RBE549: Homework0 - Alohomora

Hrishikesh Pawar
Email: hpawar@wpi.edu
Using 2 late days

I. Phase 1: Shake My Boundary

The primary objective of this phase is to develop a simplified version of the pb-lite (Probability of Boundary) boundary detection algorithm. This algorithm computes the likelihood of each pixel in an image being a part of an edge or boundary. In addition to the image intensity discontinuities considered in the classical edge detection algorithms like Canny and Sobel; pb-lite algorithm considers the texture and colour discontinuities.

The process of implementing the pb-lite algorithm is carried out in the following steps:

1) Generating sets of filter-banks.
2) Generating Texton, Brightness and Colour maps.
3) Generating Texton, Brightness and Colour gradient maps.
4) Combining features from the Texton, Brightness and Colour maps along with the Canny and Sobel baselines for boundary detection.

A. Generating set of Filter Banks

To extract textural features in an image, filters from 3 sets of filter banks are used. These filters vary in scale and orientation. Following 3 filter banks are used.:

1) Oriented Difference of Gaussian (DoG) Filters: These filters are generated by convolving a Sobel filter with a Gaussian filter and then rotating the result. The generated filter with 2 scales and 16 orientations is shown in Fig 1.

2) Leung-Malik (LM) Filters: LM filter bank consists of 48 filters varying in both scale and orientation. The first 36 filters are first-order and second-order derivative of Gaussian respectively. The first 18 filters (first-order derivatives) consists 3 scales of 6 orientations each. Likewise the next 18 filters (second-order derivatives) as well. The rest of the 12 filters consist of 8 Laplaciian of Gaussian filters and 4 Gaussian filters. This together constitutes the LM filter bank. Two versions of the LM filter bank are generated:

1) Small LM filter bank with scales $\sigma = \{1, \sqrt{2}, 2, 2\sqrt{2}\}$.
2) Large LM filter bank with scales $\sigma = \{\sqrt{2}, 2, 2\sqrt{2}, 4\}$

The first three sclaes are used for generating the first 36 first and second-order derivative of Gaussians with an elongation factor of 3 i.e ($\sigma_x = \sigma$ and $\sigma_y = 3\sigma_x$). The 8 Laplacian of Gaussian filters are generated at the four basic scales ($\sigma = \{1, \sqrt{2}, 2, 2\sqrt{2}\}$ for LM Small) and ($\sigma = \{\sqrt{2}, 2, 2\sqrt{2}, 4\}$ for LM Large) at σ and 3σ. 4 Gaussian filters are generated are 4 basic scales ($\sigma = \{1, \sqrt{2}, 2, 2\sqrt{2}\}$ for LM Small) and ($\sigma = \{\sqrt{2}, 2, 2\sqrt{2}, 4\}$ for LM Large).

3) Gabor Filters: Gabor filters are Gaussian kernels modulated by a sinusoidal plane wave. It is a linear filter used for texture analysis which emulates filters in human visual system.

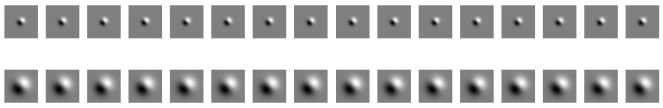

Fig. 1: DoG Filter Bank

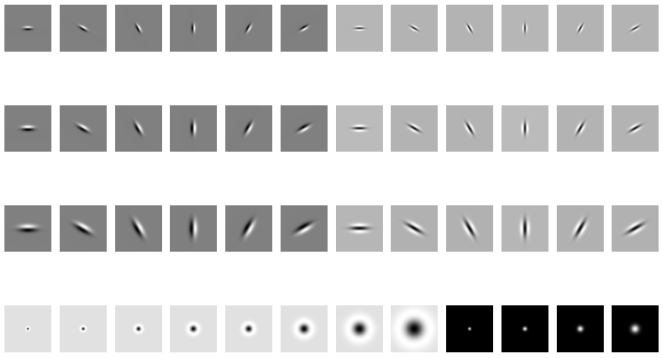

Fig. 2: LM Small Filter Bank

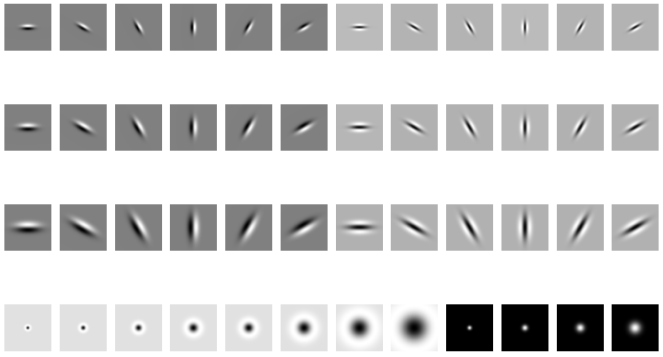

Fig. 3: LM Large Filter Bank

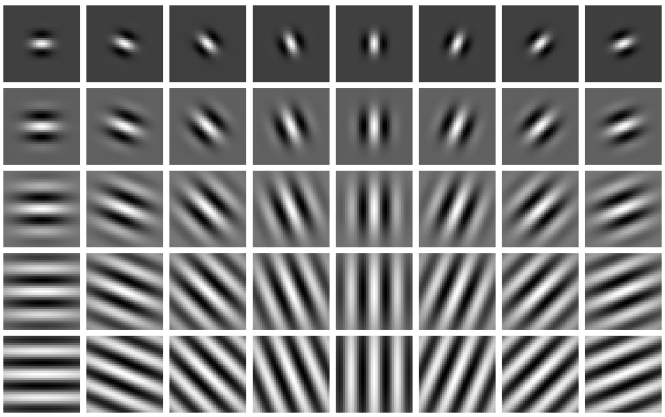

Fig. 4: DoG Filter Bank

B. Generating Texton, Brightness and Colour maps.

The next step involves applying each filter from the generated filter banks to the input image. This results in a vector or responses at each pixel in the image. Therefore, for N filters we will obtain a vector of dimension N at each pixel. This essentially can be thought of as encoding the texture properties of the image.

Following this, the next step is to replace N-dimensional vector into a distinct texton ID for each pixel. This transformation is achieved by categorizing the filter responses of each pixel into K distinct textons using the KMeans clustering. Consequently, each pixel in the image is substituted with its respective texton ID derived from the clustering, resulting in the creation of the Texton map (T). The outcome is a single-channel image where the pixel values range between 1 and K. For Texton Map generation, K was set to 64.

Similarly, the brightness and color maps are generated. The brightness map is generated by clustering the intensity values. This is achieved by applying KMeans clustering onto a gray-scale image generating brightness map (B). Likewise performing KMeans clustering on the standard three-channel color image yields the Color map (C). For both the brightness and color maps, a K value of 16 was chosen.

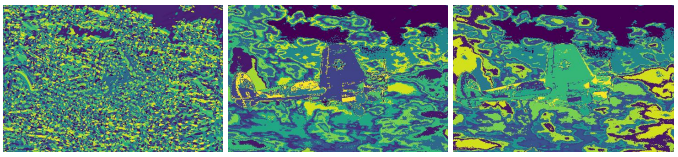

Fig. 5: T, B, and C for image 1.

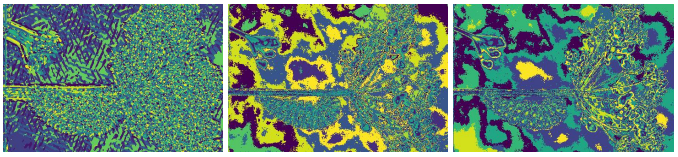

Fig. 6: T, B, and C for image 2.

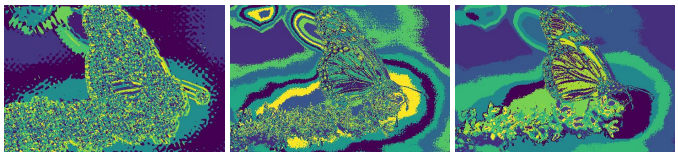

Fig. 7: T, B, and C for image 3.

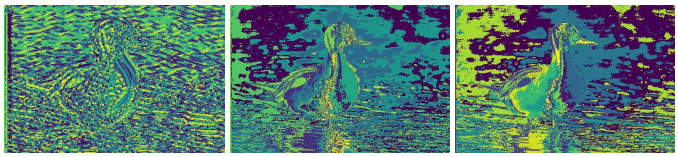

Fig. 8: T, B, and C for image 4.

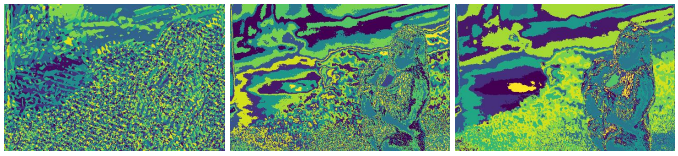

Fig. 9: T, B, and C for image 5.

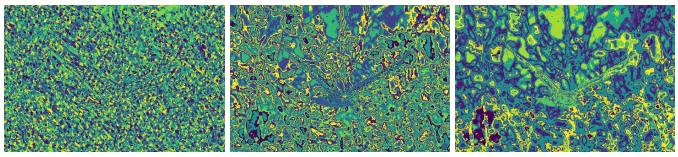

Fig. 10: T, B, and C for image 6.

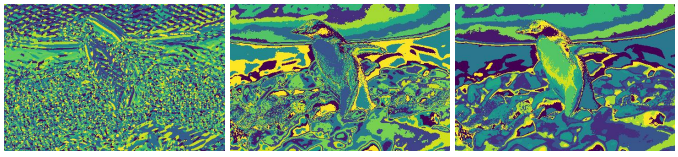

Fig. 11: T, B, and C for image 7.

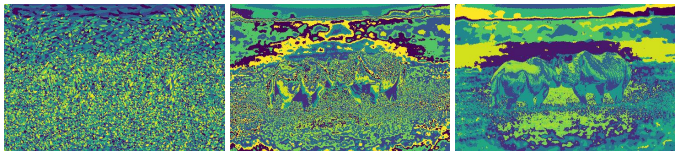

Fig. 12: T, B, and C for image 8.

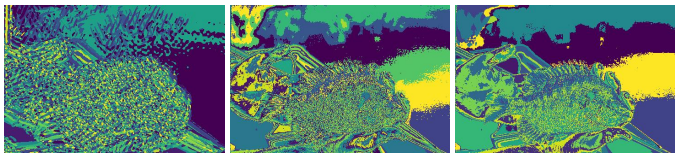

Fig. 13: T, B, and C for image 9.

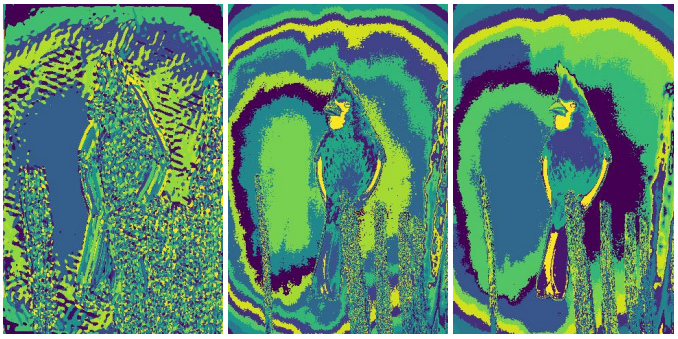

Fig. 14: T, B, and C for image 10.

C. Generating Texton, Brightness and Colour gradient maps.

The creation of the texton, brightness, and color maps for the input images serves to identify the gradients within these maps. This is crucial for understanding the areas of pixel neighborhoods where there are noticeable changes in texture, intensity, and color attributes. They can be thought as encoding the texture, brightnesss and color distribution changes at a pixel. To determine these gradients, we employ half-disc masks, as depicted in Fig. 35 (opposing directions of filters at same scale, the left/right pairs are shown one after another). These masks consist of pairs of binary images shaped like half-discs and are instrumental in computing the χ^2 distances. Half-disc masks in 8 different orientations and 3 scales were created.

The filtering of the texton, brightness, and color maps using these masks enables the calculation of the χ^2 distances between two histograms, g and h. This is achieved using the formula:

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

Employing this method, we obtain the texton gradient map (T_g), the brightness gradient map (B_g), and the color gradient map (C_g).

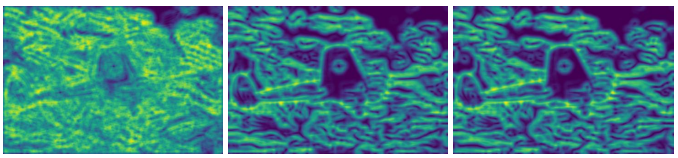

Fig. 15: T_g, B_g, and C_g for image 1.

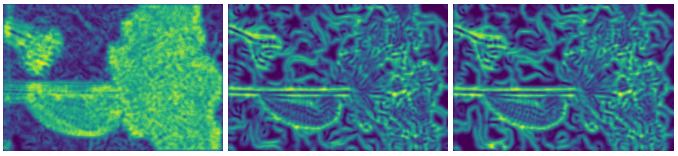

Fig. 16: T_g, B_g, and C_g for image 2.

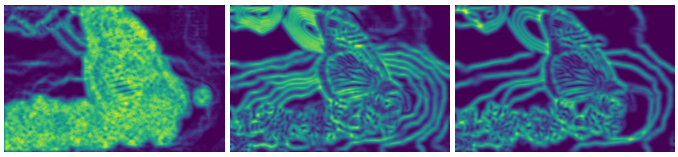

Fig. 17: T_g, B_g, and C_g for image 3.

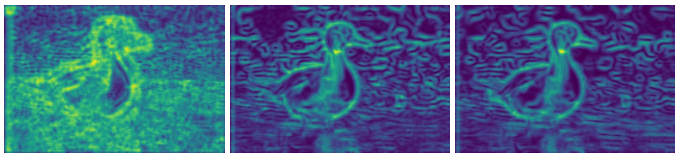

Fig. 18: T_g, B_g, and C_g for image 4.

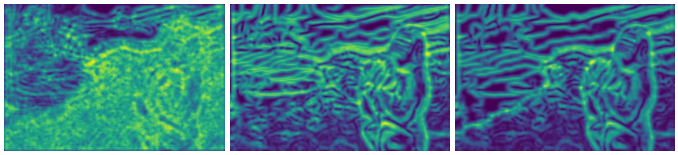

Fig. 19: T_g, B_g, and C_g for image 5.

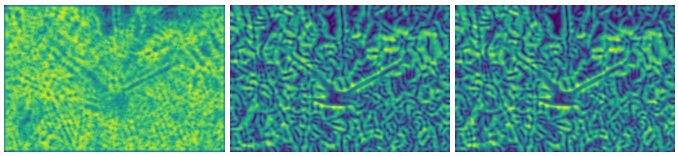

Fig. 20: T_g, B_g, and C_g for image 6.

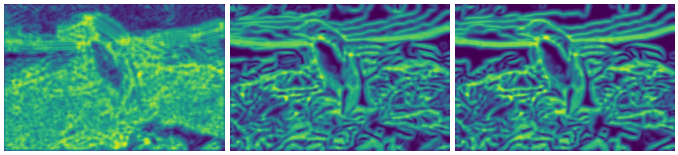

Fig. 21: T_g, B_g, and C_g for image 7.

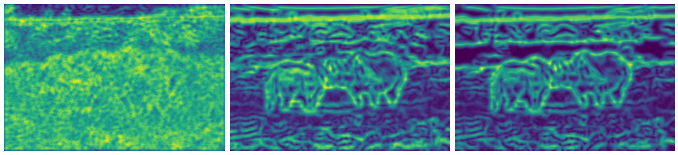

Fig. 22: T_g, B_g, and C_g for image 8.

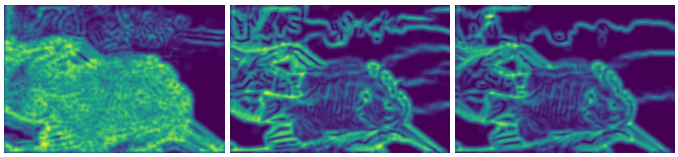

Fig. 23: T_g, B_g, and C_g for image 9.

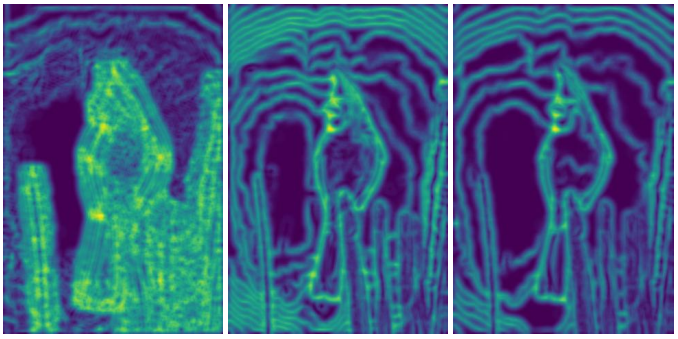

Fig. 24: T_g, B_g, and C_g for image 10.

D. Combining features from the Texton, Brightness and Colour maps along with the Canny and Sobel baselines for boundary detection.

The last step is to generate the pb-lite output using the Sobel and Canny baselines and all the Texture (T_g), Brightness (B_g) and Color (C_g) gradient maps.The pb-lite out is generated using the formula:

$$\text{PbEdges} = \frac{T_g + b_c + c_g}{3} \circ (w_1 \times \text{cannyPb} + w_2 \times \text{sobel}(Pb))$$

w_1 and w_2 are both 0.5.

The comparison of the pb-lite outputs with the Sobel and Canny baselines can be seen in Fig

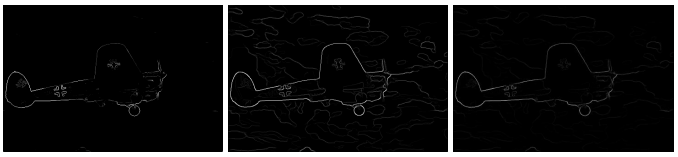

Fig. 25: Sobel, Canny and pb-lite outputs for image 1.

Fig. 26: Sobel, Canny and pb-lite outputs for image 2.

Fig. 27: Sobel, Canny and pb-lite outputs for image 3.

E. Analysis

The pb-lite implementation significantly reduced the false positives, common with the Sobel and Canny baselines particularly in images with rich texture information. This suppression is a key-advantage of pb-lite as it targets areas with

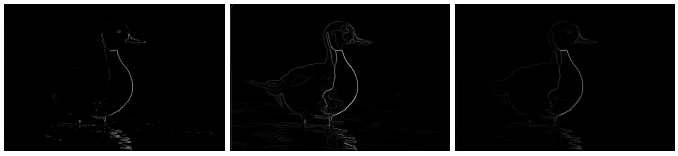

Fig. 28: Sobel, Canny and pb-lite outputs for image 4.

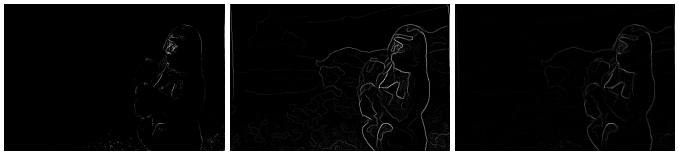

Fig. 29: Sobel, Canny and pb-lite outputs for image 5.

Fig. 30: Sobel, Canny and pb-lite outputs for image 6.

Fig. 31: Sobel, Canny and pb-lite outputs for image 7.

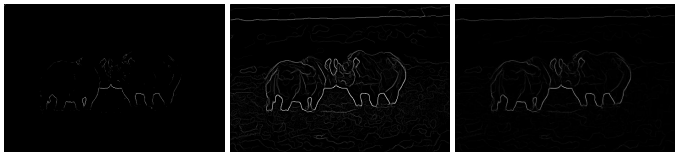

Fig. 32: Sobel, Canny and pb-lite outputs for image 8.

Fig. 33: Sobel, Canny and pb-lite outputs for image 9.

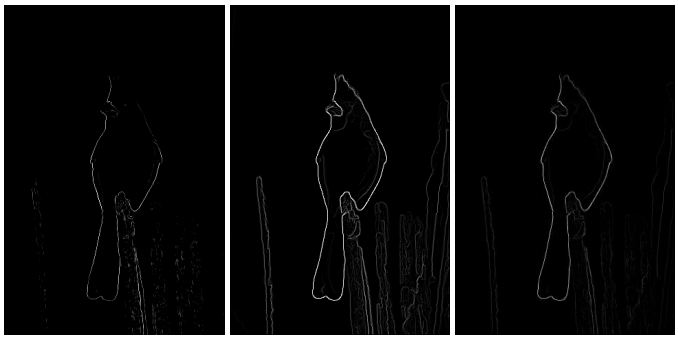

Fig. 34: Sobel, Canny and pb-lite outputs for image 10.

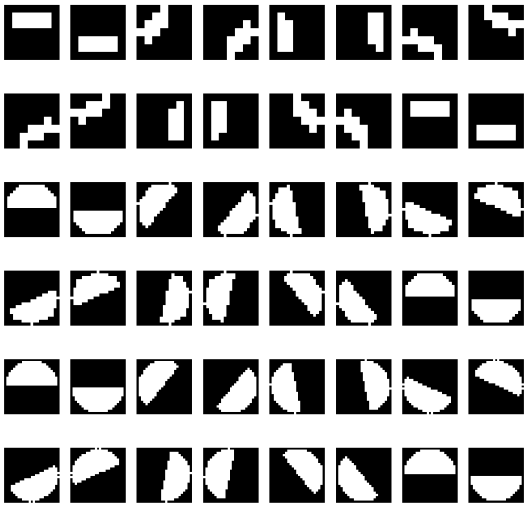

Fig. 35: Half-Disc Masks

excessive texture. However, this also seems to lead to a loss of information making outputs less clear than Canny baselines. Another observation is that pb-lite not only addresses the excessive edge detection seen in Canny but also addresses for the missed information in the Sobel baselines. Additionally, there is flexibility component to the algorithm given that it can be fine tuned based on different scales and orientations of filters in various filter banks.

Despite all the advantages I feel optimizing pb-lite can be challenging since it requires a lot of tuning in terms of the scales and orientations of the filter banks and the weights to get the desired output.

II. Phase 2: Deep Dive on Deep Learning

Using the CIFAR-10 is a dataset a set of networks are trained. The list of the trained models are as follows:

1) Baseline Network.
2) BatchNorm Network.
3) ResNet
4) ResNeXt.
5) DenseNet.

A. Baseline Network

The architecture of the Baseline Network consists of 4 convolutional layers, 3 fully-connected layers along with max pooling layers. The network begins with a series of convolutional layers (conv1 to conv4) that progressively ncrease in depth from 16 to 128 filters, extracting increasingly complex features. These layers are interspersed with MaxPool2d layers that reduce the spatial dimensions of the feature maps, enhancing the network's ability to focus on important features while reducing computational complexity. After the convolutional stages, the network transitions to fully connected layers (fc1 to fc3) where the flattened output from the convolutional layers is processed to make final class predictions.

The network was trained for 40 epochs with a batch-size of 256 with a learning-rate of 1e-3 with SGD optimiser an Cross-Entropy loss function.

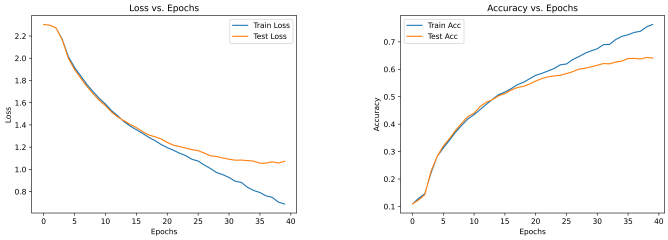

Fig. 36: Baseline Net Performance

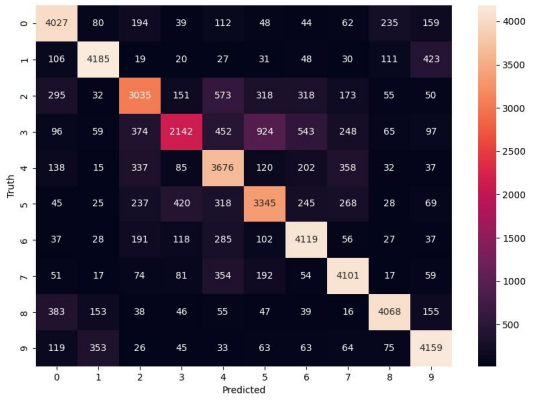

Fig. 37: Baseline Train Confusion Matrix

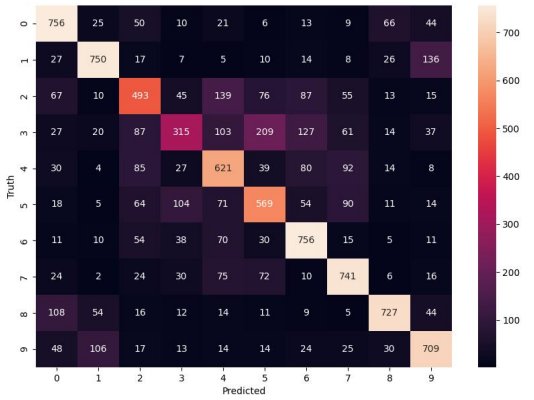

Fig. 38: Baseline Test Confusion Matrix

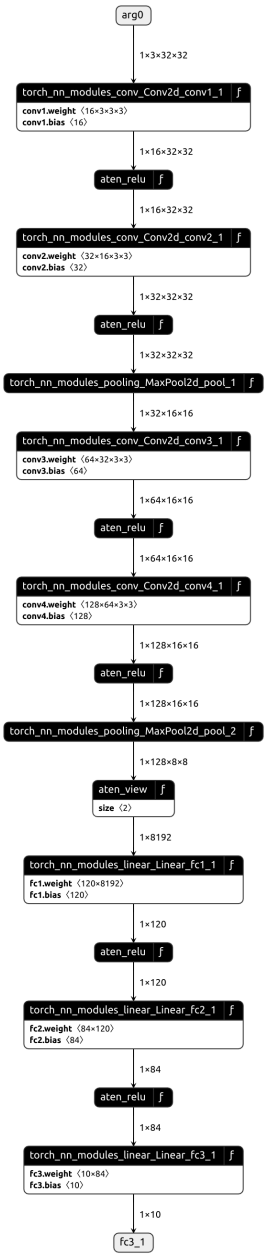

Fig. 39: Baseline Network Architecture

B. *BatchNorm Network*

This architecture retains the initial framework of convolutional layers, each followed by a ReLU activation. The sequence of convolutional layers with increasing filter depth (from 16 to 128) remains the same. The enhancement in this version is the inclusion of Batch Normalization layers (bn1 to bn4) following each convolutional layer (conv1 to conv4). Batch Normalization standardizes the output of each convolutional layer, addressing the problem of internal covariate shift. The network then transitions to fully connected layers (fc1, fc2, fc3), where the flattened output from the convolutional stages is utilised for classification.

The network was trained for 40 epochs with a batch-size of 256 with a learning-rate of 1e-3 with SGD optimiser an Cross-Entropy loss function.

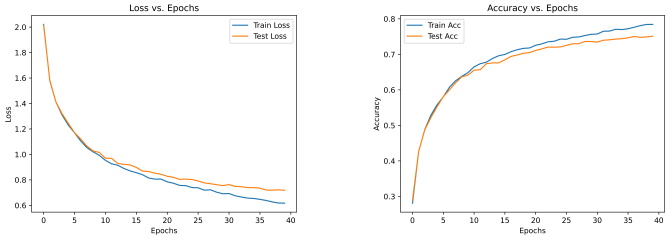

Fig. 40: BatchNorm Net Performance

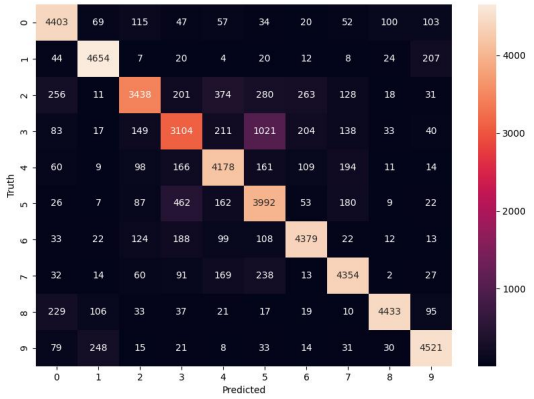

Fig. 41: BatchNorm Train Confusion Matrix

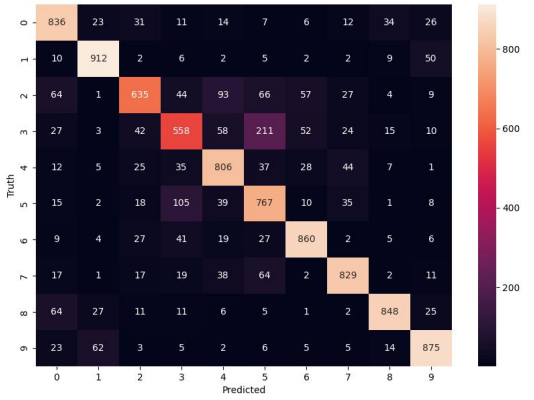

Fig. 42: BatchNorm Test Confusion Matrix

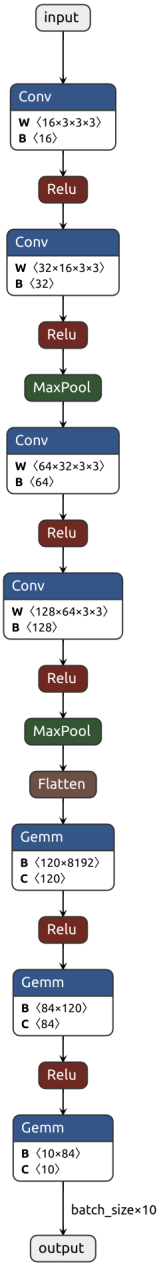

Fig. 43: BatchNorm Network Architecture

C. ResNet Network

The following architecture introduces the residual block. This block contains two convolutional layers (conv1 and conv2) each followed by batch normalization. This block integrates a residual connection that sums the input (residual) to the output of the convolutional layers. ResNet architecture is built upon this residual blocks. It begins with an initial convolution layer (conv1) and batch normalization (bn1), before the residual blocks. This model has a sequence of 5 residual blocks. This enables the model to learn complex features at various scales while handling the problem of vanishing gradients. The consequent layers include an adaptive average pooling layer, followed by a fully connected layer (fc) for classification.

The network was trained for 40 epochs with a batch-size of 256 with a learning-rate of 1e-3 with SGD optimiser an Cross-Entropy loss function.

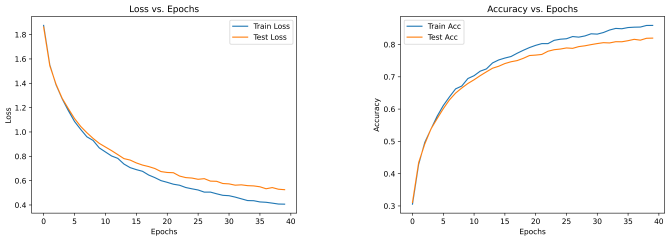

Fig. 44: ResNet Net Performance

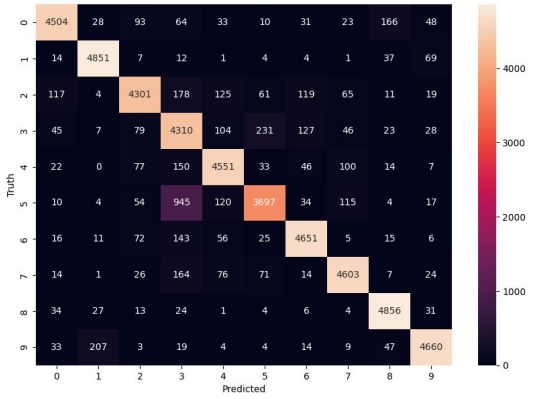

Fig. 45: ResNet Train Confusion Matrix

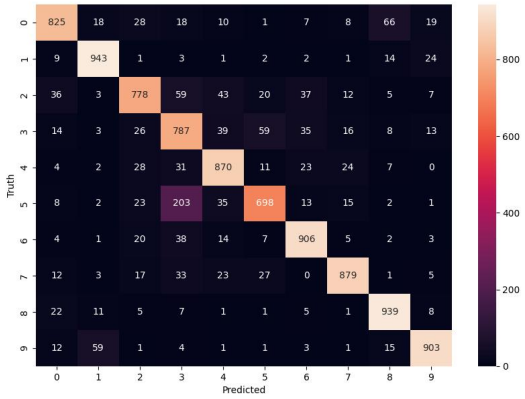

Fig. 46: ResNet Test Confusion Matrix

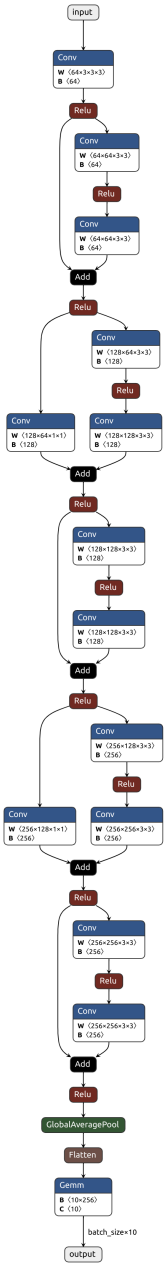

Fig. 47: ResNet Network Architecture

D. ResNeXt Network

The implementation of ResNeXt network builds upon the principle of the ResNet model while incorporating the concept of cardinality, which defines the number of parallel paths in each of the ResNeXt block. The current implementation of the ResNext block consists of a bottleneck design with three convolutional layers. The first and the third convolutional layers (conv1 and conv3) are 1x1 convolutions with an intermediate 3x3 convolutional layer (conv2) in the middle. The novelty in the ResNeXt block lies in this 3x3 layer, which operates on grouped convolutions defined by the cardinality parameter, enhancing the network's ability to learn more complex features while being computationally efficient. Cardinality for the current implementation is 16.

In this specific implementation, the ResNeXt model, initializes with a convolutional layer (conv1) followed by batch normalization (bn1) and a max-pooling layer. This is succeeded by a series of residual blocks organized into four stages, each stage characterized by a different output feature size (256, 512, 1024, 2048). The number of blocks per stage is uniformly set to three. Downsampling in the residual blocks ensures appropriate dimension matching for the shortcut connections. The network concludes with an adaptive average pooling layer (avgpool), which aggregates the feature maps to a fixed size, followed by a fully connected layer (fc) for classification.

The network was trained for 40 epochs with a batch-size of 256 with a learning-rate of 1e-3 with SGD optimiser an Cross-Entropy loss function.

The parallel layers defined by cardinality are not visible in the Netron visualisation in Fig. 51. However the implementation is see in Fig. 52.

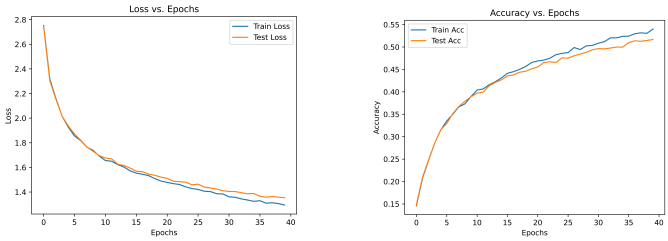

Fig. 48: ResNeXt Net Performance

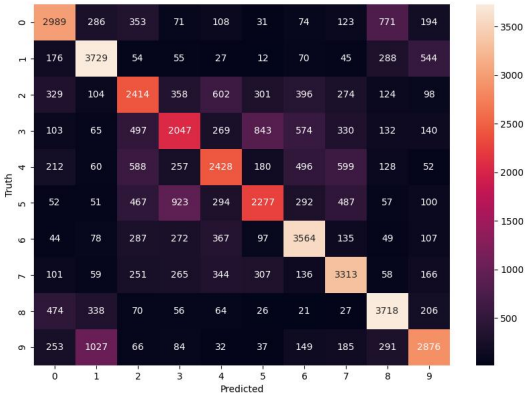

Fig. 49: ResNeXt Train Confusion Matrix

E. DenseNet Network

This implementation of DenseNet network has a sequence of DenseBlock and TransitionLayer modules. Each DenseBlock contains multiple DenseLayer units. In a DenseLayer the input undergoes batch normalization, followed by a 1x1 convolution for feature reduction. Subsequently, it is processed

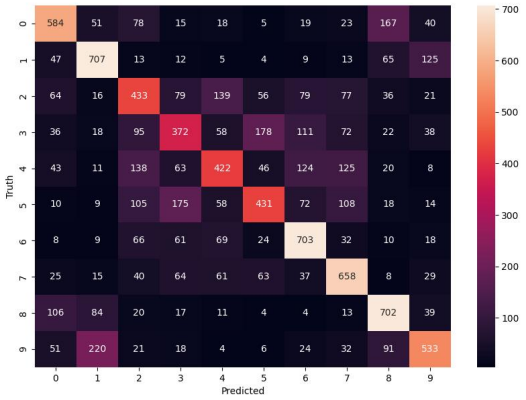

Fig. 50: ResNeXt Test Confusion Matrix

through another batch normalization and a 3x3 convolution, effectively accumulating features throughout the depth of the block. The TransitionLayer modules interspersed between the dense blocks serve to compress the feature maps. This is achieved through a combination of batch normalization, a 1x1 convolution, and average pooling. In this specific implementation the network starts with an initial convolutional layer (conv1) and batch normalization (bn1). This is followed by a series of dense blocks and transition layers as defined by the block config. The growth rate determining the increase in feature maps per dense layer is set to 16. After the dense block the network includes final bach normalization (bn2) and an adaptive average pooling layer (avgpool) followed by a fully connected layer (fc).

The network was trained for 40 epochs with a batch-size of 64 with a learning-rate of 1e-3 with SGD optimiser with a weight decay parameter of 5e-4 and Cross-Entropy loss function.

F. Network Comparison

Mostly all the models were trained with the same hyperparameters for comparing the network architectures. The hyperparameters are mentioned in each of the network architecture sections. Following table summarises the performance of each network architectures.

Model	Parameters	Train Accuracy	Test Accuracy
BaselineNet	1091614	73.714%	64.37%
BatchNormNet	1092094	82.91%	79.26%
ResNet	2705802	89.96%	85.28%
ResNeXt	20540554	58.71%	55.45%
DenseNet	1763018	95.85%	89.48%

TABLE I: The comparison of all the models.

BaselineNet, shows a decent performance but has the lowest test accuracy among all models. The gap between training and test accuracy suggests some overfitting, indicating that the model struggles to capture the complexities of the CIFAR10 dataset. This was addressed partially by the BatchNormNet with the inclusion of batch normalization layers resulting in reduced internal covariate shift.

Fig. 51: ResNeXt Network Architecture

ResNet with its deeper architecture and residual connections, demonstrates a significant improvement in both training and test accuracies. Following this, ResNeXt despite its high

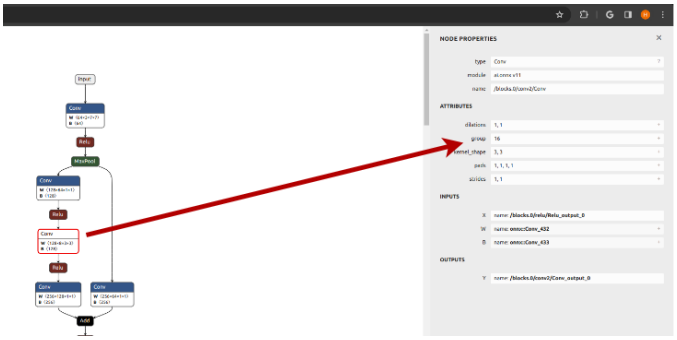

Fig. 52: ResNeXt Cardinality

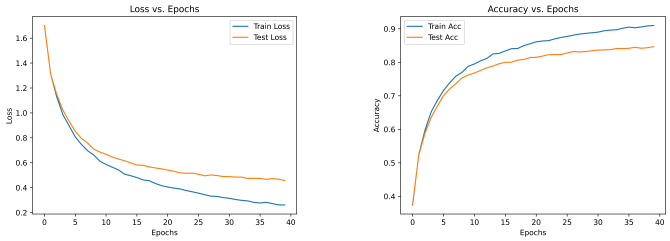

Fig. 53: DenseNet Performance

parameter count showed poor performance. I feel ResNeXt being a large model should have been trained for a couple of more epochs. Additionally, a few iterations of different hyperparameters would have resulted in a better performance. For instance experimenting with a different learning rate or using learning rate schedules could have been beneficial for optimizing. Additionally, regularization techniques, like dropout or data augmentation, might also have improved the network's ability to generalize better.

DenseNet showed the best performance among all models. Reusing features through dense connections likely contributes to its high accuracy. Additionally, smaller batch size and

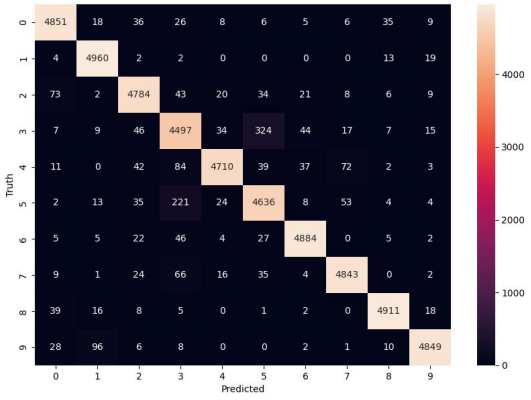

Fig. 54: DenseNet Train Confusion Matrix

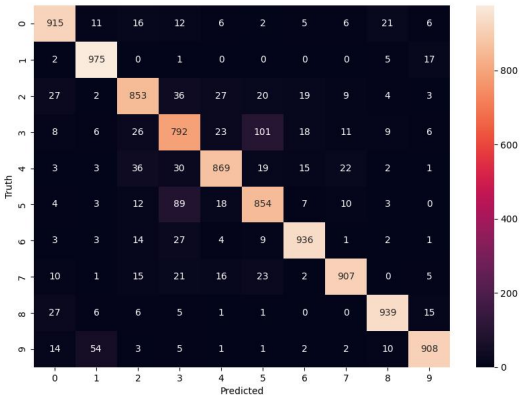

Fig. 55: DenseNet Test Confusion Matrix

the weight decay as a regularization technique could have contributed to more effective training and generalization.