Deep Learning mini project
Modified ResNet

Junyao Chen,*Denys Fenchenko,*Gustavo Sandoval*

{jc9723, df1911, gs157}@nyu.edu

Abstract

The goal of this project is to design and train a Deep Residual Network (ResNet) architecture with fewer than 5 million parameters achieving the best possible accuracy in the CIFAR-10 dataset. Our work can be broadly divided into two parts. The first is parameter reduction, which aims to reduce the model to under 5 million parameters. The second is optimization, which aims to increase the model's accuracy as much as possible, given the number of parameters. In this work, we show experimental results for two different parameter reduction techniques: **Pruning** and **Layer Removal**. We show that while both of these parameter reduction techniques produce different model sizes, the results in terms of accuracy are very similar.

Introduction

The goal of our project is to build a ResNet with a maximum of 5 million parameters and achieve the best accuracy possible given such a constraint. We decided to use untrained ResNet-18 architecture with 11 million parameters as our backbone and further optimize the size and accuracy. We applied 2 primary size reduction methods - pruning and layer removal and several different optimization methods to achieve the best performance. We use ResNet-18 from the recommended repo (Liu 2020); with default PyTorch parameter initialization.

Methodology

We will start by describing various architectural decisions we have made to minimize the parameter count from the original and maximize the accuracy. Firstly, we tried two different approaches to parameter reduction: pruning and layer removal. Secondly, we tried different options for model optimization to achieve the best accuracy possible. In the following sections we describe out work.

Parameter Reduction

Our first objective was to reduce the number of ResNet-18 parameters to under 5 million. In this section, we will describe the two approaches we experimented with: Pruning and Layer Removal. In both of the approaches, we split our data into three sets. Training set: 45,000 samples, Validation set: 5,000 samples, and Test set: 10,000 samples. We use data loaders to load and augment the data. We further use the train loop to calculate the training loss and accuracy, the evaluation loop to calculate validation loss and accuracy, and a test loop to calculate the test loss and accuracy.

Pruning Our first approach to reduce the number of parameters in the ResNet-18 was to use Neural Network Pruning (Lecun, Denker, and Solla 1989). At a high-level, pruning a Neural Network means removing certain connections and is used for the following reasons (Paperspace 2020): Regularization of over-parameterized functions; Reducing the size of the model (which is our primary case), and Reducing the energy cost during training

As noted by (Lecun, Denker, and Solla 1989), removing a percentage of parameters doesn't result in a linear percentage reduction in a model's performance. The accuracy of the model sometimes might increase. To explain this phenomenon, we can use a biological analogy. The number of synapses in the brain declines with age, which is the process referred to as synaptic pruning. This is because the brain eliminates the connections that aren't useful, leaving only the important ones. A somewhat similar process takes place during the pruning of deep neural networks, as many of the connections that wouldn't be useful are getting removed. When it comes to pruning, there are two different approaches: we can prune the network before or after training. In this project, our only option is to prune the network before we start the training process, as we must reduce the number of parameters.

In our work, we experimented with two different Python pruning modules. Our first attempt was to use the *prune* (PyTorch 2022) module. However, this module was primarily designed for pruning the model after the training since instead of removing the connections from the network during pruning; it sets the weights to be pruned to 0. As a result, if the model is getting pruned before training, during the training process, it will set the pruned weights back to the appropriate values, effectively undoing the pruning. This motivated us to adopt another pruning package - *torch_pruning* (Fang 2022).

During pruning certain layers must be pruned simultaneously to guarantee the correctness of networks. The Torch-

*These authors contributed equally.

Our repo: https://github.com/jychen630/dl-fall22-mini-project.git

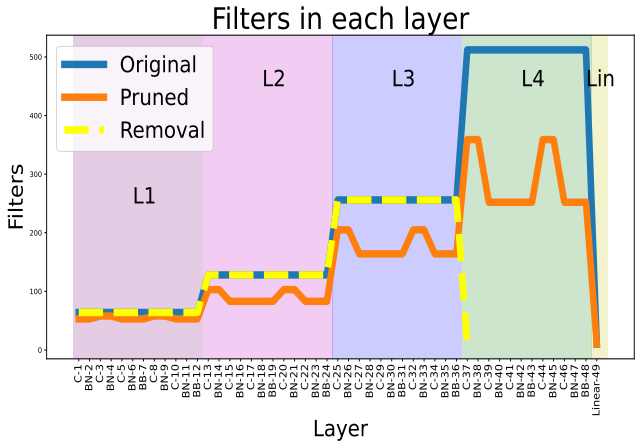

Figure 1: Filters per layer for each of our two methods: **Pruning** and **Layer Removal**. We also include the Original size per layer

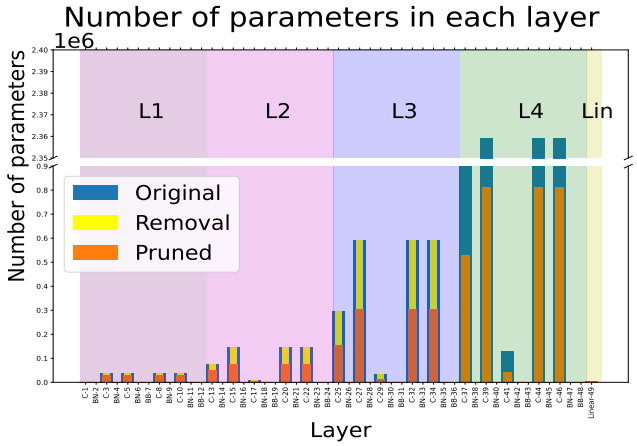

Figure 2: Number of parameters in each layer for orignal ResNet model, Pruning and Layer Removal models.

pruning module detects and handles layer dependencies for us.

There are two main steps. Tracing of the model as a computational graph, which helps us understand the flow of computations within the network, and using computational graph to build a dependency graph. Dependency Graph will record the dependencies between all possible pruning operations. If we prune a single layer, Torch-Pruning will collect all affected layers by propagating the pruning operation on the dependency graph and then return a *Pruning Plan* for final pruning.

As mentioned before, our original ResNet-18 network initially had 11.2 million parameters. The percentage of connection pruned depended on a network layer. These steps can be seen in our code repo in the `pruning_and_optuna_resnet_18_submission.ipynb` in the `prune_model` function. As a result of pruning, we reduced the number of parameters in the ResNet-18 model from 11.2 to 4.5 million. Figure 1 and Figure 2 correspondingly show number of filters and number of parameters being reduced in each layer using pruning and layer-removal comparing to those in original ResNet. Each layer drops some filters and parameters which reduces the total number of parameters.

Layer Removal The second method for parameter reduction we tried is layer removal. In this method we removed the 4th layer of ResNet. We decided to remove this layer because it has the highest number of parameters with 512 outgoing channels. After removing this layer, our network went down from 11.2 to 2.8 million parameters. Referring to Figure 1 and Figure 2, since removing one layer does not affect the number of parameters or filers in other layer, so in the first three layers (L1, L2, and L3), number of filters and parameters are the same while in the fourth layer, they drop to zero.

Model Optimization

Our next step in this project is to increase the accuracy of our compressed ResNet-18 model. To achieve that, our main focus is finding the model hyperparameters that would maximize the accuracy. Deep Neural Networks have numerous hyperparameters that can be fine-tuned to achieve better performance. Namely, we can adjust the values of the following parameters: *learning rate, optimizer, batch size, momentum, number of residual blocks, kernel size* and many more. There are several primary approaches to hyperparameter search. The first is trial and error, where the researcher tries different combinations of hyperparameters, training and evaluating the model on each combination. The second one is automated, hyperparameter search.

Dataset Augmentation Since CIFAR-10 is a small dataset by modern standards at just 60,000 images (Krizhevsky 2009), our initial approach was to augment the dataset by applying various transforms. Data augmentation is often used to increase both the amount and diversity of data by randomly "augmenting" it. The main goal of data augmentation is to teach a model about invariances in the data domain. In this project, we applied the following transforms to our data: RandomHorizontalFlip, RandomCrop and AutoAugment. The *RandomHorizontalFlip* transform randomly flips the image with a given probability. The *RandomCrop* transform crops an image at a random location. We further apply normalization to our data, using PyTorch normalization transform. While augmentation is performed on training data only, normalization is performed on all datasets.

Automated Hyperparameter search The trial and error approach is often the first step. However, if we want to test all possible combinations of hyperparameters, this approach will not be feasible. For this reason, automatic hyperparameter tuning tools have been developed. RayTune (Liaw et al. 2018) and Optuna (Akiba et al. 2019) are among the most popular ones.

In our project, we started by experimenting with several hyperparameter combinations by hand, which gave us

a sense of approximate ranges that could result in a good performance. We then conducted a number of "trials" using the Optuna software in order to establish the best possible combination of hyperparameters for our model. In this section we will describe experiments that we've performed and results we were able to achieve.

Hyperparameter tuning frameworks work by conducting several "trials". Each trial consists of a particular set of hyperparameters, and produces an accuracy score. Before each trial, the complete network needs to be set up from scratch. In the end, the best set of hyperparameters is being returned. (i.e., the one leading to the highest accuracy)

There are various optimizations that frameworks like Optuna implement to speed up hyperparameter search and make it as efficient as possible. For instance, Optuna implements pruning, which in this context means early termination of an unsuccessful trial. Furthermore, Optuna algorithms can be parallelized for an efficient search.

The first main component of Optuna is the Sampling Strategy. The sampling algorithm decides "where to look" (i.e which hyperparameters to try). On the other hand, the pruning algorithm decides whether a particular strategy looks promising. If it doesn't, it stops pursuing it without completing the full training and testing and cycle. Let us now look at each of these components in more detail.

The Samplers decide where to look. As opposed to performing a Random Search, most of the samples use Bayesian filters to find places where it has had the best results and focus future searches son those areas. This results in a much more efficient search, then brute force trial and error (Optuna 2022). We also use pruner to terminate the trials early if they are deemed unsuccessful. This dramatically speeds up the processing time and reduces the computational cost. Finally, we are visualizing our results from automated hyper parameter search using Optuna visualization functions. You can view our experiments in `pruning_and_optuna_resnet_18_submission.ipynb` notebook.

Adaptive Learning Rate Previous hyper-parameter tuning methods find a constant learning rate for the model during optimizer iterations. The idea of adaptive learning rate is that the learning rate can be adjusted according to the model's performance during the training. As an example the learning rate decreases by some order of magnitude when the model performance reaches a plateau, making the step size relatively smaller and thus accelerating the convergence. The learning rate can be increased again if the performance hardly improves.

To update the learning rate, we picked three commonly used learning rate schedulers, *CosineAnnealingLR*, *LinearLR*, *StepLR* and *CyclicLR*. Figure 3 shows the progress of training and validation accuracies for each learning rate scheduler.

Implementation To implement the update rule, we called PyTorch built-in function from `torch.optim.lr_scheduler` module. Under each update rule, the model is trained for 50 epochs.

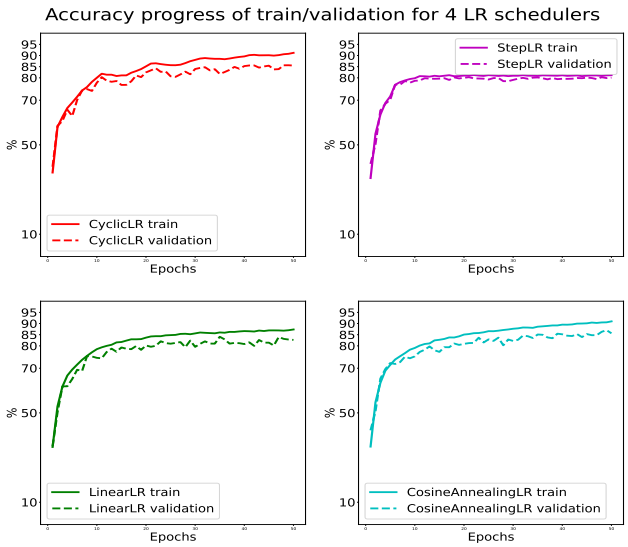

Figure 3: Progress of training and validation accuracies for each learning rate scheduler

Results

Model Reduction Results

The first part of our project focused on reducing the number of parameters in our model to under 5 million using either Pruning or Layer Removal.

Model Reduction Using Pruning Our first approach to reducing the number of parameters in the model was random local pruning. Namely, given layers 1 through 8, we pruned 10% of layers 1 and 2. 20% of layers 3 to 6 and 30% of layers 7 and 8.

As a result, the total number of parameters in our model was reduced from 11.2 million to 4.5 million.

After training the model for 200 epochs, we are able to achieve 97% training accuracy and 91% validation accuracy.

Model Reduction Using Layer Removal Our second approach is to reduce the number of parameters in a model by removing one of the layers. Since the last layer (layer 4) of our ResNet contains most of the parameters and is likely unnecessary for a small dataset like CIFAR-10, we decided to remove it. As a result, the number of parameters in our model reduced form 11.2 million down to 2.8 million. The training curves for that model can be seen in Figure 4. As we can see after training the model for 200 epochs we achieve 97.1%training accuracy and 91.5% validation accuracy.

Hyperparameter Search Results After model size reduction, the next part of our project is model optimization. For this we use hyperparameter fine-tuning. In this section we will consider two primary approaches to hyperparameter fine-tuning: manual and automatic hyperparameter search.

Manual Hyperparameter Search In this approach, we manually set hyperparameters to some fixed value. We focused on choosing the learning rate, batch size, optimizer, and the number of training epochs. Namely, Learning Rate:

Optimizer	Train Acc.	Validation Acc.	Test Acc.	Epochs	Initial LR	Hyperparams
CosineAnnealingLR	90.976%	85.620%	91.340%	50	1e-2	T_max=200
LinearLR	87.342%	82.680%	88.500%	50	1e-2	-
StepLR	81.193%	80.080%	85.650%	50	1e-2	step_size=5
CyclicLR	91.198%	85.500%	90.440%	50	1e-2	base_lr=1e-2, max_lr=1e-1

Table 1: Best training and validation accuracies for each learning rate.

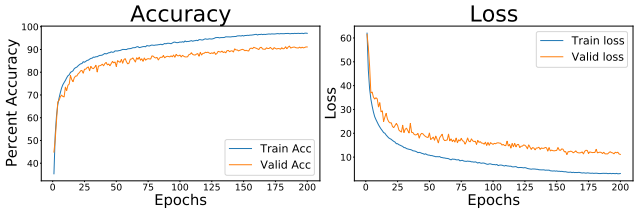

Figure 4: Training curves for model reduction using the **layer removal** method.

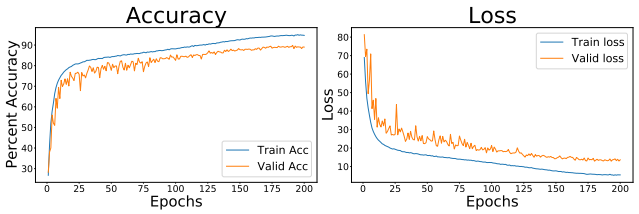

Figure 5: Training curves for model reduction using the **pruning** method.

0.001; criterion: CrossEntropyLoss; optimizer: SGD; momentum: 0.9; weight decay: 5e-4. After training the model for 118 epochs, we achieved 93 % training accuracy and 86 % validation accuracy.

Automated Hyperparameter Search Results In this project we used Optuna (Akiba et al. 2019) for automated hyperparameter search. Here we will describe the setup and results achieved using this framework. Our search space consisted of the following hyperparameters: optimizer, momentum, learning rate, batch size, and dropout rate, momentum. Optimizers: SGD, RMSprop, Adam, Adadelta, Adagrad, ASGD. For learning rate we looked at the space between 1e-5 to 1e-1. Finally we chose the batch size between 64 and 256.

We ran 30 independent trials, each one consisting of 25 epochs. Once Optuna identified the best model, we run it for 200 epochs. Optuna generates a report for us in which it describes which set of parameters yielded which accuracy as well as which hyperparameters were the most important to the model. The most important hyperparameters are momemtum and learning rate. The optimal learning rate=0.0008781984559717051, which we used for our pruning method run.

Adaptive Learning Rate Results Table 1 lists the training accuracy, validation accuracy, test accuracy and the hyperparameters we adapted in the experiments. While training accuracy of CyclicLR scheduler (91.198%) is higher than CosineAnnealngLR scheduler (90.976%), we see that the later one has higher validation accuracy and test accuracy. Therefore, we determined CosineAnnealingLR has best performance among the tested schedulers. We adapted CosineAnnealingLR as adpative learning rate scheduler for the rest of the models including one with layer removal, and one with pruning.

Conclusion

In this work we show experimental results for two different parameter reduction techniques: pruning and layer removal. Both of these methods produce different parameter size reductions with pruning reducing the original 11.2 million to 4.5 million parameters while layer removal produces 2.8 million. The difference in parameter reduction makes a big difference in training time with the layer removal being almost twice as fast to train per epoch. However, the results are very similar in terms of accuracy with a rate of 97% and validation accuracy of 91% for both methods.

References

Akiba, T.; Sano, S.; Yanase, T.; Ohta, T.; and Koyama, M. 2019. Optuna: A Next-generation Hyperparameter Optimization Framework. In *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Fang, G. 2022. Torch-Pruning.

Krizhevsky, A. 2009. Learning Multiple Layers of Features from Tiny Images. 32–33.

Lecun, Y.; Denker, J.; and Solla, S. 1989. Optimal Brain Damage. volume 2, 598–605.

Liaw, R.; Liang, E.; Nishihara, R.; Moritz, P.; Gonzalez, J. E.; and Stoica, I. 2018. Tune: A Research Platform for Distributed Model Selection and Training. *arXiv preprint arXiv:1807.05118*.

Liu, K. 2020. Pytorch-cifar.

Optuna. 2022. Samplesr. https://optuna.readthedocs.io/en/v2.0.0/reference/generated/optuna.samplers.TPESampler.html. Accessed: 2022-11-20.

Paperspace. 2020. Pruning Explained. https://blog.paperspace.com/neural-network-pruning-explained/. Accessed: 2022-11-20.

PyTorch. 2022. Pytorch Pruning. https://pytorch.org/tutorials/intermediate/pruning_tutorial.html. Accessed: 2022-11-20.