# Lab3 Report

0856091 蘇邱弘

August 2020

## 1 Introduction

For a deep neural network, while the network's depth gets deeper, the accuracy gets saturate due to vanishing gradient problem. In order to solve this problem, ResNet (Residual Network) adds "shortcut connections" into its network architecture. These shortcuts tackle the vanishing gradient problem, and make the network easier to train. In this lab, I implement two ResNet architectures: ResNet18 and ResNet50, to do the classification tasks of Diabetic Retinopathy Detection dataset on kaggle.

In this report, I will first introduce the experiment setups, including the details of my ResNet models, the dataloader, and the evaluation through confusion matrix. Next, I will show my testing results, and the comparison figures between the pretrained ResNet model and without pretraining in same architectures. Finally, I will do some discussions.

## 2 Experiment Setups

### 2.1 Details of My ResNet Models

In this lab, I implement ResNet18 and ResNet50 using Pytorch, and the architecture of the two models are illustrated in Figure 1. As shown in the figure, for ResNet18, it uses Basic Block, which composed of two 3x3 convolution layers, as the building block; for ResNet50, it uses Bottleneck Block, which first uses one 1x1 convolution layer, then followed by one 3x3 convolution layer and one 1x1 convolution layer, as the building block. Figure 2 are the architectures of the Basic Block and the Bottleneck Block.

For the implementation procedure, first I implement the building blocks for ResNet18 and ResNet50 respectively (i.e., the Basic Block and the Bottleneck Block). After finishing the building blocks, I build up ResNet18 and ResNet50 using these building blocks according to the architecture in Figure 1. Finally, for the final fully connected layer, in order to fit the dataset, I change the input width to 512 for ResNet18 and to 2048 for ResNet50, and change the output width to 5.

The implementation above are the models without pretrainig. For the pretrained version, I substitute the layers in the models I implement originally with the pretrained models which provide by torchvision. The whole implementation detail can be found in resnet.py.

| layer name | output size | 18-layer | 34-layer | 50-layer | 101-layer | 152-layer |
|---|---|---|---|---|---|---|
| conv1 | 112×112 | 7×7, 64, stride 2 | | | | |
| conv2_x | 56×56 | 3×3 max pool, stride 2 | | | | |
| | | $\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$ | $\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$ |
| conv3_x | 28×28 | $\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$ | $\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$ | $\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$ | $\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$ | $\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$ |
| conv4_x | 14×14 | $\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$ | $\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$ | $\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$ | $\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$ | $\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$ |
| conv5_x | 7×7 | $\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$ | $\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$ |
| | 1×1 | average pool, 1000-d fc, softmax | | | | |
| FLOPs | | $1.8\times10^9$ | $3.6\times10^9$ | $3.8\times10^9$ | $7.6\times10^9$ | $11.3\times10^9$ |

Figure 1: ResNet Architecture

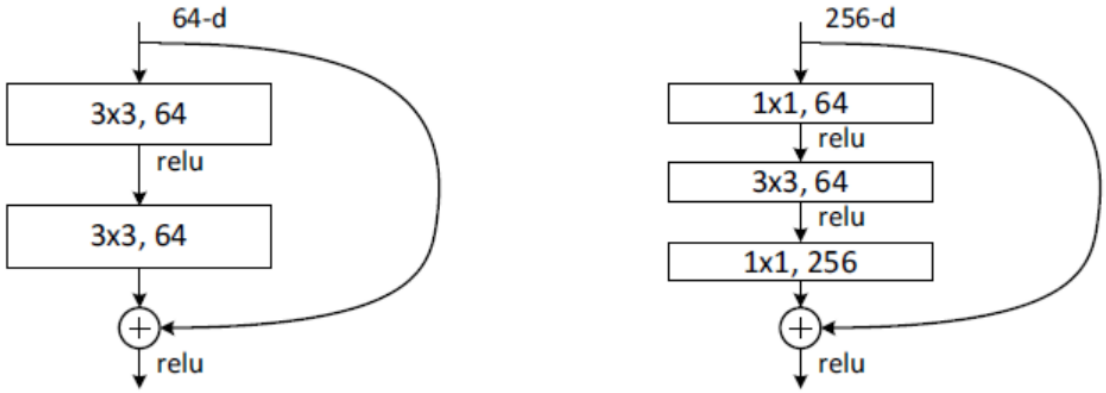



Figure 2: Architectures of the Basic Block (left) and the Bottleneck Block (right)

## 2.2 Details of My Dataloader

In this lab, I implement my own dataloader for Diabetic Retinopathy Detection dataset by using the template file providing by TA. The implementation detail is shown in Figure 3. The followings are the changes I made to the template file.

First, in the __init__ function, I add a parameter "transform" in order to perform the data augmentation to the training images, which can help increasing the data points during the training process. The augmentation I use is shown in Figure 4. For training data, I random flip and rotate the images, and then for both training and testing data, I use "ToTensor" to convert the image value to [0, 1] and transpose the image shape to [C, H, W].

Then, in the __getitem__ function, for the image, I first load the data through the PIL package, then return the tensor data after performing the transformation; for the label, I simply return it.

```python
class RetinopathyLoader(data.Dataset):
    def __init__(self, root, mode, transform):
        self.root = root
        self.img_name, self.label = getData(mode)
        self.mode = mode
        self.transform = transform

        print(f"> Found {len(self.img_name)} {mode}ing images...")

    def __len__(self):
        """'return the size of dataset"""
        return len(self.img_name)

    def __getitem__(self, index):
        label = self.label[index]

        path = self.root + self.img_name[index] + '.jpeg'
        img = self.transform(Image.open(path))

        return img, label
```

Figure 3: Implementation of my dataloader

```python
train_transform = transforms.Compose([
    transforms.RandomVerticalFlip(),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(90),
    transforms.ToTensor(),
])
test_transform = transforms.Compose([
    transforms.ToTensor(),
])

train_dataset = RetinopathyLoader(data_root_path, 'train', train_transform)
test_dataset = RetinopathyLoader(data_root_path, 'test', test_transform)
```

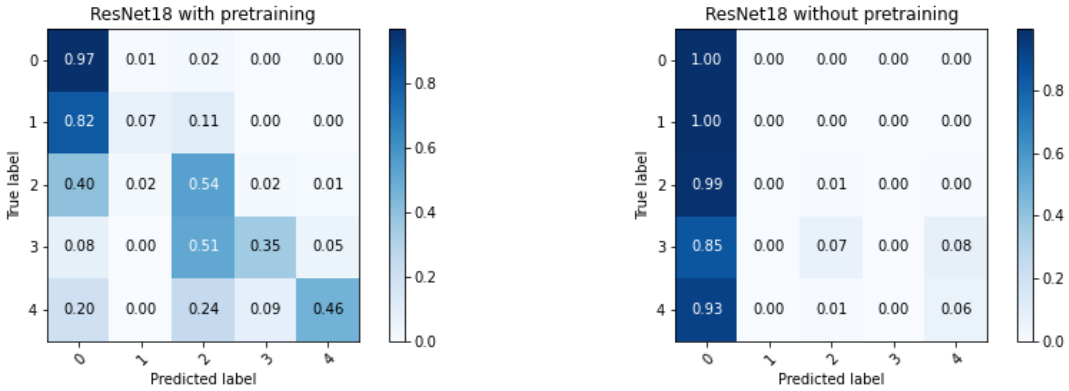
Figure 4: Implementation of data augmentation

## 2.3 Evaluation Through Confusion Matrix

A confusion matrix is a table that is often used to describe the performance of a classification model. The confusion matrices of ResNet18 and ResNet50 with/without pretraining are shown in Figure 5 and 6.

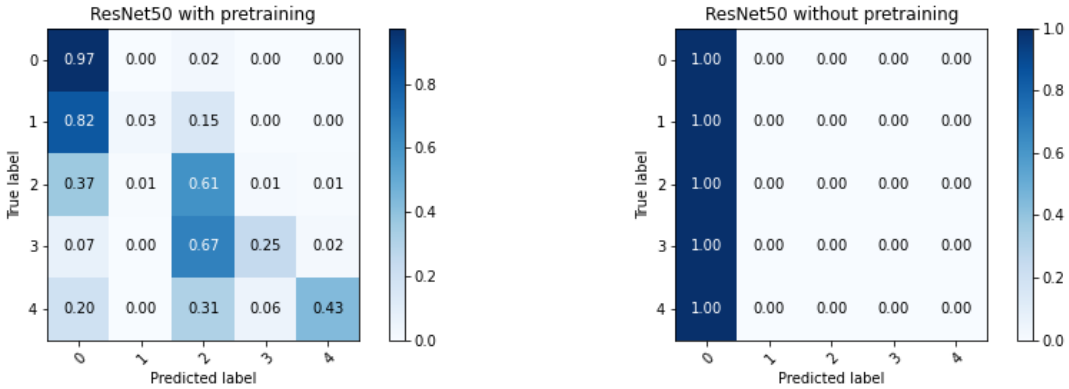
As shown in the figures, for both ResNet18 and ResNet50, when using the pretrained model, data which are labelled "2", "3", and "4" can be better classified, while almost all images are classified as class "0" when using models without pretrining. However, it is still

hard to classify data which are labelled "1", and maybe it's because that for patience with slight Retinopathy, their retina does not have obvious difference with normal retina.




(a) with pretraining (b) w/o pretraining

Figure 5: Confusion matrix of ResNet18




(a) with pretraining (b) w/o pretraining

Figure 6: Confusion matrix of ResNet50

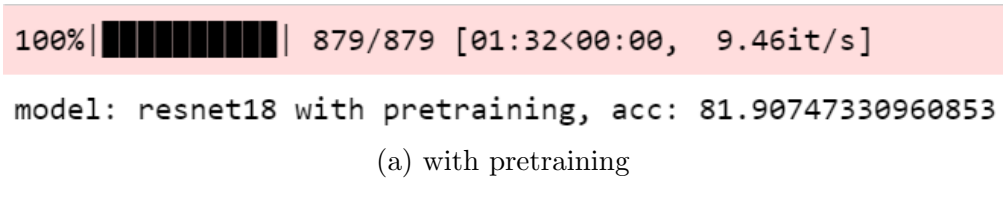
# 3 Experimental Results

The followings are the hyper-parameters I used for both ResNet18 and ResNet50:

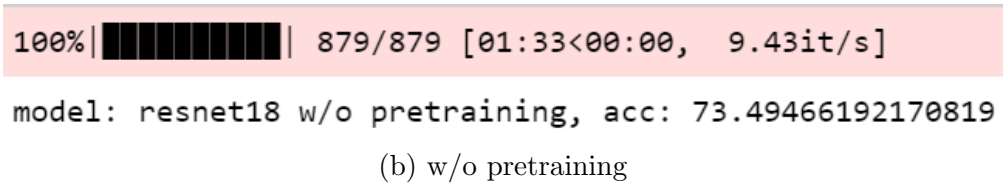
- batch size: 8
- epochs: 15
- learning rate: 0.001
- optimizer: SGD, momentum = 0.9, weight_decay = 5e-4, learning_rate=1e-3
- loss function: cross entropy

## 3.1 Highest Testing Accuracy

Figure 7 and 8 are the screenshots of the highest accuracy of the two models with/without pretraining. The highest accuracy 82.33% is achieved by the pretrainined ResNet50 in the 12-th epoch.

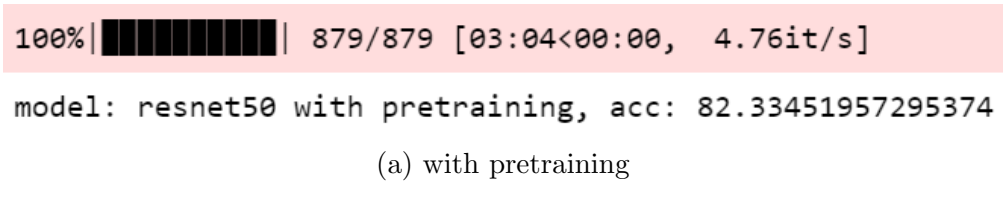



(a) with pretraining

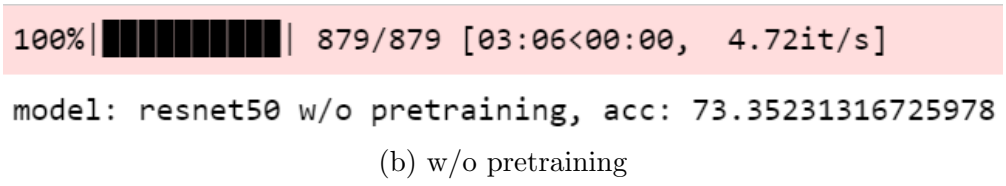



(b) w/o pretraining

Figure 7: Screenshots of ResNet18's results




(a) with pretraining

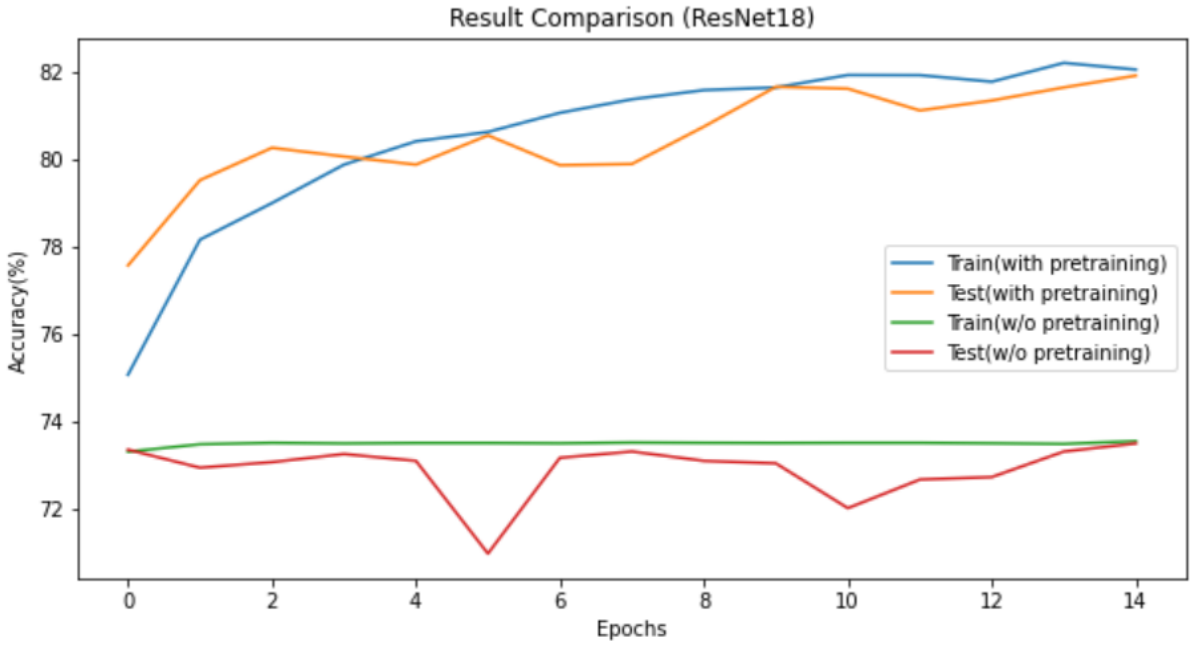



(b) w/o pretraining

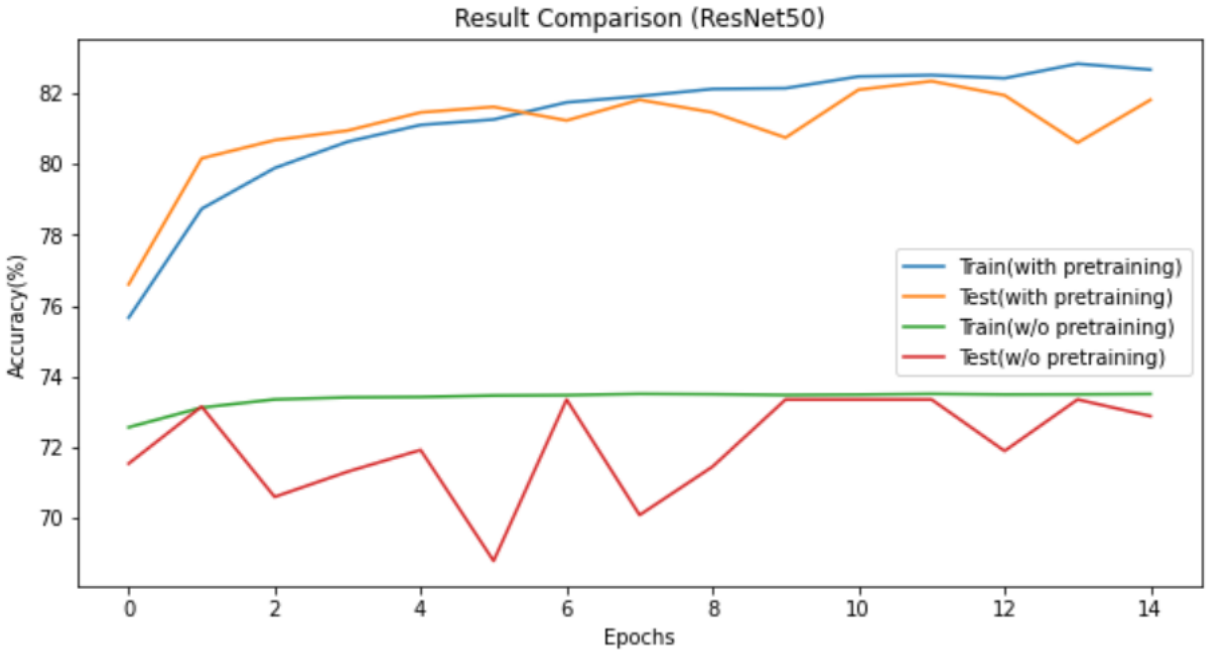
Figure 8: Screenshots of ResNet50's results

## 3.2 Comparison Figures

Figure 9 shows the comparison figures of the two models with/without pretraining. As shown in the figure, there's a large accuracy gap (almost 10%) between models with and without pretrining. In addition, for models without pretraining, the accuracy doesn't get improved after the first few epochs, which means that the models stop learning anything at some points. The reason why the models without pretrining stop learning may be because that the data is extremely unbalanced (most data are labelled "0"), so the models learn that when predicting most data as class "0", they can at least get about 70% accuracy. From the other perspective, we can say that their initial parameters are not good enough, which makes them trap into a bad local optimum during the training. On the other hands, the models with pretrining starts with better initial parameters, so that it can more easily escape some local optimum, and thus learn better.




(a) ResNet18




(b) ResNet50

Figure 9: Accuracy trend during training phase and testing phase

# 4 Discussion

## 4.1 Data augmentation

In this section, I take pretrained ResNet50 and use it to train two different models: one using training data with augmentation, while another using training data without augmentation. The experiment settings remain the same as Section 3, and the results are shown in Figure 10.

As shown in the figure, for the result which not applying data augmentation, although the training accuracy can achieve over 90%, which is much more higher than the case applying data augmentation, the testing result doesn't actually get better, and the accuracy even starts to drop in the last few epochs. I think the reason is that for the training data without augmentation, the data points are limited, and it ends up lead to the overfitting problem.

On the other hands, when using training data with augmentation, there are much more data points than the previous case, which helps the model to truly learn the pattern from the data, and thus lead to a better result on the testing data.

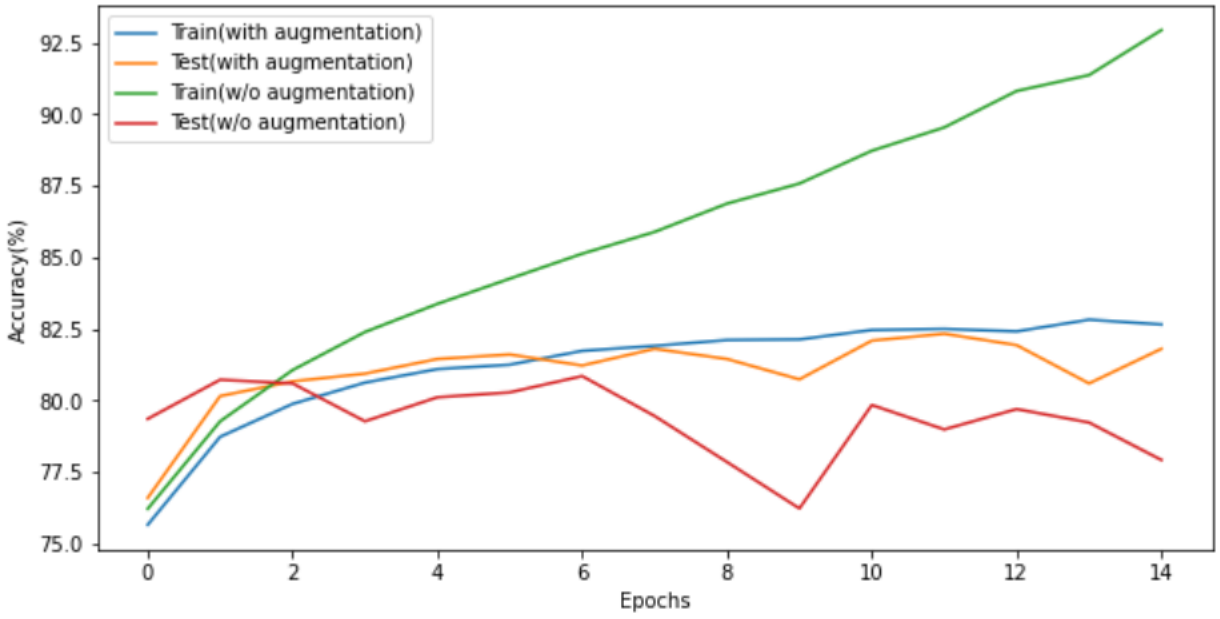



Figure 10: Accuracy trend comparison of pretrained ResNet50