Deep Learning
CS7015
Programming Assignment 3
Convolutional Neural Networks

Shubhangi Ghosh - EE15B129
Monisha J - CS15B053

March 2018

1 Objective

The objective of this assignment was to construct a convolutional neural network using *tensorflow* and experiment with different hyper-parameters and evaluate its performance on the Fashion-MNIST dataset.

2 Basic Model

2.1 Dimensions of Input and Output to each layer:

Dimensions are reported per-image input.

2.1.1 Convolutional and Pooling Layers:

CONV1:
Input : 28x28x1
Output : 28x28x64
Filters: 64x3x3
Parameters: 3x3x64 = 576
POOL1:
Input : 28x28x64
Output : 14x14x64
Parameters: 0
CONV2:

Input : 14x14x64
Output : 14x14x128
Filters: 128x3x3
Parameters: 3x3x128 = 1152
POOL2:
Input : 14x14x128
Output : 7x7x128
Parameters: 0
CONV3:
Input : 7x7x128
Output : 7x7x256
Filters: 256x3x3
Parameters: 3x3x256 = 2304
CONV3:
Input : 7x7x256
Output : 7x7x256
Filters: 256x3x3
Parameters: 3x3x256 = 2304
POOL3:
Input : 7x7x256
Output : 3x3x256
Parameters: 0

Total no of Parameters for convolutional layers: 576+1152+2304+2304 = 6336 Total no. of neurons for Convolutional layers = total no. of outputs = 28x28x64 + 14x14x64 + 14x14x128 + 7x7x128 + 7x7x256 + 7x7x256 + 3x3x256 =
50176 + 12544 + 25088 + 6272 + 12544 + 12544 +2304 = 121472

2.1.2 Fully connected Layers:

FC1:
Input: 2304
Output: 1024
Parameters: 2304x1024 + 1024(biases) = 2360320
FC2:
Input: 1024
Output: 1024
Parameters: 1024x1024 + 1024(biases) = 1049600
Softmax:

Input: 1024
Output: 10
Parameters: 1024x10 = 10240

Total no of Parameters for fully-connected layers: 2360320+1049600+10240 = 3420160
Total no. of neurons for fully-connected layers: 1024+1024+10 = 2058

2.2 Parameters of Complete Model:

Total no. of parameters in model: 6336 + 3420160 = 3426496
Total No. of neurons in model: 121472 + 2058 = 123530

2.3 Results

Training Loss : 20.09
Train Accuracy : 0.995
Validation Loss: 0.45
Validation Accuracy : 0.914

A plot of training loss and validation loss with respect to epochs is shown in Figure 1.

3 Best Performing Model

3.1 Layers

1. Convolutional Layer - 32 filters, 3 x 3 kernel, stride 1
2. Activation layer - Relu
3. Max Pooling Layer - 2 x 2 pooling, stride 2
4. Convolutional Layer - 64 filters, 3 x 3 kernel, stride 1
5. Activation layer - Relu
6. Max Pooling Layer - 2 x 2 pooling, stride 2
7. Convolutional Layer - 128 filters, 3 x 3 kernel, stride 1

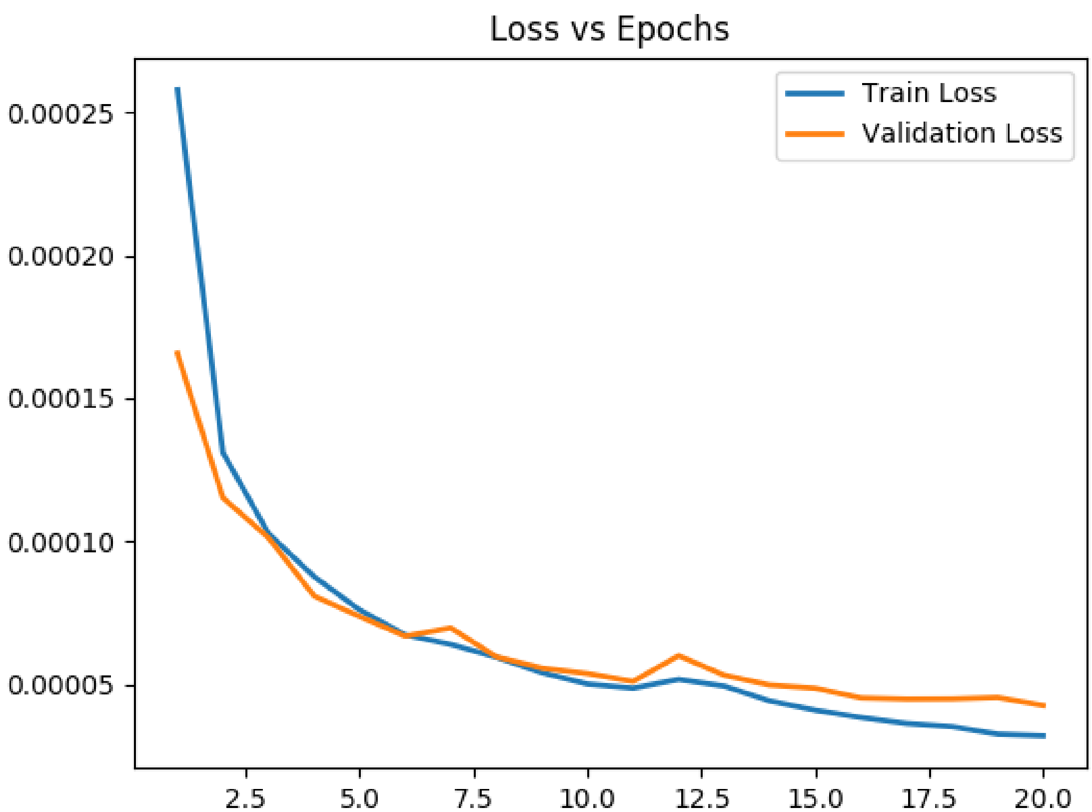

Figure 1: Basic Model - Loss vs Epochs

8. Activation layer - Relu
9. Convolutional Layer - 256 filters, 3 x 3 kernel, stride 1
10. Activation layer - Relu
11. Max Pooling Layer - 2 x 2 pooling, stride
12. Flatten to 7*7*256
13. Fully Connected Layer - 1000 neurons
14. Fully Connected Layer - 500 neurons
15. Fully Connected Layer - 500 neurons
16. Fully connected Softmax Layer - 10 neurons

3.2 Configuration

Optimizer : Adam
Loss Function : Cross Entropy
Learning Rate : 0.001
Epochs : 15
Batch Size : 50
Initialization Method : He
Early Stopping : False
Annealing : False

3.3 Data

The training data used is the original training data as well an additional set of 165000 augmented data. Thus the total size of the training set is 220000.

3.4 Parameters

1. Convolutional Layer 1- 32 filters, 3 x 3 kernel
   32*3*3 = 288 parameters
2. Max Pooling Layer - 2 x 2 pooling
   0 parameters
3. Convolutional Layer - 64 filters, 3 x 3 kernel
   64*3*3 = 576 parameters

4. Max Pooling Layer - 2 x 2 pooling
5. Convoultional Layer - 128 filters, 3 x 3 kernel
   128*3*3 = 1152 parameters
6. Convolutional Layer - 256 filters, 3 x 3 kernel
   256*3*3 = 2304 parameters
7. Max Pooling Layer
   0 parameters
8. Flatten to 7*7*256 = 12544
   0 parameters
9. Fully Connected Layer - 1000 neurons
   12544 * 1000 = 12544000 parameters
10. Fully Connected Layer - 500 neurons
    1000 * 1000 = 1000000 parameters
11. Fully Connected Layer - 500 neurons
    1000 * 500 = 500000 parameters
12. Fully connected Softmax Layer - 10 neurons
    500 * 10 = 5000 parameters

Total parameters in convolutional layers = 288+576+1152+2304 = 4320
Total neurons in convolutional layers = 28 *28*32 + 14*14*64 + 7*7 *128+ 7*7*256 = 56448

Total parameters in fully connected layers = 12554000 + 1000000 + 500000 + 5000
Total neurons in fully connected layers = 1000 + 1000 + 500 + 10 = 2560

Total number of parameters = 4320 + 14059000 = 14063320
Total number of neurons = 56448 + 2560 = 59008

3.5 Results

Train Accuracy : 95
Validation Accuracy : 93.5
Test Accuracy : 93.27

4 Additional Techniques

A few additional techniques were employed to improve the performance of the model. They are listed in the following subsections.

4.1 Data Augmentation

Data Augmentation is the process by increasing the size of the dataset by introducing artificial diversity in the data by applying certain plausible transformations in the data. This is usually applied to image data. rotation, translation, flipping, cropping, scaling, adding noise to the image, blurring the image, etc.

We have tried two kinds of image augmentation techniques :

4.1.1 imgaug library

The python library imgaug was used.
In our implementation, $augmentation_ia.py$ applies augmentation on the training set and writes the augmented data to $aug\_train_ia.csv$. The size of the augmented set is 3 times the size of the training set.

This library provides a variety of augmenting functions as listed on the github page linked above. The techniques we used are as:

- Cropping and padding
- Flipping Horizontally
- Average Blurring
- Adding Salt and Pepper Noise
- Embossing
- Sharpening
- Additive Gaussian Noise
- Contrast Normalization
- Sequential combination of one or more of these techniques

The training image as well as well as corresponding augmented images are shown for a few instances in Figures 2,3,4 and 5.

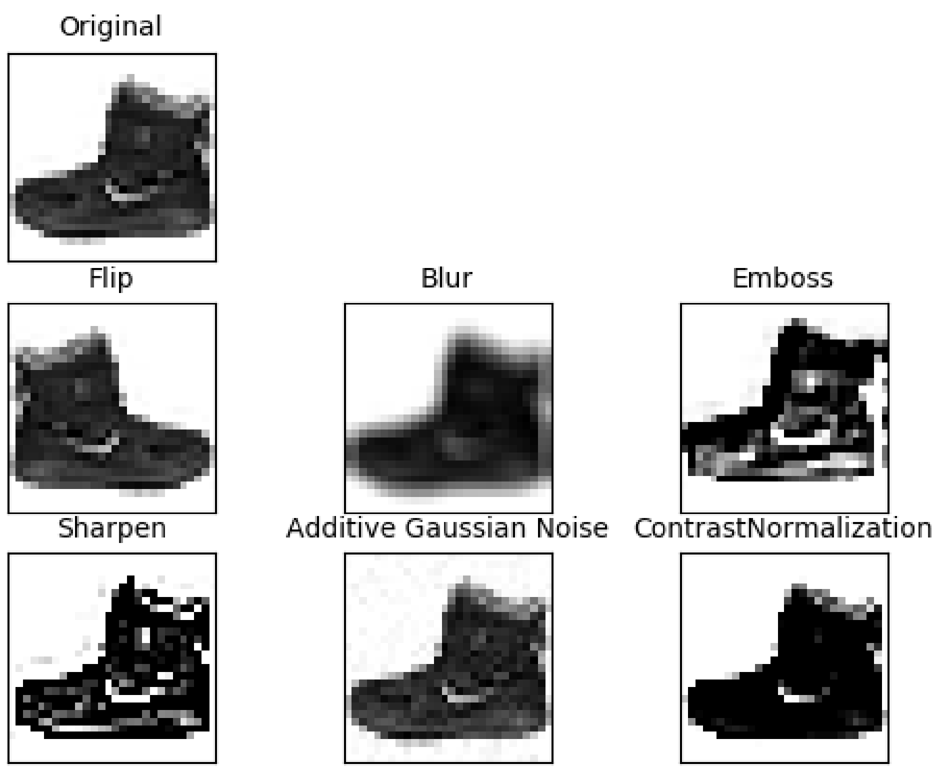

Figure 2: Data Augmentation - 1

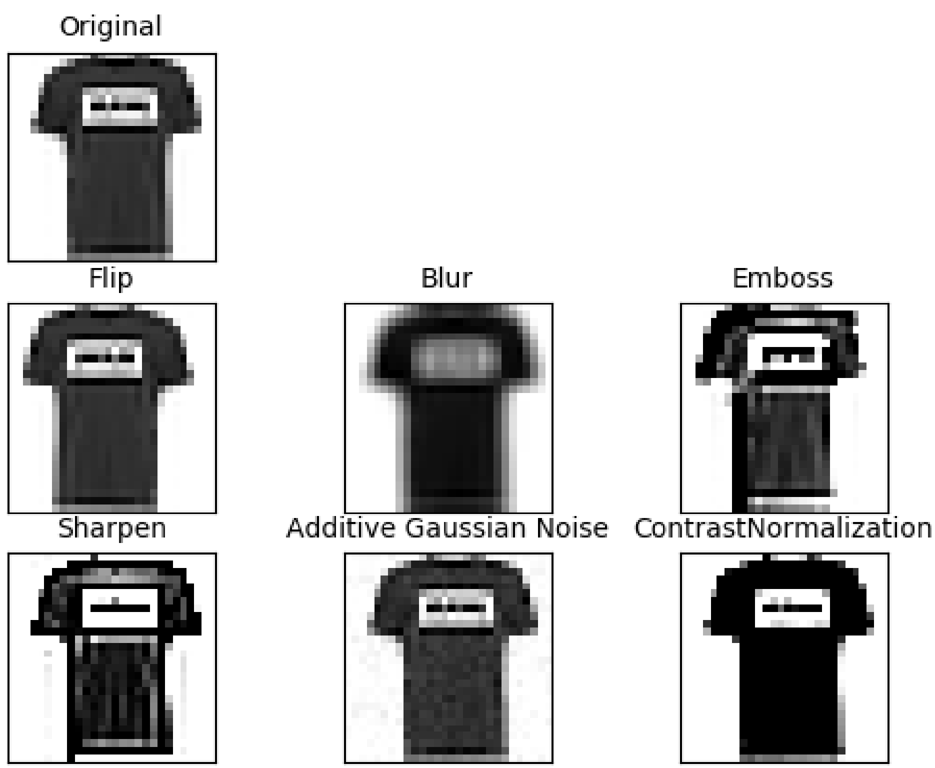

Figure 3: Data Augmentation - 2

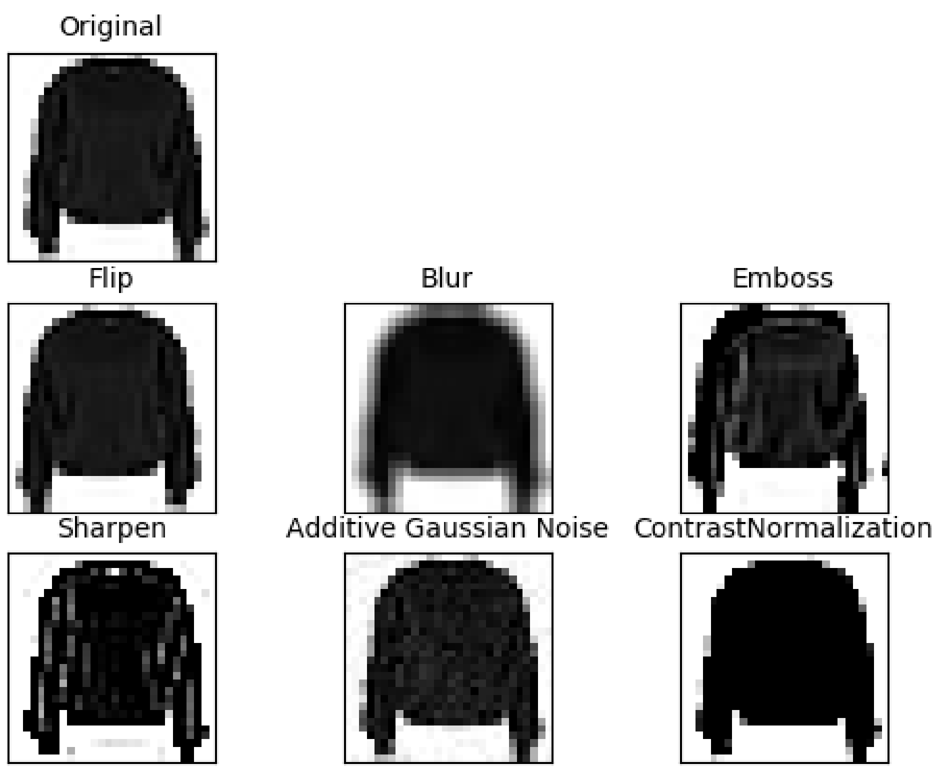

Figure 4: Data Augmentation - 3

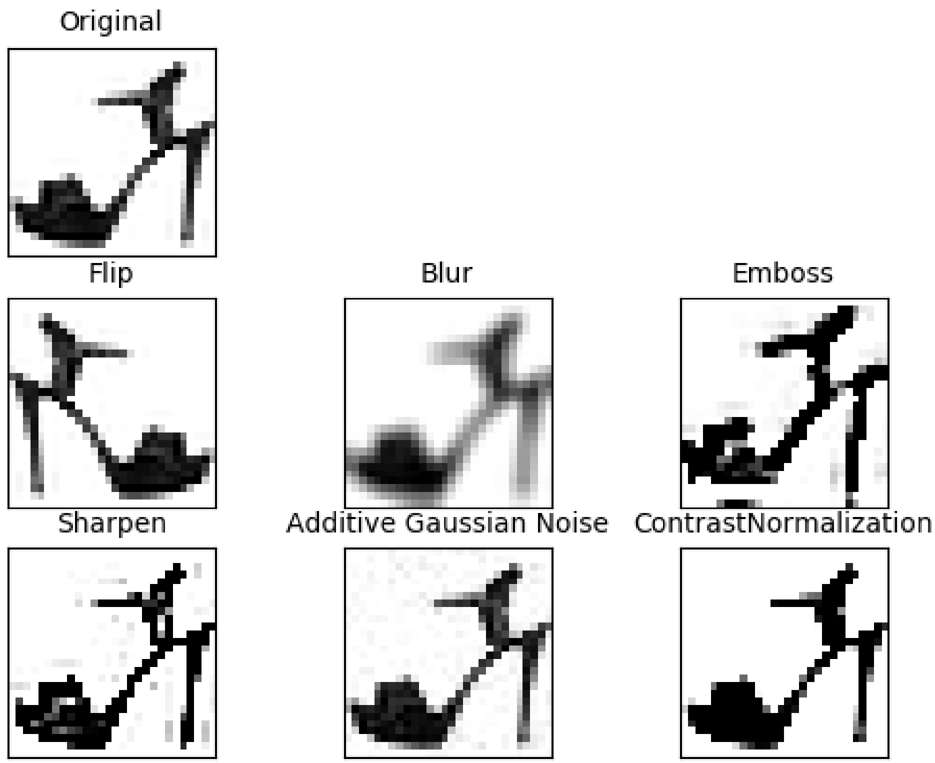

Figure 5: Data Augmentation - 4

4.1.2 Inception Preprocessing

The images were passed through the official tensorflow module $inception_preprocessing.py$. In our implementation, $augmentation_ip.py$ applies augmentation on the training set and writes the augmented data to $aug\_train_ip.csv$. The size of the augmented set is twice the size of the training set.
This module is the preprocessing module written for the Inception net. It performs cropping, scaling, distortion and flipping(left or right) of images.

4.2 Annealing

Annealing of learning rate was implemented. When the validation loss in the current epoch is higher than the validation loss in the previous epoch, the learning rate is halved and the epoch is restarted, that is, the model is restored to the one at the end of the previous epoch.

4.3 Batch Normalization

Batch Normalization is the process of normalizing the activations of a certain layer. It helps prevent overfitting. It was also observed that the learning was slower when batch normalization was used.

4.4 Dropout

Dropout was implemented using a placeholder for the dropout values. Dropout proved useful to prevent overfitting in deep convolutional networks. However, using the same dropouts on augmented data led to underfitting, and hence dropout had to controlled and reduced to almost 0 when the dataset size was huge.

4.5 Early Stopping

Early stopping with a patience of 5 epochs was used. It was implemented as follows :
If early stopping is enabled, the validation loss at any epoch is compared to the validation loss five epochs earlier. If there is no improvement in the loss, the training is terminated and the model is restored to the one five epochs before. This is implemented using the tensorflow $Saver()$ and *checkpointing* mechanisms.

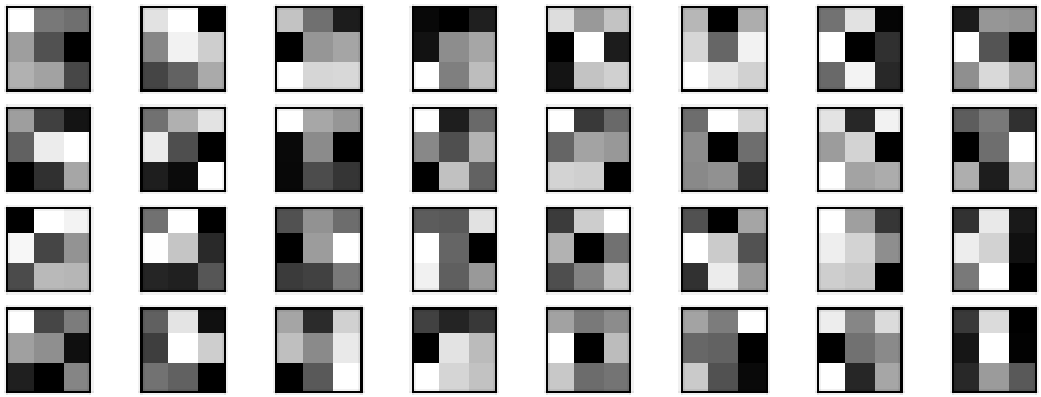

Figure 6: Convolutional Layer 1 - Filter Visualization

4.6 Xa and He Initialization

Xavier and He initiliazation were implemented. The technique that gave better results was He, which is expected, as the activation function we have used is $relu$. The idea is that these initializers vary based on the number of inputs and outputs for the neuron layer.

5 Analysis

5.1 Visualization of filters

The first convolutional layer uses 64 filters of dimensions 3x3. These filters were visualized using *matplotlib* as shown in Figure 6.
From the figure, we see that each filter has a high weight(dark black) region in a different index/combination of indices of the 3 x 3 filter. From this we can conclude that each filter is trying to detect a different edge/pattern in the image.

5.2 Guided Backpropagation

1. Here, we calculate gradients of activation functions of 10 neurons in the CONV4 layer wrt an input image in the validation set.

2. For guided backpropagation, we set the gradient to 0 if the backpropagated gradients are negative. This makes sense we are using RelU

activation which does the same for forward propagation. Adding this non-linearity helps us get clearer images of the pattern that a particular neuron identifies. The visualizations can be seen in Figure 7.

5.3 Fooling the network

Fooling of the network provided insight into its working. Fooling was implemented as follows : A certain number of pixels, say k pixels, where k was varied from 1 to 100, were randomly changed to a different value from 0 to 256. This was done for each image in the validation set and the accuracy on the validation set was measured. A plot of accuracy vs number of pixels changed for each image in the validation set is shown in Figure 8.
From the plot, we see that changing the pixels causes fall in the accuracy. And more the number of pixels changed, lesser is the validation accuracy. This can be explained by the fact that even though the number of pixels being changed is relatively less, the network is getting fooled into predicting a different classes than what it would've predicted had the pixels not been changed.

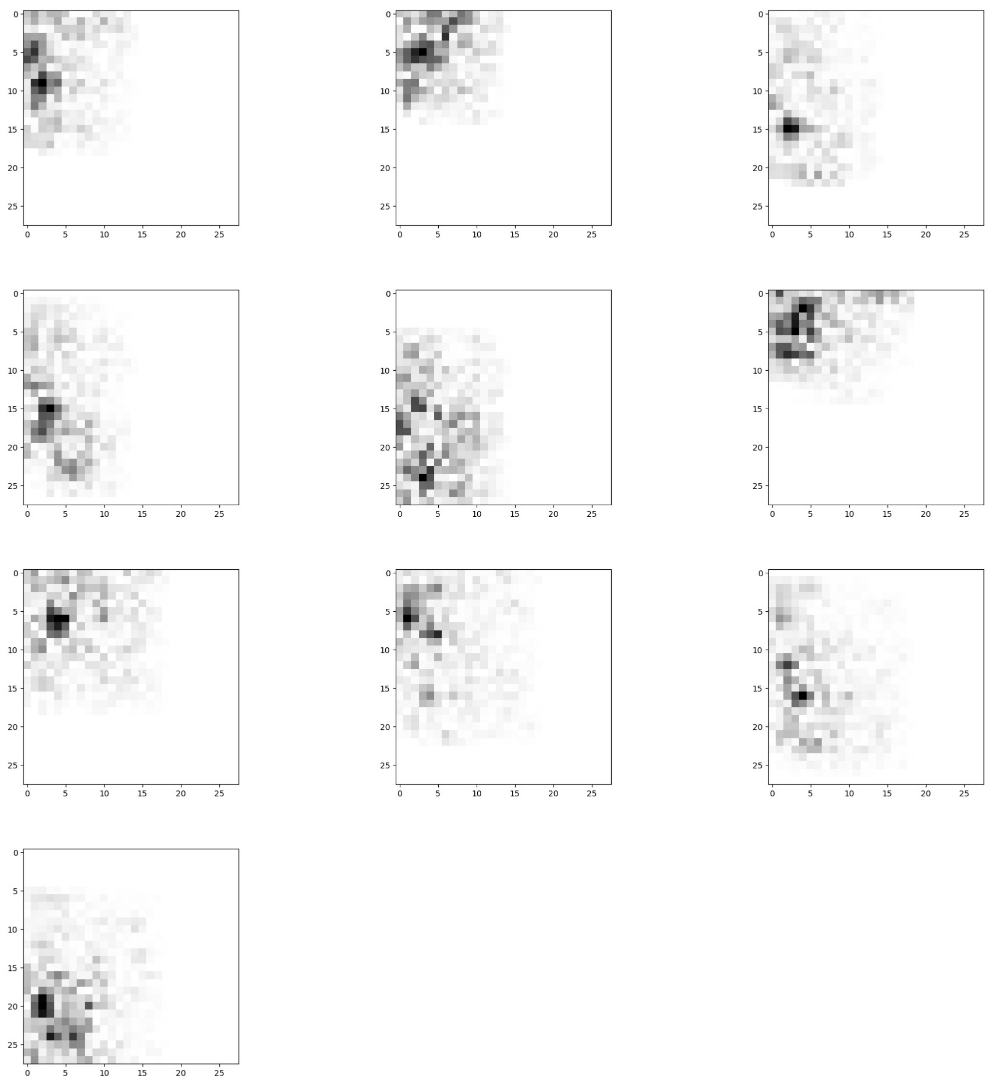

Figure 7: Guided Backpropagation

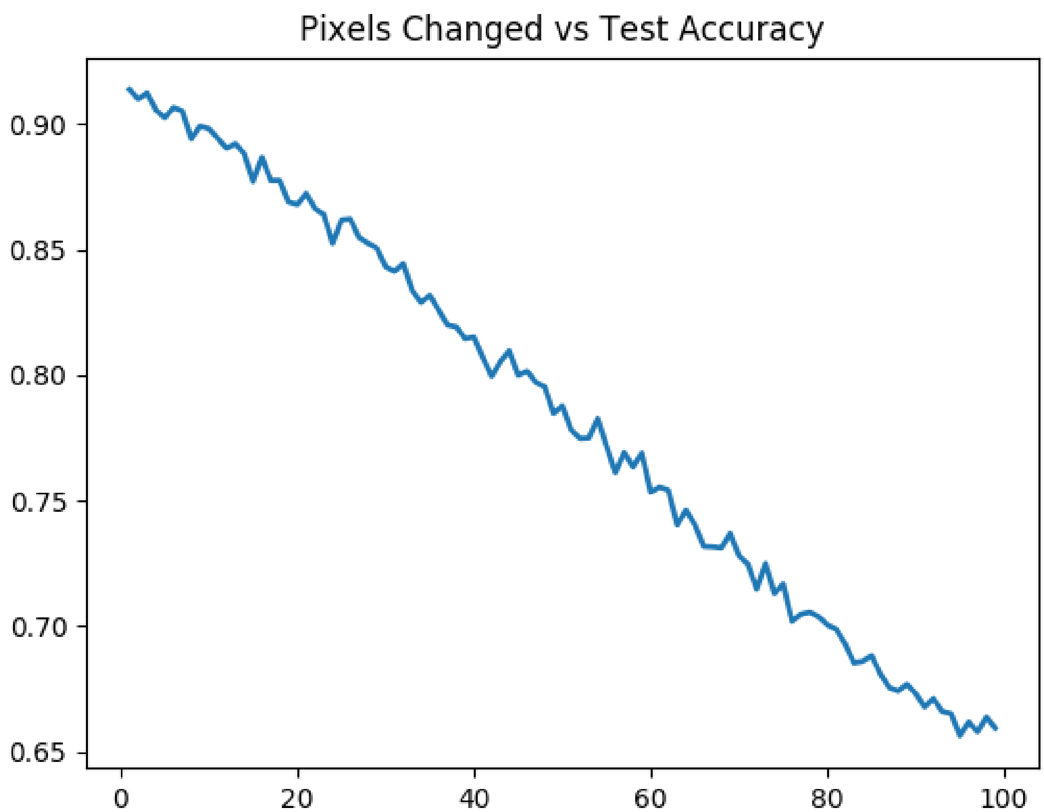

Figure 8: Fooling the Network