Homework 0
Alohomora

Shounak Naik
Masters in Robotics
WPI

I. Introduction

This report contains my understanding and implementation of the homework 0. The first phase is related to edge detection and the second phase is related to modelling neural networks.

II. Phase 1: Shake my Boundary

In this section I will explain how I approached and solved the Pb-Lite related assignment. There are 4 filter banks generated before the actual implementation of the Pb-Lite algorithm.

A. DoG Filter Bank

For creating the Derivative of Gaussian, I took a 7x7 Gaussian matrix. I took a 3x3 dimensional Sobel. Convolving Gaussian and Sobel results in an approximation of the Derivative of Gaussian. Once I had the DoG matrix, I used the OpenCV to rotate the matrix so that I could get different orientations of the same matrix. The Sobel Kernel used in my program is as follows:

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

The Gaussian matrix used for the DoG was a ready-made 7x7 Gaussian matrix. For all other banks I have the following Gaussian equation 1 to create Gaussian matrices.

$$G(x,y) = \frac{1}{2\pi\sigma_x\sigma_y} \exp -(\frac{x^2}{2\sigma_x^2}) \exp -(\frac{y^2}{2\sigma_y^2}) \qquad (1)$$

In Fig 1, you can see the DoG filter library obtained by me.

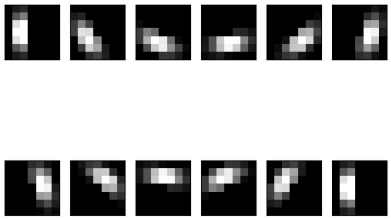

Fig. 1. Derivative of Gaussian Bank

B. LM Filter Bank

For the LMS , the scales taken were $1, \sqrt{2}$ and 2. For LML , the scales taken were $\sqrt{2}$, 2 and $2\sqrt{2}$. To generate the different orientations, I used OpenCV rotate function.

1) First Order Derivative:
The Gaussian used in this case has 2 different σ's - σ_x and σ_y. Thus Equation 1 was used to create these special Gaussian matrices. Here $\sigma_x = 3\sigma_y$. After getting these Gaussian matrices, they were convolved with the Sobel filter to get the first order derivative Gaussian. Of course this is an approximation. The left side of the first 3 rows in Figure 2 and Figure 3 are the First Order matrices

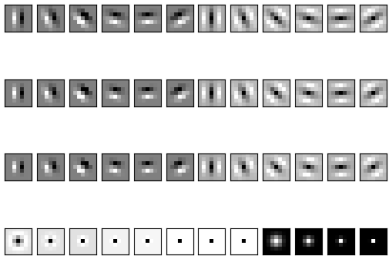

Fig. 2. LM Small

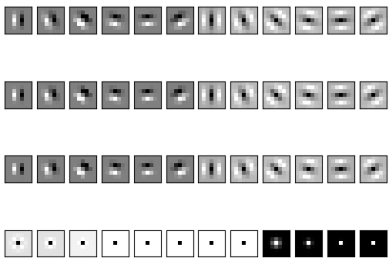

Fig. 3. LM Large

2) Second Order Derivative:
Convolution between the Sobel Filter and the First Order

Fig. 4. Gabor filter bank

Derivative matrix gives us the Second Order martix. The right side of the first 3 rows in Figure 2 and Figure 3 are the Second Order matrices.

3) Laplacian of Gaussian:
LoG was obtained directly with the equation 2. LoG occurs at σ and 3σ. LoG can be seen in the first 8 images in the last row in Figure 2 and Figure 3.

$$L(x,y) = -\frac{1}{\pi\sigma^4}[1 - \frac{x^2+y^2}{2\sigma^2}]\exp -(\frac{x^2+y^2}{2\sigma^2}) \quad (2)$$

4) Gaussian:
Gaussian is obtained by 4 basic scales. These basic scales change according to the type of LM bank that we want to create (Small or Large). Gaussian maps can be seen in the last 4 images in the last row in Figure 2 and Figure 3.

C. Gabor Filters

These filters caught my eye because they also appear in the human visual system. These filters are created by modulating a sinusodial wave onto the Gaussian kernel. There are 5 parameters - σ, γ, θ, λ and ψ. I experimented with the way sinusodial wave can be modulated onto the Gaussian kernel. I tried adding $cosine$ and sin components, but best results were obtained with using just the real component that is the $cosine$ component. Further I only varied parameters of σ, θ and γ. I observed the best visual results are when the σ parameter is near 2. The Gabor filters created can be seen in Figure 4

D. Different Maps

Texton Map was generated by convolving grayscale images with all filter in all of the 3 filter banks mentioned above. I got a 3D tensor by doing the above operation. To quantize this tensor into 2D, I used the Kmeans clustering algorithm with number of cluster as 64. To do this flattened out the multiple 2D images in the 3D tensor. I passed this flattened out images as my input to the Kmeans algorithm. Once I got the Kmeans labels, I resized the flattened array into a 2D array. The resultant Texton map can be seen in 5.

A similar algorithm was used to create the Brightness and the Color Maps. Brightness map only had one channel and the Color Map had 3 channels. For both the Brightness and the Color Map, the number of clusters used was 16.

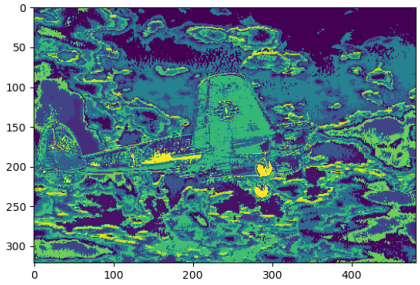

Fig. 5. Texton Map

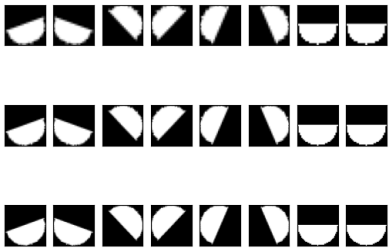

Fig. 6. Half Disk Masks

E. Gradient Maps

1) Half Disk Masks: These masks are useful for calculating the Chi-square distance later. The way I created these half disc masks was by creating masks and then applying them on a fully black image. Firstly, I created a circular mask and then a rectangular mask of side equalling the diameter of the circle. Then I AND'd these 2 masks and then applied this mask to the black image. The Half Disk bank created can be seen in Figure 6

2) Chi-Square Gradient: This was the most tough and the most interesting part of Phase 1 for me. I understood the binning concept only when I understood what a histogram of an image is. I took the bin size as the number of clusters used to create the maps earlier. Firstly for every bin , I masked the map to only contain the bin number. Then I convolved pairs of opposing filters onto the masked map and then calculated the Chi-Square distance according to the Chi-square distance formula given in the assignment.
Since the tensor created from this algorithm was of 3 dimensions, I took a mean across the 3rd dimension - across the same pixel. This gave me a 2D output.
Since many values were 0's, the division led to some NaN values. I handled these cases by simply putting a 0 in their place. This was the most logical solution I could think of as putting 0 would just give us a black pixel and won't give us a false positive pixel. The warning does pop up when executed

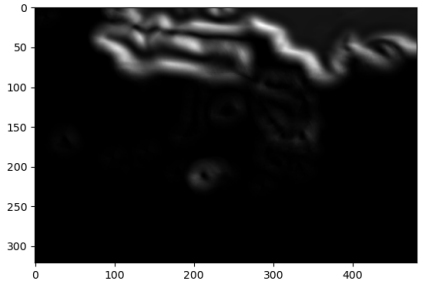

Fig. 7. Texton Gradient

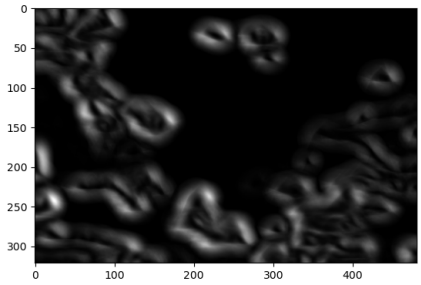

Fig. 8. Brightness Gradient

but that can be ignored as I handle the warning later.

Texton Gradient , Brightness Gradient and Color Gradient were created by using the same above concept. Texton Gradient can be seen in Figure 7. Brightness Gradient can be seen in Figure 8. Color Gradient can be seen in Figure 9

3) Pb-Lite Final Ouptut: A mean over the Texton Gradient, Brightness Gradient and Color Gradient is taken. Canny and Sobel baseline is linearly weighted together with $w1$ and $w2$ being 0.5 both. $w1$ and $w2$ are weights for Canny and Sobel respectively. The final Pb-Lite output can be seen in Figure

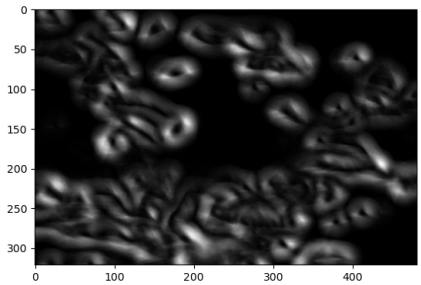

Fig. 9. Color Gradient

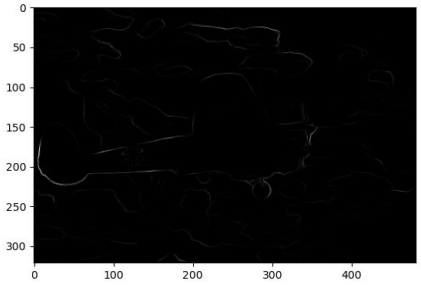

Fig. 10. Pb-Lite Output

10. The figure is light and the edges can be seen when seen closely.
I believe using this algorithm lets us introduce texture, brightness and color components into the edge detection problem. This also reduces other irrelevant edges from the scene.

III. Deep Dive on Deep Learning

This section will be a walk through the Phase2 of the assignment. It will discuss the Convolutional Neural Network architecture that I built to tackle the multiclass classification of the CIFAR-10 dataset. I will also be discussing the theoretically improvements that can be made to the first Neural Network I build in this assignment.

A. Train your First Neural Network

All images in the CIFAR-10 dataset are 3x32x32 and we are supposed to classify them into 10 classes. I went with a relatively simplistic CNN model to solve this problem. The architecture that I built is seen in Figure 11. I experimented with the number of Linear Layers - addition of another Linear layer actually dropped the accuracy significantly. (From 58 % in the current architecture to 50 % in the deeper network). I have trained my network for 10 epochs.

I have used the *torchsummary* package to summarize model architecture and parameters. My network has the following configurations:

1) Number of Parameters: 50,862 (all of them trainable)
2) Optimizer: Adam with LR $= 0.001$
3) Batch Size: 200

The Training Loss graph can be seen in 12. The Training Accuracy graph can be seen in 13. The Test Accuracy graph can be seen in 14. To generate the test accuracy plot, I loaded the model checkpoints at every epoch and calculated the test accuracy. I feel that instead of a test accuracy graph, a validation accuracy graph would be ideal. 3 splits (train, test and validation) would have been better instead of 2splits (train and test).

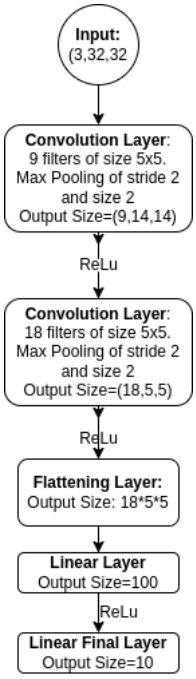

Fig. 11. Primitive CNN

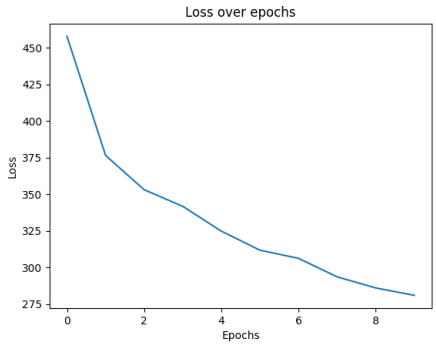

Fig. 12. Training Loss over Epochs

The test confusion matrix can be seen in Figure 15. The train confusion matrix can be seen in Figure 16. The test accuracy is 58.07 % and the train accuracy is 60.61 %.

B. *Improving Accuracy of your Neural Network*

I have tried to improve my primitive CNN I built earlier. To do this, I used the *torchvision transform* API to make a transformation composition. In this composition, I have first randomly rotated photos and then applied a random inversion of colours. This was done so that my network does not overfit on a specific colour too much. Finally, I normalized the images just before I fed them to my network. I also made some changes to my network - I added a Batch Normalization 2D to my Convolutional Network. I applied the BatchNorm just before the MaxPooling layers of my 2 convolutional layers. (See Figure 11)

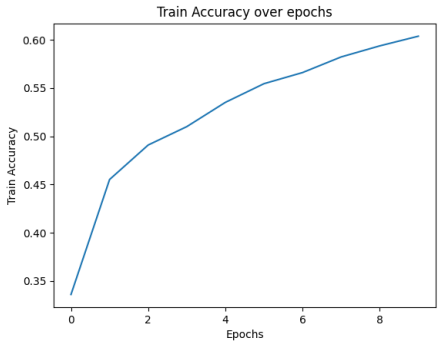

Fig. 13. Training Accuracy over Epochs

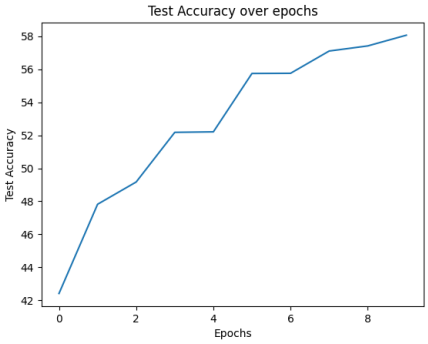

Fig. 14. Test Accuracy over Epochs

```
Test Confusion Matrix
  0%|          | 0/10000 [00:00<?, ?it/s]
[704  27  77  17  15  19  10  11  77  43] (0)
[ 46 640  22  15   6  17  10   7  44 193] (1)
[ 84   7 504  74  62 133  71  33  12  20] (2)
[ 33  12 110 353  30 317  72  32  10  31] (3)
[ 36   6 215  66 356 124  73 100  14  10] (4)
[ 11   5 107 128  23 635  22  48   6  15] (5)
[  9   7  80  88  35  59 679  18   2  23] (6)
[ 26   0  69  46  48 145   7 620   4  35] (7)
[172  41  38  18  11  27   8   7 622  56] (8)
[ 64  91  30  19   5  21  12  20  44 694] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 58.07 %
```

Fig. 15. Test Confusion Matrix

```
Train Confusion Matrix
  0%|          | 0/50000 [00:00<?, ?it/s]
[3562  114  396   89   74   79   32   59  373  222] (0)
[ 291 3355   94   56   38   72   64   46  222  762] (1)
[ 348   50 2716  362  319  577  307  174   70   77] (2)
[ 121   26  599 1900  134 1520  353  142   59  146] (3)
[ 233   27 1036  319 1945  555  310  458   47   70] (4)
[  60   21  461  715  141 3196  107  208   31   60] (5)
[  35   47  471  410  169  276 3398   55   32  107] (6)
[  97   12  308  202  210  674   39 3293   16  149] (7)
[ 745  162  172   86   49   87   38   15 3363  283] (8)
[ 287  474  104   91   23  104   81   94  165 3577] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 60.61 %
```

Fig. 16. Train Confusion Matrix

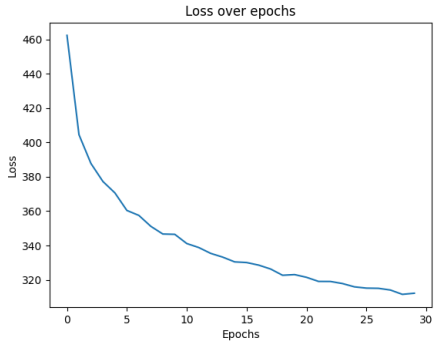

Fig. 17. Training Loss over Epochs for Improved Network

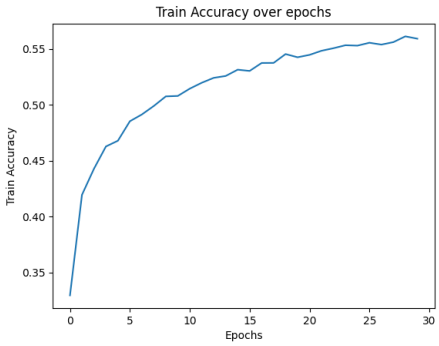

Fig. 18. Training Accuracy over Epochs for Improved Network

Since I have augmented the data using transformations, I increased the epochs to 30. Due to the addition of BatchNorm the parameters of my network increased to 50,916. Adam Optimizer already adjusts learning rate during the training process and thus I did not change my optimizer.

The Training Loss graph can be seen in 17. The Training Accuracy graph can be seen in 18. The Test Accuracy graph can be seen in 19. The test confusion matrix can be seen in Figure 20. The train confusion matrix can be seen in Figure 21. The test accuracy is 43.75 % and the train accuracy is 44.18 %.

C. Comparison

The augmentation steps I added to the training process negatively affected the accuracy of the classification task. I believe the augmentations might be too confusing for the model to understand and maybe simplifying the augmentation steps can benefit the accuracy metric. I also believe a higher learning rate might improve the accuracy as Adam and BatchNorm combined together would allow for a higher learning rate.

I did not implement the last part of the assignment due to lack of time. I would be implementing the Network architecture as soon as I get the time.

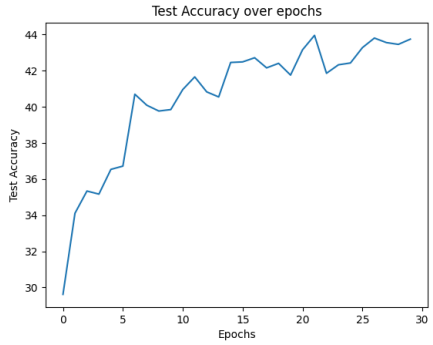

Fig. 19. Test Accuracy over Epochs for Improved Network

```
Test Confusion Matrix
  0%|          | 0/10000 [00:00<?, ?it/s]
[526  30  93  81  29  16   9  31 154  31] (0)
[ 48 451  34  61   9  44  27  41 128 157] (1)
[140  25 261  72 197  94  51  60  86  14] (2)
[ 37  30  86 250  84 228  69  76  84  56] (3)
[ 69  20  81  47 477  44  81 117  37  27] (4)
[ 43  29  50 140  70 456  29  77  71  35] (5)
[ 20  33  36  59  97  58 572  40  45  40] (6)
[ 64  40  42  71 101 103  22 436  43  78] (7)
[164  68  43  70  11  24  14  23 507  76] (8)
[ 44 168  12 109  13  38  27  55  95 439] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 43.75 %
```

Fig. 20. Test Confusion Matrix for Improved Network

IV. Conclusion

I was not at all familiar with the part of Computer Vision pertaining to Phase 1. I enjoyed solving the Phase 1 part as it was new and challenging. The first phase took me a lot of time which left me little time to complete the last part of Phase 2. I look forward to the upcoming assignments in the course.

Acknowledgment

I would like to thank the Course team of Computer Vision to allow me learn hands on various things about filters.


```
Train Confusion Matrix
  0%|          | 0/50000 [00:00<?, ?it/s]
[2654  209  441  328  137  121   37  167  740  166] (0)
[ 264 2319  151  311   48  235  123  210  604  735] (1)
[ 624  107 1516  325  956  395  256  304  403  114] (2)
[ 234  164  375 1210  407 1175  420  330  421  264] (3)
[ 271   73  440  257 2486  230  308  551  201  183] (4)
[ 216  169  338  665  375 2126  186  396  352  177] (5)
[  87  173  220  302  478  249 2792  219  270  210] (6)
[ 281  171  255  321  538  500   77 2240  197  420] (7)
[ 770  393  239  447   82  135   69  107 2439  319] (8)
[ 203  744   86  401   57  259  113  346  480 2311] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 44.186 %
```

Fig. 21. Train Confusion Matrix for Improved Network