Assignment 2 - CNN

Subhankar Chakraborty

September 24, 2019

Introduction

This assignment was based on using a CNN to classify images on the MNIST dataset. The assignment is coded in **pytorch** in a python 3.6 environment. The libraries required for running this assignment are -

- numpy
- sys
- os
- matplotlib
- torch
- torchvision

The Architecture

The layers in this CNN architecture are

- A convolution layer with 32 3x3 filters with stride 1 and padding 1.
- A 2x2 maxpool layer with stride 2.
- A convolution layer with 32 3x3 filters with stride 1 and padding 1/
- A 2x2 maxpool layer with stride 2
- A fully connected layer with 500 outputs.
- A fully connected layer with 10 outputs.

NOTE: For the final layer, I am using a log(softmax) classifier instead of the traditional softmax and so for my loss function, I am using the NLL loss. To get the softmax probabilities for any part of this assignment, I am taking the values before the log(softmax) classifier and running a softmax on it.

Training the Network

A few important points about the training procedure are mentioned below.

- The optimizer used is batch gradient descent with batch size of 128
- The learning rate used it 0.08
- The training has been run for 8 epochs.
- The loss function used is negative log likelihood (**NLL**) loss.

Plots

The error plot for the training and validation losses is shown below.

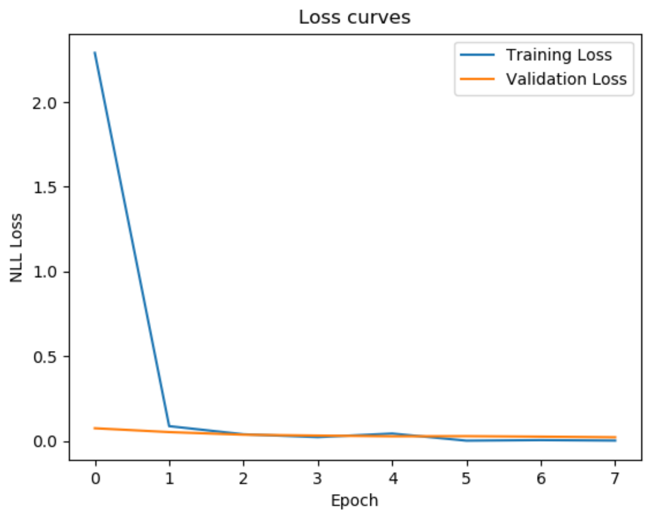


Figure 1: Training and validation loss.

The accuracy plot at the end of each epoch is shown in figure 2.

Note that as the accuracy plot starts from the end of the first epoch, the plot shows values which are already very close to 1.
The accuracy achieved on the test set was on an average is **99.070**.

Some random samples with the predicted labels are shown in figures 3,4 and 5.

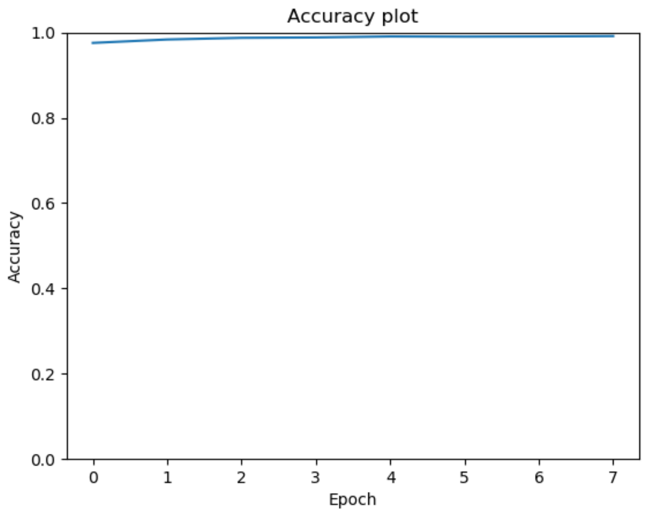


Figure 2: Test accuracy

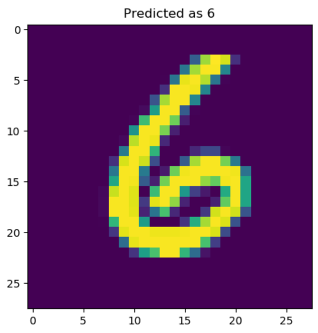


Figure 3: Random sample of 6

More on the architecture

Dimensions

The input and output dimensions at each stage are.

- First convolution layer (for each filter individually)
  - Input size = 28X28
  - Output size = 28x28
- First maxpool layer
  - Input size = 28X28

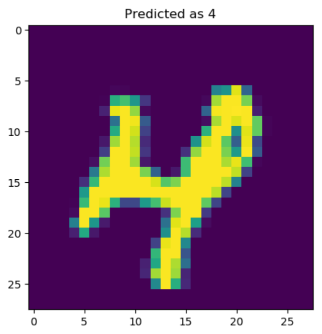


Figure 4: Random sample of 4

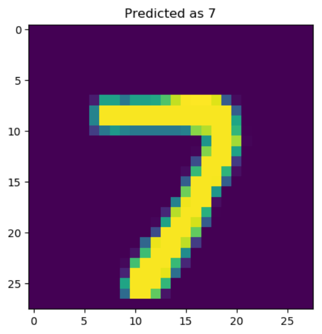


Figure 5: Random sample of 7

  - Output size = 14x14

- Second convolution layer (for each filter individually)
  - Input size = 32x14x14 (14x14 for each of the 32 filters in layer 1)
  - Output size = 14x14
- Second maxpool layer
  - Input size = 14x14
  - Output size = 7x7
- First fully connected layer
  - Input size = 32x7x7 (7x7 for each of the 32 filters previously)
  - Output size = 500
- Second fully connected layer

  - Input size = 500
  - Output size = 10

Parameters

The number of paramaters are

- First Convolution layer
  - Weights = 3x3x32 = 288
  - Biases = 32
  - total = 320
- Second Convolution layer
  - Weights = 32x3x3x32 = 9216
  - Biases = 32
  - total = 9248
- First fully connected layer
  - Weights = 7*7*32*500 = 784000
  - Biases = 500
  - total = 784500
- Second fully connected layer
  - Weights = 500*10 = 5000
  - Biases = 10
  - total = 5010

So, the total parameters are **799078** out of which **789510** are in the fully connected layers and **9568** are in the convolution layers.

Neurons

The number of neurons in each of the layers are

- First convolution layer - 3x3x32 = 288
- Second convolution layer - 3x3x32x32 = 9216
- First fully connected layer - 500
- Output layer - 10

The total number of neurons is **9564** with **510** of them being in the fully connected layers and **9054** of them being in the convolution layers.

Batch Normalization

The effects of batch normalization were

- Training accuracy - Nothing conclusive. Sometimes I got slightly higher accuracy, other times i got slightly lower accuracy.
- Training time - Same is the case with the time to train the model.

In the final model, batch normalization has not been implemented.

1 Visualizing the Model

First layer filters

The first layer filters have been shows in figure 6.

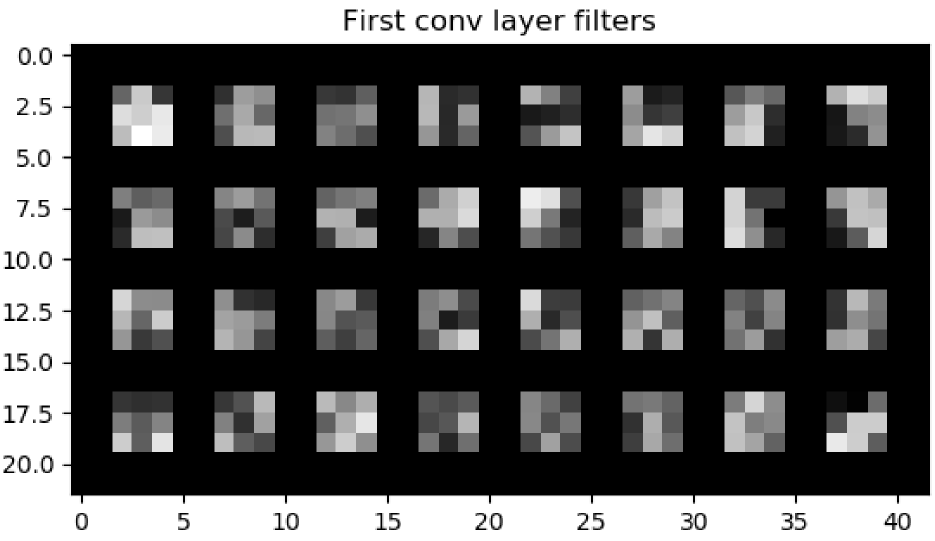


Figure 6: First convolution layer filters

Although it is difficult to claim anything in 3x3 filters, we can make a few higher level observations.

- The filters with one black patch in the centre and mostly grey patches around it are probably smoothing the images.
- The filters with bright white on the bottom row and mostly black on the top are probably looking at horizontal edges.
- The filters with bright white on the left column and mostly black on the right are probably looking at vertical edges.

Second convolution layer filters

We plot the second convolution layer filters for 3 of the 32 layers.

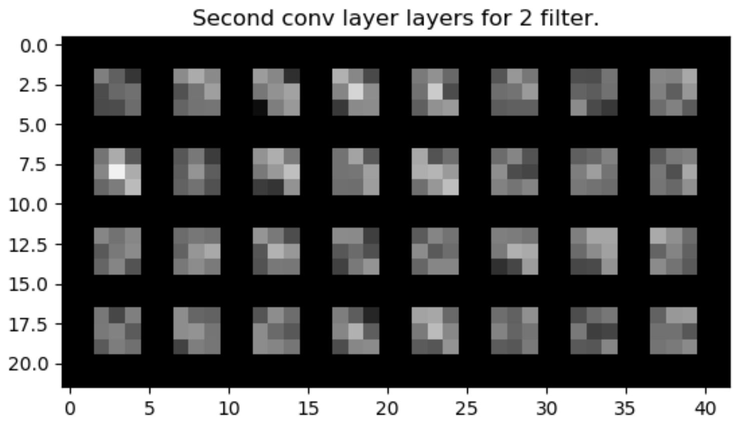


Figure 7: Second convolution layer filters

From a higher level, it is really hard to say what they filters are doing, especially in a kernel size of only 3.

First Convolution layer activations

The first convolution layer activations for a few digits are plotted in figures 10, 11 and 12.

Some observations are

- White is the highest value, black is the most negative and grey is the average.

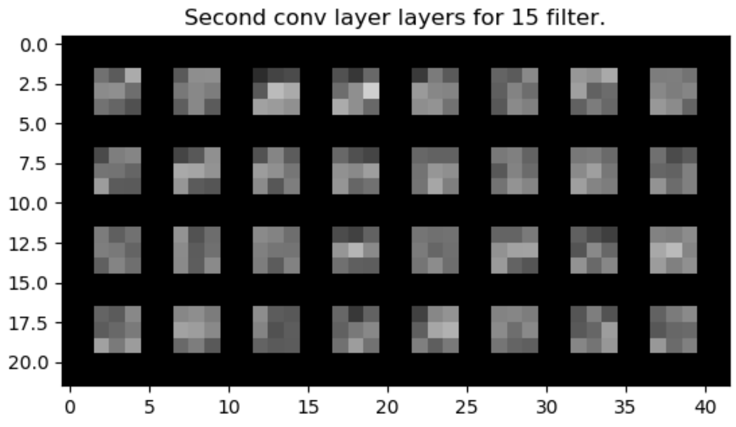


Figure 8: Second convolution layer filters

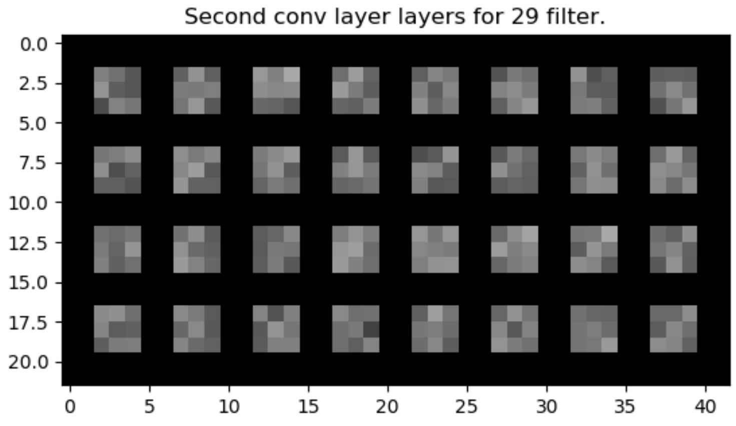


Figure 9: Second convolution layer filters

- We can see sharp whites and blacks near some of the edges, meaning some of the activations represent edge related features of the image.

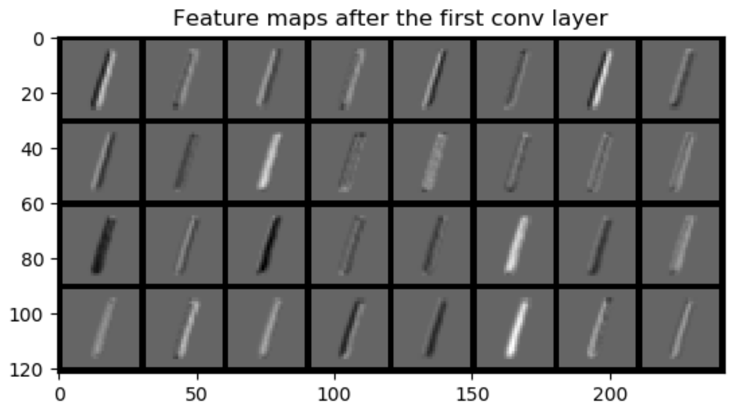


Figure 10: First layer activations for 1

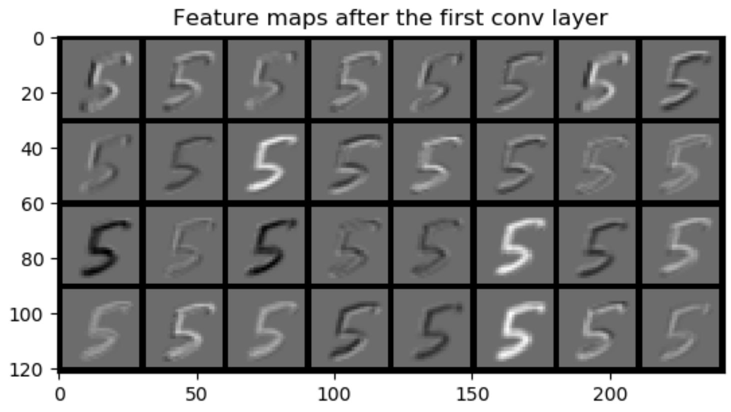


Figure 11: First layer activations for 5

Second convolution layer activations

The second convolution layer activations for some for some of the digits are shown in figures 13, 14 and 15.

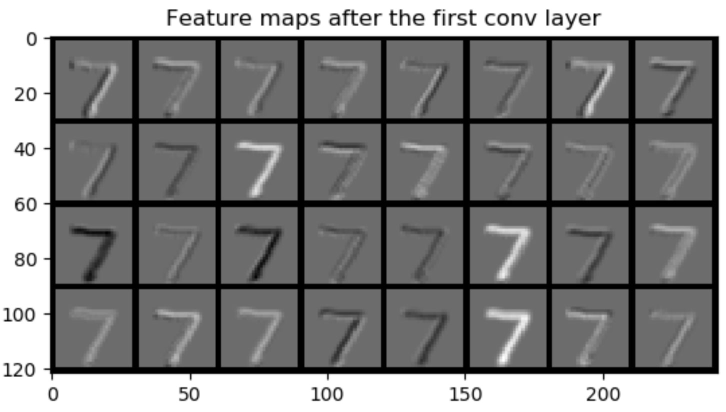


Figure 12: First layer activations for 7

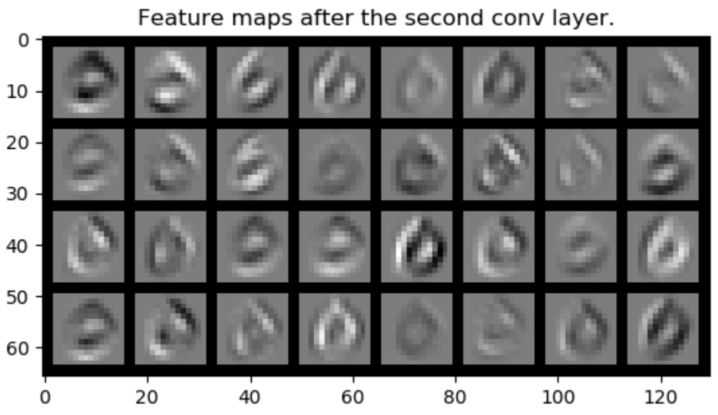


Figure 13: Second layer activations for 2

Some observations are

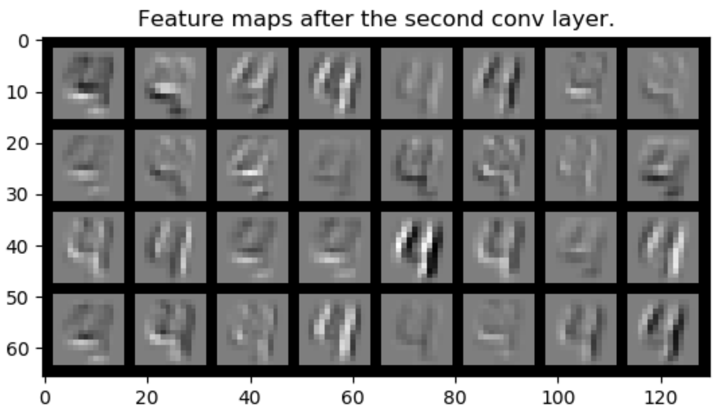


Figure 14: Second layer activations for 4

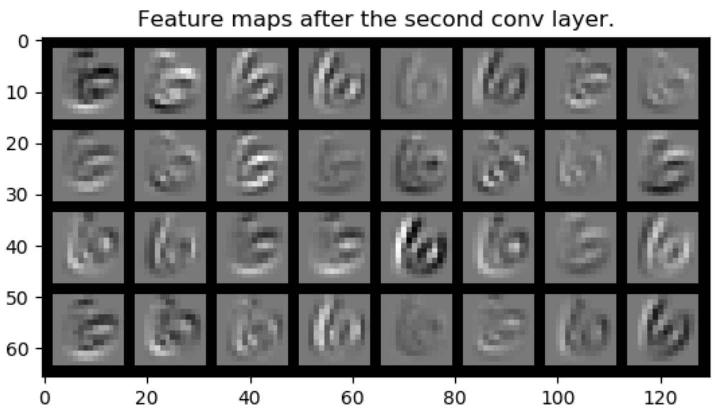


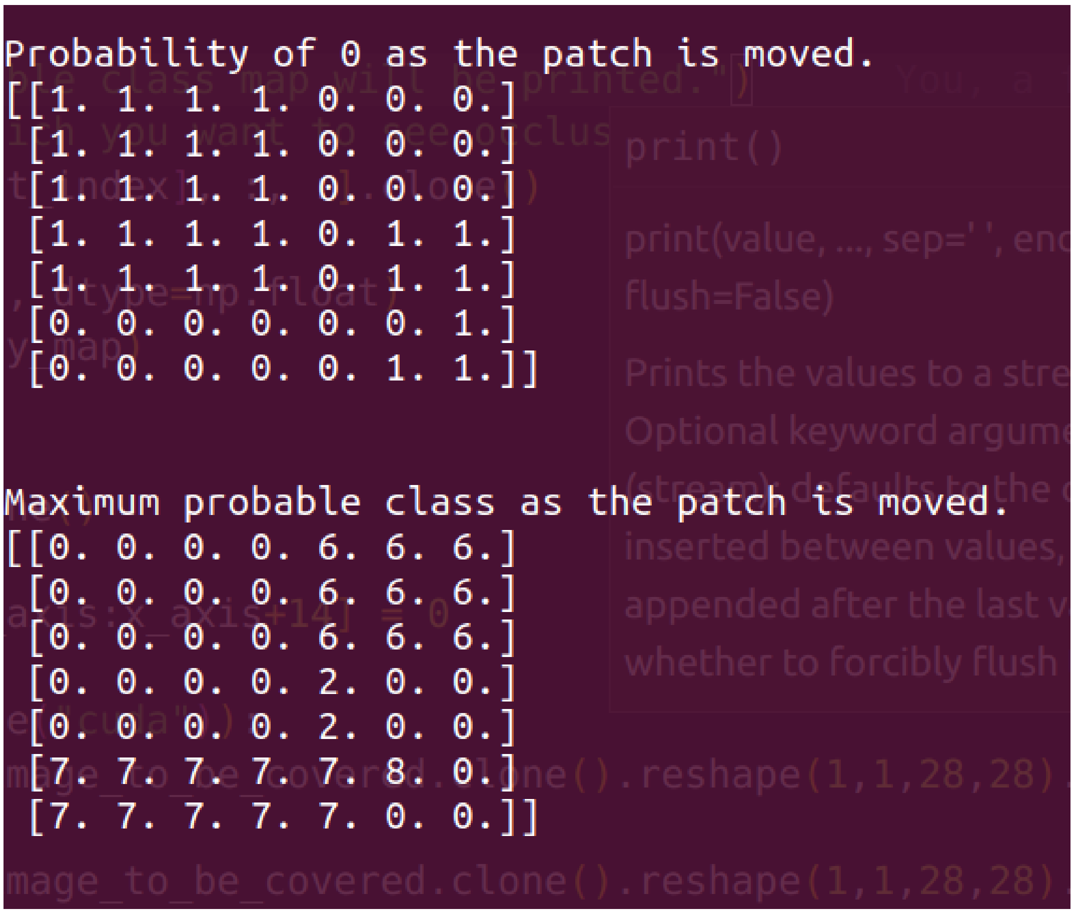
Figure 15: Second layer activations for 6

- Unlike the first layer, a lot of the second layer activations are grey blobs.

- The effects of the activation are on the whole of the image and not concentrated to particular edges like the first layer activations.
- It is hard to tell conclusively what exactly do the activations mean.

Occluding parts of the image

Instead of generating plots, what I thought will be more meaningful is varying a patch of size 14x14 across the image with stride 2 and entering the corresponding class probability and the max probable class in a matrix. A few plots with occluded parts of images are also shown. The [j,k] element of a matrix is the result when the patch [2j:2j+14, 2k:2k+14] part of the image is zeroed out. Results for the classes are presented below.



```
Probability of 0 as the patch is moved.
[[1. 1. 1. 1. 0. 0. 0.]
 [1. 1. 1. 1. 0. 0. 0.]
 [1. 1. 1. 1. 0. 0. 0.]
 [1. 1. 1. 1. 0. 1. 1.]
 [1. 1. 1. 1. 0. 1. 1.]
 [0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 1. 1.]]


Maximum probable class as the patch is moved.
[[0. 0. 0. 0. 6. 6. 6.]
 [0. 0. 0. 0. 6. 6. 6.]
 [0. 0. 0. 0. 6. 6. 6.]
 [0. 0. 0. 0. 2. 0. 0.]
 [0. 0. 0. 0. 2. 0. 0.]
 [7. 7. 7. 7. 7. 8. 0.]
 [7. 7. 7. 7. 7. 0. 0.]]
```

Figure 16: Effects of occlusion on 0

Some observations are

- As the patch is at the very edge, the classifier recognizes the image with certainty.

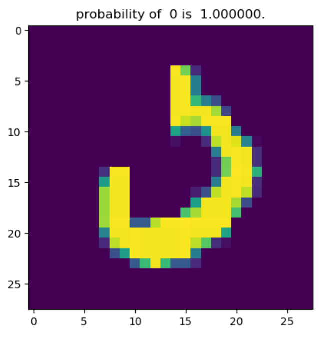


Figure 17: Effects of occlusion on 0

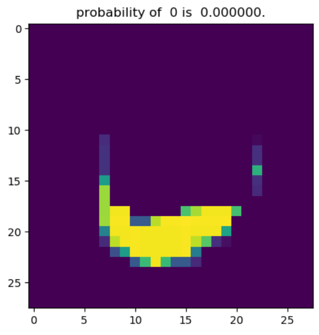


Figure 18: Effects of occlusion on 0

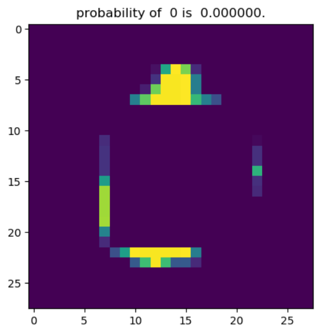


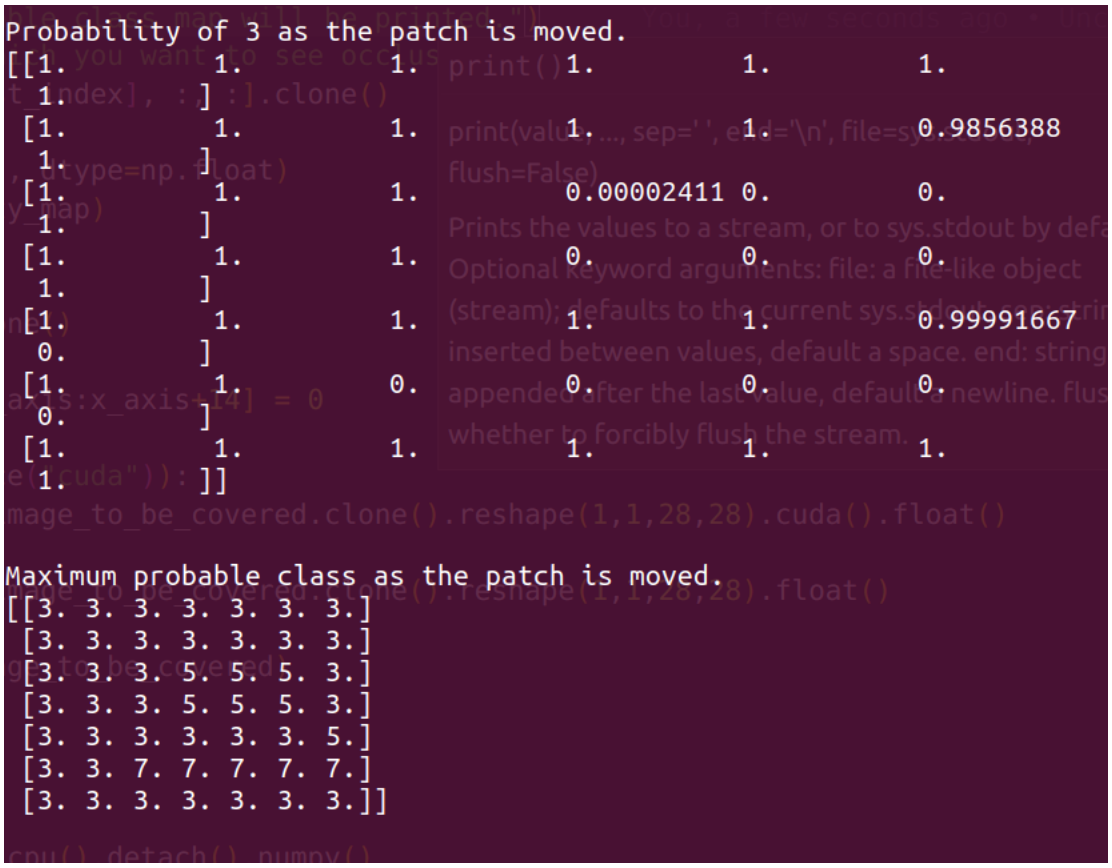
Figure 19: Effects of occlusion on 0

- As the patch is moved more centrally or begins to occlude the top half of zero, the classifier is almost certain the image is not of zero.

From the above two observations we can say that the classifier is actually looking at the digit only and not some random pixels at the side.

Similar matrices for a few other digits are plotted in figures 20 and 21. A much better way to visualize the effects real time would be to run the part of the code corresponding to it. Note that the code plots patches on stride 4 only and hence, not all entries might be visible. That can be changed by modifying a small if condition in the code.

```
if ((x_axis%4==0)&(y_axis%4==0)):
    plt.imshow(temp_image_to_be_covered.cpu().numpy().reshape(28,28))
    plt.title("probability of  {} is  {:.6f}.".format(
        test_index, probability[0]))
    plt.show()
```



```
Probability of 3 as the patch is moved.
[[1.          1.          1.          1.          1.          1.
  1.        ]
 [1.          1.          1.          1.          1.          0.9856388
  1.        ]
 [1.          1.          1.          0.00002411 0.          0.
  1.        ]
 [1.          1.          1.          0.          0.          0.
  1.        ]
 [1.          1.          1.          1.          1.          0.99991667
  0.        ]
 [1.          1.          0.          0.          0.          0.
  0.        ]
 [1.          1.          1.          1.          1.          1.
  1.        ]]

Maximum probable class as the patch is moved.
[[3. 3. 3. 3. 3. 3. 3.]
 [3. 3. 3. 3. 3. 3. 3.]
 [3. 3. 3. 5. 5. 5. 3.]
 [3. 3. 3. 5. 5. 5. 3.]
 [3. 3. 3. 3. 3. 3. 5.]
 [3. 3. 7. 7. 7. 7. 7.]
 [3. 3. 3. 3. 3. 3. 3.]]
```

Figure 20: Effects of occlusion on 3

Some generic observations are

- None of the images are affected at all when the pixels with no information are covered.

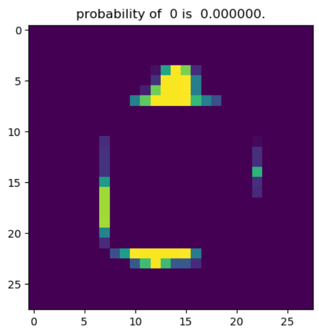
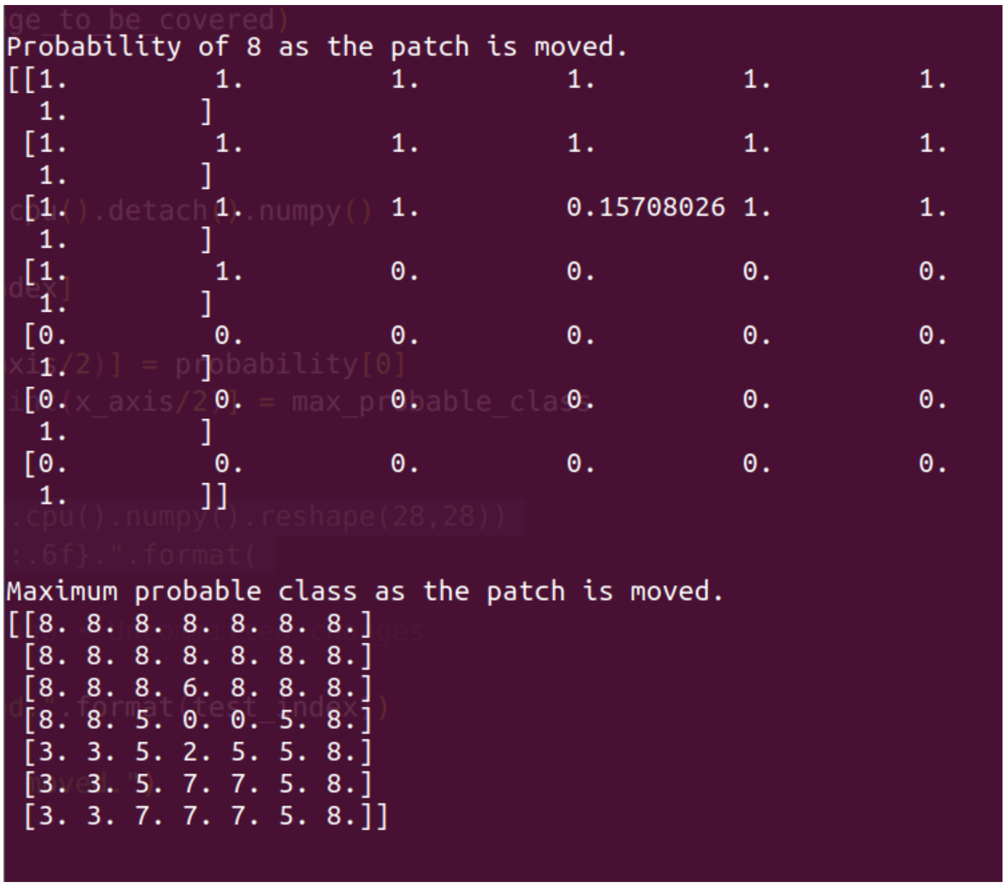


Figure 21: Effects of occlusion on 3



```
Probability of 8 as the patch is moved.
[[1.          1.          1.          1.          1.          1.
  1.        ]
 [1.          1.          1.          1.          1.          1.
  1.        ]
 [1.          1.          1.          0.15708026 1.          1.
  1.        ]
 [1.          1.          0.          0.          0.          0.
  1.        ]
 [0.          0.          0.          0.          0.          0.
  1.        ]
 [0.          0.          0.          0.          0.          0.
  1.        ]
 [0.          0.          0.          0.          0.          0.
  1.        ]]

Maximum probable class as the patch is moved.
[[8. 8. 8. 8. 8. 8. 8.]
 [8. 8. 8. 8. 8. 8. 8.]
 [8. 8. 8. 6. 8. 8. 8.]
 [8. 8. 5. 0. 0. 5. 8.]
 [3. 3. 5. 2. 5. 5. 8.]
 [3. 3. 5. 7. 7. 5. 8.]
 [3. 3. 7. 7. 7. 5. 8.]]
```

Figure 22: Effects of occlusion on 8

- Even when parts of the image is occluded, the classifier is robust to most of such cases.

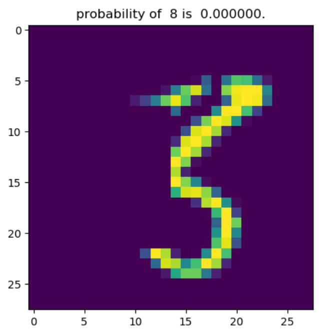


Figure 23: Effects of occlusion on 8

- When the central patches of the image are covered, the classifier goes haywire and gives random results.

2 Adversarial Examples

2.1 Non targeted Attack

Some observations

- Step size used is 0.1
- 15000 iterations are being run for each class.

The training plot for one such case is shown in figure 24.

The adversarial images for the 10 classes are plotted from figures 25 to 34.

The observations are

- The cost function **is always increasing**.
- The network always predicts the adversarial examples with high confidence. In fact, the probability value saturates at 1. after around 1000 steps while the image is being updated for 15000 iterations to get an image more perceptible to the original digit as seen by a human.
- None of the generated images looks like an exact digit. This is because how we look at a digit and recognize it and how a machine looks at it are quite different I guess. Some crude high level observations can be made.
  - In the generated image for 1, we can see a prominent vertical bar. This is probably what the classifier is looking for to classify a digit as 1.

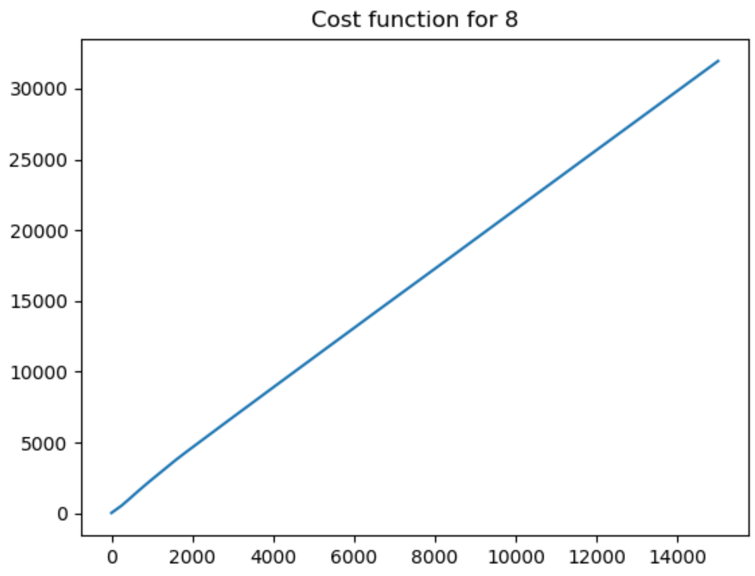


Figure 24: Adversarial cost for digit 8.

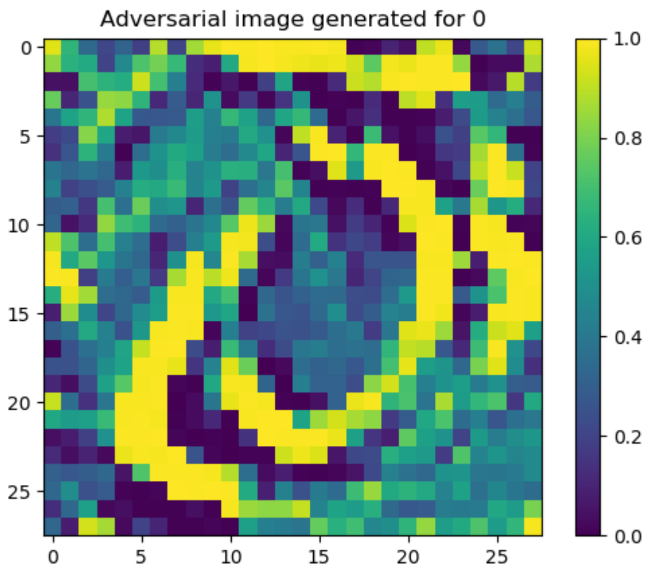


Figure 25: Adversarial image for 0.

– In the image for 3, we can partially see the two lobes. These are probably important cues for the classifier to recognize a digit as

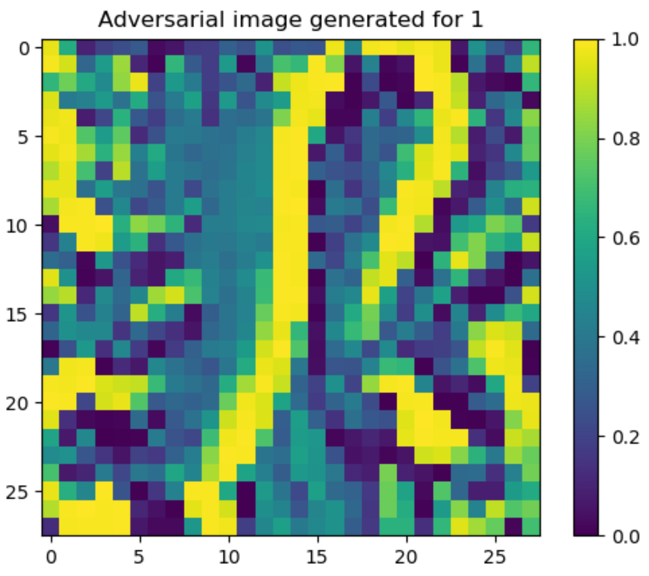


Figure 26: Adversarial image for 1.

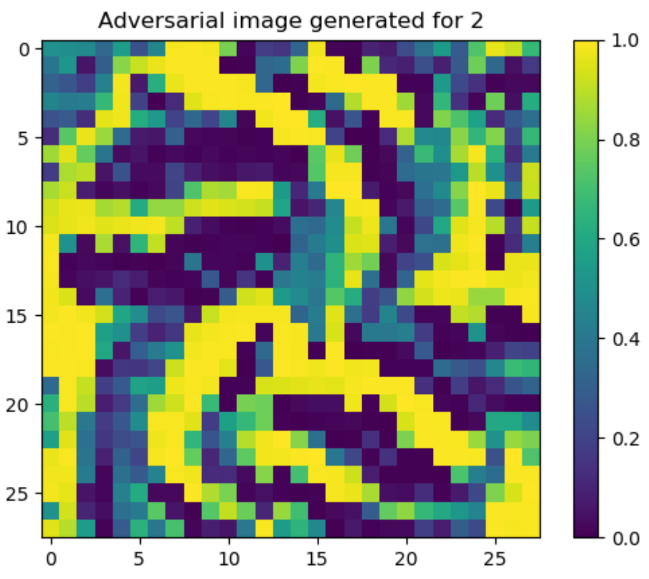


Figure 27: Adversarial image for 2.

a 3.

– Similar observations for 7 show the presence of a long horizontal

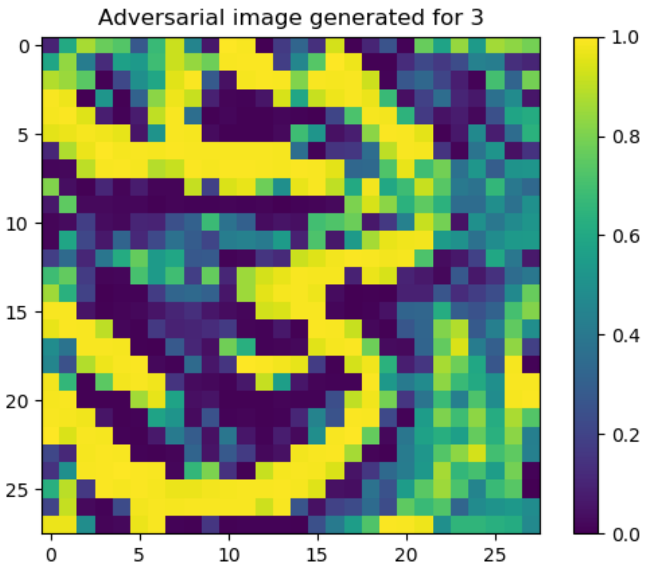


Figure 28: Adversarial image for 3.

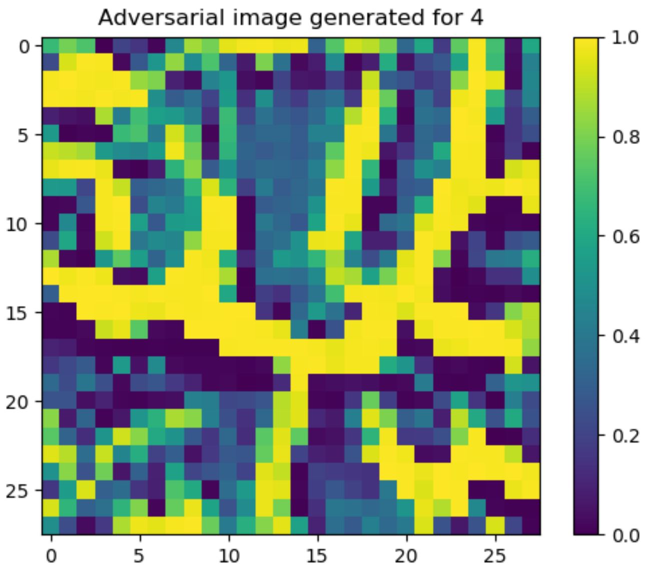


Figure 29: Adversarial image for 4.

bar.

– The images for 6 and 9 show the presence of a lobe near the

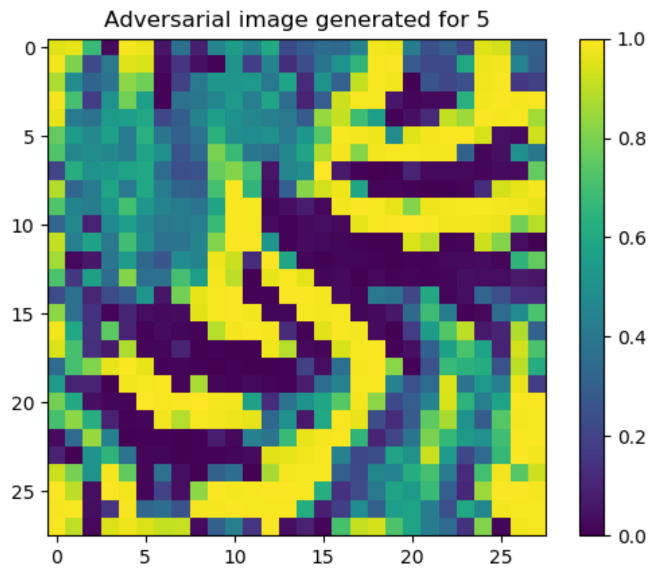


Figure 30: Adversarial image for 5.

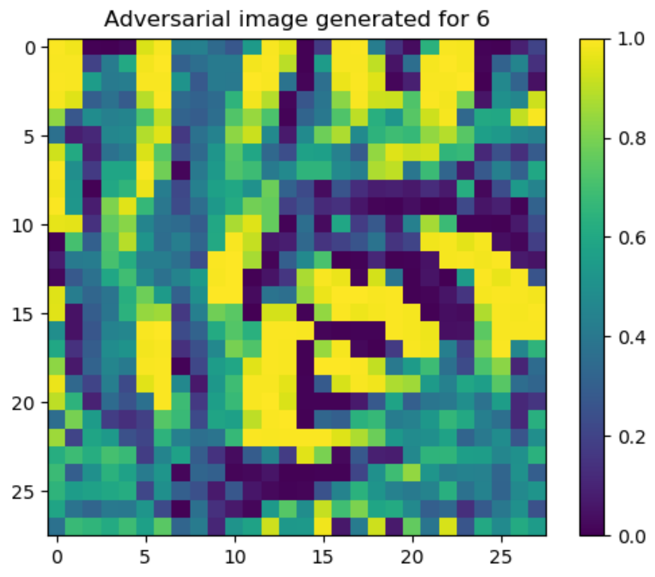


Figure 31: Adversarial image for 6.

bottom and top of the image respectively.

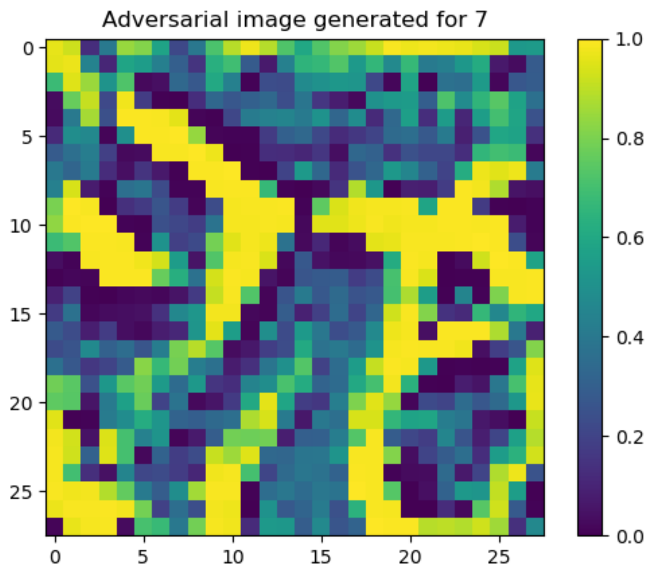


Figure 32: Adversarial image for 7.

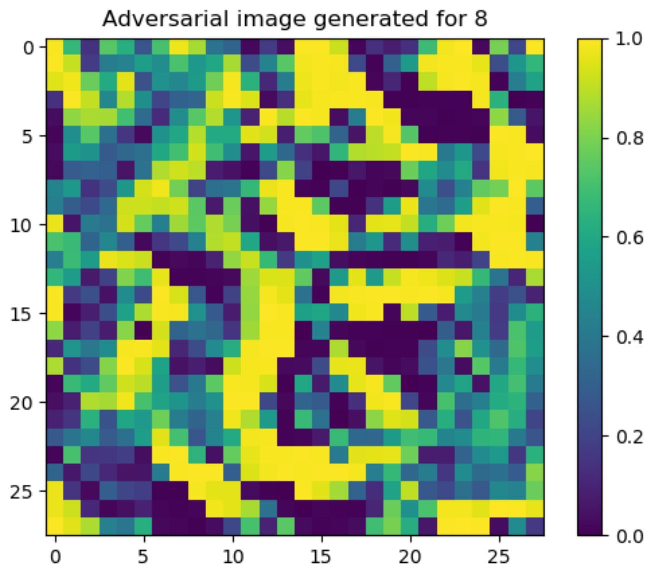


Figure 33: Adversarial image for 8.

Targeted attack

Some information

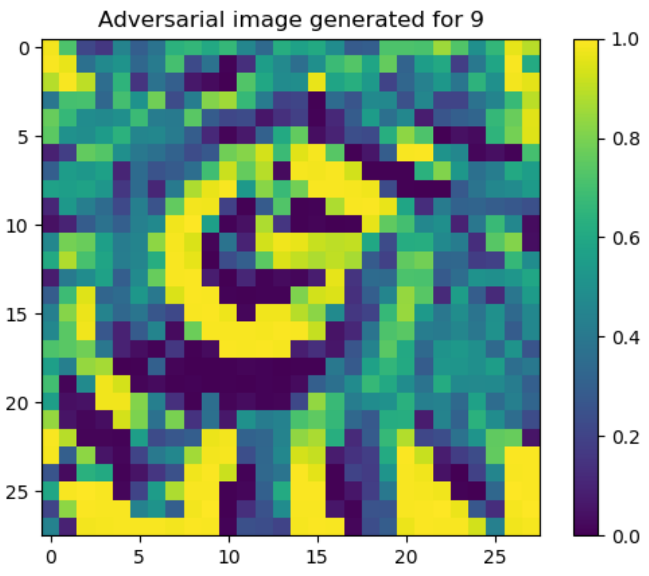


Figure 34: Adversarial image for 9.

- The value of β used is 0.185
- The original noise is being updated for 670 iterations.

The above two parameters were tuned as to strike a balance between maintaining a high confidence and preventing the mean squared error from exploding.

NOTE : It is unreasonable to plot all possible pairs of images. So, in the code it asks the user to select a target class and what he/she wants the image to look like. The user can explore all possibilities from there.

Images for some of the pairings are plotted from figures 35 to 44.

Two clear observations are

- When the target class and target image are close to each other, we can see some structure. This is the case in figures 35, 38, 39, 40, 42 and 44.
- In other when the target class and the target image are very different looking, the image generated is a blurry mess with very little structure to it.

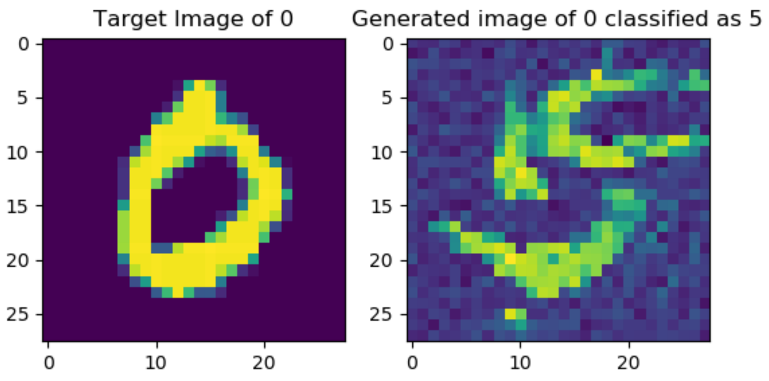


Figure 35: Generated image of 0 classified as 5.

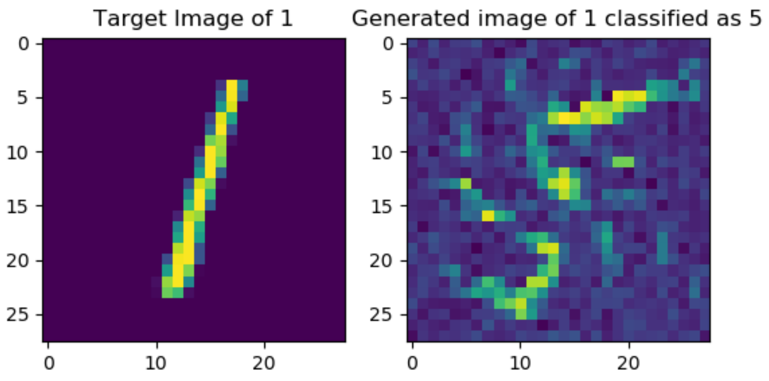


Figure 36: Generated image of 1 classified as 5.

Noise addition

Some information

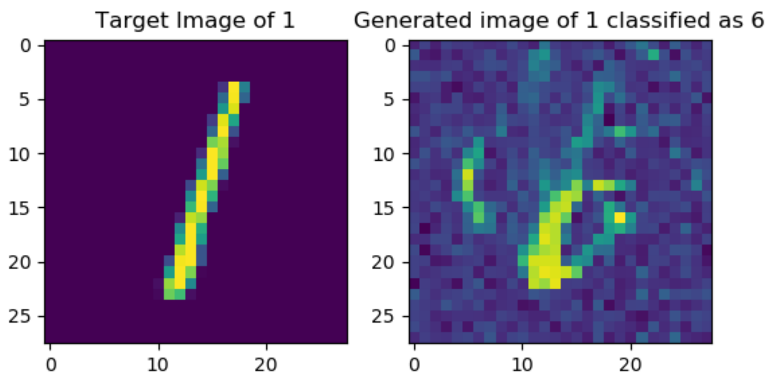


Figure 37: Generated image of 1 classified as 6.

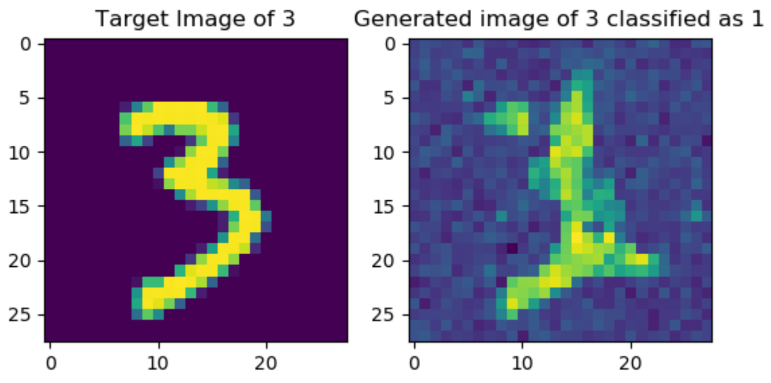


Figure 38: Generated image of 3 classified as 1.

- The learning rate used is 0.1

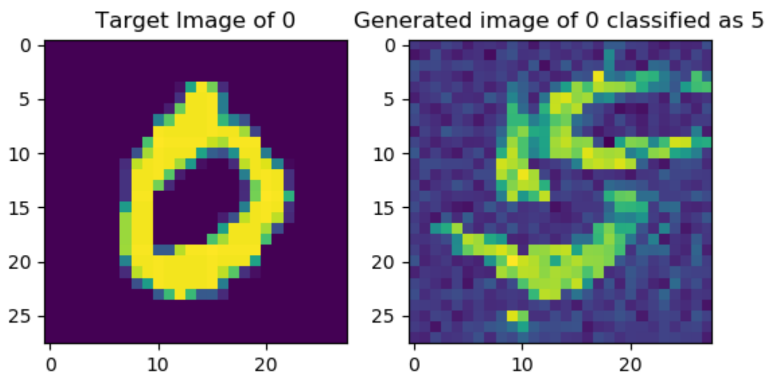


Figure 39: Generated image of 0 classified as 5.

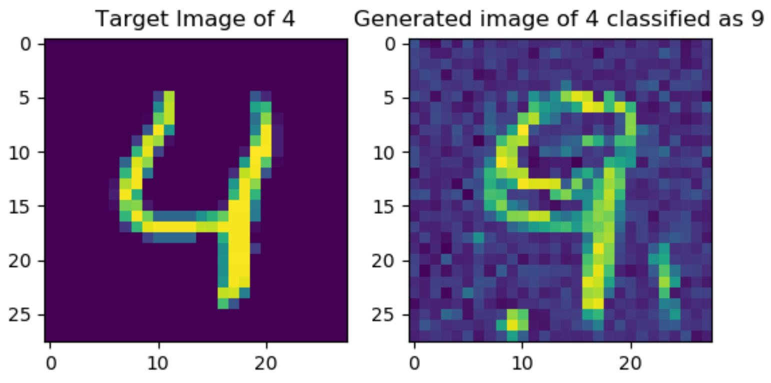


Figure 40: Generated image of 4 classified as 9.

- The iteration on the noise is run till the target class gets the highest probability.

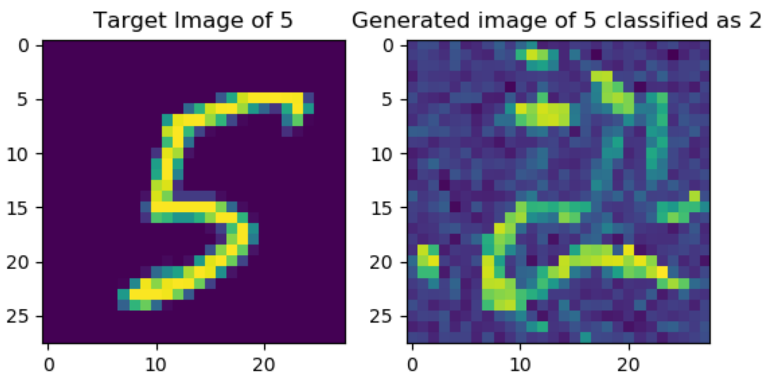


Figure 41: Generated image of 5 classified as 2.

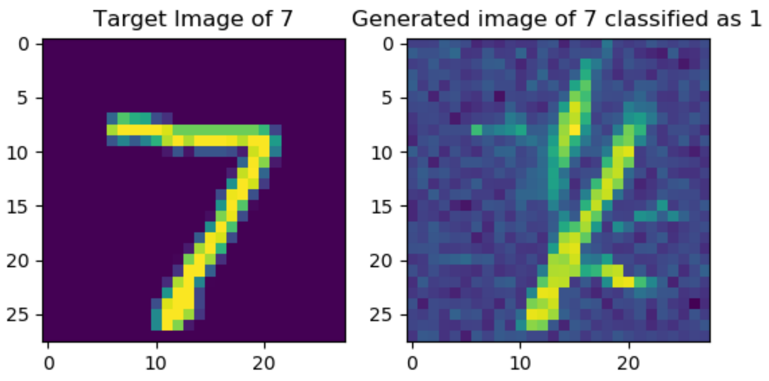


Figure 42: Generated image of 7 classified as 1.

In this section I plot 2x2 grids with the following images

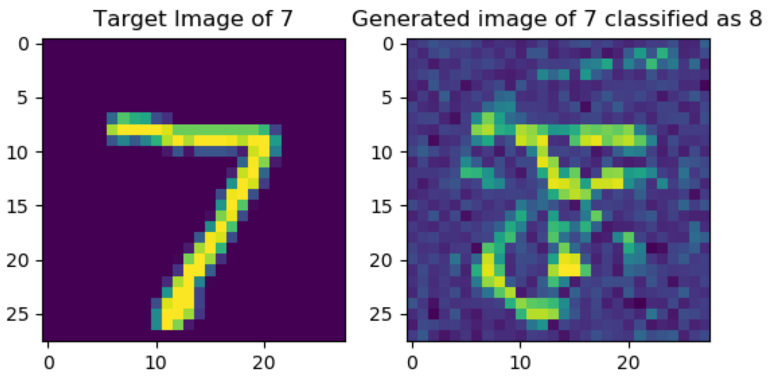


Figure 43: Generated image of 7 classified as 8.

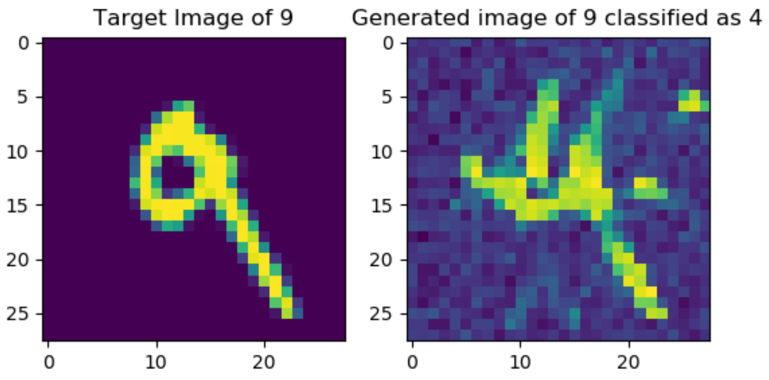


Figure 44: Generated image of 9 classified as 4.

- The original, clean image for a class

- Noise added to the image to classify it into a different class.
- The generated noise.
- The noise added to another image along with it's predicted class.

10 such cases are plotted in figures 45 to 54. The same can be tried for any triplet while running the code.

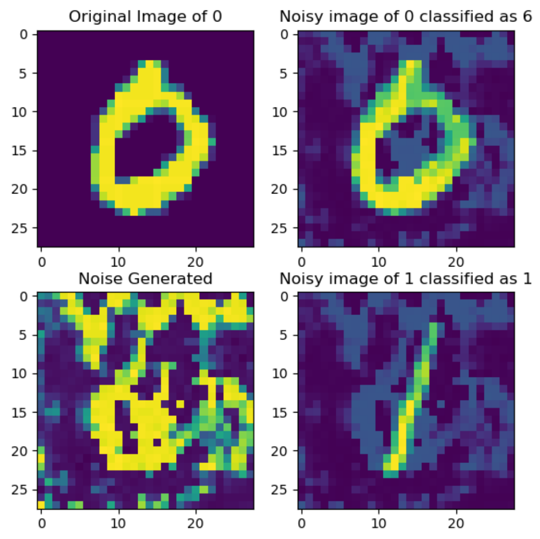


Figure 45:

Some observations are

- A lot of the images are still perceptually recognizable till after adding the noise which causes the classifier to wrongly classify them.
- In most of the cases, adding the noise generated for a certain class to an image of another class does not lead to a wrong classification.

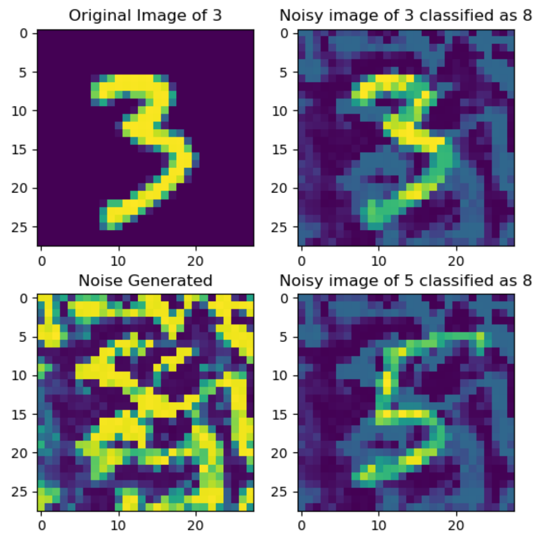


Figure 46:

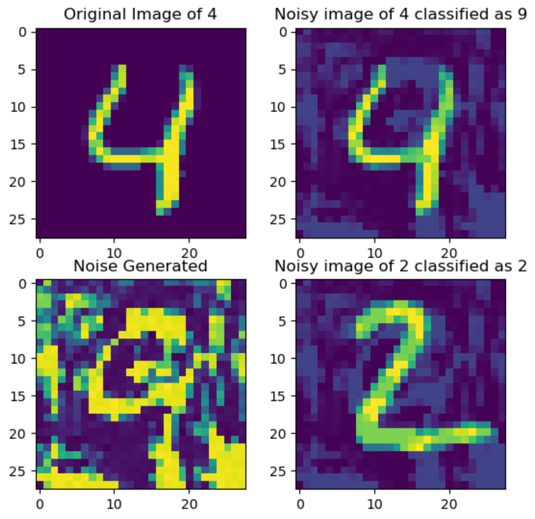


Figure 47:

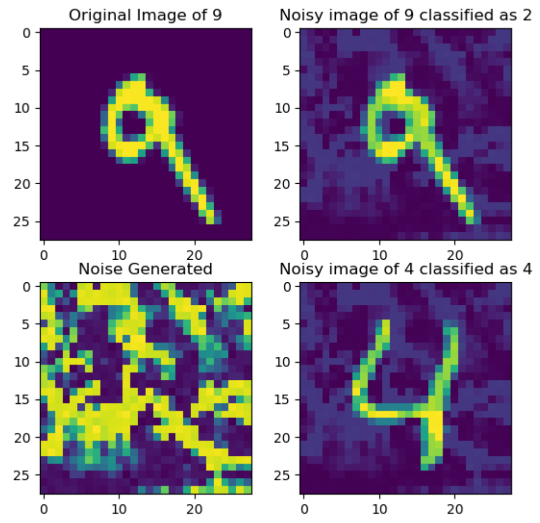


Figure 48:

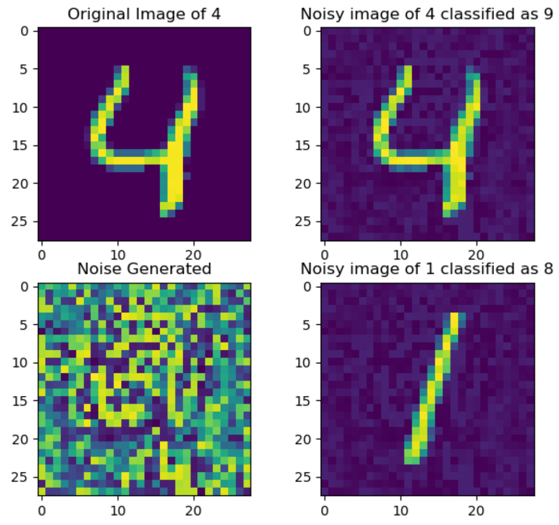


Figure 49:

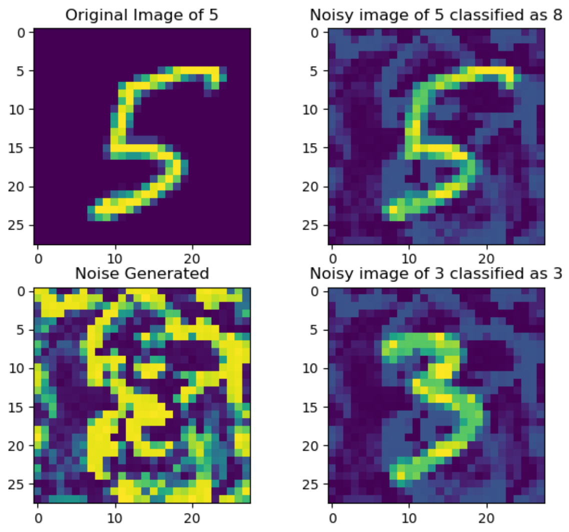


Figure 50:

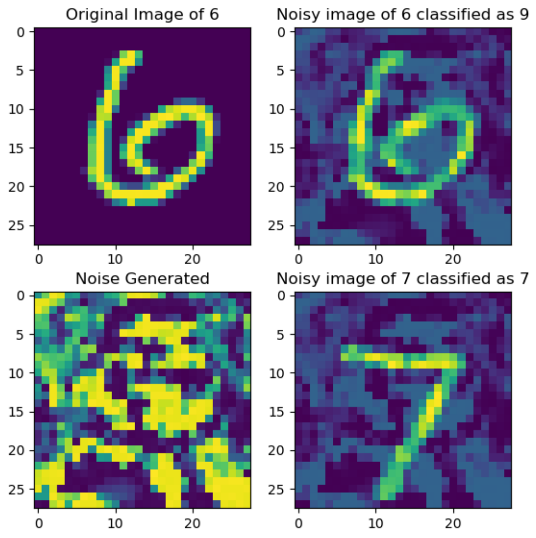


Figure 51:

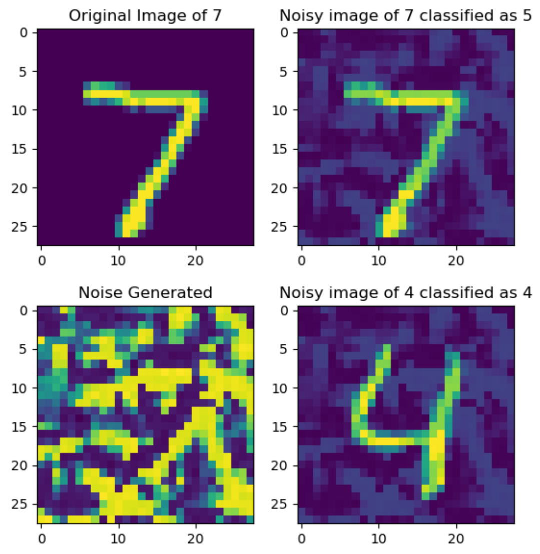


Figure 52:

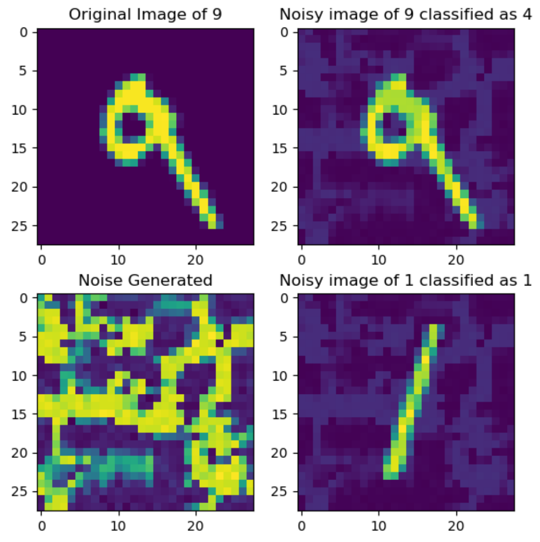


Figure 53:

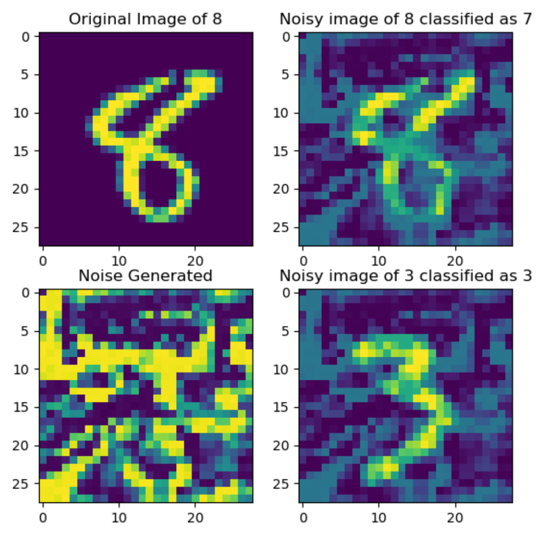

Original Image of 8
Noisy image of 8 classified as 7
Noise Generated
Noisy image of 3 classified as 3

Figure 54: