Deep Learning Lab 4

311552004 李明倫

1. Introduction

Diabetic retinopathy is the leading cause of blindness in the working-age population of the developed world. It is estimated to affect over 93 million people. This dataset provided with a large set of high-resolution retina images taken under a variety of imaging conditions.

In this lab, you are required to analysis diabetic retinopathy (糖尿病所引發視網膜病變) in PyTorch. A clinician has rated the presence of diabetic retinopathy in each image on a scale of 0 to 4, according to the following scale:

- 0 - No DR
- 1 - Mild
- 2 - Moderate
- 3 - Severe
- 4 - Proliferative DR

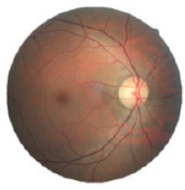
Your task is to assign a score based on this scale.

Requirements

- Implement ResNet18, ResNet50 architecture and load parameters from a pretrained model.
- Compare and visualize the accuracy trend between the pretrained model and without pretraining in the same architectures, you need to plot each epoch accuracy during training and testing phase.
- Implement your own custom DataLoader.
- Design your own data preprocessing method.
- Caluculate the confusion matrix and plotting.

Dataset

Contains 35124 images, with 28100 training data and 7026 testing data. The images' resolutions are different and are required to be preprocessed into the same resolution: 512 * 512.



2. Experiment Setups

The details of your model (ResNet)

ResNet Family (ResNet18, ResNet34) 由 Basic block 組成，而 (ResNet50, ResNet101, …) 則由 Bottleneck block 組成，因此我先分別實做了這兩個 class。

接著根據不同的 ResNet 架構，去定義各自使用的 block type、channel、number of times to repeat the block。在原始的 paper 中，作者提及 downsampling 使用在 conv3_1, conv4_1, and conv5_1，而在 torchvision 實作的 resnet v1.5 中，他把 downsampling 換成在中間 3x3 convolution layer 上，這個版本可以提升大約 0.5% 的 accuracy 在 top1 metric 上，但會多消耗一點運算資源。

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

```
import torch
import torch.nn as nn

class BasicBlock(nn.Module):
    # 'expansion' increases the output channel size in the second conv layer
    expansion = 1

    def __init__(self, in_channels, channels, stride=1):
        super(BasicBlock, self).__init__()

        self.conv1 = nn.Conv2d(in_channels, channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(channels)
        self.conv2 = nn.Conv2d(channels, channels * self.expansion, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(channels * self.expansion)
        self.relu = nn.ReLU(inplace=True)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != channels * self.expansion:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, channels * self.expansion, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(channels * self.expansion)
            )

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

        x = self.conv2(x)
        x = self.bn2(x)

        x += self.shortcut(identity)
        x = self.relu(x)
        return x

class Bottleneck(nn.Module):
    expansion = 4

    def __init__(self, in_channels, channels, stride=1):
        super(Bottleneck, self).__init__()

        self.conv1 = nn.Conv2d(in_channels, channels, kernel_size=1, stride=1, bias=False)
        self.bn1 = nn.BatchNorm2d(channels)
        self.conv2 = nn.Conv2d(channels, channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(channels)
        self.conv3 = nn.Conv2d(channels, channels * self.expansion, kernel_size=1, stride=1, bias=False)
        self.bn3 = nn.BatchNorm2d(channels * self.expansion)
        self.relu = nn.ReLU(inplace=True)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != channels * self.expansion:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, channels * self.expansion, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(channels * self.expansion)
            )

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

        x = self.conv2(x)
        x = self.bn2(x)
        x = self.relu(x)

        x = self.conv3(x)
        x = self.bn3(x)

        x += self.shortcut(identity)
        x = self.relu(x)
```

```
        return x


class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=5):
        super(ResNet, self).__init__()
        self.in_channel = 64

        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.layer1 = self.make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self.make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self.make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self.make_layer(block, 512, num_blocks[3], stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * block.expansion, num_classes)

    def make_layer(self, block, channel, num_block, stride):
        layers = []
        strides = [stride] + [1] * (num_block - 1)
        for stride in strides:
            layers.append(block(self.in_channel, channel, stride))
            self.in_channel = channel * block.expansion
        return nn.Sequential(*layers)

    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

def ResNet18():
    return ResNet(BasicBlock, [2, 2, 2, 2])

def ResNet50():
    return ResNet(Bottleneck, [3, 4, 6, 3])
```

The details of your DataLoader

- `__init__`

  這個部分我會傳入 transforms，目的是在 `__get_item__` 的時候會把 image 做對應的 transform。

- `__len__`

  這裡就簡單的回傳全部資料的數量。

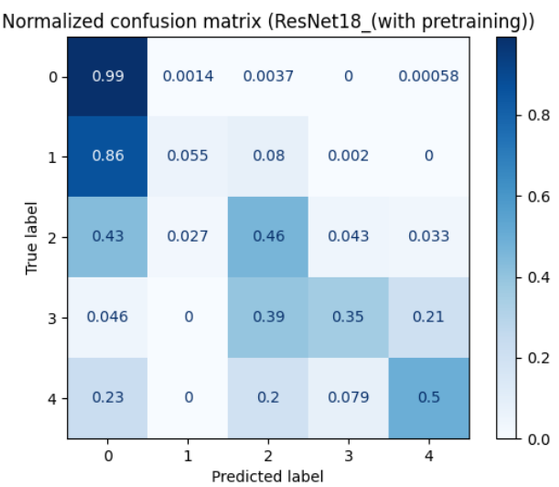
- `__get_item__`

  這個部分透過 cv2 根據 index 去讀取圖片，接著 apply 傳入的 transformation，得到轉換過後的資料，再接著與對應的 label 一起回傳。

```
import cv2
import numpy as np
import pandas as pd
import torch
from torch.utils import data


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


class RetinopathyLoader(data.Dataset):
    def __init__(self, root, mode, transforms):
        """
        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.mode = mode
        self.img_name, self.label = getData(mode)
        self.transforms = transforms

        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):
        """
           step1. Get the image path from 'self.img_name' and load it.
                  hint : path = root + self.img_name[index] + '.jpeg'

           step2. Get the ground truth label from self.label

           step3. Transform the .jpeg rgb images during the training phase, such as resizing, random flipping,
                  rotation, cropping, normalization etc. But at the beginning, I suggest you follow the hints.

                  In the testing phase, if you have a normalization process during the training phase, you only need
                  to normalize the data.

                  hints : Convert the pixel value to [0, 1]
                          Transpose the image shape from [H, W, C] to [C, H, W]

            step4. Return processed image and label
        """
        path = self.root + self.img_name[index] + ".jpeg"

        img = cv2.imread(path)
        label = self.label[index]

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

        return img, label
```

Describing your evaluation through the confusion matrix



- Confusion Matrix

  我先透過 `sklearn.metrics.confusion_matrix` (傳入 y_true, y_pred) 去建立 confusion matrix，同時 normalize 的部分有四個選項可以做設定 (true, pred, all, None)，根據講義上的範例就設成 'true'。

- ConfusionMatrixDisplay

  接者把建立好的 confusion matrix object 傳入 `sklearn.metrics.ConfusionMatrixDisplay` 中去做顯示，同時可以傳入 color map 去做顏色的設定。

透過 confusion matrix，就可以清楚的知道說每一個類別的分類狀況，實際的 label 為何，判斷正確與判斷錯誤的又各自佔有多少比例，更能夠了解 model 當前 training 的狀況。

```
def plot_confusion_matrix(y_true, y_pred, classes, normalize='true', title=None, cmap=plt.cm.Blues):
    matrix = confusion_matrix(y_true=y_true, y_pred=y_pred, normalize=normalize)
    ConfusionMatrixDisplay(confusion_matrix=matrix, display_labels=np.arange(classes)).plot(cmap=cmap)
    plt.title('Normalized confusion matrix ({})'.format(title))
    plt.savefig('./figures/confusion_matrix_' + title + '.png')
```

3. Data Preprocessing

How you preprocessed your data?

Image Resizing and Center-Cropping

因為圖片的大小都不一樣，而且每一張圖片的大小都很大，如果每次訓練的時候都要做很重的 preprocessing的話，會讓整個 training 的過程變的很沒有效率，因此我去參考了 AlexNet 和 ResNet 兩篇 paper 裡面他們處理 ImageNet 的方法，大致上採用以下方法:

- For rectangular image, rescaled the image let shorter side be length of 256 and cropped out the central 256x256 patch.
- Extracting random 224x224 patches from the 256x256 images → random crops

在這邊我就參考他們的做法，先將影像等比例縮小至最短邊為512，接著對影像做 center cropping 成 512 * 512，得到裁切後的影像之後，我就將他們另外存在 train, test 的資料夾裡面，之後要訓練的時候就去從這兩的資料夾裡面讀取圖片就好，能夠提升 load data 的效率

```
import os
import math
import cv2
import numpy as np
import pandas as pd
from tqdm import tqdm
from utils import mkdir

mkdir('./data/train')
mkdir('./data/test')

datasets = ['train', 'test']
for dataset in datasets:
    image_paths = pd.read_csv(dataset + "_img.csv", header=None)
    image_paths = np.squeeze(image_paths)           # 拿掉最外層的 dimension

    print('\n----- Preprocessing on {} data: {} -----'.format(dataset, len(image_paths)))
    for path in tqdm(image_paths):
        img_path = os.getcwd() + "/data/new_" + dataset + "/" + path + '.jpeg'
        img = cv2.imread(img_path, cv2.IMREAD_UNCHANGED)

        width, height = img.shape[1], img.shape[0]
        new_width, new_height = 0, 0

        # 找出比較短的那一邊，等比例做 downsample / upsample 成最小邊是 512
        shorter_side = min(width, height)
        ratio = 512 / shorter_side
        new_width, new_height = math.ceil(width*ratio), math.ceil(height*ratio)

        new_img = cv2.resize(img, (new_width, new_height), interpolation=cv2.INTER_AREA)

        # applies center crop
        center = new_img.shape
        w, h = 512, 512
        x = int(center[1] // 2 - 256)
        y = int(center[0] // 2 - 256)
        crop_img = new_img[y:y+h, x:x+w]

        cv2.imwrite("./data/" + dataset + "/" + path + '.jpeg', crop_img)
```

Image Transformation and Augmentation

在 `main.py` 裡面我定義了兩個 transformation，一個是用在 training data 另外一個用在 testing data。'train' 中我有另外做了 `RandomHorizontalFlip()` 與 `RandomVerticalFlip()` 兩種 augmentation，而在 'test' 中則是基本的把資料轉成 tensor type 與 normalization。

```
tranforms = {
    'train': transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225)),
        transforms.RandomHorizontalFlip(),
        transforms.RandomVerticalFlip(),
    ]),
```

```
    'test': transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225)),
    ]),
}

trainset = RetinopathyLoader(root='./data/train/', mode="train", transforms=tranforms['train'])
testset = RetinopathyLoader(root='./data/test/', mode="test", transforms=tranforms['test'])
```

What makes your method special?

- Based on previous work

  因為像是 AlexNet 與 ResNet 他們在訓練 ImageNet 上的資料時，同樣也有面臨到影像資料大小不一，且大小也蠻大的情況，因此可以借鏡他們的方法當作一個基礎的方法。先將圖片等比例縮小(或放大)之後，再將圖片做 center-cropping 可以大幅的移除旁邊的黑框，而且視網膜的影像也相對的都在中間。

- Efficiency

  我並沒有在 dataloader 中去做前面提到比較重的 preprocessing (resize + center crop)，而是選擇先把原本的資料整理好成一份新的資料集，在 training 的時候我就去讀那份整理好的 dataset 就好。如此一來能夠節省電腦需要花很大的資源去讀很大的影像，在 dataloader 讀取影像時可以快速很多，提升整體的訓練效率。

- Augmentation

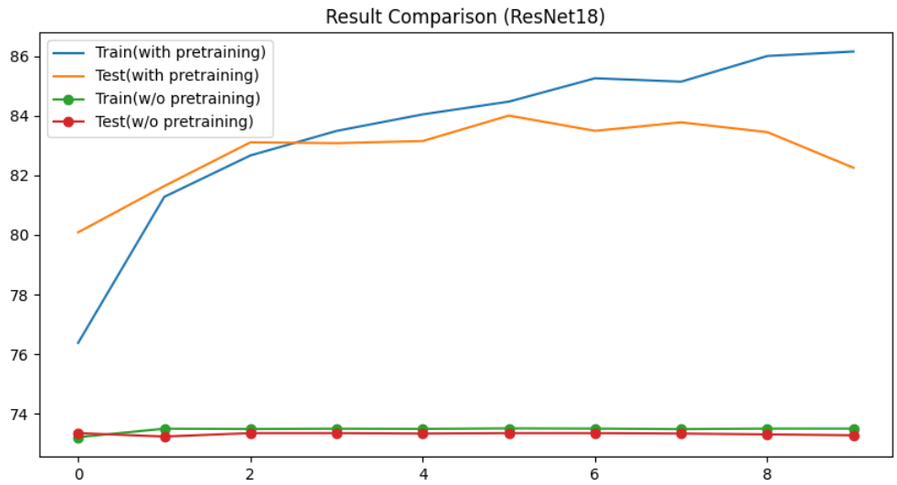
  一開始是猜想也許視網膜病變的人，左眼跟右眼的影像應該不會差異太大，可能嚴重程度跟顏色深淺比較有關係，因此我有另外加入 `RandomHorizontalFlip()` 與 `RandomVerticalFlip()` 兩種 augmentation，去翻轉一下眼球的角度，試著看看能不能讓 model 的 performance 更好，那從實驗結果來看，的確有提升一點點 model 的表現。

4. Experimental Results

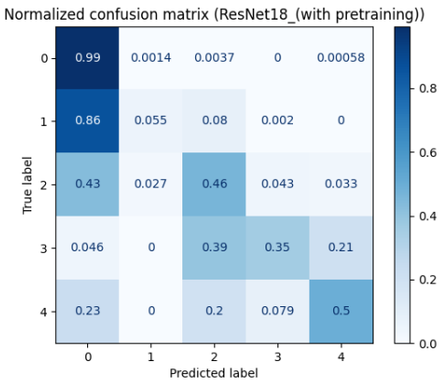
The highest testing accuracy & Comparison figures

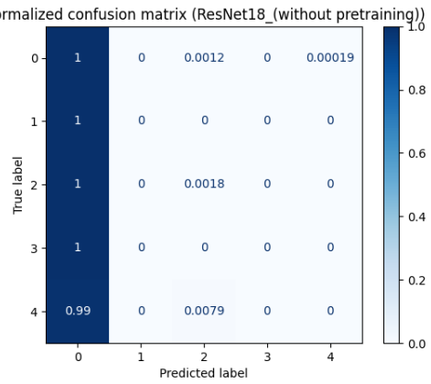
	ResNet18	ResNet18 (pretrain)	ResNet50	ResNet50 (pretrain)
Test Acc	73.36%	84.00%	73.36%	**84.69%**

- **ResNet18: w, w/o pretrain**
  - Comparison figure

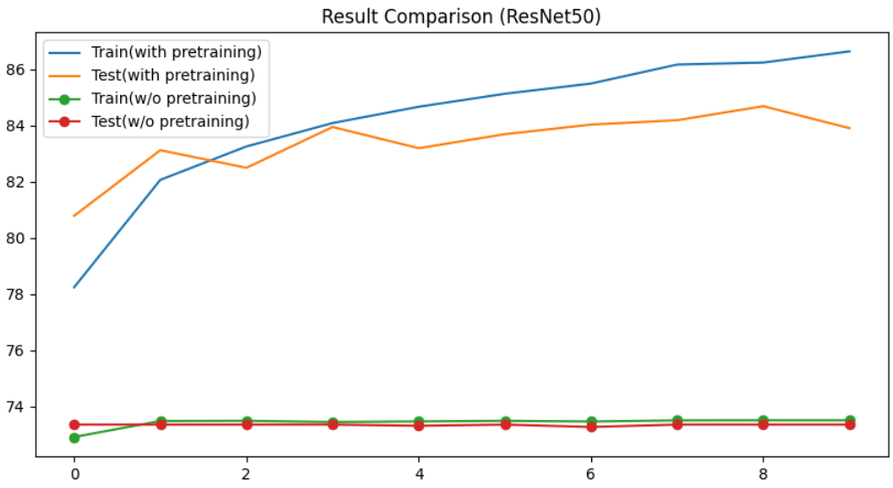


  - Confusion Matrix

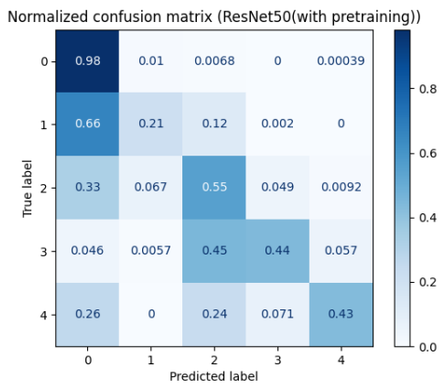


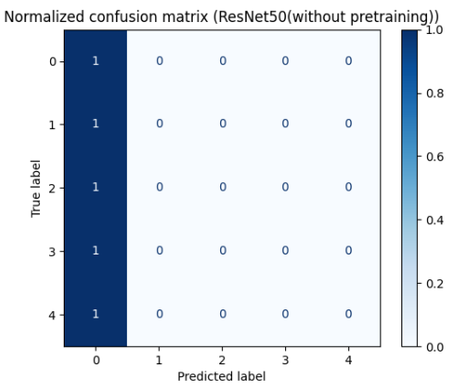


- **ResNet50: w, w/o pretrain**
  - Conparison figure



  - Confusion Matrix



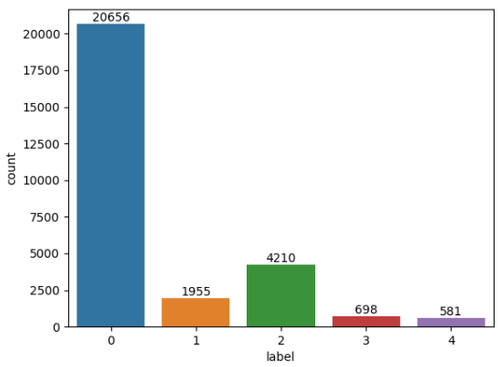


從實驗的結果來看，有用 pretrain weight 的版本表現明顯比 train from scratch 的還要好很多。Train from scratch 的版本雖然 loss 都有在變化，但變化幅度都很小，不管是 trainig 或是 testing 的 accuracy 幾乎變化都不大，都卡在 73% 左右。

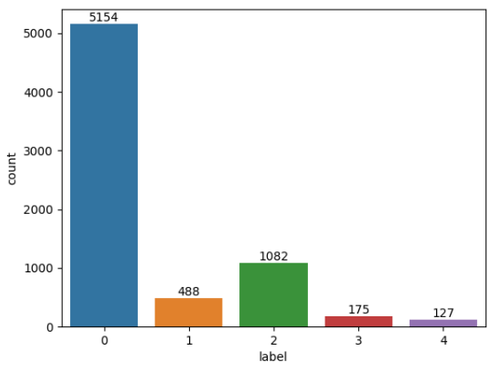
後來去看了一下 dataset 的 class 分布，發現資料的分布蠻不平均的，大部分都是 class 0 的資料，佔了全部的 73.5% 左右。發現說 train from scratch 的版本大概是放棄訓練了，全部都猜 0 的結果至少就會答對 73% 左右，從 confusion matrix 中就可以看到，不管 input 的資料是甚麼，乾脆全部猜 0。

而 pretrain 的版本卻能夠再繼續往下學習，可以從 confusion matrix 的結果中觀察到，每個 label 都還是有一定程度的準確度，感覺如果多增加一點 training data，他應該能夠做的更好。

- *Training data* distribution



- *Testing data* distribution



5. Discussion

Transfer Learning

在這個部分，我想試著用 transfer learning 的技巧看看可以把 model 訓練的怎麼樣，大概分成一下這兩種情況:

- 固定 ResNet18 與 ResNet50 前面 layer 的參數，只 train 最後的 fully connected layer

```
for param in Resnet18.parameters():
    param.requires_grad = False
```

- 將縮小圖片到 224 * 224，再做 transfer learning (原始 ResNet 的 training data 都是在 224 * 224 資料下訓練的，所以猜想 pre-train 的那些 kernel filters，也許在同樣解析度底下表現得比較好)

```
transforms.Resize((224, 224)),
```

實驗結果如下，發現還是原本 512 * 512 的表現比較好，也許 512 * 512 的影像保留了比較多的細節

	ResNet18	ResNet50
w/o Resize	75.59%	76.47%
Resize	74.34%	74.89%

Fine tuning

- 縮小圖片到224, 224，再 fine tune (全部重train)

Training Acc	Testing Acc
80.12%	81.74%

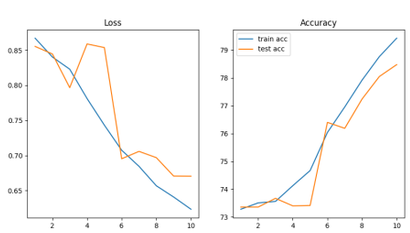
Effectiveness of Pre-train Model

為了驗證為甚麼有 pre-train 過的 model 能夠表現得比 train from scratch 的 model 還要好，我這裡就簡單做了一下實驗:

- 先將 ResNet18 拿去 train CIFAR10，簡單訓練 10 個 epoch 之後，把 model 的 weight 存下來
- 接著在訓練 Retinopathy Dataset 之前，把剛剛的 weight 讀進來，當作 model 的 initial weight
- 後續就跟一般 training 一樣，跑完整個 training process

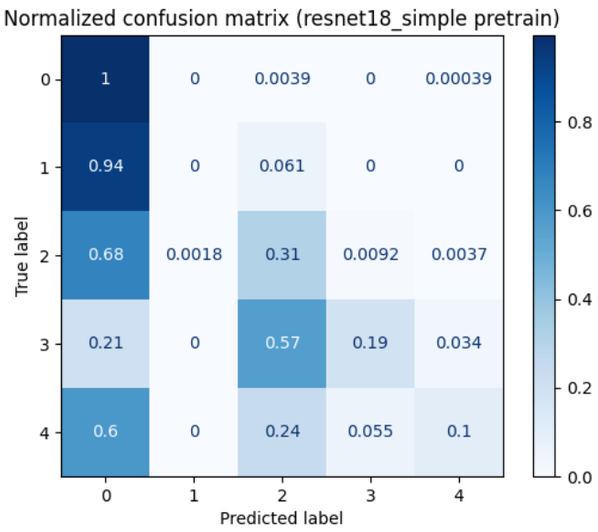
實驗結果如下:

Training Acc	Testing Acc
79.43%	78.48%



我認為是因為即使只有在 CIFAR10 上面 train 10 個 epoch，但 model 已經學會了一些 feature extractor，能夠學到要對圖片取哪些特徵，有可能是簡單的線條、局部的輪廓等等。接著如果再拿去訓練這次 lab 的醫療影像，就好比讓 model 有了一些取影像特徵的知識，而不至於讓 model 直接放棄訓練，連 filter 都不學，反正全部猜 0 就能答對 73%。

從下面的 confusion matrix 上也可以很明顯地看到，即使只是簡單的先在 CIFAR10 上面 pretrain 10 個 epoch，結果卻跟原本 without pretrain (全部猜0) 的差異很大。從結果來看這招的確讓 model 不管在 training 或是 testing accuracy 上都能夠跳出原本的 73%，而且還有持續上升的趨勢。



Explainable AI (Verifies where did the model focus on)

在這個部分，我透過 class activation mapping 的方式去把 model 最後一層 convolution 的 output 拿出來看 model 實際上是把注意力放在影像中的哪一個區塊，也去驗證一下說 pretrain 與 non-pretrain 版本的 model 差異在哪。可以從下方看到說 non-pretrain 版本的就如同之前說的放棄訓練一樣，他並沒有真的對影像去取特定的特徵，反而把注意力就直接放在整張影像上；反觀 pretrain 版本的 model，就確實把注意力集中放在影像當中的某的 pattern 上，有嘗試努力的在做學習。