RBE/CSE549 CVHW0 - Alohomora

Butchi Venkatesh Adari
Masters RBE
Worcester Polytechnic Institute
Worcester , MA , 01609
Email: badari@wpi.edu
Using 1 late day

***Abstract*—There are two phases in this Introductory homework . In the first phase a simplified version of probability of boundary detection algorithm to detect edges in the given images. To achieve this I used Oriented DOG , Leung-Malik , etc.. filter banks and extracted Texture , Brightness and Color features from the Images. By using these filters along with canny and sobel baselines yielded in better edge detection. In the second phase I implemented four Neural Networks and trained them to compare their architectures and accuracies on CIFAR10 dataset.**

I. Phase 1 : Shake My Boundary

In Phase 1 of this homework, I implemented a boundary detector algorithm called 'pb-lite' which is a simplified version of the probability of boundary detection algorithm presented in [1]. The output of the pb-lite algorithm is a per-pixel probability of boundary. Fig. 1 gives an overview of the algorithm. First, we filter the input images with the filter bank and apply k-means clustering to develop the texture, brightness and color maps for an input image. We then compute the texture gradient T , brightness gradient B and color gradient C. We then calculate the Chi-square distances with the help of half-disk filters. Finally, we combine the information from the features with baseline methods such as Sobel and Canny to get the pb-lite output.

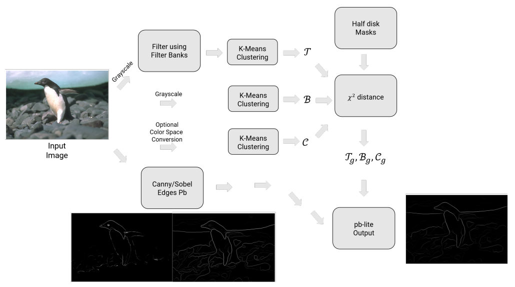


Fig. 1. Overview of pb lite pipeline.

A. Filters

Images are filtered using a collection of filters to measure texture properties and to aggregate regional texture and brightness distributions. Three different sets of filter banks are created for this purpose.

1) Oriented DOG Filters: A simple DOG Filter is created by convolving a Sobel Filter and a Gaussian kernel. This filter can then be rotated at different scales to find a series of oriented DoG Filters. The Gaussian function is given by $G(x;\sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}$.

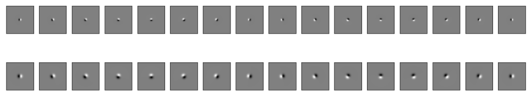


Fig. 2. Oriented DoG Filters

2) Leung-Malik Filters: The Leung-Malik filters are a set of multi-scale, multi orientation filter bank with 48 filters. Filters consists of first and second order derivatives of Gaussian with 3 different scales , 6 orientations and 8 laplacian filters and 8 basic gaussian kernels.

$$G(x, y; \sigma, \gamma, \psi, \lambda, \theta) = \frac{1}{2\pi\sigma^2} e^{-\frac{x'^2+\gamma^2 y'^2}{2\sigma^2}} \cos\left(\frac{2\pi x'}{\lambda} + \psi\right)$$

where

$$x' = x\cos(\theta) + y\sin(\theta)$$

$$y' = -x\sin(\theta) + y\cos(\theta)$$

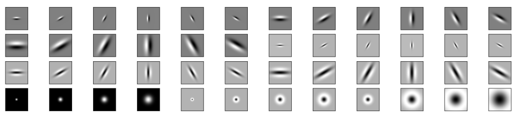


Fig. 3. Leung-Malik Filters

3) Gabor Filters: A Gabor filter is a gaussian kernel function modulated by sinusoidal plane wave.

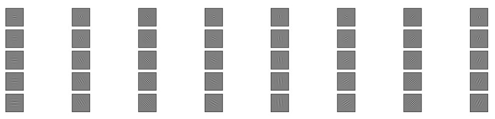


Fig. 4. Leung-Malik Filters

B. Texton , Brightness , Color Map

Textron maps, brightness maps, and color maps play crucial roles in computer graphics and data visualization. A Textron map is utilized for detailed texturing, defining surface properties like color and reflectivity in 3D rendering. On the other hand, a brightness map, represented in grayscale, influences the intensity of lighting or shading across a surface. It guides how light interacts with different parts of a 3D object based on variations in brightness. Color maps, or colormaps, are sets of colors mapped to specific values and are often employed in data visualization. These maps represent scalar values, with distinct colors indicating different data values. Together, these mapping techniques contribute to the visual richness and accuracy of rendered graphics and enhance the representation of data in visualizations.

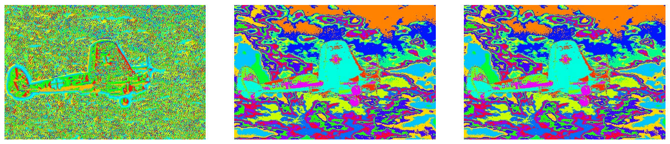


Fig. 5. $(\mathcal{T}, \mathcal{B}, \mathcal{C})$ for Image 1

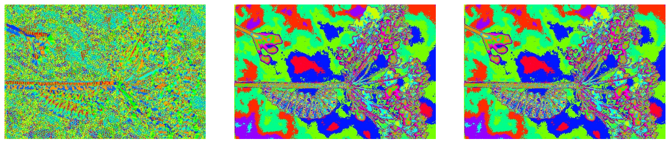


Fig. 6. $(\mathcal{T}, \mathcal{B}, \mathcal{C})$ for Image 2

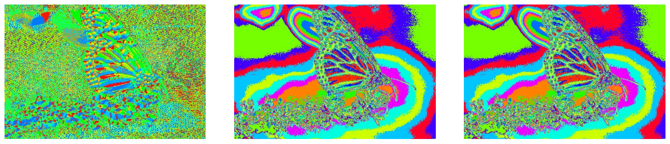


Fig. 7. $(\mathcal{T}, \mathcal{B}, \mathcal{C})$ for Image 3

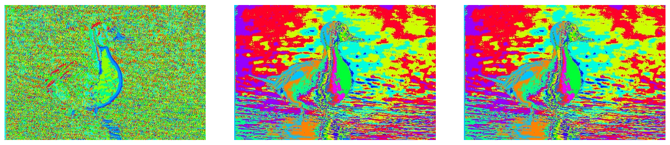


Fig. 8. $(\mathcal{T}, \mathcal{B}, \mathcal{C})$ for Image 4

1) Results:

II. Phase 2 : Deep Dive on Deep Learning

In this phase, we implemented multiple CNNs to perform classification task on the CIFAR-10 dataset. The dataset had 10 classes, 50000 training images and 10000 test images.

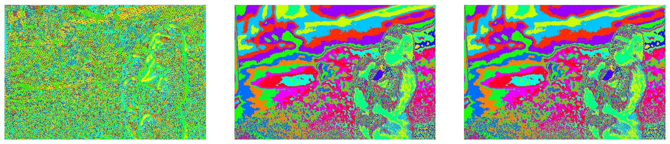


Fig. 9. $(\mathcal{T}, \mathcal{B}, \mathcal{C})$ for Image 5

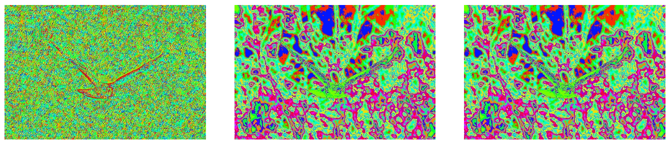


Fig. 10. $(\mathcal{T}, \mathcal{B}, \mathcal{C})$ for Image 6

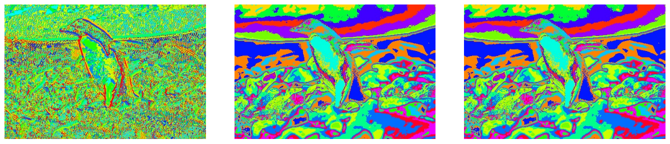


Fig. 11. $(\mathcal{T}, \mathcal{B}, \mathcal{C})$ for Image 7

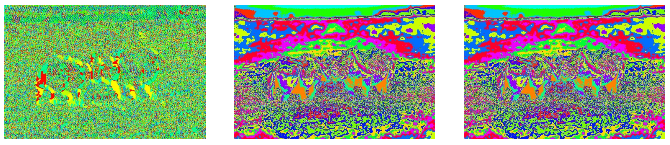


Fig. 12. $(\mathcal{T}, \mathcal{B}, \mathcal{C})$ for Image 8

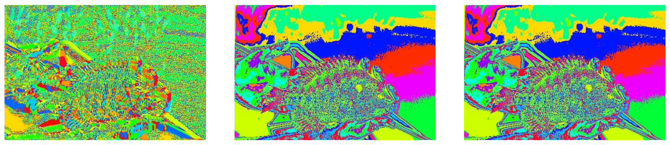


Fig. 13. $(\mathcal{T}, \mathcal{B}, \mathcal{C})$ for Image 9

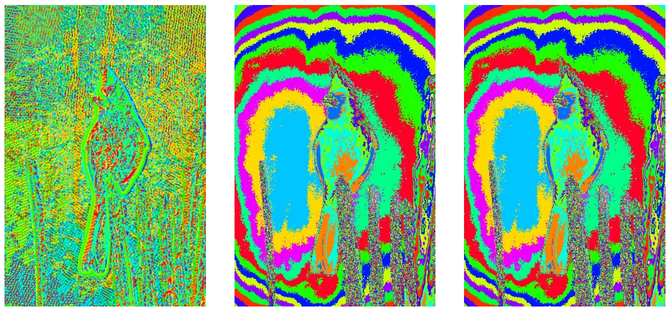


Fig. 14. $(\mathcal{T}, \mathcal{B}, \mathcal{C})$ for Image 10

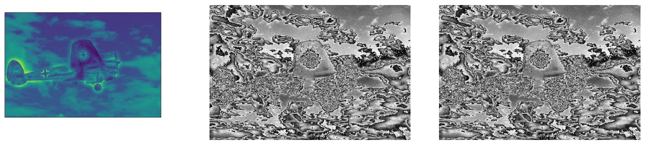


Fig. 15. $(\mathcal{T}, \mathcal{B}, \mathcal{C})$ for Image 1

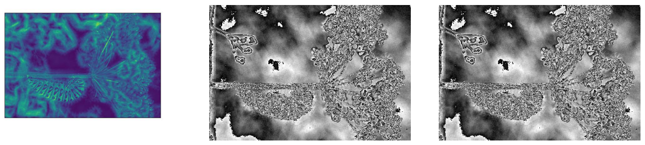


Fig. 16. $(\mathcal{T},\mathcal{B},\mathcal{C})$ for Image 2

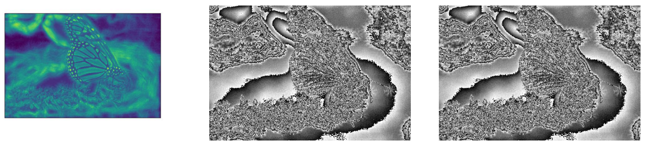


Fig. 17. $(\mathcal{T},\mathcal{B},\mathcal{C})$ for Image 3

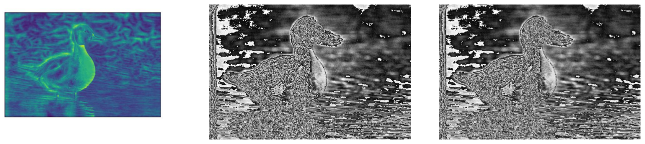


Fig. 18. $(\mathcal{T},\mathcal{B},\mathcal{C})$ for Image 4

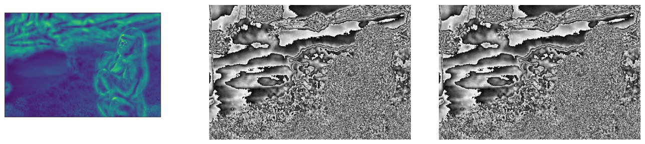


Fig. 19. $(\mathcal{T},\mathcal{B},\mathcal{C})$ for Image 5

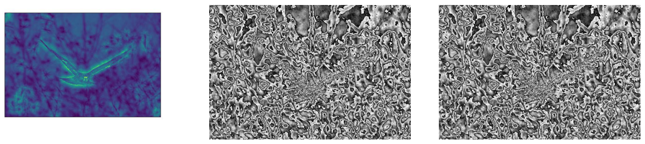


Fig. 20. $(\mathcal{T},\mathcal{B},\mathcal{C})$ for Image 6

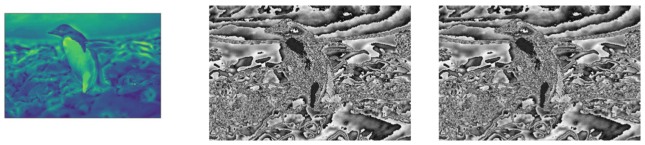


Fig. 21. $(\mathcal{T},\mathcal{B},\mathcal{C})$ for Image 7

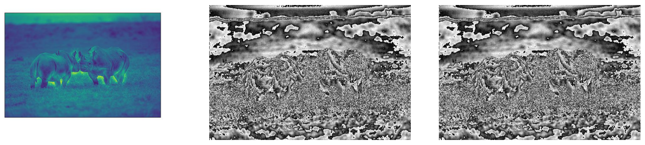


Fig. 22. $(\mathcal{T},\mathcal{B},\mathcal{C})$ for Image 8

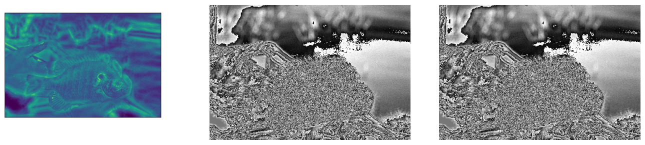


Fig. 23. $(\mathcal{T},\mathcal{B},\mathcal{C})$ for Image 9

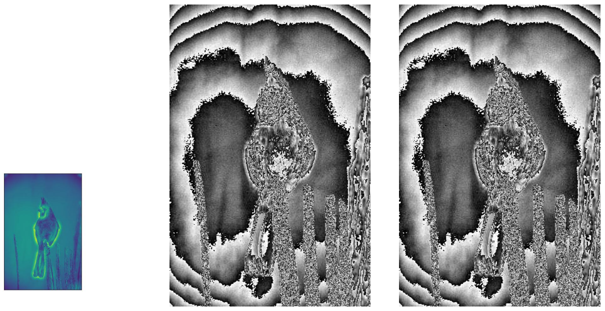


Fig. 24. $(\mathcal{T},\mathcal{B},\mathcal{C})$ for Image 10

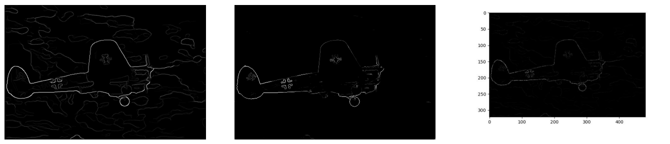


Fig. 25. Canny , sobel , pblite for Image 1

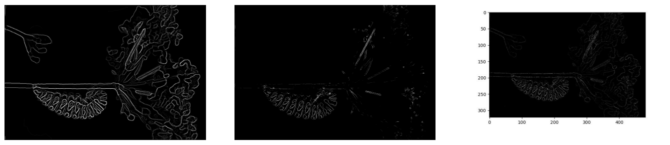


Fig. 26. Canny , sobel , pblite for Image 2

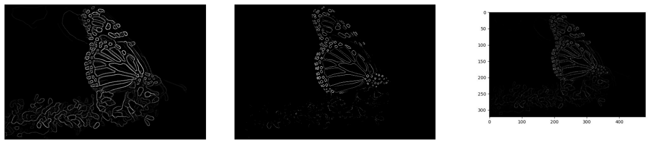


Fig. 27. Canny , sobel , pblite for Image 3

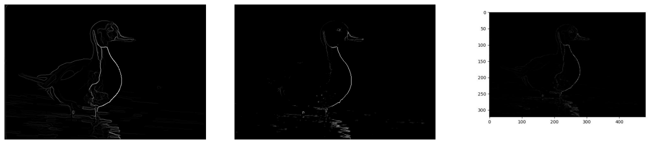


Fig. 28. Canny , sobel , pblite for Image 4

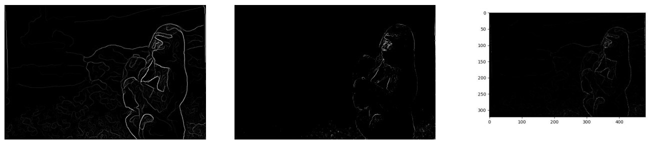


Fig. 29. Canny , sobel , pblite for Image 5

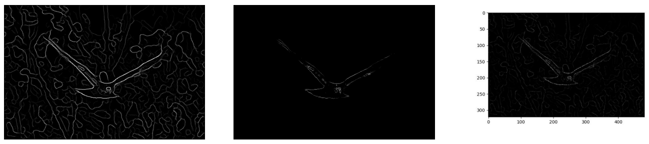


Fig. 30. Canny , sobel , pblite for Image 6

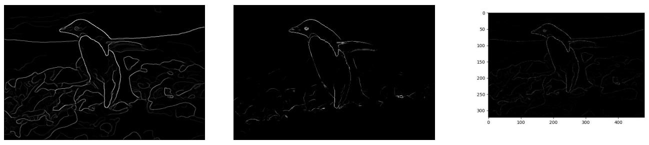


Fig. 31. Canny , sobel , pblite for Image 7

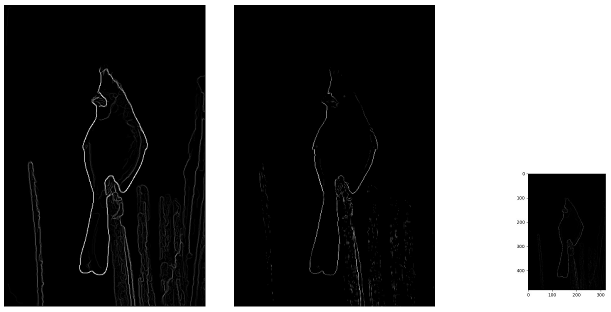


Fig. 34. Canny , sobel , pblite for Image 10

A. Custom Model

Network Architecture: The model takes an input image with three color channels (like red, green, and blue) and applies three sets of operations. First, it uses convolutional layers to detect features in the image. Then, it applies hyperbolic tangent activation functions to introduce non-linearity. After that, it uses average pooling to reduce the spatial dimensions of the features. Batch normalization is applied to normalize the output. This process is repeated twice with different numbers of channels.

Next, the network flattens the output and passes it through two fully connected layers with hyperbolic tangent activations. These layers are designed to make sense of the detected features and produce a final output, the size of which is determined by the value specified for 'OutputSize'. Overall, this architecture is a Convolutional Neural Network (CNN) commonly used for image classification tasks. The architecure is shown in Fig 35.

B. ResNet Model

Network Architecture: This ResNet architecture consists of blocks and residual blocks for feature extraction from images. Each block includes convolutional layers with batch normalization and ReLU activation, followed by max-pooling

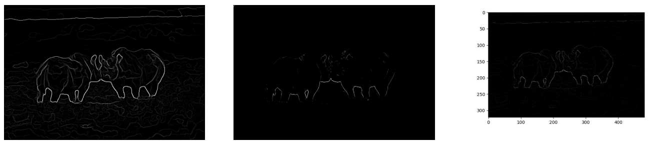


Fig. 32. Canny , sobel , pblite for Image 8

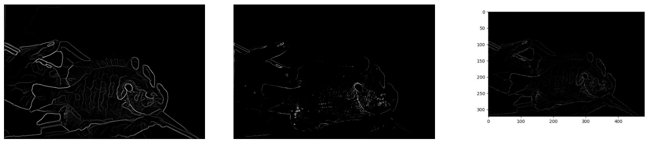


Fig. 33. Canny , sobel , pblite for Image 9

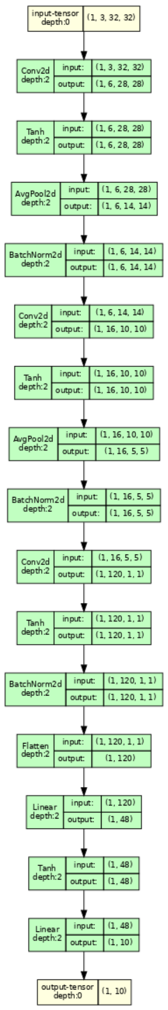

input-tensor depth:0 (1, 3, 32, 32)
Conv2d depth:2 input: (1, 3, 32, 32) output: (1, 6, 28, 28)
Tanh depth:2 input: (1, 6, 28, 28) output: (1, 6, 28, 28)
AvgPool2d depth:2 input: (1, 6, 28, 28) output: (1, 6, 14, 14)
BatchNorm2d depth:2 input: (1, 6, 14, 14) output: (1, 6, 14, 14)
Conv2d depth:2 input: (1, 6, 14, 14) output: (1, 16, 10, 10)
Tanh depth:2 input: (1, 16, 10, 10) output: (1, 16, 10, 10)
AvgPool2d depth:2 input: (1, 16, 10, 10) output: (1, 16, 5, 5)
BatchNorm2d depth:2 input: (1, 16, 5, 5) output: (1, 16, 5, 5)
Conv2d depth:2 input: (1, 16, 5, 5) output: (1, 120, 1, 1)
Tanh depth:2 input: (1, 120, 1, 1) output: (1, 120, 1, 1)
BatchNorm2d depth:2 input: (1, 120, 1, 1) output: (1, 120, 1, 1)
Flatten depth:2 input: (1, 120, 1, 1) output: (1, 120)
Linear depth:2 input: (1, 120) output: (1, 48)
Tanh depth:2 input: (1, 48) output: (1, 48)
Linear depth:2 input: (1, 48) output: (1, 10)
output-tensor depth:0 (1, 10)

Fig. 35. Custom Model Architecture

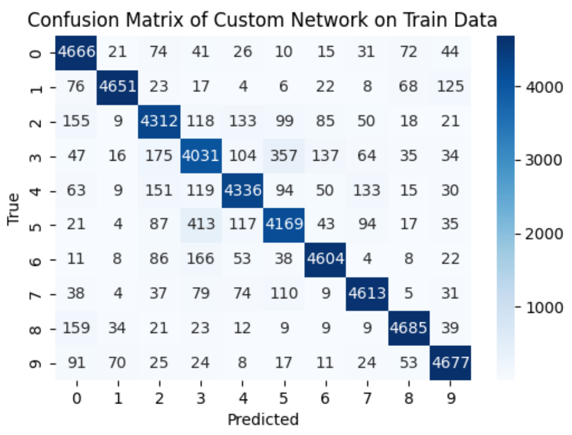


Fig. 36. Confusion Matrix on Train Dataset.

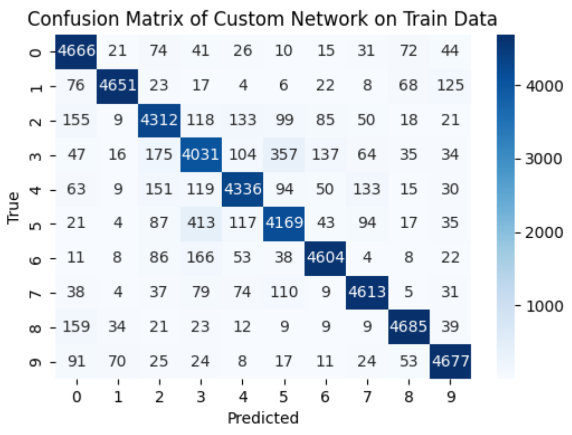


Fig. 37. Confusion Matrix on Test Dataset.

for dimension reduction. The residual blocks contain two convolutional layers with batch normalization and ReLU. The final layer applies global max-pooling, flattening, dropout, and a linear layer for classification. The architecture shown in Fig 40 and the confusion matrices for both on Train and Test datasets are shown in Fig 41, Fig 42 respectively.

C. DenseNet Model

Network Architecture: The ‘conv2d_block‘ class defines a block of operations used in a DenseNet. It performs batch nor-

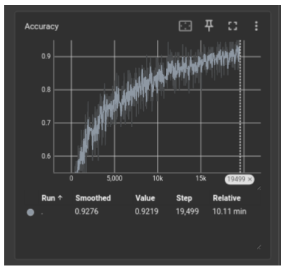


Fig. 38. Training Accuracy

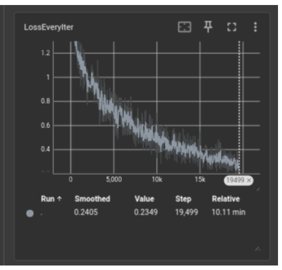


Fig. 39. Training Loss

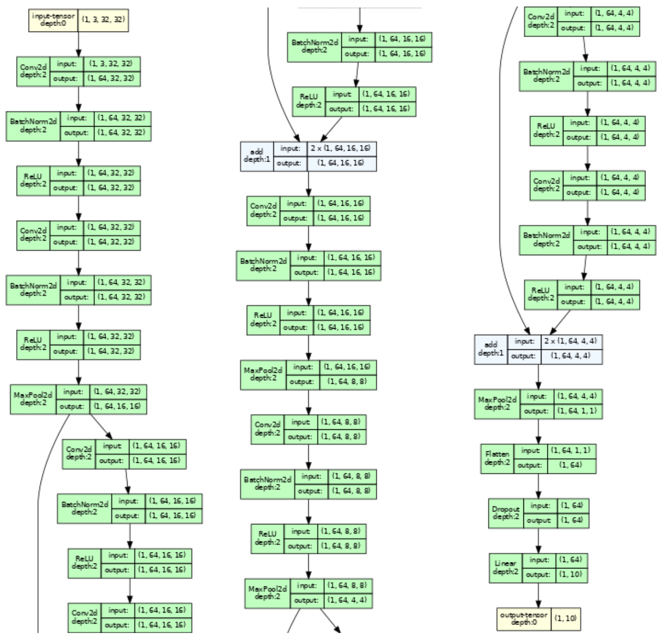


Fig. 40. ResNet Model Architecture

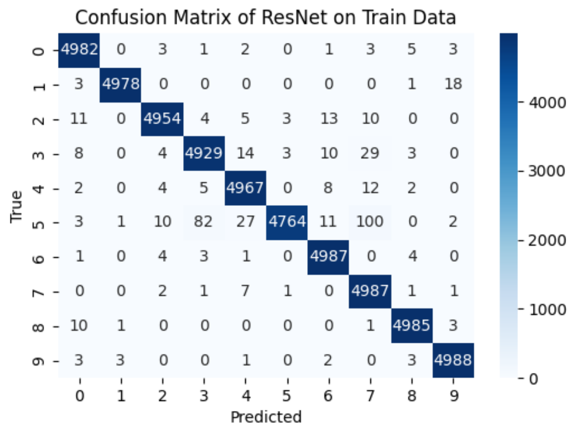


Fig. 41. Confusion Matrix on Train Dataset.

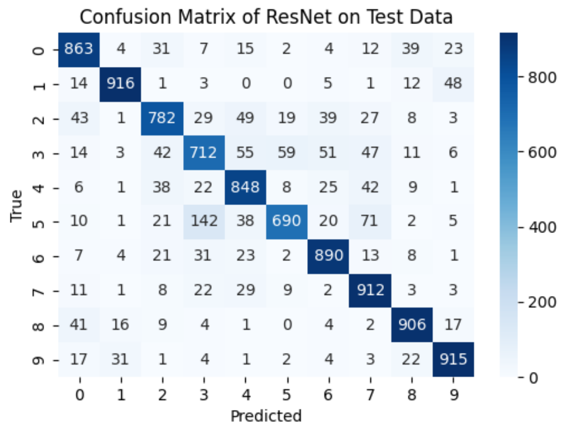
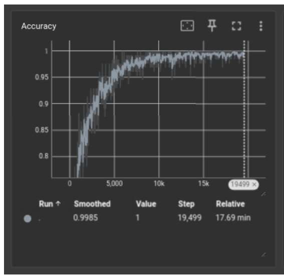
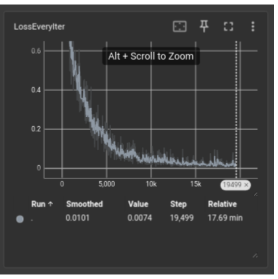


Fig. 42. Confusion Matrix on Test Dataset.



	Run	Smoothed	Value	Step	Relative
●	.	0.9985	1	19,499	17.69 min

Fig. 43. ResNet Accuracy



	Run	Smoothed	Value	Step	Relative
●	.	0.0101	0.0074	19,499	17.69 min

Fig. 44. ResNet Loss

malization, ReLU activation, and two convolutional operations (1x1 and 3x3), with the output being the concatenation of the input and the result of these operations.

The ‘DenseNet‘ class implements a DenseNet architecture for image classification. It consists of multiple dense blocks connected by transition layers. Each dense block contains several ‘conv2d_block‘ layers. The architecture ends with a classifier that includes batch normalization, average pooling, flattening, and a linear layer for the final classification into 10 classes. The architecture shown in Fig 45 and the confusion matrices for both on Train and Test datasets are shown in Fig 46, Fig 47 respectively.

D. ResNeXT Model

Network Architecture: The ‘res_block‘ class defines a residual block for the ResNeXT architecture, employing grouped convolutions with a bottleneck structure. It takes input channels, cardinality, base width, and stride as parameters.

The ‘ResNeXT‘ class implements ResNeXT for image classification. It includes an initial convolution layer, batch normalization, and three dense blocks. Each dense block contains multiple ‘res_block‘ layers, forming the architecture. The final classifier consists of average pooling, flattening, and a linear layer for classification into 10 classes.

ResNeXT uses grouped convolutions to enhance feature learning and diversity. The number of groups and base width parameters control the architecture’s characteristics. The architecture shown in Fig 50 and the confusion matrices for both on Train and Test datasets are shown in Fig 51, Fig 52 respectively.

Hyperparam - Setting	
Optimizer	AdamW
Learning Rate	1e-3
Weight Decay	1e-4
MiniBatchSize	128
Max Epochs Trained	50

TABLE I
HyperParameters made for Training

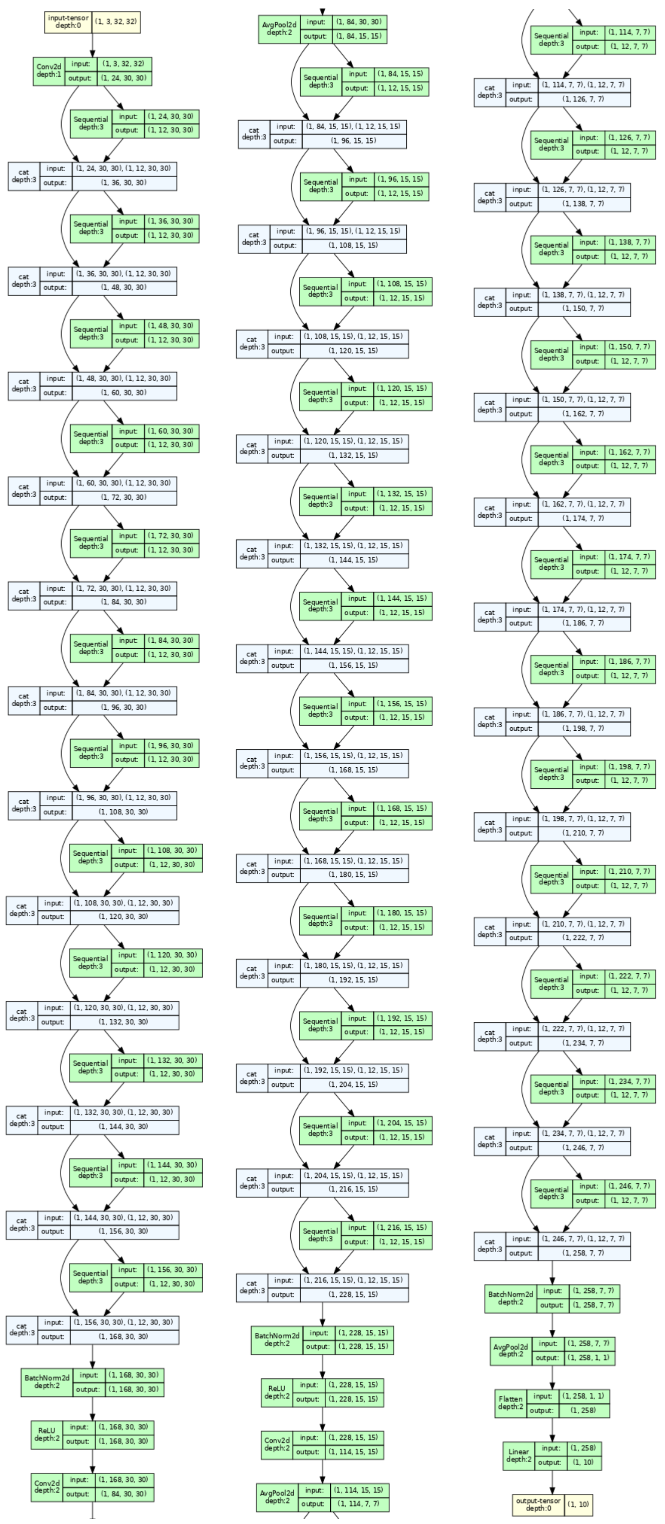


Fig. 45. ResNet Model Architecture

E. Comparison

The graphs which are shown above are for relative comparison of the variance of train and test accuracy and loss, some of

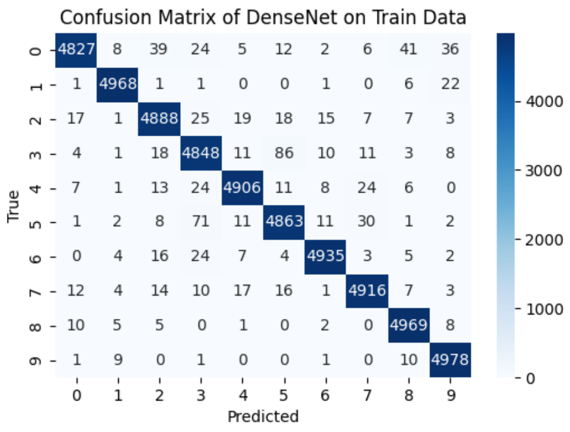


Fig. 46. Confusion Matrix on Train Dataset.

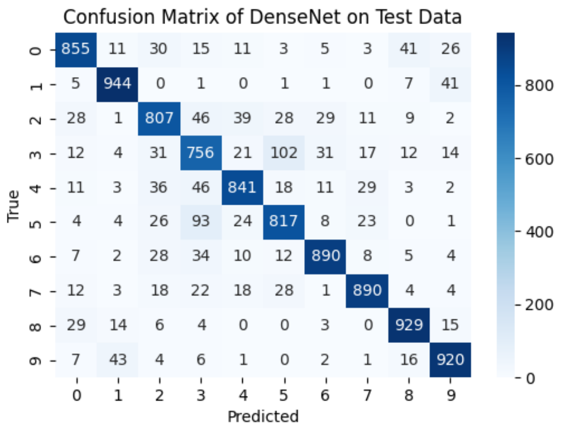
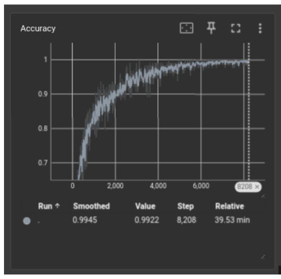


Fig. 47. Confusion Matrix on Test Dataset.



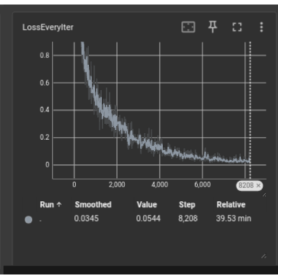


Fig. 48. DenseNet Accuracy Fig. 49. DenseNet Loss

Model Name	No. Parameters	Total Size(MB)	Params Size (MB)
CIFAR10Model	57,574	0.36	0.22
ResNet	2,61,962	5.44	4.43
ResNext	3,270,794	42.50	30.02
DenseNet	489,112	58.19	56.31

TABLE II
SIZE OF THE MODELS

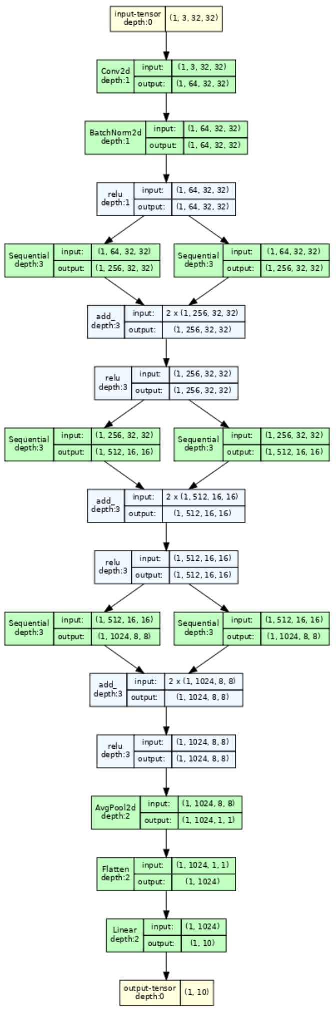

input-tensor depth:0 (1, 3, 32, 32)
Conv2d depth:1 input: (1, 3, 32, 32) output: (1, 64, 32, 32)
BatchNorm2d depth:1 input: (1, 64, 32, 32) output: (1, 64, 32, 32)
relu depth:1 input: (1, 64, 32, 32) output: (1, 64, 32, 32)
Sequential depth:3 input: (1, 64, 32, 32) output: (1, 256, 32, 32)
Sequential depth:3 input: (1, 64, 32, 32) output: (1, 256, 32, 32)
add_ depth:3 input: 2 x (1, 256, 32, 32) output: (1, 256, 32, 32)
relu depth:3 input: (1, 256, 32, 32) output: (1, 256, 32, 32)
Sequential depth:3 input: (1, 256, 32, 32) output: (1, 512, 16, 16)
Sequential depth:3 input: (1, 256, 32, 32) output: (1, 512, 16, 16)
add_ depth:3 input: 2 x (1, 512, 16, 16) output: (1, 512, 16, 16)
relu depth:3 input: (1, 512, 16, 16) output: (1, 512, 16, 16)
Sequential depth:3 input: (1, 512, 16, 16) output: (1, 1024, 8, 8)
Sequential depth:3 input: (1, 512, 16, 16) output: (1, 1024, 8, 8)
add_ depth:3 input: 2 x (1, 1024, 8, 8) output: (1, 1024, 8, 8)
relu depth:3 input: (1, 1024, 8, 8) output: (1, 1024, 8, 8)
AvgPool2d depth:2 input: (1, 1024, 8, 8) output: (1, 1024, 1, 1)
Flatten depth:2 input: (1, 1024, 1, 1) output: (1, 1024)
Linear depth:2 input: (1, 1024) output: (1, 10)
output-tensor depth:0 (1, 10)

Fig. 50. ResNeXT Model Architecture

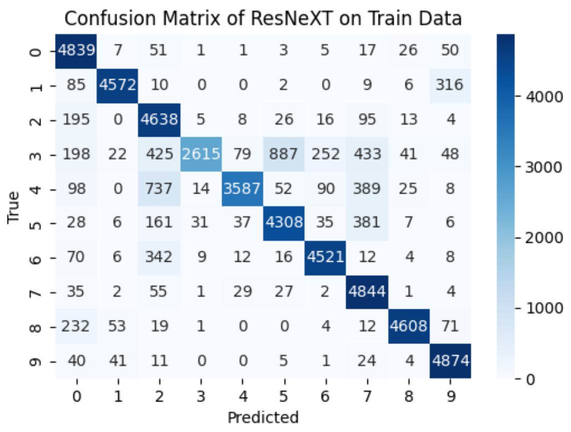


Fig. 51. Confusion Matrix on Train Dataset.

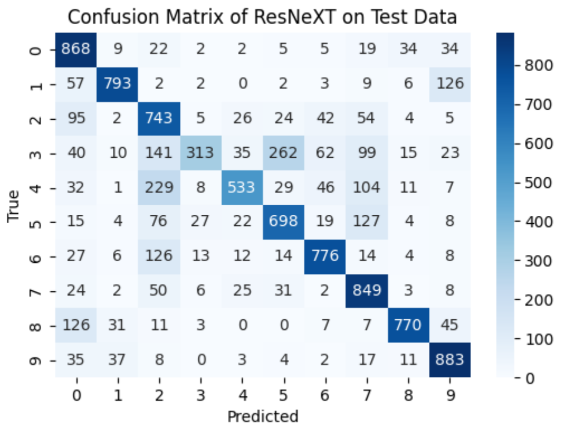
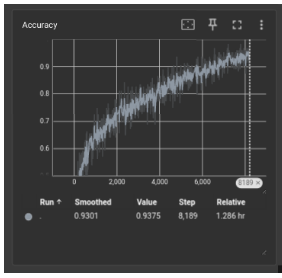


Fig. 52. Confusion Matrix on Test Dataset.



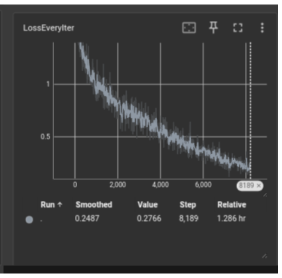


Fig. 53. ResNeXT Accuracy Fig. 54. ResNeXT Loss

the models I have trained overnight for good number of epochs generated decent accuracy which you can find below. Those checkpoints are shared along with the code. The models we have used are heavy as I wanted to test standard Architectures The architectures that I made are generalized which can be changed to get different variants of the networks. For these networks, which I have trained please refer to the table below. All the models are trained on GTX1660Ti 6GB VRAM with i9 11th gen CPU (16GB RAM) and the inference is also performed on the same device. The hyperparameters used for training are listed in TABLE 1. The details of the models are listed in TABLE 2.

III. Conclusion

Phase I: During the initial phase, I extensively utilized traditional image processing techniques to apply conventional filters on the images. One notable challenge I encountered involved debugging an issue related to the application of Gaussian first and second derivatives along specific axes. Despite experimenting with variations in taking first or second derivatives along different axes, the overall results remained largely unchanged. It was interesting to observe that the filters presented in reference [1] differed from those I implemented. This experience served as a valuable lesson in understanding the significant impact that altering such parameters can have.

Phase II: The subsequent phase focused on addressing code-related issues that were within the realm of resolution. Reviewing academic papers, even at a cursory level, consumed a considerable amount of time, prompting a realization about the importance of dedicating more time to refreshing fundamental concepts. The examination of the Custom Convolution network and other networks revealed prevalent issues of overfitting. To mitigate this, augmentation techniques and improved training strategies can be employed to optimize hyperparameters. Currently, our approach involves Random Crop Augmentation, which has shown slight improvements in certain cases.

Acknowledgment

Thank you professor!