Lab3: Diabetic Retinopathy Detection

IOC/資科工碩

學號: 310551031

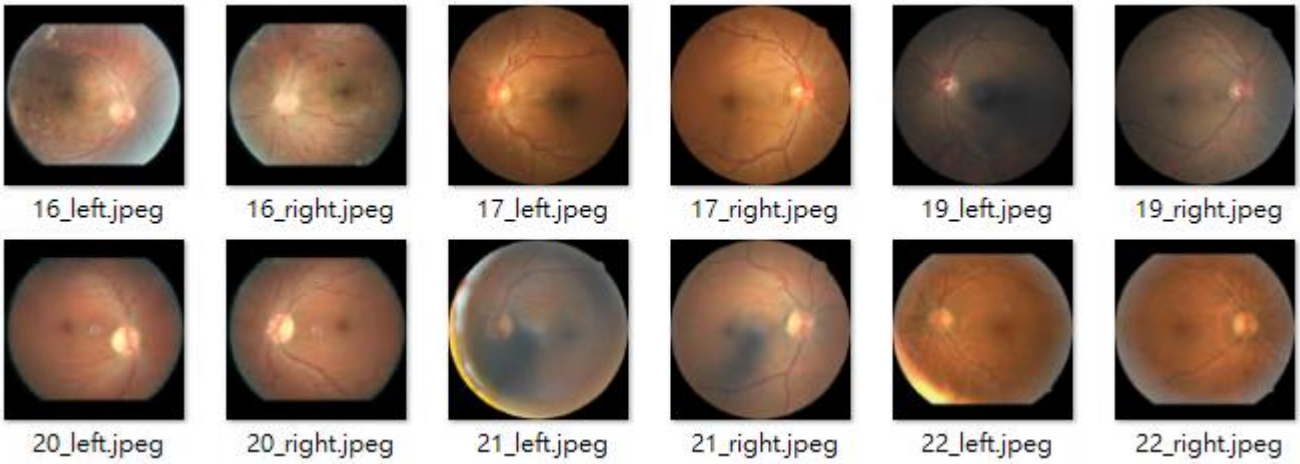
張皓雲

目錄

1. Introduction

A. 問題定義、達成目標與資料說明

本實驗注重在實作 ResNet18 與 ResNet50，並使用 pretrained model 及調整 non-pretrained model 內的超參數(hyper-parameters)，以進行 Diabetic Retinopathy Detection 的糖尿病眼球特徵分類。除此之外，本實驗希望嘗試將多個優化器(Optimizers)、資料前處理、資料增強與 freeze pretrained model 部份層等方法，加入實驗並討論調整結果。此次需要分類結果總共有五個，分別為 No DR、Mild、Moderate、Severe 及 Proliferative DR。**圖一**為本次實驗的資料，可以看到資料有左眼及右眼，兩眼資料對應的分類結果均相同。此外由於圖片大小不一，本實驗將會將這些圖片 resize 成統一的大小為 224x224。本次實驗使用的訓練資料集的圖片數量為 28100，測試資料集的圖片數量為 7025。最後**本實驗最好的結果為使用 ResNet50 的 pretrained model，測試資料集的結果為 77.48%**。



圖一、眼球分類資料示意圖

B. 實驗環境與檔案使用說明

B.1. 實驗環境

本次實驗使用的 python 套件有 os、Numpy、Pandas、Matplotlib、pkbar、tqdm 及 Pytorch 的各項子套件等。Numpy 的用途在儲存各項資料、數值的運算及統計等。Pandas 與 Collections 的用途則為進行數據結果的分析、儲存各項數據結果，例如: Confusion Matrix、儲存訓練的 accuracy 與 loss 數值等…。Matplotlib 的用途則是繪出 Training Accuracy 與 Testing Accuracy 隨著 epochs 變動的曲線圖、Training loss 與 Testing Accuracy 隨著 epochs 變動的曲線圖等….。套件的 Logo 如**圖二**、**圖三**、**圖四**與**圖五**所示。硬體設備、虛擬環境如**表一**所述。

表一、軟硬體實作環境

	實驗實作環境
處理器	**Intel(R) Core(TM) i7-6700 CPU@3.40GHz**

Python 版本	3.6.13
虛擬環境	Anaconda 2.0.4
顯示卡	GEFORCE RTX 2080 Ti
實作套件	os、Numpy、Matplotlib、Collection、Pandas、Pytorch
作業系統	Microsoft Windows 10

圖二、Pandas 套件 Logo

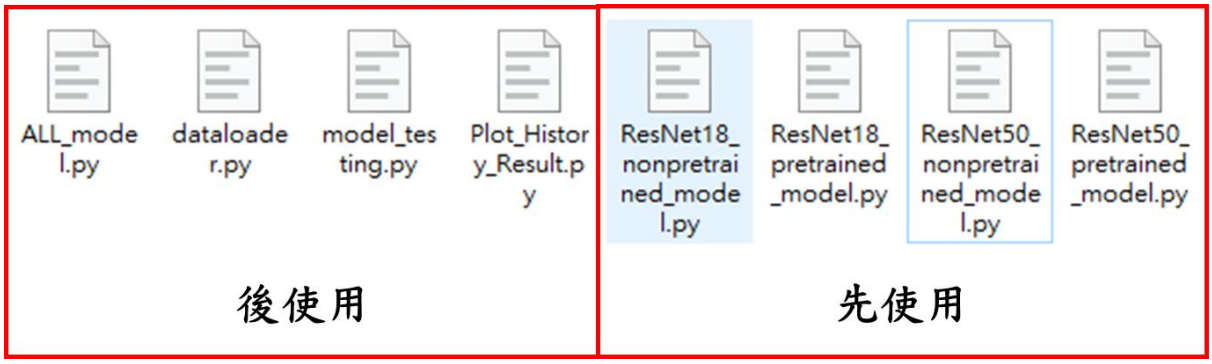
圖三、Matplotlib 套件 Logo

圖四、Numpy 套件 Logo

圖五、Pytorch 套件 Logo

B.2. 檔案使用說明

Source code 資料夾中總共有 dataloader.py、ResNet18_nonpretrained_model.py、ResNet18_pretrained_model.py、ResNet50_nonpretrained_model.py、ResNet50_pretrained_model.py、Plot_History_Result、ALL_model.py 及 model_testing.py 等八種檔案。使用這些檔案前，如需要跑新的模型權重及 acc 或 loss 曲線圖，請在檔案目錄下創建 history_csv 與 model_weight 資料夾，檔案使用順序如**圖六**所示。



圖六、檔案執行先後順序

B.2.1. dataloader.py 檔案功能說明

dataloader.py 檔案的功能為製作 iterator。這個檔案會將訓練資料與測試資料都打包成 iterator，並將此 iterator 通過 Pytorch 套件的 Dataloader 打包成很多個 batch 以供模型訓練。

B.2.2. ResNet18_nonpretrained_model.py 檔案功能說明

ResNet18_nonpretrained_model.py 檔案的功能為使用 ResNet18 nonpretrained model 架構進行訓練、產生視覺化訓練結果、產生測試結果、產生其他相關結果及示意圖、儲存最好的模行權重等...。使用此檔案時，需將 dataloader.py 放置於此檔案相同資料夾下。此檔案會將訓練與測試過程中的 loss 與 accuracy 記錄下來並存至檔名為 ResNet18_nonpretrained.csv 的檔案內(此檔案會放在 history_csv 資料夾下)，並將此模型訓練的權重檔以 ResNet18_nonpretrained.rar 的名稱儲存(此檔案會放在 model_weight 資料夾下)。

B.2.3. ResNet18_pretrained_model.py 檔案功能說明

ResNet18_pretrained_model.py 檔案的功能為使用 ResNet18 pretrained model 進行訓練、產生視覺化訓練結果、產生測試結果、產生其他相關結果及示意圖、儲存最好的模行權重等...。使用此檔案時，需將 dataloader.py 放置於跟此檔案同個資料夾下。此檔案會將訓練與測試過程中的 loss 與 accuracy 記錄下來並存至檔名為 ResNet18_pretrained.csv 的檔案內(此檔案會放在 history_csv 資料夾下)，並將此模型訓練的權重檔以 ResNet18_pretrained.rar 的名稱儲存(此檔案會放在 model_weight 資料夾下)。

B.2.4. ResNet50_nonpretrained_model.py 檔案功能說明

ResNet50_nonpretrained_model.py 檔案的功能為使用 ResNet50 nonpretrained model 架構進行訓練、產生視覺化訓練結果、產生測試結果、產生其他相關結果及示意圖、儲存最好的模行權重等...。使用此檔案時，需將 dataloader.py 放置於跟此檔案同個資料夾下。此檔案會將訓練與測試過程中的 loss 與 accuracy 記錄下來並存至檔名為 ResNet50_nonpretrained.csv 的檔案內(此檔案會放在 history_csv 資料夾下)，並將此模型訓練的權重檔以 ResNet50_nonpretrained.rar 的名稱儲存(此檔案會放在 model_weight 資料夾下)。

B.2.5. ResNet50_pretrained_model.py 檔案功能說明

ResNet50_pretrained_model.py 檔案的功能為使用 ResNet50 pretrained model 進行訓練、產生視覺化訓練結果、產生測試結果、產生其他相關結果及示意圖、儲存最好的模行權重等...。使用此檔案時，需將 dataloader.py 放置於跟此檔案同個資料夾下。此檔案會將訓練與測試過程中的 loss 與 accuracy 記錄下來並存至檔名為 ResNet50_pretrained.csv 的檔案內(此檔案會放在 history_csv 資料夾下)，並將此模型訓練的權重檔以 ResNet50_pretrained.rar 的名稱儲存(此檔案會放在 model_weight 資料夾下)。

B.2.6. ALL_model.py 檔案功能說明

在這個檔案裡面，存放 B.2.2 至 B.2.5 的模型架構，因此在 model.testing 的檔案內就可以呼叫此檔案的模型架構進行測試。

B.2.7. model_testing.py 檔案功能說明

此檔案的功能為測試所有的模型結果，並透過 pandas 將所有的結果整合並顯示出來。

B.2.8. Plot_History_Result.py 檔案功能說明

此檔案的功能為將前面產生的 loss 及 accuracy 資料繪至成各項曲線圖。其中包含全部模型訓練的 Loss 曲線圖、ResNet18 的 pretrained model 與非 pretrained model 比較曲線圖及 ResNet50 的 pretrained model 與非 pretrained model 比較曲線圖。

2. Experiment setups

A. 模型架構說明

A.1. ResNet18

本實驗使用的 ResNet18 模型架構圖與模型參數圖如**圖七**與**圖八**所示。在此模型架構中，本實驗總共使用到以下六種 Layer，分別為 Conv2d、BatchNorm2d、Avgpool2d、Flatten、Linear 及 ReLU。其中更將這些層組成 Residual Block，以盡可能的減少梯度消失的發生。本實驗在此是使用原始的 ResNet18 pretrained model 架構，並無更動模型的層。

```
ResNet(
  (classify): Linear(in_features=512, out_features=5, bias=True)
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=1)
)
```

圖七、ResNet18 模型架構圖

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 112, 112]	9,408
BatchNorm2d-2	[-1, 64, 112, 112]	128
ReLU-3	[-1, 64, 112, 112]	0
MaxPool2d-4	[-1, 64, 56, 56]	0
Conv2d-5	[-1, 64, 56, 56]	36,864
BatchNorm2d-6	[-1, 64, 56, 56]	128
ReLU-7	[-1, 64, 56, 56]	0
Conv2d-8	[-1, 64, 56, 56]	36,864
BatchNorm2d-9	[-1, 64, 56, 56]	128
ReLU-10	[-1, 64, 56, 56]	0
BasicBlock-11	[-1, 64, 56, 56]	0
Conv2d-12	[-1, 64, 56, 56]	36,864
BatchNorm2d-13	[-1, 64, 56, 56]	128
ReLU-14	[-1, 64, 56, 56]	0
Conv2d-15	[-1, 64, 56, 56]	36,864
BatchNorm2d-16	[-1, 64, 56, 56]	128
ReLU-17	[-1, 64, 56, 56]	0
BasicBlock-18	[-1, 64, 56, 56]	0
Conv2d-19	[-1, 128, 28, 28]	73,728
BatchNorm2d-20	[-1, 128, 28, 28]	256
ReLU-21	[-1, 128, 28, 28]	0
Conv2d-22	[-1, 128, 28, 28]	147,456
BatchNorm2d-23	[-1, 128, 28, 28]	256
Conv2d-24	[-1, 128, 28, 28]	8,192
BatchNorm2d-25	[-1, 128, 28, 28]	256
ReLU-26	[-1, 128, 28, 28]	0
BasicBlock-27	[-1, 128, 28, 28]	0
Conv2d-28	[-1, 128, 28, 28]	147,456
BatchNorm2d-29	[-1, 128, 28, 28]	256
ReLU-30	[-1, 128, 28, 28]	0
Conv2d-31	[-1, 128, 28, 28]	147,456
BatchNorm2d-32	[-1, 128, 28, 28]	256
ReLU-33	[-1, 128, 28, 28]	0
BasicBlock-34	[-1, 128, 28, 28]	0
Conv2d-35	[-1, 256, 14, 14]	294,912
BatchNorm2d-36	[-1, 256, 14, 14]	512
ReLU-37	[-1, 256, 14, 14]	0
Conv2d-38	[-1, 256, 14, 14]	589,824
BatchNorm2d-39	[-1, 256, 14, 14]	512
Conv2d-40	[-1, 256, 14, 14]	32,768
BatchNorm2d-41	[-1, 256, 14, 14]	512
ReLU-42	[-1, 256, 14, 14]	0
BasicBlock-43	[-1, 256, 14, 14]	0
Conv2d-44	[-1, 256, 14, 14]	589,824
BatchNorm2d-45	[-1, 256, 14, 14]	512
ReLU-46	[-1, 256, 14, 14]	0
Conv2d-47	[-1, 256, 14, 14]	589,824
BatchNorm2d-48	[-1, 256, 14, 14]	512
ReLU-49	[-1, 256, 14, 14]	0
BasicBlock-50	[-1, 256, 14, 14]	0
Conv2d-51	[-1, 512, 7, 7]	1,179,648
BatchNorm2d-52	[-1, 512, 7, 7]	1,024
ReLU-53	[-1, 512, 7, 7]	0
Conv2d-54	[-1, 512, 7, 7]	2,359,296
BatchNorm2d-55	[-1, 512, 7, 7]	1,024
Conv2d-56	[-1, 512, 7, 7]	131,072
BatchNorm2d-57	[-1, 512, 7, 7]	1,024
ReLU-58	[-1, 512, 7, 7]	0
BasicBlock-59	[-1, 512, 7, 7]	0
Conv2d-60	[-1, 512, 7, 7]	2,359,296
BatchNorm2d-61	[-1, 512, 7, 7]	1,024
ReLU-62	[-1, 512, 7, 7]	0
Conv2d-63	[-1, 512, 7, 7]	2,359,296
BatchNorm2d-64	[-1, 512, 7, 7]	1,024
ReLU-65	[-1, 512, 7, 7]	0
BasicBlock-66	[-1, 512, 7, 7]	0
AdaptiveAvgPool2d-67	[-1, 512, 1, 1]	0
Linear-68	[-1, 5]	2,565

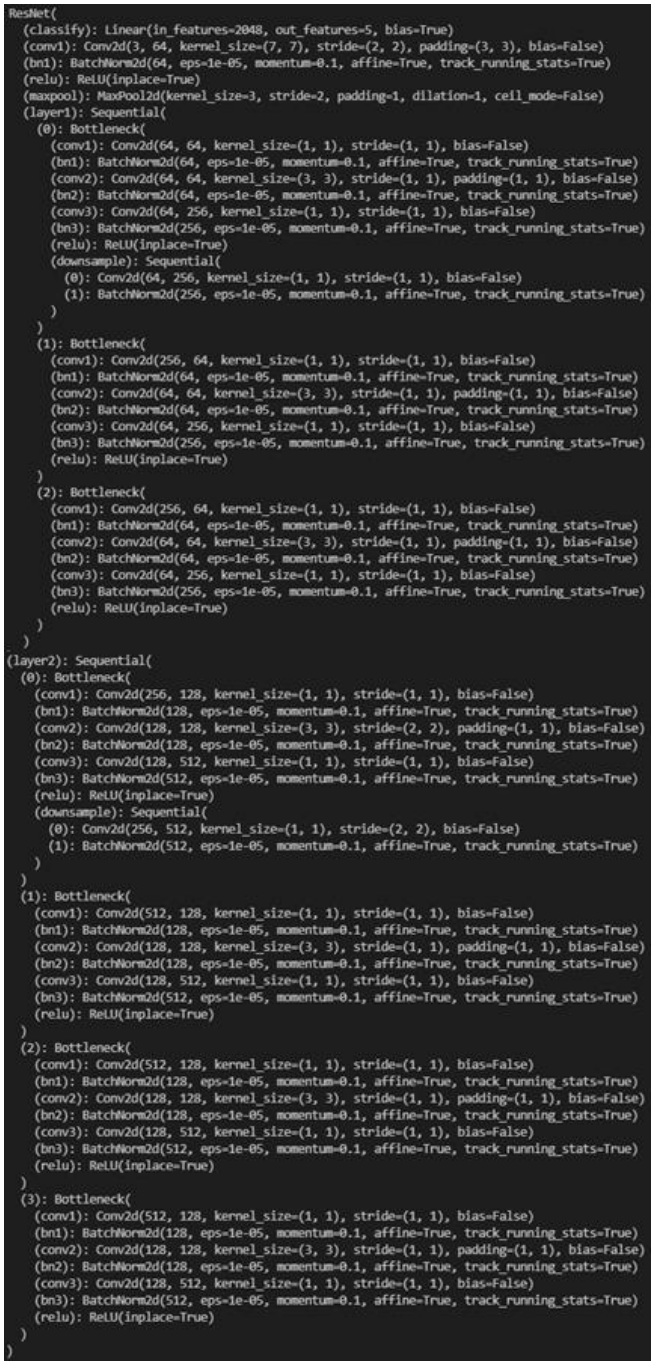
Total params: 11,179,077
Trainable params: 8,393,728
Non-trainable params: 2,785,349

Input size (MB): 0.57
Forward/backward pass size (MB): 62.79
Params size (MB): 42.64
Estimated Total Size (MB): 106.00

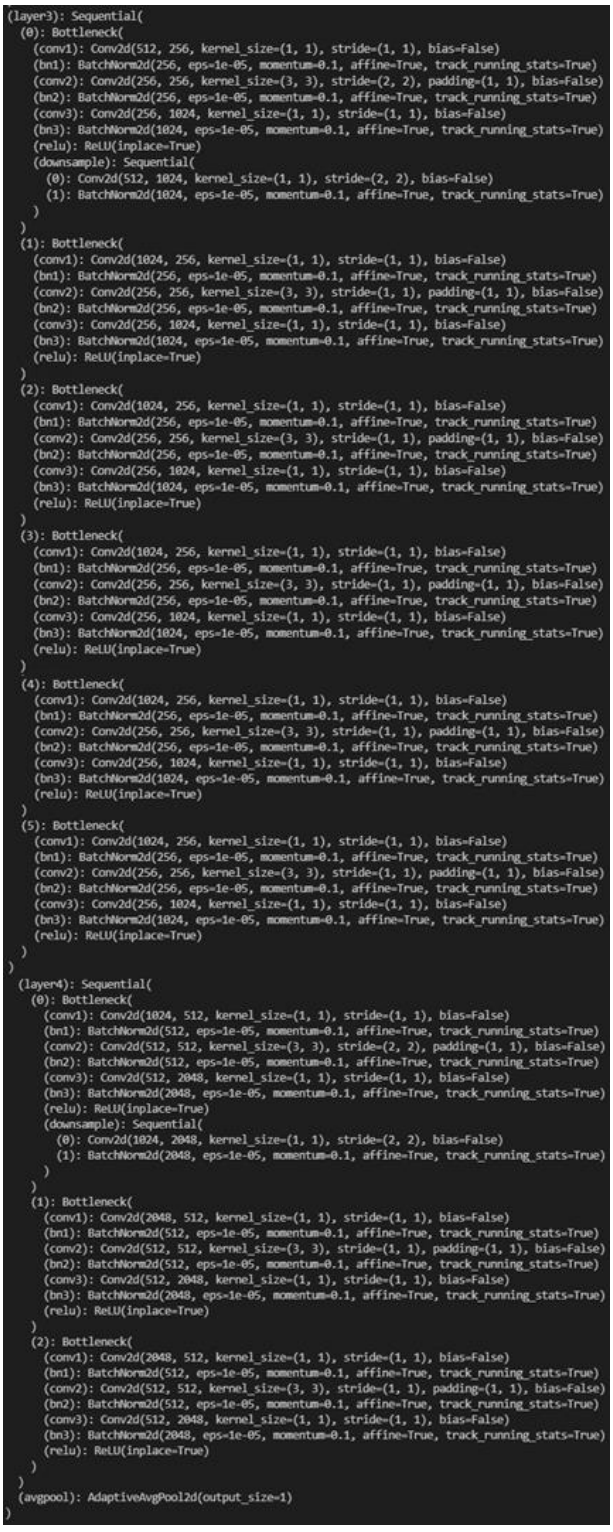
圖八、ResNet18 模型 output shape 與參數量

A.2. ResNet50

本實驗使用的 ResNet50 模型架構圖如**圖九**與**圖十**所示。在此模型架構中，本實驗總共使用到以下六種 Layer，分別為 Conv2d、BatchNorm2d、Avgpool2d、Flatten、Linear 及 ReLU。其中更將這些層組成 Residual Block，以盡可能的減少梯度消失的發生。ResNet50 與 ResNet18 差在模型的層數多寡，因此模型最後的輸出 output 也會不一樣，ResNet18 的 output 為 512，ResNet50 為 2048。因此最後須要有一個 Linear 層去接這些 output。ResNet18 最後的 Linear 層為(512,5)，ResNet50 最後的 Linear 層為(2048,5)。本實驗在此階段是使用原始的 ResNet50 pretrained model 架構，並無更動模型的層。



圖九、ResNet50 模型架構圖-1



圖十、ResNet50 模型架構圖-2

B. Dataloader 說明

在 DataLoader 中總共有兩個函式，一個物件。接下來將針對這些函式或是物件進行說明。其說明分別可以對應到 B.1.至 B.3.。

B.1. process_data_number 函式的介紹與使用

process_data_number 的功能為隨機選取指定類別數量的圖片及標籤，並將選取數量的圖片與標籤返回至 iterator 中。由於在資料探勘的過程中，本實驗發現標籤與標籤之間的圖片數量差異過大。其中標籤 0 對應到的數量為 20655，標籤 1 對應到的數量為 1955，標籤 2 對應到的數量為 4210，標籤 3 對應到的數量為 698，標籤 3 對應到的數量為 581。因此從上述的標籤資料數量中就可以知道，標籤 0 的數量遠遠超過標籤 1 至標籤 3。因此本實驗決定實作此函式，將標籤 0 的數量砍至 1/2、1/3、1/5 或是 1/10。以看哪一個資料數量對於模型分類最有幫助。**圖十一**為 process_data_number 的函式。

```python
import pandas as pd
from skimage import io
import os
from torch.utils import data
from torch.utils.data import DataLoader
from torchvision import transforms
import numpy as np


def process_data_number(img,label):
    img_0,img_1,img_2,img_3,img_4 = [],[],[],[],[]
    label_0,label_1,label_2,label_3,label_4 = [],[],[],[],[]

    print("before:",img.shape,label.shape)
    for i in range(5):
        for j in np.argwhere(label==i):
            if i == 0:
                img_0.append(img[j[0]])
                label_0.append(label[j[0]])

            elif i == 1:
                img_1.append(img[j[0]])
                label_1.append(label[j[0]])

            elif i == 2:
                img_2.append(img[j[0]])
                label_2.append(label[j[0]])

            elif i == 3:
                img_3.append(img[j[0]])
                label_3.append(label[j[0]])

            else:
                img_4.append(img[j[0]])
                label_4.append(label[j[0]])

    print(np.array(img_0).shape)
    print(np.array(label_0).shape)
    print(np.array(img_1).shape)
    print(np.array(label_1).shape)
    print(np.array(img_2).shape)
    print(np.array(label_2).shape)
    print(np.array(img_3).shape)
    print(np.array(label_3).shape)
    print(np.array(img_4).shape)
    print(np.array(label_4).shape)

    img_0 = img_0[:len(img_0)//5]
    label_0 = label_0[:len(label_0)//5]

    # img_1 = img_1[:len(img_4)]
    # label_1 = label_1[:len(label_4)]

    # img_2 = img_2[:len(img_4)]
    # label_2 = label_2[:len(label_4)]

    # img_3 = img_3[:len(img_4)]
    # label_3 = label_3[:len(label_4)]

    # print(np.array(img_0).shape)
    # print(np.array(label_0).shape)
    # print(np.array(img_1).shape)
    # print(np.array(label_1).shape)
    # print(np.array(img_2).shape)
    # print(np.array(label_2).shape)
    # print(np.array(img_3).shape)
    # print(np.array(label_3).shape)
    # print(np.array(img_4).shape)
    # print(np.array(label_4).shape)

    img,label=[],[]

    img = (img_0+img_1+img_2+img_3+img_4)
    label = (label_0+label_1+label_2+label_3+label_4)

    np.random.seed(0)
    np.random.shuffle(img)
    np.random.seed(0)
    np.random.shuffle(label)

    # print(np.array(img).shape)
    # print(np.array(label).shape)
    return np.array(img),np.array(label)
```

圖十一、process_data_number 函式去除多餘資料

B.2. getData 函式的介紹與使用

getData 函式的功能是讀取 csv 檔案內的圖片及標籤資料，以供後續 iterator 返回圖片及標籤資料。此外由於上述的資料不均問題，本實驗在此函式的訓練模式下添加上述的 process_data_number 函式，並去除訓練資料集內資料不均的問題。另外測試資料集的資料則全數返回。**圖十二**為 getData 的函式。

```
def getData(mode):
    if mode == 'train':
        img = pd.read_csv('train_img.csv',header=None)
        label = pd.read_csv('train_label.csv',header=None)

        img,label = np.squeeze(img.values), np.squeeze(label
.values)
        # img,label = process_data_number(img,label)
        return img,label
    else:
        img = pd.read_csv('test_img.csv',header=None)
        label = pd.read_csv('test_label.csv',header=None)
        return np.squeeze(img.values), np.squeeze(label.values)
```

圖十二、getData 函式示意圖

B.2. RetinopathyLoader 物件的介紹與使用

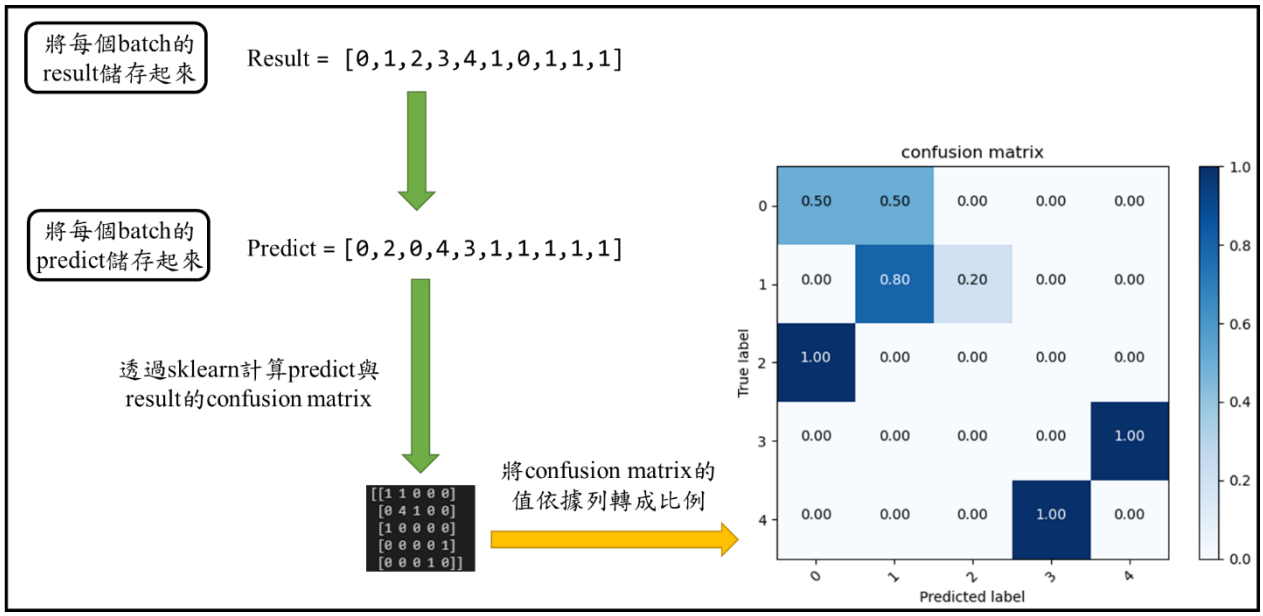
在使用 RetinopathyLoader 時，總共需要輸入三個變數，第一個變數 root 為圖片資料的根目錄，第二個變數 mode 為選擇此 iterator 為 training 還是 testing，第三個變數 transform 為資料增強選擇的方法，將其打包之後傳進 iterator 中進行使用。在 RetinopathyLoader 中，總共有三個部份需要定義，方別可以對應到**圖十三**中的紅色框框、藍色框框及綠色框框。第一個紅色框框需要定義的是 iterator 的變數基本用途，其中包含資料位置(self.root)、圖片名稱(self.img_name)與對應標籤(self.labels)、資料增強方法(self.transform)、訓練或測試模式(self.mode)。第二個藍色框框則需要告知這個 iterator 的資料長度是多少，以供後續在 DataLoader 時可以製作 batch。第三個綠色框框的動作則是依據資料的 index，打開該 index 圖片，並將此圖片經過資料增強後返回圖片與對應 index 標籤。

```
class RetinopathyLoader(data.Dataset):
    def __init__(self, root, mode, transform=None):
        """
        Args:
            root (string): Root path of the dataset.
            mode : Indicate procedure status(training or testing)

            self.img_name (string list): String list that store all image names.
            self.label (int or float list): Numerical list that store all ground truth label values.
        """
        self.root = root
        self.img_name, self.labels = getData(mode)
        self.transform = transform
        self.mode = mode
        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"""

        image_path = self.root + "/"+ self.img_name[index]+ '.jpeg'
        self.img = io.imread(image_path)
        self.label = self.labels[index]

        if self.transform:
            self.img = self.transform(self.img)

        return self.img,self.label
```

圖十三、RetinopathyLoader 物件示意圖

C. evaluation 與 confusion matrix 方法

confusion matrix 的評估流程如**圖十四**所示，首先本實驗會先使用兩個 list 分別儲存每個 batch 中的 predict 結果與真實結果，再來本實驗會使用 sklearn 計算這兩個 list 的 confusion matrix，例如實際結果為 0 的標籤最後預測為 0 的標籤為多少個、實際結果為 0 的標籤最後預測為 1 的標籤為多少個等…以此類推，最後本實驗會將 confusion matrix 的結果依據每一列換算成比例。以[1 1 0 0 0]這一列為例子，可以看到實際結果為 0 的標籤總共有兩個，其中一個被預測為 0，另外一個則被預測為 1。因此如果將預測為 0 的數量除以實際結果為 0 的標籤數量，實際結果為 0 但被預測為 0 的比例就會為 1/2＝0.5，而如果將預測為 1 的數量除以實際結果為 0 的標籤數量，實際結果為 0 但被預測為 1 的比例就會為 1/2＝0.5。因此**圖十五**及**圖十六**的程式碼則是實作**圖十四**的流程，而**圖十五**是實作**圖十四**中綠色箭頭的動作，**圖十六**則是實作**圖十四**中黃色箭頭的動作。



圖十四、Confusion matrix 評估流程圖

```python
def plot_confusion_matrix(y_pred,y_true):
    # y_pred = [0,2,0,4,3,1,1,1,1,1]
    # y_true = [0,1,2,3,4,1,0,1,1,1]

    target_names = list(range(5))
    plt.figure()
    cnf_matrix = confusion_matrix(y_true, y_pred)
    # print(cnf_matrix)
    plot_confusion_matrix_figure(cnf_matrix, classes=target_names,normalize=True,title='confusion matrix')
    # plt.savefig('confusion_matrix.jpg',dpi=300)
    plt.show()
```

圖十五、對應至圖十四的綠色箭頭

```python
def plot_confusion_matrix_figure(cm, classes,normalize=False,title='Confusion matrix',cmap=plt.cm.Blues):

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        # print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    plt.imshow(cm, interpolation='nearest', cmap='Blues')
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()
```

圖十六、對應至圖十四的黃色箭頭

3. Experimental results

A. 最高的 testing accuracy

在 ResNet18 pretrained model 的實驗中，本實驗僅有 freeze 特定的 Layer 進行訓練。在 ResNet18 非 pretrained model 的實驗中，本實驗則會調整 Activation function、Dropout 等 Layer。ResNet50 的 pretrained model 與非 pretrained model 實驗則與 ResNet18 類似。此外本實驗在這些實驗中，均有針對優化器(Optimizers)、Learning rate、weight decay 及資料增強方法等進行調整。最後得出的**表二**為本實驗經過嘗試得出的最高的 testing accuracy 參數結果。**表三**為 pretrained model 與非 pretrained model 實驗中的最高的 testing accuracy。在本次實驗中，最好的結果為 ResNet50 的 pretrained model，其測試結果可到 77.48%。而次要好的測試結果為 ResNet18 的 pretrained model，其 testing accuracy 的結果為 75.82%。實驗測試結果圖也可對應至**圖十七**。

表二、最高的模型實驗參數與方法

	Pretrained model	資料增強方法	Learning rate	Optimizers	Testing Accuracy
ResNet18	是	ToTensor Resize	5e-3	Adam	77.48%
ResNet50	是	ToTensor Resize	5e-3	Adam	75.82%

表三、三種激活函數的兩種架構的最高準確率

	Pretrained	Non-Pretrained
ResNet18	**75.82%**	70.54%

ResNet50	**77.48%**	70.25%

	Pretrained	None-Pretrained
ResNet18	0.758281	0.705448
ResNet50	0.774843	0.702573

圖十七、最好的測試結果

A.1. ResNet18 pretrained model 架構說明

在此模型架構中，本實驗有設置一個機制為取得模型訓練中 Loss 值最小的模型結果，並將當下的模型權重及參數儲存起來以做測試之用。**圖十八**及**圖十九**為模型參數配置與 freeze 的 Layer。可以看到圖中模型參數的配置與初始模型的差異在於 freeze 的 Layer (依據**圖十九**來看，本實驗將 Layer 1 至 Layer 3 freeze)。為了使得增強模型分類的效能，本實驗逐一嘗試多個資料增強方法。最終發現到僅使用 Resize 跟 ToTensor，模型的表現將會最好。**圖二十**為 ResNet18 pretrained model 中最好方法的訓練過程(受到篇幅影響，因此只放中間過程)。

```
class ResNet(nn.Module):
    def __init__(self, pretrained=True):
        super(ResNet, self).__init__()


        self.classify = nn.Linear(512, 5)
        pretrained_model = models.__dict__['resnet{}'.format(18)](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)

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

        # x = nn.Dropout(0.35)(x)

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

        # x = nn.Dropout(0.35)(x)

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

        x = self.avgpool(x)
        # print(x.shape)

        x = x.view(x.size(0), -1)
        x = self.classify(x)

        return x
```

圖十八、ResNet18 pretrained model 架構-1

```
model = ResNet()

for name,child in model.named_children():
    if name in ['layer4','fc']:
        #print(name + 'is unfrozen')
        for param in child.parameters():
            param.requires_grad = True
    else:
        #print(name + 'is frozen')
        for param in child.parameters():
            param.requires_grad = False
```

圖十九、ResNet18 pretrained model 架構-2

```
109/109 [============] - 202s 2s/step - loss: 0.8048 - train accuracy: 0.7378
100%|

 epochs: 0 loss: 0.8136753737926483 Training Accuracy: 0.7318478954081632 Testing Accuracy: 0.7603339720777443
Epoch: 2/20
109/109 [============] - 206s 2s/step - loss: 0.7080 - train accuracy: 0.7601
100%|
```

圖二十、ResNet18 pretrained model 訓練過程

A.2. ResNet50 pretrained model 架構說明

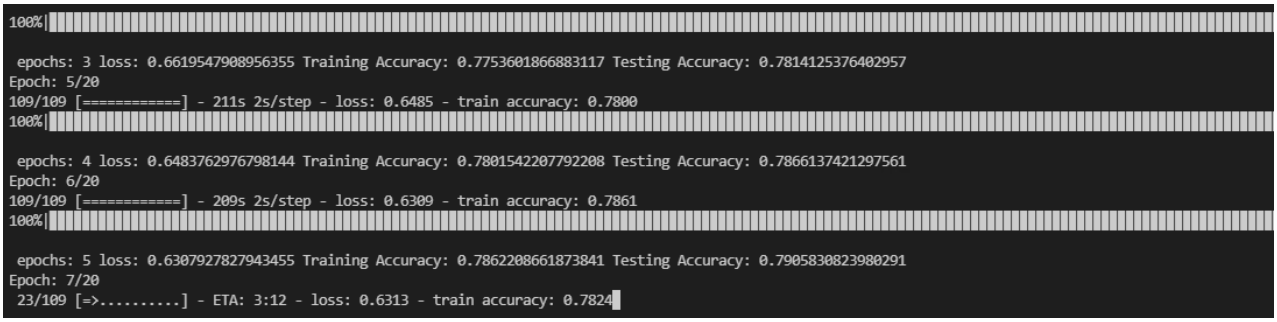
在此模型架構中，本實驗有設置一個機制為取得模型訓練中 Loss 值最小的模型結果，並將當下的模型權重及參數儲存起來以做測試之用。**圖二十一**及**圖二十二**為模型參數配置與 freeze 的 Layer。可以看到圖中模型參數的配置與初始模型的差異在於 freeze 的 Layer (依據**圖二十二**來看，本實驗將 Layer 1 至 Layer 3 freeze)。為了使得增強模型分類的效能，本實驗逐一嘗試多個資料增強方法。最終發現到僅使用 Resize 跟 ToTensor，模型的表現將會最好。**圖二十三**為 ResNet50 pretrained model 中最好方法的訓練過程(受到篇幅影響，因此只放中間過程)。

```python
class ResNet(nn.Module):
    def __init__(self, pretrained=True):
        super(ResNet, self).__init__()


        self.classify = nn.Linear(2048, 5)
        pretrained_model = models.__dict__['resnet{}'.format(50)](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)

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

        # x = nn.Dropout(0.35)(x)

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

        # x = nn.Dropout(0.35)(x)

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

        x = self.avgpool(x)
        # x = nn.Dropout(0.5)(x)
        # print(x.shape)

        x = x.view(x.size(0), -1)
        x = self.classify(x)

        return x
```

圖二十一、ResNet50 pretrained model 架構-1

```python
model = ResNet()

for name,child in model.named_children():
    if name in ['layer4','fc']:
        #print(name + 'is unfrozen')
        for param in child.parameters():
            param.requires_grad = True
    else:
        #print(name + 'is frozen')
        for param in child.parameters():
            param.requires_grad = False
```

圖二十二、ResNet50 pretrained model 架構-2

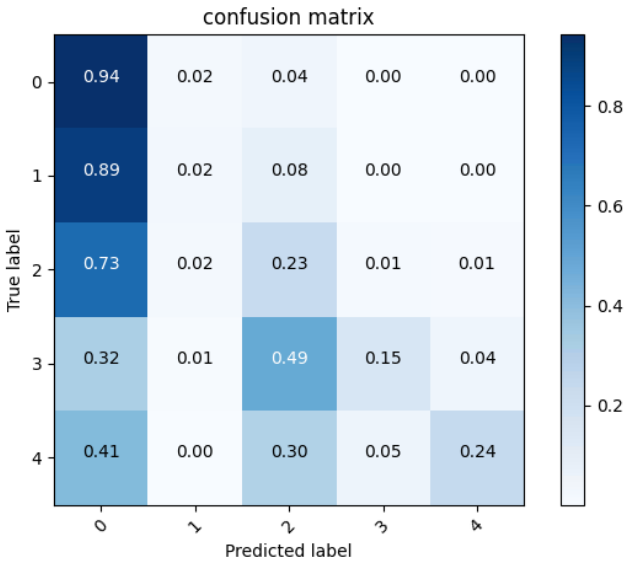


圖二十三、ResNet50 pretrained model 訓練過程

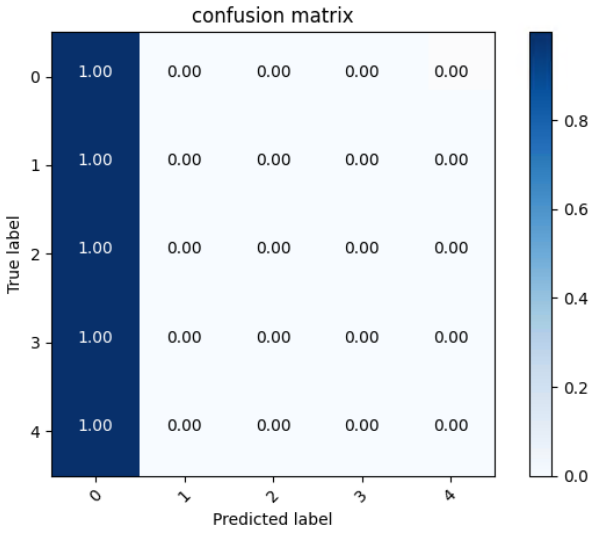
B. 圖表間的比較

B.1. ResNet18 pretrained model 與 ResNet18 nonpretrained model 的 Confusion matrix

圖二十四為 ResNet18 pretrained model 的 Confusion matrix，**圖二十五**為 ResNet18 nonpretrained model 的 Confusion matrix。可以看到在**圖二十四**中，pretrained model 在標籤 0 的預測結果最好(如同前述的資料不平均問題)，而分類較為不佳的結果為標籤 3 及標籤 4(因為這兩個標籤的資料量較少)。在**圖二十五**中，可以看到模型基本上在亂猜，所有的標籤結果都猜到 0 這個標籤(可能是資料不平均的問題)。



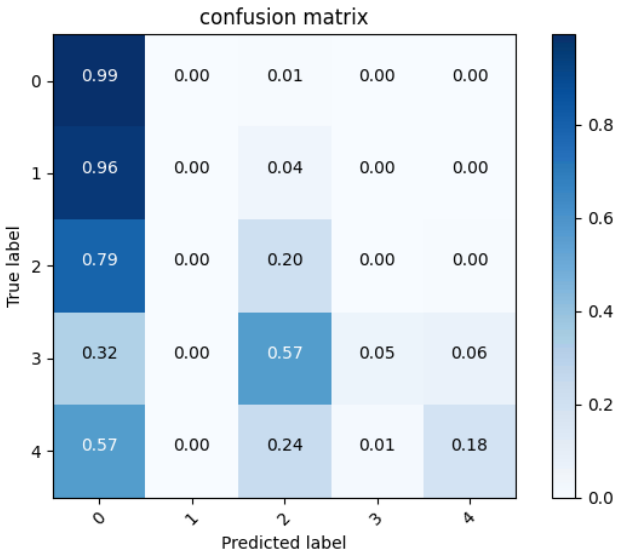
圖二十四、pretrained model confusion matrix



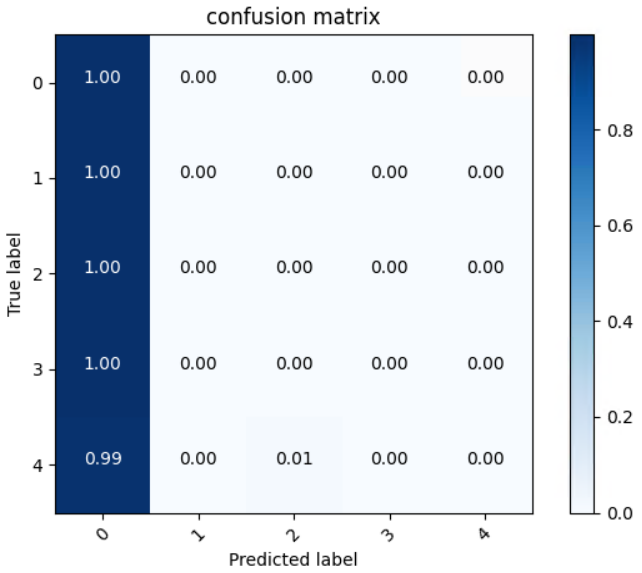
圖二十五、nonpretrained model confusion matrix

B.2. ResNet50 pretrained model 與 ResNet50 nonpretrained model 的 Confusion matrix

圖二十六為 ResNet50 pretrained model 的 Confusion matrix，**圖二十七**為 ResNet50 nonpretrained model 的 Confusion matrix。可以看到在**圖二十六**中，pretrained model 在標籤 0 的預測結果最好(如同前述的資料不平均問題)，而分類較為不佳的結果為標籤 3 及標籤 4 (這兩個標籤的資料量較少)。在**圖二十七**中，可以看到模型基本上在亂猜，所有的標籤結果都猜到 0 這個標籤(可能是資料不平均的問題)。



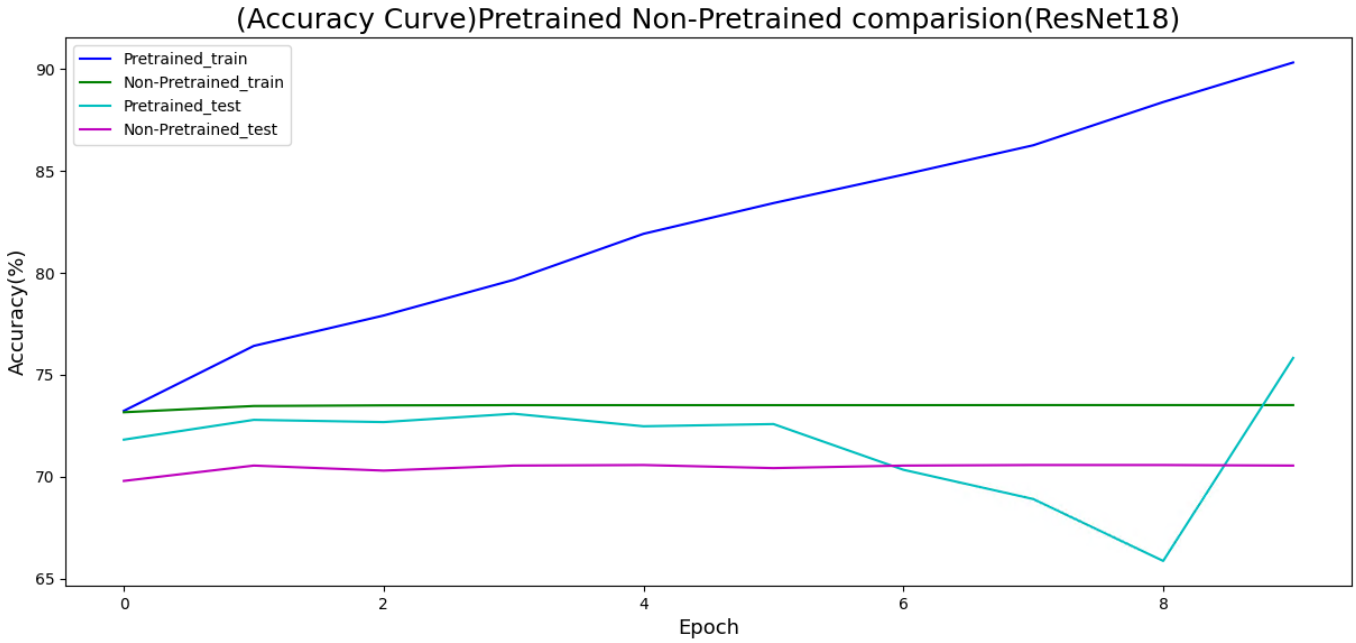
圖二十六、pretrained model confusion matrix



圖二十七、nonpretrained model confusion matrix

B.3. ResNet18 的 Accuracy 曲線

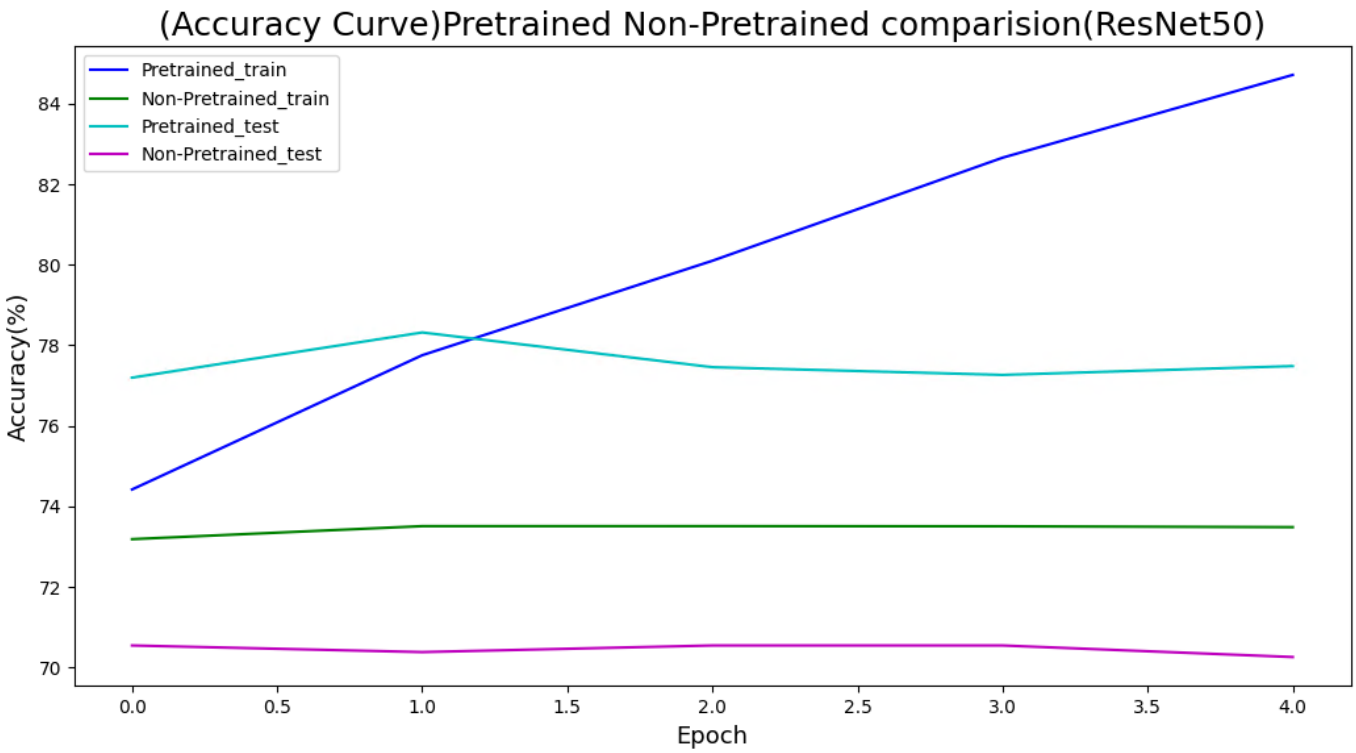
圖二十八為 ResNet18 的 Training Accuracy 曲線。可以從這張曲線圖之中發現到 pretrained model 不論是在 training 或是在 testing 都表現的比 non-pretrained model 還要好。由於資料量與模型參數過大的緣故，本實驗僅跑 10 個 epochs。



圖二十八、ResNet18 Training Accuracy 曲線

B.4. ResNet50 的 Accuracy 曲線

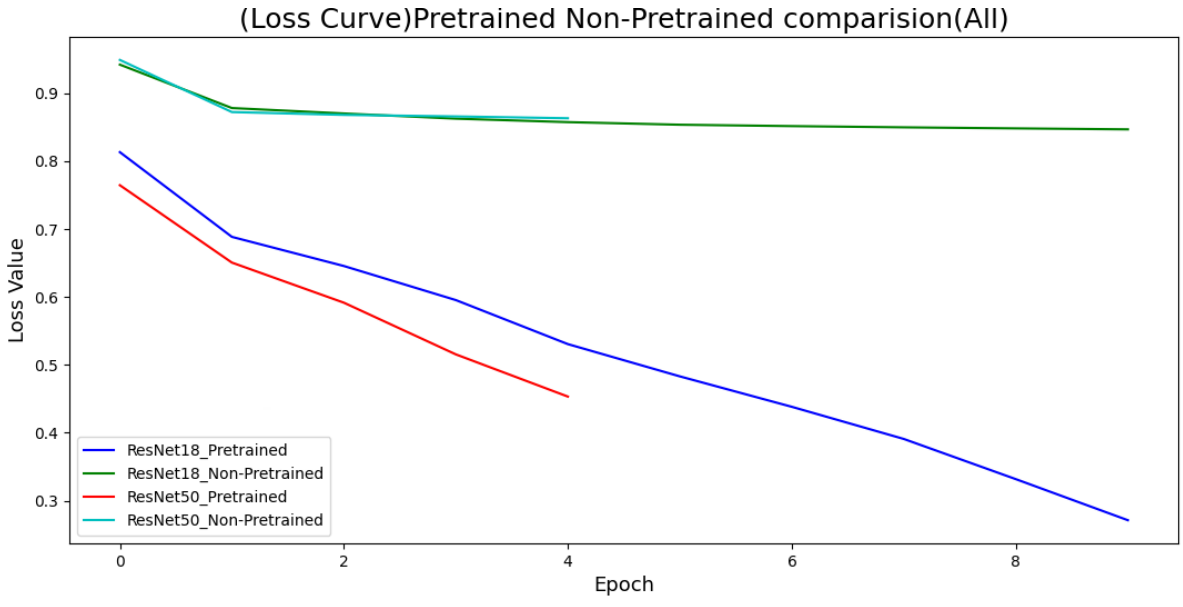
圖二十九為 ResNet50 的 Training Accuracy 曲線。可以從這張曲線圖之中發現到 pretrained model 不論是在 training 或是在 testing 都表現的比 non-pretrained model 還要好。由於資料量與模型參數過大的緣故，本實驗僅跑 5 個 epochs。



圖二十九、ResNet50 Training Accuracy 曲線

B.5. 全部 model 的 Loss 曲線

圖三十為全部 model 的 Loss 曲線。可以從這張曲線圖之中發現到 pretrained model 在 training 時表現的比 non-pretrained model 還要好。不僅如此，ResNet50 由於模型的架構比 ResNet18 還大，因此 ResNet50 能學到更多更好的特徵，其 Loss 值最終也比 ResNet18 還小。受到資料量與模型參數過大的影響，本實驗僅跑 10 個 epochs。

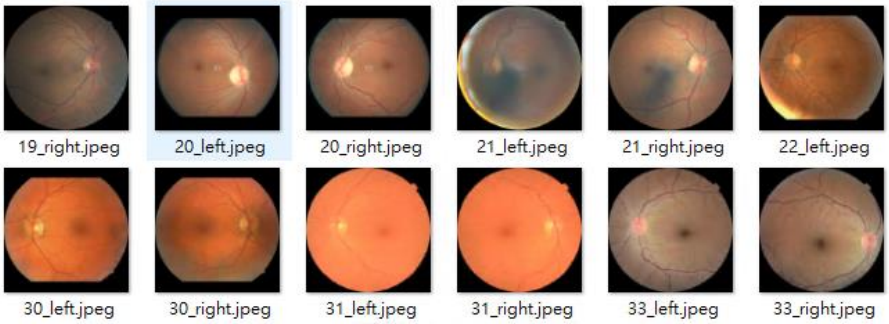


圖三十、全部 model 的 Loss 曲線

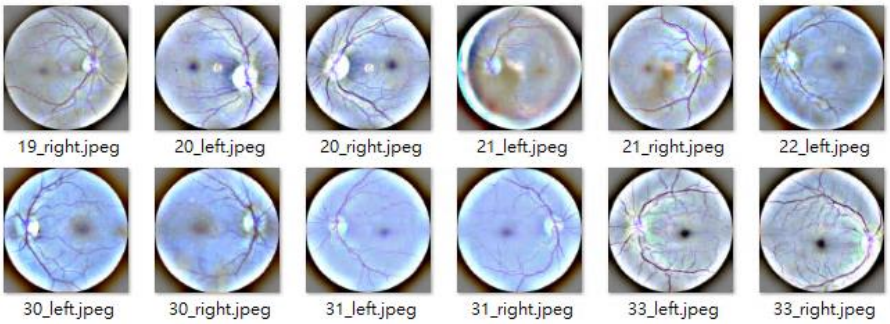
4. Discussion

A. 資料前處理方法之間的討論

在此部份的討論中，本實驗專注在討論是否有使用資料前處理方法。因此在此部份，本實驗以最好的實驗結果，也就是透過 ResNet50 pretrained model 分別進行有進行資料前處理與無進行資料前處理的訓練與測試 Accuracy 與 Loss 值討論。**圖三十一**與**圖三十二**是經過資料前處理前後的圖片差異，本實驗透過資料前處理的方法強化血管特徵(主要是透過高斯模糊)，以利進行糖尿病病癥的判斷。**圖三十三**與**圖三十四**為經過資料前處理與不經過資料前處理的 confusion matrix，可以看到雖然總體而言無經過資料前處理方法的模型較能夠辨識各個類別，但經過資料前處理的模型在標籤 3 的表現中是最為出色的。**表四**對應到的是兩種方法的比較結果。看到這三者的資料可以發現，無經過資料前處理的模型似乎較為能夠辨識糖尿病的病癥。



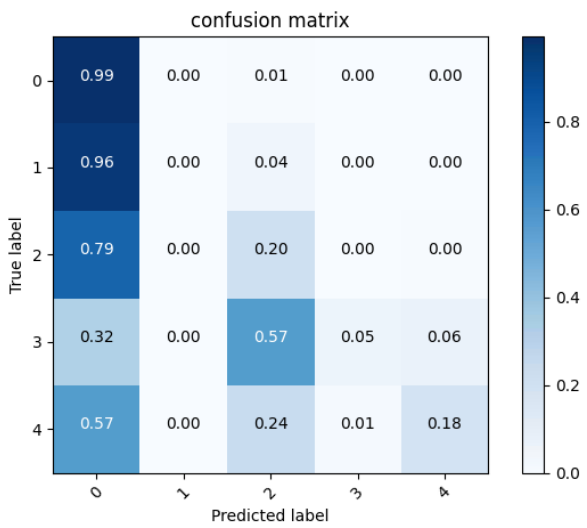
圖三十一、資料前處理前的圖片



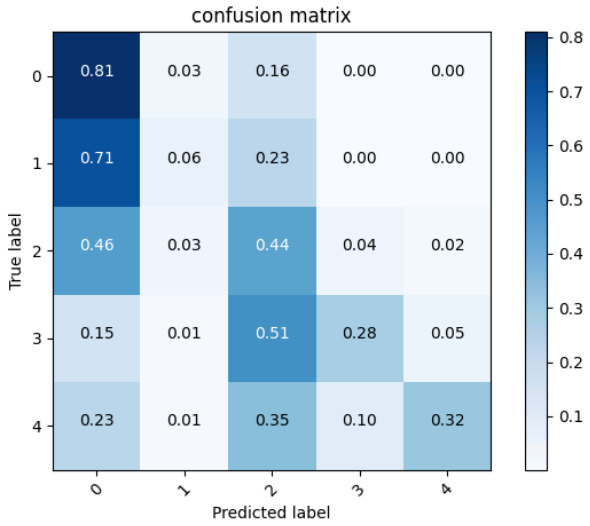
圖三十二、資料前處理後的圖片

表四、使用不同的資料前處理方法(Training Accuracy、Testing Accuracy 及 Loss)

	使用資料前處理	不使用資料前處理
Training Accuracy	99.64%	99.79%
Testing Accuracy	74.29%	77.48%
Loss	0.01	0.0075



圖三十三、無資料前處理的 pretrained model



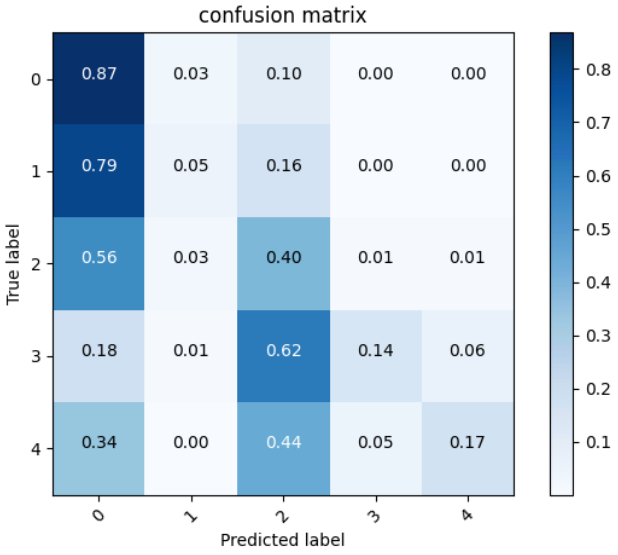
圖三十四、有資料前處理的 pretrained model

B. Pretrained model freeze 特定 Layer 的討論

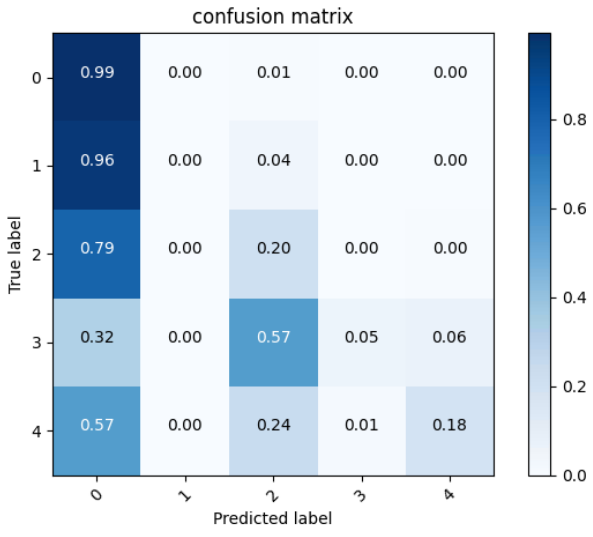
在此部份的討論中，本實驗專注在討論 freeze 哪幾層使得剩餘的模型層得以更新權重及參數。因此在此部份，本實驗以最好的實驗結果，也就是透過 ResNet50 pretrained model 分別進行 freeze 特定幾層的訓練與測試 Accuracy 與 Loss 值討論。這邊將分為兩個需要討論的部份，第一個部份(Case 1)則是將**圖九**的模型層 freeze(Layer1 與 Layer2 皆 freeze)，**圖十**的模型層 unfreeze(Layer3 與 Layer4 則皆 unfreeze)，第三個部份(Case 2)則是將 Layer1 至 Layer3 freeze，Layer4 則 unfreeze。結合這三者的資料可以發現，只將 Layer4 freeze 之後，模型的表現會較為優異。**表五**對應到的是兩種方法的比較結果。

表五、freeze 特定幾層(Training Accuracy、Testing Accuracy 及 Loss)

	Case 1	Case 2
Training Accuracy	90.66%	99.79%
Testing Accuracy	74.99%	77.48%
Loss	0.255	0.0075



圖三十五、Case 1 的 pretrained model



圖三十六、Case 2 的 pretrained model