RBE549 Computer Vision : Homework-0

USING 4 LATE DAYS

Ajith Kumar Jayamoorthy
Robotics Engineering Department
Worcester Polytechnic Institute
Worcester, MA, U.S.A.
ajayamoorthy@wpi.edu

***Abstract*—This document consist of homework implementation of pb (probability of boundary) boundary detection algorithm and Neural Network models for computer vision applications.**

I. Introduction

In this document we have explained the application and results of two phases. **Phase 1** consist of the implementation of pb (probability of boundary) algorithm which finds boundaries by examining brightness, color, and texture information across multiple scales (different sizes of objects/image). In addition to this it also utilizes the Canny and Sobel baselines to identify the edge.

In **Phase 2**, we work on the implementation of multiple deep learning architectures starting with a CNN network and then further implementation of architectures such as ResNet, ResNeXt and DenseNet on CIFAR10 Dataset. A comparative study of the following architectures are also performed.

II. PHASE 1

In this Phase 1, I will implementing the pb-lite boundary detection algorithm.The objective of this algorithm is improve boundary detection by using texture and color gradient information along with Sobel and Canny filter baselines.

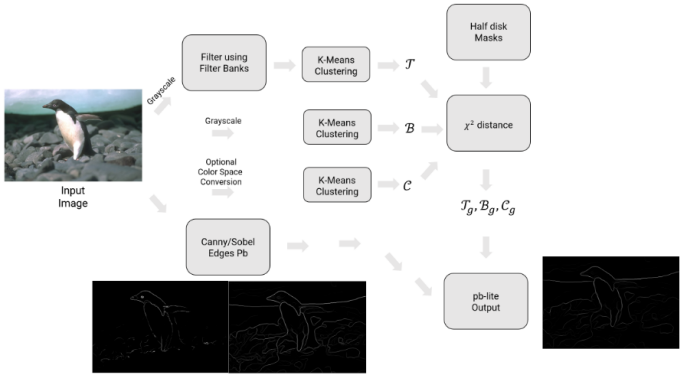

Fig. 1. Pipeline for the pb-lite algorithm [1]

The method consists of following steps:

1) Filter bank generation (Derivative of Gaussians, Leung-Malik filters, Gabor filters, Half-Disk filters)
2) Texton, Brightness, Color maps are implementation.
3) Texture Gradient (Tg), Brightness Gradient (Bg), Color Gradient (Cg) maps are computation
4) Using the above gradient maps along with Sobel and Canny baseline, the pb-lite algorithm is implemented.

A. Oriented Derivative of Gaussian Filter Bank

Derivative of Gaussian filter is a filter created by considering the gradient of the Gaussian Kernel filter. Using the Sobel filter, the Gaussian map is convoluted and the first derivative of the Gaussian map is calculated as how in Figure 2 and Figure 3.

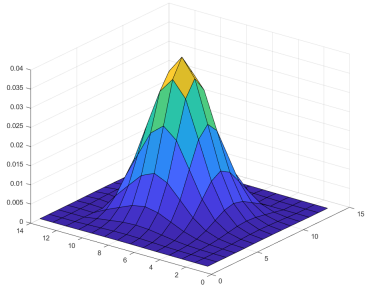

Fig. 2. Visualization of 2D Gaussian Kernel

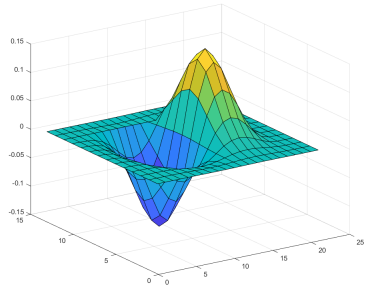

Fig. 3. Visualization of 2D Derivative of Gaussian Kernel using Sobel filter

Further, we create filters with different scales (standard deviation) and orientations and combine them into a filter bank. The Figure 4 is the visualization of the Gaussian Filter Bank.

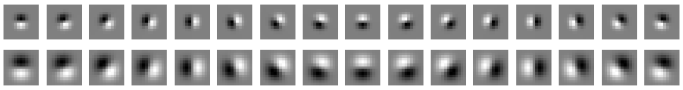

Fig. 4. Visualization of 2D Derivative of Gaussian Kernel using Sobel filter

B. The Leung-Malik Filter Bank

The Leung-Malik filters consists of first and second order derivatives of Gaussian filters (DoG), Laplacian of Gaussian (LoG) filters and Gaussian Filters. We consider two versions of the LM filter bank. In LM Small and LM Large filter bank with the scales σ = 1, $\sqrt{2}$, 2, $2\sqrt{2}$ and σ = $\sqrt{2}$, 2, $2\sqrt{2}$, 4 respectively. The First and second derivative of Gaussian use the first three scales in either case with the following relation of $\sigma_y = 3\sigma_x$. The LoG filers are created using the four basic scales at σ and 3σ resulting in 8 filters.[1] The formula [[2]] for the implementation of Laplace of Gaussian is as follows:

$$LoG(x,y) = -\frac{1}{\pi\sigma^4}\left[1 - \frac{x^2+y^2}{2\sigma^2}\right]e - \frac{x^2+y^2}{2\sigma^2}$$

Lastly, the Gaussian Kernel are developed on the four basic scales. The Figure 5 is the visualization of the Leung-Malik filter bank.

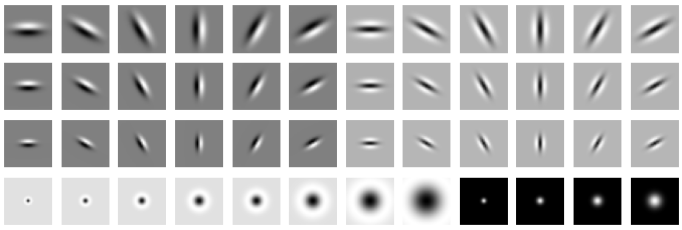

Fig. 5. Visualization of Leung-Malik filter bank

C. Gabor Filter Bank

A Gabor filter is a Gaussian kernel function modulated by a sinusoidal plane wave. The filter bank is created using 5 different scales and 8 different orientations. The Figure 6 shows the Gabor filter bank.

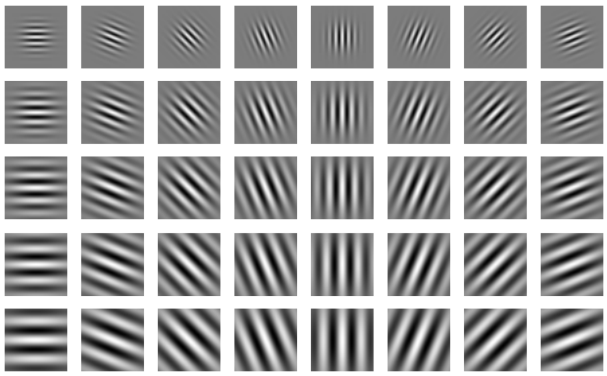

Fig. 6. Visualization of Gabor filter bank

D. Texton, Brightness and Color Map

After the implementation of all the filter banks, three other maps are created. They are the Texton, Brightness and Color Maps. The Texton map corresponds to the textures elements present in the image. The Brightness and Color Maps evaluated the intensity of the pixels in the image and the color pattern in the images respectively. Given below Figure 7 represents the original image.

Fig. 7. Original Image used for the pb-lite implementation

The following Figures 8, Figure 9 and Figure 10 represent the Texton, Brightness and Color Maps respectively.

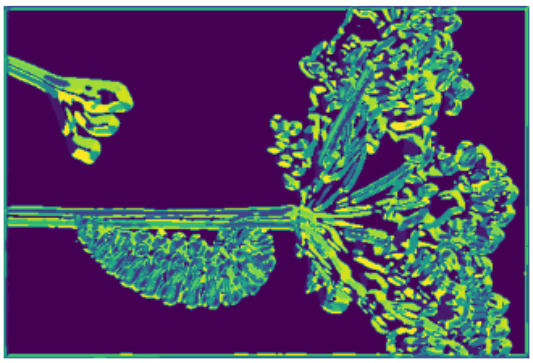

Fig. 8. Visualization of Texton Map

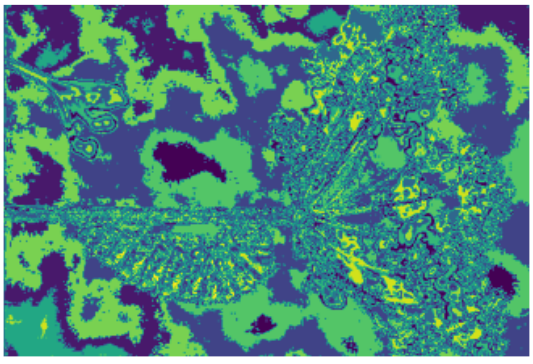

Fig. 9. Visualization of Brightness Map

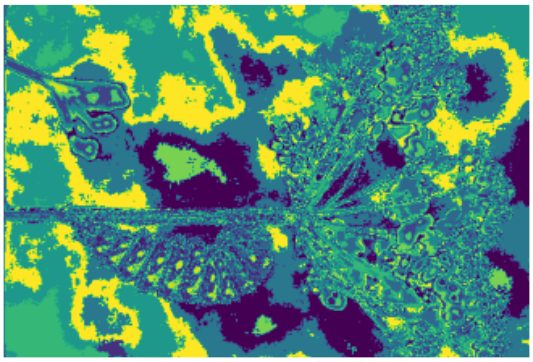

Fig. 10. Visualization of Color Map

E. Texton, Brightness and Color Gradient Map

Following the calculation of the Texton, Brightness and Color maps we start to generate the gradient map for each one of the above. A gradient map depicts the rate of change of texture, Brightness and Color between two pixels in the above maps. The Figures 11 Figure 12 and Figure 13 are Texton gradient, Brightness gradient and Color gradient maps respectively.

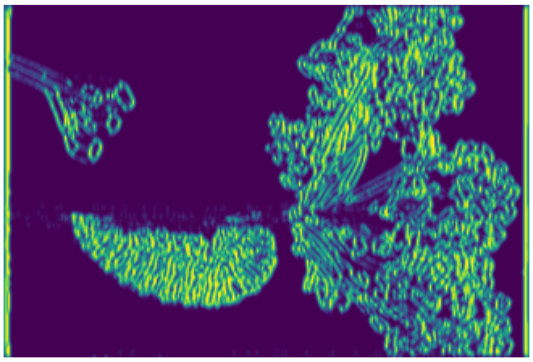

Fig. 11. Visualization of Texton gradient Map

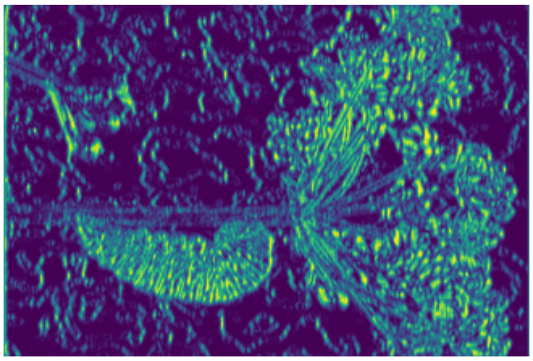

Fig. 12. Visualization of Brightness gradient Map

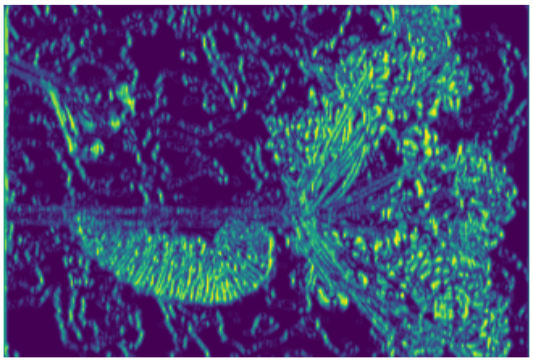

Fig. 13. Visualization of Color gradient Map

F. Pb-lite implementation

The pb-lite algorithm implementation, as mentioned before involves the various filters we have created along with the use of Sobel and Canny baselines. Figure 14 and Figure 15 represent the Sobel and Canny filter baseline outputs respectively.

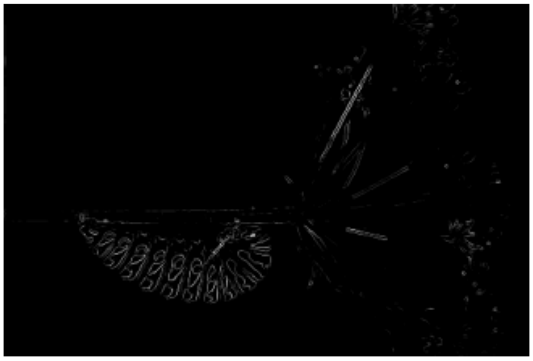

Fig. 14. Visualization of Sobel Baseline

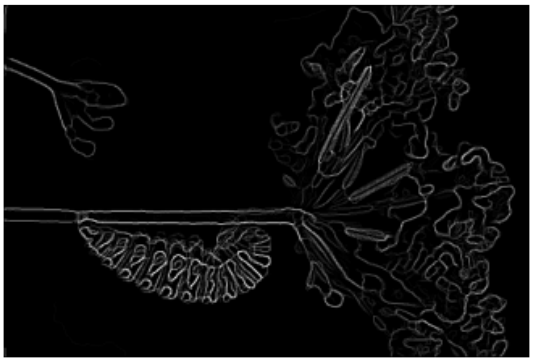

Fig. 15. Visualization of Canny Baseline

After the calculation of all the following outputs, the pl-lite output is calculated according to the follwing equations[[1]]:

$$PbEdges = \frac{\tau_g + \beta_g + C_g}{3} \odot \frac{canny + sobel}{2}$$

The following Figure 16 is the output of the pb-lite boundary detection algorithm.

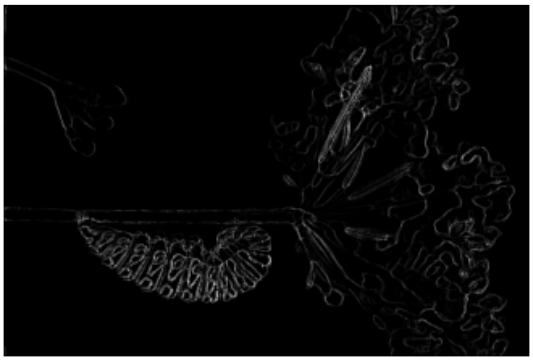

Fig. 16. Visualization of Pb-lite boundary detection algorithm

III. PHASE 2

In the section of the assignment, the implementation of own CNN network has to be done for image classification of CIFAR10 data. The CIFAR10 data is a standard dataset consisting of 10 classes or objects, namely, airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. The dataset was split into two parts: Training and Testing dataset. The Train data set consisted of 50,000 images and the testing set consisted of 10,000 images in total.

The CNN network implemented is as follows:

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 32, 32]	1,792
BatchNorm2d-2	[-1, 64, 32, 32]	128
ReLU-3	[-1, 64, 32, 32]	0
Conv2d-4	[-1, 64, 32, 32]	36,928
BatchNorm2d-5	[-1, 64, 32, 32]	128
ReLU-6	[-1, 64, 32, 32]	0
MaxPool2d-7	[-1, 64, 16, 16]	0
Conv2d-8	[-1, 128, 16, 16]	73,856
BatchNorm2d-9	[-1, 128, 16, 16]	256
ReLU-10	[-1, 128, 16, 16]	0
Conv2d-11	[-1, 128, 16, 16]	147,584
BatchNorm2d-12	[-1, 128, 16, 16]	256
ReLU-13	[-1, 128, 16, 16]	0
MaxPool2d-14	[-1, 128, 8, 8]	0
Linear-15	[-1, 4096]	33,558,528
ReLU-16	[-1, 4096]	0
Linear-17	[-1, 10]	40,970

Fig. 17. CNN model architecture for phase 2

A. Results

The following are the results from the basic implementation of own CNN architecture. The Figure 18 show the Optimizer parameters for the initial run of the model. Figure 19 shows the Training Accuracy with iteration and Figure 20 shows the Training Loss-per-iteration. The Figure 21 shows the Test accuracy with Iteration and finally, Figure 22 shows the confusion matrix for the corresponding model.

```
Number of Epochs Training will run for 50
Factor of reduction in training data is 1.0
Mini Batch Size 500
Number of Training Images 50000
Optimizer Information:
 <bound method Optimizer.state_dict of SGD (
Parameter Group 0
    dampening: 0
    foreach: None
    lr: 0.01
    maximize: False
    momentum: 0
    nesterov: False
    weight_decay: 0
)>
New model initialized....
```

Fig. 18. Optimizer parameters and Data set description

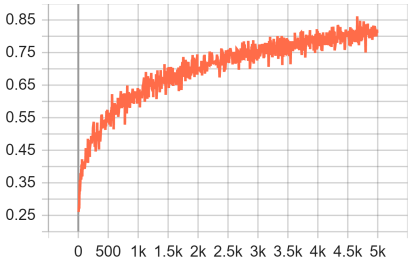

Fig. 19. Training Accuracy-vs-iteration plot

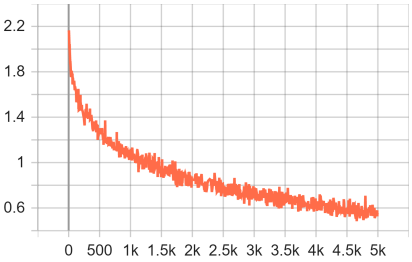

Fig. 20. Training Loss-vs-iteration plot

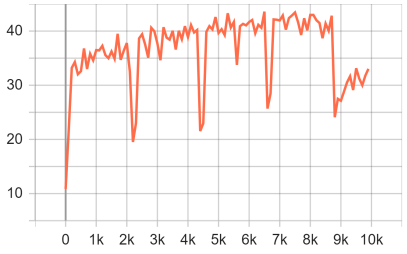

Fig. 21. Test Accuracy-vs-iteration plot

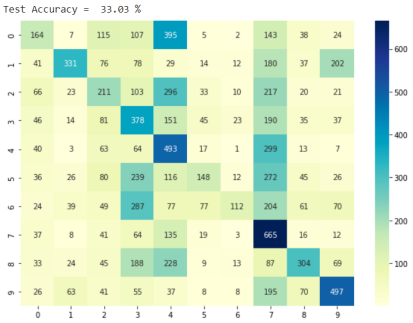

Fig. 22. Confusion Matrix and Final model accuracy

The following are the results from the basic implementation of the tuned CNN architecture. The Figure 23 show the Optimizer parameters for the final tuned run of the model. Figure 24 shows the Training Accuracy with iteration and Figure 25 shows the Training Loss-per-iteration. The Figure 26 shows the Test accuracy with Iteration and finally, Figure 27 shows the confusion matrix for the corresponding model.

```
Files already downloaded and verified
Number of Epochs Training will run for 50
Factor of reduction in training data is 1.0
Mini Batch Size 100
Number of Training Images 50000
Optimizer Information:
 <bound method Optimizer.state_dict of SGD (
Parameter Group 0
    dampening: 0
    foreach: None
    lr: 0.01
    maximize: False
    momentum: 0
    nesterov: False
    weight_decay: 0
)>
```

Fig. 23. Optimizer parameters and Data set description

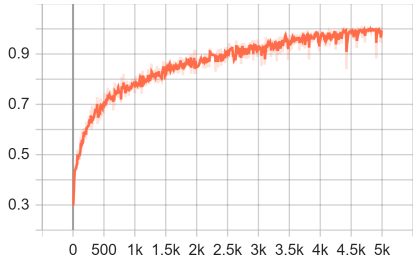

Fig. 24. Training Accuracy-vs-iteration plot

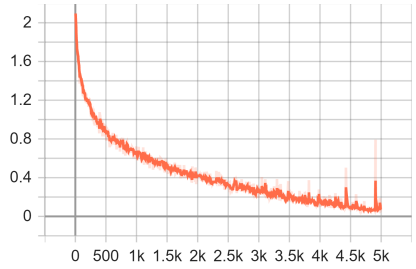

Fig. 25. Training Loss-vs-iteration plot

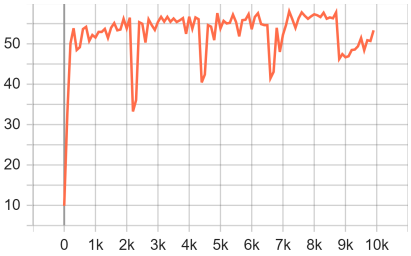

Fig. 26. Test Accuracy-vs-iteration plot

The models performance has increased from 33.03% to 53.35%. The main changes in the model made were Data

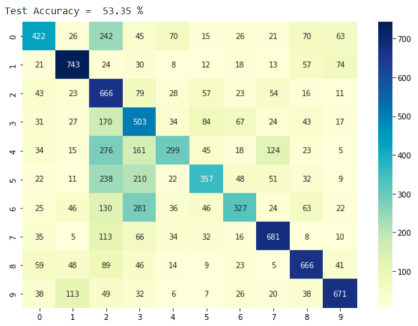

Fig. 27. Confusion Matrix and Final model accuracy

Augmentation and reducing the mini-batch size from 500 samples per batch to 100 samples per batch.

REFERENCES

[1] https://rbe549.github.io/spring2022/hw/hw0/
[2] https://academic.mu.edu/phys/matthysd/web226/Lab02.htm
[3] https://www.robots.ox.ac.uk/v̄gg/research/texclass/code/makeLMfilters.m
[4] https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
[5] Towards Data Science (stochastic-gradient-descent-with-momentum)
[6] https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html