An Implementation of the ResNet Architecture in Pytorch

David Chang[1], **Jimmy Shong**[1]

[1] New York University

Repository

Code: https://github.com/davidchiii/deep-miniproject

Overview

This paper evaluates a Residual Neural Network (ResNet) architecture for object recognition tasks using the CIFAR-10 dataset.

A ResNet neural network was first proposed by (He et al. 2015), presented a new learning framework that improved training and recognition tasks. From their findings, Resnet has improvements over the classical convolution neural network architecture while still maintaining the relative components such as convolution, pooling, and activation layers.

A ResNet network comprises ”Residual Blocks” featuring skip connections that connect activation values to layers deeper into the network. This addresses the problem of a vanishing/exploding gradient seen often in more conventional deep neural networks like CNNs. For example, when layers use an activation like the sigmoid function, the small derivatives are multiplied together during backpropagation, resulting in an exponentially decreasing gradient.

Residual networks address this by adding the value directly through a ”skip connection” to the end of the block, bypassing the derivatives. In the paper introducing the Resnet architecture, the idea of ”identity mapping” has shown that it does not add extra parameters or computation complexity, proving it to be an excellent way to improve earlier features without abstraction.

The CIFAR-10 dataset is a widely used benchmark in machine learning and computer vision. It comprises 60,000 32x32 color images in 10 classes, with 6,000 images per class. Due to its relatively small size and simplicity compared to larger datasets like ImageNet, CIFAR-10 has been used to benchmark smaller models, making it suitable as a training and test set.

In this exploration, we start with an 18-layer baseline Resnet model. Then, we fine-tune the hyperparameters in hopes of improving the accuracy of vision recognition tasks. By the end of these experiments, we will have **4.6M** ResNet that can get a **3%** increase in accuracy over the reported ResNet-18 accuracy.

Methodology

Analysis of Original ResNet-18

We first looked at the original ResNet-18 (He et al. 2015) model to get started. Its model architecture consisted of 11M parameters, with four residual layers, two residual blocks per layer, a convolution kernel size of 3, a skip connection kernel size of 1, and an average pool kernel size of 4. The first, second, third, and fourth residual layers had 64, 128, 256, and 512 channel sizes, respectively. Researchers have reported that ResNet-18 achieves a test accuracy of 93.02% (Liu 2020). In their experiments, the researchers loaded the training data with a batch size of 128 and augmented the images via random horizontal flipping, cropping, and normalization. The training was done for 200 epochs using the SGD optimizer paired with a Cosine Annealing Learning Rate Scheduler.

Model Size

Our first goal was to find a way to reduce the parameter sizes below 5M while maintaining the model’s performance. One method we tested was reducing the input channel sizes per residual layer. We explored this avenue by testing a model with three residual layers, 13 residual blocks each, and 32, 64, and 128 channel sizes, respectively. This model had 4.9M parameters and achieved a test accuracy of 95.39%, which is a decent improvement. In this instance, however, the simplest method proved superior: reducing the number of residual layers and blocks. We created a model with three residual layers and residual block amounts of 4, 4, and 3. Its input channel sizes have not been changed from the original ResNet-18, but the fourth layer, which contributed heavily to model size, has been removed. This model achieved a 95.48% test accuracy with only 4.6M parameters. Other configurations of residual layers, blocks, and channel sizes did not perform well, so we used this simple 4.6M model as our baseline for hyperparameter tuning. We chose not to change the model architecture too much as many of these hyperparameters were co-dependent on each other, and we did not find an efficient way to tune them. This can be explored in future work.

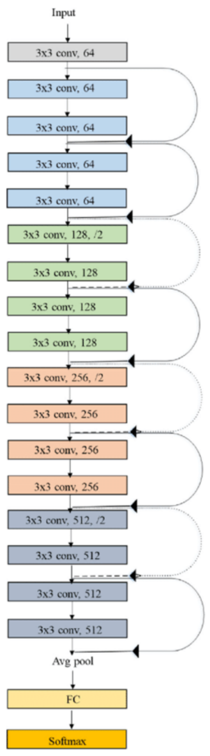

Figure 1: Model Architecture

Pooling

When experimenting with different model architectures, we found we had to change the average pool kernel size to get a valid output. We also found that different pool kernel sizes can give the same output size with different results. We initially experimented with multiple pool kernel sizes, and (8,1) achieved a slightly higher accuracy. We then found the concept of adaptive average pooling, which allowed us to specify the output size and let PyTorch pick the appropriate pool kernel size. We found that using this adaptive average pooling layer with an output size of (1,1) produced the best result, and thus, we used this in our final implementation.

Dropout

Overfitting is a highly prevalent problem in machine learning. When a neural network is adapted too well to a limited-size training set, it cannot generalize well and will perform poorly on held-out test data. While our baseline model did not seem to overfit, as it improves test accuracy at epoch 197, we still want to improve its generalization abilities through regularization. One of the most common ways to do this is Dropout (Srivastava et al. 2014). Dropout deletes a percentage p of the neurons in each training case and scales the output by a factor of $\frac{1}{1-p}$ to maintain the expected values of the model. This forces each neuron to create an activation that is generally helpful for producing the correct answer rather than an activation that is only helpful when combined with other neurons' activations. In short, dropout causes the model to avoid group representation, as the presence of other neurons becomes more unreliable.

We ran some experiments where we added a dropout layer right after the average pooling layer to the baseline model. We tuned the parameter for the dropout layer, which is the probability that any neuron gets zeroed out. We found that the probability of 0.4 worked the best, so we used it in our final model.

Augmentation

One of the most significant factors in any neural network's performance is the quality of the training data. Different data augmentations can improve a model's generalization ability by artificially enlarging the dataset. One prevalent data augmentation strategy is to do a random horizontal reflection paired with a random crop (Krizhevsky, Sutskever, and Hinton 2012). While training samples are still very interdependent, the training set size does increase significantly. Another data augmentation strategy is altering the intensity of the RGB channels, which makes it so the model is not affected by potential changes in the intensity and color of the illumination of the training images.

We tested nine different variations of this data augmentation. First, we tried two normalization values other than the baseline and found the best. Then, we put it through all 7 possible configurations between random cropping, horizontal flip, and normalization. We also did a trial with no data augmentation at all. Eventually, it was discovered that normalization does not positively affect the test accuracy for this dataset, so we just used random cropping and horizontal flips in the final model.

Training Hyperparameters

There are a lot of factors that need to be considered when training neural networks. Among them include the number of epochs, batch sizing, optimizers, and learning rate schedulers. To limit the number of experiments we conduct, we decided to tune each one by one and naively use the best options from each category. Starting with the number of epochs, we found these models tend to overfit shortly after the 200 epoch mark. Thus, we decided to keep the epoch size at 200. Different training batch sizes did not significantly impact our test accuracy, and the best-performing instance was the set initial size of 128.

We tested four different schedulers: cosine annealing scheduler, linear scheduler, polynomial scheduler, and one cycle scheduler. The baseline CosineAnnealingLR performed the best, which is surprising because PyTorch recommends OnecycleLR as state-of-the-art. We are unsure why this occurred, which will be investigated in future work. Furthermore, we would explore cosine annealing with warm restarts.

We also evaluated five optimizers in addition to the SGD optimizer used in the baseline. The optimizers included Adam, RMSProp, AdamW, AdaGrad, and Adadelta. We found none of these options came close to the set initially SGD. In hindsight, we believe we experienced these results because we did not use a warm-up in the scheduler. Including a warmup in the scheduler would prevent the gradient distribution from being distorted, leading to the optimizer being trapped in a wrong local minimum(Liu et al. 2021). In future work, we could look at RAdam, which supposedly does not have this issue, or implement a warmup in the scheduler. The RMSProp optimizer still hadn't converged within 200 epochs, so adding more epochs could have helped the accuracy at the expense of the speed. In any case, SGD with momentum and weight decay remained in the final model.

Final Model

The final model was constructed by taking the best option from each hyperparameter tuning experiment. It has the same model architecture as the baseline model, except it uses an adaptive average pooling layer and a dropout layer with $p = 4$. We also did not use color augmentation on the training and test set. It also has slightly fewer epochs than 200 since we saved it when its test accuracy was the highest. It produced the highest test accuracy, so we were pleased with the final model.

Results

Baseline and Final

This section goes over our findings. All testing is modeled off the baseline regarding other hyperparameters. This way, we can adjust each parameter individually to identify if minor improvements can be made. We established a baseline model using the following parameters and arrived at the final model via experimentation. (Table 1)

Parameter	**Baseline**	**Final**
Params	4.6M	4.6M
N	3	3
B_i	[4,4,3]	[4,4,3]
C	[64, 128, 256]	[64, 128, 256]
F	3	3
K	1	1
P	5	Adaptive(1,1)
Optimizer	SGD	SGD
Data Augments	All	Crop+Flip
Batch Size	128	128
Regularization	None	Dropout
Scheduler	Cosine Annealing	Cosine Annealing
Epochs	200	197
Acc.	**95.48**	**96.10**

Table 1: Baseline vs Final Model Hyperparameters

Scheduler

We tested four types of schedulers in our neural network. These have been implemented in the `torch.optim` package. (Table 2)

Our results showed that the Cosine Annealing scheduler has improved learning accuracy compared to other methods like Linear, Polynomial, and One-cycle schedulers. We believe this is because the Cosine Annealing scheduler allows the model's training to converge to the optimum faster(Loshchilov and Hutter 2016).

Scheduler	**Accuracy:**
Cosine Annealing (Baseline)	95.48
Linear	91.048
Polynomialr	78.413
One Cycle	91.29

Table 2: Scheduler Optimization

Data Augmentation

We tested 8 combinations of data augmentation techniques and found marginal improvements over no data augmentation techniques.

We found that the best data augmentation technique was the "crop-and-flip" method for testing accuracy. Close behind was the "random cropping" technique. (Table 3)

Data Augmentation Technique	**Accuracy:**
Baseline	95.48
None	90.71
RandomCropping	94.02
HorizontalFlip	93.39
Normalization(0.49)	89.36
Normalization(0.5)	89.15
Crop+Flip	**95.55**
Crop+Normalization	94.07
Flip+Normalization	93.51
All	95.3

Table 3: Data Augmentation

We believe data augmentation techniques help to prevent some of the overfitting trends often seen in deeper, large epoch models. Thus, introducing the subtle noise and randomness has improved our validation accuracy by 3-5%.

Optimizer

We tested five conventional optimizers from the Pytorch library. (Table 4)

Optimizer Algorithm	**Accuracy:**
SGD (baseline)	**95.48**
Adam	68.55
RMSProp	74.85
AdamW	93.06
AdaGrad	89.88
AdaDelta	94.3

Table 4: Optimizer

From testing, we found a significant deviation between each optimizer. The relative accuracy deviation between the worst scoring model using the Adam optimizer and the best AdaDelta optimizer on the validation set was 25.77%.

Epochs Trained

We identified roughly 200 epochs as our model's optimal training length. The following chart describes the epochs we tested. After 200 epochs, we identified a significant decrease per additional epoch. We believe anything above, anywhere in this range, is overfitting the Resnet model. (Table 5)

Epochs	**Accuracy:**
200 (baseline)	**95.48**
100	85.45
300	95.32
400	95.31

Table 5: Epochs Trained

Pooling

We tested using the `AdaptiveAvgPool2d()` package in PyTorch instead of the standard `avg_pool()` function. The following describes the results of this adjustment. (Table 6)

Pooling Function	**Accuracy:**
Average Pool (p = 5) (baseline)	95.48
Average Pool (p = (8,1))	95.57
Adaptive Average Pool (p = (1,1))	**95.61**

Table 6: Pooling Function Used

Dropout

We tested values from $0.2 - 0.5$ regarding the percentage of nodes dropping out and found the model performed insignificantly better using 0.4 as the parameter option. (Table 7)

Dropout Ratio	**Accuracy:**
0.0 (baseline)	95.48
0.2	95.37
0.3	95.29
0.4	**95.49**
0.5	95.43

Table 7: Dropout Optimization

Channel Size and Num. Residual Blocks

The following diagram describes the additional combinations of channel size and num. blocks we have tested in our model. From our testing, the model with 13-block layers performed best on the validation set but did not improve over the baseline. (Table 8)

Model Architecture	**Accuracy**
[64, 128, 256] (4,4,3) (baseline)	**95.48**
[32, 64, 128, 256] (4,4,3,3)	94.87
[16, 32, 64, 128, 256] (4,4,4,4,3)	93.41
[32,64,128] (13,13,13)	95.39

Table 8: Channel Size and Num. Residual Blocks

References

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In Pereira, F.; Burges, C.; Bottou, L.; and Weinberger, K., eds., *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc.

Liu, K. 2020. Train CIFAR10 with PyTorch. https://github.com/kuangliu/pytorch-cifar.

Liu, L.; Jiang, H.; He, P.; Chen, W.; Liu, X.; Gao, J.; and Han, J. 2021. On the Variance of the Adaptive Learning Rate and Beyond. arXiv:1908.03265.

Loshchilov, I.; and Hutter, F. 2016. SGDR: Stochastic Gradient Descent with Restarts. *CoRR*, abs/1608.03983.

Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I.; and Salakhutdinov, R. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(56): 1929–1958.