# Deep Learning Mini-Project: Modified Resnet

Fengze Zhang, Yi Hou, Chenhao Zhang

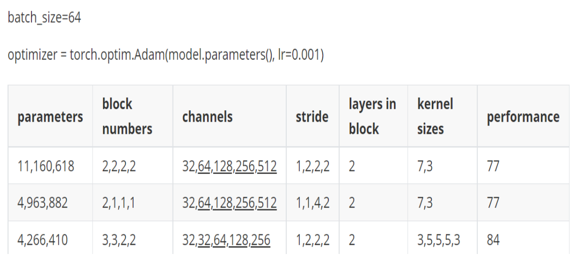
## Abstract

In this project, we tried to build a neuron network following the architecture of ResNet, which means that we have residual blocks using skip connections in our network. This model is trained on the CIFAR-10 image classification da-taset, which is a very famous and classic dataset com-posed with 6000 pictures (each of which is 32*32) in 10 categories. The only constraint for this project is that the number of parameters should be less than 5,000,000. No-ticing that the ResNet18 already has more than 10,000,000 parameters, we started this project based on this smallest model in the ResNet family. After building the ResNet18 from scratch, we experimented on it to help it perform better on CIFAR-10 while keeping its parame-ter number under the constraint. These experiments include structured pruning, data argument, making changes to convolution layers, etc. Running on Google Colab, the original ResNet18 sealed in pytorch has an ac-curacy of about 0.78 on original data while our model reached an accuracy of 0.91 on our augmented data. This report will cover the process of our project, methodology and result summary.

## Project Overview

In this project, our group built a ResNet based on the ResNet18 architecture. We first implemented the ResNet18 architecture based on the paper: *Deep Residual Learning for Image Recognition(He, 15)*[1] from scratch which helped us better understand the structure of the residual blocks and eased the following process of modification(compared with the existing model sealed in libraries). In the coding process, we referred to a youtube video focused on building ResNets[2], which inspired us on producing more general codes. After finishing the code base, we experimented on different aspects of the model and the dataset to get a better result while holding the constraint. These experiments include changing the number of blocks in the whole net, changing the channels, kernel sizes or the strides of the convolution layers, changing the place of specific convolution layers that are responsible for size decrease, trying different data argument functions, searching hyperparameters like learning rate, weight decay, batch size and optimizer functions and lr schedulers, etc. Figure 1 shows some of the remarkable experiment records we kept. After combining some of these results, we reached an accuracy of 0.91. During these experiments, we have several findings:

- Slight changes on parameters like stride and kernel sizes won't produce better performance. We guess this is because the structure of original models are the result of millions of experiments and are already structures of best practice.
- It is very important to keep the image size consistent between different kinds of blocks. In typical ResNets, the size of the image shrinks to half between blocks of different channels, which means only one convolution layer in all these blocks will have a stride of 2. Therefore, blocks of the same type are not completely the same and where to put the stride-2 layer can be a slight difference.
- Data is always the most important part in any deep learning problems. A good dataset always means a good possible result. In our experiments, we found that data argument can make a huge change to the training effect. After we used several data augmentation methods like horizontal and vertical flip or random rotation to enlarge the original CIFAR-10 dataset, this bigger and more varied dataset increased the performance of the same model greatly.
- Modifying channels can hugely affect the parameters and a suitable structured proning on specific datasets can help the performance.



batch_size=64

optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

| parameters | block numbers | channels | stride | layers in block | kernel sizes | performance |
|---|---|---|---|---|---|---|
| 11,160,618 | 2,2,2,2 | 32,64,128,256,512 | 1,2,2,2 | 2 | 7,3 | 77 |
| 4,963,882 | 2,1,1,1 | 32,64,128,256,512 | 1,1,4,2 | 2 | 7,3 | 77 |
| 4,266,410 | 3,3,2,2 | 32,32,64,128,256 | 1,2,2,2 | 2 | 3,5,5,5,3 | 84 |

Figure 1: The record of some combinations of different parameters, i.e. different models we tried.

Our github repository is : *https://github.com/DemoySegment/dl-miniproject-resnet*. In the repository, in addition to the jupyter codes, we also provide a model.pt which is a trained model we saved as backup. This one is obtained by running the *dl_miniproject.ipynb* script with at least 80 epoch training and can be tested with the *dl_miniproject_validation.ipynb* script.

## Methodology

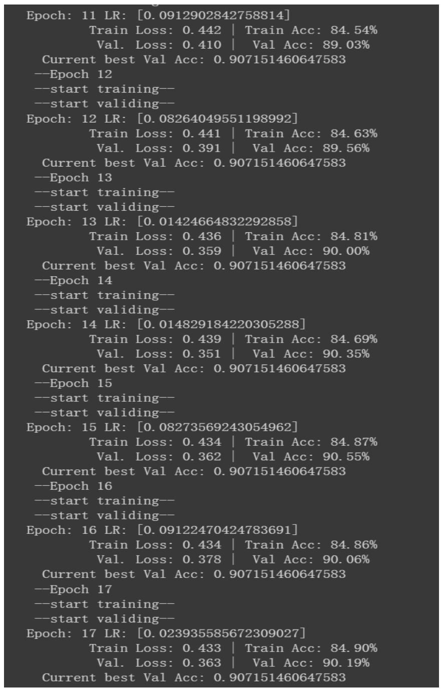
In this project, we first decided on building our network on the prototype of ResNet18. This is because ResNet18 is the smallest net in [1]. Even this net has a number of parameters of more than 10,000,000 which is about the double of the constraint. After deciding the basic structure of the net, we started building it with what are called basic blocks in [1]. [2] described codes to implement what are called bottleneck blocks for ResNet50, 101,150. We took the coding method in [2] to implement the ResNet18. To fulfill the parameter constraint, we tried the structured pruning cutting channels of all blocks to half. This changes the channels from 64-64-128-256-512 to 32-32-64-128-512. One thing to mention is that the CIFAR-10 is a dataset of relatively small pictures. Pictures are 32*32 in CIFAR-10. But ResNet in the original paper deals with the ImageNet dataset in which pictures are much larger. Therefore, reducing the channels may be suitable for this dataset. Also, in the original ResNet, after the first convolution layer before the blocks, there is a maxpool layer which reduces the size of the images. We deleted this layer considering that the original pictures are already small and a maxpool layer at the start may filter too much information for the following blocks. Experiment showed that deleting this maxpool can improve the result.

During the parameter experiments, we tried to modify some parameters like stride and kernel size and block numbers. These changes all have a slight influence on the result. A bigger kernel size can help but not much as well. What may have some effect is to change the stride-2 layer's position in the blocks. At first, our model's size-reducing layer is at the second layer of the blocks. Changing this layer to the first layer of the blocks helps improve the result. Also, it is mentioned in the original paper that the shortcut layer for skip connection may have a better result if it uses a kernel size of 1. We followed this suggestion.

Overall, making changes to the details of architecture does not help improve the result. After deciding the panorama of the block structure(number of layers in a block, how skip connection works in a block and the position of stride-2 layer in the block), ways to connect them under the constraint are limited. Therefore, we decided to handle data.

When training the model, we found that the model would overfit very quickly. In just 10 epochs, the training accuracy will reach 1 leading to a reduced validation accuracy. To overcome this problem, we copied the training dataset several times. For each of the copies, we made slight changes with different transforms increasing the number of training data. This dataset is much larger and provides more general information. Besides, we also use cosine learning rate scheduler to auto adjust to the learning rate. For the optimizer, we chose to use SGD instead of Adam because SGD is more frequently used in ResNet.

We also did some experiments on hyperparameters. To decide the batch size and learning rate initial value and weight decay, which is the regularity of the optimizer, we



```
Epoch: 11 LR: [0. 0912902842758814]
        Train Loss: 0. 442 | Train Acc: 84. 54%
         Val. Loss: 0. 410 |  Val Acc: 89. 03%
  Current best Val Acc: 0. 907151460647583
 --Epoch 12
 --start training--
 --start validing--
Epoch: 12 LR: [0. 08264049551198992]
        Train Loss: 0. 441 | Train Acc: 84. 63%
         Val. Loss: 0. 391 |  Val Acc: 89. 56%
  Current best Val Acc: 0. 907151460647583
 --Epoch 13
 --start training--
 --start validing--
Epoch: 13 LR: [0. 01424664832292858]
        Train Loss: 0. 436 | Train Acc: 84. 81%
         Val. Loss: 0. 359 |  Val Acc: 90. 00%
  Current best Val Acc: 0. 907151460647583
 --Epoch 14
 --start training--
 --start validing--
Epoch: 14 LR: [0. 014829184220305288]
        Train Loss: 0. 439 | Train Acc: 84. 69%
         Val. Loss: 0. 351 |  Val Acc: 90. 35%
  Current best Val Acc: 0. 907151460647583
 --Epoch 15
 --start training--
 --start validing--
Epoch: 15 LR: [0. 08273569243054962]
        Train Loss: 0. 434 | Train Acc: 84. 87%
         Val. Loss: 0. 362 |  Val Acc: 90. 55%
  Current best Val Acc: 0. 907151460647583
 --Epoch 16
 --start training--
 --start validing--
Epoch: 16 LR: [0. 09122470424783691]
        Train Loss: 0. 434 | Train Acc: 84. 86%
         Val. Loss: 0. 378 |  Val Acc: 90. 06%
  Current best Val Acc: 0. 907151460647583
 --Epoch 17
 --start training--
 --start validing--
Epoch: 17 LR: [0. 023935585672309027]
        Train Loss: 0. 433 | Train Acc: 84. 90%
         Val. Loss: 0. 363 |  Val Acc: 90. 19%
  Current best Val Acc: 0. 907151460647583
```

used exhaustive search to find the best tuple of these three.

After applying these methods, we found that the convergence rate became very low after reaching a training accuracy of 0.81 though validation accuracy was alway bigger than it, meaning that the model was not overfitting. As we trained the model for enough epochs(like 80 or more), the result of validation accuracy can reach 0.91.

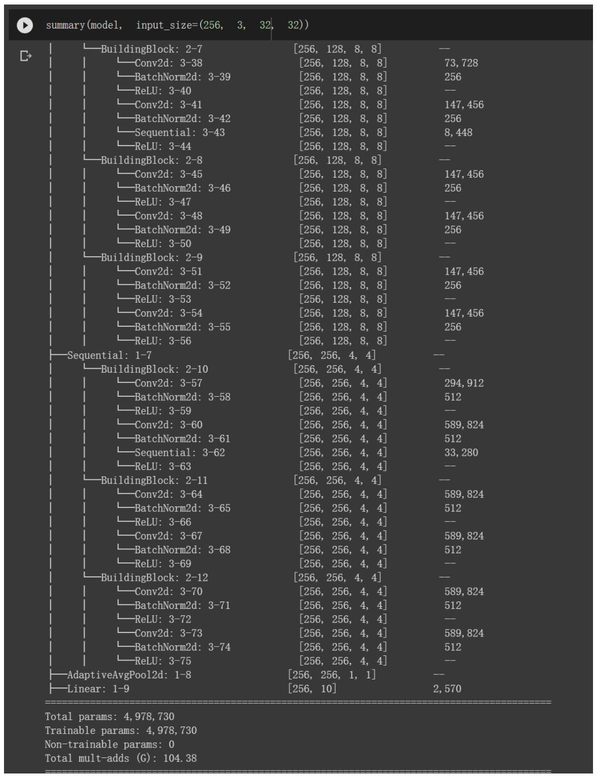
## Result

The final validation accuracy of our model can reach 0.91 after many epochs of training. It is worth noticing that after 30 epochs of training, the increase of both training accuracy and validation accuracy will be very slow. Even after 80 epochs, the model can still be trained to gain a better performance. But the process will be very slow.

Figure 2: The output of training and validation process

As shown in Figure2, though slow, the model does make better performance. This slow step starts when the training accuracy is around 0.81 while the validation accuracy is around 0.88.

The final parameter number is 4,978,730 as shown in Figure 3.

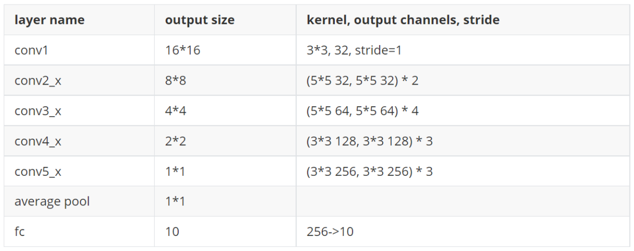


```
summary(model,  input_size=(256,  3,  32,  32))
```

```
│      └─BuildingBlock: 2-7                  [256, 128, 8, 8]          --
│      │      └─Conv2d: 3-38                  [256, 128, 8, 8]          73,728
│      │      └─BatchNorm2d: 3-39             [256, 128, 8, 8]          256
│      │      └─ReLU: 3-40                    [256, 128, 8, 8]          --
│      │      └─Conv2d: 3-41                  [256, 128, 8, 8]          147,456
│      │      └─BatchNorm2d: 3-42             [256, 128, 8, 8]          256
│      │      └─Sequential: 3-43              [256, 128, 8, 8]          8,448
│      │      └─ReLU: 3-44                    [256, 128, 8, 8]          --
│      └─BuildingBlock: 2-8                  [256, 128, 8, 8]          --
│      │      └─Conv2d: 3-45                  [256, 128, 8, 8]          147,456
│      │      └─BatchNorm2d: 3-46             [256, 128, 8, 8]          256
│      │      └─ReLU: 3-47                    [256, 128, 8, 8]          --
│      │      └─Conv2d: 3-48                  [256, 128, 8, 8]          147,456
│      │      └─BatchNorm2d: 3-49             [256, 128, 8, 8]          256
│      │      └─ReLU: 3-50                    [256, 128, 8, 8]          --
│      └─BuildingBlock: 2-9                  [256, 128, 8, 8]          --
│      │      └─Conv2d: 3-51                  [256, 128, 8, 8]          147,456
│      │      └─BatchNorm2d: 3-52             [256, 128, 8, 8]          256
│      │      └─ReLU: 3-53                    [256, 128, 8, 8]          --
│      │      └─Conv2d: 3-54                  [256, 128, 8, 8]          147,456
│      │      └─BatchNorm2d: 3-55             [256, 128, 8, 8]          256
│      │      └─ReLU: 3-56                    [256, 128, 8, 8]          --
├─Sequential: 1-7                           [256, 256, 4, 4]         --
│      └─BuildingBlock: 2-10                 [256, 256, 4, 4]          --
│      │      └─Conv2d: 3-57                  [256, 256, 4, 4]          294,912
│      │      └─BatchNorm2d: 3-58             [256, 256, 4, 4]          512
│      │      └─ReLU: 3-59                    [256, 256, 4, 4]          --
│      │      └─Conv2d: 3-60                  [256, 256, 4, 4]          589,824
│      │      └─BatchNorm2d: 3-61             [256, 256, 4, 4]          512
│      │      └─Sequential: 3-62              [256, 256, 4, 4]          33,280
│      │      └─ReLU: 3-63                    [256, 256, 4, 4]          --
│      └─BuildingBlock: 2-11                 [256, 256, 4, 4]          --
│      │      └─Conv2d: 3-64                  [256, 256, 4, 4]          589,824
│      │      └─BatchNorm2d: 3-65             [256, 256, 4, 4]          512
│      │      └─ReLU: 3-66                    [256, 256, 4, 4]          --
│      │      └─Conv2d: 3-67                  [256, 256, 4, 4]          589,824
│      │      └─BatchNorm2d: 3-68             [256, 256, 4, 4]          512
│      │      └─ReLU: 3-69                    [256, 256, 4, 4]          --
│      └─BuildingBlock: 2-12                 [256, 256, 4, 4]          --
│      │      └─Conv2d: 3-70                  [256, 256, 4, 4]          589,824
│      │      └─BatchNorm2d: 3-71             [256, 256, 4, 4]          512
│      │      └─ReLU: 3-72                    [256, 256, 4, 4]          --
│      │      └─Conv2d: 3-73                  [256, 256, 4, 4]          589,824
│      │      └─BatchNorm2d: 3-74             [256, 256, 4, 4]          512
│      │      └─ReLU: 3-75                    [256, 256, 4, 4]          --
├─AdaptiveAvgPool2d: 1-8                    [256, 256, 1, 1]         --
├─Linear: 1-9                               [256, 10]                2,570
==========================================================================================
Total params: 4,978,730
Trainable params: 4,978,730
Non-trainable params: 0
Total mult-adds (G): 104.38
==========================================================================================
```

Figure 3: The summary of the model.From the output, we can see that, the parameter number is 4,978,730

Figure 4 shows the structure of the model:



| layer name | output size | kernel, output channels, stride |
|---|---|---|
| conv1 | 16*16 | 3*3, 32, stride=1 |
| conv2_x | 8*8 | (5*5 32, 5*5 32) * 2 |
| conv3_x | 4*4 | (5*5 64, 5*5 64) * 4 |
| conv4_x | 2*2 | (3*3 128, 3*3 128) * 3 |
| conv5_x | 1*1 | (3*3 256, 3*3 256) * 3 |
| average pool | 1*1 | |
| fc | 10 | 256->10 |

Figure 4: The structure of model

In our model, each block contains two convolution layers of the same kernel size. Skip connections are applied to the two layers in one block. Note that one of the repeated blocks will have a layer of a stride of 2, in order to reduce the size of the image. Also, skip connections may involve additional convolution layers to handle the situation that two layers' input channels are not consistent with each other. More specifically, the first block of each kind may have additional convolution layers(for example, the first block of cov3 will have an input of 32*32, but output of 64*64, needing such an additional layer).

## Reference

[1] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

[2] Aladdin Persson. Pytorch ResNet implementation from Scratch [Video].
YouTube.
https://www.youtube.com/watch?v=DkNIBBBvcPs