# Deep Learning Lab4 – Diabetic Retinopathy Detection

**智能所 311581006 林子淩**

## 1. Introduction

糖尿病視網膜病變(Diabetic Retinopathy)是由糖尿病導致的併發症之一，由於高血糖會造成視網膜受損，如果不及時治療可能會導致失明等嚴重後果，因此檢測這種疾病成為一項重要的任務。
本次 lab 主題為實作 ResNet 模型來分類視網膜照片，協助糖尿病視網膜病變嚴重程度的檢測。過程包含以 PyTorch 框架完成客製化 DataLoader 撰寫、利用 torchvision 現成的 ResNet18 和 ResNet50 模型執行分類任務、比較兩種模型在使用以及不使用 pretrain weight 設定下的分類效果，並計算混淆矩陣(Confusion Matrix)以評估其分類效能。

## 2. Experiment Setup

### A. The details of your model (ResNet)

ResNet（Residual Network）是於 2015 年提出一種深度卷積神經網絡(Deep Convolutional Neural Network)。通過在幾個 layer 之後添加 input 和 output 之間的 skip connection，增加更深、層數更多的 Neural Network 訓練可能性。這種方法不僅讓 network optimization 過程更加順利，更因為後層有 skip connection 這個捷徑可以直接將梯度傳達到前面的 input，緩解了梯度消失/爆炸(gradient vanishing, exploding)問題，使得深層神經網路的效能提升。
如 Figure 1 所示，ResNet 可以分成用 basic block 或 bottleneck block 結構進行疊加，ResNet18 採用了 basic block，ResNet50 因為參數較多採用 bottleneck block。在 basic block 中，每一個 block 裡面都進行了兩次 convolution, batch normalization 及 ReLU activation，而在第二次 ReLU 前會增加一條路徑讓 block 的 output 可以直接 backpropagate 回 input，避免梯度消失(gradient vanishing)，這個設計使得多個 block 可以通過疊加來建構深層神經網絡，且不會因梯度問題降低模型表現。在 bottleneck block 中，每個 block 共進行三次 convolution，其中 1x1 的 convolution 相當於對 channel 做線性轉換，具有 rank reduction 的效果，可以降低整體運算量，讓參數量更多的 ResNet50 也可以訓練成功。

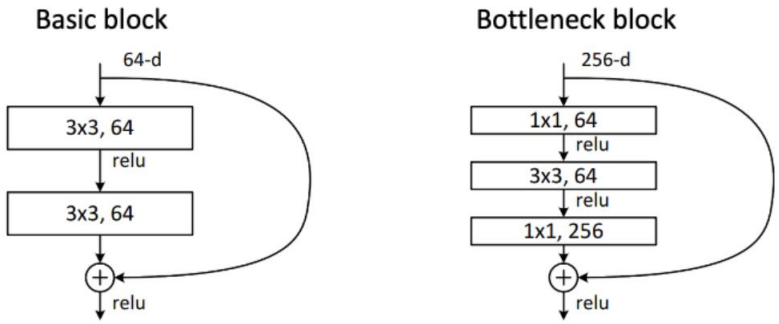



Figure 1　ResNet 內部架構(Basic block / Bottleneck block)

在本次 lab 中沒有要求必須自己搭建 ResNet 模型，因此我從 torchvision 中引入現成的 ResNet18 和 ResNet50 模型架構，並使用一個 bool 變數 `pretrain_flag` 來決定是否要載入 pretrain weight。考量一致性，針對兩種模型我都選擇使用 pretrain 在 ImageNet-1K 資料集上的 pretrain weight。
另外，ResNet 模型的最後是一層 output features 數量為 1000 的 fully-connected layer (因為 ImageNet-1K 資料集裡面共有 1,000 個分類 label)。然而本次我們要做的視網膜分類的資料集中只有 5 個

label。因此需要重新初始化最後一層(self.resnet.fc)，更改其 output features，以適應本次任務。但在模型建構中，我認為直接將 output features 數量從 1,000 降到 5，中間 dimension 驟降可能會影響分類效果，因此我將原始 ResNet 模型最後一層(self.resnet.fc)的 output features 數量改為 50，後面再新增一層全連接層(self.fc2)將 output features dimension 轉為 5。
另外，在使用 pretrain weight 的設定之下，考慮到在訓練初期經過重新初始化的 linear layer (self.resnet.fc)和我自己新增的全連接層(self.fc2)沒有 pretrain weight 的初始參數，因此直接 Finetune 整個 ResNet 時，linear layer 需要調整的幅度較大，前面有 pretrain weight 的 layer 需要的調整幅度小，不利於整體訓練。因此我使用 set_param_requires_grad()函數將除了 linear layer 外的 layer weight 凍結起來，先在前幾個 epochs 做 feature extraction (只訓練 linear layer)，等到初步訓練結束後再 finetune 整個模型，增加訓練效率(這部分在 5. Discussion 會詳細說明)。
整體實作程式碼如 Figure 2 所示。

```
6   class resnet(nn.Module):
7       def __init__(self, model_type, pretrain_flag, num_classes):
8           super(resnet, self).__init__()
9
10          """ define model"""
11          if model_type=="resnet18":
12              if pretrain_flag==True:
13                  self.resnet = resnet18(weights=ResNet18_Weights.DEFAULT)
14                  self.set_param_requires_grad(self.resnet, True)
15
16              else:
17                  self.resnet = resnet18(weights=None)
18          elif model_type=="resnet50":
19              if pretrain_flag==True:
20                  self.resnet = resnet50(weights=ResNet50_Weights.IMAGENET1K_V1)
21                  self.set_param_requires_grad(self.resnet, True)
22              else:
23                  self.resnet = resnet50(weights=None)
24
25          """reinitialize last layer of model (output_dim = num_classes of this task)"""
26          self.in_features = self.resnet.fc.in_features
27          self.resnet.fc = nn.Linear(in_features=self.in_features, out_features=50)
28
29          """self-define layers"""
30          self.fc2 = nn.Sequential(
31              nn.ReLU(),
32              nn.Dropout(0.25),
33              nn.Linear(in_features=50, out_features=num_classes)
34          )
35
36      def forward(self, x):
37          out = self.resnet(x)
38          out = self.fc2(out)
39          return out
```

Figure 2　ResNet 實作程式碼

## B. The details of your Dataloader

本次 lab 要求實作客製化 Dataloader，可以分成初始化(__init__)、取得資料(__getitem__)這兩部分說明。如 Figure 3 所示，在初始化(__init__)中會設定初始參數，並使用定義好的 getData 函數從 csv 檔讀出 train/test 資料集中的圖片名稱(self.img_name)和對應 label (self.label)。另外，根據(self.label)中共有幾種不同分類標籤定出總分類 class 數量(self.num_classes)，

以便之後回傳作為模型分類 output features 數量的參考。

```python
class RetinopathyLoader(data.Dataset):
    def __init__(self, root, mode, data_path):
        """
        Args:
            root (string): Root path of the dataset.
            mode : Indicate procedure status(training or testing)
            data_path : Root path of the train/test_img.csv and train/test_label.csv

            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.label = getData(mode, data_path)
        self.mode = mode
        # number of classes in dataset
        self.num_classes = len(set(self.label))
        print("> Found %d images..." % (len(self.img_name)))
```

Figure 3　客製化 DataLoader 實作程式碼(初始化__init__)

Figure 4 為取得資料(__getitem__)函數的具體內容，當依照 batch size 將資料分割為 dataset 時，這個函數會被自動呼叫執行。根據提示步驟，首先找到圖片名稱 list 中給定 index 的該張圖片名稱(self.img_name[index])，結合路徑(self.root)找到開啟該圖片的檔案路徑。接著取得對應 label (self.label[index])作為分類 ground truth。在 step 3 用 python 套件 Pillow 開啟圖片(img)，用 torchvision.transforms 對圖片依序做多個轉換處理(詳細在 3. Data Preprocessing 中描述)。最後將圖片轉換成 pytorch tensor，以便之後可以和模型一起放到 GPU 中加速訓練過程。最後，函數回傳處理過後的圖片(img)和對應 label (label)。

```python
    def __getitem__(self, index):
        """something you should implement here"""

        """..."""
        ImageFile.LOAD_TRUNCATED_IMAGES = True
        # step 1
        path = os.path.join(self.root, self.img_name[index]+'.jpeg')
        # step 2
        label = float(self.label[index])
        # step 3
        img = Image.open(path)

        transform = transforms.Compose(
            [
                transforms.Resize(512),
                # crop to same size
                transforms.CenterCrop([512, 512]),
                transforms.RandomHorizontalFlip(0.5),
                transforms.RandomVerticalFlip(0.5),
                # ToTensor() convert pixel value to [0,1] and transpose to [C,H,W]
                transforms.ToTensor(),
                transforms.Normalize((0.3749, 0.2602, 0.1857), (0.2526, 0.1780, 0.1291)),
            ]
        )
        img = transform(img)
        return img, label
```

Figure 4　客製化 DataLoader 實作程式碼(取得資料__getitem__)

## C. Describing your evaluation through the confusion matrix

Figure 5 是 plot_confusion() 函數的實作程式碼，函數以真實分類 label 和模型預測結果作為輸入，最後輸出 confusion matrix 結果。由於本次任務資料集包含 5 個分類類別，因此會以形狀為(5,5)的 2D array 來表示 confusion matrix。具體來說，對於每個$i, j \in 0,1,2,3,4$，confusion matrix 中的第(i, j)

項表示 label 為 i 類別，但被模型分類為 j 類別的資料數量。通過循環比較每一個 ground truth label 和 model prediction 可以完成矩陣繪製。

另外，基本的 confusion matrix 以資料數量記錄每一格，經過 normalize 之後則變成表示相對比例或機率，可以更直觀地看到模型在不同類別之間的預測錯誤率，幫助瞭解模型在不同類別上的表現。

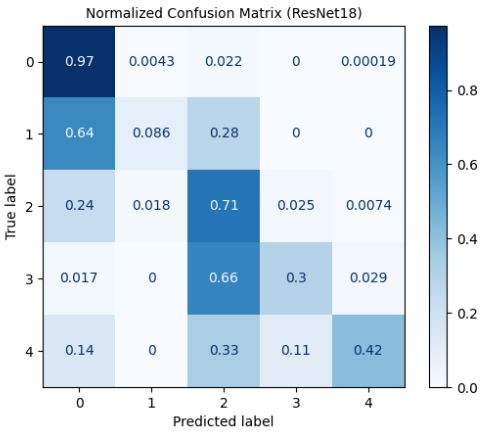
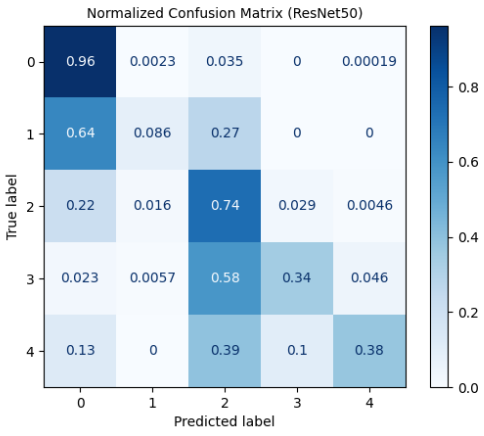
```python
def plot_confusion(args, pred, label, num_classes):
    """plot confusion matrix"""
    save_path = args.save_path
    name_dict = {'resnet18': 'ResNet18', 'resnet50': 'ResNet50'}

    if args.pretrain_flag == True:
        name = "{}_{}_{}".format(args.model_type, "pretrained", str(args.learning_rate))
    else:
        name = "{}_{}_{}".format(args.model_type, "none", str(args.learning_rate))
    cm = confusion_matrix(label, pred, labels=np.arange(num_classes), normalize='true')
    disp = ConfusionMatrixDisplay(confusion_matrix=cm)
    disp.plot(cmap='Blues')
    plt.title('Normalized Confusion Matrix ({})'.format(name_dict[args.model_type]), fontsize=10)
    plt.savefig(save_path + "/confusion_{}.png".format(args.model_type))
    plt.show()
    plt.clf()
```

Figure 5　繪製 confusion matrix 實作程式碼

Confusion Matrix 最後以圖表形式呈現，因為是 Normalize 過的結果，因此顏色深淺代表資料量比例，而非資料筆數。Table 1 為 confusion matrix 視覺化結果，可以看到在 with pretrain 的情況下，約 0.97 比例的資料為 label=0 且 prediction=0，最多分類錯誤為 label=1 分類成 prediction=0。可能是因為各 label 訓練資料數量不平衡的緣故，導致模型傾向預測結果為訓練資料數量較多的 label。整體來說，兩模型 ResNet18 和 ResNet50 的 confusion matrix 相差不大，同 column 比較之下大部分資料會落在對角線上(即分類正確，label=prediction)，可以看出 pretrain 讓模型有更好的參數起始點，有助於提升預測的準確性。

另一方面，without pretrain 的 confusion matrix 表現就不像 with pretrain 一樣好，可以看出兩種模型皆傾向於將所有資料分類到 class 0。同樣推測是因為訓練資料中類別不平衡，label 大部分為 class 0，因此模型訓練結果才會不夠 general，傾向於將大多數分類到 class 0。因為按照 data distribution，分類到 class 0 更容易會猜對(訓練時的資料量大)。這個問題可以通過針對 Imbalanced data 的處理(例如：downsampling major class data, oversampling tail class data, 調整 major/tail class 的 loss weight)來緩解，讓每個類別的資料特徵都有被模型充分學習的機會。

總體來說，with pretrain 比 without pretrain 的分類效果好，受到 Imbalance data 的影響較淺。

| With Pretrain | |
|---|---|
| **ResNet18** | **ResNet50** |
|  |  |

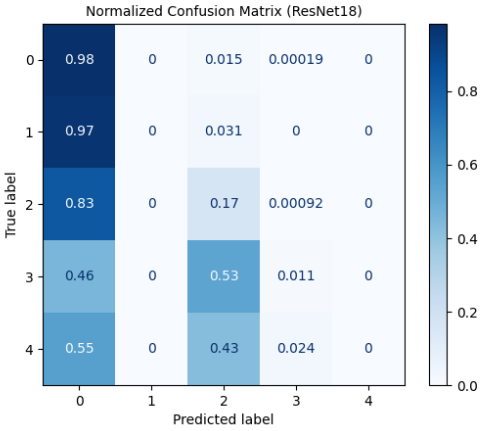
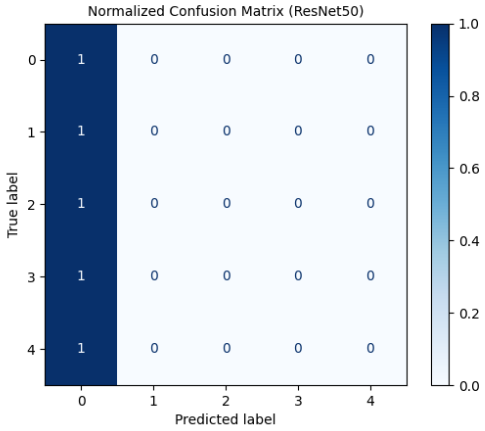
| Without Pretrain | |
|---|---|
| **ResNet18** | **ResNet50** |
|  |  |



Table 1　Confusion Matrix 視覺化結果

# 3. Data Preprocessing

## A. How you preprocessed your data?

在客製化 DataLoader 中，我首先使用 torchvision.transforms 套件依序對圖片做了 Resize, CenterCrop, RandomHorizontalFlip, RandomVerticalFlip 四種處理，再經過 ToTensor 和 Normalize 將圖片轉換成標準化的 torch.Tensor 資料型態。(實作程式碼如 Figure 4)

因為 ResNet 模型限制輸入圖片大小為[512,512]，Resize 的目的是為了讓原始圖片先按照等比例，將最短邊縮小成 512 (如 Table 2 中從[4552,3168]變成[768,512])。縮小後因比例關係寬度會超過 512，因此需要再通過 CenterCrop 將圖片裁剪成指定大小(如 Table 2 中從[768,512]變成[512,512])，以便輸入 ResNet 做運算。因為 CenterCrop 是取中心進行裁減，所以有很大的機率剛好可以將多餘的黑色邊框裁切掉，整個過程如 Table 2 所示。

另外，RandomFlip 處理則是為了增加模型的 robustness，增加模型對於不同角度的照片的判斷分類能力。後續使用 ToTensor 將 PIL 圖片轉成 torch.Tensor 形式，每個 pixel 從 0~255 縮放至 0~1，維度順序為[C, H, W]。Normalize 則負責將 Tensor 進行標準化處理，將 Tensor 中的數據轉換為 mean=0, std=1 的數據分佈。

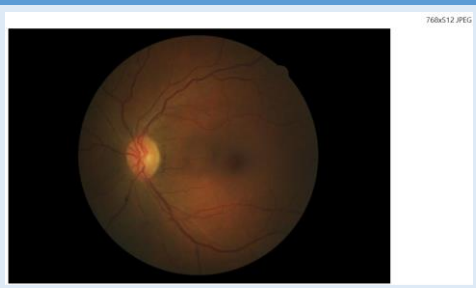
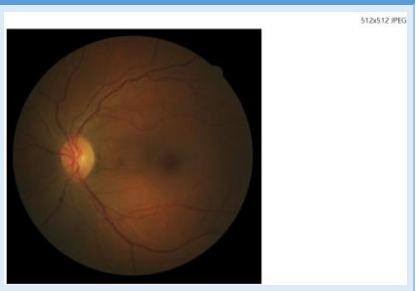
| Original<br>(size: [4752,3168]) | Resize<br>(size: [768,512]) | CenterCrop<br>(size: [512,512]) |
|---|---|---|
|  |  |  |

Table 2　圖片經過 Resize 和 CenterCrop 後的大小

## B. What makes your method special?

如前面所述，我首先使用 Resize, CenterCrop 將圖片調整到適合輸入模型的樣子，這樣做剛好可以將黑色邊框裁掉，減少多餘的雜訊(不包含特徵的部分)被輸入到模型中，避免影響分類預測的判斷。

RandomHorizontalFlip, RandomVerticalFlip 處理方法可以用來增加模型的 robustness，兩種操作各自有 0.5 的機率會發生(水平翻轉或垂直翻轉)，用來增加訓練資料的多樣性，讓模型學到在圖片翻轉但仍然保有原始特徵的情況下，還可以將其分類到正確 label 的能力。
另外還使用 Normalize，將 Tensor 中的數據轉換為 mean=0, std=1 的數據分佈。經過 Normalize 後，可以將數據分佈限制在一個較小的範圍內，提高模型訓練的穩定性和收斂速度。

# 4. Experimental results

## A. The highest testing accuracy

- **Screenshot**

Table 3 中包含了 ResNet18 和 ResNet50 兩個模型訓練過程的截圖(詳細參數設定見 Table 4)，記錄了整個 training 結果，包含 training loss, training accuracy, testing accuracy。注意這裡的兩個模型皆為使用 pretrain weight 的情況，同模型沒有使用 pretrain weight 的 test accuracy 會低約 10%左右。

| Highest testing accuracy | |
|---|---|
| **ResNet18** | **ResNet50** |
| model: resnet18, pretrain:True, lr: 0.001, bs:12<br>start training<br>epoch 0, total train loss = 1970.2425, train acc = 73.44%, test acc = 73.38%<br>epoch 1, total train loss = 1822.2690, train acc = 73.67%, test acc = 74.14%<br>epoch 2, total train loss = 1787.6213, train acc = 74.17%, test acc = 74.41%<br>epoch 3, total train loss = 1772.3634, train acc = 74.33%, test acc = 74.48%<br>epoch 4, total train loss = 1759.1394, train acc = 74.51%, test acc = 74.88%<br>epoch 5, total train loss = 1585.1519, train acc = 77.60%, test acc = 79.17%<br>epoch 6, total train loss = 1366.8799, train acc = 81.03%, test acc = 81.38%<br>epoch 7, total train loss = 1278.3099, train acc = 82.31%, test acc = 83.13%<br>epoch 8, total train loss = 1225.9624, train acc = 82.90%, test acc = 83.26%<br>epoch 9, total train loss = 1171.0716, train acc = 83.48%, test acc = 82.04%<br>epoch 10, total train loss = 1139.0606, train acc = 83.86%, test acc = 83.36%<br>epoch 11, total train loss = 1112.2099, train acc = 84.30%, test acc = 83.99%<br>epoch 12, total train loss = 1086.6739, train acc = 84.56%, test acc = 84.30%<br>epoch 13, total train loss = 1060.6569, train acc = 84.87%, test acc = 82.51%<br>epoch 14, total train loss = 1031.8625, train acc = 85.29%, test acc = 84.73%<br>epoch 15, total train loss = 1009.1438, train acc = 85.65%, test acc = 83.70%<br>epoch 16, total train loss = 994.8877, train acc = 85.61%, test acc = 83.52%<br>epoch 17, total train loss = 972.4981, train acc = 86.06%, test acc = 84.26%<br>epoch 18, total train loss = 961.5576, train acc = 86.25%, test acc = 84.43%<br>epoch 19, total train loss = 944.2298, train acc = 86.71%, test acc = 84.10%<br>best epoch 14, test acc = 84.73 | model: resnet50, pretrain:True, lr: 0.001, bs:6<br>start training<br>epoch 0, total train loss = 3925.1327, train acc = 73.48%, test acc = 73.35%<br>epoch 1, total train loss = 3689.0488, train acc = 73.53%, test acc = 73.40%<br>epoch 2, total train loss = 3637.0726, train acc = 73.66%, test acc = 73.51%<br>epoch 3, total train loss = 3626.9466, train acc = 73.77%, test acc = 74.11%<br>epoch 4, total train loss = 3600.3073, train acc = 73.94%, test acc = 74.33%<br>epoch 5, total train loss = 3271.7412, train acc = 76.96%, test acc = 80.78%<br>epoch 6, total train loss = 2819.4655, train acc = 80.62%, test acc = 81.14%<br>epoch 7, total train loss = 2608.7929, train acc = 81.87%, test acc = 82.98%<br>epoch 8, total train loss = 2499.0972, train acc = 82.46%, test acc = 83.49%<br>epoch 9, total train loss = 2415.6945, train acc = 83.15%, test acc = 82.90%<br>epoch 10, total train loss = 2350.1204, train acc = 83.53%, test acc = 83.67%<br>epoch 11, total train loss = 2298.5481, train acc = 83.80%, test acc = 83.56%<br>epoch 12, total train loss = 2260.3183, train acc = 84.13%, test acc = 83.91%<br>epoch 13, total train loss = 2227.5871, train acc = 84.33%, test acc = 84.31%<br>epoch 14, total train loss = 2177.2738, train acc = 84.52%, test acc = 83.47%<br>epoch 15, total train loss = 2159.1743, train acc = 84.63%, test acc = 83.76%<br>epoch 16, total train loss = 2114.2981, train acc = 84.94%, test acc = 83.77%<br>epoch 17, total train loss = 2087.6583, train acc = 85.28%, test acc = 82.89%<br>epoch 18, total train loss = 2083.5980, train acc = 85.06%, test acc = 83.83%<br>epoch 19, total train loss = 2057.2382, train acc = 85.08%, test acc = 83.59%<br>best epoch 13, test acc = 84.31 |

Table 3　ResNet18 和 ResNet50 的 highest test accuracy 截圖

- **Anything you want to present**

Table 4 是 ResNet18 和 ResNet50 兩個模型的訓練參數細節，按照實驗說明皆使用 SGD optimizer (Weight decay=5e-4, momentum=0.9)，Loss function 為 Cross Entropy。由於 CUDA memory 的限制，訓練 ResNet50 的時候，batch size 只能開到 6。如果能夠開到更大的 batch size 應該會有利於收斂，並達到更好的 test accuracy 結果。

| Hyperparameter | Model | |
|---|---|---|
| | **ResNet18** | **ResNet50** |
| **Learning rate** | 1e-3 | 1e-3 |
| **Batch Size** | 12 | 6 |
| **Training Epochs** | 20 | 20 |

Table 4　ResNet18 和 ResNet50 詳細訓練參數

Table 5 為實驗數據，包含四種結果(ResNet18 和 ResNet50 兩種模型以及是否使用 pretrain weight 的情況)。表格中紀錄 20 個 epochs 訓練中，出現過最高的 test accuracy。從表格中可以發現，有加載

pretrain weight 的模型表現比沒有使用 pretrain weight 的模型來的好，相較得到更高的 test accuracy 結果，皆可以達到超過 84%的分類準確度。相比之下，沒有使用 pretrain weight 讓兩種模型的 test accuracy 皆下降約 10%，可以推斷 pretrain weight 讓模型有好的訓練起始參數對結果影響很大。
回顧討論 confusion matrix 的部分，可以發現從頭開始訓練的模型(without pretrain weight)，會傾向將大多數資料往訓練 label 數多的類別去預測，造成比較低的 test accuracy 結果。如果能經過 Imbalance data 處理，或做其他圖片轉換增加模型的 robustness，或許可以提升整體模型的預測水準。

| Highest Test Accuracy | Model | |
|---|---|---|
| | ResNet18 | ResNet50 |
| With pretrain | **84.73%** | **84.31%** |
| Without pretrain | 74.73% | 73.35% |

Table 5　ResNet18 和 ResNet50 的 highest test accuracy 紀錄

## B. Comparison figures

- **Plotting the comparison figures**

Table 6 為兩種模型 accuracy learning curve，並且分為 with pretrain 和 w/o pretrain 兩種曲線。如同預期，沒有使用 pretrain weight 的模型 test accuracy 無法突破 75%，並且隨著 epoch 上升分類準確度也幾乎沒有變化。反之，使用 pretrain weight 的模型可以隨著 epoch 上升逐漸學習正確分類圖片，由於這次 lab 中我採用 feature extracting + finetuning 的方法來訓練使用 pretrain weight 的模型(詳細描述見 5. Discussion 中的 Feature Extracting and Finetuning)。因此可以觀察到當 epoch<5 時，train/test accuracy with pretrain 會先緩慢上升(此時只能更新後兩層的參數，因此上升幅度有限)。在 epoch=5 時會將模型中所有參數改成可更新狀態，因此可以看到 train/test accuracy with pretrain 會大幅上升。同時也能觀察到 train accuracy 在後期逐漸與 test accuracy 拉開距離，似乎有 overfitting 的情況發生。

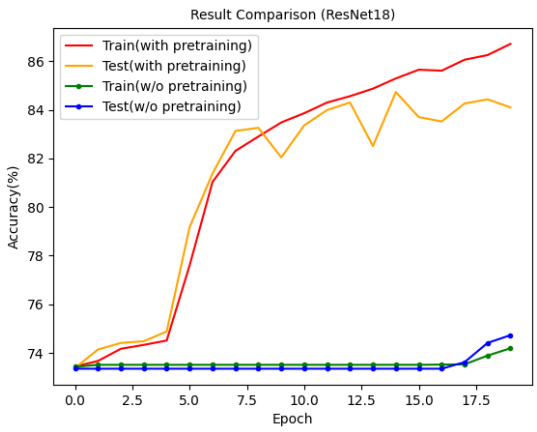
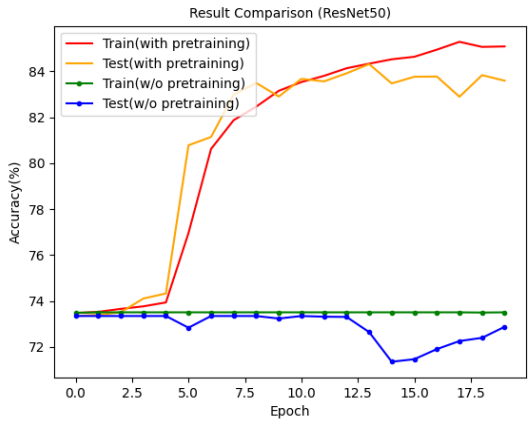
| Comparison Figure of Accuracy | |
|---|---|
| ResNet18 | ResNet50 |
|  |  |



Table 6　ResNet18 和 ResNet50 的 comparison figure (accuracy)

# 5. Discussion

## A. Anything you want to share

- **Feature Extracting and Finetuning**

回顧 Figure 2 實作模型建構，一開始從 torchvision 引入 ResNet 模型時，如果是使用 pretrain weight 的情況下，會先呼叫 Figure 6 中的函數 `set_param_requires_grad()` 來將 ResNet 中的參數先全

部設定為不進行 gradient 更新。而之後重新初始化的全連接層(self.resnet.fc)和新定義的全連接層(self.fc2)，因為是在設定 gradient 更新關閉後才重新定義的，所以 gradient 仍然會是 default 的需要更新狀態。此時允許更新 gradient 的參數只有 self.resnet.fc 和 self.fc2 裡的參數。這樣的作法是考慮到在訓練初期 linear layer self.resnet.fc 和 self.fc2 沒有 pretrain weight 的初始參數可以加載，因此如果直接 Finetune 整個 ResNet，linear layer 需要調整的幅度較大，前面有 pretrain weight 的 layer 需要的調整幅度小，不利於整體訓練。

```
def set_param_requires_grad(self, model, feature_extract):
    if feature_extract:
        for param in model.parameters():
            param.requires_grad = False
    else:
        for param in model.parameters():
            param.requires_grad = True
```

Figure 6　feature extraction 實作程式碼，控制要訓練的參數

如 Figure 7 所示，當 epoch<5 時，以 feature extract 的方法進行訓練，此時除了最後面的兩層 linear layer 參數外，其餘參數皆凍結無法進行 gradient 更新。當 epoch=5 時，此時 linear layer 參數已經經過初步訓練，有比較好的參數起始點，這時候再次使用 set_param_requires_grad() 函數將整個模型的參數皆設為允許 gradient 更新，稱作 finetune。並且調整 optimizer 需要更新的參數集合，以 finetune 模式訓練剩下的 15 個 epochs。

```
"""feature extract before finetune"""
if epoch == 5:
    self.model.set_param_requires_grad(self.model, feature_extract=False)
    optimizer = SGD(self.model.parameters(), lr=self.args.learning_rate,
                    momentum=0.9, weight_decay=5e-4)
```

Figure 7　從 feature extracting 改成 finetuning 模型

- **Layer Initialization**

Figure 8 是另一個特殊的操作，呼叫 class resnet 中 initialize_weights() 這個函數可以將 ResNet 模型中最後一層全連接層(self.resnet.fc)和自己定義的全連接層(self.fc2)的權重(weight)初始化到 Normal distribution 上，mean=0、std=0.01，bias 則初始化等於 0。
通過這個步驟可以讓 weight 在初始訓練階段有比較好的分布，增加這兩個 linear layer 的訓練穩定性，避免一開始 random weight 離更新目標太遠，導致不好收斂。

```
def initialize_weights(self):
    nn.init.normal_(self.resnet.fc.weight.data, 0, 0.01)
    self.resnet.fc.bias.data.zero_()
    nn.init.normal_(self.fc2[2].weight.data, 0, 0.01)
    self.fc2[2].bias.data.zero_()
```

Figure 8　初始化 Linear layer 的參數

- **Comparison figures of loss**

Table 7 為 ResNet18 和 ResNet50 的 loss comparison figure。可以看到使用 pretrain weight 的模型在

feature extracting 的階段(epoch<5)，loss 會先初步下降然後趨緩，對應 Table 6 中的 accuracy comparison figure 準確度初步上升然後趨緩。等到變成 finetuning 整個模型的模式時，loss 會再大幅下降並趨緩。然而因為訓練時間太長，沒有訓練到 20 個 epoch 之後，從圖形推斷 loss 應該還有下降的空間，從 Table 6 中的 accuracy comparison figure 也能看出 with pretrain 模型的 accuracy 還有上升空間。另外也能看出，沒有使用 pretrain weight 的 loss 趨於平緩，隨著 epoch 增加 loss 下降不明顯。

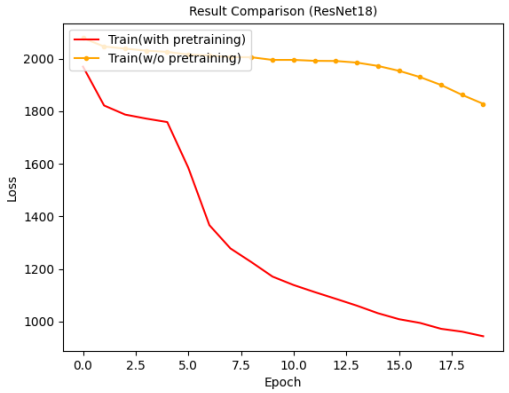
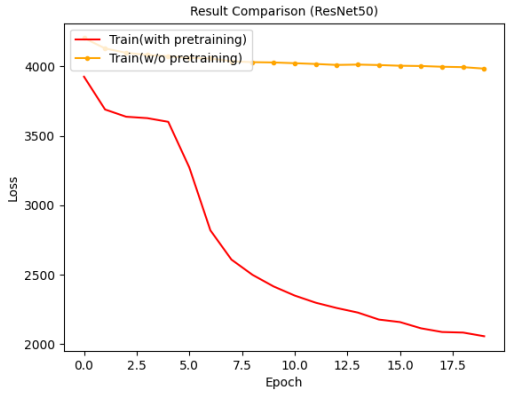
| Comparison Figure of Loss | |
|---|---|
| **ResNet18** | **ResNet50** |
|  |  |

Table 7　ResNet18 和 ResNet50 的 comparison figure (loss)