Computer Vision : Homework 0

Swati Shirke
Email: svshirke@wpi.edu

***Abstract*—This project Homework0, Alohomora has two Phases. Phase1 is to implement edge detection using pb(probability of boundary) method, in which edge detection is performed using change in texture, colour and brightness components across multiple scales and orientations. The algorithm performed pretty well in edge detection, however, it is detecting edges of background objects as well. Phase2 is to desing and implement Convolution Neural Network, ResNet, ResNext and DenseNet using CIFAR-10 image dataset.**

I. Phase 1: Shake My Boundary

This phase involves series of operations to be performed in order to achieve edge detection on input images. This involves creating 3 different filter banks, namely Derivative of Gaussian (DoG), Leung-Malik filters (LM filters) and Gabor Filters. Filtering operation is performed in input images and further, K-Means clustering is applied in order to generate texton map , brightness and colour maps, which encode how much the texture, brightness and color distributions are changing at a pixel respectively. Subsequently, we are comparing texton, brightness and color distributions with the chi-square (x^2) measure. The final step is to combine information from the features with a baseline method (based on Sobel or Canny edge detection or an average of both).

A. Designing Filter Banks

A filter bank is a set of band-pass filters designed to perform operations such blurring, noise removal on input image so as to achieve feature extractions. In this case, we are using 3 different filter banks, details are as follows.

1) Oriented DoG Bank: It is a collection of DoG filters with multiple orienations and scales. Here, we have used 16 orientations between 0 to 360 and 2 scales [2,3]. Derivation of Gaussian is implemented by convolving Gaussian filter with Sobel Kernel. In total 32 filters are there in this bank.

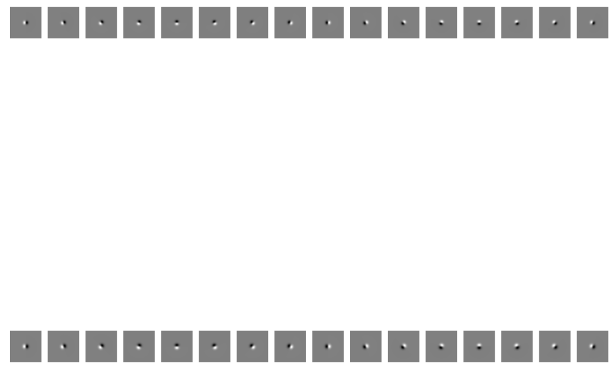


Fig. 1. DOG filter

2) Leung-Malik filters (LM filters): This filter bank consists of total 48 filters which involves first and second order derivatives of Gaussians at 6 orientations and 3 scales making a total of 36; 8 Laplacian of Gaussian (LOG) filters; and 4 Gaussians filters. Further, we created 2 LM banks, LM-small with $\sigma = [1, \sqrt{2}, 2, 2\sqrt{2}]$ and LM-large $\sigma = [\sqrt{2}, 2, 2\sqrt{2}, 4]$.The Gaussians occur at the four basic scales while the 8 LOG filters occur at σ and 3σ. Kernel size we used is 49.

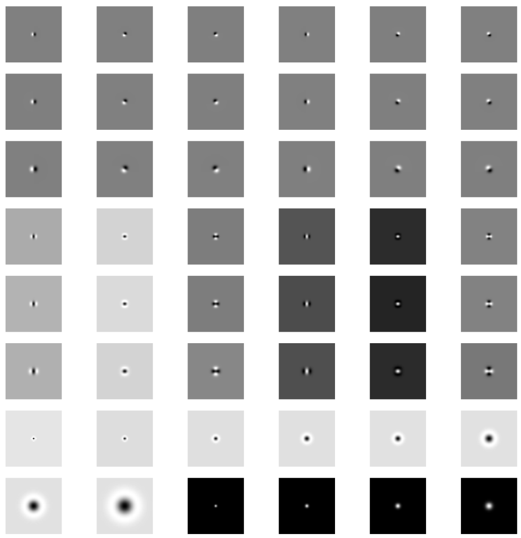


Fig. 2. LM-small filter bank

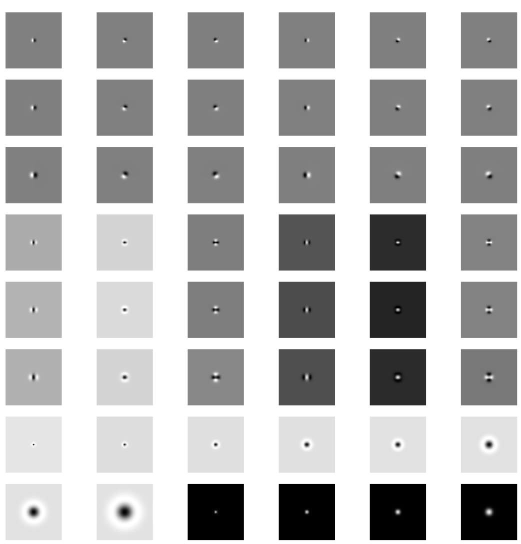


Fig. 3. LM-large filter bank

3) Gabor Filters: Gabor Filters are designed based on the filters in the human visual system. A gabor filter is a gaussian kernel function modulated by a sinusoidal plane wave. The equation for a 2D Gabor filter in the spatial domain is given by:

$$g(x,y) = e^{-\frac{x'^2+\gamma^2 y'^2}{2\sigma^2}} \cdot \cos(2\pi f_0 x' + \phi) \tag{1}$$

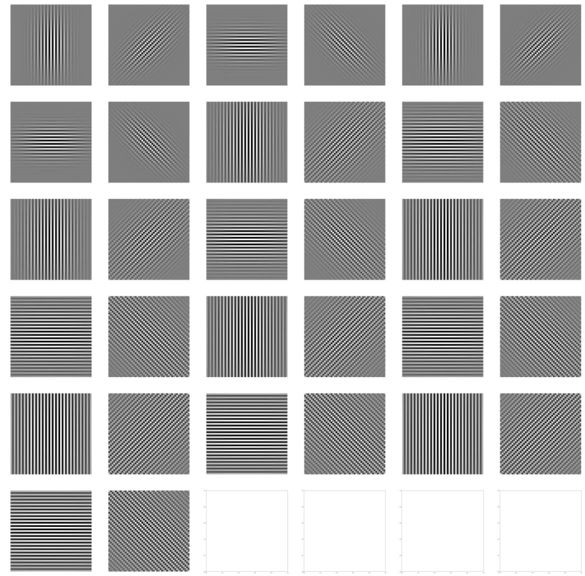


Fig. 4. Gabor filter bank

B. Texton map

Applying a filter bank to an input image generates a vector of filter responses for each pixel, centered on that pixel. The distribution of these N-dimensional filter responses can be considered as encoding texture properties. To simplify this representation, the N-dimensional vector is substituted with a texton ID. This simplification is accomplished by employing the k-means clustering algorithm from Scikit-learn to cluster filter responses at all pixels in the image into K textons.

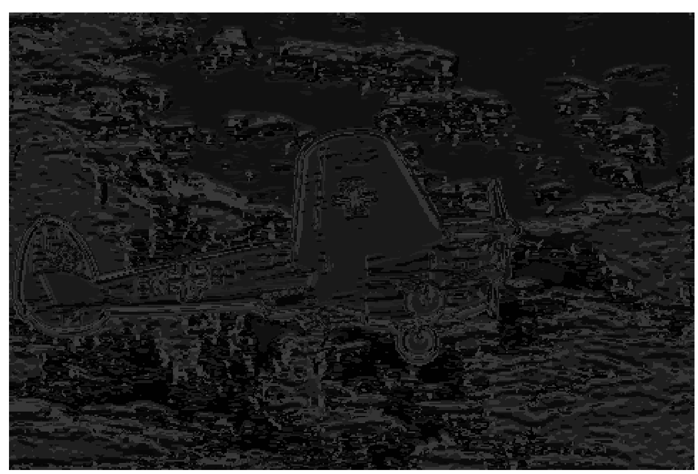


Fig. 5. Texton map

C. Brightness map

The idea behind the brightness map is straightforward—it involves capturing the variations in brightness within the image. Once again, we employ k-means clustering on the brightness values (equivalent to grayscale representation of the color image), grouping them into a specified number of clusters (optimal at 16 clusters, but feel free to explore other values). The resulting clustered output is referred to as the brightness map, denoted as B.

Fig. 6. Brightness map

D. Colour map

The idea behind the color map is to encapsulate the variations in color or chrominance within the image. Once again, we utilize k-means clustering on the color values (assuming three values per pixel for RGB color channels), and you have the flexibility to explore alternative color spaces like YCbCr, HSV, or Lab. The color values are grouped into a specified number of clusters (16 clusters have proven effective, but experimentation is encouraged). The resulting grouped output is termed the color map, denoted as C. It's worth noting that you have the option to cluster each color channel separately, providing room for experimentation with different methods.

Fig. 7. Colour map

E. Texture, Brightness and Color Gradients Tg ,Bg ,Cg

To calculate Tg, Bg, and Cg, it is necessary to compute variations in values across various shapes and sizes. This process can be efficiently accomplished by utilizing Half-disc masks.

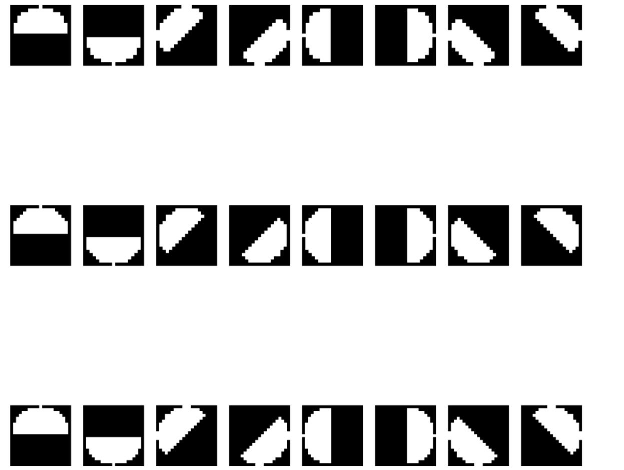


Fig. 8. Half Disk Mask

F. Sobel and Canny baseline

The output images obtained from the Sobel and Canny operations serve as the baselines. Our Pb-lite algorithm is then compared to these baseline results.

G. Pb-Lite Algorithm

The performance of the Pb-lite algorithm in edge detection is evidently strong. From a semantic perspective, Pb-lite demonstrates the ability to detect edges that are meaningful.

H. Results

Sample input image is shown

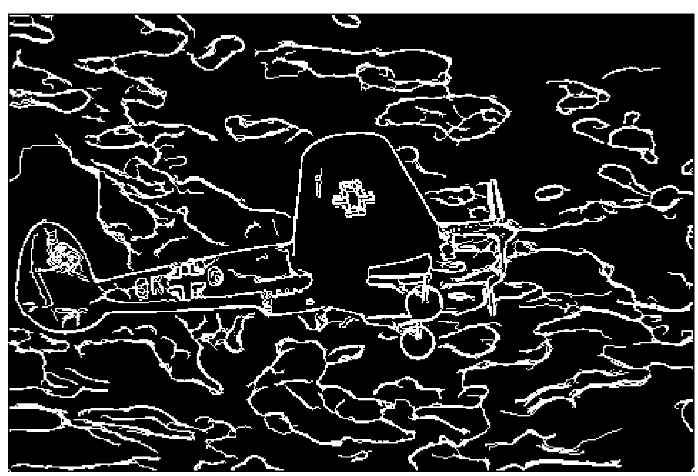


Fig. 10. Result of Edge detection

Fig. 9. Input image

II. PHASE2: DEEP DIVE ON DEEP LEARNING

In addressing this problem, a basic convolutional neural network (convnet) is constructed using PyTorch. The model is trained and tested using images sourced from the CIFAR-10 dataset. Subsequently, for the later stages of this problem, I am implementing 2 additional neural network models: ResNet ResNeXt.

III. DATA SET

The CIFAR-10 dataset comprises 60,000 32x32 images distributed among 10 distinct classes, with 6,000 images per class. Among these, 50,000 images are designated for training, while the remaining 10,000 are allocated for testing. The dataset demonstrates a balanced distribution across the various classes.

A. Model the first neural network

As part of this section, I have build a 5 layer Convolution Neural Network (CNN). First 2 layers are convolution layers and remaining are linear layers. Its architecture is shown in the figure 9. Total number of learnable parameters are 62,006.

```
class CIFAR10Model(ImageClassificationBase):
    def __init__(self, InputSize, OutputSize):
        super().__init__()


        ############################
        self.conv1 = nn.Conv2d(3,6,5)
        self.pool = nn.MaxPool2d(2,2)
        self.conv2 = nn.Conv2d(6,16,5)

        self.fc1 = nn.Linear(16*5*5, 120)
        self.fc2 = nn.Linear(120,84)
        self.fc3 = nn.Linear(84,OutputSize)

    def forward(self, xb):

        out = self.pool(F.relu(self.conv1(xb)))
      #   print(out.shape)
        out = self.pool(F.relu(self.conv2(out)))
      #   print(out.shape)
        out = out.view(-1, 16*5*5)
      #   print(out.shape)
        out = F.relu(self.fc1(out))
        out = F.relu(self.fc2(out))
        out = self.fc3(out)

        return out
```

Fig. 11. CNN network

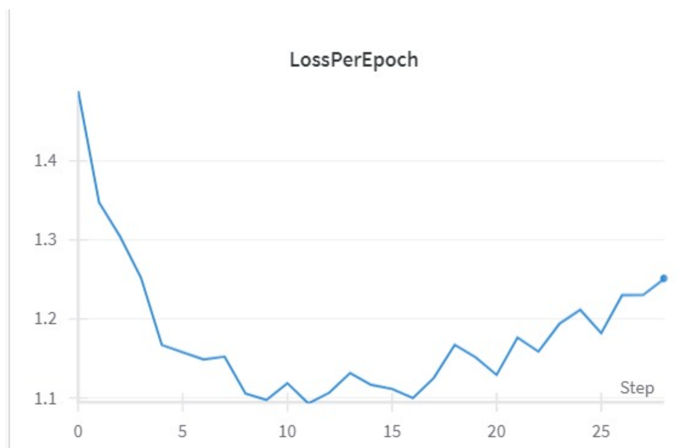
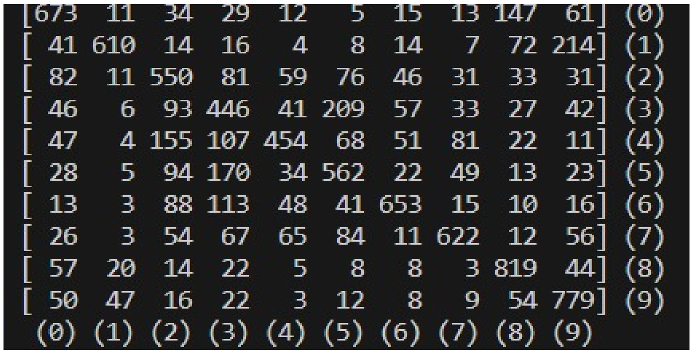


Fig. 12. Training loss of CNN

3859	35	178	61	45	27	24	23	534	214	(0)
139	3591	30	56	6	19	27	9	237	886	(1)
342	18	3450	326	195	258	164	63	103	81	(2)
138	13	391	2849	141	898	217	83	111	159	(3)
183	4	661	364	2954	259	174	273	68	60	(4)
87	8	306	812	104	3305	81	139	36	122	(5)
39	17	386	464	113	166	3685	17	43	70	(6)
74	6	228	282	203	319	30	3658	45	155	(7)
196	39	49	32	12	10	21	4	4531	106	(8)
111	106	33	49	5	22	16	19	156	4483	(9)

Fig. 13. Training Confusion matrix of CNN

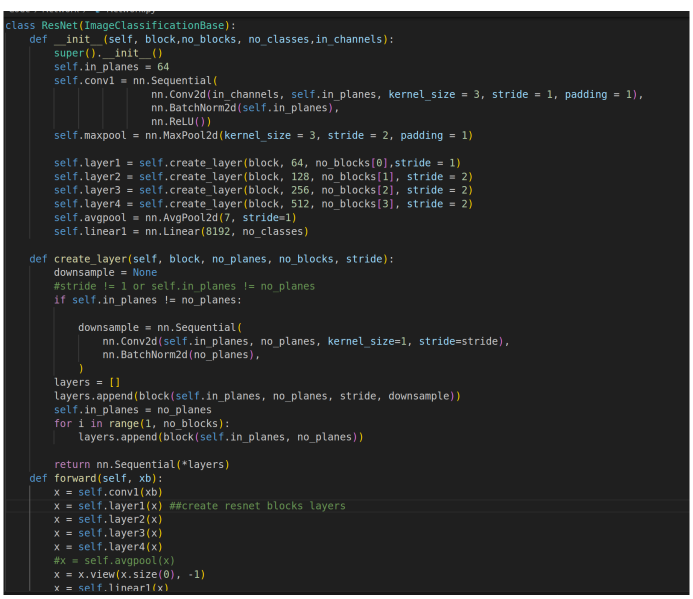


	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
(0)	673	11	34	29	12	5	15	13	147	61
(1)	41	610	14	16	4	8	14	7	72	214
(2)	82	11	550	81	59	76	46	31	33	31
(3)	46	6	93	446	41	209	57	33	27	42
(4)	47	4	155	107	454	68	51	81	22	11
(5)	28	5	94	170	34	562	22	49	13	23
(6)	13	3	88	113	48	41	653	15	10	16
(7)	26	3	54	67	65	84	11	622	12	56
(8)	57	20	14	22	5	8	8	3	819	44
(9)	50	47	16	22	3	12	8	9	54	779

Fig. 14. Testing Confusion matrix of CNN

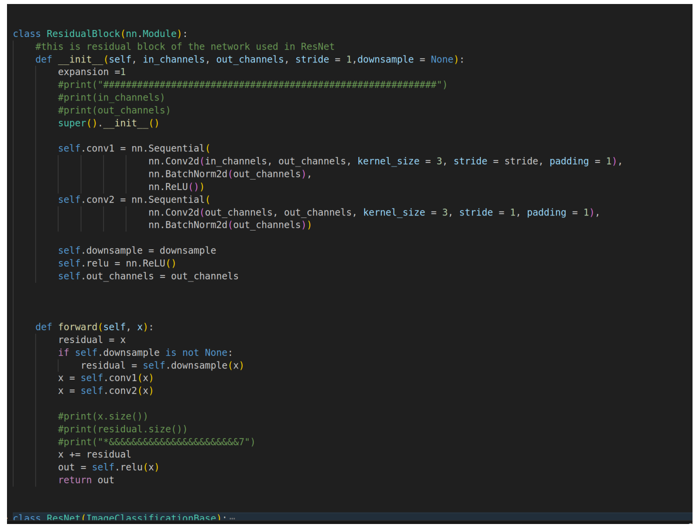
B. Designing ResNet

As a part of this, I have built a ResNet with architecture as shown in figure 13. It has total 8 block, and 4 skip 8 skip connections. Total number of learnable parameters are 11,255,562.



```
class ResNet(ImageClassificationBase):
    def __init__(self, block,no_blocks, no_classes,in_channels):
        super().__init__()
        self.in_planes = 64
        self.conv1 = nn.Sequential(
                        nn.Conv2d(in_channels, self.in_planes, kernel_size = 3, stride = 1, padding = 1),
                        nn.BatchNorm2d(self.in_planes),
                        nn.ReLU())
        self.maxpool = nn.MaxPool2d(kernel_size = 3, stride = 2, padding = 1)

        self.layer1 = self.create_layer(block, 64, no_blocks[0],stride = 1)
        self.layer2 = self.create_layer(block, 128, no_blocks[1], stride = 2)
        self.layer3 = self.create_layer(block, 256, no_blocks[2], stride = 2)
        self.layer4 = self.create_layer(block, 512, no_blocks[3], stride = 2)
        self.avgpool = nn.AvgPool2d(7, stride=1)
        self.linear1 = nn.Linear(8192, no_classes)

    def create_layer(self, block, no_planes, no_blocks, stride):
        downsample = None
        #stride != 1 or self.in_planes != no_planes
        if self.in_planes != no_planes:

            downsample = nn.Sequential(
                nn.Conv2d(self.in_planes, no_planes, kernel_size=1, stride=stride),
                nn.BatchNorm2d(no_planes),
            )
        layers = []
        layers.append(block(self.in_planes, no_planes, stride, downsample))
        self.in_planes = no_planes
        for i in range(1, no_blocks):
            layers.append(block(self.in_planes, no_planes))

        return nn.Sequential(*layers)
    def forward(self, xb):
        x = self.conv1(xb)
        x = self.layer1(x) ##create resnet blocks layers
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        #x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.linear1(x)
```

Fig. 15. ResNet architecture



```
class ResidualBlock(nn.Module):
    #this is residual block of the network used in ResNet
    def __init__(self, in_channels, out_channels, stride = 1,downsample = None):
        expansion =1
        #print("############################################################")
        #print(in_channels)
        #print(out_channels)
        super().__init__()

        self.conv1 = nn.Sequential(
                        nn.Conv2d(in_channels, out_channels, kernel_size = 3, stride = stride, padding = 1),
                        nn.BatchNorm2d(out_channels),
                        nn.ReLU())
        self.conv2 = nn.Sequential(
                        nn.Conv2d(out_channels, out_channels, kernel_size = 3, stride = 1, padding = 1),
                        nn.BatchNorm2d(out_channels))

        self.downsample = downsample
        self.relu = nn.ReLU()
        self.out_channels = out_channels


    def forward(self, x):
        residual = x
        if self.downsample is not None:
            residual = self.downsample(x)
        x = self.conv1(x)
        x = self.conv2(x)

        #print(x.size())
        #print(residual.size())
        #print("*&&&&&&&&&&&&&&&&&&&&&&&&7")
        x += residual
        out = self.relu(x)
        return out

class ResNet(ImageClassificationBase):…
```

Fig. 16. ResNet Residual block

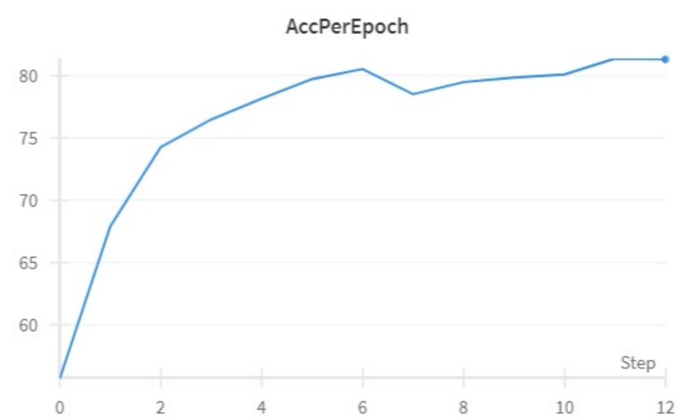


Fig. 17. ResNet Testing Accuracy

Fig. 18. ResNet Testing Loss

802	8	31	21	21	4	9	7	73	24	(0)
16	863	2	3	0	5	6	3	30	72	(1)
43	1	690	56	86	53	43	13	12	3	(2)
12	1	40	655	65	156	36	21	9	5	(3)
8	1	26	30	861	25	20	20	8	1	(4)
4	2	26	128	40	757	9	27	4	3	(5)
6	0	23	40	34	36	845	6	9	1	(6)
10	0	17	37	59	51	5	813	4	4	(7)
23	5	8	11	4	2	6	2	924	15	(8)
8	21	5	8	4	4	3	5	20	922	(9)

Fig. 19. ResNet Testing Confusion Matrix

4821	18	51	19	14	2	22	23	6	24	(0)
37	4871	6	7	0	1	0	2	1	75	(1)
106	4	4668	79	34	40	36	31	1	1	(2)
36	10	145	4491	15	156	97	36	2	12	(3)
24	0	414	119	4072	128	115	125	2	1	(4)
5	2	99	332	7	4487	22	42	0	4	(5)
24	10	139	39	4	21	4756	4	1	2	(6)
15	1	46	84	11	62	4	4773	0	4	(7)
415	101	16	16	12	3	12	7	4281	137	(8)
22	46	5	8	1	2	1	3	0	4912	(9)
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	

Fig. 20. ResNet Training Confusion Matrix

C. Designing ResNext

As a part of this, I have built a ResNext with architecture as shown in figure 19 It has total 12 blocks and cardinality is 32. Total number of learnable parameters are 37,052,554.

```
class ResNextBlock(nn.Module):
    def __init__(self, in_channels, cardinality, bottleneck_width, downsample=None, stride = 1 ):
        super().__init__()
        self.expansion =2
        out_channels = cardinality * bottleneck_width
        self.conv1 = nn.Sequential(
                        nn.Conv2d(in_channels, out_channels, kernel_size = 1, stride = 1, padding = 0),
                        nn.BatchNorm2d(out_channels),
                        nn.ReLU())
        self.conv2 = nn.Sequential(
                        nn.Conv2d(out_channels, out_channels, kernel_size = 3,groups=cardinality, stride = stride, paddi
                        nn.BatchNorm2d(out_channels),
                        nn.ReLU())

        self.conv3 = nn.Sequential(
                        nn.Conv2d(out_channels, out_channels*self.expansion, kernel_size = 1, stride = 1, padding = 0),
                        nn.BatchNorm2d(out_channels * self.expansion))

        self.downsample = downsample
        self.relu = nn.ReLU()

    def forward(self, x):
        residual = x
        #print(x.size())
        if self.downsample is not None:
            residual = self.downsample(x)
            #print("Here")
        #print(residual.size())
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        #print(x.size())
        x += residual
        out = self.relu(x)
        return out
```

Fig. 21. ResNext architecture

```
class ResNet(ImageClassificationBase):⋯

class ResNextBlock(nn.Module):
    def __init__(self, in_channels, cardinality, bottleneck_width, downsample=None, stride = 1 ):
        super().__init__()
        self.expansion =2
        out_channels = cardinality * bottleneck_width
        self.conv1 = nn.Sequential(
                        nn.Conv2d(in_channels, out_channels, kernel_size = 1, stride = 1, padding = 0),
                        nn.BatchNorm2d(out_channels),
                        nn.ReLU())
        self.conv2 = nn.Sequential(
                        nn.Conv2d(out_channels, out_channels, kernel_size = 3,groups=cardinality, stride = stride, paddi
                        nn.BatchNorm2d(out_channels),
                        nn.ReLU())

        self.conv3 = nn.Sequential(
                        nn.Conv2d(out_channels, out_channels*self.expansion, kernel_size = 1, stride = 1, padding = 0),
                        nn.BatchNorm2d(out_channels * self.expansion))

        self.downsample = downsample
        self.relu = nn.ReLU()

    def forward(self, x):
        residual = x
        #print(x.size())
        if self.downsample is not None:
            residual = self.downsample(x)
            #print("Here")
        #print(residual.size())
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        #print(x.size())
        x += residual
        out = self.relu(x)
        return out
```

Fig. 22. ResNext Residual block

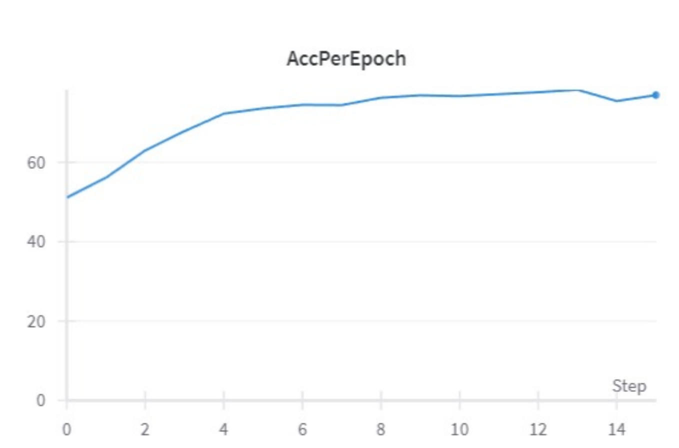


Fig. 23. ResNext Testing Accuracy

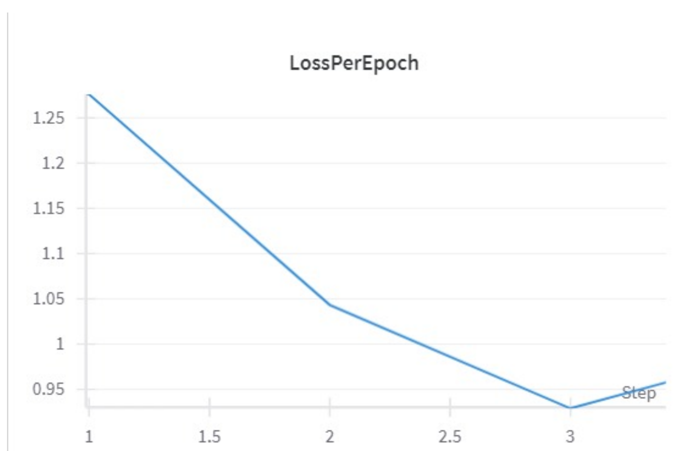
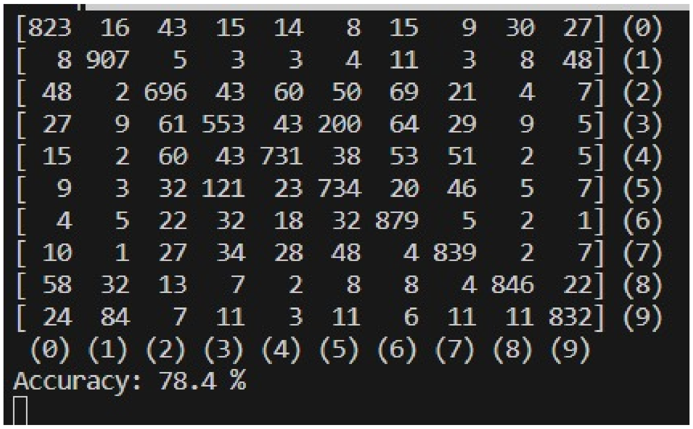


Fig. 24. ResNext Testing Loss



```
[823  16  43  15  14   8  15   9  30  27] (0)
[  8 907   5   3   3   4  11   3   8  48] (1)
[ 48   2 696  43  60  50  69  21   4   7] (2)
[ 27   9  61 553  43 200  64  29   9   5] (3)
[ 15   2  60  43 731  38  53  51   2   5] (4)
[  9   3  32 121  23 734  20  46   5   7] (5)
[  4   5  22  32  18  32 879   5   2   1] (6)
[ 10   1  27  34  28  48   4 839   2   7] (7)
[ 58  32  13   7   2   8   8   4 846  22] (8)
[ 24  84   7  11   3  11   6  11  11 832] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 78.4 %
```

Fig. 25. ResNext Testing Confusion Matrix

mds
August 26, 2015

D. Subsection Heading Here

Subsection text here.

References

[1] https://www.robots.ox.ac.uk/ vgg/research/texclass/filters.html
[2] https://hannibunny.github.io/orbook/preprocessing/04gaussianDerivatives.html
[3] https://medium.com/@anuj$_s hah/through - the - eyes - of - gabor - filter - 17d1fdb3ac97https : //cs.brown.edu/courses/csci1430/2011/results/proj2/schangpi/$