Summer 2022 Deep Learning

Report of Lab #3

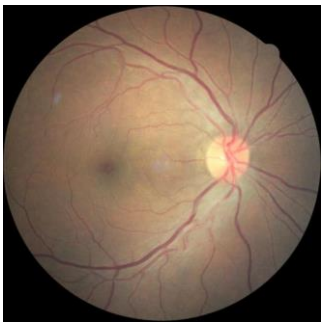
Student name: 劉子齊 Jonathan
Student ID: 311605004

Part 1: Introduction

In lab3, we need to implement ResNet18 and ResNet50 to analysis Diabetic Retinopathy. Also, we need to implement our own DataLoader through the PyTorch framework. In this lab, our goal is to classify the grading of the diabetic retinopathy through the ResNet architecture. Besides, we need to compare the performances of original ResNets and pretrained ResNets. For last, we need to output the result as accuracy curves and confusion matrixes.

Before moving on to the next part, I would to talk about the dataset I made use of in this lab. The dataset incudes thousands of pictures of human retina, which include labels that tells the level of diabetes from 0 to 4. The following is how the dataset looks. As we can see below, the pictures are directional. We can easily tell whether the picture is from the retina of right eye or from left eye. Hence, there might be some constraints when conducting data augmentation, and I'll mention this in part 2-B, which is about the detail of my Dataloader.

Picture of Left Eye



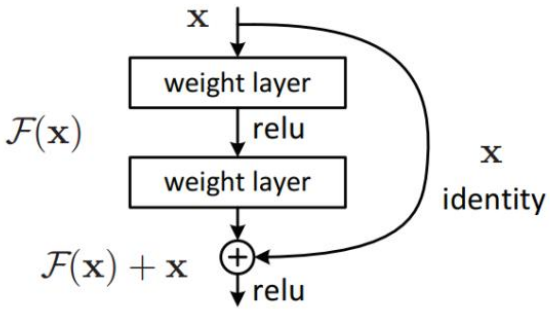
Picture of Right Eye

Part 2: Experiment set up

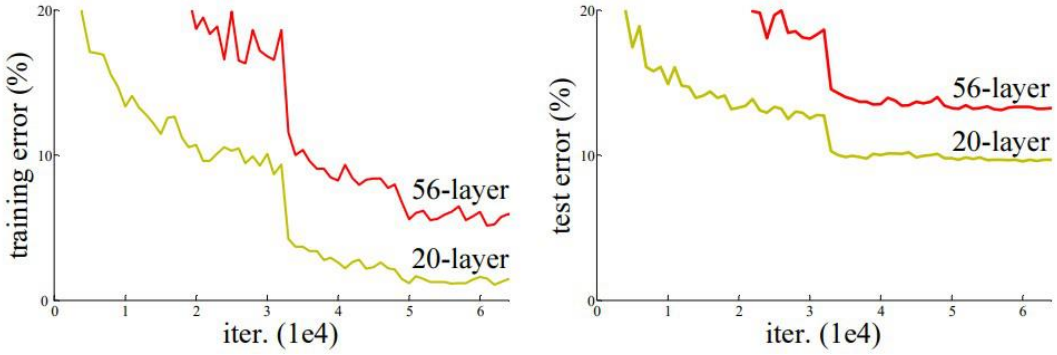
Part 2-A: Details of the ResNets

ResNet, also known as Residual Network, is a type of neural network introduced in 2015. The biggest difference between ResNet and other neural networks is the residual connection.

The residual network skips some layers to provide an alternative for data to reach the later regions of the neural network. If we feed the input data into the model today, just like the following function, $y = f(x)$, we will get the corresponding output. In the residual block, we will add input x to the output, which is $y = f(x) + x$. By this we can make the weight layers of the neural network to learn the residual between the input and output.



Besides, the residual network also gives model the ability to train much deeper networks. When we try to increase the layers of the neural network, we may also encounter the problem that the gradient to become 0 or too large, which is also known as "vanishing gradient". This problem might also raise the training error of the network with more layers, just like the phenomenon shown in the figure below. Through the skip connection of residual network, we can solve the problem of vanishing gradient.



The following is how I implemented the Resnet architecture. First, I constructed the two residual networks, the Basic Block and the Bottleneck Block, which correspondingly belongs to ResNet18 and ResNet50.

```python
class BasicBlock(nn.Module):
    expansion: int = 1

    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super().__init__()

        self.block = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(out_channels)
        )

        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample

    def forward(self, x):
        residual = x
        output = self.block(x)

        if self.downsample is not None:
            residual = self.downsample(x)

        output += residual
        output = self.relu(output)

        return output
```

```python
class Bottleneck(nn.Module):
    expansion: int = 4

    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super().__init__()

        self.bottleneck = nn.Sequential(
            nn.Conv2d(inplanes, planes, kernel_size=1, bias=False),
            nn.BatchNorm2d(planes),
            nn.ReLU(inplace=True),
            nn.Conv2d(planes, planes, kernel_size=3, stride=stride, padding=1, bias=False),
            nn.BatchNorm2d(planes),
            nn.ReLU(inplace=True),
            nn.Conv2d(planes, planes * self.expansion, kernel_size=1, bias=False),
            nn.BatchNorm2d(planes * self.expansion)
        )

        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        residual = x

        output = self.bottleneck(x)

        if self.downsample is not None:
            residual = self.downsample(x)

        output += residual
        output = self.relu(output)

        return output
```

After the residual networks, I build a ResNet class which took the source code of the resnet.py of torchvision as base and made the inplane modifiable according to the image channels. By passing the corresponding layer and num of classes, I can get both ResNet18 and ResNet50 through this class.

```
class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes, image_channels=64):
        super().__init__()

        self.inplanes = image_channels

        self.layer0 = nn.Sequential(
            nn.Conv2d(3, self.inplanes, kernel_size=7, stride=2, padding=3, bias=False),
            nn.BatchNorm2d(self.inplanes),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        )

        self.layer1 = self._make_layer(block, 64, layers[0], stride=1)
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))

        self.fc = nn.Linear(512 * block.expansion, num_classes)

    def _make_layer(self, block, planes, blocks, stride=1):
        downsample = None

        if stride != 1 or self.inplanes != planes * block.expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.inplanes, planes * block.expansion, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(planes * block.expansion)
            )

        layers = []
        layers.append(block(self.inplanes, planes, stride, downsample))
        self.inplanes = planes * block.expansion

        for i in range(1, blocks):
            layers.append(block(self.inplanes, planes))

        return nn.Sequential(*layers)

    def forward(self, x):

        x = self.layer0(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        return x
```

Pretrained Resnets also played an important role in this lab. The following figure is how I took all the layers except the last layer from the pretrained model, and reinitialize the specific layer.

```python
class PretrainedResNet(nn.Module):
    def __init__(self, num_classes, num_layers):
        super(PretrainedResNet, self).__init__()

        pretrained_model = torchvision.models.__dict__['resnet{}'.format(num_layers)](pretrained=True)

        self.conv1 = pretrained_model._modules['conv1']
        self.bn1 = pretrained_model._modules['bn1']
        self.relu = pretrained_model._modules['relu']
        self.maxpool = pretrained_model._modules['maxpool']

        self.layer1 = pretrained_model._modules['layer1']
        self.layer2 = pretrained_model._modules['layer2']
        self.layer3 = pretrained_model._modules['layer3']
        self.layer4 = pretrained_model._modules['layer4']

        self.avgpool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Linear(pretrained_model._modules['fc'].in_features, num_classes)

        del pretrained_model

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        return x
```

Finally, the functions in the figure below is how I get the corresponding ResNet through the train() function. I can determine whether to call the pretrained function or not here through a flag. Also, as I mentioned before, through passing the corresponding layer and num of classes, I can get both ResNet18 and ResNet50 through this class. The following is where I pass them in.

```python
def ResNet18(pretrained=False):
    model = ResNet(BasicBlock, layers=[2, 2, 2, 2], num_classes=5)

    if pretrained == True:
       model =  PretrainedResNet(num_classes=5, num_layers=18)

    return model


def ResNet50(pretrained=False):
    model = ResNet(Bottleneck, layers=[3, 4, 6, 3], num_classes=5)

    if pretrained == True:
        model =  PretrainedResNet(num_classes=5, num_layers=50)

    return model
```

Part 2-B: Details of the Dataloader

The figure below is how I implemented the Dataloader. First, I get the images from the image path through PIL.Image. Then I can chose to do some augmentation to the image or just simply conduct ToTensor() through the augmentation variable.

Let's say we chose to augment the image, it will randomly vertical flip the image in the probability of 0.5. Then it will randomly rotate the image in the degree of −30° to 30°. Finally, I convert the PIL image which is in the size of [512, 512, 3] into tensor [3, 512, 512].

The reason why I only applied the RandonVerticalFlip() here is I found the data is directional, which we can find the white circle in the retina on the left if the picture is taken from the left eye. I consider it a bit useless if we apply horizontal flip again, since we got both picture from the left eye and right eye already.

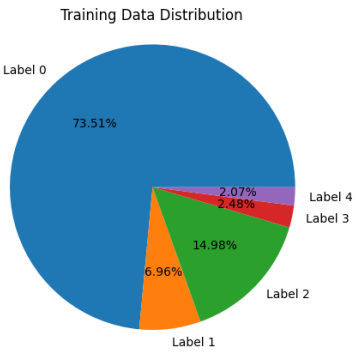
Finally, after all the transformations, I return the image data and label for further process.

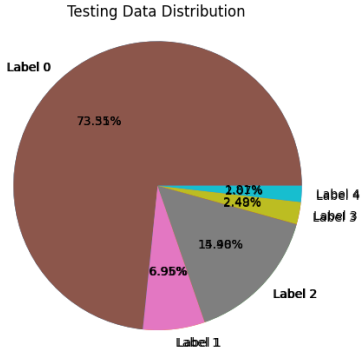
```
class RetinopathyLoader(data.Dataset):
    def __init__(self, root, mode, augmentation=None):

        """...
        self.root = root
        self.img_name, self.label = getData(mode)
        self.mode = mode

        if augmentation == 1:
            print("Data Augmentation Activated!!!")
            self.transformation = transforms.Compose([transforms.RandomVerticalFlip(p=0.5),
                                                      transforms.RandomRotation(30),
                                                      transforms.ToTensor()])
        else:
            self.transformation = transforms.Compose([transforms.ToTensor()])

        print("> Found %d images..." % (len(self.img_name)))

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

    def __getitem__(self, index):
        """something you should implement here"""

        """...

        img_path = os.path.join(self.root, self.img_name[index] + ".jpeg")
        img = Image.open(img_path)

        # print("====== pixel value ========")
        # pix_val = list(img.getdata())
        # print(pix_val)

        img = self.transformation(img)

        label = self.label[index]

        return img, label
```

Part 2-C: Evaluation through the confusion matrix

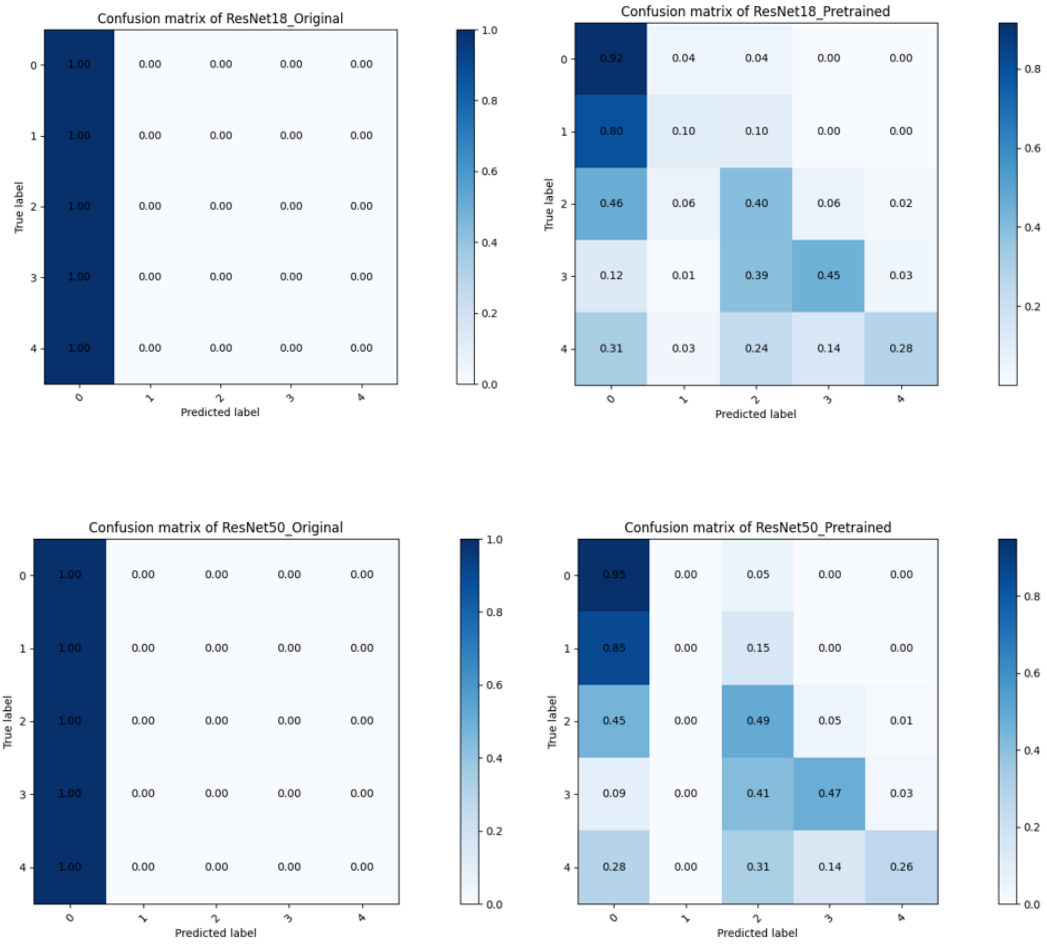
The following is the confusion matrix of the 1st experiment. In this experiment, I set all the parameters according to the parameters provided by the spec. Through the confusion below, we can find the origin models, which is also known as the "not" pretrained models, classified all the images in the testing dataset to label "0". This made me so confused, so then I checked the distribution of the dataset. Through printing out the nums of each label, I found the dataset is severely biased, which is shown in the figure below.

```
========== Training Data Distribution ==========
num of label  0  :  20655
num of label  1  :  1955
num of label  2  :  4210
num of label  3  :  698
num of label  4  :  581
```





We can easily tell that over 70% of the dataset are label "0" data. I think this might be caused by the way they get the images. When they were collecting data, I think they just randomly pick respondents and took picture of their retinas. Therefore, among all respondents, the number of people with diabetes was relatively low, which made the dataset to be biased.Besides, it is apparent that pretrained models performed better. However, except for label "0", most of them didn't exceed 50%.



Part 3: Experimental results

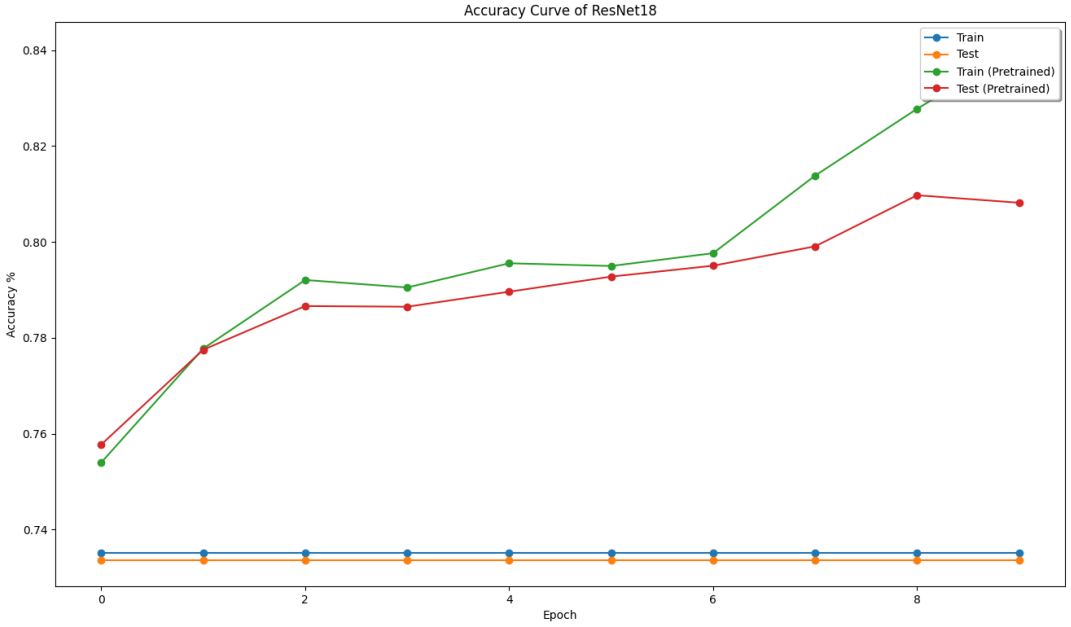
Part 3-A: Highest testing accuracy

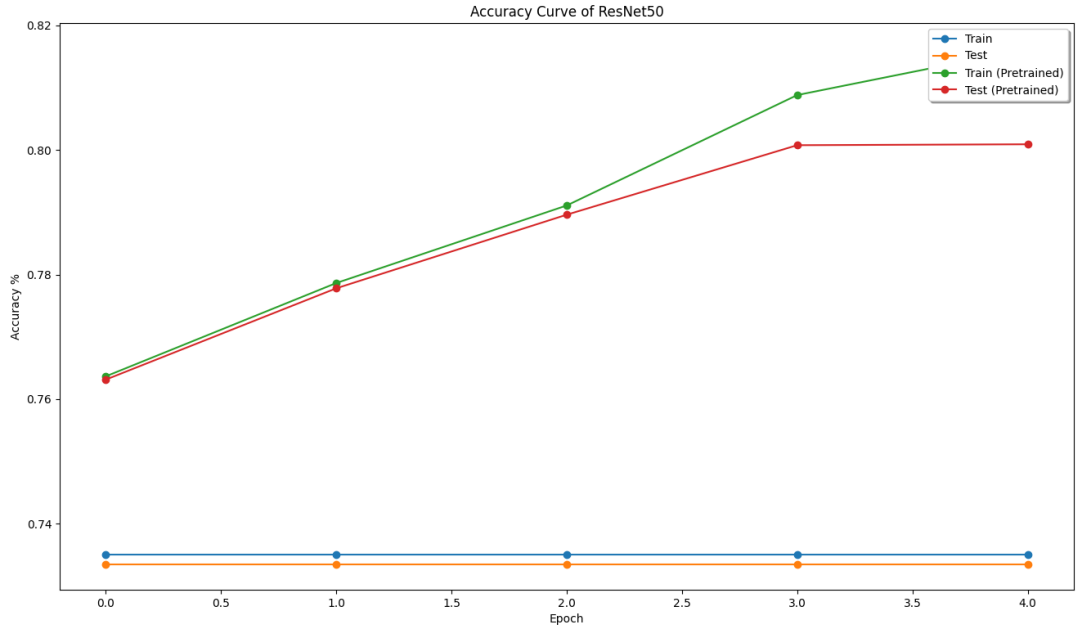
The following chart recorded the highest testing accuracies of the experiments I conducted. Below the chart I'll explain the things I modified which made me got the corresponding accuracy. For the parameters I didn't mention, I followed the settings the spec recommended.

Model Name	Accuracy	What I modified
ResNet18	73.352 %	➢ Data Augmentation ➢ Loss Function Weight ➢ Set Weight Decay to 1e-3
ResNet50	80.972 %	
Pretrained ResNet18	73.532 %	
Pretrained ResNet50	80.092 %	

Part 3-B: Comparison figures

In this part, I'll show the comparison figures of the experiments which occurred the models with the best testing accuracy results I mentioned above.





Part 4: Discussion

Part 4-1: Does Data Augmentation Help?

In my experiments, I compared the result with data augmentation and the result without it, I found that it did make the accuracy curve to look "healthier".

For “heathier”, I meant that the training curve and the testing curve act more similar, not act like overfitted. To some extent, I think this proved that data augmentation did help to prevent model from overfitting.

Part 4-2: Potential Solution to the Biased Dataset

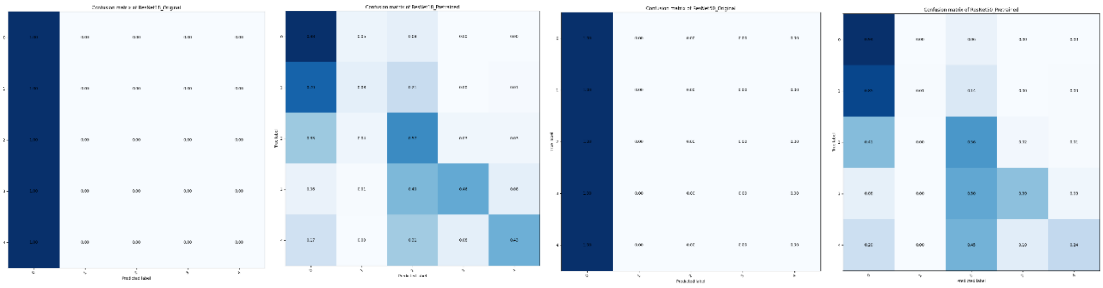
As I mentioned above, I found the data biased through printing out the total nums of each label. Then it came up to me that the loss function I made use of, which is the CrossEntropyLoss(), can actually kind of solve this problem. In the CrossEntropyLoss function, we can actually set the normalization weight, and it will time the loss with the weight given.

In the figure below is how I implemented the weight list. The get_train_label_nums() function is implemented in the dataloader.py. By subtracting the result of result of dividing x, which is the number of each label, by the total number of the whole training dataset, I can get the weight. Then I passed it to the loss function and wait for the result.

```
train_label_nums = get_train_lable_nums()
norm_train_weight = [1 - (x / sum(train_label_nums)) for x in train_label_nums]
norm_train_weight  = torch.FloatTensor(norm_train_weight).to(device)
```

```
loss_func = nn.CrossEntropyLoss(weight=norm_weight)
```

However, the result didn’t come out so well performed as I expected. Below is the confusion matrix which I obtained after modifying the CrossEntropyLoss function (with all the parameters obtained the same as the spec recommended), we can find that it is still strongly influenced by the label “0” data.



ResNet18 Pretrained Resnet18 ResNet50 Pretrained ResNet50

I started to think maybe I should attempt to change the data itself to make

the distribution balanced. Since even I attempt to solve the bias problem through the loss function, the model is still trained with the biased data. In this case, the model might be super strong at predicting label “0” data, but learned a little bit about other labels.

Hence, if I got a chance to work on biased dataset, I would try to save the model through balancing the dataset through duplicating data and balance the data distribution. I believe this will outcome some fabulous result.