Homework 0: Alohamora
RBE 549
(Using 2 late days)

Karter Krueger
Robotics Engineering Department
Worcester Polytechnic Institute
Worcester, MA 01609
Email: kkrueger2@wpi.edu

I. Phase 1

Phase 1 of the homework is focused on the implementation of a simplified version of the PB (probability of boundary) edge detector outlined in [1]. Detecting edges and boundaries of objects is a very common problem in computer vision, as you often must look for objects and features in the image to be used for applications such as counting or tracking. Our boundary detector goes through several steps to extract features from the image to be combined for a final edge detection output.

A. Step 1: Filter Banks

The first step of detecting the boundaries requires the extraction of low-level features and textures using a bank of filters. Features are found by performing a 2D convolution across the image with the filter as the kernel. Three filter generation methods are used with a variety of parameters to maximize the variety of discovered features.

1) Oriented DoG (Derivative of Gaussian) Filters: Derivative of Gaussian filters are generated by convolving a Sobel kernel operator across a Gaussian distribution matrix. First, a Gaussian matrix G is generated by filling each (x, y) cell by the formula

$$G(x,y) = \frac{1}{\pi * \sigma^2} * e^{\frac{-x^2+y^2}{2*\sigma^2}}$$

The Sobel kernel is defined by: [[1, 0, -1], [2, 0, -2], [1, 0, -1]]. After convolving the Sobel kernel across a Gaussian matrix of the target size, the resulting filter can be rotated to a target angle to achieve a variety of oriented DoG filters. Filters from 16 orientations $[0, 2*\pi)$ and 2 scales are shown in 1 below.

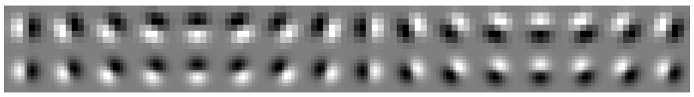


Fig. 1. Oriented DoG Filters with 16 orientations and 2 scales.

2) Leung-Malik Filters: Leung-Malik (LM) filters are made up of both angled and circular Gaussian filters. The standard 48 filters include 1st and 2nd order Gaussian derivatives at 6 angles and 3 scales, 8 Laplacian of Gaussian (LoG), and 4 circular Gaussian filters, shown in Fig. 2 below. LoG filters are generated by convolving the Laplacian kernel [2] across a Gaussian matrix, with the Laplacian kernel defined as: [[0, -1, 0], [-1, 4, -1], [0, -1, 0]]. Filters are then rotated to a target rotation angle.

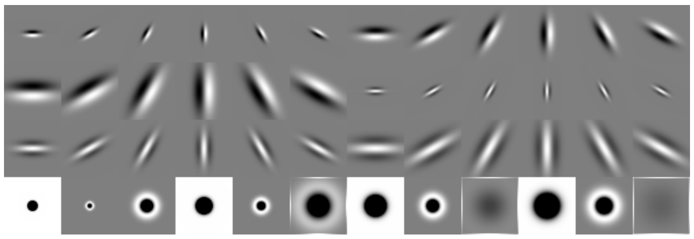


Fig. 2. Leung-Malik Filter Bank.

3) Gabor Filters: Gabor filters are generated to be similar to how humans see as they look for specific image frequencies, explained further in [3]. As defined by [3], the Gabor extracts image features using:

$$G_{cos}[i,j] = Be^{-\frac{i^2+j^2}{2\sigma^2}} cos(2\pi f(icos\theta + jsin\theta))$$

Gabor filters at 8 orientations (θ) $[0, 2\pi)$ and 5 scales are shown in 3 below. The scales were altered by increasing the value of σ while decreasing the number of standard deviations. Parameters γ, λ, and ψ were held constant.

B. Step 2: Textons, Brightness, and Color Maps

After generating filter banks, they must be applied to the images to be useful for detecting features and boundaries. The filters are used to create three types of maps, based on Textons, Brightness, and Color as explained below.

1) Texton Map: Texton maps are created by convolving N filters across the target image to generate an N "channels" for the resulting array. Each pixel can then be viewed as a vector with N values. Next, KMeans clustering is performed on the

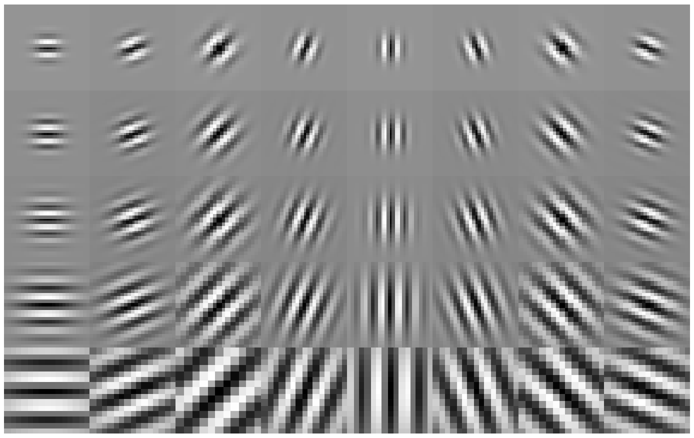


Fig. 3. Gabor Filter Bank.

pixel vectors to cluster into $K = 64$ Texton ID values. An example of a Texton ID map after clustering is shown in Fig. 4.

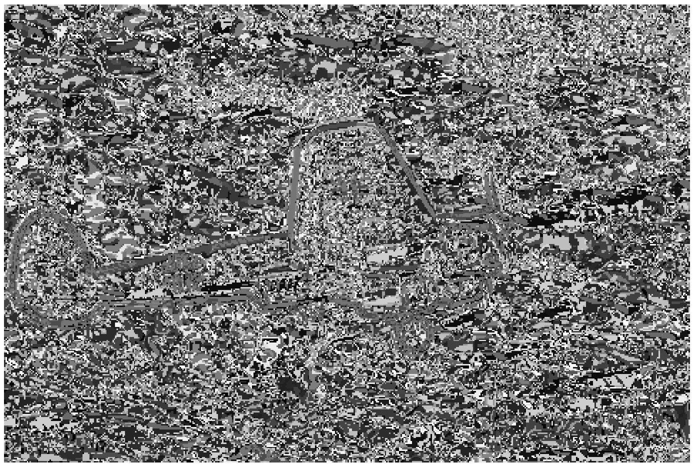


Fig. 4. KMeans Clustering of a Filtered Image1 to produce a Texton Map.

2) Brightness Map: Brightness maps are created by KMeans clustering on the greyscale (brightness) image to cluster pixel brightness values into $K = 16$ levels of intensity. An example of the brightness clustered image is shown in Fig. 5.

3) Color Map: Color maps are created by KMeans clustering on the color (RGB) image to cluster pixel colors into $K = 16$ levels of intensity. An example of the brightness clustered image is shown in Fig. 6.

C. Step 3: Map Gradients

Next, gradients are computed from the maps by the differences in values of different sizes and shapes. First, Half-disc masks are generated, to be used for the difference algorithm, by creating a white half-circle on a black background at a target angle. Opposite angle pairs are generated to get left and right opposite masks for finding difference in images. Examples of the half-disk masks are shown below in Fig. 7 at 8 orientations and 3 scales. Now, Texton, Brightness, and Color gradients are computed at the pixel-level using the above half-disk masks as part of the following equation, where g_i is the result of the image convolved with the left disk mask and h_i from the right matching mask.

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

This generates a 3D matrix with shape $m \times n \times N$ from (m, n) image and N filters.

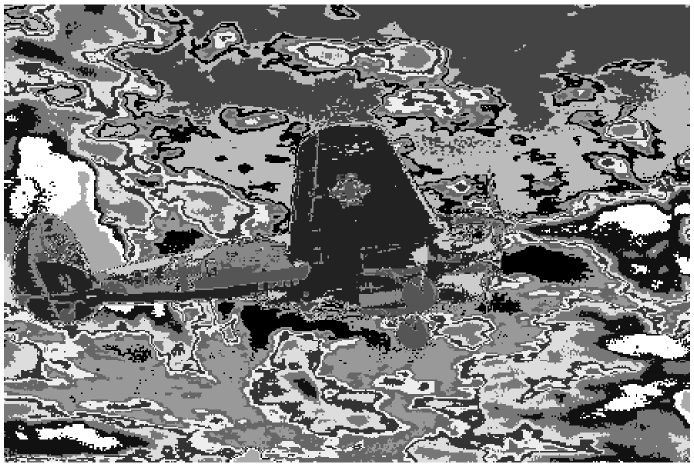


Fig. 5. KMeans Clustering of Brightness (greyscale) Image1.

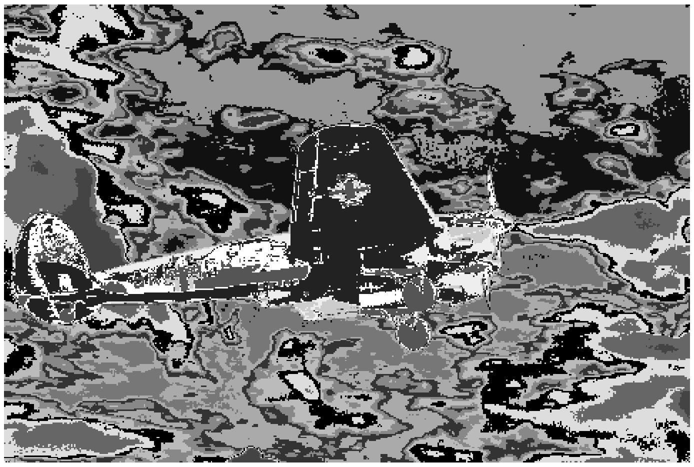


Fig. 6. KMeans Clustering of RGB Color Image1.

D. Step 4: Sobel and Canny Edge Detection

Sobel and Canny are both well-known edge detectors that are often used as a baseline. They are still important as an input to this PB-lite boundary detector. Examples of the Sobel and Canny edges are shown in Fig. 8.

E. Step 5: Pb-Lite Final Output

The fine step of the Pb-lite boundary detector combines the map gradients with the Sobel and Canny outputs using the

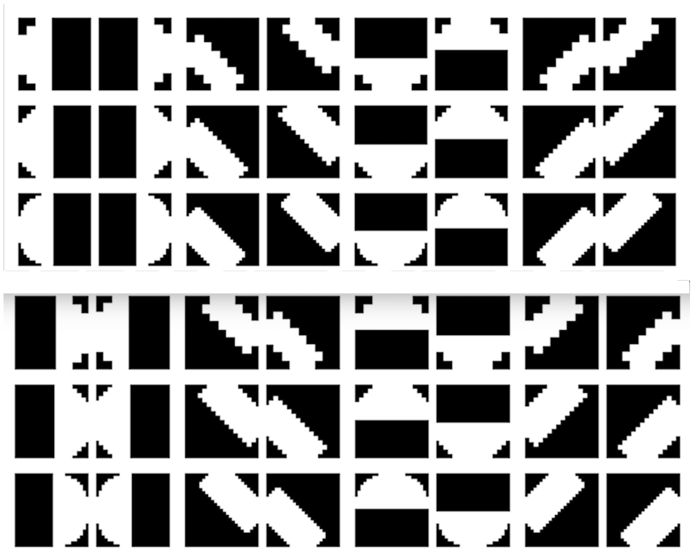


Fig. 7. Half-disk Mask Pairs.

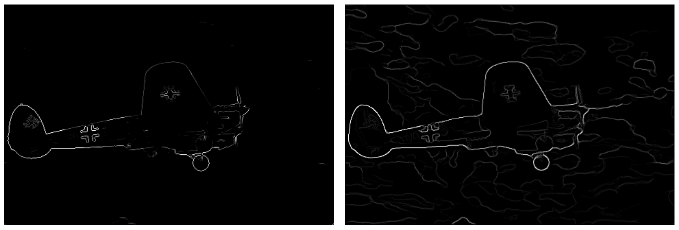


Fig. 8. Sobel and Canny from Image 1

following equation:

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

Boundaries appear brighter and more solid if they are stronger with a higher magnitude output of the equation. Values of $w_1 = .02$ and $w_2 = .02$ are used in the following example output images.

Overall this Pb-Lite boundary detector performs fairly well at detecting the object boundaries and scene edges. It is seen that background edges are shown more clearly than the Canny and Sobel baselines.

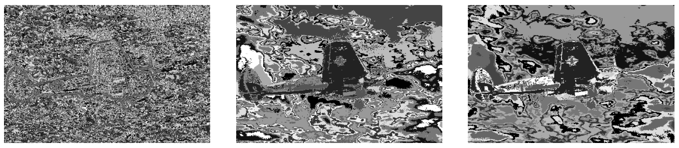


Fig. 9. Image 1 Texton, Brightness, Color

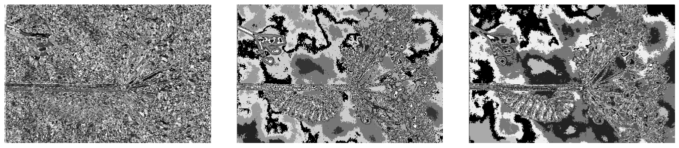


Fig. 10. Image 2 Texton, Brightness, Color

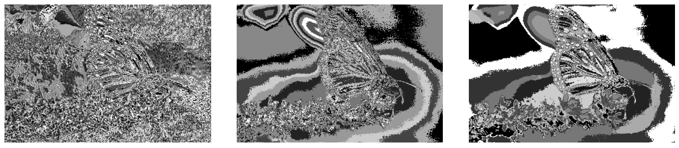


Fig. 11. Image 3 Texton, Brightness, Color

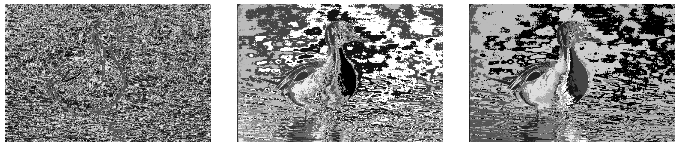


Fig. 12. Image 4 Texton, Brightness, Color

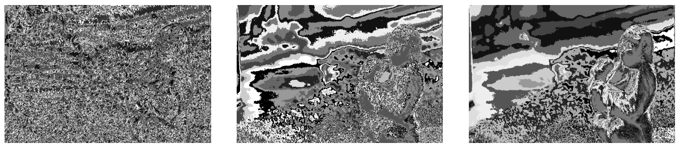


Fig. 13. Image 5 Texton, Brightness, Color

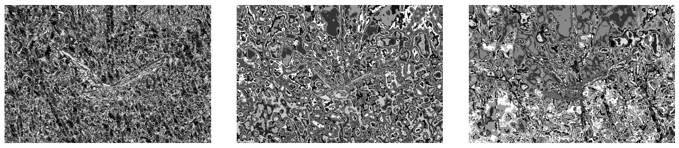


Fig. 14. Image 6 Texton, Brightness, Color

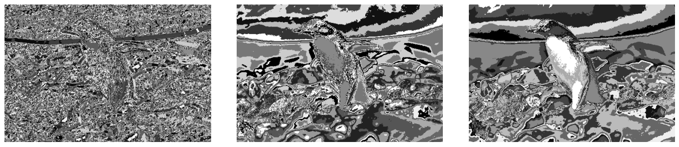


Fig. 15. Image 7 Texton, Brightness, Color

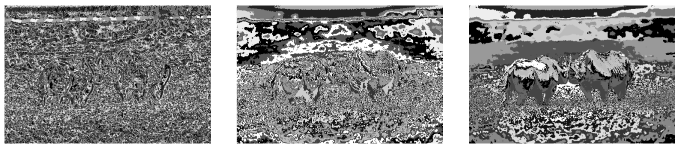


Fig. 16. Image 8 Texton, Brightness, Color

II. Phase 2: Deep Learning Cifar10

Phase 2 of the homework implemented four different neural networks that were trained on the CIFAR-10 classification dataset of 50,000 images across 10 classes. All networks are implemented and trained in PyTorch. Metrics are reported and compared across networks with confusion matrices along with plots of training loss and accuracy. Data was standardized across all network training from the $[0, 255]$ range down to $[0, 1]$. A network architecture comparison is shown in 39 of Simple network and Resnet.

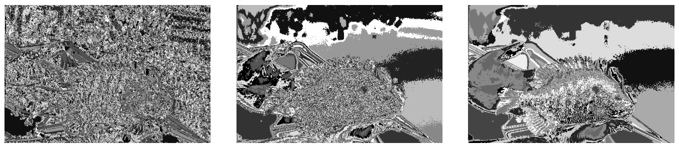


Fig. 17. Image 9 Texton, Brightness, Color

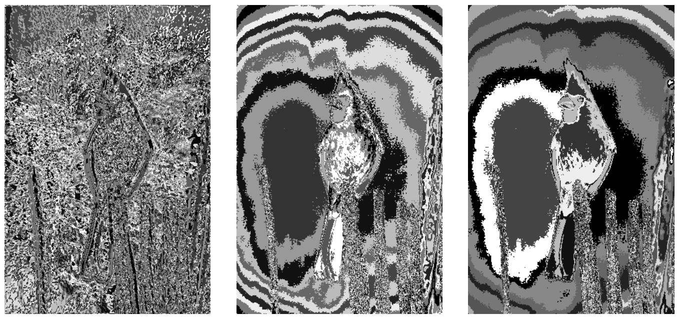


Fig. 18. Image 10 Texton, Brightness, Color

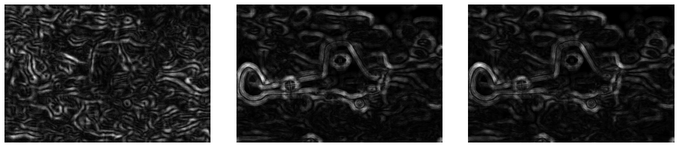


Fig. 19. Image 1 Texton, Brightness, and Color Gradients

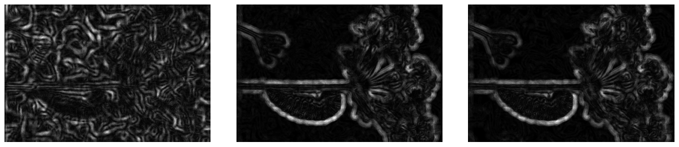


Fig. 20. Image 2 Texton, Brightness, and Color Gradients

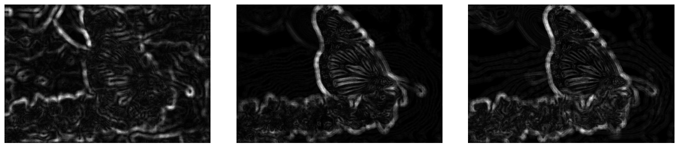


Fig. 21. Image 3 Texton, Brightness, and Color Gradients

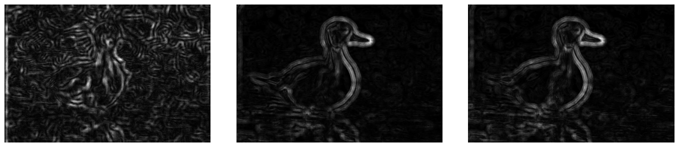


Fig. 22. Image 4 Texton, Brightness, and Color Gradients

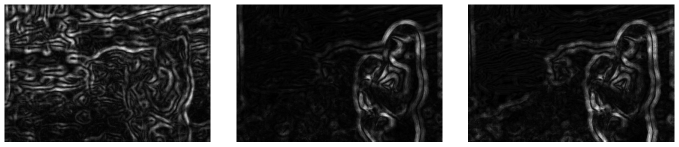


Fig. 23. Image 5 Texton, Brightness, and Color Gradients

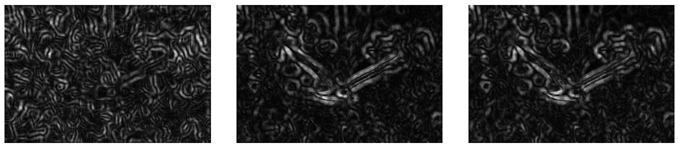


Fig. 24. Image 6 Texton, Brightness, and Color Gradients

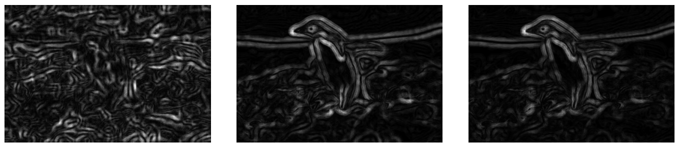


Fig. 25. Image 7 Texton, Brightness, and Color Gradients

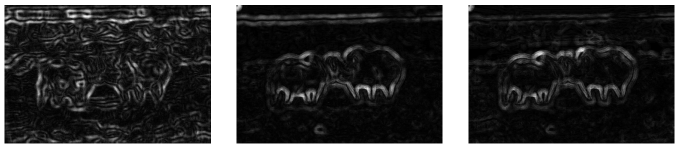


Fig. 26. Image 8 Texton, Brightness, and Color Gradients

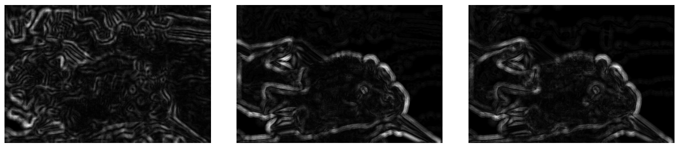


Fig. 27. Image 9 Texton, Brightness, and Color Gradients

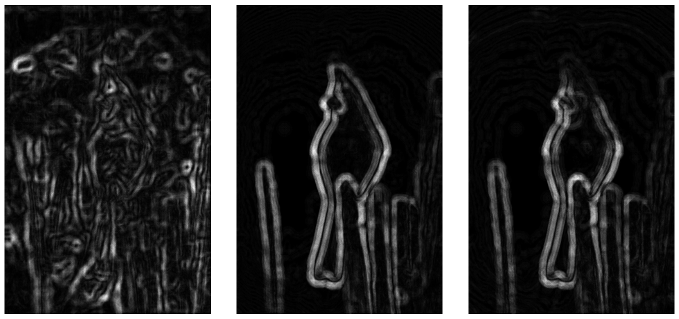


Fig. 28. Image 10 Texton, Brightness, and Color Gradients

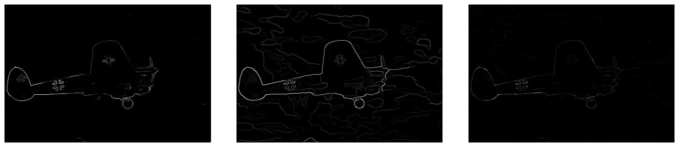


Fig. 29. Image 1 Sobel Baseline, Canny Baseline, (Ours) Pb-Lite Output

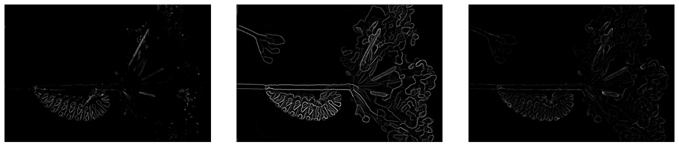


Fig. 30. Image 2 Sobel Baseline, Canny Baseline, (Ours) Pb-Lite Output

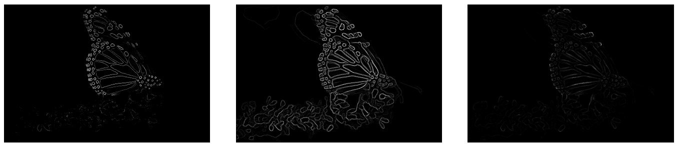


Fig. 31. Image 3 Sobel Baseline, Canny Baseline, (Ours) Pb-Lite Output

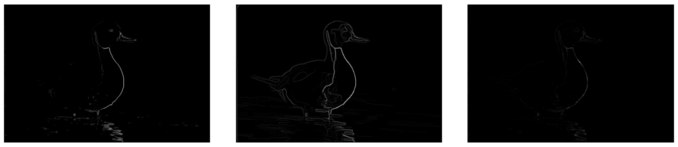


Fig. 32. Image 4 Sobel Baseline, Canny Baseline, (Ours) Pb-Lite Output

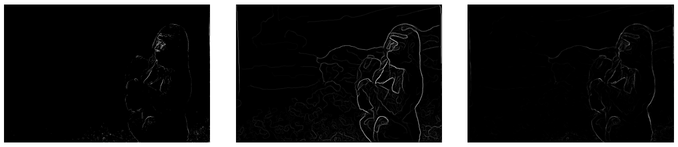


Fig. 33. Image 5 Sobel Baseline, Canny Baseline, (Ours) Pb-Lite Output

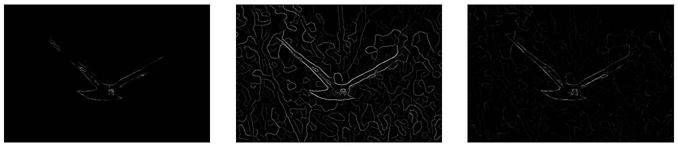


Fig. 34. Image 6 Sobel Baseline, Canny Baseline, (Ours) Pb-Lite Output

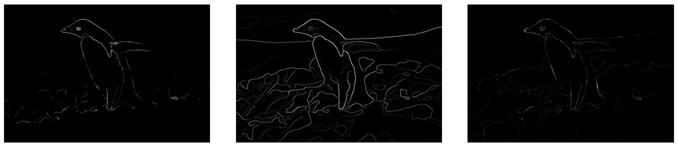


Fig. 35. Image 7 Sobel Baseline, Canny Baseline, (Ours) Pb-Lite Output

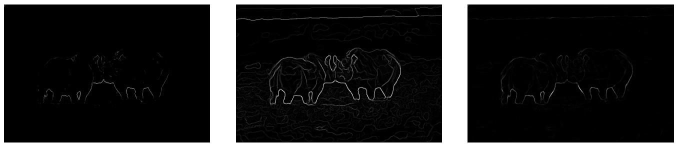


Fig. 36. Image 8 Sobel Baseline, Canny Baseline, (Ours) Pb-Lite Output

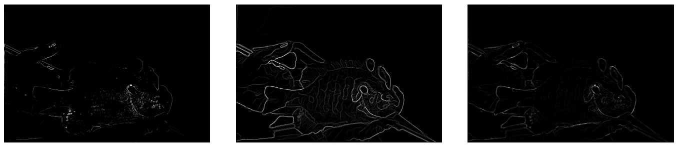


Fig. 37. Image 9 Sobel Baseline, Canny Baseline, (Ours) Pb-Lite Output

A. Network 1: Initial Custom Network

The first network is a quick and simple custom network composed of 4 convolutional layers and 2 fully-connected (FC) final layers. Each convolutional layer is followed by a ReLU non-linear activation function. The filter sizes of the 4 layers

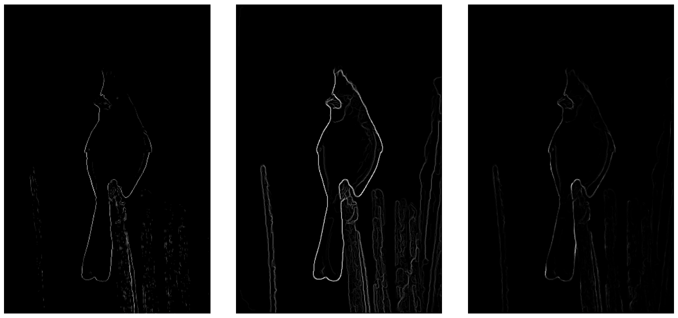


Fig. 38. Image 10 Sobel Baseline, Canny Baseline, (Ours) Pb-Lite Output

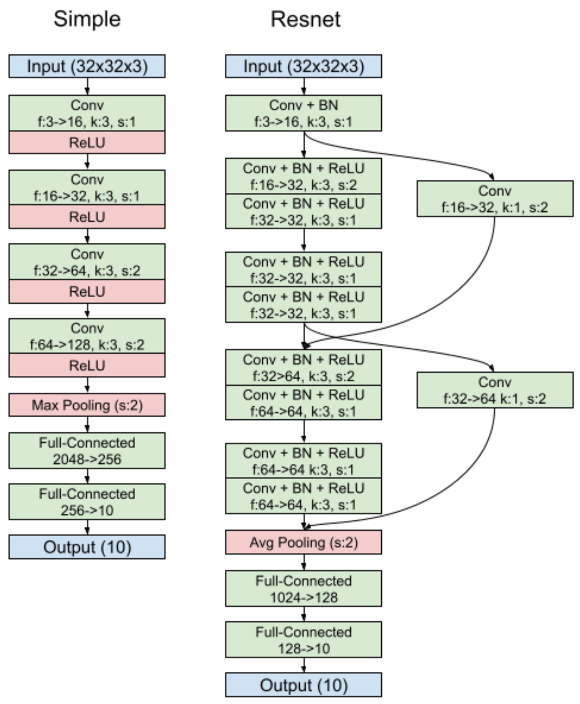
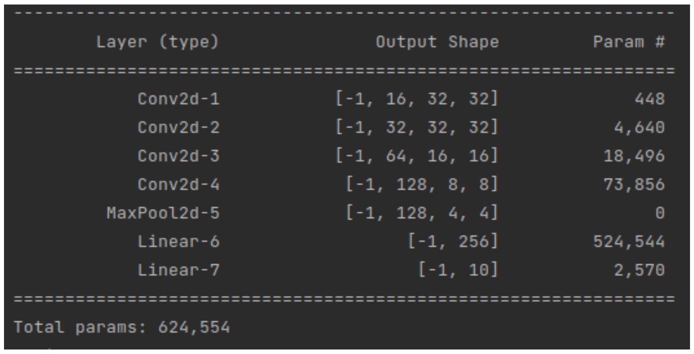


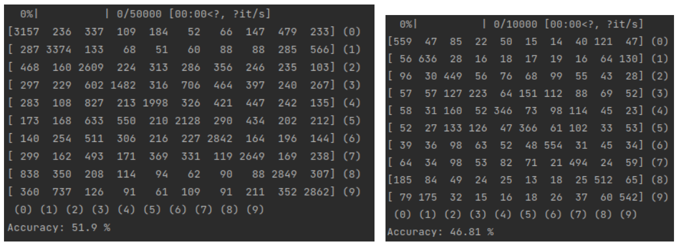
Fig. 39. Simple Network vs Resnet Architecture

are [16, 32, 64, 128], all with kernel sizes of 3x3. The last 2 layers use a stride of size 2 to start shrinking the feature space. Lastly the space is flattened into a 2048 vector before an FC layer of 256 and then 10 for the 10 output classes. The network is trained with a batch size of 128 from 64 images with 1 random flip applied to each to get 128 total batch size. The simple, common SGD (Stochastic Gradient Descent) optimizer is used with a learning rate of 0.01 and momentum of 0.9. This learning rate was found to work best after trying 0.1 and 0.001 options. The simple custom architecture is shown in Fig. 40 with 624,554 parameters. Training accuracy reached 51.9% and test accuracy was 46.8%. Training accuracy and loss plots are shown in Fig. 42. The second, improved version of this network added dropout layers between all convolutional layers to randomly drop connections 20% of the time to prevent worse overfitting. Adding the dropout during training did improve the performance, as expected. Additional convolutional layers were also added with higher numbers of channels between the first and second versions as well.



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 16, 32, 32]             448
            Conv2d-2           [-1, 32, 32, 32]           4,640
            Conv2d-3           [-1, 64, 16, 16]          18,496
            Conv2d-4            [-1, 128, 8, 8]          73,856
         MaxPool2d-5            [-1, 128, 4, 4]               0
            Linear-6                  [-1, 256]         524,544
            Linear-7                   [-1, 10]           2,570
================================================================
Total params: 624,554
```

Fig. 40. Simple Network Architecture



```
  0%|          | 0/50000 [00:00<?, ?it/s]
[3157  236  337  109  184   52   66  147  479  233] (0)
[ 287 3374  133   68   51   60   88   88  285  566] (1)
[ 468  160 2609  224  313  286  356  246  235  103] (2)
[ 297  229  602 1482  316  706  464  397  240  267] (3)
[ 283  108  827  213 1998  326  421  447  242  135] (4)
[ 173  168  633  550  210 2128  290  434  202  212] (5)
[ 140  254  511  306  216  227 2842  164  196  144] (6)
[ 299  162  493  171  369  331  119 2649  169  238] (7)
[ 838  350  208  114   94   62   90   88 2849  307] (8)
[ 360  737  126   91   61  109   91  211  352 2862] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 51.9 %
```

```
  0%|          | 0/10000 [00:00<?, ?it/s]
[559  47  85  22  50  15  14  40 121  47] (0)
[ 56 636  28  16  18  17  19  16  64 130] (1)
[ 96  30 449  56  76  68  99  55  43  28] (2)
[ 57  57 127 223  64 151 112  88  69  52] (3)
[ 58  31 160  52 346  73  98 114  45  23] (4)
[ 52  27 133 126  47 366  61 102  33  53] (5)
[ 39  36  98  63  52  48 554  31  45  34] (6)
[ 64  34  98  53  82  71  21 494  24  59] (7)
[185  84  49  24  25  13  18  25 512  65] (8)
[ 79 175  32  15  16  18  26  37  60 542] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 46.81 %
```

Fig. 41. Simple Network Confusion Matrix for Train and Test

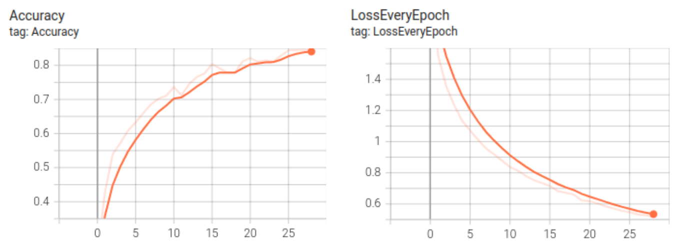
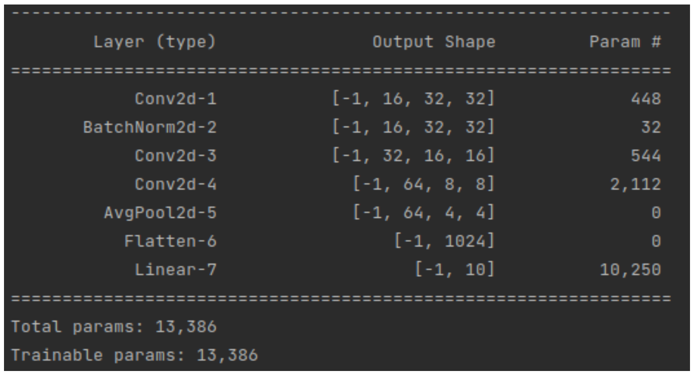


Fig. 42. Simple network training accuracy and loss over epochs

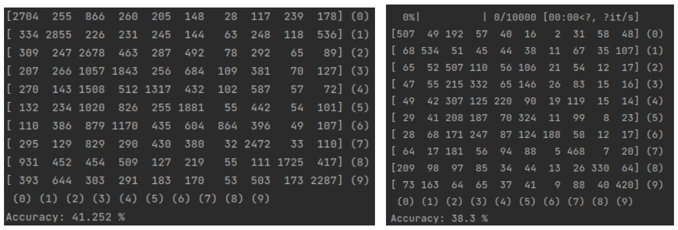
B. Network 2: ResNet

ResNet is a very well known network from the ImageNet dataset challenge. ResNet improves on original convolutional networks by adding residual connections that connect earlier feature maps to be added after further convolutional layers are performed. This helps with several issues including the vanishing-gradient problem, among others. The ResNet implemented in this homework is a miniaturized version since the full version is unnecessarily large for the small CIFAR-10 images that are sized 32x32 instead of the much larger ImageNet database. This implementation uses 2 conv block units with 2 sets of filters in each. The network is trained with a batch size of 128 from 64 images with 1 random flip applied to each to get 128 total batch size. The sipmle, common SGD (Stochastic Gradient Descent) optimizer is used with a learning rate of 0.01 and momentum of 0.9. This learning rate was found to work best after trying 0.1 and 0.001 options. The ResNet architecture is shown in Fig. 43 with 13,386 parameters. Training accuracy reached 51.9% and test accuracy was 46.8%. Training accuracy and loss plots are shown in Fig. 45.



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 16, 32, 32]             448
       BatchNorm2d-2           [-1, 16, 32, 32]              32
            Conv2d-3           [-1, 32, 16, 16]             544
            Conv2d-4             [-1, 64, 8, 8]           2,112
         AvgPool2d-5             [-1, 64, 4, 4]               0
           Flatten-6                 [-1, 1024]               0
            Linear-7                   [-1, 10]          10,250
================================================================
Total params: 13,386
Trainable params: 13,386
```

Fig. 43. Resnet Network Architecture



```
[2704  255  866  260  205  148   28  117  239  178] (0)
[ 334 2855  226  231  245  144   63  248  118  536] (1)
[ 309  247 2678  463  287  492   78  292   65   89] (2)
[ 207  266 1057 1843  256  684  109  381   70  127] (3)
[ 270  143 1508  512 1317  432  102  587   57   72] (4)
[ 132  234 1020  826  255 1881   55  442   54  101] (5)
[ 110  386  879 1170  435  604  864  396   49  107] (6)
[ 295  129  829  290  430  380   32 2472   33  110] (7)
[ 931  452  454  509  127  219   55  111 1725  417] (8)
[ 393  644  303  291  183  170   53  503  173 2287] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 41.252 %
```

```
  0%|          | 0/10000 [00:00<?, ?it/s]
[507  49 192  57  40  16   2  31  58  48] (0)
[ 68 534  51  45  44  38  11  67  35 107] (1)
[ 65  52 507 110  56 106  21  54  12  17] (2)
[ 47  55 215 332  65 146  26  83  15  16] (3)
[ 49  42 307 125 220  90  19 119  15  14] (4)
[ 29  41 208 187  70 324  11  99   8  23] (5)
[ 28  68 171 247  87 124 188  58  12  17] (6)
[ 64  17 181  56  94  88   5 468   7  20] (7)
[209  98  97  85  34  44  13  26 330  64] (8)
[ 73 163  64  65  37  41   9  88  40 420] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 38.3 %
```

Fig. 44. Resnet Confusion Matrix for Train and Test

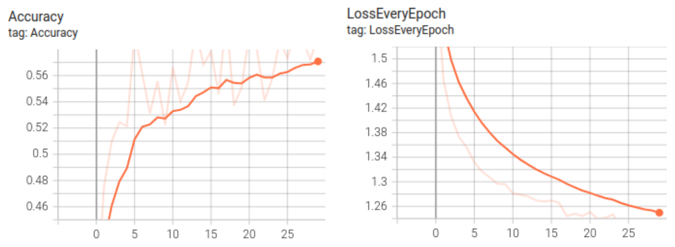
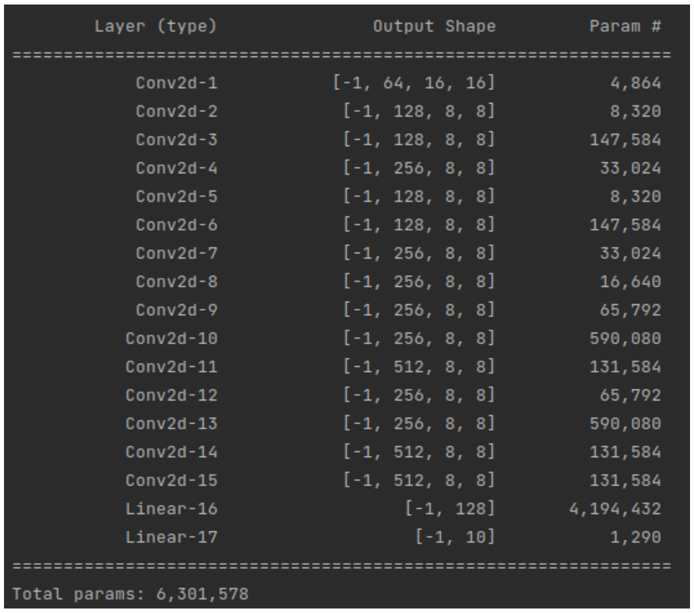


Fig. 45. Resnet training accuracy and loss over epochs

C. Network 3: ResNext

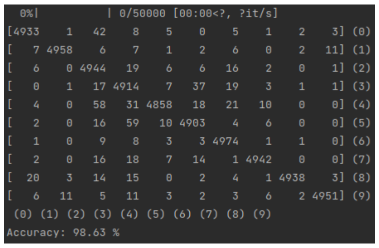
ResNext builds on the original ResNet by modifing the conv block units to have several branches of mini conv layers inside to collect additional features. The branches are summed and then passed to the next conv block unit to be split and re-joined again. In this homework implementation, it is again a smaller version of the originaal due to the CIFAR-10 application. This network has 2 conv block units of 2 mini branches ("cardinality" = 2) in each. The network is trained with a batch size of 128 from 64 images with 1 random flip applied to each to get 128 total batch size. The sipmle, common SGD (Stochastic Gradient Descent) optimizer is used with a learning rate of 0.01 and momentum of 0.9. This learning rate was found to work best after trying 0.1 and 0.001

options. The ResNext architecture is shown in Fig. 46 with 6,301,578 parameters. Training accuracy reached 98.63% and test accuracy was 73.91%. Training accuracy and loss plots are shown in Fig. 48.

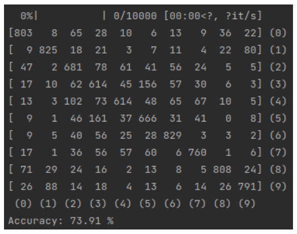


```
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 16, 16]           4,864
            Conv2d-2            [-1, 128, 8, 8]           8,320
            Conv2d-3            [-1, 128, 8, 8]         147,584
            Conv2d-4            [-1, 256, 8, 8]          33,024
            Conv2d-5            [-1, 128, 8, 8]           8,320
            Conv2d-6            [-1, 128, 8, 8]         147,584
            Conv2d-7            [-1, 256, 8, 8]          33,024
            Conv2d-8            [-1, 256, 8, 8]          16,640
            Conv2d-9            [-1, 256, 8, 8]          65,792
           Conv2d-10            [-1, 256, 8, 8]         590,080
           Conv2d-11            [-1, 512, 8, 8]         131,584
           Conv2d-12            [-1, 256, 8, 8]          65,792
           Conv2d-13            [-1, 256, 8, 8]         590,080
           Conv2d-14            [-1, 512, 8, 8]         131,584
           Conv2d-15            [-1, 512, 8, 8]         131,584
           Linear-16                  [-1, 128]       4,194,432
           Linear-17                   [-1, 10]           1,290
================================================================
Total params: 6,301,578
```

Fig. 46. ResNext Network Architecture



```
  0%|          | 0/50000 [00:00<?, ?it/s]
[4933    1   42    8    5    0    5    1    2    3] (0)
[   7 4958    6    7    1    2    6    0    2   11] (1)
[   6    0 4944   19    6    6   16    2    0    1] (2)
[   0    1   17 4914    7   37   19    3    1    1] (3)
[   4    0   58   31 4858   18   21   10    0    0] (4)
[   2    0   16   59   10 4903    4    6    0    0] (5)
[   1    0    9    8    3    3 4974    1    1    0] (6)
[   2    0   16   18    7   14    1 4942    0    0] (7)
[  20    3   14   15    0    2    4    1 4938    3] (8)
[   6   11    5   11    3    2    3    6    2 4951] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 98.63 %
```



```
  0%|          | 0/10000 [00:00<?, ?it/s]
[803   8  65  28  10   6  13   9  36  22] (0)
[  9 825  18  21   3   7  11   4  22  80] (1)
[ 47   2 681  78  61  41  56  24   5   5] (2)
[ 17  10  62 614  45 156  57  30   6   3] (3)
[ 13   3 102  73 614  48  65  67  10   5] (4)
[  9   1  46 161  37 666  31  41   0   8] (5)
[  9   5  40  56  25  28 829   3   3   2] (6)
[ 17   1  36  56  57  60   6 760   1   6] (7)
[ 71  29  24  16   2  13   8   5 808  24] (8)
[ 26  88  14  18   4  13   6  14  26 791] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 73.91 %
```

Fig. 47. ResNext Confusion Matrix for Train and Test

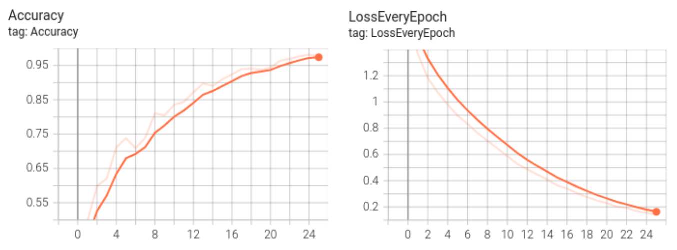
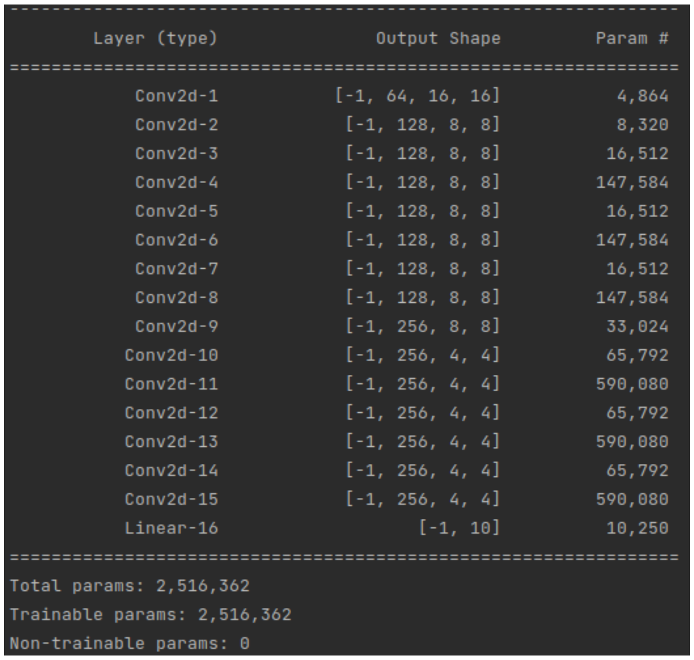


Fig. 48. ResNext training accuracy and loss over epochs

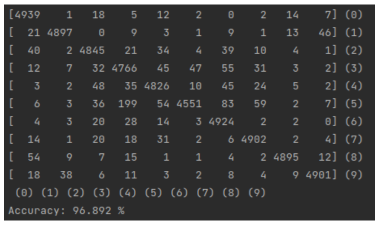
D. Network 4: DenseNet

DenseNet also builds on ideas from ResNet with residual connections. Dense blocks are structured with several convolution layers inside with residual connections running from every layer to every future layer. The network implemented in this homework is again a reduced size for this application, with 2 dense blocks with 3 convolution sections (of 2 Conv layers) in each. The network is trained with a batch size of 128 from 64 images with 1 random flip applied to each to get 128 total batch size. The sipmle, common SGD (Stochastic Gradient Descent) optimizer is used with a learning rate of 0.01 and momentum of 0.9. This learning rate was found to work best after trying 0.1 and 0.001 options. The DenseNet architecture is shown in Fig. 49 with 2,516,362 parameters. Training accuracy reached 96.8% and test accuracy was 76%. Training accuracy and loss plots are shown in Fig. 51.

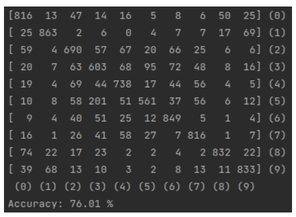


```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 16, 16]           4,864
            Conv2d-2            [-1, 128, 8, 8]           8,320
            Conv2d-3            [-1, 128, 8, 8]          16,512
            Conv2d-4            [-1, 128, 8, 8]         147,584
            Conv2d-5            [-1, 128, 8, 8]          16,512
            Conv2d-6            [-1, 128, 8, 8]         147,584
            Conv2d-7            [-1, 128, 8, 8]          16,512
            Conv2d-8            [-1, 128, 8, 8]         147,584
            Conv2d-9            [-1, 256, 8, 8]          33,024
           Conv2d-10            [-1, 256, 4, 4]          65,792
           Conv2d-11            [-1, 256, 4, 4]         590,080
           Conv2d-12            [-1, 256, 4, 4]          65,792
           Conv2d-13            [-1, 256, 4, 4]         590,080
           Conv2d-14            [-1, 256, 4, 4]          65,792
           Conv2d-15            [-1, 256, 4, 4]         590,080
           Linear-16                   [-1, 10]          10,250
================================================================
Total params: 2,516,362
Trainable params: 2,516,362
Non-trainable params: 0
```

Fig. 49. DenseNet Network Architecture



```
[4939    1   18    5   12    2    0    2   14    7] (0)
[  21 4897    0    9    3    1    9    1   13   46] (1)
[  40    2 4845   21   34    4   39   10    4    1] (2)
[  12    7   32 4766   45   47   55   31    3    2] (3)
[   3    2   48   35 4826   10   45   24    5    2] (4)
[   6    3   36  199   54 4551   83   59    2    7] (5)
[   4    3   20   28   14    3 4924    2    2    0] (6)
[  14    1   20   18   31    2    6 4902    2    4] (7)
[  54    9    7   15    1    1    4    2 4895   12] (8)
[  18   38    6   11    3    2    8    4    9 4901] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 96.892 %
```



```
[816  13  47  14  16   5   8   6  50  25] (0)
[ 25 863   2   6   0   4   7   7  17  69] (1)
[ 59   4 690  57  67  20  66  25   6   6] (2)
[ 20   7  63 603  68  95  72  48   8  16] (3)
[ 19   4  69  44 738  17  44  56   4   5] (4)
[ 10   8  58 201  51 561  37  56   6  12] (5)
[  9   4  40  51  25  12 849   5   1   4] (6)
[ 16   1  26  41  58  27   7 816   1   7] (7)
[ 74  22  17  23   2   2   4   2 832  22] (8)
[ 39  68  13  10   3   2   8  13  11 833] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 76.01 %
```

Fig. 50. DenseNet Confusion Matrix for Train and Test

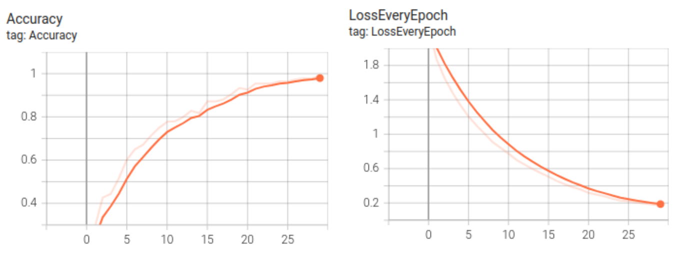


Fig. 51. DenseNet training accuracy and loss over epochs

E. Conclusion and Comparison

Overall, the DenseNet performed the best on the CIFAR-10 dataset with 76% on the test set vs ResNext in 2nd place at 73.9%, despite DenseNet having only around 30% the number of parameters as ResNext. It is noticed that the Resnet architecture has an especially small number of parameters in comparison to the others. This is due to it's reduced scale being reduced a bit too much, and it would likely perform more similar to ResNext if it had a larger number of parameters from more blocks of layers.

III. Conclusion

Overall, this homework was a nice intro to the computer vision class, with both classic CV and deep learning. The classic-CV implemented basic filtering methods to create an improved edge detector modeled after (and greatly simplified from) the Berkeley PB boundary detector [1]. The deep learning section was more advanced to learn about three popular architectures and implement an adaptation of all three to perform on the CIFAR-10 dataset instead of ImageNet.

References

[1] P. Arbeláez, M. Maire, C. Fowlkes and J. Malik, "Contour Detection and Hierarchical Image Segmentation," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 33, no. 5, pp. 898-916, May 2011.

[2] https://www.l3harrisgeospatial.com/docs/laplacianfilters.html

[3] https://en.wikipedia.org/wiki/Gabor$_{filter}$