DL-Lab4

0816095 Hsuan Wang

April 23, 2023

1 Introduction

In this lab, we are asked to implement a diabetic retinopathy classifier using ResNet architecture. We are also asked to preprocess the images by ourselves, this include cropping and resizing the image to the same size that is 512x512 for the network to process. We also need to implement a custom dataloader that can load, randomly rotate, flip. Finally normalize the data images ideally into a distribution of mean 0 and standard deviation 1 for faster convergence.

I implemented two architecture of ResNet, ResNet18 and ResNet50, and I am going to compare the results and discuss the difference.

2 Experiment Setup

2.1 ResNet

The emergence of ResNet is to solve the problem of vanishing/exploding gradient. We expect that the deeper the network becomes, the better it performs, but in reality it is exactly the opposite. This is the result of vanishing/-exploding gradient, and this problem becomes more serious the deeper the network goes.

Thus, in order to solve this problem, people come up with skip/shortcut connection that maps the input of a block to the output. This way, when performing back-propagation, there is a shortcut connection to propagate the information onward even if vanishing/exploding occur.

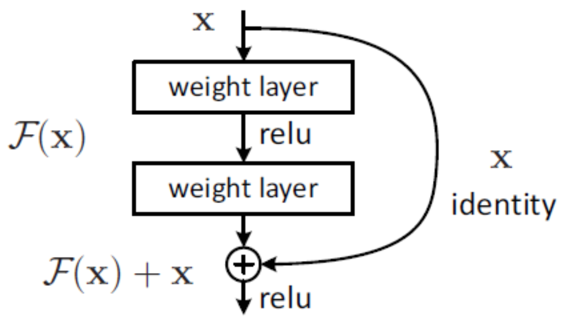

x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 1: Skip connection

So, the output of a single block becomes H(x)= F(x) + x, thus the weight layers are actually learning a kind of residual mapping of the actual output: F(x)=H(x)-x. Hence the name Residual Network (ResNet).

2.1.1 ResNet18

This is the first architecture of the two that I have implemented. The number 18 refers to the depth of the network, so this is a relatively shallow network (compared to ResNet50).

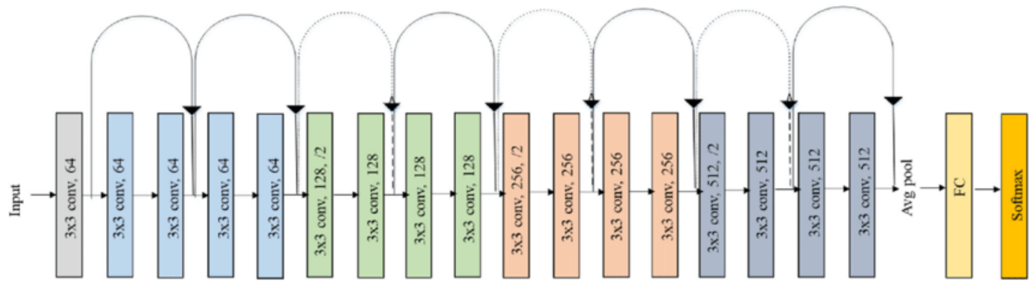


Figure 2: Architecture of ResNet18

2.1.2 Blocks in ResNet18

As we can see in figure 2, the network is made up of multiple basic blocks with skip connection, the implementation is show below:

```
class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, d_stride: Optional[int] = None):
        super(BasicBlock, self).__init__()
        self.d_stride = 1
        self.downsample = None
        if d_stride is not None:
            self.d_stride = d_stride
            self.downsample = downsample(in_channels, out_channels, d_stride)
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=(1, 1), stride=self.d_stride, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=(3, 3), stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        identity = x
        out = self.bn1(self.conv1(x))
        out = self.relu(out)
        out = self.bn2(self.conv2(out))
        out = self.relu(out)
        if self.downsample is not None:
            identity = self.downsample(x)
        out = out + identity
        out = self.relu(out)
        return out
```

2.1.3 ResNet50

Originally, this is called ResNet34, just a deeper version of ResNet18, but as the network gets deeper, time complexity becomes a problem. In order to solve this problem people come up with the bottleneck design:

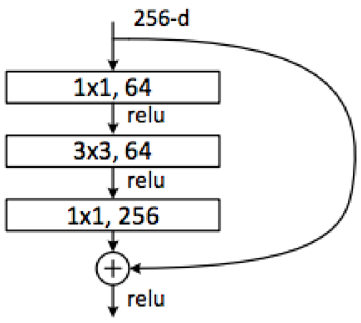


Figure 3: Bottleneck design

The added 1x1 convolution layer at the start and end of each block reduces the number of parameters while not degrading the performance too much. ResNet34 then becomes ResNet50, so ResNet50 can be seen as a deeper, improved version of ResNet18.

2.1.4 Bottlenecks in ResNet50

The bottleneck implementation is as below:

```python
class Bottleneck(nn.Module):
    def __init__(self, in_channels, width, out_channels, d_stride: Optional[int] = None):
        super(Bottleneck, self).__init__()
        self.d_stride = 1
        self.downsample = None
        if d_stride is not None:
            self.d_stride = d_stride
            self.downsample = downsample(in_channels, out_channels, d_stride)
        self.conv1 = nn.Conv2d(in_channels, width, kernel_size=(1, 1), stride=1, bias=False)
        self.bn1 = nn.BatchNorm2d(width)
        self.conv2 = nn.Conv2d(width, width, kernel_size=(3, 3), stride=self.d_stride, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(width)
        self.conv3 = nn.Conv2d(width, out_channels, kernel_size=(1, 1), stride=1, bias=False)
        self.bn3 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        identity = x
        out = self.bn1(self.conv1(x))
        out = self.relu(out)
        out = self.bn2(self.conv2(out))
        out = self.relu(out)
        out = self.bn3(self.conv3(out))
        if self.downsample is not None:
            identity = self.downsample(x)
        out = out + identity
        out = self.relu(out)
        return out
```

2.2 Dataloader

The main use of a dataloader is to transform the data into our desirable form. According to the hint given by TA, we need to first transform the range of image data from 0 255 to 0 1, and I do so using the ToTensor() function provided by torchvision. Next, we need to normalize the data into a distribution of desirably mean 0 and standard deviation 1, and I do so by using the Normalize() function, also provided by torchvision.

```
    def __getitem__(self, index):
        transform = T.Compose([
            T.RandomRotation(degrees=20),
            T.RandomHorizontalFlip(p=0.5),
            T.RandomVerticalFlip(p=0.5),
        ])
        normalization = T.Compose([
            T.ToTensor(),    # range [0, 255] -> [0.0,1.0]
            T.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])    #[mean, std]
        ])

        img = Image.open(f'{self.root}/{self.mode}/{self.img_name[index]}.jpeg')
        if self.mode == 'train':
            img = transform(img)
        img = normalization(img)
        label = self.label[index]
        return img, label
```

The question is how to get the mean and std of the original data. I could statistically calculate it, but I found it tedious, so I just look online for someone else's results.

2.3 Evaluation through Confusion Matrix

Confusion matrix is a specific graph used for visualizing the results of a classifier. Each row represents the ground truth and each column represents the predicted results. The darker the color means the higher the value in that cell, so we can have a pretty good grasp of how go a model is doing from just a glance at the matrix. Usually darker colours around the diagonal means better performance.

In my case, the value in the cells are ratio of range 0 1, and are meant to be read row-wise. In simple terms, each row has a sum of 1 and represents the distribution of the model prediction.

As we can see in the confusion matrix, the model has a tendency to predict "No DR", especially the one with out pretraining. Despite the model predict almost every class into "No DR" it still got an accuracy of 73%, this suggest that most of the ground truth is "No DR", this causes the model to guess "No DR" unless there are very obvious diabetic retinopathy syndromes.

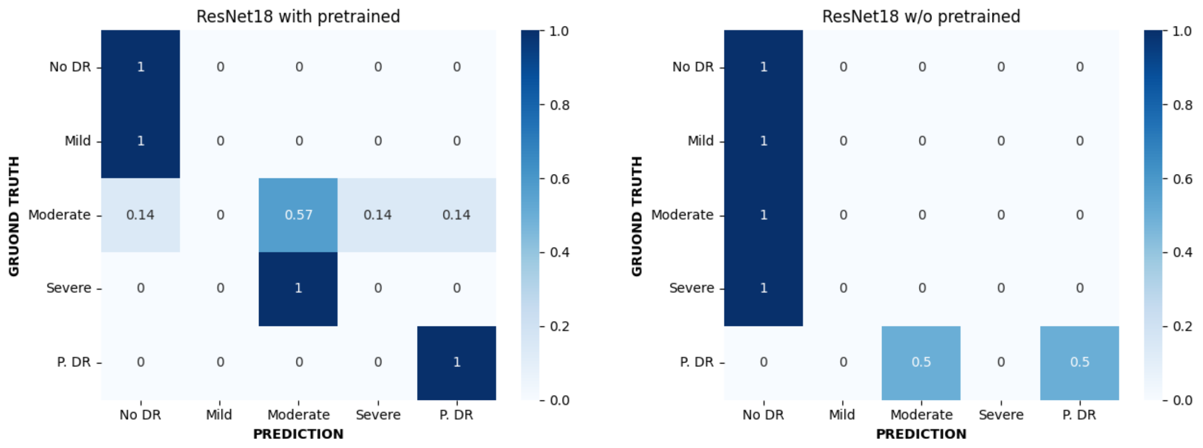


Figure 4: ResNet18's confusion matrix

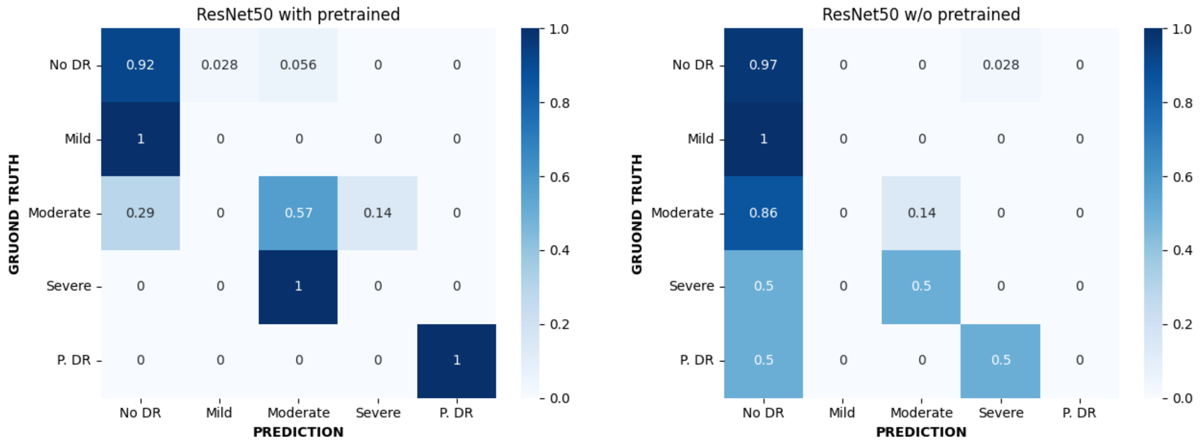


Figure 5: ResNet50's confusion matrix

3 Data Preprocessing

In this lab, we are given a dataset of various high resolution images of eyeball. Thus we are asked to preprocess them into a unanimous resolution of 512x512. Usually this is easily accomplished by resizing the image into 512x512, but this operation would squeeze the eyeball into an oval shape which is undesirable for training. Thus, we need to find the contour of the eyeball and crop the image accordingly, only then can we resize the image, below is an illustration:

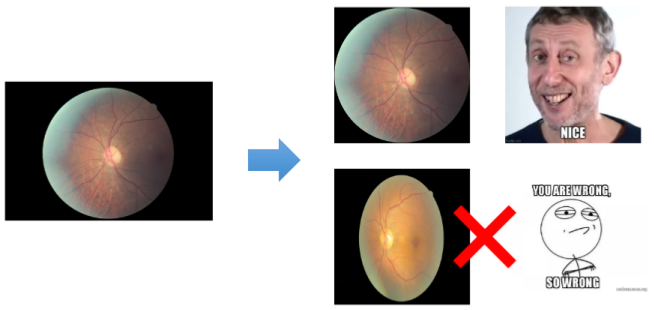


Figure 6: No squeezing the eyeball

```python
def crop_image(image):
    output = image.copy()
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    ret,gray = cv2.threshold(gray,25,255,cv2.THRESH_BINARY)
    contours,hierarchy = cv2.findContours(gray,cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE)
    if not contours:
        print('no contours!')
        return cv2.resize(image, (512, 512), interpolation=cv2.INTER_NEAREST)
    cnt = max(contours, key=cv2.contourArea)
    ((x, y), r) = cv2.minEnclosingCircle(cnt)
    x = int(x)
    y = int(y)
    r = int(r)
    H, W, C = output.shape
    if r > 100:
        if y - r < 0 or y + r > H:
            y_min = 0
            y_max = H
        else:
            y_min = y - r
            y_max = y + r

        if x - r < 0 or x + r > W:
            x_min = 0
            x_max = W
        else:
            x_min = x - r
            x_max = x + r

        output = output[y_min:y_max, x_min:x_max]
        if output.size == 0:
            output = cv2.resize(image, (512, 512), interpolation=cv2.INTER_NEAREST)
        else:
            output = cv2.resize(output, (512, 512), interpolation=cv2.INTER_NEAREST)
        return output
    else:
        output = cv2.resize(image, (512, 512), interpolation=cv2.INTER_NEAREST)
        return output
```

I used external package cv2 to accomplish this whole process. First transform the image into a grayscale image, then turn the image into a binary image containing only white and black using a threshold of (25, 255) (that is, all the pixels lower then 25 will be turned white and vice versa for every pixel higher then 25). Then find the contour using cv2.findContours() and crop the image accordingly, finally resize and return. I also hand remove a few corrupted images that I found during training for minimal interference to the model.

The point of turning the image into binary colour is for it to find the contour more easily. Setting the threshold to 25 is to prevent any noise from interfering. It is true that some of the eyeballs may still get squeezed, but it is negligible and unavoidable if we want to resize the image into 512x512.

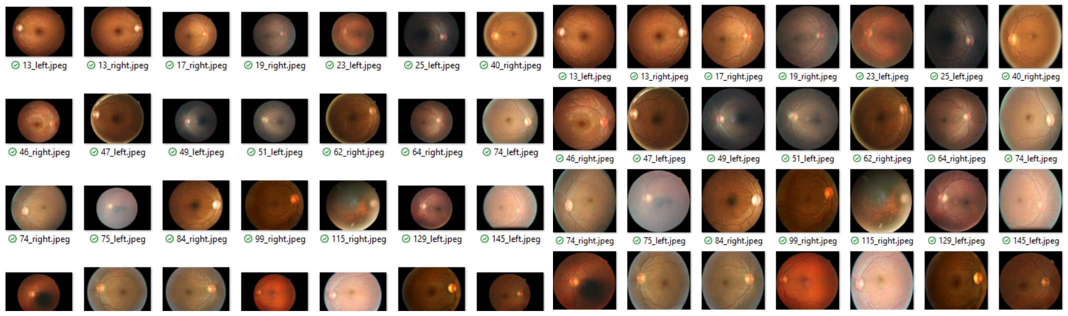
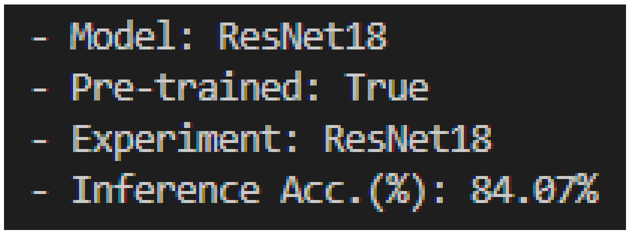


Figure 7: Testing dataset before (left) and after (right) preprocess

4 Experimental results

4.1 The highest testing accuracy

As we can see in the table, ResNet18 performs slightly better with pretraining, but ResNet50 has the upper hand when there are no pretraining.



```
- Model: ResNet18
- Pre-trained: True
- Experiment: ResNet18
- Inference Acc.(%): 84.07%
```

Figure 8: Screen shot of best result

	with pretraining	w/o pretraining
ResNet18	84.07%	73.57%
ResNet50	83.64%	74.45%

Table 1: Comparison of best performance between ResNet18 and ResNet50

4.2 Comparison figure

4.2.1 ResNet18 and ResNet50

From the figure we can see that both ResNet18 and ResNet50 performs significantly better when pretrained, both better then the baseline by over 1 percentage. I trained both networks with 20 epochs under basically the same condition, but we can see that pretrained ResNet50 is starting to overfit, although the training accuracy still goes up, the testing accuracy is gradually dropping. Whereas ResNet18, although the testing accuracy is showing some signs of slowing down, seems like it hasn't reached it's peek.

Models without pretraining performs roughly the same throughout the entire training process, with training accuracy just very slightly better then testing accuracy.

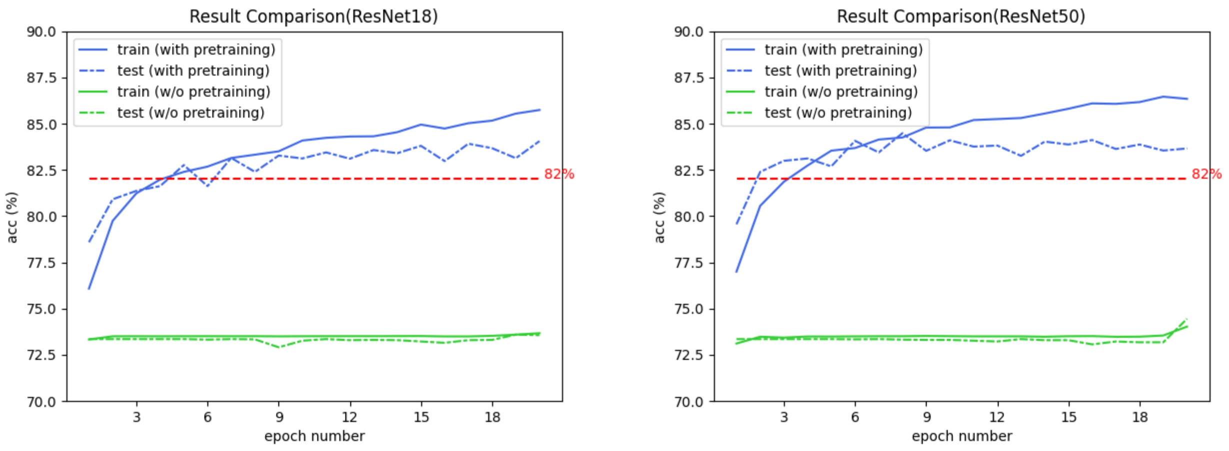


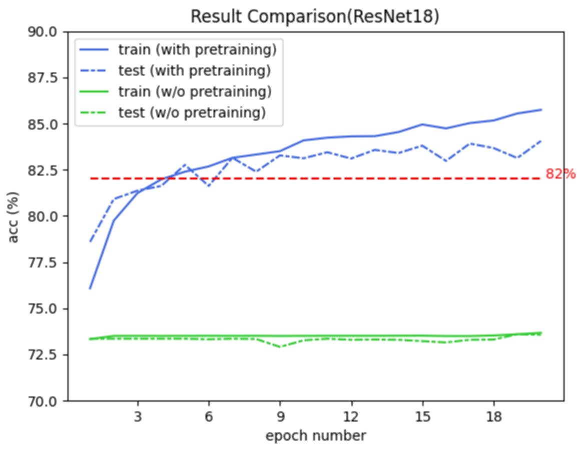
Figure 9: Comparison figure of ResNet18 and ResNet50

4.2.2 With and without flipping or rotating the training set

As previously mentioned, I flipped and rotated the training dataset randomly before feeding it into the model. Because in the testing dataset, eyeballs are not necessarily positioned upright, they may include some noise of rotation, so, mixing up the training set is extremely important, it not only makes the model able to recognize rotation, but also prevents over-fitting by a huge margin as we will see in the comparison graph later.

I used my best performing model and trained under the same condition, one with the randomly rotated training set (left), and one without (right). We can see that without this random rotation, the model becomes very overfitted, eventhough the training accuracy approaches 100%, testing accuracy starts to drop at about 5 epoch.

Whereas the one with randomly rotated training set has a much smoother training process, and ultimately reached a better score then the one without rotation.



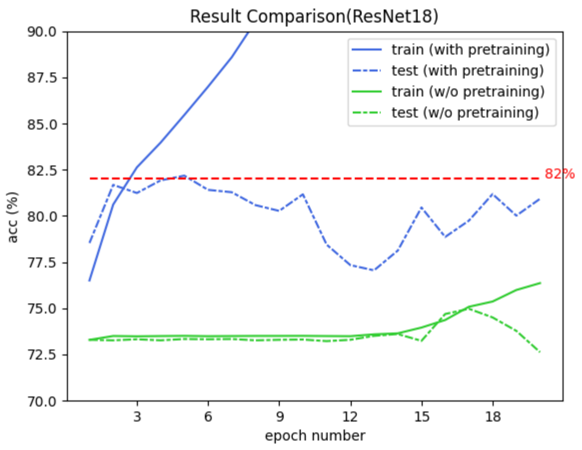


Figure 10: Comparison figure of with (left) and without (right) flipping or rotating training set

5 Discussion

All in all, ResNet18 has a slightly better score then ResNet50, and randomly rotating the training set proved to be a very effective way of boosting the model's performance. I am actually very surprised that ResNet18 performs better, because intuitively, the deeper the better. Maybe the bottleneck design took a toll on the performance of ResNet50, and the trade of between model depth and time complexity isn't worth it after all.

To find out, maybe I could implement a ResNet34 without the bottleneck design and run some experiment, then compare it with ResNet18, but unfortunately I am running out of time.