Alohomora!
RBE549 Homework 0

Sumukh Porwal
MS Robotics Engineering
Worcester Polytechnic Institute
Email: sporwal@wpi.edu

I. PHASE 1: SHAKE MY BOUNDARY

The first part of the assignment explores classical methods in computer vision for boundary detection. I present a detailed analysis of my implementation of the pb (probability of boundary) boundary detection algorithm. Here, I have implemented its lite version - "pb-lite". The algorithm works by using texture, brightness and color information to improve the boundary detection result of the Sobel and Canny baseline. The method consists of 4 main steps: (1) Filter Bank Generation (2) Texton, Brightness and Color Map computation (3) Texture, Brightness and Color Gradients computation, and (4) Boundary Detection.

A. Filter Bank Generation

The first step in the boundary detection algorithm is to create filter banks so that they can be used to capture texture information from the input images. In this implementation, we use three types of filters: Oriented Derivative of Gaussian (DoG) filters, Leung-Malik filters, and Gabor filters.

1) Oriented Derivative of Gaussian (DoG) Filters: As mentioned in the problem statement, these filters are created by convolving a simple Sobel filter and a Gaussian kernel and then rotating the result. The filter bank generated with 4 scales and 16 orientations is shown in Figure 1.

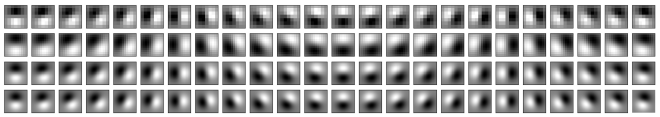

Fig. 1. Oriented Derivative of Gaussian (DoG) filters

2) Leung-Malik (LM) Filters: These are a set of multi scale, multi orientation filter bank with 48 filters. In this implementation, we consider two versions of Leung-Malik filter banks. The Leung-Malik Small filter bank is generated using the scales $\sigma = \{1, \sqrt{2}, 2, 2\sqrt{2}\}$, whereas the Leung-Malik Large filter bank is generated using the scales $\sigma = \{\sqrt{2}, 2, 2\sqrt{2}.4\}$ The generated Leung-Malik Large filter bank is shown in Figure 2, and the generated Leung-Malik Small filter bank is shown in Figure 3.

3) Gabor Filters: These filters which are approximated versions of how human visual system works. As mentioned in the problem statement, a Gabor filter is a Gaussian kernel function modulated by a sinusoidal plane wave. The filter bank generated with 6 scales and 16 orientations is shown in Figure 4.

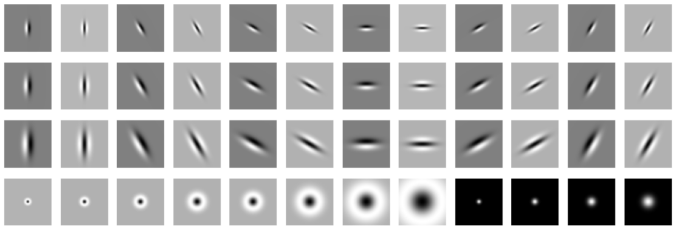

Fig. 2. Leung-Malik Large (LML) filters

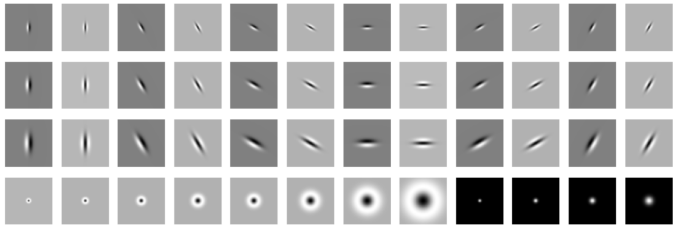

Fig. 3. Leung-Malik Small (LMS) filters

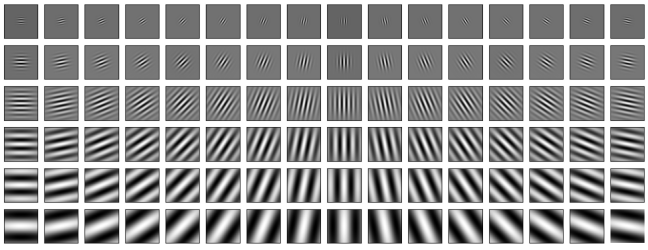

Fig. 4. Gabor Filters

B. Texton, Brightness, Color Map Computation

The next step is to filter the input image with each and every filter in our generated filter bank. Assuming there are N filters in total in our filter bank, we get an N-dimensional vector as a response for each pixel after filtering. Then, we use KMeans clustering on these vectors for all pixels, to get discrete Texton IDs (cluster size, K = 64) for each pixel. Replacing each pixel in the image with its corresponding Texton ID gives

us the Texton map. Similarly, we follow the same process to generate the Brightness and Color map (cluster size, K = 16). For Brightness, we use the grayscale pixel value and for Color, we use the RGB value of each pixel. The generated Texton Maps $\mathcal{T}$, Brightness Maps $\mathcal{B}$ and Color Maps $\mathcal{C}$ for all the 10 provided images is shown in Figure 6.

C. Texton, Brightness, Color Gradients Computation

To generate the Texton, Brightness and Color Gradients from the corresponding maps, we first need to generate halfdisc masks. The half-disc masks are simply (pairs of) binary images of half-discs. The half-disc masks generated with 3 scales and 8 orientations is shown in Figure 5.

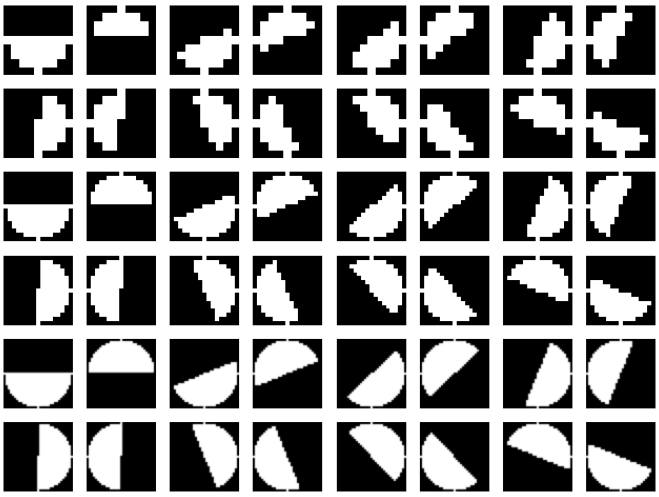

Fig. 5. Half-Disc Masks

Using these half-disc masks along with the Chi-square distance χ^2, the gradients of the Texton, Brightness, and Color maps are generated for each input image. The generated Texton Gradients $\mathcal{T}_g$, Brightness Gradients $\mathcal{B}_g$ and Color Gradients $\mathcal{C}_g$ for all the 10 provided images is shown in Figure 7.

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

D. Boundary Detection

The final step of the boundary detection pipeline is to combine the Texton, Brightness and Color gradients with the Sobel and Canny baselines to generate the final output. The generated pb-lite boundaries for all the 10 provided images is shown in Figure 8.

$$PbLite = (\frac{\mathcal{T}_g + \mathcal{B}_g + \mathcal{C}_g}{3}) * (0.5 * Sobel + 0.5 * Canny)$$

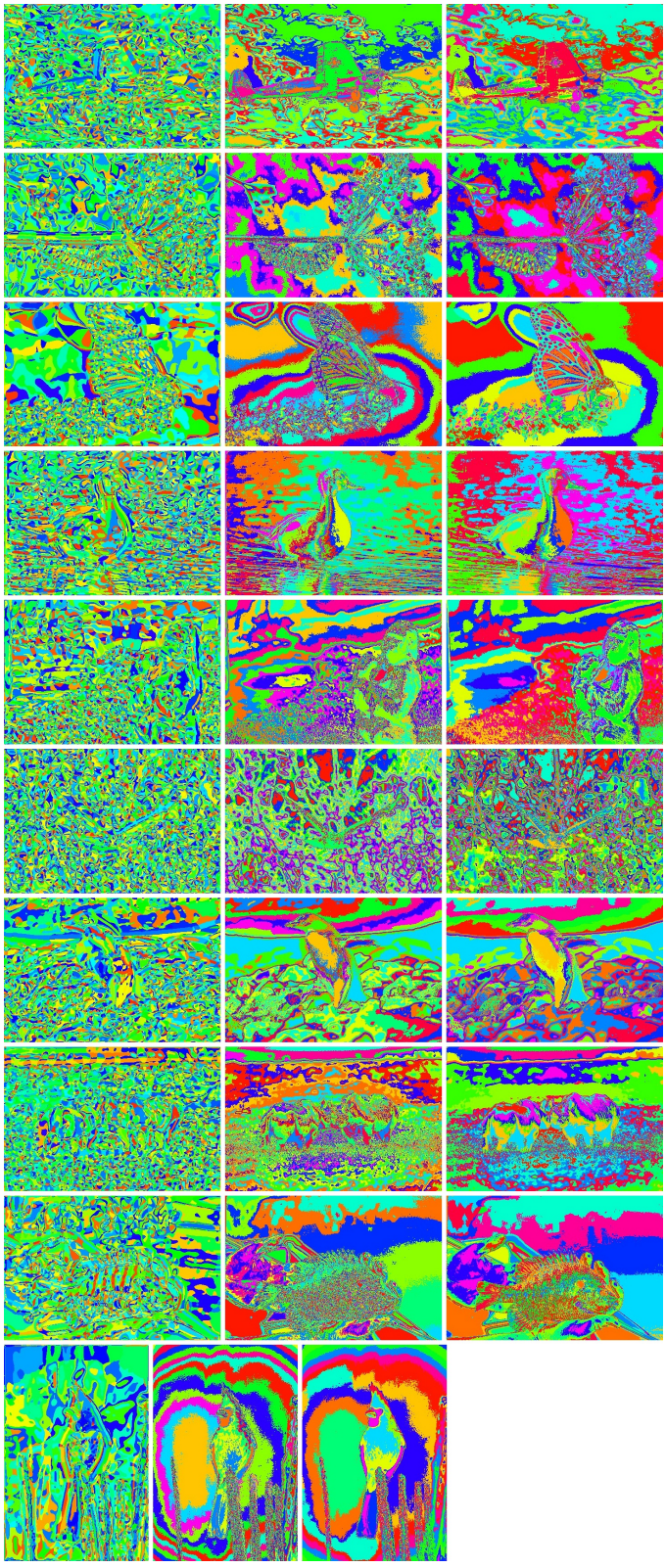

Fig. 6. Texton Map $\mathcal{T}$, Brightness Map $\mathcal{B}$, Color Map $\mathcal{C}$ for all images 1 through 10

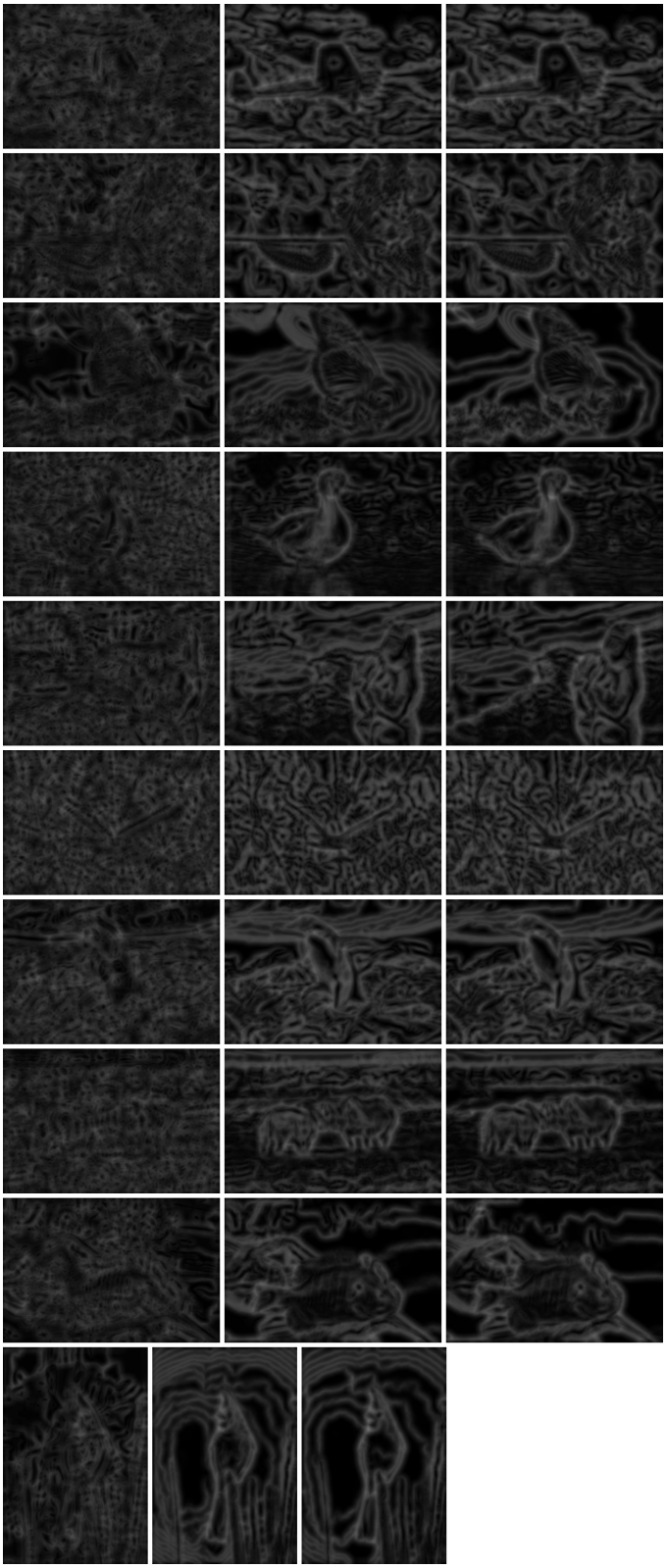

Fig. 7. Texton Gradients $\mathcal{T}_g$, Brightness Gradients $\mathcal{B}_g$, Color Gradients $\mathcal{C}_g$ for all images 1 through 10

Fig. 8. Comparison of the Sobel baseline, Canny baseline and the Pb-lite outputs for all images 1 through 10

II. PHASE 2: DEEP DIVE ON DEEP LEARNING

The second part of the assignment focuses on computer vision through deep learning. Here, I have implemented multiple neural networks to perform image classification. All the neural networks are trained on the CIFAR-10 dataset, which consists of 50,000 training images and 10,000 testing images.
To be able to compare the performance of all networks with each other, most of the hyperparameters are kept the same wherever possible. All the networks are trained for 50 epochs with a batch size of 512. All the networks are implemented and trained on the WPI Turing Cluster.

A. Initial Neural Network Architecture

The initial implementation of my neural network is a simplified version inspired by the AlexNet architecture. It consists of a series of convolutional and fully connected layers, specifically designed to demonstrate a straightforward approach to image classification. The network contains approximately one million parameters and does not incorporate advanced features such as dropout or batch normalization layers. This design was chosen to establish a baseline performance and to explore the fundamental capabilities of a neural network using only basic components. The architecture is shown in Figure 29.

This initial architecture uses convolutional layers to extract spatial features from the input images, followed by fully connected layers that map these features to the desired output classes. The absence of regularization mechanisms, such as batch normalization or dropout, makes the network susceptible to overfitting, as it fails to generalize well to unseen data.

The model was trained using the Adam optimizer with a learning rate of 0.001. The Adam optimizer, known for its adaptive learning rate and momentum properties, helped stabilize the training process while achieving faster convergence compared to traditional stochastic gradient descent. The total number of parameters in this model are **71092**.

The simplicity of the architecture results in a relatively quick training time, with the model completing training in approximately 30 minutes on the WPI Turing Cluster. However, the performance is limited, achieving a test accuracy of about 64% and train accuracy of about 86%, as illustrated in Figure 14. The corresponding loss curve is shown in Figure 9, and the confusion matrices in Figure 19 and Figure 20 provide further insights into the classification performance.

Despite its limitations, this initial network serves as a valuable reference point for understanding the impact of architectural choices and regularization techniques on neural network performance. The primary drawback of this model is its tendency to overfit the data, which is evident from the training and testing results. This overfitting is attributed to the lack of proper normalization and data augmentation techniques, which will be addressed in subsequent iterations of the model.

B. Improving Accuracy of the Neural Network

To address the limitations observed in the initial network, several enhancements were introduced to improve its accuracy and generalization capabilities. These improvements include the incorporation of batch normalization, data augmentation techniques, and normalization of the input data.

1) Adding Batch Normalization: Batch normalization was integrated into the network to address the issue of internal covariate shift, which often hampers the training process. This technique normalizes the input of each layer, ensuring that the mean and variance of the activations remain consistent during training. By stabilizing the learning process, batch normalization enables the model to converge faster and reduces its sensitivity to the choice of hyperparameters. Furthermore, it acts as a regularizer, mitigating the risk of overfitting and enhancing the network's ability to generalize to unseen data.

2) Data Augmentation: To improve the robustness of the model, data augmentation techniques were employed. These techniques artificially expand the training dataset by applying random transformations to the input images. Specifically, random horizontal flips, vertical flips, and cropping were used to create diverse variations of the original images. By exposing the network to a wider range of input patterns, data augmentation helps the model learn invariant features, reducing overfitting and improving performance on the test set.

3) Normalization of Input Data: Normalization was applied to the input images to standardize their pixel values. The pixel intensity range of [0, 255] was scaled to a normalized range of [-1, 1]. This preprocessing step ensures that the input data is centered around zero, which accelerates the convergence of gradient-based optimization algorithms and improves numerical stability. Normalization also ensures that all input features contribute equally during training, leading to more effective learning.

4) Learning Rate Decay: To further enhance the training process, a learning rate decay mechanism was implemented using a step scheduler. The learning rate decay helps prevent overfitting and ensures that the optimization process converges to a more optimal solution by gradually reducing the learning rate during training. Specifically, the `torch.optim.lr_scheduler.StepLR` scheduler was employed, with a step size of 10 epochs and a decay factor (γ) of 0.1. This means that the learning rate was reduced by a factor of 10 every 10 epochs, allowing the optimizer to take smaller steps as the training progressed.

This approach not only helped to stabilize the training process but also allowed the model to fine-tune its weights during the later stages of training, achieving better generalization on the test data. By balancing the initial rapid convergence and the later precision adjustments, the learning rate decay significantly contributed to the model's improved performance.

5) Improving Model Complexity: To further enhance the performance of the neural network, several changes were made to increase the model complexity. The most notable modification was the increase in the number of filters in the convolutional layers, which allowed the network to capture more intricate features in the input data. By expanding the

network's capacity, the model was able to learn more detailed representations, leading to improved feature extraction.

Additionally, average pooling operations were introduced to reduce the spatial dimensions of the feature maps while maintaining the most relevant information. This not only helped to reduce the computational burden but also contributed to the model's ability to generalize better by summarizing the features in a more compact form.

These changes, when combined with the previously discussed techniques, played a crucial role in improving the network's accuracy and robustness. In the improved model too Adam optimizer with a initial learning rate of 0.001 was used. The total number of parameters in this model are **427658**.

After incorporating batch normalization, data augmentation, input normalization, learning rate decay, and increasing model complexity, the final neural network achieved better accuracy and generalization. The model reached a training accuracy of 83.84% and a testing accuracy of 80.08%. These improvements demonstrate the effectiveness of the applied enhancements over the initial naive architecture. The improved architecture is shown in Figure 30.

The accuracy curve for both training and testing data versus epochs is shown in Figure 15, while the loss versus epochs curve is presented in Figure 10. Furthermore, the confusion matrices for the training and testing datasets are depicted in Figures 21 and 22, respectively.

These results emphasize the significance of both architectural modifications and training techniques in developing a neural network capable of generalizing well to unseen data.

C. ResNet, ResNeXt, DenseNet

To enhance the performance and efficiency of the image classification task, three advanced deep learning architectures were implemented: ResNet, ResNeXt, and DenseNet. Each of these architectures represents a significant milestone in deep learning, introducing novel techniques to address challenges such as vanishing gradients, feature reuse, and computational efficiency. For this project, the ResNet-18, ResNeXt-29, and DenseNet-100 variants were utilized. The total number of trainable parameters for these networks is approximately 11173962 for ResNet-18, 27104586 for ResNeXt-29, and 769162 for DenseNet-100. Block diagrams of these models are shown in Figures 31, 32, and 33.

1) ResNet-18: ResNet (Residual Network) revolutionized deep learning by introducing residual connections that help mitigate the vanishing gradient problem in deep neural networks. These shortcut connections bypass one or more layers, allowing the network to learn identity mappings and focus on refining features. The ResNet-18 architecture, with 18 layers, achieves a good balance between complexity and accuracy, making it a popular choice for image classification tasks. The architec

The model employs the Stochastic Gradient Descent (SGD) optimizer with a learning rate of 0.1, facilitating effective parameter updates during training. Momentum is set to 0.9, helping to stabilize training by incorporating prior gradients, while a weight decay of 5×10^{-4} is applied to regularize the model and mitigate overfitting by discouraging excessively large weight values. Additionally, a cosine annealing learning rate scheduler is utilized, with a maximum number of epochs ($T_{\max}$) set to 200. This scheduler gradually decreases the learning rate following a cosine curve, which helps improve convergence by allowing the optimizer to make finer adjustments as training progresses.

2) ResNeXt-29: ResNeXt extends the concept of ResNet by introducing the cardinality dimension, which refers to the number of parallel paths in the network. Instead of increasing the depth or width, ResNeXt enhances representational power by using grouped convolutions. The ResNeXt-29 model, which contains 29 layers, strikes a balance between efficiency and performance by leveraging these grouped convolutions.

The model employs the Stochastic Gradient Descent (SGD) optimizer with a learning rate of 0.1, facilitating effective parameter updates during training. Momentum is set to 0.9, helping to stabilize training by incorporating prior gradients, while a weight decay of 5×10^{-4} is applied to regularize the model and mitigate overfitting by discouraging excessively large weight values. Additionally, a cosine annealing learning rate scheduler is utilized, with a maximum number of epochs ($T_{\max}$) set to 200. This scheduler gradually decreases the learning rate following a cosine curve, which helps improve convergence by allowing the optimizer to make finer adjustments as training progresses.

3) DenseNet-100: DenseNet (Densely Connected Convolutional Network) introduces dense connections between layers, ensuring that each layer has access to the feature maps of all its preceding layers. This approach encourages feature reuse, reduces redundancy, and enhances gradient flow, making DenseNet highly parameter-efficient. The DenseNet-100 model consists of 100 layers and has the fewest parameters among the three models implemented in this project.

The model utilizes the Stochastic Gradient Descent (SGD) optimizer, configured with a learning rate of 0.1 to control the step size during training. A momentum value of 0.9 is included to accelerate convergence by dampening oscillations and accumulating gradients from previous steps. Nesterov acceleration is enabled, which adjusts the gradients based on a look-ahead step to improve optimization efficiency and accuracy. Furthermore, a weight decay of 1×10^{-4} is applied as a regularization term to reduce overfitting by penalizing large weights and promoting simpler models that generalize better to unseen data.

4) Comparison: The implementation of ResNet-18, ResNeXt-29, and DenseNet-100 demonstrated significant improvements over the initial naive architecture. Key observations from these models are summarized below:

- **DenseNet-100:** DenseNet-100, while having the fewest parameters, delivered the lowest performance among the three models. It achieved a training accuracy of 90.16% and a testing accuracy of 86.96%. The architecture's densely connected layers effectively reused features, en-

suring efficient learning and decent generalization. The training and testing accuracy curves are presented in Figure 18, while the loss vs. epochs curve is shown in Figure 13. The confusion matrices for training and testing are depicted in Figures 27 and 28, respectively.

- **ResNeXt-29:** ResNeXt-29, leveraging grouped convolutions, provided a balance between complexity and accuracy. It almost same as DenseNet-100, achieving a training accuracy of 91.87% and a testing accuracy of 86.71%. The training and testing accuracy curves are displayed in Figure 17, while the loss curve is shown in Figure 12. The confusion matrices for training and testing are depicted in Figures 25 and 26.
- **ResNet-18:** ResNet-18 demonstrated the highest performance among the three models, highlighting the effectiveness of residual connections in mitigating vanishing gradients and accelerating convergence. It achieved a training accuracy of 95.346% and a testing accuracy of 89.79%. Figures 16 and 11 illustrate the accuracy and loss trends, respectively. The confusion matrices for training and testing are shown in Figures 23 and 24.

D. Comparison of all the models trained

Model	Number of Parameters	Train Accuracy	Test Accuracy	Inference run-time*
Naive Neural Network	71K	86%	64%	0.362 ms
Improved Neural Network	427K	83.84%	80.08%	0.402 ms
ResNet	11.1M	95.346%	89.79%	1.053 ms
ResNext	27.1M	91.87%	86.71%	1.878 ms
DenseNet	769K	90.16%	86.96%	5.885 ms

* The inference run-time is calculated as the total time taken to process the entire test set, divided by 10,000 (the number of test images). The WPI Turing Cluster, equipped with 2 A30 GPUs, was used for this calculation.

The custom model achieved an accuracy of 80%, compared to ResNet-18 (89%), ResNeXt-29 (87%), and DenseNet-100 (86%) on the CIFAR-10 dataset. While the custom model demonstrates reasonable performance for its simplicity, the advanced architectures outperform it due to their innovative design principles.

ResNet-18 achieved the highest accuracy by leveraging residual connections, which mitigate the vanishing gradient problem and enable deeper feature hierarchies. These residual blocks allow the model to efficiently learn complex features, contributing to its superior performance.

ResNeXt-29 performed slightly below ResNet-18, benefiting from grouped convolutions to increase cardinality. This approach strikes a balance between complexity and accuracy, allowing ResNeXt to learn diverse features without significantly increasing parameters.

DenseNet-100, while achieving slightly higher accuracy than ResNeXt-29, is highly parameter-efficient due to its densely connected layers. These connections promote feature reuse and effective gradient flow, ensuring good generalization despite having fewer parameters.

The custom model, with its simpler architecture of three convolutional and fully connected layers, lacks advanced components like residual or grouped connections. Although it incorporates batch normalization and dropout, it cannot capture the complex hierarchical features that the other models effectively learn. This limitation explains its lower accuracy compared to the advanced architectures.

These results demonstrate the impact of architectural innovations, with ResNet-18 emerging as the most effective due to its ability to train deeper networks while maintaining stability and performance.

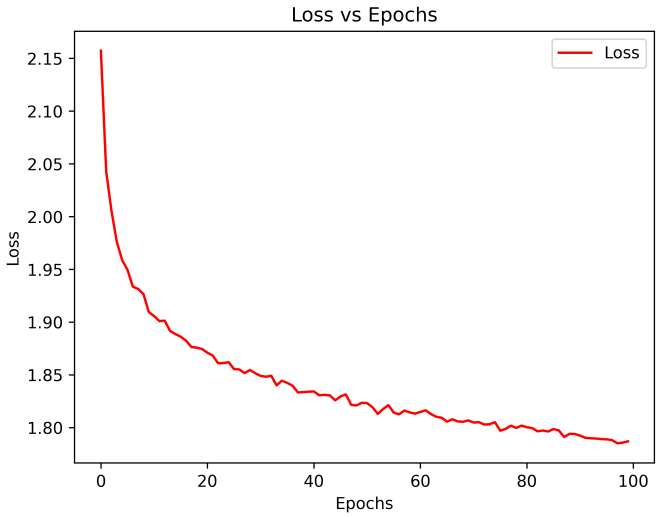

Fig. 9. Training and Testing Loss - My Neural Network

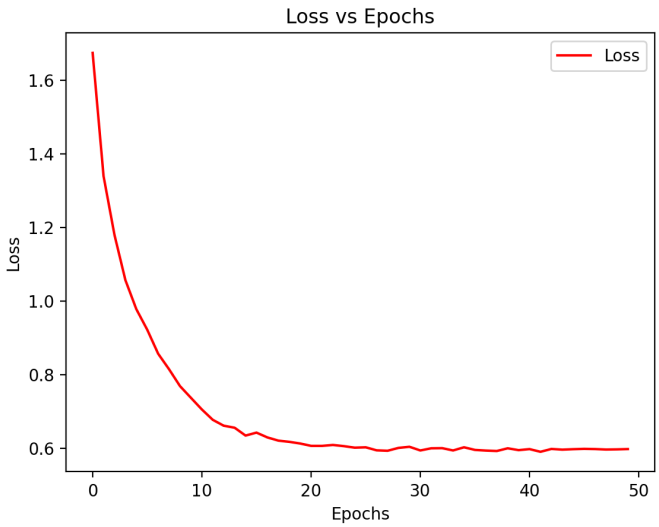

Fig. 10. Training and Testing Loss - My Improved Neural Network

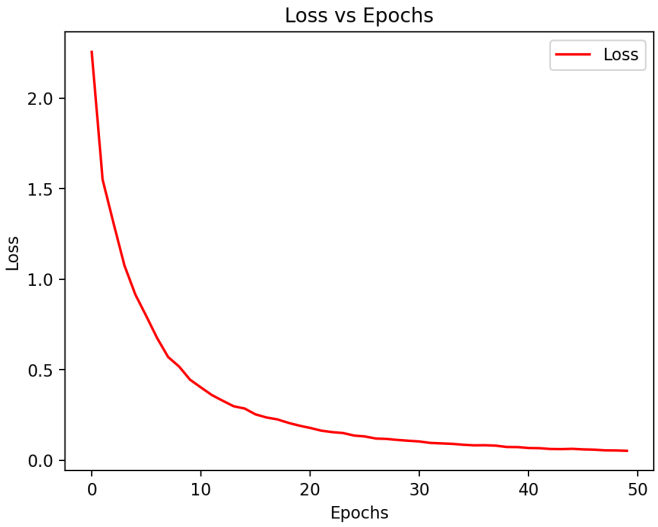

Fig. 11. Training and Testing Loss - ResNet

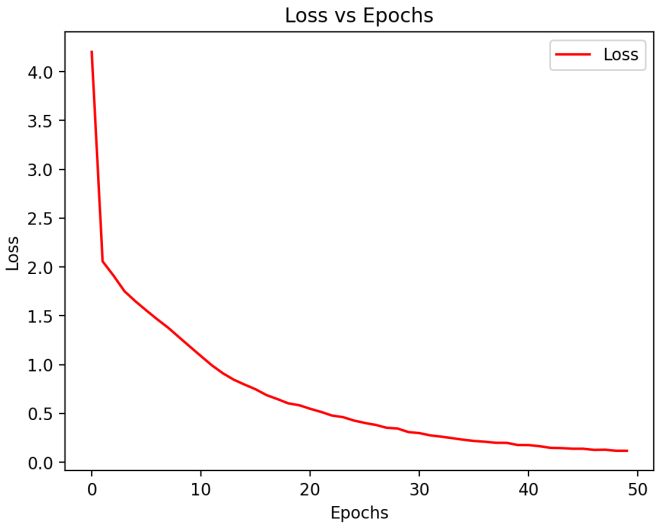

Fig. 12. Training and Testing Loss - ResNext

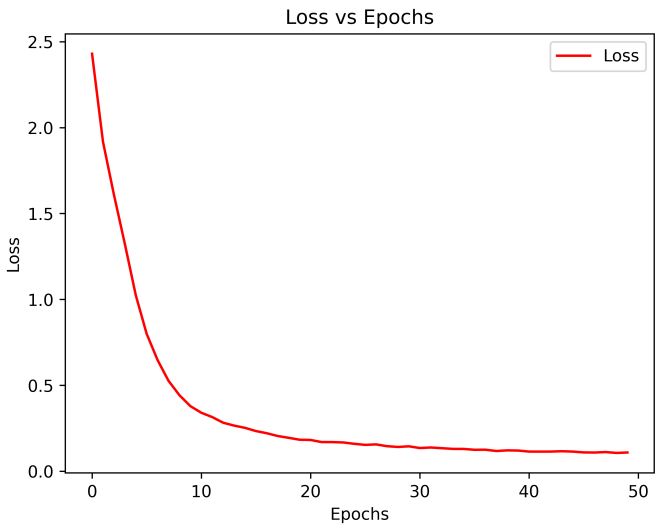

Fig. 13. Training and Testing Loss - DenseNet

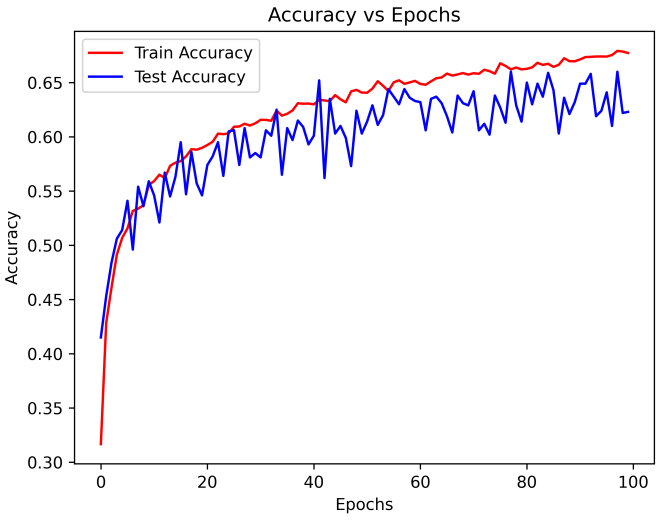

Fig. 14. Training and Testing Accuracy - My Neural Network

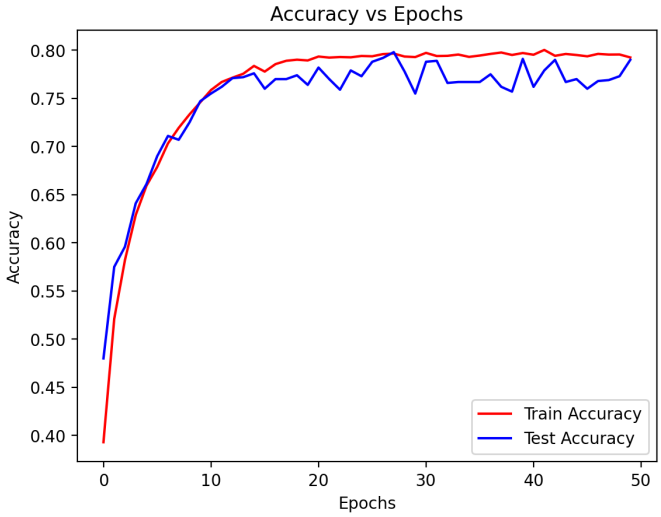

Fig. 15. Training and Testing Accuracy - My Improved Neural Network

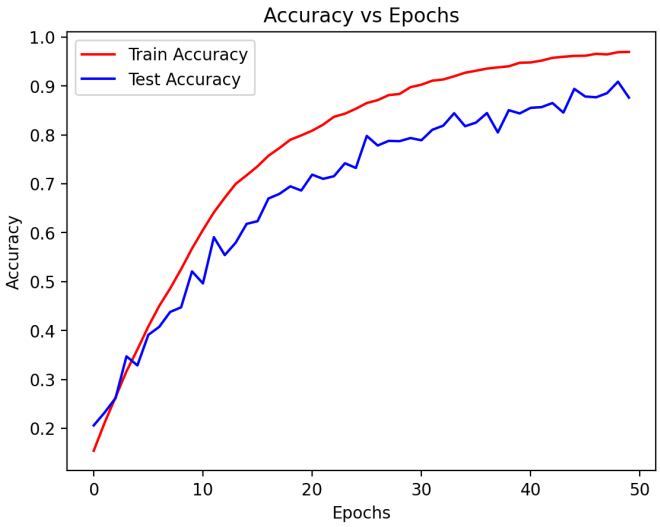

Fig. 17. Training and Testing Accuracy - ResNext

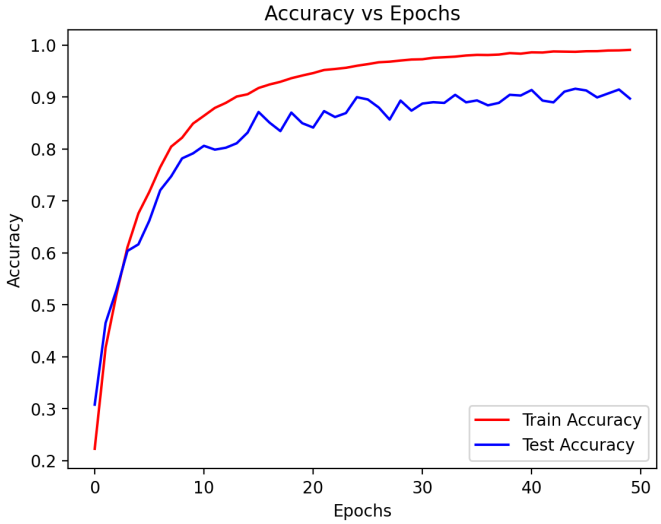

Fig. 16. Training and Testing Accuracy - ResNet

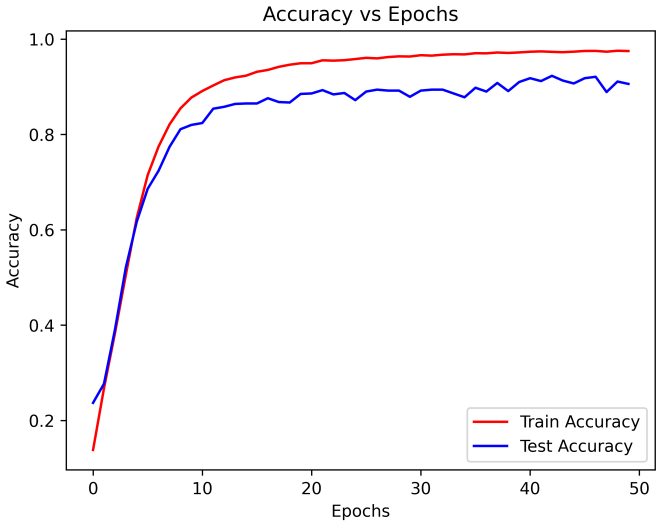

Fig. 18. Training and Testing Accuracy - DenseNet

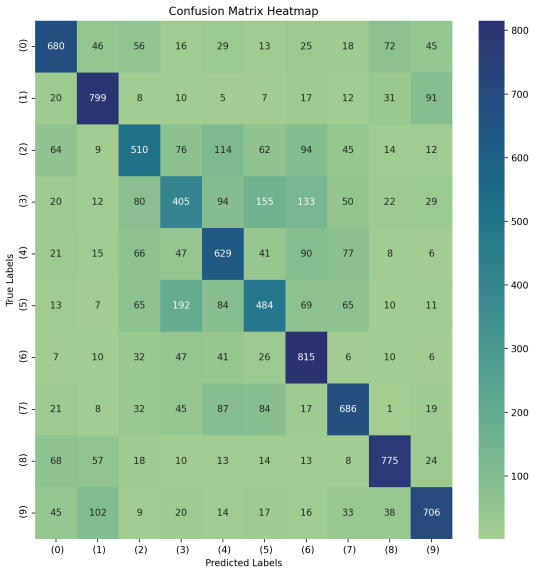

Fig. 19. Training Confusion Matrix - My Neural Network

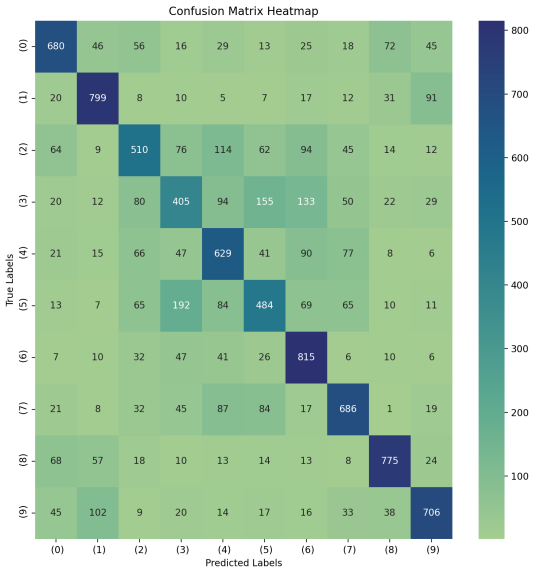

Fig. 20. Testing Confusion Matrix - My Neural Network

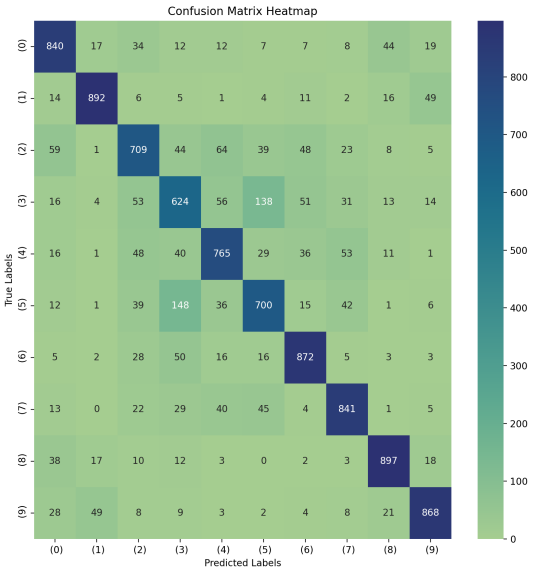

Fig. 21. Training Confusion Matrix - My Improved Neural Network

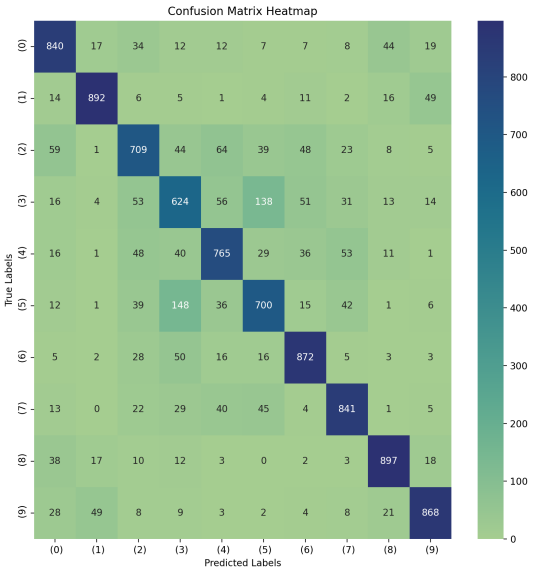

Fig. 22. Testing Confusion Matrix - My Improved Neural Network

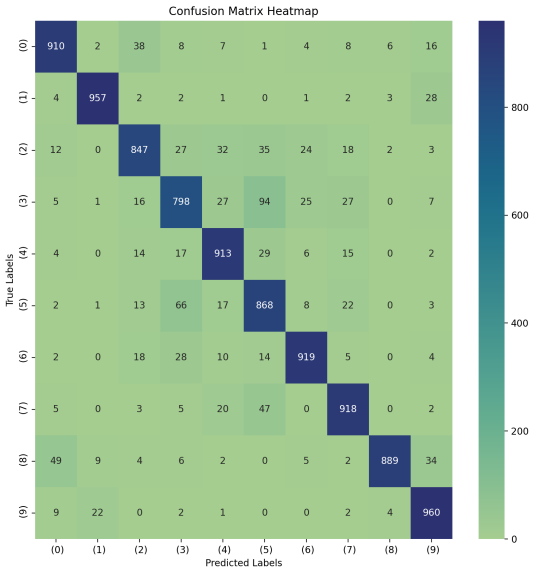

Fig. 23. Training Confusion Matrix - ResNet

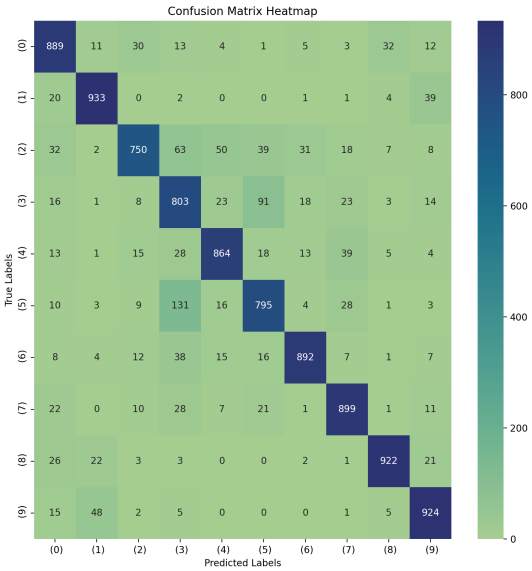

Fig. 25. Training Confusion Matrix - ResNext

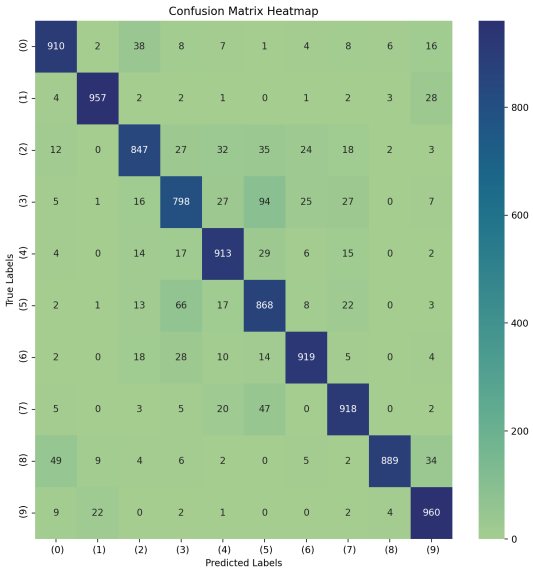

Fig. 24. Testing Confusion Matrix - ResNet

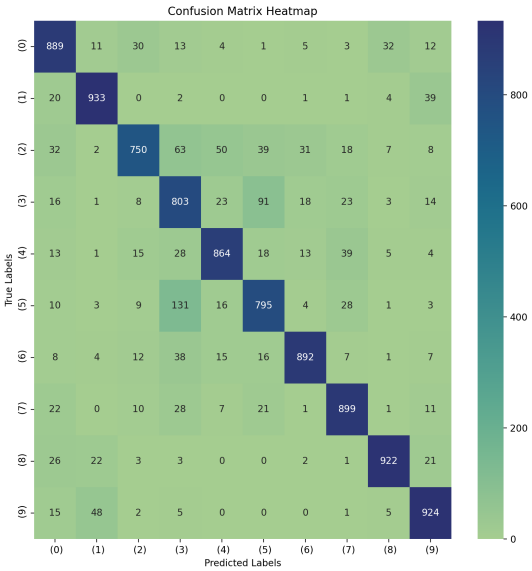

Fig. 26. Testing Confusion Matrix - ResNext

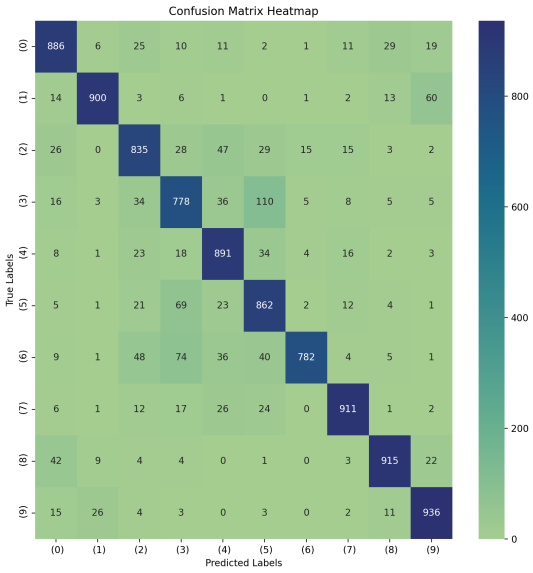

Fig. 27. Training Confusion Matrix - DenseNet

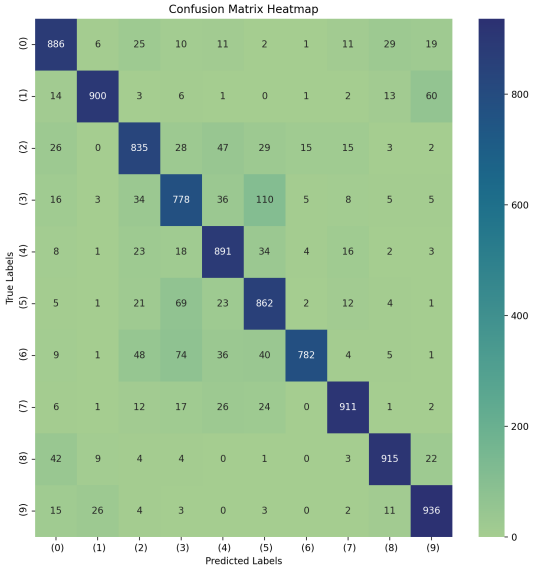

Fig. 28. Testing Confusion Matrix - DenseNet

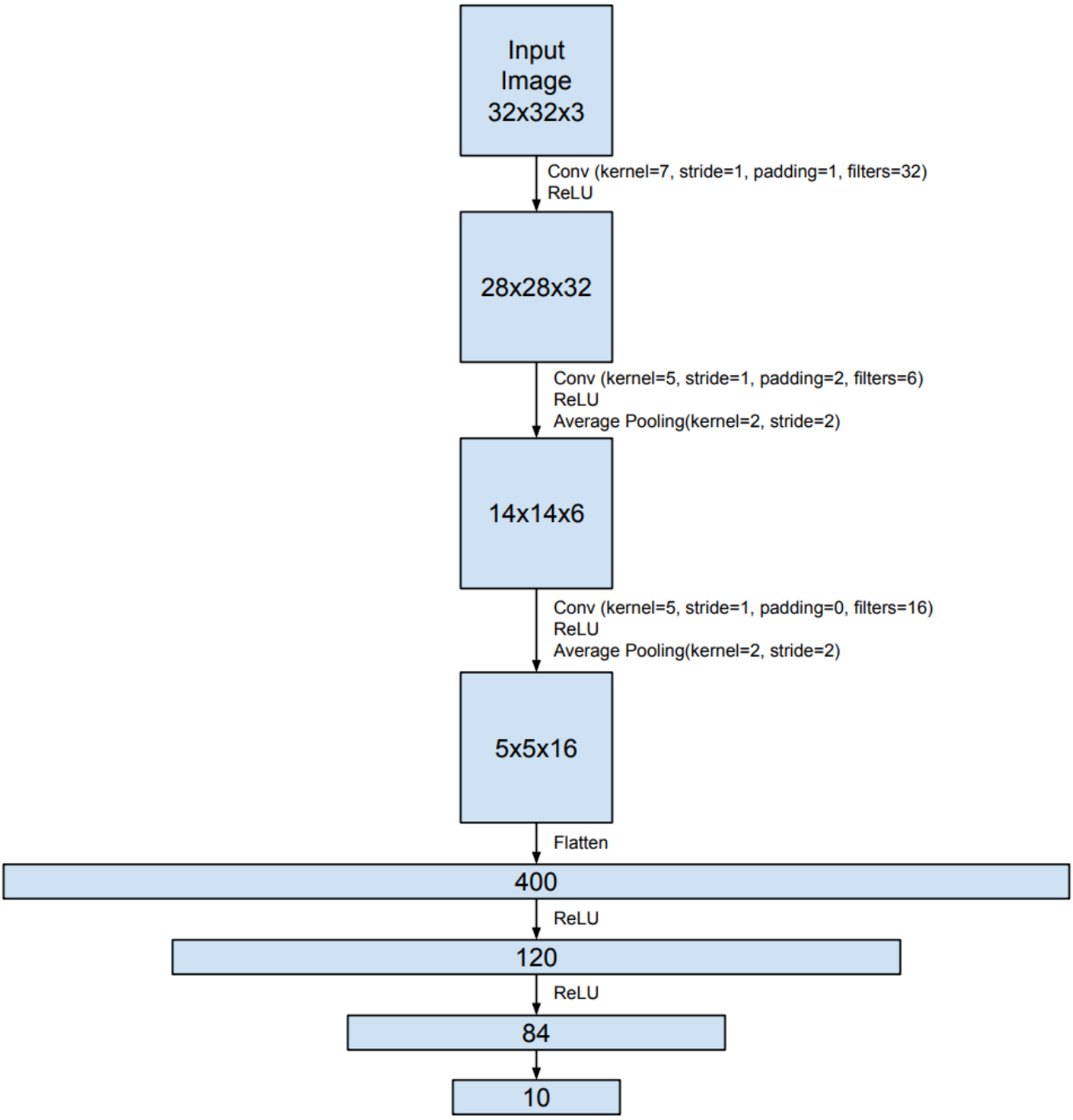

Fig. 29. My Neural Network Architecture

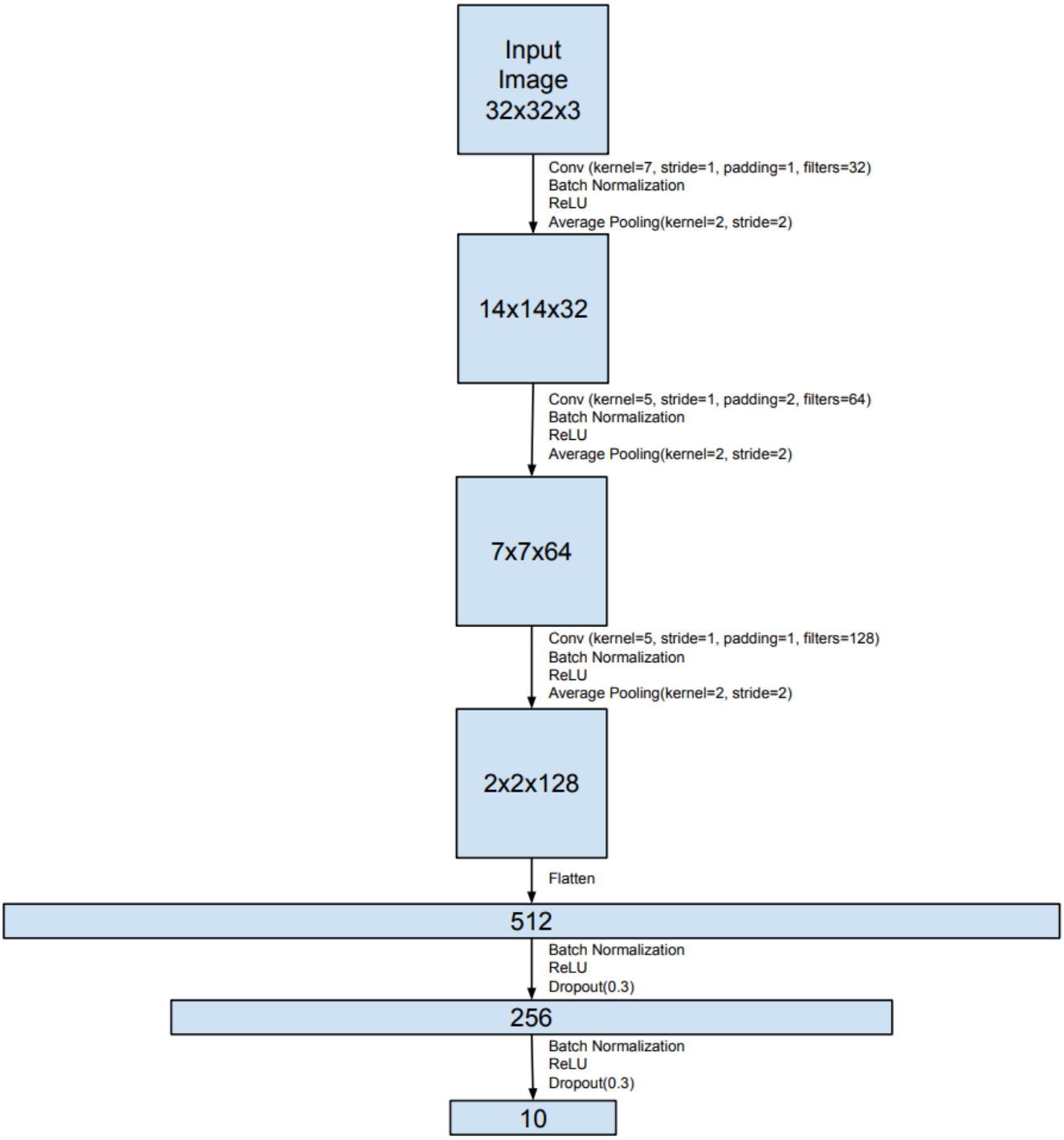

Fig. 30. Improved Neural Network Architecture

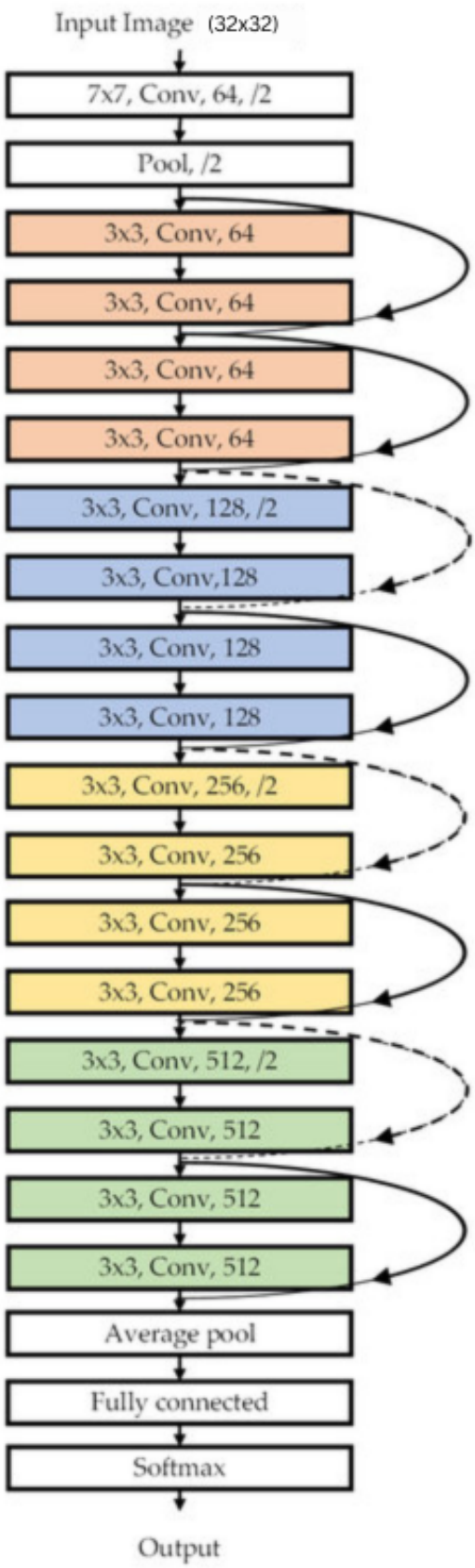

Fig. 31. Block Diagram of ResNet

Fig. 32. Block Diagram of ResNext

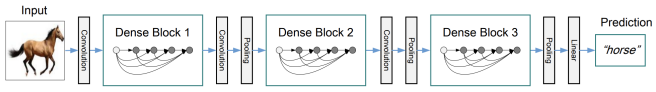

Figure 2: A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature-map sizes via convolution and pooling.

Fig. 33. Block Diagram of DenseNet