Computer Vision HW0: Alohomora

Abhijeet Sanjay Rathi
MS Robotics Engineering
asrathi@wpi.edu
901015394

***Abstract*—Here, I have implemented a pb (Probability of Boundary) boundary detection algorithm which is far more better than the classical edge detection algorithms. This was for the Phase 1. For Phase 2, I have implemented Simple Neural Network, ResNet, ResNext and DenseNet for CIFAR 10-dataset.**

I. Phase 1: Shake My Boundary

In this section, I implemented the Probability of Boundary detection algorithm. It is better than the classical edge detection algorithms including Sobel and Canny as it considers texture discontinuities and color discontinuities along with intensity discontinuities. This is done in the following 4 steps:

1) Filter Banks Creation
2) Texton, Brightness and Color Map
3) Texton, Brightness and Color Gradient Map
4) Probability of Boundary

A. Filter Banks Creation

The first step of pb lite boundary detection algorithm is to filter the images with a set of filters. Filters are used to extract the low level features which represents the texture properties. So, I created a set of three different filter banks: Oriented DoG Filters, Leung-Malik Filters and Gabor Filters.

1) Oriented Difference of Gaussian (DoG) Filters
   This is a simple and effective filter bank. They are created by convolving simple Sobel filter and Gaussian filter and then rotating the output. For this, I have considered 2 scales, 16 orientations and kernel size as 41.
2) Leung-Malik Filters
   The Leung-Malik Filters, also referred to as LM Filters, constitute a set of multi-scale and multi-orientation filters. The LM filter comprises 48 filters, encompassing first and second-order derivatives of Gaussians at 6 orientations and 3 scales, totaling 36 filters, along with 8 Laplacian of Gaussian (LOG) filters and 4 Gaussians. Two versions of the LM filter are examined in this context. In LM Small (LMS), the filters occur at basic sigma scales $(1, \sqrt{2}, 2, 2\sqrt{2})$. The first and second derivative filters manifest at the first three scales with an elongation factor of 3. The Gaussians manifest at the four basic scales, while the 8 LOG filters manifest at σ and 3σ. For LM Large (LML), the filters occur at the basic sigma scales $(\sqrt{2}, 2, 2\sqrt{2}, 4)$.

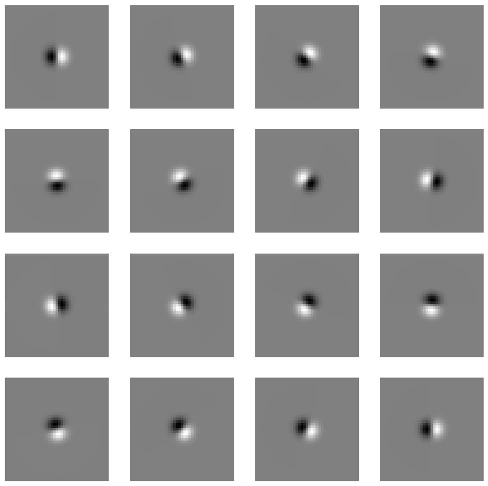


Fig. 1: *Oriented Difference of Gaussian (DoG) Filters*

3) Gabor Filters
   Gabor Filters are crafted with inspiration from the filters found in the human visual system. A Gabor filter is essentially a Gaussian kernel function modulated by a sinusoidal plane wave. The Gabor filter is characterized by scales of 3, 6, and 12, featuring 8 orientations and 4, 8, 16 frequencies, with a kernel size of 41.

B. Texton, Brightness and Color Map

1) Texton Map
   In essence, the approach involves the application of filter banks to generate N-dimensional responses for individual pixels in an image. Subsequently, these responses undergo k-means clustering to form K textons, effectively reducing the dimensionality from N to 1. This process, known as Vector Quantization, assigns each pixel a one-dimensional cluster ID, representing its texture properties. The typical choice for K in this context is 64 clusters. To elaborate further, the method employs filter banks, as explained in the preceding section, to discern texture properties within an image.

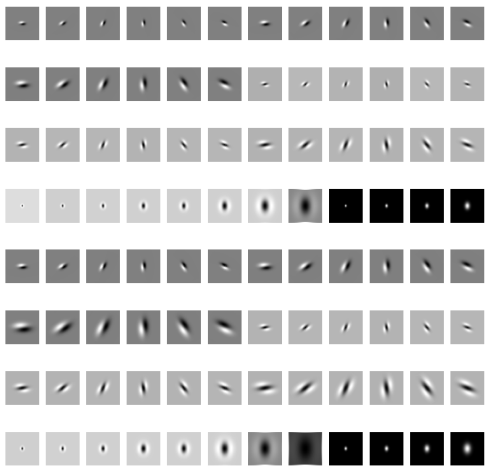


Fig. 2: *LM Filters (LMS + LML)*

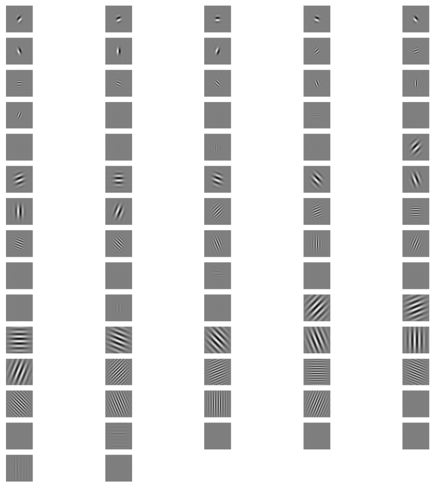
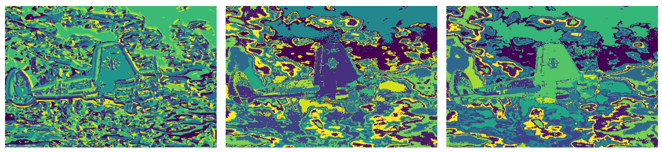


Fig. 3: *Gabor Filters*

Utilizing multiple filters in each bank and employing three filter banks, the method produces a vector of filter responses for each pixel, encoding texture properties. By applying the K-means algorithm with K set to 64, pixels with similar texture properties are grouped together based on their filter response vectors. The result of this K-means clustering is a Texton map, indicating discrete cluster IDs assigned to each pixel.

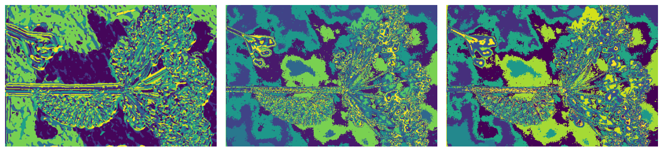
2) Brightness Map
The concept of the brightness map is quite straightforward, involving the capture of brightness changes in an image. In this process, the brightness values are clustered using k-means clustering on the grayscale equivalent of the color image. The image is initially converted to grayscale, and then the K-means algorithm with K=16 is applied to obtain brightness maps. The resulting clustered output is termed the brightness map.

3) Color Map
The idea behind the color map is to record the color variations or chrominance content within an image. In this process, we once again employ k-means clustering to cluster the color values (since there are 3 values per pixel with RGB color channels) into a 16 number of clusters. The resulting output of this clustering is referred to as the color map.



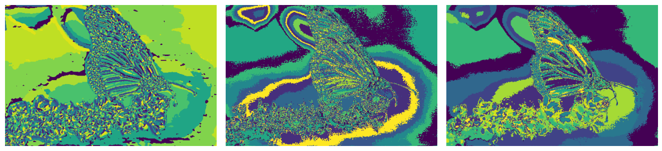
(a) T (b) B (c) C

Fig. 4: *T, B and C for Image 1*



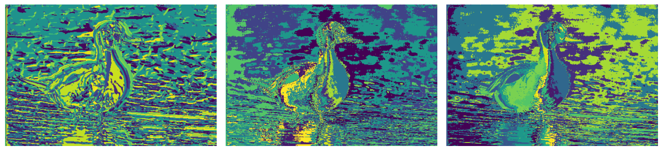
(a) T (b) B (c) C

Fig. 5: *T, B and C for Image 2*



(a) T (b) B (c) C

Fig. 6: *T, B and C for Image 3*



(a) T (b) B (c) C

Fig. 7: *T, B and C for Image 4*

C. *Texton, Brightness and Color Gradients (T_g, B_g, and C_g)*

T_g, B_g, and C_g represent the extent of changes in texture, brightness, and color distributions at a pixel. To derive T_g, B_g, and C_g, it is necessary to compute the differences in values

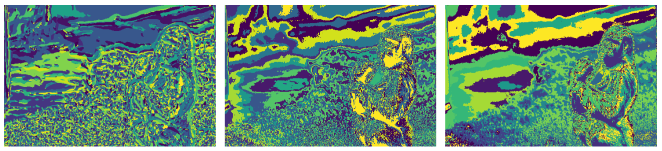

(a) T (b) B (c) C

Fig. 8: *T, B and C for Image 5*

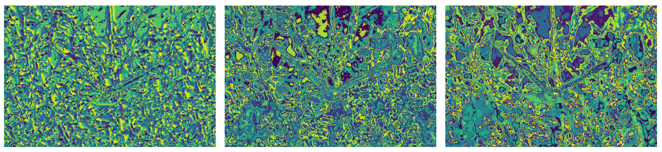

(a) T (b) B (c) C

Fig. 9: *T, B and C for Image 6*

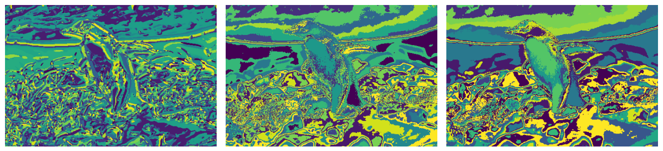

(a) T (b) B (c) C

Fig. 10: *T, B and C for Image 7*

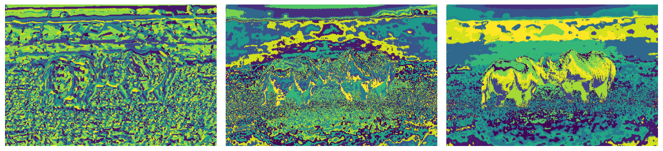

(a) T (b) B (c) C

Fig. 11: *T, B and C for Image 8*

across various shapes and sizes. These values are calculated by comparing distributions in pairs of left/right half-discs, which correspond to opposing directions of filters at the same scale. This process can be efficiently achieved through the utilization of half-disc masks.

1) Half Disk Masks
Half-disk masks are binary images representing half circles. They play a crucial role in calculating changes in texture or brightness distributions across various scales and angles. These masks are employed for comparing texture, color, and brightness properties over the disk area, offering a softer filter compared to the Sobel operator, which focuses on immediate pixels. The creation of these masks involves using different radii, such as 2, 4, 8, 16 and 32, and 18 orientations in this particular case.
2) Chi - Square Distance
In the computation of the X^2 distance, the images undergo convolution with left/right half-disc pairs, resulting in two histograms denoted as g and h. The X^2 distance is then calculated using the formula below:

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

This process is applied separately to each texton, brightness, and color map. For N half-disk pairs, N matrices are obtained, where each pixel value corresponds to the X^2 distance. To derive the gradients of T_g, B_g, and C_g, the average of these N matrices is taken.
3) Sobel and Canny Baselines
The Sobel and Canny images are loaded using the *cv2.imread()* function.

D. Probability of Boundary (Pb)

In the final step, I integrated information from the features with a baseline method, relying on Sobel and Canny edge detection, using the following equation:

$$PbEdges = \frac{T_g + B_g + C_g}{3} \circ (w_1 * cannyPb + w_2 * sobelPb)$$

The $\circ$ symbol denotes the Hadamard product operator, representing elementwise multiplication between two matrices. The weights w1 and w2 are both set to 0.5, resulting in a simple mean for the weighted average.

E. Analysis

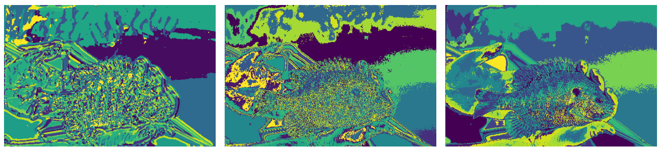
It is observed that pb-lite edges exhibit a reduced amount of noise compared to Canny and Sobel. This improvement is attributed to its effectiveness in suppressing false positives, which contribute to the noise present in Sobel and Canny outputs. The final result is enhanced, and further improvements can be achieved by exploring and selecting filters that outperform the existing ones.

II. Phase 2: Deep Dive on Deep Learning

In this section, I implemented multiple neural network architectures and compared them based on the number of parameters, accuracy, and loss. The four architectures employed are: Simple Neural Network, ResNet, ResNext, and DenseNet. A simple neural network is characterized by fewer filters and shallower depth compared to the other architectures.

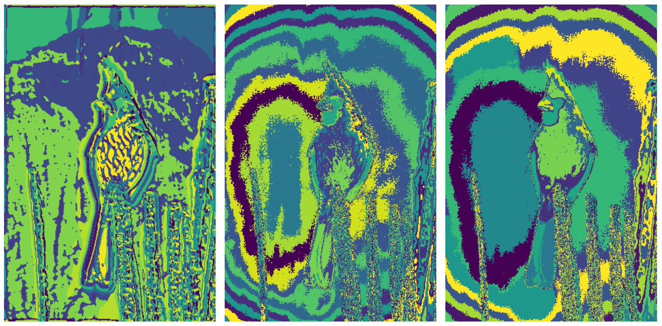
For each of the four architectures, two types of models are trained: one with the transformation of the training dataset and one without. I implemented a series of image transformations using the *torchvision.transforms* module in PyTorch, tailored for image preprocessing in a neural network. These transformations include a random horizontal flip with a 50% probability, introducing variability during training. Images are then converted to the torch float32 format and scaled, ensuring consistent data representation. A normalization step follows, adjusting pixel values based on predefined mean and standard deviation values, which aids in stabilizing and accelerating the training process.

Additionally, a random resized crop operation is applied, where images are randomly cropped and resized to a specified



(a) T (b) B (c) C

Fig. 12: *T, B and C for Image 9*



(a) T (b) B (c) C

Fig. 13: *T, B and C for Image 10*

size of 32x32 pixels with antialiasing enabled. This set of transformations, commonly employed in image classification tasks, serves the dual purpose of augmenting the training dataset for improved model generalization and preparing the data for input into a neural network designed for image classification on the CIFAR-10 dataset.

While generating batches for each model, I implemented a code to ensure that 80% of the images in the mini-batch size go for testing, and the remaining 20% are allocated for validation. This approach helps in assessing the model's performance on unseen data during training. The training configurations for each model are as follows:

1) Number of epochs: 20
2) Mini-batch size: 30
3) Optimizer: AdamW
4) Learning rate: 0.0001
5) Weight decay: 0.01

A. Simple Neural Network (SNN)

The SNN model consists of three convolutional layers with batch normalization and ReLU activation, followed by max-pooling to extract hierarchical features from input images. The output is then flattened and passed through two fully connected layers with ReLU activation. The final layer produces the classification output. The model is designed to take images with three color channels as input and output predictions for the specified output size. This architecture is a common pattern for image classification tasks.

B. ResNet

Here, I have implemented a ResNet-based neural network architecture. It consists of two main classes: 'BottleneckResnet' and 'ResNet'. The 'BottleneckResnet' class defines a

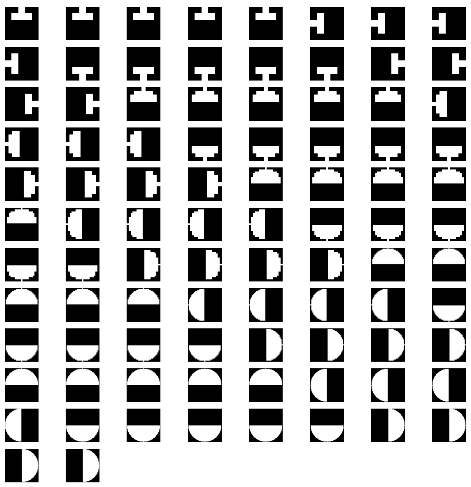
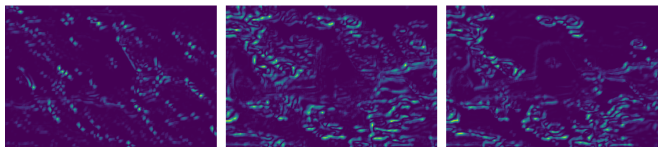


Fig. 14: *Half Disk Masks*



(a) T_g (b) B_g (c) C_g

Fig. 15: $T_g, B_g and C_g$ *for Image 1*

bottleneck residual block with three convolutional layers and shortcut connections, while the 'ResNet' class constructs the overall ResNet architecture by stacking multiple blocks. Additionally, a class named 'CIFAR10ModelResnet' is created, inheriting from 'ImageClassificationBase', and serves as a wrapper for the ResNet model with specific configurations. The model is initialized with a bottleneck block type, an array specifying the number of blocks in each layer, and the number of output classes (set to 10 for CIFAR-10). This ResNet-based model is designed such that it is able to capture complex features in images for accurate classification, and the provided instantiation uses bottleneck blocks with a configuration that is known to be effective in practice.

The bottleneck layer is a key component, often associated with architectures like ResNet. It consists of three consecutive layers: a 1x1 convolution for dimensionality reduction, a 3x3 convolution for feature extraction, and another 1x1 convolution for dimensionality expansion. This design efficiently balances computational complexity, capturing intricate features in a reduced channel space.

C. ResNext

I have implemented the ResNeXt architecture such that it comprises of two main classes: 'BlockResnext' and 'ResNeXt'. The 'BlockResnext' class defines a block with

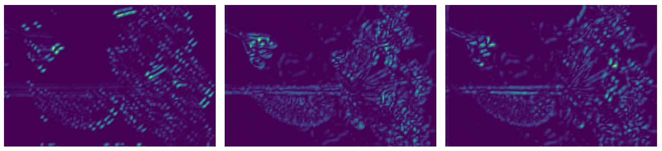

(a) T_g (b) B_g (c) C_g

Fig. 16: $T_g, B_g and C_g$ *for Image 2*

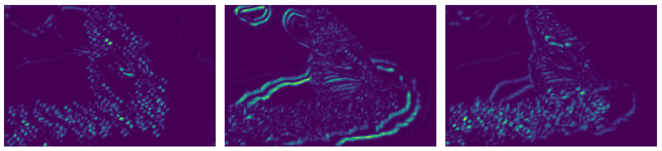

(a) T_g (b) B_g (c) C_g

Fig. 17: $T_g, B_g and C_g$ *for Image 3*

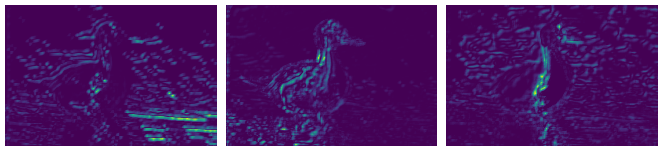

(a) T_g (b) B_g (c) C_g

Fig. 18: $T_g, B_g and C_g$ *for Image 4*

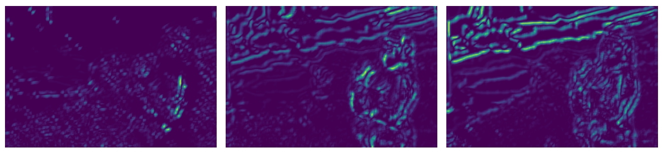

(a) T_g (b) B_g (c) C_g

Fig. 19: $T_g, B_g and C_g$ *for Image 5*

grouped convolutions and a shortcut connection, while the 'ResNeXt' class constructs the overall architecture by stacking multiple blocks. The number of blocks in each layer, cardinality, and bottleneck width are configurable parameters. Additionally, a class named 'CIFAR10ModelResnext' serves as a wrapper for the ResNeXt model with specific configurations. The model is initialized with the specified number of blocks, cardinality, and bottleneck width.

Here, Cardinality refers to the number of groups in grouped convolutions within a block. A higher cardinality allows the model to capture more diverse features by considering different subsets of channels in parallel.

D. DenseNet

Here, the DenseNet architecture is implemented such that it consists of three main classes: 'BottleneckDensnet', 'Transition', and 'DenseNet'. The 'BottleneckDensnet' class defines a bottleneck block with batch normalization and 1x1 and 3x3 convolutions. The 'Transition' class represents transition blocks that include batch normalization, 1x1 convolution, and average pooling to reduce spatial dimensions. The 'DenseNet' class constructs the overall architecture by stacking multiple instances of the bottleneck and transition blocks. It comprises four dense blocks, each followed by a transition block. The model uses global average pooling and a fully connected layer for classification. Additionally, a class named 'CIFAR10ModelDensenet' serves as a wrapper for the 'DenseNet' model with specific configurations. The model is initialized with the specified bottleneck block type, the number of blocks per dense block, and the growth rate.

E. Analysis

The summary table below provides an overview of different neural network architectures, including Simple Neural Network, ResNet, ResNext, and DenseNet, each with various parameters, transformation options, and associated accuracy metrics for testing and training. Simple Neural Network exhibits relatively low accuracy, influenced by the presence of transformations. ResNet and ResNext show better accuracy, with ResNet maintaining consistency regardless of transformations. DenseNet, on the other hand, demonstrates consistently low accuracy. This happens due to very less epochs i.e. 20 for a DenseNet. To improve overall performance, potential strategies include hyperparameter tuning, data augmentation, ensemble learning, transfer learning, architecture modifications, regularization techniques, and experimenting with learning rate schedules. Systematic exploration of these approaches can help enhance the models' accuracy and generalization capabilities.

III. Conclusion

All the neural network models were trained under consistent configurations with 20 epochs, a mini-batch size of 30, and utilized the AdamW optimizer. The learning rate was set at 0.0001, providing a balance between convergence speed and precision during training. Additionally, a weight decay of 0.01 was applied, contributing to the regularization of the models by penalizing large weights. These training settings offer a standardized foundation for model training across the various architectures, facilitating a fair comparison of their performances. It's worth noting that further refinement of these hyperparameters, in conjunction with the recommended strategies for improvement, could potentially yield even better results in terms of accuracy and generalization.

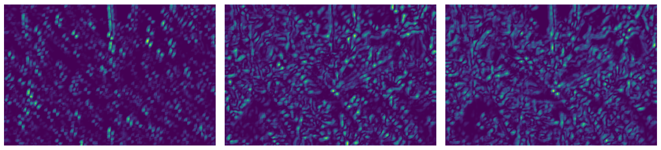

(a) T_g (b) B_g (c) C_g

Fig. 20: $T_g, B_g and C_g$ *for Image 6*

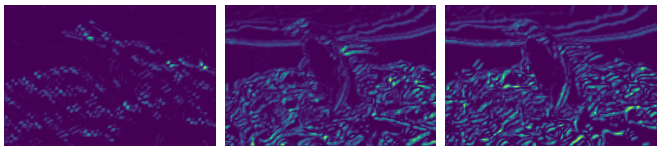

(a) T_g (b) B_g (c) C_g

Fig. 21: $T_g, B_g and C_g$ *for Image 7*

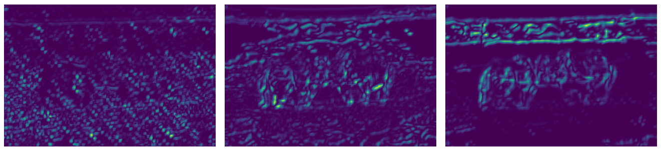

(a) T_g (b) B_g (c) C_g

Fig. 22: $T_g, B_g and C_g$ *for Image 8*

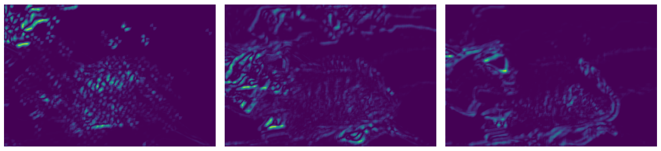

(a) T_g (b) B_g (c) C_g

Fig. 23: $T_g, B_g and C_g$ *for Image 9*

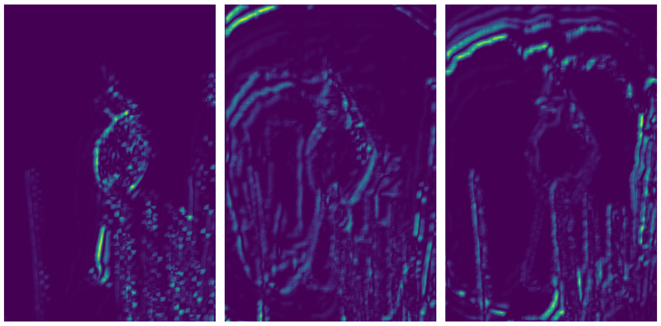

(a) T_g (b) B_g (c) C_g

Fig. 24: $T_g, B_g and C_g$ *for Image 10*

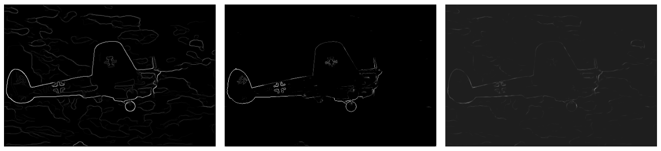

(a) Canny (b) Sobel (c) Pb

Fig. 25: *Canny, Sobol and Pb for Image 1*

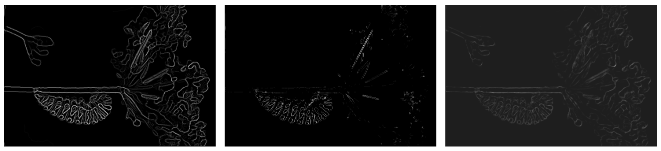

(a) Canny (b) Sobel (c) Pb

Fig. 26: *Canny, Sobol and Pb for Image 2*

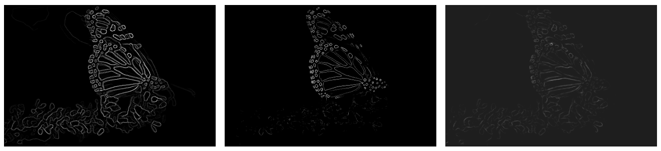

(a) Canny (b) Sobel (c) Pb

Fig. 27: *Canny, Sobol and Pb for Image 3*

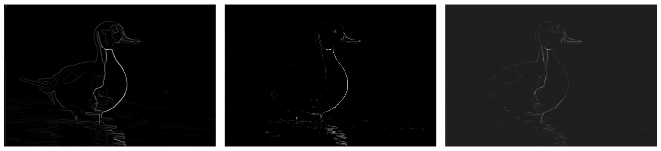

(a) Canny (b) Sobel (c) Pb

Fig. 28: *Canny, Sobol and Pb for Image 4*

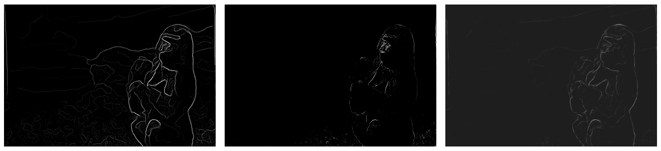

(a) Canny (b) Sobel (c) Pb

Fig. 29: *Canny, Sobol and Pb for Image 5*

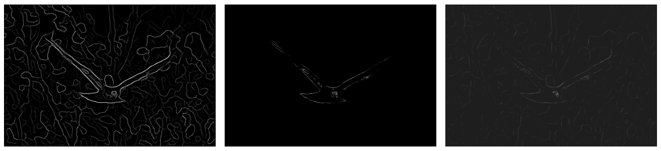

(a) Canny (b) Sobel (c) Pb

Fig. 30: *Canny, Sobol and Pb for Image 6*

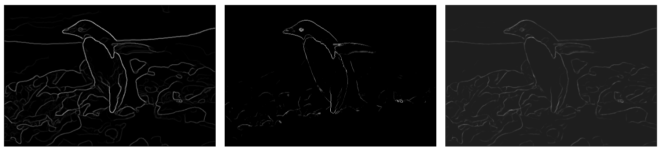
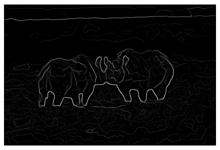
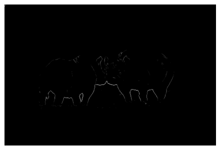

(a) Canny (b) Sobel (c) Pb

Fig. 31: *Canny, Sobol and Pb for Image 7*


(a) Canny


(b) Sobel

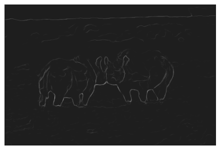
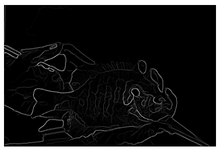
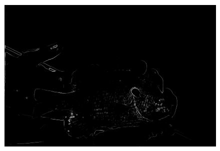

(c) Pb

Fig. 32: *Canny, Sobol and Pb for Image 8*


(a) Canny


(b) Sobel

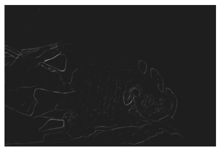
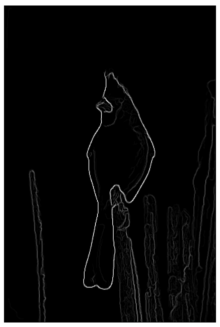
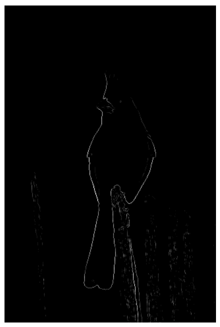

(c) Pb

Fig. 33: *Canny, Sobol and Pb for Image 9*


(a) Canny


(b) Sobel

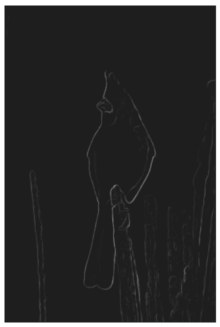

(c) Pb

Fig. 34: *Canny, Sobol and Pb for Image 10*

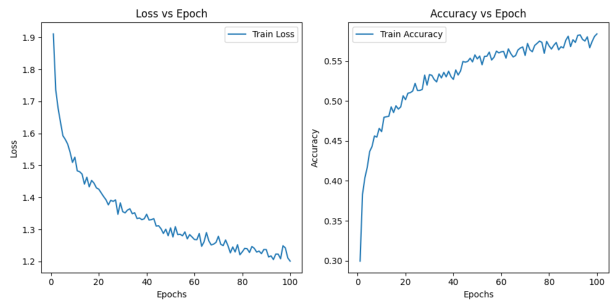


Fig. 35: *Loss and Accuracy Plot vs Epoch: SNN with Transformations*

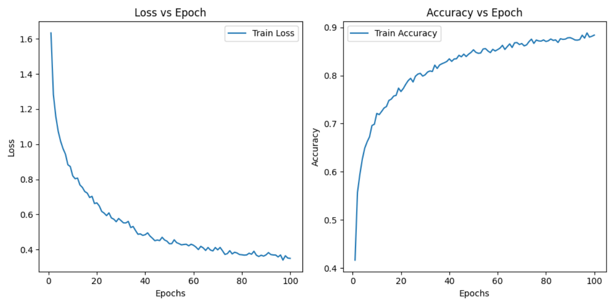
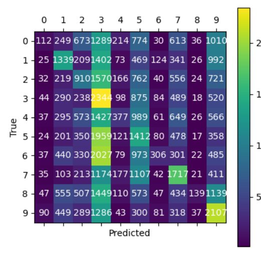
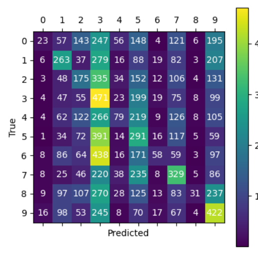


Fig. 36: *Loss and Accuracy Plot vs Epoch: SNN without Transformations*

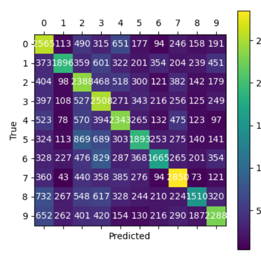


(a) Train

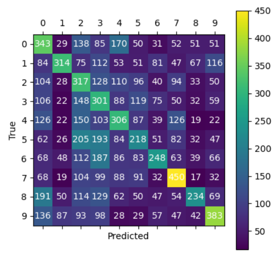


(b) Test

Fig. 37: *Train and Test Confusion Matrix: SNN with Transformations*



(a) Train



(b) Test

Fig. 38: *Train and Test Confusion Matrix: SNN without Transformations*

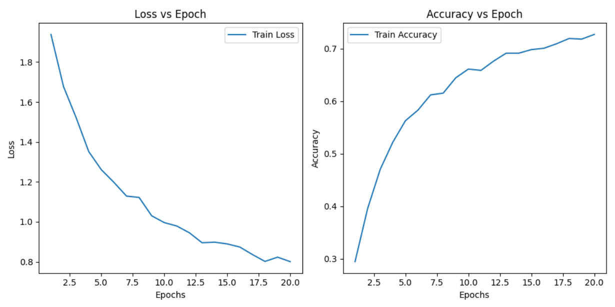


Fig. 39: *Loss and Accuracy Plot vs Epoch: ResNet with Transformations*

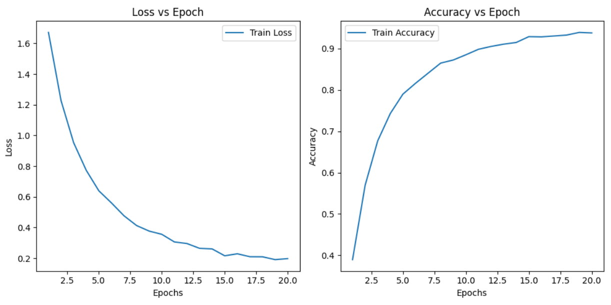
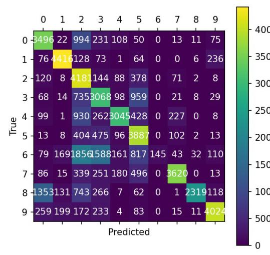
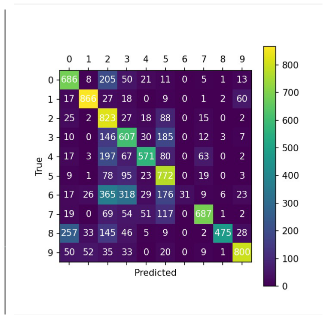


Fig. 40: *Loss and Accuracy Plot vs Epoch: ResNet without Transformations*

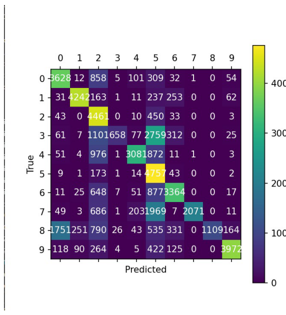


(a) Train

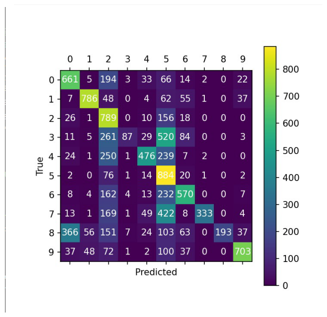


(b) Test

Fig. 41: *Train and Test Confusion Matrix: ResNet with Transformations*



(a) Train



(b) Test

Fig. 42: *Train and Test Confusion Matrix: ResNet without Transformations*

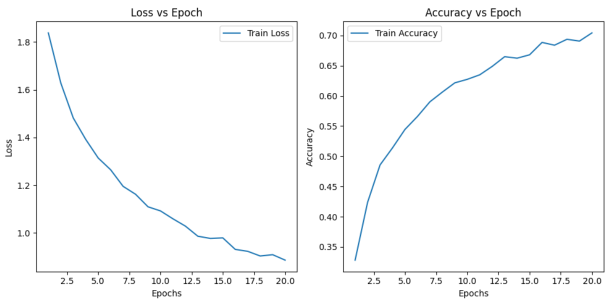


Fig. 43: *Loss and Accuracy Plot vs Epoch: ResNext with Transformations*

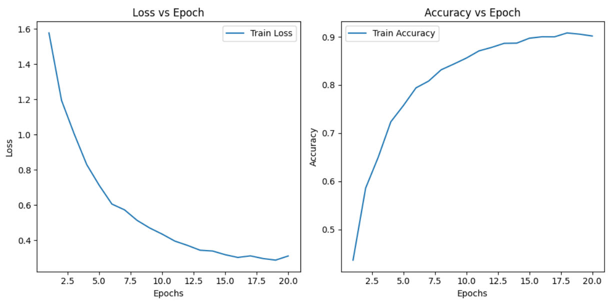
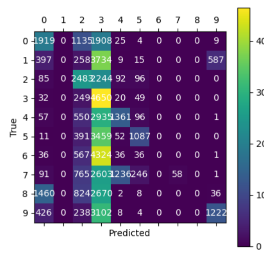
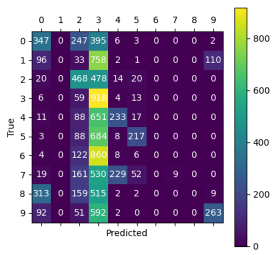


Fig. 44: *Loss and Accuracy Plot vs Epoch: ResNext without Transformations*

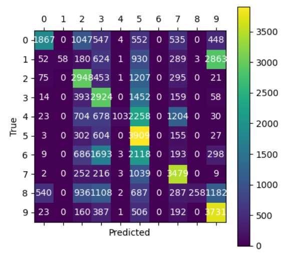


(a) Train

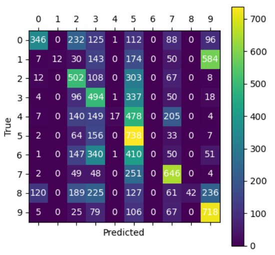


(b) Test

Fig. 45: *Train and Test Confusion Matrix: ResNext with Transformations*



(a) Train



(b) Test

Fig. 46: *Train and Test Confusion Matrix: ResNext without Transformations*

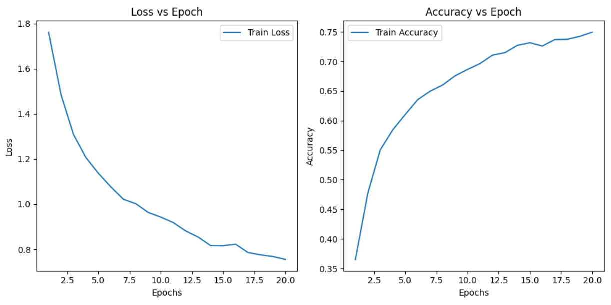


Fig. 47: *Loss and Accuracy Plot vs Epoch: DenseNet with Transformations*

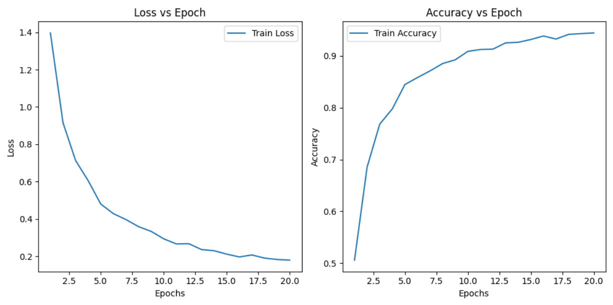
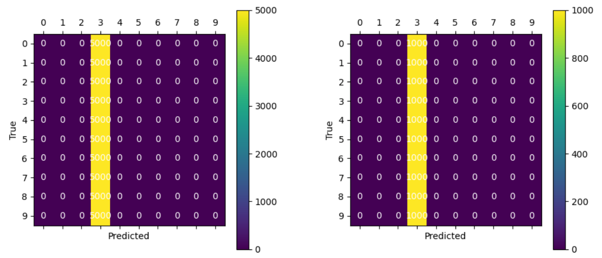
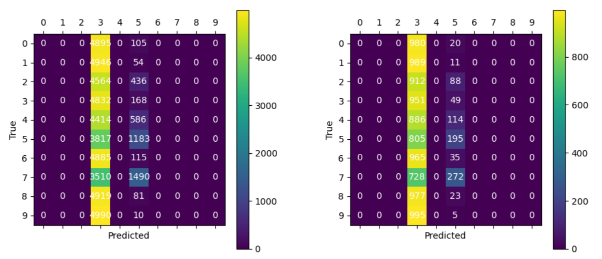


Fig. 48: *Loss and Accuracy Plot vs Epoch: DenseNet without Transformations*



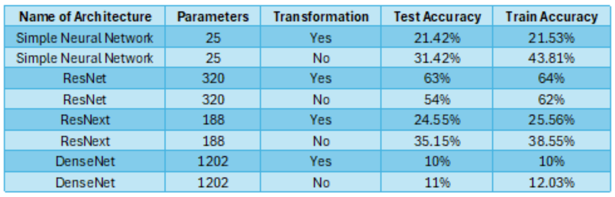
(a) Train (b) Test

Fig. 49: *Train and Test Confusion Matrix: DenseNet with Transformations*



(a) Train (b) Test

Fig. 50: *Train and Test Confusion Matrix: DenseNet without Transformations*



Name of Architecture	Parameters	Transformation	Test Accuracy	Train Accuracy
Simple Neural Network	25	Yes	21.42%	21.53%
Simple Neural Network	25	No	31.42%	43.81%
ResNet	320	Yes	63%	64%
ResNet	320	No	54%	62%
ResNext	188	Yes	24.55%	25.56%
ResNext	188	No	35.15%	38.55%
DenseNet	1202	Yes	10%	10%
DenseNet	1202	No	11%	12.03%

Fig. 51: *Summary of all Neural Networks*