Homework 0 - Alohomora

Uday Sankar
Email: usankar@wpi.edu
Using 1 late day

I. PHASE 1: SHAKE MY BOUNDARY

The objective of this section is to implement the pb-lite boundary detection algorithm in which 'pb' stands for Probability of Boundary. Its called so because the process returns the probability of each pixel of the input image being part of an edge. In this method, in addition to gradients of intensity values, discontinuities in texture and color of the image is also considered for the purpose of detecting edges. This process can be divided into four steps:

1) Generation of filter bank
2) Generation of texton, brightness and color maps
3) Generation of texton, brightness and color gradient maps
4) Boundary detection using the maps, Sobel and Canny baselines

A. Generation of Filter Bank

In order to get the texture information from the images, a number of different filters, collected into a filter bank is applied on to the images. The filters applied may be of different scales and orientations. Three main kinds of filters are used in this phase:

1) Oriented Derivative of Gaussian (DoG) filters: This is a collection Derivative of Gaussian filters in different orientations. These DoG filters are obtained by convolving a sobel operator on to a Gaussian kernel. Oriented DoG filters of 2 scales and 16 orientations are shown in figure 1.
2) Leung-Malik Filters: It is a set of 48 filters of multiple scales and multiple orientations. It contains 36 first and second order derivatives of Gaussians at 6 orientations and 3 scales, 8 Laplacian of Gaussian (LoG) filters, and 4 Gaussians. For this experiment, 2 Leung-Malik filter banks, namely, LM small and LM large was generated. The LM small and LM large filter banks that I generated for this experiment is shown in figures 2 and 3 respectively.
3) Gabor filters: These filters are designed based on the operation of human eye. It is basically a gaussian kernel modulated by a sinusoidal wave. The Gabor filter bank that I generated dor this experiment is shown in figure 4.

Fig. 1: Oriented Derivative of Gaussian Filter Bank.

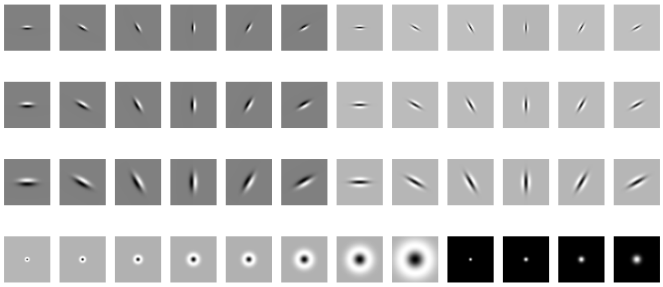

Fig. 2: Small Leung-Malik Filter Bank.

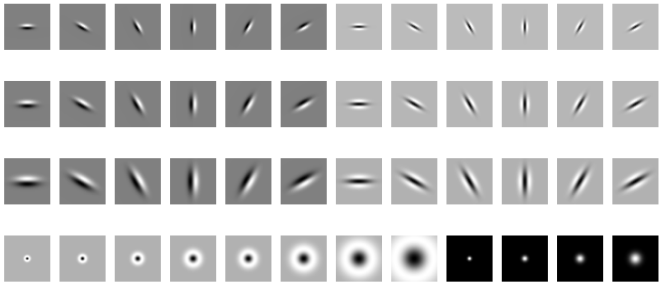

Fig. 3: Large Leung-Malik Filter Bank.

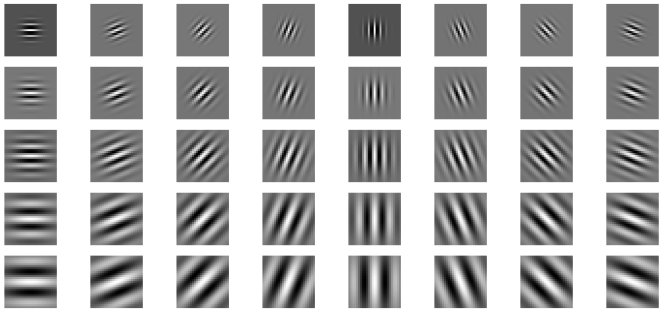

Fig. 4: Gabor Filter Bank.

B. Texton, Brightness and Color Maps

Once the filter banks are generated, each filter in the filters banks generated in the previous step needs to be applied on to the input images. This will result in a vector of filter responses

centered around each pixel in the image. This vector of filter responses is basically the encoding of of texture properties of the image. If the filter banks contain N filters in total, the output will be an N-dimensional vector corresponding to each pixel.

The next step is to replace each of these N-dimensional vectors with a discrete texton ID. This is done by clustering the filter responses for each pixel into K textons by means of KMeans clustering. The each pixel in the image is replaced by the discrete texton ID obtained through KMeans clustering to obtain the Texton map ($\mathcal{T}$). So, the output will be a singe channel image with values in the range of [1,2,...,K]. For this experiment, a value of 64 was selected for K.

The brightness and color maps can be generated in similar fashion. To obtain the Brightness map (B), we have to cluster the brightness or intensity values. This is done by performing KMeans clustering on the grayscale equivalent of the color image. Similary, when KMeans clustering is performed on the default three channel color image, the Color map (C) is obtained. A K value of 16 was chosen for both brightness map and color map in this experiment.

The texton maps ($\mathcal{T}$), the brightness maps ($\mathcal{B}$), and the Color maps ($\mathcal{C}$) for each input image under study is shown in the figures 5 through 14. Different color maps were used in the showing of the maps for proper visualization.

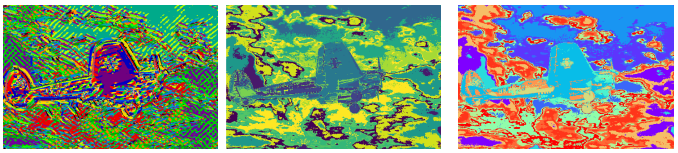

Fig. 5: $\mathcal{T}$, $\mathcal{B}$, and $\mathcal{C}$ for image 1.

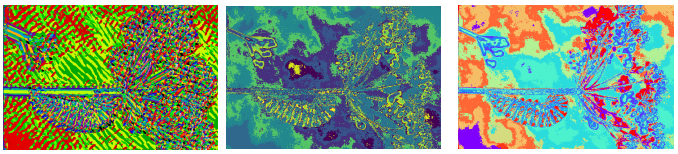

Fig. 6: $\mathcal{T}$, $\mathcal{B}$, and $\mathcal{C}$ for image 2.

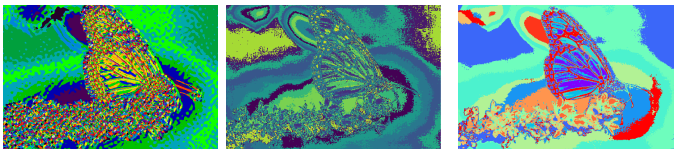

Fig. 7: $\mathcal{T}$, $\mathcal{B}$, and $\mathcal{C}$ for image 3.

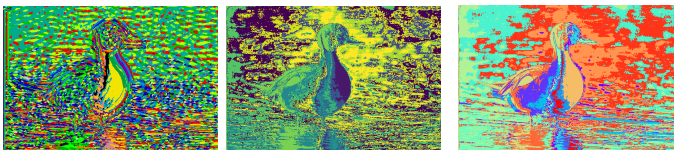

Fig. 8: $\mathcal{T}$, $\mathcal{B}$, and $\mathcal{C}$ for image 4.

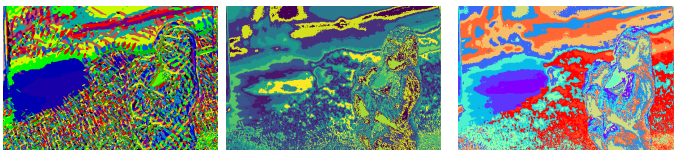

Fig. 9: $\mathcal{T}$, $\mathcal{B}$, and $\mathcal{C}$ for image 5.

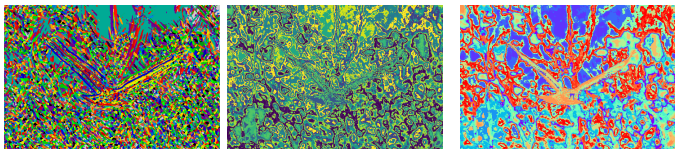

Fig. 10: $\mathcal{T}$, $\mathcal{B}$, and $\mathcal{C}$ for image 6.

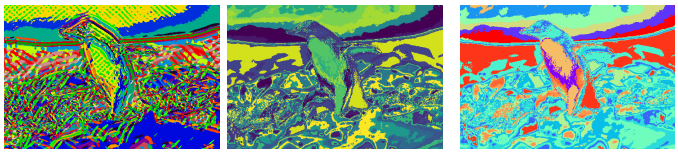

Fig. 11: $\mathcal{T}$, $\mathcal{B}$, and $\mathcal{C}$ for image 7.

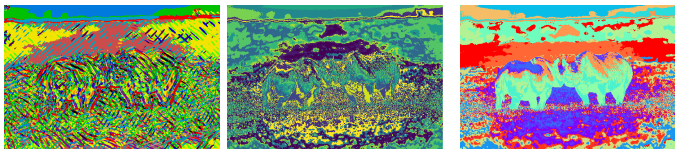

Fig. 12: $\mathcal{T}$, $\mathcal{B}$, and $\mathcal{C}$ for image 8.

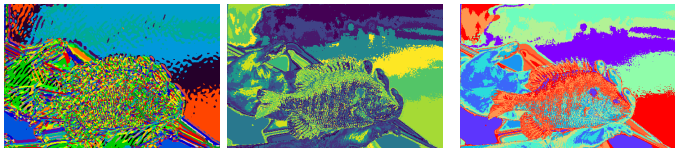

Fig. 13: $\mathcal{T}$, $\mathcal{B}$, and $\mathcal{C}$ for image 9.

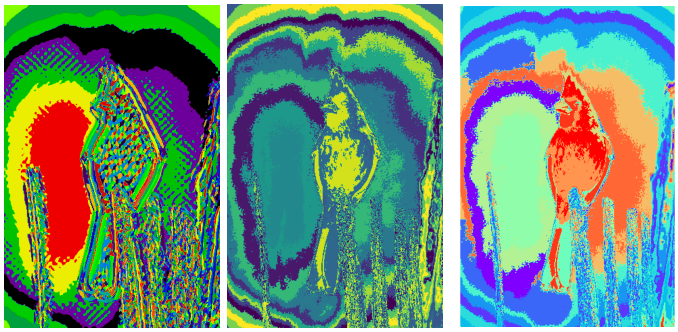

Fig. 14: $\mathcal{T}$, $\mathcal{B}$, and $\mathcal{C}$ for image 10.

C. Texton, Brightness and Color Gradient Maps

The reason why we generated the texton, brightness and color maps of the input images was to find the gradients of these maps in order to understand the pixel neighbourhoods where change in texture, intensity and color properties was happening. To find the gradients of the images, we use the half-disc masks shown in figure 15. The half-disc masks are pairs of binary images of half-discs. The purpose of the masks is to calculate the χ^2 distances using a simple filtering operation. This completely simplifies the process where χ^2 distances are calculated by aggregating counts for histogram by looping over entire pixel neighbourhoods. For this experiment, I generated a half-disc mask pairs of 8 orientations and 3 scales.

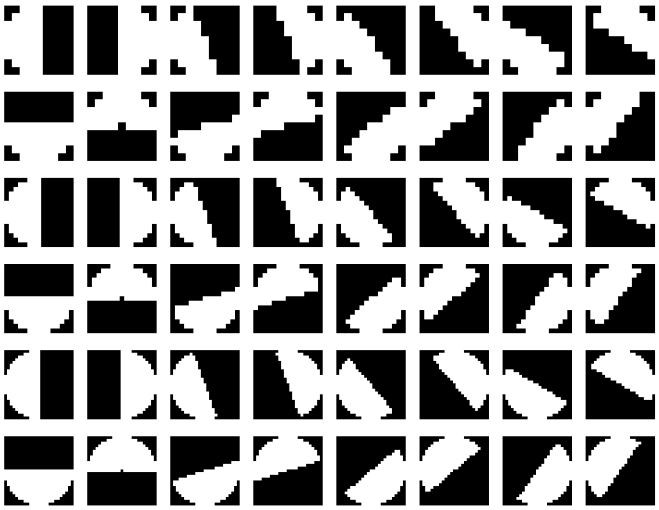

Fig. 15: Half-disc Masks.

The texton, brightness and color maps are filtered using the masks. Then, the χ^2 distances between two histograms g and h can be obtained by using the equation shown below:

$$\chi^2 = \frac{1}{2}\sum_{i=1}^{K}\frac{(g_i - h_i)^2}{(g_i + h_i)}$$

where K indexes through the bins. Using this method, the texton gradient map ($\mathcal{T}_g$), the brightness gradient map ($\mathcal{B}_g$), and the color gradient map ($\mathcal{C}_g$) can be obtained. The $\mathcal{T}_g$, $\mathcal{B}_g$ and $\mathcal{C}_g$ for the given input images that I obtained during the experiment are shown in the figures 16 through 25.

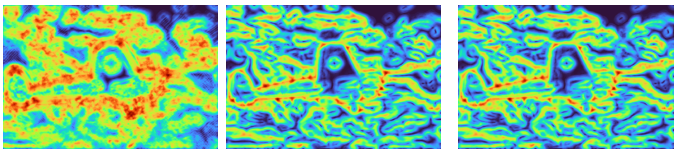

Fig. 16: $\mathcal{T}_g$, $\mathcal{B}_g$, and $\mathcal{C}_g$ for image 1.

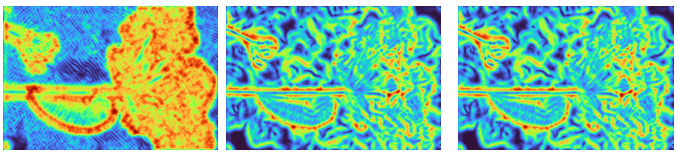

Fig. 17: $\mathcal{T}_g$, $\mathcal{B}_g$, and $\mathcal{C}_g$ for image 2.

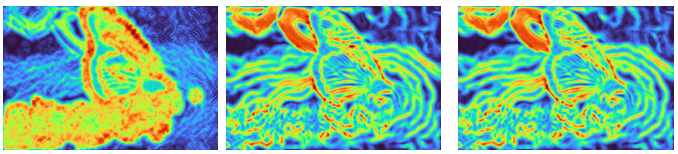

Fig. 18: $\mathcal{T}_g$, $\mathcal{B}_g$, and $\mathcal{C}_g$ for image 3.

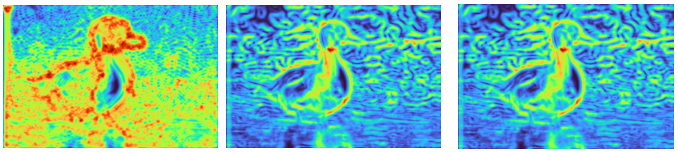

Fig. 19: $\mathcal{T}_g$, $\mathcal{B}_g$, and $\mathcal{C}_g$ for image 4.

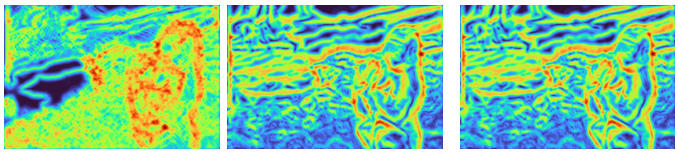

Fig. 20: $\mathcal{T}_g$, $\mathcal{B}_g$, and $\mathcal{C}_g$ for image 5.

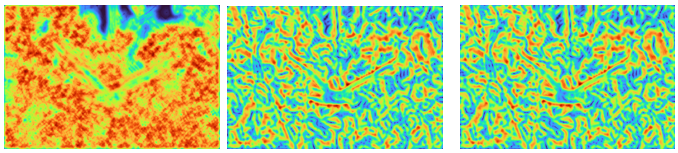

Fig. 21: $\mathcal{T}_g$, $\mathcal{B}_g$, and $\mathcal{C}_g$ for image 6.

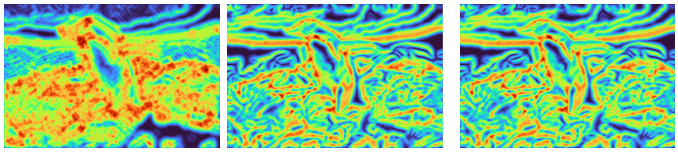

Fig. 22: $\mathcal{T}_g$, $\mathcal{B}_g$, and $\mathcal{C}_g$ for image 7.

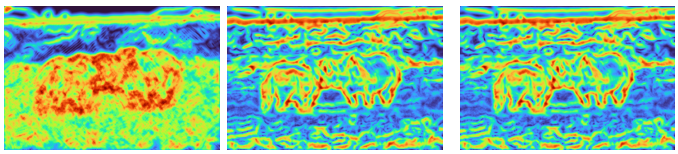

Fig. 23: $\mathcal{T}_g$, $\mathcal{B}_g$, and $\mathcal{C}_g$ for image 8.

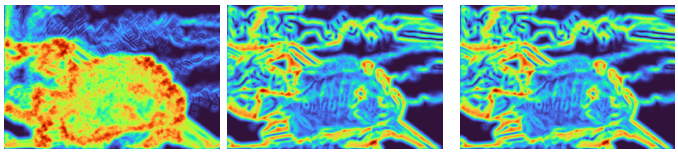

Fig. 24: $\mathcal{T}_g$, $\mathcal{B}_g$, and $\mathcal{C}_g$ for image 9.

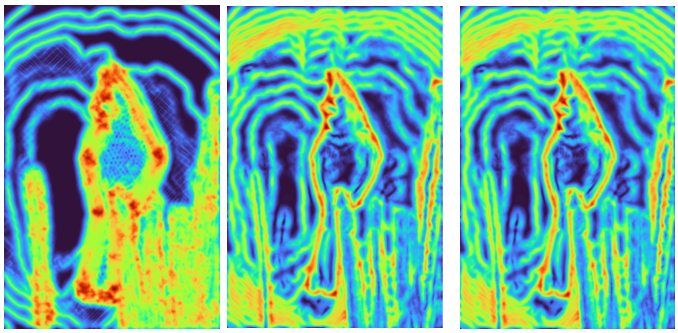

Fig. 25: $\mathcal{T}_g$, $\mathcal{B}_g$, and $\mathcal{C}_g$ for image 10.

D. Boundary Detection

Now that we have all the gradient maps of the input images, we can proceed to generating the pb-lite output. But before doing there is a one last step. That is to read all the Sobel and Canny baselines of these images that were provided with the input data. Once these baselines are obtained, they can be used to generate the pb-lite output with the given equation:

$$PbEdges = \frac{\mathcal{T}_g + \mathcal{B}_g + \mathcal{C}_g}{3} \odot (w_1 * cannyPb + w_2 * sobelPb)$$

For this experiment I have chosen the value of 0.5 for both w_1 and w_2. The pb-lite outputs that I obtained during my

experiment is show in figures 26 through 35, along with their corresponding Sobel and Canny baselines for comparison.

Fig. 26: Sobel, Canny, and pb-lite output of image 1.

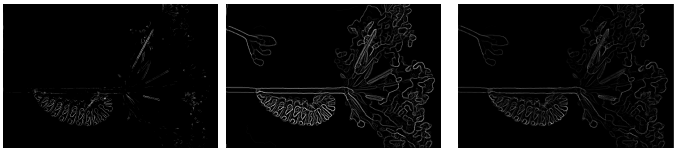

Fig. 27: Sobel, Canny, and pb-lite output of image 2.

Fig. 28: Sobel, Canny, and pb-lite output of image 3.

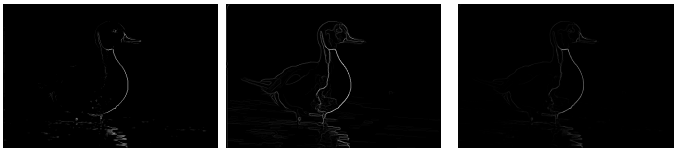

Fig. 29: Sobel, Canny, and pb-lite output of image 4.

Fig. 30: Sobel, Canny, and pb-lite output of image 5.

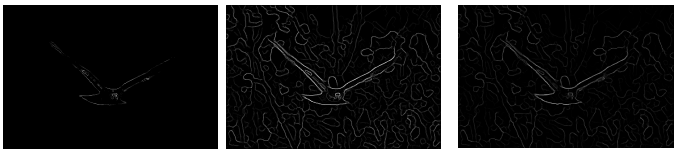

Fig. 31: Sobel, Canny, and pb-lite output of image 6.

Fig. 32: Sobel, Canny, and pb-lite output of image 7.

Fig. 33: Sobel, Canny, and pb-lite output of image 8.

Fig. 34: Sobel, Canny, and pb-lite output of image 9.

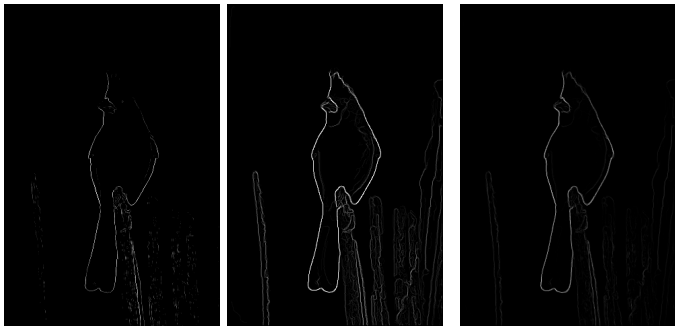

Fig. 35: Sobel, Canny, and pb-lite output of image 10.

E. Analysis

Upon comparing the pb-lite results with the Sobel and Canny baselines, pb-lite has managed to remove a lot of false positives in Canny baseline but also include a lot of neglected information in Sobel baseline. Also, the fact that the user has more freedom in pb-lite, in terms of choosing the filter banks of different scales and orientations, the boundary detection output can be fine tuned to satisfy the requirements of any user. Thus, it can be stated that pb-lite boundary detection method is better than both Sobel and Canny boundary detection algorithms.

II. PHASE 2: DEEP DIVE ON DEEP LEARNING

In this phase, I aim to implement a number of neural network architecture in order to perform classification operation on the CIFAR-10 dataset which contains 50,000 training and 10,000 testing images of size 32x32 belonging to 10 classes. I trained a total of five models in the experiment, which are shown below:

1) My Basic Neural Network
2) My Improved Neural Network
3) Resnet
4) ResNeXt
5) DenseNet

Now lets take a closer look at all these networks and their performance.

A. My Basic Neural Network

For this section, I implemented a basic classification neural network with two convolutional layers each followed by a ReLU activation function. The outputs of these ReLU activation functions are passed through maxpool layers. Finally, the output of the second maxpool layer is flattened and then passed through a set of fully connected layers which then through argmax gives the predicted class. I decided to give this network the name BasicNet. The architecture of BasicNet is shown in figure 36.

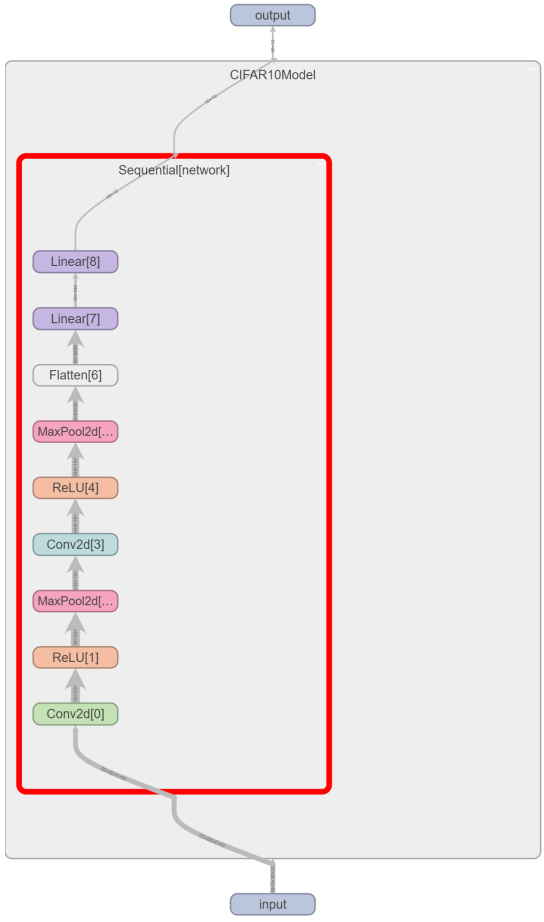

Fig. 36: Architecture of BasicNet.

For this experiment, I selected a batch size of 32 and the optimizer selected was Adam with a learning rate of 1e-3. I also selected Cross Entropy loss for training the network. Upon training the model for 15 epochs, these results were obtained are shown in figures 37 through 40.

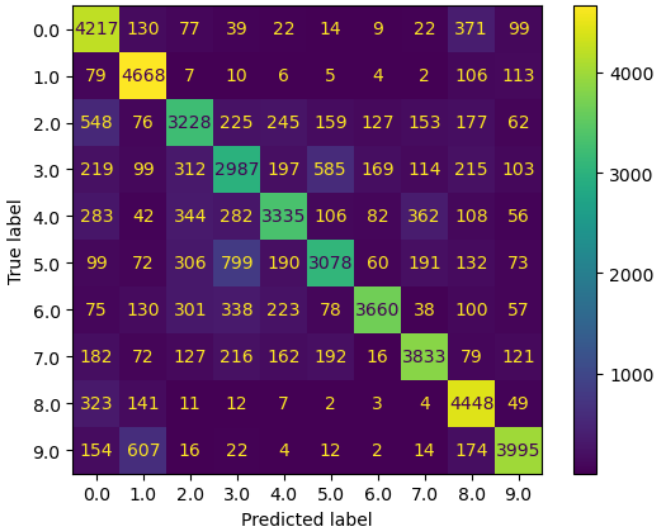

Fig. 37: Confusion of trained model on training data for BasicNet.

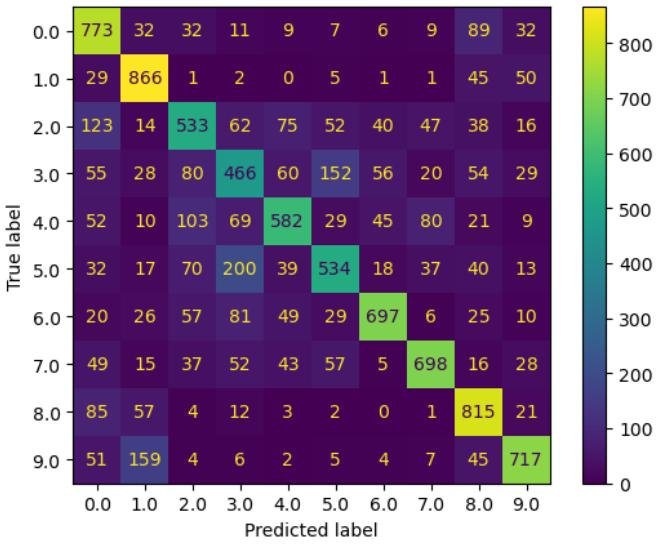

Fig. 38: Confusion of trained model on testing data for BasicNet.

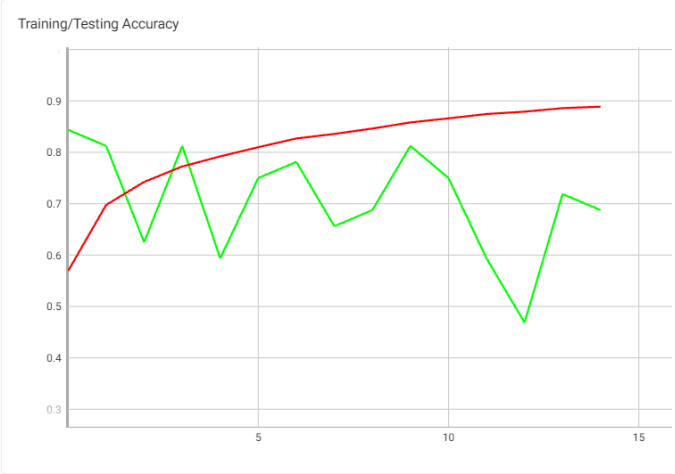

Fig. 39: Train and Test accuracy over epochs for BasicNet.

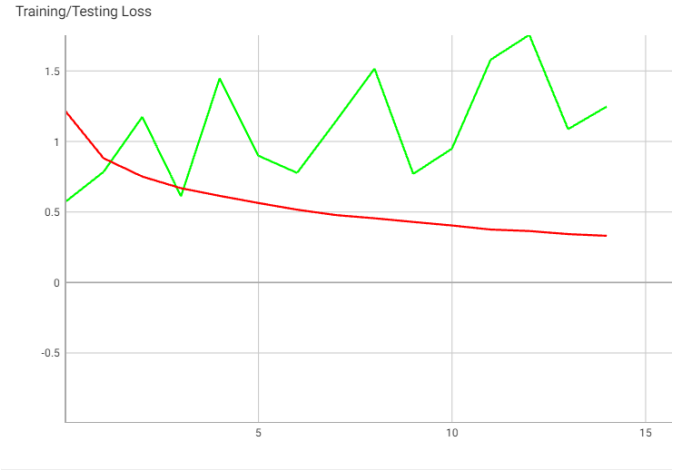

Fig. 40: Train and Test loss over epochs for BasicNet.

In the loss and accuracy per epoch plots, the red line represents the training and green line represents the testing. Also, the number of parameters in the model was found to

be 430102. Alongside these, the confusion matrix for both training data and testing data of the trained model is also provided for the better visualization of the performance of the model. Overall, an accuracy of 68.42% was obtained for the testing data.

B. My Improved Neural Network

For this section, I tried to improve the BasicNet. From the given instructions, I chose to use a number of Batch Normalization layers between the convolutional layers. Also, I increased the number of convolutional layers from the BasicNet. For convenience, I decided to call this network BatchNormNet. Apart from the batch normalization layers and the extra convolutional layers, this network similar to the BasicNet. The architecture of this network is shown in figure 41.

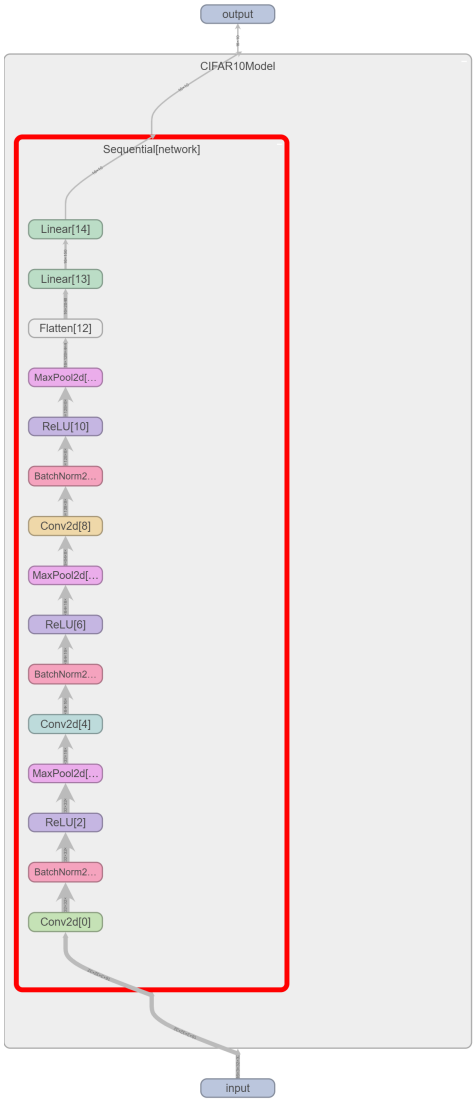

Fig. 41: Architecture of BatchNormNet.

For this experiment, I selected a batch size of 32 and the optimizer selected was Adam with a learning rate of 1e-3. I also selected Cross Entropy loss for training the network. Upon training the model for 15 epochs, the results obtained are shown in figures 42 through 45.

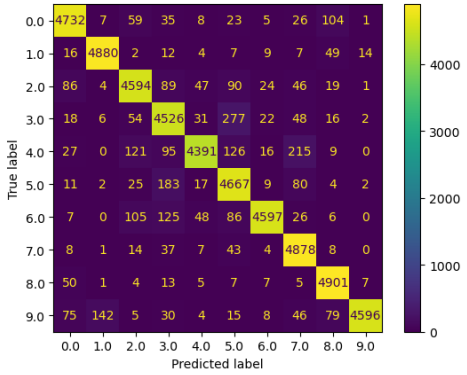

Fig. 42: Confusion of trained model on training data for BatchNormNet.

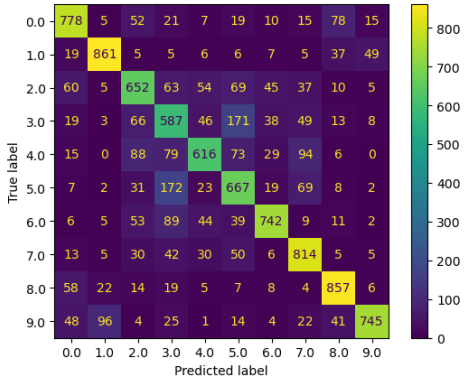

Fig. 43: Confusion of trained model on testing data for BatchNormNet.

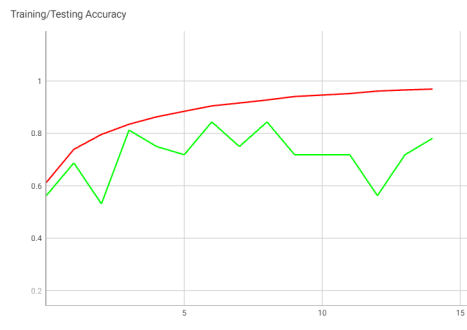

Fig. 44: Train and Test accuracy over epochs for BatchNormNet.

Fig. 45: Train and Test loss over epochs for BatchNormNet.

In the loss and accuracy per epoch plots, the red line represents the training and green line represents the testing. Also, the number of parameters in the model was found to be 402556. Alongside these, the confusion matrix for both training data and testing data of the trained model is also provided for the better visualization of the performance of the model. Overall, an accuracy of 73.19% was obtained for the testing data.

C. ResNet

For this section, I tried to implement a Residual Network (ResNet). My implementation of the ResNet has four layers containing ResNet blocks. The architecture of the whole network is shown in figure 46 and the structure of a sample ResNet block from the network is shown in figure 47. ResNet blocks passes the down sampled input to the output of the block so that it can remove the issue of vanishing gradient. Its capabilities are visible when the model is necessarily deep.

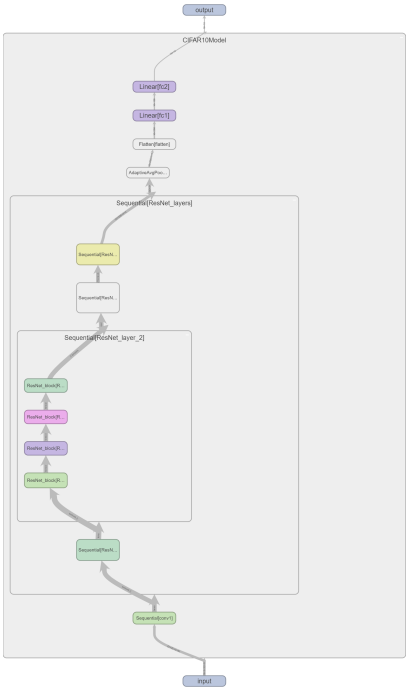

Fig. 46: Architecture of ResNet.

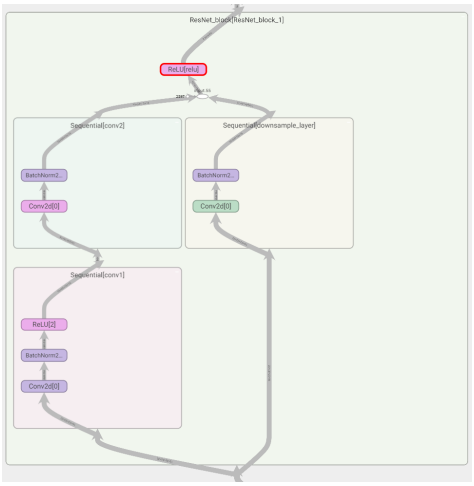

Fig. 47: The ResNet Block.

For this experiment, I selected a batch size of 32 and the optimizer selected was Adam with a learning rate of 1e-3. I also selected Cross Entropy loss for training the network. Upon training the model for 15 epochs, the results obtained are shown in figures 42 through 45.

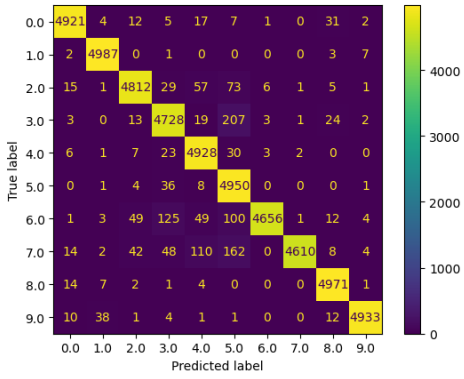

Fig. 48: Confusion of trained model on training data for ResNet.

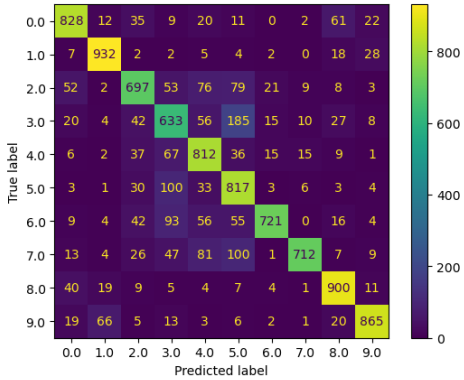

Fig. 49: Confusion of trained model on testing data for ResNet.

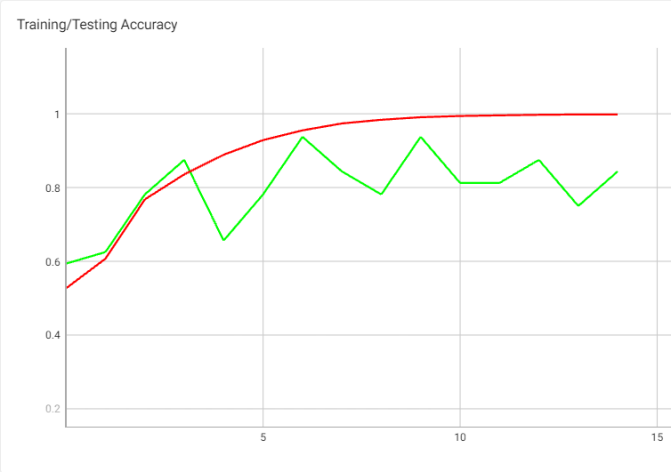

Fig. 50: Train and Test accuracy over epochs for ResNet.

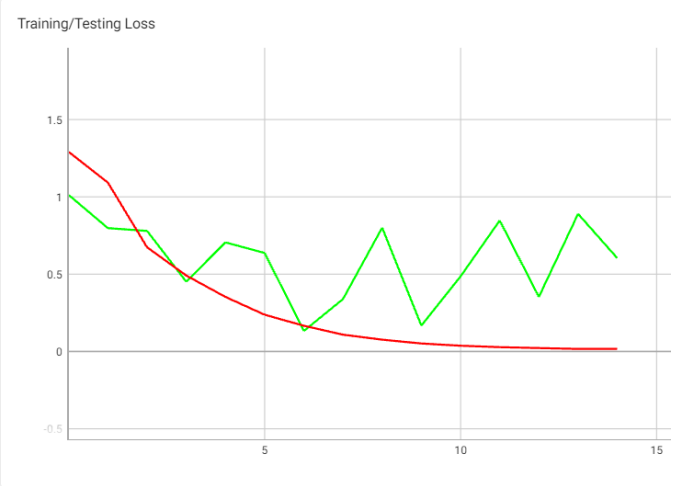

Fig. 51: Train and Test loss over epoch for ResNet.

In the loss and accuracy per epoch plots, the red line represents the training and green line represents the testing. Also, the number of parameters in the model was found to be 5350422. Alongside these, the confusion matrix for both training data and testing data of the trained model is also provided for the better visualization of the performance of the model. Overall, an accuracy of 79.17% was obtained for the testing data.

D. ResNeXt

For this section, I tried to implement a ResNeXt deep learning network. My implementation of the ResNeXt has four layers containing ResNeXt blocks. The architecture of the whole network is shown in figure 52 and the structure of a sample ResNeXt block from the network is shown in figure 53. In ResNext, a new dimension called cardinality is implemented in order to handle model complexity efficiently. In my network, a cardinality of 32 was selected.

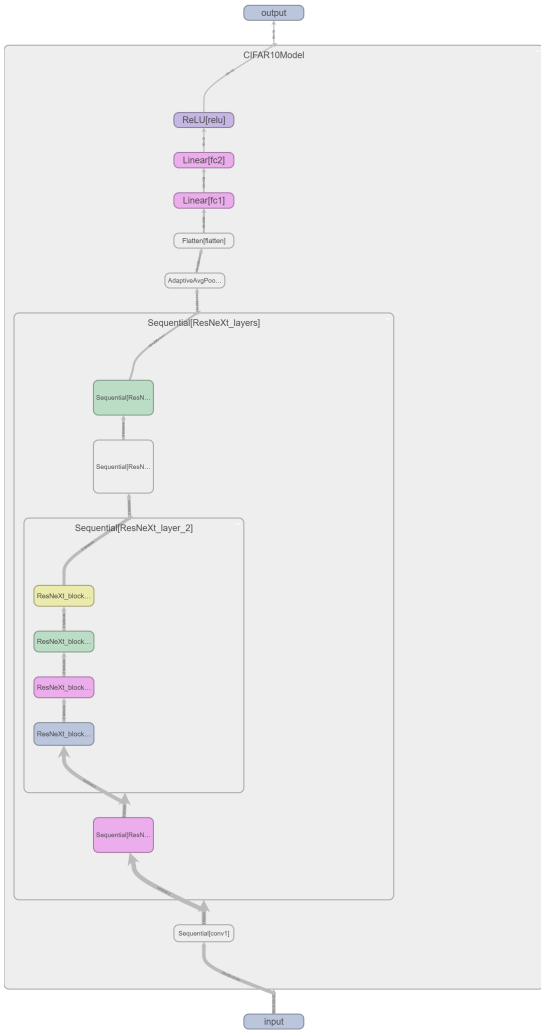

Fig. 52: Architecture of ResNeXt.

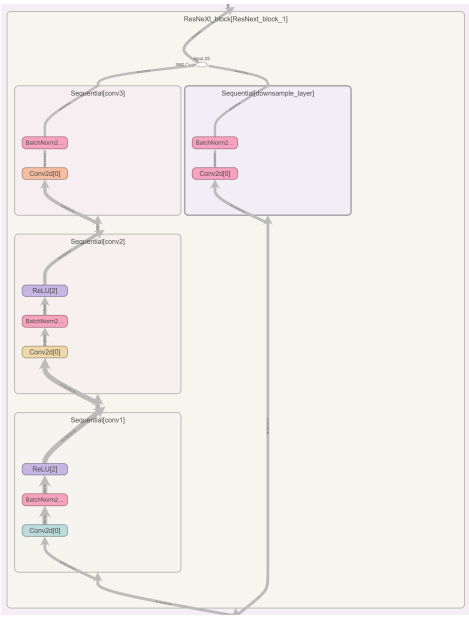

Fig. 53: The ResNeXt Block.

The tensorboard graph has not done a great job visualizing

the cardinality feature of ResNeXt. The cardinality feature can be properly seen in figure 54.

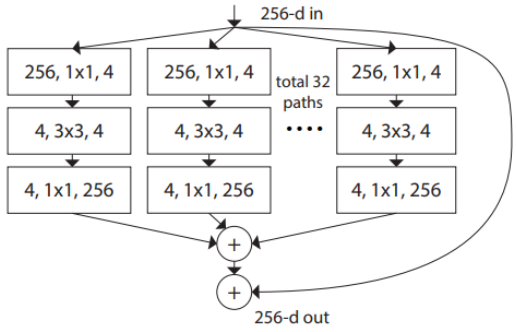

Fig. 54: The cardinality feature of ResNeXt.

For this experiment, I selected a batch size of 32 and the optimizer selected was Adam with a learning rate of 1e-3. I also selected Cross Entropy loss for training the network. Upon training the model for 15 epochs, the results obtained are shown in figures 55 through 58.

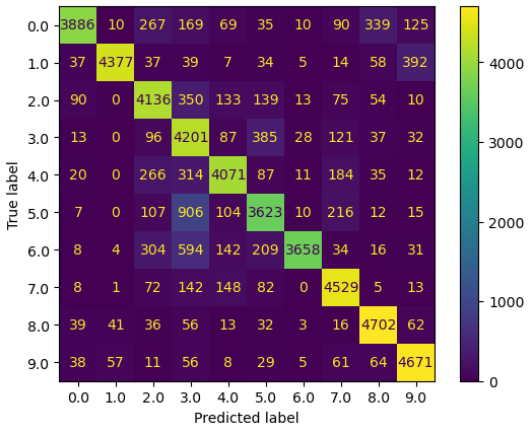

Fig. 55: Confusion of trained model on training data for ResNeXt.

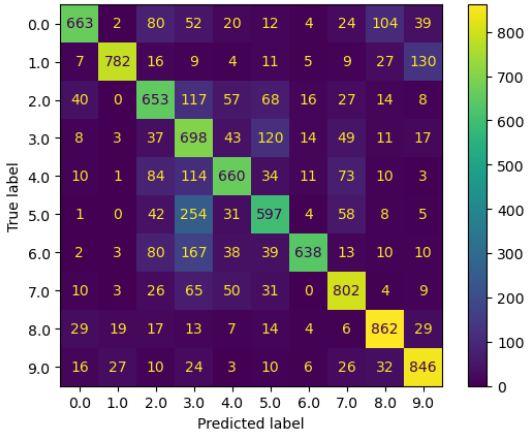

Fig. 56: Confusion of trained model on testing data for ResNeXt.

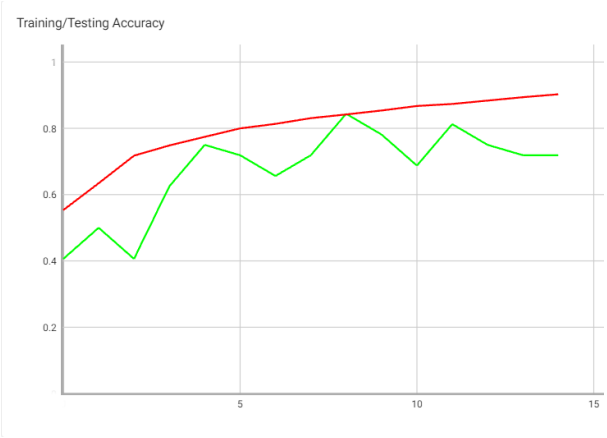

Fig. 57: Train and Test accuracy over epochs for ResNeXt.

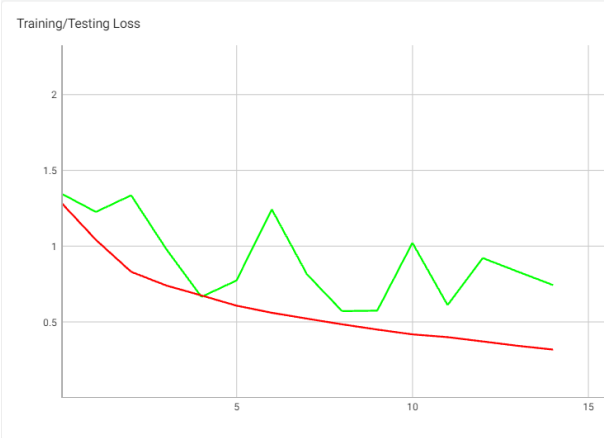

Fig. 58: Train and Test loss over epoch for ResNeXt.

In the loss and accuracy per epoch plots, the red line represents the training and green line represents the testing. Also, the number of parameters in the model was found to be 616438. Alongside these, the confusion matrix for both training data and testing data of the trained model is also provided for the better visualization of the performance of the model. Overall, an accuracy of 72.01% was obtained for the testing data.

E. DenseNet

In this section, I will be implementing a DenseNet deep learning model. By far, this is the most complex network with different DenseNet Bottleneck Blocks connected by DenseNet transition layers. The whole DenseNet architecture is shown in figure 59 and the basic DenseNet block is shown in figure 60. Also, the transition layer used to connect DenseNet layers is shown in figure 61.

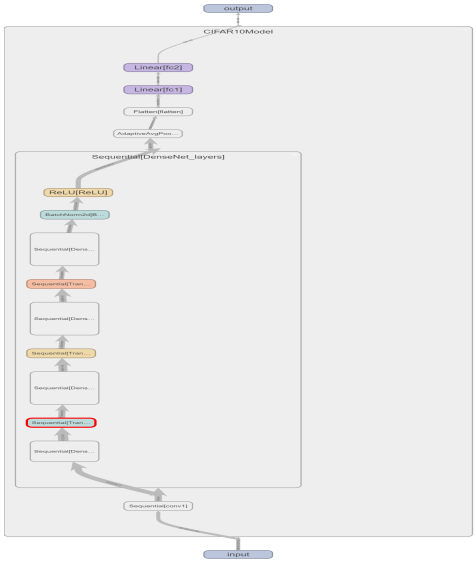

Fig. 59: Architecture of DenseNet.

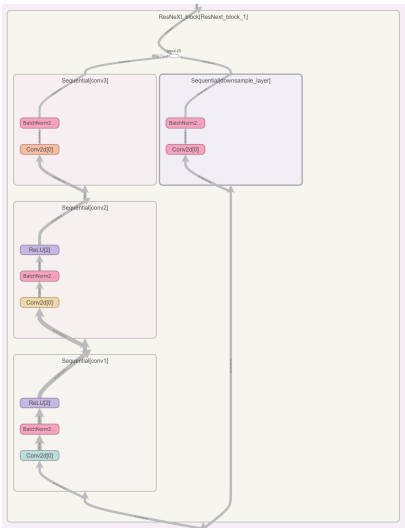

Fig. 60: The DenseNet Block.

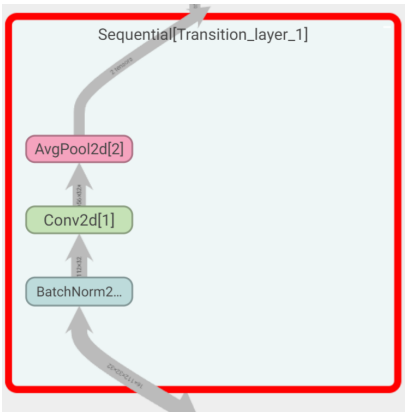

Fig. 61: The DenseNet Transition Layer.

For this experiment, I selected a batch size of 32 and the optimizer selected was Adam with a learning rate of 1e-3. I also selected Cross Entropy loss for training the network. Upon training the model, the results obtained are shown in figures 62 through 65.

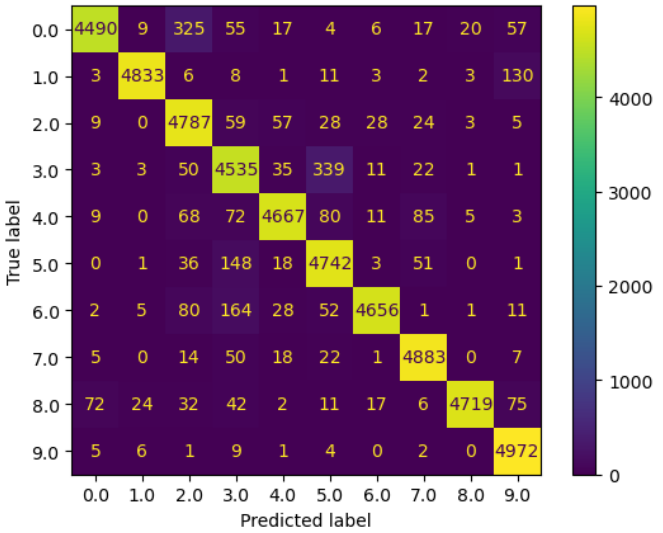

Fig. 62: Confusion of trained model on training data for DenseNet.

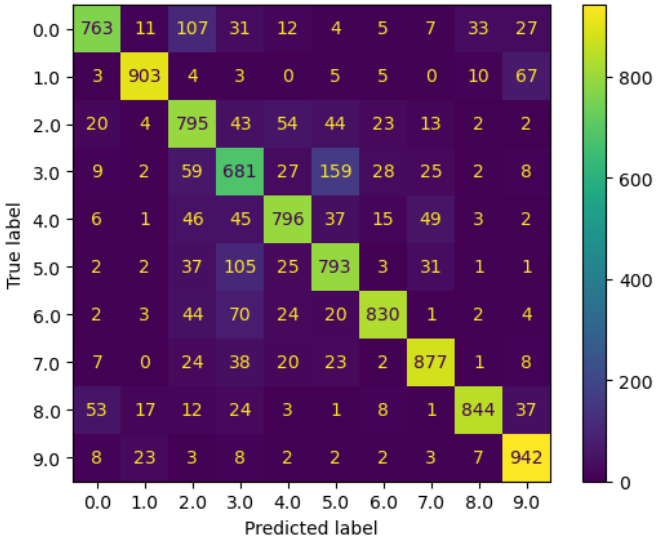

Fig. 63: Confusion of trained model on testing data for DenseNet.

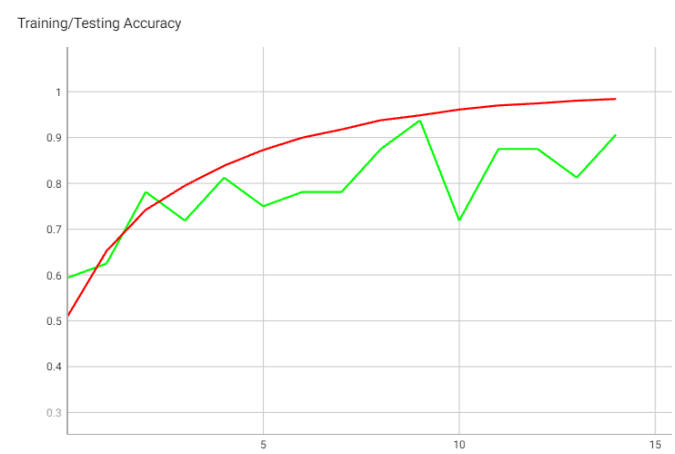

Fig. 64: Train and Test accuracy over epochs for DenseNet.

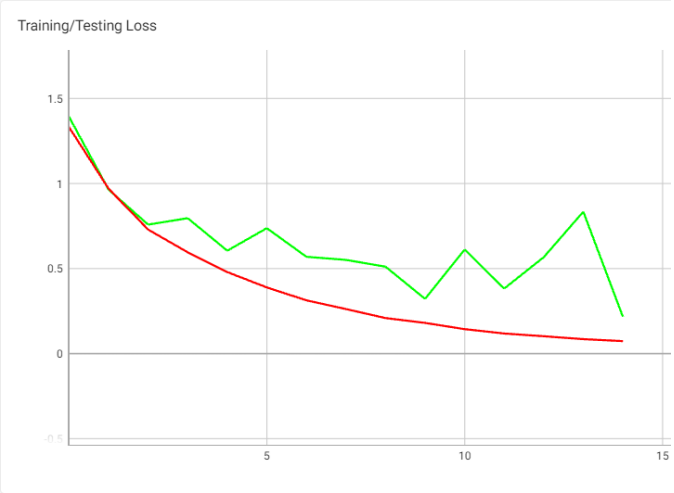

Fig. 65: Train and Test loss over epoch for DenseNet.

In the loss and accuracy per epoch plots, the red line represents the training and green line represents the testing. Also, the number of parameters in the model was found to be 1782394. Alongside these, the confusion matrix for both training data and testing data of the trained model is also provided for the better visualization of the performance of the model. Overall, an accuracy of 82.24% was obtained for the testing data.

F. Comparison

The table below is a summarization of performance of all the models. The inference time of each model is also included in this comparative analysis. This calculated by using the get_model_complexity_info function from the 'ptflops' library which returns the Mac value corresponding to the input model. This value is an indicator of the complexity of the model and is roughly equal to twice the FLOPS involved. This value when divided by the TFLOPS of the GPU used for training will give the inference time.

Model	Parameters	Train Accuracy	Test Accuracy	Inference Time
BasicNet	430102	83.004%	68.42%	0.186µs
BatchNormNet	402556	93.524%	73.19%	0.329µs
ResNet	5350422	96.992%	79.17%	8.677µs
ResNeXt	616438	83.708%	72.01%	0.198µs
DenseNet	1782394	94.568%	82.24%	6.575µs

TABLE I: The comparison of all the models.

It is clearly visible from the table that the BasicNet initially gave a lesser testing accuracy of 68.42% and the accuracy improved to 73.19% when batch normalization layers were included (BatchNormNet). ResNet and Densenet gave much higher accuracies as expected. But the performance of my implementation of ResNeXt is not as great as that of ResNet and DenseNet even though it gave an higher accuracy compared to the BasicNet. This accuracy can be improved by choosing different hyperparameters apt for this specific task.