1. Introduction

這次的 lab 是透過實作出 ResNet18 和 ResNet18 架構，以及撰寫自己的 data loader 來完成分類因糖尿病所造成的視網膜病變嚴重程度。同時，在這次 lab 中，也比較了 model 的是否先被 pretrain 過，而對後續訓練所造成的影響。

比起普通的 CNN 架構，ResNet 的架構中因為包含了直接連結前面輸入至輸出的部分，可以減少當網路串的很深的時候 gradient 容易變成 0 的情況。

2. Experiment setups

A. The details of your model (ResNet)

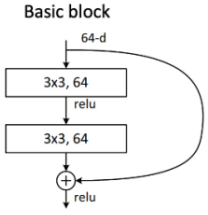
我的實作分成 BasicBlock、BottleneckBlock 以及由這些 block 所組成的 ResNet18 和 ResNet50

a. BasicBlock

以下是我的 BasicBlock 實作方式：

```python
class BasicBlock(nn.Module):
    def __init__(self, in_channels: int, out_channels: int, stride: int = 1, downsample: bool = False):
        super(BasicBlock, self).__init__()  # call constructor of nn.Module

        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=(3,3), stride=stride, padding=(1, 1),bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(out_channels, out_channels, kernel_size=(3,3), stride=1, padding=(1, 1),bias=False),
            nn.BatchNorm2d(out_channels),

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

        if downsample:
            self.downsample = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=(1,1), stride=stride, bias=False),
                nn.BatchNorm2d(out_channels),

            )
        else:
            self.downsample = lambda x:x

    def forward(self, x) -> Tensor:
        output = self.conv1(x)
        output = self.conv2(output)
        identity = self.downsample(x)
        output = self.relu(output + identity)

        return output
```

在實作的時候，Basic Block 裡面 layer 的架構我參考了 spec 中所附的圖。詳細的 Conv2d、BatchNorm2d function 中的參數，我則參考了 torchvision 中所提供的 resnet18 model，我直接把這個 model 印出來，就可以知道它裡面各層所放的參數是什麼，就照著去放入我的 model 中。



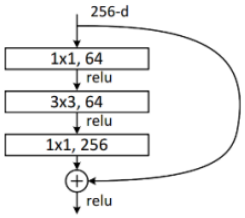

```
(layer2): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 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)
  )
```

b. BottleneckBlock

以下是我的 BottleneckBlock 實作方式：

```python
class BottleneckBlock(nn.Module):

    def __init__(self, in_channels: int, mid_channels: int, out_channels: int, stride: int = 1, downsample: bool = False):
        super(BottleneckBlock, self).__init__()  # call constructor of nn.Module
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, mid_channels, kernel_size=(1,1), stride=1,bias=False),
            nn.BatchNorm2d(mid_channels),
            nn.ReLU(inplace=True)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(mid_channels, mid_channels, kernel_size=(3, 3), stride=stride, padding=(1, 1), bias=False),
            nn.BatchNorm2d(mid_channels),
            nn.ReLU(inplace=True)
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(mid_channels, out_channels, (1, 1), stride=1, bias=False),
            nn.BatchNorm2d(out_channels)
        )

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

        if downsample:
            self.downsample = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=(1,1), stride=stride, bias=False),
                nn.BatchNorm2d(out_channels),

            )
        else:
            self.downsample = lambda x:x

    def forward(self, x) -> Tensor:
        out = self.conv1(x)
        out = self.conv2(out)
        out = self.conv3(out)
        identity = self.downsample(x)
        out = self.relu(out + identity)

        return out
```

實作時一樣是參考講義中附上的架構以及 torchvision 中所提供的 resnet50 model 的參數。




```
(layer1): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=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)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=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)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
```

c. ResNet18

以下是我的 ResNet18 實作方式：

```python
class ResNet18(nn.Module):
    def __init__(self):
        super(ResNet18, self).__init__()

        self.name = 'resnet18'

        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        )
        self.layer1 = self.make_layer(64, 64, stride=1, downsample=False)
        self.layer2 = self.make_layer(64, 128, stride=2, downsample=True)
        self.layer3 = self.make_layer(128, 256, stride=2, downsample=True)
        self.layer4 = self.make_layer(256, 512, stride=2, downsample=True)

        self.avgpool = nn.AdaptiveAvgPool2d(output_size=(1, 1))
        self.fc = self.classify = nn.Sequential(
            nn.Flatten(),
            nn.Linear(512, 5)
        )


    def make_layer(self, in_channels: int, out_channels: int, stride: int = 1, downsample: bool = False):
        return nn.Sequential(
            BasicBlock(in_channels, out_channels, stride=stride, downsample=downsample),
            BasicBlock(out_channels, out_channels)
        )

    def forward(self, x) -> Tensor:
        out = self.conv1(x)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.avgpool(out)
        out = self.fc(out)

        return out
```

我一樣是參考了 torchvision 中所提供的 resnet18 model 各是串了哪些東西，以及觀察他們 feature 的數量之後，找出它裡面的 layer 串接 basic block 的 feature 數量關聯性之後，照著打出我的 model 連接方式。我有將網路最後 fc layer 的輸出 feature 數改成 5，因為我們預測的 class 數是 5 個。

```
ResNet(
  (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)
      )
```

在我打完我的 model 之後，我還使用了 torchsummary 提供的 summary 函式，看看 torchvision 內建的 model 和我的 model 有沒有哪裡不同，以下的圖即為 summary 所印出的一部分的 model 資訊。

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

d. ResNet50

以下是我的 ResNet18 實作方式：

```python
class ResNet50(nn.Module):
    def __init__(self):
        super(ResNet50, self).__init__()

        self.name = 'resnet50'
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        )
        self.layer1 = self.make_layer(3, 64, 64, 256, stride=1, downsample=True)
        self.layer2 = self.make_layer(4, 256, 128, 512, stride=2, downsample=True)
        self.layer3 = self.make_layer(6, 512, 256, 1024, stride=2, downsample=True)
        self.layer4 = self.make_layer(3, 1024, 512, 2048, stride=2, downsample=True)
        self.avgpool = nn.AdaptiveAvgPool2d(output_size=(1, 1))
        self.fc = self.classify = nn.Sequential(
            nn.Flatten(),
            nn.Linear(2048, 5)
        )


    def make_layer(self, block_num: int, in_channels: int, mid_channels: int, out_channels: int, stride: int = 1, downsample: bool = False):

        blocks = []

        blocks.append(BottleneckBlock(in_channels, mid_channels, out_channels, stride=stride, downsample=downsample))
        for i in range(block_num - 1):
            blocks.append(BottleneckBlock(out_channels, mid_channels, out_channels))
        return nn.Sequential(*blocks)

    def forward(self, x) -> Tensor:
        out = self.conv1(x)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.avgpool(out)
        out = self.fc(out)

        return out
```

實作時一樣是參考了 torchvision 中所提供的 resnet50 model 各是串了哪些東西，以及觀察他們 feature 的數量之後，找出它裡面的 layer 串接 bottleneck block 的 feature 數量關聯性之後，照著打出我的 model 連接方式。最後一樣用 summary 來確認是否的我的 model 和 torchvision 提供的有所不同。

B. The details of your Dataloader

每次 training 或 testing 時，pytorch 的 dataloader 都會去呼叫 dataset class 的__getitem__() function 來去拿出 dataset 中的資料。

```
117  class RetinopathyLoader(data.Dataset):
```

而在一開始這個 dataset class 建立的時候，會先去讀存有 training/testing image 的檔名及 label 的 csv 檔

```
13  def getData(mode):
14      if mode == 'train':
15          img = pd.read_csv('./train_img.csv')
16          label = pd.read_csv('./train_label.csv')
17          return np.squeeze(img.values), np.squeeze(label.values)
18      else:
19          img = pd.read_csv('./test_img.csv')
20          label = pd.read_csv('./test_label.csv')
21
22          # img.values.shape = label.values.shape = (7025, 1) = (# of testing data, 1)
23          # np.squeeze(img.values) =  np.squeeze(label.values) = (7025, ) = (# of testing data, )
24          # np.squeeze(a, axis=None): remove axes of length one from a.
25          return np.squeeze(img.values), np.squeeze(label.values)
```

```
120     def __init__(self, root, mode):
121         """
122         Args:
123             root (string): Root path of the dataset.
124             mode : Indicate procedure status(training or testing)
125
126             self.img_name (string list): String list that store all image names.
127             self.label (int or float list): Numerical list that store all ground truth label values.
128         """
129         self.root = root
130         self.img_name, self.label = getData(mode)
131         self.mode = mode
132
133         self.transform = imageTransformation(mode)
134         # self.transform = transforms.Compose([transforms.ToTensor()])
135         print("> Found %d images..." % (len(self.img_name)))
```

之後用__getitem__()拿出第 i 筆資料的時候，就會去檔案路徑中讀第 i 筆資料的影像圖片，並對他做一些圖片的處理。

```
141     def __getitem__(self, index):
142         """something you should implement here"""
143
144         """
145            step1. Get the image path from 'self.img_name' and load it.
146                   hint : path = root + self.img_name[index] + '.jpeg'
147
148            step2. Get the ground truth label from self.label
149
150            step3. Transform the .jpeg rgb images during the training phase, such as resizing, random flipping,
151                   rotation, cropping, normalization etc. But at the beginning, I suggest you follow the hints.
152
153                   In the testing phase, if you have a normalization process during the training phase, you only need
154                   to normalize the data.
155
156                   hints : Convert the pixel value to [0, 1]
157                           Transpose the image shape from [H, W, C] to [C, H, W]
158
159            step4. Return processed image and label
160         """
161         path = os.path.join(self.root, self.img_name[index] + '.jpg')
162         label = self.label[index]
163         img = PIL.Image.open(path)      # PIL image
164         img = self.transform(img)       # tensor, shape = (C, H, W)
165         return img, label
```

做的處理如下圖所示。在 train 的時候，每次 data loader 把照片拿出來的時候，則會隨機的做正負旋轉 20 度(以內的值都有可能)、隨機水平翻，隨機上下翻，之後把照片從 PIL image 變成 tensor 後，對 image 的各個 pixel rgb 值做 normalization。前面旋轉和翻轉的作用是因為，有病變的眼睛照片即使旋轉、或是上下左右翻轉了，也都還是有病變的眼睛(label 不變)，這樣可以讓不同 epoch 訓練時看到的照片有一點不同，增加看過的 data 的多樣性。做 normalization 的用意，是不希望照片本身的色調去影響預測的判斷，因為真正重要的應該是照片中可能像血絲之類的花紋特徵。

在 test 時，因為就是要拿固定的資料來比較 test accuracy，因此旋轉翻轉就不需要做了，只需要做 normalization 就好。

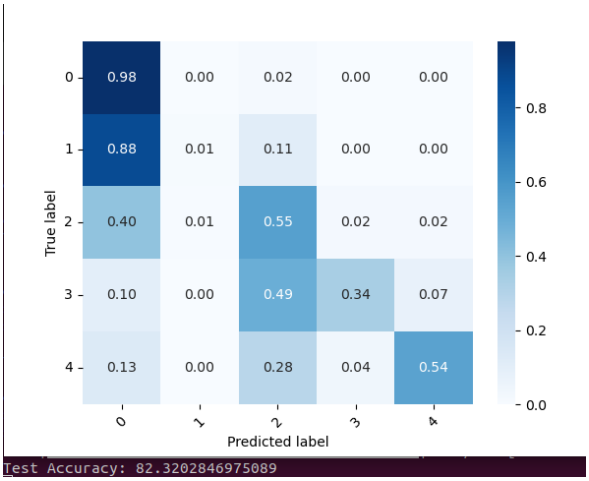
除了 data loader 這裡的圖片處理之外，我在開始 train 和 test 之前有另外先對圖片做處理並存起來，詳細的內容我放在下面的 data preprocessing 做說明。

```python
def imageTransformation(mode):
    normalize = transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    )

    if mode == 'train':
        # torchvision.transforms.Compose(transforms): composes several transforms together
        # torchvision.transforms.ToTensor: converts a PIL Image or numpy.ndarray (H x W x C) in the range [0, 255]
        #                                  to a torch.FloatTensor of shape (C x H x W) in the range [0.0, 1.0]
        transform = transforms.Compose([
            transforms.RandomRotation(degrees=20),
            transforms.RandomHorizontalFlip(),
            transforms.RandomVerticalFlip(),
            transforms.ToTensor(),
            normalize
        ])
    else:
        transform = transforms.Compose([
            transforms.ToTensor(),
            normalize
        ])

    return transform
```

C. Describing your evaluation through the confusion matrix

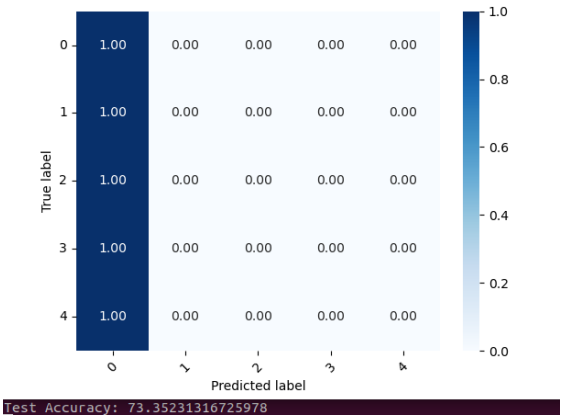
下方的 confusion matrix 是以列方向來 normalize，因此圖中第 i 列第 j 欄數字的意思為”真正 label 為 i 的影像中，有百分之多少 model 預測其 label 為 j”

下面是用 pretrained resnet50 訓練後得到的 test 結果的 confusion matrix。



由 confusion matrix 的圖中可以發現，model 的 test accuracy 雖有 82 左右，不過從 confusion matrix 可看出，model 在不同 label 的影像預測準確率並不太相同。原始影像 label 為 0 的資料，model 幾乎可以非常準確地預測其為 label，準確率有大約 98。然而，原始影像 label 為 1 的資料，model 預測出其真正 label 的準確率幾乎是 0，model 幾乎都把它錯誤預測為 label0，有可能是因為病變初期的眼球其實和沒病變的眼球本身原本就相差不大，再加上 label 1 的訓練資料也比 label 0 的訓練資料少很多所致。原始影像 label 為 2 的資料，和原始影像 label 為 1 的資料相似，model 也容易預測其為沒病變，但 model 已能有大約 5 盛的預測準確率預測其為 label 2。
原始影像 label 為 3, 4 的資料則和原始影像 label 為 1, 2 的結果相似，model 有機會把他們預測為較不嚴重的 label 2。

下面是則是用沒有 pretrained 過的 resnet50 訓練後得到的 test 結果的 confusion matrix。



由於原本的影像資料中，各個 label 的 data 並不平均，label 為 0 的影像佔了大約 7 成左右，所以沒有 pretrained 過的 model，在訓練過程中，即使不去學習真正分類疾病程度的圖片 feature，只要把所有 data 都預測成 0，仍能達到 7 成左右的準確率，從上圖的訓練 accuracy 折線圖可以看到，不管 train 的幾個 epoch，training accuracy 都沒有變高，confusion matrix 中也可看到沒有 pretrained 過的 ResNet18 和 ResNet50 都把所有資料預測成 label 0。
而 pretrained 過的 model，雖然他原本用來訓練的資料並不是糖尿病的眼球資料，但它仍從原本的訓練資料中學到了一些有用的辨識圖片 feature 的方法，因此再用糖尿病的眼球資料去 train 它後，就能真正去學習分類疾病程度的圖片 feature，因此 training accuracy 和 testing accuracy 都隨著訓練 epoch 數變多而增加，從 confusion matrix 也可看出它大部分預測的都是正確的。

3. Data Preprocessing

A. How you preprocessed your data?

因為 data 中的影像原始尺寸幾乎都不相同，長寬比也幾乎都不是 1:1，直接 resize 圖片成固定的 512*512 可能會導致眼球被伸縮。因此我先計算出原本圖片中有顏色的地方(3 個 channel 只要有任一個不是 0 就算有顏色)，然後找到這些有顏色的地方的 row 和 column 的最大值和最小值，這個最大值和最小值維出來的地方就是圖片中有眼球的地方，其他地方都是黑的，因此可以裁掉。我為了讓裁出來的圖片可以為正方形，我把 row 的最大最小值相差及 column 的最大最小值相差中比較大的那一個當作裁出來圖片的寬和高，原本眼球照片比較短的那一邊的其他部份就用黑色來填滿。

```python
def cropImage(img):
    non_zeros = img.nonzero()   # find indeices of non zero elements, non_zeros -> Tuple with three arrays as the element
                                # non_zeros[0] = array for the row index of non zero elements
                                # non_zeros[1] = array for the column index of non zero elements
                                # non_zeros[1] = array for the channel index of non zero elements

    non_zero_rows = [min(np.unique(non_zeros[0])), max(np.unique(non_zeros[0]))]      # the first and the last row with non zero elements
    non_zero_cols = [min(np.unique(non_zeros[1])), max(np.unique(non_zeros[1]))]      # the first and the last column with non zero elements

    dim = max(non_zero_rows[1] - non_zero_rows[0] + 1, non_zero_cols[1] - non_zero_cols[0] + 1)
    crop_img = np.zeros((dim, dim, 3), dtype='uint8')

    if non_zero_rows[1] - non_zero_rows[0] + 1 > non_zero_cols[1] - non_zero_cols[0] + 1:
        # rows more than columns
        diff = non_zero_rows[1] - non_zero_rows[0] - non_zero_cols[1] + non_zero_cols[0]
        col_start = int(diff/2)
        cols_num = non_zero_cols[1] - non_zero_cols[0] + 1
        crop_img[:, col_start:(col_start+cols_num), :] = img[non_zero_rows[0]:(non_zero_rows[1]+1), non_zero_cols[0]:(non_zero_cols[1]+1), :]

    else:

        # columns more than rows
        diff = non_zero_cols[1] - non_zero_cols[0] - non_zero_rows[1] + non_zero_rows[0]
        row_start = int(diff/2)
        rows_num = non_zero_rows[1] - non_zero_rows[0] + 1
        crop_img[row_start:(row_start+rows_num), :, :] = img[non_zero_rows[0]:(non_zero_rows[1]+1), non_zero_cols[0]:(non_zero_cols[1]+1), :]

    return crop_img
```

將照片裁成正方形之後，因為 data loader 在 load 資料的時候，必須要每筆資料的 size 是相同的，而前面裁出來的照片只確保是正方形，並不一定尺寸就相同，所以我在裁完之後再把他們 resize 成 512*512，因為照片已經是正方形了，所以這裡 resize 就不會拉伸眼球。之後就把這些處理成 512*512 的照片存起來，等 train 和 test 的時候用。

```python
def imageTransformation():

    transform = transforms.Compose([
        transforms.Resize(512)
    ])

    return transform
```

```python
        img = cropImage(img)
        img = PIL.Image.fromarray(img)
        img = transform(img)
        img.save(save_root + img_name[i] + '.jpg')
```

前面的做法是在 train 和 test 開始之前另外做的，目的是為了避免 train 和 test 的時候每次從 data loader 拿照片出來的時候都要重新再算一次要裁掉的地方跟 resize，速度會太慢。

B. What makes your method special?

其實 torchvision 的 transform 中有直接提供像是 CenterCrop(size)這樣，只要輸入想要切出的影像大小，他就可以直接幫你完成從圖片中心切出 image 的 function。但是這樣的問題是，我並不知道裁出的影像需要多大，才能夠剛好涵蓋到眼球的範圍，而且眼球也可能並不再圖片的正中央。因此，用我自己寫的 crop image function，可以盡量確保不會自己裁掉眼球。

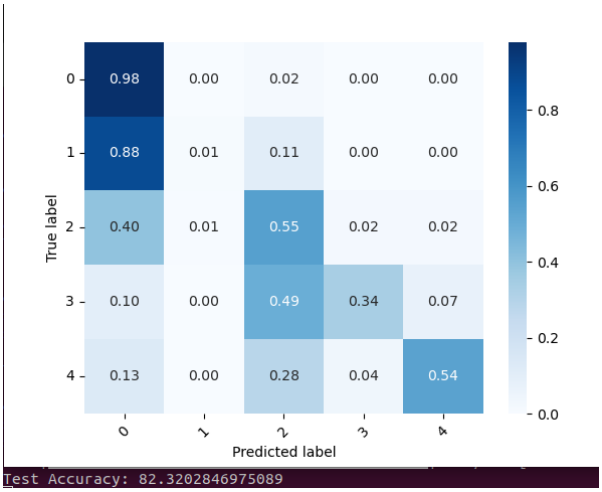
4. Experimental results

A. The highest testing accuracy

• Screenshot

以下的圖為 Pretrained ResNet50 用以下 hyperparameter 設定 train 至第 12 個 epoch 的結果:

batch size = 12, learning rate = 1e-3, optimizer = SGD, momentum = 0.9, weight_decay = 5e-4, loss function = Cross Entropy Loss



• Anything you want to present

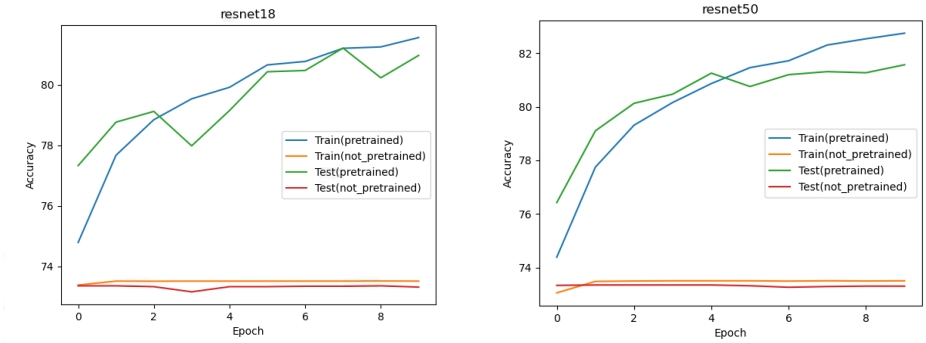
雖然我的最高結果是用 Pretrained ResNet50 得到的，但我覺得只要也讓 Pretrained ResNet18 再多 train 幾個 epoch(因為我只有讓他 train 到第 10 個 epoch)，應該也能用 Pretrained ResNet18 得到大於 82 的 testing accuracy。

B. Comparison figures

• Plotting the comparison figures

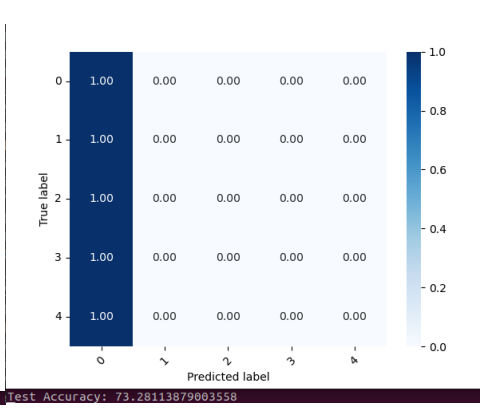
以下的圖為用以下 hyperparameter 設定 train 至 10 個 epoch 後的結果

batch size = 16 (ResNet18) / 12 (ResNet50), learning rate = 1e-3, optimizer = SGD, momentum = 0.9, weight_decay = 5e-4, loss function = Cross Entropy Loss

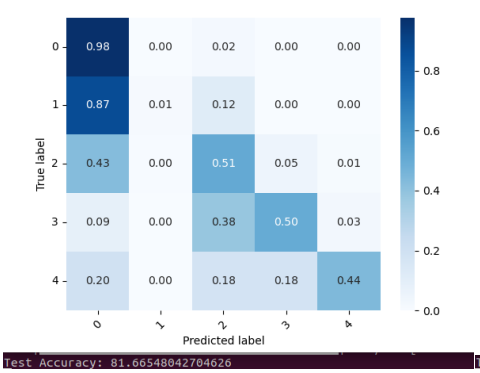


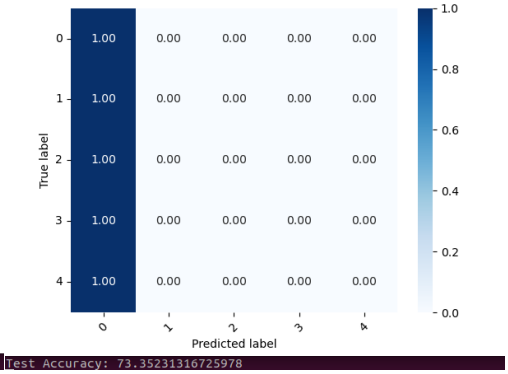
• (RseNet18/50, with/without pretraining)

Pretrained ResNet18　　Not-Pretrained ResNet18



Pretrained ResNet50　　Not-Pretrained ResNet50





其實從訓練過程的準確率和 epoch 的折線圖還有 confusion matrix 的圖來看，我覺得用 ResNet18 和 ResNet50 這兩個 model 的差異其實不大，對於結果來說，差異大的是是否 model 有 pretrained 過。

5. Discussion

A. Anything you want to share

其實我一開始在實作 model 的時候有先去看過原本 paper 裡提供的這張架構圖，但一開始其實對裡面的這些參數都沒有非常理解，也不太懂到底該怎麼串，所以才想說直接去看 torchvision 裡面的 model 看他們是怎麼接的。

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	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

但是實作完 model 之後，我再回去看原本 paper 中的圖，還有一篇網路上找到的解釋之後，就看得懂 paper 中這張架構圖的內容了！
Reference: https://blog.csdn.net/EasonCcc/article/details/108474864