Homework 0 - Alohomora! Report

Sigurthor Bjorgvinsson
Department of Computer Science
University of Maryland
College Park, Maryland 20740

I. Introduction

In this project, we touched on boundary detection and deep learning. In Section II I will describe the steps taken to implement a simplified version of the Pb algorithm[1] and in section III I will describe the first network I created, how I improved it and how I introduced ideas from ResNet[3], ResNeXt[4] and DenseNet[5] into my model. I will then end with a comparison between the four different networks implemented.

II. Phase 1: Shake My Boundary

In this section I will describe my attempt at implementing the Pb-Lite boundary detection algorithm along with few extra filters. Finally I end this section with discussion about how to execute my code.

A. Derivative of Gaussian (DoG) Filter Bank

To created the DoG Filter Bank, I needed to create a gaussian matrix. In later filters, I needed to have seperate σ_x and σ_y values so my gaussian function used the following equation

$$M_{x,y} = \frac{1}{2\pi\sigma_x\sigma_y} e^{(\frac{x^2}{2\sigma_x^2} + \frac{y^2}{2\sigma_y^2 i})} \tag{1}$$

To get the derivative of the gaussian matrix, I convolved gradient matrices shown below with the 2d Gaussian Matrix generated with equation 1 where σ_x and σ_y were the same.

$$\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

When these matrices are convolved, with zero padding, with the Gaussian matrix, the output are two matrices that are the gradient G_x and G_y respectively. This gradient is a close estimate of the derivative of the Gaussian function. I started out with the derived function of the Gaussian equation but in later filters I was having issues introducing the σ_x and σ_y so this approach was selected.

The rotation of the filters is performed by using equation 2

$$F = \cos(rad) * G_x + \sin(rad) * G_y \tag{2}$$

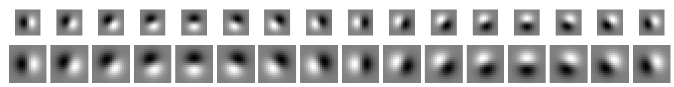

Fig. 1. DoG Filter Bank

The size ($kSize\ x\ kSize$) of the filters are selected by the following equation to make sure that most of distribution is included in the filter:

$$\begin{aligned} &kSize = \lceil \sigma * 6 \rceil + 1 \\ &if\ kSize\%2\ ==\ 0: \\ &\quad kSize+ = 1 \end{aligned} \tag{3}$$

Example filters are shown in figure 1. If the size of the filter is smaller than other filters in the image, the smaller filters are padded with white background. When the images were resized or extended, the differences between the filters was not apparent which lead to this decision.

B. Leung-Malik (LM) Filter Bank

The LM Filter Bank includes 4 sets of filters which I will describe in this section. For the Gaussian derivatives, the sigmas used in the LM smaller, figure 2 are $\sigma = \{1, \sqrt{2}, 2\}$ and in the LM Larger, figure 3 are $\sigma = \{\sqrt{2}, 2, 2\sqrt{2}\}$. Derivatives are rotate in 6 different orientations split over 180°.

A issue occured when I needed to rate these morphed Gaussian distribution filters. Equation 2 did not work for the derivatives in this bank so a function was implemented using the openCV rotate functionality. This resulted in some smudges but I was unable to find another way.

1) First Derivative Gaussian: The first set(first 3 rows on the left) is the first derivative of a morphed Gaussian distribution. These filters are similar to DoG filters, except that they are stretched in one direction. Here the separate σ values in equation 1 comes in and the σ_y is set to $3 * \sigma_x$. Convolution to get the G_y is applied once with zero padding to preserve the filter size.

2) Second Derivative Gaussian: The second set (first 3 rows on the right) is the second derivative of the same Gaussian distribution as above. Here, the matrix used to get G_y is convolved twice to get the second derivative.

3) Laplacian of Gaussian (LoG): The LoG filters are the third set (left most 8 filters in the last row). These filters

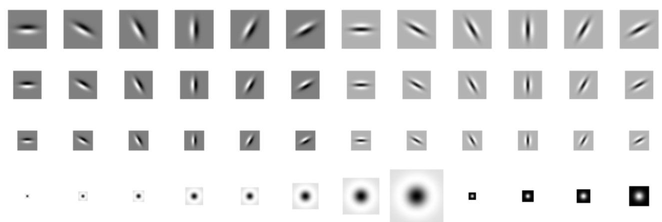

Fig. 2. LM Filter Bank Small

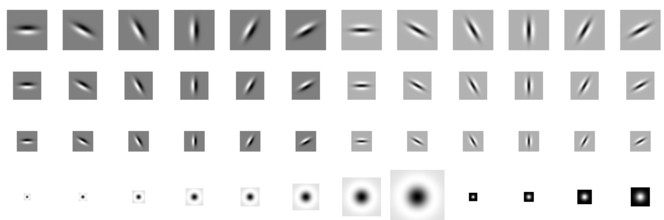

Fig. 3. LM Filter Bank Large

are generated by convolving the following matrix, without padding, to a Gaussian distribution matrix:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

The reason why the convolution was done without padding was that the edges would of the output matrix would be wrong because it detects an edge. This edge is only there because of the zero padding which cuts the small numbers on the edges.

Another filter was tested with all -1 around a centered 8 but that filter provided filters further from the examples which lead to the decision of using the 4 centered.

These filters were generated with these $\sigma = \{1, \sqrt{2}, 2, 2\sqrt{2}\}$ and 3σ for LM small and $\sigma = \{\sqrt{2}, 2, 2\sqrt{2}, 4\}$ and 3σ for LM large

4) Gaussian: These filters are just simple Gaussian filters generated with the same σ using the gaussian equation 1. These filters were generated with these $\sigma = \{1, \sqrt{2}, 2, 2\sqrt{2}\}$ for LM small and $\sigma = \{\sqrt{2}, 2, 2\sqrt{2}, 4\}$ for LM large.

C. Gabor Filter Bank

The Gabor filters are generated with a Gaussian distribution with is modulated by a sinusoidal plane wave. The filter size was preset so that the filters could occupy the entire filter size so they do not use equation 3. To generate these filters, there is an equation that takes 5 variables. These variables are σ, θ, λ, psi and γ. The way that I understand these variables are the following. σ is the standard deviation, θ is the rotation, λ is the wave length, psi is the offset and γ is the offset of the filter in terms of width. Figure 4 shows the generated filters filter size of 13 and 6 orientations of 360°. Gamma was not used but the σ, θ, psi pairs used were

$$\{(2.0, 1.5, 0.0), (3.0, 3.0, 0.0), (4.0, 4.5, 0.0), (5.0, 7.0, 0.0), (7.5, 10.0, 0.0), (7.5, 10.0, 15.0)\}$$

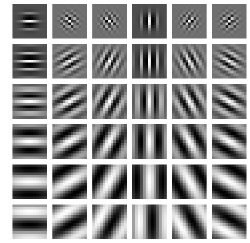

Fig. 4. Gabor Filter Bank

where the last change in psi was to invert the wave or shift it so that it seems inverted.

D. Maps

In this subsection I will discuss how the Texton map, Brightness map and Color map was generate and clustered. The texton map is generated after convolving some of the filters selected from the banks above and then clustering into 64 clusters using k-means. My tests showed that using only the Gaussian filters from step one produced the best results. The brightness map was the default gray scale values clustered into 16 clusters and the colors was normalized RGB values clustered into 16 values.

Before clustering, data manipulation was required reshape the data from a 3D to 2D. The colors used in the texton map were generated using a HSV color distribution which was borrowed from stack overflow (link in comment)

E. Map Gradients

To get the gradients of the maps, I first needed to create half-disks to get the difference changes in cluster ids.

1) Half-disk Filter Bank: The half-disk masks were implemented by starting the loop on the right half of the filter with all values set to 0. Looping over the right side, a check was created to see if the euclidean distance of the coordinates were within a certain radius. If it was within the radius, the cell got the value 1. These filters were then flipped to create a matching disk. These disks were rotated 8 times using OpenCV and each time a matching disk was created. The filter had 3 sizes which were $\{7, 17, 27\}$ and are displayed in figure 6.

2) Chi-Distance: The Chi-Distance calculates the distance between two histograms. the two histograms here are the values for each pair of half disks for all the bins. Once all the filters were applied, the Chi-Distance was calculated with the following equation:

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

I selected, after few tests of mean, median and max that the mean of all the Chi-Distances was the most optimal final distance value for that pixel.

3) Gradients: The output of the Chi-Distance was the gradient of change in texture, brightness and color. figures 7 8 9 display these gradients which are scaled to grayscale values from $0 - 255$

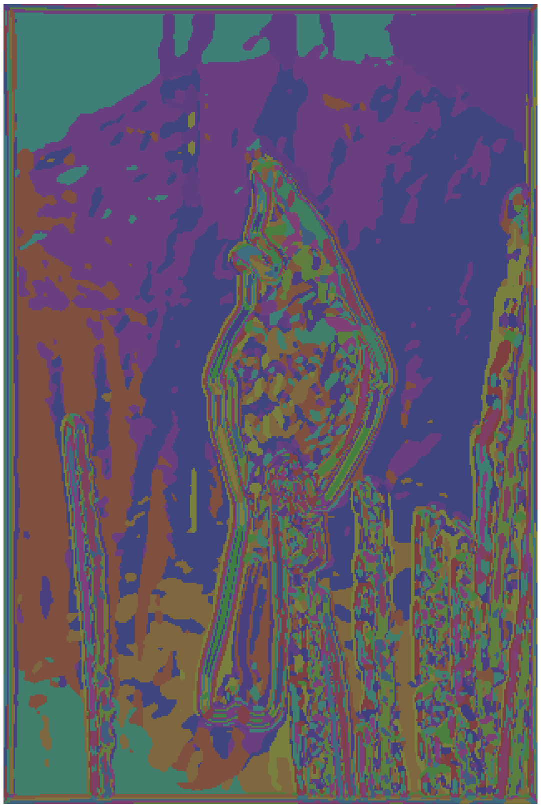

Fig. 5. Texton Map

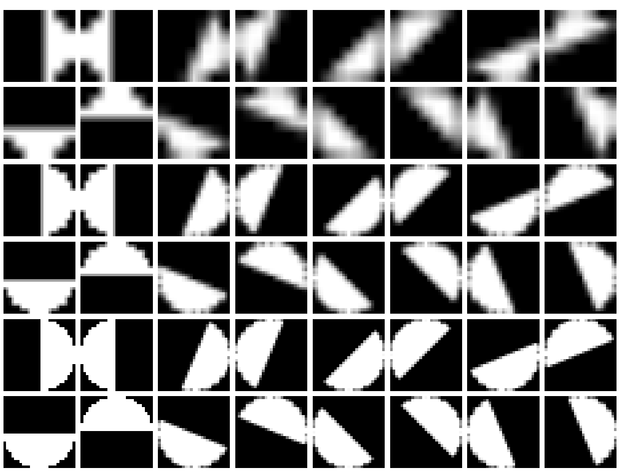

Fig. 6. Half-disk Filter Bank

4) Final Pb-lite Output: The final output from the detector was then not a edge by it self but rather a weight on the Canny and Sobel edge detection algorithms. The weights were the gradient values combined:

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

The weights that I selected for Canny and Sobel were 0.4 and 0.6 respectively. These weights were selected because with

Fig. 7. Brightness Gradient

Fig. 8. Color Gradient

Fig. 9. Texton Gradient

manual inspection, Sobel seemed to have better edge detection on the main feature/item in the image while Canny found all edges. The final output is then shown in figure [**?**]

F. Execution Notes

To execute the Wrapper.py, you need pass in a path to the base folder which should have the SobelBaseline, CannyBaseline and Images folder as the first argument and then the image name as the second argument.

Because of issues with cv2.imshow where if multiple windows were opened consecutively, the last two images would be distorted or not displayed, the images are only saved to the disk.

Example: 'python Wrapper.py ../BSDS500/ 1'

Fig. 10. Pb-lite Final Output

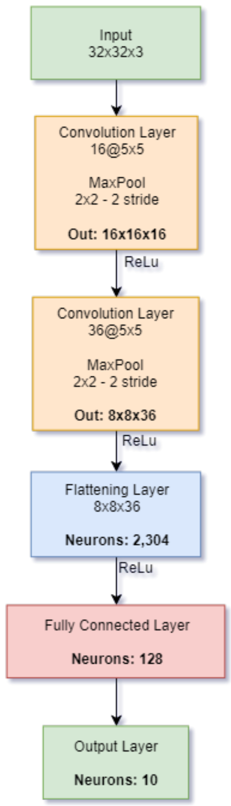

Fig. 11. My First Network

III. Phase 2: Deep Dive on Deep Learning

In this section I will describe the network I created, what improvements I did to it which were inspired from and then on compare it to ResNet, ResNeXt and DenseNet. I was limited by the number of epochs I could do because of the slow training of the network.

A. My First Neural Network

I used the Hvass Labs tutorial [2] on convolutional networks to guide me through creating a convolutional network. This came of great help because I had not implemented a neural network in TensorFlow before. My network was structured as had the following parameters.

- Number of parameters: 311,982
- Optimizer: AdamOptimizer
- Learning Rate: 0.001
- Batch Size: 16
- Epochs: 15

My network architecture is shown in figure 11

The model was best trained after 6 epochs with 58.19% accuracy on the test data. figure 12 shows train accuracy, figure 13 shows training loss, figure 14 the confusion matrix on the best trained model and figure 15 shows the test accuracy.

I ran my network with multiple different values for Epoch and Mini Batch Size without any noticeable difference in test accuracy.

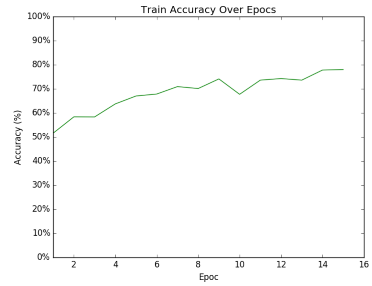

Fig. 12. First Network: Train accuracy over epochs

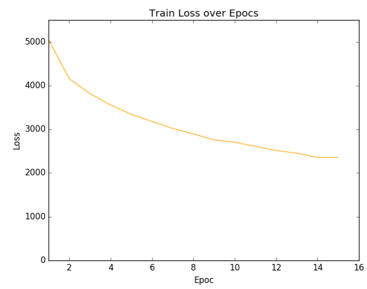

Fig. 13. First Network: Train Loss over epochs

```
Best Confusion Test:
[677  32  43  26  40  11   6  18  84  63] (0)
[ 24 760   2   9  13   9   8   9  24 142] (1)
[113  12 379 101 164 116  45  34  16  20] (2)
[ 48  26  57 354 105 271  35  36  28  40] (3)
[ 32  11  85  69 600  60  41  78  11  13] (4)
[ 22   6  61 132 101 553  24  52  26  23] (5)
[ 16  24  42 113 144  71 541  14  14  21] (6)
[ 23  10  34  39 104 114   9 630   8  29] (7)
[164  89  13  11  23  10   4   9 627  50] (8)
[ 50 135   9  21  13  17   5  24  28 698] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)

Best Confusion Train:
[4103   67  106   63  122   44   12   47  223  213] (0)
[  68 4323    5   27   52   19   20    7   74  405] (1)
[ 429   70 2722  323  697  399  124   93   68   75] (2)
[ 156   73  172 2392  475 1196  144  137  122  133] (3)
[ 201   29  246  217 3683  193  101  252   40   38] (4)
[  52   39  161  506  347 3514   62  162   59   98] (5)
[  62   93  169  443  637  247 3177   51   38   83] (6)
[  65   31   70  150  422  390   14 3750   24   84] (7)
[ 652  245   56   48   68   21    6   15 3715  174] (8)
[ 145  446   10   77   38   34   14   46  106 4084] (9)
  (0)  (1)  (2)  (3)  (4)  (5)  (6)  (7)  (8)  (9)
```

Fig. 14. First Network: Test and Train confusion matrix on best model

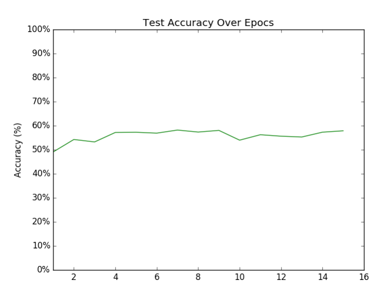

Fig. 15. First Network: Test accuracy over epochs

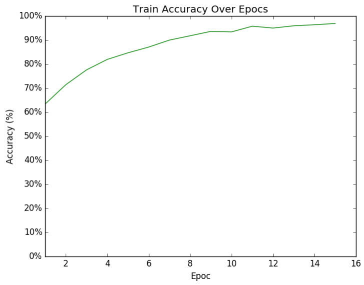

Fig. 16. Improved Network: Train accuracy over epochs

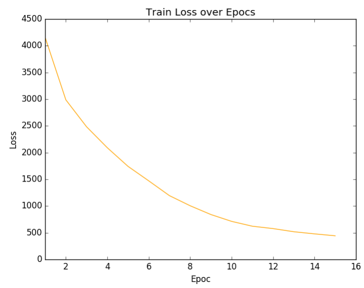

Fig. 17. Improved Network: Train Loss over epochs

B. Improving Accuracy

When trying to improve accuracy of my model, I attempted to standardize the values on my images. For all images, I divided all pixel values by 127.5 and subtracted 1 which resulted in all values being in the range [-1,1]This increased the accuracy of my model to 69.69% on test data. Because of time restrictions I was unable to evaluate more improvements.

I attempted decaying the learning rate but that did not work for me. I read online that the AdamOptimizer already has decaying learning rate but was unable to verify that. It did not increase my accuracy but rather lowered it. I did not augmenting my data because of the time taken to train my current data set.

The model was best trained after 3 epochs with 69.69% accuracy on the test data. figure 16 shows train accuracy, figure 17 shows training loss, figure 18 the confusion matrix on the best trained model and figure 19 shows the test accuracy.

```
Best Confusion Test:
[750  14  44  11   9   8   7   6 102  49] (0)
[ 19 820   8   9   2   3   3   3  38  95] (1)
[ 78  10 672  42  60  49  34  28  16  11] (2)
[ 41  16 129 453  40 169  53  32  29  38] (3)
[ 41   8 126  65 593  36  48  56  18   9] (4)
[ 24   6  86 162  34 602  21  38  14  13] (5)
[  9  11 112  57  41  19 716   8  10  17] (6)
[ 25   9  51  34  79  50   6 704  12  30] (7)
[ 57  26  12   4   3   2   4   1 862  29] (8)
[ 36  86  12  13   3   9   2   9  33 797] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)

Best Confusion Train:
[4317   29  122   26   22   17    3   14  324  126] (0)
[  40 4587   19   12    7    4    4    0   98  229] (1)
[ 236   18 4121   99  154  109   85   41   90   47] (2)
[ 117   32  414 3118  151  675  162  106  106  119] (3)
[ 129   18  467  192 3727  124   99  110   81   53] (4)
[  48   24  322  557  140 3616   79  111   49   54] (5)
[  25   30  429  224   98   98 4009    6   37   44] (6)
[  79    8  183   92  247  112    7 4168   35   69] (7)
[  82   47   28   13    3    3    6    2 4755   61] (8)
[  49  213   27   20    2    8    7   10  100 4564] (9)
  (0)  (1)  (2)  (3)  (4)  (5)  (6)  (7)  (8)  (9)
```

Fig. 18. Improved Network: Test and Train confusion matrix on best model

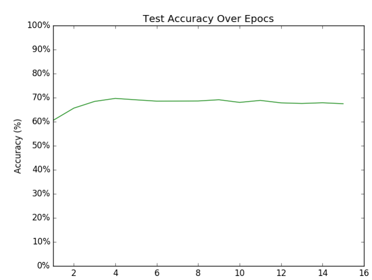

Fig. 19. Improved Network: Test accuracy over epochs

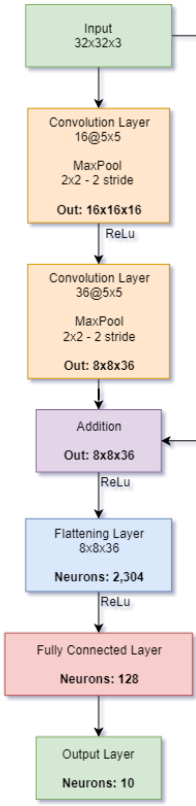

Fig. 20. My ResNet Network

C. ResNet

I decided that I was going to introduce the architecture improvements that the networks included in their paper into my first network instead of implementing their network. ResNet[3] introduced shortcut paths over layers.

I added the shortcut from the raw input to the output of the second convolution layer. The shortcut was first pooled with a filter of size 4 with a stride 4 and the dimensions increased from 3 to 36 by concatenating 8x8x33 zeroed matrices before adding. This implementation performed best by all I tried which I will talk about in the comparison section.

My ResNet shown in figure 20 had 336,558 parameters with all the same configurations as above for comparison purposes.

The model was best trained after 4 epochs with 71% accuracy on the test data. figure 21 shows train accuracy, figure 22 shows training loss, figure 23 the confusion matrix on the best trained model and figure 24 shows the test accuracy.

After multiple attempts at implementing the batch normalization defined in the network I ended up skipping that. It either did nothing or degraded the performance of my network greatly. After consulting with a peer, he set the training argument as true for both training and testing. For me, it worked best when i set it as false for both when training and when testing but I know that was wrong to do so I decided to not use batch normalization.

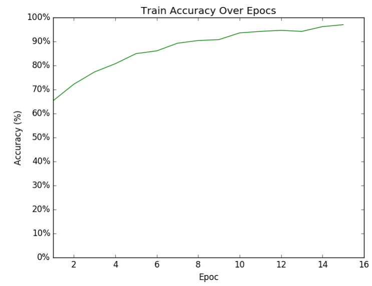

Fig. 21. ResNet: Train accuracy over epochs

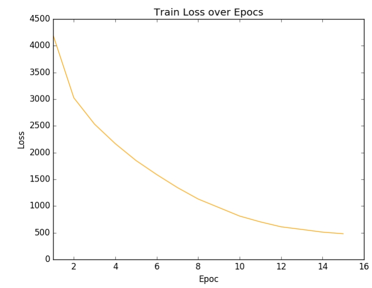

Fig. 22. ResNet: Train Loss over epochs

```
Best Confusion Test:
[740  14  55  16  25   5  18  12  84  31] (0)
[ 19 830   6   8   2   2   7   4  37  85] (1)
[ 61   3 552  79 162  48  50  24  12   9] (2)
[ 16  12  57 565 100 117  61  38  20  14] (3)
[ 17   4  40  64 731  31  41  62   9   1] (4)
[ 11   2  68 223  67 527  20  68   9   5] (5)
[  4  11  37  72  77  20 759   6  10   4] (6)
[ 16   6  27  42  76  40   5 774   6   8] (7)
[ 57  23  10   6  13   4   7   2 859  19] (8)
[ 32  94   9  22   8  11   6  19  36 763] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
```

```
Best Confusion Train:
[4381   30  121   42   99   13   15   27  207   65] (0)
[  38 4639   12   16   13    5   18    8  112  139] (1)
[ 140    6 3616  220  581  108  154  101   58   16] (2)
[  70   12  129 3763  275  379  166  129   54   23] (3)
[  37    3   92  130 4474   55   59  128   17    5] (4)
[  22   11  135  781  246 3545   58  182    7   13] (5)
[  12   18  118  200  208   63 4341   19   17    4] (6)
[  26    3   56  123  185   98   10 4472   12   15] (7)
[  99   44   12   29   31    1   12    8 4721   43] (8)
[  69  200   22   40   19    5   10   25   89 4521] (9)
  (0)  (1)  (2)  (3)  (4)  (5)  (6)  (7)  (8)  (9)
```

Fig. 23. ResNet: Test and Train confusion matrix on best model

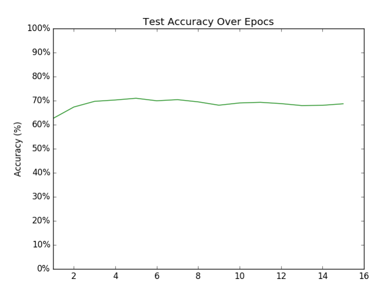

Fig. 24. ResNet: Test accuracy over epochs

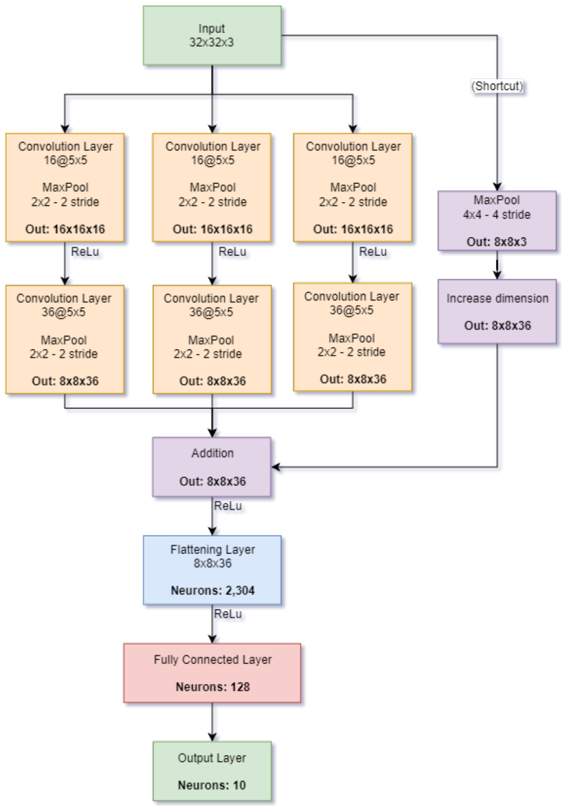

Fig. 25. My ResNeXt Network

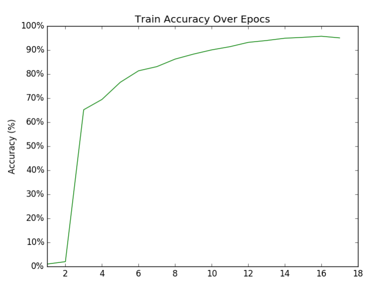

Fig. 26. ResNeXt: Train accuracy over epochs

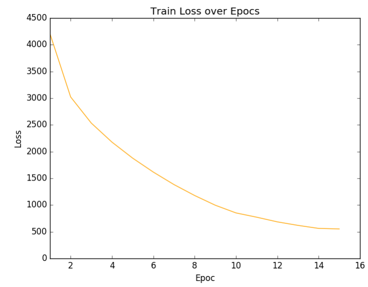

Fig. 27. ResNeXt: Train Loss over epochs

```
Best Confusion Test:
[733  15  49  37  14   7   5  12  75  53] (0)
[ 14 831  10  14   3   5  10   2  43  68] (1)
[ 62   8 584  72  78  56  60  44  17  19] (2)
[ 16  10  67 607  53 107  66  36  20  18] (3)
[ 23   7  78  71 635  21  59  85  17   4] (4)
[ 13   4  53 241  48 521  35  54  17  14] (5)
[  3   9  41  75  34  13 803  13   7   2] (6)
[ 13   5  39  57  49  44   5 767   4  17] (7)
[ 57  33  13  16  10   3   5   5 832  26] (8)
[ 28 119   6  31   1   7  11  15  27 755] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
```

```
Best Confusion Train:
[4205   59  157   82   37   14   29   30  220  167] (0)
[  49 4540   13   29    4    4   26   12  148  175] (1)
[ 199   16 3650  291  277  132  238  111   48   38] (2)
[  54   22  190 3730  186  389  226  100   62   41] (3)
[  80    7  236  215 3896   70  198  249   29   20] (4)
[  24   15  179 1188  166 3082  108  198   18   22] (5)
[  12   21  173  280   80   45 4338   17   22   12] (6)
[  24    4   82  189  137  126   22 4363   12   41] (7)
[ 156   68   25   42   17    8   14    8 4581   81] (8)
[  45  410   13   79   10   12   16   26   83 4306] (9)
  (0)  (1)  (2)  (3)  (4)  (5)  (6)  (7)  (8)  (9)
```

Fig. 28. ResNeXt: Test and Train confusion matrix on best model

D. ResNeXt

ResNeXt[4] proposed instead of making the network deeper, to make the network wider. I implemented this idea into my network by tripling the number of convolution layers. All of the outputs of the convolution layers and from the shortcut were added together (not concatenated) before ReLu and the flattening layer.

My ResNeXt shown in figure 25 had 343,286 parameters with all the same configurations as above for comparison purposes.

The model was best trained after 3 epochs with 70.68% accuracy on the test data. figure 26 shows train accuracy, figure 27 shows training loss, figure 28 the confusion matrix on the best trained model and figure 29 shows the test accuracy.

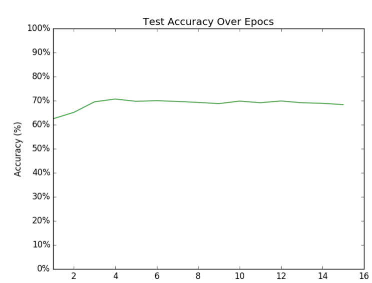

Fig. 29. ResNeXt: Test accuracy over epochs

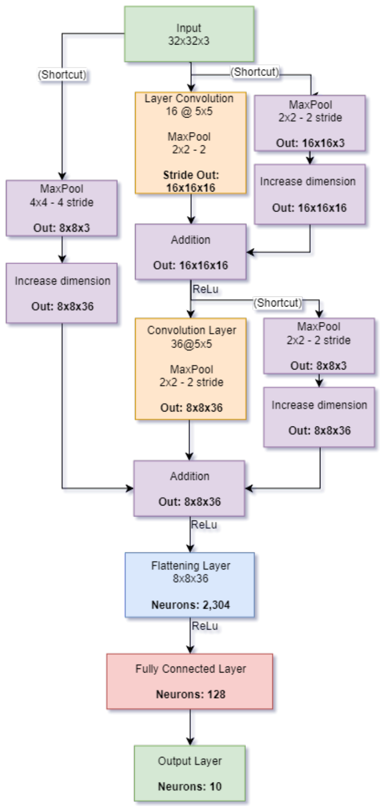

Fig. 30. My DenseNet Network

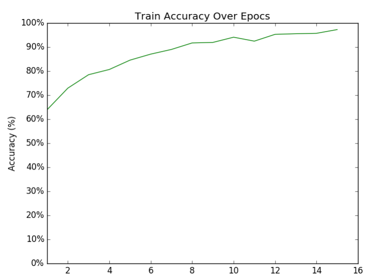

Fig. 31. DenseNet: Train accuracy over epochs

E. DenseNet

DenseNet[5] introduced forward connections that between layers where each output layer got a shortcut from all other layers before it. To implement this I added a shortcut after my convolution first layer (i.e. 1 dense block) to the flattening layer along with the shortcut from the input layer.

My DenseNet shown in figure 30 had 311,982 parameters with all the same configurations as above for comparison purposes.

The model was best trained after 4 epochs with 70.68% accuracy on the test data. figure 31 shows train accuracy, figure 32 shows training loss, figure 33 the confusion matrix on the best trained model and figure 34 shows the test accuracy.

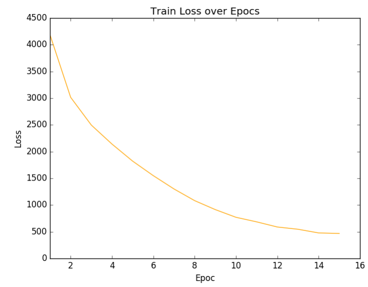

Fig. 32. DenseNet: Train Loss over epochs

```
Best Confusion Test:
[761  10  48  25  20   4  14  12  85  21] (0)
[ 15 762  13  15   2   1  15  10  68  99] (1)
[ 63   4 641  49  81  35  82  21  16   8] (2)
[ 25  11  97 492  82 112 126  35  13   7] (3)
[ 30   4  96  38 675  21  72  55   7   2] (4)
[ 17   2  86 179  53 530  53  64  11   5] (5)
[  6   1  46  42  41  14 829   7  10   4] (6)
[ 16   0  42  46  72  49  21 738   7   9] (7)
[ 48  16  19   8  11   6   5   1 874  12] (8)
[ 42  66  19  13   4   7  10  25  48 766] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)

Best Confusion Train:
[4467    9  130   50   45   11   21   29  205   33] (0)
[  46 4497   16   20    4   10   29   12  136  230] (1)
[ 198    5 4085  106  204   79  223   45   41   14] (2)
[  65    8  244 3451  258  360  448  102   45   19] (3)
[  75    4  320  106 4141   37  155  138   23    1] (4)
[  29    5  253  703  244 3372  209  166   13    6] (5)
[  28   10  115   83   89   19 4626    9   15    6] (6)
[  40    3  138  112  200   84   28 4384    7    4] (7)
[  99   24   33   26   15   11   14    5 4749   24] (8)
[ 104  110   31   47   21    9   19   29  117 4513] (9)
  (0)  (1)  (2)  (3)  (4)  (5)  (6)  (7)  (8)  (9)
```

Fig. 33. DenseNet: Test and Train confusion matrix on best model

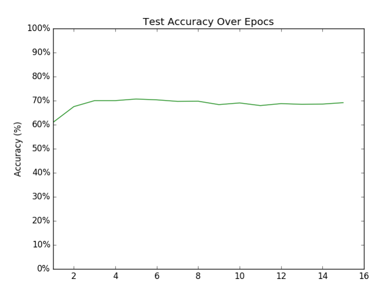

Fig. 34. DenseNet: Test accuracy over epochs

F. Comparison

If we only look at the accuracy, ResNet had the best accuracy with 71%. The ResNet was by far the easiest to understand and to implement.

I tested many approaches to connect the shortcut paths to the results from the layers. The most successful one was the addition and not the concatenation which was recommended from the paper. The concatenation required more parameters to implement because of the added 1x1 convolution required to decrease the dimensions.

To fully evaluate and compare these architecture, I believe that a much larger network is required that just the two convolution layers I had. These architectures provide a solution for a deep network which I do not have. The problem with deep networks is that the deeper it goes, the harder the training becomes because of the diminished gradient. These shortcuts are suppose to help with that problem.

G. Execution Notes

I created a new file called Coach.py. This program uses the Train.py and Test.py functions directly. The Train and Test files still work as expected.

Example:
python Coach.py –BasePath ../CIFAR10 –NumEpochs 15 –MiniBatchSize 16 –OutFolder ./output –LearningRate 0.001 –CheckPointPath ../Checkpoints/ –Normalize True

IV. Conclusion

I learned a lot from being thrown into the deep end like this homework did. I am looking forward to implement more vision and deep learning algorithms in the coming projects. For next projects I need to focus on understanding the problem and solution before starting to implement. This project took too much time with going back and forth with implementations that I didn't know if they would work or not. Never the less, I'm ready to take on this semester.

References

[1] P. Arbelaez and C. Fowlkes, *Contour Detection and Hierarchical Image Segmentation* 2010.

[2] M. E. H. Pedersen, Tutorial #02 - Convolutional Neural Network GitHub, 2016. [Online]. Available: https://github.com/Hvass-Labs/TensorFlow-Tutorials/commits/master/02_Convolutional_Neural_Network.ipynb. [Accessed: 20-Jan-2019].

[3] K. He, X. Zhang, S. Ren, and J. Sun, Deep Residual Learning for Image Recognition, 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[4] S. Xie, R. Girshick, P. Dollar, Z. Tu, and K. He, Aggregated Residual Transformations for Deep Neural Networks, 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017.

[5] G. Huang, Z. Liu, L. V. D. Maaten, and K. Q. Weinberger, Densely Connected Convolutional Networks, 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017.