with Validation loss: 0.023
50841

Epoch: 7/10, Train Loss: 0.01248081, Val Loss: 0.02666156, Val Accuracy:
0.99066667

Epoch: 8/10, Train Loss: 0.01181077, Val Loss: 0.02510144, Val Accuracy:
0.99211111

Epoch: 9/10, Train Loss: 0.01081694, Val Loss: 0.03471209, Val Accuracy:
0.98866667

Epoch: 10/10, Train Loss: 0.00784686, Val Loss: 0.03297928, Val Accuracy:
0.99066667
```



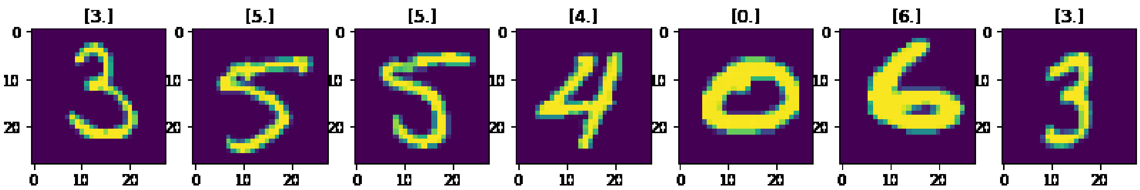


In [15]:

```
for itr, (image, label) in enumerate(test_dataloader):
        pred = model(image)
        pred_label = np.zeros((64,1))
        pred = torch.nn.functional.softmax(pred, dim=1)
        for i, p in enumerate(pred):
            pred_label[i] = int(torch.max(p.data, 0)[1])
```

In [16]:

```
fig=plt.figure(figsize=(20, 10))
for test_images, test_labels in test_dataloader:
    for i in range(1,8):
        sample_image = test_images[i]
        sample_label = pred_label[i]
        img = transforms.ToPILImage(mode='L')(sample_image)
        fig.add_subplot(1, 10, i)
        plt.title(str(sample_label))
        plt.imshow(img)
plt.show()
```

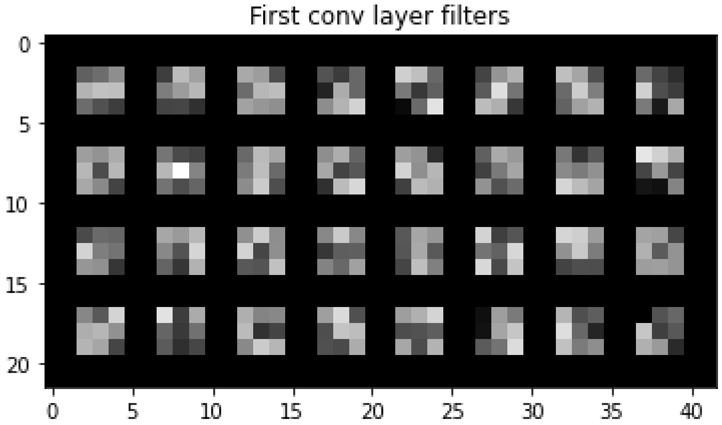


2. Visualizing the Convolutional Neural Network ¶

Convolution layer filters

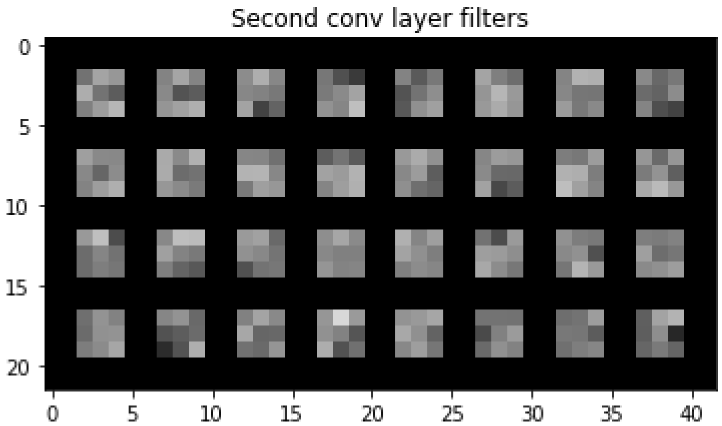
In [17]:

```
kernel1 = model.layer1[0].weight.detach().clone()
kernel1 = kernel1 - kernel1.min()
kernel1 = kernel1/kernel1.max()
img1 = make_grid(kernel1)
plt.imshow(img1.permute(1, 2, 0))
plt.title("First conv layer filters")
plt.show()
```



In [18]:

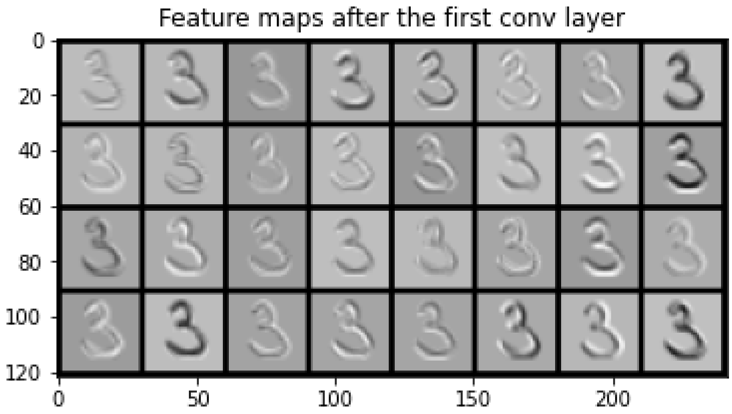
```
kernel2 = model.layer2[0].weight.detach().clone()
kernel2 = kernel2 - kernel2.min()
kernel2 = kernel2/kernel2.max()
temp_kernel = kernel2[10,:, :, :].reshape(32, 1, 3, 3)
img2 = make_grid(temp_kernel)
plt.imshow(img2.permute(1, 2, 0))
plt.title("Second conv layer filters")
plt.show()
```



Feature maps

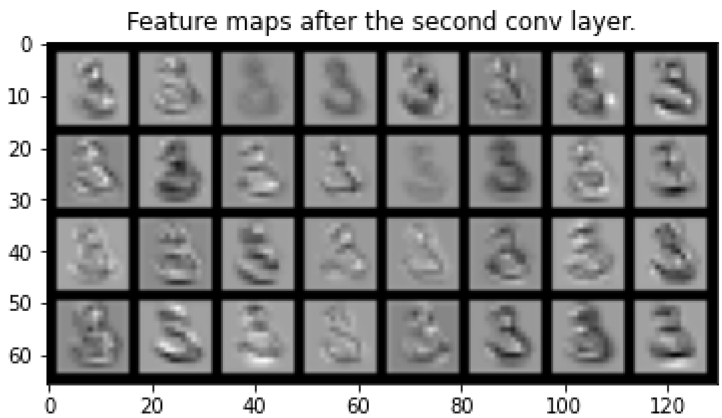
In [19]:

```
for test_images, test_labels in test_dataloader:
    test_image = test_images[1].clone().reshape(
                    1, 1, 28, 28).clone().float()
with torch.no_grad():
    layer_1_output = model.layer1[0].forward(test_image).reshape(32, 1, 28, 28)
    layer_1_output = layer_1_output.cpu()
    layer_1_output = layer_1_output - layer_1_output.min()
    layer_1_output = layer_1_output/layer_1_output.max()
    img = make_grid(layer_1_output)
    plt.imshow(img.permute(1, 2, 0))
    plt.title("Feature maps after the first conv layer")
    plt.show()
```



In [20]:

```
with torch.no_grad():
    layer_2_output = model.layer1.forward(test_image)
    layer_2_output = model.layer2[0].forward(layer_2_output)
    layer_2_output = layer_2_output.cpu()
layer_2_output = layer_2_output - layer_2_output.min()
layer_2_output = layer_2_output/layer_2_output.max()
layer_2_output = layer_2_output.reshape(32, 1, 14, 14)
img = make_grid(layer_2_output)
plt.imshow(img.permute(1, 2, 0))
plt.title("Feature maps after the second conv layer.")
plt.show()
```



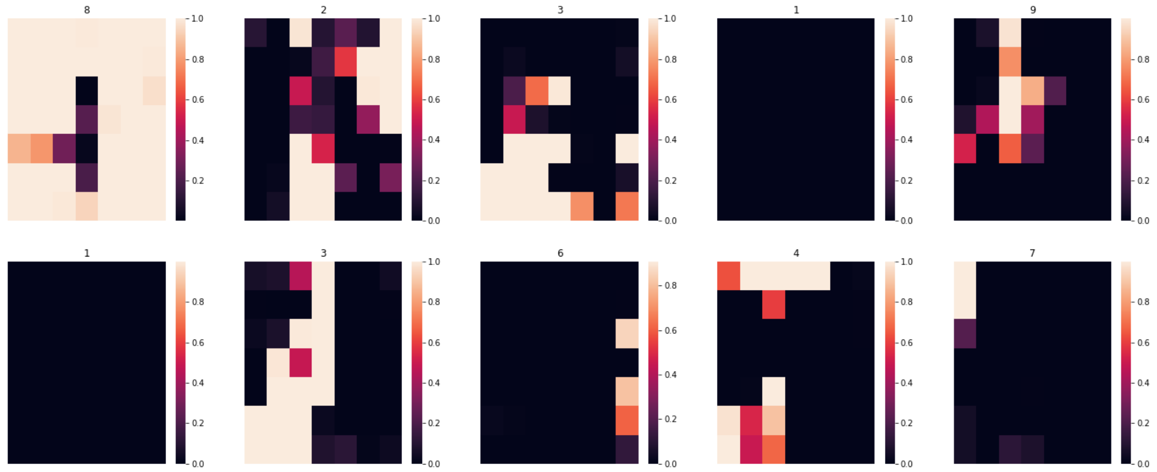
Occlusion experiment

In [21]:

```
def occlusion(model, image, label, occ_size, occ_stride, occ_pixel):

    #get the width and height of the image
    width, height = image.shape[-2], image.shape[-1]

    #setting the output image width and height
    output_height = int(np.ceil((height-occ_size)/occ_stride))
    output_width = int(np.ceil((width-occ_size)/occ_stride))

    #create a white image of sizes we defined
    heatmap = torch.zeros((output_height, output_width))

    #iterate all the pixels in each column
    for h in range(0, height):
        for w in range(0, width):

            h_start = h*occ_stride
            w_start = w*occ_stride
            h_end = min(height, h_start + occ_size)
            w_end = min(width, w_start + occ_size)

            if (w_end) >= width or (h_end) >= height:
                continue

            input_image = image.clone().detach()

            #replacing all the pixel information in the image with occ_pixel(grey) in t
he specified location
            input_image[:, :, w_start:w_end, h_start:h_end] = occ_pixel

            #run inference on modified image
            output = model(input_image)
            output = nn.functional.softmax(output, dim=1)
            prob = output.tolist()[0][label]

            #setting the heatmap location to probability value
            heatmap[h, w] = prob

    return heatmap
```

In [22]:

```
import seaborn as sns
#looking at data using iter
dataiter = iter(train_dataloader)
images, labels = dataiter.next()
f = plt.figure(figsize=(25,10))
for i in range (1,11):
    #shape of images bunch
    image = images[i].unsqueeze(1)
    #print(image.shape)

    outputs = model(image)
    outputs = nn.functional.softmax(outputs, dim = 1)
    prob_no_occ, pred = torch.max(outputs.data, 1)
    #print(pred)
    prob_no_occ = prob_no_occ.item()
    heat_map = occlusion(model, image, labels[i].item(), occ_size=14, occ_stride=2, occ
_pixel=2)
    ax = f.add_subplot(2,5,i)
    #plt.subplot(1,10,i)
    sns.heatmap(heat_map, xticklabels=False, yticklabels=False, vmax=prob_no_occ)
    #plt.title(str(sample_label))
    plt.title(str(labels[i].item()))
```



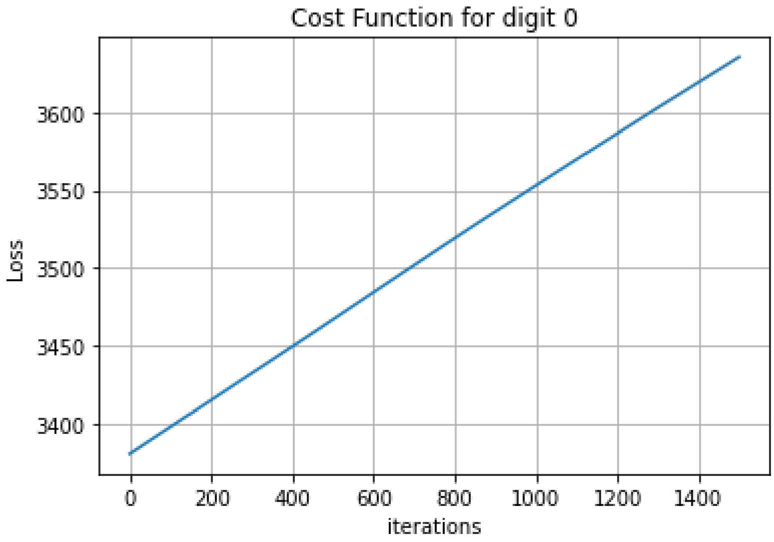
3. Adversarial Examples

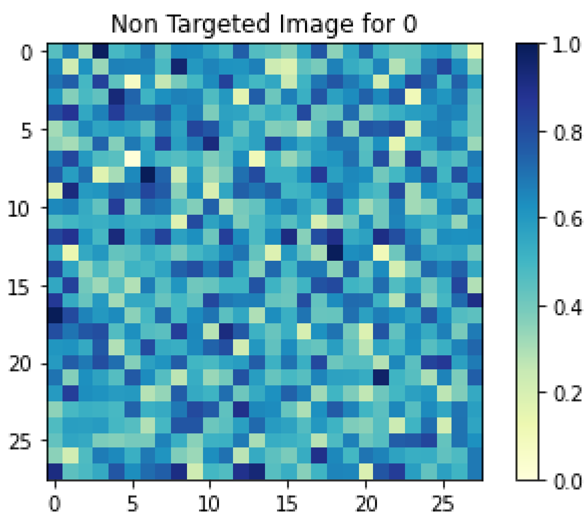
3.1 Non-Targeted Attack

In [23]:

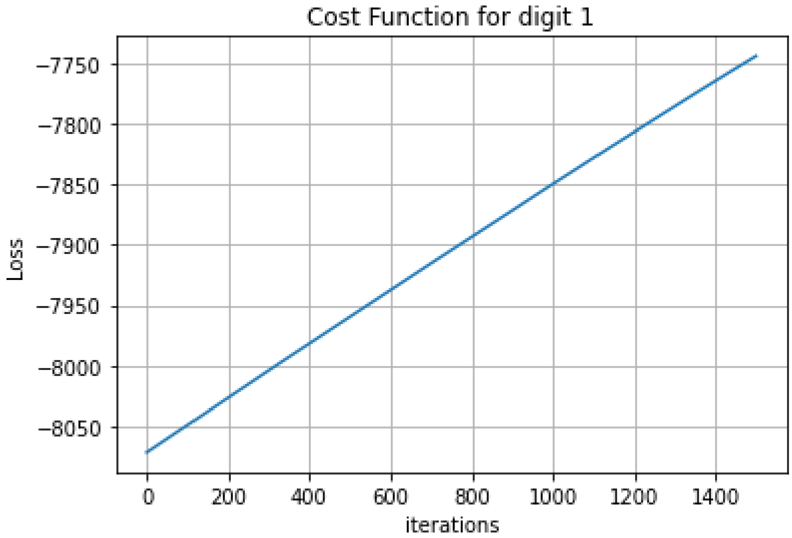
```
from torch import flatten
for j in range(10): # j takes the values from 0 to 9

    target_class = j #target class
    noise = np.random.normal(loc = 128, scale = 8, size = (28,28)).reshape(1,1,28,28) #
centred around 128 with standard deviation 8
    noise = torch.from_numpy(noise)

    noise_tensor = torch.tensor(noise.type(torch.FloatTensor), requires_grad=True)
    #calculating logit values : logit is the network output just before softmax
    logit_cost = []

    optim=torch.optim.Adam([noise_tensor], lr=0.0003)

    for i in range(1500):
        optim.zero_grad()

        #forward pass
        output = model.layer1.forward(noise_tensor)
        output = model.layer2.forward(output)
        output = flatten(output,1)
        output = model.layer3.forward(output)
        logit  = model.layer4.forward(output)
        loss   = -logit[:,target_class] #to convert Gradient Descent to a Gradient Asce
nt

        logit_cost.append(logit[:,target_class].detach().numpy())

        loss.backward(retain_graph = True)

        optim.step()

    plt.plot(np.asfarray(logit_cost))
    plt.title('Cost Function for digit '+str(target_class))
    plt.xlabel('iterations')
    plt.ylabel('Loss')
    plt.grid()
    plt.show()

    #normalization
    NT_plot = noise_tensor.cpu().reshape(28,28).detach().numpy()
    NT_plot = NT_plot - np.min(NT_plot)
    NT_plot = NT_plot/np.max(NT_plot)

    plt.imshow(NT_plot,cmap = "YlGnBu")
    plt.colorbar()
    plt.title("Non Targeted Image for "+str(target_class))
    plt.show()

    pred = model.forward(noise_tensor).detach().numpy() #as it is a single image we dir
ectly run the forward pass
    pred_class = np.argmax(pred) #predicted class
    #print(pred_class)

    confidence = np.exp(pred, order = 'K')[:,target_class] #now in the form of probabil
ities

    print('Confidence in prediction:', confidence)
```

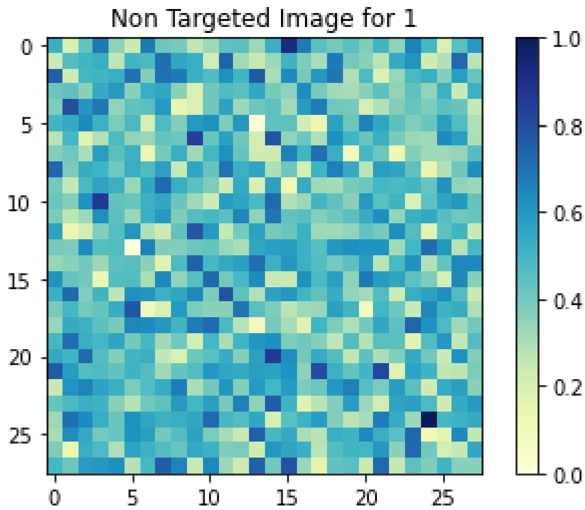
```
/tmp/ipykernel_2182682/1305307155.py:8: UserWarning: To copy construct fro
m a tensor, it is recommended to use sourceTensor.clone().detach() or sour
ceTensor.clone().detach().requires_grad_(True), rather than torch.tensor(s
ourceTensor).
  noise_tensor = torch.tensor(noise.type(torch.FloatTensor), requires_grad
=True)
```



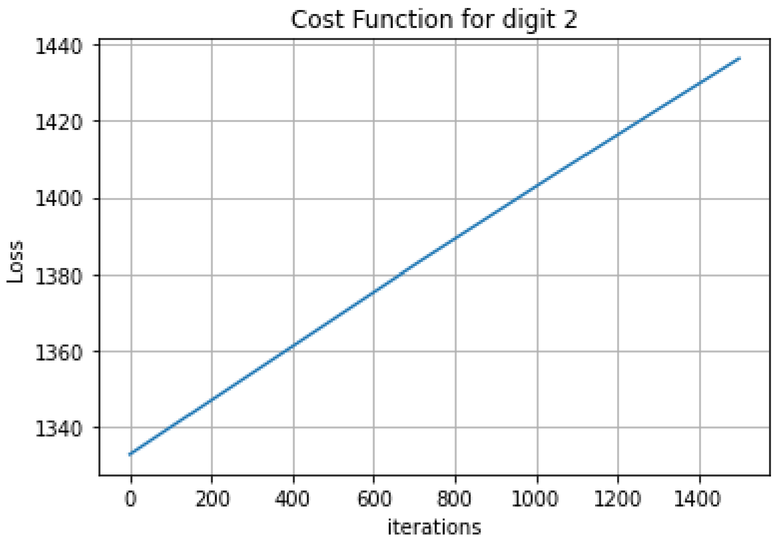


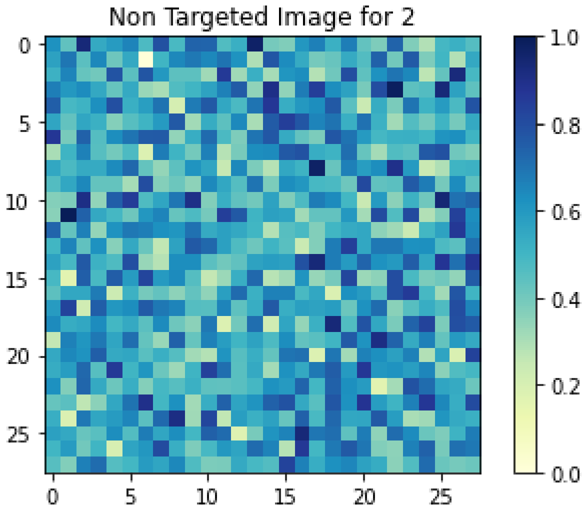

```
Confidence in prediction: [0.]
```



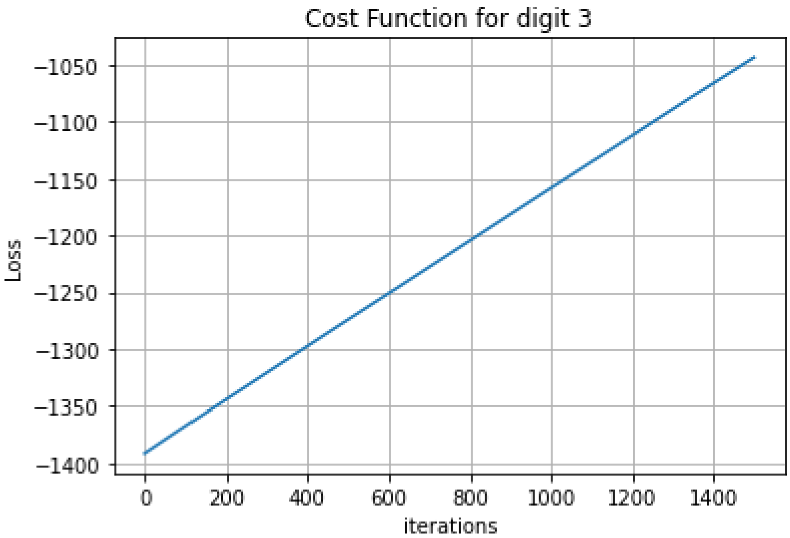


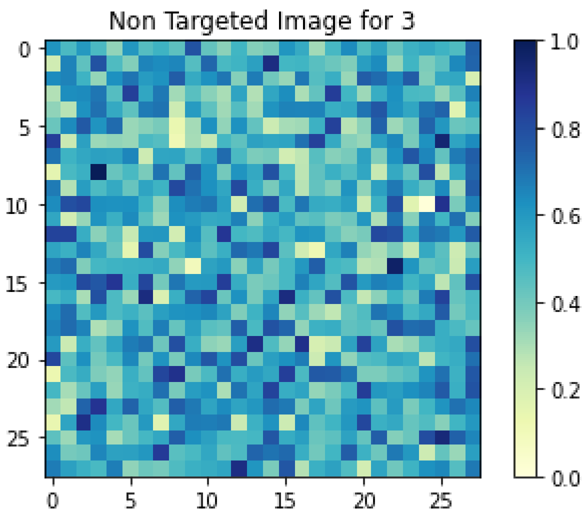

```
Confidence in prediction: [0.]
```



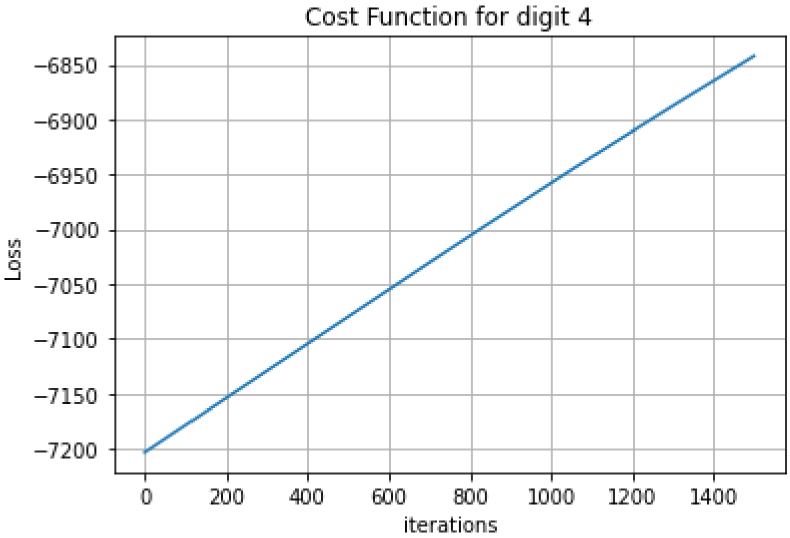


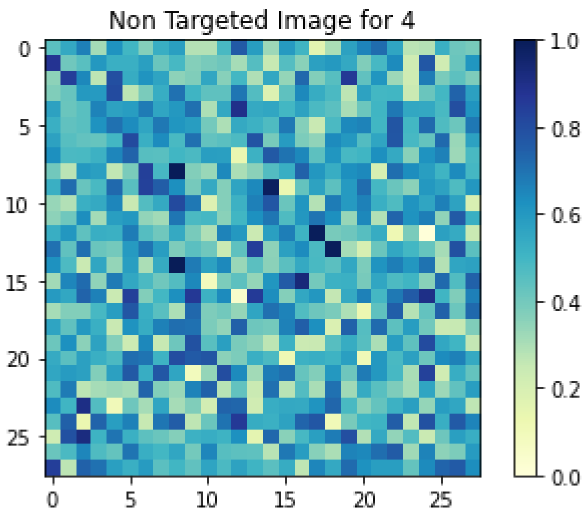

```
Confidence in prediction: [0.]
```



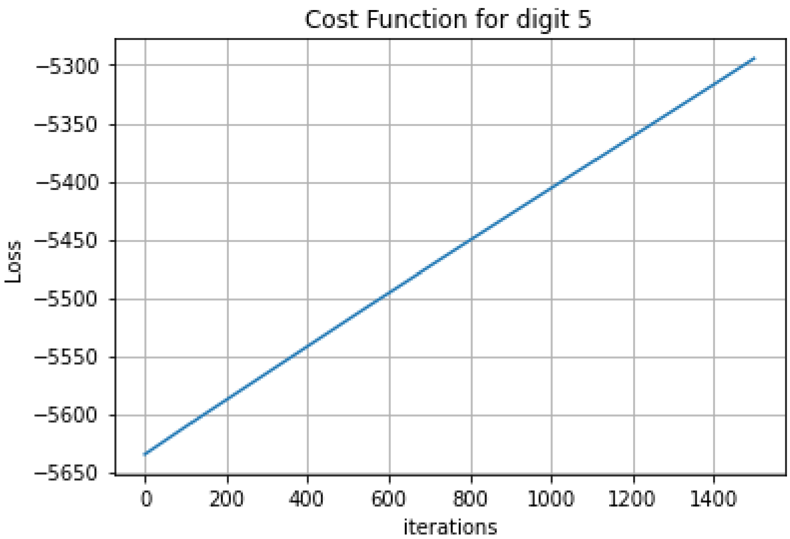


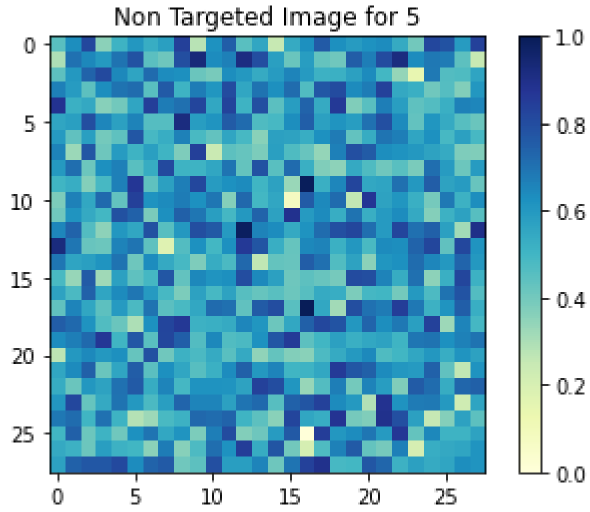

```
Confidence in prediction: [0.]
```



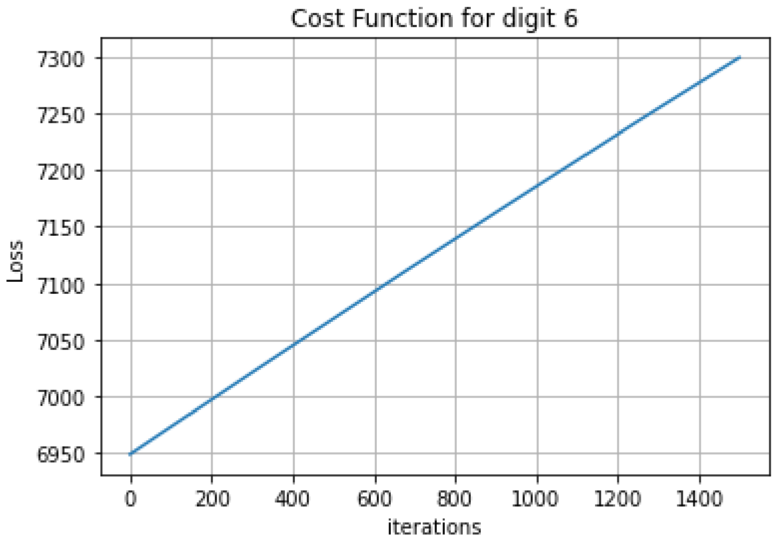


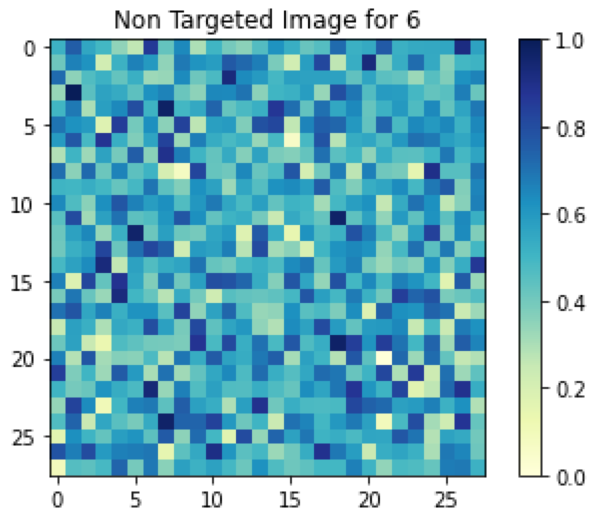

```
Confidence in prediction: [0.]
```



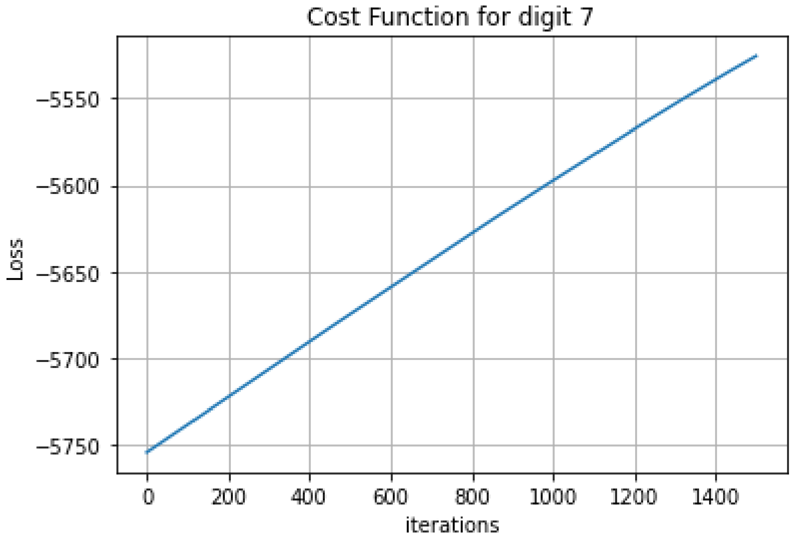


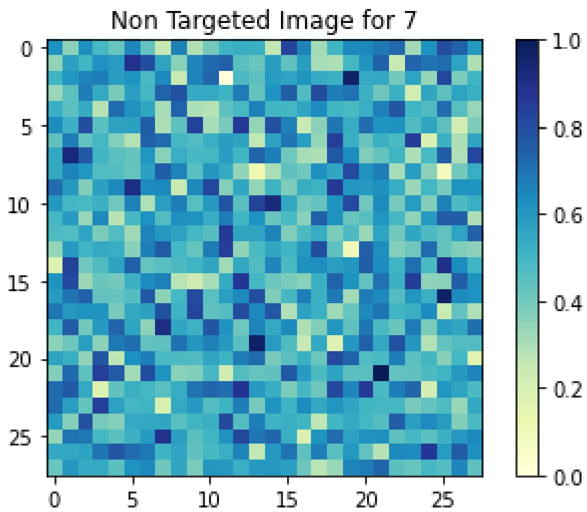
Confidence in prediction: [0.]



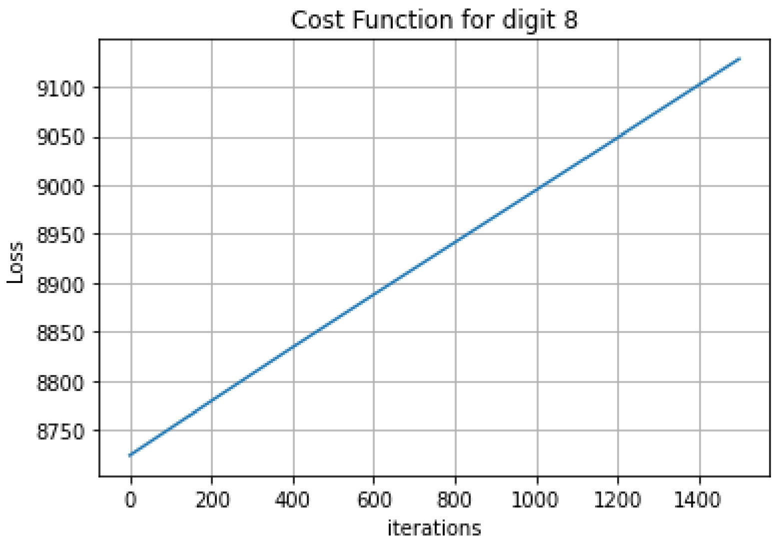


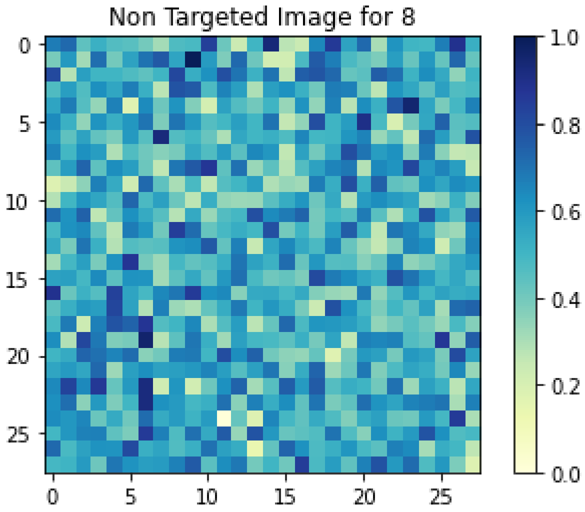
Confidence in prediction: [0.]



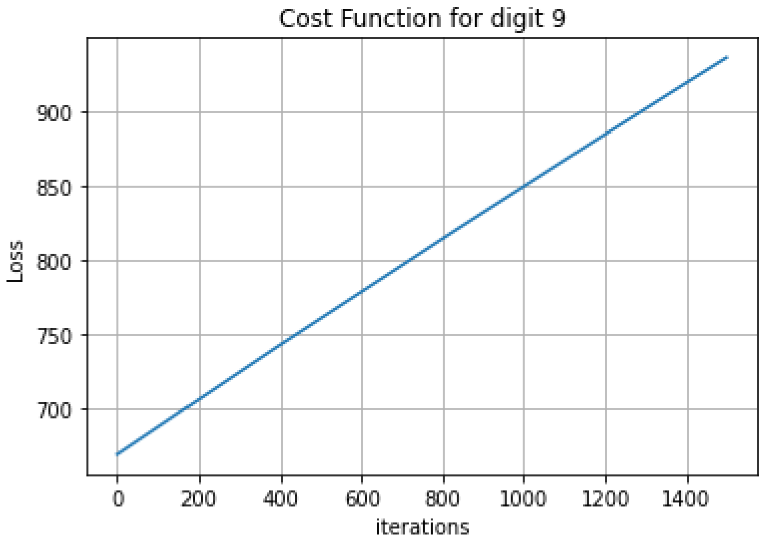


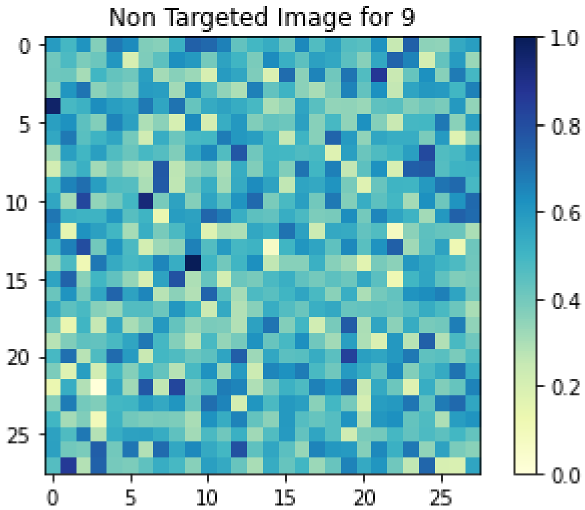

```
Confidence in prediction: [0.]
```






```
Confidence in prediction: [1.]
```






```
Confidence in prediction: [0.]
```

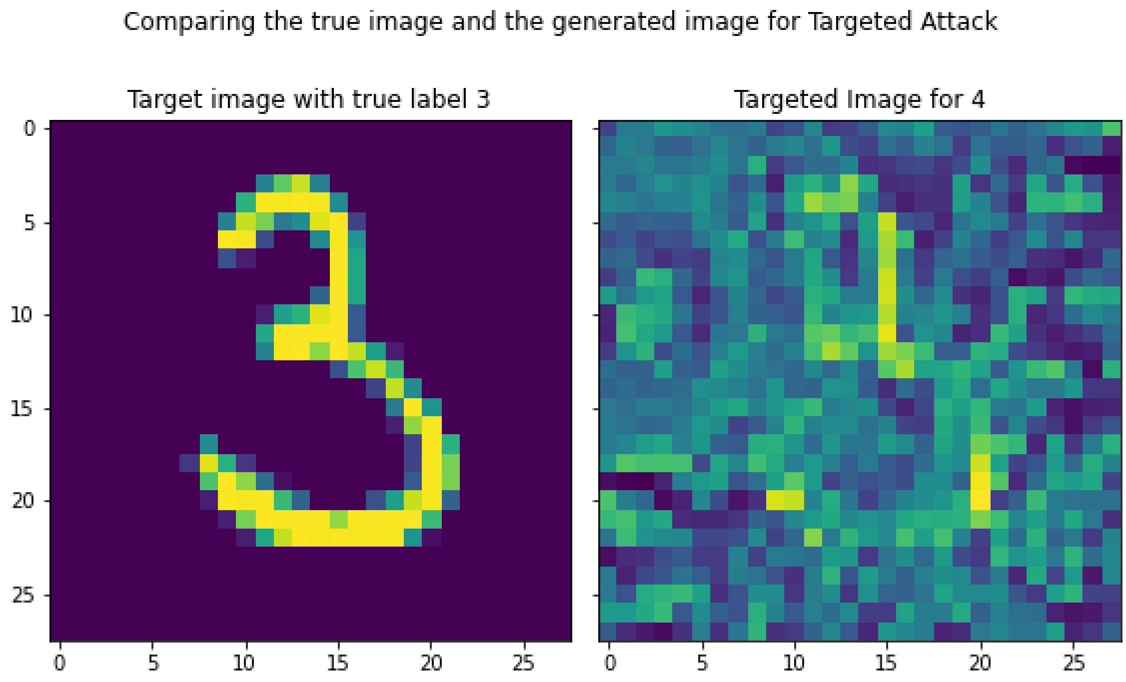
3.2 Non-targeted attack

In [24]:

```
for test_images, test_labels in test_dataloader:
    target_image = test_images[1]
    orig_class = test_labels.data[1].detach().numpy()


beta = 0.001
if orig_class == 9:
    target_class = 0
else:
    target_class = orig_class + 1

target_image = target_image.clone().reshape(1,1,28,28).float()

fig, axs = plt.subplots(1, 2,figsize = (8,8))
axs[0].imshow(target_image.detach().numpy().reshape(28,28)) #our target image
axs[0].set_title("Target image with true label "+str(orig_class))

X = target_image.detach().clone()
X_tensor = torch.tensor(X.type(torch.FloatTensor), requires_grad=True)

optim=torch.optim.Adam([X_tensor], lr=0.0006)
mse_loss = nn.MSELoss()

for i in range(1500):
    optim.zero_grad()
    output = model.layer1.forward(X_tensor)
    output = model.layer2.forward(output)
    output = flatten(output,1)
    output = model.layer3.forward(output)
    logit  = model.layer4.forward(output)

    logit_cost = -logit[:,target_class] #the loss component pertaining to the logits ,
 taking negative of this as we wanna perform gradient ascent
    mse_cost = beta*mse_loss(X_tensor,target_image) #mse loss b/w noise and target imag
e
    t_loss = logit_cost + mse_cost

    t_loss.backward(retain_graph = True)
    optim.step()

#after all the iterations, find the confidence with which it classifies
pred = model.forward(X_tensor).detach().cpu().numpy() #as it is a single image we direc
tly run the forward pass
pred_class = np.argmax(pred) #predicted class
print(pred_class)

#pred is in the form of log_softmax and therefore, we take an exponent to convert them
 to probability
confidence = np.exp(pred)[:,target_class] #now in the form of probabilities

print('Confidence in prediction:', confidence)

#normalization
NT_plot = X_tensor.cpu().reshape(28,28).detach().numpy()
NT_plot = NT_plot - np.min(NT_plot)
NT_plot = NT_plot/np.max(NT_plot)

axs[1].imshow(NT_plot)
axs[1].set_title("Targeted Image for "+str(target_class))
```

```
# Hide x labels and tick labels for top plots and y ticks for right plots.
for ax in axs.flat:
    ax.label_outer()

fig.suptitle("Comparing the true image and the generated image for Targeted Attack ",x
= 0.5,y=0.8)
fig.tight_layout()
fig.show()
```

```
/tmp/ipykernel_2182682/1767801413.py:19: UserWarning: To copy construct fr
om a tensor, it is recommended to use sourceTensor.clone().detach() or sou
rceTensor.clone().detach().requires_grad_(True), rather than torch.tensor
(sourceTensor).
  X_tensor = torch.tensor(X.type(torch.FloatTensor), requires_grad=True)

4
Confidence in prediction: [1.]
```



3.3 Adding Noise

In [25]:

```
for test_images, test_labels in test_dataloader:
    original_image1 = test_images[2]
    original_class1 = test_labels.data[2].detach().numpy()
    original_image2 = test_images[1]
    original_class2 = test_labels.data[1].detach().numpy()

if original_class1 == 9:
    target_class1 = 0
else:
    target_class = original_class1 + 1

fig,axs = plt.subplots(1,5,figsize = (20,20))

axs[0].imshow(original_image1.detach().cpu().numpy().reshape(28,28)) #our original imag
e
axs[0].set_title("Original image with true label "+str(original_class1))

noise = np.zeros((1,1,28,28),dtype = np.float) #initializing noise matrix
noise = torch.from_numpy(noise)
noise_tensor = torch.tensor(noise.type(torch.FloatTensor), requires_grad=True) #get noi
se tensor

#using ADAM for gradient ascent
optim=torch.optim.Adam([noise_tensor], lr=0.0003)
for i in range(1500):
    X = original_image1 + noise_tensor

    optim.zero_grad()
    output = model.layer1.forward(X)
    output = model.layer2.forward(output)
    output = flatten(output,1)
    output = model.layer3.forward(output)
    logit  = model.layer4.forward(output)

    n_logit_cost = -logit[:,target_class] #the loss component pertaining to the logits
 , taking negative of this as we wanna perform gradient ascent
    n_logit_cost.backward(retain_graph = True)

    optim.step() #gradient ascent wrt noise this time

X = original_image1 + noise_tensor
pred = model.forward(X).detach().numpy() #as it is a single image we directly run the f
orward pass
pred_class = np.argmax(pred) #predicted class
print('Predicted Class: ',pred_class)
confidence = np.exp(pred)[:,target_class] #now in the form of probabilities
print('Confidence in prediction of noise added image:', confidence)

NT_plot = X.reshape(28,28).detach().numpy()
NT_plot = NT_plot - np.min(NT_plot)
NT_plot = NT_plot/np.max(NT_plot)

axs[1].imshow(NT_plot)
axs[1].set_title("Noise Added Image predicted as "+str(pred_class))

#plot for noise
NT_plot = noise_tensor.cpu().reshape(28,28).detach().numpy()
NT_plot = NT_plot - np.min(NT_plot)
NT_plot = NT_plot/np.max(NT_plot)
```

```
axs[4].imshow(NT_plot,vmin = 0,vmax = 1)
axs[4].set_title("Noise Matrix")

X = original_image2 + noise_tensor
pred = model.forward(X).detach().numpy() #as it is a single image we directly run the f
orward pass
pred_class = np.argmax(pred) #predicted class
print('Predicted Class: ',pred_class)
confidence = np.exp(pred)[:,target_class] #now in the form of probabilities
print('Confidence in prediction:', confidence)

#normalization
#plot for the noise added image
NT_plot = X.cpu().reshape(28,28).detach().numpy()
NT_plot = NT_plot - np.min(NT_plot)
NT_plot = NT_plot/np.max(NT_plot)

axs[3].imshow(NT_plot)
axs[3].set_title("Noise Added Image predicted as  "+str(pred_class))

#plot for new image
axs[2].imshow(original_image2.detach().cpu().numpy().reshape(28,28)) #our original imag
e
axs[2].set_title("Original image with true label "+str(original_class2))

for ax in axs.flat:
    ax.label_outer()

fig.suptitle("Noise Addition Experiment with target class: "+str(target_class),x = 0.5,
y=0.7,fontsize = 20)
fig.tight_layout()
fig.show()
```

```
/tmp/ipykernel_2182682/580360504.py:17: DeprecationWarning: `np.float` is
a deprecated alias for the builtin `float`. To silence this warning, use `
float` by itself. Doing this will not modify any behavior and is safe. If
you specifically wanted the numpy scalar type, use `np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.or
g/devdocs/release/1.20.0-notes.html#deprecations
  noise = np.zeros((1,1,28,28),dtype = np.float) #initializing noise matri
x
/tmp/ipykernel_2182682/580360504.py:19: UserWarning: To copy construct fro
m a tensor, it is recommended to use sourceTensor.clone().detach() or sour
ceTensor.clone().detach().requires_grad_(True), rather than torch.tensor(s
ourceTensor).
  noise_tensor = torch.tensor(noise.type(torch.FloatTensor), requires_grad
=True) #get noise tensor

Predicted Class:  6
Confidence in prediction of noise added image: [1.]
Predicted Class:  6
Confidence in prediction: [1.]
```