Deep Learning: Mini Project Report

Chenxi Liu, Jalil Douglas, Dailin Ji

New York University, Tandon School of Engineering
https://github.com/TigaJi/DL-mini-project

Overview

In this project, our goal is to develop a lightweight version of ResNet (Zhang, Ren and Sun 2015) for image classification tasks. We achieved this by modifying the residual block within ResNet, resulting in a significant reduction in the number of parameters without sacrificing much of the model's capability. Additionally, we employed various preprocessing techniques, such as normalization and data augmentation, to enhance performance and achieve faster convergence. Finally, we experimented with multiple combinations of model parameters and optimizer to obtain an optimal model. As a result, we achieved a prediction accuracy of 91.47% on the CIFAR-10 (Krizhevsky and Hinton 2009) dataset with approximately 2.8 million trainable parameters. The codebase could be found through this publicly accessible github repository (https://github.com/TigaJi/DL-mini-project). (Ji, Liu and Douglas 2023)

Methodology

The architecture of this work consulted the github repository by Mountchicken (Mountchicken 2021) and class demo repository (Garimella 2023)

Datasets

In this project, the training dataset and testing dataset use the predetermined datasets by the CIFAR10 creators, which contains 50,000 training images and 10,000 test images.

Data Preprocessing

In order to help the model to learn features that are more invariant to the overall pixel values distribution and scale, we firstly calculate the mean and standard deviation of the training set data then apply the normalization function for each channel, this can help improve the convergence and overall performance of the neural network during training. Before applying the normalization, we also performed some data augmentation to artificially increase the diversity of the dataset. This could help with generating more training examples and mitigate potential overfitting. We performed random rotation, random horizontal flip, random crop, color alteration for the data augmentation for the training dataset and kept the test dataset plain.

ResNet Architecture

The ResNet architecture used in this project consists of multiple layers, mainly convolutional (Conv2d), batch normalization (BatchNorm2d), ReLU activation, and modified residual blocks followed by a fully connected layer. The network starts with an input size of 32x32 and gradually reduces spatial dimensions while increasing the number of channels. The final output of the network is a 10-dimensional vector, which represents class probabilities for each input.

Each ModifiedResidualBlock consists of two convolutional layers (Conv2d) with kernel size 3, followed by batch normalization (BatchNorm2d) and ReLU activation functions. Then a shortcut connection is included to allow the input to bypass the two convolutional layers. The shortcut connection consists of a 1x1 convolution followed by batch normalization when the input and output channels differ or the stride is not 1. The forward function defines the flow of data through the block, applying the two convolutional layers and merging the output with the shortcut connection before applying the final ReLU activation. In our model that gave the best result, each residual layer contains two residual blocks.

The ModifiedResNet consists of a convolutional layer with batch normalization and ReLU activation followed by four layers of residual blocks in our best trial. An adaptive average pooling layer reduces the spatial dimensions to 1x1, and the output is then flattened and passed through a fully connected layer to produce the final class predictions.

Increase the ModifiedResNet kernel size from 3 (with padding = 1) to 5 (with padding = 2) improved the pace of learning process as well as the highest accuracy from 40% to 60% in our early stage study, thus we kept using kernel size of 5 for all later trials. The padding size is also changed to maintain the consistent spatial dimensions of the output.

In order to properly initialize the parameter, we used *nn.init.kaiming_normal_* and *nn.init.constant_* to initialize the weights and biases of the convolutional and normalization layers in the architecture. This is supposed to ensure that the model could learn efficiently.

The *_make_layer* function is used to create a sequence of

ModifiedResidualBlock instances and stack them to form a layer within the ModifiedResNet architecture.

Training Process

From our preliminary trials, we figured out that in our architecture, 70 epoch training cycles could sufficiently represent the overall performance of the model. All of our training and testing cycles reached plateaus within 70 epochs, thus we kept this value for all later testing models. The three key evaluation parameters, train losses, test accuracies, learning rates are initialized to record the learning progress.

In each epoch, the model is firstly set to train mode, and iterates through the training data via loading data through DataLoader. The optimizer's gradients are initially set to zero, and apply the current model to the input images. CrossEntropy loss function is used to compute the loss between the model's predictions and the true labels. The gradients are then calculated by backpropagation, and the optimizer updates the model's weights. The learning rate scheduler (used CosineAnnealingLR in our case) adjusts the learning rate after each step. We stored the loss and learning rate in the initialized array accordingly. In our test scenarios, we tried both Adaptive Moment Estimation (Adam) and Stochastic Gradient Descent (SGD) as the optimizer. The outcome of the two optimizers are summarized in the next section.

Evaluating Process

The model is then set to evaluation mode. The test accuracy is assessed using the test dataset. Each test data batch is used for making a prediction where the predicted class is determined by selecting the class with the highest probability. The number of correct predictions and total samples is updated, and the test accuracy is calculated as the ratio of correct predictions to total samples. The best model with the highest accuracy is saved into *best_model.pth*.

Results and Analysis

There are many hyperparameters that could be optimized for the model. In our scenarios, we tried altering the batch size (128, 256, 512), optimizer (Adam, SGD), learning rate (0.001, 0.01, 0.1), number of residual layers (4 or 5) and the number of residual blocks in each layer (2 or 3). The total number of trainable parameters are also changed along with these hyperparameters (2,799,146, 4,587,306, 4,367,786).

When trying to increase the output channel numbers in the ModifiedResidualBlock, we figured out that it could easily expand the trainable parameters over 5 million thus we finally keep the architecture with output channels $32 \rightarrow 64 \rightarrow 128 \rightarrow 256$ in the ModifiedResNet.

The combination of hyperparameters are summarized in Table.1, all of our models reached the testing accuracy over 80%, with 7 of the models reaching the highest testing accuracy over 90%. The testing accuracy progress along with the number of epochs are summarized in Figure 1 to Figure 9. In the trials that yielded over 90% highest accuracy, the Model 5 reached a highest accuracy of 91.47%.

In Model 1, 5, 7, 8, by keeping the testing batch size the same, a 128 training batch size overall results in a smoother learning curve and more stable ending plateau after 70 epochs. This could be due to the fact that a smaller training batch size will result in a more frequent weights update for the model per epoch. This more frequent update acts as a gradually fine-tuning process of the parameter updating that could give a smoother accuracy curve over the course of training. Furthermore, using a smaller training batch size could introduce more noise in the gradient estimates, which can sometimes help the model to escape local minima and find better solutions. This batch size is considered optimal since it does not significantly slow down the learning process. The four models with this batch size all reached 80% around 10 epochs.

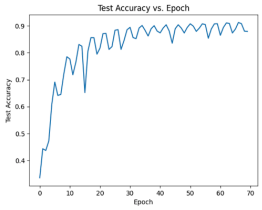

Figure 1: Model 0

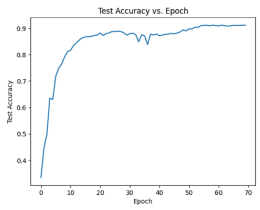

Figure 2: Model 1

The comparison of Model 0 and Model 2 (the only difference between these two models is the optimizer) shows that Adam is a better optimizer instead of SGD giving better accuracy of 91.28% instead of 83.25%. Therefore, we used SGD for the rest of the training.

Model 5, 6 and 7 show the impact of learning rate with all other parameters kept the same. The Model 6 shows that a learning rate of 0.1 is too slow for our scenario while Model 5 and Model 7 show competitive learning progress with Model 5 has a slightly faster learning progress and a slightly better final learning accuracy. The explanation could be that a larger learning rate, in theory, can speed up the learning process but may also result in overshooting the optimal solution thus causing oscillations around the optimal point and slow down the overall learning speed conversely.

Model 1 and Model 5 show the impact of the number of residual block layers. Model 1 contains one more layer with 3 residual blocks than Model 5, in which case, Model 1 has 60% more trainable parameters than Model 5 but did not yield a higher accuracy and a slightly slower learning speed. Therefore, we conclude that Model 5 is a better model with fewer trainable parameters that could save the computing power.

Model 5 and Model 8 show the impact of complexity of residual blocks, where Model 5 contains two residual blocks each residual layer while Model 8 contains three residual blocks each residual layer. The more complex architecture results in a near 60% increase in the total number of trainable parameters but did not improve the overall performance, therefore, we conclude that Model 5 is a better model with fewer trainable parameters but even slightly better final testing accuracy.

The total running times of each model are all very close on the scale of this project. In our testing environment (CoLab), the total cost of time for each trial is about 55min.

Model	Batch Size(training)	Batch Size(testing)	Optimizer	lr	#residual layer	#epoch	#params	Acc(%)
0	512	512	Adam	0.01	5	70	4,587,306	91.28
1	128	128	Adam	0.01	5	70	4,587,306	91.23
2	512	512	SGD	0.01	5	70	4,587,306	83.25
3	256	256	Adam	0.01	5	70	4,587,306	90.97
4	256	128	Adam	0.01	5	70	4,587,306	90.90
5	**128**	**128**	**Adam**	**0.01**	**4**	**70**	**2,799,146**	**91.47**
6	128	128	Adam	0.1	5	70	4,587,306	88.87
7	128	128	Adam	0.001	5	70	4,587,306	91.14
8	128	128	Adam	0.01	4	70	4,367,786	91.24

Table 1: Hyperparameters used for training process and results

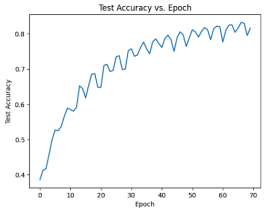

Figure 3: Model 2

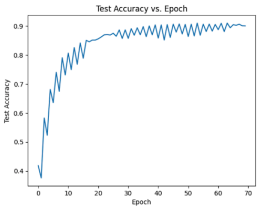

Figure 4: Model 3

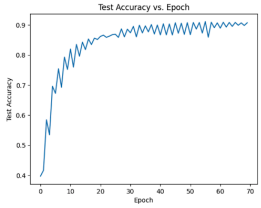

Figure 5: Model 4

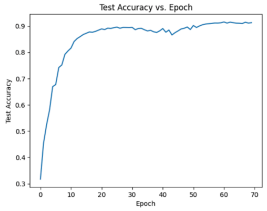

Figure 6: Model 5

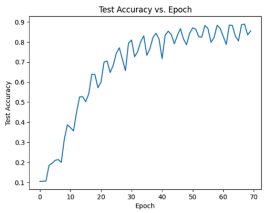

Figure 7: Model 6

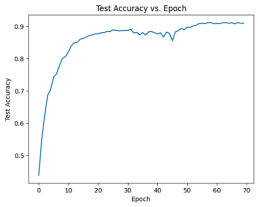

Figure 8: Model 7

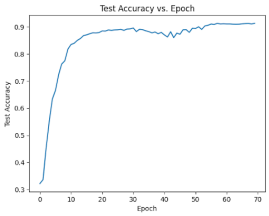

Figure 9: Model 8

Conclusion

Overall, the best model is Model 5 with training batch size of 128, Adam optimizer, learning rate of 0.01, four residual layers with two residual blocks each layer. Model 5 contains a total trainable parameter of 2,799,146 and yielding final testing accuracy of 91.47%. This outcome shows that it is not necessary that a more complicated model could yield a better performance.

For the future work of this project, we could increase trials to stress the stability of each model; Hyperparameter such as filter size, pool size and different scheduler could be studied. The learning rate could also be studied with finer increments along with the combinations of different optimizers since it could be one of the most impactive hyperparameters.

References

Garimella, K. V. 2023. Demo 05 - Implement ResNet and Visualize Loss Landscape. https://github.com/kvgarimella/dl-demos. Accessed: 2023-04-13.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2015. Deep residual learning for image recognition. arXiv:1512.03385.

Ji, D.; Liu, C.; and Douglas, J. 2023. DL-mini-project. https://github.com/TigaJi/DL-mini-project. Accessed: 2023-04-13.

Krizhevsky, A.; and Hinton, G. 2009. Learning multiple layers of features from tiny images. Technical report, University of Toronto.

Mountchicken. 2021. ResNet18-CIFAR10. https://github.com/Mountchicken/ResNet18-CIFAR10. Accessed: 2023-04-13.