DLP-Lab4-2

ILE 310553043 You-Pin, Chen

April 24, 2022

1 Introduction

In this assignment, we are asked to implement a custom Dataloader which include three functions:*__init__*, *__len__* and *__getitem__*. *__init__* is the initialization of parameters in the class. *__len__* returns the number of whole dataset and *__getitem__* is the way to get the data. Inside the custom Dataloader, we also implemented *Transforms* from Pytorch package which is used to do data augmentation. The dataset we used in this lab is Diabetic Retinopathy Detection from Kaggle[link]. It provided with a large set of high-resolution retina images taken under a variety of imaging conditions. However, TA had already resized the image into 512x512 as preprocessing. This dataset contains five classes which is No DR, Mild, Moderate, Severe and Proliferative DR. The task is a classification problem that we also need to implement the ResNet18 and ResNet50 architecture and load parameters from a pretrained model. The baseline is 82% to get the whole points and we got 83.13% with pretrained ResNet50. The code is available in the github[link].

2 Experiment Setup

2.1 ResNet

ResNet is a network architecture using residual block to solve the issue of gradient dispersion and accuracy degradation (training set) in deep networks. Therefore the network can get more deeper while still controlling the accuracy and speed. The gradient dispersion is happened as the number of layers is increasing, the gradient in backpropagation in the network become unstable. While as the layer is increased, the accuracy will reach into saturation at one point then degrade and this issue is not caused by overfitting. In order to solve the degradation issue, the shallower the network will perform better. By using this idea, ResNet skip the training of few layers by using skip connections or residual connections. Fig.1 showing the single residual block used in ResNet and Fig.2 showing the ResNet architecture. The layers in a residual network are learning the residual ($R(x)$), hence the name is Residual Block[1].

$$R(x) = Output|Input = H(x)|x \tag{2.1.1}$$

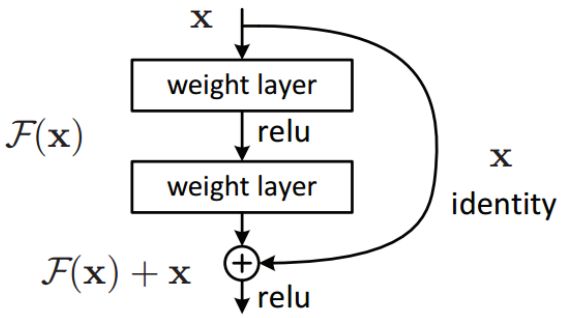


Figure 1: ResNet single residual block

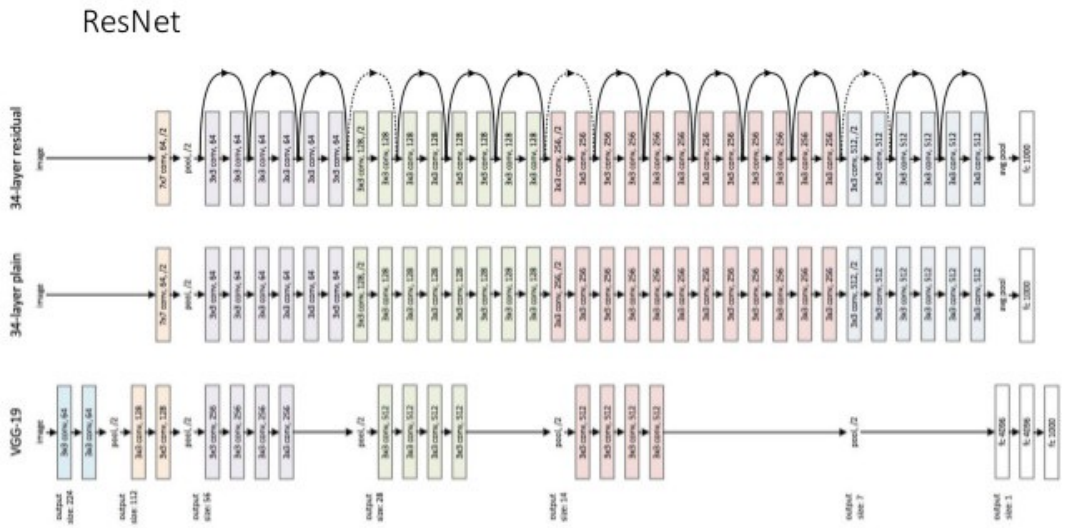


Figure 2: ResNet architecture

2.1.1 Block in ResNet

Different between ResNet18 and ResNet50 is that ResNet18 use Basic block to solve the gradient vanishing problem which ResNet50 use Bottleneck block. A skip connection is added to add the input which is named identity to the output after few weight layers. Basic block contains two 3x3 convolutional layers and Bottleneck block contains 1x1 Conv layer with 3x3 Conv layer and another 1x1 Conv layer as output layer. The name *Bottleneck* is because the shape of architecture. The implementation of two blocks are as 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
```

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

2.1.2 ResNet18 Implementation

```
class ResNet18(nn.Module):
    def __init__(self):
        super(ResNet18, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=(7, 7), stride=2, padding=3, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True)
        )
        self.conv2 = nn.Sequential(
            nn.MaxPool2d(kernel_size=(3, 3), stride=2, padding=1),
            BasicBlock(64, 64),
            BasicBlock(64, 64)
        )
        self.conv3 = nn.Sequential(
            BasicBlock(64, 128, 2),
            BasicBlock(128, 128)
        )
        self.conv4 = nn.Sequential(
            BasicBlock(128, 256, 2),
            BasicBlock(256, 256)
        )
        self.conv5 = nn.Sequential(
            BasicBlock(256, 512, 2),
            BasicBlock(512, 512)
        )
        self.avgpool = nn.AdaptiveAvgPool2d(output_size=(1, 1))
        self.fc = nn.Linear(512, 5)
```

```
    def forward(self, x):
        out = self.conv1(x)
        out = self.conv2(out)
        out = self.conv3(out)
        out = self.conv4(out)
        out = self.conv5(out)
        out = self.avgpool(out)
        out = self.fc(out.reshape(out.shape[0], -1))
        return out
```

2.1.3 ResNet50 Implementation

```
class ResNet50(nn.Module):
    def __init__(self):
        super(ResNet50, self).__init__()
        self.input_stem = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=(7, 7), stride=2, padding=3, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True)
        )
        self.stage_1 = nn.Sequential(
            nn.MaxPool2d(kernel_size=(3, 3), stride=2, padding=1),
            Bottleneck(64, 64, 256, 1),
            Bottleneck(256, 64, 256),
            Bottleneck(256, 64, 256)
        )
        self.stage_2 = nn.Sequential(
            Bottleneck(256, 128, 512, 2),
            Bottleneck(512, 128, 512),
            Bottleneck(512, 128, 512),
            Bottleneck(512, 128, 512)
        )
        self.stage_3 = nn.Sequential(
            Bottleneck(512, 256, 1024, 2),
            Bottleneck(1024, 256, 1024),
            Bottleneck(1024, 256, 1024),
            Bottleneck(1024, 256, 1024),
            Bottleneck(1024, 256, 1024),
            Bottleneck(1024, 256, 1024)
        )
        self.stage_4 = nn.Sequential(
            Bottleneck(1024, 512, 2048, 2),
            Bottleneck(2048, 512, 2048),
            Bottleneck(2048, 512, 2048)
        )
        self.avgpool = nn.AdaptiveAvgPool2d(output_size=(1, 1))
        self.fc = nn.Linear(2048, 5)

    def forward(self, x):
        out = self.input_stem(x)
        out = self.stage_1(out)
        out = self.stage_2(out)
        out = self.stage_3(out)
        out = self.stage_4(out)
        out = self.avgpool(out)
        out = self.fc(out.reshape(out.shape[0], -1))
        return out
```

3 Experimental Results

For our experiment in the Table 1, we tried several combinations in each ResNet. All of methods use Cross Entropy as loss function and Stochastic Gradient Decent (SGD) as optimizer which set momentum to 0.9. For data augmentation, all the experiments using 20 degrees of rotation randomly and both horizontal and vertical flipping. In (b), we tried *RandomAutocontrast*, *RandomAdjustSharpness* and *RandomEqualize*. And in (d), we tried *ColorJitter* to change brightness, saturation and hue, but the performance did not improve, instead time became double. Our final best accuracy is **83.07%** in (g) and **83.13%** in (D). Screenshot shows in Fig.3.

	epochs	batch size	learning rate	weight decay	Acc.(%)
*ResNet*18 (a)	20	16	0.001	5e-04	81.72%
*ResNet*18 (b)	20	16	0.001	5e-04	81.00%
*ResNet*18 (c)	(a) + 20	16	0.001	5e-04	80.81%
*ResNet*18 (d)	20	16	0.001	5e-04	81.58%
*ResNet*18 (e)	(a) + 5	64	0.001	5e-03	**82.55%**
*ResNet*18 (f)	(a) + 5	64	0.001	1e-03	**82.72%**
*ResNet*18 (g)	(a) + 5	64	0.0005(slr)	5e-03	**83.07%**
*ResNet*50 (B)	10	16	0.001	5e-04	**82.88%**
*ResNet*50 (C)	(B) + 3	16	0.0002(slr)	5e-03	**83.04%**
*ResNet*50 (D)	(B) + 5	16	0.0002(slr)	5e-03	**83.13%**

Table 1: Comparison in different combination experiments.

3.1 The highest testing accuracy

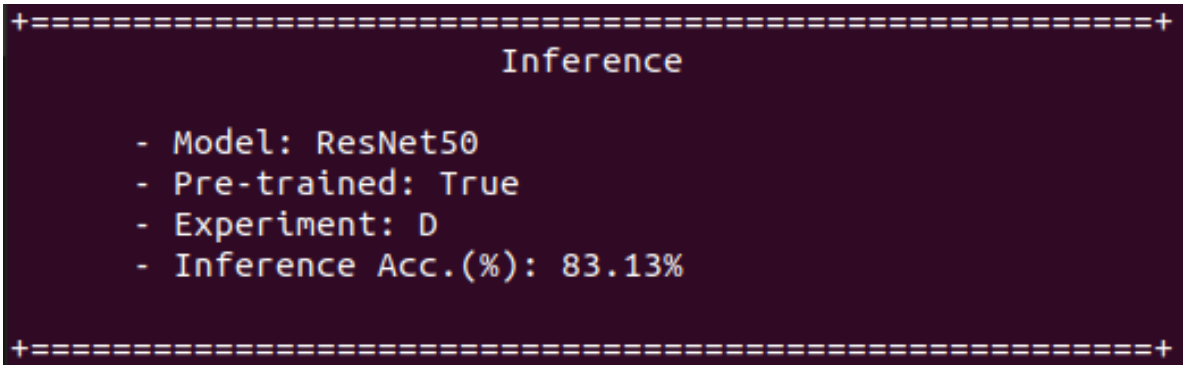


Figure 3: Screenshot for our best result.

	Non-pretrained	Pretrained
ResNet18	76.77% (c)	**83.07%** (g)
ResNet50	72.91% (D)	**83.13%** (D)

Table 2: Performance Comparison in ResNet18 and ResNet50.

3.2 Comparison figures

For our best result, we used (a) and (B) in Table 1 as baseline for ResNet18 and ResNet50 respectively. As you can see the left picture in the Fig.4 and Fig.5, the approach of learning got overfitting which testing accuracy maintain around 81% (sometimes achieved 82% with lucky, but it is not stable). We analyzed the loss and adjusted our strategy, the experiment (g) and (D) is the final result which led us to the best accuracy in both networks. In our test experience, the performance is no big different between ResNet18 and ResNet50. But in (B), we had already achieved baseline 82% with just 20 epochs.

3.2.1 ResNet18

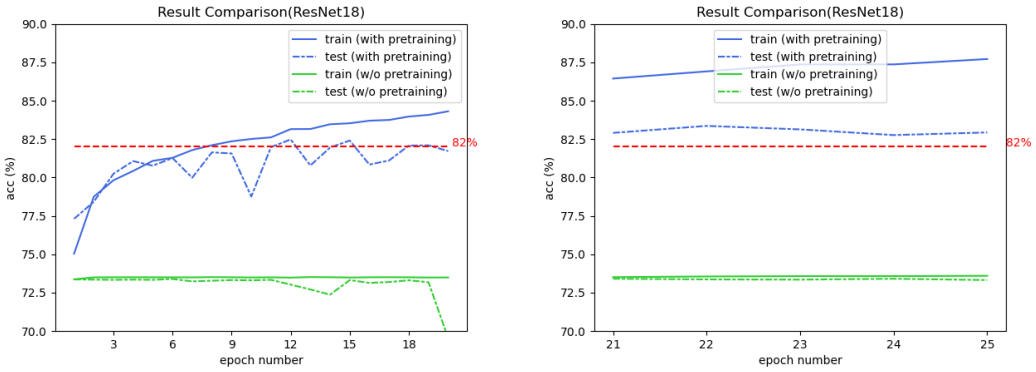


Figure 4: An illustration of the best result with ResNet18. $(left : (a), right : (g))$

3.2.2 ResNet50

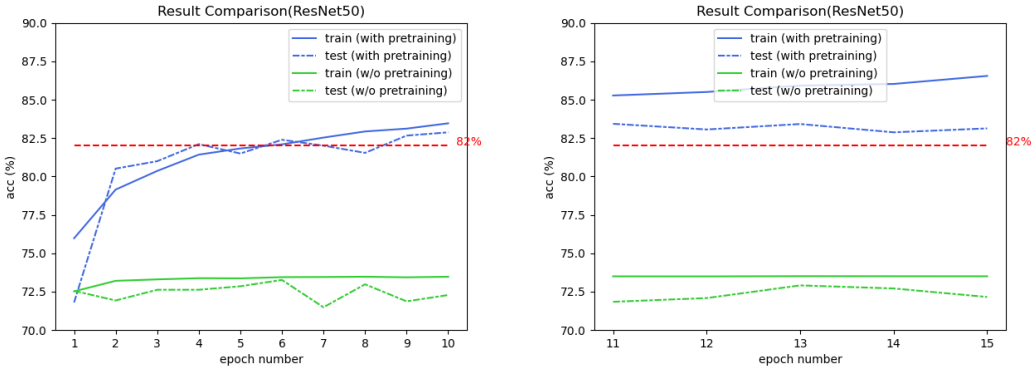
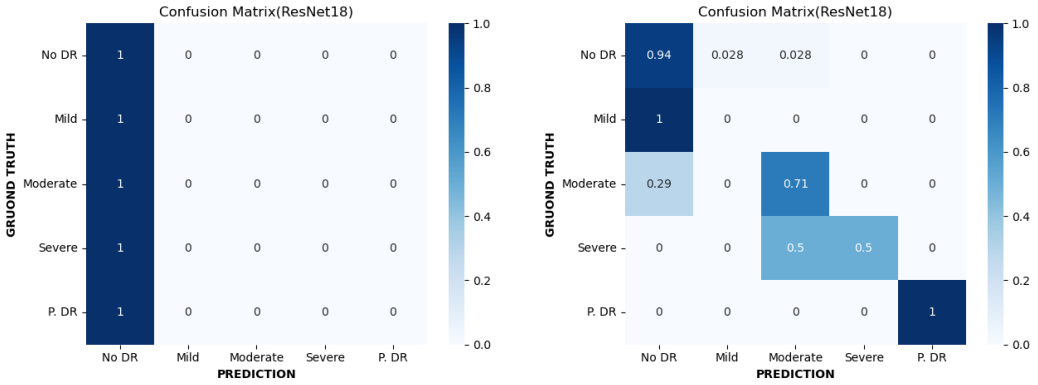


Figure 5: An illustration of the best result with ResNet50. $(left : (B), right : (D))$

3.3 Confusion Matrix

Confusion Matrix is a specific table layout that allows visualization of the performance of an algorithm, typically a supervised learning one (in unsupervised learning it is usually called a matching matrix). Each row of the matrix represents the instances in an actual class while each column represents the instances in a predicted class, or vice versa – both variants are found in the literature. The name stems from the fact that it makes it easy to see whether the system is confusing two classes (i.e. commonly mislabeling one as another). In our cases, confusion matrix is a 5x5 matrix because of the 5 classes. x-axis is the prediction from our model and y-axis is the ground truth. Since our best results are all based on pretrained model, you can see that it really learned something in the matrix right below.

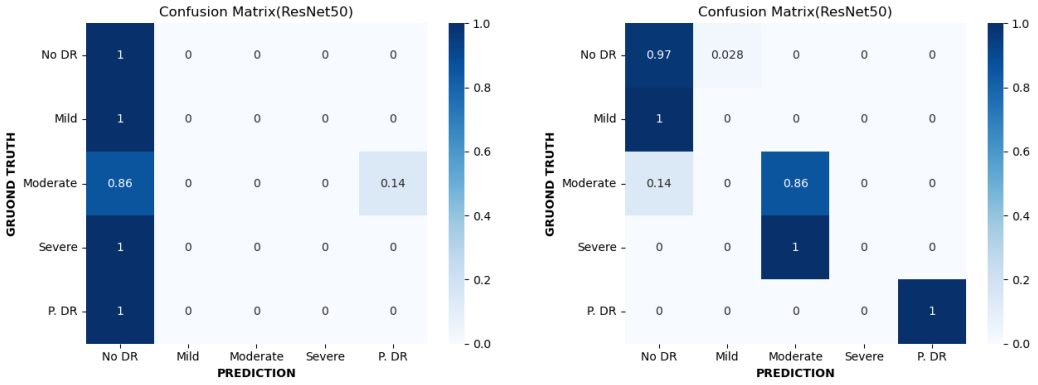
3.3.1 ResNet18



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

Figure 6: Illustrates the confusion matrix in ResNet18's best experiment (g)

3.3.2 ResNet50



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

Figure 7: Illustrates the confusion matrix in ResNet50's best experiment (D)

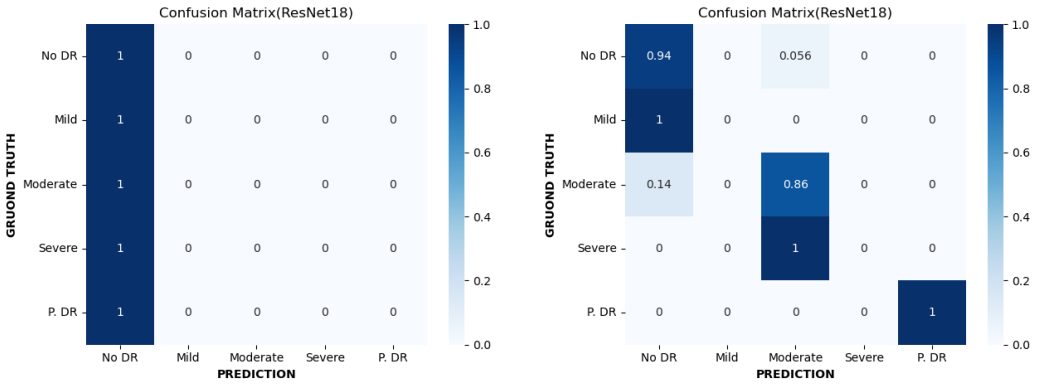
4 Discussion

4.1 Overfitting Problem

Our experiments all facing overfitting problem. We found out that the testing accuracy maintain around 81% after 10 epochs that training accuracy still going up with decreasing training loss. In that case, our first adjustment is increasing weight decay which is a regularization method to tackle the overfitting problem and set batch size to 64 with ResNet18 model. The experiment shows in Table 6.

	batch size	weight decay	w Pretrained Acc.(%)
Experiment (e)	64	5e-03	**82.55%**
Experiment (f)	64	1e-03	**82.72%**

Table 3: Performance Comparison in different weight decay values.



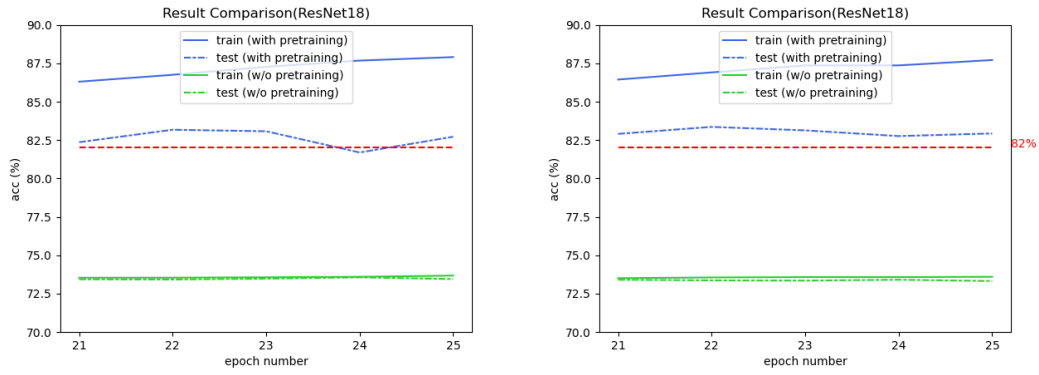
(a) w/o pretrained (b) with pretrained

Figure 8: Illustrates the confusion matrix in experiment (f)

And then, we found out that the loss stopped going down in both (e) and (f). The accuracy is stable around 82% which had already beat our baseline. We still want to see if there is any possible to get more improvement. So we tried to reset the learning rate to 0.0005 which is half in original one and added a Exponential learning rate as scheduler with $\gamma = 0.96$. Then we achieved our best: **83.07%**.

	learning rate	weight decay	w Pretrained Acc.(%)
Experiment (e)	0.001	5e-03	**82.55%**
Experiment (f)	0.001	1e-03	**82.72%**
Experiment (g)	0.0005 (w scheduler)	5e-03	**83.07%**

Table 4: Performance Comparison in different learning rate strategies.



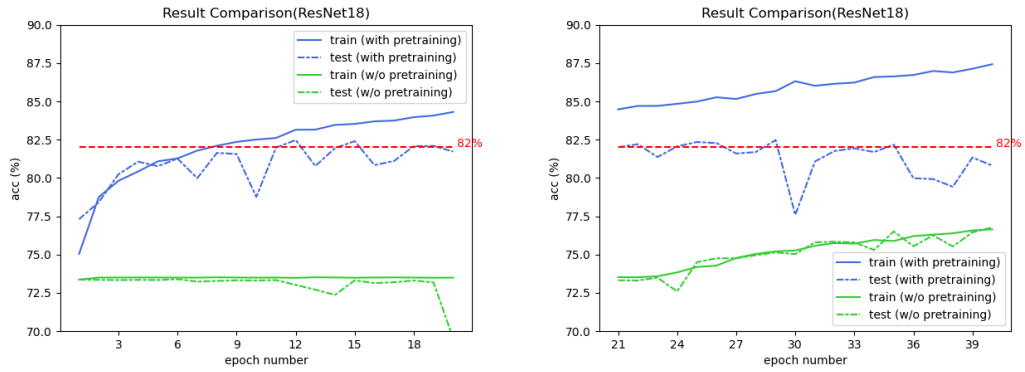
(a) Exp.(f) with 0.001 LR. (b) Exp.(g) with 0.0005 LR and scheduler.

Figure 9: Illustrates learning curve in experiment (f) and (g).

4.2 Data Imbalance

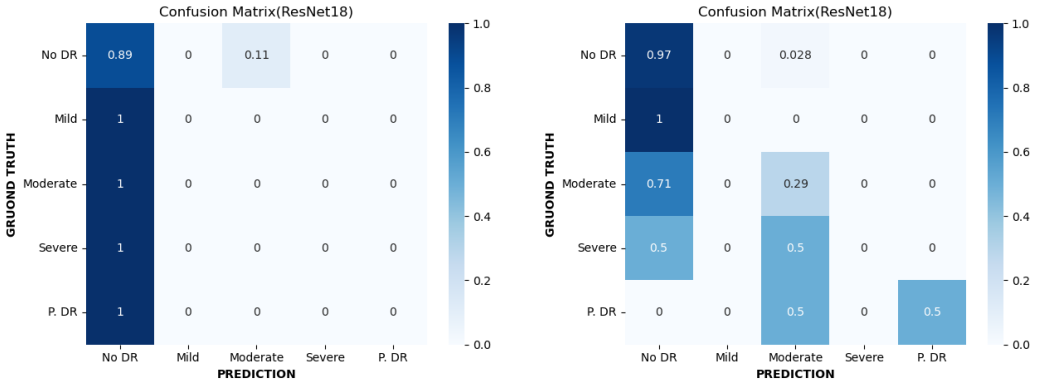
Since our data are extremely imbalance, we tried to tackle this problem. The percentage of each class is 73.5%, 7%, 15%, 2.5% and 2% in No DR, Mild, Moderate, Severe and Proliferative DR respectively. That is the reason which every experiment without pretrained all got around 73%. Because the model did not learn anything, you will get 73% accuracy if you guess all prediction to class No DR. About this, we tried to add more epochs for training to see if the model learned something. In Fig.10 (green

lines), you can see that after adding 20 more epochs in experiment (a), the model accuracy increasing and we can also observe the change on confusion matrix in Fig.11.



(a) 20 epochs. (b) 40 epochs.

Figure 10: Illustrates how training time affect the model by adding epochs.

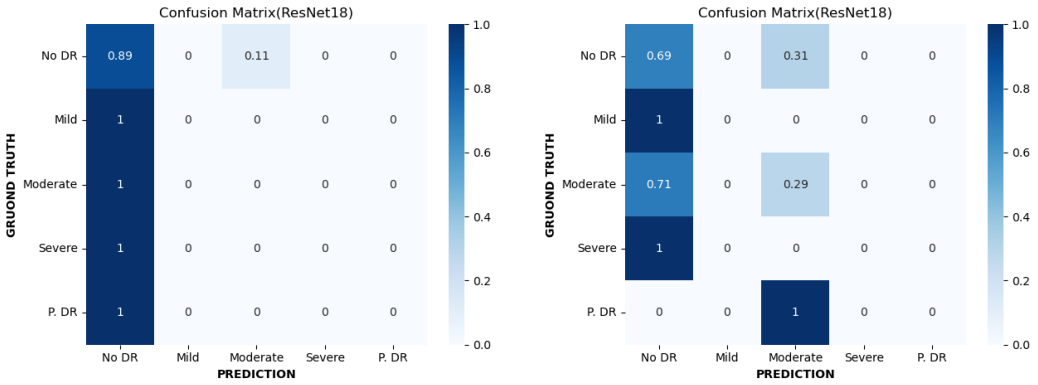


(a) 20 epochs w/o pretrained (b) 40 epochs w/o pretrained

Figure 11: Illustrates the confusion matrix between 20 and 40 epochs.

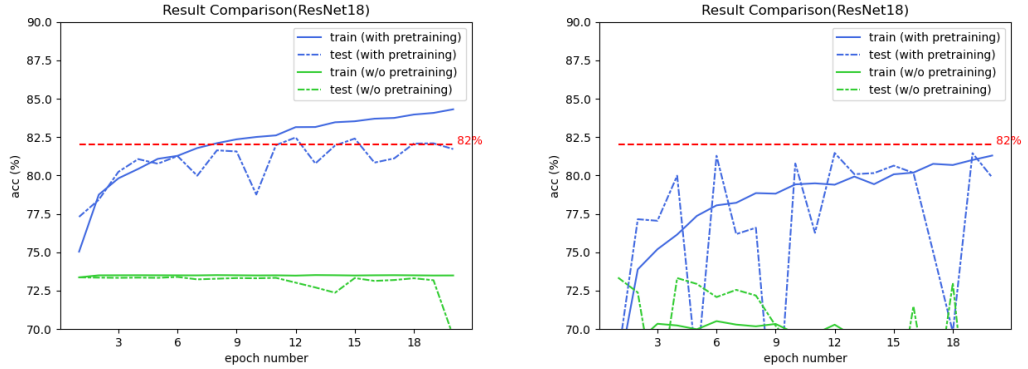
And then we also attempt to tackle data imbalance problem from loss function, so we adjusted the weights to each class by setting weight parameter to Cross Entropy. *sample* is a list about the number of each class and if the scale of class is small, the weight will be more bigger and so on. Then we normalized the value which is said will be better and turned them to float tensor. The code shows in below. However, the results were not good. We did not know the main reason, maybe it also is relative to the number of training time. But we are not allowed to train 100 or even more epochs because of the deadline. Although the result did not perform well, it also shows that the model are learning something on confusion matrix in Fig.12.

```
'''
No DR: 20656 (73.5%)
Mild: 1955 (7%)
Moderate: 4210 (15%)
Severe: 698 (2.5%)
Proliferative DR: 581 (2%)
'''
sample = [20656, 1955, 4210, 698, 581]
nweights = [1 - (x / sum(sample)) for x in sample]
nweights = torch.FloatTensor(nweights).to(device)
criterion = nn.CrossEntropyLoss(weight=nweights)
```



(a) 20 epochs. (b) 20 epochs with weight adjustment.

Figure 12: Illustrates the confusion matrix with or w/o weight adjustment.



(a) 20 epochs. (b) 20 epochs with weight adjustment.

Figure 13: Accuracy curve between weight adjustment or not.

	epoch	w Pretrained Acc.(%)	w/o Pretrained Acc.(%)
w/o adjustment	20	81.72%	69.54%
w/o adjustment	40	80.81%	**76.77%**
with adjustment	20	79.87%	61.38%

Table 5: Performance Comparison in different learning rate strategies.

4.3 Augmentation

The last discussion is about data augmentation. Even though augmentation is useful to increase performance, however the selection of augmentation shall be carefully chosen. Center crop is not really useful in this case. For center crop, we thought that the center of image do not have sufficient information of the disease feature since we had already resized our data into 512x512. For the flipping augmentation in both horizontal and vertical, the performance did not grow significantly due to the shape of retina. However, when we added 20 degrees of rotation, the combination of flipping and rotation led us to our best accuracy. We also made a experiment about color adjustment which we adjusted the brightness, contrast, saturation and hue. For our intuitive thoughts, color adjustment will highlight the retinopathy. But it seems like the results did not support our ideas. Same conclusion in other adjustment like sharpness, histogram equalization...etc.

Figure 14: Original 512x512 images.

Figure 15: Horizontal and Vertical flipping.

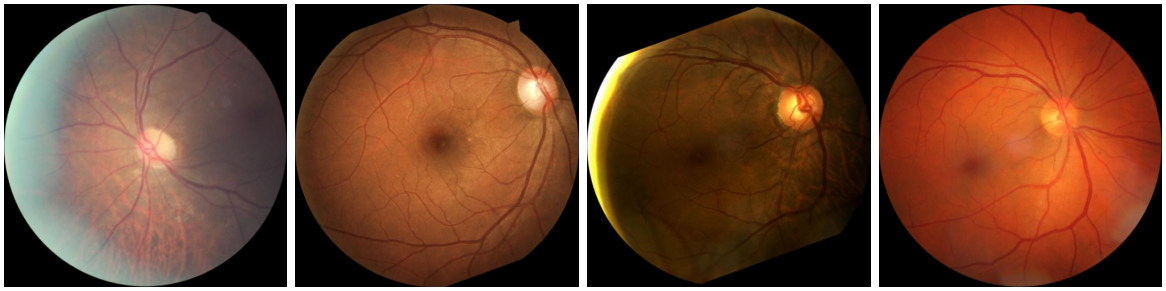


Figure 16: 20 degrees of rotation.

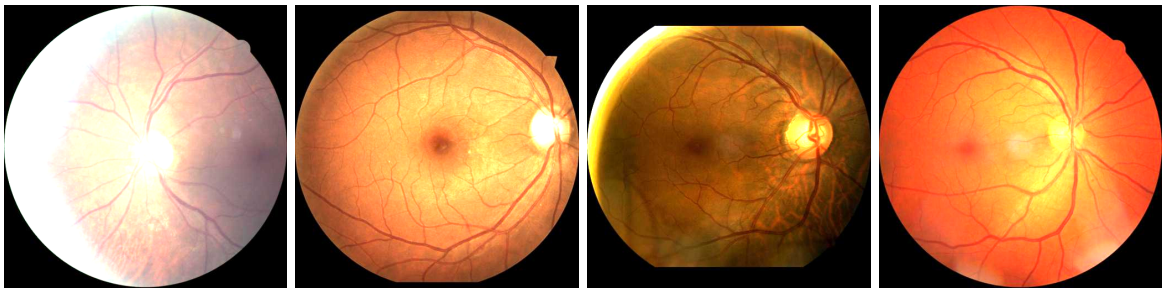


Figure 17: Color adjustment.

	Flipping	Rotation	Color Adjustment	w Pretrained Acc.(%)
Experiment 1	✓	✗	✗	81.03%
Experiment 2	✓	✓	✗	**81.72%**
Experiment 3	✓	✓	✓	81.58%

Table 6: Performance Comparison in different data augmentation strategies.

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.