Homework 0 - Alohomora

Karthik Mundanad
Robotics Department
Worcester Polytechnic Institute
Email: krmundanad@wpi.edu

Using 1 late day

***Abstract*—This report is subdivided into two parts - Phase 1 and Phase 2. Phase 1 deals with Probability of Boundary (Pb) based Boundary/edge detection which is aimed to improve the traditional methods of Edge detection such as Canny and Sobel by incorporating Texture, Brightness and Color of image. We discuss this in deatil in the next section. The next part Phase 2 deals with Deep Learning approaches for classification wherein we particularly discuss Networks such as CNNs, ResNets, ResNeXt and DenseNet.**

I. Phase - 1 Shake my Boundary

A. Introduction

This section deals with Pb-based Boundary detection. Pb-based Boundary detection algorithm outperforms traditional methods such as Canny and Sobel by taking a weighted sum of these along with other image parameters such as Texture, Color, and Brightness. The algorithm can be divided into subparts as follows.

B. Filter Bank Generation

Filter Banks are a set of filters with differences in tuneable parameters such as size and orientation. We mainly implement three filter banks that aggregate low-level regional features such as texture and brightness.

1) Difference of Gaussian (DoG): The Gaussian Kernel is fundamental in computer vision applications. We generate $s \times o$ sized filter bank DoG by convolving this Gaussian kernel with Sobel kernel. s is the scale (here 2) and o is orientation (here 8). Figure 1 shows the result for the same

2) Leung-Malik (LM): The LM filter bank is an extensive filter bank consisting of 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. Here, 2 different LM filter banks are generated - LM small (LMS) and LM large (LML) at two different sets of scales. Both these are shown by figure 2 and figure 3 respectively.

3) Gabor: Gabor filters are generated by modulating Gaussians with sinusoidal waves of different scales and orientations. This is represented in figure 4

C. Map Generation

Next, we use the above filters are generate Texton Maps. In addition, we also generate Brightness and Color Maps for each image

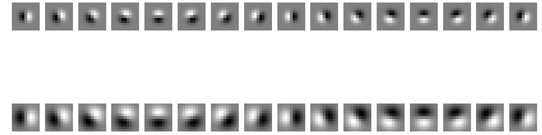


Fig. 1: DoG Filter Bank

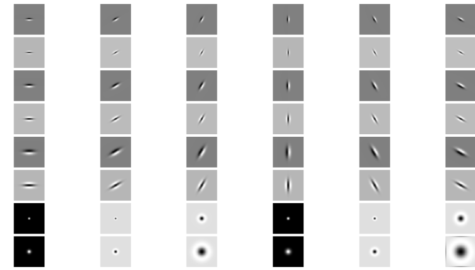


Fig. 2: LM Small Filter Bank

1) Texton Maps $\mathcal{T}$: Texton Map for each image is generated by applying all the filters (here DoG, LM small and Gabor) on the image. This results in an N-dimensional vector response consisting of the textural information for each image where N is the number of filters. To encode this vector and assign an ID - called the Texton ID we use KMeans Clustering which results in a Texton map. (Note K here is 64)

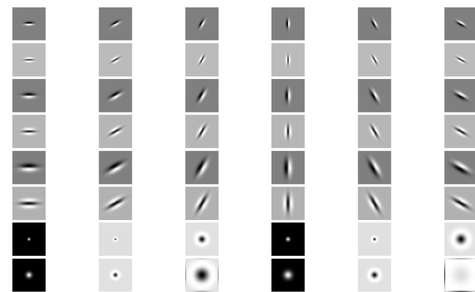


Fig. 3: LM large Filter Bank

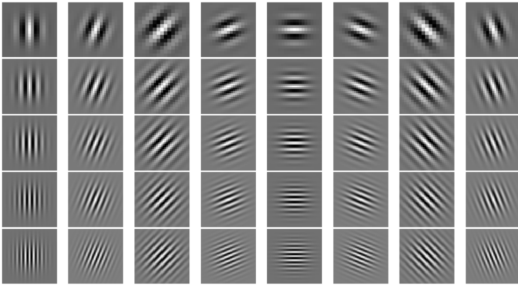
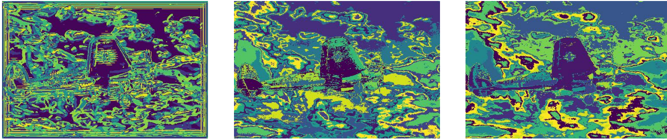


Fig. 4: Gabor Filter Bank

2) Brightness Maps $\mathcal{B}$: To generate the Brightness Maps, we cluster the intensity using KMeans (K = 16) after converting the image to grayscale.

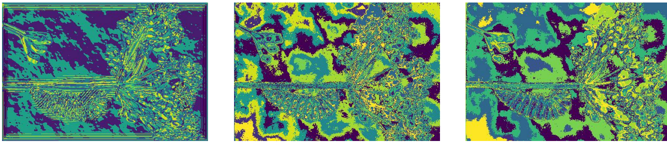
3) Color Maps $\mathcal{C}$: Similar to Brightness Maps, we get the Color map for each image by clustering the image using KMeans (K =16).

The Texton, Brightness, and Color Maps are shown in Figures 5 - 14



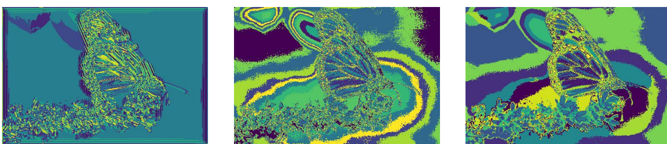
(a) Texton Map (b) Brightness Map (c) Color Map

Fig. 5: Maps for Image-1



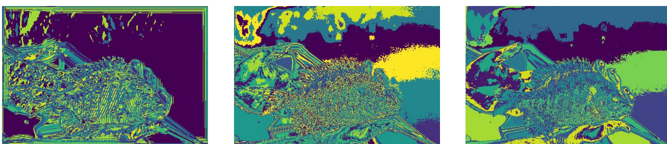
(a) Texton Map (b) Brightness Map (c) Color Map

Fig. 6: Maps for Image-2



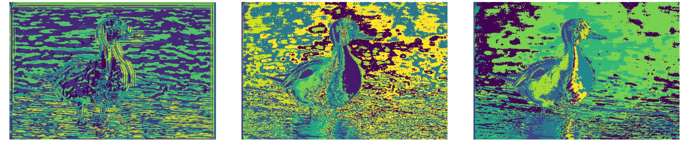
(a) Texton Map (b) Brightness Map (c) Color Map

Fig. 7: Maps for Image-3



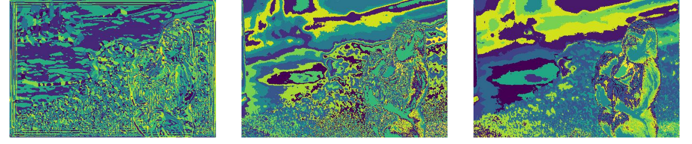
(a) Texton Map (b) Brightness Map (c) Color Map

Fig. 13: Maps for Image-9



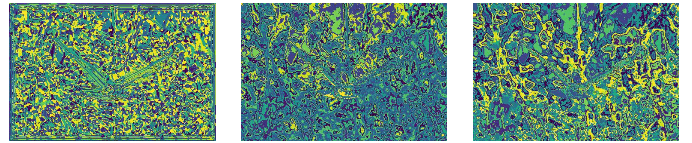
(a) Texton Map (b) Brightness Map (c) Color Map

Fig. 8: Maps for Image-4



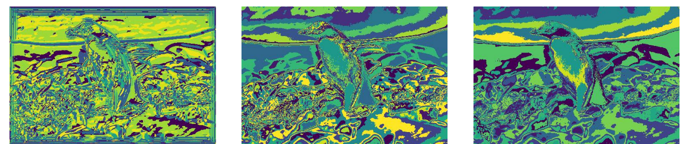
(a) Texton Map (b) Brightness Map (c) Color Map

Fig. 9: Maps for Image-5



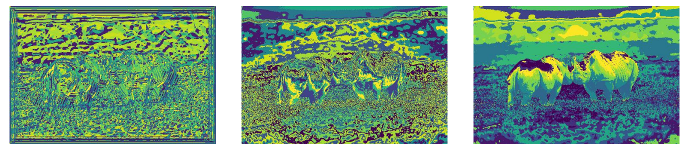
(a) Texton Map (b) Brightness Map (c) Color Map

Fig. 10: Maps for Image-6



(a) Texton Map (b) Brightness Map (c) Color Map

Fig. 11: Maps for Image-7

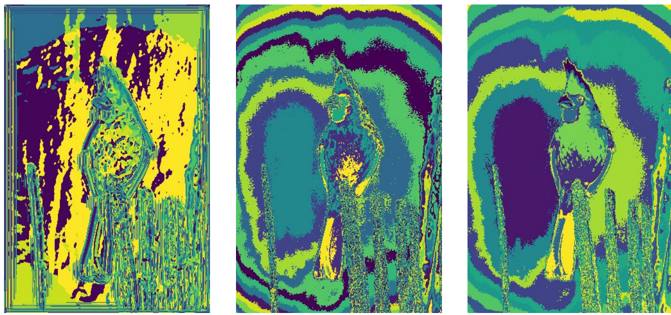


(a) Texton Map (b) Brightness Map (c) Color Map

Fig. 12: Maps for Image-8

D. Gradient Generation

Once we have the maps, we need to find how these factors (texture, brightness, color) change in the image. For this, we calculate the gradients corresponding to each of the maps. These can be generated by looping over neighbors of each pixel but that's not optimal. Instead, we use halfmasks shown in Figure 15. These are complementary masks that are applied to masks and generate the two sets of aggregated histograms. The chi-square distance between the histogram gives the change of texture, brightness, and color. The higher the difference, the higher the change. The gradient maps ($\mathcal{T}_g$, $\mathcal{B}_g$, $\mathcal{C}_g$) are shown for all images from figure 16 - 25



(a) Texton Map (b) Brightness Map (c) Color Map

Fig. 14: Maps for Image-10

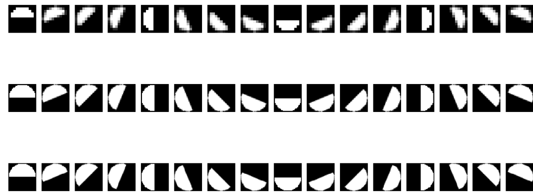
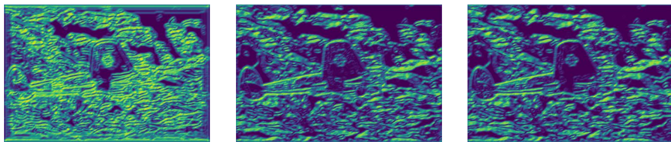
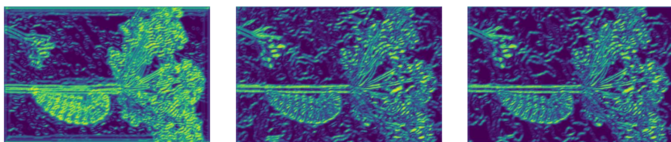


Fig. 15: Half Disk Makss



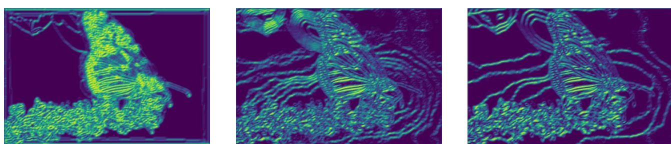
(a) Texton Gradient (b) Brightness Gradient (c) Color Gradient

Fig. 16: Gradients for Image-1



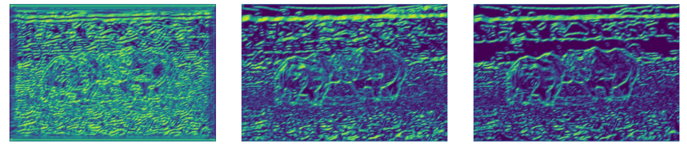
(a) Texton Gradient (b) Brightness Gradient (c) Color Gradient

Fig. 17: Gradients for Image-2



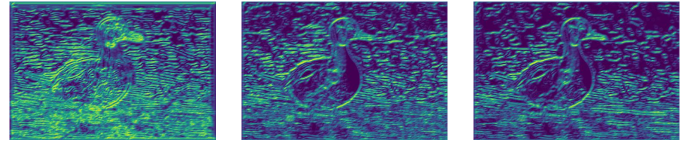
(a) Texton Gradient (b) Brightness Gradient (c) Color Gradient

Fig. 18: Gradients for Image-3



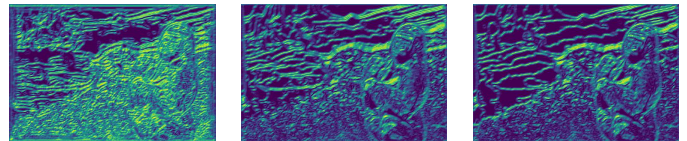
(a) Texton Gradient (b) Brightness Gradient (c) Color Gradient

Fig. 23: Gradients for Image-8



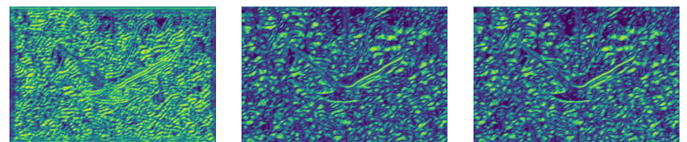
(a) Texton Gradient (b) Brightness Gradient (c) Color Gradient

Fig. 19: Gradients for Image-4



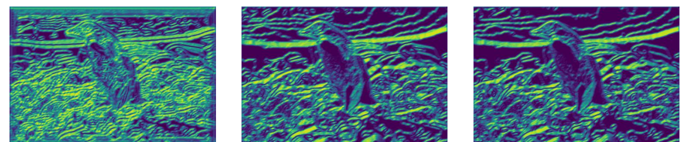
(a) Texton Gradient (b) Brightness Gradient (c) Color Gradient

Fig. 20: Gradients for Image-5



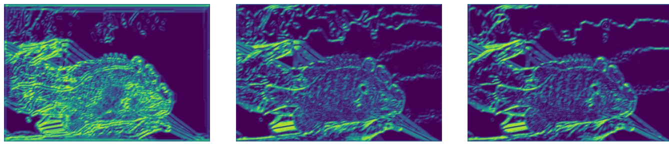
(a) Texton Gradient (b) Brightness Gradient (c) Color Gradient

Fig. 21: Gradients for Image-6



(a) Texton Gradient (b) Brightness Gradient (c) Color Gradient

Fig. 22: Gradients for Image-7

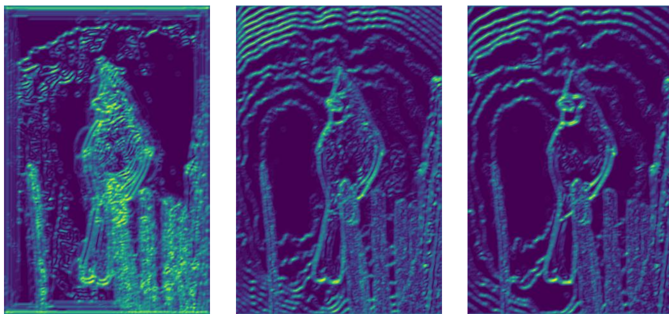


(a) Texton Gradient (b) Brightness Gradient (c) Color Gradient

Fig. 24: Gradients for Image-9

(a) Canny Baseline (b) Sobel Baseline (c) PB result

Fig. 26: Comparison of Sobel, Canny and PB based Edge Detection for Image-1



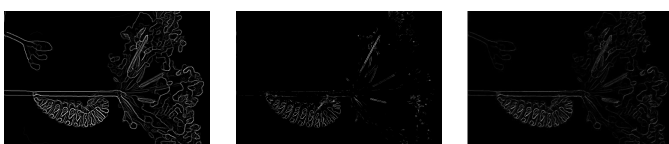
(a) Texton Gradient (b) Brightness Gradient (c) Color Gradient

Fig. 25: Gradients for Image-10

1) Combining Sobel, Canny, and Gradients for Edge Detection: Now to detect boundaries, we take the weighted sum of Sobel and Canny's results and multiply this by the mean of the three gradients. Here we have assigned Sobel and Canny edges equal weights. This resultant Pb boundary is given by

$$\mathcal{PB} = \frac{\mathcal{T}_g + \mathcal{B}_g + \mathcal{C}_g}{3} \odot (w_1 \cdot S + w_2 \cdot C) \tag{1}$$

The resultant Canny, Sobel and PB images are shown in Figures 26-35

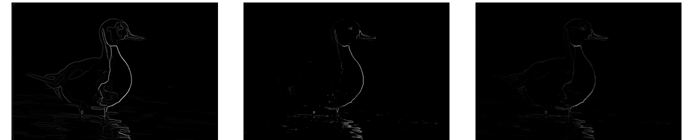


(a) Canny Baseline (b) Sobel Baseline (c) PB result

Fig. 27: Comparison of Sobel, Canny and PB based Edge Detection for Image-2

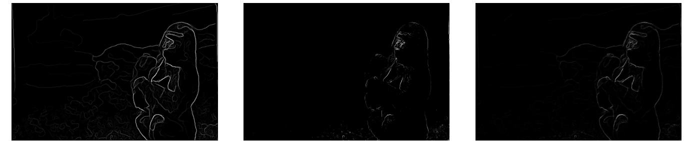
(a) Canny Baseline (b) Sobel Baseline (c) PB result

Fig. 28: Comparison of Sobel, Canny and PB based Edge Detection for Image-3



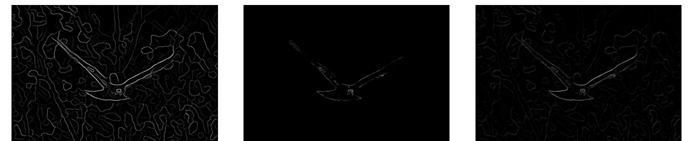
(a) Canny Baseline (b) Sobel Baseline (c) PB result

Fig. 29: Comparison of Sobel, Canny and PB based Edge Detection for Image-4



(a) Canny Baseline (b) Sobel Baseline (c) PB result

Fig. 30: Comparison of Sobel, Canny and PB based Edge Detection for Image-5

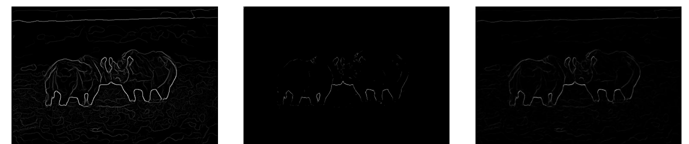


(a) Canny Baseline (b) Sobel Baseline (c) PB result

Fig. 31: Comparison of Sobel, Canny and PB based Edge Detection for Image-6

(a) Canny Baseline (b) Sobel Baseline (c) PB result

Fig. 32: Comparison of Sobel, Canny and PB based Edge Detection for Image-7

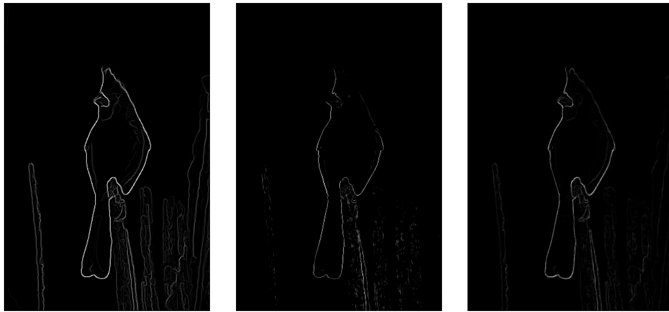


(a) Canny Baseline (b) Sobel Baseline (c) PB result

Fig. 33: Comparison of Sobel, Canny and PB based Edge Detection for Image-8

(a) Canny Baseline (b) Sobel Baseline (c) PB result

Fig. 34: Comparison of Sobel, Canny and PB based Edge Detection for Image-9



(a) Canny Baseline (b) Sobel Baseline (c) PB result

Fig. 35: Comparison of Sobel, Canny and PB based Edge Detection for Image-10

E. Conclusion

Although it might seem that PB edges are not legible, it is because it is producing fine edges for the object(s) in focus. Thus we can see that it suppresses are lot of unwanted background edges produced in Canny and Sobel. The performance can even be improved by tuning the weights for Sobel and Canny, producing texton maps using diverse filter banks, etc. Thus this fine tuning can allow us to produce desired results.

II. Phase 2 - Deep Learning

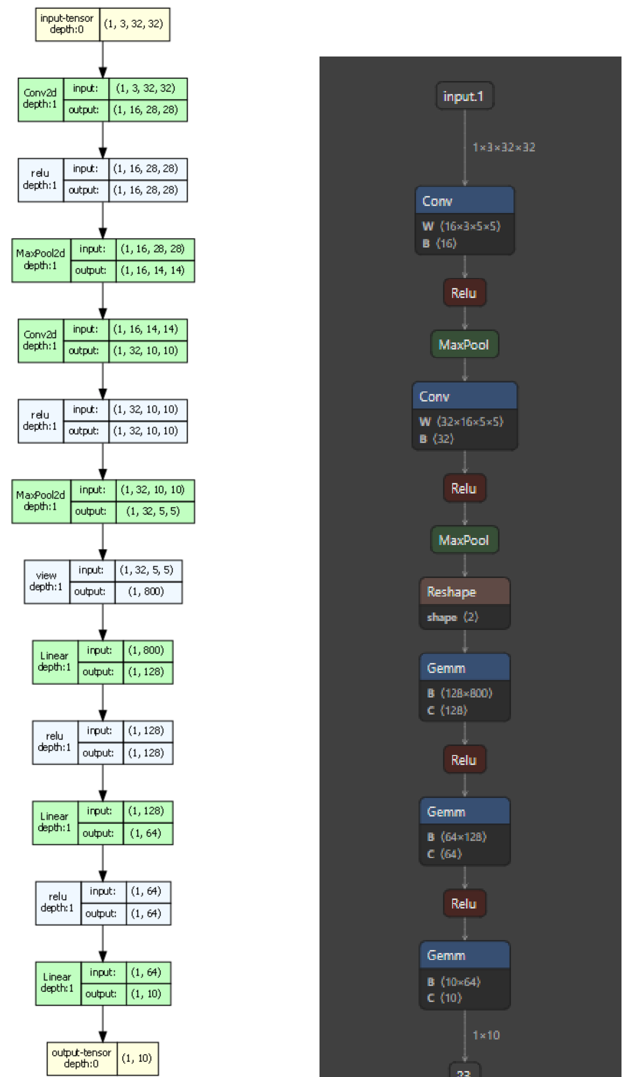
In this section, Deep Learning models on the CIFAR-10 Dataset are discussed. We implement a classification task on the CIFAR-10 dataset which contains 50,000 training and 10,000 testing images of size 32x32 belonging to 10 classes. There are several networks implemented for the same which will be discussed in the next sections

A. Convolution Neural Networks

Firstly, I implemented a CNN with 5-layer CNN with 2 convolution blocks and 3 linear layers. Cross Entropy Loss and Adam Optimizer were used for training. The hyperparameters including a number of epochs, batch size, and learning rate (LR) are shown in Table I. The number of parameters chosen for the implementation and the results for Train Loss, Train Accuracy, and Test Accuracy are also given in Table II.

Epochs	Batch Size	L.R	Parameters
30	64	0.001	125,482

TABLE I: Parameters and Network details for CNN



(a) Architecture Overview (b) Netron Architecture

Fig. 36: Architecture for CNN based Classifier

Train Loss	Train Acc	Test Acc
0.6467	81.25	63.96

TABLE II: Training and Testing Results for CNN

The plots for architecture and Netron details are attached below in Figure 36.

The variation of training loss, training accuracy, and testing accuracy with epochs is given in Figure 37 - 44

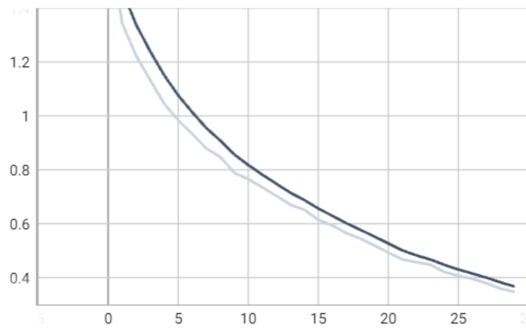


Fig. 37: Training loss vs Epochs for CNN

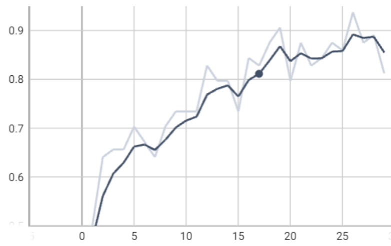


Fig. 38: Training accuracy vs Epochs for CNN

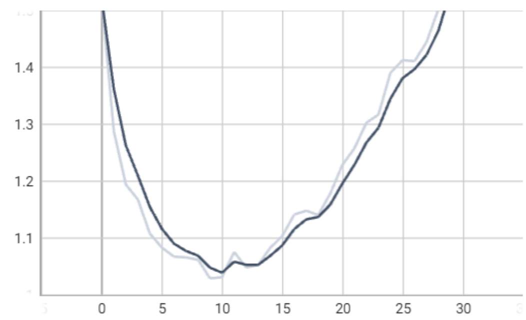


Fig. 39: Testing loss vs Epochs for CNN

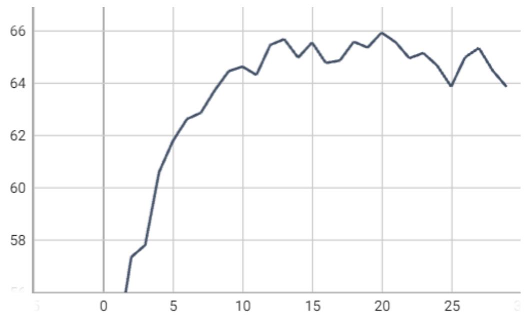


Fig. 40: Testing accuracy vs Epochs for CNN

The confusion matrix is given in figure 41 and 42

```
[4618   21   96   20   17   14    1   14  153   46] (0)
[  39 4686   10   16    0   14    6    4  135   90] (1)
[ 168    9 4202  155  148  189   24   62   29   14] (2)
[  71    6  131 3711   85  813   25  105   34   19] (3)
[  99    5  225  181 4013  251   14  167   31   14] (4)
[  37    6   83  271   70 4373    5  129   15   11] (5)
[  36   38  241  382  189  296 3723   35   34   26] (6)
[  30    5   78   73   97  151    0 4537   10   19] (7)
[  95   21   15   21    3   15    1    2 4809   18] (8)
[  56  127   23   45    5   32    2   23  109 4578] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
```

Fig. 41: Training Confusion matrix for CNN

```
[ 27 771  11   5   8   9   5   9  63  92] (1)
[ 96   5 545  77  79 108  20  37  22  11] (2)
[ 37  12  69 446  47 255  30  52  24  28] (3)
[ 40   4 109  94 526  88  22  88  19  10] (4)
[ 21   5  63 162  31 620  12  68  10   8] (5)
[ 18  14  80 126  59  85 577  13  21   7] (6)
[ 28   8  42  44  74  96   7 677  11  13] (7)
[ 84  34  23   9  11  14   3   5 798  19] (8)
[ 64 105   9  29   5  19  10  19  52 688] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
```

Fig. 42: Testing Confusion matrix for CNN

B. Improved CNN

I added BatchNorms after each convolution block and implemented horizontal flip as well as normalization on the data to improve. The horizontal flips did not yield better outputs as they gave 75.2% training accuracy and 47% testing accuracy. So I only took into account the BatchNorms and have presented the results for the same. The Table III and Table IV show the parameters and results.

Epochs	Batch Size	L.R	Parameters
30	64	0.001	125,538

TABLE III: Parameters and Network details for CNN

Train Loss	Train Acc	Test Acc
0.8025	90.3	71.37

TABLE IV: Training and Testing Results for CNN

The architecture is given Figure 43.

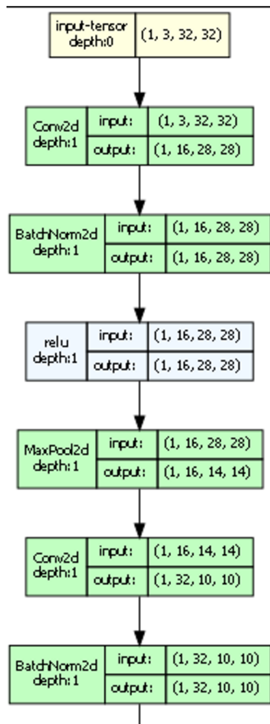


Fig. 43: Training accuracy vs Epochs for new CNN

The variation of training loss, training accuracy, and testing accuracy with epochs is given in Figure 44 - 47.

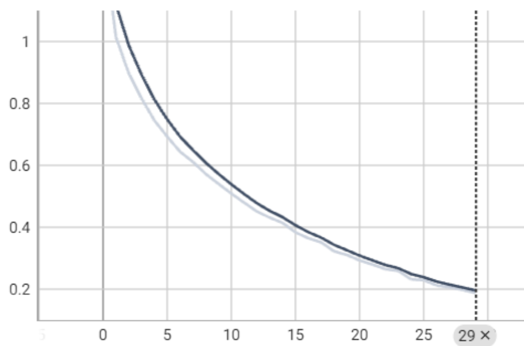


Fig. 44: Training loss vs Epochs for New CNN

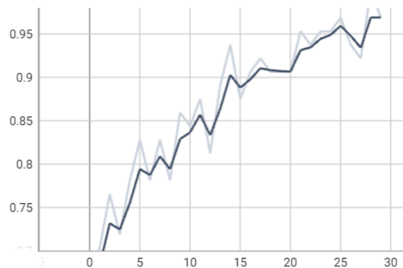


Fig. 45: Training accuracy vs Epochs for new CNN

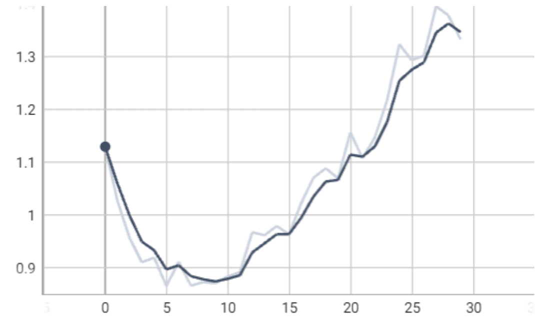


Fig. 46: Testing loss vs Epochs for new CNN

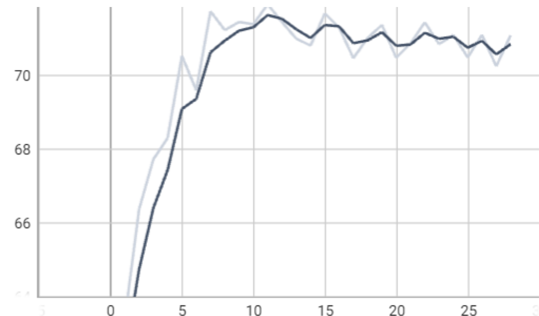


Fig. 47: Testing accuracy vs Epochs for new CNN

The confusion matrix is given in figure 48 and 49

	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
(0)	4618	21	96	20	17	14	1	14	153	46
(1)	39	4686	10	16	0	14	6	4	135	90
(2)	168	9	4202	155	148	189	24	62	29	14
(3)	71	6	131	3711	85	813	25	105	34	19
(4)	99	5	225	181	4013	251	14	167	31	14
(5)	37	6	83	271	70	4373	5	129	15	11
(6)	36	38	241	382	189	296	3723	35	34	26
(7)	30	5	78	73	97	151	0	4537	10	19
(8)	95	21	15	21	3	15	1	2	4809	18
(9)	56	127	23	45	5	32	2	23	109	4578

Fig. 48: Training Confusion matrix for new CNN

	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
(0)	723	19	75	20	23	13	18	14	72	23
(1)	17	820	13	12	3	5	13	2	36	79
(2)	60	2	597	51	76	74	82	29	18	11
(3)	23	3	76	501	53	198	87	32	17	10
(4)	17	2	79	54	685	43	45	58	13	4
(5)	7	5	54	152	44	648	41	36	7	6
(6)	5	4	39	69	23	25	820	2	10	3
(7)	14	2	39	41	70	81	11	726	6	10
(8)	60	31	13	19	7	9	10	3	826	22
(9)	40	64	16	17	4	6	7	9	46	791

Fig. 49: Testing Confusion matrix for new CNN

C. ResNet

For the ResNet Implementation, I looked up the PyTorch implementation but the layers vary from the original implementation. There are 2 skip connections and the tables below show the data as above.

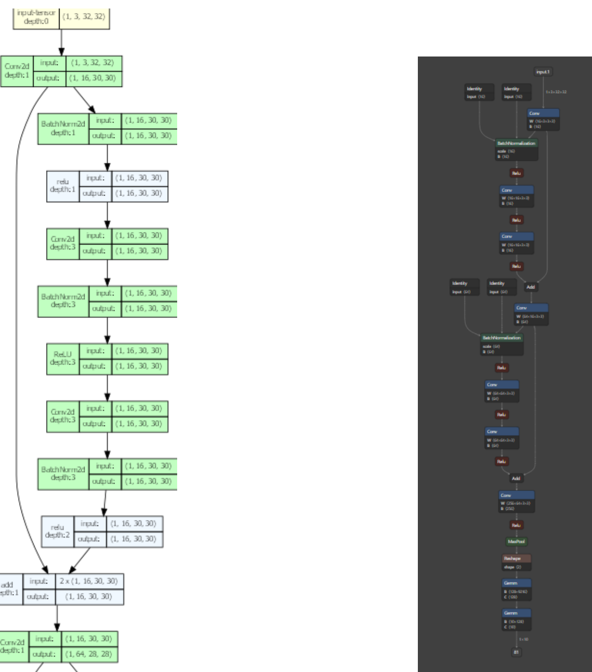
Epochs	Batch Size	L.R	Parameters
30	64	0.001	1,417,994

TABLE V: Parameters and Network details for ResNet

Train Loss	Train Acc	Test Acc
0.4628	82.81	76.01

TABLE VI: Training and Testing Results for ResNet

The snippet of architecture from torchview and netron are also provided in figure 50.



(a) Architecture Overview

(b) Netron Architecture

Fig. 50: Architecture for ResNet based Classifier

The variation of training loss, training accuracy, and testing accuracy with epochs is given in Figure 51 - 54

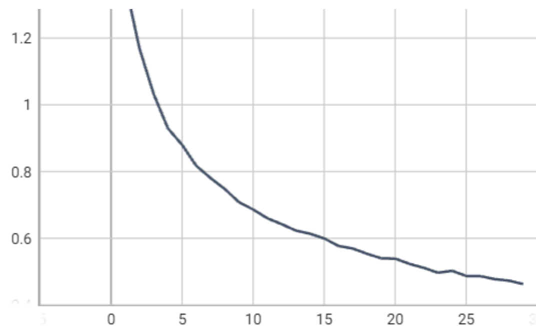


Fig. 51: Training loss vs Epochs for ResNet

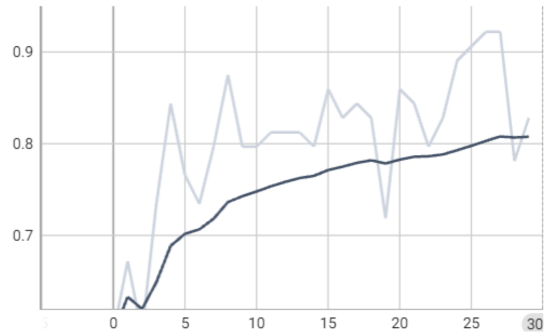


Fig. 52: Training accuracy vs Epochs for ResNet

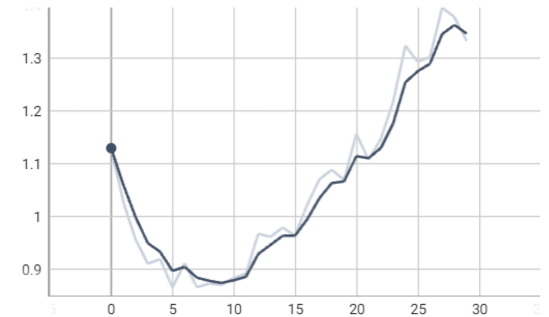


Fig. 53: Testing loss vs Epochs (20) for ResNet

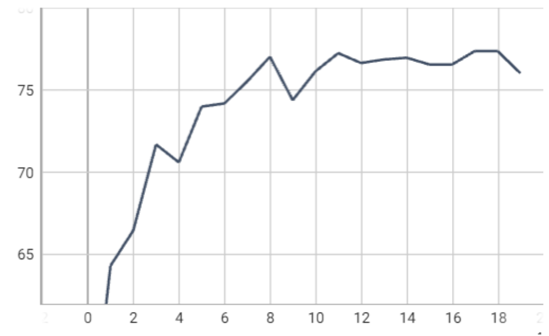


Fig. 54: Testing accuracy vs Epochs (20) for ResNet

The confusion matrix is given in figure 55 and figure 56

```
[4804   10   81    3    8    4    6    6   66   12] (0)
[   3 4936    2    3    1    3    4    1   13   34] (1)
[  11    1 4895   16   14   19   30    3    8    3] (2)
[  14    1   40 4784   17   94   29   11    4    6] (3)
[   6    1   46   26 4848   32   28   10    2    1] (4)
[   0    2   34   42   10 4889   12   10    1    0] (5)
[   0    0   13    7    3    8 4968    0    0    1] (6)
[   6    2   28    8   21   40    1 4888    3    3] (7)
[   8    1    8    1    1    2    3    4 4969    3] (8)
[   2   15    0    2    0    1    3    1    6 4970] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
```

Fig. 55: Training Confusion matrix for ResNet

```
[740  23  77  11  14   8  13   9  74  31] (0)
[ 10 868   4   2   2   5   7   3  32  67] (1)
[ 40   2 692  44  60  60  59  22  13   8] (2)
[ 13   8  87 574  41 163  66  24  12  12] (3)
[ 12   1  75  69 674  58  42  54  12   3] (4)
[ 10   1  58 121  29 713  22  30   9   7] (5)
[  1   3  51  45  20  22 851   2   2   3] (6)
[ 11   3  39  35  46  67   7 780   5   7] (7)
[ 34  13  17   7   3   5   7   2 899  13] (8)
[ 16  53   5  11   2   5   6   6  26 870] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
```

Fig. 56: Testing Confusion matrix for ResNet

D. *ResNeXt*

For the ResNeXt architecture, which is as shown in Figure 57. My implementation includes the same cardinality of 32 and I have used the grouping twice in the network. The parameters are shown in Table VII and results in VIII.

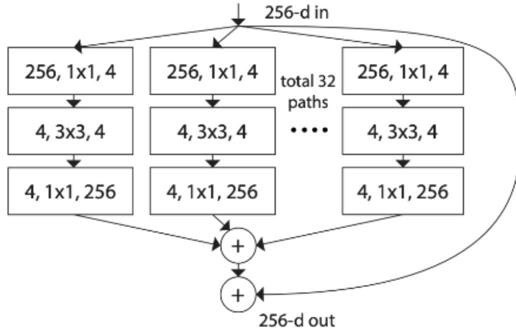


Fig. 57: Reference Architecture for ResNeXt

Epochs	Batch Size	L.R	Parameters	Card.
20	64	0.001	531,210	32

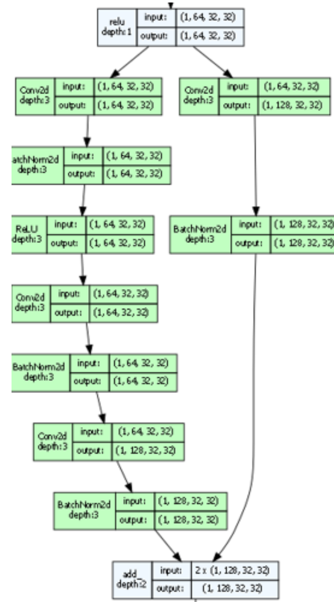
TABLE VII: Parameters and Network details for ResNeXt

Train Loss	Train Acc	Test Acc
0.2258	87.29	79.05

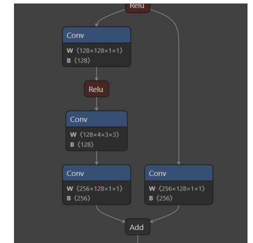
TABLE VIII: Training and Testing Results for ResNeXt

The snippet of architecture from torchview and netron are also provided in figure 58. The architecture doesn't show the split in groups because of torch view limitations but comparing the torch and netron images shown, we can understand that there is a block dividing the incoming channels to outgoing channels as shown.

The variation of training loss, training accuracy, and testing accuracy with epochs is given in Figure 59 - 62



(a) Architecture Overview



(b) Netron Architecture

Fig. 58: Architecture for ResNeXt based Classifier

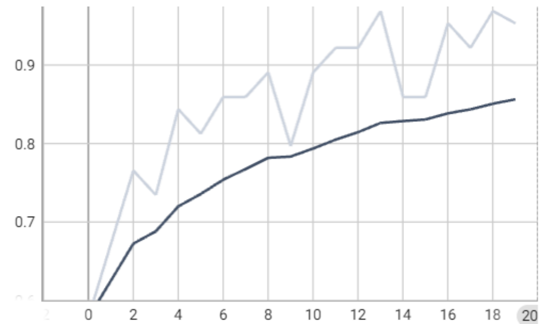


Fig. 59: Training loss vs Epochs for ResNeXt

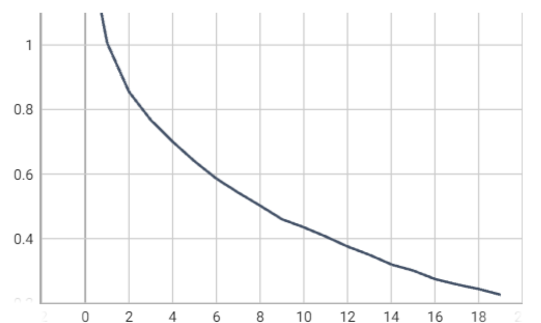


Fig. 60: Training loss vs Epochs for ResNeXt

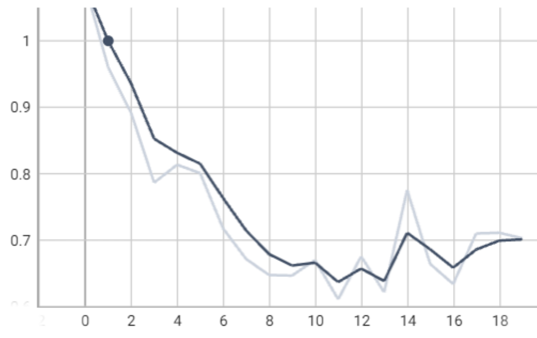


Fig. 61: Testing loss vs Epochs for ResNeXt

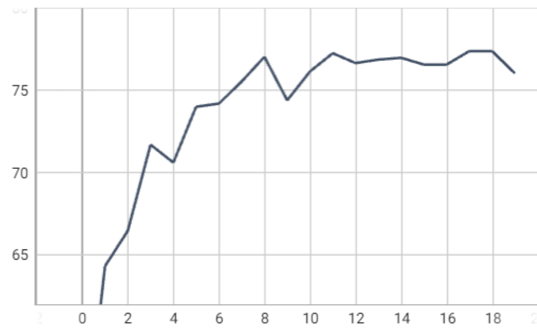


Fig. 62: Testing accuracy vs Epochs for ResNeXt

The confusion matrix is given in figure 63 and figure 64

```
[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. 63: Training Confusion matrix for ResNeXt

```
[884  17  25   9  10   1   5  11  14  24] (0)
[ 19 908   0   3   1   1   3   1   1  63] (1)
[ 55   6 769  43  24  31  38  28   2   4] (2)
[ 23  10  66 706  16  95  47  22   4  11] (3)
[  8   2 153  56 622  45  57  52   3   2] (4)
[  9   4  37 174  11 714  16  32   0   3] (5)
[  6   6  73  32   8   7 863   2   2   1] (6)
[ 14   0  30  40  21  51   7 831   0   6] (7)
[132  36   6  11   4   2   5   6 753  45] (8)
[ 22  42   3   5   3   2   0   6   4 913] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
```

Fig. 64: Testing Confusion matrix for ResNeXt

E. DenseNet

For the Denset architecture, I referred to the PyTorch implementation. I have kept the growth rate as 12 and the number of dense blocks as 2 of 6 and 12 layers each and 1 transition layer. In my bottleneck, I have directly reduced to k(growth rate) channels instead of upsampling to 4*k channels and then k channels as traditionally done. This was to reduce the parameters. The parameters are shown in Table VII and results in VIII.

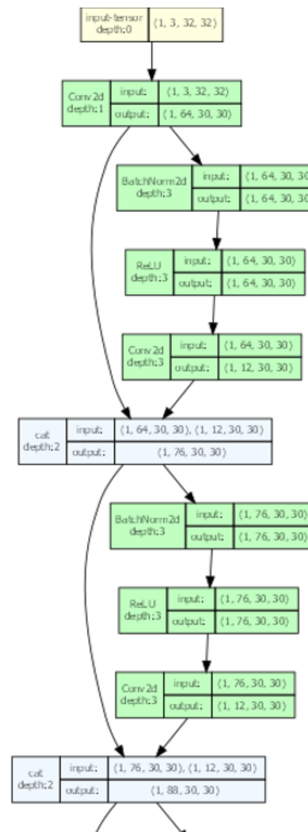
Epochs	Batch Size	L.R	Parameters	Growth Rate
20	64	0.001	253,070	12

TABLE IX: Parameters and Network details for DenseNet

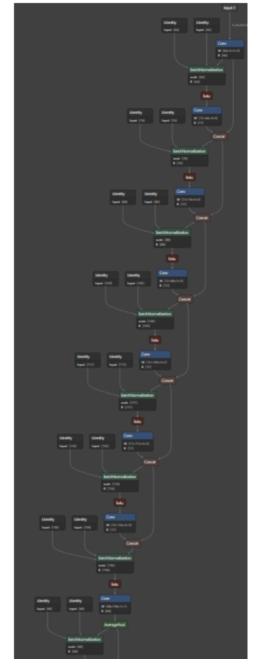
Train Loss	Train Acc	Test Acc
0.1831	95.69	81.31

TABLE X: Training and Testing Results for DenseNet

The snippet of architecture from torchview and netron are also provided in figure 65



(a) Architecture Overview



(b) Netron Architecture

Fig. 65: Architecture for DenseNet based Classifier

The variation of training loss, training accuracy, and testing accuracy with epochs is given in Figure 66 - 69

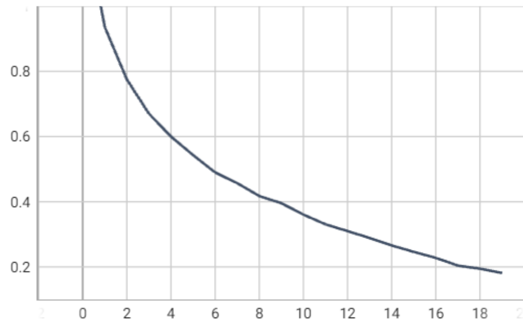


Fig. 66: Training loss vs Epochs for DenseNet

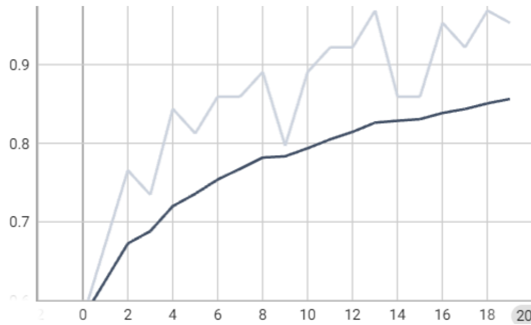


Fig. 67: Training accuracy vs Epochs for DenseNet

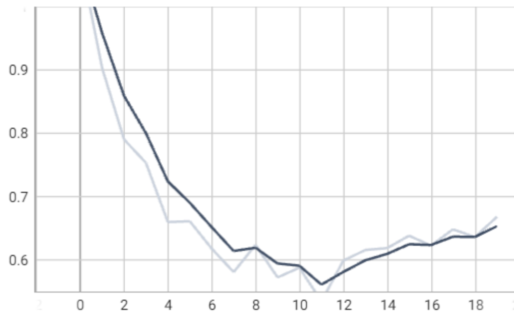


Fig. 68: Testing loss vs Epochs for DenseNet

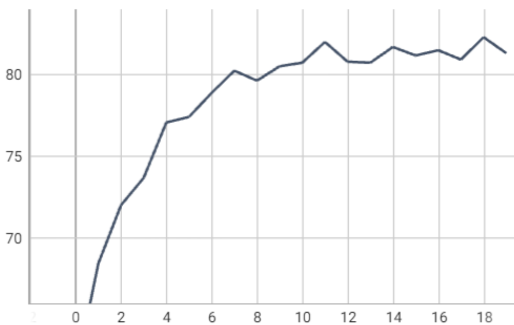


Fig. 69: Testing accuracy vs Epochs for DenseNet

The confusion matrix is given in figure 70 a and 71

```
[835  24  20   9   5   1   6   4  73  23] (0)
[  9 918   1   1   0   0   2   0  25  44] (1)
[ 68   3 690  45  41  30  91   2  25   5] (2)
[ 25   8  44 639  48 136  52  13  29   6] (3)
[ 26   2  36  43 802  21  41   7  20   2] (4)
[ 13   5  25 112  39 750  18  24  12   2] (5)
[  4   5  13  36  11   8 905   2  14   2] (6)
[ 21   6  23  39  82  45   7 753   7  17] (7)
[ 22   9   2   4   0   0   2   0 948  13] (8)
[ 12  54   6   6   1   1   2   2  25 891] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
```

Fig. 70: Training Confusion matrix for DenseNet

```
[835  24  20   9   5   1   6   4  73  23] (0)
[  9 918   1   1   0   0   2   0  25  44] (1)
[ 68   3 690  45  41  30  91   2  25   5] (2)
[ 25   8  44 639  48 136  52  13  29   6] (3)
[ 26   2  36  43 802  21  41   7  20   2] (4)
[ 13   5  25 112  39 750  18  24  12   2] (5)
[  4   5  13  36  11   8 905   2  14   2] (6)
[ 21   6  23  39  82  45   7 753   7  17] (7)
[ 22   9   2   4   0   0   2   0 948  13] (8)
[ 12  54   6   6   1   1   2   2  25 891] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
```

Fig. 71: Testing Confusion matrix for DenseNet

F. Conclusion

From the above graphs, confusion matrix and parameter comparison, I conclude that DenseNet performed better in terms of its accuracy even though it had far fewer parameters compared to ResNeXt and ResNet. It significantly outperformed traditional CNNs as it was deeper and preserving the information by concatenating the previous channels. Adding BatchNorms improved CNNs but it had limitations towards the end as it was not get higher testing accuracy as epochs increased. ResNet and ResNeXt seemed to perform slightly better compared to CNN but the there were lot more parameters resulting in higher inference time. ResNeXt performed better to ResNet even though it was not as deep and had less parameters. To conclude DenseNet was the most suitable architecture and it can be further improved by deeper CNN blocks in it. The table

	Epochs	Parameters	Training Acc.	Testing Acc.	Inf. Time(s)
CNN	30	125,482	81.25	63.96	0.0010
New CNN	30	125,538	90.3	71.27	0.0013
ResNet	20	1,417,994	82.81	76.01	0.020
ResNeXt	20	531,210	87.29	79.05	0.008
DenseNet	20	253,070	95.69	81.31	0.003

TABLE XI: Parameters and Network details for DenseNet