HW0: Alohomora!

: RBE/CS546: Computer Vision

Sarthak Mehta
Department of Robotics Engineering
Worcester Polytechnic Institute
Email: smehta1@wpi.edu

***Abstract*—This document is a report for homework0 that presents two phases. Phase 1 is focused on detection from a single image. To accomplish this, an algorithm named Pb(Probability of Boundary) is implemented, using some basic texture, brightness, and color information combined with classical methods such as Canny edge and Sobel baselines. For phase 2, various deep learning algorithms are implemented for image-based classification of the CIFAR10 dataset.**

I. Phase 1: Shake my boundary

This section focuses on the implementation of Pblite. While most classical methods, such as Canny edge and Sobel baseline, only consider the discontinuity in intensity, the PbLite algorithm also considers the discontinuities in the color and texture of the images. The output of PbLite is the probability of boundary per pixel.

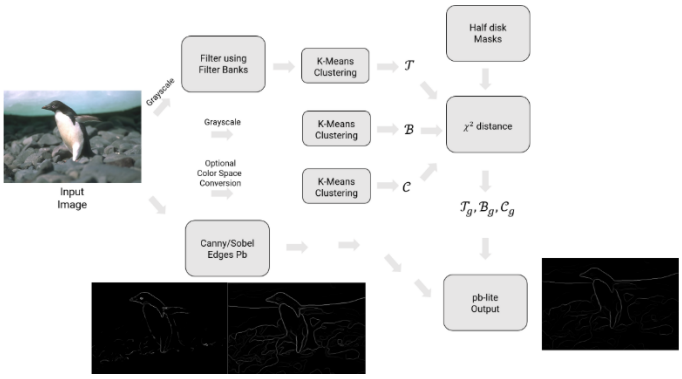

Fig. 1. Pb Lite Pipeline

As shown in Figure 1, the Pb Lite Pipeline illustrates the key stages of processing:

1) Applying a filter on the grayscale of the input image which results in obtaining texture.
2) Applying K-Means clustering to create Texture, Brightness, and Color map.
3) For each pixel, finding the gradient for the attributes using half-disc masks.
4) Combining the information obtained with Sobel and Canny baselines.

Firstly we need a texture map, brightness map, and color map. Now, to create the texture map, we will first pass the image through a set of filter banks namely Oriented DoG Filters, Leung-Malik Filters, and Gabor Filters, and cluster the different responses.

A. Oriented Derivative of Gaussian(DoG) Filter

The DoG is an image filter mainly used for the edge detection technique. Incorporating the orientation to make it more robust and better suited for detecting curves. For this filter we will create a gaussian kernel, after which we will convolute a rotated sobel filter onto this kernel. This results in a filter that has both smoothening and edge detection features. This filter created after convolution is called Derivative of Gaussian(DoG) filter.

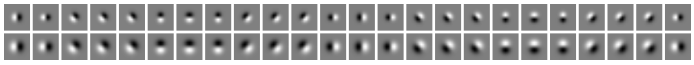

Fig. 2. Oriented DoG Filter Bank

This filter is generated by using two different scale values $\sigma = [1.0, 1.5]$ and a kernel size of 13×13, along with 24 different orientations starting from 0° to 360°. Hence, a filter bank of 2×24 filters is created.

B. Leung-Malik(LM) Filter Bank

The Leung-Malik (LM) filter bank is designed for texture classification and segmentation tasks. It is a multi-scale, multi-orientation filter bank that combines various types of filters to capture diverse texture features. The LM filters include first and second derivatives of Gaussian filters, Laplacian of Gaussian (LoG) filters, and simple Gaussian filters. This filter bank is generated using three scales for the Gaussian derivatives $\sigma = \{1.0, \sqrt{2}, 2.0\}$, and each derivative is computed for six different orientations ranging from 0° to 180°. Additionally, it includes eight Laplacian of Gaussian (LoG) filters, generated at scales σ and 3σ, along with four simple Gaussian filters at four scales. In total, the LM filter bank consists of 36 filters from the Gaussian derivatives, 8 LoG filters, and 4 Gaussian filters, resulting in a total of 48 filters. These filters are designed to capture edge, and texture features. Using these scales the filters generated are called LM Small version of the LM Filter Bank. In the case of LM Large, the scales are $\sigma = \{\sqrt{2}, 2.0, 2.0\sqrt{2}\}$. Figure 3 and Figure 4 shows LMS and LML filter banks.

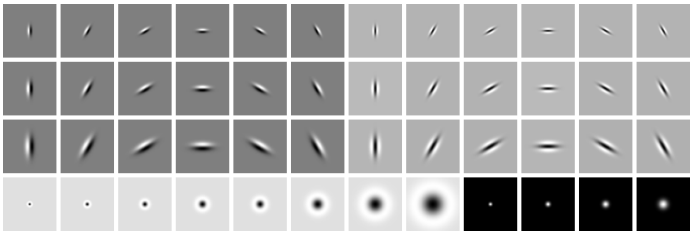

Fig. 3. LMS Filter Bank

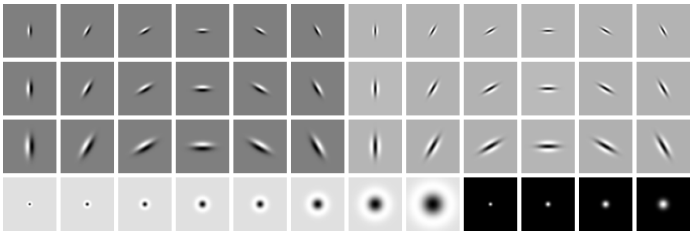

Fig. 4. LML Filter Bank

C. Gabor Filter Bank

The Gabor filter is widely used for edge detection and texture analysis, as it captures spatial frequency, orientation, and phase information simultaneously. Incorporating orientation and scale makes it robust for detecting edges and texture patterns in multiple directions and scales. A Gabor filter is essentially a Gaussian-modulated sinusoidal wave. This filter bank is generated using five different Gaussian spreads $\sigma = [6, 7, 9, 11, 13]$, five sinusoidal frequencies $\omega = [0.3, 0.4, 0.5, 0.6, 0.7]$, and a kernel size of 37×37, ensuring compact and localized filters. Each filter is designed for one combination of scale and frequency, along with multiple orientations, capturing a wide variety of spatial and frequency features. Hence, with 8 orientations ranging from 0° to 180°, a filter bank of $5 \times 8 = 40$ filters is created.

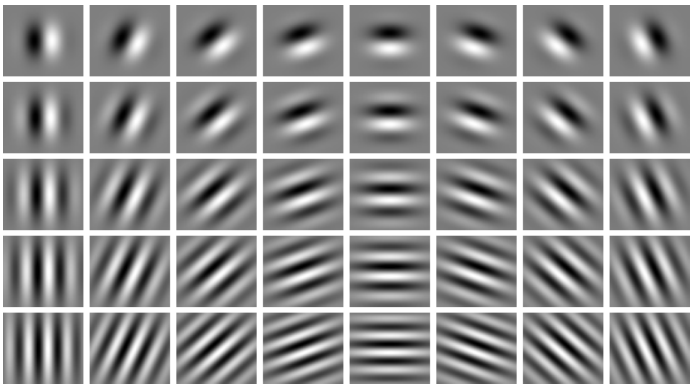

Fig. 5. Gabor Filter Bank

D. Texton Map

After filtering the image we need to create a texton map. The reason for this is that after filtering the input image through 3 different filter banks, we will have a vector of filter responses which is centered at each pixel. In this case we have used 3 different filter banks which will make the output for one image 3xwxh which would be stack of images. Now, we will use k-means clustering to simplify this representation by replacing each 3-dimensoinal vector with a unique texton ID. This process is called vector quantization. Now the output image can be represented with a single channel with values in a particular range which in this case is [1, 2, 3...., 64]. Texton Maps are shown in Figure 6 below.

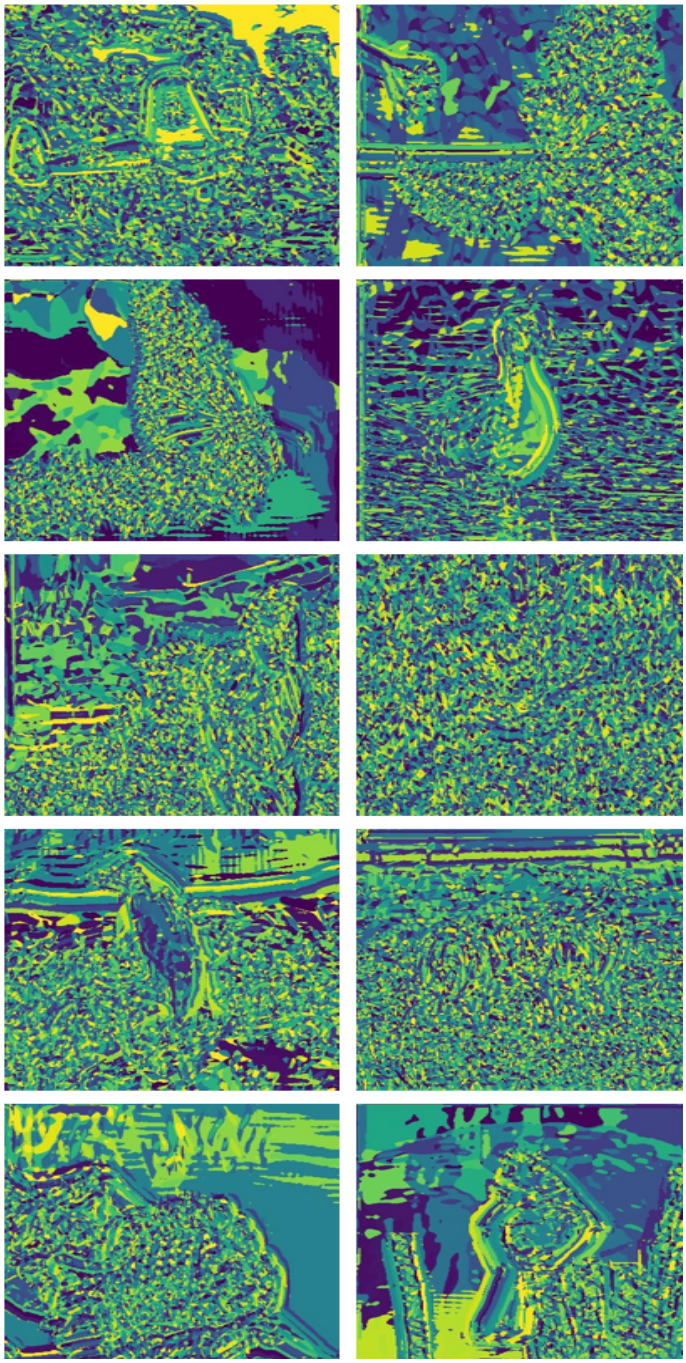

Fig. 6. Texton Maps

E. Brightness Map

The brightness map captures the brightness change in an image. Using k-means cluster on gray-scale of the image, we represent the image in a single channel with values in a particular range which in this case is [1, 2, ... , 16]. Brightness Maps are shown in Figure 7 below.

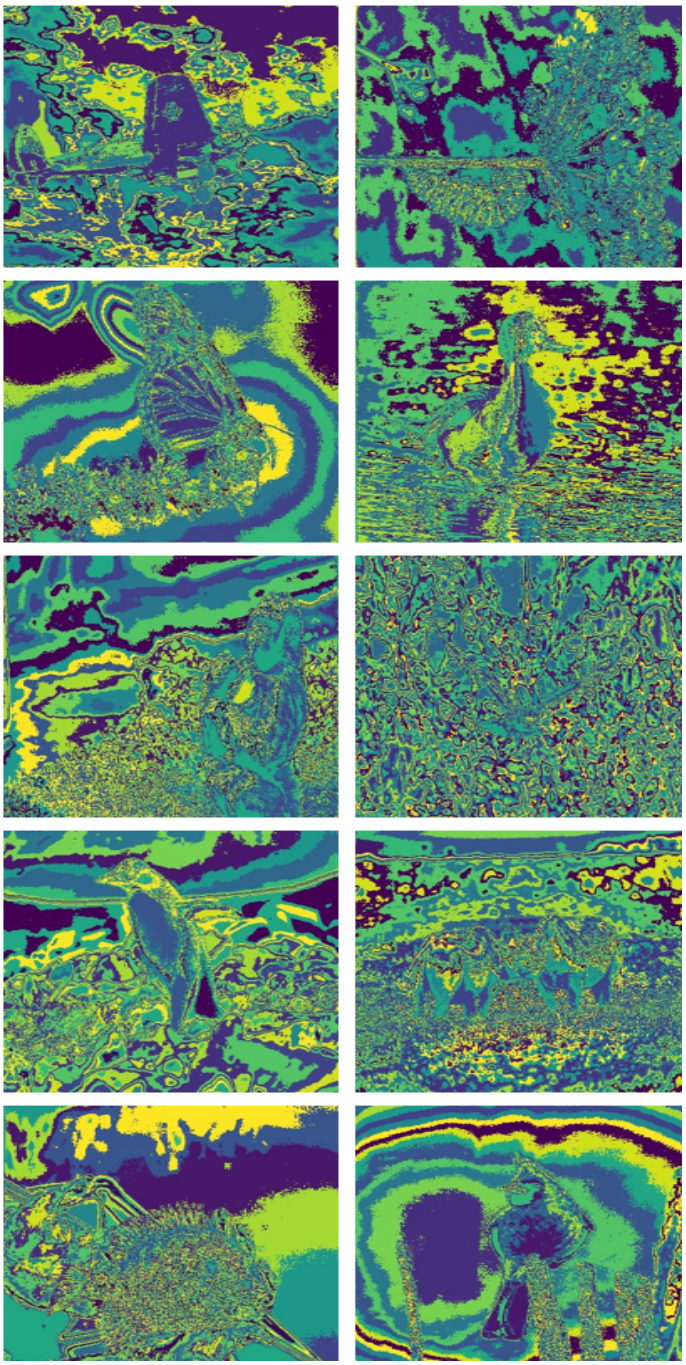

Fig. 7. Brightness Maps

F. Color Map

The brightness map captures the color changes in the image. Using k-means cluster on the RGB image which means in this case we have 3 channels, we represent the image in a single channel with values in a particular range which in this case is [1, 2, ... , 16]. Brightness Maps are shown in Figure 8 below.

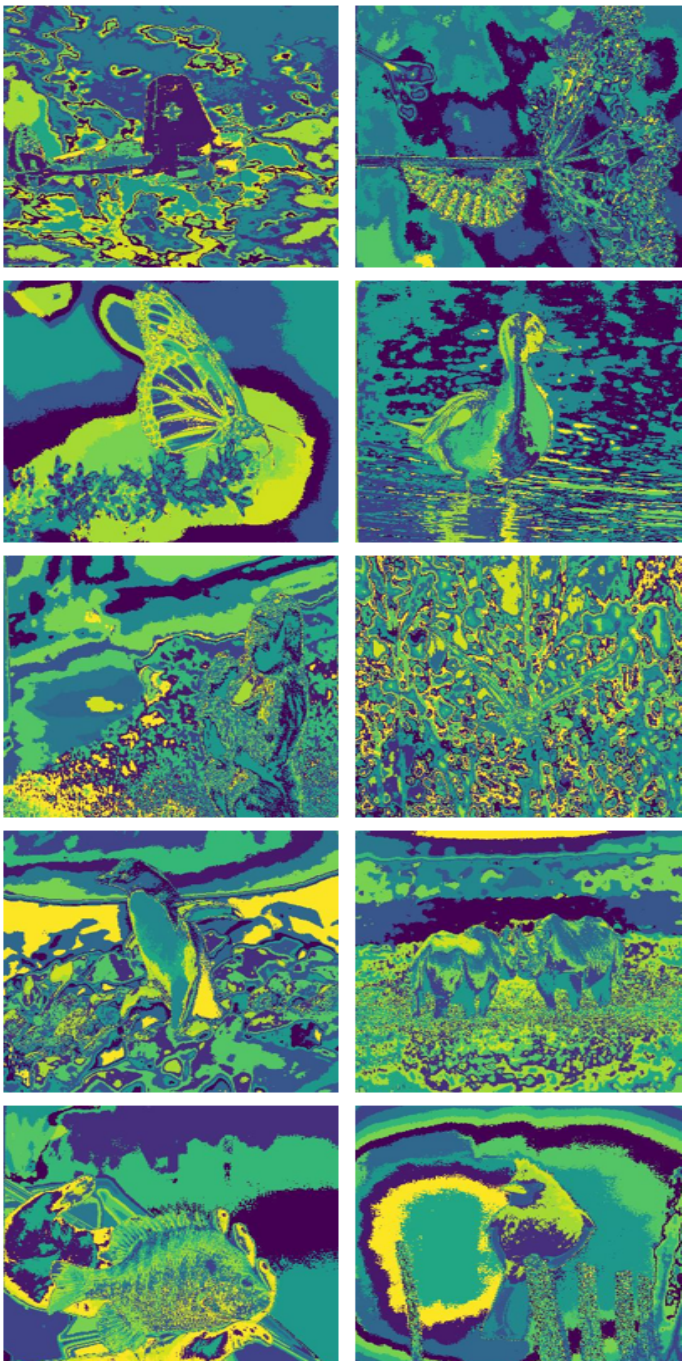

Fig. 8. Color Maps

G. Gradient Maps (T_g, B_g, C_g)

Now to obtain T_g, B_g, C_g we need to compute the differences of various values for shapes and sizes. To achieve this we will use half-discs masks. Half-discs are nothing but pairs of binary images of half-discs. We will use it to compute the

χ^2 (chi-square) distances. The half-discs mask are shown in Figure 9 below.

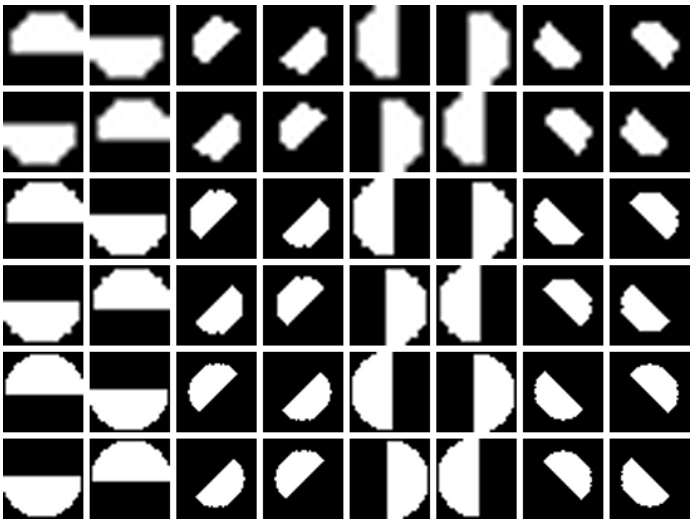

Fig. 9. Half Discs Mask

T_g, B_g, C_g basically encodes the change in texture, brightness and color distribution in a pixel are changing. These halfsics masks has in total 8 orientations ranged from 0° to 360° and scales ranged with different radiuses of size [5, 10, 15]. T_g, B_g, C_g for each image is shown in Figure 10.

We will compare texton, brightness and color distribution with χ^2 measure.

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

H. PBLite:

Now for the final step we will combine the outputs from PbLite and use the Canny and Sobel baseline to compute the probability of boundary for each pixel using the following equation.

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

The term

$$\frac{T_g + B_g + C_g}{3}$$

represents an averaged combination of these gradient maps, providing a balanced edge detection mechanism across texture, brightness, and color domains.

The term $w_1 * (\text{cannyPb})$ represents the convolution of the Canny edge detector's response (cannyPb) with a weight w_1, capturing edges detected by the Canny method. Similarly, $w_2 * (\text{sobelPb})$ represents the convolution of the Sobel edge detector's response (sobelPb) with a weight w_2, capturing edges detected by the Sobel method. These weighted responses are combined to leverage the strengths of both edge detectors, providing a more robust and complementary edge detection mechanism. The output for Canny, Sobel and PbLite is shown in Figure 11.

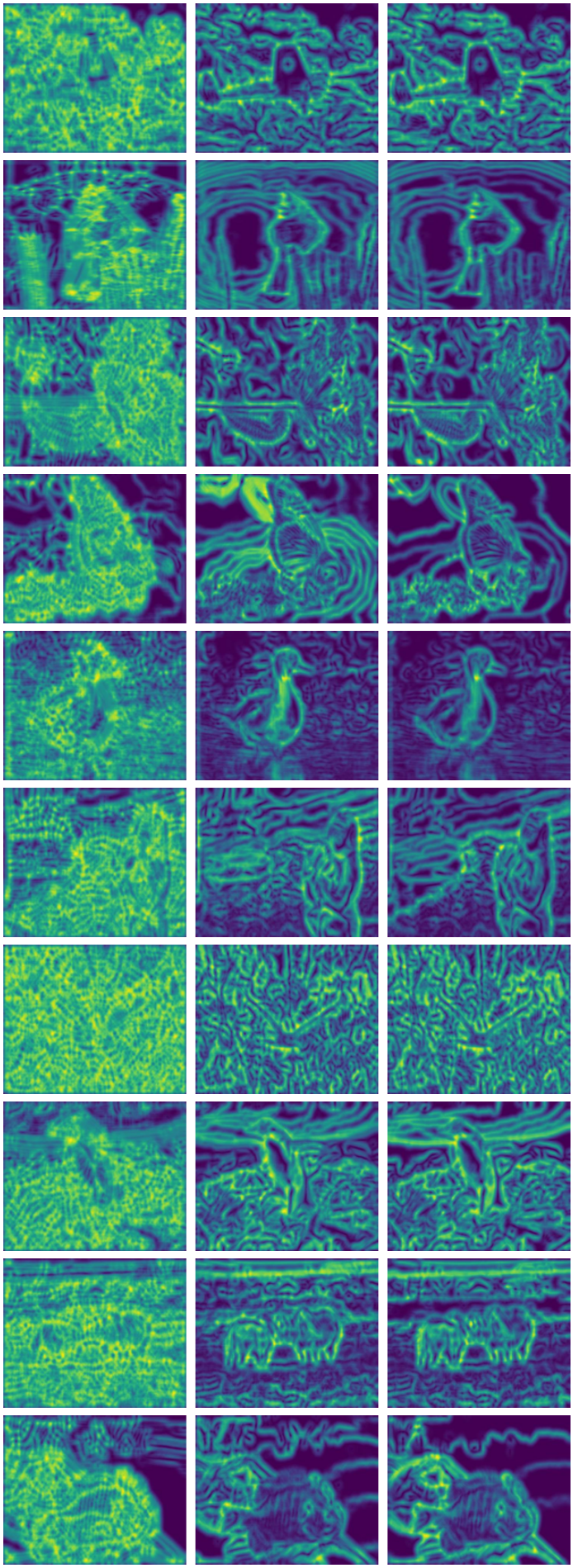

Fig. 10. T_g, B_g, C_g

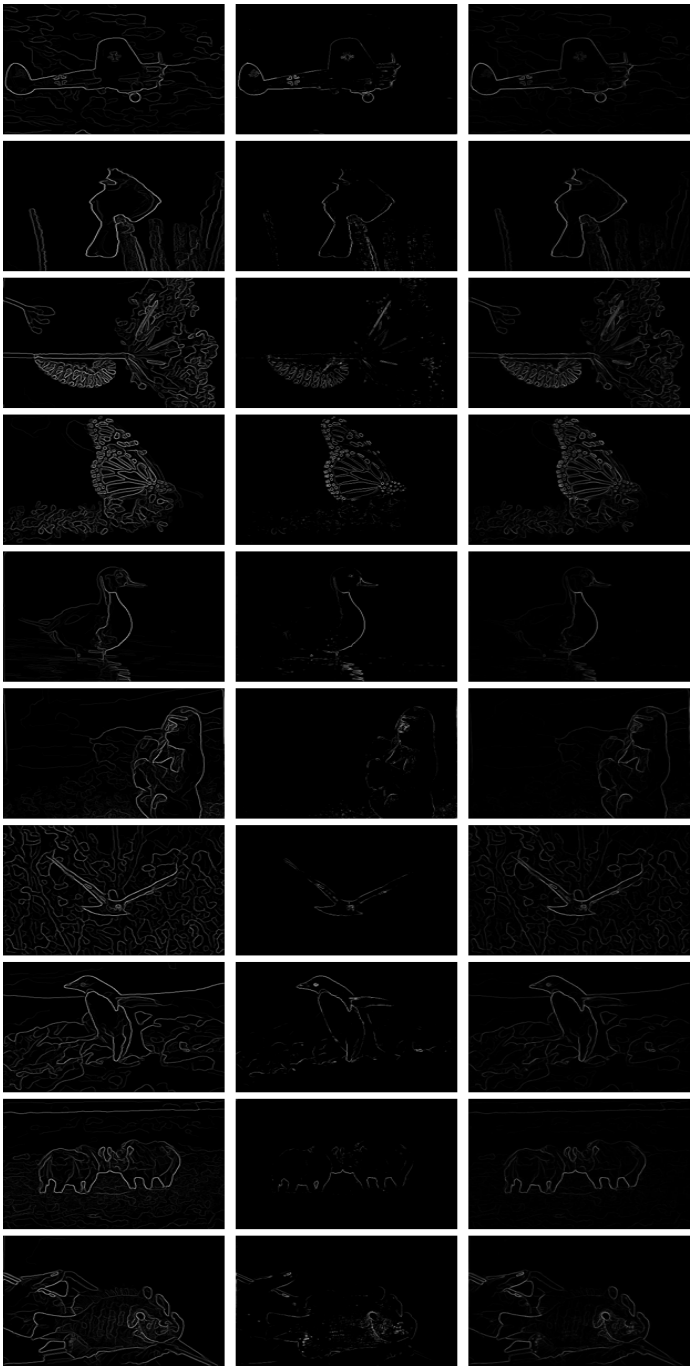

Fig. 11. Canny, Sobel, PbLite

II. Phase 2: Deep Dive into Deep Learning

In this section various deep learning algorithms are implemented for image-based classification of the CIFAR10 dataset. For the first part we need to implement a simple CNN model and test it on CIFAR-10 dataset. For the second part we need to optimize the same model to achieve a higher accuracy. And for the last part we need implement three neural network models ResNet, ResNeXt and DenseNet.

A. Dataset

Rather then using the dataset provided by pytorch we will use a custom dataset. The dataset has been randomized and in total it contains 60000 images, 50000 images are for training the model and 10000 images are test images. We need to classify this into 10 classes, size of each given image is 32x32.

B. 3.3: Building the first model

The first implementation is very simple. It has two convolutional layers with ReLU activations, doubling the feature maps from *InputSize* to $2 \times$ *InputSize*, each followed by 2×2 max-pooling to reduce spatial dimensions. The extracted features are passed through four fully connected layers (*fc1* to *fc4*) that progressively reduce dimensions and map the features to 10 output classes.

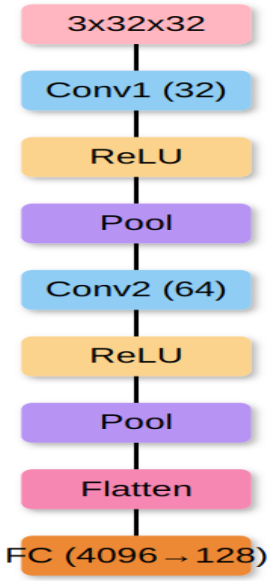

Fig. 12. Architecture

1) Loss Function: The loss function used in this implementation is Cross-Entropy loss function. This is commonly used for multi-class classification.

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c}\log(\hat{y}_{i,c})$$

As we can see in this the model doesn't achieve a good accuracy because it is not able to generalise better.

Parameter	Value
Number of Parameters	34,522
Hyper Param	lr = 0.001
Mini Batch Size	50
Layers	8
Epochs	20
Optimizer	Adam

TABLE I
Model Training Parameters

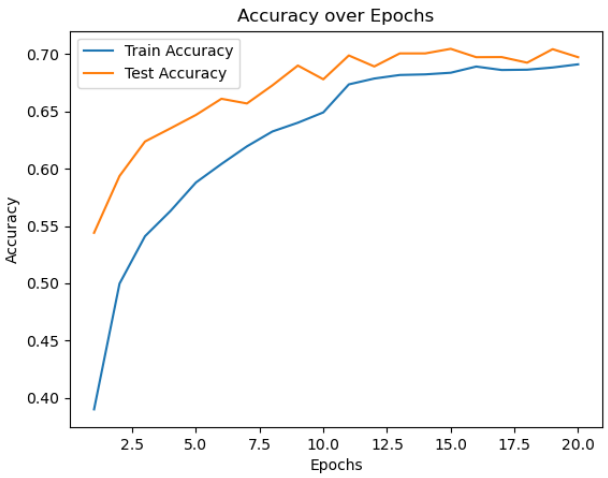

Fig. 13. Train and Test Accuracy over Epochs

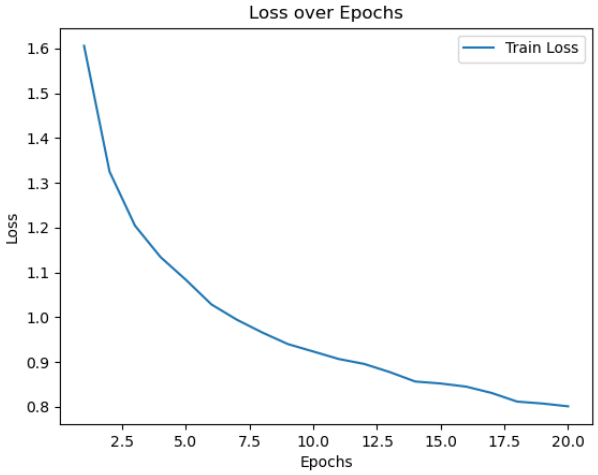

Fig. 14. Training Loss over Epochs

```
Confusion Matrix:
[713  13  52  41   9  16   3  10 105  29] (0)
[ 47 697   8  18   5   9   9   3  67 105] (1)
[ 50   5 515 105 109 105  57  59  31  12] (2)
[ 45   3  69 483  68 195  34  37  29  19] (3)
[ 30   2  87  91 569  56  41  77  22   3] (4)
[ 29   2  43 165  38 640  13  37  18  10] (5)
[  7   2  50 104  55  43 778   5  14   5] (6)
[ 13   1  36  51  58  92  15 661   5  21] (7)
[ 60  38   9  15  11  13   2   4 847  30] (8)
[ 42  83  15  18   7  29   3  21  53 722] (9)
  (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
```

Fig. 15. Confusion Matrix on Testing data

C. 3.4: Improving accuracy of your Neural Network

Now, to improve the accuracy we have added two batch normalization layer, each after a convolution layer. Normalization ensures the input distribution after every layer doesn't change. Also added the decaying learning rate which is reduced after every 10 epochs by a factor of 1/10.

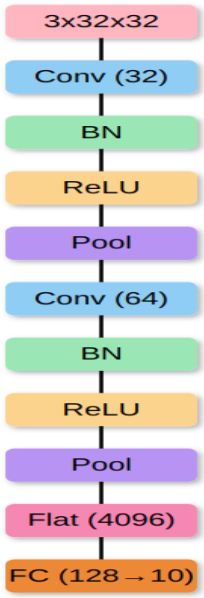

Fig. 16. Architecture

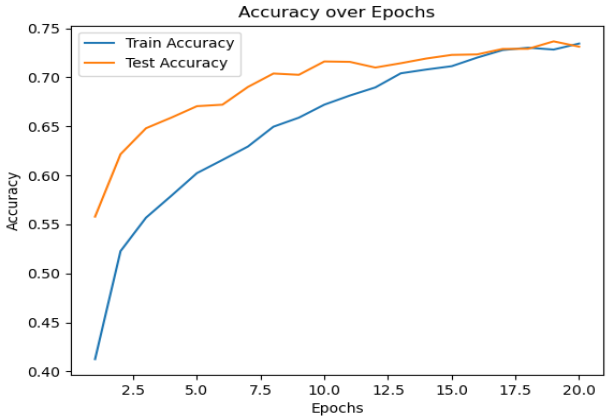

Fig. 17. Train and Test over Epochs

Parameter	Value
Number of Parameters	136,970
Hyper Param	lr = 0.1 (Start learning rate)
Mini Batch Size	100
Layers	10
Epochs	30
Optimizer	Adam

TABLE II
MODEL TRAINING PARAMETERS

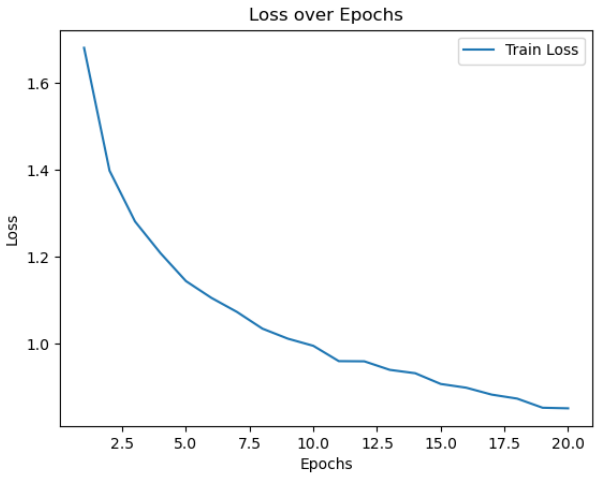

Fig. 18. Training Loss over Epochs

```
Confusion Matrix:
[769  15  71  14  10   2  11  13  96  47] (0)
[ 26 788   8   5   3   3   6   7  26 110] (1)
[ 66  13 545  41  97  87  74  38  23  17] (2)
[ 18  10  76 507  39 202  71  41  25   9] (3)
[ 30   0  58  67 592  31  67  62  16   4] (4)
[  8   2  65 236  31 566  19  54   7   8] (5)
[  4   9  55  62  39  16 827   4   6   3] (6)
[  8   2  40  49  60  73   2 799   1   7] (7)
[ 73  51  21  10   7   2   5   3 862  27] (8)
[ 32  89   2  16  11   4   8  19  21 719] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
```

Fig. 19. Training Confusion Matrix

D. 3.5: ResNet, ResNeXt and DenseNet

1) ResNet: It has three convolutional layers (conv1 to conv3), with Batch Normalization (bn1 to bn3) and ReLU activation, followed by a residual connection between the input and the output of conv3. Afterward, the feature map is downsampled by conv4 using a stride of 2, followed by conv5 to further refine features. A second residual connection between the downsampled input and the output of conv5 is implemented using the shortcut2 layer, ensuring efficient backpropagation of gradients. Finally, an adaptive average pooling layer (avgpool) reduces spatial dimensions, and a fully connected layer maps the features to 10 output classes.

In this it can be observed that we have better accuracy then the improvised model. The reason for that is The residual connections allow gradients to flow directly to earlier layers, mitigating the vanishing gradient problem, especially in deep networks. This ensures that lower layers continue to learn effectively, even as the network grows deeper. Additionally, the use of Batch Normalization improves training stability and convergence.

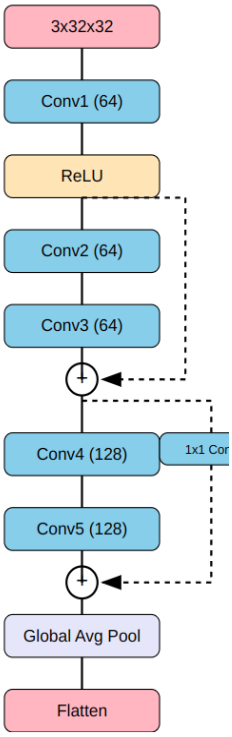

Fig. 20. Simplified ResNet18

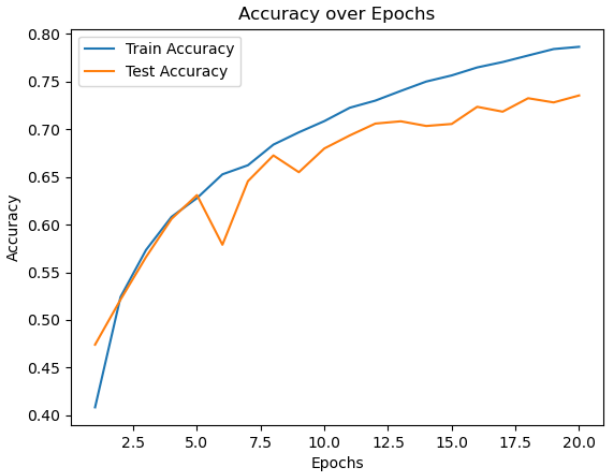

Fig. 21. ResNet18: Train and Test Accuracy

Parameter	Value
Number of Parameters	3,07,466
Hyper Param	lr = 0.001
Mini Batch Size	100
Layers	10
Epochs	20
Optimizer	Adam

TABLE III
RESNET18: MODEL TRAINING PARAMETERS

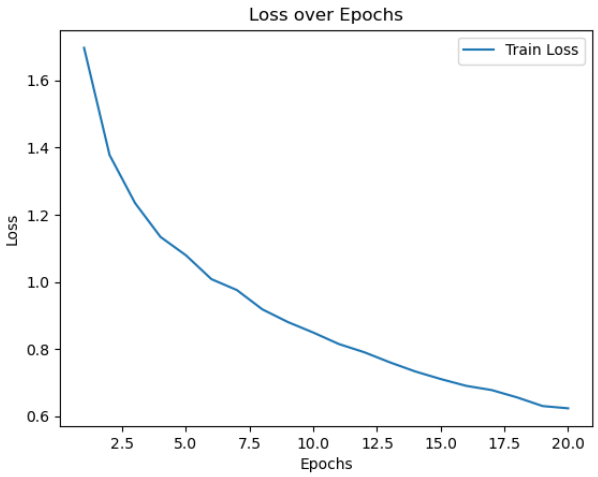

Fig. 22. ResNet18: Train Loss over Epochs

```
Confusion Matrix:
[760  34  27  14   5  15  26   1  67  16] (0)
[ 17 915   3   5   0   1   7   1   9  21] (1)
[ 77   3 477  72 102  79 171  15   8   3] (2)
[ 12  20  34 588  34 234  98  11   8  11] (3)
[ 21   6  34  61 628  37 120  30   5   2] (4)
[  2   5  19 160  32 724  42  12   0   5] (5)
[  4   3  12  43  15  14 910   1   4   0] (6)
[ 32   7  10  48  66 110  13 672  13   6] (7)
[ 52  38   0  11   9   6   6   2 846   2] (8)
[ 45 164   1  13   5   2   9   0  25 835] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
```

Fig. 23. ResNet18: Training confusion matrix

2) ResNeXt: The model is a simplified ResNeXt architecture designed for image classification on the CIFAR-10 dataset. It begins with an initial convolutional layer followed by a max-pooling layer to reduce the spatial dimensions. The core of the network consists of two ResNeXt blocks, each comprising grouped convolutions (with a cardinality of 32) for parallel feature extraction, followed by a 1×1 convolution to merge these features. Residual connections are implemented in each block using shortcut connections to facilitate gradient flow and mitigate vanishing gradient problems. After the ResNeXt blocks, max-pooling further reduces the spatial dimensions, and the flattened output is passed through three fully connected layers for classification into 10 output classes.

3) DenseNet: This model implements a simplified **DenseNet** for image classification on the **CIFAR-10** dataset. It consists of three dense blocks, where each block contains multiple `DenseLayer` modules that concatenate the input with newly generated features, ensuring feature reuse and reducing redundancy. Each dense block is followed by a transition layer, consisting of a batch normalization and a pooling operation to reduce the spatial dimensions. The

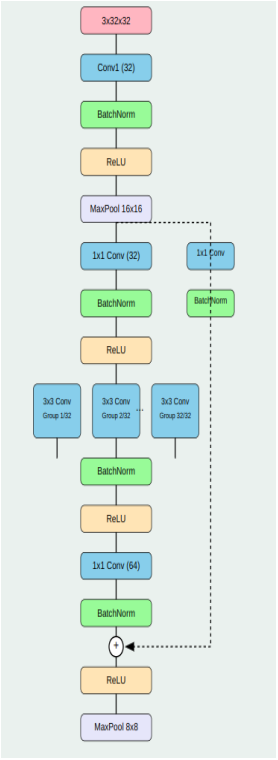

Fig. 24. Simplified ResNeXt

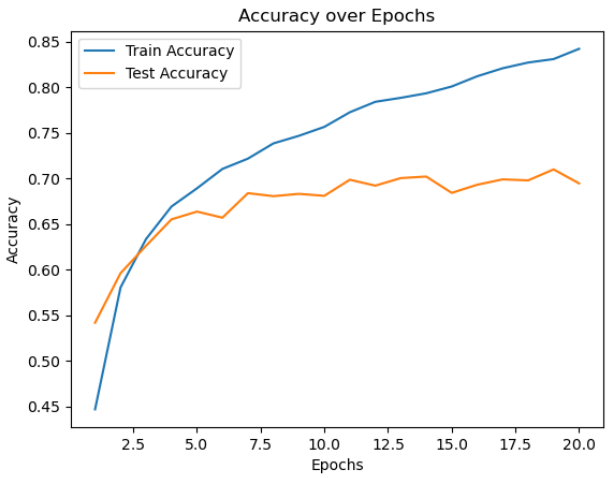

Fig. 25. ResNeXt: Train and Test Accuracy

Parameter	**Value**
Number of Parameters	104,398
Hyper Param	lr = 0.001
Mini Batch Size	100
Layers	10
Epochs	20
Optimizer	Adam

TABLE IV
RESNEXT: MODEL TRAINING PARAMETERS

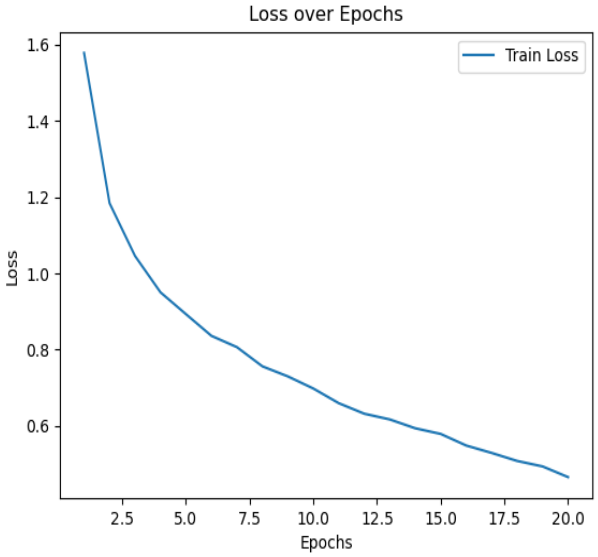

Fig. 26. ResNeXt: Train Loss over Epochs

```
Confusion Matrix:
[705  42  37  35  14   0   8   9 116  30] (0)
[ 24 802   2  15   1   6  19   1  50  61] (1)
[ 87  18 595  69 128  59  65  27  15   9] (2)
[ 30  12  58 598  60  89  91  26  39   6] (3)
[ 30   8  33  67 640  24  55  88  18   1] (4)
[ 16   1  33 278  42 510  18  67  15   2] (5)
[  7  14  30  75  48  10 810  12   8   7] (6)
[  8   7  28  67  64  36  10 730   4  14] (7)
[ 73  27   7  16   2   2  10   4 849  16] (8)
[ 33 114  13  35   2   6  12  24  55 707] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
```

Fig. 27. ResNeXt: Training confusion matrix

model starts with an initial convolutional layer and concludes with a global average pooling layer and a fully connected layer for classification. The *growth rate* controls how many features each `DenseLayer` adds, while the *block_config* determines the number of layers in each dense block, making the model efficient.

Parameter	Value
Number of Parameters	581,130
Hyper Param	lr = 0.001
Mini Batch Size	100
Layers	10
Epochs	20
Optimizer	Adam

TABLE V
DENSENET: MODEL TRAINING PARAMETERS

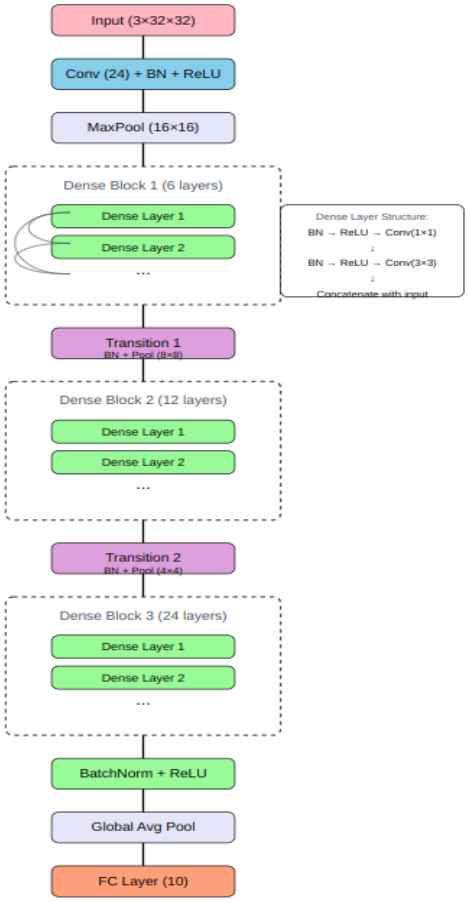

Fig. 28. DenseNet

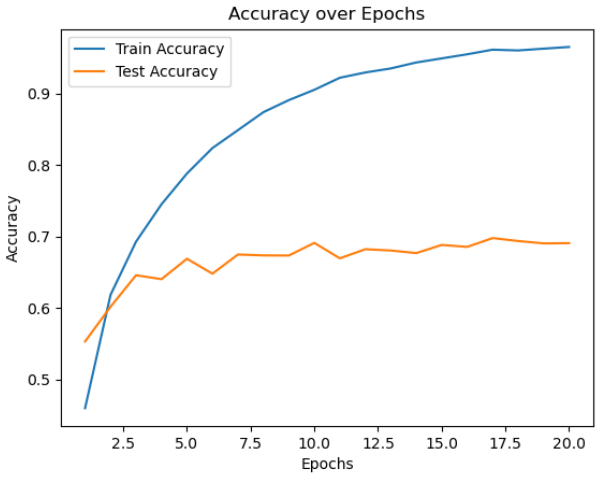

Fig. 29. DenseNet: Train and Test Accuracy

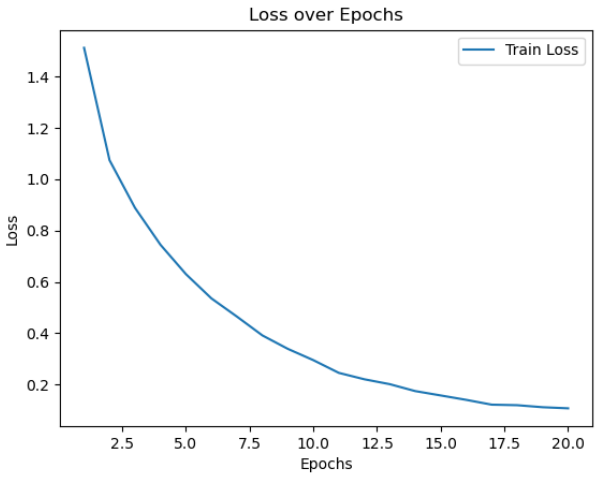

Fig. 30. DenseNet: Train Loss over Epochs

```
Confusion Matrix:
[643  11  73  22  40  17  21   6 130  34] (0)
[ 12 826   3  18   3   8   5   4  62 105] (1)
[ 68   7 605  81  82  42  90  22  21   4] (2)
[ 13   9  60 568  59 145  89  37  21  35] (3)
[ 10   5 109  53 627  28  72  54  10   8] (4)
[ 13   3  50 239  53 583  35  36   9  13] (5)
[  3   6  62  67  50  22 726   3  13   9] (6)
[  8   6  40  53  79  54   8 698   7  34] (7)
[ 22  24   4  27  10  12  15   3 865  18] (8)
[ 11  66   2  15   5   3   5   9  56 769] (9)
  (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
```

Fig. 31. DenseNet: Training confusion matrix

E. Conclusion

In this we implemented multiple deep learning architectures and test for the CIFAR-10 dataset. Including ResNet18, ResNeXt and DenseNet. Each model has its own way to optimisation for instance ResNet18 uses residual connections which ensures that lower layers learns effectively even as the network grows deeper, ResNeXt uses grouped convolutions to efficiently process features parallel, and DenseNet uses dense connectivity for enhanced feature reuse and gradient flow.

Among these models, ResNet18 has achieved the highest test accuracy of 74%, this highlights the effectiveness of residual networks. Hence, we can say that it is a optimal choice for CIFAR-10 classification.

Model	Parameters	Train Accuracy (%)	Test Accuracy(%)
Simplified CNN	34,522	68.56	69.38
improved CNN	136,740	72.34	71.92
ResNet18	3,07,466	77.26	72.45
ResNeXt	1,04,398	83.6	68.32
DenseNet	5,81,130	97.23	67.37

TABLE VI
COMPARISON BETWEEN MODELS

REFERENCES

1) https://www.geeksforgeeks.org/apply-a-gauss-filter-to-an-image-with-python/
2) https://www.geeksforgeeks.org/rotate-a-picture-using-ndimage-rotate-scipy/
3) https://rbe549.github.io/spring2025/hw/hw0/
4) https://www.robots.ox.ac.uk/~vgg/research/texclass/filters.html
5) https://en.wikipedia.org/wiki/Gabor_filter
6) https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html
7) https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html