Alohomora!

Computer Vision (RBE549) Homework 0

Tejas Rane
MS Robotics Engineering
Worcester Polytechnic Institute
turane@wpi.edu

I. Phase 1: Shake My Boundary

The first part of the assignment explores classical methods in computer vision for boundary detection. I present a detailed analysis of my implementation of the pb (probability of boundary) boundary detection algorithm. Here, we implement its lite version - "pb-lite". The algorithm works by using texture, brightness and color information to improve the boundary detection result of the Sobel and Canny baseline. The method consists of 4 main steps: **(1)** Filter Bank Generation **(2)** Texton, Brightness and Color Map computation **(3)** Texture, Brightness and Color Gradients computation, and **(4)** Boundary Detection.

A. Filter Bank Generation

The first step in the boundary detection algorithm is to create filter banks so that they can be used to capture texture information from the input images. In this implementation, we use three types of filters: Oriented Derivative of Gaussian (DoG) filters, Leung-Malik filters, and Gabor filters.

1) Oriented Derivative of Gaussian (DoG) Filters: As mentioned in the problem statement, these filters are created by convolving a simple Sobel filter and a Gaussian kernel and then rotating the result. The filter bank generated with 4 scales and 16 orientations is shown in Figure 1.

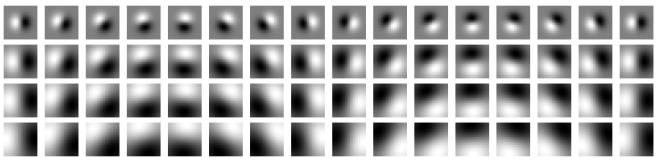

Fig. 1: Oriented Derivative of Gaussian (DoG) filters.

2) Leung-Malik (LM) Filters: These are a set of multi scale, multi orientation filter bank with 48 filters. In this implementation, we consider two versions of Leung-Malik filter banks. The Leung-Malik Small filter bank is generated using the scales $\sigma = \{1, \sqrt{2}, 2, 2\sqrt{2}\}$, whereas the Leung-Malik Large filter bank is generated using the scales $\sigma = \{\sqrt{2}, 2, 2\sqrt{2}, 4\}$ The generated Leung-Malik Small filter bank is shown in Figure 2, and the generated Leung-Malik Large filter bank is shown in Figure 3.

3) Gabor Filters: These filters which are approximated versions of how human visual system works. As mentioned in the problem statement, a Gabor filter is a Gaussian kernel function modulated by a sinusoidal plane wave. The filter bank generated with 5 scales and 8 orientations is shown in Figure 4.

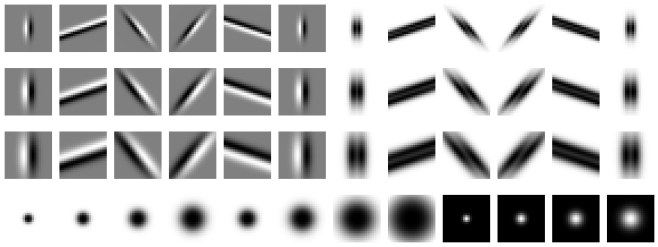

Fig. 2: Leung-Malik Small (LMS) filters.

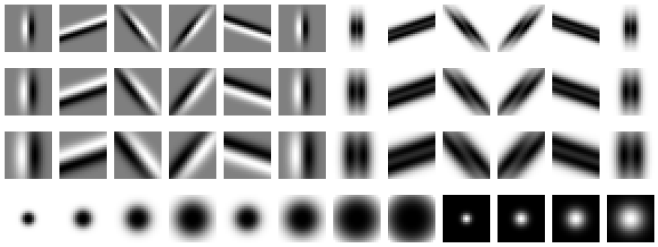

Fig. 3: Leung-Malik Large (LML) filters.

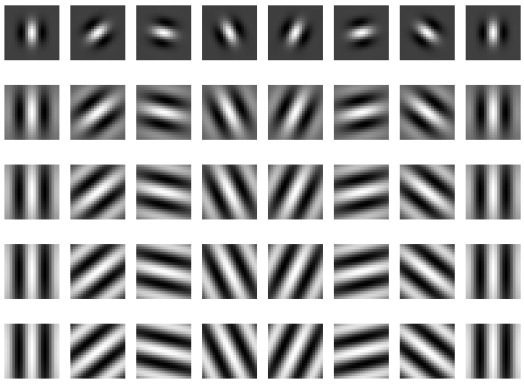

Fig. 4: Gabor Filters.

B. Texton, Brightness, Color Map Computation

The next step is to filter the input image with each and every filter in our generated filter bank. Assuming there are N filters in total in our filter bank, we get an N-dimensional vector as a

response for each pixel after filtering. Then, we use KMeans clustering on these vectors for all pixels, to get discrete Texton IDs for each pixel. Replacing each pixel in the image with its corresponding Texton ID gives us the Texton map. Similarly, we follow the same process to generate the Brightness and Color map. For Brightness, we use the grayscale pixel value and for Color, we use the RGB value of each pixel.

The generated Texton Maps $\mathcal{T}$, Brightness Maps $\mathcal{B}$ and Color Maps $\mathcal{C}$ for all the 10 provided images is shown in Figure 6.

C. Texton, Brightness, Color Gradients Computation

To generate the Texton, Brightness and Color Gradients from the corresponding maps, we first need to generate half-disc masks. The half-disc masks are simply (pairs of) binary images of half-discs. The half-disc masks generated with 3 scales and 8 orientations is shown in Figure 5.

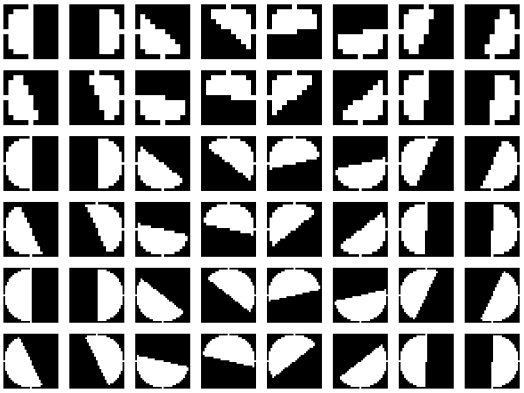

Fig. 5: Half-Disc Masks.

Using these half-disc masks along with the Chi-square distance, the gradients of the Texton, Brightness, and Color maps are generated for each input image. The generated Texton Gradients $\mathcal{T}_g$, Brightness Gradients $\mathcal{B}_g$ and Color Gradients $\mathcal{C}_g$ for all the 10 provided images is shown in Figure 7.

D. Boundary Detection

The final step of the boundary detection pipeline is to combine the Texton, Brightness and Color gradients with the Sobel and Canny baselines to generate the final output. The generated pb-lite boundaries for all the 10 provided images is shown in Figure 8.

E. Analysis

We can see that the pb-lite boundary detection results in Figure 8 are much more noisy as compared to the Sobel and Canny baselines. The algorithm is able to detect the subject, but along with that it also detects many false positives, mainly related to the texture in the image. In some cases the results are better compared to the others. For example, the boundary detection on images 6, 8 and 9 are so noisy that it is difficult to find the subject in the image. But in case of images 2 and 3, the algorithm is able to detect all the intricate details in the subject.

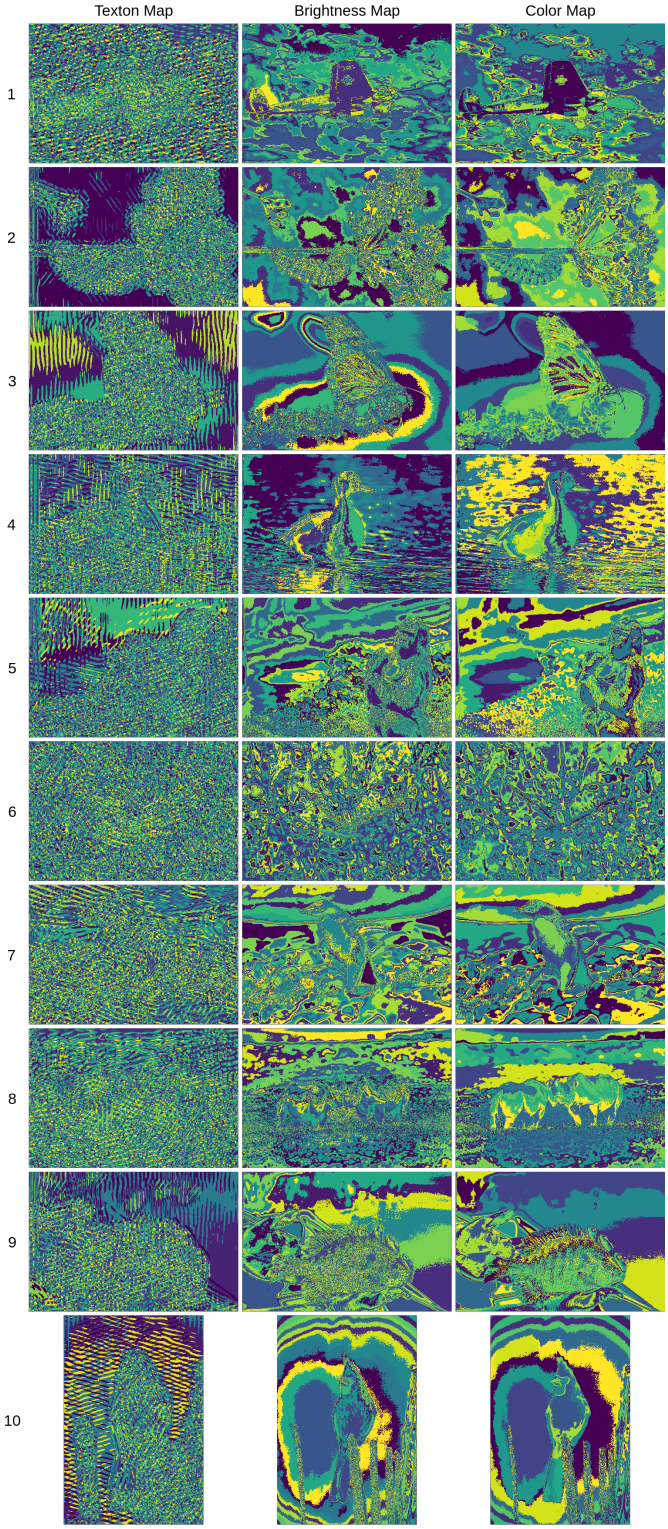

Fig. 6: Texton Map $\mathcal{T}$, Brightness Map $\mathcal{B}$, Color Map $\mathcal{C}$ for all images 1 through 10.

F. Future Work

Due to time constraints, I was not able to perform an extensive hyperparameter search. There are multiple parameters in

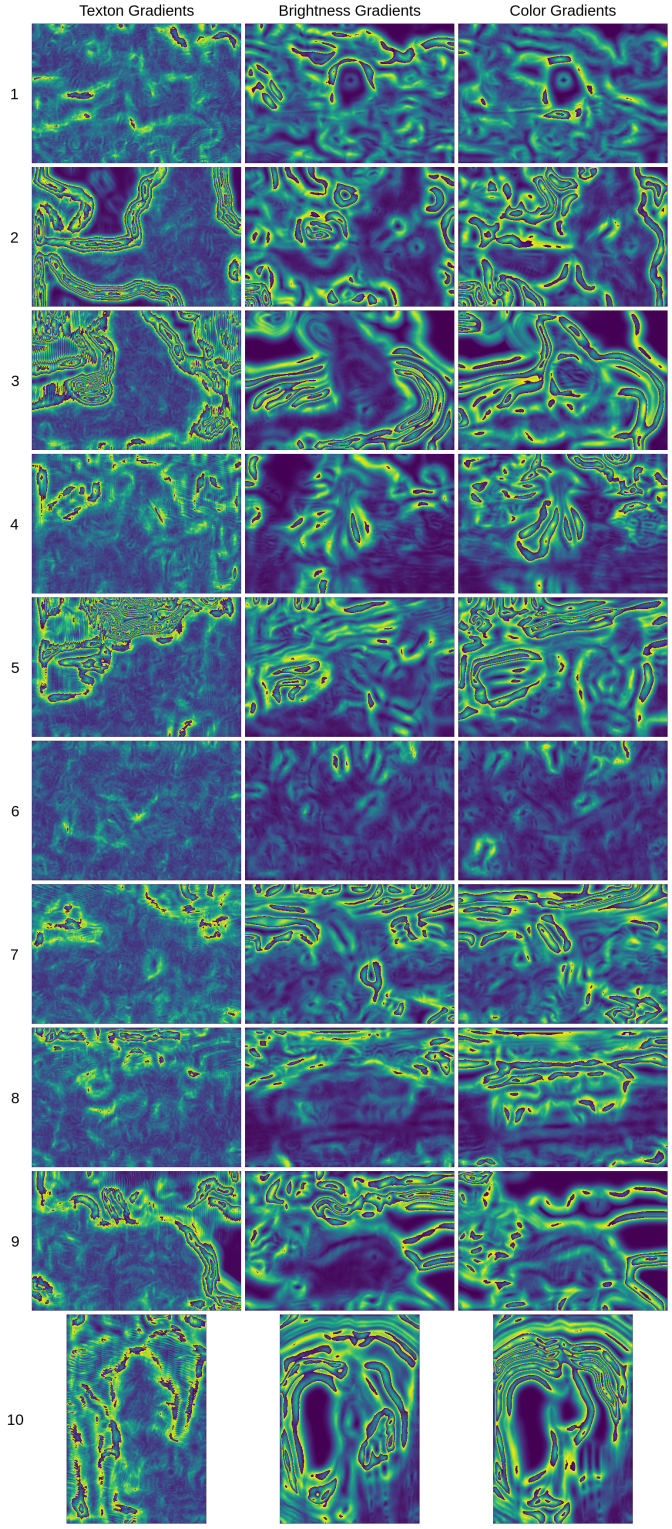

Fig. 7: Texton Gradients $\mathcal{T}_g$, Brightness Gradients $\mathcal{B}_g$, Color Gradients $\mathcal{C}_g$ for all images 1 through 10.

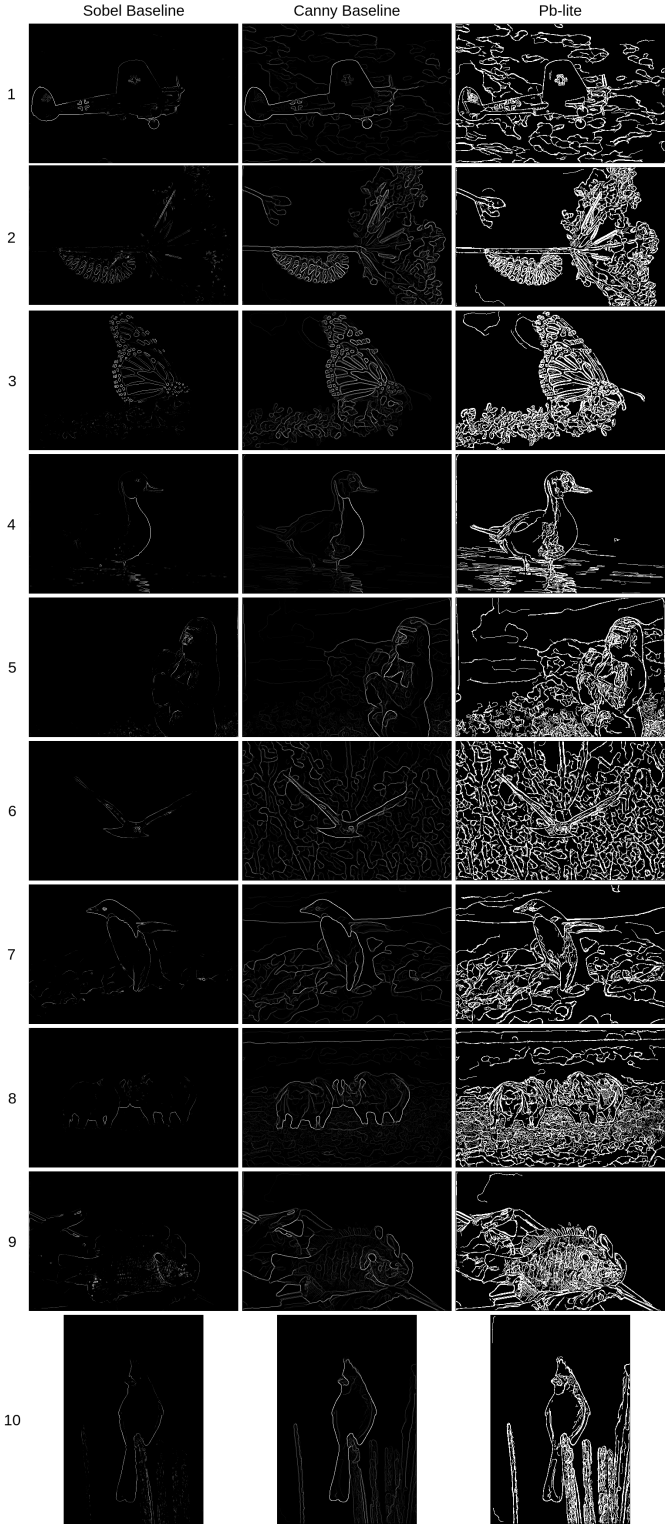

Fig. 8: Comparison of the Sobel baseline, Canny baseline and the pb-lite outputs for all images 1 through 10.

this implementation which can affect the quality of the final boundary detection. A thorough study and tweaking of these hyperparameters can help improve the quality of results. All the filters used to generate the corresponding gradients must be chosen properly, as too many convolutions are creating the noisy results seen in Figure 8.

II. Phase 2: Deep Dive on Deep Learning

The second part of the assignment focuses on computer vision through deep learning. Here, we implement multiple neural networks to perform image classification. All the neural networks are trained on the CIFAR-10 dataset, which consists of $50,000$ training images and $10,000$ testing images.

To be able to compare the performance of all networks with each other, most of the hyperparameters are kept the same wherever possible. All the networks are trained for 50 epochs with a batch size of 128. The optimizer chosen for training is AdamW with an initial learning rate of 0.001 which is decayed exponentially with a rate of 0.9, and a regularization strength (weight decay) of 0.0001. All the networks are implemented and trained on the WPI Turing Cluster.

A. *My first Neural Network*

My first network implementation is a simple linear model with only fully connected linear layers. The network has four layers and around 37.8M trainable parameters. There are no dropout, batch normalization layers, or non-linear activation functions used in this network. This network is implemented as the most naive approach to image classification using the simplest form of neural network layers - fully connect linear layers. A block diagram of the model architecture is shown in Figure 37. The training and testing loss of this model is shown in Figure 9, accuracy in Figure 10 and confusion matrices in Figure 11. This network took about 20 minutes to train.

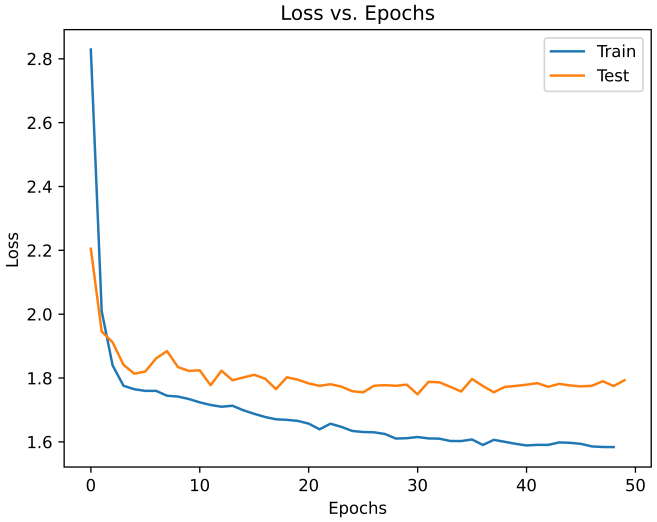

Fig. 9: Training and Testing Loss of fully connected linear model.

As it can be seen from Figure 10, the initial model is able to achieve only about 40% accuracy on the test set. A linear model with only fully connected layers has multiple disadvantages when the input data is an image. The biggest disadvantage is that the fully connected hidden layers cannot capture the spatial structure of images. Hence, a tiny shift in the input image can result in an arbitrary large change in the activations of the hidden layers, thus changing the output drastically.

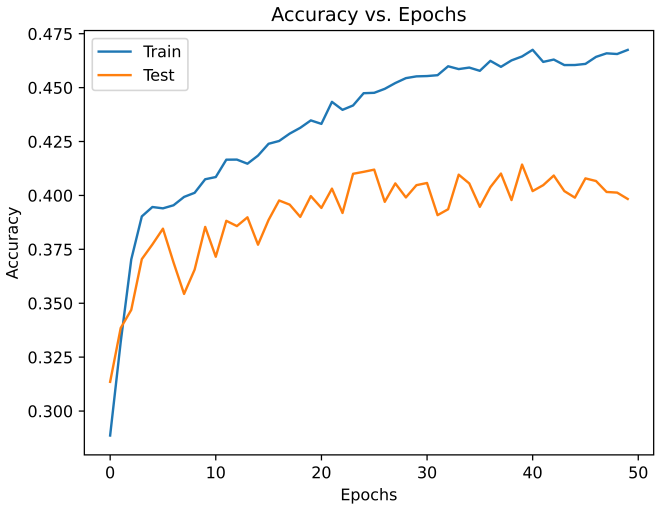

Fig. 10: Training and Testing Accuracy of fully connected linear model.

To overcome these issues, I next implemented a famous and widely used network architecture for image classification - VGG19[1]. VGG19 takes advantage of convolutional layers along with non-linear activation functions like ReLU to generate better performance with almost the same number of trainable parameters. The VGG19 model has 19 layers, of which 16 are convolutional layers which represent the encoder, and 3 are fully connected layers. A block diagram of the VGG19 architecture is shown in Figure 38. My implementation of VGG19 has around 39M trainable parameters. This network took about 42 minutes for training. Plots comparing training and testing loss for both the models is shown in Figure 12, and comparing training and testing accuracy is shown in Figure 13.

As it is evident from the plots, the VGG19 model performs much better as compared to the fully connected linear model. We can see in Figure 13 that VGG19 achieves much better training and testing accuracy as compared to the linear model. But it also overfits easily to the training data as seen in Figure 12. The confusion matrix for the VGG19 model is shown in Figure 14.

B. *Improving accuracy of the neural network*

In the previous section, we saw that the performance of the VGG19 model for the image classification task is much better as compared to the fully connected linear model, as the VGG19 model uses convolutional layers along with non-linear activation function like ReLU. In this section, we will employ different strategies to further improve the performance of VGG19.

[1]I have implemented and trained the VGG19 models from scratch. No pre-trained weights have been used.

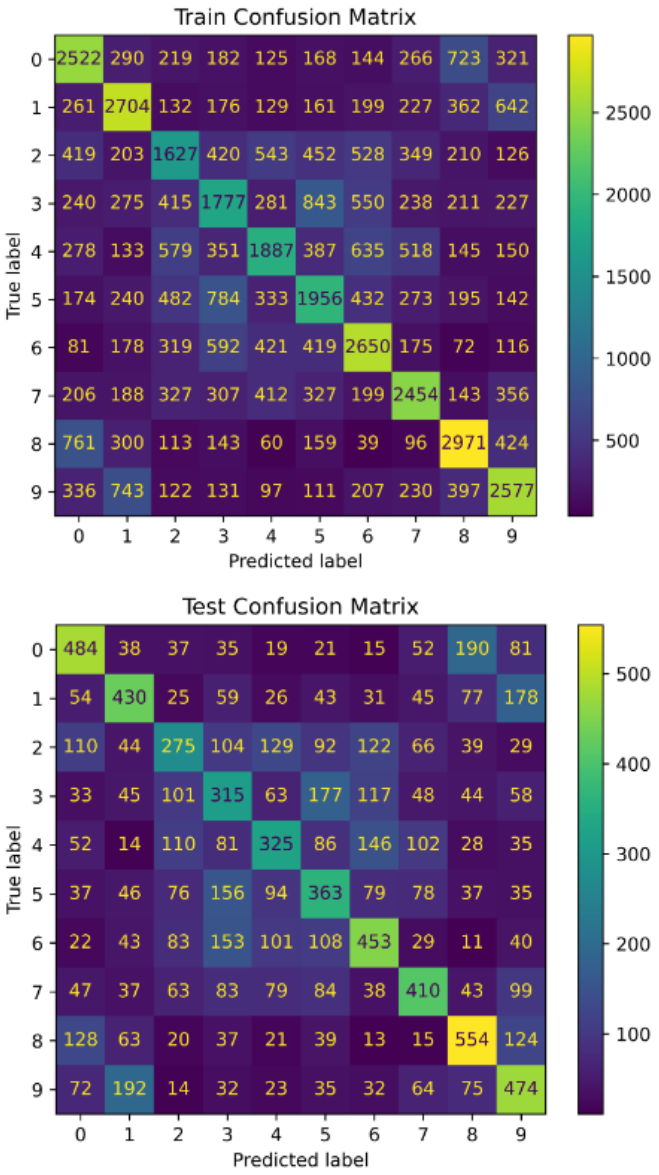

Fig. 11: Train and Test Confusion Matrix of fully connected linear model.

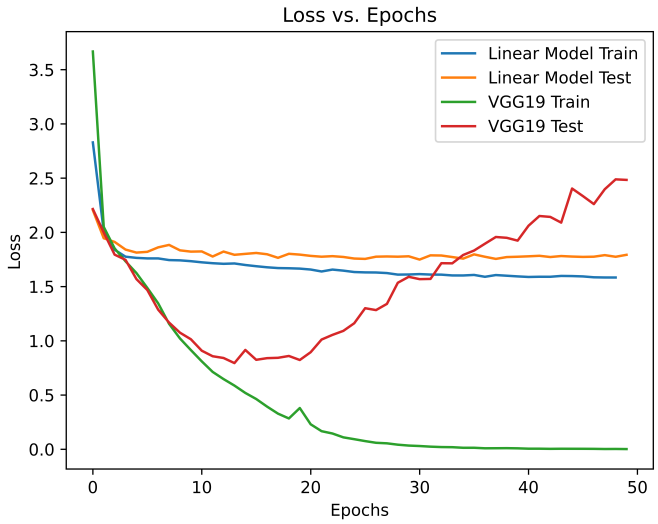

Fig. 12: Training and Testing Loss of VGG19 model compared to the fully connected linear model.

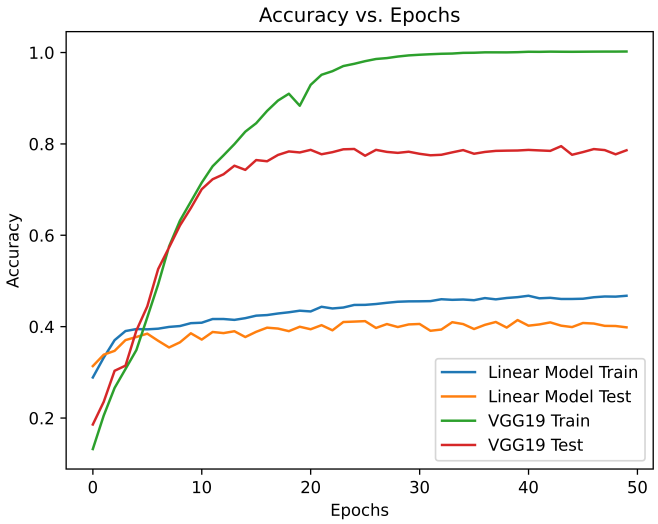

Fig. 13: Training and Testing Accuracy of VGG19 modelcompared to the fully connected linear model.

1) Adding Batch Normalization: The original VGG19 network did not have any batch normalization layers. But with advancements made in the field of deep learning, it was found that adding batch normalization layers improves the performance of network. Following this, I added the batch normalization layers in my VGG19 implementation. The number of trainable parameters increased from $38,952,010$ to $38,958,922$. A plot comparing the training and testing loss of the two models is shown in Figure 15.

As it can be seen from the plot, there is not much improvement in the training loss, but the batch normalization layers help to curb the overfitting in the model as evident from the test loss. The confusion matrix for the VGG19 (batchnorm) model is shown in Figure 16.

2) Data Augmentation: From the results and plots till now, we can see that the VGG19 model easily overfits to the training data. This is because the number of training examples in the dataset is far less than the number of trainable parameters in the model. One way to increase the number of training examples is to apply data augmentation. I have implemented data augmentation techniques like horizontally flipping the image, vertically flipping the image, adding Gaussian noise and changing the intensity of the image. Results of different data augmentation methods on a given input image is shown in Figure 17. A plot comparing the training and testing loss for the models trained with and without data augmentation is shown in Figure 18.

From the plots, we can see that the magnitude of loss

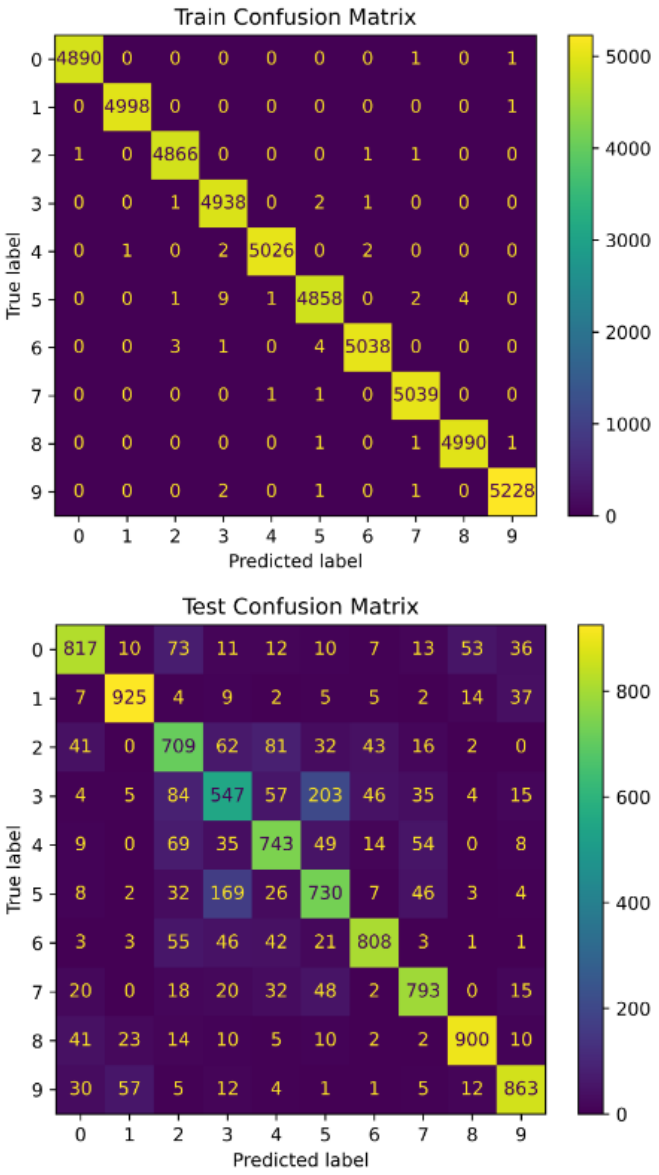

Fig. 14: Train and Test Confusion Matrix of VGG19 model.

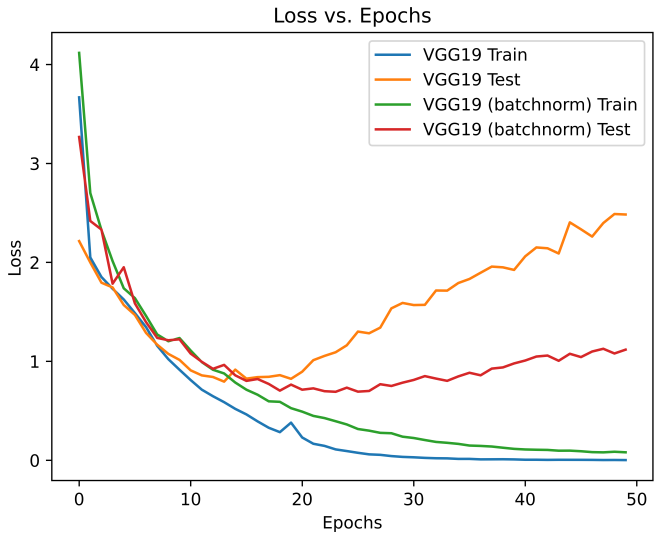

Fig. 15: Training and Testing Loss of VGG19 and VGG19 (batchnorm) models.

increases after adding data augmentation, as the number of training samples increases. But adding data augmentation helps to control the overfitting in the model.

3) Curriculum Learning: One problem that I came across while applying data augmentation to the VGG19 models is that, since I am not using any pre-trained weights, the models are not able to directly train on the augmented training data. This effect is more pronounced in the VGG19 model.

To avoid this problem, I devised a curriculum learning approach, similar to the curriculum employed in schools and colleges. Students in younger grades are taught easier concepts as compared to students in older grades who are taught much complicated and involved concepts. Similarly, a network in the initial training phase is exposed to easier training data, and the difficulty of the training data (data augmentation) increases with the training iterations.

I implemented curriculum learning as follows: in the initial 5 epochs, the network is trained on the training data as it is, without any augmentation. This ensures that all the layers of the network get a good training start. In the next 5 epochs, the probability of encountering an augmented training sample is increased to 20%. Then, in the next 20 epochs the probability is increased to 50%, and finally in the last 20 epochs, the probability reaches 80%.

The advantages of using a curriculum for data augmentation can be seen in the Figures 19, 20, 21, 22. Figure 19 and 20 show that VGG19 model trained with just data augmentation is not able to learn anything as compared to training with a curriculum for data augmentation. Similarly, Figures 21 and 22 show that VGG19 (batchnorm) model trained with curriculum learning gives better performance (both, in terms of lower train and test loss and higher train and test accuracy) as compared to training with just data augmentation.

The confusion matrices for all these variants of the VGG19 models is shown in Figures 25, 27, 26, 28.

Finally, after applying all these strategeis to improve the accuracy of the model, we see that the VGG19 (batchnorm) model outperforms the VGG19 model in Figures 23 and 24.

C. ResNet, ResNeXt, DenseNet

In an attempt to improve the accuracy and efficiency of the image classification, ResNet, ResNeXt and DenseNet models were implemented. I chose to implement the ResNet-34[2], ResNeXt-50[3], and DenseNet-121[4] variants of the corresponding models. The total number of trainable parameters for these

[2]Implementation following this article.
[3]Implementation following this article.
[4]Implementation following this article.

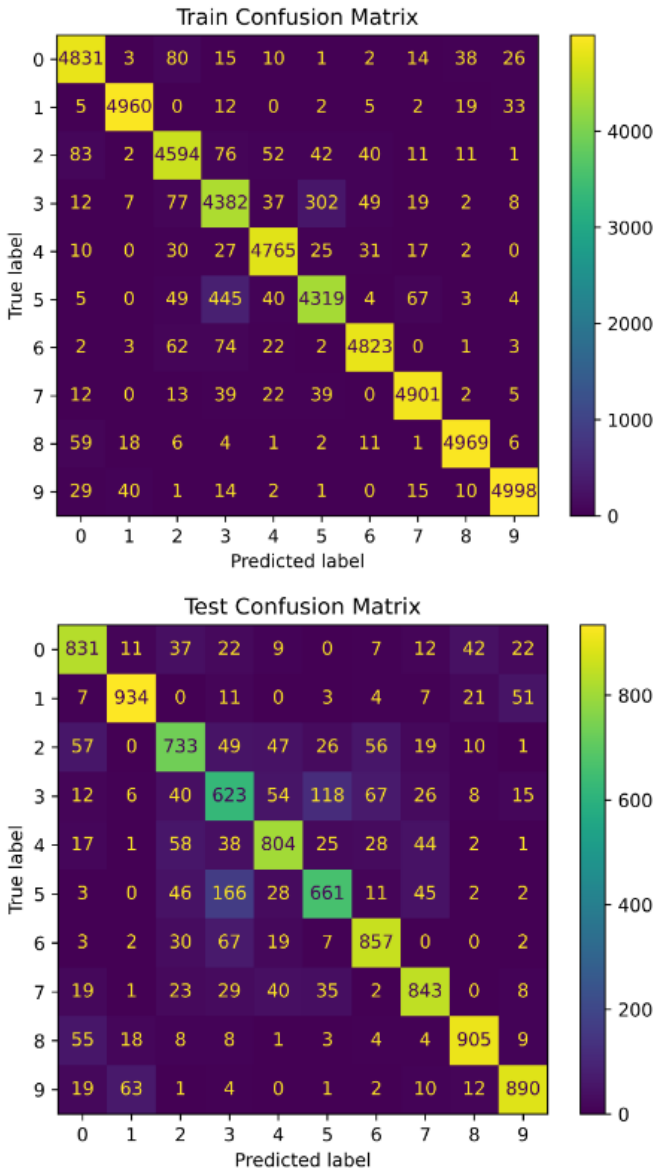

Fig. 16: Train and Test Confusion Matrix of VGG19 (batch-norm) model.

networks are around 21.3M for ResNet-34, 23M for ResNeXt-50 and 7.2M for DenseNet. Block diagrams of these models is shown in Figures 39, 40 and 41.

Since these network architectures are efficient as compared to the previously implemented networks, they tend to overfit easily on the given training data, even after using data augmentation and curriculum. To control this overfitting, a higher regularization strength (weight decay) of 0.001 was used. The current implementation of DenseNet is very memory inefficient, which results in GPUs running out of memory. Hence, for DenseNet, a smaller batch size of 8 was used. This resulted in a drastic increase in training time - from around 3.5 minutes per epoch for ResNet and 7 minutes per epoch for ResNeXt to 50 minutes per epoch for DenseNet. Hence, in order to complete all the trainings in the given time, the number of epochs for DenseNet was reduced to 15. The total time required to train ResNet was around 3 hours, for ResNeXt 6 hours, and for DenseNet 13 hours.

The training and testing loss for all these models is shown in Figure 29 and the accuracy in Figure 30. As it can be observed in Figure 30, the accuracy curve on both train set and test set for ResNet and ResNext is very similar. But looking at Figure 29 we see that ResNeXt overfits faster as compared to ResNet. As for DenseNet, since the training could not be completed for 50 epochs, the results are sub-optimal. For the initial 15 epochs that it was trained, the accuracy on both sets is lower than ResNet and ResNeXt.

The training and testing confusion matrices for the corresponding networks is shown in Figures 31, 32 and 33.

D. Analysis

Figures 34 and 35 show the loss and accuracy of all the models on the test set. Looking at these plots, we can conclude that strategies like adding batch normalization layers, data augmentation and curriculum learning definitely help in improving the accuracy of the models. Models with efficient architectures like ResNet, ResNeXt and DenseNet achieve better performance than other models in less number of epochs, but their training time per epoch is significantly higher. In my current implementation of all these networks, ResNet34 performs the best on the test set, with around 82% accuracy. A comparison of all these models in a tabular form is shown in Figure 36.

E. Future Work

Due to time and compute constraints, I could not perform an extensive hyperparameter search for all the models. For the ResNet, ResNeXt and DenseNet models, I could experiment with only one network architecture. Due to this, the results for all the models in suboptimal, and not exhaustive.Especially in case of ResNet and ResNeXt, implementing ResNet-50 and comparing it with ResNeXt-50 could generate a better comparison study. All these hyperparameter tuning and ablation studies can be conducted to further improve the performance of each of the models. More data augmentation strategies and varied curriculum can also be explored.

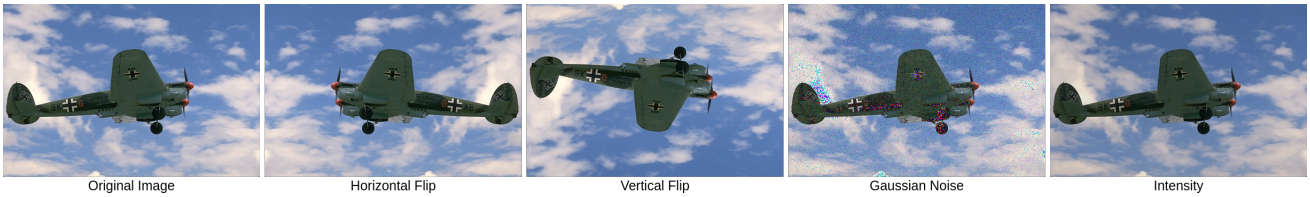

Fig. 17: Data Augmentation strategies.

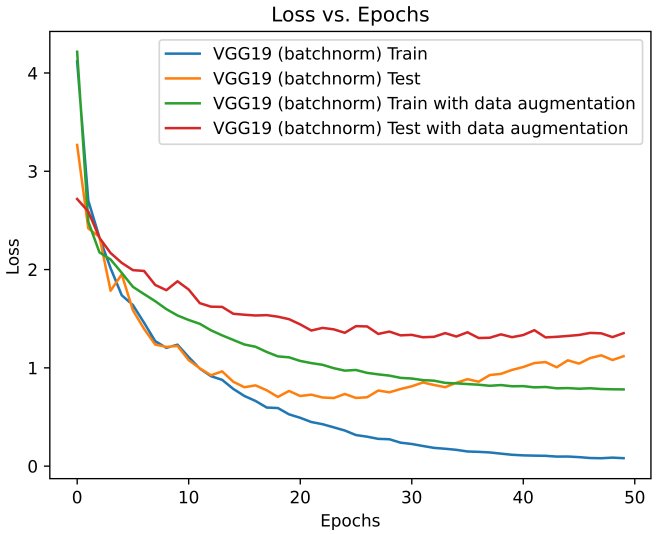

Fig. 18: Training and Testing Loss of VGG19 (batchnorm) model with and without data augmentation.

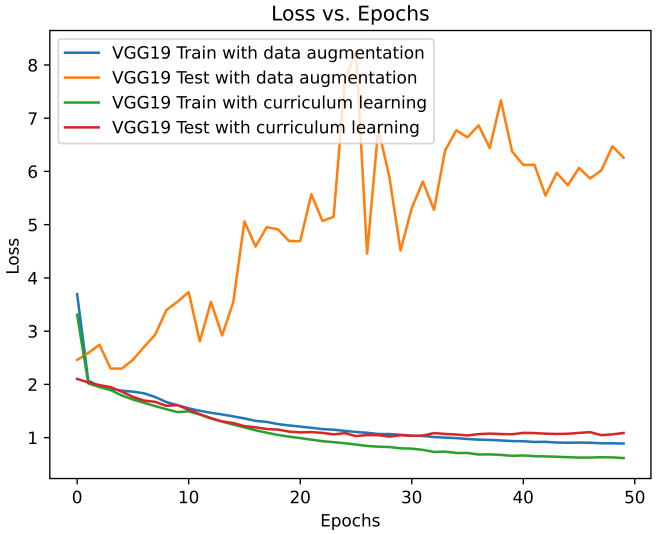

Fig. 19: Training and Testing Loss of VGG19 model with and without curriculum for data augmentation.

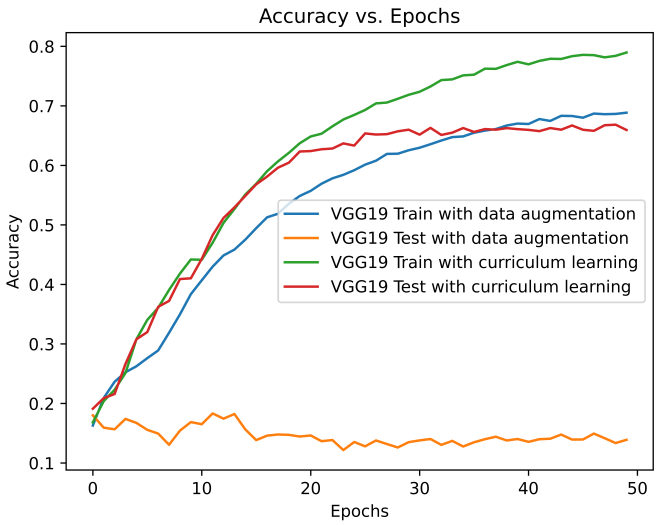

Fig. 20: Training and Testing Accuracy of VGG19 model with and without curriculum for data augmentation.

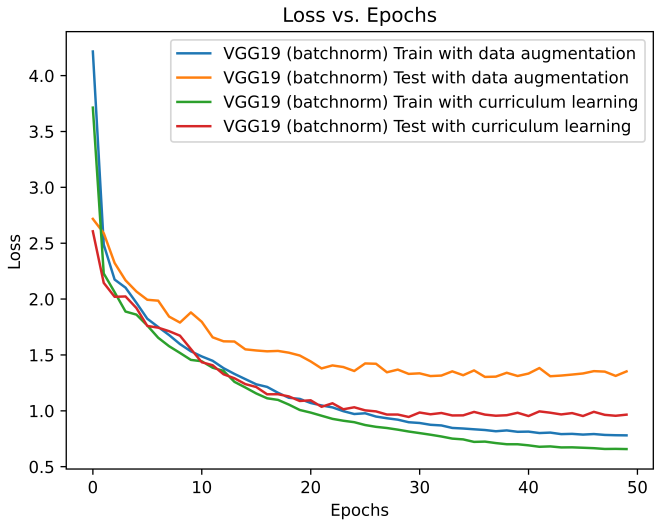

Fig. 21: Training and Testing Loss of VGG19 (batchnorm) model with and without curriculum for data augmentation.

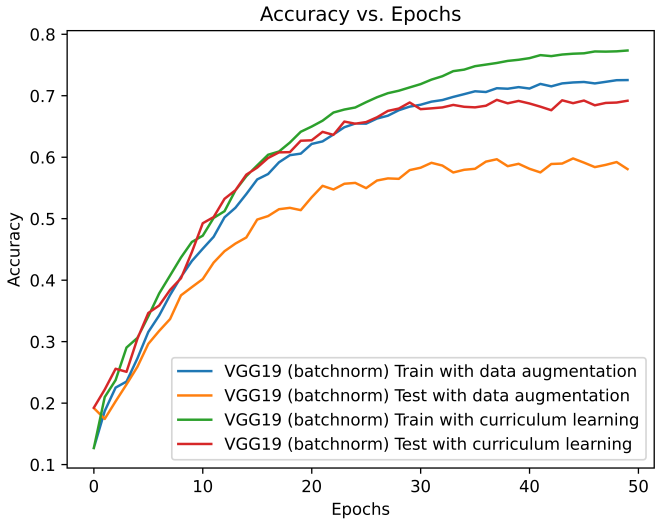

Fig. 22: Training and Testing Accuracy of VGG19 (batchnorm) model with and without curriculum for data augmentation.

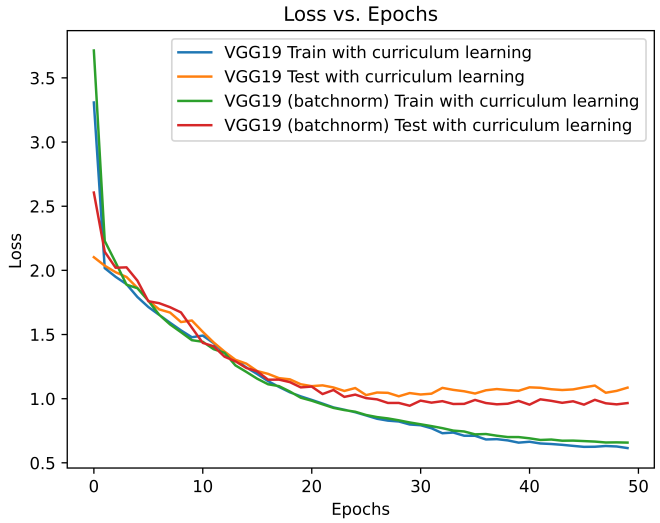

Fig. 23: Training and Testing Loss of VGG19 and VGG19 (batchnorm) models after applying all the strategies to improve accuracy.

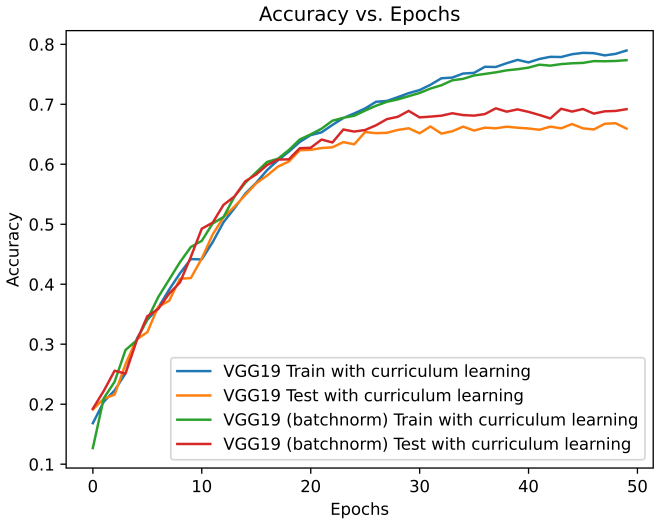

Fig. 24: Training and Testing Accuracy of VGG19 and VGG19 (batchnorm) models after applying all the strategies to improve accuracy.

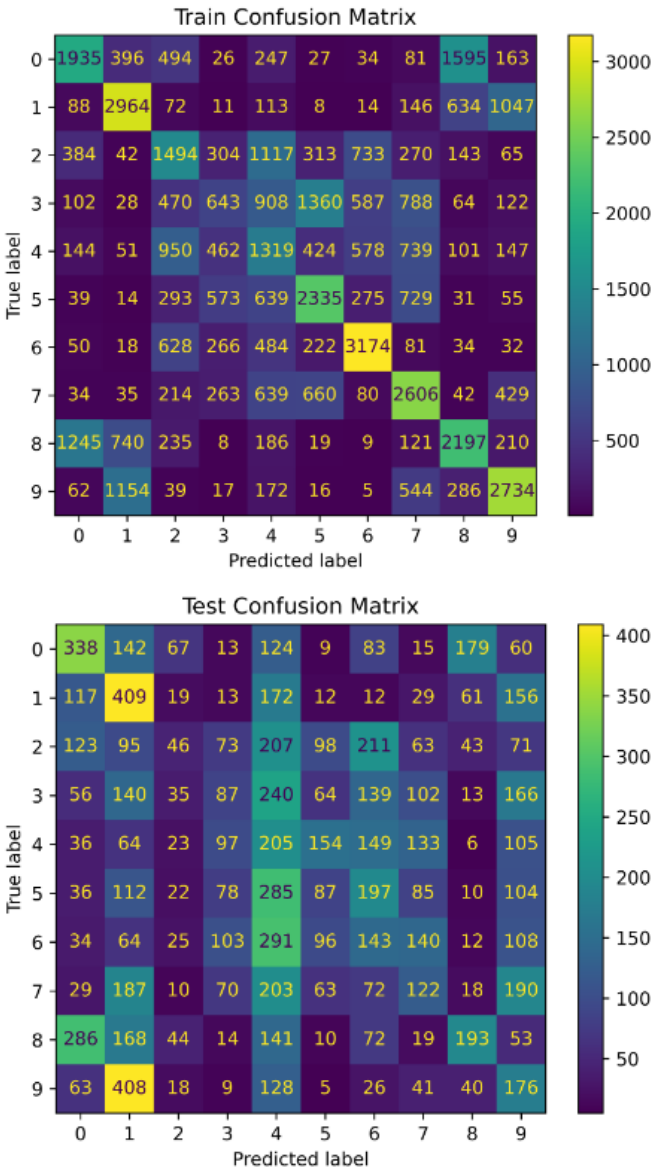

Fig. 25: Train and Test Confusion Matrix of VGG19 model trained with data augmentation.

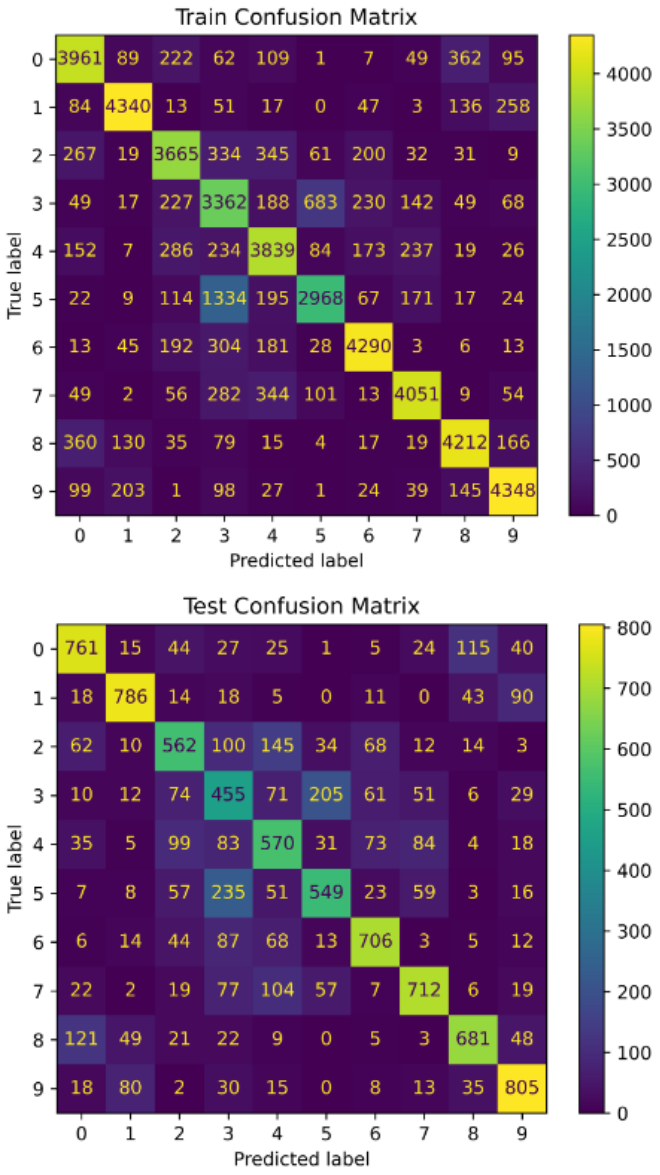

Fig. 26: Train and Test Confusion Matrix of VGG19 model trained with curriculum learning.

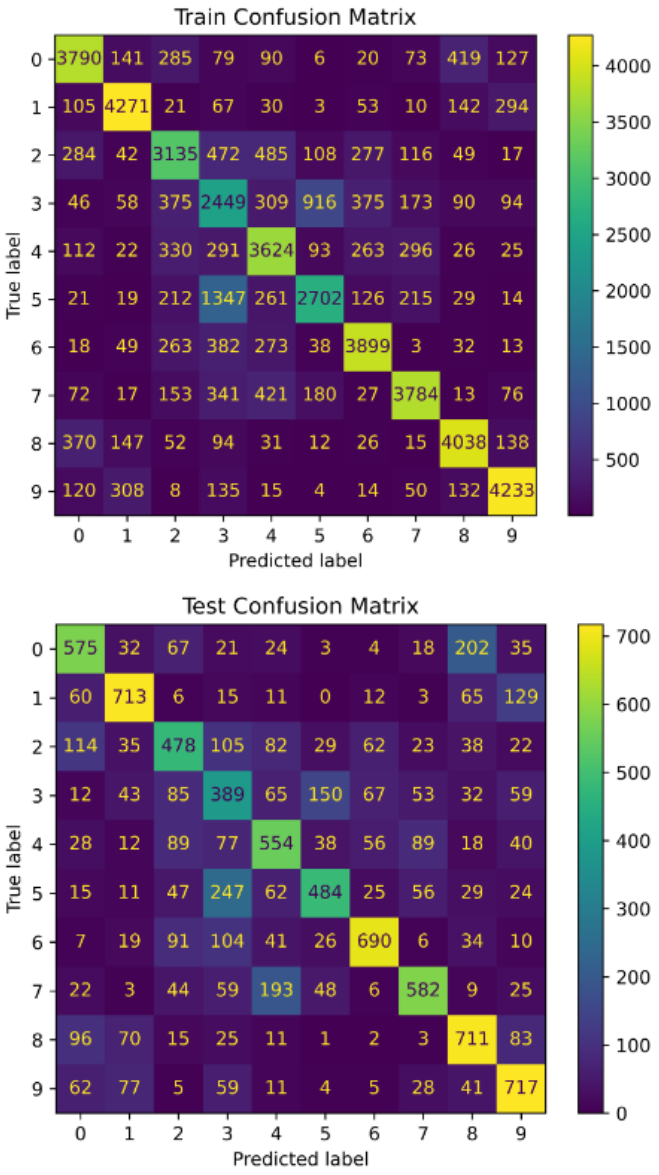

Fig. 27: Train and Test Confusion Matrix of VGG19 (batch-norm) model trained with data augmentation.

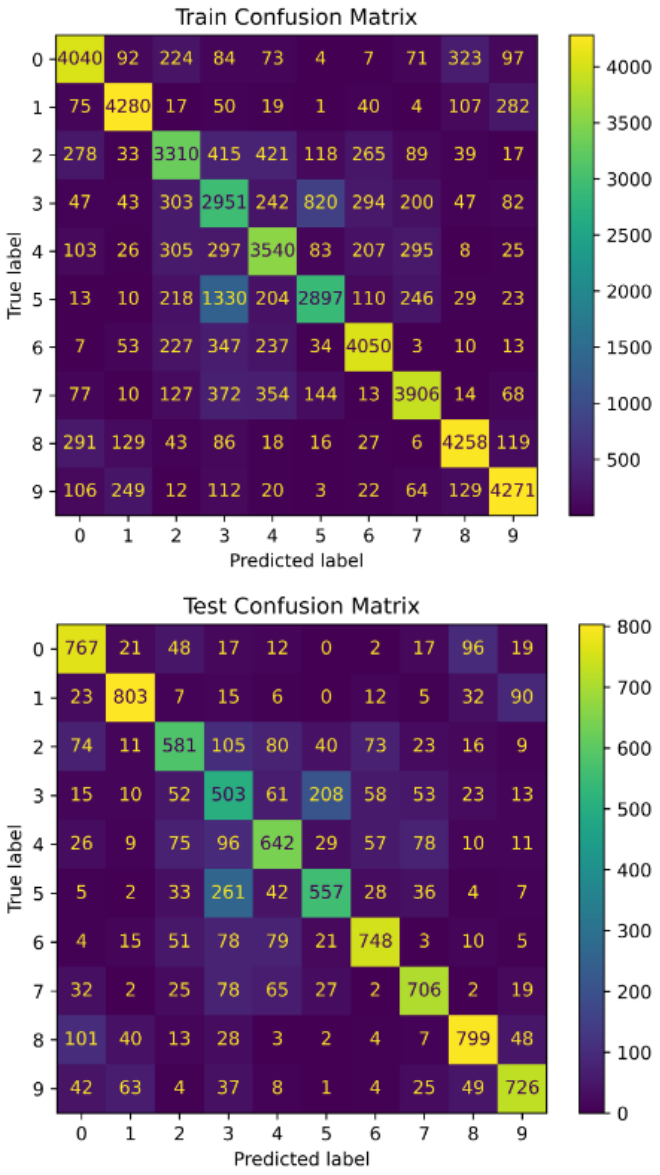

Fig. 28: Train and Test Confusion Matrix of VGG19 (batch-norm) model trained with curriculum learning.

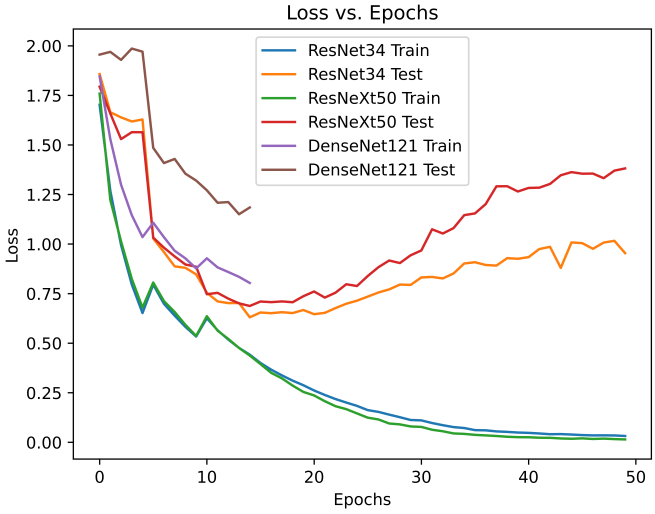

Fig. 29: Training and Testing Loss for ResNet, ResNeXt, DenseNet models.

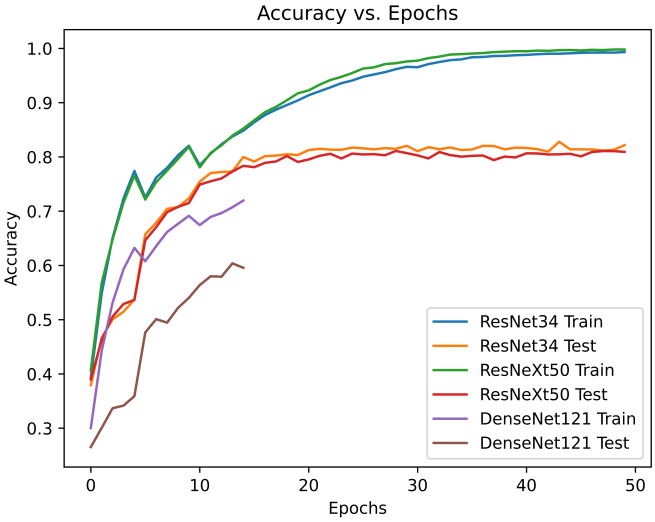

Fig. 30: Training and Testing Accuracy for ResNet, ResNeXt, DenseNet models.

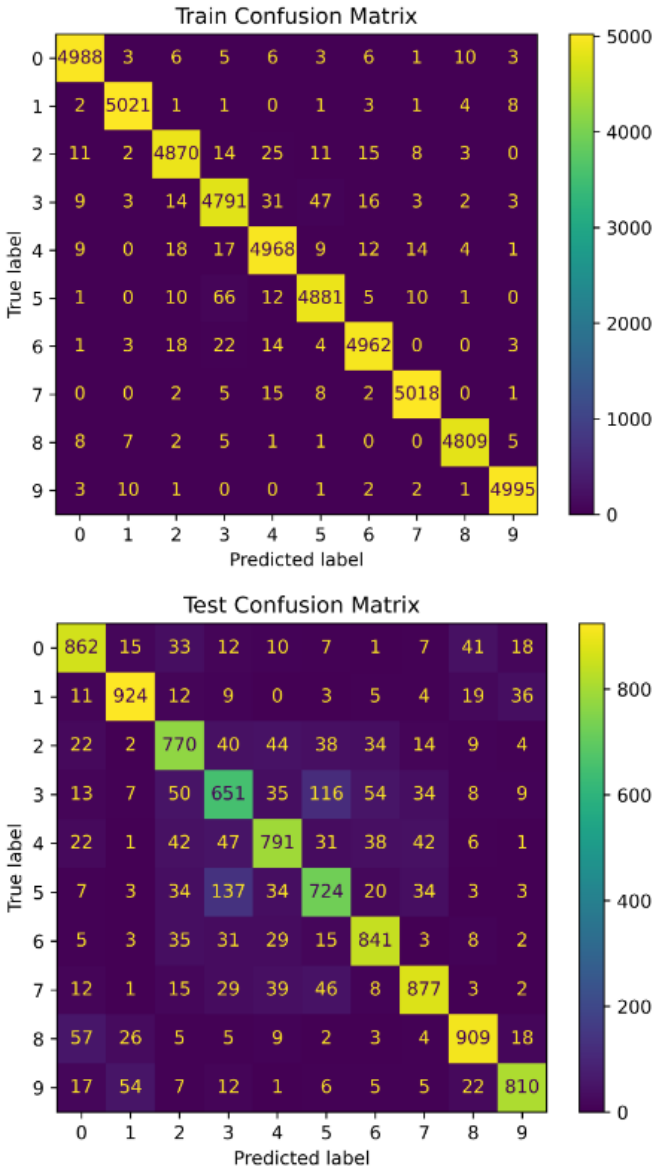

Fig. 31: Train and Test Confusion Matrix for ResNet-34.

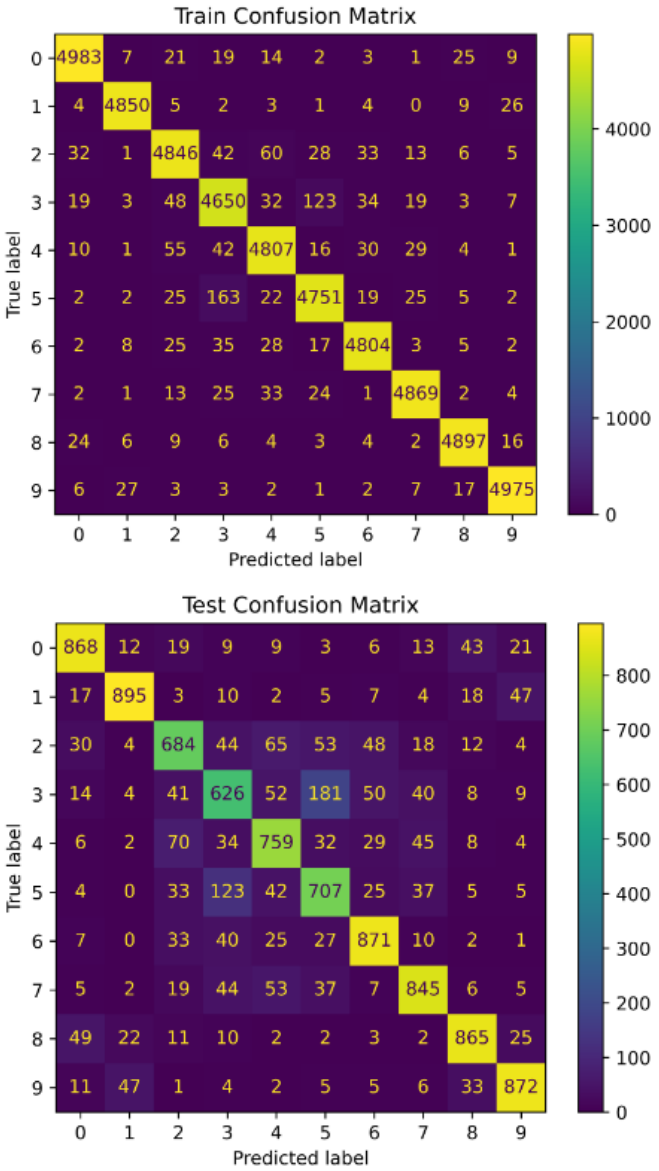

Fig. 32: Train and Test Confusion Matrix for ResNeXt-50.

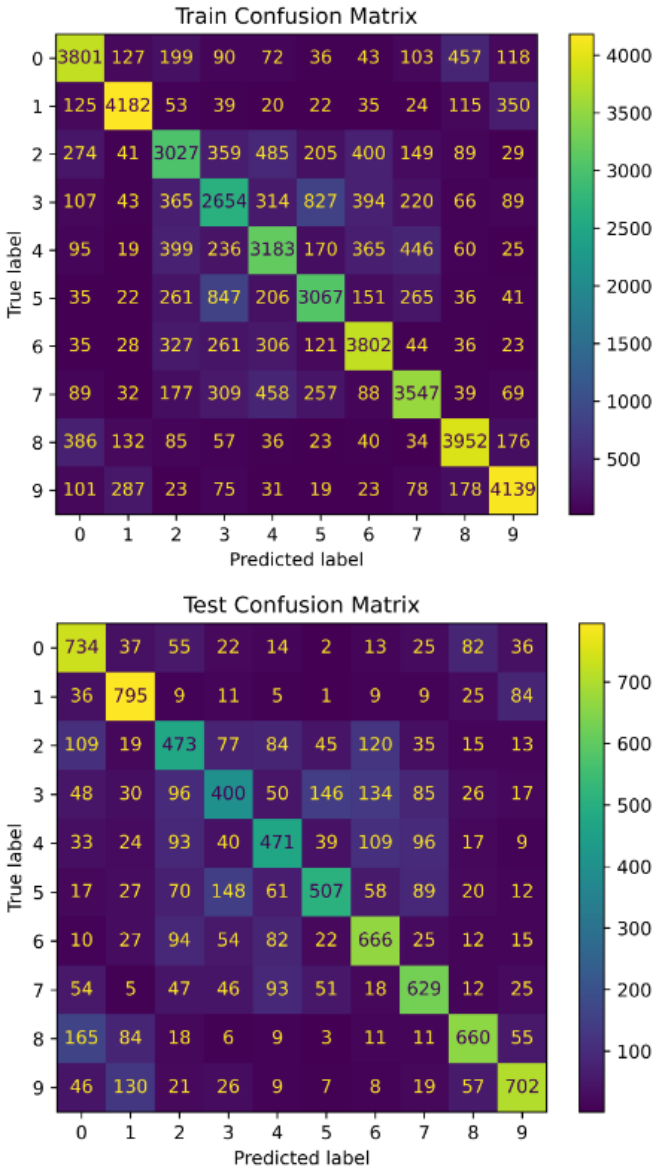

Fig. 33: Train and Test Confusion Matrix for DenseNet121.

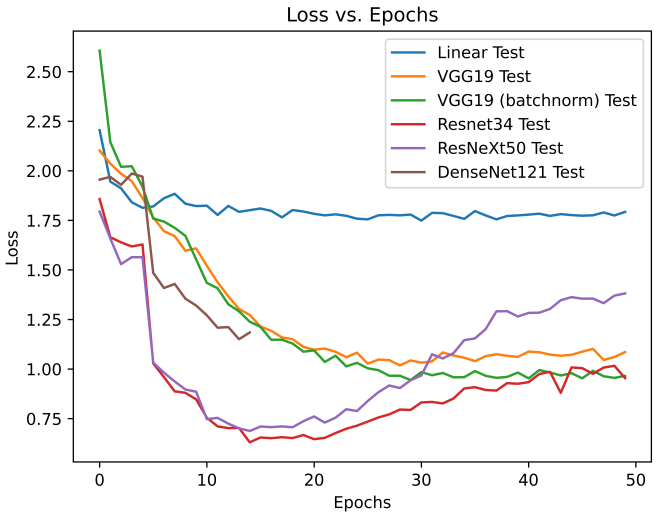

Fig. 34: Testing Loss for all models.

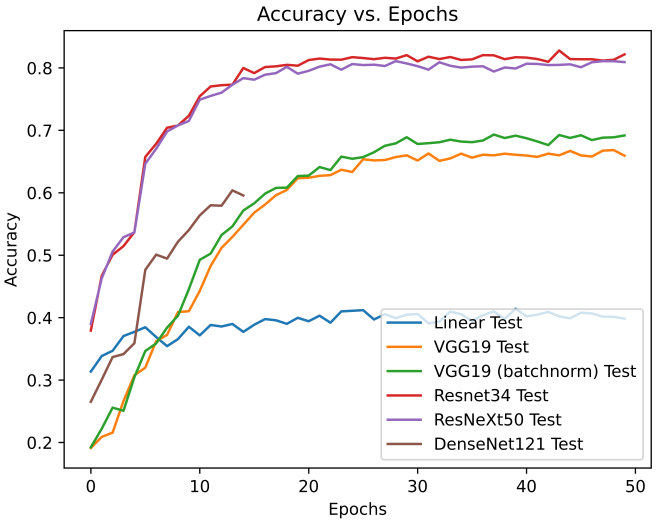

Fig. 35: Testing Accuracy for all models.

Model	Number of trainable parameters	Train Accuracy	Test Accuracy	Train Time	Inference Time (avg)
Fully Connected Linear Model	37.8M	46.55%	41.42%	20 minutes	1 ms
VGG19	38.9M	78.39%	66.83%	42 minutes	0.4 ms
VGG19 (batchnorm)	38.9M	77.21%	69.31%	48 minutes	1.6 ms
ResNet34	21.3M	99.22%	82.78%	3 hours	2.5 ms
ResNeXt50	23M	99.79%	81.08%	6 hours	4.2 ms
DenseNet121 (15 epochs)	7.2M	70.71%	60.38%	13 hours	26.1 ms

Fig. 36: Table summarizing all models.

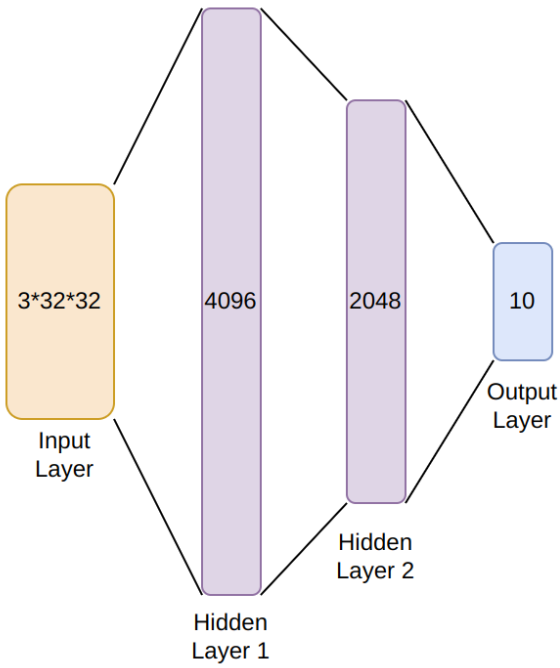

Fig. 37: Block Diagram of fully connected linear model.

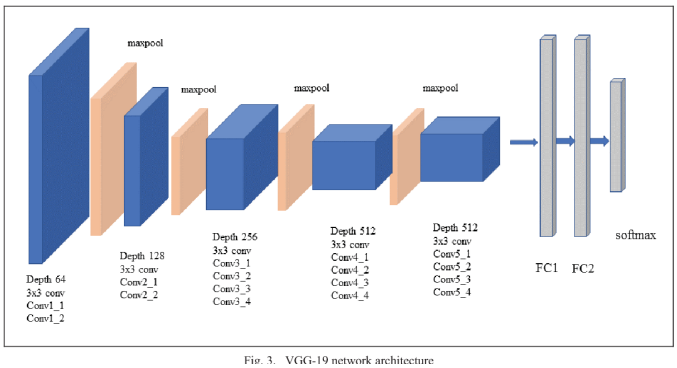

Fig. 3. VGG-19 network architecture

Fig. 38: Block Diagram of VGG19 model.

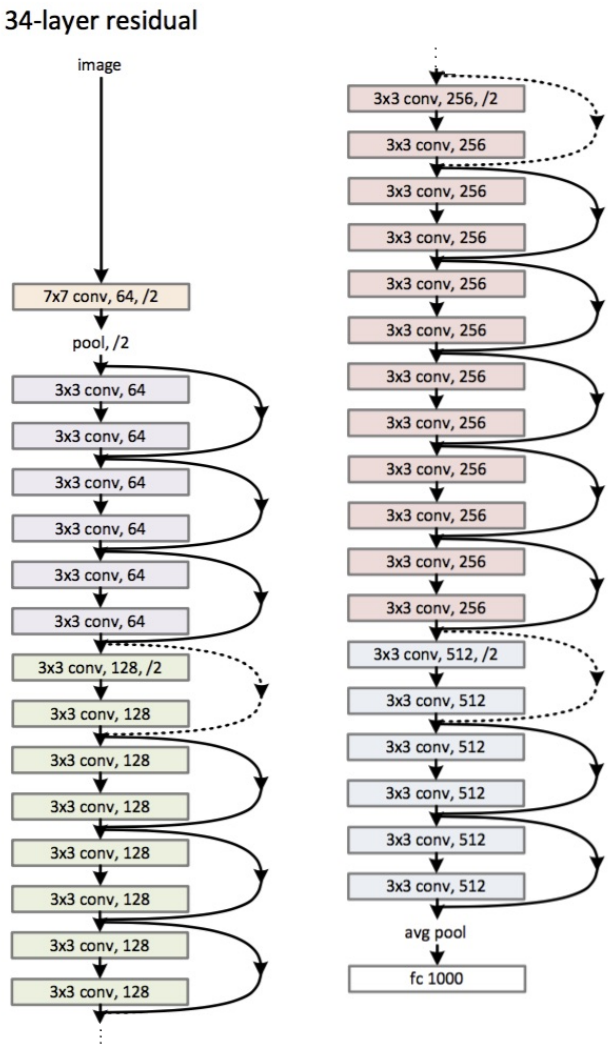

Fig. 39: Block Diagram of ResNet34 model.

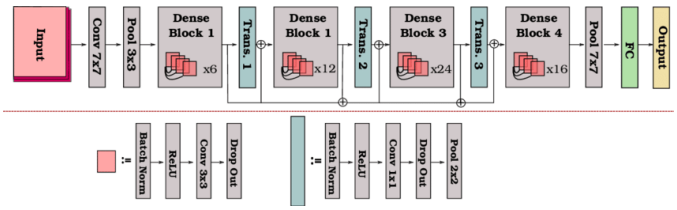

Fig. 41: Block Diagram of DenseNet121 model.

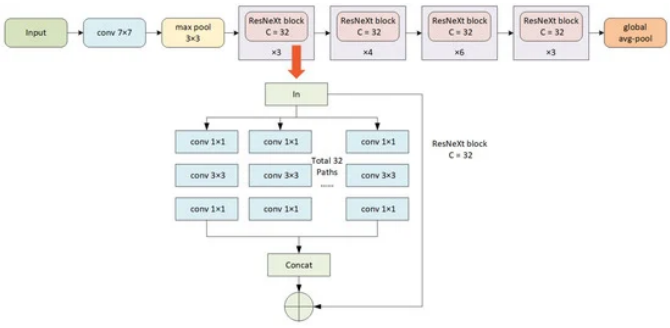

Fig. 40: Block Diagram of ResNeXt50 model.