HW0 : Alohomora

Soumik Saswat Patnaik
Dept. of Robotics Engineering,
Worcester Polytechnic Institute
Worcester,MA
email : sspatnaik@wpi.edu
Using 2 late days

***Abstract*—This report is aimed at presenting my results for Homework0 which introduces the basic concepts of computer vision for the course RBE/CS 549. The report presents the results divided into two parts - Phase 1 and phase 2. In phase 1, we created some of the most commonly used filter banks, namely Derivative of Gaussian, Leung-Malik Filters, and Gabor Filters, which were used to capture the texture properties which in turn are used to create a probability-based edge detector using brightness, color and texture gradients combined with a weighted sum of the traditional sobel and canny methods. In phase 2, we build a deep learning model from scratch to classify images in CIFAR-10 dataset, evaluate it and improve it. Finally, we also make use of the ResNet, ResNeXt, DenseNet models to see and evaluate their performance and present our findings.**

I. Phase 1: Shake My Boundary

A. Filter Banks

Three Banks were created:-

1) Oriented Difference of Gaussian:
Difference of Gaussian (DoG) filters are a set of oriented filters used in image processing to highlight edges, textures, and fine details within an image. These filters are created by subtracting two Gaussian-blurred versions of an image with different scales, resulting in responses that emphasize variations in pixel intensities. In the specific context of a paper, the DoG filter bank comprises filters of size 9x9 at 16 different orientations and across 2 scales (1 and 2). The rotational invariance of DoG filters is achieved by convolving a basic Sobel filter and a Gaussian kernel and subsequently rotating the result. This approach ensures the effectiveness of the filters in detecting features at different orientations. DoG filters find applications in edge detection, object recognition, and feature extraction, serving as a fundamental tool in various image processing and computer vision tasks. Figure 1 visually represents these filters, providing insight into their orientations and scales for a comprehensive understanding of their impact on image features.
2) Leung-Malik Filters: The Leung-Malik filter bank is well known for its adaptability and effectiveness in representing a wide range of textures. The LM set is a multi-scale, multi-orientation filter bank with 48 filters. It consists of first and second derivatives of Gaussians at 6 orientations and 3 scales making a total of 36; 8 Laplacian of Gaussian (LOG) filters; and 4 Gaussians, enabling the extraction of texture information across multiple spatial frequencies.

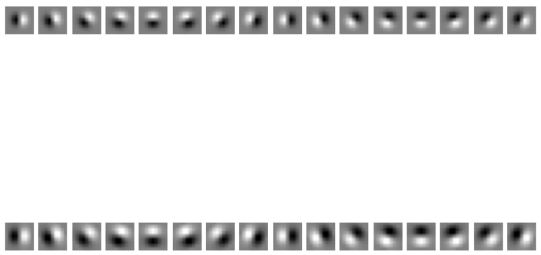


Fig. 1. *Oriented DoG Filter Bank*

These filters are widely used in computer vision tasks such as texture classification, segmentation, and pattern recognition. Their ability to discern texture variations makes them valuable for applications like medical imaging, satellite image analysis, and scene understanding. Fig.2 shows the LM-Filter Bank.

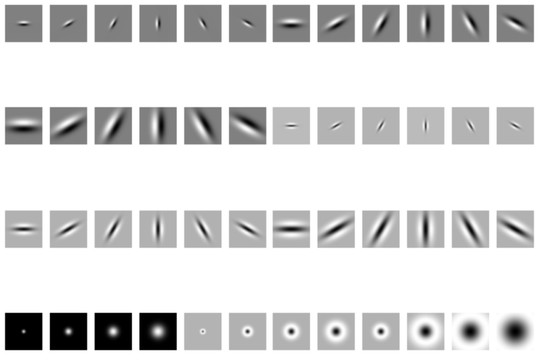


Fig. 2. *LM Filter Bank*

3) Gabor Filters: Gabor filters, essential components in image processing and computer vision, are essentially sine waves constrained by a Gaussian envelope. As sinusoidal wavelets, they exhibit an infinite number of zero-crossings within their domain. The sinusoidal function is bounded by the Gaussian envelope, ensuring that its amplitude never surpasses the envelope. Figure 3 visually illustrates Gabor filters at 5 different scales and 6 orientations, showcasing their adaptability to capture variations in spatial frequencies and orientations. This combination of sinusoidal and Gaussian characteristics gives Gabor filters the ability to effectively represent and respond to specific patterns, mak-

ing them valuable for tasks such as edge detection, texture analysis, and feature extraction in diverse applications, including fingerprint recognition and image segmentation.

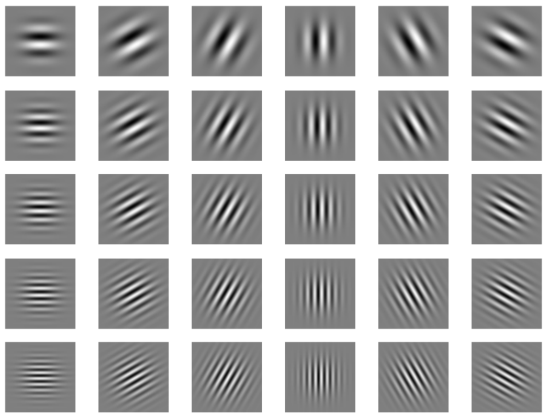


Fig. 3. *Gabor Filter Bank*

B. Texton, Brightness, Color Maps

Filtering an input image with a filter bank gives a vector of filter responses centered on each pixel, representing texture properties. Each N-dimensional vector is replaced with a discrete texton ID obtained through K-means clustering, reducing the dimensionality to a one-dimensional cluster ID. The number of clusters K was set to 64. Normalizing the filtered output in the 0-255 range after observing low-intensity values significantly improved clustering. The texton map, derived from the filter bank's responses, distinguishes object boundaries and texture edges. The implementation involves applying filters, resulting in N-dimensional vectors at each pixel. K-means clustering condenses these vectors into a single feature map, providing contextually rich information about texture. The observation that filter properties impact output more significantly than the number of clusters led to optimal results with N=64. Varying filter sizes influenced the texture map output, emphasizing the importance of filter characteristics.
The concept of the brightness map is to capture the brightness changes in the image. Here, again we cluster the brightness values using kmeans clustering into 16 clusters (K=16). We call the clustered output as the brightness map B.
The color map is created to capture the chrominance in the image. Here, again we cluster the RGB color values using kmeans clustering into 16 clusters (K=16). We call the clustered output as the color map C.

C. Texton, Brightness, Color Gradients

The changes from one pixel to the neighboring pixel can be encoded by using texture gradient (g), Brightness gradient (Bg) and Color gradient (Cg). We compute these by calculating the Chi2 distance with the help of half-disk filters obtained from previous calculations which is displayed in Fig. 14.

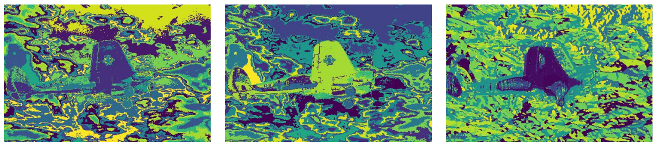


Fig. 4. *Brightness, color, texton maps for image1.jpg*

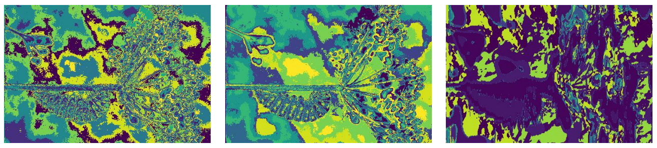


Fig. 5. *Brightness, color, texton maps for image2.jpg*

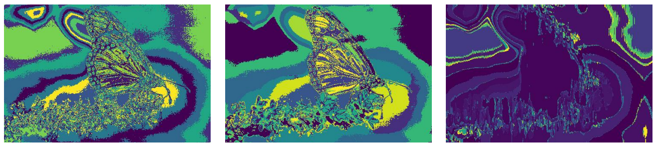


Fig. 6. *Brightness, color, texton maps for image3.jpg*

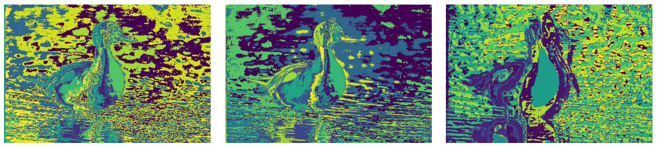


Fig. 7. *Brightness, color, texton maps for image4.jpg*

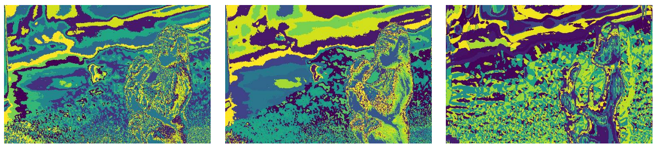


Fig. 8. *Brightness, color, texton maps for image5.jpg*

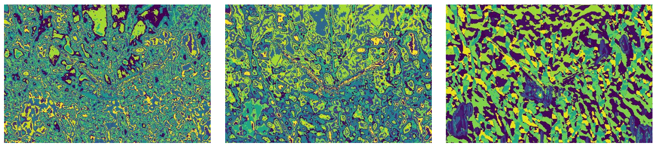


Fig. 9. *Brightness, color, texton maps for image6.jpg*

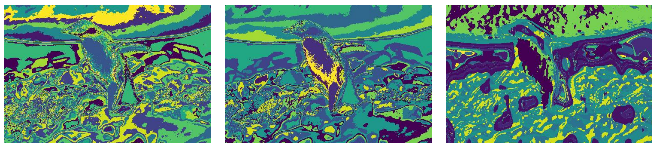


Fig. 10. *Brightness, color, texton maps for image7.jpg*

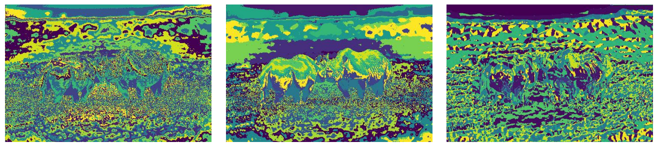


Fig. 11. *Brightness, color, texton maps for image8.jpg*

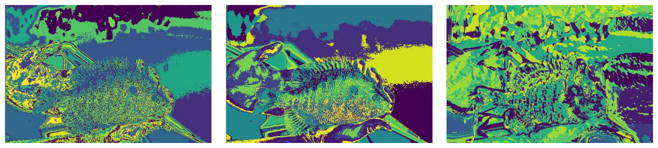


Fig. 12. *Brightness, color, texton maps for image9.jpg*

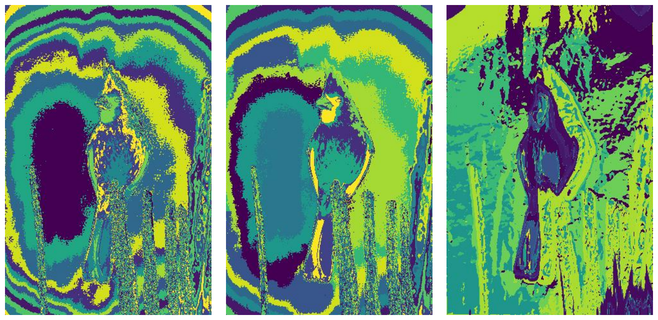


Fig. 13. *Brightness, color, texton maps for image10.jpg*

D. Pb-Lite Boundary Detection

For the Pb-lite boundary detector, first, we take the canny and Sobel baselines as input and take a weighted sum of both. Then we take the average of the Gradients calculated and finally use the Hadamard operator to get out the final Pb-lite output. The expression can be written as:-

$$pb = \frac{Tg+Bg+Cg}{3} \odot (w1 * \text{Canny} + w2 * \text{Sobel})$$

E. Discussion on Pb-Lite output

The final Pb-lite output does a good job reducing of most of the noise that Canny and sobel contain Owing to its ability to suppress the false positives that are produced by Canny and Sobel. However, some of the edges that should be a part of the output are also suppressed in the process. The different combinations of weights of canny and Sobel were tried and I was able to get a bit better performance. However, various other methods could be tried to achieve better results in this case like changing the orientation, scales, and size of filter banks and also allowing dynamics weight allocation.

II. Phase 2: Deep Dive on Deep Learning

In this phase, we implemented multiple Network Architectures to perform classification tasks on the CIFAR-10 dataset. The dataset had 10 classes, 50000 training images, and 10000 test images and each image size is 32x32.

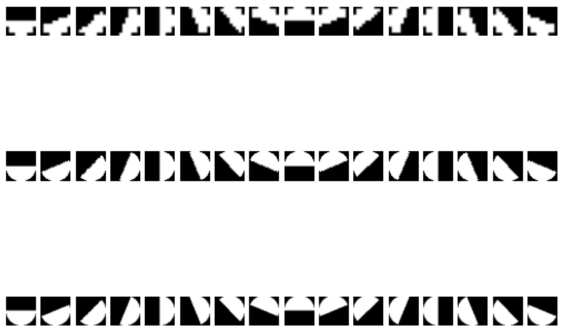


Fig. 14. *Half-Disc Filter Bank*

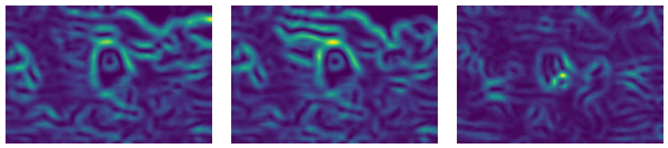


Fig. 15. *Brightness, color, texton Gradients for image1.jpg*

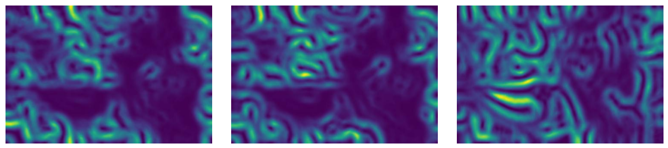


Fig. 16. *Brightness, color, texton Gradients for image2.jpg*

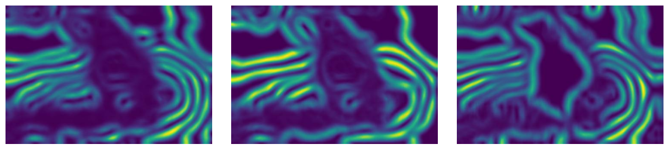


Fig. 17. *Brightness, color, texton Gradients for image3.jpg*

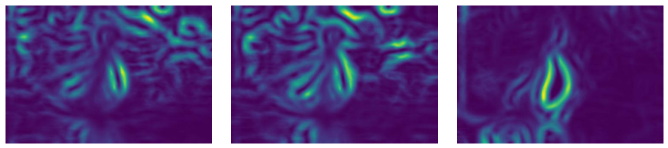


Fig. 18. *Brightness, color, texton Gradients for image4.jpg*

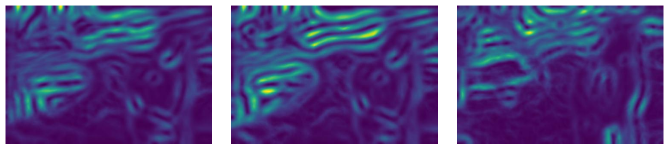


Fig. 19. *Brightness, color, texton Gradients for image5.jpg*

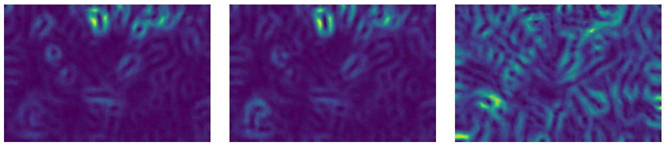


Fig. 20. *Brightness, color, texton Gradients for image6.jpg*

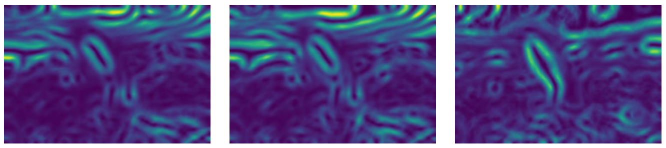


Fig. 21. *Brightness, color, texton Gradients for image7.jpg*

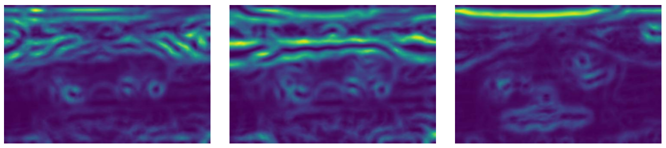


Fig. 22. *Brightness, color, texton Gradients for image8.jpg*

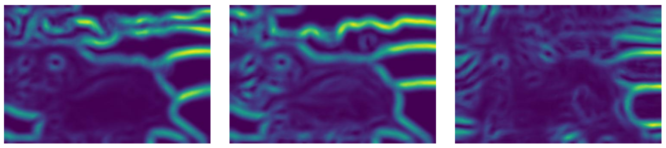


Fig. 23. *Brightness, color, texton Gradients for image9.jpg*

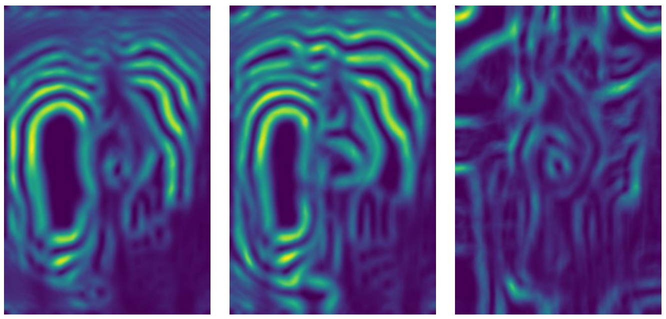


Fig. 24. *Brightness, color, texton Gradients for image10.jpg*

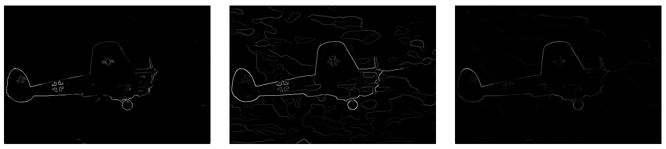


Fig. 25. *Sobel, canny, PbLite outputs for image1.jpg*

A. First Model Architecture (Without Batch Normalization):

For the initial CIFAR10 model, I opted for a straightforward linear neural network setup involving three layers. The input

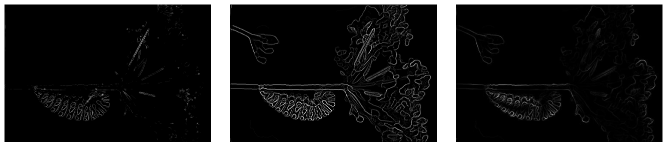


Fig. 26. *Sobel, canny, PbLite outputs for image2.jpg*

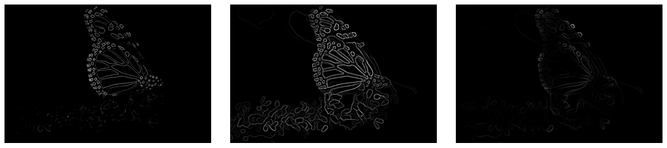


Fig. 27. *Sobel, canny, PbLite outputs for image3.jpg*

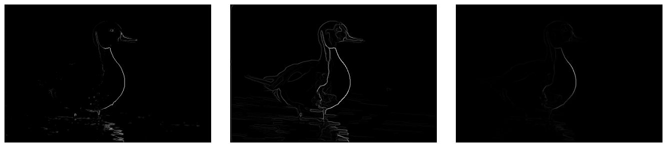


Fig. 28. *Sobel, canny, PbLite outputs for image4.jpg*

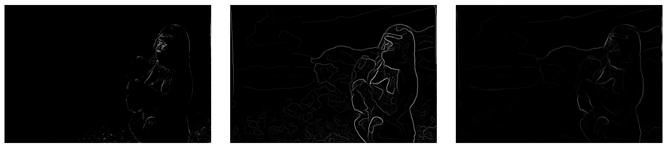


Fig. 29. *Sobel, canny, PbLite outputs for image5.jpg*

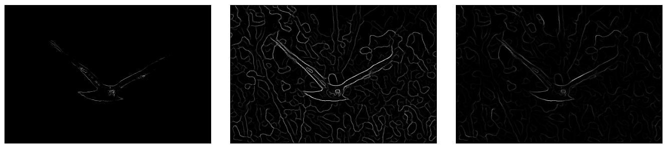


Fig. 30. *Sobel, canny, PbLite outputs for image6.jpg*

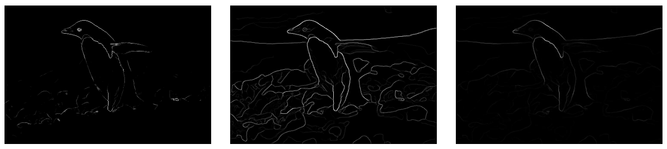


Fig. 31. *Sobel, canny, PbLite outputs for image7.jpg*

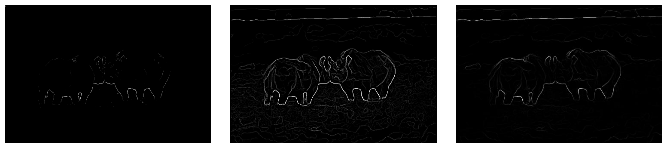


Fig. 32. *Sobel, canny, PbLite outputs for image8.jpg*

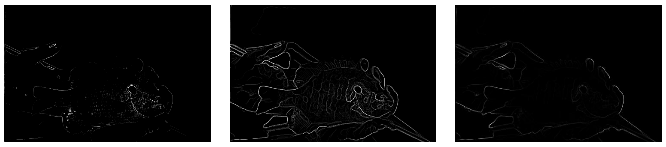


Fig. 33. *Sobel, canny, PbLite outputs for image9.jpg*

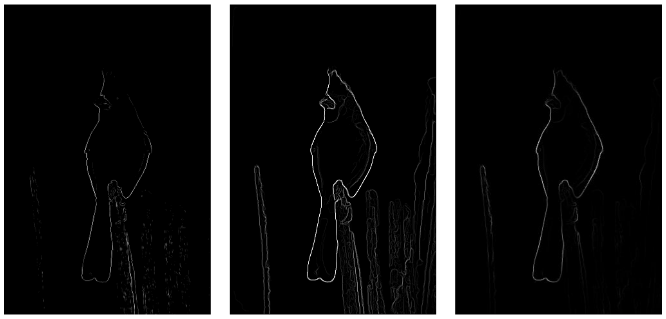
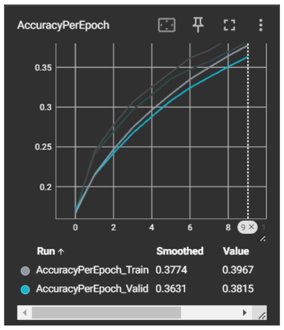
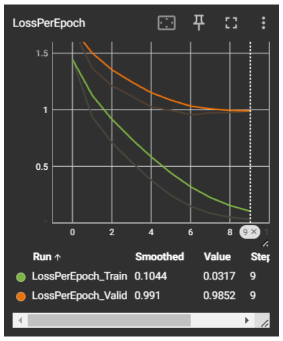


Fig. 34. *Sobel, canny, PbLite outputs for image10.jpg*



Run ↑	Smoothed	Value
AccuracyPerEpoch_Train	0.3774	0.3967
AccuracyPerEpoch_Valid	0.3631	0.3815

(a) Training Accuracy



Run ↑	Smoothed	Value	Step
LossPerEpoch_Train	0.1044	0.0317	9
LossPerEpoch_Valid	0.991	0.9852	9

(b) Training Loss

Fig. 35. First Model Architecture (Without Batch Normalization)

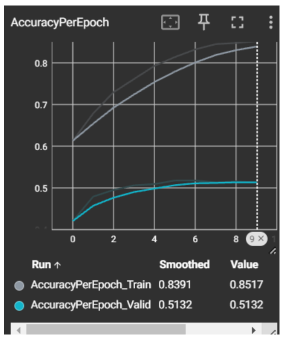
layer matched the input dimensions, followed by hidden layers featuring output sizes of 1024, 512, and 128. To introduce non-linearity, I applied ReLU activation functions after each linear layer. However, I chose not to include batch normalization in this model. This absence of batch normalization may lead to slower convergence during training, parameters were used to train the model. However, the absence of batch normalization could contribute to better generalization, as the model is less likely to overfit the training data

1) Number of Epochs = 10
2) Mini Batch Size = 128
3) Optimizer: Stochastic Gradient Descent
4) Learning Rate: 0.01

The model achieved a training accuracy of 31% and validation accuracy of 29%

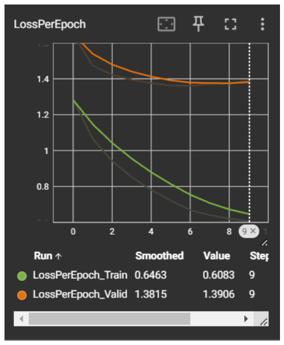
B. *Improved Model Architecture (With Batch Normalization):*

In the upgraded CIFAR10 model, I stuck to the same linear architecture with three layers – the input layer aligned with the input dimensions and hidden layers having output sizes of 1024, 512, and 128. The game-changer here was the addition of batch normalization after each linear layer. This tweak significantly stabilized the training process, leading to quicker convergence and higher training and validation accuracies. The training and validation losses(85 % and 55% respectively) showcased a more prompt and controlled reduction, emphasizing the influence of batch normalization on training efficiency. However, this improvement in accuracy came at the cost of overfitting, as batch normalization can lead to a model being overly adapted to the training data, resulting in reduced generalizability and robustness. All other parameters were kept the same as in the first model.



Run ↑	Smoothed	Value
AccuracyPerEpoch_Train	0.8391	0.8517
AccuracyPerEpoch_Valid	0.5132	0.5132

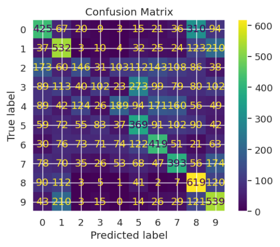
(a) Training Accuracy



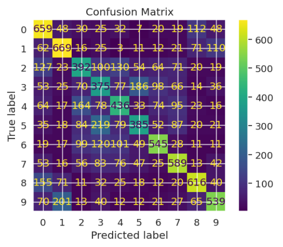
Run ↑	Smoothed	Value	Step
LossPerEpoch_Train	0.6463	0.6083	9
LossPerEpoch_Valid	1.3815	1.3906	9

(b) Training Loss

Fig. 36. Improved Model Architecture (With Batch Normalization)



(a) Confusion matrix basic model



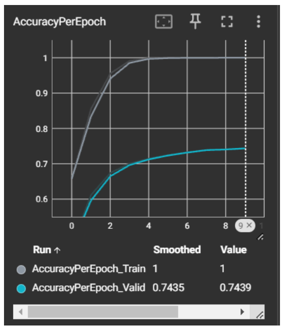
(b) Confusion matrix improved model

Fig. 37. Confusion matrix

C. *ResNet Model Architechture*

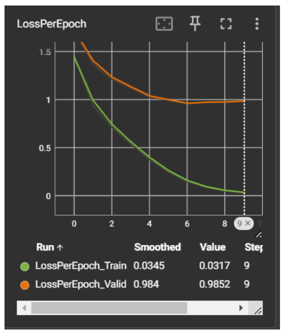
Residual Networks, commonly known as ResNets have played a key role in addressing a critical issue known as the vanishing gradient problem. This problem used to limit the depth of neural networks. The breakthrough idea behind ResNets is the introduction of shortcut connections, creating direct links between layers. These shortcuts facilitate smoother flow of gradients during training, allowing ResNets to successfully train much deeper models than their predecessors.

The model employs stacked "ResNet blocks," featuring shortcut connections that facilitate efficient gradient flow during training. The architecture comprises four layers of increasing complexity, progressively extracting intricate features from input images. The initial convolutional layer and max pooling extract preliminary features, followed by four residual layers with either BasicBlock or Bottleneck building blocks (choice affects model



Run ↑	Smoothed	Value
● AccuracyPerEpoch_Train	1	1
● AccuracyPerEpoch_Valid	0.7435	0.7439

(a) Training Accuracy



Run ↑	Smoothed	Value	Ste
● LossPerEpoch_Train	0.0345	0.0317	9
● LossPerEpoch_Valid	0.984	0.9852	9

(b) Training Loss

Fig. 38. ResNet Model Architecture

depth and computational cost). Each layer progressively increases feature complexity and downsamples spatial dimensions. Finally, a fully connected layer maps extracted features to the desired output classes. Achieved around 98% training accuracy and 74% validation accuracy.

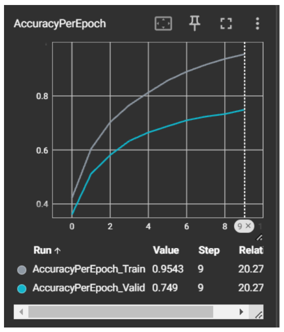
D. DenseNet Model Architechture

DenseNet, short for Densely Connected Convolutional Network, is a deep learning architecture that aims to address the vanishing gradient problem and promote feature reuse by connecting each layer to every other layer in a dense manner. Unlike traditional convolutional neural networks (CNNs), where layers are connected sequentially, DenseNet introduces dense connections, enabling direct communication between layers at different depths. This design results in highly compact models that are computationally efficient and exhibit improved parameter efficiency. I implemented DenseNet architecture called BottleneckDenseNet. This particular variant uses bottleneck layers, consisting of 1x1 and 3x3 convolutions, to reduce the number of parameters and computational cost.

The DenseNet model shows promise with a training accuracy of 95.43%, indicating it effectively learns from the given data. However, the validation accuracy of 74.9% raises concerns about overfitting. This gap suggests the model might memorize the training data but struggle with generalizing to new examples. The decreasing loss curves for both datasets imply ongoing learning, but the consistently higher validation loss reinforces the overfitting possibility. Implementing early stopping and regularization techniques could help mitigate this issue and improve the model's generalizability.

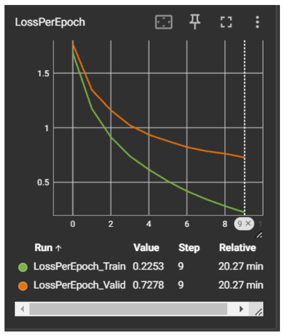
E. ResNext Model Architechture

ResNeXt is a convolutional neural network (CNN) architecture that builds on the success of ResNet by introducing the concept of "cardinality," representing the number of paths within a block. This innovative approach involves parallel pathways with different transformations, enhancing model capacity and feature diversity. ResNeXt's scalable design allows it to adapt efficiently to different tasks and datasets, leading to superior performance in image classification benchmarks. The training loss and accuracy of Resnext is plotted in Fig. 40



Run ↑	Value	Step	Relat
● AccuracyPerEpoch_Train	0.9543	9	20.27
● AccuracyPerEpoch_Valid	0.749	9	20.27

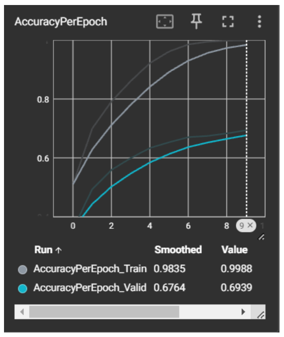
(a) Training Accuracy



Run ↑	Value	Step	Relative
● LossPerEpoch_Train	0.2253	9	20.27 min
● LossPerEpoch_Valid	0.7278	9	20.27 min

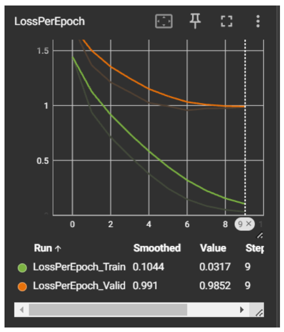
(b) Training Loss

Fig. 39. DenseNet Model Architecture



Run ↑	Smoothed	Value
● AccuracyPerEpoch_Train	0.9835	0.9988
● AccuracyPerEpoch_Valid	0.6764	0.6939

(a) Training Accuracy



Run ↑	Smoothed	Value	Ste
● LossPerEpoch_Train	0.1044	0.0317	9
● LossPerEpoch_Valid	0.991	0.9852	9

(b) Training Loss

Fig. 40. ResNext Model Architecture