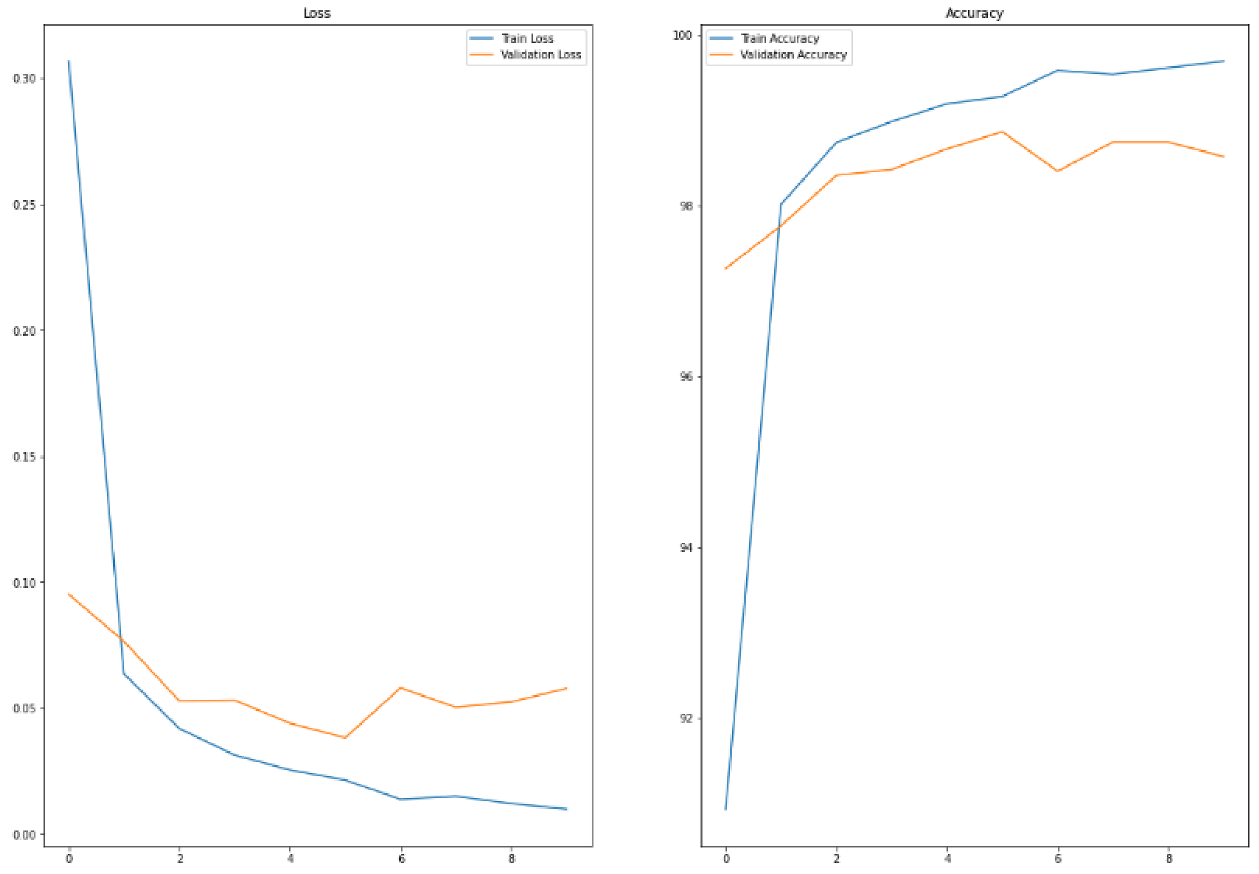
Programming Assignment 2: Convolutional Neural Networks
Done by: Anirud N CE21014

Part 1 Training using CNN

Plots of training and validation errors and Accuracies



After 10 epochs:

Test loss : 0.06

Test Accuracy : 98.57%

Train Loss : 0.01

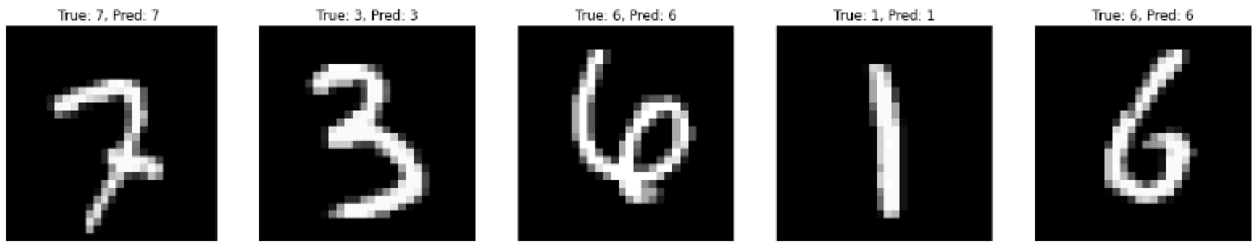
Train accuracy : 99.69%

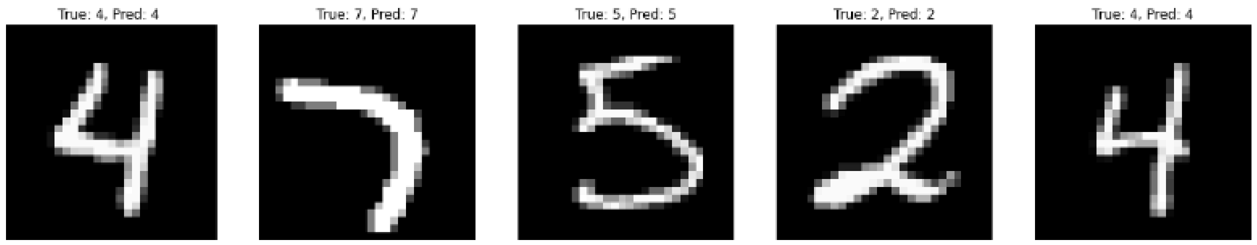
Validation loss : 0.06

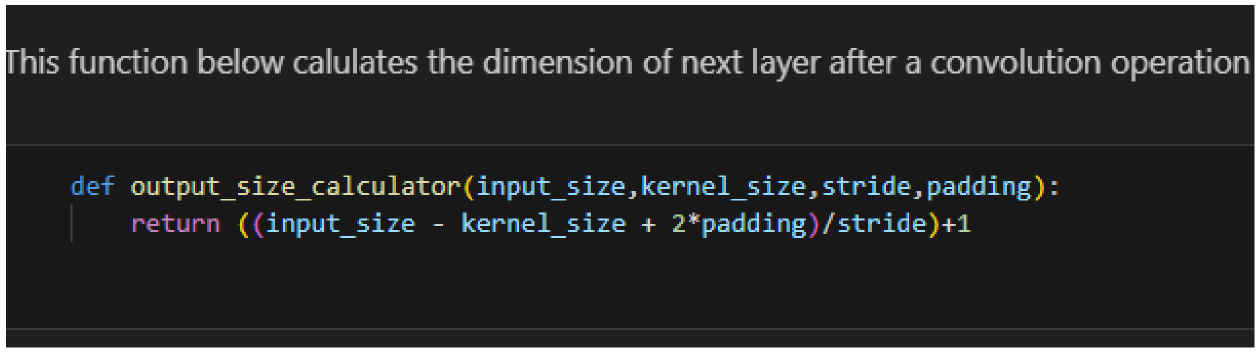
Validation accuracy : 98.57%

Note that, from the plots it is evident that after 5 epochs the model tends to overfit a lot, and so the validation errors increase and the validation accuracy decreases.
SO LATER, I HAVE TRAINED IT AGAIN BUT ONLY FOR 5 EPOCHS AND USED THIS FINAL MODEL ON THE QUESTION 2 AND 3 FOR PREVENTING OVERFITTING

Predictions from Random images of MNIST:







This function below calulates the dimension of next layer after a convolution operation

```
def output_size_calculator(input_size,kernel_size,stride,padding):
    return ((input_size - kernel_size + 2*padding)/stride)+1
```

DIMENSIONS OF OUTPUT CALCULATION:

Conv1 Layer

Input: (batch_size, 1, 28, 28)

Kernel size: 3x3, padding: 1, stride: 1
Output: (batch_size, 32, 28, 28) (32 is the number of output channels)

MaxPool1 Layer

Input: (batch_size, 32, 28, 28)
Kernel size: 2x2, stride: 2
Output: (batch_size, 32, 14, 14)

Conv2 Layer

Input: (batch_size, 32, 14, 14)
Kernel size: 3x3, padding: 1, stride: 1
Output: (batch_size, 32, 14, 14)

MaxPool2 Layer

Input: (batch_size, 32, 14, 14)
Kernel size: 2x2, stride: 2
Output: (batch_size, 32, 7, 7)

Flattening Step

Input: (batch_size, 32, 7, 7)
After flattening: (batch_size, 32 * 7 * 7) = (batch_size, 1568)

Fully Connected Layer (fc1)

Input: (batch_size, 1568)
Output: (batch_size, 500)

Fully Connected Layer (fc2)

Input: (batch_size, 500)
Output: (batch_size, 500)

Output Layer (log_softmax)

Input: (batch_size, 500)
Output: (batch_size, 10)

NUMBER OF PARAMETERS CALCULATION

Conv1 Layer:

Input: 1 channel, Output: 32 channels, Kernel size: 3x3
Parameters = (3 * 3 * 1 * 32) + 32 (bias) = 320 parameters

Conv2 Layer:

Input: 32 channels, Output: 32 channels, Kernel size: 3x3
Parameters = (3 * 3 * 32 * 32) + 32 (bias) = 9,248 parameters

Fully Connected Layers:

fc1: Input size = 1568, Output size = 500
Parameters = 1568 * 500 + 500 (bias) = 784,500 parameters
fc2: Input size = 500, Output size = 500
Parameters = 500 * 500 + 500 (bias) = 250,500 parameters

Batch Normalization Layers:

The batch normalization layers each have 64 trainable parameters (32 for scaling and 32 for shifting for each layer).

Batch Norm1: 32 parameters for scaling and 32 for bias = 64

Batch Norm2: 32 parameters for scaling and 32 for bias = 64

Total Parameters:

Summing up:

Conv1: 320
Conv2: 9,248
fc1: 784,500
fc2: 250,500
BatchNorm1: 64
BatchNorm2: 64

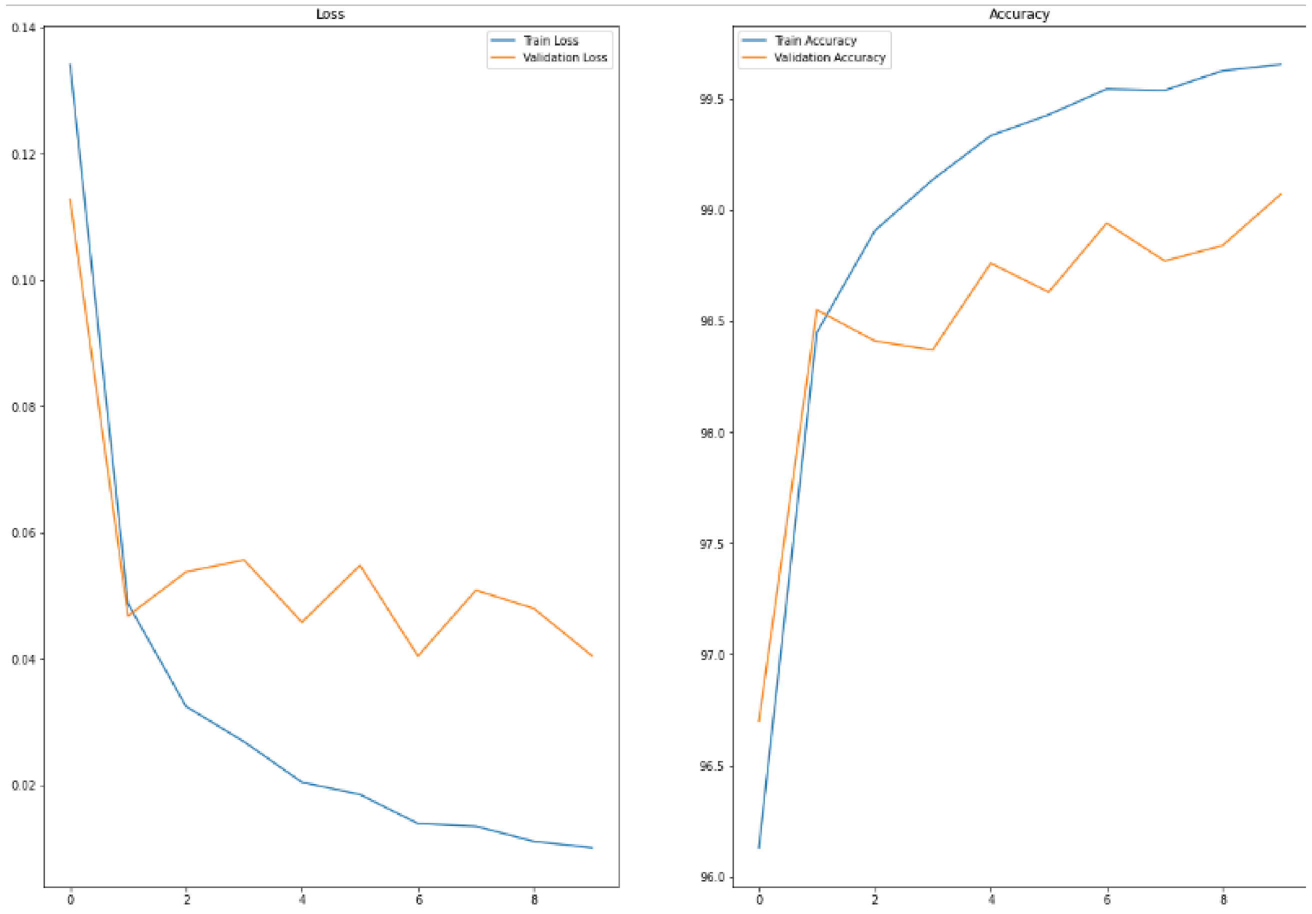
Total = 1,044,696 parameters

Parameters in fully connected layers: 784,500 (fc1) + 250,500 (fc2) = 1,035,000
Parameters in convolutional layers: 320 (conv1) + 9,248 (conv2) = 9,568

NUMBER OF NEURONS:
Conv1 Output: 32 channels * 28 * 28 = 25,088 neurons
Conv2 Output: 32 channels * 14 * 14 = 6,272 neurons
Total conv neurons = 31360
fc1: 500 neurons
fc2: 500 neurons
Total FC neurons = 1000
Total neurons = 32360

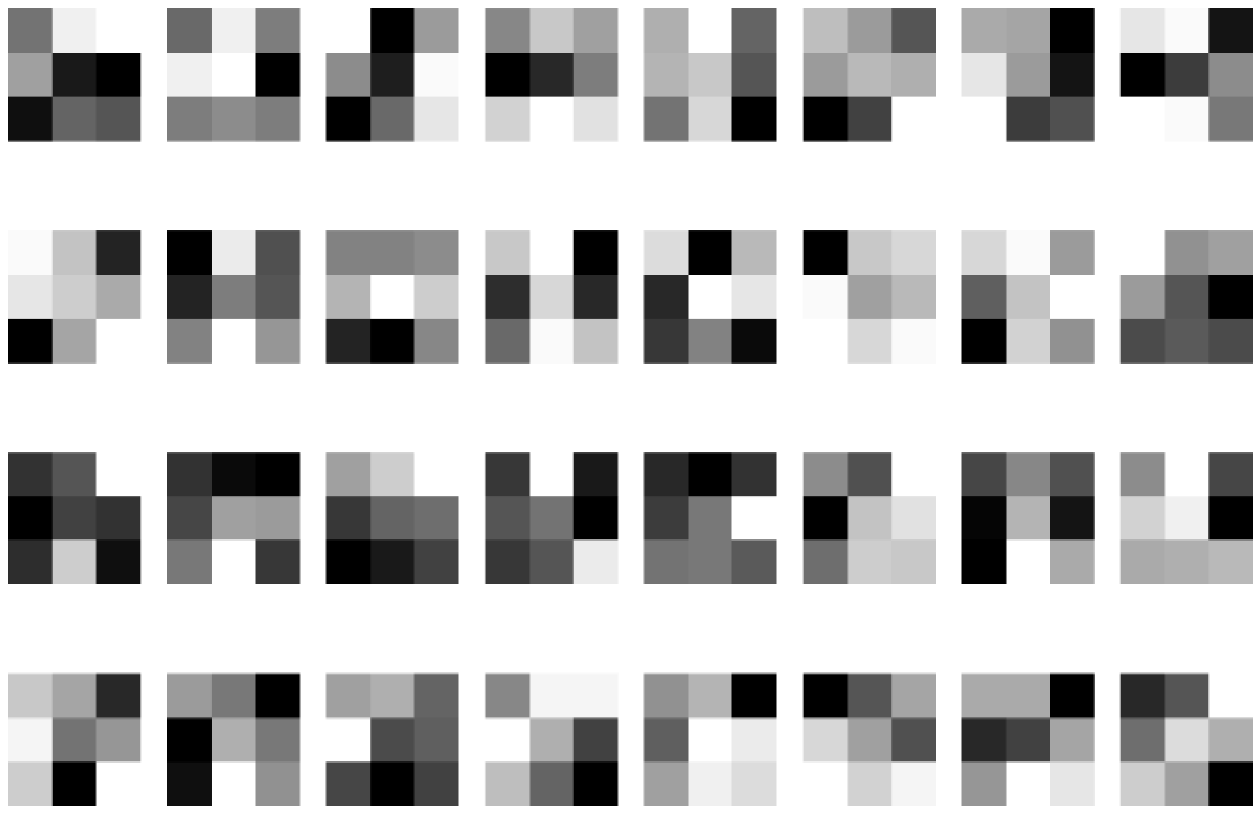
Comparison with BATCH NORMALISATION:

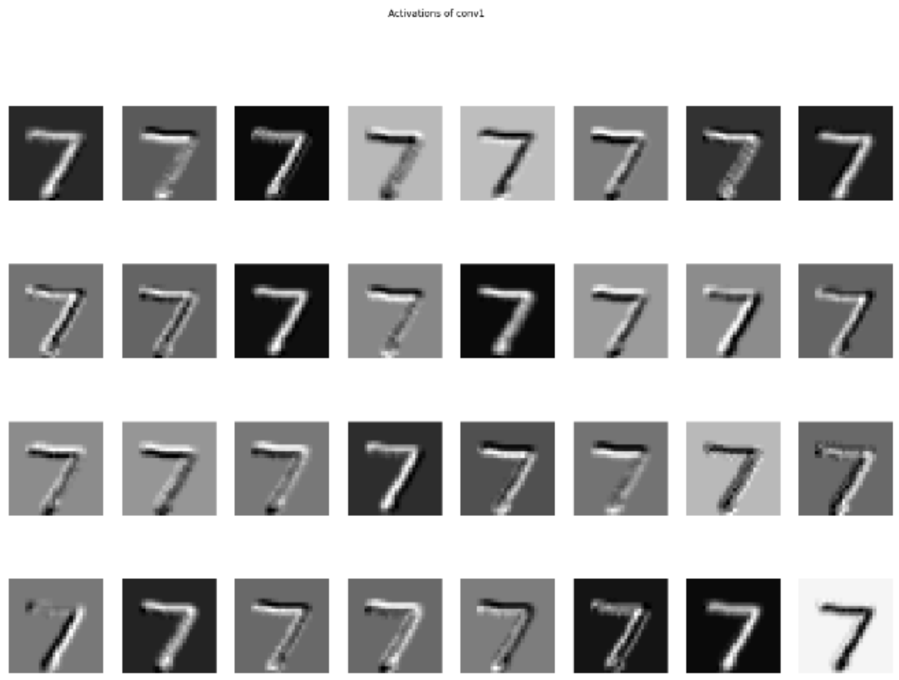
For 10 epochs : CNN without batch normalization took **5 minutes 51 seconds** to train. Whereas the CNN with Batch Normalisation took **6 minutes and 32 seconds** for the same number of epochs. In terms of the learning curve, from below it could be seen that Batch normalization achieves errors of less than 0.05 within 2 epochs, which in our original case without batch norm took 5 epochs. **Therefore, we can say that Batch Normalisation accelerated training.** But note that, from the curves, the degree of overfitting increases a lot with the number of epochs in batch normalization than when compared to the original case due to accelerated training. So, it is essential to stop within the few epochs to prevent Overfitting. **Within 2 epochs, batch normalization achieves accuracies > 98% in train and valid set whereas in without batch normalization case the accuracies were below 98% by 2 epochs of training. These statements could be understood from the plots.**



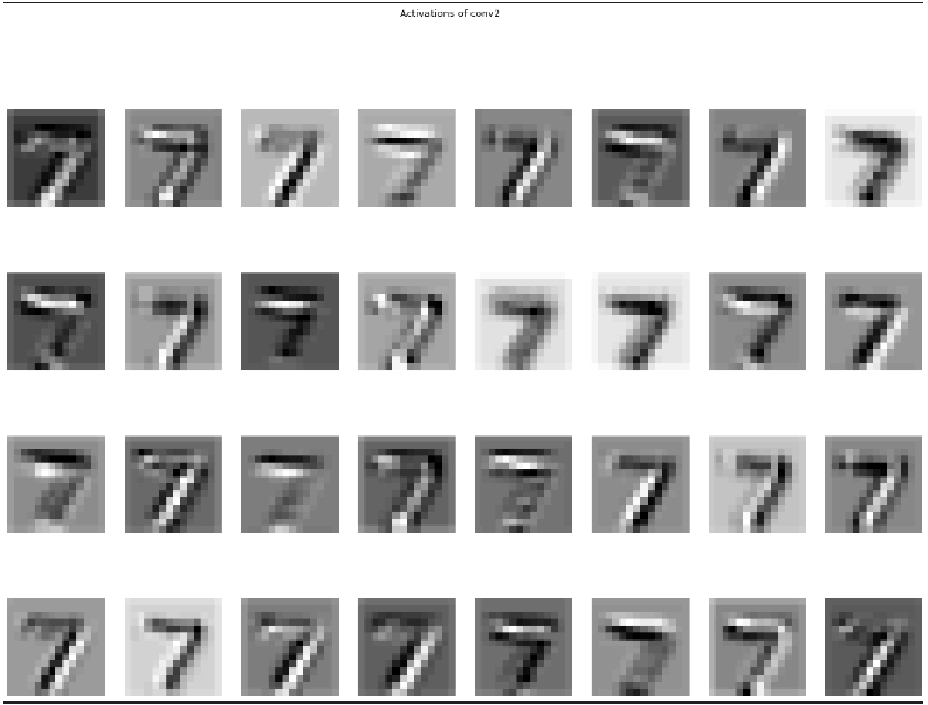
Part 2 Visualizing Convolutional Neural Network :

The above is the visual representation of the conv1 filters - 32.


The above is the visual representation of the conv2 filters - 32.



The above image shows the activation layer after conv1- output on a sample image.

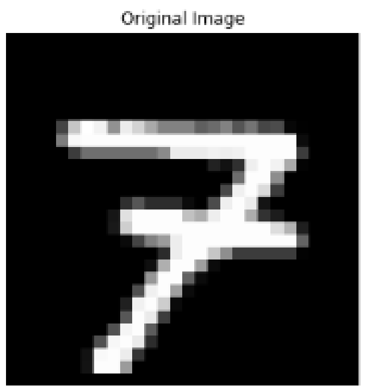


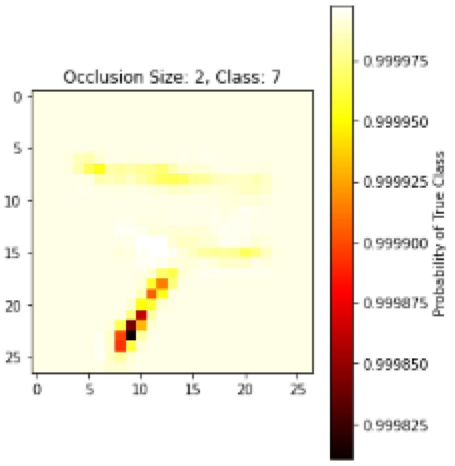
The above image shows the activation layer after conv2- output on a sample image.

From the above images on activations and the visualization of the convolution layers suggest that after the first activation - **the background seems more suppressed. and the edges and the foreground - especially edges are more prominent.** So this means the first convolution layer might be some sort of **high frequency feature capturing or an edge detection.**
The second activation layer shows prominent **BLURRING**, meaning that the **second convolution layer leads to some sort of a blurring filter**.
The outputs of the conv layer 1 might be some sort of edge detection filters, but they need not be the edge detection filters along x and y direction, rather they are the edge detection filters under other directions

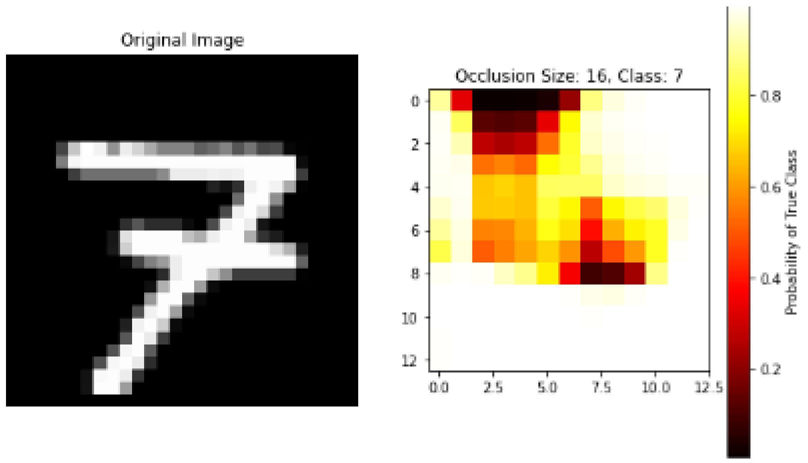
OCCLUDING PARTS OF IMAGE:

For this part, I plotted the heatmaps for the probability of true class predictions with the region of occlusion - and this occlusion was done with different sizes - 2,4,6,8,10,12,16 and for different images. All the images could be found in the output in the notebook, but here I am just attaching a few examples for commenting- detailed heatmaps for many images and different sizes of occlusion could be found in the notebook.
When the occlusion size is less, the probabilities are mostly higher.

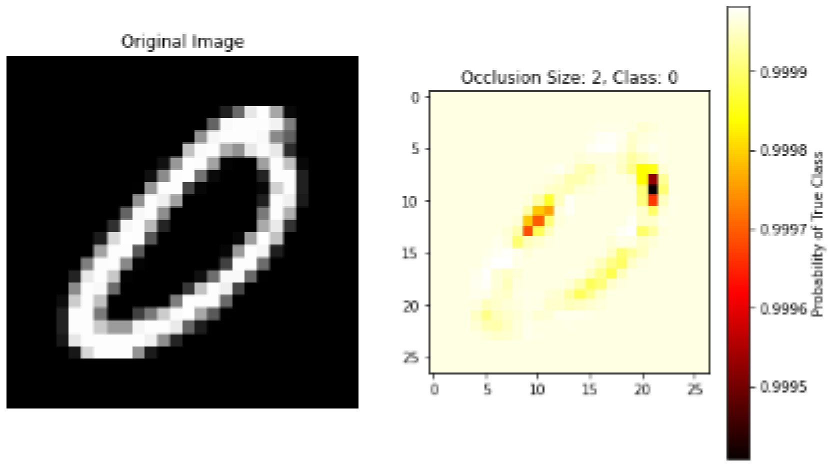


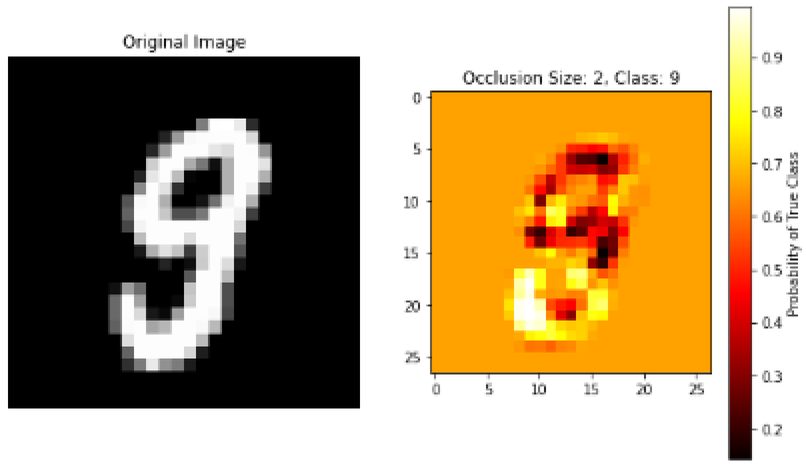


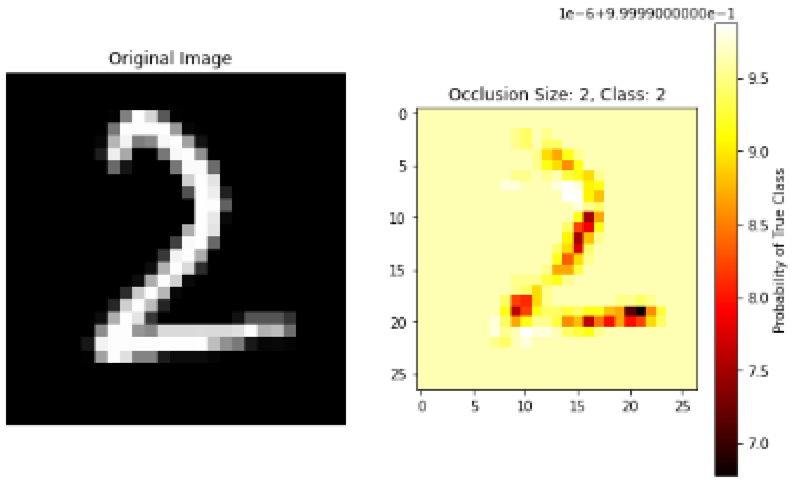
NOw i try to increase the size , you could find that the probabilities decreases by a lot:(occlusion size 16)



Few more examples to illustrate the model performance:
THis image below shows a proper heatmap - the edges of 0 is shown well in the heatmap , meaning that occlusion in these edges would reduce the probability of detecting that the image is 0 - this strengthens out trust on the model that the model actually sees the 0 in the image and only decides.





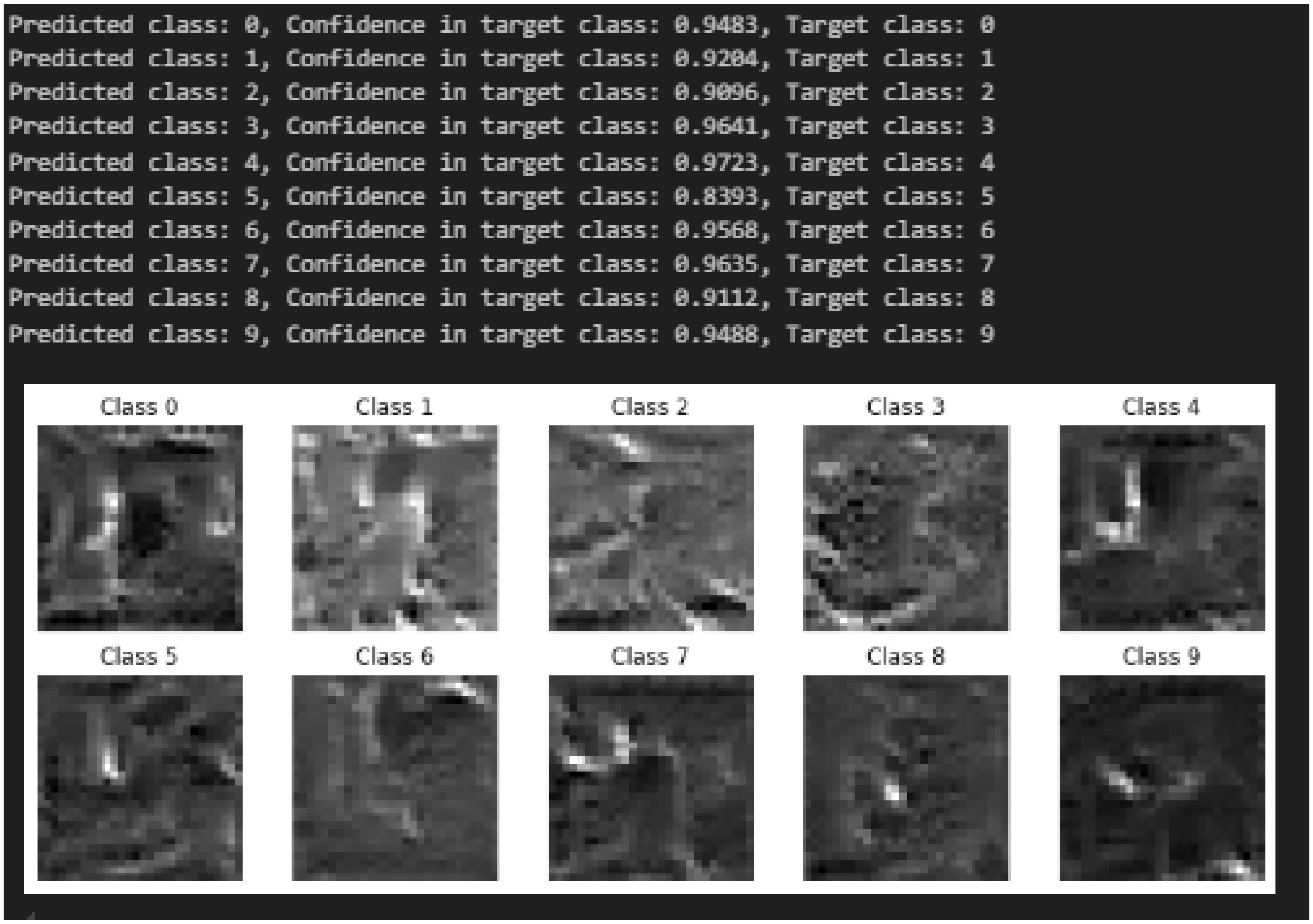


This guarantees that our model uses the features that we want the model to use to classify the image into numbers.

Part 3 Adversarial Examples

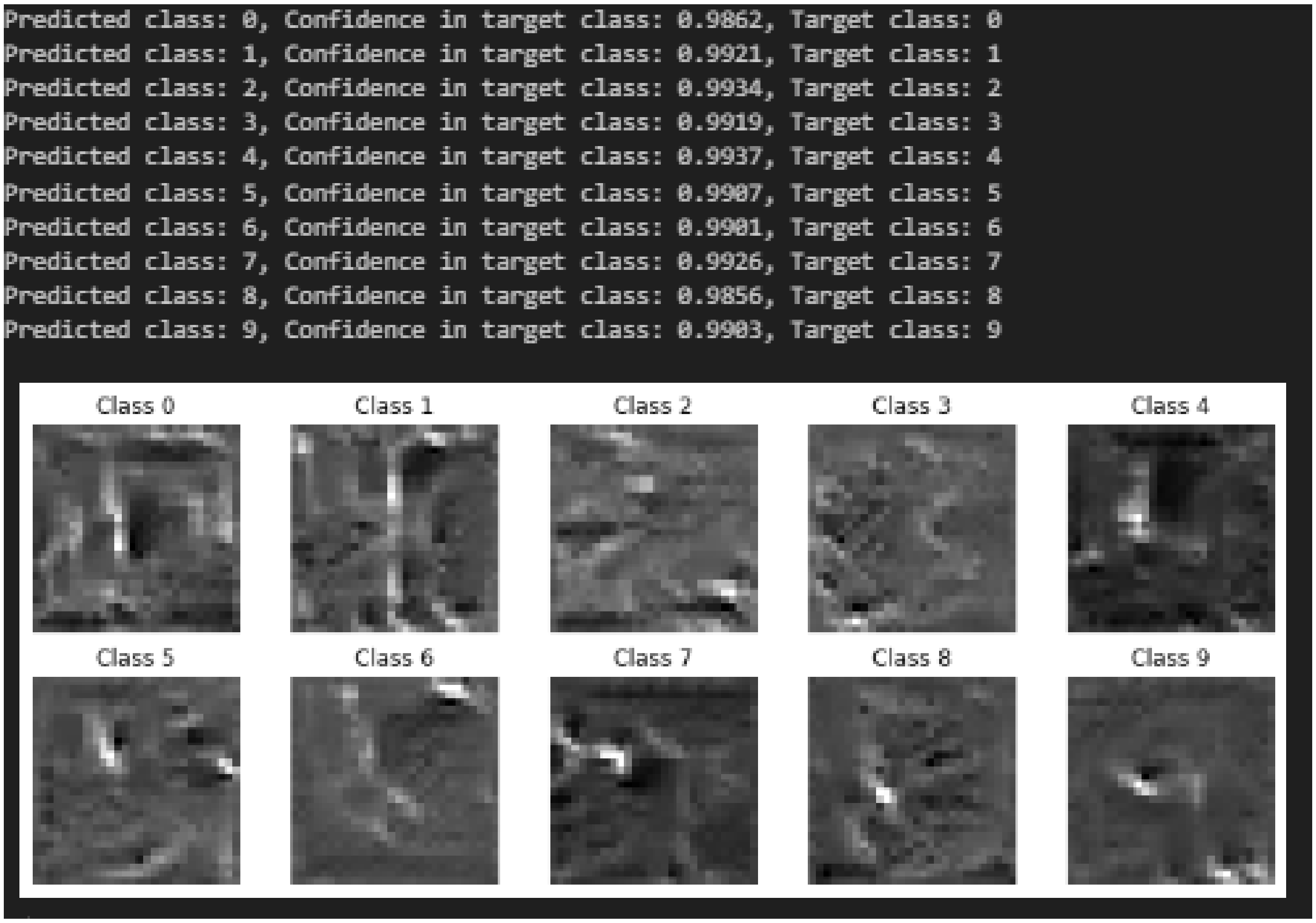
Non Targeted Attack:

Training for 100 steps and lr =0.01

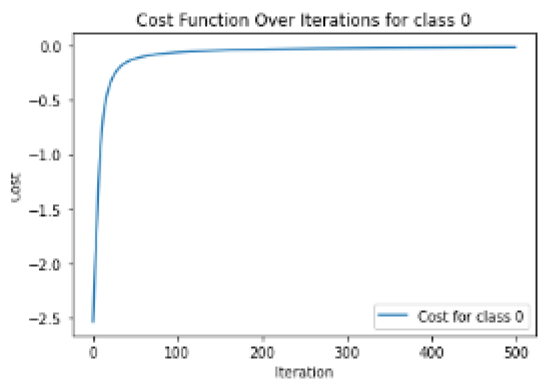


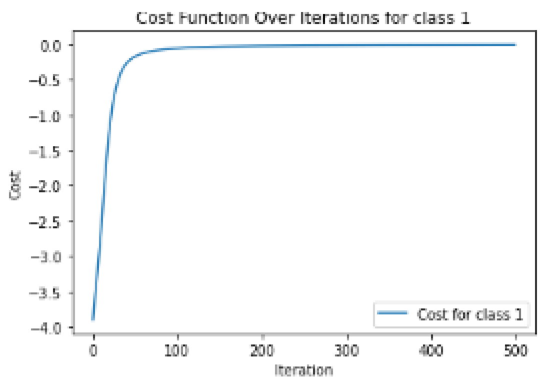
```
Predicted class: 0, Confidence in target class: 0.9483, Target class: 0
Predicted class: 1, Confidence in target class: 0.9204, Target class: 1
Predicted class: 2, Confidence in target class: 0.9096, Target class: 2
Predicted class: 3, Confidence in target class: 0.9641, Target class: 3
Predicted class: 4, Confidence in target class: 0.9723, Target class: 4
Predicted class: 5, Confidence in target class: 0.8393, Target class: 5
Predicted class: 6, Confidence in target class: 0.9568, Target class: 6
Predicted class: 7, Confidence in target class: 0.9635, Target class: 7
Predicted class: 8, Confidence in target class: 0.9112, Target class: 8
Predicted class: 9, Confidence in target class: 0.9488, Target class: 9
```

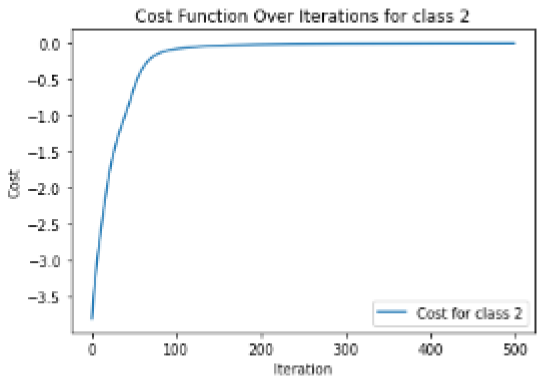
Training for 500 steps:

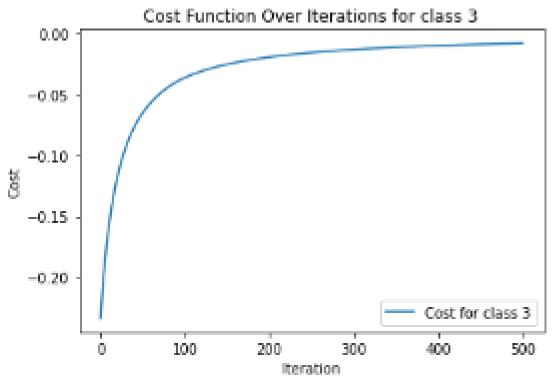


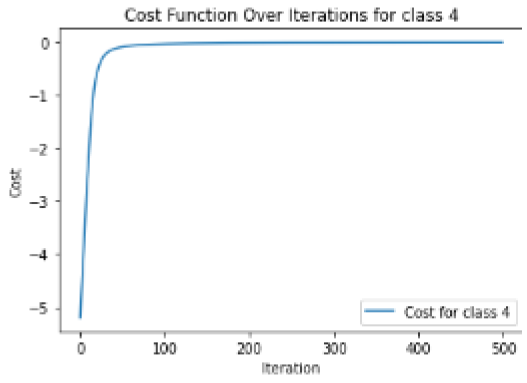
```
Predicted class: 0, Confidence in target class: 0.9862, Target class: 0
Predicted class: 1, Confidence in target class: 0.9921, Target class: 1
Predicted class: 2, Confidence in target class: 0.9934, Target class: 2
Predicted class: 3, Confidence in target class: 0.9919, Target class: 3
Predicted class: 4, Confidence in target class: 0.9937, Target class: 4
Predicted class: 5, Confidence in target class: 0.9907, Target class: 5
Predicted class: 6, Confidence in target class: 0.9901, Target class: 6
Predicted class: 7, Confidence in target class: 0.9926, Target class: 7
Predicted class: 8, Confidence in target class: 0.9856, Target class: 8
Predicted class: 9, Confidence in target class: 0.9903, Target class: 9
```

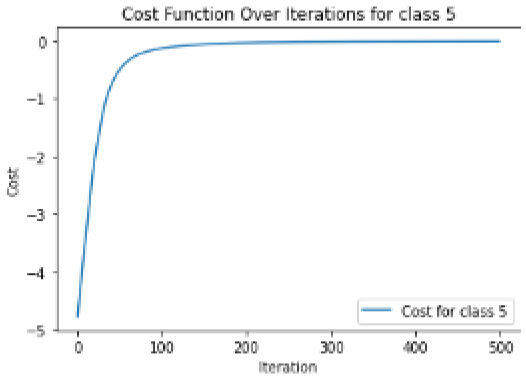

Cost Function Over Iterations for class 0
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
Cost
0
100
200
300
400
500
Iteration
Cost for class 0

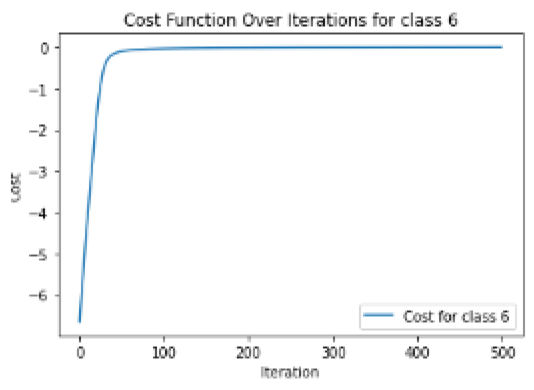

Cost Function Over Iterations for class 1
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
−4.0
Cost
0
100
200
300
400
500
Iteration
Cost for class 1

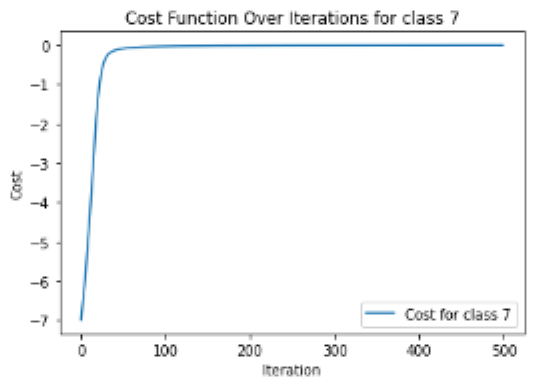

Cost Function Over Iterations for class 2
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
Cost
0
100
200
300
400
500
Iteration
Cost for class 2

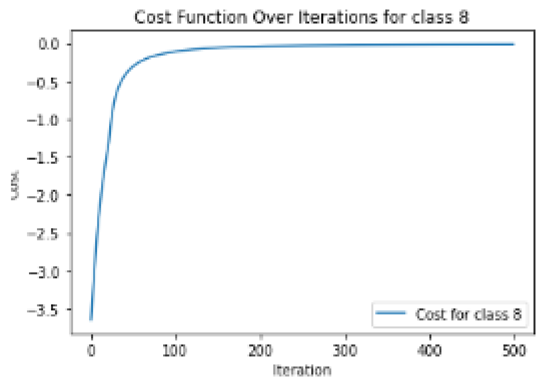

Cost Function Over Iterations for class 3
0.00
−0.05
−0.10
−0.15
−0.20
Cost
0
100
200
300
400
500
Iteration
Cost for class 3

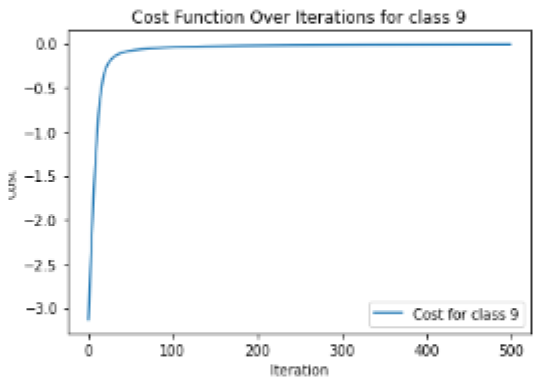

Cost Function Over Iterations for class 4
0
−1
−2
−3
−4
−5
Cost
0
100
200
300
400
500
Iteration
Cost for class 4


Cost Function Over Iterations for class 5
0
−1
−2
−3
−4
−5
Cost
0
100
200
300
400
500
Iteration
Cost for class 5







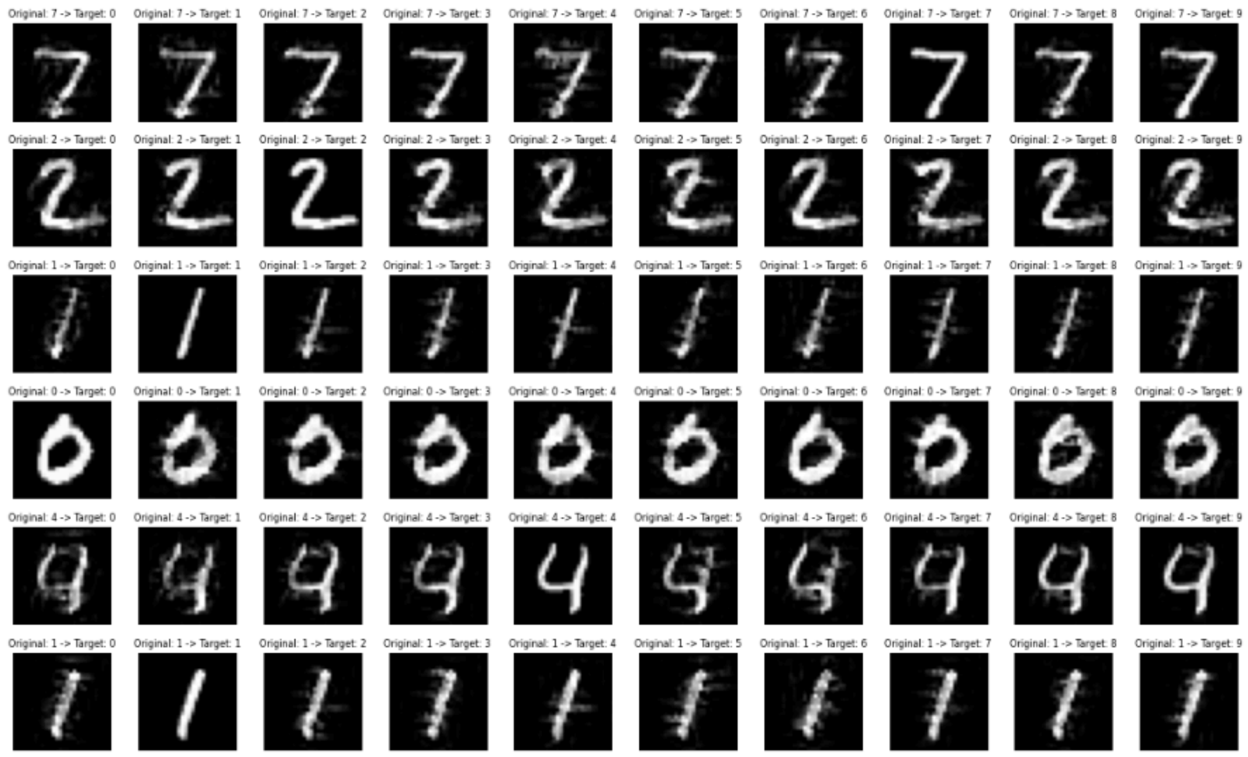


We see that, as we increase the number of steps to 200 and then to 500, the confidence values of predicting the classes also increase. Now they are as high as 0.98. The generated images dont look exactly like the numbers, but we can see that there are very small features that look like derived similar to the numbers, FOR example, the two ovals in the generated image for 8, or the oval in 9(not visible in the doc much due to loss of quality of image while pasting in google docs - but it can be zoomed and checked in the notebook attached). They don't look similar to the generated images, because, all the algorithm tries to do is to **increase the logit values to increase the probabilities of the target class my modifying the image.** The algorithm, therefore tries to modify the input images so that they have all the "**features"** of the target class that the model has **"learnt"** during its training. The objective function set, has not much control on the "**similarity**" of the generated image to the actual image. So, the image generated, may have a lot of "reconstructed" loss as it does not try to reconstruct the target class image, but it tries to maximize the similarity feature representation of the image with the target class at a higher dimension level - inside the neural network.

Because we do a gradient ascent, we see the cost function increases and is asymptotic to x axis

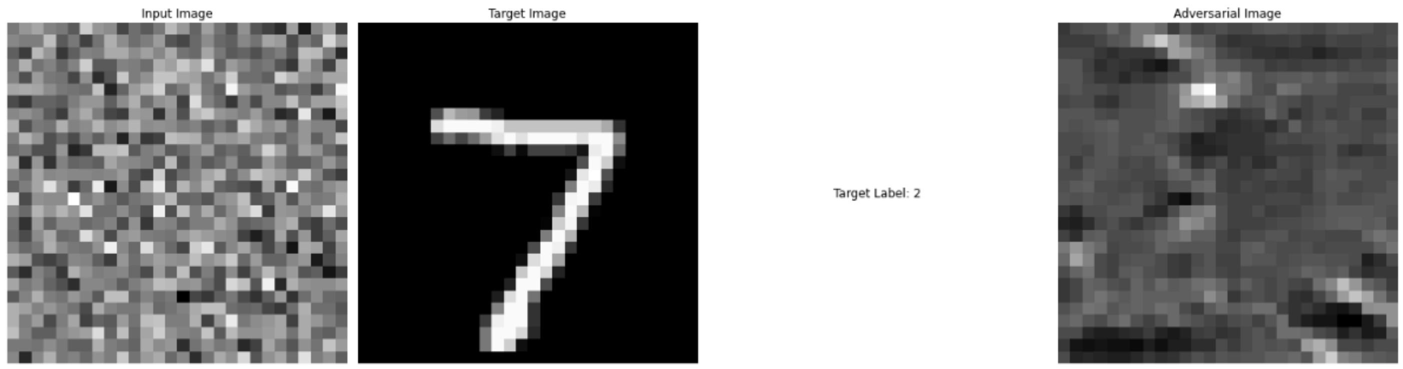
Targeted Attack:
Since the question paper has no information about what sort of input images should be passed onto a targetted attack, I did the following two cases below
1) Case 1: Input image and target image is same: and the target label is different, Ie the input image and target image are both the same from the MNIST dataset
2) Case 2: Input image is the gaussian noisy image - mean 128 and std dev 1. The target image is taken from the MNIST dataset and the target label is the label different from the target image

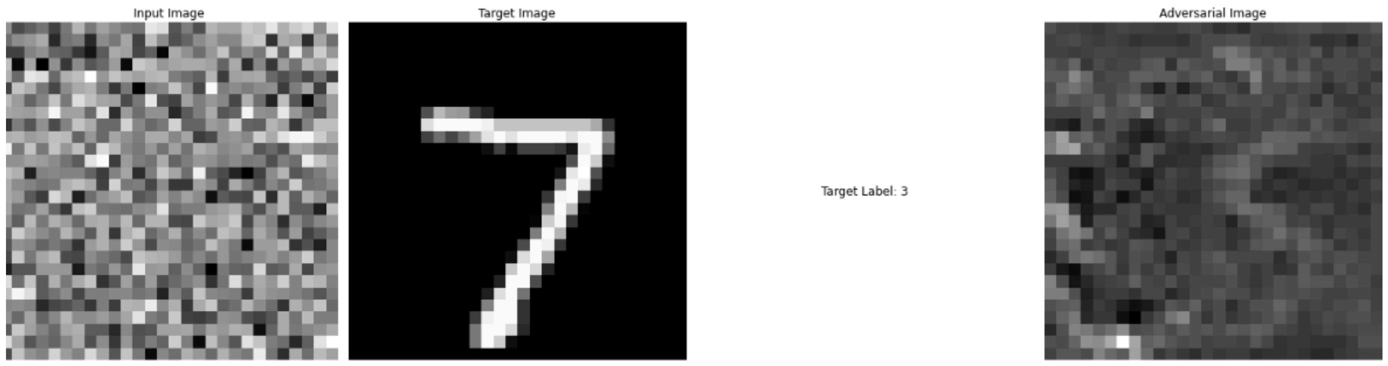
CASE 1:



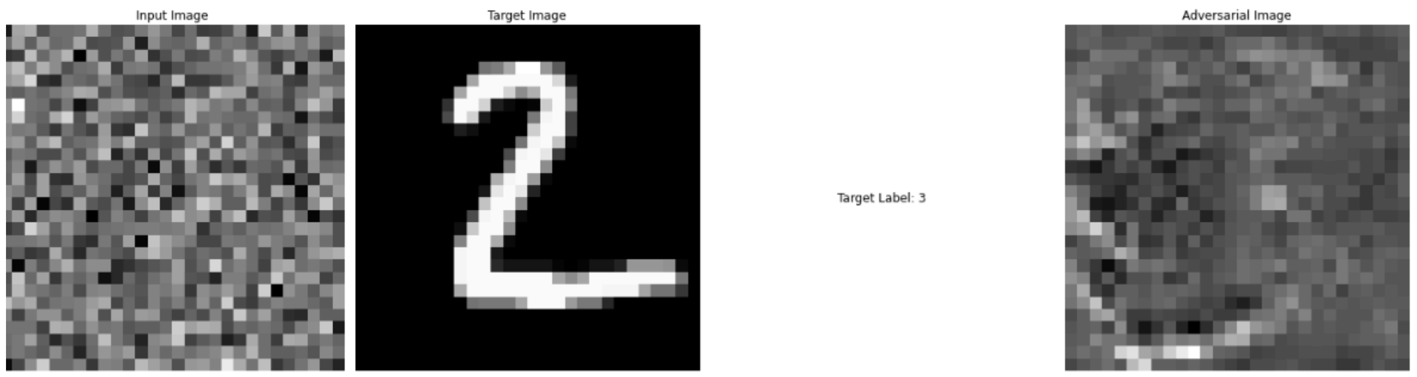
Here since I started the input image as the
CASE 2:

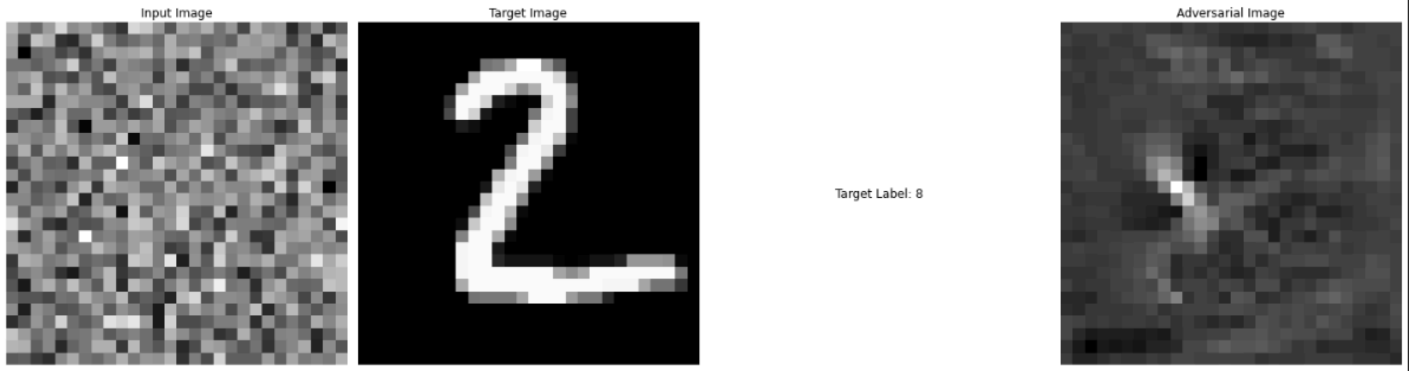
I am attaching a few examples, but all the images are found in the notebook.

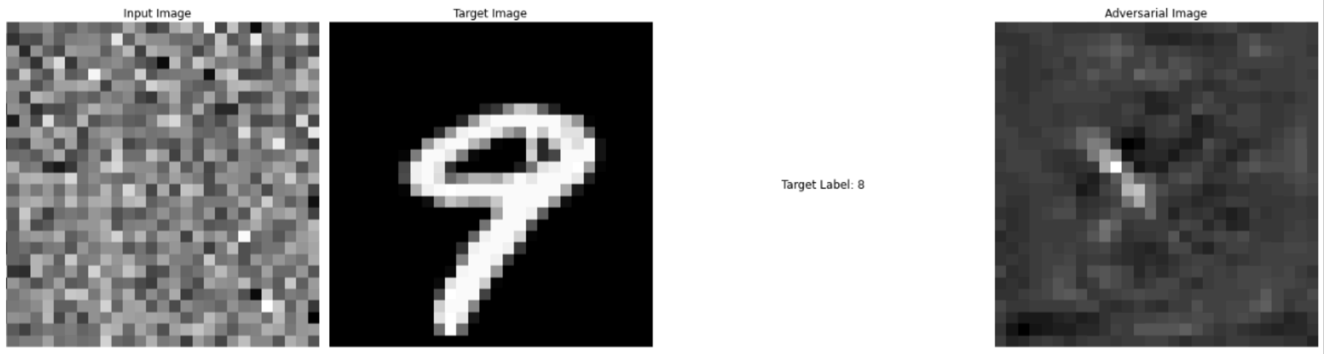




From the above image we could see that the generated image has some features of 3 in it - the edges of 3 among the noises. So these images sort of resemble the targeted class.







SO what I observe is that the final image has edges that resemble the edges of the target labels.