RBE/CS549: Homework 0

Mihir Kulkarni
MS, Robotics
Worcester Polytechnic Institute
Email: mmkulkarni@wpi.edu
Using 1 Late Day

***Abstract*—This homework is divided into 2 phases. Phase 1 explores the Probability of Boundary(Pb) boundary detection algorithm and its comparison with the well known and standard Canny and Sobel edge detection techniques. The Pb algorithm examines the brightness and texture information along with the intensity information which gives it an advantage over the Canny and Sobel techniques. Phase 2 involves designing and training our own neural network architecture using PyTorch. The CIFAR-10 dataset containing a total of 60,000 images is used for training and testing the model.**

I. PHASE-I: SHAKE MY BOUNDARY

The standard edge detection algorithms like Canny and Sobel utilise the intensity discontinuities of an image to determine an edge which is the classical method. On the other hand, the Pb technique uses texture, brightness and intensity data thereby improving the performance significantly. We will perform a few initial steps required to implement the Pb detection algorithm. Firstly, we start by creating 4 filter banks.

- Oriented Difference of Gaussian (DoG)
- Leung-Malik Filters
- Gabor Filters
- Half Disc Masks

A. Generating Filter Bank

The DoG filter is the difference of two Gaussian filters that can be applied to an image. This filter is generated by convolving the Sobel operator with the 2D Gaussian kernel. The result is then rotated to get the oriented versions of the filter collectively forming the filter bank. The generated 2*16 DoG filter bank is shown in the Figure 1. I have taken sigma values as [1,3] and 16 orientations of 22.5 degrees each.

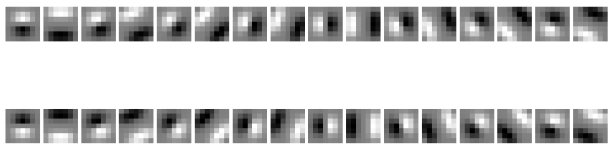

Fig. 1: DoG Filter Bank.

The generated LM filter bank is shown in the figure 2. It consists of 48 filters that are first and second order derivatives of Gaussians at various orientations and scales. The Gabor filter bank is generated when a Gaussian function is modulated by a sinusoidal plane wave. Shown in figure 3 is the generated Gabor filter bank. I used 3 different combinations of theta and omega to generate these filters. First 2 rows are for [theta,omega] = [1,0.4]. Next 2 rows are for [0.5, 0.3] and the last 2 rows are for [0.25,0.2].

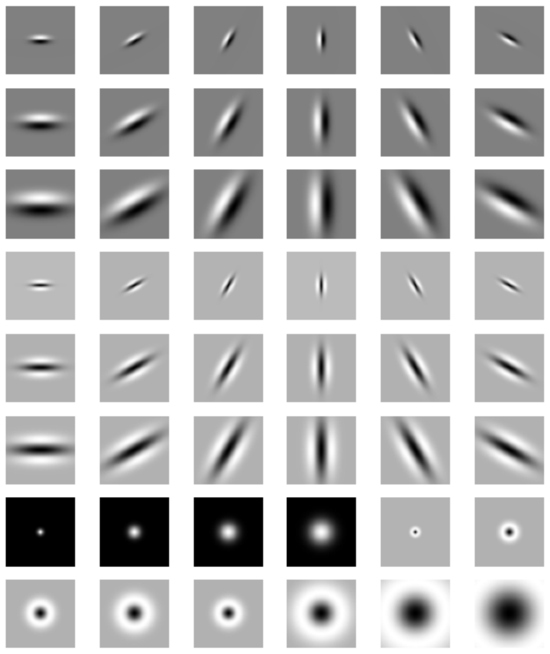

Fig. 2: LM Filter Bank.

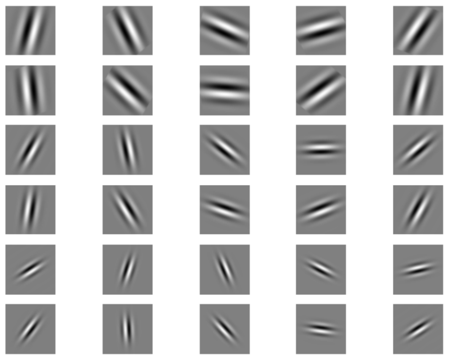

Fig. 3: Gabor Filter Bank.

Lastly, the Half Disk Mask filters are generated. We will be using these later on while calculating the Chi square distance. The following figure displays a total of 80 masks with 16 orientations and radii of [5, 10, 15].

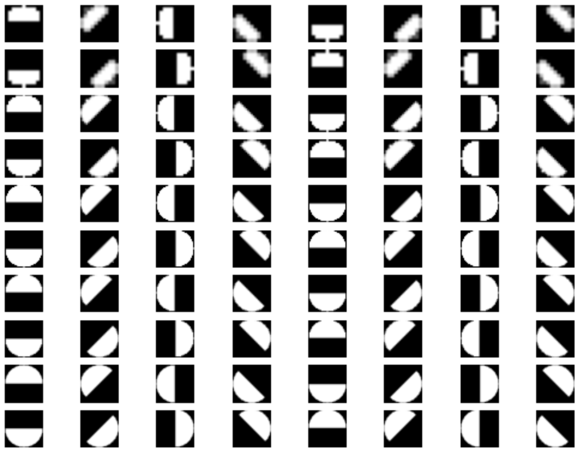
Fig. 4: Half Disk Masks.

B. Generating Texton, brightness, Color Maps

For the Texton map, I combined all the 3 filter banks (DoG, LM and Gabor) to produce a large combined filter bank. Convoluting these filters on the input image and performing K-Means clustering with K=64 and plotting the results gives us the following output in figure 5.

C. Generating Gradients for Texton, brightness, Color Maps

To calculate the gradients of the maps, we convolve the left and right half disk masks over the input image. These values of T_g, B and C_g indicate how the values of Texture, brightness and color are changing at a pixel respectively. The gradient turns out to be small if the distributions are similar and large if the distributions are dissimilar. The gradients plotted as an image are shown in the figure 6.

D. Generating the final PbLite output

To produce the PbLite output, we first need the filtered outputs from the standard Canny and Sobel baseline filters. In the final step, we combine the information using a simple equation as follows.

$$PbEdges = \frac{(T_g + B_g + C_g)}{3}(w_1 * cannyPb + w_2 * sobelPb) \tag{1}$$

The values of w_1 and w_2 can be varied accordingly. In my case, both values are 0.5. The output of the PbLite boundary detection filter along with the Canny and Sovel filter outputs is shown in figure 7 for all the 10 test images to give a better understanding of the filters and their comparison.

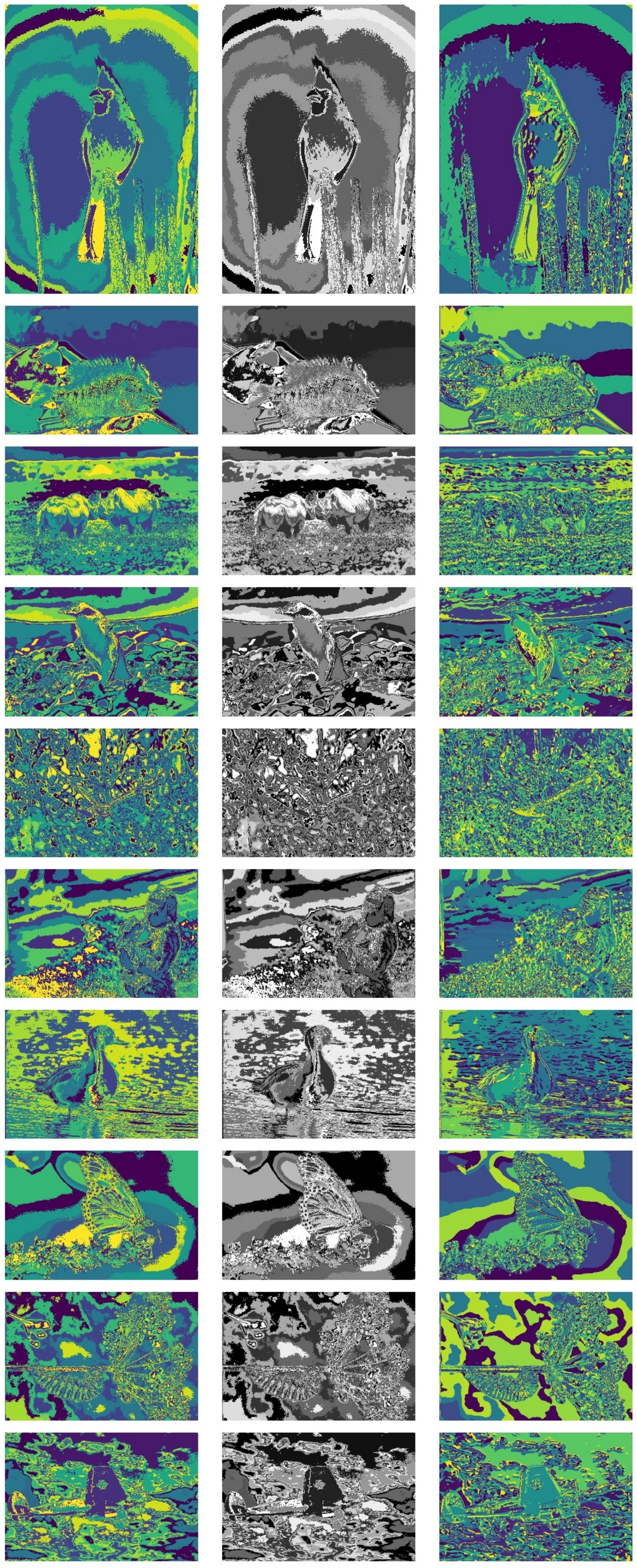
Fig. 5: T, B, C of Images 1 to 10 (Bottom to Top).

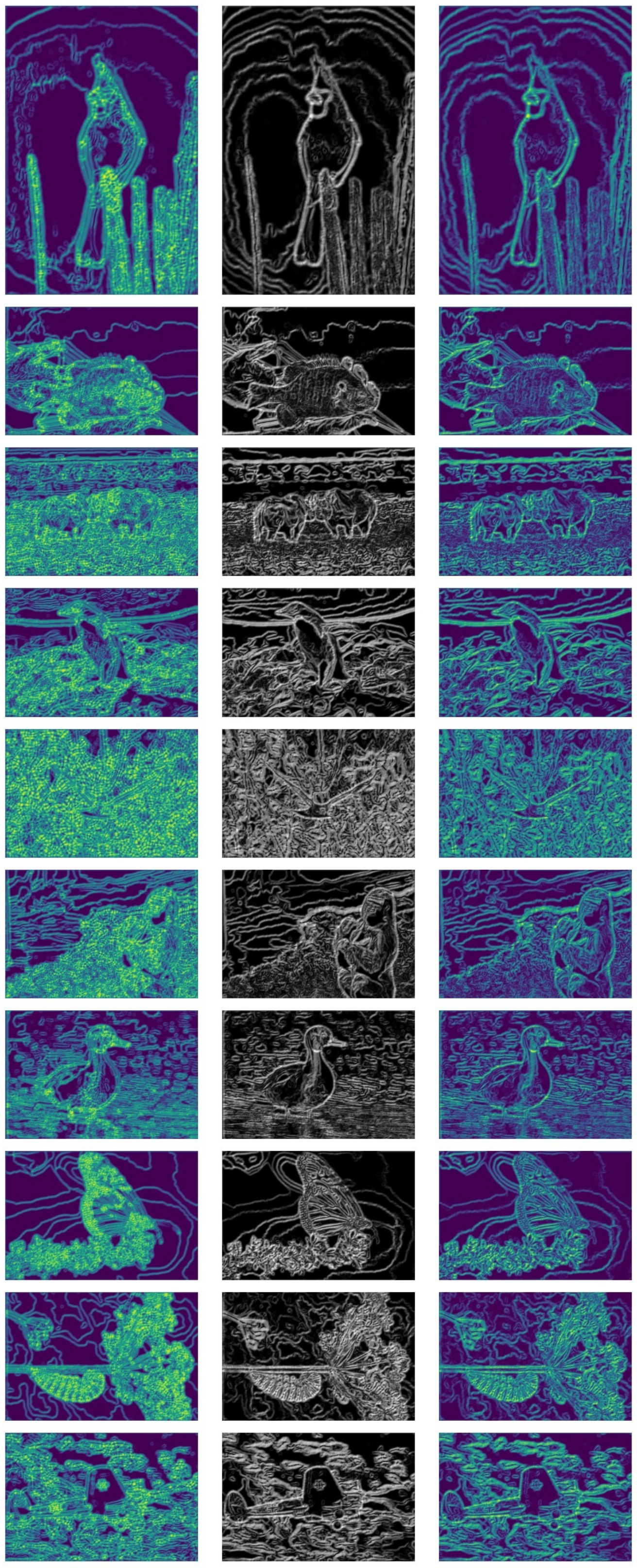
Fig. 6: T_g, B_g, C_g of Images 1 to 10.

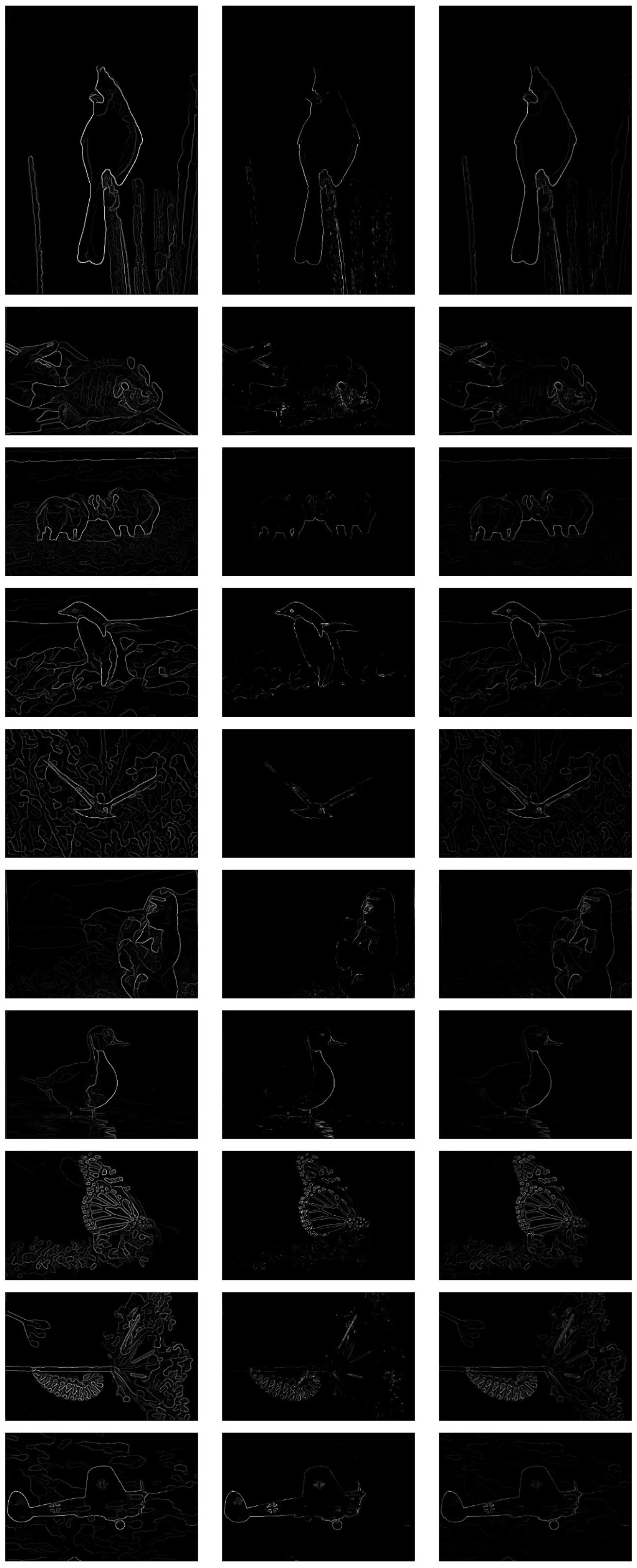
Fig. 7: Canny, Sobel and PbLite filters resp of Images 1 to 10.

E. Evaluating the results

For the images 10 and 9, we can see that PbLite image looks significantly better than the other 2 as all the required edges have been detected which is not the case with the Sobel filter. The PbLite also ignores the background edges slightly better than the Canny. For images 8 to 5 and 1, I feel the Sobel output images are better than the Canny and PbLite as it filters the background edges in a better way leaving just the subject in focus, although the edges do not appear to be as prominent. For images 4, 3 and 2, I feel PbLite does a better job.

Overall, we can see a regular pattern among the filters. The Canny tends to highlight all the edges very prominently and often preserves more information than needed. The Sobel appears to less highlight the edges in almost all the images. The PbLite gives the user control over how much texture, brightness or color information to be used to perform edge detection.

II. Phase-II: Deep Dive on Deep Learning

In this section, the task is to design and implement a custom CNN architecture using PyTorch to perform image classification. We will be using the popular CIFAR-10 dataset which contains a total of 60,000 images divided into 10 classes. In our case, we will be using 50,000 images as the training dataset and the remaining 10,000 images for testing.

A. First CNN Architecture

The first architecture I implemented is shown in the figure 8.

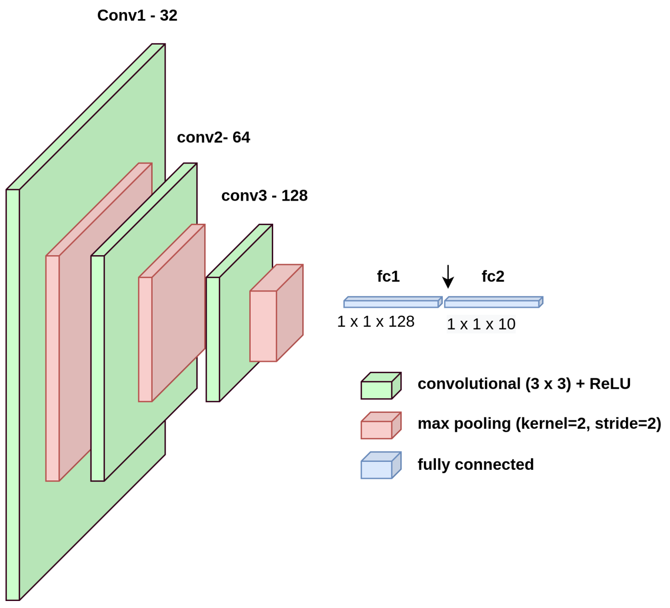

Fig. 8: First basic architecture.

This architecture consists of 3 convolution layers each followed by a ReLU and Max pooling layer of kernel size = 2 and stride = 2. Then there are 2 fully connected layers of outputs 128 and 10. The total number of parameters in this model is 160202.

1) Training:

1) Training: The parameters for training are constant and as follows for all the networks.

- Epochs: 50
- Batch size: 16
- Learning Rate: 0.0001
- Loss Function: Cross Entropy

I tried 2 optimizers namely Stochastic gradient descent (SGD) and ADAM for this model. Figure 9 shows the training loss and accuracy plotted over Epochs for the SGD optimizer and figure 10 shows the same when the Adam optimizer is used. The accuracy and loss comes out to be 61.34% and 1.21 respectively for the SGD optimizer. Whereas it is 93.76% and 0.31 respectively for the ADAM optimizer.

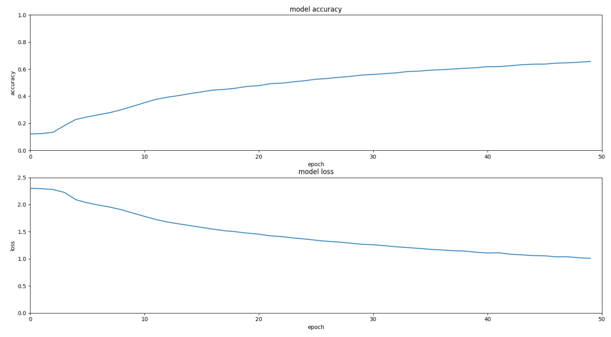

Fig. 9: Training Loss and Accuracy using SGD

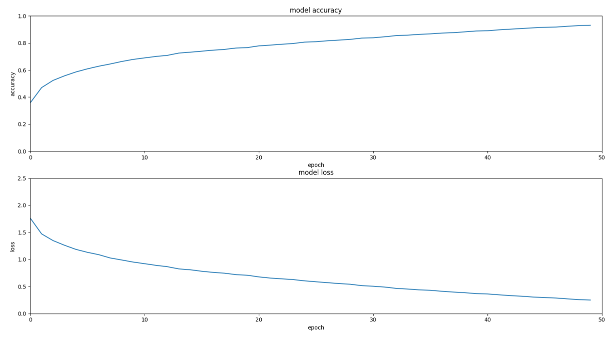

Fig. 10: Training Loss and Accuracy using ADAM

2) Testing: The testing accuracy and confusion matrix for the model for the SGD optimizer is shown in figures 11 and 12 respectively. Whereas the testing accuracy and confusion matrix for the ADAM optimizer is plotted in figures 13 and 14 respectively. The testing accuracy is 62.32% for the SGD and 69.57% for the ADAM.

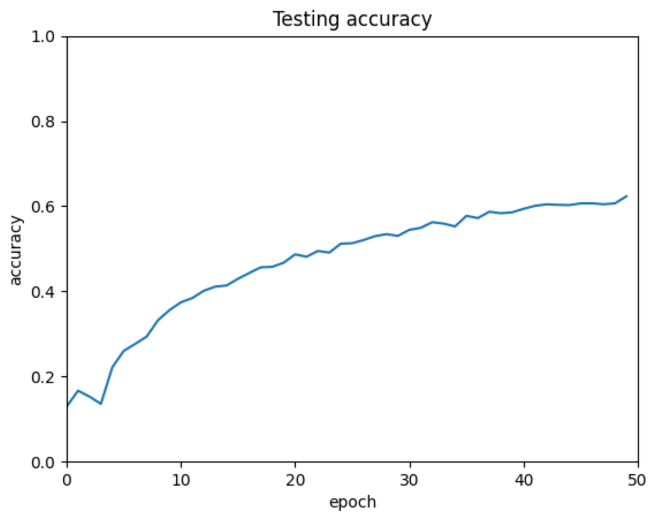

Fig. 11: Testing Accuracy with SGD optimizer

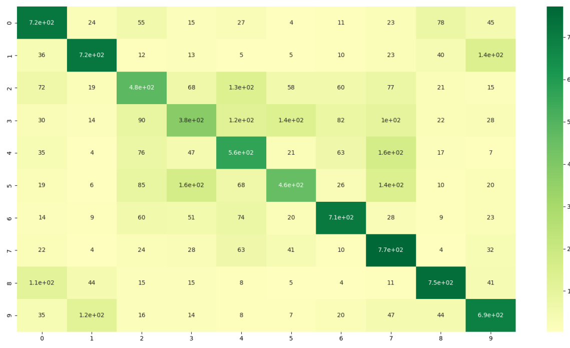

Fig. 12: Confusion matrix for SGD optimizer

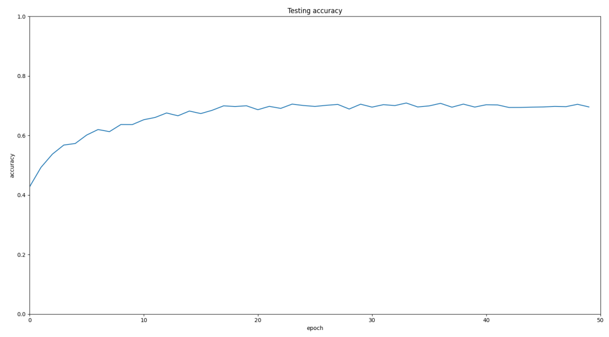

Fig. 13: Testing Accuracy with ADAM optimizer

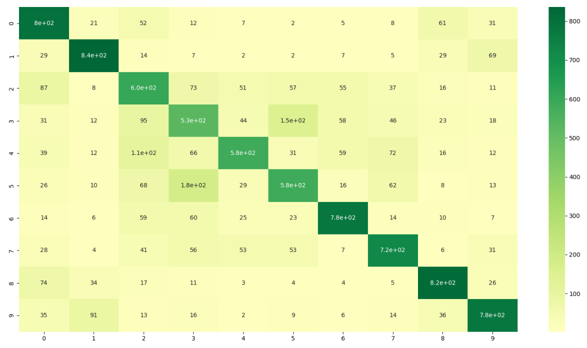

Fig. 14: Confusion matrix for ADAM optimizer

B. New CNN Architecture

In this new architecture, I added a convolution layer making a total of 4, in a pair of 2 consecutively. I also added a batch normalization layer after each pair of convolution layers. At the end, another fully connected layer is added to the model architecture. The total number of parameters in this model is 509802. The diagram of this CNN model is shown in figure 15.

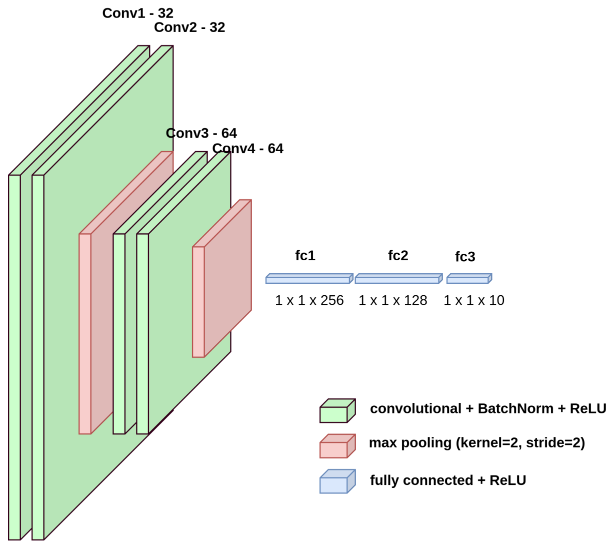

Fig. 15: New improved Architecture

1) Training: The parameters used in the earlier model are kept as it is while training this model. As done previously, I have experimented for 2 different optimizers here as well i.e. SGD and ADAM. The training loss and accuracy plotted over Epochs for the SGD optimizer is shown in figure 16 and figure 17 shows the same when the Adam optimizer is used. The accuracy and loss comes out to be 92.54% and 0.44 respectively for the SGD optimizer. Whereas it is 95.26% and 0.32 respectively for the ADAM optimizer.

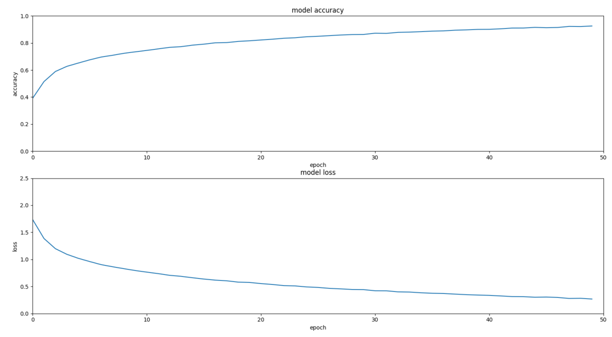

Fig. 16: Training Loss and Accuracy using SGD

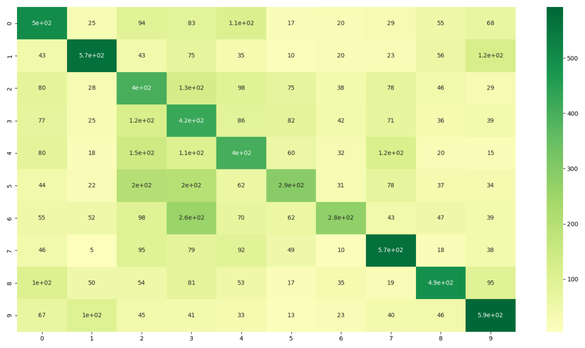

Fig. 19: Confusion matrix for SGD optimizer

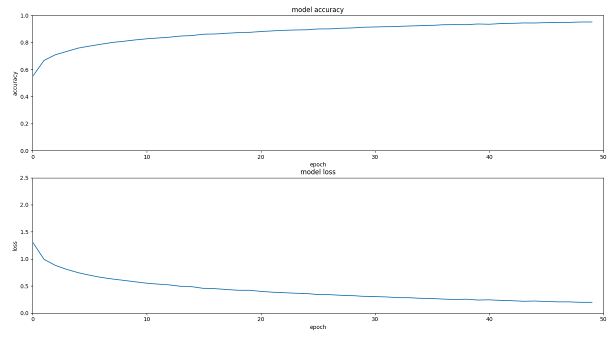

Fig. 17: Training Loss and Accuracy using ADAM

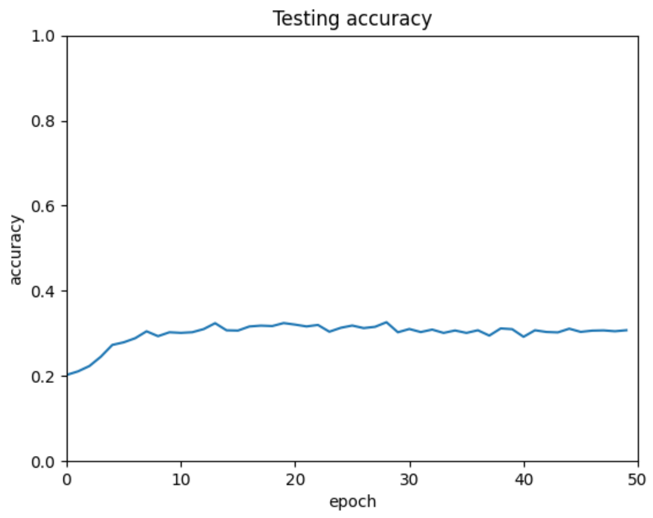

Fig. 20: Testing Accuracy with ADAM optimizer

2) *Testing:* The testing accuracy and confusion matrix for the model for the SGD optimizer is shown in figures 18 and 19 respectively. Whereas the testing accuracy and confusion matrix for the ADAM optimizer is plotted in figures 20 and 21 respectively. The testing accuracy is 43.21% for the SGD and 36.57% for the ADAM.

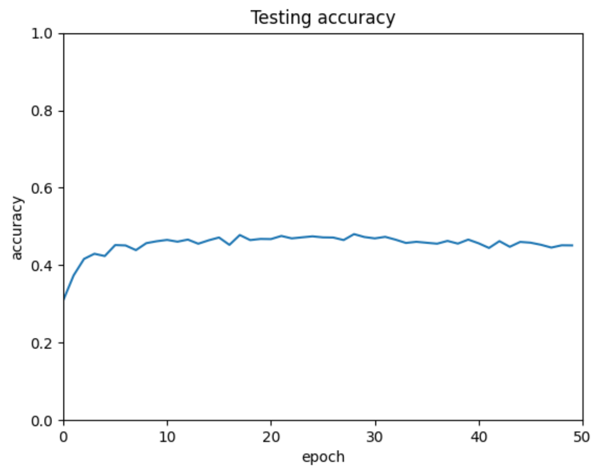

Fig. 18: Testing Accuracy with SGD optimizer

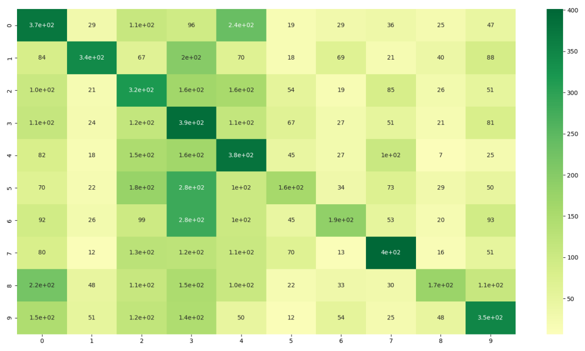

Fig. 21: Confusion matrix for ADAM optimizer

C. Evaluating the results

We can infer from the loss and accuracy plots of $Training$, that the first CNN architecture has relatively less training accuracy as compared to the New CNN architecture when using the SGD optimizer and almost same when using the ADAM optimizer. The same can be said w.r.t the training loss. the new architecture performs better as the loss falls down to a lower value whereas the value is comparatively high for the first architecture using the SGD optimizer.

For the $Testing$ part, the first architecture has a increasing curve for the testing accuracy although the accuracy is not the best. Whereas the new architecture performs extremely poorly in the testing part with a flat accuracy curve that only increases for the initial $\tilde{5}$ epochs.

This indicates that the New architecture model has been **Overfit** largely and is of little use. Overall, the First architecture with the ADAM optimizer seems to perform the best out of all the architecture and optimizer combinations.