RBE549 : Homework 0 - Alohomora

Mihir Deshmukh
Worcester Polytechnic Institute
Worcester, MA
Email: mpdeshmukh@wpi.edu

***Abstract*—This homework consists of two phases- 1) Phase 1: Shake my boundary, 2) Phase 2: Deep dive on deep learning. The first phase implements the Pb(probability of boundary) lite algorithm for boundary detection. It uses multiple filterbanks and Kmeans for vector quantization of features. The second phase implements a basic deep learning model as well as 3 well-known models, namely ResNet, DenseNet, and ResNeXt.**

Using one late day.

I. Phase 1: Shake my Boundry

Classical approaches for edge detection utilize intensity discontinuities to identify edges. Here, we use the simplified version of the recent Pb-lite algorithm, which uses texture, brightness, and intensity maps for boundary detection. This approach helps texture and color maps to outperform the classical baselines. Figure 1 shows the overview of the pb-lite algorithm. The pb-lite algorithm consists of these 4 basic steps:

- Filtering
- K-Means to get Texton, Brightness, and Color Maps.
- Chi-square distance with Half Disks (Gradients)
- Combine Texton, Color, and Brightness gradients with Canny and Sobel Outputs.

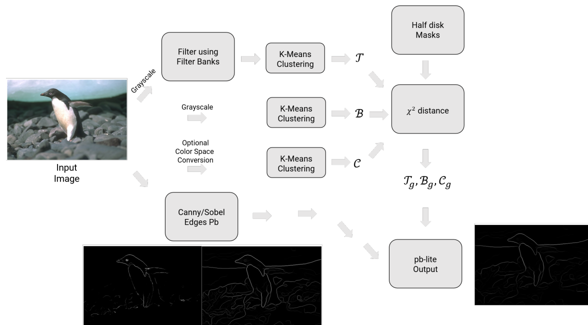

Fig. 1. Pb-Lite algorithm overview

A. Filter Bank Generation

We use the following three filter banks:

- Oriented Derivative of Gaussian (DoG)
- Leung-Malik Filters
- Gabor Filters

1) Oriented DoG: The oriented Derivative of the Gaussian filter bank is constructed by convolving the X and Y Sobel filter with a Gaussian kernel and summing the result. The filter size was set to 49*49, and the two scales(standard deviation of the Gaussian) were set to [4, 8]. I have 16 orientations of each scale. Figure 2 is the visualization of the Oriented DoG filter bank.

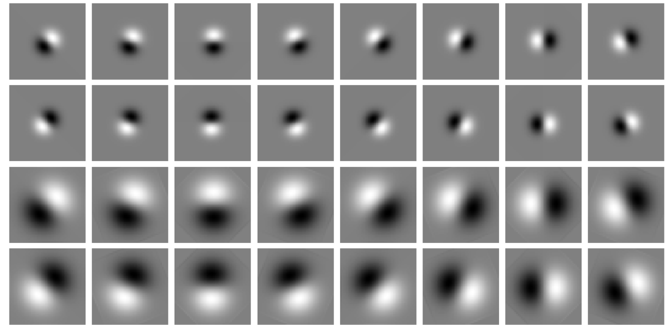

Fig. 2. Oriented DoG Filter Bank

2) Leung-Malik Filters: The LM filters consist of 48 filters. We have Gaussian, 8 Laplacian of Gaussian(LoG), and 18 first-order Gaussian derivative, and 18 second-order Gaussian derivative kernels. LM small used these $[1, \sqrt{2}, 2, 2\sqrt{2}]$ scale, while the LM large uses $[\sqrt{2}, 2, 2\sqrt{2}, 4]$. The filter size was set to 49*49. Figure 3 is the visualization of the Leung-Malik Small filter bank.

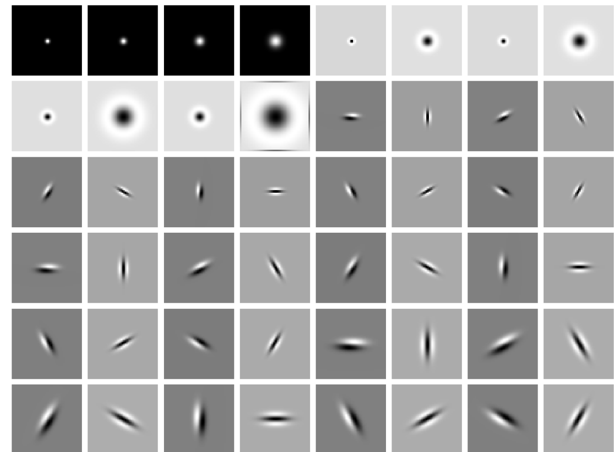

Fig. 3. Leung-Malik Small Filter Bank

3) Gabor Filters: The Gabor filter bank uses a Gaussian filter modulated with a sinusoidal plane wave with a total of 40 filters. The sigma were were set to [2, 4, 7, 9, 12]. Figure 4 is the visualization of the Gabor filter bank.

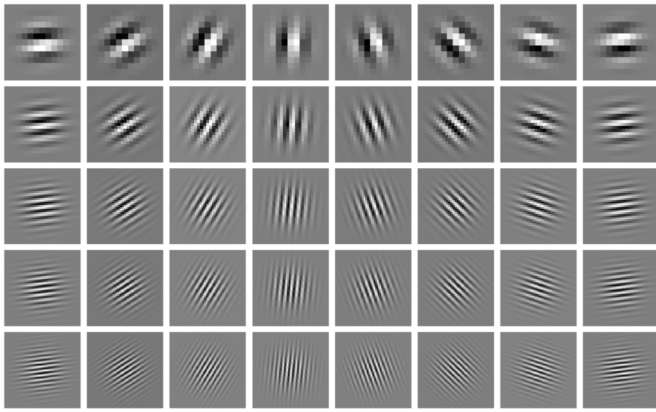

Fig. 4. Gabor Filter Bank

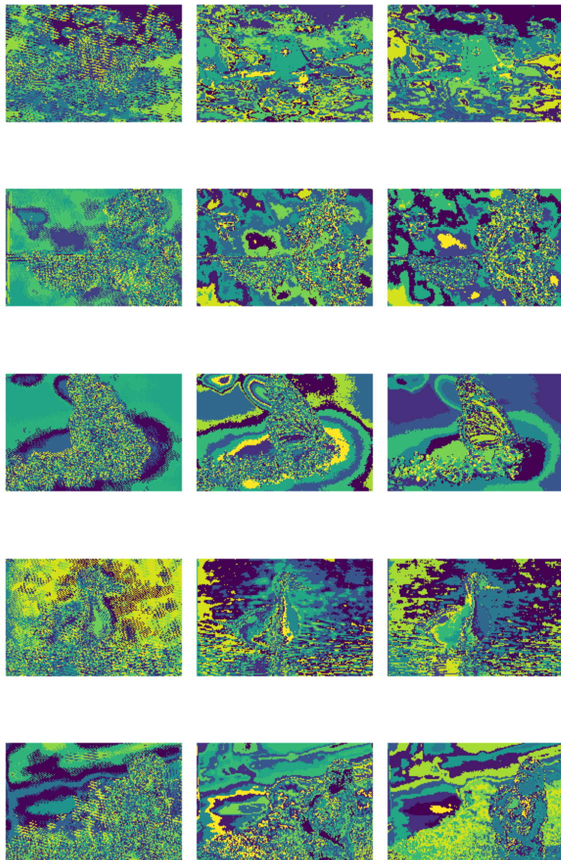

Fig. 5. Images 1-5: Texton Map, Brightness Map, Color Map

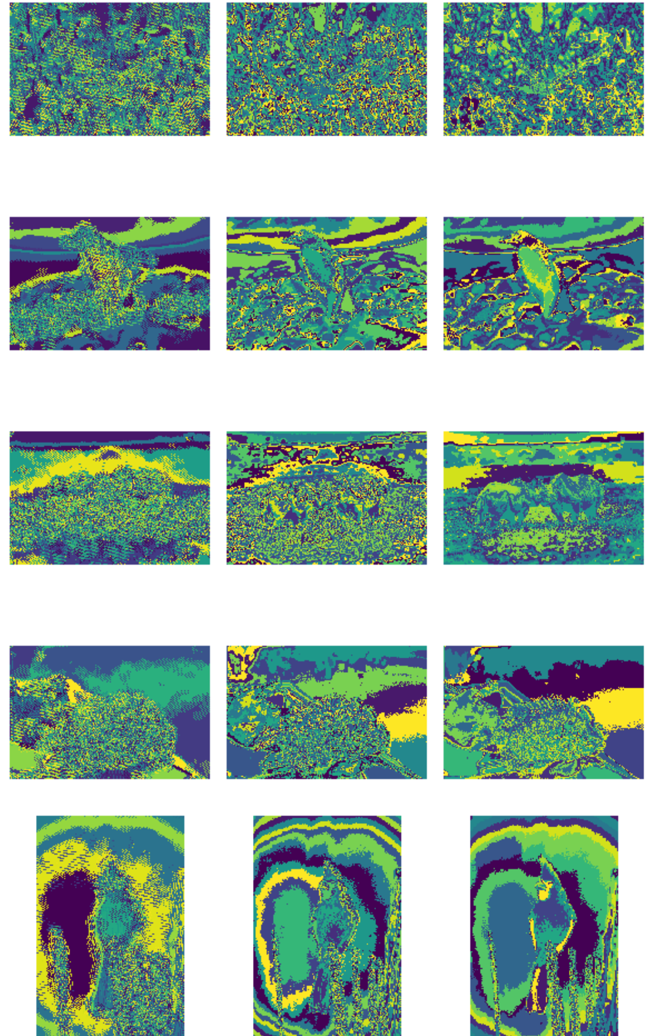

Fig. 6. Images 5-10: Texton Map, Brightness Map, Color Map

B. Texton, Brightness, Color Maps

Figure 5 and Figure 6 illustrate the texton, brightness, and color maps, respectively, for all the images in the dataset.

1) Texton Map: Now that our filter bank is ready, I convolved each image with the filter bank. I have used the Oriented DoG, LM small, and Gabor filters to get the image features. After getting the filter responses, we have an N-dimensional (N is the number of filters) feature vector for each pixel in the image. These feature vectors can be thought of as containing information pertaining to the image textures. Using K-Means clustering to aggregate these features, we assign them to one of the 64 bins. This cluster bin number can be viewed as the Texton ID for each pixel. F

2) Brightness Map: The brightness map is used to capture the brightness fluctuations in the image. Here, we first convert the RGB image to Grayscale and apply K-Means to cluster the different brightness levels. For brightness maps, we use K=16.

3) Color Map: Now, to get the colormap, we cluster the RGB values of the input image and segregate them in 16 bins using K-Means.

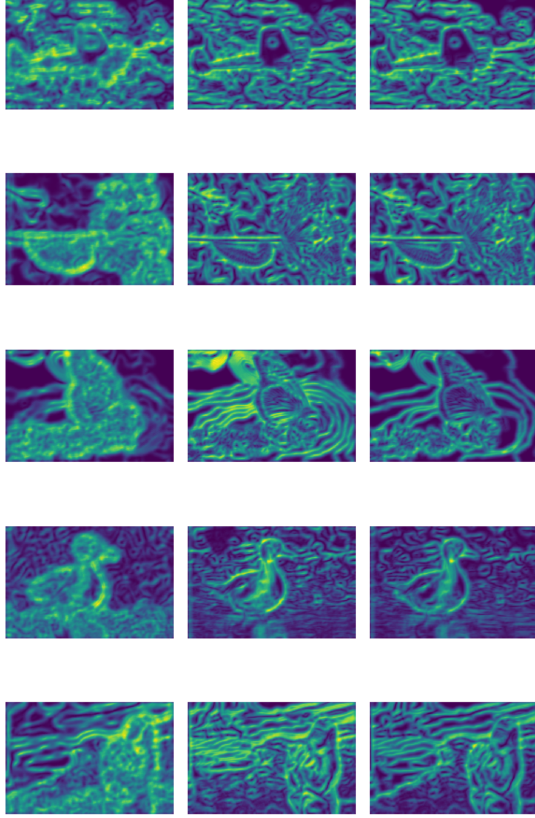

Fig. 7. Images 1-5: Texton Gradients, Brightness Gradients, Color Gradients

C. Texton, Brightness, color Gradients

1) Half-Step disc masks: We generate half-step disc masks to calculate the gradients of the texture, brightness, and color map in an efficient manner. These masks are semi-circular binary masks in 16 orientations and [5, 10, 15] scales. Figure 8 depicts the half-step disc masks.

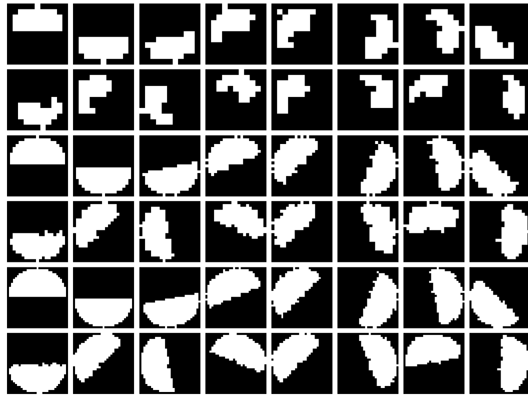

Fig. 8. Half Step Disk Masks

The gradients depict how the corresponding map distribution changes at a pixel. We calculate this by using a left-right pair of half-disk masks of the same scale to filter the map. We then use the chi-square distance to compute the gradients. Figure 7 and Figure 9 illustrate the gradients for the texton, brightness, and color maps, respectively.

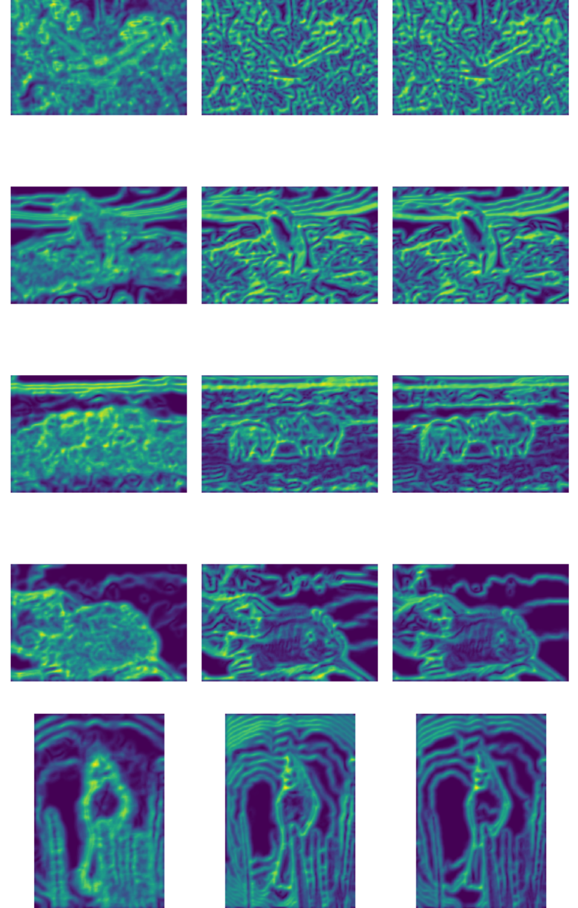

Fig. 9. Images 5-10: Texton Gradients, Brightness Gradients, Color Gradients

D. Pb-lite output

The Pb lite output is calculated using the below mentioned formula. I used w1=0.3 and w2=0.7 to get the PB-lite output.

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

The values of w1 and w2 can be changed further to fine-tune the result of the pb-lite algorithm. Figure 10 and 11 showcase the comparison of outputs for the Canny, Sobel, and Pb-lite outputs.

E. Result Comparison & Discussion

We can see from the comparison the Sobel output is the most suppressed, while Canny, on the other hand, has a lot of noise. The Pb-lite output is the most balanced among the three and is a clearer and more stable output on the boundaries.

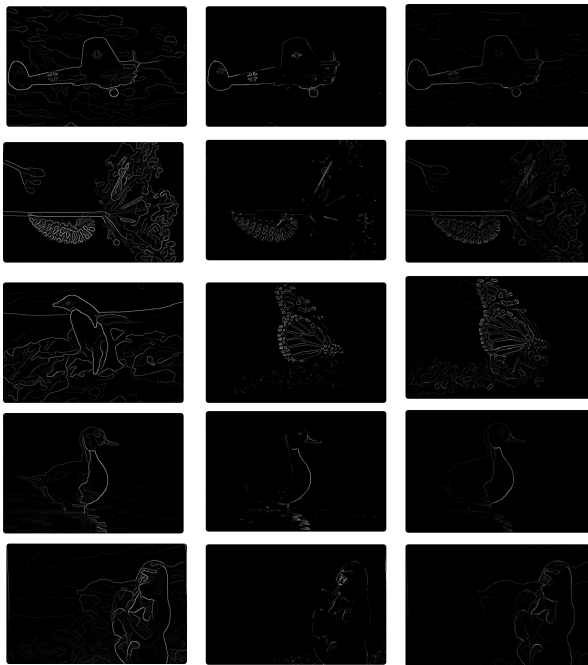

Fig. 10. Images 1-5: Canny, Sobel, Pb-lite

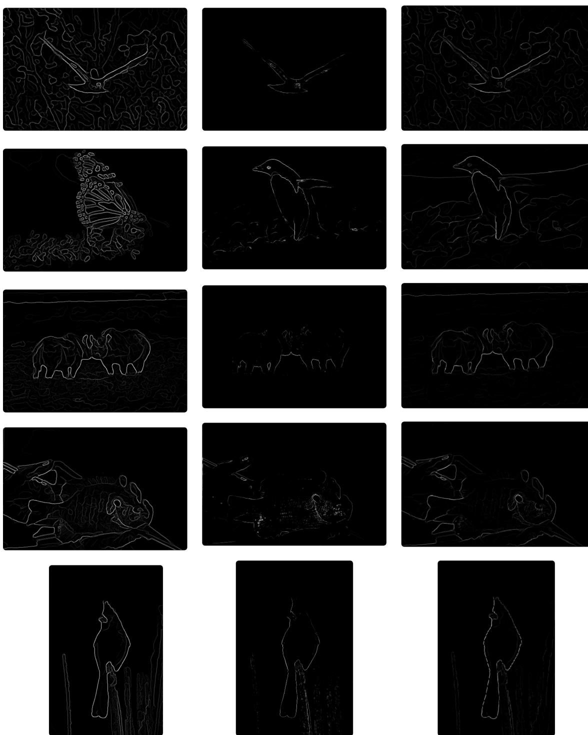

Fig. 11. Images 5-10: Canny, Sobel, Pb-lite

It is a good mix between the Canny, which has a lot of false positives, and Sobel, which has only a few details. Overall, we can further improve the performance of the Pb-lite by adjusting the weights w1 & w2 and the Filter Bank as well, depending on the input images.

II. Phase 2: Deep Dive on Deep Learning

In this phase of the homework, we train and test various Neural network architectures for a classification task. Here, we have used the CIFAR-10 dataset, which consists of 60000, 32×32 RGB images encompassing 10 classes. There are 50000 training images and 10000 test images. In the following sections, I go over the various models and methodologies I used for the classification task.

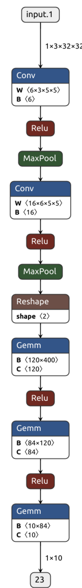

Fig. 12. Simple CNN Model

A. *Simple Convolution neural network:*

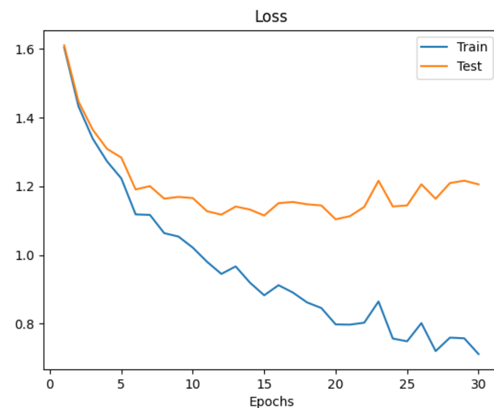

Fig. 13. SimpleCNN: Loss over epochs

First, I used a simple convolutional neural network with 2 convolutional layers with ReLU activation functions for the given task. The model architecture can be seen in Figure 12. The model has a total of 62,006 parameters. Below are the training hyperparameters:
Optimizer: Adam

Learning Rate: 0.001
Batch Size: 64
Epochs: 30

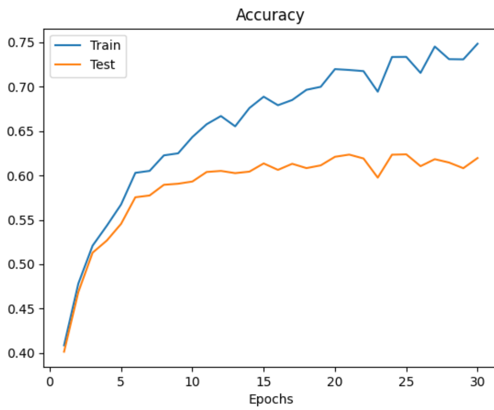

Fig. 14. SimpleCNN: Accuracy over epochs

The input images are scaled between 0 and 1 by dividing each pixel value by 255. This helped improve the model's performance. The training accuracy was 74.8%, and the test accuracy was 61.9%. The confusion matrix for training and testing can be seen in Figure 15 and 16.

```
[4111   83  249   40   34   14   27   22  300  120] (0)
[ 128 4287   28   12   10    4   32    5  197  297] (1)
[ 252   45 3632  186  294  129  208   93   84   77] (2)
[ 118   61  486 2763  219  492  414  172  144  131] (3)
[ 174   30  426  215 3462   71  243  253   54   72] (4)
[  63   44  393  930  245 2724  175  275   53   98] (5)
[  38   83  257  173  147   57 4103   17   71   54] (6)
[  96   30  227  158  299  144   30 3844   43  129] (7)
[ 268   83   54   23   11    4   14    5 4462   76] (8)
[ 195  445   43   35   16    7   25   16  177 4041] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 74.858 %
```

Fig. 15. SimpleCNN: Train Confusion Matrix

```
[713  28  75  15  19   6  10  10  83  41] (0)
[ 38 758  11  10   3   0  13   1  68  98] (1)
[ 74  16 548  61  99  51  68  29  34  20] (2)
[ 31  23 110 402  71 111 108  57  39  48] (3)
[ 44  15 101  59 560  25  86  74  19  17] (4)
[ 22   4 104 221  62 423  46  77  13  28] (5)
[ 13  17  73  50  65  16 715  15  17  19] (6)
[ 39   7  65  48  80  55  16 635   6  49] (7)
[ 95  41  25  15   7   2   7   5 777  26] (8)
[ 68 129  16  21   3   7  12  11  68 665] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 61.96 %
```

Fig. 16. SimpleCNN: Test Confusion Matrix

From the train and test loss & accuracies in 13 and 14, we can see the model starts overfitting after 10 epochs as the test loss stagnates while the training loss is decreasing. To help alleviate this, we modify the network architecture to try to improve the model performance.

B. *Modified CNN*

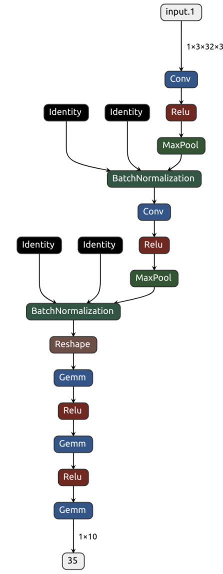

Fig. 17. Modified CNN Model

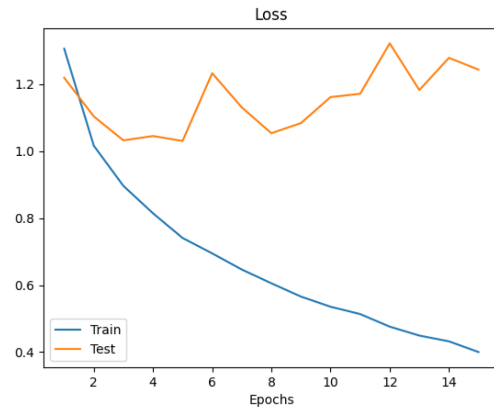

Fig. 18. ModifiedCNN: Loss over epochs

To try and improve the model performance I added batch normalization layers between the convolution layers as well as added weight decay for regularization so that the model can generalize better. Figure 17 shows the modified architecture. The model has 62,050 parameters. Below are the training hyperparameters:
Optimizer: Adam
Learning Rate: 0.001
Batch Size: 64
Epochs: 30
Weight Decay: 1e-4

The batch normalization layers help prevent internal covariate shifts in deep networks, thus improving performance. This helped the model perform a little better and converge faster.

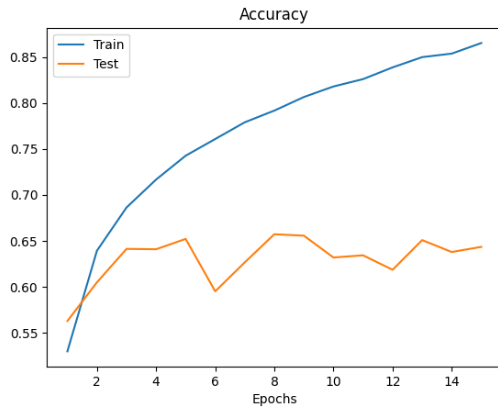

Fig. 19. ModifiedCNN: Accuracy over epochs

However, as this network is shallow, only a small benefit was observed from the introduction of batch normalization. It still began to over around 5 epochs as seen in the loss and accuracy graphs Figure 18 and Figure 19.

There can be two solutions to solving this issue. One would be to augment training data so as to help the model generalize better and not overfit. Another would be to go for a more complex model, which can model the training data better.

The training accuracy was 84.7%, and the test accuracy was 64.4%. The confusion matrix for training and testing can be seen in Figure 20 and 21.

```
[4233   62  125   43   32   12   24   21  370   78] (0)
[  41 4654   16   17    6    7   17    7   75  160] (1)
[ 221   10 3993  162  200   99  144   79   61   31] (2)
[  62   15  194 3690  143  467  219   69   91   50] (3)
[ 143   31  234  175 3905  110  165  142   51   44] (4)
[  42   24  171  556  124 3780  100  136   40   27] (5)
[  19   20  152  151   56   60 4463   12   46   21] (6)
[  46   14   71  111  124  120   22 4429    8   55] (7)
[  94   59   22   29   11    8   14    6 4671   86] (8)
[  77  176   25   42   10   12   20   14   86 4538] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 84.712 %
```

Fig. 20. ModifiedCNN: Train Confusion Matrix

```
[678  33  52  20  20  11  13  12 111  50] (0)
[ 28 772  12   9   6   5  15   4  33 116] (1)
[ 82   4 534  78  85  53  69  50  26  19] (2)
[ 28  17  84 463  56 169  74  48  31  30] (3)
[ 48  11  81  82 553  45  76  73  25   6] (4)
[ 18   9  59 203  53 515  40  71  20  12] (5)
[ 16   4  42  72  35  33 761   8  17  12] (6)
[ 28  11  54  54  66  69  12 662   5  39] (7)
[ 50  49  16  20   6   6   6   5 793  49] (8)
[ 41 114  11  21  11  12  16  24  41 709] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 64.4 %
```

Fig. 21. ModifiedCNN: Test Confusion Matrix

C. ResNet

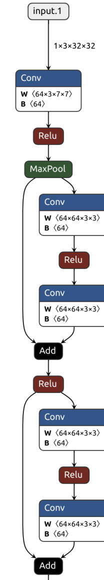

Fig. 22. View of a Residual blocks in ResNet18

The motivation behind ResNet is that when we train deeper CNNs, the accuracy seems to decrease, which should not be the case. To mitigate this, ResNet introduces skip connections, which help provide identity maps so that the model performance doesn't degrade even if we go deeper even if it doesn't improve. Figure 22 depicts the structure of a residual block of ResNet18.

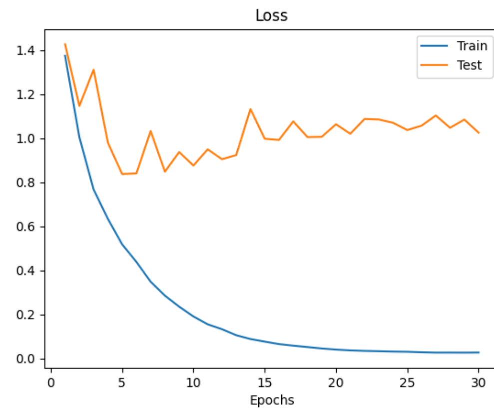

Fig. 23. ResNet18: Loss over epochs

Following this idea, I implemented the ResNet18 network according to the paper. It consists of 4 residual layers, which total 18 convolutional layers. The model has a total of 11,186,442 parameters. Below are the training hyperparameters:
Optimizer: SGD
Learning Rate: 0.01
Batch Size: 64

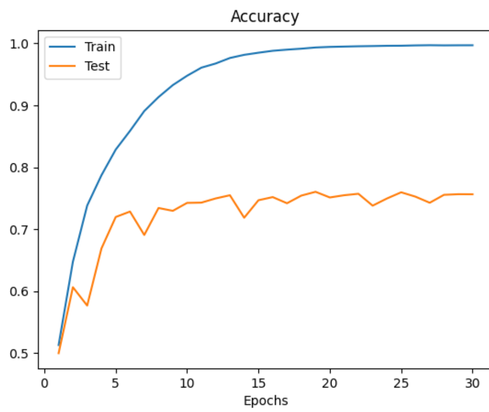

Fig. 24. ResNet18: Accuracy over epochs

Epochs: 30
Weight Decay: 5e-4
Momentum: 0.9

```
[4921   11   10    4   19    2    1    6   12   14] (0)
[   6 4887    2    5    1    4   11    0    8   76] (1)
[  48    0 4704   64  107   37   14   16    8    2] (2)
[   6    2   17 4815   52   63   28    6    7    4] (3)
[   2    0   11   17 4906   17   25   15    4    3] (4)
[   8    1   29   92   55 4766    9   32    5    3] (5)
[   2    0   16   45   21   18 4895    1    1    1] (6)
[   0    1    6   41   30   24    4 4892    1    1] (7)
[  41    9    5    1    3    1    4    0 4916   20] (8)
[   7   22    4   12    1    5    3    7    4 4935] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 97.274 %
```

Fig. 25. ResNet: Train Confusion Matrix

```
[800  11  30  24  21   6  13  15  44  36] (0)
[ 21 816   7  12   3   6  11   3  17 104] (1)
[ 52   3 617  78 119  51  42  21  12   5] (2)
[ 28   6  30 597  79 160  49  21  17  13] (3)
[  9   2  45  48 788  30  32  38   5   3] (4)
[ 12   2  38 187  59 622  24  39   9   8] (5)
[  7   1  34  56  33  28 828   4   4   5] (6)
[ 16   4  19  38  59  36   7 814   1   6] (7)
[ 62  30  13   8   8   7   6   6 832  28] (8)
[ 34  53   7  11   5   6   5  11  15 853] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 75.67 %
```

Fig. 26. ResNet: Test Confusion Matrix

The training accuracy was 97.27%, and the test accuracy was 75.67%. The confusion matrix for training and testing can be seen in Figure 25 and 26.

Though ResNet18 performs better than the basic CNN, it still starts overfitting after about 6 epochs as seen clearly from the loss and accuracy plots. One solution, as explained earlier, could be augmenting the dataset, which will help the model generalize better. We need to tune the hyperparameters as well as think about implementing even deeper ResNets like ResNet50 or ResNet101.

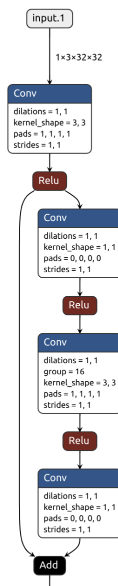

Fig. 27. View of a ResNeXt block

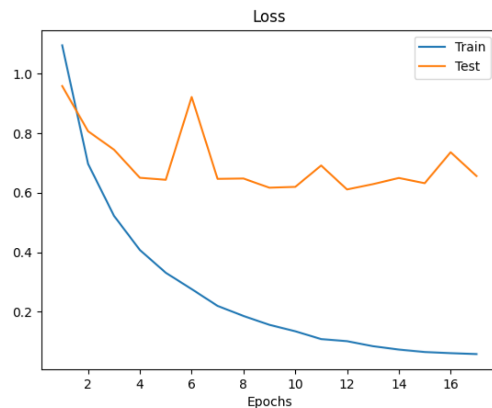

Fig. 28. ResNeXt: Loss over epochs

D. ResNeXt

ResNext uses aggregation of grouped convolutions allowing more parallel paths in the network. This is controlled by the cardinality parameters, and here I have used a cardinality of 32. Following this idea, I implemented the ResNeXt network according to the paper. It consists of 4 resNeXt layers. The model has a total of 1,803,658 parameters. Below are the training hyperparameters:
Optimizer: Adam
Learning Rate: 0.01
Batch Size: 64
Epochs: 16
Weight Decay: 1e-4

The training accuracy was 95.5%, and the test accuracy was 81.5%. The confusion matrix for training and testing can be seen in Figure 30 and 31.

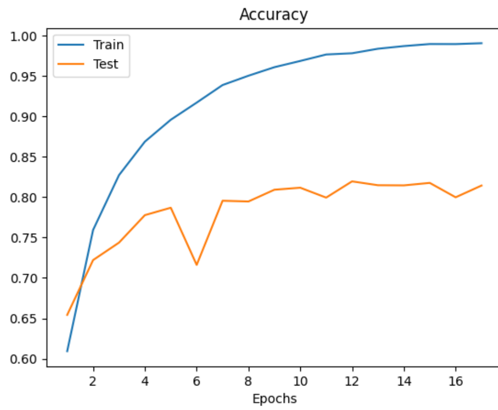

Fig. 29. ResNeXt: Accuracy over epochs

```
[4910    8   24    6    5    2    2    1   37    5] (0)
[  10 4946    2    1    1    0    7    2    8   23] (1)
[ 112    1 4779   28   30   17   16    6   10    1] (2)
[  41    3   70 4733   37   68   16   14   16    2] (3)
[  29    0   77   65 4771   15   18   19    4    2] (4)
[  24    0   84  264   57 4454   13   98    6    0] (5)
[  22    8  104  105   24   12 4701    9   15    0] (6)
[  22    2   59   62   53   17    4 4775    5    1] (7)
[  66   17   13    3    2    0    1    0 4880   18] (8)
[  63   47    8   19    0    3    8    8   16 4828] (9)
  (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 95.554 %
```

Fig. 30. ResNeXt: Train Confusion Matrix

```
[893   9  30  12   5   2   2   5  32  10] (0)
[ 14 929   1   4   2   0   4   2  15  29] (1)
[ 78   3 776  34  47  17  26   8   8   3] (2)
[ 32   7  61 698  40  88  30  23  19   2] (3)
[ 24   1  66  48 802  19  18  18   4   0] (4)
[ 10   1  58 144  40 688  12  40   5   2] (5)
[ 13   3  59  80  22  15 793   4  11   0] (6)
[ 22   1  43  36  39  23   3 822   6   5] (7)
[ 60  13  10   5   1   2   3   1 882  23] (8)
[ 34  50   7  12   0   3   3   8  19 864] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 81.47 %
```

Fig. 31. ResNeXt: Test Confusion Matrix

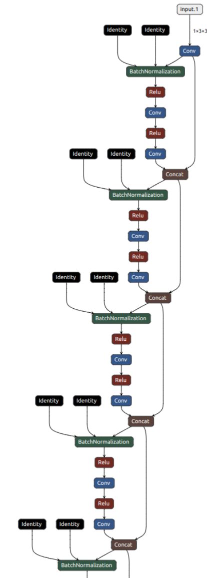

Fig. 32. View of a Dense block in DenseNet

ResNeXt performs better than ResNet, but we can still see some overfitting after about 8 epochs, so there is a scope for improvement by adding more layers as well as adding data augmentation.

E. DenseNet

DenseNet uses a unique approach for dense connectivity between layers where each layer receives input from all preceding layers. This improves information flow and reduces the vanishing gradient problem. It is more efficient in terms of parameters than ResNet and focuses on feature map fusion with its dense connectivity. Following this idea, I implemented the 40-depth DenseNet network according to the paper. It consists of 3 dense layers and 2 transition layers. The model has a total of 5,564,248 parameters. Below are the training hyperparameters:
Optimizer: SGD
Learning Rate: 0.01
Batch Size: 64
Epochs: 18
Weight Decay: 5e-4
Momentum: 0.9

The training accuracy was 98.8%, and the test accuracy was 87.7%. The confusion matrix for training and testing can be seen in Figure 35 and 36.

DenseNet performs the best out of all three models, but we can still see some overfitting after about 10 epochs, so there is a scope for improvement by adding data augmentation to improve performance.

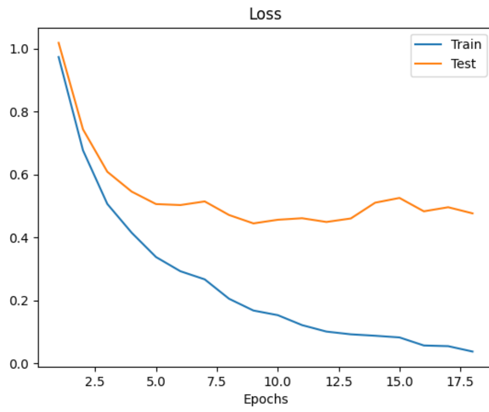

Fig. 33. DenseNet: Loss over epochs

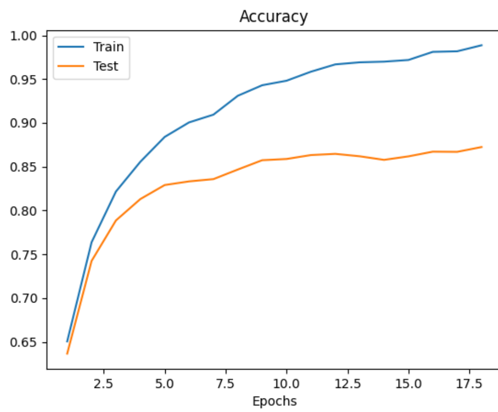

Fig. 34. DenseNet: Accuracy over epochs

```
[4928    2   33    2    6    0    0    4   21    4] (0)
[   0 4985    6    2    0    0    1    2    1    3] (1)
[   4    0 4965   15    6    4    0    5    1    0] (2)
[   3    0    6 4953    8   16    2    7    2    3] (3)
[   4    0   27    2 4953    1    0    8    5    0] (4)
[   0    1   35   84    5 4843    0   30    2    0] (5)
[   0    0   45   42   10    4 4891    5    3    0] (6)
[   2    0    5    4    3    1    0 4982    2    1] (7)
[   3    0    1    0    0    0    0    0 4995    1] (8)
[   6   27    7    4    4    0    2    4   11 4935] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 98.86 %
```

Fig. 35. DenseNet: Train Confusion Matrix

```
[870  10  41  12  12   1   4   7  34   9] (0)
[  6 946   2   6   1   0   3   1  11  24] (1)
[ 16   0 856  43  28  15  16  21   4   1] (2)
[  9   1  36 789  41  74  12  26   7   5] (3)
[  4   1  59  20 875  12   7  20   2   0] (4)
[  3   1  41 105  22 784   4  37   2   1] (5)
[  8   0  46  59  26   9 841   9   2   0] (6)
[  4   1  14  16  22  12   0 925   3   3] (7)
[ 22   7   6   7   0   0   2   0 948   8] (8)
[ 15  50   4   9   0   1   1   6  21 893] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 87.27 %
```

Fig. 36. Dense: Test Confusion Matrix