EE6132 Assignment No. 2: Report

-Akash Reddy A, EE17B001

September 26, 2019

Introductory Notes

PyTorch has been leveraged to construct all the models in this assignment and answer the sub-questions as well. `torchvision.datasets` and `torchvision.transforms` have been used to extract the MNIST datasets directly from the website and convert them into Tensors. The code has been written into a Google Colab notebook, linked here.

1 MNIST Classification Using CNN

Question 1

Validation error and accuracy are plotted once every epoch just to reduce the training time of the program. We observe a clearly decreasing loss and an increasing accuracy with training/iterations. Final Test Accuracy is 97.96%.

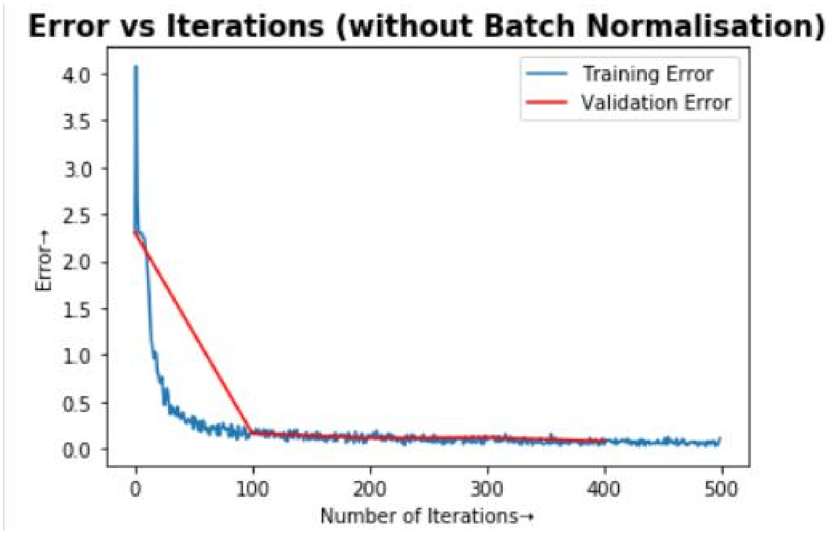


Figure 1: Training and Validation Errors vs Iterations

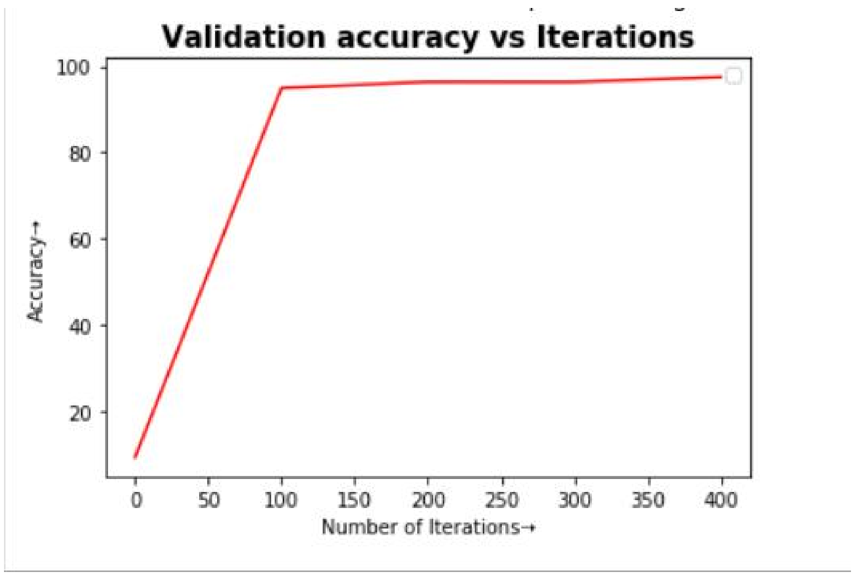
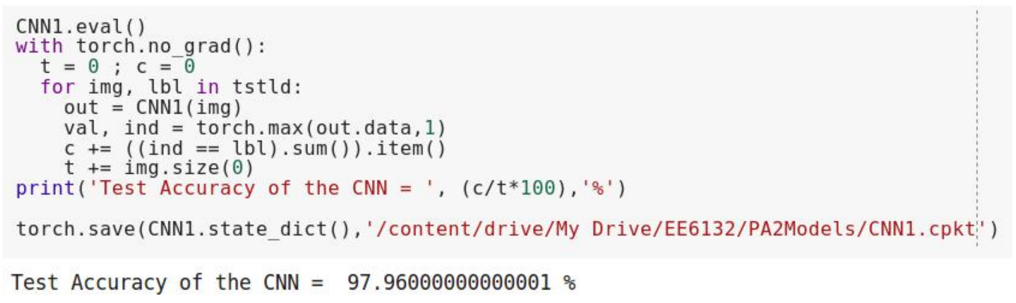


Figure 2: Validation Accuracy vs Iterations(Epochs)



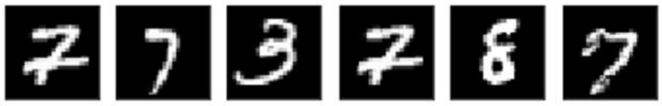
```
CNN1.eval()
with torch.no_grad():
  t = 0 ; c = 0
  for img, lbl in tstld:
    out = CNN1(img)
    val, ind = torch.max(out.data,1)
    c += ((ind == lbl).sum()).item()
    t += img.size(0)
print('Test Accuracy of the CNN = ', (c/t*100),'%')

torch.save(CNN1.state_dict(),'/content/drive/My Drive/EE6132/PA2Models/CNN1.cpkt')
```

```
Test Accuracy of the CNN =  97.96000000000001 %
```

Figure 3: Final Test Accuracy = 97.96%

Question 2

6 images are randomly selected from the test set, and the true labels and predicted labels are printed along with the images themselves.



```
True labels =  [7, 7, 3, 7, 8, 7] , Predicted labels =  [2, 7, 3, 2, 8, 7]
```

Figure 4: True and Predicted Labels of 6 Random Images

We see that the first image, 7, is mis-classified as a 2. This is a testament to the fact that our test accuracy is ≈ 97% and not 100%. The first image in fact does look like a 2, with the lower horizontal line. Therefore, this error is understandable.

Question 3

Convolutional Layer 1

At Layer 1, there are 1 input channel, 32 output channels, and 28 $\times$ 28 pixels in images. Thus,

$$\begin{aligned}\text{Input size} &= 28 \times 28\\ \text{Output size} &= 32 \times 28 \times 28\end{aligned}$$

Max Pool Layer 1

Max Pooling is done with a kernel size 2 $\times$ 2, and the stride value is 2. Thus,

$$\begin{aligned}\text{Input size} &= 32 \times 28 \times 28\\ \text{Output size} &= 32 \times 14 \times 14\end{aligned}$$

Convolutional Layer 2

There are 32 input channels, 32 output channels, and 14 $\times$ 14 pixels in images. Thus,

$$\begin{aligned}\text{Input size} &= 32 \times 14 \times 14\\ \text{Output size} &= 32 \times 14 \times 14\end{aligned}$$

Max Pool Layer 2

Max Pooling is done with a kernel size 2 $\times$ 2, and the stride value is 2. Thus,

$$\begin{aligned}\text{Input size} &= 32 \times 14 \times 14\\ \text{Output size} &= 32 \times 7 \times 7\end{aligned}$$

Fully Connected Layer 1

It has the previous layer's output coming into it, and has 500 neurons with 500 outputs. Thus,

$$\begin{aligned}\text{Input size} &= 32 \times 7 \times 7\\ \text{Output size} &= 1 \times 500\end{aligned}$$

Fully Connected Layer 2

It has the previous layer's output coming into it, and has 10 neurons with 10 outputs. Thus,

$$\begin{aligned}\text{Input size} &= 1 \times 500\\ \text{Output size} &= 1 \times 10\end{aligned}$$

Question 4

Convolutional Layer 1

There are 32 convolutional output layers. This would mean 32 different 3 $\times$ 3 filters and 32 different biases for the single input channel.

$$\begin{aligned} \text{Weights} &= 32 \times 3 \times 3 = 288 \\ \text{Biases} &= 32 \\ \text{Total Parameters} &= 320 \end{aligned}$$

Max Pool Layer 1

No parameters.

Convolutional Layer 2

There are 32 convolutional output layers. This would mean 32 different 3 $\times$ 3 filters for each of the 32 input channels and 32 different biases, 1 for each output channel!

$$\begin{aligned} \text{Weights} &= 32 \times 32 \times 3 \times 3 = 9216 \\ \text{Biases} &= 32 \\ \text{Total Parameters} &= 9248 \end{aligned}$$

Max Pool Layer 2

No parameters.

Fully Connected Layer 1

The output of the convolutional layer 2 is now stretched into a single-dimensional vector. The length of this vector would be 32 $\times$ 7 $\times$ 7 = 1568. The fully connected layer itself has 500 neurons. Therefore,

$$\begin{aligned} \text{Weights} &= 1568 \times 500 = 784000 \\ \text{Biases} &= 500 \\ \text{Total Parameters} &= 784500 \end{aligned}$$

Fully Connected Layer 2

This fully connected layer has 10 neurons and a 500-long input. Therefore,

$$\begin{aligned} \text{Weights} &= 500 \times 10 = 5000 \\ \text{Biases} &= 10 \\ \text{Total Parameters} &= 5010 \end{aligned}$$

Ultimately, there are a total of 799078 parameters, of which 789510 are in the fully connected layers and 9568 are in the convolutional layers.

Question 5

Convolutional Layer 1

There are 32 convolutional output layers. Each is a virtual neuron. Thus,

$$\text{Number of neurons} = 32$$

Convolutional Layer 2

Just as in convolutional layer 1, there are 32 convolutional output layers.

$$\text{Number of neurons} = 32$$

Fully Connected Layer 1

$$\text{Number of neurons} = 500$$

Fully Connected Layer 2

$$\text{Number of neurons} = 10$$

Ultimately, there are a total of 574 neurons, of which 510 are in the fully connected layers and 64 are in the convolutional layers.

Question 6

Applying batch-normalisation does improve the test accuracy by ≈1.5%! However, this comes at the cost of slower training time- almost twice the training time as without batch-normalisation- because there is an extra normalisation step involved at each layer of the CNN, at each iteration.

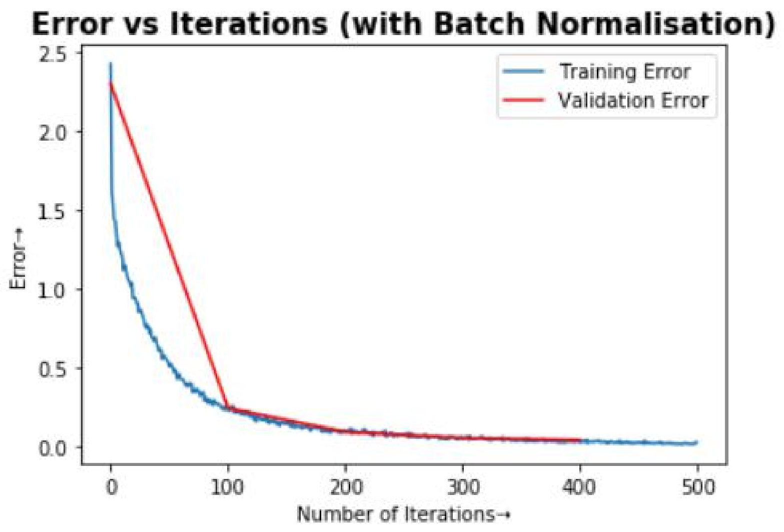


Figure 5: Training and Validation Errors vs Iterations

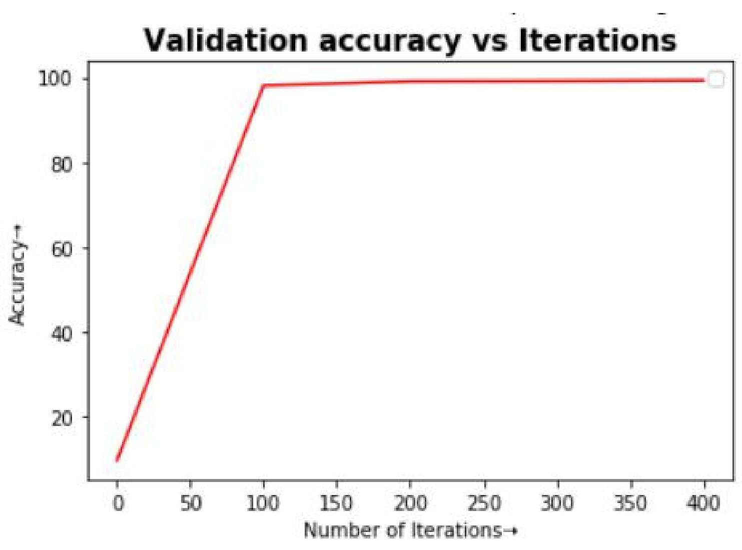
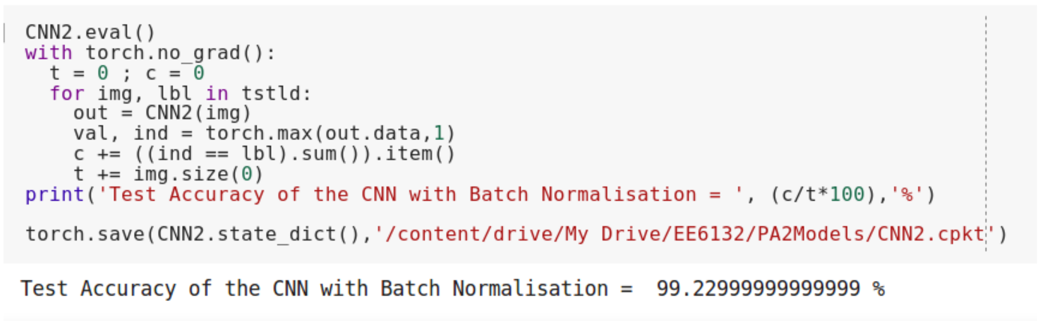


Figure 6: Validation Accuracy vs Iterations(Epochs)



```
CNN2.eval()
with torch.no_grad():
  t = 0 ; c = 0
  for img, lbl in tstld:
    out = CNN2(img)
    val, ind = torch.max(out.data,1)
    c += ((ind == lbl).sum()).item()
    t += img.size(0)
print('Test Accuracy of the CNN with Batch Normalisation = ', (c/t*100),'%')

torch.save(CNN2.state_dict(),'/content/drive/My Drive/EE6132/PA2Models/CNN2.cpkt')
```

```
Test Accuracy of the CNN with Batch Normalisation =  99.22999999999999 %
```

Figure 7: Final Test Accuracy = 99.22%

2 Visualising Convolutional Neural Network

Question 1

The conv1 layer filters seem to be varying less between adjacent pixels. They seem to be divided more clearly into distinct areas of black and white. This is so that the surface-level features are captured, such as the position and rough shape of the digit in the image.

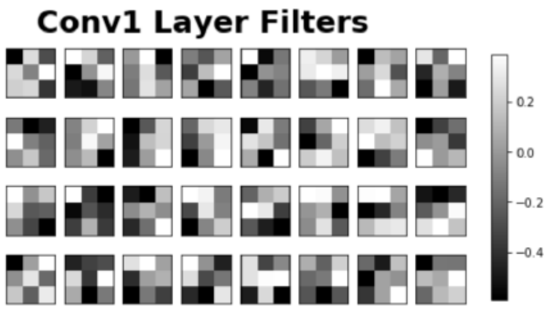


Figure 8: conv1 Layer Filters

Question 2

The conv2 layer filters seem to be varying more between adjacent pixels. They seem to be more intricate as compared to the conv1 filters. This is probably to capture deeper features, such as edges, local shapes, and colour changes in the image.

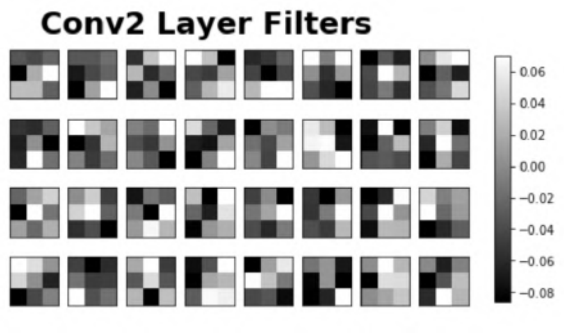


Figure 9: conv2 Layer Filters for the First Input Channel out of 32

Question 3

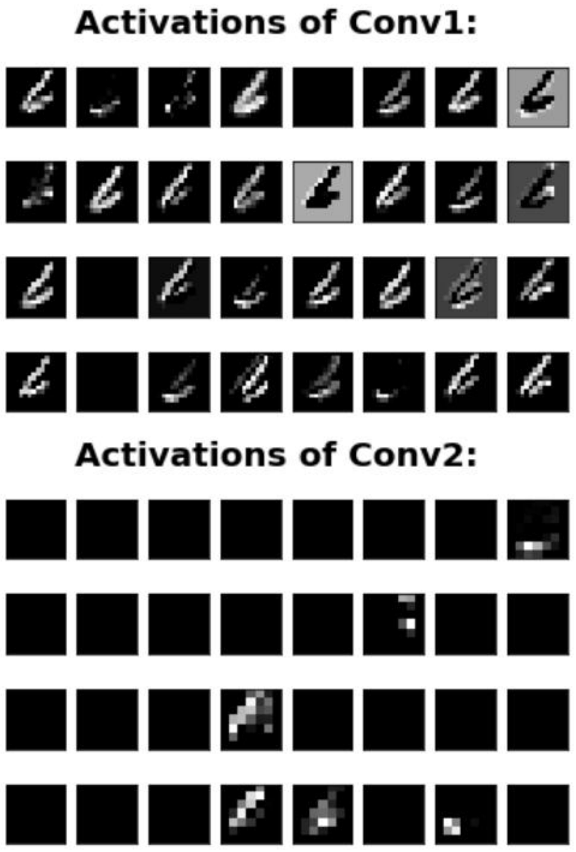


Figure 10: conv1, conv2 Layers' Activations for a Random Input

As we go deeper, we see that the less obvious, deeper features of the input image are extracted. The first layer seems to extract the rough shape and visibility of the 6, the more obvious surface-level features.

Question 4

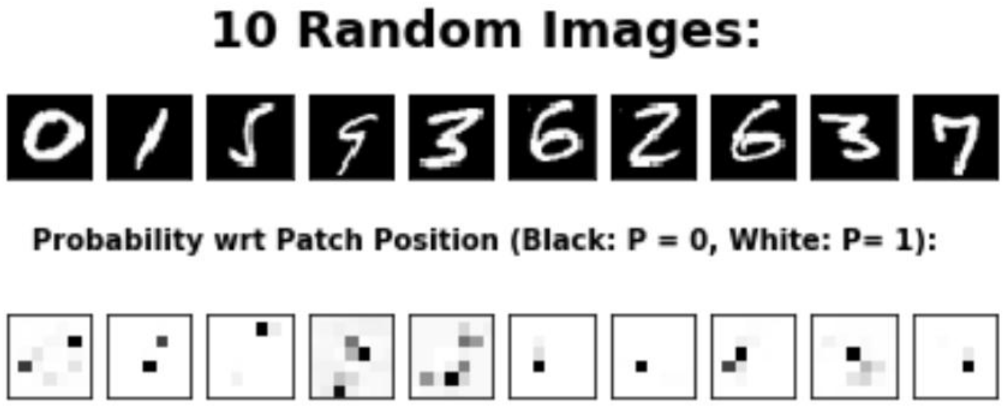


Figure 11: Probability Plots as a Function of Position of an Occluding a Grey Patch

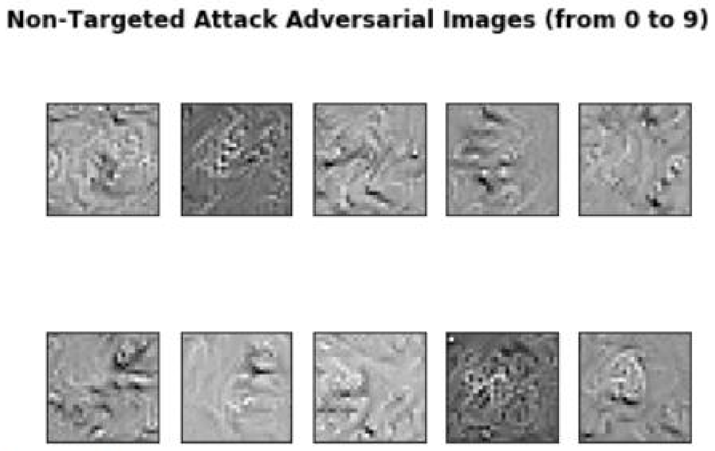
The plots here are meaningful enough to draw useful conclusions. When we occlude the first image (the digit 0) with the grey patch, we find that the probability somewhat drops as the patch covers the important areas of the 0. The highest drop seems to be where, if the patch is present, the 0 can be mistaken to be a 6 or a 9.

Even in the other probability grids, we find that the probability drops visibly when the patch reaches the digit. It drops the most when the patch is at such a location that it is possible to mistake it to be another digit. As another example, the probability for the digit 6 drops the most when they grey patch covers it so that it looks like a 5.

3 Adversarial Examples

Non-Targeted Attack

Question 1



```
Predictions =
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Confidences =
[100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0]
```

Figure 12: Non-Targeted Attack: Generated Images for Each Class

Question 2

Yes, the confidence for each prediction of the target class is very high (100%).

Question 3

The generated images do not look clearly like a number. However, if one looks closely, one can notice the vague resemblances to the digits.

In the first image, there is a round-shaped form located at the centre, which is characteristic of a 0. Looking at the image for 6, one can vaguely see a round-shaped form located towards the lower-half which is typical of a 6. We find a round-shaped form to the top of the last image- typical of a 9. All the other images are also very vaguely representative of the digit they are classified as.

In conclusion, even though the images don't exactly look like the digits (because they start off as random Gaussian noise before they are transformed into the images we see above), they seem to contain some very rough characteristic features of the digits.

Question 4

The cost for each digit is plotted below.

The cost increases with iterations. This is as expected, as we are declaring the cost function to be the output of the target class being predicted (before the softmax function). We are trying to maximise this probability of the target class being predicted through gradient ascent. Therefore, it is no surprise that the cost increases.

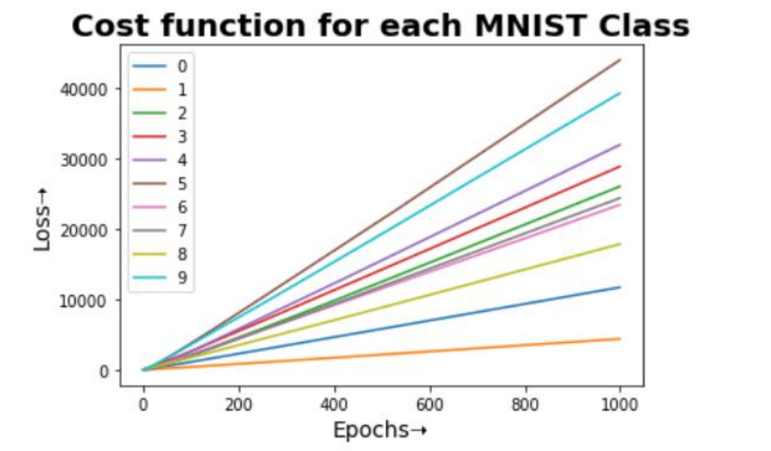
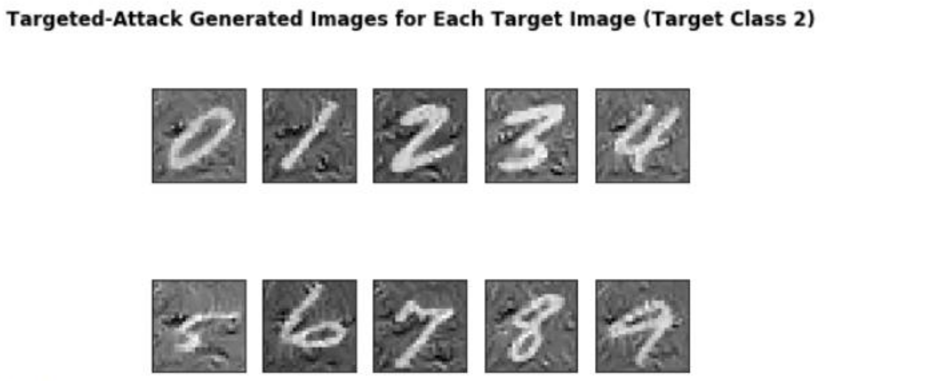


Figure 13: The Value of the logits Cost vs Iterations for All 10 Digits

Targeted Attack

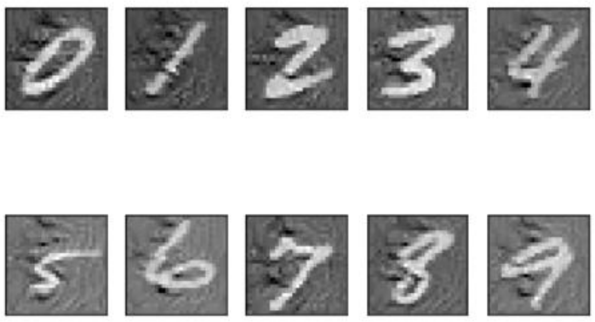
Question 1


Targeted-Attack Generated Images for Each Target Image (Target Class 2)

```
Predictions =
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
Confidences =
[100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0]
```

Figure 14: Targeted Attack: Generated Images for Each Class, Target Class 2

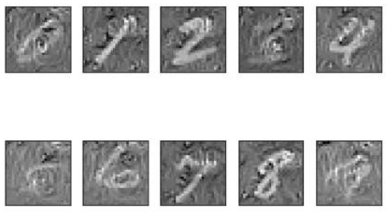
The images have been plotted for all MNIST target *image* classes each, for 3 different *prediction* target classes 2, 3, and 6.



```
Predictions =
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
Confidences =
[100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0]
```

Figure 15: Targeted Attack: Generated Images for Each Class, Target Class 3

Targeted-Attack Generated Images for Each Target Image (Target Class 6)



```
Predictions =
[6, 6, 6, 6, 6, 6, 6, 6, 6, 6]
Confidences =
[100.0, 99.9998688697815, 99.99985694885254, 100.0, 99.9995470046997, 100.0, 100.0, 99.99847412109375, 99.99581575393677, 100.0]
```

Figure 16: Targeted Attack: Generated Images for Each Class, Target Class 6

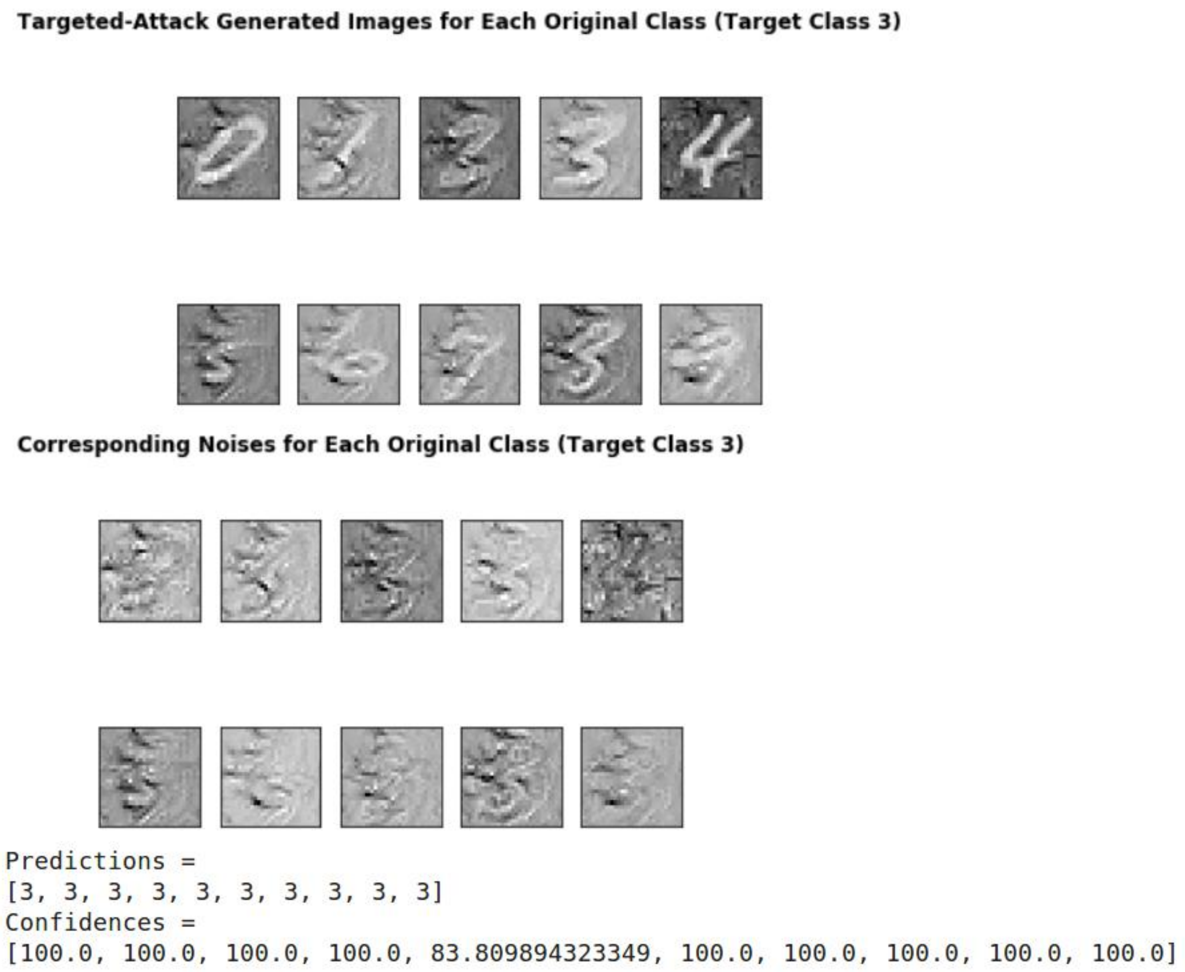
The generated images now do look like numbers as we start from the original number image. However, more subtle features are introduced as the input takes gradient ascent steps, that make the CNN predict *target_class*. In the above figures, the generated image is shown for each target image of the MNIST, keeping the prediction target class fixed.

Question 2 (Adding Noise)

1

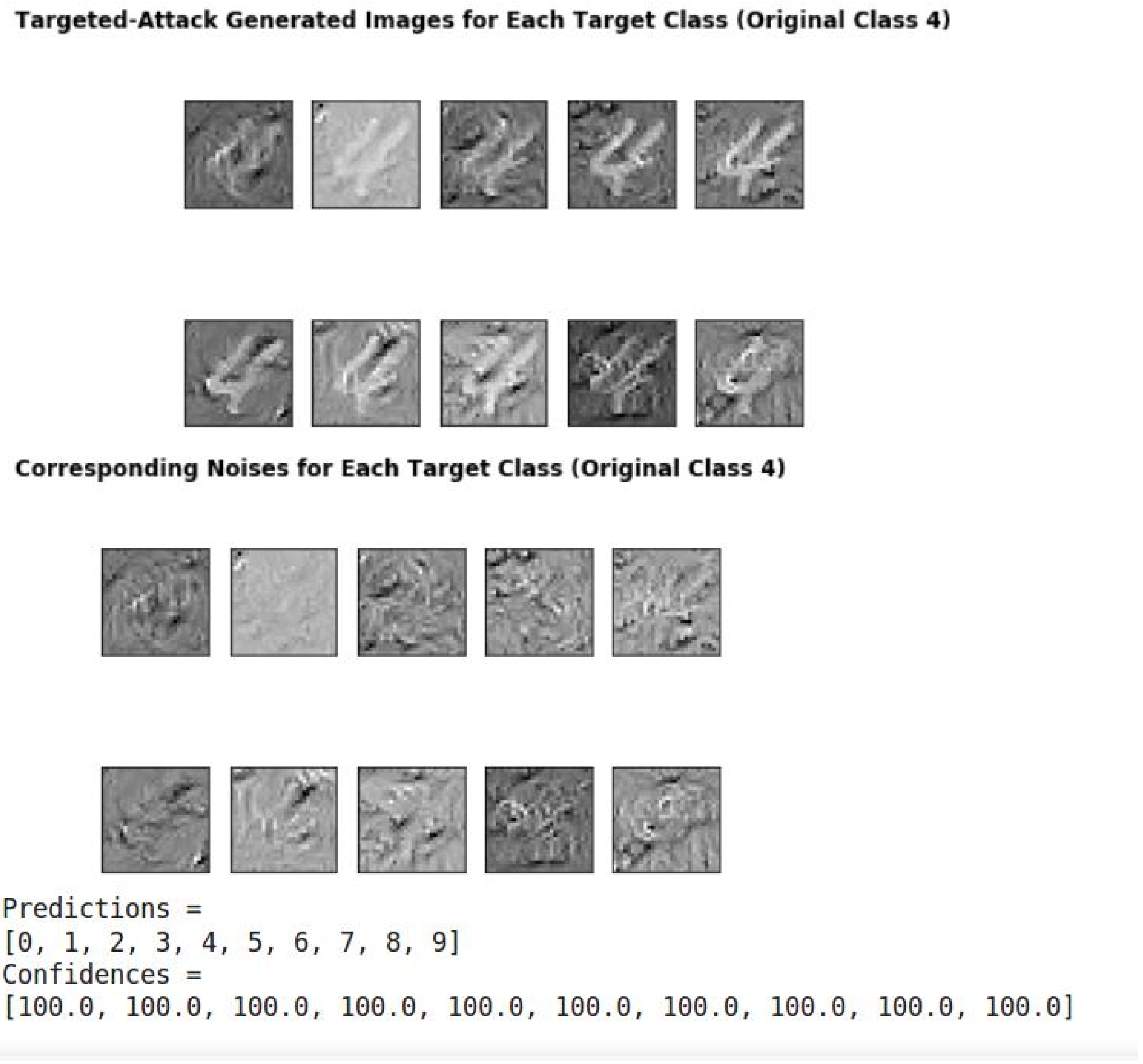
There are too many plots to be plotted if we consider all target classes and all original classes.

Therefore, plots have been made for all original classes keeping target class fixed at 3. Also, plots have been made for all target classes with original classes 4 and 5.


Targeted-Attack Generated Images for Each Original Class (Target Class 3)
Corresponding Noises for Each Original Class (Target Class 3)

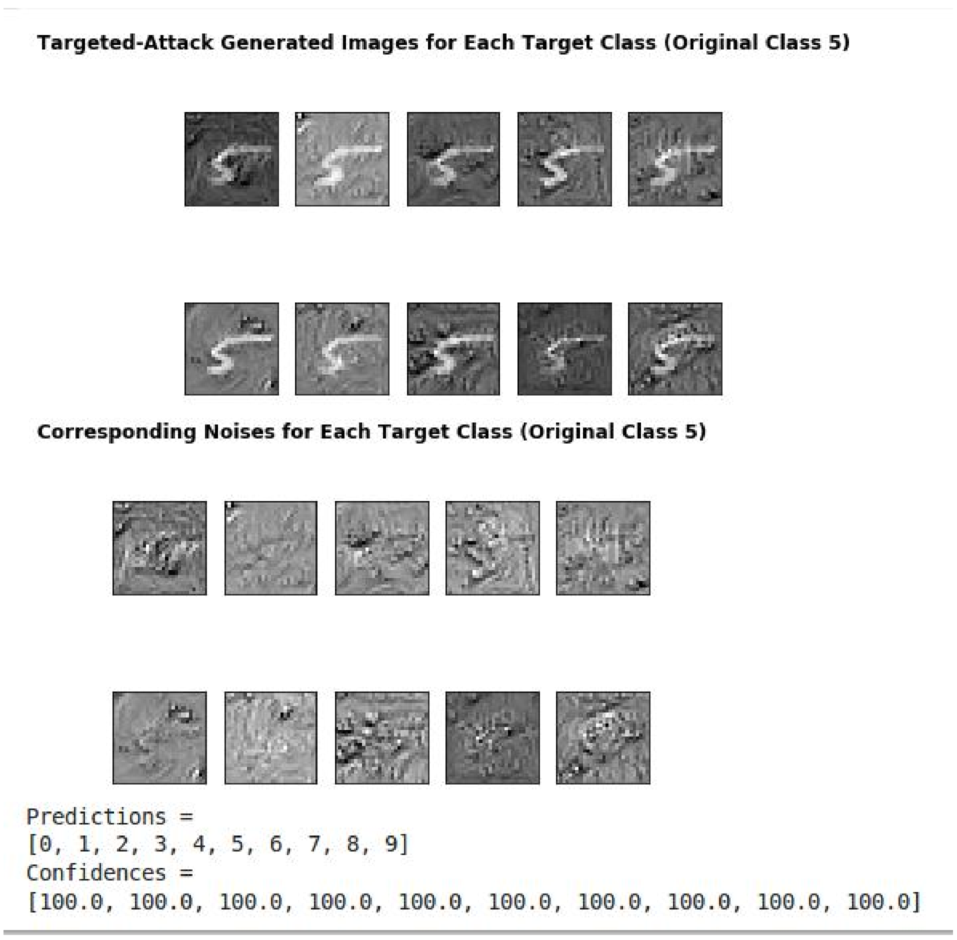
```
Predictions =
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
Confidences =
[100.0, 100.0, 100.0, 100.0, 83.809894323349, 100.0, 100.0, 100.0, 100.0, 100.0]
```

Figure 17: Targeted Attack With Noise: Generated Images and Noise for Each Original Class, Target Class 3. We can see that the noise has vague hints of the digit 3.


Targeted-Attack Generated Images for Each Target Class (Original Class 4)
Corresponding Noises for Each Target Class (Original Class 4)

```
Predictions =
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Confidences =
[100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0]
```

Figure 18: Targeted Attack With Noise: Generated Images and Noise for Each Target Class, Original Class 4. We can see that the noises vaguely visualise 0-9.


Targeted-Attack Generated Images for Each Target Class (Original Class 5)

Corresponding Noises for Each Target Class (Original Class 5)

```
Predictions =
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Confidences =
[100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0]
```

Figure 19: Targeted Attack With Noise: Generated Images and Noise for Each Target Class, Original Class 5. We can see that the noises vaguely visualise 0-9.

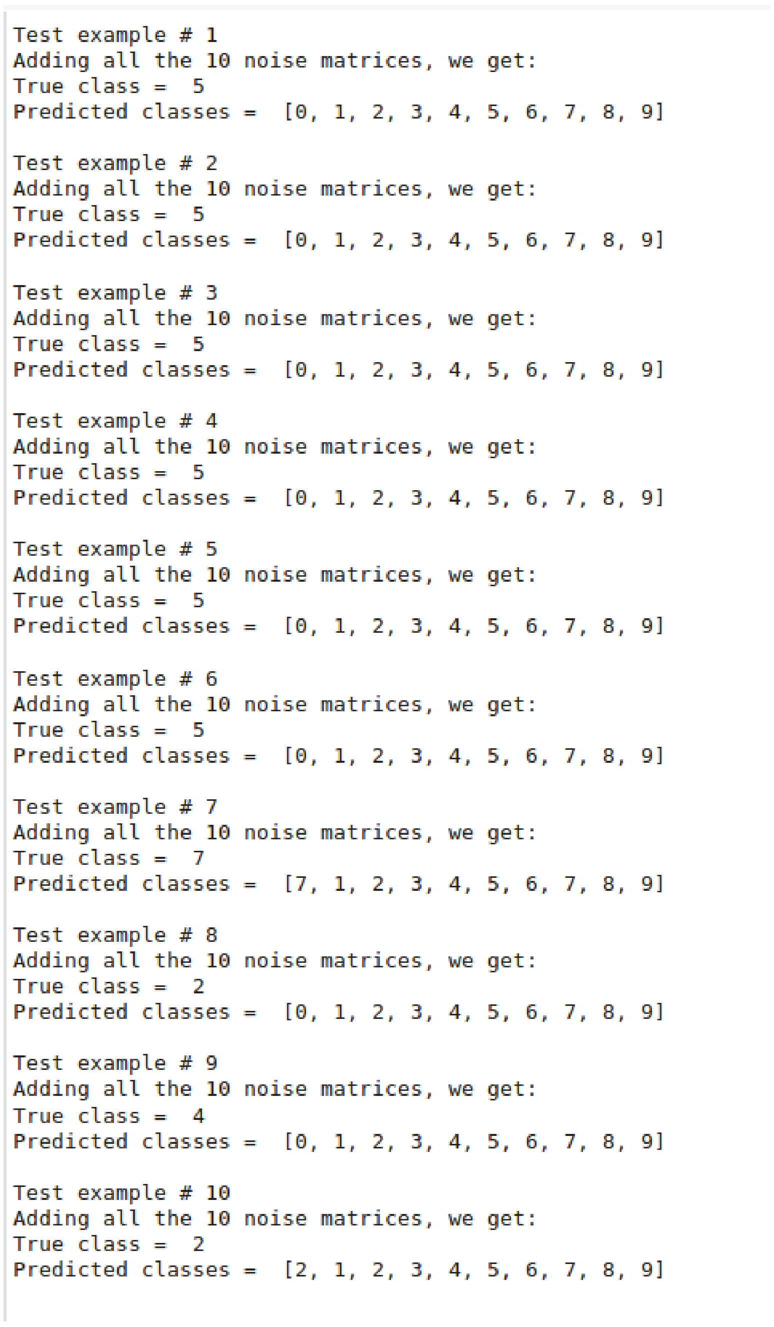
2

We choose the noises we generated for each target class, keeping the original class fixed at 5.

Now, a fixed set of 10 test images is generated: the first 6 images are of the digit 5, and the final 4 images are randomly selected images which are not 5.

After applying the 10 adversarial noise matrices on these 10 images, we find that the predicted classes for all the 5's are the list [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] in that order. However, the latter 4 images sometimes get predicted as the original class itself, therefore our adversarial noise fails.

This is because the adversarial noise has been trained while keeping the original class at 5, therefore it gets predicted as the target class with more effectiveness when the true class is 5.



```
Test example # 1
Adding all the 10 noise matrices, we get:
True class =  5
Predicted classes =  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Test example # 2
Adding all the 10 noise matrices, we get:
True class =  5
Predicted classes =  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Test example # 3
Adding all the 10 noise matrices, we get:
True class =  5
Predicted classes =  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Test example # 4
Adding all the 10 noise matrices, we get:
True class =  5
Predicted classes =  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Test example # 5
Adding all the 10 noise matrices, we get:
True class =  5
Predicted classes =  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Test example # 6
Adding all the 10 noise matrices, we get:
True class =  5
Predicted classes =  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Test example # 7
Adding all the 10 noise matrices, we get:
True class =  7
Predicted classes =  [7, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Test example # 8
Adding all the 10 noise matrices, we get:
True class =  2
Predicted classes =  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Test example # 9
Adding all the 10 noise matrices, we get:
True class =  4
Predicted classes =  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Test example # 10
Adding all the 10 noise matrices, we get:
True class =  2
Predicted classes =  [2, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Figure 20: Targeted Attack With Noise: True and Predicted Labels for 10 Test Images

* *