Custom ResNet Model Performance on the CIFAR-10 Dataset

Nishant Aswani, Alex Biehl, Oscar Shi

[1]New York University
Tandon School of Engineering
Brooklyn, New York 11221 USA
nsa325@nyu.edu, ab7289@nyu.edu, zs1616@nyu.edu

Abstract

We experimented with a wide range of custom ResNet architectures, training methods, and relevant hyperparameters and obtained a test accuracy of 91.5% with 4.997 million parameters on the CIFAR-10 dataset.

We found that best performances in architectures that were extremely deep and built using many residual blocks per layer. In contrast, increasing the number of channels did not always result in better performances. Changing kernel sizes did not result in better accuracy, either.

We also tried various way to improve training. Dropout was helpful as long as the dropout probability was kept low. Cyclic Learning Rate schedules helped add a few percentage points of accuracy to an already well-performing model, but only if the scheduler's parameters are carefully constructed, lest the model diverge.

Links to Supporting Material

Github Repository Here is all the code we used for this project, along with notebooks demonstrating our methodology.

Weights & Biases Results and Visualization The result of all of our experiments were pushed to Weights & Biases, by model and by epoch. The most noteworthy runs can be found here, which we reference in Table 1. The runs can be corroborated by the ID number.

Introduction

ResNet (He et al. 2016) is a popular convolutional neural network (CNN) for computer vision tasks.

A Residual Block The distinguishing feature of ResNet models is *residual mapping*, represented by the skip connections shown in Figure 1. Within each *residual block*, the output of stacked convolutional layers is added to the original input.

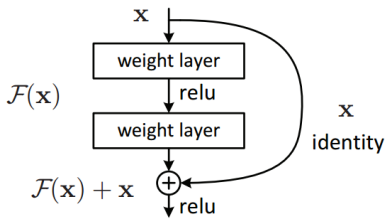

Figure 1: Residual block (He et al. 2016)

Methodology

Data Augmentation

Two forms of data augmentation were utilized. The first is a simple transformation pipeline including random rotations, random horizontal flips, and a random 32x32 crop with a padding of two, based on (He et al. 2016). The second form utilized Pytorch's $AutoAgument$ transform coupled with the CIFAR10 $AutoAugmentPolicy$. Given that $AutoAugment$ is significantly more complex (source code) than the first form, we were surprised to find only negligible difference between the two augmentation strategies during training.

Architecture

We employed a flexible version of the ResNet architecture to facilitate testing multiple configurations. Our implementation of the ResNet architecture borrowed heavily from both (Zhang et al. 2021) and (Liu et al. 2020). We made a wide variety of changes to model architecture in search of better performance. We adjusted the net stem that feeds into the residual layers. We tried changing the number of residual blocks (depth) and the number of channels (width). Within each residual blocks, we experimented with kernel size, bottlenecks, and dropouts.

A typical residual block (see Figure 1 consists of two consecutive convolutional layers, each with a kernel size n and output dimension C. The ResNet (He et al. 2016) authors introduced a bottleneck block with three convolutional layers, where the first convolutional layer serves to downsample C by a factor k, the second layer performs a standard convolution operation with kernel size

n, and the third convolutional layer upsamples back to the original input dimension C. As a result, we obtain residual blocks with fewer parameters, allowing for deeper networks.

Neural Net Stem Configuration As in many deep convolutional networks, ResNet's utilize a convolutional "stem":

- Kernel: 7×7, stride 3, padding 2

This convolutional layer is then followed by batch normalization, ReLU activation, and a Max Pooling layer, before feeding data into the first residual layer (He et al. 2016).

While this stem worked well for Imagenet, whose images are 224 by 224, in practice we found this default stem was inappropriate for CIFAR10, whose images are tiny in comparison at 32 by 32 pixels; CIFAR images left this default stem with only 16 by 16 tensor dimension retained. This, we believed, was excessive downsampling, throwing away far too much information too early.

So we modified the stem by removing the Max Pooling layer and reducing the convolution's kernel size, stride, and padding to either of the two following configurations.

- Kernel: 5 x 5, stride 2, padding 1
- Kernel: 3 x 3, stride 1, padding 1

Both of these configurations produced similar results in our testing. In all of our trials, one of the above convolutional layers was used, followed by a batch normalization layer and ReLU activation, but no Max Pooling layer.

Kernel Size Adjustment ResNet18 as presented in (He et al. 2016) contains 11.7 million parameters, far exceeding this project's five million constraint. Because ResNet18's parameters mostly reside in its convolutional layers, and because a convolutional layer' parameter count is in proportion to the square of its kernel size, a logical way to reduce ResNet18's parameters count would be to reduce its kernel size from 3×3 to 2×2.

But to use a 2×2 kernel, we had to first solve the issue of shape matching between a block's two addends: the output of its two convolutional layers and the shortcut. The original VGG's 3×3 kernel size combined with its 1-pixel padding preserved spatial resolution after convolution (Simonyan and Zisserman 2014). Implementing a 2×2 kernel size without changing the rest of a residual block would cause spatial resolution to grow by one pixel per convolution.

We devised two designs that offset this unwanted growth of spatial resolution. The first design is to change a block's second convolution to zero padding (Row 1 in Table 1 and notebook here):

- $conv1$: kernel 2, padding 1: resolution $+1$
- $conv2$: kernel 2, padding 0: resolution -1

A second design is to bring back resolution using average pooling layers (see Row 6 in Table 1 and the notebook here):

- $conv1$: kernel 2, padding 1: resolution $+1$
- $avgpool1$: kernel 2: resolution -1
- $conv2$: kernel 2, padding 1: resolution $+1$
- $avgpool2$: kernel 2: resolution -1

Both designs kept ResNet18's architecture otherwise unchanged, and they both had 5.069 million parameters. Despite their parameters count exceeding this project's requirement, they only achieved 87.45% and 85.89% test accuracy, respectively. In comparison, our other designs below achieved closer to 90% test accuracy while preserving the 3×3 kernel size.

We believe that the reason that 2×2 kernel sizes materially worsened model performance was because a model requires at least 3×3 kernel sizes to capture the notion of left-/right, up/down, and center (Simonyan and Zisserman 2014). In contrast, our 2×2 kernel sizes probably failed to extract and retain the notion of center.

Network Depth and Number of Channels Because CIFAR-10 has only 10 classes, we hypothesized that the last residual block could output a small number of channels (i.e. narrow), and those channels still contain sufficient information to distinguish the 10 classes.

This intuition to meet the five-million parameter constraint by narrowing the net was corroborated by an initial architecture in which we modified ResNet18: we significantly reduced the last residual layer's channels from 512 to 196, while increasing the channels in earlier blocks from 64 to 128, lowering the parameter count to just below 5 million, as well as adding more layers and thus more downsampling than ResNet18. This deep and narrow model did remarkably well, achieving an acceptable 86% test accuracy after just 100 epochs and a constant learning rate.

For the sake of completeness, we constructed and tested architectures that are wider, for example with 1024 channels in the last residual block, but shallower. We consistently observed that all else being equal, higher performance is achieved via narrower and deeper architectures. Model number 4 referenced in Table 1 consisted of 2 blocks of 128 channels, 6 blocks of 256 channels, 6 blocks of 256 channels, and finally 2 blocks with 1024 channels before the classification head. Despite the wide bottlenecks built into the model, it was only able to achieve a final test accuracy of 85.7% after 200 epochs.

A corollary of keeping architecture narrow is that depth should be achieved not through adding residual layers, but through adding more residual blocks per residual layer. This is because crossing into a subsequent residual layers causes channels to double or at least increase, which widens a net.

Basic vs Bottleneck Blocks As shown in Table 1, we experimented with a variety of depths in model architecture. We discovered that by replacing basic blocks with bottleneck blocks we could produce deeper networks, while remaining under the five million parameter constraint.

In hopes to push the general rule of thumb regarding deepening neural networks to its limit, we focused our attention on adding as many blocks as possible to our model. However, we discovered that building deeper networks did not result in a significant boost in model accuracy. Another work (Wu, Shen, and Van Den Hengel 2019) also discovered that

tested models "performed comparably", despite one of them having a "larger depth."

In our experiments, we found that the increased depth did not have a negative impact, so we proceeded with a deep network consisting of bottleneck blocks. Our final model consists of thirty nine blocks (five 6-channel basic, five 128-basic, twenty six 256-channel bottleneck, five 512-channel bottleneck) and obtained 91.46% accuracy, while previous attempts with fifteen blocks and twenty-six blocks (maintaining a similar total number of parameters) obtained 90.67% (Row 2 in Table 1) and 90.31% (Row 5 in Table 1) test accuracy, respectively. Similar to the work referenced above, we found that varying depth yielded models with comparable performance, but there was some benefit to deeper networks.

Training

We experimented with various ways to improve training such as adding dropout layers, trying different optimizers, running a learning rate range test, and adjusting batch size and the number of epochs.

Both Google Colab and AWS SageMaker Studio Lab were used, along with a Lambda workstation for longer runs.

Dropout In early experiments, we noticed that validation accuracy begun to significantly lag training accuracy after about 30 epochs. This indicated overfitting and called for regulariation, which we implemented through dropout layers. Specifically, we inserted one dropout layer between each block's $conv \rightarrow bn \rightarrow relu$ and its $conv \rightarrow bn$. Doing so successfully narrowed the gap between training and validation losses.

We also tried dropout probabilities either greater or smaller than the default $p = 0.5$, and found that probabilities near and above 0.5 negatively impacted model performance. We found that $p = 0.1$ led to the highest and most balanced training and validation performances.

Learning Rate and Optimizers Initially, most of our models were trained using an Adam optimizer with a 0.001 learning rate. The resulting test accuracy ranged from 85% to 90%. We also observed that after 100-200 epochs, validation loss was still declining and not yet bottomed. So we hypothesized that our gradient descent was too slow, and that higher performance could be achieved by improving our learning rate and optimizer.

To accelerate our gradient descent, we tried increasing learning rate from 0.001 to 0.01 and then to 0.1. But such arbitrary increases resulted in diverging behavior.

We then searched for optimizers beyond Adam, and found a modified Adam called AdamW proposed by (Loshchilov and Hutter 2017) that fixes an error that was found in the original Adam paper. . It improved speed of convergence: whereas most of our models under Adam saw their training losses starting to flatten at around 100 epochs, the same models under AdamW descented much faster and started to flatten at around 40-60 epochs. We found that (Loshchilov and Hutter 2017)'s claim that the updated AdamW optimizer performed much better than Adam on image classification was in fact correct.

Learning Rate Range Test The Learning Rate Range Test (LRRT) was first proposed by Leslie N. Smith in (Smith 2018). The LRRT consists of running several short tests (a few hundred iterations at most) on an untrained model with an increasing learning rate. Plotting the training loss and accuracy against the learning rate, the paper proposes that an optimal range can be found such that with a Cyclic Learning Rate schedule, better and faster convergence can be achieved than with a fixed learning rate. Linked here is an example of one LRRT we ran for a particular model. It suggests that we use a maximum learning rate around 0.30.

That LRRT, however, did not tell us what minimum learning rate to use. (Smith 2018) recommended a factor of 3 or 4 less than the maximum bound. But when we followed that recommendation, it destablized our training. In general, we found cyclic learning rate schedule not to be a panacea, even though it was used in our final best performing model: several models that we tried with Cyclic LR schedules failed wholly to converge, with some unable to break 10% validation accuracy after 60 epochs. See Row 7 and 8 on Table 1. Our takeaway was that while the LRRT and Cyclic Learning Rate schedules were sometimes helpful, they were a tool to be used with caution.

Batch Size The choice of batch size was largely constrained by the availability and memory bandwidth of the free GPU accelerators available to us on Google Colab and AWS Sagemaker StudioLab. Batch sizes that we tried were 128, 256, 512, and 1024. We occasionally ran into out of memory (OOM) errors when training with batch size of 512–even more so with 1024–as well as increased difficulty gaining convergence, so we ended up training the majority of our models with a batch size 256.

Epochs Similarly to batch size, the amount of time we could spend training any one model was constrained by the timeouts and time limits built into both Colab and Sagemaker. As such, the majority of our models were ran between 100 and 200 epochs. We often found that learning progressed quickly in the early stages of training, with some models achieving 80% validation accuracy after as few as 10 epochs, but that progress would slow down immensely between 50 and 80 epochs. A few models we ran for as many as 400 epochs, but found that this had little impact on the final performance of the model.

Results

Final Model

- Performance:
  - Test accuracy: 91.4550781%
- Architecture:
  - Stem: 1 convolution with 64 channels
  - Res. Layer 1: 3 Basic Res. Blocks, 64 channels
  - Res. Layer 2: 5 Basic Res. Blocks, 128 channels
  - Res. Layer 3: 26 Bottleneck Res. Blocks, 256 channels
  - Res. Layer 4: 5 Bottleneck Res. Blocks, 512 channels
  - A dropout layer with $p = 0.1$ is inserted between each block's $conv \rightarrow bn \rightarrow relu$ and its $conv \rightarrow bn$

  - Classifier: adaptive average pooling → flatten → fully connected layer with 10 output classes
- Number of parameters: 4,997,194

References

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

Liu, K.; Yang, W.; Peiwen, Y.; and Ducau, F. 2020. pytorch-cifar.

Loshchilov, I.; and Hutter, F. 2017. Decoupled weight decay regularization. *arXiv:1711.05101*.

Simonyan, K.; and Zisserman, A. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556*.

Smith, L. N. 2018. A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820*, 5510–026.

Wu, Z.; Shen, C.; and Van Den Hengel, A. 2019. Wider or deeper: Revisiting the resnet model for visual recognition. *Pattern Recognition*, 90: 119–133.

Zhang, A.; Lipton, Z. C.; Li, M.; and Smola, A. J. 2021. Dive into Deep Learning. *arXiv preprint arXiv:2106.11342*.

Appendix

Table 1: Selected runs referenced in this report. Each residual layer is formatted as [Block Type, Number of Blocks, Number of Channels, (Optional Dropout), (Optional Kernel Size)]

ID	Architecture	Final Test Accuracy	Final Train Accuracy
1	[[Basic,2,64,0.1,2],[Basic,2,128,0.1,2],[Basic,2,256,0.1,2],[Basic,2,512,0.1,2]]	0.8745	0.9566
2	[[Basic,2,128,0.1],[Basic,2,256,0.1],[Bottleneck,5,256,0.1],[Bottleneck,6,512,0.1]]	0.9067	0.9825
3	[[Basic,3,64,0.1],[Basic,5,128,0.1],[Bottleneck,26,256,0.1],[Bottleneck,5,512,0.1]]	0.9146	0.9821
4	[[Basic,1,128,0.5],[Basic,2,128,0.5],[Basic,2,128,0.5],[Basic,2,128,0.5],[Basic,2,196,0.5],[Basic,2,196,0.5]]	0.8842	0.9158
5	[[Basic,3,64,0.1],[Basic,5,128,0.1],[Bottleneck,9,256,0.1],[Bottleneck,9,512,0.1]]	0.9031	0.9806
6	[[Basic,2,64,0.1,2],[Basic,2,128,0.1,2],[Basic,2,256,0.1,2],[Basic,2,512,0.1,2]]	0.8589	0.9386
7	[[Bottleneck,2,128,0.1],[Bottleneck,6,256,0.1],[Bottleneck,6,512,0.1],[Bottleneck,2,1024,0.1]]	N/A	Very Low
8	[[Bottleneck,2,128,null],[Bottleneck,6,256,null],[Bottleneck,6,512,null],[Bottleneck,2,1024,null]]	0.857	0.9513
9	N/A	0.8986	0.9666