DLP Lab3 Diabetic Retinopathy Detection

學號:311605015 姓名:張哲源

1. Introduction

- 分析糖尿病所引發視網膜病變
- 編寫自定義的 dataloader
- 實踐 ResNet18,ResNet50 等網路架構，並 pretrained
- 比較有無 pretrain 並可視化準確率
- 繪製混淆矩陣
- 這次資料集依照危險程度被分成五類

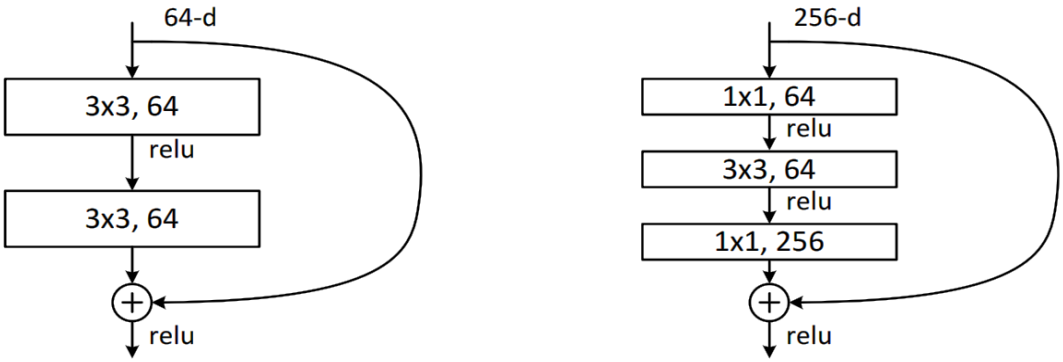
2. Experiment setups

2.1 The details of your model (ResNet)

ReseNet 有兩種建立方式，一種是用 pytorch 刻出建立其結構，另一種是從 torchvision 裡面的 model 這個模組將 ResNet 抓出來做使用，在下面我介紹我的 ResNet 建立方式

首先先建立一個 block，分別有兩種建立方式分別是 bottlenet block 和 basic block，而 BottleNetBlock 的好處是所需參數比 basic block 少但卻能達到一樣的結果

```
class BottleneckBlock(nn.Module):
    """
    output = (channels * 4, H, W) -> conv2d (1x1) -> (channels, H, W) -> conv2d (3x3) -> (channel
            -> conv2d (1x1) -> (channels * 4, H, W) + (channels * 4, H, W)
    """
    expansion: int = 4
    def __init__(self, in_channels: int, out_channels: int, stride: int = 1, down_sample= None):
        super(BottleneckBlock, self).__init__()
        external_channels = out_channels * self.expansion
        self.activation = nn.ReLU(inplace=True)
        self.block = nn.Sequential(
            nn.Conv2d(in_channels=in_channels,
                      out_channels=out_channels,
                      kernel_size=1,
                      bias=False),
            nn.BatchNorm2d(out_channels),
            self.activation,
            nn.Conv2d(in_channels=out_channels,
                      out_channels=out_channels,
                      kernel_size=3,
                      stride=stride,
                      padding=1,
                      bias=False),
            nn.BatchNorm2d(out_channels),
            self.activation,
            nn.Conv2d(in_channels=out_channels,
                      out_channels=external_channels,
                      kernel_size=1,
                      bias=False),
            nn.BatchNorm2d(external_channels),
        )
        self.down_sample = down_sample
```



Basic block/Bottlenet block 架構

而 ResNet 架構如下，本次作業要實作 ResNet18 和 ResNet50，例如 ResNet18 就是分別建立 4 個 2 層的 BottleNetblok，因此實踐程式碼較為複雜如下

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	[3×3, 64; 3×3, 64] ×2	[3×3, 64; 3×3, 64] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3
conv3_x	28×28	[3×3, 128; 3×3, 128] ×2	[3×3, 128; 3×3, 128] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×8
conv4_x	14×14	[3×3, 256; 3×3, 256] ×2	[3×3, 256; 3×3, 256] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×23	[1×1, 256; 3×3, 256; 1×1, 1024] ×36
conv5_x	7×7	[3×3, 512; 3×3, 512] ×2	[3×3, 512; 3×3, 512] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

```
class ResNet(nn.Module):
    def __init__(self, block, layers):
        super(ResNet, self).__init__()
        self.current_channels = 64
        self.conv_1 = nn.Sequential(
                nn.Conv2d(
                    in_channels=3,out_channels=64,kernel_size=7,stride=2,padding=3,bias=False),
                nn.BatchNorm2d(64),
                nn.ReLU(inplace=True),
                nn.MaxPool2d(kernel_size=3,
                             stride=2,
                             padding=1)
            )
        self.conv_2 = self.make_layer(block=block,
                                      num_of_blocks=layers[0],
                                      in_channels=64)
        self.conv_3 = self.make_layer(block=block,
                                      num_of_blocks=layers[1],
                                      in_channels=128,
                                      stride=2)
        self.conv_4 = self.make_layer(block=block,
                                      num_of_blocks=layers[2],
                                      in_channels=256,
                                      stride=2)
        self.conv_5 = self.make_layer(block=block,
                                      num_of_blocks=layers[3],
                                      in_channels=512,
                                      stride=2)
```

再設計 make_layer 函示將每層架構好

```
def make_layer(self,block,num_of_blocks,in_channels,stride=1):
    down_sample = None
    if ( stride != 1 or self.current_channels != in_channels * block.expansion):
        down_sample = nn.Sequential(
            nn.Conv2d(in_channels=self.current_channels,
                      out_channels=in_channels * block.expansion,
                      kernel_size=1,
                      stride=stride,
                      bias=False),
            nn.BatchNorm2d(in_channels * block.expansion),
        )
    layers = [
        block(in_channels=self.current_channels,
              out_channels=in_channels,
              stride=stride,
              down_sample=down_sample)
    ]
    self.current_channels = in_channels * block.expansion
    layers += [block(in_channels=self.current_channels, out_channels=in_channels) for _ in range(1, num_of_blocks)]
    return nn.Sequential(*layers)
```

2.2 The details of Dataloader

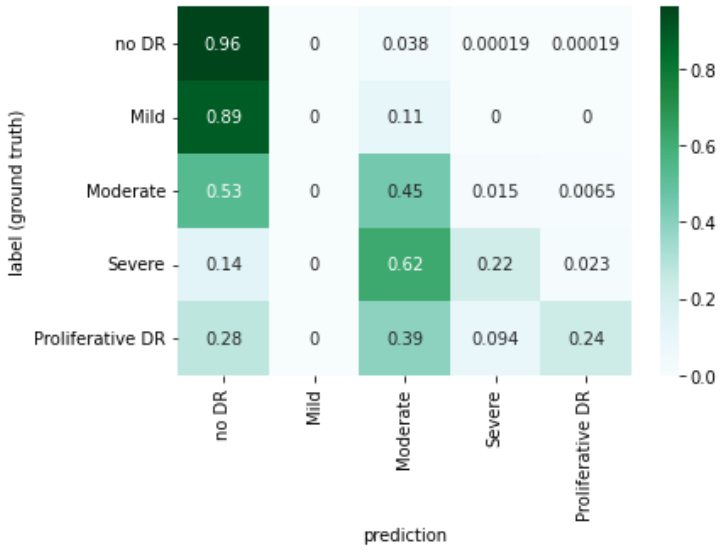
```
self.root = root
self.img_name, self.label = getData(mode)
self.mode = mode
trans_augmentation = []
if augmentation == True :
    trans_augmentation = trans_augmentation + augmentation

trans_augmentation = trans_augmentation + [transforms.ToTensor()]
self.trans_res = transforms.Compose(trans_augmentation)
```

Dataloader 會在__init__ function 中取得 images 所在的 folder，並讀取得到的 augmentation 方法做資料擴充，並印出數量

```
path = os.path.join(self.root, self.img_name[index] + '.jpeg')
img=Image.open(path)
img_data = self.trans_res(img)
label = self.label[index]
return img_data, label
```

在__getitem__函示則會根據取得的 index 取出對應的照片並透過初始化的擴充方式將資料擴充後，透過 PIL 讀取並轉成對應的 tensor，最後回傳轉換過後的 image 以及其對應的 label

2.3 Describing your evalution through the confusion matrix




```python
def calculate_confusion(model):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    test_loader=DataLoader(test_dataset,batch_size=8)
    model.to(device)
    y_pred=[]
    y_true=[]
    with torch.no_grad():
        for i ,(images,target) in enumerate(test_loader):
            torch.cuda.empty_cache()
            images=images.to(device)
            target=target.to(device)
            output=model(images)
            _, preds = torch.max(output, 1)
            y_pred.extend(preds.view(-1).detach().cpu().numpy())
            y_true.extend(target.view(-1).detach().cpu().numpy())
            print(i+1,'/',len(test_loader))
```

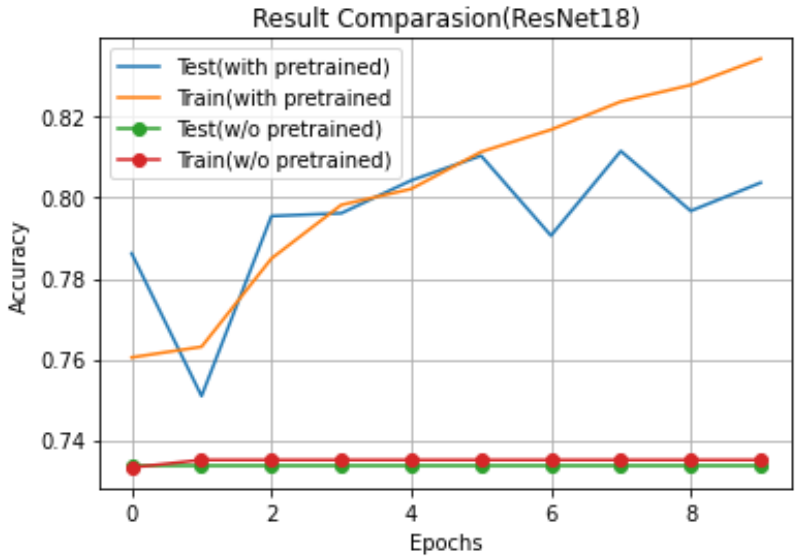
首先將預測結果與真實 label 存成一個矩陣，並利用 Skylearn 將結果算成混淆矩陣並 normalize，再利用 seaborn 與 pandas 等套件繪製成圖表，圖上的數字分別代表該類數量與準確率

```python
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
def plot_confusion_matrix(y_true,y_pred,title):
    torch.cuda.empty_cache()
    cf_matrix = confusion_matrix(y_true, y_pred,normalize='true')
    class_names = ['no DR', 'Mild', 'Moderate', 'Severe', 'Proliferative DR']
    df_cm = pd.DataFrame(cf_matrix, class_names, class_names)
    sns.heatmap(df_cm, annot=True, cmap='Oranges')
    plt.title(title)
    plt.xlabel("prediction")
    plt.ylabel("label (ground truth)")
plot_confusion_matrix(y_true,y_pred,'res18 w/o pretrained ')
```

3. Experimental results

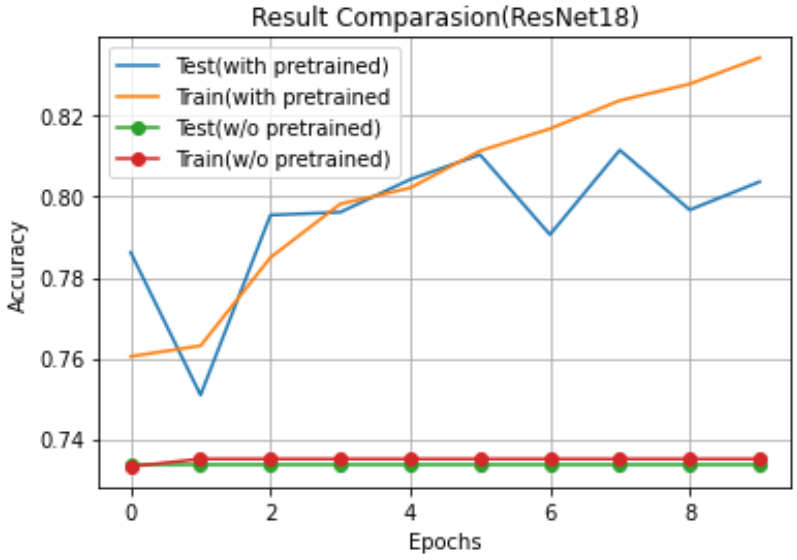
3.1 The highest testing accuracy

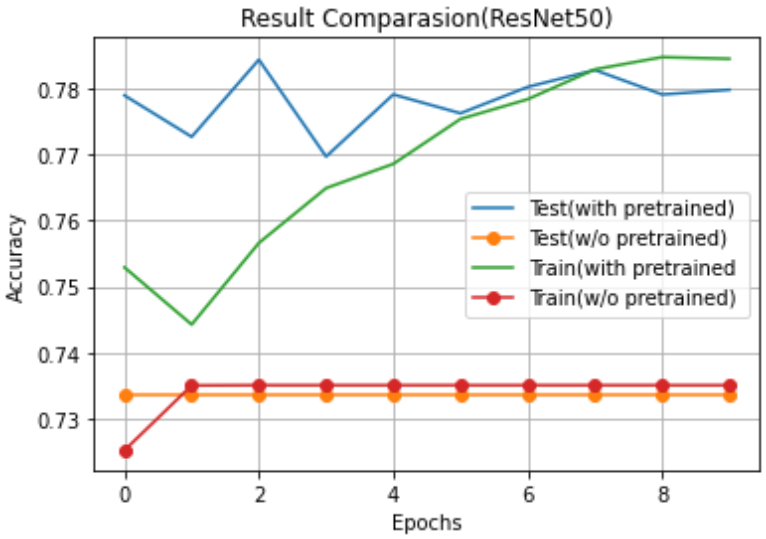
最高準確率我選用 pretrained 的 ResNet18 並將 batch_size 設為 16，epoch 為 10，並透過 augmentation 將資料集水平與垂直翻轉與隨機旋轉進行擴充，最後最高準確率可以到達 **81.34%**如下圖



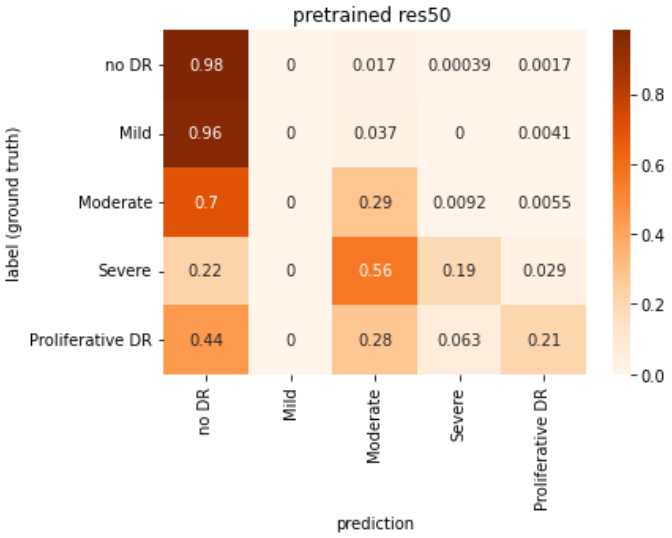
3.2 Comparison figures

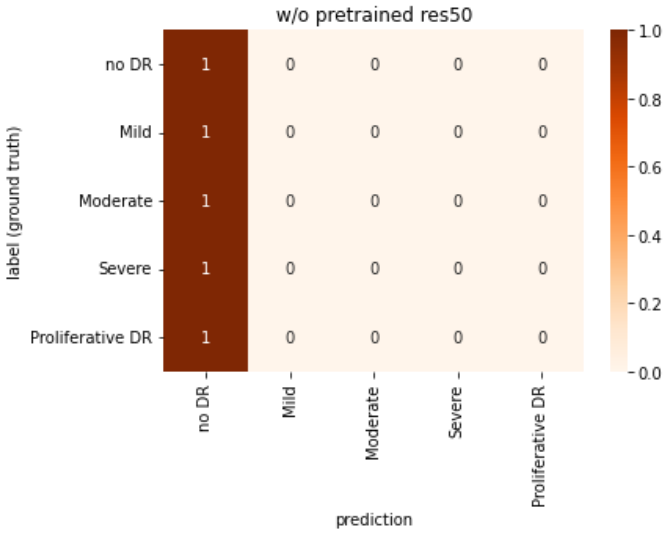
比較有無 pretrained 的 ResNet18，可以看出有 pretrained 過的 model，在提升 train accuracy 方面非常效果非常顯著，而沒有 pretrained 過的雖然 weight 都有改變，但修正不夠大，因此準確率沒有明顯上升。



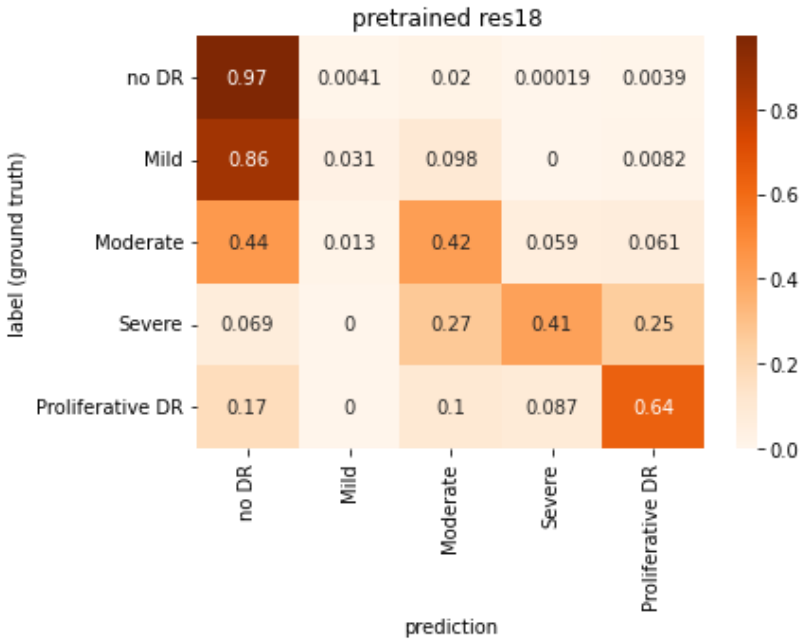


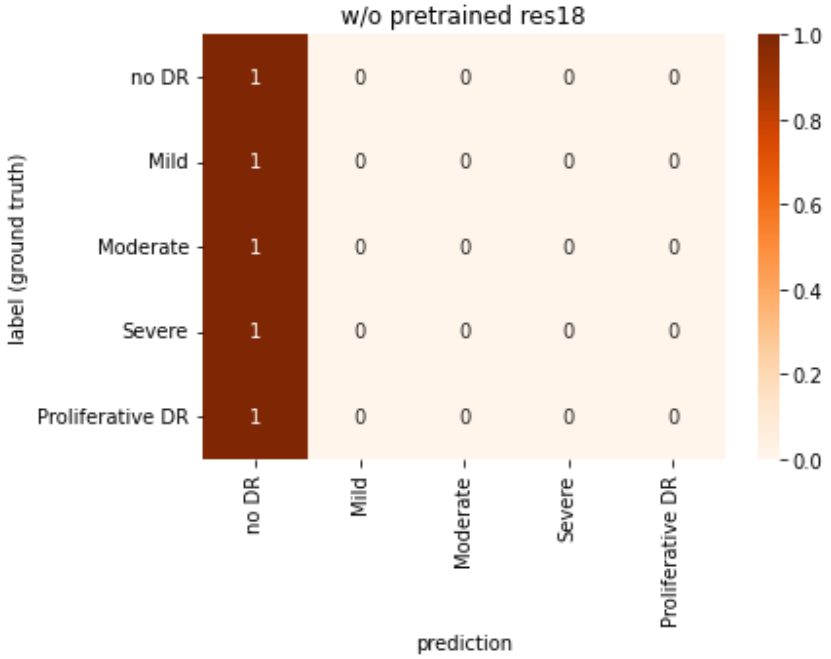
3.2 Confusion Matrix comparison





我們透過混淆矩陣觀察有 pre train 過的 model 再分類也比較合理，但除了第一類，其他種類的準確率均不高，在實際情況下還有很大改進空間，我認為很有可能是因為資料不太平均，像是第二類的資料數量較少，因此很難預測到第二類的答案，因此第二類的準確率極低，而沒有 pretrain 過的矩陣效果非常差，幾乎將所有東西都分到第一類





ResNet18 在有沒有 pretrained 也有類似的情況，第一類的分類結果還不錯但其他種類效果都非常差，沒有 pretrained 過的 model 準確率也沒有明顯上升並且很容易把結果全都分到第一類

4. Dicussion

4.1 Confusion Matrix

在機器學習或深度學習中，最常見的就是分類，但要去判斷分類的好不好，單憑準確率是不夠的，因此混淆矩陣(Confusion Matrix)的各項指標會被拿來參考。

混淆矩陣的 4 個元素(TP,TN,FP,FN)

TP(True Positive):正確預測成功的樣本。

TF(True Negative):正確預測錯誤的樣本

FP(False Positive):錯誤預測成正樣本，實際上是負樣本

FN(False Negative):錯誤預測成負樣本

而對混淆矩陣做 Normalization 則能看到每個種類的機率

混淆矩陣有三種主要計算的方式，第一種是準確率(Accuracy)計算公式為 AC=(TP+TN)/TP+FP+FN+TN)

第二種是精確率(Precision)=(TP)/(TP+FP)，代表判斷為陽性的樣本有機個是預測正確的。

第三種是召回率(Recall)=TP/(TP+FN) ，代表真實為陽性的樣本中有幾個是預測正確的。

4.2 Augmentation 資料增強

在本次作業我發現一種問題，因為大部分的資料都屬於第一類(NODN)，而第二種種類的資料很少，因此透過混淆矩陣看出很難預測第二種的成果，因此若能增加第二種資料的話，對於整體的預測也能更為精準。

而資料的擴充除了最基本的左右翻轉，若能做裁切旋轉也能有效提升準確率，本次實驗最高準確率也是在進行 Augmentation 下得到的