RBE/CS 549 Computer Vision

HW0 - Alohomora

Uthiralakshmi Sivaraman
Robotics Engineering Department
Worcester Polytechnic Institute
Worcester, MA, USA
usivaraman@wpi.edu

Using 1 late day

***Abstract*—The assignment consists of two parts: A)"Shake my Boundary" where we use a probability based edge detection by calculating Texture, Brightness and Color Map and gradients along with Sobel and Canny Baselines B) "Deep Dive on Deep Learning " where we compare multiple deep learning architectures to classify objects from CIFAR10 BSDS500 dataset.**

***Index Terms*—Edge Detection, Sobel, Canny, CIFAR10, BSDS500, ResNet, DenseNet, ResNeXt**

I. PHASE 1 : SHAKE MY BOUNDARY

Boundary detection is an interesting problem statement. Given an image, we find the boundary based on how one object transitions to another. Although boundary detection seems straightforward for human being, it is difficult to achieve boundary or edge detection from single image. Most of the existing techniques use just intensities variations in the image to obtain edges.

In this assignment, we use a probability based edge detection which consists of three different parameters:texture, brightness as well as color variations to detect boundaries along with three different filters: Oriented Derivative of Gaussian, Leung-Malik (LM), Gabor Filter-banks.

A. Oriented Derivative of Gaussian Filter Bank

We obtain the Oriented DOG Filter, Convolution of a Sobel filter over a Gaussian kernel, rotating the kernel with 2 different scales and 16 orientations.

Equation of a Gaussian operator :
$g(x, y) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/(2\sigma^2)}$

B. Leung-Malik (LM) Filter Bank

Leung-malik filter-banks are formed by multi-scale, multi-orientation filter-bank consisting 48 different filters. There are three different types of Leung-malik filters. In first type of filters, first and second derivative filters occur at the first 3 scales with an elongation factor of 3, i.e. $sigma_x = sigma$ $sigma_y = 3 * sigma_x$. In second type of filter, Leung-malik small filters occurs at basic scales, sigma =$1, \sqrt{2}, 2, 2\sqrt{2}$. In third type of filter, Leung-malik large filters occurs at basic scales, sigma = $\sqrt{2}, 2, 2\sqrt{2}, 4$.

Leung-Malik filters are obtained by combining 4 different combinations of filters: 1) First Derivative of Gaussian Filter 2) Second Derivative of Gaussian Filter 3) Laplacian of Gaussian Filter 4) Gaussian Filter

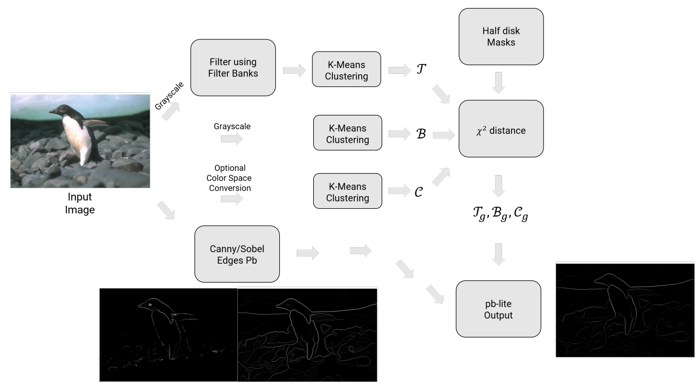

Fig. 1. PbLite Edge Detection

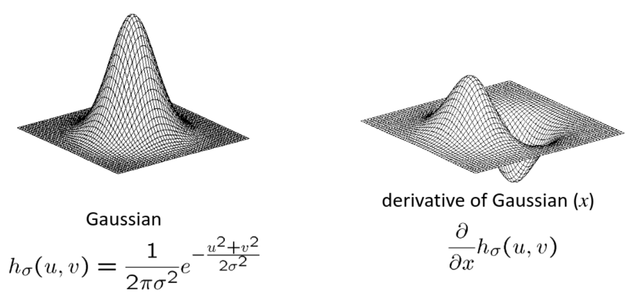

Fig. 2. Gaussian Filter and its derivative

C. Gabor Filter Bank

Gabor filters mostly occur in the human visual system. Gaussian kernel function modulated by a sinusoidal plane wave. It analyses whether there is any specific frequency change.

D. Texton Map, Brightness Map, Color Map

1) Texton Map: We find Texton Map by capturing the texture changes in the image and cluster the texture variations

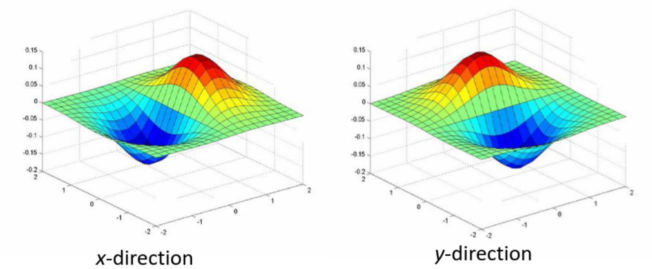

Fig. 3. Derivative of Gaussian in 2 Dimensions

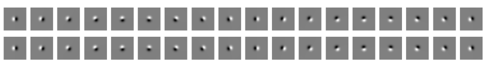

Fig. 4. Oriented DOG Filter-bank

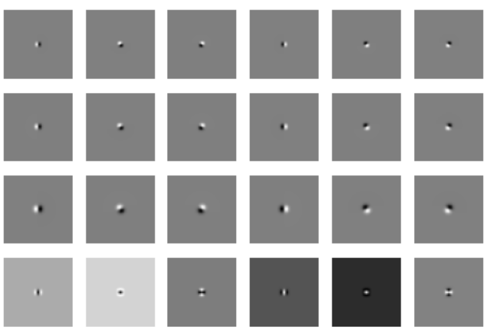

Fig. 5. Leung-Malik Small Filter-bank

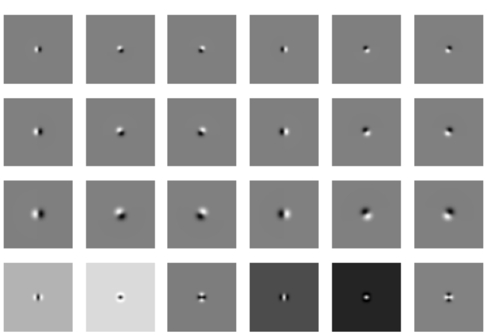

Fig. 6. Leung-Malik Large Filter-bank

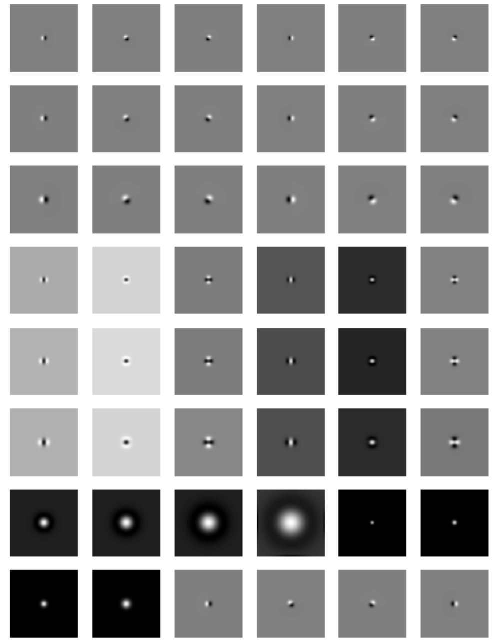

Fig. 7. Leung-Malik Filter-bank

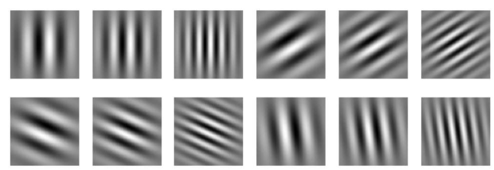

Fig. 8. Gabor Filter-bank

with an N-dimensional vector for clustering all the responses at all pixels in the image for K textons using Kmeans.

2) Brightness Map: We find Brightness Map by capturing the brightness change in the image and cluster the brightness values for gray-scale equivalent of a color image using Kmeans clustering by choosing a set of cluster bins.

3) Color Map: We find Color Map by capturing color changes or chrominance content in the image and cluster the color values (3 values per pixel (RGB color channels)) using Kmeans clustering by choosing a set of cluster bins.

E. Half Disc Masks

Half Disc Masks refer to pairs of binary images of Half-Discs using equation of circles constraining either x and y or both within a particular range and variation of angles.

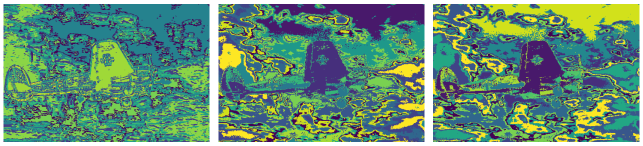

Fig. 9. Image 1 (a) Texton Map (b)Brightness Map (c) Color Map

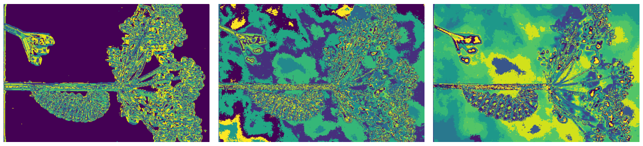

Fig. 10. Image 2 (a) Texton Map (b) Brightness Map (c) Color Map

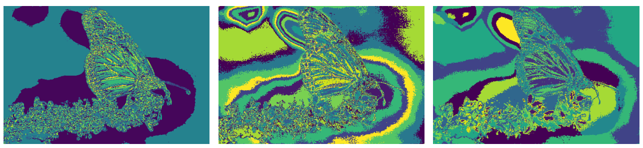

Fig. 11. Image 3 (a) Texton Map (b) Brightness Map (c) Color Map

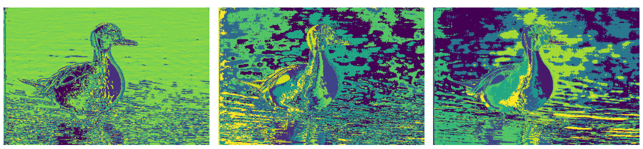

Fig. 12. Image 4 (a) Texton Map (b) Brightness Map (c) Color Map

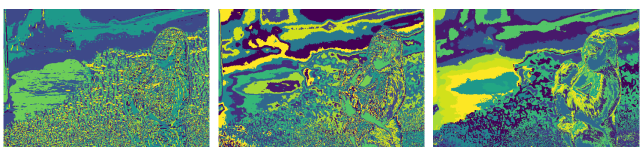

Fig. 13. Image 5 (a) Texton Map (b) Brightness Map (c) Color Map

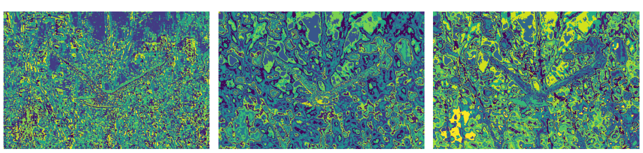

Fig. 14. Image 6 (a) Texton Map (b) Brightness Map (c) Color Map

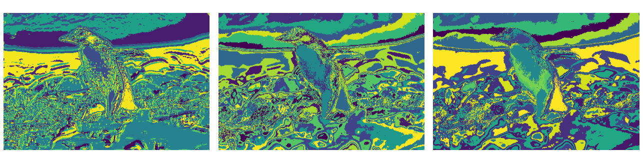

Fig. 15. Image 7 (a) Texton Map (b) Brightness Map (c) Color Map

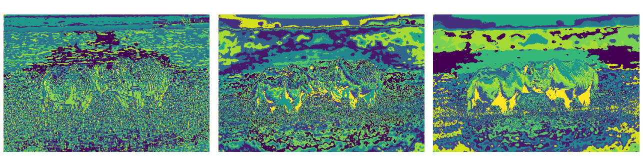

Fig. 16. Image 8 (a) Texton Map (b) Brightness Map (c) Color Map

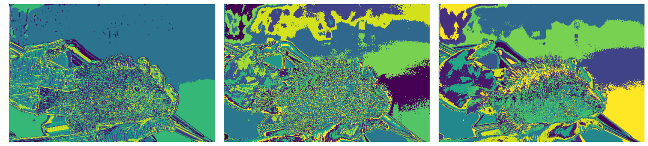

Fig. 17. Image 9 (a) Texton Map (b) Brightness Map (c) Color Map

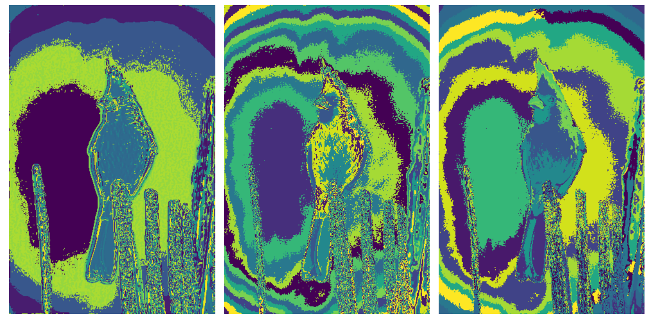

Fig. 18. Image 10 (a) Texton Map (b) Brightness Map (c) Color Map

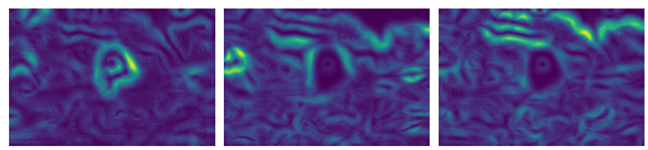

Fig. 19. Image 1 (a) Texton Gradient (b) Brightness Gradient (c) Color Gradient

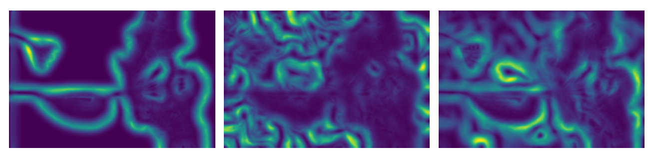

Fig. 20. Image 2 (a) Texton Gradient (b)Brightness Gradient (c) Color Gradient

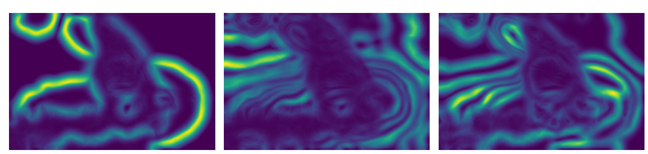

Fig. 21. Image 3 (a) Texton Gradient (b) Brightness Gradient (c) Color Gradient

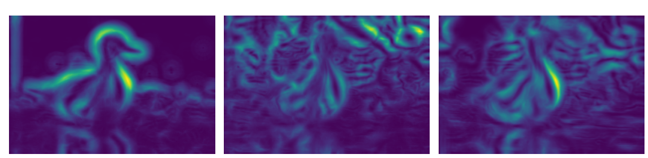

Fig. 22. Image 4 (a) Texton Gradient (b) Brightness Gradient (c) Color Gradient

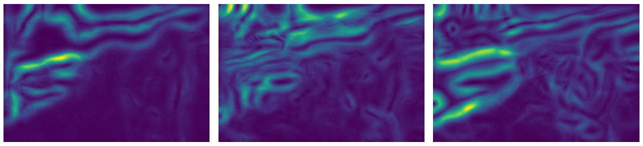

Fig. 23. Image 5 (a) Texton Gradient (b) Brightness Gradient (c) Color Gradient

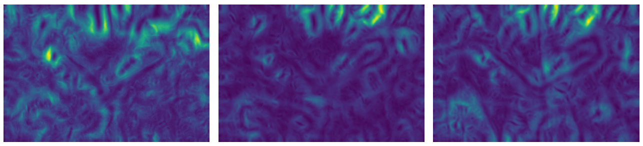

Fig. 24. Image 6 (a) Texton Gradient (b) Brightness Gradient (c) Color Gradient

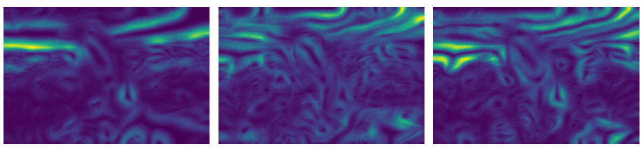

Fig. 25. Image 7 (a) Texton Gradient (b) Brightness Gradient (c) Color Gradient

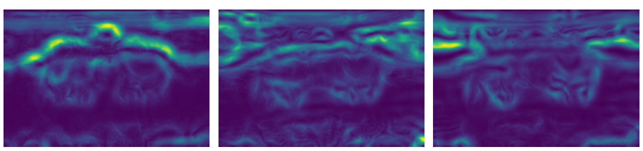

Fig. 26. Image 8 (a) Texton Gradient (b) Brightness Gradient (c) Color Gradient

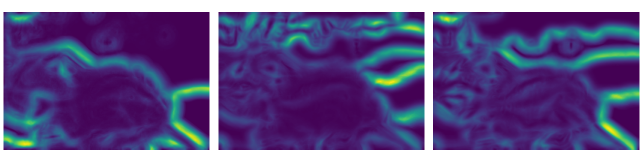

Fig. 27. Image 9 (a) Texton Gradient (b) Brightness Gradient (c) Color Gradient

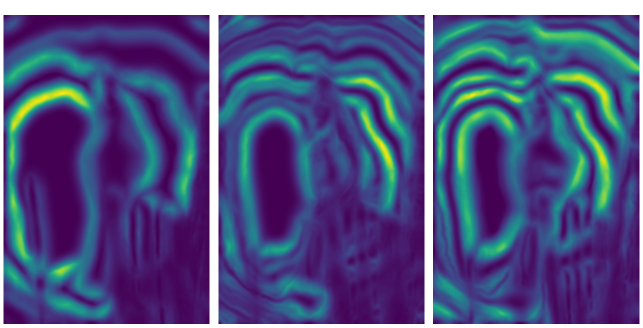

Fig. 28. Image 10 (a) Texton Gradient (b) Brightness Gradient (c) Color Gradient

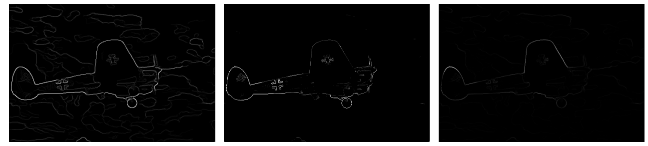

Fig. 29. Image 1 (a) Canny (b) Sobel (c) Pblite

Fig. 30. Image 2 (a) Canny (b) Sobel (c) Pblite

Fig. 31. Image 3 (a) Canny (b) Sobel (c) Pblite

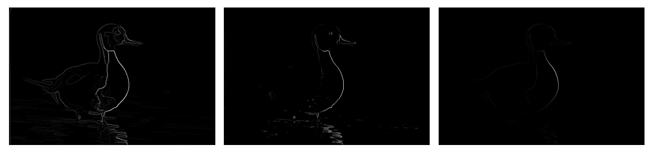

Fig. 32. Image 4 (a) Canny (b) Sobel (c) Pblite

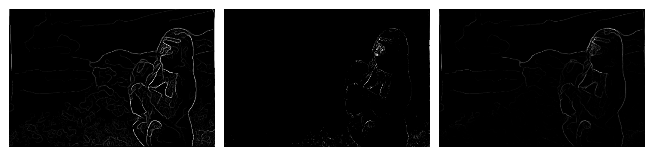

Fig. 33. Image 5 (a) Canny (b) Sobel (c) Pblite

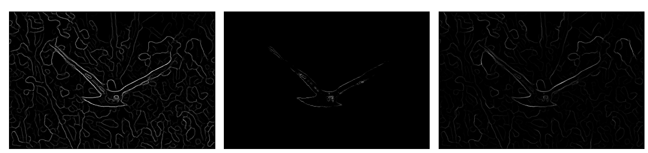

Fig. 34. Image 6 (a) Canny (b) Sobel (c) Pblite

Fig. 35. Image 7 (a) Canny (b) Sobel (c) Pblite

Fig. 36. Image 8 (a) Canny (b) Sobel (c) Pblite

Fig. 37. Image 9 (a) Canny (b) Sobel (c) Pblite

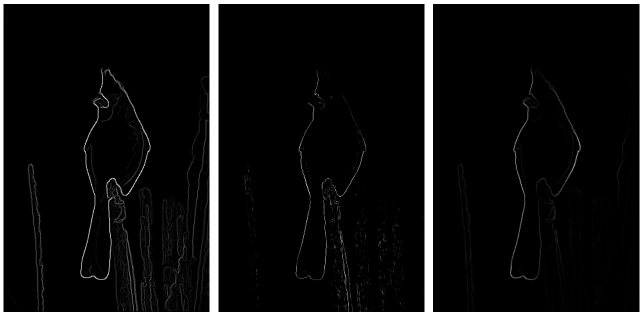

Fig. 38. Image 10 (a) Canny (b) Sobel (c) Pblite

F. Chi Square Distance

Chi-square distance is a statistical method to measure similarity between 2 feature matrices (h, g) and used in many applications like similar image retrieval, image texture, feature extractions. It has the property of distributional equivalence, meaning that it ensures that the distances between rows and columns are invariant. We use chi-sqaure distance to find the various gradient values by comparing each map with particular bins against half disk filter bank.

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

G. K-means Clustering

K-means algorithm clusters data by trying to separate samples in group of equal variance by minimizing inertia or within cluster sum of squares.

Kmeans algorithm divides a set of N samples X into K disjoint clusters C, each described by mean u_i of samples in the cluster.

We first start with initialising the number of clusters and randomly initialise the centroid within the clusters and compute new centroids of each cluster by assigning each point to its closest centroid until the centroid positions remain constant and unaffected by further iterations.

H. Probability based detection

As a final step, we combine all the filter data to obtain texture brightness and color gradients by applying chi square distances. In order to obtain the final edge from these gradients, we use a weighted sum over Sobel and Canny baseline images for the images. The end result, is weighted sum of gradients over these baselines.

Although many approaches just use either Sobel or canny edge detectors to find edges in a image which is also Incorporated in many packages available open source online, it is found that on using Sobel and Canny edge detectors, we find the edges of all the objects and variations present in the image. IN this assignment we use an rigorous approach to use various filter operations on the image and finally detect the edges of particular objects in the image as we can see the difference as shown in Fig. 37.

II. PHASE 2 : DEEP DIVE INTO DEEP LEARNING

We compare multiple neural network architectures by varying the number of parameters to analyse the training and testing accuracy and loss values for training with CIFAR-10 data-set which consists of 60000 32*32 color images in 10 classes with 6000 image per class. The training and testing data-set are split as having 50000 and 10000 images respectively. First, I started my implemented with Convolution neural network to train for minimum epochs. I have used ADAM optimizer and Cross Entropy function for computing loss and Learning rate of $1e^{-2}$ for all the network architecture. I have trained and tested the models using Cluster- turing.wpi.edu. For simple CNN architecture, I have not used any standardization or normalization and for all other variations: ResNet18, ResNet34, DenseNet, ResNeXt. I have used annotations and standardization from torchvision.transforms.

Normalisation and standardization:

- 1. CenterCrop(10)
- 2. Normalise((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
- 3. RandomRotation((30,70)

One of the key observations I made while using different combinations of annotations is that, it has direct impact on the output filter size of the network and I have used MaxPool2d() and AvgPool2d() functions to adjust the filter size input to the final classifier layer.

Sno	**Model**	**No of Epochs**	**Train Accuracy**	**Test Accuracy**
1.	CNN	477	77.78	58.82
2.	ResNet18	150	95.99	43.22
3.	ResNet34	150	91.24	45.56
4.	DenseNet	1200	70.06	45.15
5.	ResNeXt	7	45.41	10.88

TABLE I
DEEP LEARNING ARCHITECTURES AND ACCURACY

With respect to computational time, ResNext took longer time to train while basic and Resnet18 were faster to train. It was a great learning process to try different annotations and various layers for training the network. Further work would be on improving test accuracy even over shorter training.

REFERENCES

[1] https://medium.com/@sergioalves94/deep-learning-in-pytorch-with-cifar-10-dataset-858b504a6b54.

[2] https://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm.

[3] https://github.com/miraclewkf/ResNeXt-PyTorch.

[4] Arbelaez, Pablo, et al. "Contour detection and hierarchical image segmentation." IEEE transactions on pattern analysis and machine intelligence 33.5 (2010): 898-916.

[5] https://en.wikipedia.org/wiki/Gaussian$_{filter}$

```
[678  37  68  11  21  12  15   0 131  27] (0)
[ 36 755  19  18   5  15   9   0  39 104] (1)
[ 74  24 565  72  75  75  77   0  20  18] (2)
[ 32  22 108 508  68 150  57   0  26  29] (3)
[ 34  14 120  72 586  65  73   0  24  12] (4)
[ 19  14  73 145  53 645  17   0  17  17] (5)
[  9  12  85  67  80  31 702   0   8   6] (6)
[ 56  34 154 124 279 209  25   0  29  90] (7)
[ 99  41  19  23  10   5   3   0 742  58] (8)
[ 38 105  21  21  14  20  11   0  69 701] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 58.82 %
Test Accuracy =  58.82 %
```

Fig. 39. Test Confusion Matrix for CNN Basic Layer

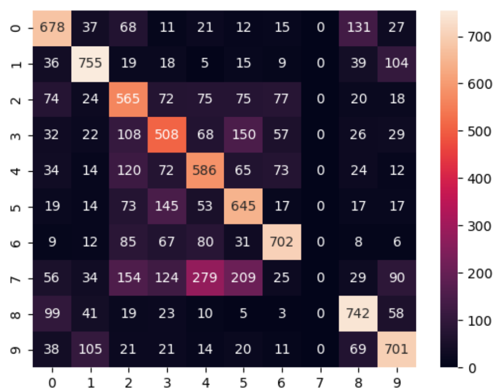

Fig. 40. Test Confusion Matrix HeatMap for CNN Basic Layer

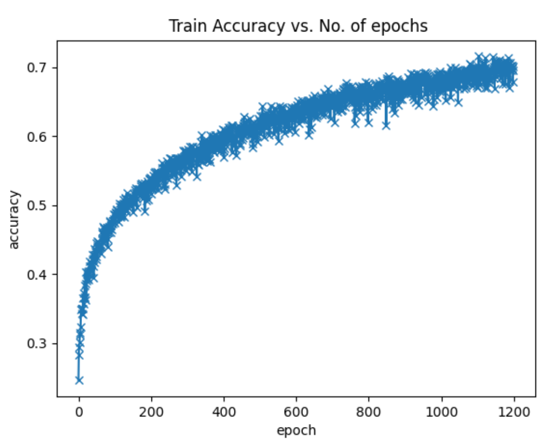

Fig. 41. Training Accuracy for DenseNet Layer

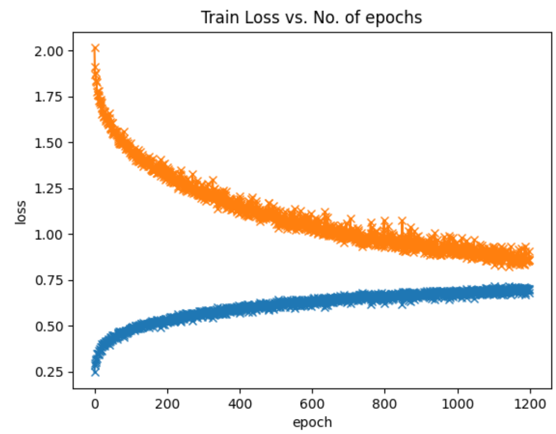

Fig. 42. Training Loss for DenseNet Layer

```
  0%|          | 0/10000 [00:00<?, ?it/s]
[394 110  48  45  57  31  24  13 188  90] (0)
[ 93 530  17  26  22  22  15  13 102 160] (1)
[ 64  32 291 126 178  88  99  37  34  51] (2)
[ 33  16  70 412 116 157  88  28  41  39] (3)
[ 26  20  88 118 462  64 101  65  38  18] (4)
[ 26  17  53 239 101 403  36  36  31  58] (5)
[  7  13  63 139 121  47 565  10  12  23] (6)
[ 36  21  46 104 149  84  29 450  22  59] (7)
[119  75  32  45  39  32  30  13 488 127] (8)
[ 70  86  32  62  50  33  31  14 102 520] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 45.15 %
```

Fig. 43. Test Confusion Matrix for DenseNet Layer

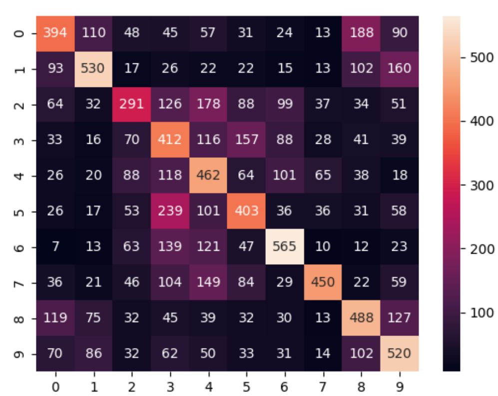

Fig. 44. Test Confusion Matrix HeatMap for DenseNet Layer

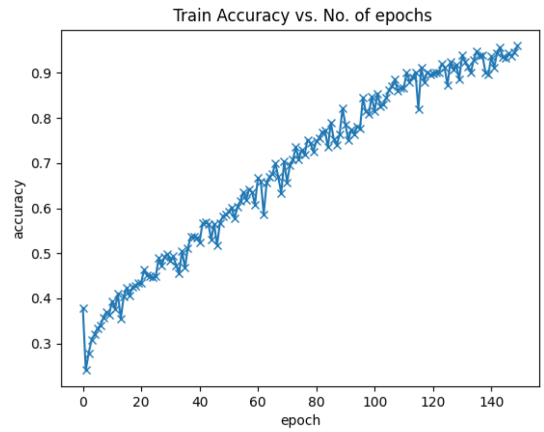

Fig. 45. Training Accuracy for ResNet 18 Layer

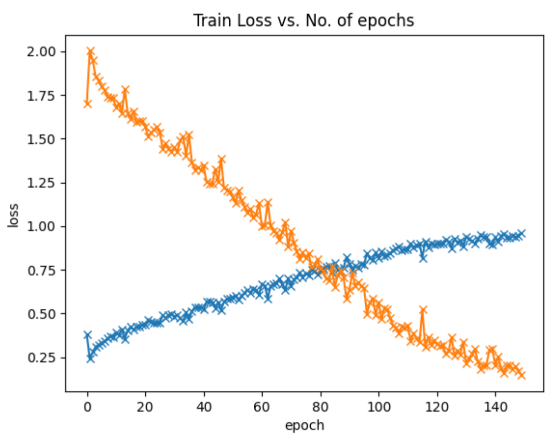

Fig. 46. Training Loss for ResNet 18 Layer

```
[403 122  73  44  26  53  19  26 196  38] (0)
[111 533  32  30  16  35  15  28 128  72] (1)
[ 73  23 351 107 101 109 102  65  36  33] (2)
[ 49  11  92 366  57 212  91  54  51  17] (3)
[ 37  14 127 104 360 103  94 101  38  22] (4)
[ 29  15  83 186  54 445  44  82  43  19] (5)
[ 23  17 107 118  66  76 535  35  13  10] (6)
[ 58  18  75  74  79 123  33 491  24  25] (7)
[117  82  44  49  25  44  29  28 514  68] (8)
[107 137  59  64  33  56  31  52 137 324] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 43.22 %
Test Accuracy =  43.22 %
```

Fig. 47. Test Confusion Matrix for ResNet 18 Layer

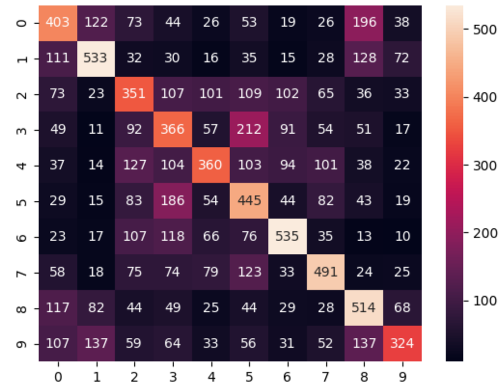

Fig. 48. Test Confusion Matrix HeatMap for ResNet 18 Layer

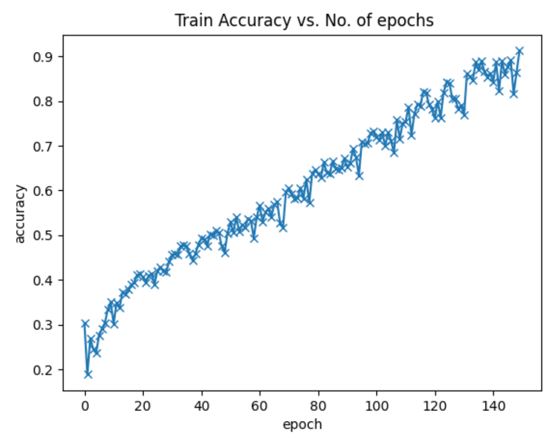

Fig. 49. Training Accuracy for ResNet 34 Layer

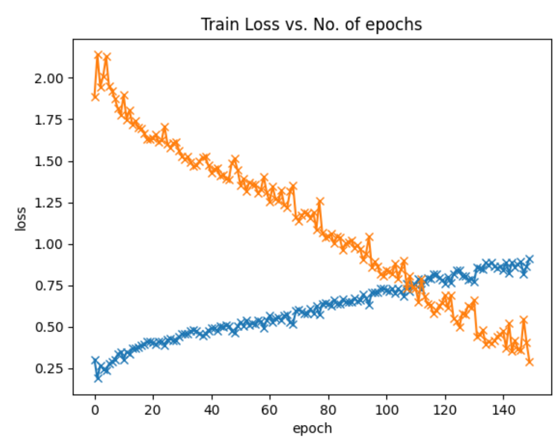

Fig. 50. Training Loss for ResNet 34 Layer

```
[426 111  66  25  26  66  26  29 141  84] (0)
[117 552  27   9   6  42  19  16  84 128] (1)
[ 74  37 335  73  70 167  98  63  35  48] (2)
[ 26  22  60 286  40 308 107  65  35  51] (3)
[ 32  18 115  74 322 148  97 123  28  43] (4)
[ 30  30  52 101  33 558  36  75  35  50] (5)
[  7  15  94  82  50 105 581  27  17  22] (6)
[ 41  24  49  43  42 164  23 536  23  55] (7)
[131 120  32  22  19  60  25  27 433 131] (8)
[ 70 121  39  23  22  81  28  25  64 527] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 45.56 %
Test Accuracy =  45.56 %
```

Fig. 51. Test Confusion Matrix for ResNet 34 Layer

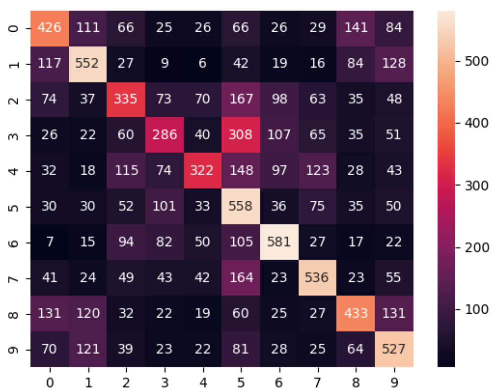

Fig. 52. Test Confusion Matrix HeatMap for ResNet 34 Layer

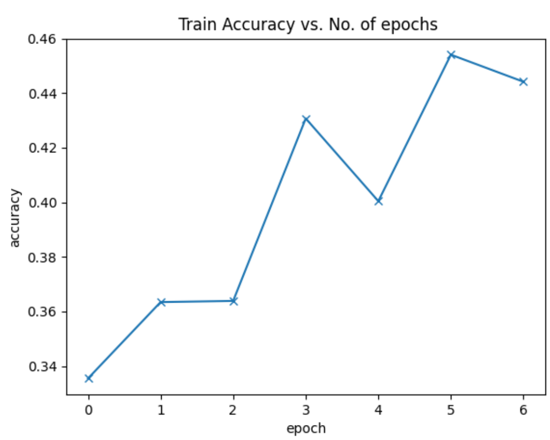

Fig. 53. Training Accuracy for ResNeXt Layer

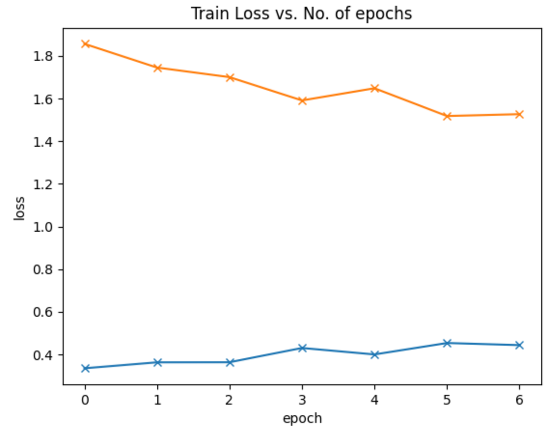

Fig. 54. Training Loss for ResNeXt Layer

```
  0%|          | 0/10000 [00:00<?, ?it/s]
[105   0 892   0   0   0   0   0   3   0] (0)
[  2   0 998   0   0   0   0   0   0   0] (1)
[ 16   0 983   0   0   0   0   0   1   0] (2)
[   0    0 1000    0    0    0    0    0    0] (3)
[  2   0 998   0   0   0   0   0   0   0] (4)
[   0    0 1000    0    0    0    0    0    0] (5)
[  0   0 999   0   1   0   0   0   0   0] (6)
[  1   0 999   0   0   0   0   0   0   0] (7)
[ 39   0 961   0   0   0   0   0   0   0] (8)
[  4   0 995   0   1   0   0   0   0   0] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
Accuracy: 10.88 %
```

Fig. 55. Test Confusion Matrix for ResNeXt Layer

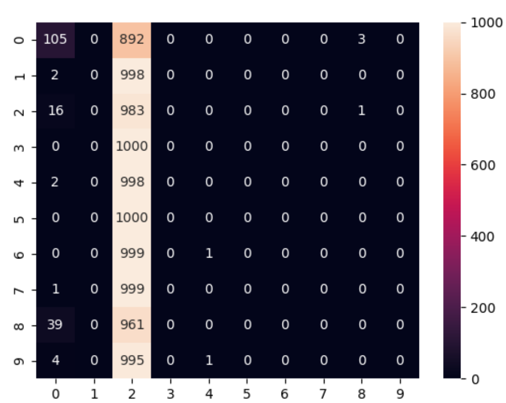

Fig. 56. Test Confusion Matrix HeatMap for ResNeXt Layer

```
CIFAR10Model(
  (layer1): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
  )
  (layer2): Sequential(
    (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer3): Sequential(
    (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=10, stride=10, padding=0, dilation=1, ceil_mode=False)
  )
  (fc): Sequential(
    (0): Linear(in_features=128, out_features=512, bias=True)
    (1): ReLU()
  )
  (fc1): Sequential(
    (0): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

Fig. 57. Basic Model Architecture

```
ResNet(
  (conv_1): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (res_1): Sequential(
    (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
  )
  (conv_2): Sequential(
    (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU()
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (res_2): Sequential(
    (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
  )
  (fc): Sequential(
    (0): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

Fig. 58. ResNet18 Model Architecture

```
ResNet(
  (conv_1): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU()
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (res_1): Sequential(
    (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU()
    (9): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU()
  )
  (conv_2): Sequential(
    (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU()
    (9): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU()
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (14): ReLU()
    (15): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (16): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (17): ReLU()
  )
  (res_2): Sequential(
    (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU()
  )
  (avgpool): AvgPool2d(kernel_size=5, stride=5, padding=0)
  (fc): Sequential(
    (0): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

Fig. 59. ResNet34 Model Architecture

```
ResNeXt(
  (conv1): Conv2d(3, 64, kernel_size=(1, 1), stride=(1, 1))
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (block1): Sequential(
    (0): res_block(
      (layers): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=2)
        (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (4): Conv2d(128, 256, kernel_size=(1, 1), stride=(1, 1))
        (5): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (residual): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1))
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
  )
  (block2): Sequential(
    (0): res_block(
      (layers): Sequential(
        (0): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1))
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=2)
        (3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (4): Conv2d(256, 512, kernel_size=(1, 1), stride=(1, 1))
        (5): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (residual): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2))
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
  )
  (block3): Sequential(
    (0): res_block(
      (layers): Sequential(
        (0): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1))
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=2)
        (3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (4): Conv2d(512, 1024, kernel_size=(1, 1), stride=(1, 1))
        (5): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (residual): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2))
        (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
  )
  (avgpool): AvgPool2d(kernel_size=8, stride=8, padding=0)
  (fc): Sequential(
    (0): Linear(in_features=1024, out_features=10, bias=True)
  )
)
```

Fig. 60. ResNeXt Model Architecture