Implementation of Pb-lite and Comparison of Network Architectures on CIFAR-10

Yaşar İdikut
Worcester Polytechnic Institute
Worcester, Massachusetts
Email: yidikut@wpi.edu
Using 4 Late Days

***Abstract*—This introduction project consists of two independent phases. In phase one, we implement a lighter version of Pb edge detection algorithm from [1] and compare against Sobel and Canny baselines. In phase two, we compare between the following architectures for the CIFAR-10 dataset [2]: simple CNN-based model, hyperparameter-tuned CNN-based model, ResNet-based model, ResNeXt-based model, and DenseNet-based model.**

I. Introduction

The first phase focuses on improving upon the existing methods for edge detection. A rather recent work from 2011 [1] on edge detection essentially relies on texture, color, and brightness discontinuities and assigns each pixel a probability for being an edge. This project doesn't implement the whole algorithm introduced in that work. This project takes its main ideas from there and implements a lighter version of it.

The second phase deals with implementing and comparing different network architectures to be used for the CIFAR-10 dataset [2]. CIFAR-10 is a dataset of labeled images where each image belongs to one of ten classes. CNN-based architectures are generally preferred over simple neural network layers because CNNs can better consider the spatial information. All the architectures implemented and compared here are CNN-based but vary in complexity.

II. Phase I: Pb-lite Edge Detection

The lighter version of the original Pb algorithm [1] is implemented as follows. First, I created three types of filter banks. Filter banks are basically a collection of filters that can be used to detect texture changes in the image. In total, I created 32 DoG filters, 96 Leung-Malik filters (48 small scale, another 48 large scale), and 40 Gabor filters. Then, these filter are used to create the Texton map, which can be used as a measure of different textures on the image. After filtering the image using selected filters, I create a K-means cluster and assign each pixel an ID that represents its texture. The brightness maps and the color maps are generated using the gray scale and colored version of the image, respectively. I create another K-means cluster and assign each pixel an ID that represents its brightness and color, respectively. Afterwards, I take gradients of texton, brightness, and color maps to highlight the edges. A higher gradient would signal that there is a higher probability that pixel is an edge. This is intuitive because objects are separated from the background or other objects by certain discontinuities in the map values. Finally, these gradient maps are averaged and combined with the outputs of the baseline Sobel and Canny edge detection algorithms. The resulting map is the edge probability value for each pixel.

Figure 1: The image I use to evaluate and compare my implementation of Pb-lite

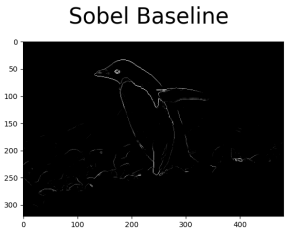

Figure 2: Visualization of the Sobel baseline

In this phase, I only use the image in Figure 1

A. Sobel and Canny baselines

B. Evaluation

I wasn't able to do this.

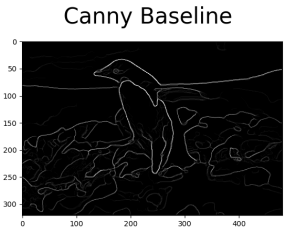

Figure 3: Visualization of the Canny baseline

III. Phase II: Network Architectures on CIFAR-10

Here, I present my brief explanations for each network architecture and analyze the results at the end. The total inference time and number of parameters comparison is provided in the last subsection.

A. Initial CNN-based Model

The first model is a very simple one. It consists of two convolution layers and two fully connected layers. For a detailed view of the network, please see figure 4. The Adam optimizer is used. I trained this model for 20 epochs with a minibatch size of 16 and a learning rate of 0.001. The plots show signs of convergence both on the training and testing dataset, however, the accuracy and loss are not great. These can be seen in figures 5 6 7 8. Lastly, confusion matrix for this model can be seen in figures 9 and 10. As expected, the model performs better in the training dataset and has higher errors on the testing dataset.

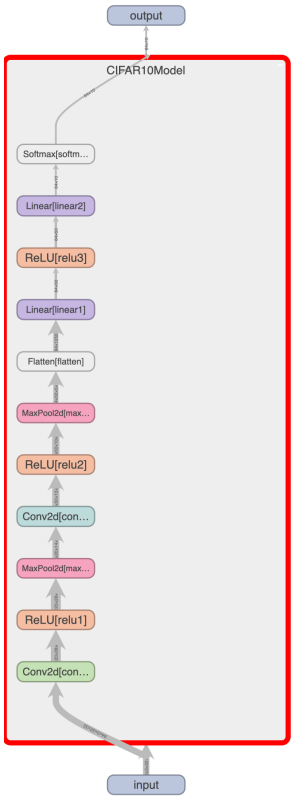

Figure 4: Initial CNN-based model architecture

Figure 5: Initial CNN-based model training loss

Figure 6: Initial CNN-based model training accuracy

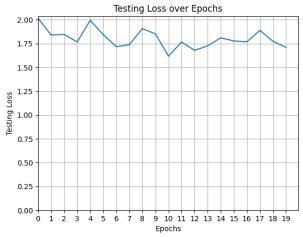

Figure 7: Initial CNN-based model testing loss

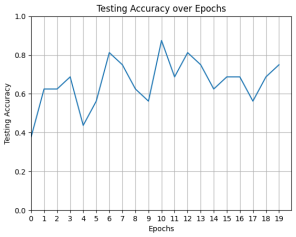

Figure 8: Initial CNN-based model testing accuracy

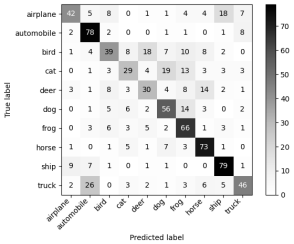

Figure 9: Initial CNN-based model confusion matrix on training data

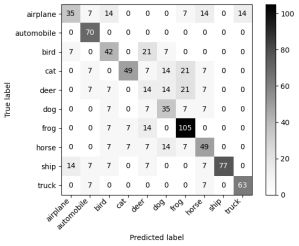

Figure 10: Initial CNN-based model confusion matrix on testing data

B. *Hyper-parameter Optimized CNN-based Model*

To improve upon the initial model, I used hyperparameter tuning. I iterated over different minibatch sizes and learning rates for a small number of epochs. Then, I picked the combination of parameters with the lowest loss. Finally, I used a minibatch size of 64 and a learning rate of 0.001. This time, results got better as can be seen in figures 11 12 13 14. Lastly, confusion matrix for this model can be seen in figures 15 and 16.

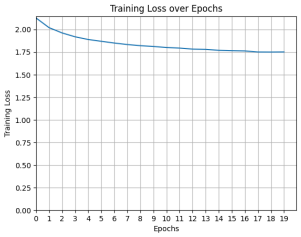

Figure 11: Hyper-parameter optimized CNN-based model training loss

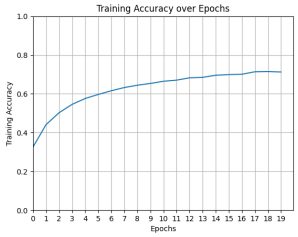

Figure 12: Hyper-parameter optimized CNN-based model training accuracy

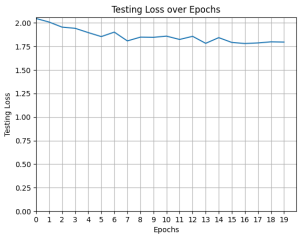

Figure 13: Hyper-parameter optimized CNN-based model testing loss

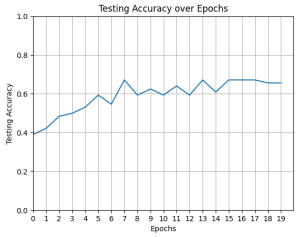

Figure 14: Hyper-parameter optimized CNN-based model testing accuracy

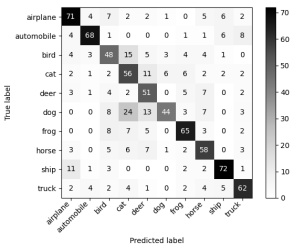

Figure 15: Hyper-parameter optimized CNN-based model confusion matrix on training data

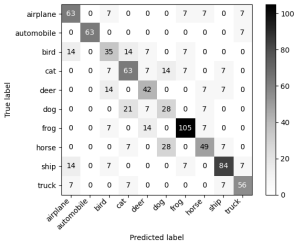

Figure 16: Hyper-parameter optimized CNN-based model confusion matrix on testing data

C. *ResNet-based Model*

The next architecture I experimented with is a ResNet-based model. This architecture also relies on convolution layers, but it is less prone to information getting lost/vanished as data proceeds through the layers. The reason is because this architecture is able to directly connect some non-neighboring layers by simply skipping them. The visualization of the architecture can be seen in 17. Specifically, I implemented the ResNet-50 architecture and resized images from 32x32 to 224x224 before supplying them to the model.

The Adam optimizer is used. I trained this model for 20 epochs with a minibatch size of 128 and a learning rate of 0.001. The plots show very good signs of convergence both on the training and testing dataset. These can be seen in figures 18 19 20 21. Lastly, confusion matrix for this model can be seen in figures 22 and 23.

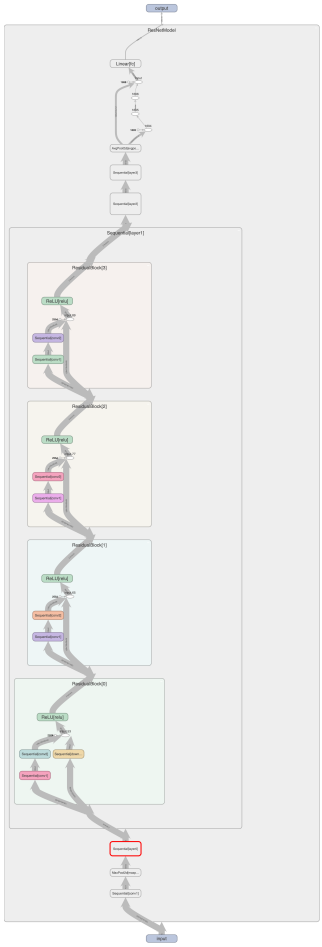

Figure 17: ResNet-based model architecture

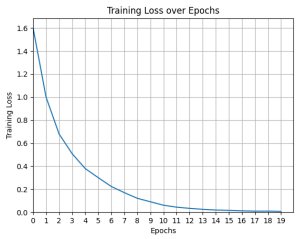

Figure 18: ResNet-based model training loss

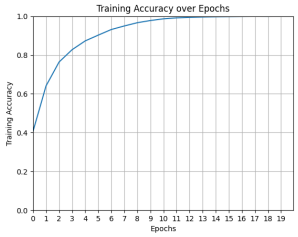

Figure 19: ResNet-based model training accuracy

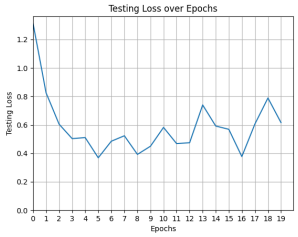

Figure 20: ResNet-based model testing loss

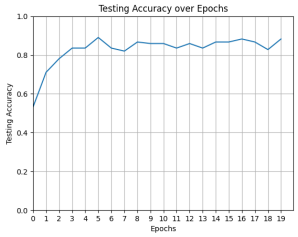

Figure 21: ResNet-based model testing accuracy

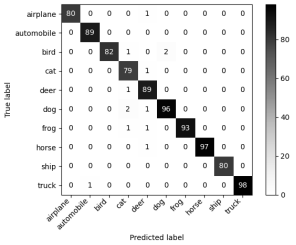

Figure 22: ResNet-based model confusion matrix on training data

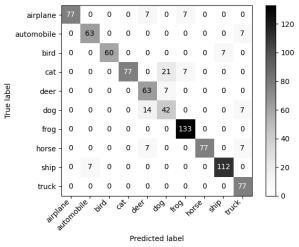

Figure 23: ResNet-based model confusion matrix on testing data

D. ResNeXt-based Model

The next architecture I experimented with is a ResNeXt-based model. This architecture builds upon ResNet and instead of enlarging the network in depth, it prioritizes growing wide. Layers are grouped into blocks and inputs feed into the blocks k times. Output from these blocks are aggregated and fed into the next block. This widens the network while not changing the depth. The benefit of this approach is again to retain long-term information. I used a cardinality value (k) of 2, and created 3 blocks. The visualization of the architecture can be seen in 24. In the figure, I show a detailed setup of one of the blocks.

Many of the following setup is similar to the ResNet model. The Adam optimizer is used. I trained this model for 20 epochs with a minibatch size of 128 and a learning rate of 0.001. The plots show very good signs of convergence both on the training and testing dataset. These can be seen in figures 25 26 27 28. Lastly, confusion matrix for this model can be seen in figures 29 and 30.

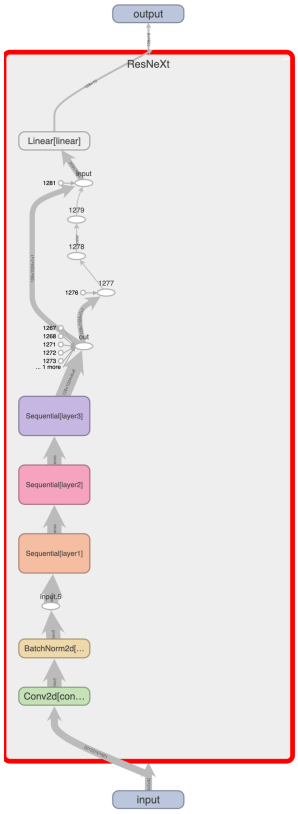

Figure 24: ResNeXt-based model architecture

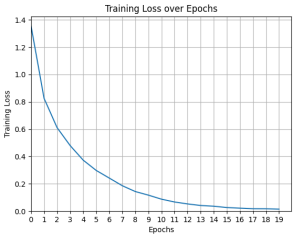

Figure 25: ResNeXt-based model training loss

Figure 26: ResNeXt-based model training accuracy

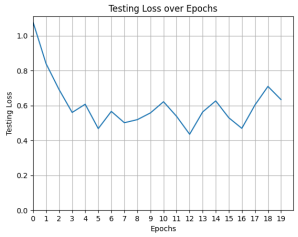

Figure 27: ResNeXt-based model testing loss

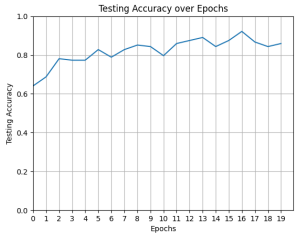

Figure 28: ResNeXt-based model testing accuracy

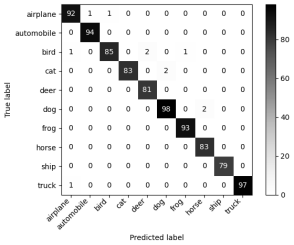

Figure 29: ResNeXt-based model confusion matrix on training data

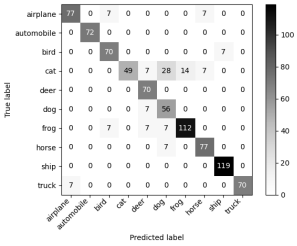

Figure 30: ResNeXt-based model confusion matrix on testing data

E. DenseNet-based Model

The next architecture I experimented with is a DenseNet-based model. In this architecture, each layer in a dense block are all connected with each other. This is possible made possible by passing both a previous layer's input and output. This way, the next layer, also processes the inputs from all the previous layers. In between the dense blocks, there exists a transition layer that ensures that the dimension of the data is kept to the configured constant value. I implemented a version of DenseNet-121 that is compatible with our dataset image size. This model consists of 4 dense blocks and each block has 6, 12, 24, and 16 layers in order. The visualization of the architecture can be seen in 31. In the figure, I show a detailed setup of one of the blocks.

Many of the following setup is similar to the previous models. The Adam optimizer is used. I trained this model for 20 epochs with a minibatch size of 128 and a learning rate of 0.001. The plots show very good signs of convergence both on the training and testing dataset. These can be seen in figures 32 33 34 35. Lastly, confusion matrix for this model can be seen in figures 36 and 37.

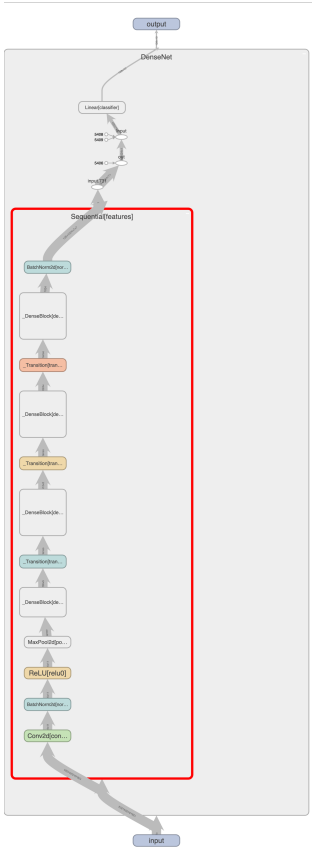

Figure 31: DenseNet-based model architecture

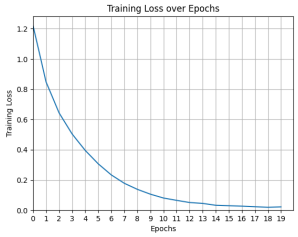

Figure 32: DenseNet-based model training loss

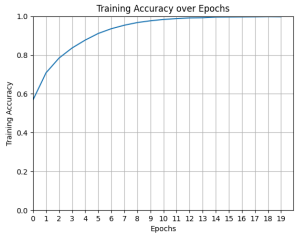

Figure 33: DenseNet-based model training accuracy

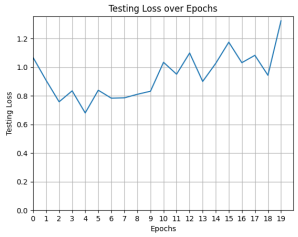

Figure 34: DenseNet-based model testing loss

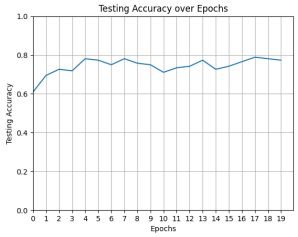

Figure 35: DenseNet-based model testing accuracy

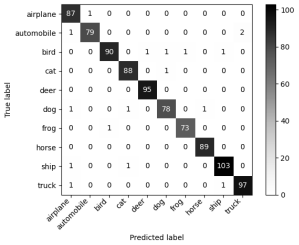

Figure 36: DenseNet-based model confusion matrix on training data

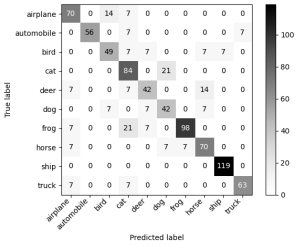

Figure 37: DenseNet-based model confusion matrix on testing data

F. Comparison of Models

Overall, all three of the complex models with a rather complex network architecture showed great accuracy while the initial model, even hyperparameter optimized, gave only acceptable results. The models with complex architectures perform almost perfectly on the training dataset and still do a good job in the testing dataset. This signals to me that there is a need for increased regularization such that the model performs better on an unseen dataset.

In table I, you can see the number of parameters used in each model. There were different ways to calculate the number of parameters in a model. The one recommended by the assignment page (”Named” parameters) yields much lower number. This is because each ”named” parameter is actually a matrix with a lot more trainable parameters. The number of trainable parameters are calculated by counting all the trainable parameters in the model object's *parameters* method. In table II, you can see the inference speed of each model. To make the inference times more realistic, I report the time it takes to evaluate a batch of inputs rather than evaluating a single image.

Model	**Number of ”Named” Parameters**	**Number of Trainable Parameters**
Simple CNN	8	89630
ResNet	144	21298186
ResNeXt	95	9128778
DenseNet	364	6852074

Table I: Number of parameters per model

Model	**Minibatch (32) Inference Time**
Simple CNN	0.0011 seconds
ResNet	0.0104 seconds
ResNeXt	0.01080 seconds
DenseNet	0.0413 seconds

Table II: Minibatch inference time per model.

REFERENCES

[1] P. Arbeláez, M. Maire, C. Fowlkes, and J. Malik, “Contour detection and hierarchical image segmentation,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 5, pp. 898–916, 2011.

[2] A. Krizhevsky, “Learning multiple layers of features from tiny images,” *University of Toronto*, 05 2012.