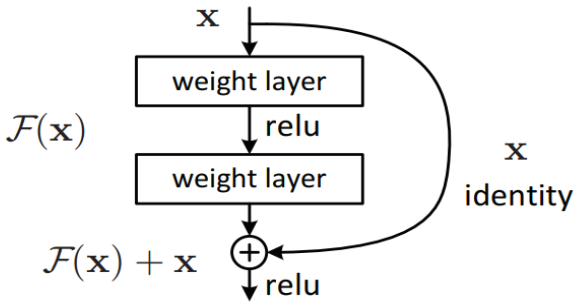
DLP_LAB3_309551124_涂景智

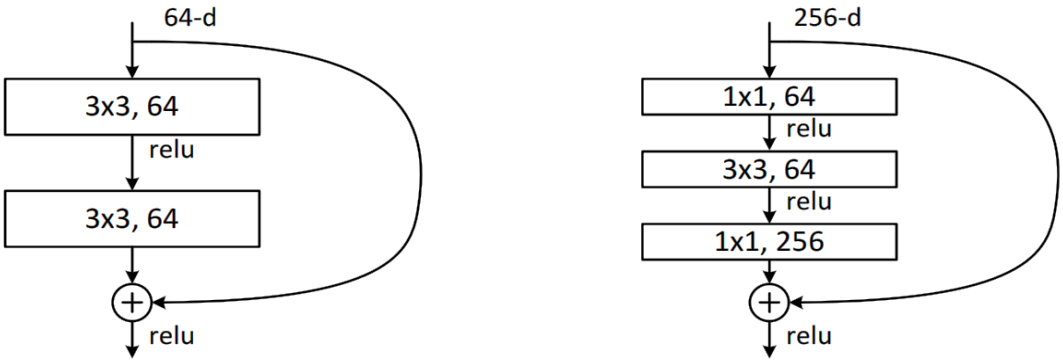
1. Introduction

在 Lab3 中，我們需要實作 Resnet。Resnet 是一種透過增加模型層數來提高模型表現的深度學習方法。相較於其他高層數的機器學習模型，Resnet 透過殘差學習的方式，來降低 Degradation 產生的問題。



Residual Block

另外，我們也可以根據模型深度的不同，而使用不同層數的 Residual Block。譬如我們這次需要實作的 Resnet18 和 Resnet50，就是使用了不同的二種 Block 作為 Residual Block。



上方左圖為 Basic block，residual mapping 為兩個 64 通道的 3x3 卷積，輸入輸出均為 64 通道。該 block 主要使用在相對淺層網路，比如我們這次要實作的 Resnet18。

而上方右圖為針對深層網路提出的 block，稱為 Bottleneck block，主要目的是為了降維。。先通過一個 1x1 卷積將 256 維 channel 降到 64，最後通過一個 256 channel 的 1x1 卷積恢復。該 block 主要使用在相對深層網路，比如我們這次要實作的 Resnet50。

2. Experiment setups

A. The details of your model (ResNet)

a. Basic Block

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

    def __init__(self, inplanes, planes, stride=1, downsample=None, groups=1,
                 base_width=64, dilation=1, norm_layer=None):
        super(BasicBlock, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d

        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = norm_layer(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

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

        out += identity
        out = self.relu(out)

        return out
```

b. Bottleneck Block

```python
class Bottleneck(nn.Module):
    expansion = 4
    def __init__(self, inplanes, planes, stride=1, downsample=None, groups=1,
                 base_width=64, dilation=1, norm_layer=None):
        super(Bottleneck, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        width = int(planes * (base_width / 64.)) * groups

        self.conv1 = conv1x1(inplanes, width)
        self.bn1 = norm_layer(width)
        self.conv2 = conv3x3(width, width, stride, groups, dilation)
        self.bn2 = norm_layer(width)
        self.conv3 = conv1x1(width, planes * self.expansion)
        self.bn3 = norm_layer(planes * self.expansion)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

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

        out = self.conv3(out)
        out = self.bn3(out)

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

        out += identity
        out = self.relu(out)

        return out
```

c. Resnet

```
class ResNet(nn.Module):

    def __init__(self, block, layers, num_classes=5, zero_init_residual=False,
                 groups=1, width_per_group=64, norm_layer=None):
        super(ResNet, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        self._norm_layer = norm_layer

        self.inplanes = 64
        self.dilation = 1

        self.groups = groups
        self.base_width = width_per_group
        self.conv1 = nn.Conv2d(3, self.inplanes, kernel_size=7, stride=2, padding=3,
                               bias=False)
        self.bn1 = norm_layer(self.inplanes)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(block, 64, layers[0])
        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 _resnet(arch, block, layers, **kwargs):
    model = ResNet(block, layers, **kwargs).to(device)
    return model


def resnet18( **kwargs):
    return _resnet('resnet18',  BasicBlock, [2, 2, 2, 2], **kwargs)



def resnet50( **kwargs):
    return _resnet('resnet50', Bottleneck, [3, 4, 6, 3], **kwargs)
```

比較要特別注意的是，當我們呼叫 resnet18() 和 resnet50() 時，我們除了要選定我們是要使用 Basic Block 還是 Bottleneck Block 之外，我們還需要把每一「大層」需要使用的 Block 數傳入 Resnet（譬如圖中的 [2, 2, 2, 2] 和 [3, 4, 6, 3]）。

要傳入的 Block 數是參考下圖：

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

最後，在 Pretrained Model 的部分，也要記得更改 Resnet18 和 Resnet50 的 Fc 層：

```
resnet50.fc = nn.Linear(2048, 5)

resnet18.fc = nn.Linear(512, 5)
```

B. The details of your Dataloader

在 Dataloader 的 「__getitem__」中：

```
path = self.root + self.img_name[index] + '.jpeg'

pre_img = Image.open(path)
data_transform = transforms.Compose([
transforms.RandomGrayscale(p=0.2),
transforms.RandomHorizontalFlip(p=0.5),
transforms.RandomVerticalFlip(p=0.5),
transforms.ToTensor(),
transforms.Normalize(mean = (0.5, 0.5, 0.5), std = (0.5, 0.5, 0.5))
])

img = data_transform(pre_img)
label = self.label[index]
```

將 index 對應的圖片名字加上 root 的名稱後，用 PIL 的套件載入該圖片。將其以一定機率灰階或水平垂直選轉後，將其轉乘 Tensor 並標準化。隨機選轉和灰階的目的在於可使不同的圖片產生，某種程度上也代表了 Training data 的增加。

另外，觀察 Train Data 的 Label，發現 Class 0 出現的機會比其他 Class 多得多（Label 為 0 的有 20655 個，1 有 1955 個，2 有 4210 個，3 有 698 個，4 有 581 個。）若直接將 Train data 做訓練會使模型全部都猜 0。為了改善這現象，我有刻意在 Dataloader 中儲存那些 Label 值不是 0 的 index：

```
self.nonezero = []
for i in range(len(self.label)):
    if self.label[i] != 0:
        self.nonezero.append(i)
```

當取資料的 index 超過原本的 Data 的長度（在訓練時，我有將會取得到 index 設為原始 Data 長度的 1.5 倍）時，我們就會從這些 Label 不為 0 的 index 裡取資料。

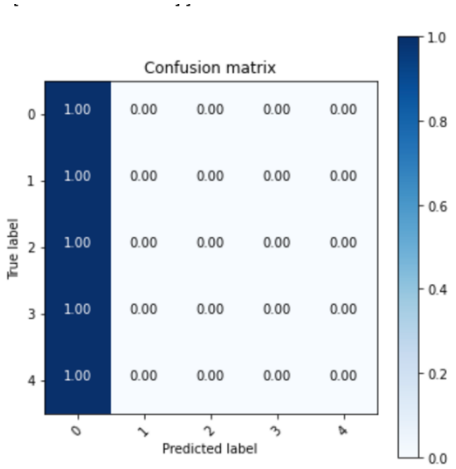
C. Describing your evaluation through the confusion matrix

a. Without Pretrained Data

在 Without Pretrained Data 的部分，無論是 Resnet 18 還是 Resnet50，因為模型出來的結果都是 0，因此 confusion matrix 都是長這樣：

```
array([[5153,    0,    0,    0,    0],
       [ 488,    0,    0,    0,    0],
       [1082,    0,    0,    0,    0],
       [ 175,    0,    0,    0,    0],
       [ 127,    0,    0,    0,    0]])
```

Normalize 後，畫成 Heat Map 就是：



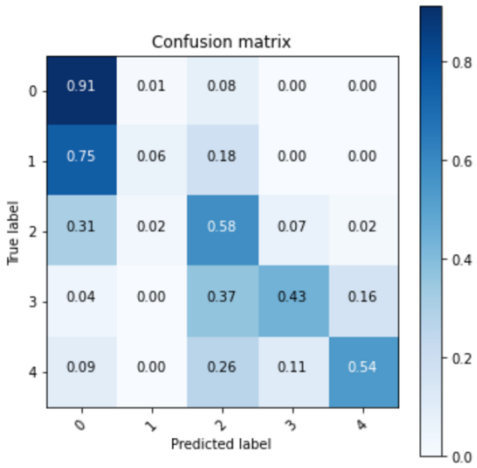
b. Test with Pretraining

(1) Resnet18 (Accuracy = 0.8015)

Confusion Matrix:

```
array([[4694,   41,  400,    3,   15],
       [ 367,   30,   90,    0,    1],
       [ 334,   24,  629,   73,   21],
       [   7,    0,   64,   76,   28],
       [  12,    0,   33,   14,   68]])
```

Normalize & Heat Map:

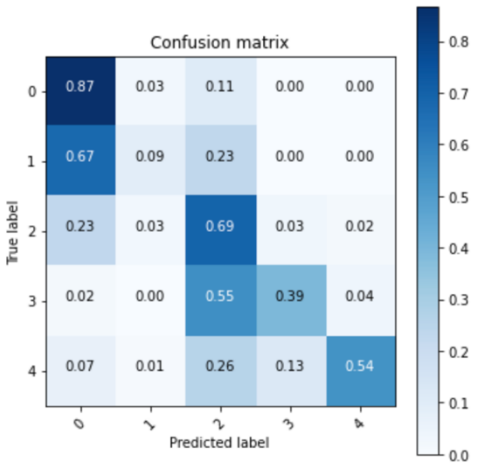


(2) Resnet50 (Accuracy = 0.7826)

Confusion Matrix:

```
array([[4462,  133,  546,    1,   11],
       [ 329,   44,  114,    0,    1],
       [ 249,   29,  751,   35,   17],
       [   3,    0,   96,   69,    7],
       [   9,    1,   33,   16,   68]])
```

Normalize & Heat Map:



3. Experimental results

A. The highest testing accuracy

☐ Screenshot

	Without Pretrained	With Pretrain
Resnet18	73.16%	80.15%
Resnet50	73.16%	78.26%

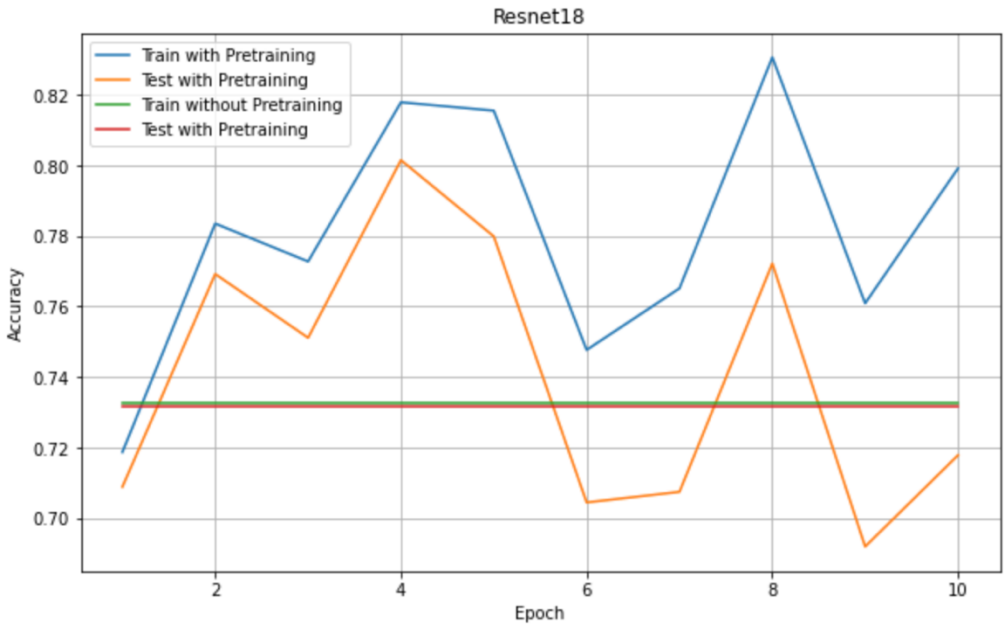
最高的 testing accuracy 為 Resnet18 With Pretrained 的 Model：

```
Epoch: 4, Train Accuracy = 0.8179456150341685, Test Accuracy = 0.8015375854214123
```

B. Comparison figures

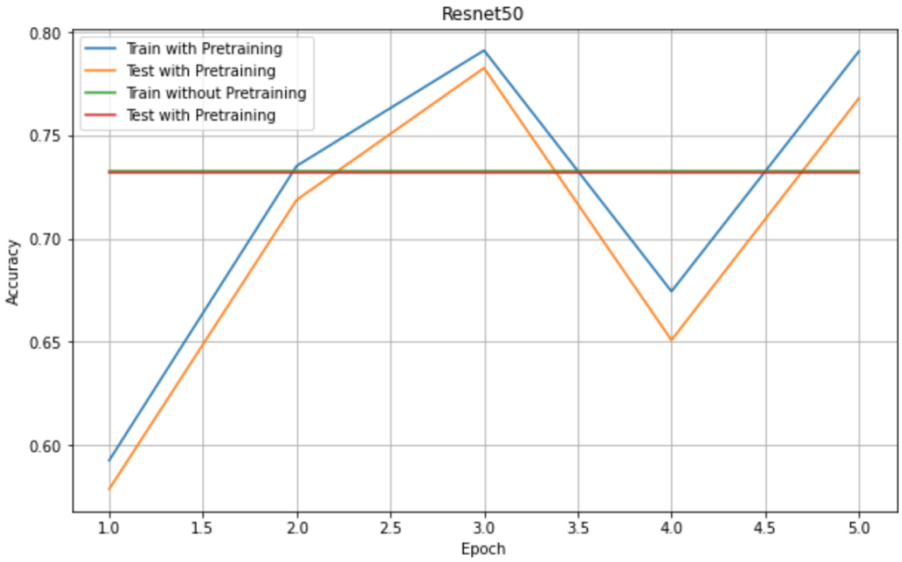
☐ Plotting the comparison figures (RseNet18/50, with/without pretraining)

a. Resnet18



結果毫不意外的顯示， Train with Pretraining 的準確率最高，可達大約 0.82，而 Test with Pretraining 的最高結果為 0.8015 左右。至於沒有 Pretrained 的 Model，無論是 Test Data 還是 Train Data 都是全猜 0，準確率分別為 0.7316 和 0.7325。

b. Resnet50

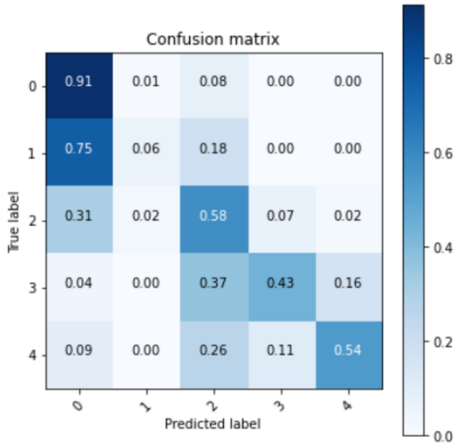


一樣是 Train with Pretraining 的準確率最高，可達大約 0.79。比較特別的是，Pretrained 的 Model 剛開始的 Accuracy Rate 都相當低，大約都只有 58% 左右而已。

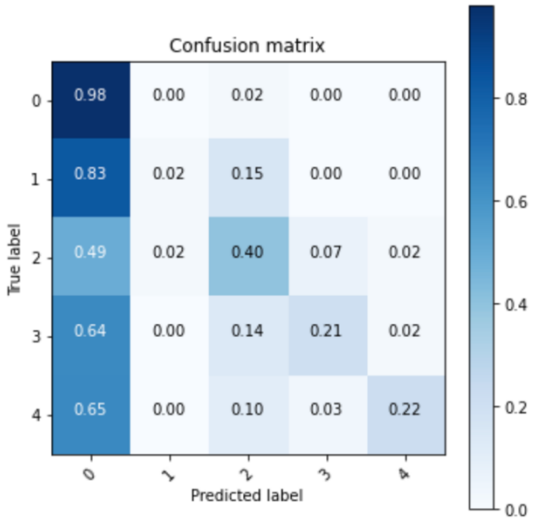
4. Discussion

前面有提過，因為 Data Set 本身 Label 為 0 的資料筆數眾多，所以我有透過增加 Label 非 0 的 image 被 Training 的次數，來試圖避免讓模型將預測過多的 Label 為零。在此比較「有 / 無透過增加 Label 非 0 的 Data 被 Training 的次數」的模型表現差異。

在 Resnet18 中，若有增加 Label 非 0 的 Data（確切的說，我將 Label 非 0 的訓練次數增加為原本的 7 倍），則模型的結果的 Confusion Matrix 如右圖（與前面相同）：



然而，若無增加 Label 非 0 的 Data 訓練數，則出來的 Confusion Matrix 為：



可以發現 Label 被誤判為 0 的結果顯著的增加了。雖然這並不一定代表增加 Label 非 0 的 Data 訓練數 會使 Accuracy 增加（因為 Label 為 0 的 Data 有高達 73%，比例相當的高），但至少證明這種做法對於提高 F1-score，或是 AUC 會有幫助。