RBE 549 - Computer Vision HW0 - Alohomora

Ashwin Disa
M.S Robotics Engineering
Worcester Polytechnic Institute (WPI)
Worcester, MA 01609
Email: amdisa@wpi.edu

***Abstract*—For phase 1, We develop a simplified version of pb (probability of boundary) detection algorithm, which finds boundaries by examining brightness, color, and texture information across multiple scales. It outperforms the classical methods (Canny and Sobel) by considering texture and color discontinuities. For phase 2, We implemented the ResNet architecture and trained on the CIFAR-10 dataset. Performance in terms of accuracy and losses is analysed.**

I. Phase 1: Shake my Boundary

Boundary detection is an important, well-studied computer vision problem. Clearly it would be nice to have algorithms which know where one object transitions to another. But boundary detection from a single image is fundamentally difficult. Determining boundaries could require object-specific reasoning, arguably making the task hard. A simple method to find boundaries is to look for intensity discontinuities in the image, also known of edges.

We compare against classical edge detection algorithms, including the Canny and Sobel baselines, look for intensity discontinuities. The more recent pb (probability of boundary) boundary detection algorithm significantly outperforms these classical methods by considering texture and color discontinuities in addition to intensity discontinuities. Qualitatively, much of this performance jump comes from the ability of the pb algorithm to suppress false positives that the classical methods produce in textured regions.

We develop a simplified version of pb, which finds boundaries by examining brightness, color, and texture information across multiple scales (different sizes of objects/image). The overview of the algorithm is shown in fig. 1.

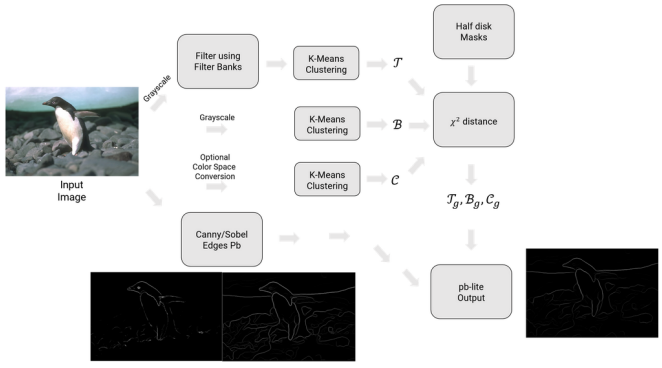


Fig. 1: Overview of the pb lite pipeline.

A. Filter Banks

The first step of the pb lite boundary detection pipeline is to filter the image with a set of filter banks. We create three different sets of filter banks for this purpose. Once we filter the image with these filters, we'll generate a texton map which depicts the texture in the image by clustering the filter responses. Let us denote each filter as $\mathcal{F}_i$ and texton map as $\mathcal{T}$.

Filtering is at the heart of building the low level features we are interested in. We will use filtering both to measure texture properties and to aggregate regional texture and brightness distributions.

1) Oriented DoG filters: A simple but effective filter bank is a collection of oriented Derivative of Gaussian (DoG) filters. These filters are created by convolving a simple Sobel filter and a Gaussian kernel and then rotating the result. Let $\wr$ be the orientations (from 0 to 360°) and s scales, we should end up with a total of $\wr \times \wr$ filters. A sample filter bank of size 2 $\times$ 16 with 2 scales and 16 orientations is shown below. We expect you to read up on how these filter banks are generated and implement them. is shown in fig. 2.

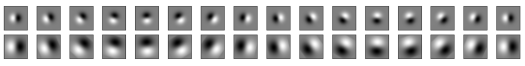


Fig. 2: Oriented DoG filter bank.

2) Leung-Malik filters: LM filters are a set of multi scale, multi orientation filter bank with 48 filters. It consists of first and second order derivatives of Gaussians at 6 orientations and 3 scales making a total of 36; 8 Laplacian of Gaussian (LOG) filters; and 4 Gaussians. We consider two versions of the LM filter bank. In LM Small (LMS), the filters occur at basic scales $\sigma = 1, \sqrt{2}, 2, 2\sqrt{2}$. The first and second derivative filters occur at the first three scales with an elongation factor of 3, i.e. ($\sigma_x = \sigma$ and $\sigma_y = 3\sigma_x$). The Gaussians occur at the four basic scales while the 8 LOG filters occur at σ and 3σ. For LM Large (LML), the filters occur at the basic scales $\sigma = \sqrt{2}, 2, s\sqrt{2}, 4$.

3) Gabor filters: Gabor Filters are designed based on the filters in the human visual system. A gabor filter is a gaussian kernel function modulated by a sinusoidal plane wave.

B. Texton, Brighness, Color Map

Filtering an input image with each element of your filter bank results in a vector of filter responses centered on each

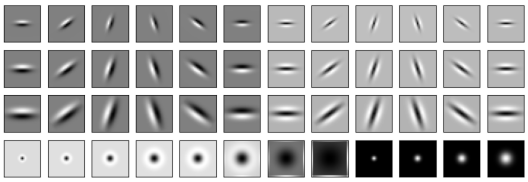


Fig. 3: Leung-Malik filter bank.

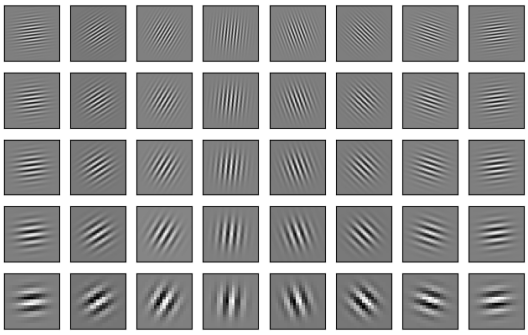


Fig. 4: Gabor filter bank.

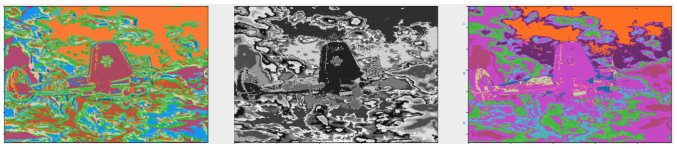
pixel. For instance, if the filter bank has $\mathcal{N}$ filters, you'll have $\wr$ filter responses at each pixel. A distribution of these N-dimensional filter responses could be thought of as encoding texture properties. We simplify this representation by replacing each N-dimensional vector with a discrete texton ID. This is done by clustering the filter responses at all pixels in the image in to $\mathcal{K}$ textons using KMeans function. Each pixel is then represented by a one dimensional, discrete cluster ID instead of a vector of high-dimensional, real-valued filter responses.

The concept of the brightness map is as simple as capturing the brightness changes in the image. Here, again we cluster the brightness values using kmeans clustering (grayscale equivalent of the color image) into a chosen number of clusters (16 clusters seems to work well, feel free to experiment). We call the clustered output as the brightness map $\mathcal{B}$.

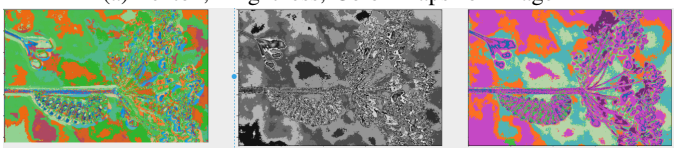
The concept of the color map is to capture the color changes or chrominance content in the image. Here, again we cluster the color values (you have 3 values per pixel if you have RGB color channels) using kmeans clustering (feel free to use alternative color spaces like YCbCr, HSV or Lab) into a chosen number of clusters (16 clusters seems to work well, feel free to experiment). We call the clustered output as the color map $\mathcal{C}$.

C. Texture, Brightness and Color Gradients

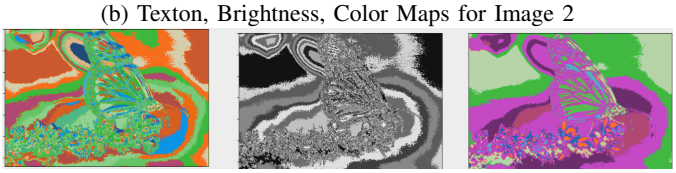
To obtain $\mathcal{T}_g, \mathcal{B}_g, \mathcal{C}_g$ we need to compute differences of values across different shapes and sizes. This can be achieved very efficiently by the use of Half-disc masks. The half-disc masks are simply (pairs of) binary images of half-discs. This is very important because it allows us to compute the $\mathcal{X}^2$



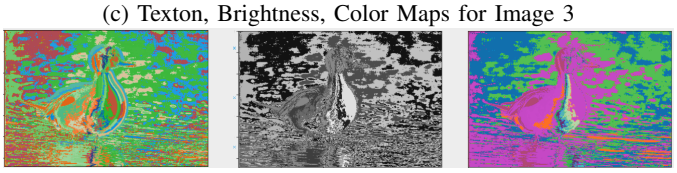
(a) Texton, Brightness, Color Maps for Image 1



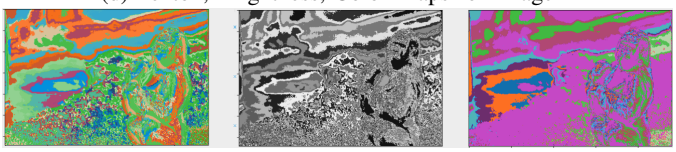
(b) Texton, Brightness, Color Maps for Image 2



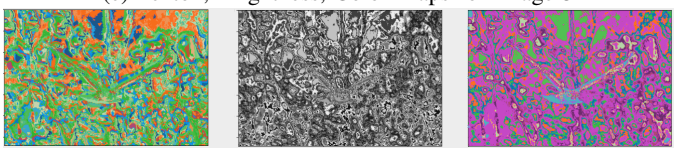
(c) Texton, Brightness, Color Maps for Image 3



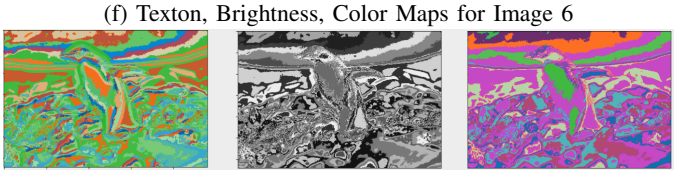
(d) Texton, Brightness, Color Maps for Image 4



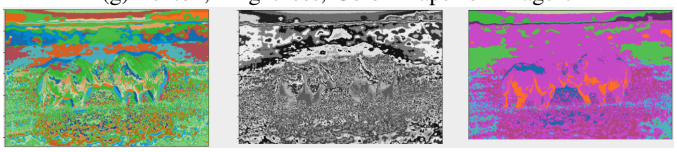
(e) Texton, Brightness, Color Maps for Image 5



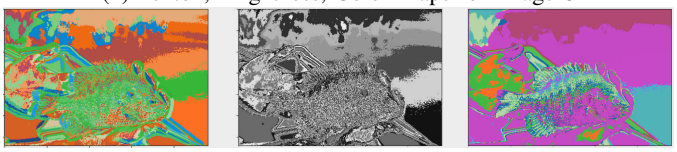
(f) Texton, Brightness, Color Maps for Image 6



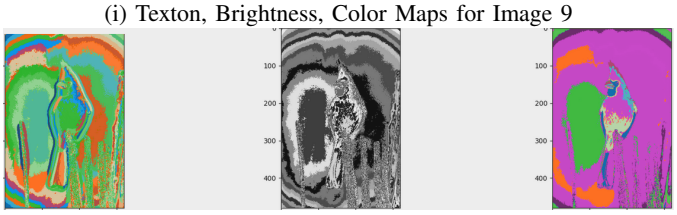
(g) Texton, Brightness, Color Maps for Image 7



(h) Texton, Brightness, Color Maps for Image 8



(i) Texton, Brightness, Color Maps for Image 9



(j) Texton, Brightness, Color Maps for Image 10

(chi-square) distances using a filtering operation, which is much faster than looping over each pixel neighborhood and aggregating counts for histograms. Forming these masks is quite trivial. A sample set of masks (8 orientations, 3 scales) is shown in fig. 6.

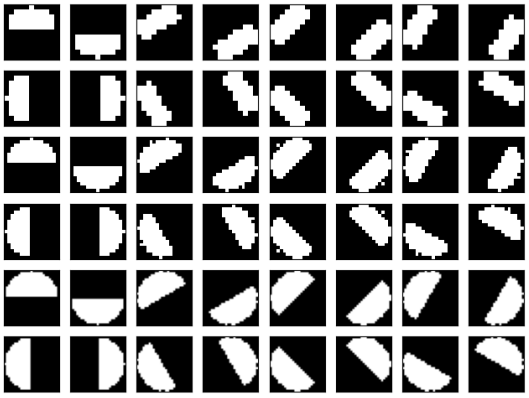


Fig. 6: Half disc masks at different scales and sizes.

$\mathcal{T}_g, \mathcal{B}_g, \mathcal{C}_g$ encode how much the texture, brightness and color distributions are changing at a pixel. We compute $\mathcal{T}_g, \mathcal{B}_g, \mathcal{C}_g$ by comparing the distributions in left/right half-disc pairs (opposing directions of filters at same scale, in fig. 6, the left/right pairs are shown one after another, these are easy to create as you have control over the angle) centered at a pixel. If the distributions are the similar, the gradient should be small. If the distributions are dissimilar, the gradient should be large. Because our half-discs span multiple scales and orientations, we will end up with a series of local gradient measurements encoding how quickly the texture or brightness distributions are changing at different scales and angles.

D. Pb-lite Output

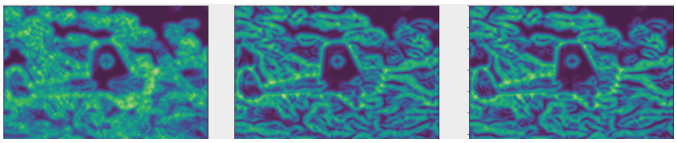
The final step is to combine information from the features with a baseline method (based on Sobel or Canny edge detection or an average of both) using a simple equation as shown below

$$PbEdges = \frac{\mathcal{T}_g + \mathcal{B}_g + \mathcal{C}_g}{3} \odot (w_1 * cannyPb + w_2 * sobelPb) \tag{1}$$

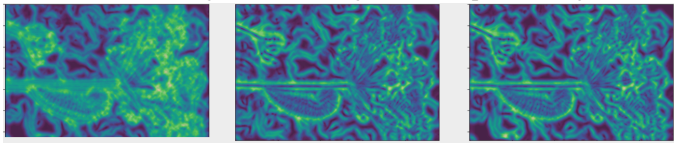
Here, $\odot$ is the Hadamard product operator. A simple choice for $w1$ and $w2$ would be 0.5 (they have to sum to 1).

E. Results

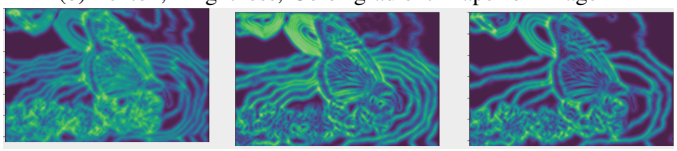
It is observed the Canny baseline output gives high number of false positives whereas Sobel baseline outputs are suppressed. The performance of Pb-lite lies between the too as can be seen in the comparison images.



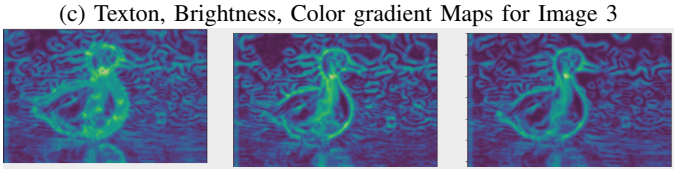
(a) Texton, Brightness, Color gradient Maps for Image 1



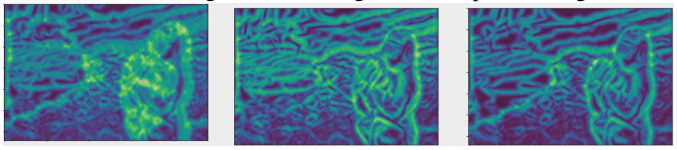
(b) Texton, Brightness, Color gradient Maps for Image 2



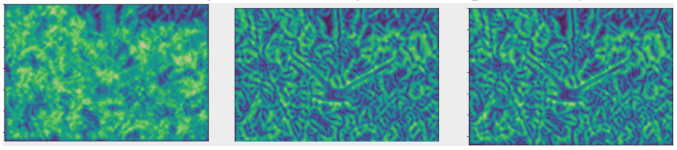
(c) Texton, Brightness, Color gradient Maps for Image 3



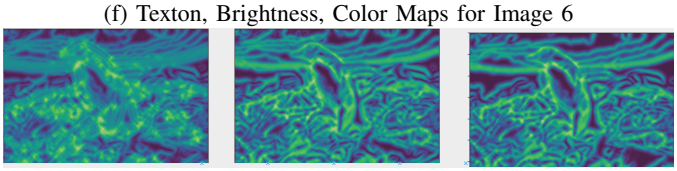
(d) Texton, Brightness, Color gradient Maps for Image 4



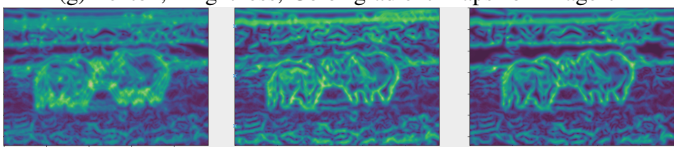
(e) Texton, Brightness, Color gradient Maps for Image 5



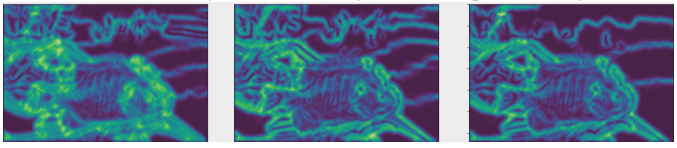
(f) Texton, Brightness, Color Maps for Image 6



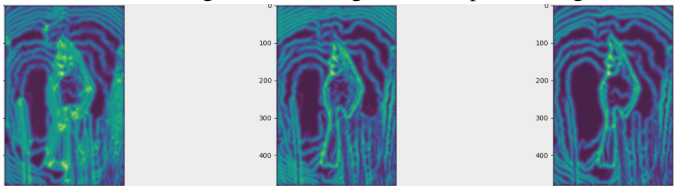
(g) Texton, Brightness, Color gradient Maps for Image 7



(h) Texton, Brightness, Color gradient Maps for Image 8

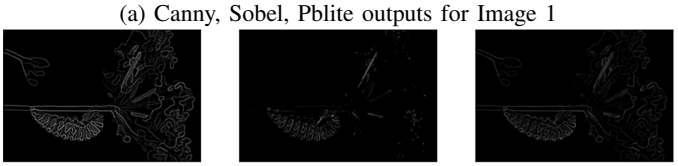


(i) Texton, Brightness, Color gradient Maps for Image 9

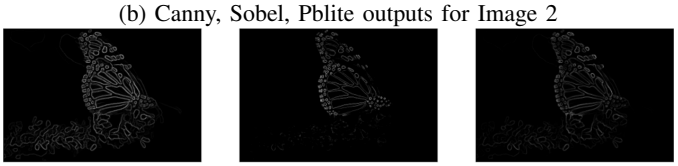


(j) Texton, Brightness, Color gradient Maps for Image 10

(a) Canny, Sobel, Pblite outputs for Image 1

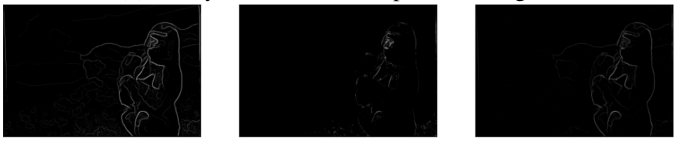


(b) Canny, Sobel, Pblite outputs for Image 2

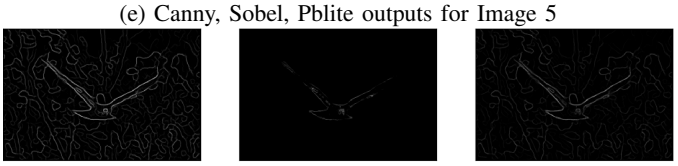


(c) Canny, Sobel, Pblite outputs for Image 3

(d) Canny, Sobel, Pblite outputs for Image 4



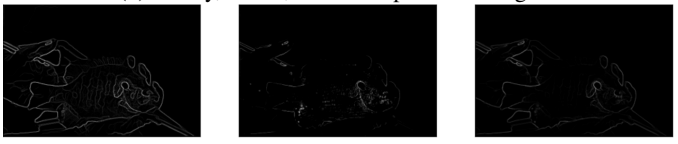
(e) Canny, Sobel, Pblite outputs for Image 5



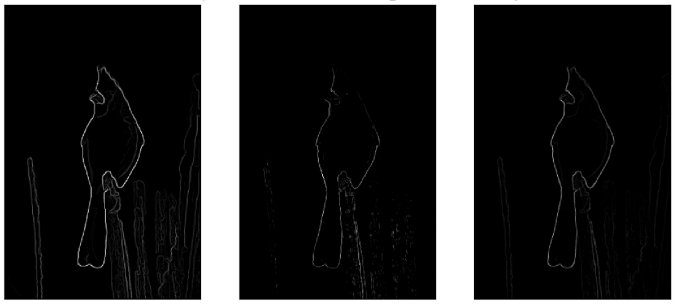
(f) Canny, Sobel, Pblite outputs for Image 6

(g) Canny, Sobel, Pblite outputs for Image 7

(h) Canny, Sobel, Pblite outputs for Image 8



(i) Canny, Sobel, Pblite outputs for Image 9



(j) Canny, Sobel, Pblite outputs for Image 10

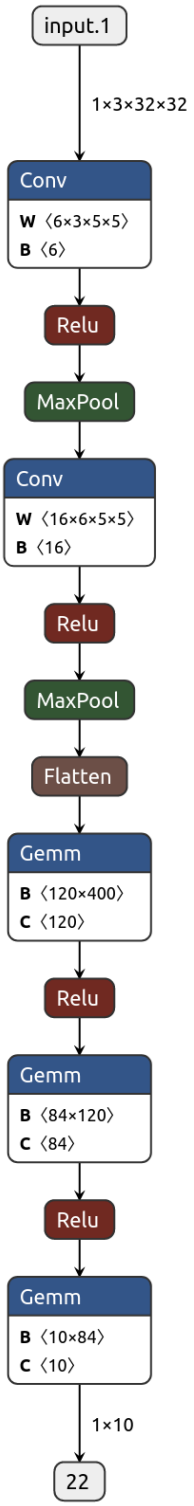


Fig. 9: Basic CNN architecture.

II. Phase 2: Deep Dive on Deep Learning

Here, multiple neural network architectures are implemented on the CIFAR-10 dataset and their performance are analysed using the loss and accuracy by comparing against

each other. The input images from the dataset are of size $(3 \times 32 \times 32)$, and there are 50000 training and 10000 testing images.

A. Basic Convolutional Neural Network

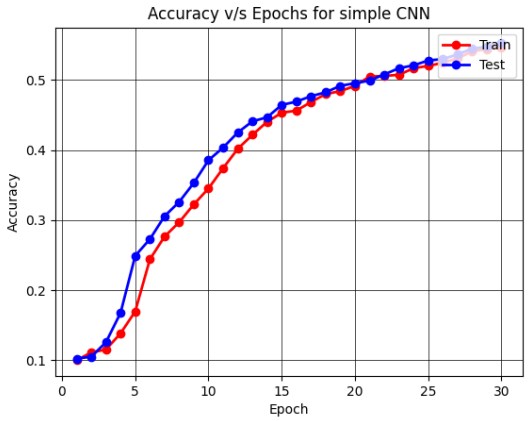


Fig. 10: Accuracy v/s number of epoch for train and test sets.

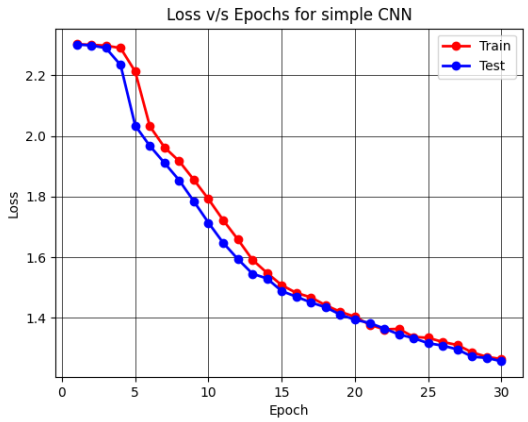


Fig. 11: Loss v/s number of epoch for train and test sets.

```
[735 113   0   5   0  18   0  12  73  44] (0)
[ 57 802   0   4   0   8   0   8  48  73] (1)
[272  75  52  42   5 342   2 127  35  48] (2)
[136  75   2 131   0 426   2  92  29 107] (3)
[213  83  16  45  38 300   2 230  33  40] (4)
[ 95  43   4  60   4 616   1 114  17  46] (5)
[ 76 140   6 143   6 346  33  85  54 111] (6)
[119  52   0  14   2 107   0 599   8  99] (7)
[271 213   0   2   0  15   0  13 425  61] (8)
[ 78 429   0   3   0  11   1  16  32 430] (9)
  (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 38.61 %
```

Fig. 12: Confusion matrix of Test set.

The implemented CNN has a total of 2 convolutional layers in the network. The activation function for each layer is a $ReLu()$ activation function. $MaxPool2D()$ function is used to scale down the feature map after every convolutional layer. 3 fully connected layers are implemented after the convolutional layers using the $Linear()$ function and the final output of the third fully connected layer is used to classify the images into 10 classes. The architecture is shown in fig. 9.

The hyperparameters for this architecture are as follows, SGD (Stochastic Gradient Descent) is the optimizer, the learning rate used is $1e-3$, the mini-batch size is set to 128 and number of Epochs are 30.

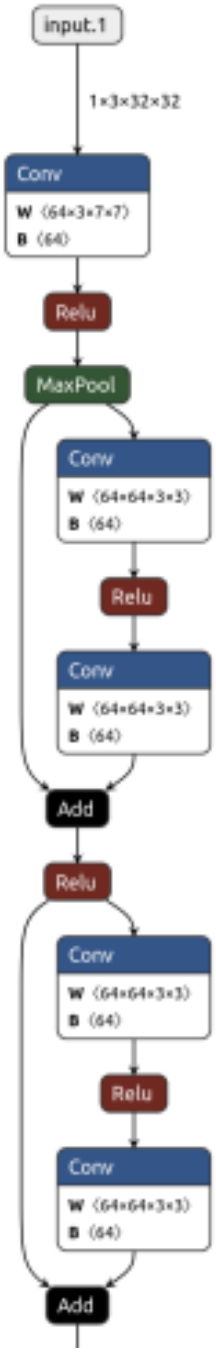


Fig. 13: ResNet model architecture.

The train and test set accuracy are found out to be very similar over 30 epochs. Similar with the model accuracy. This is shown in fig. 11 and fig. 10. And the confusion matrix for the Test set is shown in fig. 12.

B. ResNet

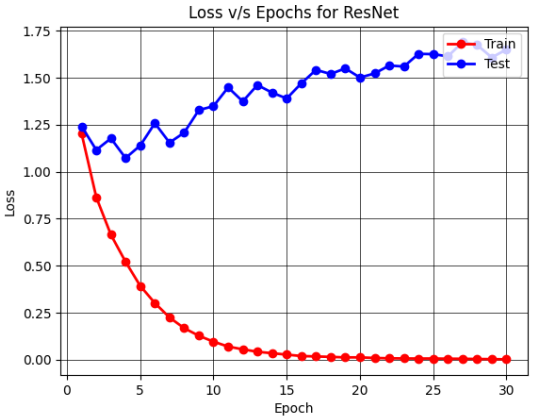


Fig. 14: Loss v/s number of epoch for train and test sets.

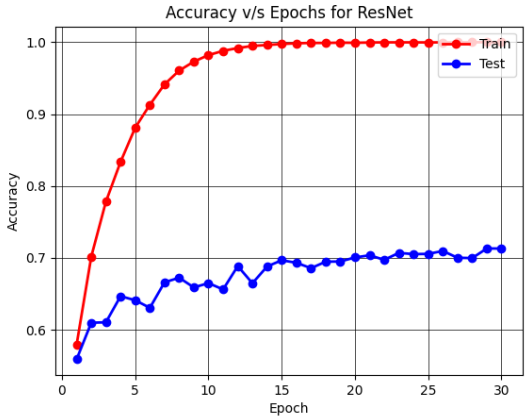


Fig. 15: Accuracy v/s number of epoch for train and test sets.

In ResNet, we add a skip connection from the previous layer. This ensures the model does not go in the other way if it does not improve and maintain it's accuracy. The fundamental building blocks of ResNet are residual blocks. These blocks contain a shortcut connection, or a skip connection, that allows the gradient to flow directly through the network without passing through too many layers. This helps in mitigating the vanishing gradient problem and enables the training of very deep networks.

The shortcut connection adds the original input to the output of the residual block. If the input and output dimensions are not the same, a linear projection is applied to match the dimensions. This identity shortcut connection aids in the efficient training of deep networks. The model architecture is shown in fig. 13.

The hyperparameters for this architecture are as follows, SGD (Stochastic Gradient Descent) is the optimizer, the learning rate used is $1e-3$, the mini-batch size is set to 128 and number of Epochs are 30.

The plots for Loss and model accuracy for the train and test sets are shown in fig. 14 and 15. The final accuracy for the model in the Test set is about 70%.

After comparing, the Resnet model shows overall better performance than the simple CNN architecture implementation.