RBE549 HW0 Alohomora

Haoying Zhou
Department of Robotics Engineering
Worcester Polytechnic Institute
Worcester, MA, 01609
Email: hzhou6@wpi.edu

***Abstract*—Note: Use 5 late days. For your information, I have 2 extra late days because HW0 has time conflict with my summer internship**

I. Phase 1: Shake My Boundary

A. Filter Banks

1) Oriented DoG Filters: The Gaussian kernel can be calculated by Equation 1:

$$g(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{1}$$

where σ is the scale of the Gaussian kernel.

To calculate the derivative of Gaussian (DoG) kernel, I use Sobel operator[1] with:

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

As mentioned in the description[2], the derivative of the Gaussian kernel $g(x,y)$ in x and y direction can be denoted as Equation 2:

$$\begin{aligned} \frac{\partial g}{\partial x} &= G_x \circledast g(x,y) \\ \frac{\partial g}{\partial y} &= G_y \circledast g(x,y) \end{aligned} \tag{2}$$

where $\circledast$ represents the convolution operation.

The oriented derivative of Gaussian function is the product of gradient of Gaussian kernel and the rotation vector $r = \left[\begin{smallmatrix} \cos\theta \\ \sin\theta \end{smallmatrix}\right]$. Therefore, the oriented derivative of Gaussian can be denoted as Equation 3:

$$\nabla_r g = \nabla g \cdot r = \cos\theta \frac{\partial g}{\partial x} + \sin\theta \frac{\partial g}{\partial y} \tag{3}$$

Then, with different σ values and orientations, I can obtain the oriented DoG filters as shown in Figure 1:

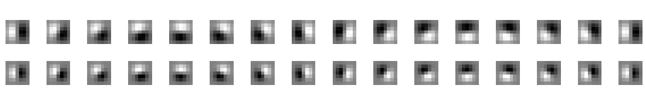


Fig. 1: Oriented DoG filter bank

2) Leung-Malik Filters: Compared to DoG filter, the Gaussian kernel in Leung-Malik (LM) filter [3] has different variances along x and y. Therefore, the Gaussian kernel formula can be rewritten as Equation 4:

$$g(x,y) = \frac{1}{2\pi\sigma_x\sigma_y} e^{-(\frac{x^2}{\sigma_x{}^2} + \frac{y^2}{\sigma_y{}^2})} \tag{4}$$

then the first (DoG) and second derivative (Laplacian of Gaussian, LoG) of the Gaussian kernel can be calculated analytically as Equation 5 :

$$\begin{aligned} \nabla g(x,y) &= -\frac{x}{\sigma_x{}^2}(-\frac{y}{\sigma_y{}^2})\frac{1}{2\pi\sigma_x\sigma_y} e^{-(\frac{x^2}{\sigma_x{}^2} + \frac{y^2}{\sigma_y{}^2})} \\ \nabla^2 g(x,y) &= -\frac{1}{\pi\sigma_x{}^2\sigma_y{}^2}(1 - \frac{x^2}{2\sigma_x{}^2} - \frac{y^2}{2\sigma_y{}^2}) e^{(\frac{x^2}{\sigma_x{}^2} + \frac{y^2}{\sigma_y{}^2})} \end{aligned} \tag{5}$$

Correspondingly, the mixed first and second derivative filters can be calculated as Equation 6:

$$\begin{aligned} \frac{\partial g(x,y)}{\partial x} &= -\frac{x}{\sigma_x{}^2}(-\frac{y}{\sigma_y{}^2})\frac{1}{2\pi\sigma_x\sigma_y} e^{-(\frac{x^2}{\sigma_x{}^2} + \frac{y^2}{\sigma_y{}^2})} \\ \frac{\partial^2 g(x,y)}{\partial x^2} &= \frac{x^2 - \sigma_x{}^2}{\sigma_x{}^4}\frac{1}{2\pi\sigma_x\sigma_y} e^{-(\frac{x^2}{\sigma_x{}^2} + \frac{y^2}{\sigma_y{}^2})} \end{aligned} \tag{6}$$

To obtain oriented filters, I can rotate the x and y coordinates by angle θ and substitute the rotated coordinates into Equation 5 and Equation 6. The rotated coordinates can be denoted as:

$$\begin{bmatrix} x_r \\ y_r \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Eventually, with different σ values and orientations, I can obtain the LMS and LML filters shown in Figure 2 and Figure 3:

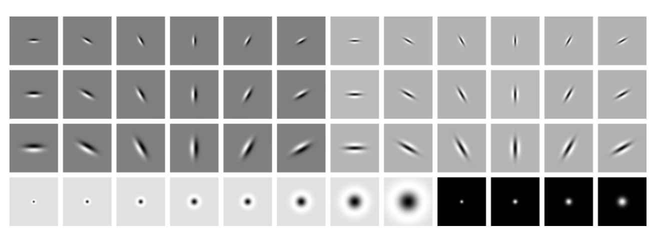


Fig. 2: Leung-Malik Small filter bank

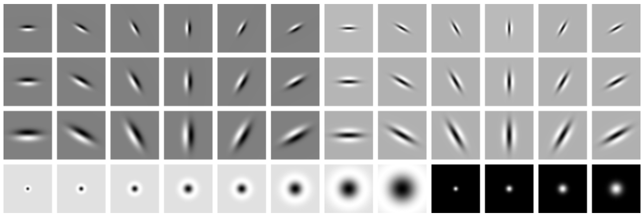


Fig. 3: Leung-Malik Large filter bank

3) Gabor Filters: The Gabor filters can be calculated using Equation 7[4]:

$$
\begin{aligned}
f(x, y, \theta) &= e^{-\frac{x'^2 + y'^2}{2\sigma^2}} \cos{(2\pi \frac{x'}{\lambda} + \psi)} \\
x' &= x \cos\theta + y \sin\theta \\
y' &= -x \sin\theta + y \cos\theta \\
\lambda &= \sigma^{1.1} \\
\psi &= 0 \\
\gamma &= 1
\end{aligned}
\tag{7}
$$

where θ is the rotation angle.

Then, with different σ values and orientations, I can obtain the Gabor filters shown in Figure 4:

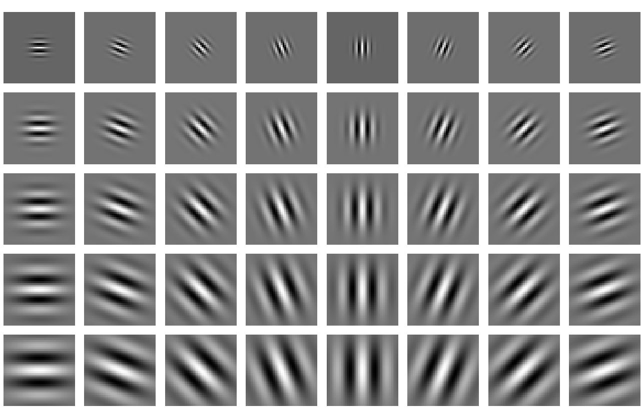
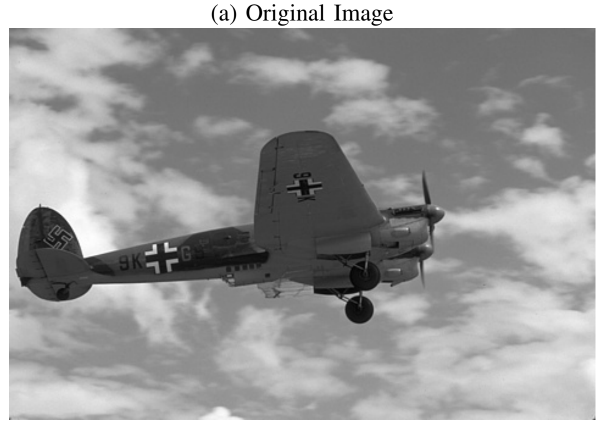


Fig. 4: Gabor filter bank

B. Texton Map

To obtain the texton map, I need to firstly implement the above filter banks to the images. To implement the filters, I need to firstly grayscale the input image and convolute the filter onto the input image. Here is an example image:

(a) Original Image



(b) Grayscale Image

Fig. 5: Example Input Image

Then, I implement the filter banks using convolution one by one:

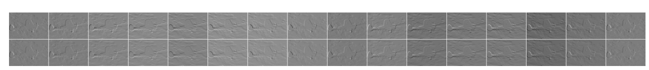


Fig. 6: Implement DoG filter bank

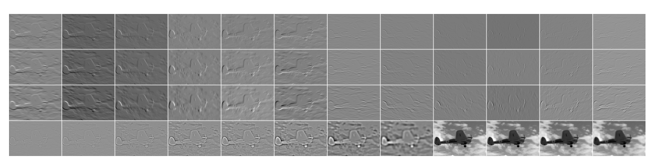


Fig. 7: Implement Leung-Malik Small filter bank

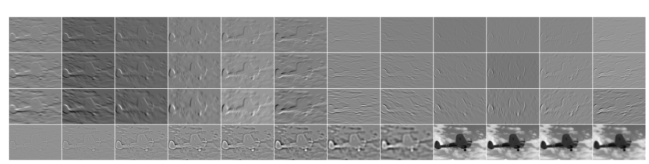


Fig. 8: Implement Leung-Malik Large filter bank

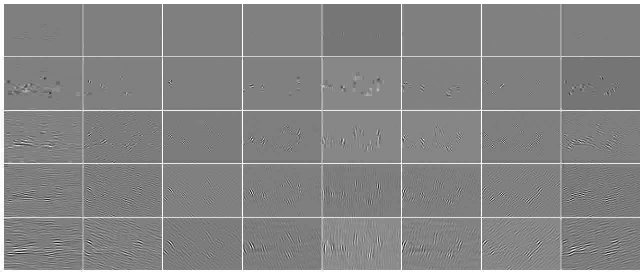


Fig. 9: Implement Gabor filter bank

The filter responses for DoG, LMS, LML and Gabor filters are shown in Figure 6, Figure 7, Figure 8 and Figure 9. Then, concentrate all the filter responses into a $N \times W \times H$ array. $N = 168$ is the total number of filters, W and H are the dimensions of the input image. Next, the filter responses of the images are then clustered into the $K = 64$ textons using K-means algorithm. Then, the texton map $\mathcal{T}$ of the example image is shown in Figure 10:

Fig. 10: Texton Map $\mathcal{T}$ of the example image

Moreover, generate the texton maps $\mathcal{T}$ of all images and visualize in Figure 11:

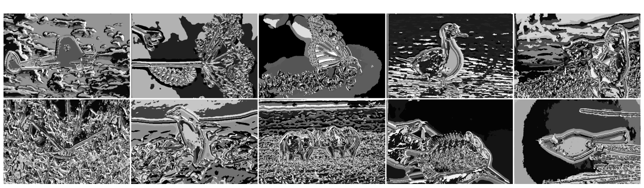


Fig. 11: Texton Map $\mathcal{T}$ of all images

C. Brightness Map

To obtain the brightness map, I firstly need to convert the image to the Lab[5] color space. The L channel represents the brightness of the image. Then I perform K-means algorithm with $K = 16$ clusters on the input data. Eventually, the brightness map $\mathcal{B}$ of all images are obtained and shown in Figure 12:

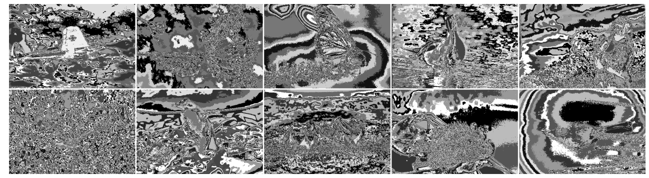


Fig. 12: Brightness Map $\mathcal{B}$ of all images

D. Color Map

To obtain the color map, I firstly get the RGB channels of the images directly using `cv2.imread()`. Then, I feed the RGB channel data to K-means algorithm with $K = 16$. Eventually, the color map $\mathcal{C}$ of all images are obtained and shown in Figure 13:

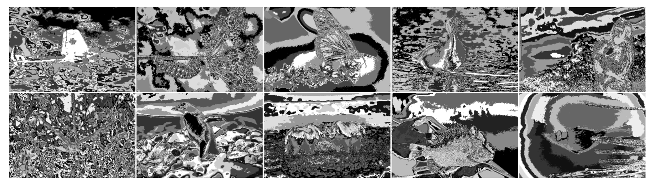


Fig. 13: Color Map $\mathcal{C}$ of all images

E. Texture, Brightness, Color Gradients

To calculate the texture gradient $\mathcal{T}_g$, brightness gradient $\mathcal{B}_g$ and color gradient $\mathcal{C}_g$, I will follow the instruction[2] using half-disc masks. The half-disc masks are pairs of binary images (HD_{left}, HD_{right}) of half-discs at different orientations and scales. The half-disc masks implemented are shown in Figure 14, and I resize the kernels for better visualization.

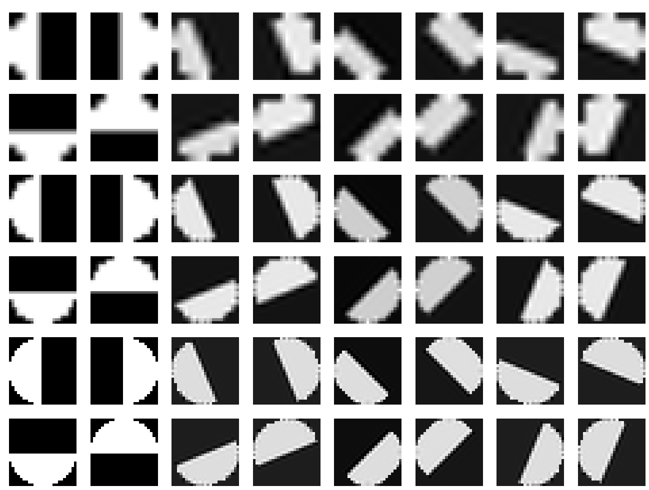


Fig. 14: Half disc masks

To calculate out the gradients, I will implement algorithm 1. K represents the number of clusters in K-means algorithm, and $\circledast$ represents convolution operation.

Moreover, calculated texture gradient $\mathcal{T}_g$, brightness gradient $\mathcal{B}_g$ and color gradient $\mathcal{C}_g$ for Figure 5 are shown in Figure 15, Figure 16 and Figure 17.

Algorithm 1: Gradient Calculation Algorithm

Input: Texton Map $\mathcal{T}$, Brightness Map $\mathcal{B}$ or Color Map $\mathcal{C}$, shown as `img`

Output: Texture Gradient $\mathcal{T}_g$, Brightness Gradient $\mathcal{B}_g$ or Color Gradient $\mathcal{C}_g$, shown as χ^2

$\chi^2 = \texttt{img} * 0$

for $i = 1 : K$ **do**

  $\texttt{tmp} = \texttt{img} * 0$

  Implement following if-statement for every pixel p in `img` :

  if $\texttt{img}[p] = i$ **then**

    $\texttt{tmp}[p] == 1$

  end if

  $g_i = HD_{left} \circledast \texttt{tmp}$

  $h_i = HD_{right} \circledast \texttt{tmp}$

  ${\chi_i}^2 = \frac{(g_i - h_i)^2}{2(g_i + h_i)}$

  $\chi^2 += {\chi_i}^2$

end for

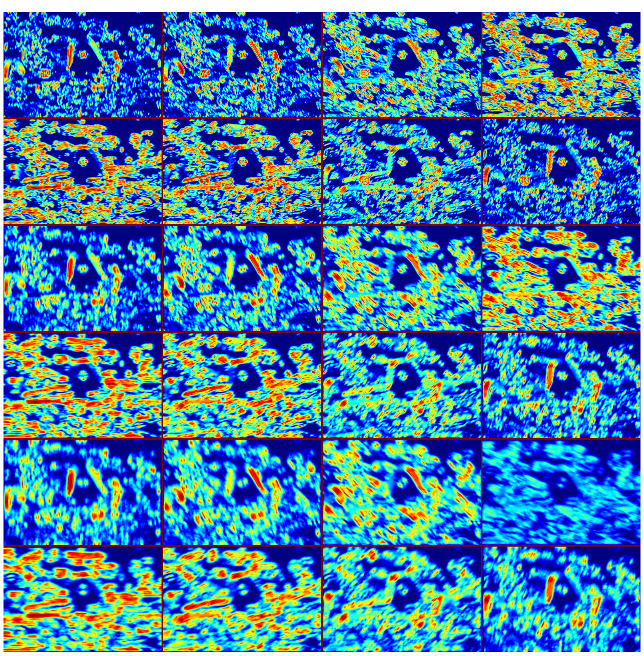


Fig. 15: Texture Gradient $\mathcal{T}_g$ of the example image

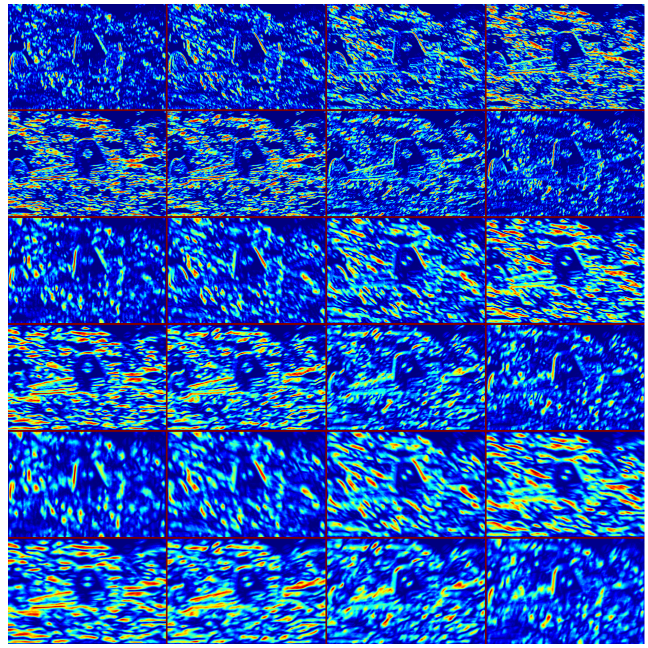


Fig. 16: Brightness Gradient $\mathcal{B}_g$ of the example image

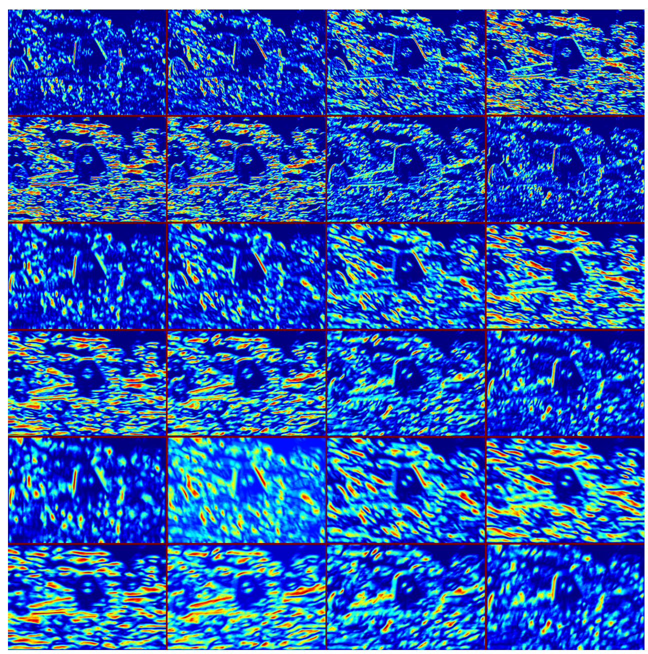


Fig. 17: Color Gradient $\mathcal{C}_g$ of the example image

F. Sobel and Canny baselines

The Sobel and Canny[6] baseline results are shown in Figure 18 and Figure 19:

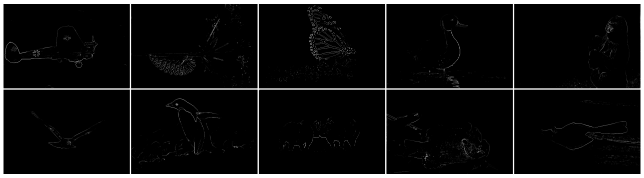


Fig. 18: Sobel Baseline

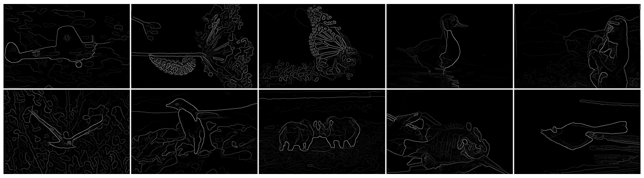


Fig. 19: Canny Baseline

G. Pb-lite Output

To obtain the edge based on Pb-lite[7], I can use a simple equation as shown in Equation 8:

$$Pb_{edge} = \frac{\mathcal{T}_g + \mathcal{B}_g + \mathcal{C}_g}{3} \odot (0.5 * Pb_{canny} + 0.5 * Pb_{sobel}) \tag{8}$$

Therefore, the detected edge using Pb-lite is shown in Figure 20:

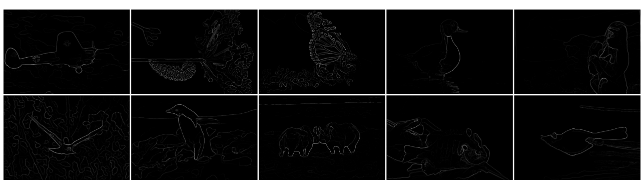


Fig. 20: Pb-Lite Baseline

Furthermore, the ground truth is shown in Figure 21:

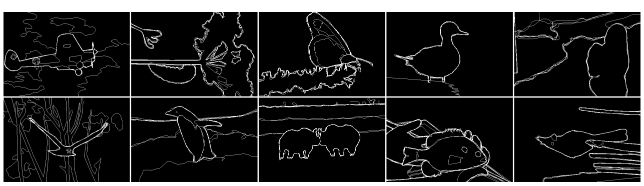


Fig. 21: Pb-Lite Baseline

Comparing the above results, I can see that false positive edges of the Canny and Sobel baselines are suppressed in the Pb-Lite baseline while true edges still remain. This is because Pb-Lite baseline is able to use the global information of the image and also combine multi-scale information [7].

II. PHASE2: DEEP DIVE ON DEEP LEARNING

For all neural networks trained in this section, I will use stochastic gradient descent(SGD)[8] as the optimization method and cross-entropy loss[9] as the loss function. The number of epochs is 25, expect for the improving accuracy section (which is 40).

Note: some drop-out layer may not be presented on the network architectures. And if you cannot see the architecture, you may find the corresponding <model_name>.png in the attached submission.

A. Train your first neural network

The first neural network designed is a simple convolutional neural network(CNN). The CNN architecture is shown Figure 22.

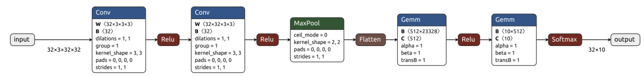


Fig. 22: CNN Architecture

There are 11969866 parameters in this model. I use a stochastic gradient decent optimizer for learning, with a learning rate $lr = 0.01$ and weight decay $decay = 0.0004$ and a batch size of 32. The train and test accuracy over epochs are visualized in Figure 23 and Figure 24. Loss over epochs is visualized in Figure 25.

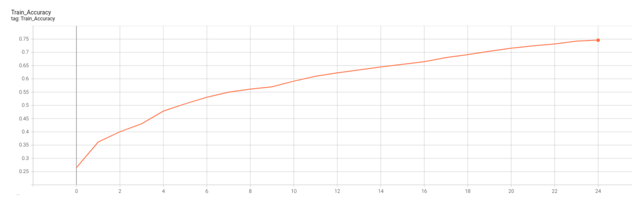


Fig. 23: CNN Train Accuracy over Epochs

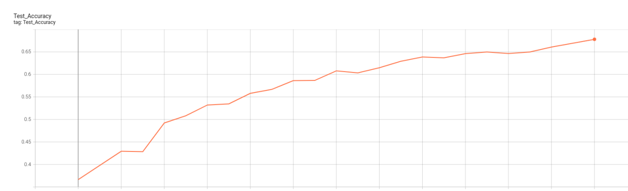


Fig. 24: CNN Test Accuracy over Epochs

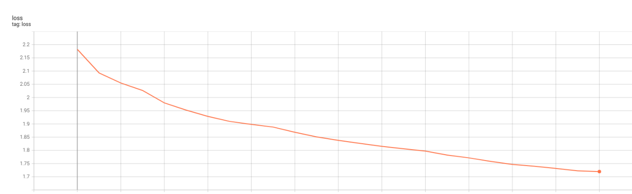


Fig. 25: CNN Loss over Epochs

The confusion matrix of the trained model on training data is:

$$\begin{bmatrix} 4080 & 106 & 127 & 32 & 55 & 32 & 44 & 57 & 308 & 159 \\ 69 & 4432 & 24 & 13 & 15 & 23 & 39 & 27 & 108 & 250 \\ 253 & 31 & 3390 & 131 & 305 & 256 & 332 & 151 & 90 & 61 \\ 116 & 25 & 243 & 3143 & 161 & 672 & 324 & 154 & 91 & 71 \\ 149 & 19 & 190 & 122 & 3716 & 145 & 230 & 310 & 62 & 57 \\ 55 & 27 & 180 & 318 & 175 & 3769 & 158 & 216 & 45 & 57 \\ 28 & 26 & 133 & 72 & 109 & 91 & 4457 & 26 & 26 & 32 \\ 43 & 11 & 95 & 96 & 126 & 224 & 37 & 4283 & 24 & 61 \\ 231 & 94 & 41 & 22 & 12 & 21 & 24 & 15 & 4441 & 99 \\ 97 & 243 & 21 & 34 & 15 & 39 & 28 & 41 & 108 & 4374 \end{bmatrix}$$

The confusion matrix of the trained model on testing data is:

$$\begin{bmatrix} 749 & 19 & 40 & 13 & 9 & 9 & 19 & 13 & 83 & 46 \\ 19 & 809 & 6 & 8 & 7 & 6 & 9 & 4 & 23 & 109 \\ 85 & 7 & 511 & 34 & 84 & 87 & 101 & 48 & 23 & 20 \\ 33 & 17 & 75 & 376 & 63 & 230 & 92 & 60 & 26 & 28 \\ 43 & 4 & 66 & 52 & 548 & 42 & 104 & 111 & 25 & 5 \\ 21 & 1 & 68 & 101 & 40 & 634 & 35 & 66 & 22 & 12 \\ 6 & 9 & 42 & 35 & 27 & 27 & 824 & 9 & 11 & 10 \\ 21 & 2 & 31 & 27 & 46 & 71 & 6 & 765 & 4 & 27 \\ 80 & 46 & 8 & 8 & 3 & 8 & 9 & 11 & 792 & 35 \\ 33 & 87 & 4 & 14 & 4 & 9 & 10 & 27 & 40 & 772 \end{bmatrix}$$

B. Improving Accuracy of your neural network

The improve CNN architecture is almost identical to the CNN architecture. The only difference is the batch size. And the improve CNN architecture is shown in Figure 26.

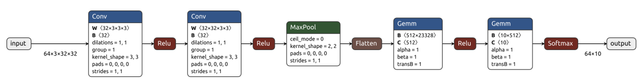


Fig. 26: Improved CNN Architecture

Multiple approaches are implemented to improve the accuracy of the original CNN:

- Standardize the data input. The data is normalized from [0, 255] to [0, 1].
- Increase the batch size to 64. [10]
- Increase the number of epochs to 40.

There are 11969866 parameters in this model. I use a stochastic gradient decent optimizer for learning, with a learning rate $lr = 0.01$ and weight decay $decay = 0.00025$ and a batch size of 64. The train and test accuracy over epochs are visualized in Figure 27 and Figure 28. Loss over epochs is visualized in Figure 29.

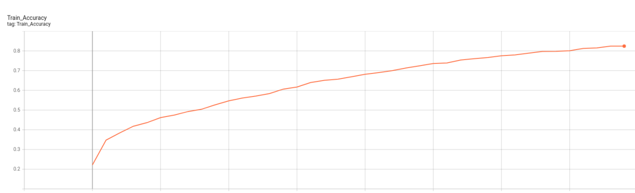


Fig. 27: Improved CNN Train Accuracy over Epochs

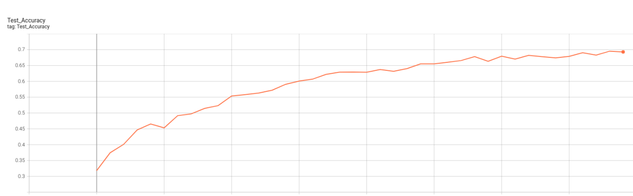


Fig. 28: Improved CNN Test Accuracy over Epochs

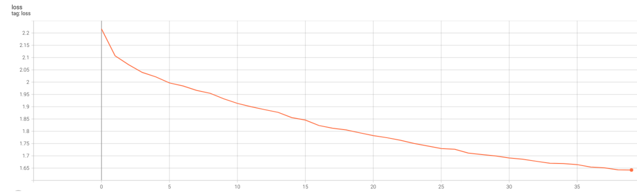


Fig. 29: Improved CNN Loss over Epochs

The confusion matrix of the trained model on training data is:

$$\begin{bmatrix} 4496 & 51 & 108 & 31 & 54 & 24 & 12 & 41 & 151 & 92 \\ 47 & 4705 & 20 & 19 & 9 & 11 & 15 & 22 & 52 & 100 \\ 154 & 22 & 4070 & 93 & 253 & 106 & 131 & 84 & 51 & 36 \\ 58 & 21 & 198 & 3995 & 163 & 254 & 97 & 120 & 46 & 48 \\ 88 & 9 & 138 & 88 & 4356 & 66 & 49 & 163 & 19 & 24 \\ 30 & 18 & 168 & 241 & 163 & 4104 & 65 & 153 & 25 & 33 \\ 21 & 22 & 137 & 72 & 95 & 49 & 4535 & 15 & 26 & 28 \\ 32 & 8 & 108 & 78 & 118 & 102 & 18 & 4498 & 14 & 24 \\ 144 & 55 & 61 & 15 & 17 & 10 & 7 & 18 & 4620 & 53 \\ 79 & 143 & 32 & 21 & 17 & 23 & 7 & 29 & 63 & 4586 \end{bmatrix}$$

The confusion matrix of the trained model on testing data is:

$$\begin{bmatrix} 754 & 26 & 53 & 17 & 13 & 6 & 9 & 12 & 65 & 45 \\ 18 & 810 & 14 & 9 & 6 & 5 & 6 & 5 & 27 & 100 \\ 66 & 7 & 581 & 48 & 117 & 44 & 58 & 46 & 21 & 12 \\ 26 & 19 & 100 & 454 & 84 & 152 & 52 & 62 & 15 & 36 \\ 25 & 3 & 73 & 48 & 664 & 31 & 44 & 92 & 15 & 5 \\ 19 & 5 & 82 & 151 & 57 & 580 & 20 & 70 & 9 & 7 \\ 9 & 8 & 54 & 53 & 64 & 30 & 743 & 13 & 9 & 17 \\ 12 & 3 & 49 & 29 & 59 & 51 & 2 & 776 & 4 & 15 \\ 81 & 40 & 19 & 8 & 3 & 5 & 4 & 4 & 805 & 25 \\ 35 & 94 & 15 & 14 & 4 & 6 & 8 & 30 & 39 & 761 \end{bmatrix}$$

C. ResNet, ResNeXt, DenseNet

1) ResNet: The ResNet architecture[11] is shown in Figure 30.

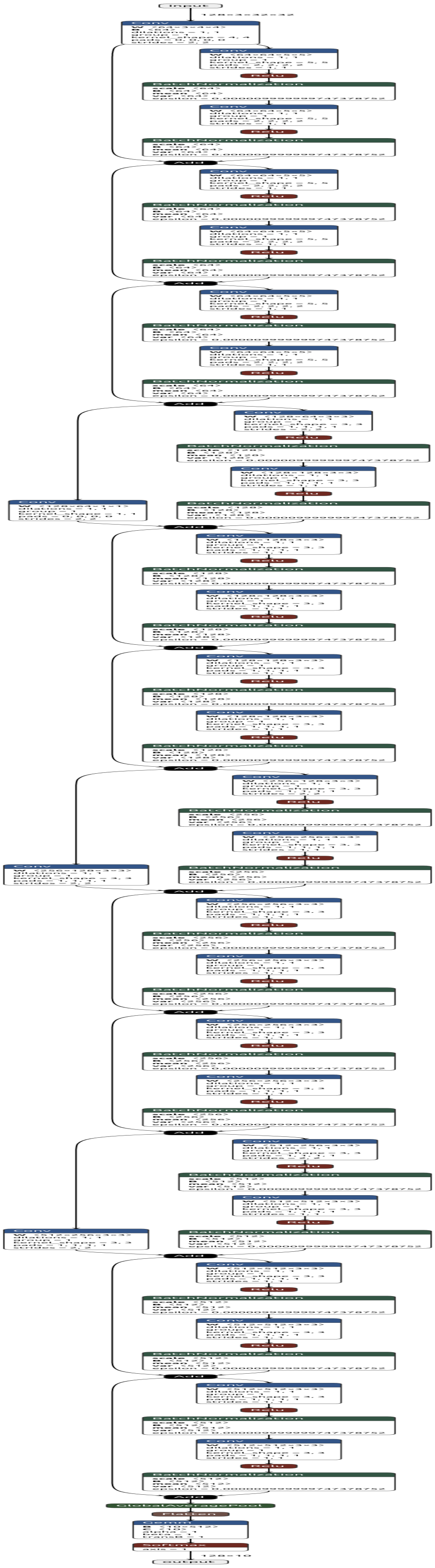


Fig. 30: ResNet Architecture, may check attachment for a better view

There are 19148106 parameters in this model. I use a stochastic gradient decent optimizer for learning, with a learning rate $lr = 0.01$ and weight decay $decay = 0.0004$ and a batch size of 128. The train and test accuracy over epochs are visualized in Figure 31 and Figure 32. Loss over epochs is visualized in Figure 33.

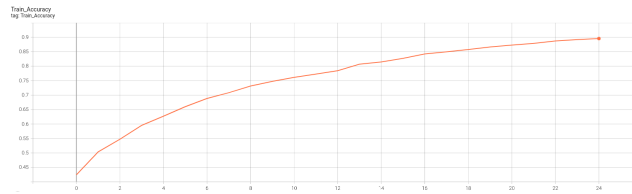


Fig. 31: ResNet Train Accuracy over Epochs

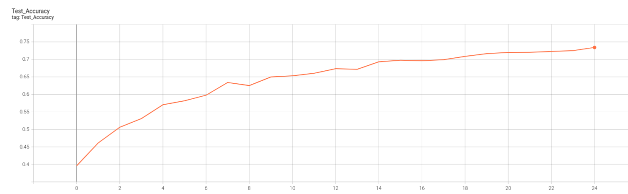


Fig. 32: ResNet Test Accuracy over Epochs

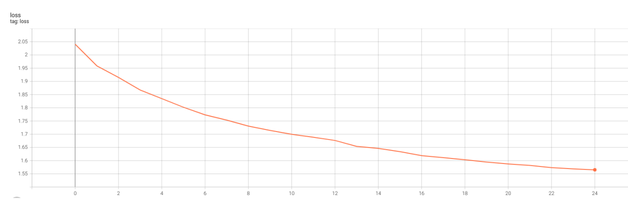


Fig. 33: ResNet Loss over Epochs

The confusion matrix of the trained model on training data is:

```
[4367   20  216   35   33   24   14   22  216   53] (0)
[  32 4745   15   14    5   23   14    7   68   77] (1)
[ 128   11 4283  142  111  120   92   47   44   22] (2)
[  57   13  162 3972   81  418  106   70   69   52] (3)
[  67    2  172  164 4215  134   40  156   35   15] (4)
[  36    8  118  296   94 4199   50  149   23   27] (5)
[  21   11  130  146  111  107 4402   25   27   20] (6)
[  32    3   90  101   85  110   11 4544   15    9] (7)
[  69   43   34    9   12   14    6   10 4777   26] (8)
[  78  165   28   21   10   23    6   26   81 4562] (9)
 (0)  (1)  (2)  (3)  (4)  (5)  (6)  (7)  (8)  (9)
```

The confusion matrix of the trained model on testing data is:

```
[754  14  69  20  12   3   5  13  84  26] (0)
[ 22 837   8  10   3  10  11   3  33  63] (1)
[ 47   3 688  64  60  54  37  25  14   8] (2)
[ 30   5  64 531  46 191  51  35  30  17] (3)
[ 20   4  81  80 649  36  33  79  16   2] (4)
[ 12   6  43 142  32 667  18  59  13   8] (5)
[  5   6  41  73  38  48 756  13  10  10] (6)
[ 29   3  32  35  43  50   4 794   3   7] (7)
[ 33  19  15  11   5   6   1   3 892  15] (8)
[ 40  77  14  17   0  11   4  19  44 774] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
```

2) ResNeXt: The ResNeXt architecture[12] is shown in Figure 34.

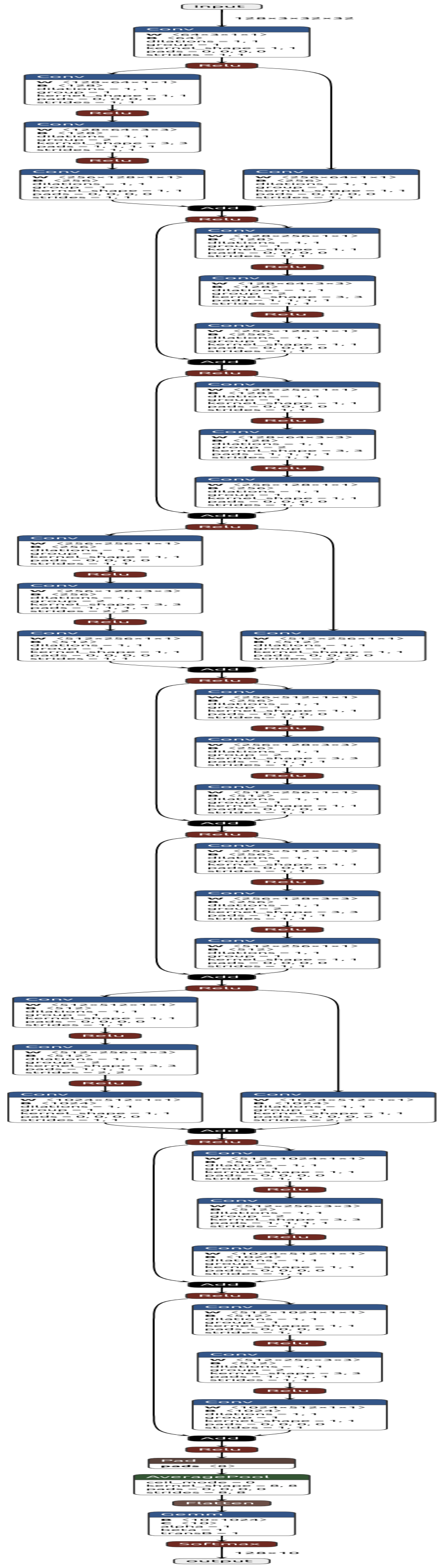


Fig. 34: ResNeXt Architecture, may check attachment for a better view

There are 9128778 parameters in this model. I use a stochastic gradient decent optimizer for learning, with a learning rate $lr = 0.01$ and weight decay $decay = 0.0004$ and a batch size of 128. The train and test accuracy over epochs are visualized in Figure 35 and Figure 36. Loss over epochs is visualized in Figure 37.

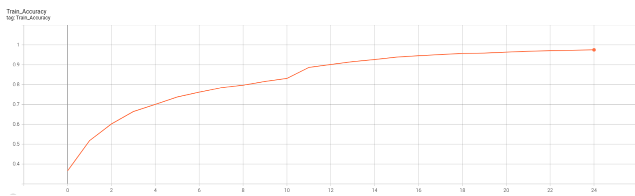


Fig. 35: ResNeXt Train Accuracy over Epochs

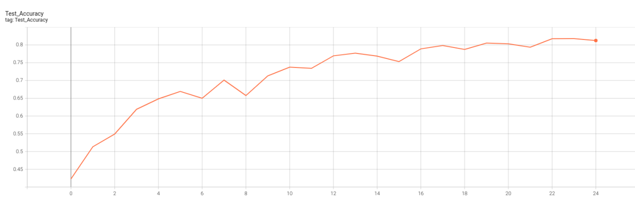


Fig. 36: ResNeXt Test Accuracy over Epochs

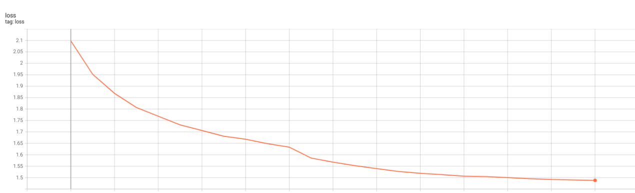


Fig. 37: ResNeXt Loss over Epochs

The confusion matrix of the trained model on training data is:

[4850	5	54	11	6	13	5	15	10	31]	(0)
[7	4894	6	3	0	12	2	1	5	70]	(1)
[47	0	4807	9	15	54	41	19	3	5]	(2)
[8	0	89	4175	50	544	84	34	4	12]	(3)
[10	1	62	25	4782	43	33	38	4	2]	(4)
[2	0	49	48	31	4794	23	49	2	2]	(5)
[4	1	45	14	5	56	4866	5	1	3]	(6)
[7	1	35	18	17	61	8	4851	1	1]	(7)
[28	13	11	2	2	5	5	4	4920	10]	(8)
[14	13	17	2	0	13	4	4	8	4925]	(9)
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	

The confusion matrix of the trained model on testing data is:

[806	12	70	9	10	6	11	15	28	33]	(0)
[9	891	6	2	3	9	6	1	5	68]	(1)
[25	1	776	11	37	67	66	12	2	3]	(2)
[6	4	86	502	32	268	68	18	8	8]	(3)
[8	1	89	15	758	40	41	44	4	0]	(4)
[3	0	38	41	25	841	17	29	1	5]	(5)
[5	2	36	7	11	46	888	4	1	0]	(6)
[4	1	31	10	21	68	4	852	1	8]	(7)
[48	12	14	5	2	4	3	1	893	18]	(8)
[14	23	10	8	3	5	3	8	9	917]	(9)
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	

3) DenseNet: The DenseNet architecture[13] is shown in `densenet.png`. It is too large to be put in the report, please check the attachment to get a better view.

There are 6956298 parameters in this model. I use a stochastic gradient decent optimizer for learning, with a learning rate $lr = 0.01$ and weight decay $decay = 0.0004$ and a batch size of 128. The train and test accuracy over epochs are visualized in Figure 38 and Figure 39. Loss over epochs is visualized in Figure 40.

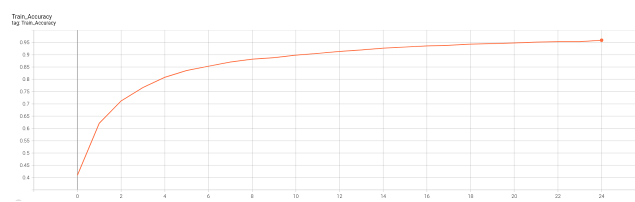


Fig. 38: DenseNet Train Accuracy over Epochs

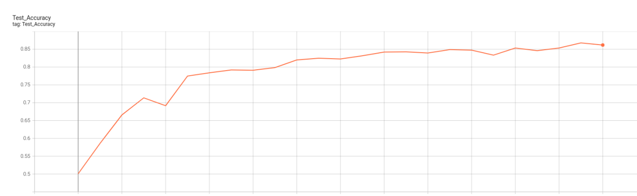


Fig. 39: DenseNet Test Accuracy over Epochs

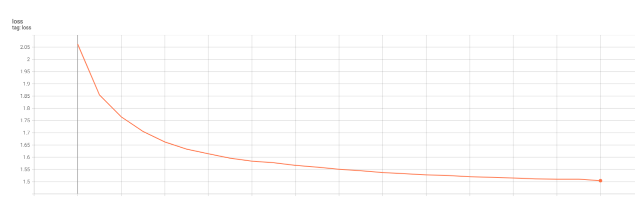


Fig. 40: DenseNet Loss over Epochs

The confusion matrix of the trained model on training data is:

```
[4787    4   84   18   16   20    8   35   22    6] (0)
[  22 4929    6   10    4    3    3    6    5   12] (1)
[  74    1 4609   80   51   75   82   28    0    0] (2)
[   8    1   46 4579   48  253   51   12    2    0] (3)
[  10    0   76   78 4727   45   36   27    1    0] (4)
[   3    4   30  195   39 4677   11   41    0    0] (5)
[   7    0   34   48   12   37 4857    4    1    0] (6)
[   2    0   26   91   26  132    6 4717    0    0] (7)
[ 130   12   37   28   15    6   23   16 4721   12] (8)
[  33  118   16   28   13   16   10   32    3 4731] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
```

The confusion matrix of the trained model on testing data is:

```
[890   4  44  12   8   2   7  16  11   6] (0)
[  6 959   5   4   1   0   1   2   5  17] (1)
[ 35   0 799  39  28  40  49  10   0   0] (2)
[  6   2  24 782  23 109  38  11   3   2] (3)
[  5   0  49  38 838  25  23  21   1   0] (4)
[  3   0  20  85  18 854   6  14   0   0] (5)
[  7   0  17  33   5  20 917   0   1   0] (6)
[  4   0  11  32  27  59   4 862   0   1] (7)
[ 67  14  17   7   9   4   6   4 863   9] (8)
[ 16  67   6  12   7   6   6  13  13 854] (9)
 (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
```

D. Comparison

All neural network performance comparsion is shown in Table I.

Network	Parameter Number	Train Accuracy	Test Accuracy	Inference Time
CNN	11,969,866	80.17 %	67.8 %	0.294 ms
Improved CNN	11,969,866	87.81 %	69.28 %	0.280 ms
ResNet	19,148,866	88.132 %	73.42 %	2.948 ms
ResNeXt	9,128,778	95.728 %	81.24 %	2.839 ms
DenseNet	6,956,298	94.668 %	86.18 %	12.081 ms

TABLE I: Neural Network Performance

According to the given performance, my opinion is that ResNeXt is the better than the others.

Firstly, the two CNN have relatively poor performance: they use a large number of parameters but achieve relatively low train and test accuracy. Also, their convergence rates are low. Then, ResNet is a little bit better than the two CNN: it uses a large number of parameters and achieves fairly good train and test accuracy, their convergence is also fairly well. Both ResNeXt and DenseNet have amazing performances and they achieve high train and test accuracy. Nevertheless, even though DenseNet has been shown to have better feature use efficiency, outperforming ResNeXt with fewer parameters, DenseNet requires heavy GPU memory due to concatenation operations and it is not memory-efficient.

Therefore, ResNeXt is a better choice compared to the other neural networks introduced above.

References

[1] Wikipedia. *Sobel operator*. URL: https://en.wikipedia.org/wiki/Sobel_operator.

[2] Nitin J. Sanket, Lening Li, and Gejji Vaishnavi Vivek. *HW0 Guidence*. URL: https://rbe549.github.io/fall2022/hw/hw0/.

[3] University of Oxford. *The Leung-Malik(LM) Bank*. URL: https://www.robots.ox.ac.uk/~vgg/research/texclass/filters.html.

[4] Wikipedia. *Gabor filter*. URL: https://en.wikipedia.org/wiki/Gabor_filter.

[5] Mark D Fairchild and Garrett M Johnson. "Image appearance modeling". In: *Human Vision and Electronic Imaging VIII* 5007 (2003), pp. 149–160.

[6] John Canny. "A Computational Approach to Edge Detection". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.6 (1986), pp. 679–698. DOI: 10.1109/TPAMI.1986.4767851.

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

[8] Wikepedia. *Stochastic gradient descent*. URL: https://en.wikipedia.org/wiki/Stochastic_gradient_descent.

[9] Wikepedia. *Cross Entropy*. URL: https://en.wikipedia.org/wiki/Cross_entropy.

[10] Samuel L Smith et al. "Don't decay the learning rate, increase the batch size". In: *arXiv preprint arXiv:1711.00489* (2017).

[11] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[12] Saining Xie et al. "Aggregated residual transformations for deep neural networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1492–1500.

[13] Gao Huang et al. "Densely connected convolutional networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4700–4708.