Project-2 of “Neural Network and Deep Learning”

Wenzhe Zhang
19307110450

1. Train a Network on CIFAR-10

1.1. Introduction

CIFAR-10 is a widely used dataset for visual recognition task. The CIFAR-10 dataset contains 60,000 32×32 color images in 10 different classes. The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. There are 6,000 images of each class. In this project, I trained neural network models on CIFAR-10 to optimize performance.

1.2. Network Structure

In this project, I use ResNet-18 and did some modifications to obtain better performance on CIFAR10 according to the characteristics of the dataset. The idea of ResNet is that every additional layer should more easily contain the identity function as one of its elements. There are 4 convolutional layers in each module. Together with the first 7×7 convolutional layer and the final fully connected layer, there are 18 layers in total. Figure 1 depicts the full ResNet-18.

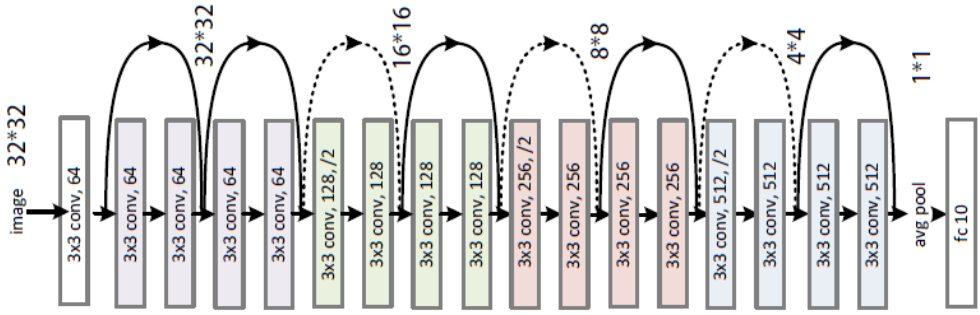

Figure 1: Network structure of ResNet-18

In my network, I changed the kernel size of the first convolution layer to 3×3, since the 7×7 filter in the first layer seems too large for a 32×32 image in CIFAR10. And the image is too small to have enough features, so here the pooling layer is disabled by a 1×1 pooling kernel. But my network still has a average pooling as its pooling method and ReLU as its default activation function. Thus, my network contains all the required components. (Fully-Connected layer, 2D convolutional layer, 2D pooling layer, Activations)

I use Batch-Norm layer in my network, and I didn't use Drop out. Since Some papers suggest that batch normalization may be a way to control model complexity by adding noise to each small batch, there is no need to mix it with the Drop out.

Here is the implementation of my residual block for the network:

```python
class BasicBlock(nn.Module):

    def __init__(self, in_features, out_features) -> None:
        super().__init__()
        self.in_features = in_features
        self.out_features = out_features

        stride = 1
        _features = out_features
        if self.in_features != self.out_features:

            if self.out_features / self.in_features == 2.0:
                stride = 2
            else:
                raise ValueError("输出特征数最多为输入特征数的2倍！")

        self.conv1 = Conv2d(in_features, _features, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = BatchNorm2d(_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        self.relu = ReLU(inplace=True)
        self.conv2 = Conv2d(_features, _features, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = BatchNorm2d(_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)


        self.downsample = None if self.in_features == self.out_features else nn.Sequential(
            Conv2d(in_features, out_features, kernel_size=1, stride=2, bias=False),
            BatchNorm2d(out_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )

    def forward(self, x):
        identity = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)

        if self.in_features != self.out_features:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)
```

```
42          return out
```

1.3. Optimize network with difference structures

1.3.1. Try different activations

I discussed three different activation functions in this part: ReLU, LeakyReLU and ELU.

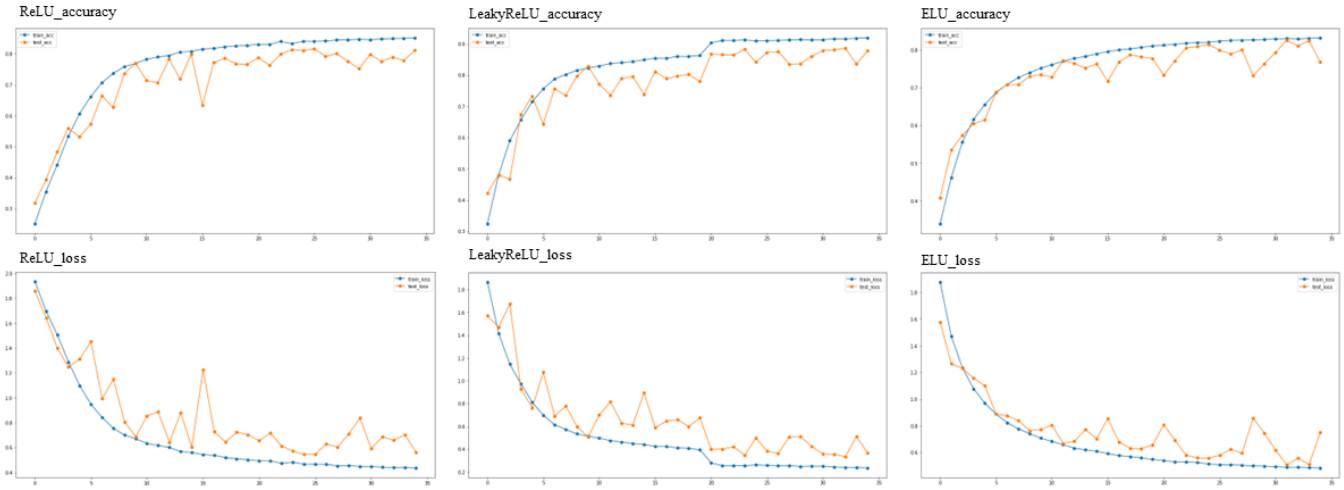

Figure 2: Accuracy and loss of different activation function

Activation	Test Accuracy	Time
ReLU	81.22%	1352s
LeakyReLU	88.67%	1267s
ELU	82.64%	1550s

Table 1: Test accuracy and training time of different activations(35 epoches)

As can be seen from the results, LeakyReLU has a better performance and can accelerate the process. Thus, I will use LeakyReLU in following training processes. This may because LeakyReLU can solve the 'Dead Neuron' problem. ReLU outputs 0 in negative inputs, which may prevent the network from updating certain parameters. LeakyReLU can produce a small gradient in negative inputs, avoiding the problem. Thus, LeakyReLU can imporve expression ability and accelerate convergence.

1.3.2. Try different loss functions

Crossentropy loss function is a commonly used choice because it can help the model to quickly converge during training and also takes into account the relative probability size between samples,

typically achieving decent classification performance. However, CIFAR10 dataset has complex samples that contain noise, rotation and other deformations, so I try L2-regulization with different weight decay.

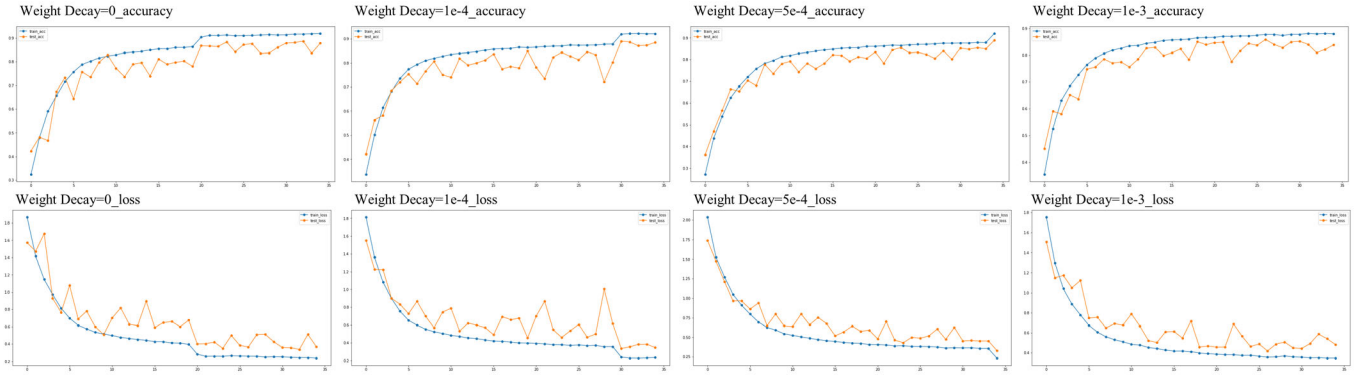

Figure 3: Accuracy and loss of different Weight Decay

Weight Decay	Test Accuracy	Time
0	88.67%	1267s
1e-4	89.06%	1273s
5e-4	88.93%	1326s
1e-3	85.89%	1056s

Table 2: Test accuracy and training time of different Weight Decay(35 epoches)

We can see that appropriate weight decay can help improve the performance and an improper weight decay can impair the performance of the model.

1.3.3. Try different number of neurons/filters

In order to change the number of filters, I try difference number of channels(32, 64, 128).

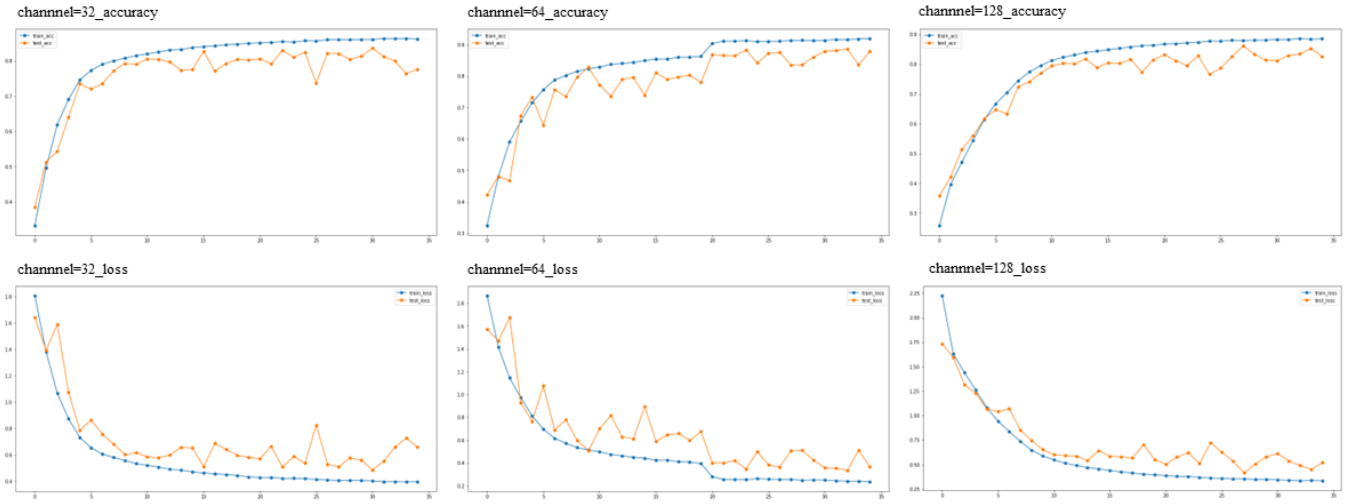

Figure 4: Accuracy and loss of different channel size

Channels	Test Accuracy	Time
32	83.59%	1135s
64	88.67%	1267s
128	86.11%	1670s

Table 3: Test accuracy and training time of different channel size(35 epoches)

As can be seen from the figure and table, more filters can lead to a higher accuracy on test set. But the training time for more filters is significantly higher than fewer filters. This is because the network with more filters can extract more features of the input image.

1.3.4. Try different optimizers using torch.optim

I tested four different optimizers in this part: SGD, Adam, Adagrad, RMSprop. We can see that there are several models doesn't seem to converge and have poor performance. This may because the dataset cantains only 60000 images, which is relatively small compared to some complex datasets. SGD is more likely to find a suitable descent direction, while Adam and Adagrad are more suitable for large-scale datasets. Besides, Adam and Adagrad require more hyperparameters and it can be challenging to find proper hyperparameter values.

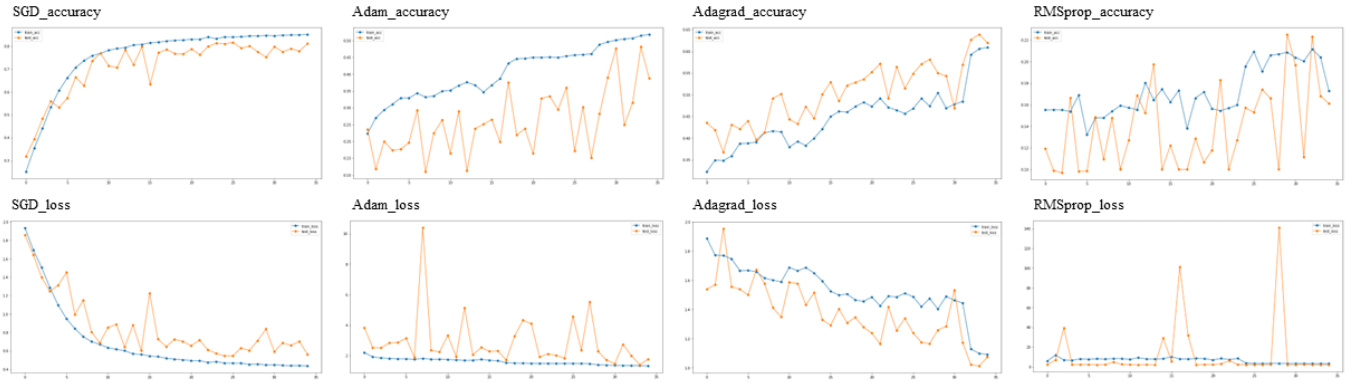

Figure 5: Accuracy and loss of different optimizers

Optimizer	Test Accuracy	Time
SGD	88.67%	1267s
Adam	48.17%	1779s
Adagrad	63.91%	1350s
RMSprop	31.15%	1445s

Table 4: Test accuracy and training time of different optimizers(35 epoches)

1.4. Final Model

My model is designed based on the idea of ResNet-18, and I optimized the model to make it more suitable for CIFAR10 dataset. I changed the kernel size and implement the weight decay method.

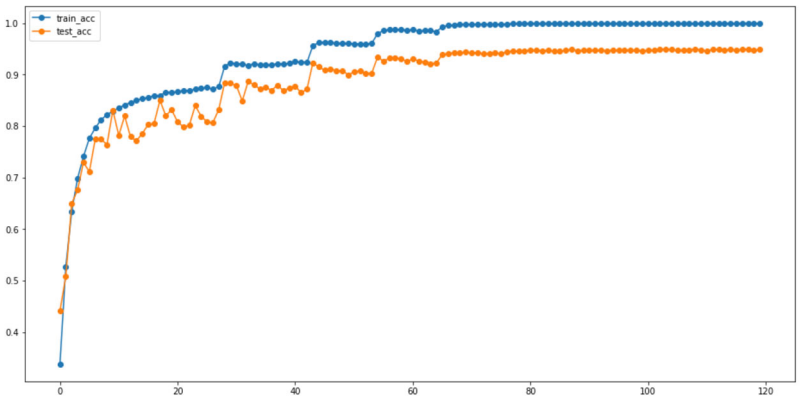

Figure 6: Accuracy curves of my final model

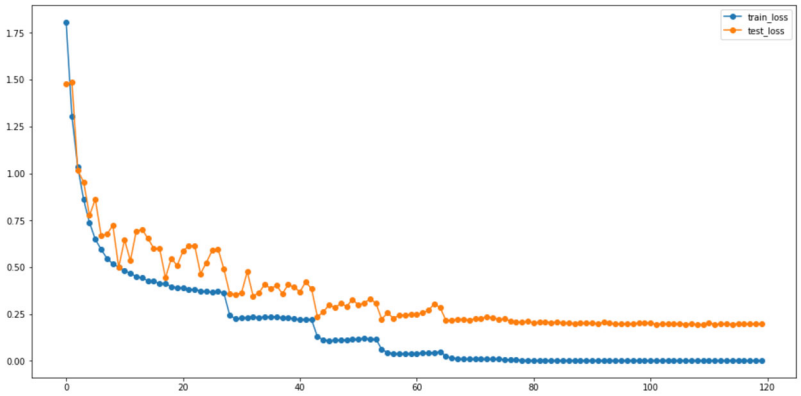

Figure 7: Loss curves of my final model

The accuracy on **validation set** can achieve **93.39%** with **55** epoch. The best result I achieve on this model is **99.99%** accurate on **training set**, **95.09%** on **validation set**, with **116** epochs of training. The hyperparameter setting of my model is as followed:

Hyperparameter	Value
epoch	120
batch size	128
initial learning rate	0.1
weight decay	0.0001
activation function	LeakyReLU
optimizer	SGD
model	ResNet-18
ResNet 1st convolution layer	(Kernel_size, stride, padding) = (3, 1, 1)

2. Batch Normalization

2.1. VGG-A with and without BN

I added BN layer with corresponding parameters after each CNN layer. Code of VGG_A_BatchNorm is in VGG.py. And I plot the curves of accuracy and curves of loss. Detailed code for this part is in VGG_train.py.

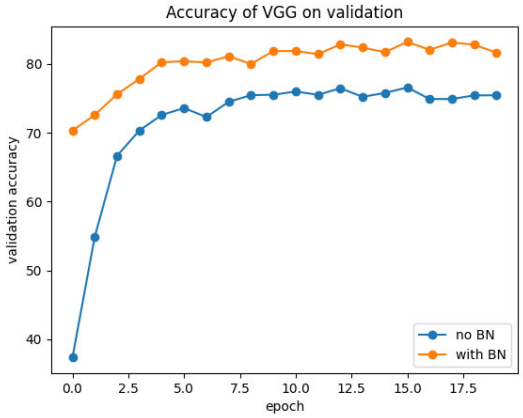

Figure 8: Accuracy curves with and without BN

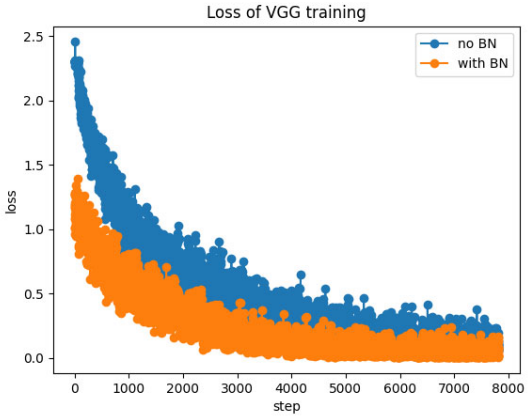

Figure 9: Loss curves with and without BN

From the curves, we can see that the network with BN layer has a better performance. And the accuracy on validation set with BN layer is 83% while 77% without it. Beside this, batch normalization can also accelerate the convergence and lower the loss.

The advantage of Batch Normalization lies in its characteristics. Batch Normalization can solve the problem of internal covariate shift by applying mean normalization and nromalization to each mini-batch input to change the distribution of neural network, making the input of each layer more stable and accelerating the convergence process.

2.2. Loss Landscape

Obviously, after adding BN layer, the loss drops faster and the convergence speed is accelerated. And the loss changes during the training process are more stable. This is because batch normalization can change the distribution of the neural network, making the input closer to the standard normal distribution, thereby increasing the nonlinear response of the neural network to the activation function, reducing the fluctuation of small parameter adjustments, and making the neural network more stable.

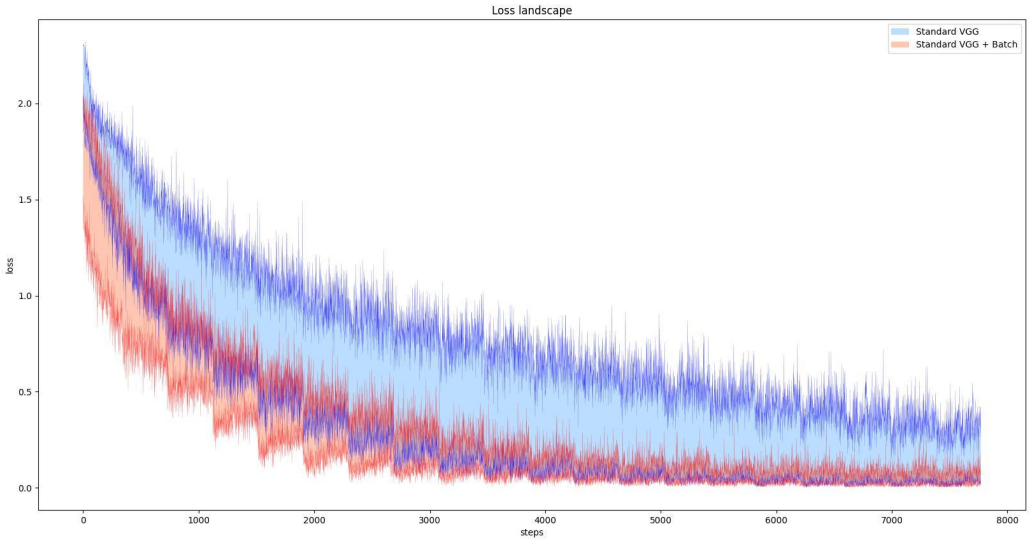

Figure 10: Loss Landscape

References

[1] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition." 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) 2016 (2016): 770-78. Web.

[2] Santurkar, Shibani, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. "How Does Batch Normalization Help Optimization?" (2018). Web.

[3] https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html.

[4] https://d2l.ai.