Computer Vision - Homework 0 - Alohomora

Muhammad Sultan
Robotics Engineering
msultan@wpi.edu
Used one late day for this assignment

I. Phase 1: Shake My Boundary

In Phase 1, our goal was to implement a simplified version of the Probability of Boundary (pb) algorithm called pb lite. This algorithm computes the probability of a pixel being a part of a boundary between two object, hence the name. Classical edge detection algorithms like Canny and Sobel only use the gradients of intensity values, however, pb uses discontinuities in texture, brightness, and color of the image as well. The steps in the implementation of pb lite in this homework are generating filter banks, generating texton maps, color maps and brightness maps, generating gradient maps, and lastly combining gradient maps with canny and sobel baselines to generate final edges.

A. Step 1: Generating Filter Banks

A total of three filter banks were generated, namely, oriented Derivative of Gaussian (DoG) filter bank, Leung-Malik (LM) filter bank (both Small and Large), and Gabor filter bank.

The **DoG** filters were obtained by convolving a Gaussian kernel with a sobel operator to get approximations of the partial derivatives of the Gaussian kernel with respect to x and y. These partial derivatives were than combined to get a Derivative of Gaussian filter. This filter was then rotated through different angles to generate more filters for the filter bank. This bank is shown in Fig 1.

The small and large **LM banks** were made using the same algorithm but different scales. The LM bank has four different kinds of filters. The first kind is just the DoG filters but with standard deviation in the y-direction three times more than in the x-direction. The second type of filters are the second order derivative of Gaussian filters, which are obtained by convolving the first order partial derivatives with the sobel kernels again. The third type are simple Gaussian filters, and the forth type are Laplacian of Gaussian filters (LOG). LOG filters are obtained by convolving a gaussian filter with a Laplacian kernel which gives an approximation of the Laplacian when convolved with. LM filters are shown in Fig 2.

Gabor filters are just gaussian filters modulated by a sinosoidal wave. These filters are shown in Fig 3.

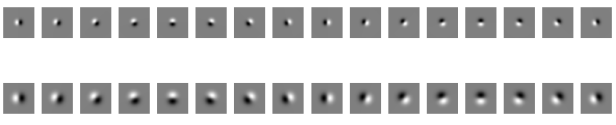

Fig. 1: Oriented Derivative of Gaussian Filter Bank

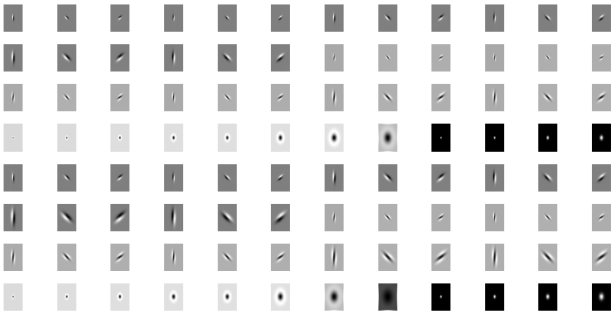

Fig. 2: Leung-Malik filter bank (Upper half is LM Small and lower half is LM Large

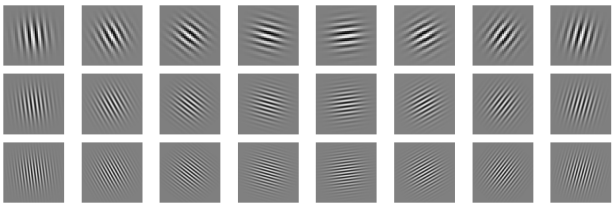

Fig. 3: Gabor filter bank

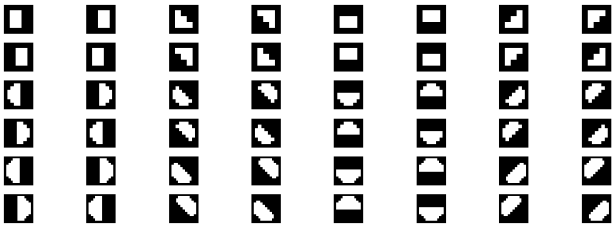

Fig. 4: Half-disk mask pairs

B. Step 2: Generating Texton, Brightness, and Color Maps

In this step, we first compute the texton maps for each image. To do this we start by applying each filter generated in the previous step to an image separately and then combining the filtered images into one. This results in an $mxnxN$ matrix, each element of which is an N-dimensional vector corresponding to a single pixel. K-means clustering is then applied to this matrix to compute the Texton map. K-means clustering replaces each N-dimensional vector with a discrete texton ID so the final image is a single channel image with values ranging from **1** to **K**. The suggested value of **64** was chosen for **K**.

The Brightness Maps are computed by converting the original image to grey-scale and performing K-means clustering on it.

The Color Maps are computed by performing K-means clustering on the original images themselves. For both Color and Brightness maps a value of **16** is chosen for **K**. These maps are shown in Figures 5 - 14 (Left: Texton Map, Right: Brightness Map, Bottom: Color Map).

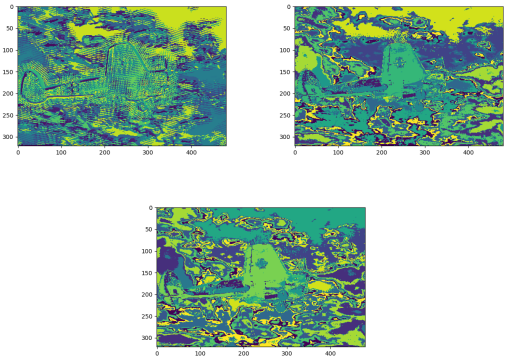

Fig. 5: Image 1: Texton, Brightness and Color Map

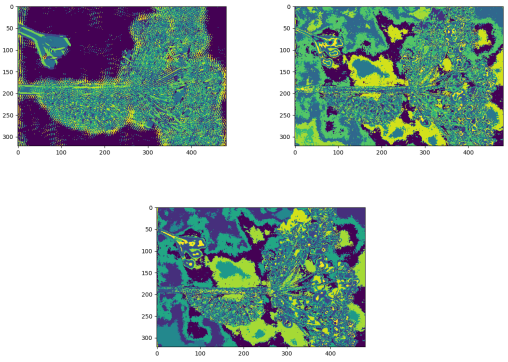

Fig. 6: Image 2: Texton, Brightness and Color Map

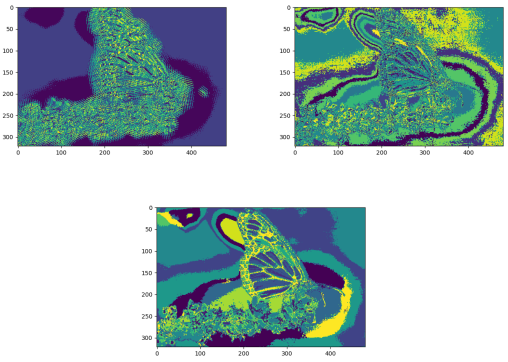

Fig. 7: Image 3: Texton, Brightness and Color Map

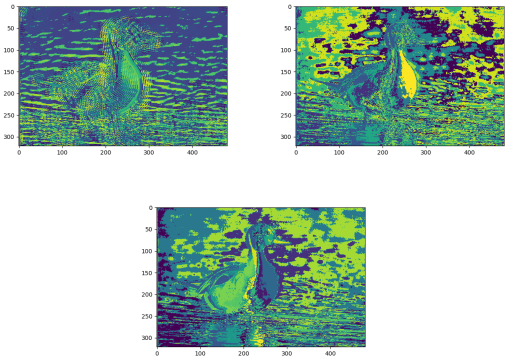

Fig. 8: Image 4: Texton, Brightness and Color Map

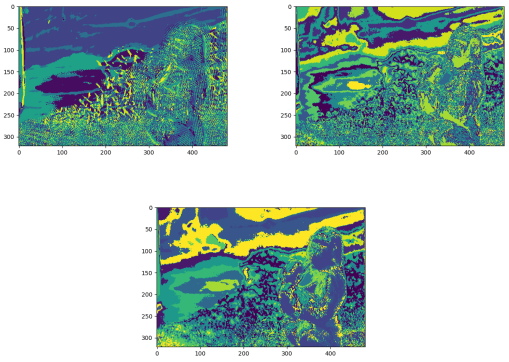

Fig. 9: Image 5: Texton, Brightness and Color Map

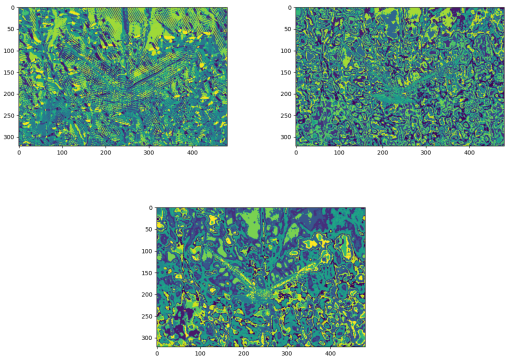

Fig. 10: Image 6: Texton, Brightness and Color Map

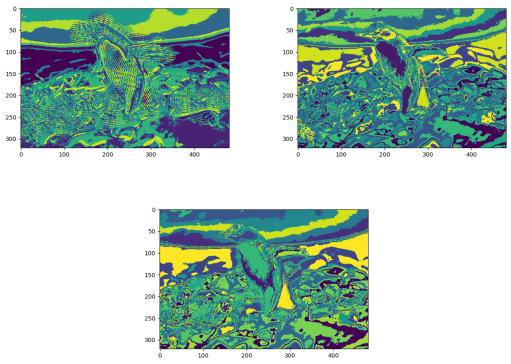

Fig. 11: Image 7: Texton, Brightness and Color Map

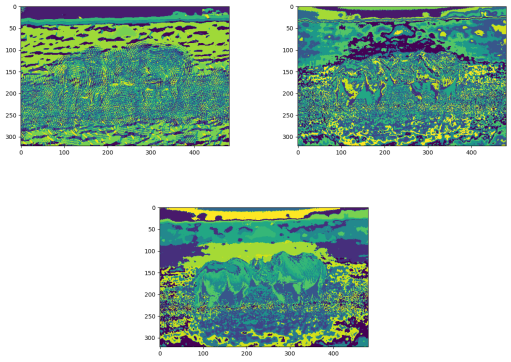

Fig. 12: Image 8: Texton, Brightness and Color Map

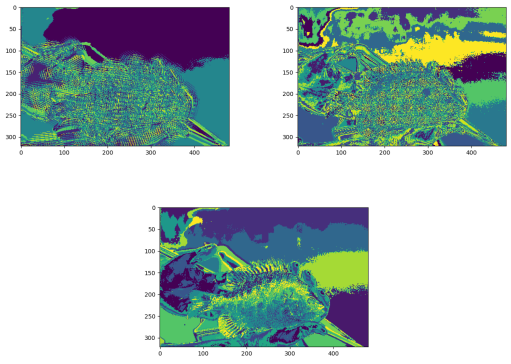

Fig. 13: Image 9: Texton, Brightness and Color Map

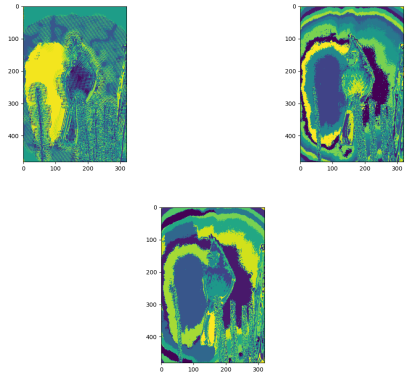

Fig. 14: Image 10: Texton, Brightness and Color Map

C. Step 3: Generating Texton, Brightness, and Color Gradient Maps

The gradient maps are computed using the Texton, Brightness, and Color maps obtained in step 2. For this, we first generate half-disk masks, which we use to calculate chi-square distances (χ^2) for different orientations and scales of the half-disk masks (shown in Fig 4). These (χ^2) distances are then aggregated together to get the gradient map of the input (any of the three maps). These gradient maps are shown in Figures 15 - 24 (Left: Texton Gradient Map, Right: Brightness Gradient Map, Bottom: Color Gradient Map).

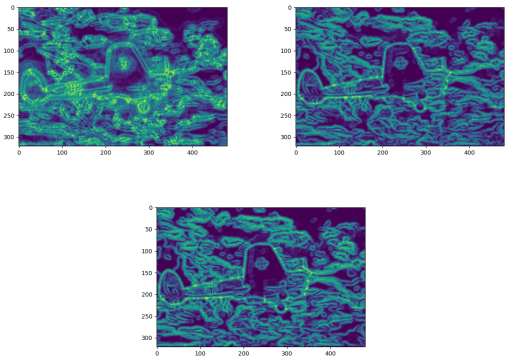

Fig. 15: Image 1: Texton, Brightness and Color Map Gradients

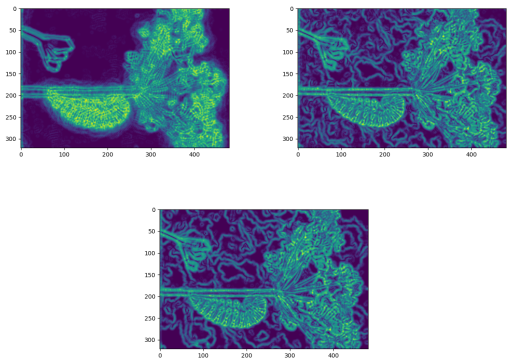

Fig. 16: Image 2: Texton, Brightness and Color Map Gradients

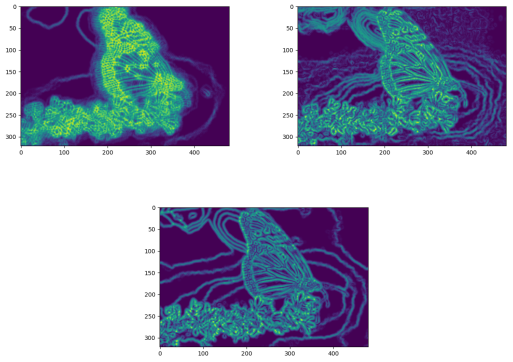

Fig. 17: Image 3: Texton, Brightness and Color Map Gradients

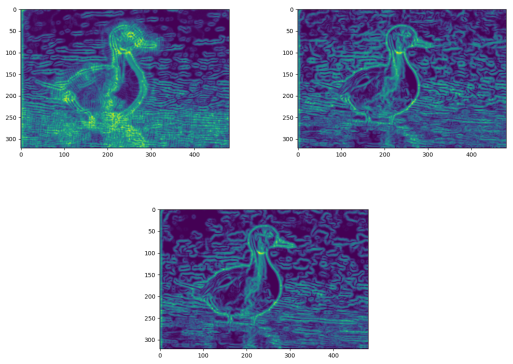

Fig. 18: Image 4: Texton, Brightness and Color Map Gradients

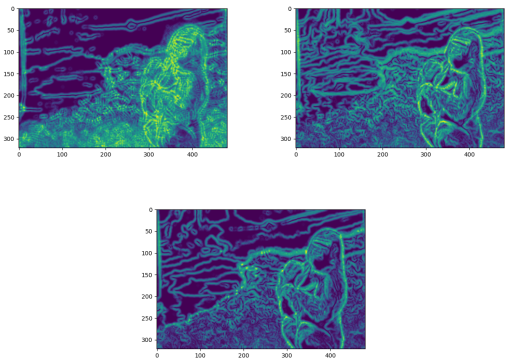

Fig. 19: Image 5: Texton, Brightness and Color Map Gradients

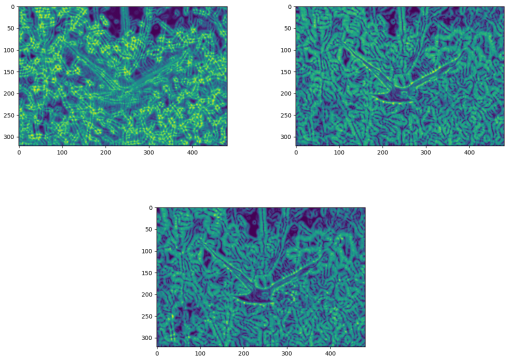

Fig. 20: Image 6: Texton, Brightness and Color Map Gradients

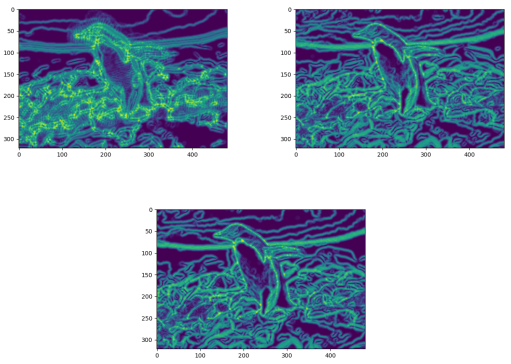

Fig. 21: Image 7: Texton, Brightness and Color Map Gradients

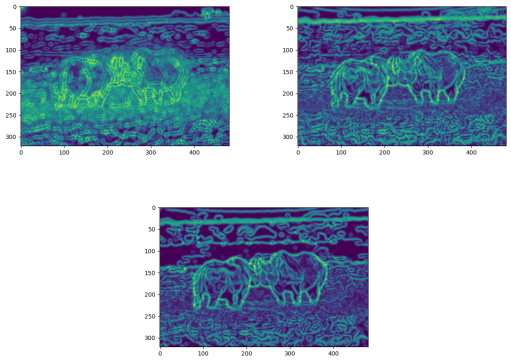

Fig. 22: Image 8: Texton, Brightness and Color Map Gradients

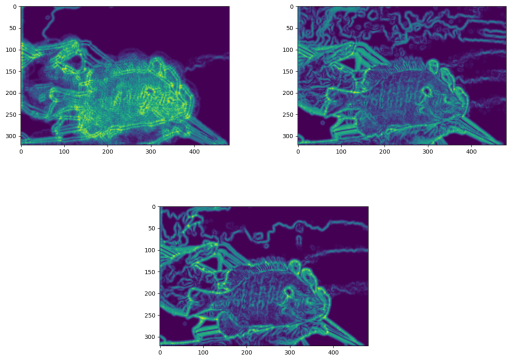

Fig. 23: Image 9: Texton, Brightness and Color Map Gradients

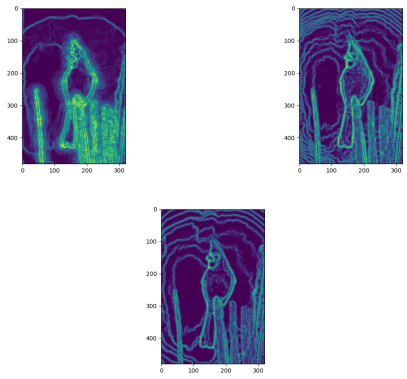

Fig. 24: Image 10: Texton, Brightness and Color Map Gradients

D. Step 4: Edge Detection

The final step is to combine the three gradient maps by averaging them together and computing the Hadamard product with a weighted sum of the canny and sobel baseline images. This gives us the final pb lite output images. The formula used is

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

. I chose w_1 as 0.8 and w_2 as 0.2. These images are shown in Figures 15 - 34 (Left: Canny Baseline, Middle: Sobel Baseline, Right: Pb-lite result)

Fig. 25: Image 1: Canny, Sobel and Pb-lite

Fig. 26: Image 2: Canny, Sobel and Pb-lite

Fig. 27: Image 3: Canny, Sobel and Pb-lite

Fig. 28: Image 4: Canny, Sobel and Pb-lite

Fig. 29: Image 5: Canny, Sobel and Pb-lite

Fig. 30: Image 6: Canny, Sobel and Pb-lite

Fig. 31: Image 7: Canny, Sobel and Pb-lite

Fig. 32: Image 8: Canny, Sobel and Pb-lite

Fig. 33: Image 9: Canny, Sobel and Pb-lite

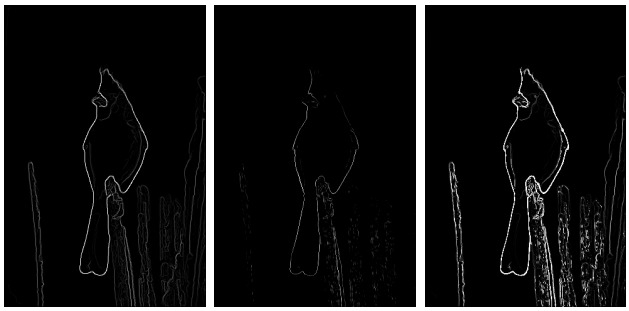

Fig. 34: Image 10: Canny, Sobel and Pb-lite

E. Analysis

As apparent from the final results, the pb-lite algorithm brought out a lot of the edges that were not detected in the canny or sobel baselines. However, in my implementation, it seems that the false positives have not been decreased. Theoretically, pb-lite should also reduce false positives from the images to only have important lines. I suspect this is due to sub-optimal parameters used during the algorithm like filter bank size, filter size, filter orientations, filter scales, etc. However, we can still see that in the pb-lite results, important edges have not been ignored. Aside from my implementation, PbLite is excellent on discerning the main object which is subject to detection from the background, hence it is better than canny in that regard. When comapred to sobel, it does not miss out on edges of the object like sobel does. So, it is a best of both worlds in a sense.

II. Phase 2 - Deep Dive on Deep Learning

In phase 2, we have implemented 4 different types of neural networks on the CIPHAR-10 dataset given to us and compared their results. The models include a basic CNN, a ResNet, a ResNext, and a DenseNet architecture.

A. Basic CNN

For the basic CNN, no pre-processing was done on the pictures before training. I implemented three convolutional layers. For each convolutional layer, I further applied ReLU activation to their output and then pass them through a Batch normalization layer. Finally, I reshaped the output of the third convolutional layer and passed it into 2 fully connected layers, one after the other. The Model architecture is shown in Fig **??**.

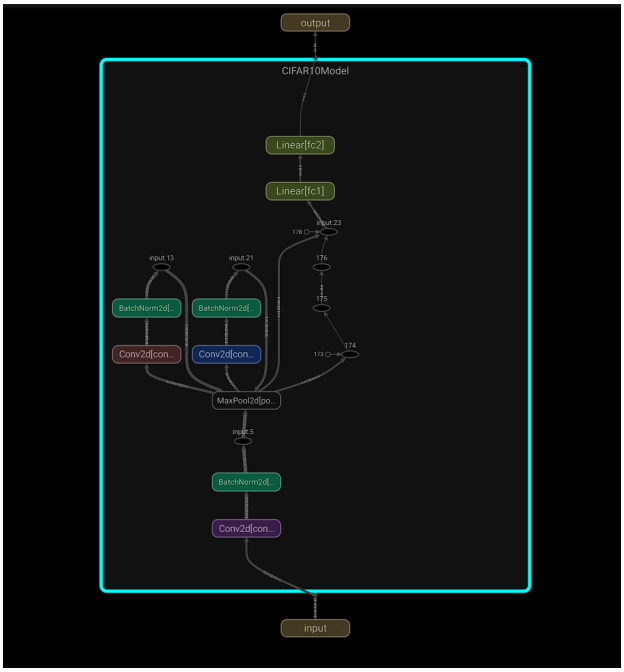

Fig. 35: Basic CNN: Architecture

This model was trained using a batch size of 32 over 16 epochs. The number of parameters of this model were **1147914**. The CNN gave 97% accuracy on the training set, and only 61% accuracy on the test set, which highlights an over-fitting problem. The plots of accuracy and loss vs epochs are shown in Fig 59. The plot of test accuracy vs epochs is shown in Fig 60. The confusion matrices for the test and training set are shown in Fig 61 and Fig 62, respectively.

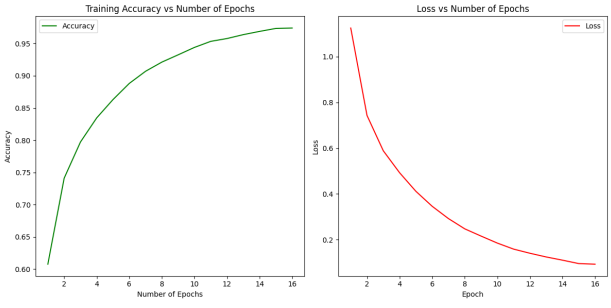

Fig. 36: Basic CNN: Plot of Training accuracy and loss over each epoch

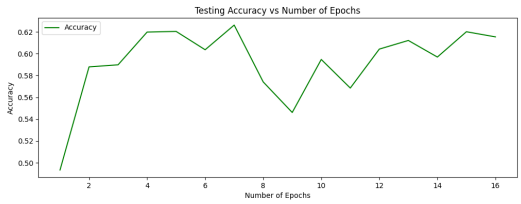

Fig. 37: Basic CNN: Plot of Test accuracy over each epoch

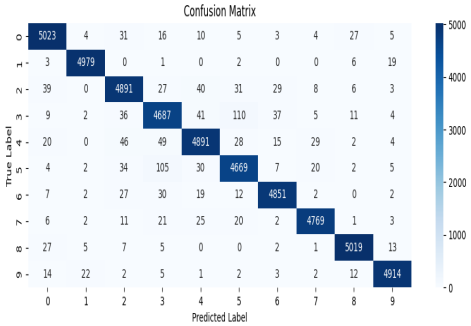

Fig. 38: Basic CNN: Confusing matrix for training data

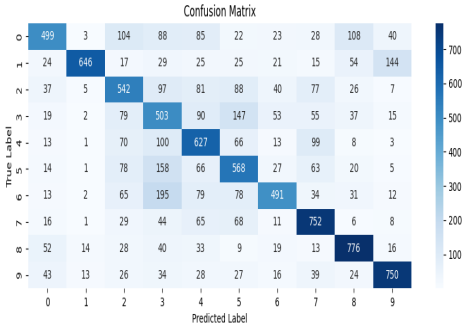

Fig. 39: Basic CNN: Confusing matrix for training data

This same model was attempted to be improved (to Basic CNN 2.0), by adding pre-processing of the images by applying transformations, including normalization, to the images. The number of epochs was kept 16 and batch size was change to 64. However, this actually resulted in a drop in accuracy over both training and testing set. The results for this model are shown in Figures 40 - 43.

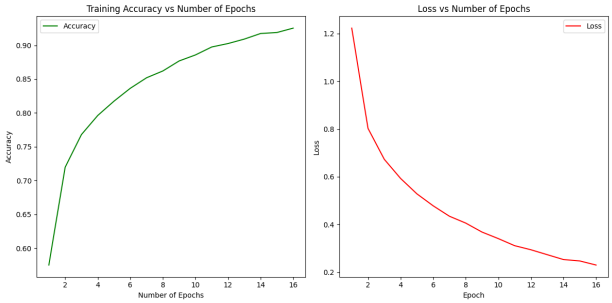

Fig. 40: Basic CNN 2.0: Plot of Training accuracy and loss over each epoch

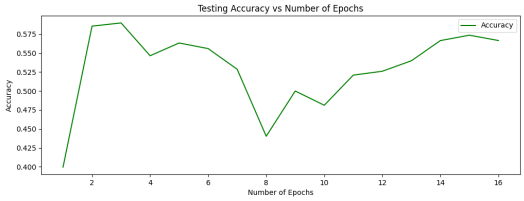

Fig. 41: Basic CNN 2.0: Plot of Test accuracy over each epoch

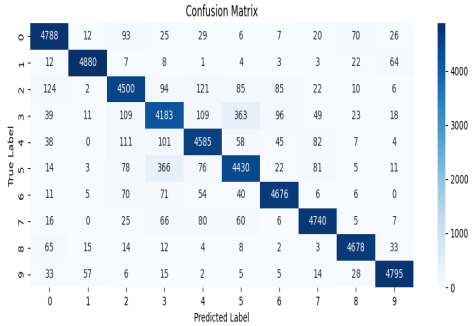

Fig. 42: Basic CNN 2.0: Confusing matrix for training data

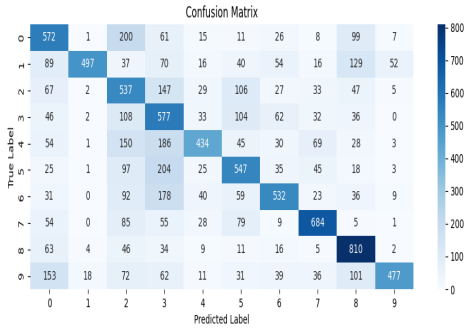

Fig. 43: Basic CNN 2.0: Confusing matrix for training data

Overall, the results who that the model is overfitting, since it is excellent at classifiying training images but not as good at testing images.

B. ResNet

My ResNet implementation is very basic (for faster computation). It consists of a convolutional layer, followed by three layers, each having two Residual Blocks in them. Each Residual Block has two convolutional layers with batch normalization and ReLU activation. Whenever the stride of the residual block is not 1 or the number of input and output channels of the block are different, it adds a downsampled input (downsampling done on the input that the block got) to the output of the block. This is the residual connection, and it helps address the vanishing gradient problem. The model architecture is shown in Figures 44 and 45.

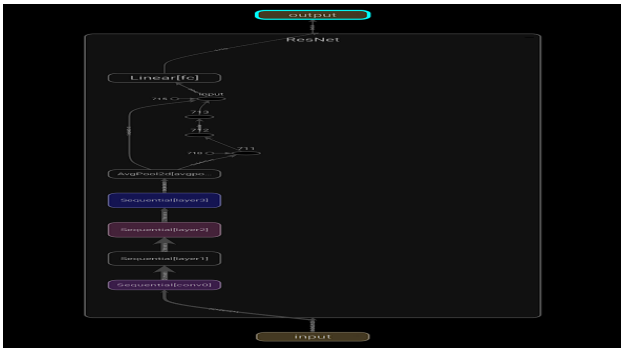

Fig. 44: ResNet: Model Architecture

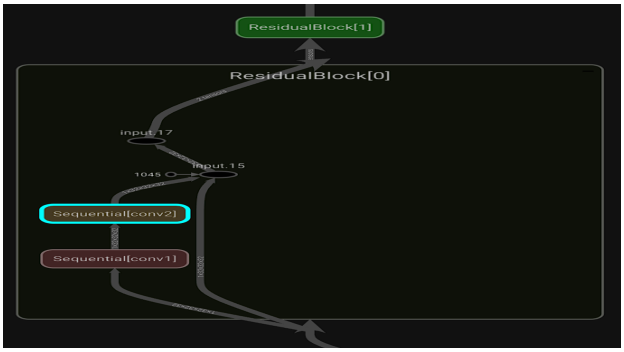

Fig. 45: ResNet: Residual Block

This model was trained for 16 epochs with a batch size of 64. The number of parameter of this model are **342026**. The results are not very good for this model. It has an even more severe overfitting problem than the Basic CNN. The training accuracy is 95% but the testing accuracy is only 38%. These results are shown in Figures 46 - 49.

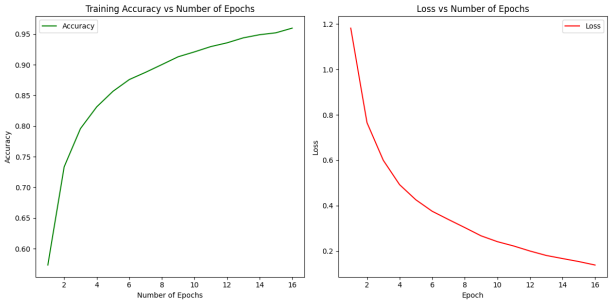

Fig. 46: ResNet: Plot of Training accuracy and loss over each epoch

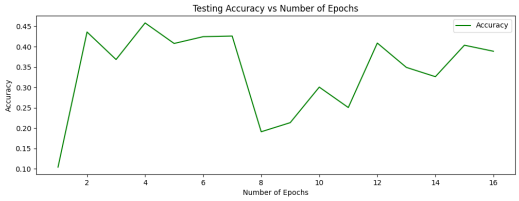

Fig. 47: ResNet: Plot of Test accuracy over each epoch

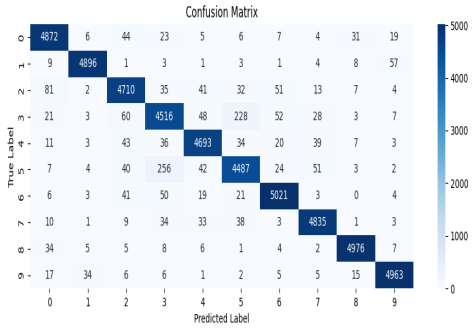

Fig. 48: ResNet: Confusing matrix for training data

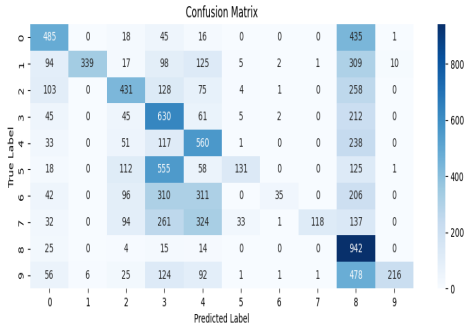

Fig. 49: ResNet: Confusing matrix for training data

C. ResNext

For the ResNext, I just modified my ResNet model to make it fit the description of a ResNext (added cardinality), and added a few extra layers to the residual block. Cardinality is

basically the number of parallel paths in a Residual Block. Combining information from multiple pathways enables diverse feature learning. Hence, it is an improvememnt on ResNet.

The cardinality was set to 32, the number of epochs were 16 and the batch size was 64. The number of parameters of this model are **2438036**. The model architecture is shown in Figures 50 and 51.

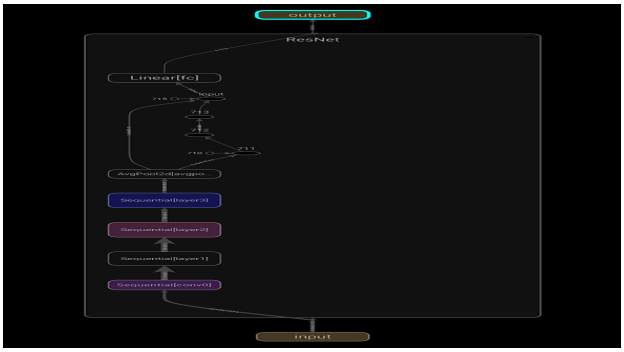

Fig. 50: ResNext: Model Architecture

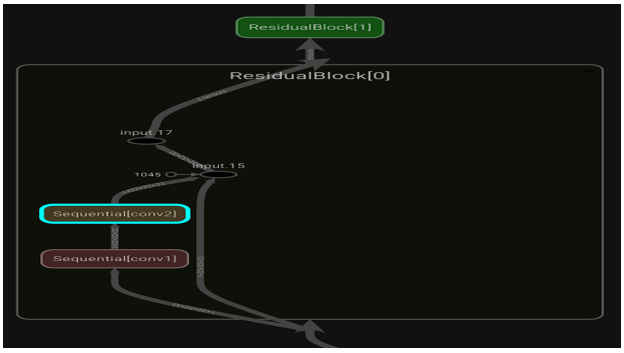

Fig. 51: ResNext: Residual Block

The results for this model actually got worse, even though they were meant to improve the results of the ResNet. It has a training accuracy of 98% but a testing accuracy of only 25%. The results are shown in Figures 52 - 55.

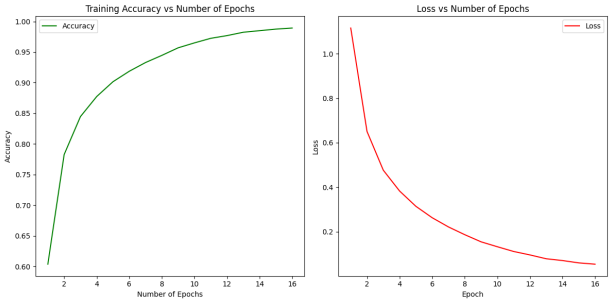

Fig. 52: ResNext: Plot of Training accuracy and loss over each epoch

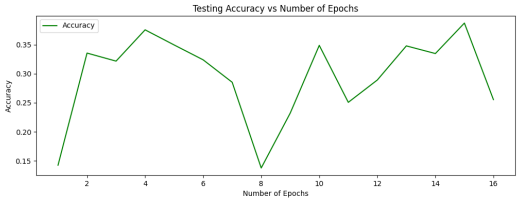

Fig. 53: ResNext: Plot of Test accuracy over each epoch

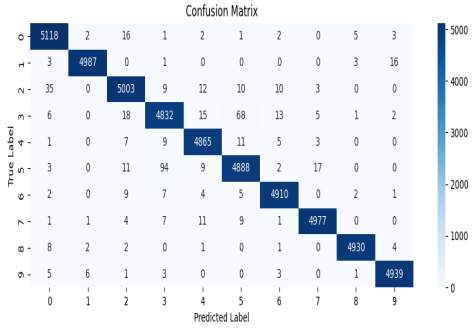

Fig. 54: ResNext: Confusing matrix for training data

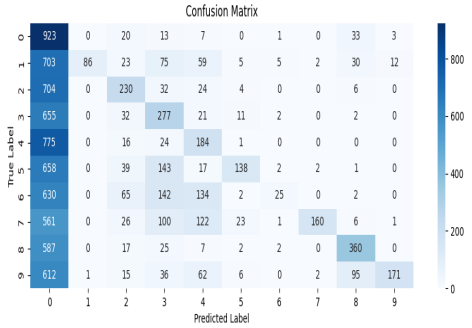

Fig. 55: ResNext: Confusing matrix for training data

D. DenseNet

The final model implemented was DenseNet, which has two types of blocks, a dense block and a tranisition block. In Dense blocks each layer receives input from all previous layer. These Dense blocks increase the number of channels making the model too complex. The Transition blocks are there to control the complexity of the model by reducing the number of channels by applying a 1 x 1 convolution. The architecture of the model is shown in Figures 56, 57 and 58.

Fig. 56: DenseNet: Model Architecture

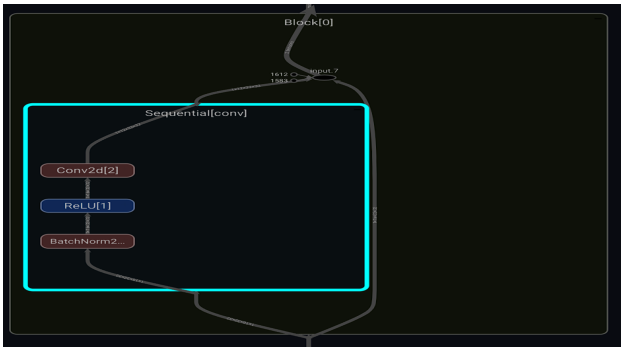

Fig. 57: DenseNet: Dense Block

Fig. 58: DenseNet: Transiton Block

The number of parameters for this model are **35661**. This model was trained using a a growth rate (number of channels added to the input of a subsequent layer in a dense block) of 12 , batch size 0f 64, and 16 epochs. The results for DenseNet are not good either, as it suffers even more from overfitting. It has a training accuracy of 82% but a testing accuracy of only 25%. These results are shown in Figures .

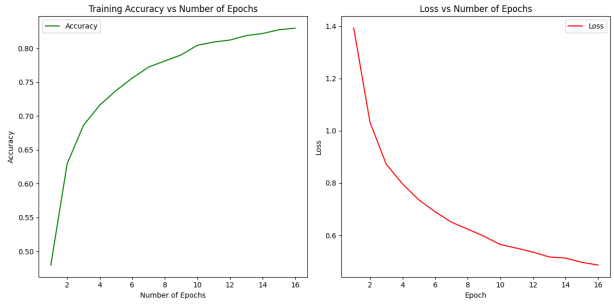

Fig. 59: DenseNet: Plot of Training accuracy and loss over each epoch

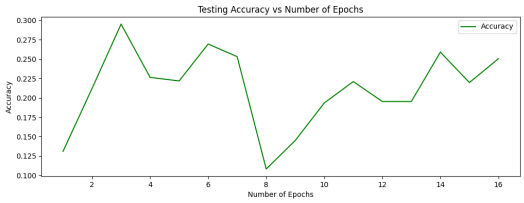

Fig. 60: DenseNet: Plot of Test accuracy over each epoch

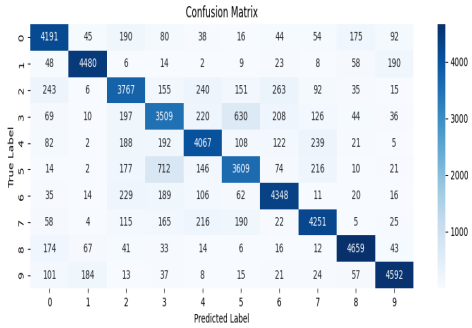

Fig. 61: DenseNet: Confusing matrix for training data

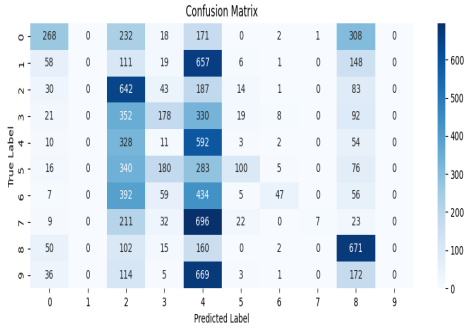

Fig. 62: DenseNet: Confusing matrix for training data

E. Analysis

The final comparison of each model is shown in the Table I.

Model	Train Accuracy	Test Accuracy	Number of parameters
Basic CNN	97%	61%	1147914
Basic CNN 2.0	92%	56%	1147914
ResNet	95%	38%	342026
ResNeXt	98%	25%	2438026
DenseNet	82%	25%	35661

TABLE I: Accuracy comparison of Neural Nets

Overall, with my implementation the Basic CNN performed the best even tough it should have performed the worst. From the results, we can see that as we increase the complexity of the model, the testing accuracy decreases, which makes sense (in a way). Models that are more complex can fit the training data very closely, as compared to simpler models which could result in even more overfitting. Another possible reason for this could be sub-optimal hyper-parameters. It seems like these models need to be tuned a lot better than I have attempted to. With these things in mind, the performance of these models can be improved significantly. Model architectures also need to be further improved and could be the reason this is happening in the first place.