multirow

Homework 0 - Alohomara

Krunal M. Bhatt
Masters of Science in Robotics Engineering
Worcester Polytechnic Institute
Worcester, MA - 01609
Email: kmbhatt@wpi.edu

Using 1 Late Day

***Abstract*—This document presents the two phases of homework 0 for the course RBE 549: Computer Vision. Phase 1 is focused on detecting edges by implementing the pb (probability of boundary) algorithm with the help of some basic texture, brightness, and color information in combination with Canny and Sobel baselines. Phase 2 is focused on a deep-learning approach for image classification on the CIFAR-10 dataset.**

I. Phase 1: Shake my Boundary

This section focuses on the development of a simplified version of pb, which finds boundaries by examining brightness, color, and texture information across multiple scales (different sizes of objects/images). The pipeline for the same can be observed in the image below in Fig. 1. Traditional Approaches to solving this problem are Canny and Sobel Edge Detectors. In a grayscale image, Sobel computes the intensity across neighboring pixels while Canny detects the edge by checking the similarity between the gradients at that pixel and the neighboring pixel.

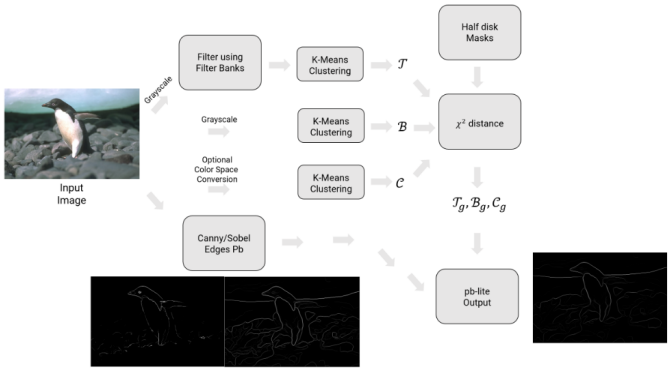

Fig. 1. Overview of pb lite pipeline

Both algorithms take into account the change in the intensity of a pixel. The algorithm is implemented through these steps:

1) Apply filter on input image resulting in obtaining texture
2) Quantizing low-level attributes like Texture, Color, and Brightness
3) Finding the gradients of these attributes for each pixel
4) Combining the result with baseline Canny and Sobel results

Here we use three filter banks namely: Oriented DoG filters, Leung-Malik Filters, and Gabor Filters. We will describe them in brief with their technical details below:

A. The Oriented Derivative of Gaussian (DoG) :

DoG is an image filter utilized in computer vision and image processing to accentuate image features along particular orientations while suppressing noise and irrelevant details. It is constructed by convolving a Gaussian kernel with a Sobel kernel oriented along a specific direction. The resulting filter response emphasizes edges and features aligned with the specified orientation, making it valuable for tasks such as edge detection, texture analysis, and object recognition in images. Fig. 2 shows the same.

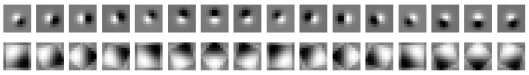

Fig. 2. Orienrted Derivative of Gaussian(DoG) filter

The filter is generated by using 2 different scale values 16 orientations from 0 deg to 360 deg, therefore getting a filter bank of 2 x 16 filters.

B. The $Leung - Malik\ Filters$ (LMF) :

It is a set of image filters designed to capture a wide range of image features, including edges, textures, and other visual patterns. The Leung-Malik filters, also known as LM filters, comprise a collection of 48 filters that are multi-scale and multi-orientation. This filter bank includes 36 filters derived from first and second-order derivatives of Gaussians at 6 orientations and 3 scales, along with 8 Laplacian of Gaussian (LOG) filters and 4 Gaussian filters. There are two versions of the LM filter bank, with the LM Small (LMS) version featuring filters at fundamental scales such as $\sigma = \{1, \sqrt{(2)}, 2, 2\sqrt{(2)}\}$. Derivatives occur at the first three scales with an elongation factor of 3, i.e. $(\sigma_x = \sigma\ and\ \sigma_y = 3\sigma_x)$. In the LM Large (LML), the filters occur at the basic scales such as $\sigma = \{\sqrt{(2)}, 2, 2\sqrt{(2)}, 4\}$. Fig. 3 shows LMS and Fig. 4 shows the LML filter bank.

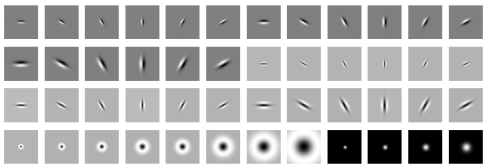

Fig. 3. LMS filter bank

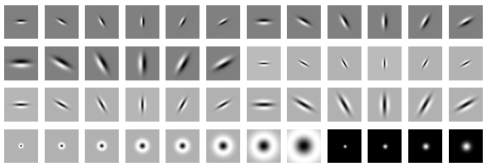

Fig. 4. LML filter bank

C. The $GaborFilters$:

The Gabor filters are kernels that have been made by modulating the Gaussian kernel with a sinusoidal plane kernel. This is a filter that analyses if there is any specific frequency content in a particular direction in the image. The filter bank includes Gabor filters with varying standard deviations for the Gaussian component and different frequencies for the sinusoidal components. These filters are then rotated to produce a range of orientations. Gabor filters with 3 different standard deviation values and a sinusoid frequency were implemented in Fig. 5.

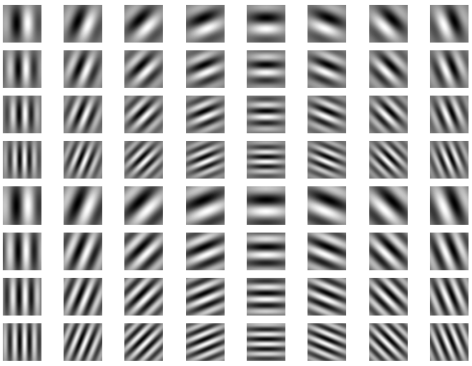

Fig. 5. Gabor filter bank

Now, we saw three tools to extract the low-level information and use it to make filter banks. Further, we will see some representations of the filtered outputs presented differently. We will look at three core representations for this below:

D. $Texton\ Map\ (\mathcal{T})$:

When the image is filtered we make the filter bank and stack it up with the size of images m x n xN, where the image is

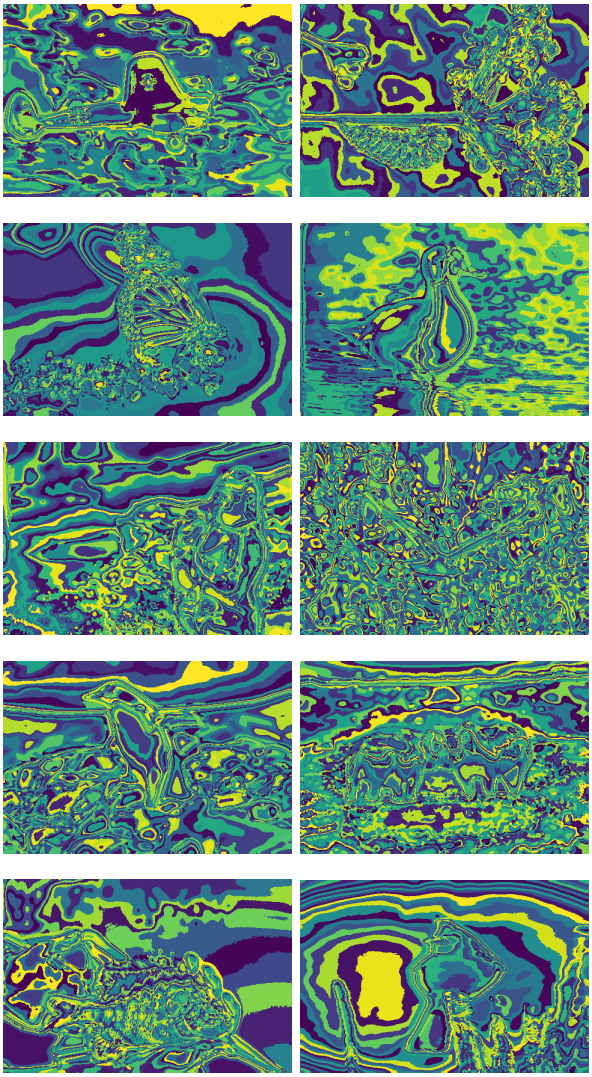

Fig. 6. Texton Maps for the set of test images

m x n size and N is the number of filters used. Hence, we can now represent each pixel as a distribution of these N values. Each distribution can be represented with a unique Texton ID. Different distributions are then clustered using K-means clustering into K textons. As a result, an image is generated, shown in Fig. 6, capturing the texture changes in the original image.

E. $Brightness\ Map\ (\mathcal{B})$:

The brightness map's concept is to identify brightness fluctuations (intensity variations) within the image. Once more, we arrange the brightness values (corresponding to the color image's grayscale representation) into a predetermined number of clusters (K = 16) using k-means clustering. The output can be seen in Fig. 7:

F. $Color\ Map(\mathcal{C})$:

Color maps represent the change in the chrominance(color) in the image. The same approach is employed using K-

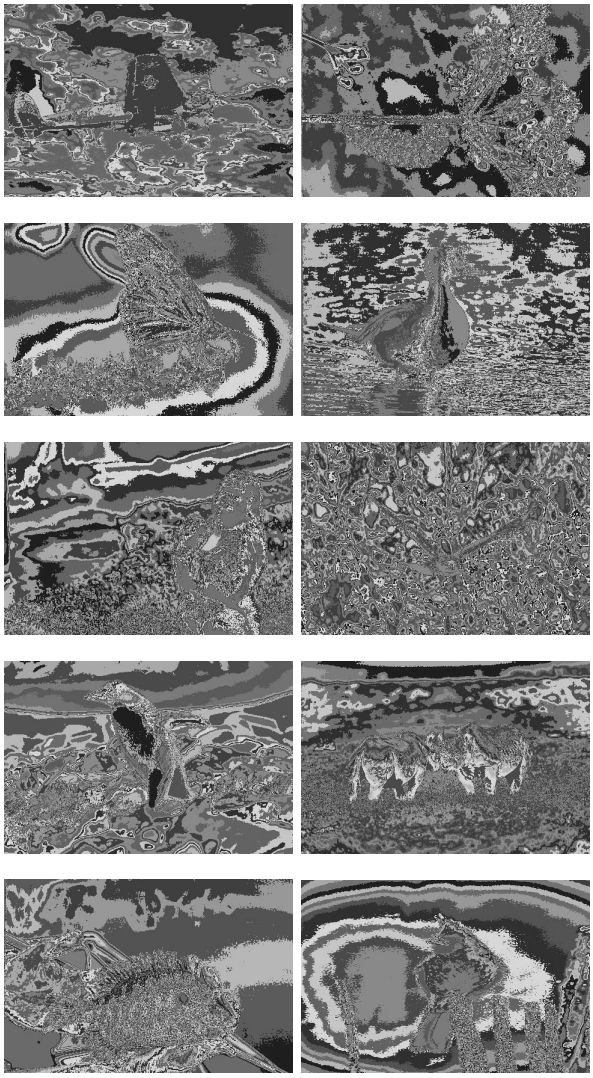

Fig. 7. Brightness Maps for the set of test images

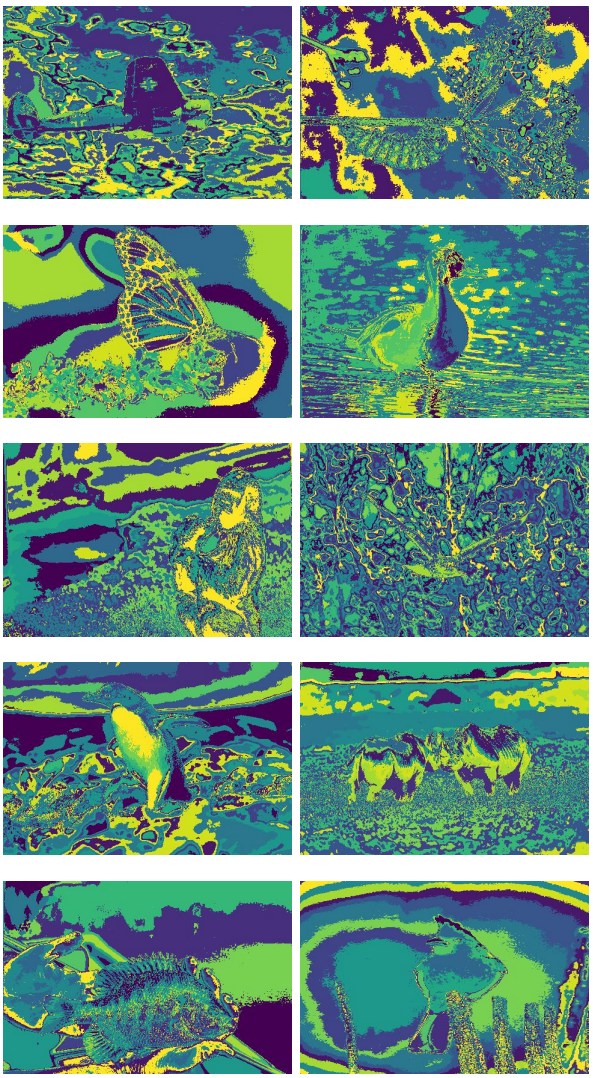

Fig. 8. Color Maps for the set of test images

Means to cluster the color values. Fig. 8 shows the color map representation for the set of test images provided.

G. $Gradient\ Maps(\mathcal{T}_g, B_g, \mathcal{C}_g)$:

Another step into the process, we calculate the gradients of the above texture, color, and brightness measures. We do it by implementing the Half-Disc masks. These masks are just a pair of binary images of half-discs. This way we can calculate the $\mathcal{X}^2$ distances for calculating the textures using a filtering operation. This is much faster than looping over each pixel and averaging the histograms. A sample set of masks (8 orientations and 3 scales) is shown in Fig. 9.

$\mathcal{T}_g, B_g, \mathcal{C}_g$ represent how much texture, brightness, and color distributions are changing at a pixel. We compute them by comparing the distributions in left/right half-disc pairs, which are straightforward to generate since you have control over the angle, and are centered at a pixel. The gradient ought to be modest if the distributions are similar. The gradient ought to

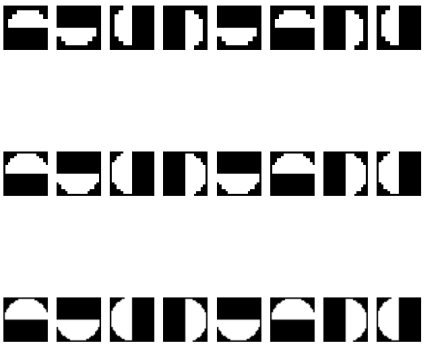

Fig. 9. Half disc masks

be substantial if the distributions are not identical. Our half-discs cover several orientations and scales, thus in the end we will have a set of local gradient measurements that encode the rate at which the texture or brightness distributions change at various angles and scales.

$\mathcal{X}^2$ on the other hand is a frequently used metric for the comparison between two histograms. It is calculated as follows:

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

here, K = index through bins. The numerator is the sum of the squared difference between histogram elements. The denominator adds a normalization to each bin so that less frequent elements also have a meaningful impact. Fig. 10 shows the $\mathcal{T}_g, B_g, \mathcal{C}_g$ respectively.

H. Color Map($\mathcal{C}$):

Color maps represent the change in the chrominance(color) in the image. The same approach is employed using K-Means to cluster the color values. Fig. 8 shows the color map representation for the set of test images provided.

I. PBLite:

In the end, we refer to the pipeline in Fig. 1 and see that we combine these gradient maps generated with the Canny and Sobel baselines. Let us first see the Canny and Sobel baseline images that are being combined with the gradient maps shown in Fig. 10. The Fig. 11 we see Canny baseline images and in Fig. 12 we see Sobel baselines.

We then use these baselines and an equation to combine and find the probability of the boundary. The following equation is used:

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

J. Discussion and Conclusion: Phase 1

Working with the equation above and the weights of Canny and Sobel baselines, there were significant changes in the output pb lite image. The Fig. 13 represents the output for all the 10 test set images. Reading about the compared baselines, it was evident that Canny's baseline also includes weaker unwanted edges as can be seen in Fig. 11. This happens due to the averaging operation being performed when the Canny edges are calculated. Similarly, When considering the compared baselines, it becomes apparent that the Sobel operator, similar to Canny's baseline, may produce weaker edges. This can be observed in Fig. 12. The presence of these weaker edges can be attributed to the Sobel operator's reliance on gradient-based edge detection, which may result in susceptibility to noise or variations in intensity resulting in faulty edge detection. For PbLite, when the edges are calculated, the weights are distributed between Canny and Sobel. This results in several outputs with good and bad outputs respectively. The output shown in Fig. 13 has equally distributed weight between the two baselines that are mentioned in the pipeline. Also, the sum of these weights adds up to 1. Pb Lite, hence, is beneficial

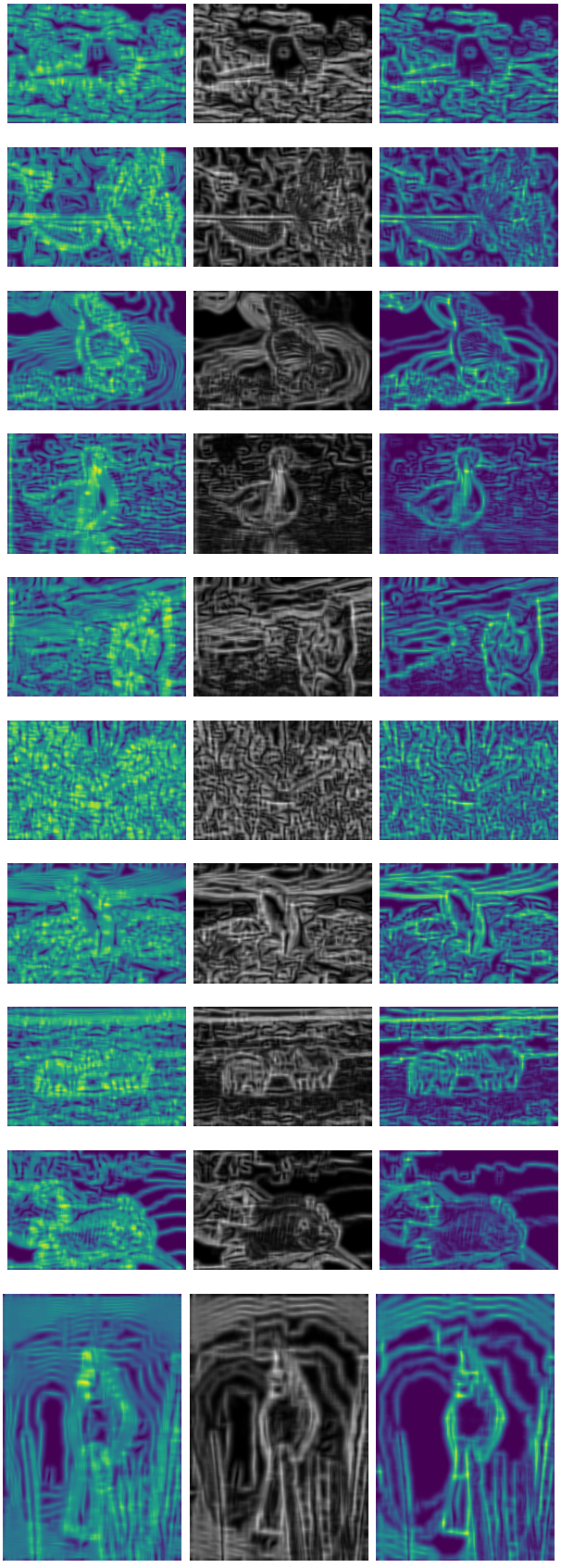

Fig. 10. $\mathcal{T}_g, B_g, \mathcal{C}_g$ for the set of test images respectively

Fig. 11. Canny Baselines

Fig. 12. Sobel Baselines

because it can be modified by varying the orientations and scales of the filter banks used as well as the weights of the baselines.

II. PHASE 2: Deep Dive on Deep Learning

This section aims to implement multiple neural network architectures and compare them on various criteria like the number of parameters, training, and test set accuracies and provide an analysis of why one architecture works better than another.

A. Dataset:

A random version of the CIFAR-10 dataset has been provided with 50,000 train images and 10,000 test images. The original dataset has 10,000 photos in a test batch, the dataset is split into five training batches and one test batch. There are precisely 1000 randomly chosen photos from each class in the test batch. The remaining photographs are divided into training batches and are arranged randomly; however, certain training batches may have more images from a particular class than others. The training batches have exactly 5000 photos from each class combined. The Fig. 14 represents the classes and images of the dataset.

For the scope of this project, we will be using the random version of the dataset since we aim to understand and implement various architectures and learn about them in the Pytorch Environment.

B. Training First Neural Network:

This is a simple PyTorch Implementation for the task of image classification. The input for this network is a single image and the output is probabilities of 10 classes. Here we learn about optimizer, loss function, and a PyTorch Neural Network Architecture and we also do some evaluation around it.

Fig. 13. Pb Lite output

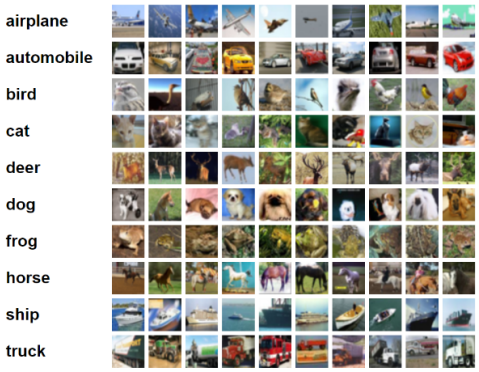

Fig. 14. Dataset

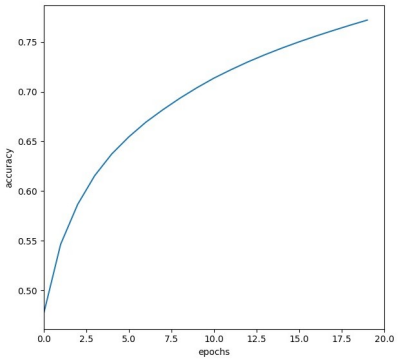

Fig. 15. Train Accuracy over Epochs

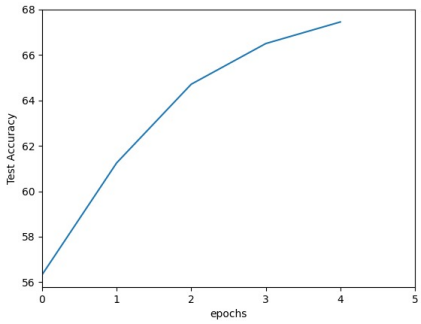

Fig. 16. Test Accuracy over Epochs

1) Model and its loss function:: Determining the model architecture, and loss function, are basic steps in the CIFAR10 Model training process in PyTorch. The model consists of ”conv1” and ”conv2” convolution layers with 16 and 32 output channels respectively. They use a ReLU activation function. In the network, there are ”fc1” and ”fc2” layers which are fully connected layers with 64 output features. These layers also utilize the ReLU activation function. A forward method defines the flow of data through the network including the application of these layers with their respective activation functions. Here we use a Cross-Entropy Loss function which is commonly used for multi-class classification tasks.

$$CrossEntropyLoss = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(p_i) + (1-y_i)\log(1-p_i)\right) \tag{1}$$

where: - N is the number of samples, y_i is the true label for sample i, p_i is the predicted probability for sample i. We now present the figures for the simple first neural network.

C. Improving the Accuracy:

We first standardize the dataset within the interval [-1, 1]. Then, we can employ data augmentation methods, such as random noise addition and random rotation of the left and right images. To make sure that the input distribution of every layer

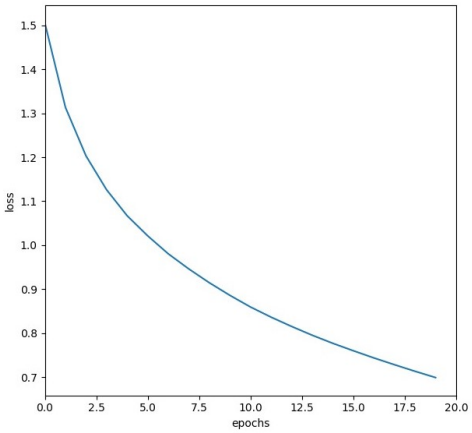

Fig. 17. Loss over Epochs

Number of Parameters	136,874
Optimizer	Adam
Hyp. Params	Lr = 0.001
Batch Size	32
Epochs	20

TABLE I
FIRST SIMPLE NEURAL NETWORK

stays roughly the same after each training stage, we also added the batch normalization layers after each convolution layer. As seen in Fig. 20, this helps avoid the internal covariance shift issue and shortens training time. Figs. 21, 22, 23, and 24 display the model statistics.

D. Residual Neural Network (ResNet):

A neural network architecture known as a residual network, or ResNet, finds the most straightforward solution to the issue of vanishing gradients. That is, as illustrated in Fig. 25, by implementing skip connections in a general residual block. As a result, the network may support deep layers without experiencing the vanishing gradient issue. Figs. 26 and 27 show the training and test set accuracy versus epochs for this network, which was only trained for 25 epochs. Fig. 28 Shows the graph of Loss over Epochs. Table II shows the other model parameters. Model shows accuracy around 58% .

E. ResNext:

ResNeXt is an extension of the ResNet architecture that introduces a new dimension called ”cardinality”. The cardinality represents the number of parallel paths within a ResNeXt block. The network is, as illustrated in Fig. 29,. As a result, by increasing the cardinality, ResNeXt can capture more diverse

Fig. 18. Confusion Matrix for Training Data of first NN

Fig. 19. Confusion Matrix for Testing Data of first NN

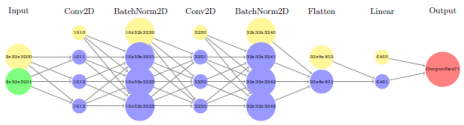

Fig. 20. Model Architecture

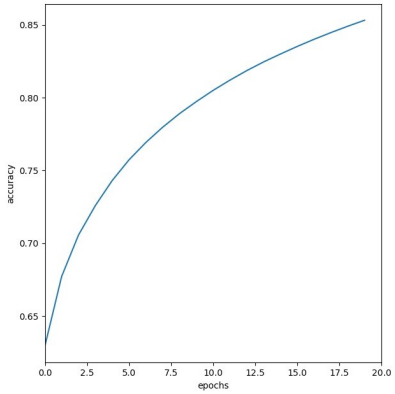

Fig. 21. Train Accuracy for improvised model

Number of Parameters	2333002
Optimizer	Adam
Hyp. Params	Lr = 0.001
Batch Size	32
Number of epochs	30

TABLE II
RESNET

features and improve the model's performance. Figs. **??** and **??** show the training and test set accuracy versus epochs for this network, which was only trained for 25 epochs. Fig. **??** Shows the graph of Loss over Epochs. Table III shows the other model parameters.

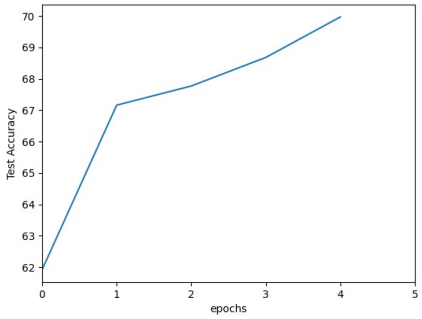

Fig. 22. Test Accuracy for improvised model

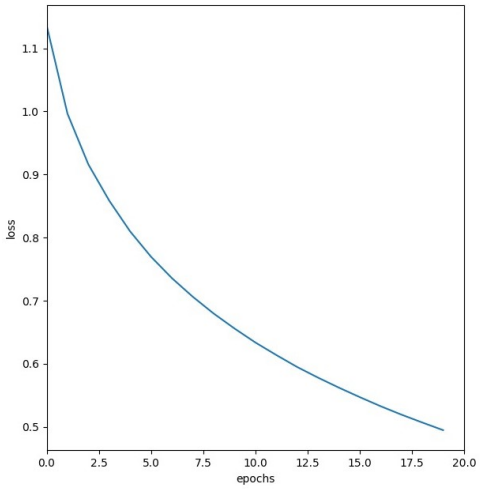

Fig. 23. Loss for improvised model

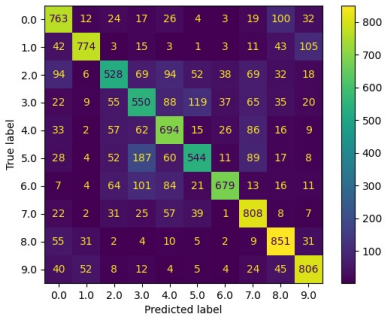

Fig. 24. Confusion Matrix for Test Data of improvised model

Number of Parameters	
Optimizer	Adam
Hyp. Params	0.001
Batch Size	32
Number of epochs	25

TABLE III
RESNEXT

F. DenseNet:

DenseNet is a prominent neural network architecture notable for its dense connectivity pattern. Every layer in a DenseNet is feed-forward connected to every other layer. This form of connectivity promotes feature propagation throughout the network, minimises the number of parameters, and allows features to be reused. The network is, as illustrated in Fig. 30. The model is made up of dense blocks with several dense layers inside of them. The input and output of each previous layer are concatenated within each thick layer. The following layers then get this concatenated input. Each dense layer also includes convolutional processes, rectified linear unit (ReLU) activation, and batch normalisation. The number of feature maps that each dense layer generates is determined by the

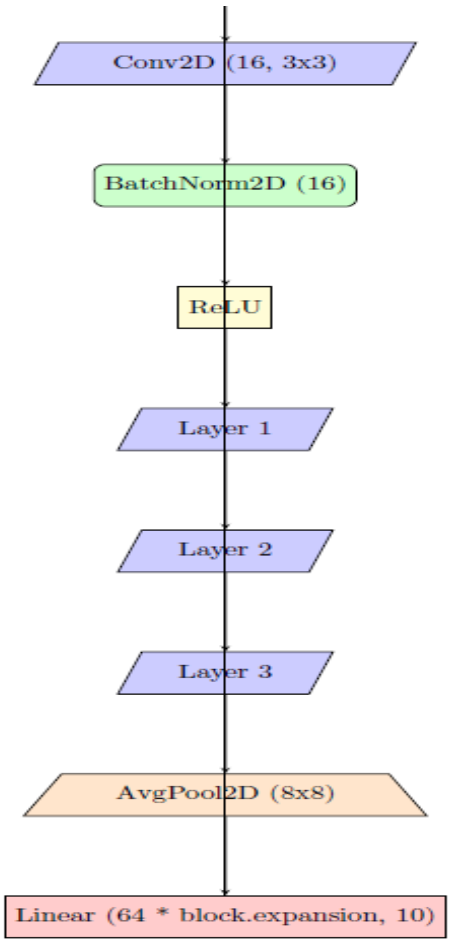

Fig. 25. ResNet Architecture

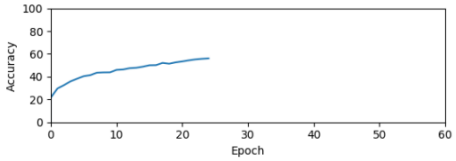

Fig. 26. ResNet Train Accuracy over Epochs

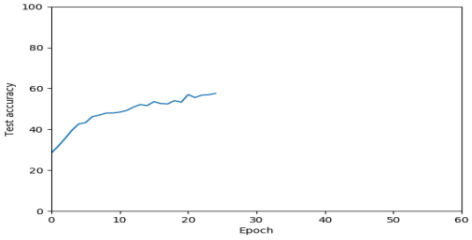

Fig. 27. ResNet Test Accuracy over Epochs

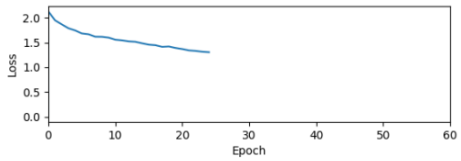

Fig. 28. ResNet Loss over Epochs

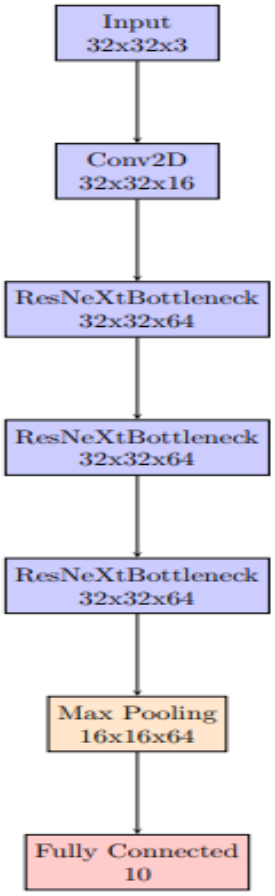

Fig. 29. ResNext Architecture

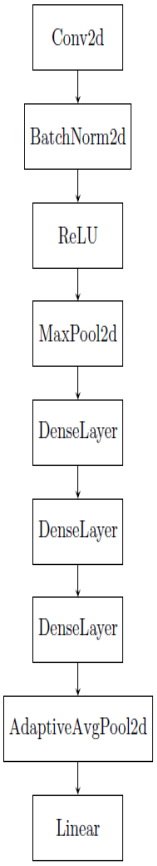

Fig. 30. DenseNet Architecture

Number of Parameters	1738714
Optimizer	Adam
Hyp. Params	Lr = 0.001
Batch Size	32
Number of epochs	30

TABLE IV
DENSENET

growth rate parameter. The number of feature maps increases as the network moves through the dense blocks, capturing progressively more intricate patterns in the input data.

Figs. **??** and **??** show the training and test set accuracy versus epochs for this network, which was only trained for 25 epochs. Fig. **??** Shows the graph of Loss over Epochs. Figs. **??** and **??** show the confusion matrix of train and test respectively. Table IV shows the other model parameters.

G. Discussion and Conclusion: Phase 2

In order to create an image classifier for this project, we used the CIFAR-10 dataset to implement various neural network topologies. Each relevant paragraph contains a table containing the hyperparameters that were utilised to train each network design. Each network's accuracy and results are given in the relevant part. It has been discovered through the exercise of implementing several types of Neural Nets that every architecture has pros and cons of its own. A basic CNN trains in a very short amount of time, however its test results are not very accurate.

It is theoretically possible to argue that as CNN gets deeper and deeper, the network's accuracy will increase, yet this is untrue. The network becomes more complex as the number of convolution layers rises, and after a certain point, the accuracy of the network drops. We refer to this issue as "overfitting" the data. In order to address this issue, many architectures were developed. By adding skip connections (ResNet), introducing cardinality (ResNext), or feed-forwarding the input straight to

each successive layer (DenseNet), these new networks each solve the overfitting issue in a different way. These are some learnings that I had while doing this homework.

REFERENCES

[1] https://www.tensorflow.org/tutorials/images/deepcnn
[2] https://www.cs.toronto.edu/ kriz/cifar.html
[3] https://docs.scipy.org/doc/scipy/tutorial/signal.html
[4] https://hannibunny.github.io/orbook/preprocessing/04gaussianDerivatives.html
[5] https://rbe549.github.io/spring2024/hw/hw0/pbliteout
[6] https://github.com/akathpal/
[7] https://en.wikipedia.org/wiki/Gabor$_{filter}$