1. Introduction

此次 lab 重點在於解決神經網路太多層時，所引發的 gradient vanish 問題。我們將實做 ResNet18 與 ResNet50，並與 pretrained 過的 model 比較成果差別。另外我們還會實做 pytorch 的 Dataloader，以及利用 sklearn 的 confusion matrix 幫助觀察結果。

2. Experiment setups

A) **model detail**

ResNet 18:

如右圖，先經過 convolution 後，batch norm 再進入 relu function。
接著進入四次 layer，每個 layer 皆由兩個 basic block 組成。
四次之後再丟到 average pool，最後 fully connect 輸出類別可能高低。

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

    out = self.layer1(out)
    out = self.layer2(out)
    out = self.layer3(out)
    out = self.layer4(out)

    out = self.avg_pool(out)
    out = out.view(out.size()[0], -1)
    out = self.full(out)
    return out
```

而 Basic Block 則如右下圖，每次都將進行兩次 convolution network，而主要重點有兩點
一是倒數第三行，在 output 輸出前加入了前方的 input，如此可防止 gradient vanish
二是如果輸出與輸入維度不同時，需再額外加入 down sample 來使彼此可以相加
如此就完成 ResNet 18 了

```
out = self.conv1(x)
out = self.bn1(out)
out = self.relu(out)
out = self.conv2(out)
out = self.bn2(out)
if self.down_sample is not None:
    res = self.down_sample(x)
else:
    res = x
out += res
out = self.relu(out)
return out
```

ResNet 50:

基本架構如右圖，可以看到跟 ResNet 18 整體架構非常相像。差別在於其中的維度，以及 layer 的配置

四個 layer 分別是由 Bottleneck 重複 [3, 4, 6, 3] 次，且每次 layer 的輸出皆為輸入維度的 4 倍
Bottleneck 的配置如右下圖，每次都將經過三次 convolution network，再將 intput 維度 down sample 成跟 output 一樣後相加就完成了。詳細的配置可以參考下方圖片

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

    out = self.layer1(out)
    out = self.layer2(out)
    out = self.layer3(out)
    out = self.layer4(out)

    out = self.avg_pool(out)
    out = out.view(out.size()[0], -1)
    out = self.full(out)
    return out
```

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

```
def forward(self, 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.down_sample is not None:
        res = self.down_sample(x)
    else:
        res = x
    out += res
    out = self.relu(out)
    return out
```

B) **Dataloader detail**

Dataloader 主要分三個部份，initial, len 以及 get_item，分別如下

initial 功能在於準備好所有會用到的東西，例如從 csv 檔讀取圖片名稱以及其 label，或是準備好之後要對 data 做的 normalize 或是 flip

```
def __init__(self, root, mode, batch_size=1):
    self.root = root
    self.img_name, self.label = getData(mode)
    self.mode = mode
    self.batch_size = batch_size
    self.length = int(len(self.img_name))
    self.transform_0 = transforms.Compose([
            transforms.ToTensor(),  # range [0, 255] -> [0.0,1.0]
    ])
    self.transform_n0 = transforms.Compose([
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),
    ])
    print(f">> Found {self.length} {mode} images")
```

len 就是很單純的回傳圖片個數

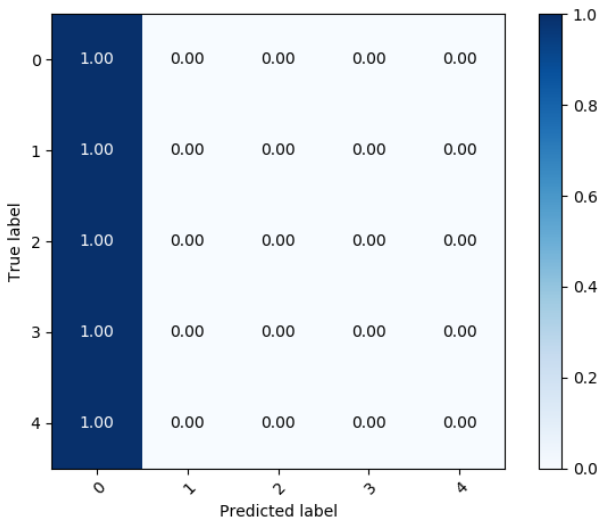
```
def __len__(self):
    return self.length
```

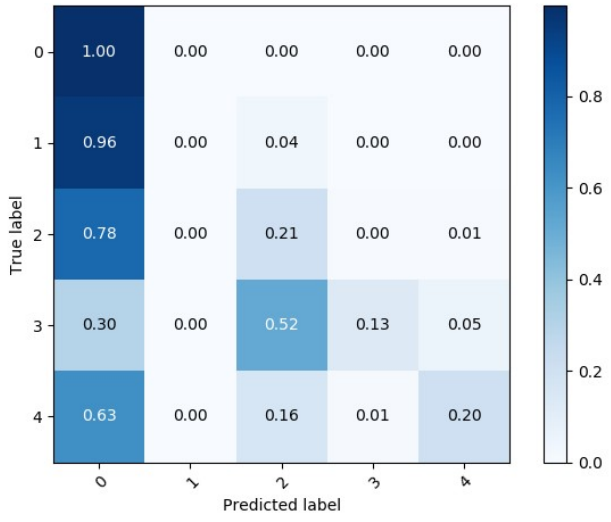
getitem 則是每次在 iterate 此 dataloader 時會呼叫的函式，這邊先用 Image 以 RGB 形式開啟之後，再對其做 normalize，另外我還額外對 label 非零的 data 做了 flipping 使其 data 個數可以更多

```
def __getitem__(self, idx):
    label = self.label[idx]
    path = os.path.join(self.root, self.img_name[idx] + '.jpeg')
    img = Image.open(path).convert('RGB')
    if label == 0 or self.mode=='test':
        img = self.transform_0(img)
    else:
        img = self.transform_n0(img)
    return img, np.array(label)
```

C) evaluation with confusion matrix

首先是 ResNet 18



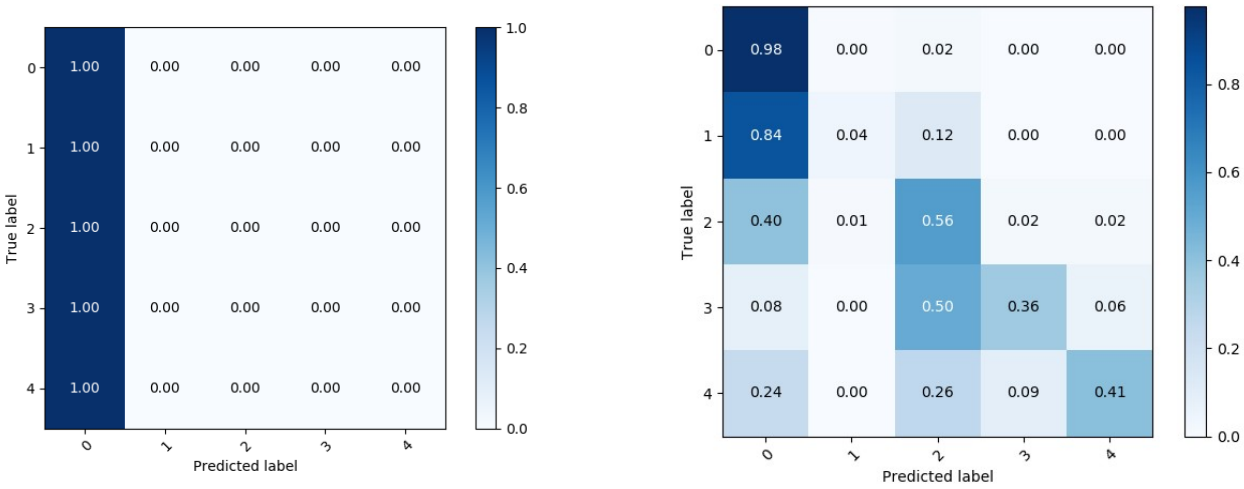


左圖是 without pretrained model 在 testing 上的表現，右圖則是 pretrained model

可以看到 without pretrained 會因為 training data 裡面 0 太多而使 model 只會 predict 0 也因此表現的不好

而 pretrained 則能跳過這個限制，嘗試其他可能，因此可以訓練的上去

再來是 ResNet 50



在左圖，without pretrained 一樣會陷入全部 predict 成 0 的結果

而右圖有 pretrained 的 ResNet 50 則能很好的，但一樣可以看到在 class 1 上表現實在不是很好，不過相較於 ResNet 18 以及 without pretrained 的 ResNet 50 來說都算好很多了

3. Experimental results

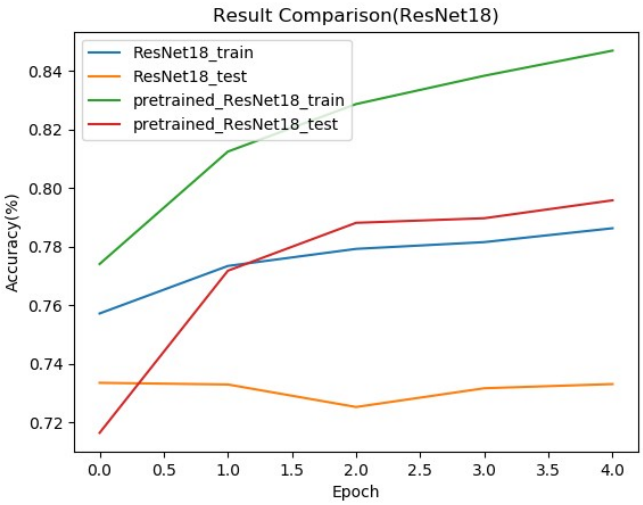
A) Highest accyract: 0.8200711743772242

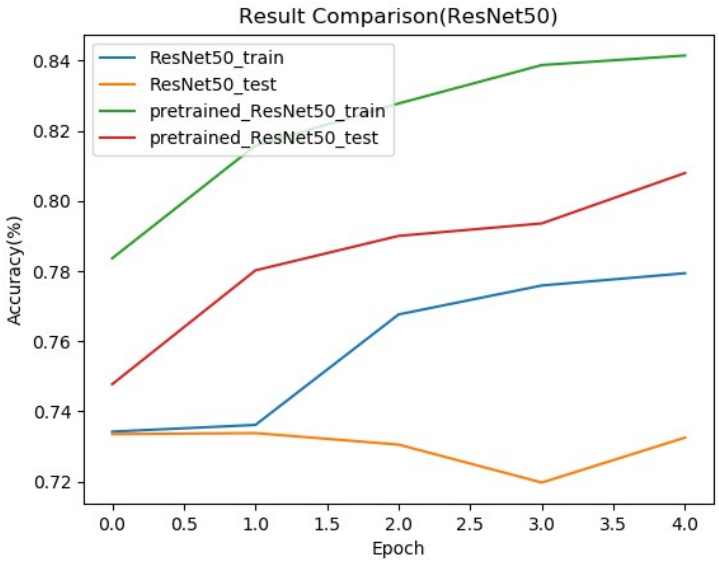
```
(kk) karljackab@pc3421:~/DL/lab3$ python3 lab3_0516003.py
use Cuda
>> Found 28099 train images
>> Found 7025 test images
pretrained ResNet50 testing accuracy: 0.8200711743772242
```

這個超過 0.82 的 model 實在是訓練非常的辛苦，首先我先觀察 training data 的 label，發現 label 為 0 的 data 多達 20655 個，相較於其他 label 多了非常多，因此我對非零的 training data 做 0.5 機率性的水平翻轉，使其算是增加了 training data 的數量。

效果也還不錯，在 Epoch 5 時，testing accuracy 就達到了 0.8078。但後面升上去就很困難了，我試了很多 Learning rate, batch size, weight_decay，以及後期時加入 label 0 的翻轉與接連的 training，才在作業 deadline 當天達到這個成績。實在是一項很令人印象深刻的作業。

B) Comparison figures





3. Discussion

這次作業非常的印象深刻，包含：

1. 硬體資源的重要性：好的顯卡可以讓我們有更多犯錯的機會，差的顯卡就要自求多福了
2. pretrained model 的重要：早些時候還在為 without pretrained 都 train 不上去而煩惱，想著到底是哪邊寫錯了，在助教寄信說這是正常的現象時才安心的繼續做下去
3. 了解 data 很重要：這次的 training data label 分佈非常不均勻，如果不是因為當時 without pretrained model 都上不去，而去看裡面的 data，我也不會想到要解決非 0 太多的問題
4. learning rate 在不同階段可以有很大的差別：這個作業中，我試過前面 learning rate 很小，然後 train 很多次，卻發現 train 不起來。或是 learning rate 適中，但到後期就上不去了。因此我才會分段去 train，越到後期越觀察有沒有開始往下掉而調整。

其他的當然還有 ResNet 的建制，以及 dataloader 的寫法，這次作業真是太棒了，學到很多。