EE6132 Advanced Topics in Signal Processing - Assignment No 2

Name: Atishay Ganesh
Roll Number: EE17B155

September 23, 2019

1 Abstract

The goal of this assignment is the following.

- To experiment with Convolutional Neural Networks.
- To study the effects of Batch Normalisation.
- To visualise the filters and activations of the CNN.
- To explore adversarial attacks on CNN's.

2 Assignment

2.1 Part 0

Implementation Details: We use Pytorch in Python 3.6 for implementing the Convolutional Neural Networks. We use torch as the Deep Learning Framework, Tensorboard and pyplot for plotting the graphs and torchvision for the MNIST dataset and the appropriate transforms.

Importing the standard libraries

```
import sys
import torch
import torch.nn as nn
from torchvision import datasets, transforms
import torch.nn.functional as F
import matplotlib.pyplot as plt
from torch.utils.tensorboard import SummaryWriter
from torch.utils.data import RandomSampler
import numpy as np
```

```
from torch.utils.data.sampler import SubsetRandomSampler
from functools import partial
```

We write code to load the MNIST Data and split it into train, validation and test sets.(90-10 split) We normalise the images to -1 to 1.

```
    train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('../data', train=True, download=True,
                       transform=transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.5,), (0.5,))
                       ])),batch_size= batch_size,
                       shuffle=False,sampler=train_sampler,**kwargs)
```

2.2 MNIST Classifiaction using CNN

First we construct the CNN Class that will be used for the rest of the assignment.
Forward Pass function:

```
class CNN(nn.Module):
    def __init__(self,batchnorm=False):
        super(CNN, self).__init__()
        self.batchnorm = batchnorm
        if self.batchnorm:
            self.conv2_bn = nn.BatchNorm2d(32)
            self.fc1_bn = nn.BatchNorm1d(500)
        self.conv1 = nn.Conv2d(in_channels=1,out_channels =32,
        kernel_size= 3, stride = 1,padding=1)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=32,
        kernel_size= 3, stride = 1,padding=1)
        self.fc1 = nn.Linear(4*4*98, 500)
        self.fc2 = nn.Linear(500, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2, 2)
        x = self.conv2(x)
        if self.batchnorm:
            x = self.conv2_bn(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 4*4*98)
        x = (self.fc1(x))
```

```
        if self.batchnorm:
            x = self.fc1_bn(x)
        x = F.relu(x)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

We train the network using negative log likelihood loss with optimizer SGD.

```
    def train_epoch(self,epoch):
        for batch_idx, (data, target) in enumerate(self.train_loader):
            data, target = data.to(self.device), target.to(self.device)
            self.optimizer.zero_grad()
            output = self.model(data)
            correct = 0

            loss = F.nll_loss(output, target)
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

            loss.backward()
            self.optimizer.step()
            self.writer.add_scalar('Loss/train', loss.item(),self.steps)
            self.writer.add_scalar('Accuracy/train',
            100*correct / len(target),self.steps)
            self.steps +=1

            if batch_idx % self.logging_interval == 0:
                print(
                'Train Epoch: {} [{}/{} ({:.0f})%]\tLoss: {:.6f}\n'.format(
                    epoch+1, batch_idx * len(data), self.size[0],
                    100.*batch_idx*len(data) /self.size[0],
                    loss.item()),self.steps)
                self.test_epoch('valid')
                self.model.train()

def test_epoch(self,type_set):
    self.model.eval()
    curr_loader = self.valid_loader if type_set == 'valid' else self.test_loader
    size_value = 1 if type_set =='valid' else 2
    correct = 0
    loss = 0
    with torch.no_grad():
        for data, target in curr_loader:
            data, target = data.to(self.device), target.to(self.device)
            output = self.model(data)
```

```
            loss += F.nll_loss(output, target, reduction='sum').item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()
    loss /= self.size[size_value]
    if type_set== 'valid':

        self.writer.add_scalar('Loss/valid',
                 loss,self.steps)
        self.writer.add_scalar('Accuracy/valid',
                  100.*correct / self.size[size_value],self.steps)

    print('{} set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        type_set,loss, correct, self.size[size_value],
        100.*correct/self.size[size_value]),self.steps)
```

2.2.1 Without Batch Normalisation

We train it with ReLU Non-linearity for 10 epochs. The results of the same are shown.

Test Set: Average loss: 0.0461, Accuracy: 9845/10000 (98%)

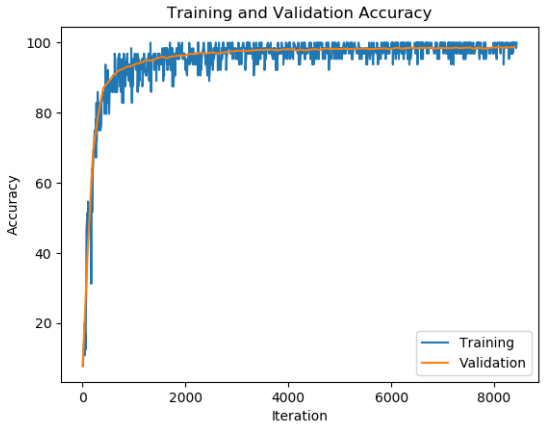

Figure 1: Training and Validation Accuracy

2.2.2 Description of the CNN

We list the dimensions of the input and output at each layer.

- Conv1 Input: 1,28,28 Output: 32,28,28

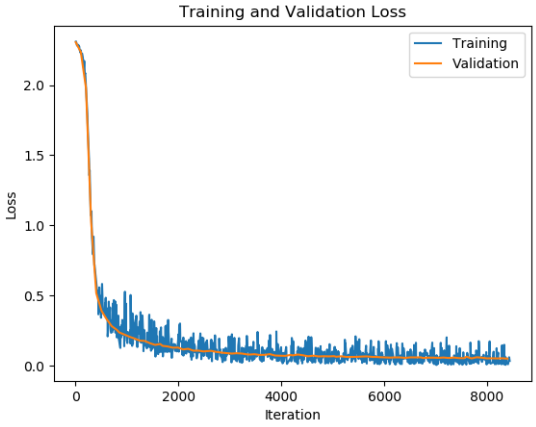

Figure 2: Training and Validation Loss

Figure 3: True Labels: [8,9,5,7,0] Predicted Labels:[8,9,5,7,0] respectively

- Maxpool Input: 32,28,28 Output: 32,14,14
- Conv2 Input: 32,14,14 Output: 32,14,14
- Maxpool Input: 32,14,14 Output: 32,7,7
- Linear 1 Input: 1568 Output: 500
- Linear 2 Input: 500 Output: 10

We discuss the number of parameters in our network

- Conv1: 320 (288 Weights + 32 Biases)
- Conv2: 9248 (9216 Weights +32 Biases)
- Linear1: 784500 (784000 Weights and 500 Biases)
- Linear2: 5010 (5000 Weights and 10 Biases)

9568 Parameters belong to the convolutional layers. 789510 Parameters belong to the fully connected layers.
The number of neurons in our network is

- Conv1: 25088 Neurons

- Conv2: 25088 Neurons
- Linear1: 500 Neurons
- Linear2: 10 Neurons

50176 Neurons belong to the convolutional layers. 510 Neurons belong to the fully connected layers.

2.2.3 Batch Normalization

We use batch normalization between conv2 and its relu, and Linear 1 and its relu. The use of batch normalization was initially introduced to combat the effect of the internal covariate shift. However, recent studies state that batch normalization smoothens the optimization landscape, and this is reason the performance is better. Test Set: Average loss: 0.0316, Accuracy: 9897/10000 (99%) Batch-normalization improved the test accuracy by a small amount that was reducing as the network trained more, and the training time to reach a certain validation accuracy threshold did reduce by a good amount. The conditions were same as the previous case.

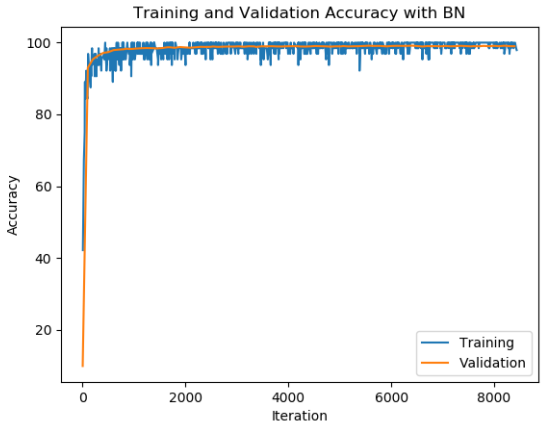

Figure 4: Training and Validation Accuracy

2.3 Visualing Convolutional Neural Networks

2.3.1 Plot filters

We observe certain patterns which are difficult to interpret as such, since the filter size is extremely small. If the filter size was higher we would expect the first layer to have patterns like a sobel filter. The later layers do not

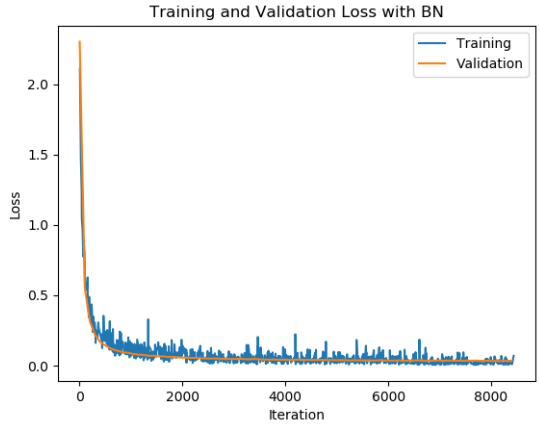

Figure 5: Training and Validation Loss

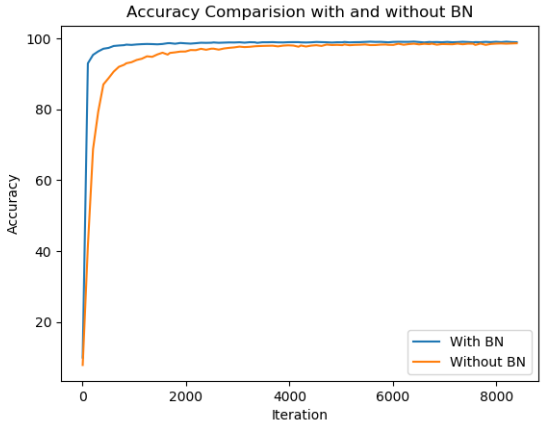

Figure 6: Accuracy with and without Batch Norm

much visual significance.

2.3.2 Visualising Activations

We Visualise some of the activations of the convolutional layers. As we go deeper the activations get smaller and the activations itself are not so clear, but the digit can still be made out in the second layer.

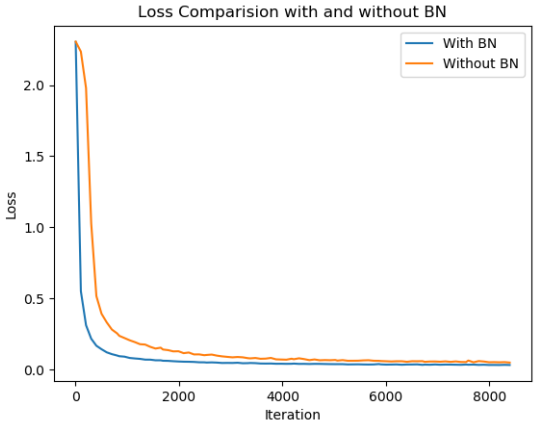

Figure 7: Loss with and without Batch Norm

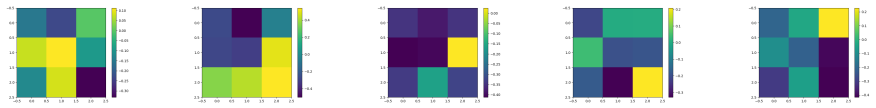

Figure 8: Filters from First Convolutional Layer

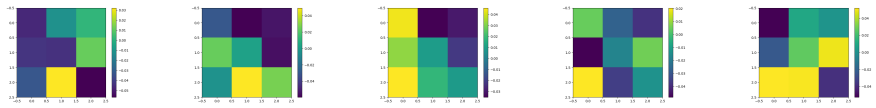

Figure 9: Filters from Second Convolutional Layer

Figure 10: Activations from First Convolutional Layer

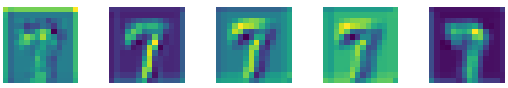

Figure 11: Activations from Second Convolutional Layer

2.3.3 Occlusion of Image

In this part we occlude parts of the image to test if the learning is proper. With a gray patch of 5x5, we observe that in most cases the probability is

pretty good, but when it over the image, the probabilities of the class reduce, in one particular case it reduced dramatically, presumably since the digit in question there (7) looked quite similar to 1. Hence it appears the neural network is learning quite well. It is interesting to observe that when we occlude at the digit or near the digit, the probability of the class decreases quite a lot. The probability of the class does not decrease too much as the patch is relatively small,but with a larger patch size it would decrease more substansially.

2.4 Adversarial Examples

We load one of the MNIST models from the first question and try and construct adversarial examples for the network.

2.4.1 Non Targeted Attack

We try to make a matrix with noise that will be classified by the network as some class with 99% probability. The plotting code is emitted from the code snippet for brevity reasons.

```
elif sys.argv[1] == 'Part_30':
    '''Non Targeted Attack'''
    X = np.random.normal(0,0.1,(1,1,28,28))
    X = X.astype('d')
    X = np.clip(X,-1,1)
    model = CNN()
    model.load_state_dict(torch.load('mnist_cnn.pt'))
    model.train()
    step = 0.1
    for i in range(10):
        p = 0
        print(X.shape)
        z = np.array([i])
        target =torch.from_numpy(z).type('torch.LongTensor')

        X1 = torch.tensor(X, requires_grad=True).type('torch.FloatTensor')
        device = torch.device( "cpu")
        plist= []
        while(p<0.99):
            X1, target = X1.to(device), target.to(device)
            loss = model.forward_logits(X1)
            prob = model(X1)
            p = np.exp(prob.detach().numpy())[0,i]
            g = torch.autograd.grad(loss[0][i],X1)
```

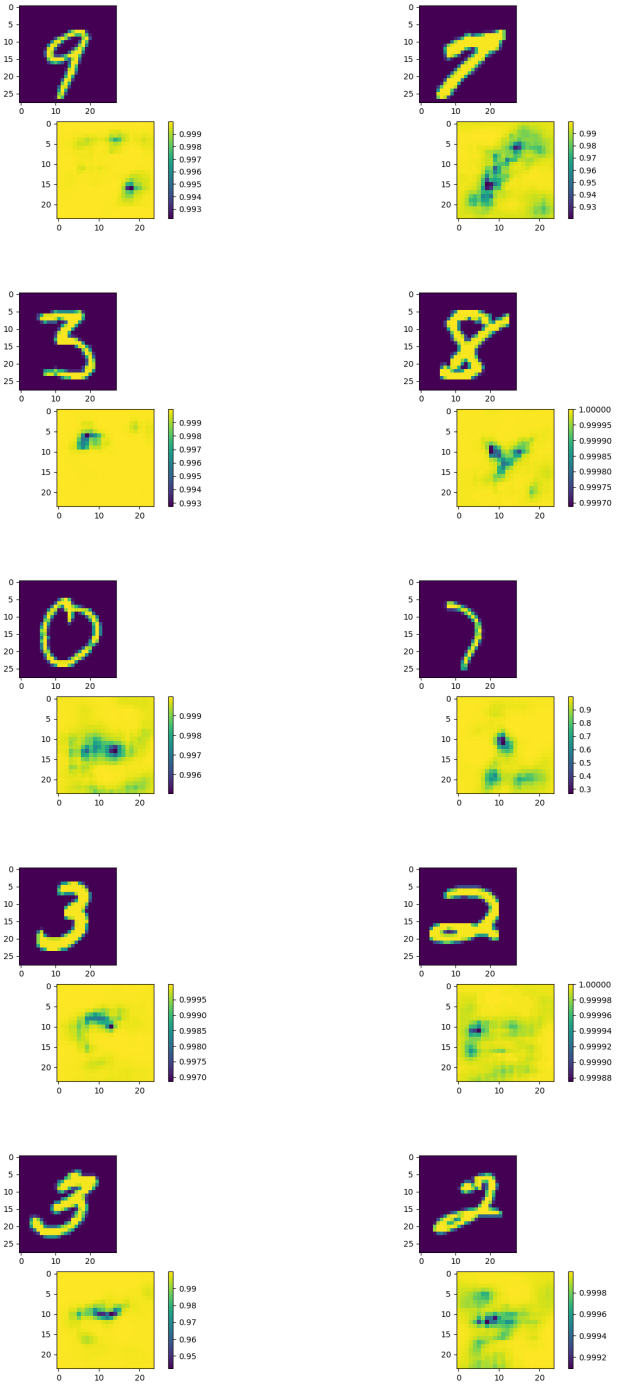

Figure 12: Probabilities after occluding parts of the image.

```
X1 = X1+step*g[0]
print(p)
plist.append(loss[0][i].detach().numpy())
```

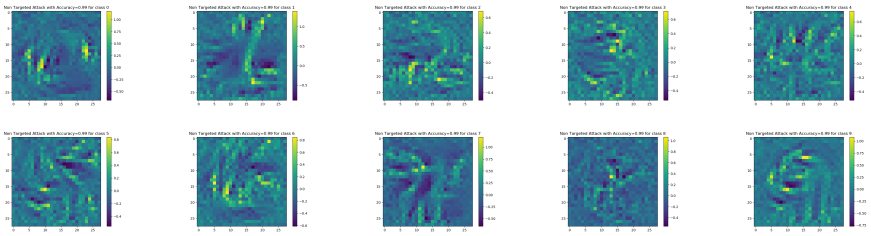

Figure 13: Non Targeted Attacks for 0-9 target classes (Row Major).

- Yes, because we have trained it till it predicts target_class with high confidence.
- No. They do not look like a number but mostly noise. The network has not been trained on these mostly noise images, hence it predicts without too much logic, hence we can pass off random noise as numbers.
- We observe that the cost is linearly increasing.

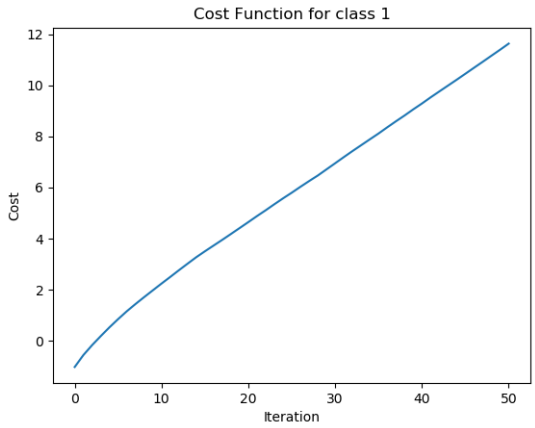

Figure 14: Class 1 output cost

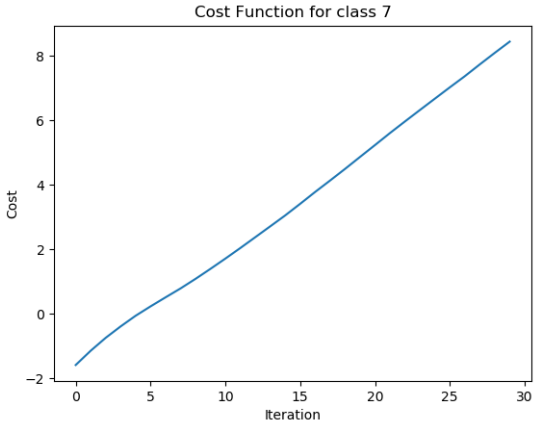

Figure 15: Class 7 output cost

2.4.2 Analysis

2.4.3 Targeted Attack

We generate some adversarial example that looks like a particular digit, which the network classifies as another digit with 90% probability.
Relavant Code Snippet

```
while(p<0.90):
    X1 = X1.to(device)
    loss = model.forward_logits(X1)[0][target] - beta*F.mse_loss(
            start_image,target_image)
    prob = model(X1)
    p = np.exp(prob.detach().numpy())[0,target]
    g = torch.autograd.grad(loss,X1)
    X1 = X1+step*g[0]
    print(p)
```

The images now look like the original image but are classified as the target image. All images have been created and are in the submission, but for brevity only 1 per input class and 1 per target class are included in this report.

The Generated Images look similar to the original images but with some amount of noise or blur.

2.4.4 Adding noise

We try to add noise to fool the network.

Relevant Code Snippet:

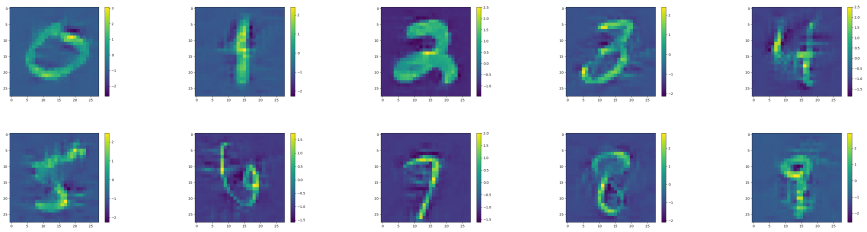

Figure 16: Targeted Attack: True Label:[0,1,2,3,4,5,6,7,8,9]
Predicted Label:[5,4,3,2,6,1,7,9,0,8]

```
while(p<0.99):
    X2 = X2.to(device)
    loss = model.forward_logits(X2)[0][target]
    prob = model(X2)
    p = np.exp(prob.detach().numpy())[0,target]
    g = torch.autograd.grad(loss,X3)
    X3 = X3+step*g[0]
    X2 = X1 +X3
    print(p)
```

The noise got from the noise attack can be added to other examples to fool the network, but it does not seem to be very robust.

Noise Targets:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
True Image:
[8, 5, 0, 9, 7, 1, 3, 1, 9, 3]
Predictions
(each row is all the true images for a single noise target.)

$$\begin{bmatrix} 1 & 3 & 1 & 1 & 2 & 1 & 3 & 1 & 1 & 3 \\ 2 & 2 & 0 & 2 & 2 & 2 & 2 & 2 & 8 & 2 \\ 8 & 8 & 0 & 9 & 7 & 1 & 3 & 3 & 9 & 3 \\ 8 & 5 & 0 & 4 & 7 & 8 & 9 & 4 & 9 & 9 \\ 8 & 5 & 0 & 9 & 9 & 5 & 3 & 9 & 9 & 5 \\ 8 & 8 & 0 & 4 & 7 & 1 & 3 & 1 & 9 & 3 \\ 8 & 5 & 0 & 9 & 7 & 7 & 3 & 3 & 9 & 3 \\ 8 & 8 & 0 & 8 & 8 & 8 & 3 & 8 & 8 & 3 \\ 8 & 5 & 0 & 9 & 7 & 8 & 3 & 9 & 9 & 3 \\ 8 & 8 & 0 & 8 & 7 & 0 & 3 & 0 & 9 & 3 \end{bmatrix}$$

In a good number of cases the network is fooled by adding the noise, to predict the noise target instead of the True output. This method does not appear to be infallible, but it does very well at times, especially for 2,8,1 in that order.

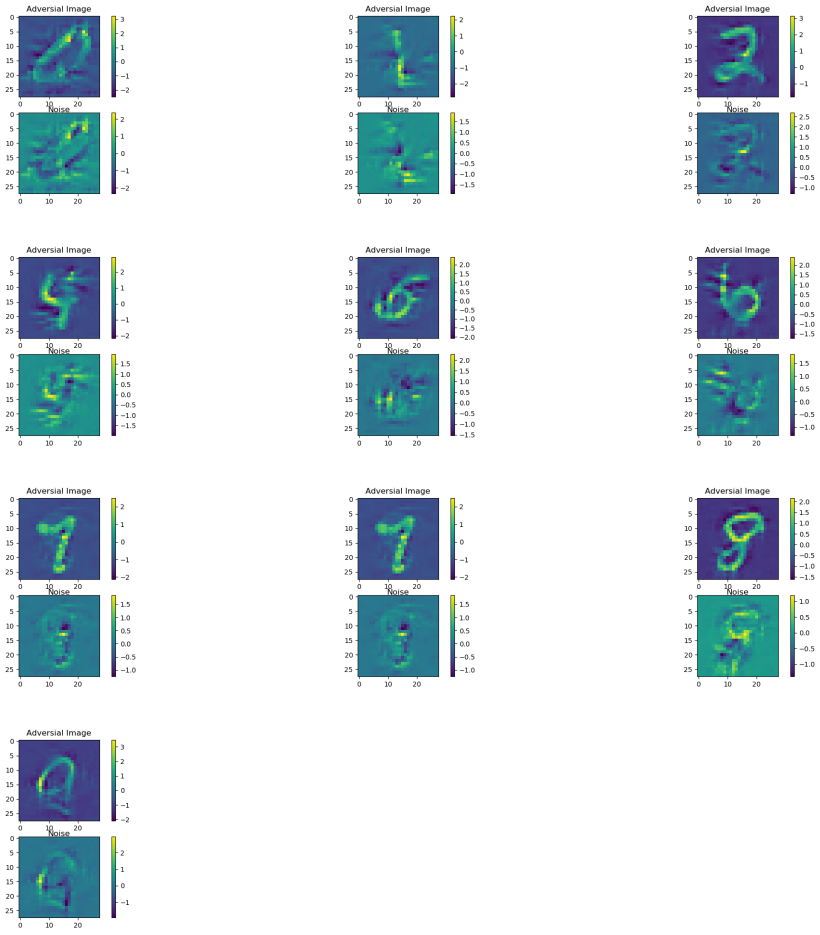

Figure 17: Attack with noise. The true labels :[0-9]
Predicted Labels: True Label +1 (mod 10)
The Top image is the adversarial example and the bottom one is the noise

3 Conclusions

- We experimented with Convolutional Neural Networks for image classification.
- We studied the effects of Batch Normalisation in improving the performance of the neural network.
- We visualised the filters and activations of the CNN.
- We explored adversarial attacks on CNN's (targeted, non targeted and noise driven attacks).