Deep Learning Mini Project

Team members: Jesse Inouye, Shashvat Shah, Flavjo Xhelollari

[1]New York University Tandon School Of Engineering
{*jai9962, sss9772, fx2078*}*@nyu.edu*

Abstract

In this project we modify the ResNet architecture, in order to achieve high accuracy on CIFAR-10 dataset for image classification. The model has 4.6M parameters and reaches a test accuracy of 91.41%. The link to the code is https://github.com/jesseinouye/ECE-GY-6953-Mini-Project

Overview

In this project, our primary objective was to modify the ResNet architecture (He et al. 2015) to achieve the highest possible accuracy on the CIFAR-10 dataset, while adhering to the given parameter constraints. Our challenge was to maintain a parameter limit of 5 million while achieving a minimum accuracy benchmark of 80%. To accomplish this goal, we employed a carefully crafted approach that involved a thorough review of the ResNet architecture and a series of trial-and-error experiments with various implementations of the model.

After several iterations of experimentation and analysis, we finally succeeded in achieving an impressive accuracy rate of 91% while keeping the parameter count within the 5M limit. Our approach involved a meticulous study of the ResNet architecture, identifying potential bottlenecks, and implementing novel techniques to overcome them. We have utilized a combination of these two techniques: learning rate scheduling (Brownlee 2019) and early stopping.

Methodology

First things first, the CIFAR-10 dataset is a benchmarking image classification dataset that consists of 50'000 training and 10'000 testing images, each belonging in one of the 10 possible classes. It is also important to mention how the ResNet Architecture works, and how it is applicable to the CIFAR-10 optimization task. ResNet stands for Residual Network, and it is an ubiquitous Deep Learning Network Architecture, specifically used in image classification tasks. What is interesting about ResNet, is that it uses residual blocks (one might say connections), so that the information is passed directly to an arbitrary network layer, instead of being passed forward, as you can see in traditional Neural Network Architectures. This approach helps overcome the vanishing gradient problem, making training on deep networks more feasible, thus bearing better results in tasks such as image classification.

Taking the constraint that we have mentioned above into account, one might be able to approach this problem in different ways. To reduce the number of parameters for the model, it might come as natural to reduce the size of the input, i.e. to reduce the quality of the images we feed into the network. We approached this problem in two different methods: one by building a number of models varying in architecture with differences in layout, kernel sizes, strides, and channel dimensions; two by adjusting the training method and hyperparameters.

In the first method, we created models with total trainable parameters ranging from roughly 2.5 million to 4.9 million, attaining various levels of accuracy. The models with fewer parameters contained four layers, with channel sizes increasing from 3-64-128-256. These smaller models consisted of 2.6 million parameters and expectedly performed worse, achieving accuracy scores on the test dataset around 79%, just short of the 80% goal. Maintaining the four layers, but adjusting the channel sizes to 64-128-256-512 achieved much higher test accuracy scores around 89%, but drastically increased the number of parameters to 4.9 million. This accuracy increase is likely attributed to the overall larger model, with more channels able to capture finer details in the training images.

Shifting focus, we decided to adjust the convolutional kernel sizes in an attempt to better capture more global information in the training images at higher levels. In our implementation, every residual block uses the same convolutional kernel, stride, and padding parameters for the two convolution sub-layers within - thus when adjusting the parameters for either the first or second convolution sub-layer, it affects every residual block in every residual layer. With this design, we decided to increase the size of the kernel in the first convolution sub-layer from 3x3 to 5x5, adjusting the padding accordingly from 1 to 2. In the previous iteration with four layers and channel sizes 64-128-256-512, this larger kernel pushed the number of trainable parameters above 5 million, so we adjusted the last layer down to 256 channels from 512, giving us roughly 4.2 million parameters. This new model increased test accuracy slightly to around 90%.

After a few small additional adjustments that provided little improvement (or decreased accuracy), we decided to add an additional residual layer and modify the channel sizes to fit within the 5 million parameter budget. We ended with a five-layer model with channel sizes 64-64-128-258-258, containing roughly 4.6 million parameters, as seen in Figure 1 below. This model marginally increased the test accuracy to 91.4%, the highest accuracy we were able to achieve.

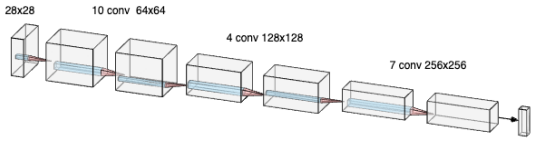

Figure 1: Model outline

In parallel with the testing above, we pursued the second aforementioned method - adjusting the training hyperparameters, adding early stopping, and trying ensemble learning. For the latter, we trained a few models and combined their predictions so that the accuracy further improved. After averaging these outcomes, the early models were able to achieve a high accuracy of 85%. Furthermore, we have successfully implemented early-stopping and hyperparameter tuning via a rate scheduler, while adhering to the prescribed parameters constraint. With these adjustments and the aforementioned models, we managed to achieve a 91% accuracy. For a more detailed view, refer to Table 1.

Early stopping is a very good technique when it comes to training deep learning models. It is very useful in preventing overfitting (Rice, Wong, and Kolter 2020) and improves the generalization capabilities of the model. When training the model, we monitor the performance on the validation set at regular intervals and stop training if the performance starts to deteriorate. This helps avoid overfitting and improves performance on unseen data. This technique also helped us train for an optimal number of epochs, yielding the highest possible performance, whilst complying to the constraint. We have also incorporated a learning rate scheduler when training the model. This is a technique used in deep learning models that adjusts the learning rate in the training process. It involves adjusting (typically reducing) the learning rate following a specific type of scheduler. In our final implementation, we used a cosine annealing scheduler with warm restart (CosineAnnealingWarmRestarts). This type of scheduler reduces the learning rate by following a cosine curve from the initial rate down to a specified minimum over a given number of epochs, after which it "restarts" the learning rate back to the initial rate and repeats the reducing process. We found this worked best with a starting learning rate of 0.01, a minimum rate of 0, restarting after 100 epochs - though we occasionally never reached 100 epochs as the "early stopping" mechanism usually kicked in after 80-90 epochs.

We managed to combine the use of these aforementioned techniques in solving the original problem. In a bird-eye view, we adjust the learning rate during training in order to help the model converge quicker, and we stop the training when the early-stopping dictates that the performance of the model stops improving, hence we avoid overfitting. By using this combination, we managed to achieve a better performance while adhering to the 5 million parameter limit. This way, the model converges relatively faster, overfitting is avoided, and we don't waste further computational resources, as we avoid training beyond what we aim at.

Results

Our final model architecture is composed of five residual layers, with channel sizes 64, 64, 128, 256, and 256 respectively. Each residual layer has two residual blocks, each with two convolutional sub-layers with kernel sizes of 5x5 and 3x3, and padding of 2 and 1 respectively. The stride of each convolution is determined by the layer, with the first two layers using a stride of 1, and the last three layers using a stride of 2. The smaller stride in the earlier layers helps maintain more local information while the larger strides in later layers helps capture more global information. A summary with the entire structure can be seen in the Jupyter Notebook containing the code, but it generally follows a sequence of: convolution, batch norm, convolution, batch norm, ReLU; repeating for each residual block.

This model contains around 4.6 million parameters and reaches a test accuracy score of 91.4%. The training and validation accuracies reach 98% and 97% respectively, as seen in Figure 2 below. The training and validation losses reach 0.053 and 0.057 respectively, as seen in Figure 3 below.

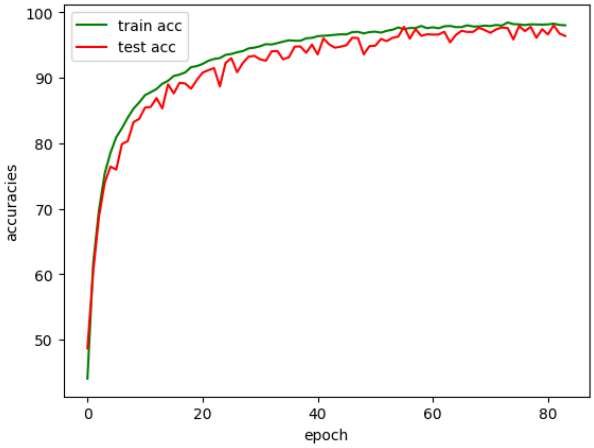

Figure 2: Training and validation accuracies over epochs

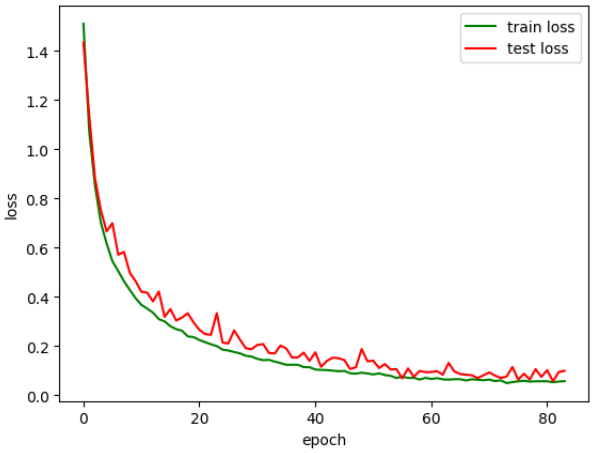

Figure 3: Training and validation losses over epochs

References

Brownlee, J. 2019. Learning Rate Schedules and Adaptive Learning Rate Methods for Deep Learning. *Machine Learning Mastery*, 3.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2015. Deep Residual Learning for Image Recognition. *CoRR*, abs/1512.03385.

Rice, L.; Wong, E.; and Kolter, J. Z. 2020. Overfitting in adversarially robust deep learning. arXiv:2002.11569.

Table 1: Appendix - Model Architecture Overview

Layer (type)	Output Shape	Param #	
Conv2d-1	[-1, 64, 28, 28]	1,728	
BatchNorm2d-2	[-1, 64, 28, 28]	128	
Conv2d-3	[-1, 64, 28, 28]	102,400	
BatchNorm2d-4	[-1, 64, 28, 28]	128	
Conv2d-5	[-1, 64, 28, 28]	36,864	
BatchNorm2d-6	[-1, 64, 28, 28]	128	
ResnetBlock-7	[-1, 64, 28, 28]	0	
Conv2d-8	[-1, 64, 28, 28]	102,400	
BatchNorm2d-9	[-1, 64, 28, 28]	128	
Conv2d-10	[-1, 64, 28, 28]	36,864	
BatchNorm2d-11	[-1, 64, 28, 28]	128	
ResnetBlock-12	[-1, 64, 28, 28]	0	
Conv2d-13	[-1, 64, 28, 28]	102,400	
BatchNorm2d-14	[-1, 64, 28, 28]	128	
Conv2d-15	[-1, 64, 28, 28]	36,864	
BatchNorm2d-16	[-1, 64, 28, 28]	128	
ResnetBlock-17	[-1, 64, 28, 28]	0	
Conv2d-18	[-1, 64, 28, 28]	102,400	
BatchNorm2d-19	[-1, 64, 28, 28]	128	
Conv2d-20	[-1, 64, 28, 28]	36,864	
BatchNorm2d-21	[-1, 64, 28, 28]	128	
ResnetBlock-22	[-1, 64, 28, 28]	0	
Conv2d-23	[-1, 128, 14, 14]	204,800	
BatchNorm2d-24	[-1, 128, 14, 14]	256	
Conv2d-25	[-1, 128, 14, 14]	147,456	
BatchNorm2d-26	[-1, 128, 14, 14]	256	
Conv2d-27	[-1, 128, 14, 14]	8,192	
BatchNorm2d-28	[-1, 128, 14, 14]	256	
ResnetBlock-29	[-1, 128, 14, 14]	0	
Conv2d-30	[-1, 256, 7, 7]	819,200	
BatchNorm2d-31	[-1, 256, 7, 7]	512	
Conv2d-32	[-1, 256, 7, 7]	589,824	
BatchNorm2d-33	[-1, 256, 7, 7]	512	
Conv2d-34	[-1, 256, 7, 7]	32,768	
BatchNorm2d-35	[-1, 256, 7, 7]	512	
ResnetBlock-36	[-1, 256, 7, 7]	0	
Conv2d-37	[-1, 256, 4, 4]	1,638,400	
BatchNorm2d-38	[-1, 256, 4, 4]	512	
Conv2d-39	[-1, 256, 4, 4]	589,824	
BatchNorm2d-40	[-1, 256, 4, 4]	512	
Conv2d-41	[-1, 256, 4, 4]	65,536	
BatchNorm2d-42	[-1, 256, 4, 4]	512	
ResnetBlock-43	[-1, 256, 4, 4]	0	
Linear-44	[-1, 10]	2,570	
Total params	**4,662,346**		
Trainable params	4,662,346		
Non-trainable params	0		