# Project-2 of “Neural Network and Deep Learning”

Haitian Jiang

Fudan University 19307110022@fudan.edu.cn

## 1 Train a Network on CIFAR-10

### 1.1 Dataset Construction

We obtained the latest dataset of CIFAR-10 from its open websites[1] and the size of the dataset is 60k. Then divided them into training set, validation set and test set according to 10:1:1.

### 1.2 Network Architecture

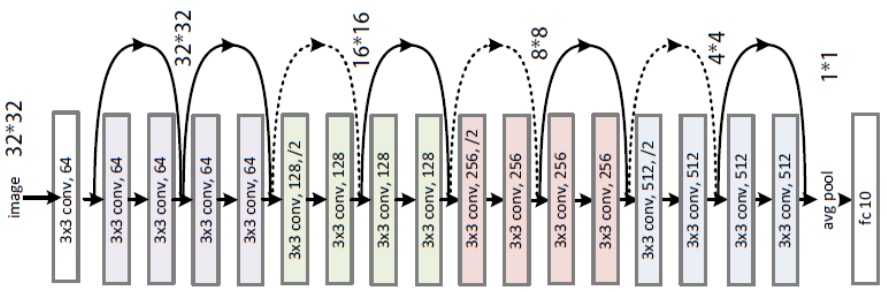

**Fig. 1.** The network structure of ResNet-18

In this project, I employed ResNet[2] as my backbone framework and did lots of modifications. As shown in Fig.1, the basic framework consists of 17 convolutional Layer and 1 fully connected layer. I implemented the network architecture myself and added or replaced some key components like channels size, activation functions and so on. Therefore it is by no means a public available model.

My network has average pooling as its pooling method and ReLU as its default activation function, with parameters to adjust to other options like ELU, sigmoid and others. So it contains all the requiered components: fully-connected layer, 2D convolutional layer, 2D pooling layer, activations.

I use batch normalization in the network architecture without dropout. On one hand, Batch normalization can introduce the statistical information from

[1] https://www.cs.toronto.edu/ kriz/cifar.html

other samples in the same batch to each sample so as to add noise to the data and thus reducing overfitting. On the other hand, adding dropout together with batch normalization will harm the performance for the convolutional layer[3]. Moreover, dropout is mostly used after a fully-connected layer, whereas my network only has it as the output layer, hence there is no place to put dropout even if I want. Therefore my network contains batch-norm layer and residual connection as its components.

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

```
class ResBlock(nn.Module):
    def __init__(self, in_channel, out_channel, stride=1,
                 batch_norm=True, activation='relu'):
        super().__init__()

        if activation == 'relu':
            self.activation = nn.ReLU(inplace=True)
        elif activation == 'sigmoid':
            self.activation = nn.Sigmoid()
        elif activation == 'tanh':
            self.activation == nn.Tanh()
        elif activation == 'leakyrelu':
            self.activation = nn.LeakyReLU(inplace=True)
        elif activation == 'elu':
            self.activation = nn.ELU(inplace=True)

        self.conv1 = nn.Conv2d(in_channel, out_channel, kernel_size=3,
                               stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channel) if batch_norm else nn.Sequential()
        self.conv2 = nn.Conv2d(out_channel, out_channel, kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channel) if batch_norm else nn.Sequential()

        self.shortcut = nn.Sequential()
        if stride != 1 or in_channel != out_channel:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channel, out_channel, kernel_size=1,
                          stride=stride, bias=False),
                nn.BatchNorm2d(out_channel) if batch_norm else nn.Sequential()
            )

    def forward(self, x):
        identity = self.shortcut(x)

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.activation(out)
        out = self.conv2(out)
        out = self.bn2(out)
        out += identity
        out = self.activation(out)

        return out
```

### 1.3 Optimize networks with different structures

**Try different number of neurons/filters**
Use the channel size parameter in my network to control the number of filters. I tried different channel size from 16 to 32 to 64. The training and validation accuracy and losses are shown in Fig.2

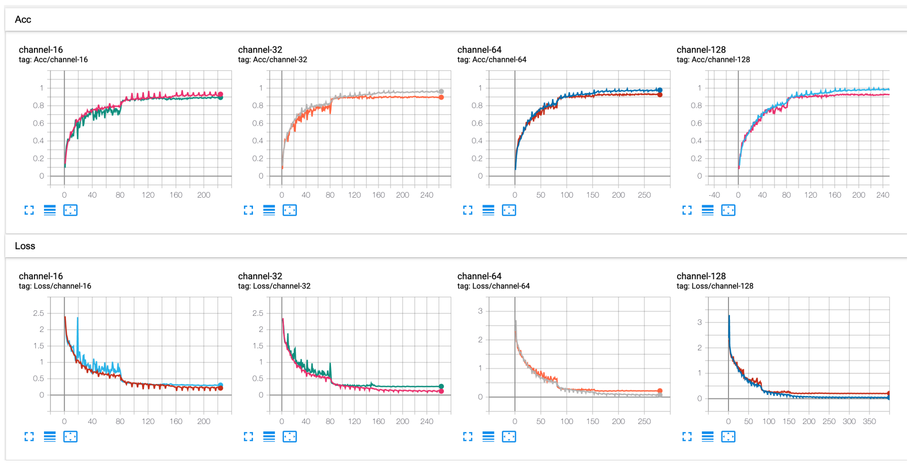

**Fig. 2.** Accuracy and loss of different channel size

| Channels | Test Accuracy | Test loss | Training time |
|---|---|---|---|
| 16 | 87.7586% | 0.3624 | 787s |
| 32 | 89.7845% | 0.3363 | 966s |
| 64 | 91.3793% | 0.2861 | 1132s |
| 128 | 92.7155% | 0.2672 | 1984s |

**Table 1.** Test accuracy and loss of different channel size

We can see that more filters produce a lower loss and higher accuracy on all training validation and test set. But the training time for more filters is significantly higher than fewer filters.

With the increasing of filters, the network can extract more ample features of the input image, thus produce a better result.

**Try different loss functions (with different regularization)**

Use different weight decay as different regularization, since this is a classification problem and cross entropy loss is the most suitable loss function.

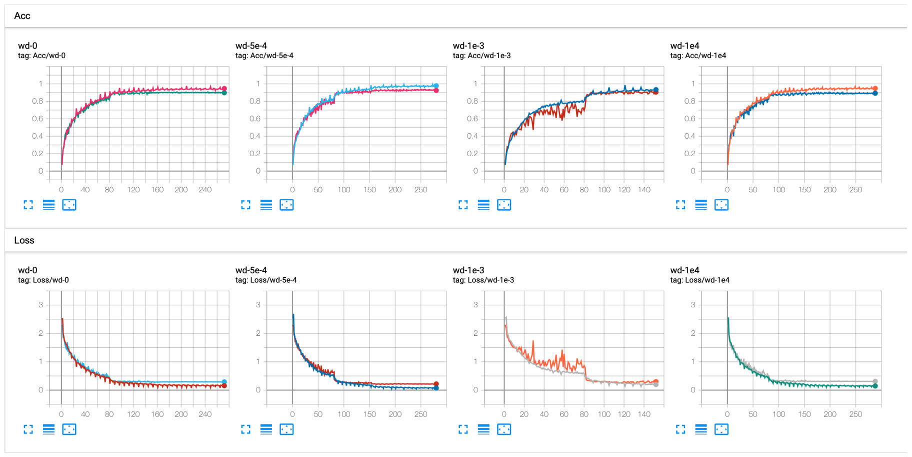

**Fig. 3.** Accuracy and loss of different loss function

| Weight Decay | Test Accuracy | Test loss | Training time |
|---|---|---|---|
| 0 | 88.3189% | 0.4015 | 1120s |
| 1e-4 | 88.8362% | 0.3704 | 1126s |
| 5e-4 | 91.3793% | 0.2861 | 1132s |
| 1e-3 | 88.9655% | 0.3191 | 1133s |

**Table 2.** Test accuracy and loss of different loss function

From the above result we can see that either too big or too small weight decay will hamper the performance, a moderate weight decay like 5e-4 is best for my network.

**Try different activations**

Use ReLU, ELU, LeakyReLU and sigmoid as activation functions.

We can see that the network performs bad with sigmoid as activation. That's why seldom does anyone still use sigmoid as activation function nowadays. For Leaky ReLU, the performance is nearly the same as ReLU, while ELU suffers from longer training time and slower convergence due to the innate computation intricacy brought by its exponential component. Therefore it is wise to just use ReLU as default activation function for its outstanding performance and eminent simplicity for calculation.

The reason of this phenomenon partially lies in the fact that the input of sigmoid with large numeric value will produce a tiny gradient, thus make it hard to propagate the gradient information, leading to a slow convergence and bad performance.

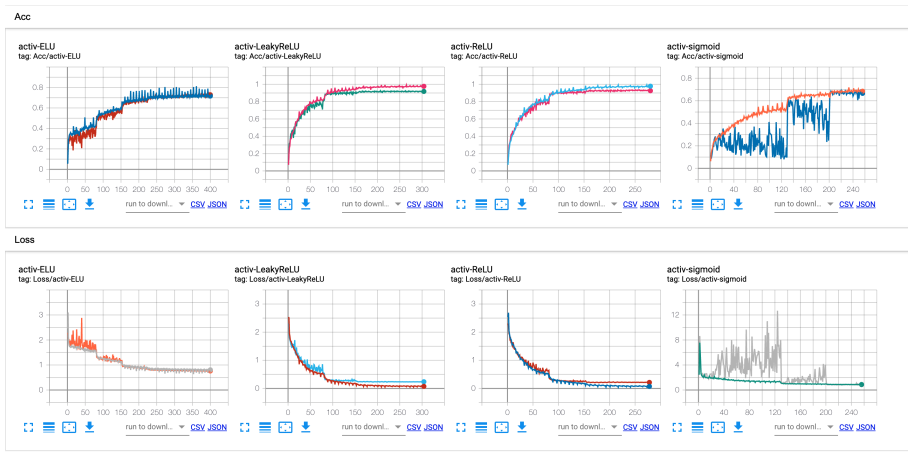

**Fig. 4.** Accuracy and loss of different activation function

| Activation | Test Accuracy | Test loss | Training time |
|---|---|---|---|
| sigmoid | 68.2327% | 0.8774 | 1084s |
| ELU | 88.3189% | 0.4015 | 1839s |
| LeakyReLU | 90.9913% | 0.2952 | 1211s |
| ReLU | 91.3793% | 0.2861 | 1246s |

**Table 3.** Test accuracy and loss of different channel size

## 1.4 Different optimizers

In this part, I tried different optimizers including stochastic gradient descent(SGD), SGD with momentum, AdaGrad and Adam.

Thanks to the sophisticated weight initialization offered natively by PyTorch, the SGD obtained similar accuracy both with or without momentum, suggesting a good configuration of hyper parameters, albeit the initial training loss and accuracy of mere SGD suffers from a conspicuous fluctuation, indicating that the momentum method provide a more robust training process.

When it comes to the adaptive learning methods like AdaGrad and Adam, great accuracy defect is shown in the figure, the reason of which lies in that adaptive learning methods are designed to find feasible weights even with poorly selected hyper-parameters. Therefore these methods tends to be stuck in sharp local minima or saddle points, from which momentum allows gradient descent method to escape. Also, a clever scheduling of learning rate endows these rather simple methods with the potency to outperform adaptive methods.

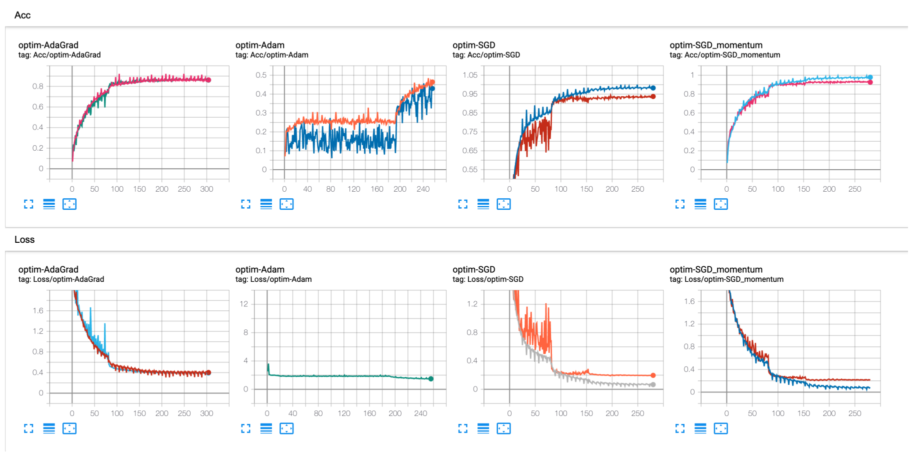

**Fig. 5.** Accuracy and loss of different activation function

| Activation | Test Accuracy | Test loss | Training time |
|---|---|---|---|
| Adam | 44.5258% | 1.5449 | 1431s |
| AdaGrad | 84.4396% | 0.4729 | 1202s |
| SGD-momentum | 91.3793% | 0.2861 | 1217s |
| SGD | 92.2413% | 0.2359 | 1059s |

**Table 4.** Test accuracy and loss of different channel size

## 1.5 Insights and visulization

### CNN filters

As the crucial part of convolutional neural networks, convolution kernels(filters) carries great weight in how the network extract features from images. Since only the filters in the first layer directly interact with the input image, and those in the deeper layers focus more on hidden representation and higher levels of pattern, it is reasonable to depict only the filters in the first layer as images.

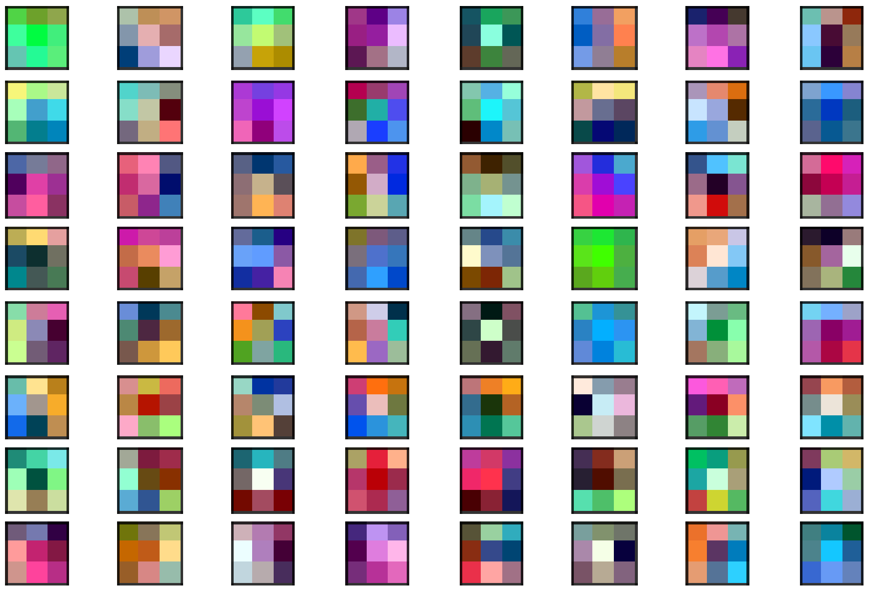

**Fig. 6.** Visualization of filters in first layer

I use the three channel in the filter to represent the corresponding RGB channels in the input image. Doing dot product seen as a way of examining similarity, the filters' color and intensity embody the similar patterns they are looking for. We can see that different filters learned different features to extract from the image, varying color patterns to the edge information. Certainly, due to the small size of input image(32x32), I used a relatively small input filter(3x3), which shows less salient features compared to previous work on 7x7 filters or 11x11 filters[4].

### Loss landscape

Loss landscapes are all shown and analyzed in the previous section, as a supplement to the analysis on accuracy.

**Hyper parameter choosing**

Here in all the above experiments, I was using a set of hyper-parameters with the performance and just changing one of them to show the contrast of different strategies. But the selection of hyper-parameter itself is a quite tough work to do.

Two methods of finding the desirable hyper-parameters were proposed. One is that we can arrange the these parameters and do grid search. This method needs an exponential of time with regard to the number of hyper-parameters, on account of the product rule for different parameters. If train in full procedure for each combination of these exponentially large parameter space, the time would be unreasonable. Another method is to randomly select a fixed number of parameter combinations, leading to a more scattered distribution in the parameter space, from which we can obtain more insights in how different values of parameters affect the performance.

Moreover, when doing parameter selection, only a few epochs of the initial training can embody how this set of parameter would perform in future training. Selecting parameters based on initial outcome can produce some decent final result.

**Best result**

The best result I achieve on this part is 99.22% accurate on training set, 93.58% on validation set and 92.72% on test set, with 32 epochs of training.

## 2 Batch Normalization

### 2.1 VGG-A with and without BN

The detailed code for this part of experiment is in `VGG_train.py`.

| Model | Training speed | validation acc |
|---|---|---|
| VGG_A | 7.26s/epoch | 76.58% |
| VGG_A_BN | 8.04s/epoch | 83.12% |

**Table 5.** Performance of VGG with and without BN

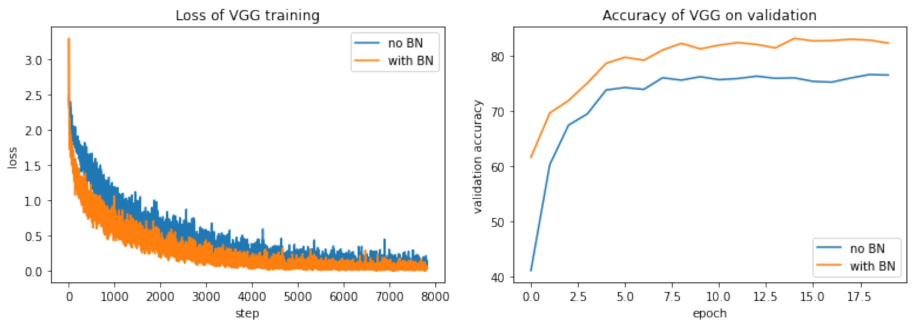

**Fig. 7.** Training loss and validation accuracy of VGG with or without BN

The tabular result suggests adding batch normalization layers slightly impairs the training speed while brings about a great boost in the accuracy. Other than better accuracy, batch normalization also has the merit of accelerate the convergence and lower the loss, as shown in Fig7.

The inherent reason for the better performance, as discuss in the previous section, lies in the capability of batch normalization to mix statistical information from other samples in the same batch to each sample, thus adding noise to the input image to reduce overfitting, which accounts for the better accuracy on the validation set. On the other hand, the quick drop of loss comes from the more smooth surface of the loss function, thanks to the redistributed data within each layer, which also enhance the gradient flows between layers.

### 2.2 How does BN help optimization: Loss Landscape

The detailed code for this part of experiment is in `VGG_extra.py`. The learning rate varies in 2e-3, 1e-3, 5e-4 and 1e-4.

Comparing the area sizes of the light-colored areas of the two models of Loss Landscape images, it can be seen that using the batch standardized model, the loss changes during the training process are less unstable, and the loss tends to zero more stably and accurately at each iteration. The reason for this is that batch normalization maintain the distribution of data to be close to zero, where the non-linearity of ReLU is best emphasized, thus making the loss landscape more smooth and less steep, leading to the similar loss at each step for different learning rates.

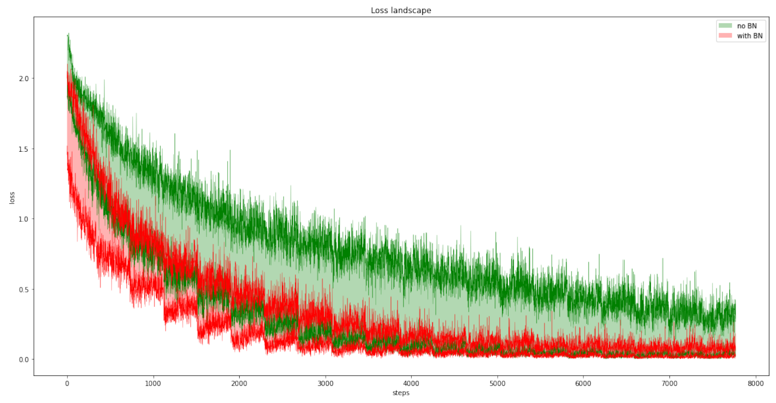

**Fig. 8.** Loss landscape of VGG with and without BN

## 2.3 Extra Bonus: Gradient Predictiveness

The detailed code for this part of experiment is also in `VGG_extra.py`. The learning rates are set the same as those in the loss landscape part: 2e-3, 1e-3, 5e-4, 1e-4.

Comparing the initial stage of training in the gradient predictiveness outcome figure, we can see that for the batch normalized model, the gradient value of the initial several iterations is often higher than that without batch normalization, but it drops down below its non-normalized counterpart in the latter stage. When batch normalization is not used, the data distribution in the early stage of model training is random, and it is likely that the data distribution is too extreme to pass into the nonlinear layer value, which leads to the disappearance of the gradient in back propagation during the first few steps, making the first few steps of iterations feckless, as the results shown from the loss landscape. After using batch normalization to qualify the data distribution, the value passed into the nonlinear layer is the data distribution fixed, so there is no disappearance of gradient in back propagation caused by extreme data distribution.

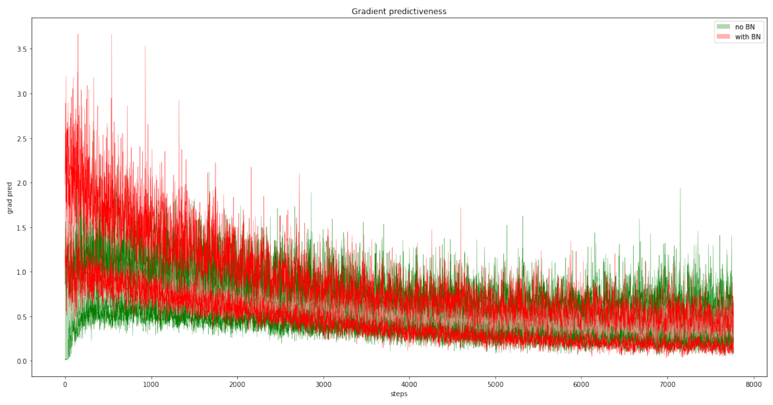

**Fig. 9.** Gradient predictiveness of VGG with and without BN

## 2.4 Extra Bonus: "Effective" $\beta$-Smoothness

The detailed code for this part of experiment is also in `VGG_extra.py`. The learning rates are set the same as those in the loss landscape part: 2e-3, 1e-3, 5e-4, 1e-4. According to the definition of Lipschitz continuity, a function $f(x)$ is $\beta$-smooth if $||\nabla f(x) - \nabla f(y)|| \leqslant \beta||x - y||$. The result figure shows that the $\beta$ for batch normalized network is similar to that for the not normalized network, but dropped lower with the increase of steps, showing that the local minima found by network with batch normalization is more smooth and flat than that by the non-normalized network.

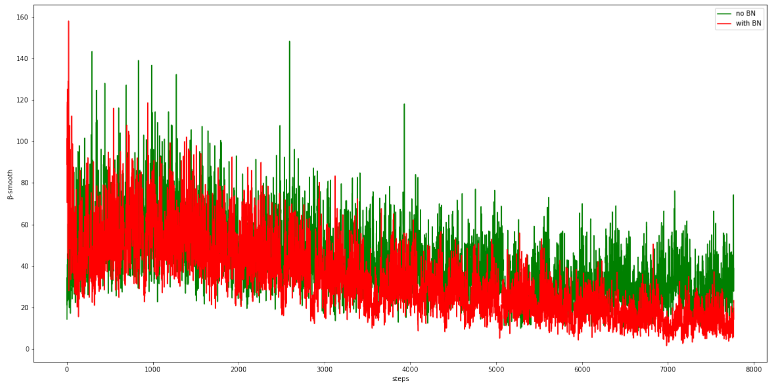

**Fig. 10.** $\beta$-smoothness of VGG with and without BN

## 3 Extra Bonus 2: DessiLBI

In this part, I use DessiLBI[1] as an ancillary to the Adam optimizer for the LeNet[5] on MNIST dataset to do the handwritten digits recognition task. The codes can be found in `dessilbi.py`.

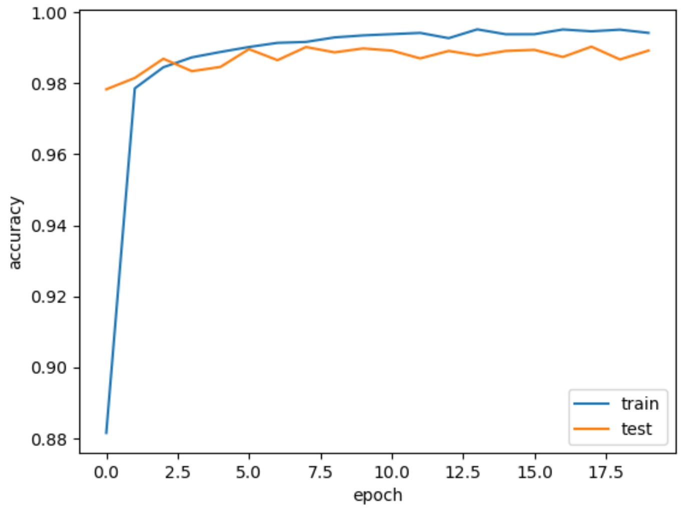

**Fig. 11.** Accuracy of using DessiLBI

The tiny difference between training loss and testing loss suggests the regularization effect of DessiLBI.

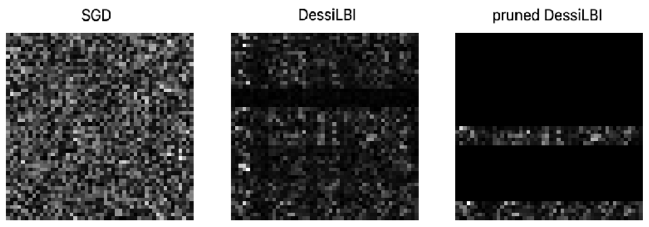

**Fig. 12.** Value in the last convolutional layer

Also, I tried the to prune on the network and visualize the last convolutional layer in the network. The luminous intensity in Fig.12 shows the magnitude of the weight value in the layer. We can see that using DessiLBI indeed added sparsity to the network parameter, without losing the accuracy, and the pruned model has less parameter. And from the table we can see that even so many parameters are pruned, the performance of the model didn't drop dramatically.

| Model | Test Accuracy |
|---|---|
| DessiLBI | 98.67% |
| prune last conv | 93.63% |
| prune last conv & last fc | 92.53 % |

**Table 6.** Test accuracy and loss of different channel size

## References

1. Fu, Y., Liu, C., Li, D., Sun, X., Zeng, J., Yao, Y.: Dessilbi: Exploring structural sparsity of deep networks via differential inclusion paths. In: International Conference on Machine Learning. pp. 3315–3326. PMLR (2020)
2. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 770–778 (2016)
3. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: International conference on machine learning. pp. 448–456. PMLR (2015)
4. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems **25** (2012)
5. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (1998)