Deep Learning Mini Project

Dharu Piraba Muguntharaman, Raunak Bhupal, Shamyukta Rajagopal

New York University
Github Repository: https://github.com/raunakbhupal/resnet-cifar10

Abstract

Our project involved designing a modified ResNet architecture for classifying images in the CIFAR-10 dataset. We imposed a constraint on the model to have no more than 5 million parameters. By training various models on this dataset, we achieved a test set accuracy of **93%**. To enhance the model's performance, we utilized hyperparameter tuning and optimization techniques.

Introduction

Deep Neural Networks perform well on image classification tasks despite being difficult to train. Compared to other neural networks, deep residual networks (ResNets) can enhance accuracy and accelerate the training process. Residual blocks serve as the foundation of the ResNet model, where each layer feeds into the next layer and directly into the layers that are 2-3 hops away, known as skip links or connections[1].

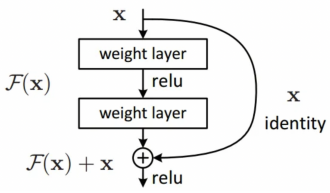

Figure 1: Single residual block

The output of the model after using the skip connections is:

$$H(x) = R(x) + x$$

Our project involved utilizing the ResNet architecture to classify images in the CIFAR-10 dataset. The ResNet model comprises a convolutional layer, a series of residual layers composed of Residual blocks, an average pooling layer, and a fully connected layer. The building blocks of the ResNet model [2] consists of 2 convolutional layers with a skip connection. We conducted experiments with different blocks and layers and applied various optimization techniques to obtain optimal results while adhering to a constraint of no more than 5 million parameters.

Methodology

Idea Behind Our Approach

The basic idea for this project is taken from Deep Residual Learning from Image Recognition Paper [2]. The architecture in this paper consists of 11.17 million parameters with channel sizes as 64,128,256,512.

Initially, we loaded the CIFAR-10 dataset and divided it into training and testing sets. To improve the performance of the model, we applied data augmentation techniques such as random cropping and horizontal flipping to the training data. Then, we implemented the ResNet architecture described in [2][3] which includes a BasicBlock module, the building block for the ResNet architecture. The BasicBlock contains two convolutional layers followed by batch normalization layers, with a skip connection that adds the input to the output of the second convolutional layer. The ResNet module consists of four layers, each containing a certain number of BasicBlocks. The first layer starts with a convolutional layer, followed by batch normalization and the first set of BasicBlocks. The remaining layers follow the same pattern, with increasing numbers of filters in each layer. Finally, the output is passed through an average pooling layer, flattened, and connected to a fully connected layer with softmax activation for classification. To improve the model's accuracy, we experimented with different hyperparameters, including optimization methods and schedulers. Additionally, we adjusted the number of BasicBlocks and filters for each layer to obtain better results.

Data Augmentation

We used two different augmentation techniques i.e. Random Crop and Random Horizontal Flip as part of data augmentation [4]

Random Crop : This technique involves creating a smaller, random subset of an original image, which can improve the generalization of a model. It returns the cropped

image at a random location of given size.

Random Horizontal Flip : This technique horizontally flips the given image randomly with a given probability. The idea behind this technique is that an object should be recognizable in the same way even if it appears as its mirror image.

Optimizers Used

An optimizer is a function or algorithm that can modify the weights and learning rates of a neural network, among other features. The main objective of the optimizer is to modify the model's parameters in such a way that the loss function is minimized. In order to enhance the performance of the model, we experimented with different optimizers. The optimizers which we used are :

Stochastic Gradient Descent (SGD) : It is a variation of the gradient descent algorithm. In SGD, rather than processing the entire dataset in each iteration, we randomly choose batches of data and then shuffle the data randomly at each iteration. By doing this, we can reach an approximate minimum of the loss function based on the values of hyperparameters such as the learning rate and momentum [5].

AdaDelta : AdaDelta is an enhanced version of the AdaGrad optimizer, which is designed to address the limitations of AdaGrad and RMSprop optimizers. AdaDelta uses adaptive learning and overcomes the challenge of decaying learning rates, which requires manual setup in other optimizers. AdaDelta keeps track of two state variables, a leaky average of the second moment gradient, and a leaky average of the second moment of the parameter changes in the model [5].

Adam : Adam is a modified version of Stochastic Gradient Descent, which updates the learning rate separately for each weight in the neural network. Adam combines the advantages of Adagrad and RMSprop optimizers by utilizing the second moment of gradients and computing the uncentered variance mean using these moments. Unlike stochastic gradient descent, Adam is designed to prioritize faster computation, as it focuses less on individual data points [5]. Due to its emphasis on faster computation and less on individual data points, the Adam optimizer may not perform as well as the SGD optimizer in certain cases, as we will see in the subsequent discussion of our model's performance.

The base model was taken with 4 ResNet layers and 3 blocks on the first three layers and 2 blocks on the last layer. This resulted in 4,049,994 parameters sticking to the parameters constraint mentioned. Initially, we used the Adam optimizer, which resulted in an accuracy of 86%. We further tested the model using AdaDelta and Stochastic Gradient Descent optimizers to determine if accuracy could be improved. Although AdaDelta improved the training accuracy, the test accuracy remained similar to that of Adam. In comparison, we improved our accuracy using SGD, achieving a 92% test accuracy. The requirement for significant hyperparameter tuning in Adam and Adadelta may have contributed to their poorer results compared to SGD. Overall, the results of our experiments show that a neural network's performance may be significantly impacted by the optimizer that is used.

To train our base model, we began with an initial learning rate of 0.01 and implemented a learning rate scheduler to decrease the learning rate as the number of epochs increases. Specifically, we used the CosineAnnealingLR scheduler [6], which reduces the learning rate by a cosine function. This approach helps the model to converge more effectively by adjusting the learning rate according to the training progress, which can prevent the optimizer from overshooting the optimal solution.

Varying Channel size

Channel sizes refer to the number of feature maps in a convolutional layer of a neural network. Adjusting the channel size can impact the network's performance significantly. Decreasing the channel size can reduce computational cost and memory usage. The four residual blocks used by authors in [2] have channel sizes of 64, 128, 256, and 512, yielding a total of 11.17 million parameters.

We modified the channel sizes to 64, 128, 128 and 256, correspondingly, to reduce the parameter count. We tried altering the block sizes and optimizer to improve accuracy. We attained an accuracy of 86% with 3,953,834 parameters while using the Adam optimizer with block sizes of 3, 2, 2, and 3 with channel sizes of 16, 64, 128, and 256. However, even with hyperparameter tuning, accuracy could not be increased any more. It's likely that the model's capacity to learn and generalize from the training data has been achieved. By using SGD optimizer with block sizes of 3, 3, 3, and 2, we reduced the parameter count to 3,142,570 while achieving an accuracy of 91%. Further, with block sizes of 2, 2, 2, and 2, we reduced the parameters to 2,768,490 while maintaining the same accuracy.

Shallow Network

The computational expense and memory requirements of the model can be decreased by reducing residual layers in a ResNet architecture. By utilizing skip connections, which make it easier for gradients to move through the network, the ResNet architecture is made to address the disappearing gradient problem. But as the network's depth rises, so do the number of parameters and the cost of computation, which can make it challenging to train the model effectively. By simplifying the model, lowering the number of learnable parameters, and enhancing the model's adaptability to new data, decreasing the number of residual layers in a ResNet can help with these problems.

Since we had the constraint of no more than 5 million parameters, we tried reducing the number of layers to 3. The last residual layer with channel size 512 was removed

resulting in a 3 layer model with channel sizes of 64,128 and 256. We used this model with block sizes of 3 for the first layer and 2 for the remaining 2 layers. This configuration resulted in 2,851,658 parameters and an accuracy of 86% when trained with the Adam optmiizer.

We tested the same model with the Stochastic Gradient Descent (SGD) optimizer, and were able to improve the accuracy to 92%. We also tried a similar model with a different block size of 3 for each of the 3 layers, resulting in 4,327,754 parameters. With this model and optimizer, we achieved an accuracy of **93%**.

Main Results

Model Architecture	Parameters	Accuracy
Block Size : 3, 3, 3 Channel Size : 64, 128, 256	4,327,754	93%
Block Size : 3, 2, 2 Channel Size : 64, 128, 256	2,851,658	92%
Block Size : 3, 3, 3, 2 Channel Size : 64, 128, 128, 256	4,049,994	92%

Results Discussion

In our study, we experimented with different hyperparameters for the ResNet architecture to classify images in the CIFAR-10 dataset. Our best performing model, available as "final_model.ipynb" on our github repository, used SGD optimizer and trained the dataset for 100 epochs with a CosineAnnealingLR scheduler, achieving **93%** accuracy on the test data. The model includes 3 convolutional layers with 3 residual blocks each and channel sizes of 64, 128, and 256. Additionally, we were successful in reducing the parameter count to less than 5 million, specifically 4,327,754. The loss and accuracy curves for the model is displayed in Figure 2.

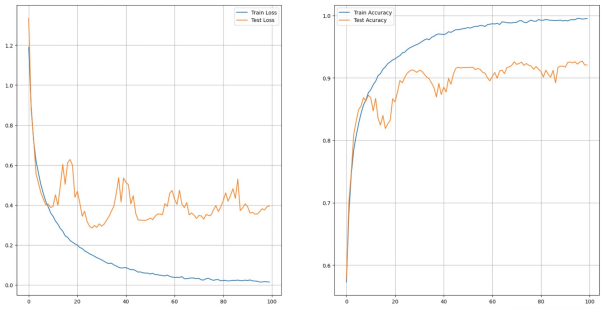

Figure 2: Results of Final Model

Conclusion

We performed different experiments on the CIFAR-10 dataset to identify a modified ResNet architecture to improve the performance. We achieved an accuracy of **93%** with 3 residual layers using SGD optimizer. Overall, Our experiments showed that modifying the model architecture and optimizer can significantly improve accuracy within the given parameter constraints.Therefore, this project highlights the importance of optimizing model architecture and hyperparameters to improve performance in deep learning applications.

References

[1] Sahoo, Sabyasachi. "Residual Blocks Building Blocks of ResNet." 7 Nov. 2018,https://towardsdatascience.com/residual-blocks-building-blocks-of-resnet-fd90ca15d6ec.

[2] He, Kaiming, et al. "Deep Residual Learning for Image Recognition." 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, https://doi.org/10.1109/cvpr.2016.90.

[3] Kuangliu. "Kuangliu/Pytorch-CIFAR." GitHub, https://github.com/kuangliu/pytorch-cifar.

[4] Nouman. "Writing Resnet from Scratch in Pytorch." Paperspace Blog, Paperspace Blog, 21 June 2022, https://blog.paperspace.com/writing-resnet-from-scratch-in-pytorch/.

[5] Gupta, Ayush. "A Comprehensive Guide on Optimizers in Deep Learning." Analytics Vidhya, 3 Mar. 2023, https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-on-deep-learning-optimizers/.

[6] Monigatti, Leonie. "A Visual Guide to Learning Rate Schedulers in Pytorch." Medium, Towards Data Science, 6 Dec. 2022, https://towardsdatascience.com/a-visual-guide-to-learning-rate-schedulers-in-pytorch-24bbb262c863#fad1.