Homework 0 - Alohomora
CMSC 733

Ameya Patil
Department of Computer Science
University of Maryland
College Park, Maryland 20740
Email: ameyap@umd.edu

I. Phase 1

The objective of the first phase is to implement the Pb-Lite edge detection algorithm. 'Pb' stands for probability of boundary and accordingly gives the probability of each pixel of an input image, belonging to a true edge in the image. The algorithm is implemented in 4 steps:

A) Filtering the input image to get textures
B) Quantizing the per pixel texture, brightness and color attributes
C) Finding the gradients of texture, brightness and color for each pixel
D) Combining the result with the baseline Canny or Sobel edge detection results

Following is a short description of each of the four steps:

A. Filtering the image

Filter banks were created with filters of different characteristics and sizes with the aim of identifying edges in the image. The filters have a particular type of impulse response which enables them to do so. Textures are although repetitive patterns of edges, they are not necessarily strong edges in an image and can thus be falsely identified as edges. The process of identifying textures from edges will be explained in I-B and I-C

For this homework, 3 filter banks were created:

A) Derivative of Gaussian Filters
B) Leung-Malik Filters
C) Gabor Filters

All these filters have the gaussian function common among them, with different values for the standard deviation (scale) σ and different angles of rotation. Multiple scale values help in detecting edges of different thicknesses, while multiple angles of rotation help in detecting edges along different direction. Each filter bank differs with respect to the modification applied to the basic gaussian function.

1) Derivative of Gaussian: As the name suggests, this filter bank has the first order derivative of the 2D gaussian function. The first order derivative is computed by convolving the gaussian kernel with the Sobel filter. Figure 1

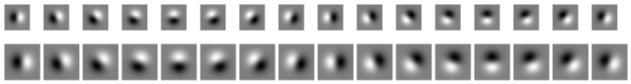


Fig. 1. Derivative of Gaussian Filter Bank - with 2 different scales and 16 different orientations for each scale

2) Leung-Malik: This filter bank consists of 4 different types of filters - the gaussian filter, first order derivative of gaussian, second order derivative of gaussian and the laplacian of gaussian. The derivative of gaussian filters have different standard deviation along the X and Y axes giving the filter an elongated shape. Further, these elongated filters are rotated to have 8 different orientations. A fixed set of 4 standard deviation values was used to create the filters and depending on those fixed set of values, 2 LM filter banks were implemented - LM Small and LM Large. Derivative of gaussian filters were created only for the first 3 scale values of the set. Gaussian and laplacian of gaussian were created for all the 4 scale values.

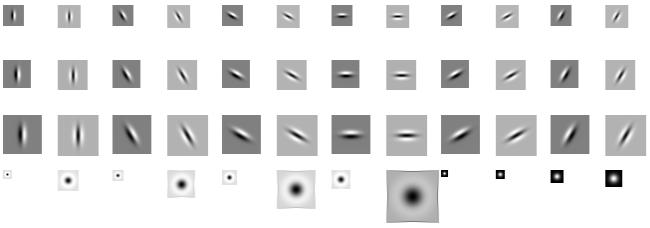


Fig. 2. LM Small filter bank having 48 filters. Top 3 rows show 8 different orientations of the first and second derivative of gaussian. First 8 filters in the last row are the laplacian of gaussian and the remaining 4 filters are gaussian filters. Standard deviation values of 1, $\sqrt{2}$, 2 and $2\sqrt{2}$ were used

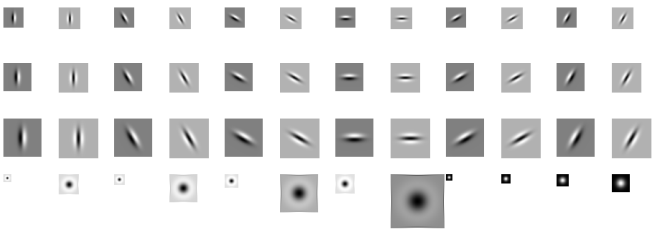


Fig. 3. LM Large filter bank having 48 filters. Top 3 rows show 8 different orientations of the first and second derivative of gaussian. First 8 filters in the last row are the laplacian of gaussian and the remaining 4 filters are gaussian filters. Standard deviation values of $\sqrt{2}$, 2, $2\sqrt{2}$ and 4 were used

3) Gabor Filters: The gabor filter has a gaussian kernel modulated with a sinusoidal plane kernel. The bank consists of gabor filters having different standard deviation for the gaussian and different frequency for the sinusoids, which are further rotated to get different orientations. Gabor filters with 3 different standard deviation values and 2 different sinusoid frequencies were implemented.

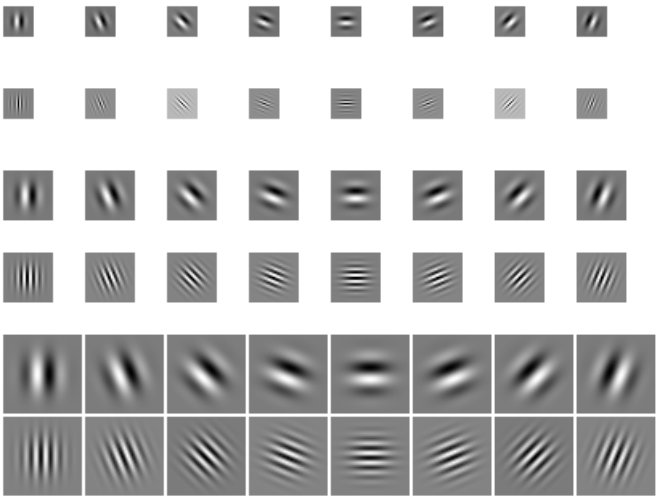


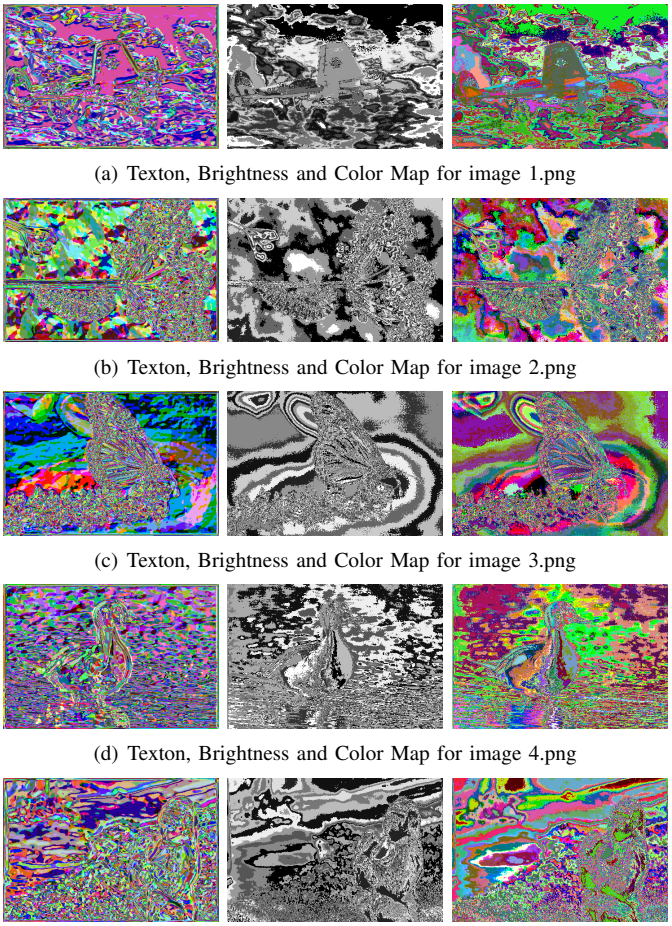
Fig. 4. Gabor filter bank having 8 different orientations (8 columns) for 3 different gaussian kernels and 2 different sinusoid kernels

B. Texture, Brightness and Color Maps

The next step is to get 2 different kinds of per-pixel map for the input image. The texton map or the texture map encodes the information of a certain pixel being a part of a texture in the image. This is done by convolving the image with the individual filters in the filter banks generated in I-A. The brightness and color maps accordingly encode the intensity and color values for each pixel. These maps are then processed with dimensionality reduction operation to get a lower dimension space for the maps. This is done using the k-means clustering algorithm. For the texture map, convolution was performed for all the 3 image channels - R,G,B. The color maps were also generated for all the 3 image channels. In the resulting texton and color maps, the 3 chanels were merged to create an RGB image. Figures 5 and 6 show the corresponding maps where the Derivative of Gaussian filter bank was used.

C. Finding Gradients of Texture, Brightness and Color Maps

The next step is to find the gradients of the texture, brightness and color at each pixel. This step is executed efficiently using half disk masks of varying sizes and orientations. Half disk masks are a set of pairs of masks having a semicircular disk centred at the centre of the mask, refer Figure 7. By applying pairs of half disk masks at each pixel of the maps generated in I-B via convolution, and computing the difference between the results of left mask and right mask application, we get a measure of the gradient of the attribute at the pixel.



(a) Texton, Brightness and Color Map for image 1.png

(b) Texton, Brightness and Color Map for image 2.png

(c) Texton, Brightness and Color Map for image 3.png

(d) Texton, Brightness and Color Map for image 4.png

(e) Texton, Brightness and Color Map for image 5.png

Fig. 5. Texture, Brightness anc Color Maps for Images 1 to 5

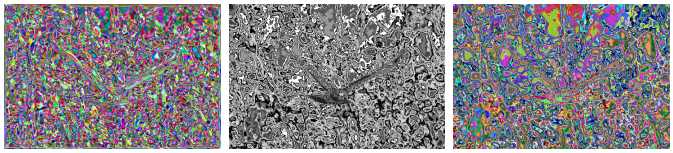
The difference between the left and right mask application is calculated using the χ^2 distance after binarizing the image over all the quantization bins of the individual map. The resulting gradient maps are shown in Figures 8 and 9

D. Combine with baselines

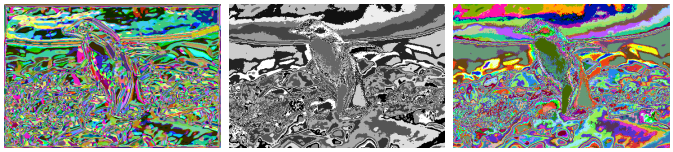
In the last step, the results obtained after finding the texture gradients, brightness gradients and color gradients, were combined with the Canny and Sobel baselines using the Hadamard matrix product, with different weights for each of the 5 components - texture gradient, brightness gradient, color gradient, sobel baseline and canny baseline. The texture gradient information scans for the existence of textures in the image and acts as a weightage for the baseline results to reduce the false negative edges in textures in the input image. Final output is as shown in Fig 10

Why is Pb-Lite better than Sobel and Canny edge detectors?

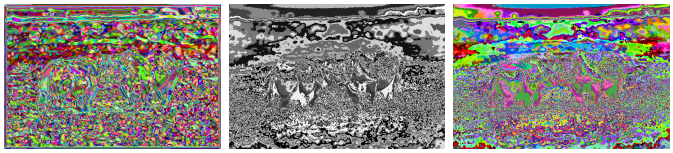
Pb-Lite algorithm gives control to the user as to how much visibility should be given to texture edges in the final output by changing the relative contribution of the texture gradient, brightness gradient and the color gradient. This makes it more



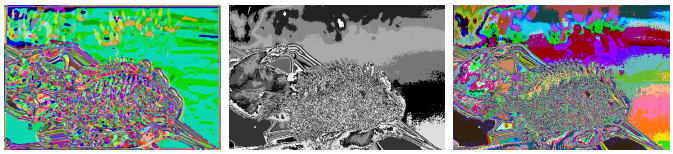
(a) Texton, Brightness and Color Map for image 6.png



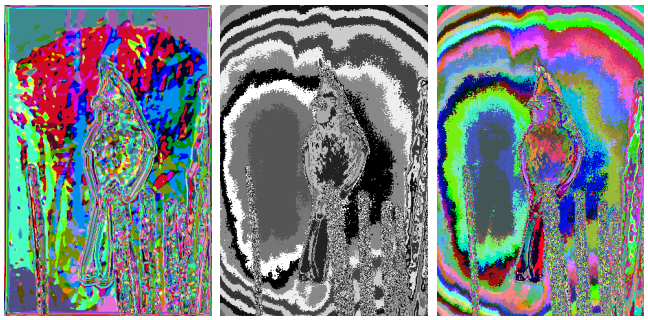
(b) Texton, Brightness and Color Map for image 7.png



(c) Texton, Brightness and Color Map for image 8.png



(d) Texton, Brightness and Color Map for image 9.png



(e) Texton, Brightness and Color Map for image 10.png

Fig. 6. Texture, Brightness anc Color Maps for Images 6 to 10

flexible and better compared to the Sobel and Canny edge detectors which necessarily use the derivative operator on images to find all kind of intensity and/or color changes.

II. Phase 2

The first task in phase 2 is about creating a convolutional network for image classification on the CIFAR-10 dataset. First, a simple convolutional network was created with 2 convolution layers having a ReLU activation function, followed by 2 fully connected layers and finally a softmax layer, as shown in image 11. The softmax cross entropy loss function was used on the network logits and it was optimized using the Adam optimizer. The network was trained for 10 epochs. This network is being referred to as the 'first' network in the succeeding text.

Network architecture:
Optimizer: Adam Optimizer

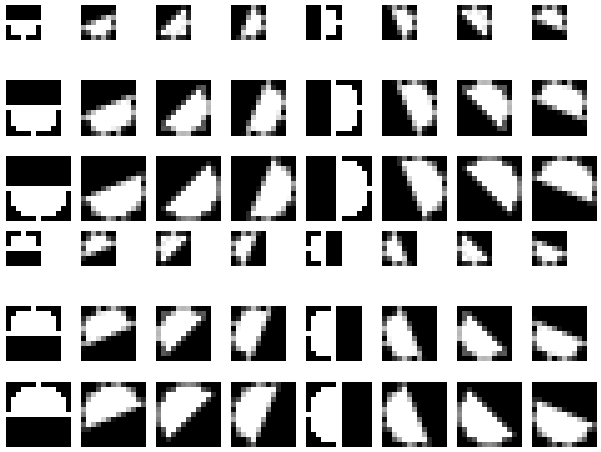


Fig. 7. Half Disk masks having 3 different sizes and 8 different orientations. First 3 rows show the first of the pairs while the last 3 rows show the corresponding second of the pairs

Class	0	1	2	3	4	5	6	7	8	9
0	4846	12	54	19	11	12	1	2	25	18
1	17	4889	4	17	2	1	4	2	11	53
2	55	6	4771	64	19	48	13	12	5	7
3	12	5	50	4708	16	164	16	12	13	4
4	59	7	179	90	4500	59	24	64	11	7
5	6	2	48	161	20	4661	24	67	3	8
6	13	17	92	218	48	67	4510	10	7	18
7	26	2	50	54	29	44	1	4772	1	21
8	84	16	22	24	4	2	1	0	4839	8
9	22	117	8	28	3	8	2	15	20	4777

TABLE I
CONFUSION MATRIX FOR THE 'FIRST' NETWORK ON TRAINING DATASET AFTER 10 EPOCHS

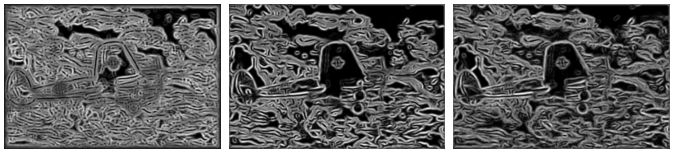
Learning Rate: 0.0001
Batch Size: 1 for all the epochs
Number of parameters in the model: 50451690

The per epoch loss and accuracy on train and test dataset is provided in Figure 12. The confusion matrices for 'first' network on train and test data after 10 epochs are at Tables I and II. Inference time on train dataset (50000 images) was 13 min 48 sec and that on the test dataset was 2 min 16 sec.

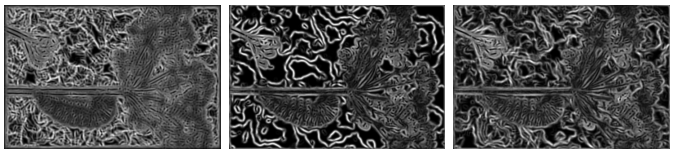
Enhanced Network - 3.4 A newer version of the same network was created with a few enhancements, referred to as

Class	0	1	2	3	4	5	6	7	8	9
0	655	27	103	53	15	10	10	11	72	44
1	36	705	10	31	6	11	6	6	45	144
2	93	9	443	132	85	108	51	45	18	16
3	34	25	107	439	58	210	40	50	8	29
4	37	11	180	135	372	72	63	109	13	8
5	21	11	96	257	54	421	37	83	8	13
6	25	16	92	183	71	62	492	25	17	17
7	31	16	59	98	65	81	11	591	7	41
8	119	50	20	45	12	3	7	6	690	48
9	62	183	11	49	7	11	6	20	40	611

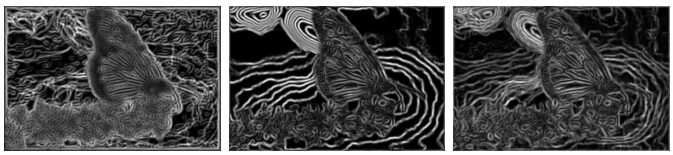
TABLE II
CONFUSION MATRIX FOR THE 'FIRST' NETWORK ON TEST DATASET AFTER 10 EPOCHS



(a) Texture, Brightness and Color Gradients for image 1.png

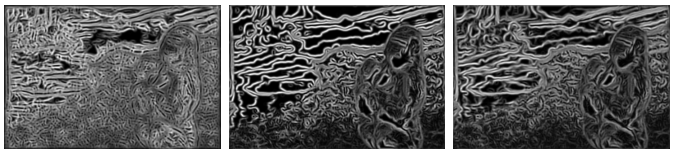


(b) Texture, Brightness and Color Gradients for image 2.png



(c) Texture, Brightness and Color Gradients for image 3.png

(d) Texture, Brightness and Color Gradients for image 4.png

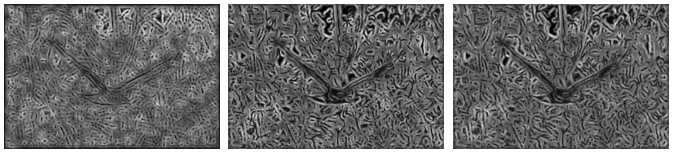


(e) Texture, Brightness and Color Gradients for image 5.png

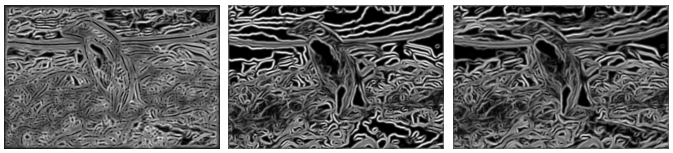
Fig. 8. Texture, Brightness and Color Gradients for Images 1 to 5

the 'enhanced' network in the succeeding text. The input data batch was standardized by mapping the pixel values between -1 and 1. This helped in ensuring that the ReLU activation did actually introduce some non-linearity in the system, otherwise if all the pixel values are between 0 and 255, the ReLU activation would not have much effect on the network output. Further, batch normalization layers were added to each of convolution layers in the network before applying the ReLU activation. The idea behind batch normalization is to reduce the internal covariate shift of the data being passed from one convolution layer to the next. This helps in faster convergence as the learning rate can be increased, because the parameters in each network only need to be adjusted slightly to accommodate the activations for different inputs. In the presence of internal covariate shift, the parameters in each layer might need huge modifications for every forward pass, thus needing low learning rate. The network was trained for 10 epochs and the batch size was increased by 100 after every 3 epochs. Increasing the batch size after a few epochs helps in reaching convergence faster as updates to the parameters are not made for each sample but are aggregated over a batch of samples.

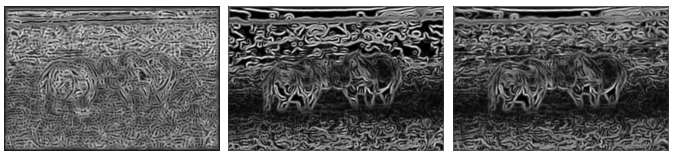
Network architecture:



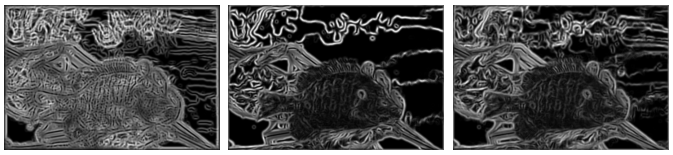
(a) Texture, Brightness and Color Gradients for image 6.png



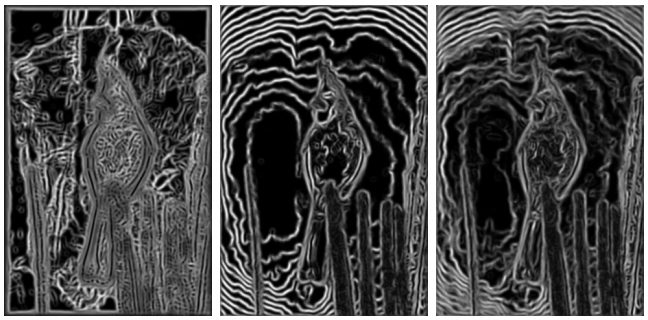
(b) Texture, Brightness and Color Gradients for image 7.png



(c) Texture, Brightness and Color Gradients for image 8.png



(d) Texture, Brightness and Color Gradients for image 9.png



(e) Texture, Brightness and Color Gradients for image 10.png

Fig. 9. Texture, Brightness and Color Gradients for Images 6 to 10

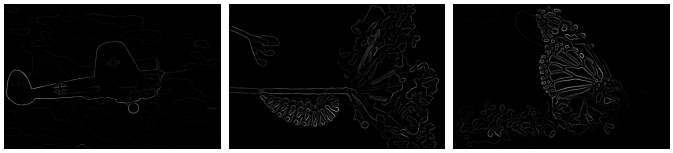
Optimizer: Adam Optimizer
Learning Rate: 0.0001
Batch Size: 1 for first epoch, then multiples of 100 for next 3 epochs and so on
Number of parameters in the model: 50452634

The per epoch loss and accuracy on train and test dataset is provided in Figure 13. The confusion matrices for the 'enhanced' network on train and test data after 15 epochs are at Tables III and IV. Inference time on train dataset (50000 images) was 4 min 37 sec and that on the test dataset was 55 sec.

Next, new modifications were attempted with the network. This included the ResNet architecture, the ResNeXt architecture and the DenseNet architecture.

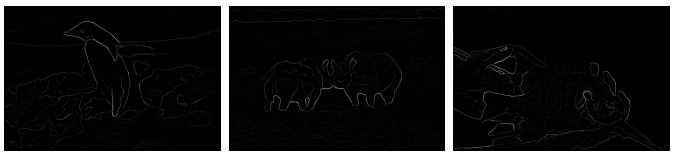
ResNet

For the ResNet architecture, the 2 convolution layers in the

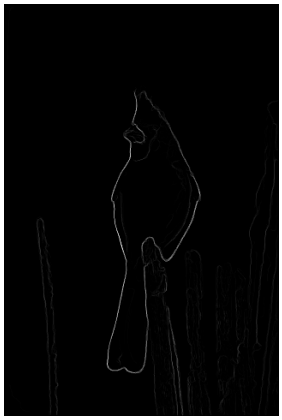


(a) PbLite outputs for images 1 to 3

(b) PbLite outputs for images 4 to 6



(c) PbLite outputs for images 7 to 9



(d) PbLite output for image 10

Fig. 10. PbLite outputs for images 1 to 10

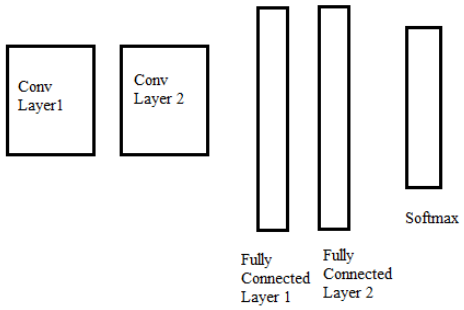


Fig. 11. 'First' Network schematic

'first' network were also accompanied by 1 skip connection as shown in Figure 14. This implementation is not a full-fledged replication of the ResNeXt architecture, but just a toy model.

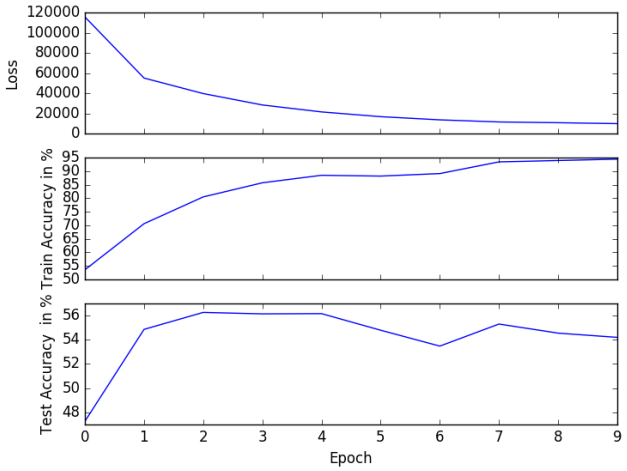


Fig. 12. Per epoch loss and accuracy on train and test dataset for 'first' network, after 10 epochs

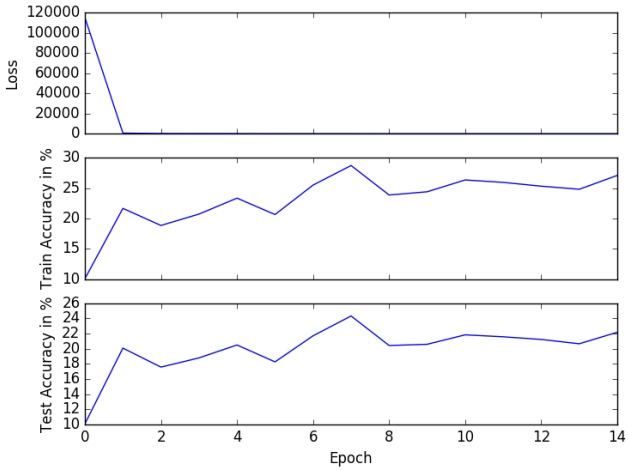


Fig. 13. Results for enhanced network. The loss value dropped below 1000 units after the first epoch itself

The network was trained for 15 epochs.

Network architecture:

Optimizer: Adam Optimizer
Learning Rate: 0.0001
Batch Size: 1 for first epoch, then (epochnumber/3 + 1) * 100
Number of parameters in the model: 50453786

The per epoch loss and accuracy on train and test dataset is provided in Figure 15. The confusion matrices for ResNet on train and test data after 15 epochs are at Tables V and VI

ResNeXt

For the ResNeXt architecture, the input image batch was split among 2 different low dimensional sub-parts, here the cardinality parameter of the network was fixed to 2. Both these sub-parts were transformed by passing them through 2 different convolution layers and the results of the layers from

Class	0	1	2	3	4	5	6	7	8	9
0	3121	376	154	5	4	0	111	0	155	1074
1	6	2465	258	479	1	4	237	0	334	1216
2	15	468	3320	104	6	5	87	166	0	829
3	13	651	167	2899	6	8	171	0	182	903
4	1	602	100	2	3355	3	62	0	122	753
5	11	586	102	8	6	3080	120	7	200	880
6	9	599	88	2	3	72	3427	11	99	690
7	30	829	232	6	1	87	2808	211	796	0
8	4	481	180	4	4	216	0	252	2378	1481
9	20	448	648	2	3	204	0	309	872	2494

TABLE III
CONFUSION MATRIX FOR 'ENHANCED' NETWORK ON TRAINING DATASET AFTER 15 EPOCHS

Class	0	1	2	3	4	5	6	7	8	9
0	634	92	22	0	0	1	25	0	24	202
1	2	494	47	87	0	0	57	0	70	243
2	0	90	685	20	1	3	15	0	34	152
3	2	133	26	598	0	2	28	0	41	170
4	1	109	13	0	666	1	14	0	23	173
5	1	110	18	2	1	643	25	0	38	162
6	1	149	17	12	0	0	678	0	23	120
7	4	157	54	0	0	1	24	556	32	172
8	2	89	50	46	0	1	46	0	450	316
9	1	97	128	188	0	0	34	0	60	492

TABLE IV
CONFUSION MATRIX FOR 'ENHANCED' NETWORK ON TEST DATASET AFTER 15 EPOCHS

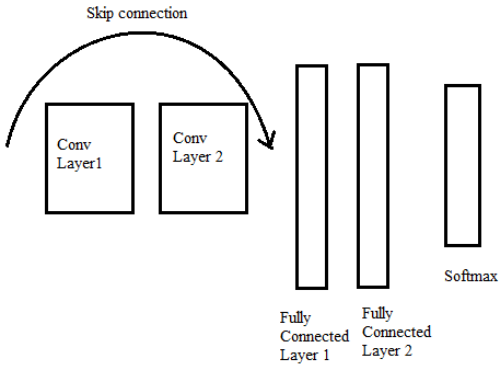


Fig. 14. ResNet schematic

Class	0	1	2	3	4	5	6	7	8	9
0	4644	2	1	2	0	4	0	0	1	346
1	1827	1833	1	15	0	37	0	6	1	1280
2	3510	76	262	104	0	291	5	14	5	733
3	2301	76	6	969	1	472	2	19	14	1140
4	3719	64	17	171	21	209	8	12	2	777
5	1756	39	7	112	0	2488	2	13	1	582
6	2982	188	8	142	0	328	251	6	6	1089
7	2036	91	7	89	0	357	1	711	2	1706
8	2935	33	0	17	1	14	0	4	208	1788
9	537	20	1	4	0	15	0	1	0	4422

TABLE V
CONFUSION MATRIX FOR RESNET ON TRAINING DATASET AFTER 15 EPOCHS

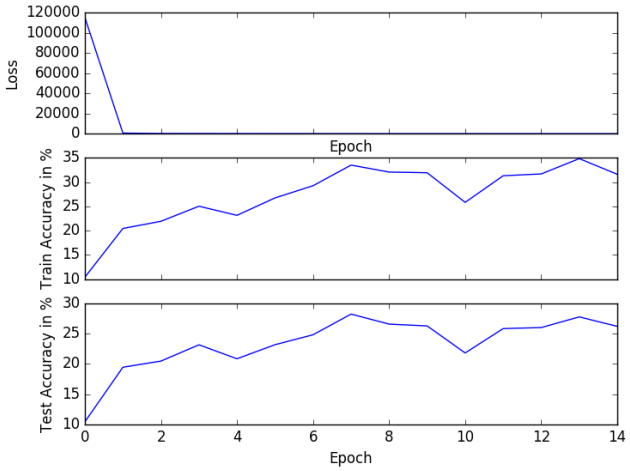


Fig. 15. Per epoch loss and accuracy on train and test dataset for ResNet, after 15 epochs. The per epoch loss dropped below 1000 units immediately after the first epoch

Class	0	1	2	3	4	5	6	7	8	9
0	872	5	1	3	0	4	0	1	0	114
1	410	275	0	2	0	11	1	4	0	297
2	664	12	21	32	0	89	2	6	1	173
3	474	27	5	103	0	157	1	3	2	128
4	710	9	2	51	1	50	3	2	0	172
5	413	13	1	52	0	380	2	4	0	135
6	592	30	5	37	0	72	40	2	3	219
7	431	13	0	19	0	90	0	110	1	336
8	591	12	4	4	0	3	0	0	35	351
9	177	21	1	0	0	15	1	1	1	183

TABLE VI
CONFUSION MATRIX FOR RESNET ON TEST DATASET AFTER 15 EPOCHS

both the sub-parts were merged. This split-transform-merge graph was also accompanied by a skip connection, refer Figure 16. This implementation is not a full-fledged replication of the ResNeXt architecture, but just a toy model.

Network architecture:

Optimizer: Adam Optimizer
Learning Rate: 0.0001
Batch Size: 1 for first epoch, then (epochnumber/3 + 1) * 100
Number of parameters in the model: 50571434

The per epoch loss and accuracy on train and test dataset is provided in Figure 17. The confusion matrices for ResNeXt on train and test data after 10 epochs are at Tables VII and VIII. Inference time on train dataset (50000 images) was 4 min 43 sec and that on the test dataset was 57 sec.

DenseNet

In the final modification, the enhanced network was added with 4 more convolution layers to give a total of 6 convolution layers. 3 of these layers have the same number of input and output channels while the other layers have the same number of input and output channels different from the first 3 layers. Each of the 3 layers was connected to all its subsequent networks of the same input size, thus yielding 7 connections in

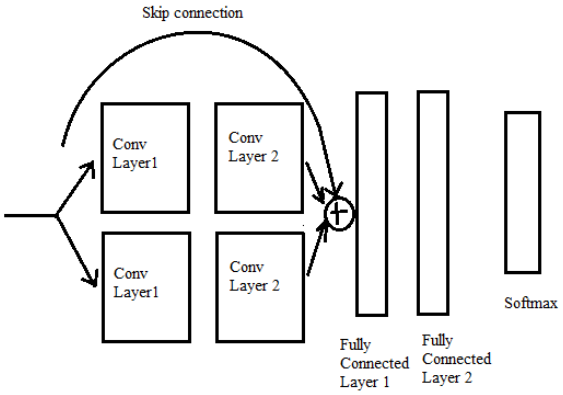


Fig. 16. ResNeXt schematic

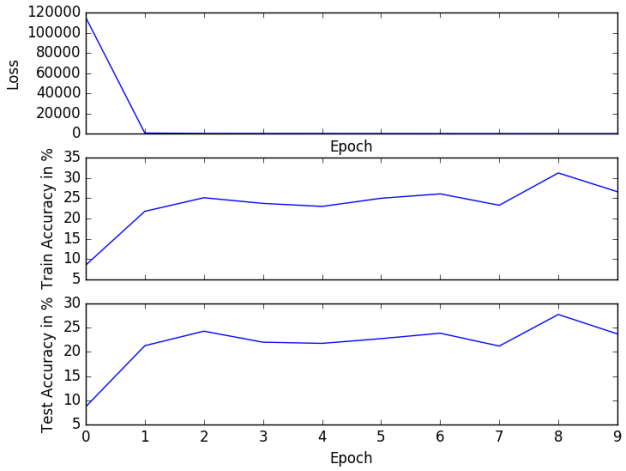


Fig. 17. Per epoch loss and accuracy on train and test dataset for ResNeXt, after 10 epochs. The per epoch loss dropped below 1000 units immediately after the first epoch

6 networks, which in the normal scenario would have been 5 connections only, as shown in Figure 18. This implementation is not a full-fledged replication of the DenseNet architecture, but just a toy model.

Network architecture:

Class	0	1	2	3	4	5	6	7	8	9
0	4761	2	1	1	0	4	0	131	8	92
1	3146	950	1	8	0	24	0	247	39	585
2	4218	27	164	83	0	174	0	165	19	150
3	3136	72	13	511	0	569	0	178	40	481
4	4273	26	8	73	0	209	2	191	13	205
5	2636	43	20	168	0	1661	0	165	11	296
6	3935	140	2	103	0	431	34	67	18	270
7	2515	52	7	77	0	196	0	1769	8	376
8	3585	12	3	5	0	3	0	340	747	305
9	1801	40	1	6	0	27	0	377	28	2720

TABLE VII
CONFUSION MATRIX FOR RESNEXT ON TRAINING DATASET AFTER 10 EPOCHS

Class	0	1	2	3	4	5	6	7	8	9
0	935	3	1	0	0	1	0	30	3	27
1	641	157	0	1	0	4	0	47	9	141
2	846	4	19	17	0	42	0	32	8	32
3	611	16	7	69	0	147	0	37	3	110
4	836	7	2	24	0	42	0	36	3	50
5	573	10	4	45	0	277	0	38	3	50
6	783	23	1	20	0	101	3	10	4	55
7	515	7	3	7	0	47	0	324	3	94
8	705	4	1	3	0	5	0	69	136	77
9	431	20	0	0	0	8	0	90	2	449

TABLE VIII
CONFUSION MATRIX FOR RESNEXT ON TEST DATASET AFTER 10 EPOCHS

Class	0	1	2	3	4	5	6	7	8	9
0	1947	2149	888	2	0	0	0	13	1	0
1	1618	3118	249	1	0	2	0	11	0	1
2	1013	3506	434	2	0	6	0	36	3	0
3	1268	3514	120	20	0	9	0	67	1	1
4	869	3933	151	0	0	4	0	43	0	0
5	1302	3491	134	3	0	20	0	47	3	0
6	781	4141	56	0	0	3	0	19	0	0
7	1994	2616	134	0	0	7	0	248	1	0
8	1660	2328	985	0	0	4	0	16	6	1
9	2110	2519	313	0	0	7	0	45	0	6

TABLE IX
CONFUSION MATRIX FOR DENSENET ON TRAINING DATASET AFTER 10 EPOCHS

Optimizer: Adam Optimizer
Learning Rate: 0.0001
Batch Size: 1 for first epoch, then multiples of 100 for next 3 epochs and so on
Number of parameters in the model: ??

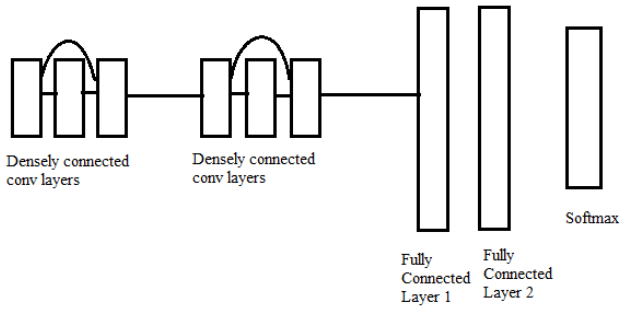


Fig. 18. DenseNet schematic

The per epoch loss and accuracy on train and test dataset is provided in Figure 19. The confusion matrices for DenseNet on train and test data after 15 epochs are at Tables IX and X. Inference time on train dataset (50000 images) was 1 min 9 sec and that on the test dataset was 14 sec.

Ideally, the ResNet, ResNeXt and DenseNet architectures are better in terms of faster convergence than the 'first'

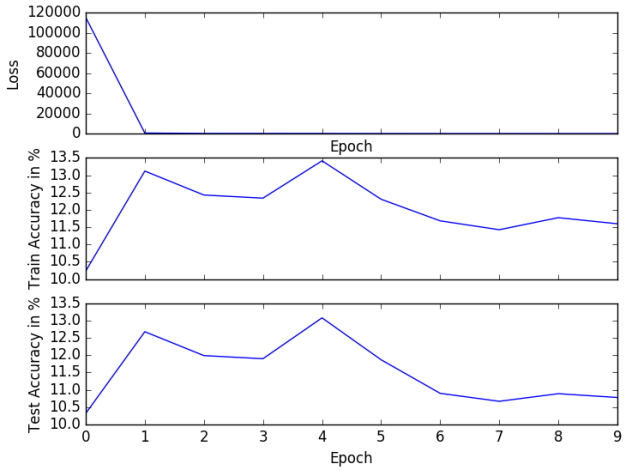


Fig. 19. Per epoch loss and accuracy on train and test dataset for 'DenseNet' after 15 epochs

Class	0	1	2	3	4	5	6	7	8	9
0	342	446	211	0	0	0	0	1	0	0
1	348	595	55	0	0	0	0	1	0	1
2	203	699	82	0	0	3	0	12	1	0
3	277	688	20	1	0	4	0	9	1	0
4	191	770	28	0	0	0	0	11	0	0
5	263	701	19	2	0	1	0	14	0	0
6	162	825	7	1	0	0	0	5	0	0
7	448	480	19	0	0	1	0	52	0	0
8	354	457	177	1	0	3	0	3	5	0
9	397	541	53	0	0	0	0	9	0	0

TABLE X
CONFUSION MATRIX FOR DENSENET ON TEST DATASET AFTER 10 EPOCHS

network, but since the versions of ResNet, ResNeXt and DenseNet implemented here were toy versions, that could be the reason why the results were not as expected.

REFERENCES

1) https://github.com/tensorflow/models/blob/master/official/resnet/resnetmodel.py
2) https://www.cs.cornell.edu/courses/cs6670/2011sp/lectures/lec02filter.pdf
3) http://cs.brown.edu/courses/cs143/2011/proj2/