NYU Deep Learning Mini-Project — Training A ResNet For CIFAR10 Image Classification

Written by Marc Chiu, Sathish Vijay, Hafez ElBoghdadi

mmc9967@nyu.edu, svk319@nyu.edu, ae2405@nyu.edu,

Abstract

In this mini-project we were tasked with coming up with a modified residual network (ResNet) architecture with the highest test accuracy on the CIFAR- 10 image classification dataset, under the constraint that the model has no more than 5 million parameters. We modified the ResNet-18 architecture by reducing the number of channels to create an architecture with fewer than 5 million parameters and preformed data augmentation and normalization to improve test accuracy.

Loading and Transforming CIFAR-10 Data

The CIFAR-10 data set consists of 60,000 32x32 colored images in ten classes, with 6,000 images per class. The data is comprised of 50,000 training images and 10,000 testing images. To make our model more robust we preformed data augmentation by rotating every image by five degrees in a random direction, randomly cropping each image, and randomly selecting images with a probability of 0.5 to be flipped horizontally. Through data augmentation with are able to add more variety to our training data making it more robust and preform better on data with more noise. To improve performance when training and optimizing we normalize all the images and create train, test, and validation data loaders.

ResNet Architecture

A Residual Network is a specific type of Convolutions Neural Net that includes skip connections. We modified the ResNet-18 architecture by decreasing the number of channels in each convolution after the first block. To preform classification we added a linear layer and a softmax activation after all the residual blocks. By nearly halving the number of channels we were able to create an architecture with 4,999,718 parameters.

Methodology

We have followed the below steps

- The number of layers is inversely proportional to loss generally, but too many layers cause vanishing/exploding gradient problem[3].
- Skip connections in residual networks helps resolve the above problem[3]. When the number of layers are large, Resnets are beneficial else it may be detrimental.
- Since our problem statement is constrained to use Resnet architecture, we decided not to reduce the layers, instead we reduced the number of channels/ output units of each hidden layer so that params are less than 5M. If we reduce the layers, the skip connections may retard our learning to certain extent in resnet18 architecture.
- Our logic was to increase the output units(or output channels) in conv layers in each residual layers progressively so as to increase the features extracted. So we modified them accordingly(32,96,156,350) for the four residual layers[4].
- Neural Networks (however deep) acts as an approximation of an ideal function. The more layers/nodes are available, the more the Neural Network successfully approximate that ideal function[Universal approximation algorithm].
- The rest we followed standard practices in representation, loss function and optimization stages of our model pertaining to image classification models.

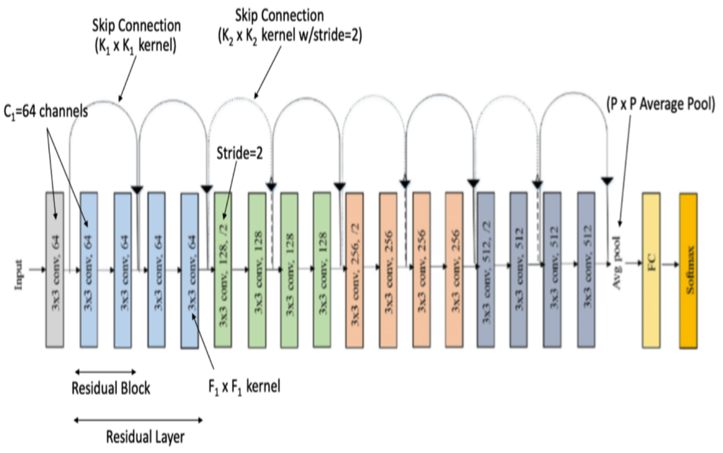

Figure 1: ResNet-18

Initializing Parameters

Once the model had been created the next step was to initialize all the parameters. The parameters were initialized using two different normal distributions. To initialize the weights of the linear layer we used a normal distribution with a mean of 0 and a standard deviation of 1e-3. To initialize the weights of the convolutional layers we used Kaiming initialization. We chose to initialize the convolutional layers using Kaiming initialization because it uses a scaling factor which aids in mitigating the exploding/vanishing gradient problem for networks that use the ReLu activation. The Kaiming initialization changes the normal distributions we are sampling from by changing the standard deviation at each layer, meaning layers further down the network will have slightly incremented means, thus making our model more robust. We set our hyperparameters (N, B, F, K) as N=4, B=2, F=[3,3], and K=F in all layers. We also set our stride as Strides=[[1,1] , [1,1] , [2,2] , [2,2], [2,2]], our learning rate as 1e-3, and our batch size as 256.

$$\text{std} = \sqrt{\frac{2}{(1 + a^2) \times \text{fan\_in}}}$$

Figure 2: Kaiming Distriubtion

Training and Testing

The model was initially trained using a train-validation split of 90-10 percent over 25 epochs using the ADAM optimizer and the cross entropy loss function. With only 25 epoch the test accuracy was approximately 88 percent so we decided to increase the number of epochs to 80. Each epoch took between 45-60 second on Google Colab GPUs. The best preforming model from training was saved and used to evaluate the test data. Using 80 epochs a training accuracy of 99 percent and validation accuracy of 90.2 percent was achieved. However, this model was still only able to achieve a 88.9 percent accuracy on the test data.

	Train Loss	Train Acc	Validation Loss	Validation Acc
Epoch 1	1.55	40.1%	1.490	47.8%
Epoch 10	0.375	87.1%	0.500	83.8%
Epoch 20	0.214	92.4%	0.493	85.5%
Epoch 30	0.137	95.0%	0.419	88.3%
Epoch 40	0.097	96.5%	0.481	88.6%
Epoch 50	0.068	97.6%	0.500	88.9%
Epoch 60	0.051	98.2%	0.446	90.2%
Epoch 70	0.040	98.6%	0.528	88.9%
Epoch 80	0.039	98.6%	0.460	90.2%

	Loss	Acc
Test Results	0.37	88.9%

Table 1: Results

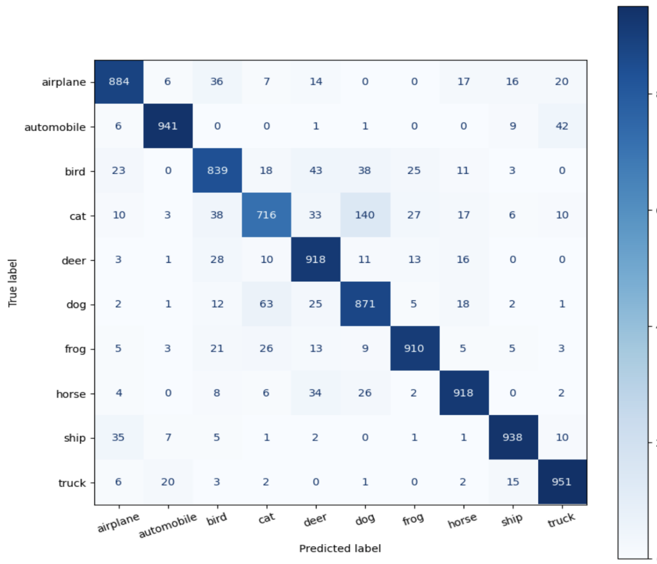

Figure 3: Confusion Matrix

Improvements

Our model was getting close to 90 percent but it was not breaking the threshold. Since the goal was to get greater than 90 percent we looked for ways to boost our accuracy. One way we though to boost our accuracy was to change the validation and train data so that we could incorporate all 50,000 images into training. Similar to cross validation we split our data in folds so that we could validate on one fold while training on the rest. By splitting the data into 5 folds, each fold trained with 10 epochs, we were able to incorporate another 5000 images into training and that was enough to improve our test accuracy to 90.47 percent. We believe this is a reasonable improvement because increasing our training data set by 10 percent is a small increase but not negligible, which results in a small increase in accuracy. Another potential improvement that we considered but did not implement is altering the way the current data augmentation in preformed. Currently, the data is being edited instead of being copied, altered, and concatenated to the original images. We believe that by adding to the original data set we could see an improvement in accuracy simply because there is more training data available. Although the improvement would be small because we would not be introducing data with new features.

	Train Loss	Train Acc	Validation Loss	Validation Acc
Fold 1	0.423	85.1%	0.550	80.8%
Fold 2	0.239	91.5%	0.394	87.2%
Fold 3	0.166	94.1%	0.276	91.1%
Fold 4	0.111	95.9%	0.271	91.4%
Fold 5	0.084	97.1%	0.147	95.0%

	Loss	Acc
Test Results	0.37	90.47%

Table 1: Results After Improvements

Observations

The ADAM optimizer converged faster than SGD. Training on all 50,000 data images compared to 45,000 was able to improve accuracy. We tried using learning rate scheduler but later reverted back owing to simplicity. Also, we tried increasing learning rate but data began to overfit. We did not use gridsearch, Raytune [5]etc. because we wanted to tune the hyperparameters manually. After experimenting with various hyperparameter spaces we resorted to the above mentioned methodology.

Github Repository

https://github.com/Svk319/Resnet18-CIFAR10.git
We have used some part of code from [6].

References

[1] https://towardsdatascience.com/understand-kaiming-initialization-and-implementation-detail-in-pytorch-f7aa967e9138.
[2] https://towardsdatascience.com/understanding-and-visualizing-resnets-442284831be8
[3] Residual Networks (ResNet) - Deep Learning - GeeksforGeeks
[4] https://dspace.cuni.cz/bitstream/handle/20.500.11956/127372/120390840.pdf?sequence=1isAllowed=ysubsection.3.1.2
[5] https://pytorch.org/tutorials/beginner/hyperparametertuningtutorial.html
[6] https://github.com/kuangliu/pytorch-cifar