

## 311551069 余忠旻 Lab4 : Diabetic Retinopathy Detection

### 1. Introduction

這次作業我們要用 ResNet 來分析 diabetic retinopathy (糖尿病所引發視網膜病變)，而這次視網膜病變分類總共有五類，我們主要是用 ResNet18 和 ResNet50 這兩個 model 進行分類，這兩個 model 都在 torchvision 函式庫中宣告了，可以直接呼叫使用。並且在 dataloader 我們還需要實作 `getitem()` 的部分。它的功能不只是回傳圖片本身，更要做一些前處理，以增加模型預測的正確率，並且觀察 model 有無 pretraining 所帶來的影響。

### 2. Experiment setups:

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

ResNet 透過殘差學習(Residual learning)，來解決深度 CNN 模型難訓練的問題，尤其是 Degradation problem。

#### Degradation problem

上面所說的更深的網路有時反而帶來更差的效果，也就是出現了退化問題。當網路深度增加時，網路準確度出現飽和，甚至出現下降。這個現象可以在下圖中觀察出來：56 層的網路比 20 層網路效果還要差。這不會是 overfitting 問題，因為 56 層網路的訓練誤差同樣高。我們知道深層網路存在著梯度消失或爆炸的問題，這使得深度學習模型很難訓練。

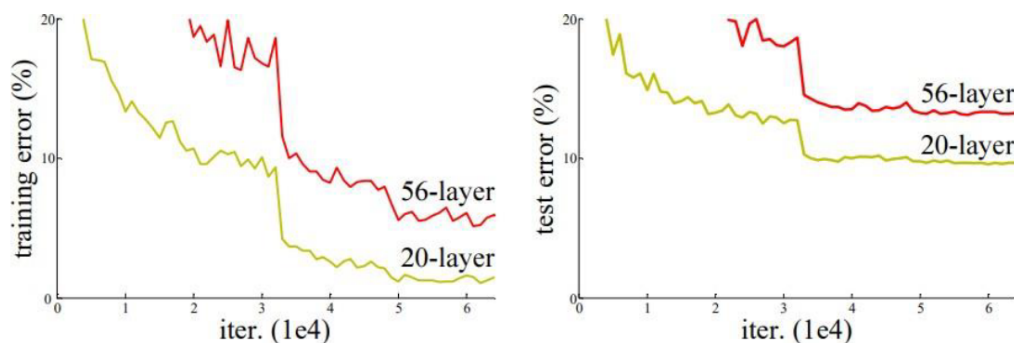
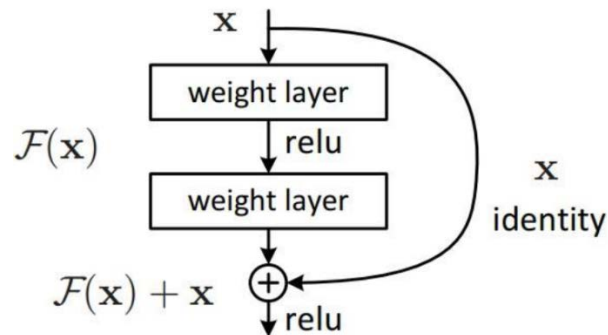


Figure. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error.

### Skip/Shortcut connection

為了解決梯度消失或爆炸的問題，添加了一個 skip / shortcut connection，在幾個 weight layers 之後將輸入  $x$  添加到輸出，如下圖所示



### Residual learning

Residual learning 主要就是解決上述所講深度網路所造成 Degradation problem，讓模型很難被訓練起來，Residual learning 的想法是當想要通過向上堆積新層來建構深層網路，一種極端狀況是這增加的層甚麼也沒學習到，只是複製之前網路的特徵，這稱新層為恆等映射(Identity mapping)，這種情況下，準確度至少會跟淺層網路一樣，不會出現退化問題。

如何將上述想法應用在 CNN 網路，就是用了 Skip/Shortcut connection 這個技巧，根據上圖，輸入為  $x$ ，學到的特徵為  $H(x)$ ，而殘差  $F(x) = H(x) - x$ ，將殘差也學習到，這樣其實學習到的特徵  $H(x) = F(x) + x$ ，這樣當殘差為 0 時，堆積層僅僅是做了 Identity mapping，能不讓模型產生 degradation problem，因此能更好訓練深層網路。

### Why ResNet can avoid vanishing gradient problem?

$$y_l = h(x_l) + F(x_l, W_l)$$

$$x_{l+1} = f(y_l)$$

$x_\ell$  和  $x_{\ell+1}$  分別表示的是第  $\ell$  個殘差單元的輸入和出，每個殘差單元一般包含多層結構。 $F$  是殘差函數，表示學習到殘差，而  $h(x_\ell) = x_\ell$  表示恆等映射， $f$  是 ReLU 激活函數。因此從淺層  $\ell$  到深層  $L$  的學習特徵為：

$$x_L = x_l + \sum_{i=l}^{L-1} F(x_i, W_i)$$

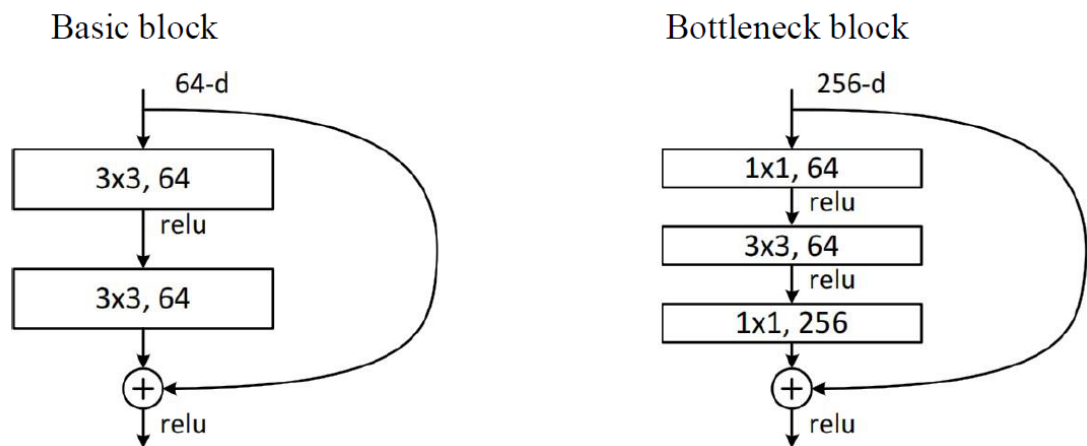
利用 Chain rule，可以求得反向過程的梯度：

$$\frac{\partial loss}{\partial x_l} = \frac{\partial loss}{\partial x_L} \cdot \frac{\partial x_L}{\partial x_l} = \frac{\partial loss}{\partial x_L} \cdot \left( 1 + \frac{\partial}{\partial x_l} \sum_{i=l}^{L-1} F(x_i, W_i) \right)$$

式子的第一個因子  $\frac{\partial \text{loss}}{\partial x_L}$  表示的 loss function 到達 L 的梯度，而殘差梯度則需要經過 weight layers，梯度不是直接傳遞過來，小括號另外一項 1 表明短路機制可以無損地傳播梯度，不會導致梯度消失。所以殘差學習會更容易訓練。

### Building residual basic block and bottleneck block:

而上述所講的其實就是左下圖的 basic block，ResNet 就是由 basic block 所組成的，而在疊更深的網路(ResNet50)時，ResNet 設計了 bottleneck block，降低 3x3 convolution 的寬度，大幅減少了所需的運算量。



### Implementation

接下來是實作的部分，model 我使用 torchvision 函式庫的 resnet18 和 resnet50，函式庫已經把 model 部分做好了，而需要修改的部分則是在最後一層 fully connected layer，也就是要在 linear 層輸出成 5 類 classes。

```
if model_name == 'resnet18':
    if pretrained_flag == True:
        model = models.resnet18(weights=models.ResNet18_Weights.DEFAULT)
        for param in model.parameters():
            param.requires_grad = False
    else:
        model = models.resnet18(weights=None)
        for param in model.parameters():
            param.requires_grad = True
    ftrs_num = model.fc.in_features
    model.fc = nn.Linear(ftrs_num, 5)
elif model_name == 'resnet50':
    if pretrained_flag == True:
        model = models.resnet50(weights=models.ResNet50_Weights.DEFAULT)
        for param in model.parameters():
            param.requires_grad = False
    else:
        model = models.resnet50(weights=None)
        for param in model.parameters():
            param.requires_grad = True
    ftrs_num = model.fc.in_features
    model.fc = nn.Linear(ftrs_num, 5)
```

在上圖可以看到，我先取得 input neuron 數，也就是上圖中的 `fters_num`，再把它和 output class 數 (5) 一起傳進 `nn.Linear` 裡面，最後指派給 `model.fc` 即完成，就可以建出這次作業 Diabetic Retinopathy Detection 的 5 classes 分類器。另外，在 `torchvision.models` 有 pretrained weight 供我們使用參數可以使用。

### Initializing pre-trained models

As of v0.13, TorchVision offers a new **Multi-weight support API** for loading different weights to the existing model builder methods:

```
from torchvision.models import resnet50, ResNet50_Weights

# Old weights with accuracy 76.130%
resnet50(weights=ResNet50_Weights.IMAGENET1K_V1)

# New weights with accuracy 80.858%
resnet50(weights=ResNet50_Weights.IMAGENET1K_V2)

# Best available weights (currently alias for IMAGENET1K_V2)
# Note that these weights may change across versions
resnet50(weights=ResNet50_Weights.DEFAULT)


# Strings are also supported
resnet50(weights="IMAGENET1K_V2")

# No weights - random initialization
resnet50(weights=None)
```

pretrained 的意思就是模型已經由先人經過 大量的 dataset 訓練(ex: ImageNet)，並儲存 weights 和 biases 供後人使用。這樣我們就不需要從 random initialize 的參數開始訓練。當然也可以設定 `weights = None` 將其關掉。另外，pretrained model 通常會搭配 feature extraction 使用。Feature extraction 是指在前幾個 epoch 單獨使用一個 optimizer 只更新最後一層的 fc layer，其他中間層則不計算 gradient 也不更新。其目的是為了更加運用已經先訓練好的 weight，固定前面的 CNN layers 有取出 input image 的特徵的用意。可以看到上圖實作 feature extraction 時先會把 `requires_grad` 設為 false，再指派 fc 層 (預設剛被指派的 fc 層的 `requires_grad` 會為 true)，這樣就能使 model 只有一個 fc 層的 `requires_grad` 是 true、其他層都是 false 的 model，後面的 epoch 就會恢復 `requires_grad` 是 true，進行 fine-tuning 的訓練。

我使用的 torchvision 版本是 0.14.1，新版的 torchvision 有 Multi-weight support，可以看到上圖 initializing pretrained models 可以選擇多種 weight 使用，而我是選擇 DEFAULT，也就是新版準確度比較高的 weight，另外，我使用的是 ResNet v1.5，因為它的準確度比較高，而 ResNet v1.5 和 ResNet v1 最主要也就是 downsampling 的大小，會影響它的些許的準確度和效能，可以看到下面是 ResNet v1.5 和 ResNet v1 的差異：

## ResNet v1.5 for PyTorch



DEEP LEARNING EXAMPLES

**Description**

With modified architecture and initialization this ResNet50 version gives ~0.5% better accuracy than original.

**Publisher**

NVIDIA

**Overview** | Version History | File Browser | Release Notes | Related Collections | More

The ResNet50 v1.5 model is a modified version of the [original ResNet50 v1 model](#).

The difference between v1 and v1.5 is that, in the bottleneck blocks which requires downsampling, v1 has stride = 2 in the first 1x1 convolution, whereas v1.5 has stride = 2 in the 3x3 convolution.

This difference makes ResNet50 v1.5 slightly more accurate (~0.5% top1) than v1, but comes with a small performance drawback (~5% imgs/sec).

The model is initialized as described in [Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification](#)

This model is trained with mixed precision using Tensor Cores on Volta, Turing, and the NVIDIA Ampere GPU architectures. Therefore, researchers can get results over 2x faster than training without Tensor Cores, while experiencing the benefits of mixed precision training. This model is tested against each NGC monthly container release to ensure consistent accuracy and performance over time.

We are currently working on adding [NHWC data layout](#) support for Mixed Precision training.

### OPTIMIZER

This model uses SGD with momentum optimizer with the following hyperparameters:

- Momentum (0.875)
- Learning rate (LR) = 0.256 for 256 batch size, for other batch sizes we linearly scale the learning rate.
- Learning rate schedule - we use cosine LR schedule
- For bigger batch sizes (512 and up) we use linear warmup of the learning rate during the first couple of epochs according to [Training ImageNet in 1 hour](#). Warmup length depends on the total training length.
- Weight decay (WD)= 3.0517578125e-05 (1/32768).
- We do not apply WD on Batch Norm trainable parameters (gamma/bias)
- Label smoothing = 0.1
- We train for:
  - 50 Epochs -> configuration that reaches 75.9% top1 accuracy
  - 90 Epochs -> 90 epochs is a standard for ImageNet networks
  - 250 Epochs -> best possible accuracy.
- For 250 epoch training we also use [MixUp regularization](#).

### DATA AUGMENTATION

This model uses the following data augmentation:

- For training:
  - Normalization
  - Random resized crop to 224x224
    - Scale from 8% to 100%
    - Aspect ratio from 3/4 to 4/3
  - Random horizontal flip
- For inference:
  - Normalization
  - Scale to 256x256
  - Center crop to 224x224

## B. The details of your Dataloader

這次作業要實作的部分是 RetinopathyLoader 裡的 `getitem()` 函式，接下來 `torch.utils.data.DataLoader` 才能依據 `getitem()` 拿取相應的資料。

如下圖可以看到 `getitem()` function 實作部分，根據 hint 的 step 步驟就能完成。也就是會先根據路徑和 index 取得 image 和 label，再對資料進行一些前處理並回傳。而這邊前處理的目的主要是為了將圖片轉成適合傳入模型的型態以及讓圖片資料多一點變化性和多樣性。

額外要注意的則是 train 跟 test 的 transform 要不一樣。隨機裁剪、翻轉是為 train data 增加多樣性、增加 training 難度，而 testing 時則應降低難度，取消

隨機裁剪、翻轉，才能使 test accuracy 提高，另外，因為每個 image 的解析度都不太相同，長寬也不太相同，所以需要做前處理，先將它等比例縮放到相同解析度，並且不能做擠壓或變形等動作，否則會讓準確度降低，可以看到我先用 Resize(256)，會伸縮成(h, w) = (256, 256 \* width/ height)，接著用 CenterCrop 把中間像素取出來，然後再做旋轉、翻轉等動作，就不會讓 image 有擠壓或變形的行為了。

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

    """
    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
    """

    # transform image (training & testing transform is different)
    # mini-batches of 3-channel RGB images of shape (3 x H x W), where H and W are expected to be at least 224
    train_transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(256),
        transforms.RandomCrop(224),
        transforms.RandomRotation(degrees=20),
        transforms.RandomHorizontalFlip(),
        transforms.RandomVerticalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ])
    test_transform = transforms.Compose([
        transforms.Resize(224),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ])

    # step 1
    path = self.root + self.img_name[index] + '.jpeg'
    # step 2
    label = self.label[index]

    # step 3
    image = Image.open(path).convert('RGB')
    if self.mode == 'train':
        img = train_transform(image)
    else:
        img = test_transform(image)

    # step 4
    return img, label
```

- Resize: Resize(size)表示 image 會被縮放，if width > height, (h, w) = (size, size \* width/ height)，反之，if height > width, (h, w) = (size \* height /



width, size)

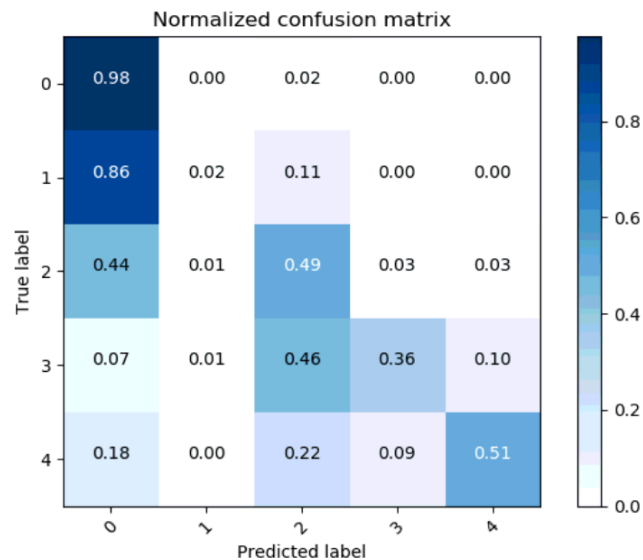
- RandomCrop: 依據給定的 size 隨機剪裁。
- CenterCrop: 以圖片中心為準，依據給定的 size 剪裁，假如裁剪的比原圖還大時，多出來的部分會用黑色填補。
- RandomRotation(degrees=d): 會隨機在 (-d, d) 的角度範圍作旋轉，旋轉造成周圍缺的部分會用黑色填補。
- RandomHorizontalFlip: RandomHorizontalFlip(p=0.5)表示圖片有 0.5 的機率進行水平翻轉。
- RandomVerticalFlip: RandomVerticalFlip(p=0.5) 表示圖片有 0.5 的機率進行垂直翻轉。
- ToTensor: 將 PIL image 或 ndarray 轉成 tensor 型態，並將值轉成[0, 1]。而影像大小(H x W x C)也會被轉成(C x H x W)。
- Normalize: 正規化，需給定影像平均值(mean)和標準差(std)。根據 ImageNet 資料集訓練，訓練集所計算出來的平均值(mean)和標準差(std)分別為 mean = [0.485, 0.456, 0.406] 和 std = [0.229, 0.224, 0.225]。

### C. Describing your evaluation through the confusion matrix

Confusion matrix 是經常用來描述一個 classification model 的 performance 指標，而 confusion matrix 的畫法我則是直接使用 sklearn.metrics 裡的 ConfusionMatrixDisplay 函式來畫。只需要傳 true\_label 和 prediction 結果，並指定 normalize = true 即可。

Parameters:	<p><b>y_true : array-like of shape (n_samples,)</b> True labels.</p> <p><b>y_pred : array-like of shape (n_samples,)</b> The predicted labels given by the method <code>predict</code> of an classifier.</p> <p><b>labels : array-like of shape (n_classes,), default=None</b> List of labels to index the confusion matrix. This may be used to reorder or select a subset of labels. If <code>None</code> is given, those that appear at least once in <code>y_true</code> or <code>y_pred</code> are used in sorted order.</p> <p><b>sample_weight : array-like of shape (n_samples,), default=None</b> Sample weights.</p> <p><b>normalize : {'true', 'pred', 'all'}, default=None</b> Either to normalize the counts display in the matrix:</p> <ul style="list-style-type: none"><li>• if <code>'true'</code>, the confusion matrix is normalized over the true conditions (e.g. rows);</li><li>• if <code>'pred'</code>, the confusion matrix is normalized over the predicted conditions (e.g. columns);</li><li>• if <code>'all'</code>, the confusion matrix is normalized by the total number of samples;</li><li>• if <code>None</code> (default), the confusion matrix will not be normalized.</li></ul>
-------------	---

我以下面示範的 confusion matrix 來說明，縱軸是 true label、橫軸是 predicted label。舉例來說，下圖中左上方的 0.98 那格就是預測是 0、實際也是 0 的比例是 0.98；0.86 那格就是預測是 0、實際是 1 的比例為 0.86。而我指定 `normalize = true` 來代表使用真實的數量(true label)做 normalize，也就是每個 row 總和為 1。



### 3. Data Preprocessing (20%)

#### A. How you preprocessed your data?

在前面實作 dataloader 有提到，會對資料進行一些前處理。而前處理的目的主要是為了將圖片轉成適合傳入模型的型態且增加圖片資料的多樣性。

我先把 PIL 開啟的 RGB image 做 `resize`，將它等比例縮放到相同解析度，伸縮成  $(h, w) = (256, 256 * \text{width} / \text{height})$ ，接著用 `CenterCrop` 把中間像素取出來，接著要讓圖片資料有多一點變化性和多樣性，我用 `RandomCrop`、`RandomRotation`、`RandomHorizontalFlip` 和 `RandomVerticalFlip`，讓它有一定機率做隨機裁切、旋轉、水平或垂直翻轉，並且把它轉成 `tensor` 且做 `normalize`，讓 `torch.utils.data.DataLoader` 可以取的相對應資料，這樣就能讓 data 經過前處理取到多樣性的 `preprocessed data`。

在 testing 時候，這些產生圖片多樣性 `RandomCrop`、`RandomRotation`、`RandomHorizontalFlip` 和 `RandomVerticalFlip` 應該要取消，才能使 `test accuracy` 提高，也 testing transform 只做 `resize` 把圖片等比例縮放，接著做 `CenterCrop` 把中間像素取出來，最後把它轉成 `tensor` 且做 `normalize` 就好。

#### B. What makes your method special?

除了前面 dataloader 做的前處理，我在 training 的時候有發現有幾張圖片是殘缺的，可能是圖片本身是殘缺的或者是下載解壓縮時讓圖片受損，而這其實



是用圖片訓練時，做前處理蠻常遇到且重要的一部份，這次作業的 dataset 是 jpeg 檔，jpeg 檔的特徵是：

- Start of Image(SOI)，圖像開始標記：2 字節，固定值為\xff\xd8
- End of Image(EOI)，圖像結束標記：2 字節，固定值為\xff\xd9

也就是正常圖片的字節流是以\xff\xd8 開始，\xff\xd9 結束，假如圖片的字節流有缺失，也就是損壞，會報出 `OSError: image file is truncated (xxx bytes not processed)`，這時候就可以透過下面程式檢查圖片是否有損壞：

```
pimg = "data/"+img_name[index]+ ".jpeg"
with open(pimg, 'rb') as f:
    f = f.read()
    if f[-1] == 217 and f[-2] == 255:
        # EOI = \xff\xd9
        pass
    else:
        # borken image
        print(img_name[index])
```

當偵測到圖片有損壞的時候，最直接的方式就是丟棄那筆 data，但有時候數據缺損的很多，直接拋棄可能就會造成成本太高，另外的一種方式就是補齊再用，也就是在後面補上正確的 EOI (\xff\xd9)。

```
pimg = "data/"+img_name[index]+ ".jpeg"
with open(pimg, 'rb') as f:
    f = f.read()
    if f[-1] == 217 and f[-2] == 255:
        # EOI = \xff\xd9
        pass
    else:
        # borken image
        print(img_name[index])
        f = f+B'\xff'+B'\xd9'
im = Image.open(BytesIO(f))
```

## 4. Experimental results

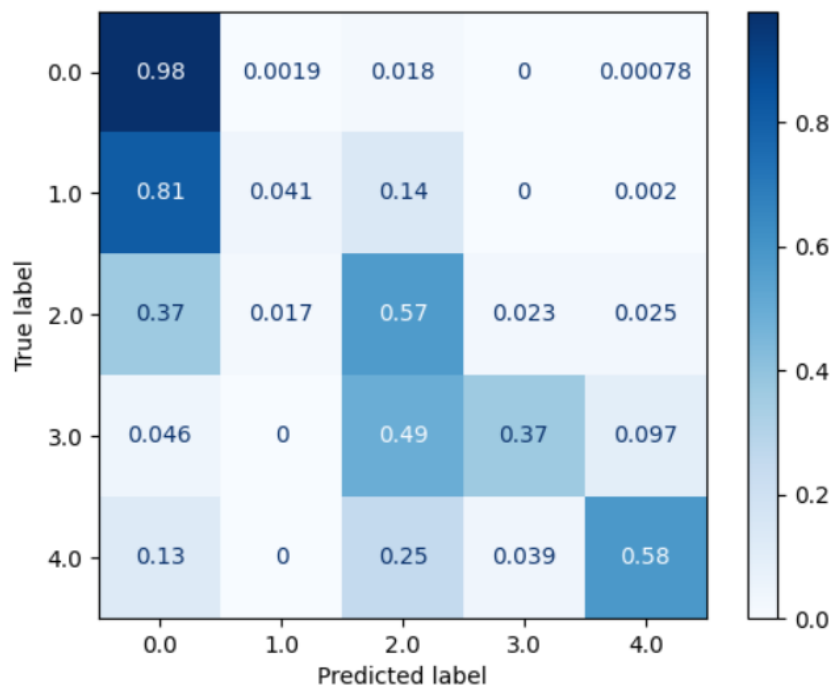
- Batch size= 64 for resnet18 / Batch size= 16 for resnet50
- Learning rate =  $1e-3$
- Epochs = 25 : 5(feature extracting) + 20(finetuning)
- Optimizer: SGD
- Momentum = 0.9
- Weight Decay =  $5e-4$
- Loss function: Cross Entropy Loss

其中 batchsize 和 epoch 有所調整，因為 batchsize 變大時，能讓確定的梯度下降方向較準，引起的訓練震盪較小，因此訓練效果比較好，另外 epoch 數也有所增加，因為 batchsize 變大會使參數的修正較為緩慢。

### A. The highest testing accuracy

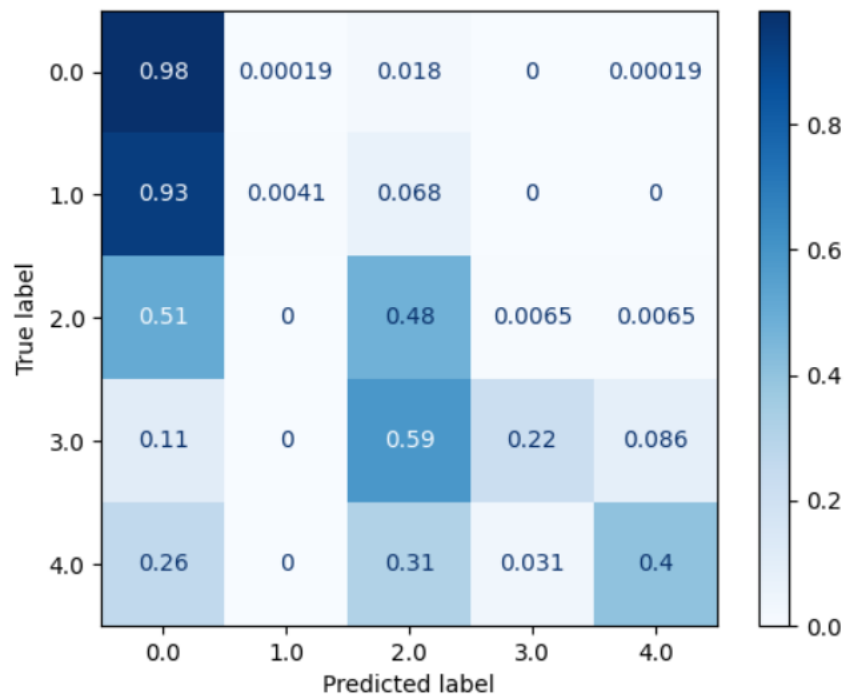
**Resnet50** with pretraining: highest testing accuracy (**82.8612**) at epoch 23

```
epoch 23 :  
training accuracy: tensor(84.8982, device='cuda:0')  
testing accuracy: tensor(82.8612, device='cuda:0')  
epoch 23 save weights
```



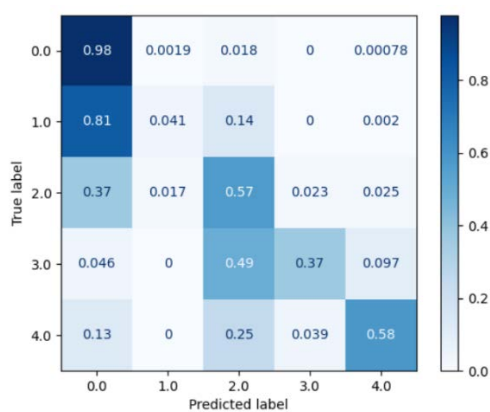
**Resnet18** with pretraining: highest testing accuracy (**80.6690**) at epoch 23

```
epoch 23 :  
trainig accuracy: tensor(82.4708, device='cuda:0')  
testing accuracy: tensor(80.6690, device='cuda:0')  
epoch 23 save weights
```

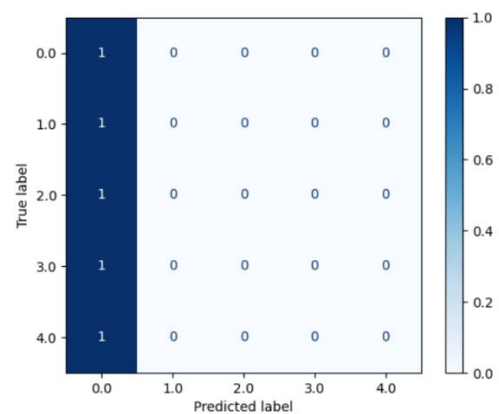


## B. Comparison figures

### Resnet50

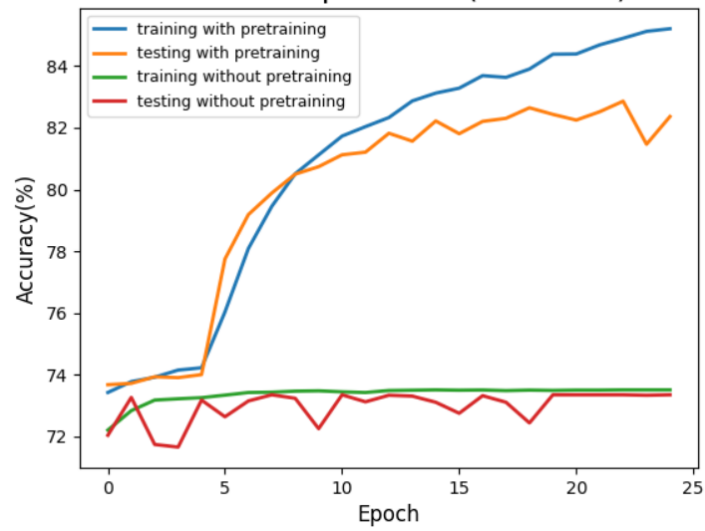


with pretraining

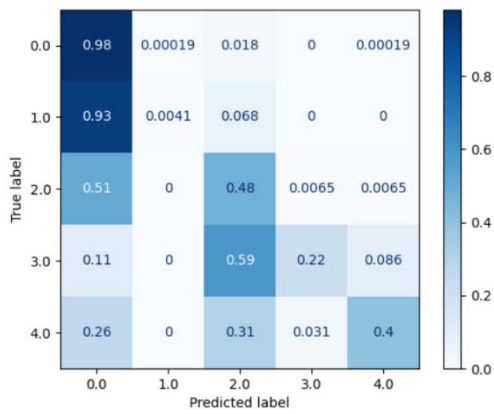


without pretraining

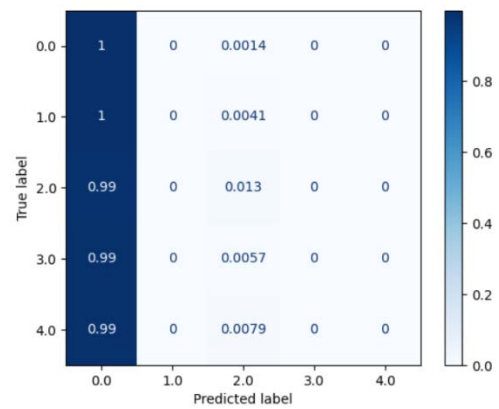
Result comparaisn (resnet50)



Resnet18

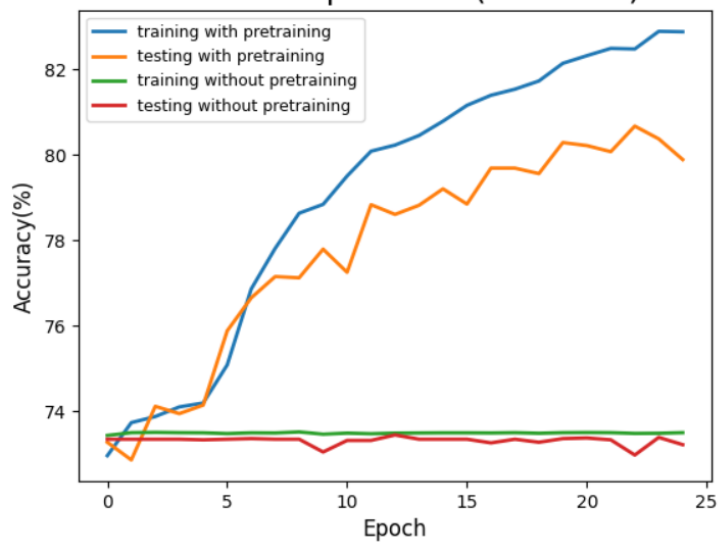


with pretraining



without pretraining

Result comparaisn (resnet18)



從上面比較圖中可以看到，有用 pretrain 明顯比沒用 pretrain 表現得更好。另外，用 pretrain 因為前五個 epoch 在做 feature extraction，所以 accuracy 較低，過前五個 epoch 後做 finetuning 時，accuracy 會有一段明顯的提升。

## 5. Discussion

### A. Augmentation (資料增強)

這次作業因為有些類別的訓練資料比較少，所以可以透過對經過旋轉、調整大小比例尺寸，或者改變亮度色溫翻等處理增加圖片的數量，因為人眼仍能辨識出來是相同的片但對機器說完全不新圖像，因此 Data augmentation 就是將 dataset 中既有的圖片予以修改變形，創造出更多的圖片來讓機器學習，彌補資料量不足困擾。且是當的 augmentation 可以對影像產出的結果訓練出更好的模型。

1. 資料的正規化(normalization)：可針對 Sample-wise（每次取樣的 sample batch）或 Feature-wise（整體的 dataset）去做 normalization
2. 資料白化（Whitening）處理：提供 ZCA Whitening 處理。（Whitening 是一種將資料去冗餘的技術）
3. 影像處理 翻轉、旋轉、切裁、放大縮小、偏移 ...等。

以下為各種 以下為各種 augmentation 方法：

#### (1) 裁剪(Crop)

- a. 隨機裁剪： `transforms.RandomCrop`
- b. 中心裁剪： `transforms.CenterCrop`
- c. 隨機長寬比裁剪
- d. 上下左右中心裁剪： `transforms.FiveCrop`
- e. 上下左右中心裁剪後翻轉： `transforms.TenCrop`

#### (2) 翻轉和旋轉(Flip and Rotation)

- a. 依概率 p 水平翻轉： `transforms.RandomHorizontalFlip`
- b. 依概率 p 垂直翻轉： `transforms.RandomVerticalFlip`
- c. 隨機旋轉： `transforms.RandomRotation`

#### (3) 圖像變換

- a. resize： `transforms.Resize`
- b. 標準化： `transforms.Normalize`
- c. 轉為 tensor： `transforms.ToTensor`
- d. 填充： `transforms.Pad`
- e. 修改亮度、對比和飽： `transforms.ColorJitter`
- f. 轉灰度圖： `transforms.Grayscale`
- g. 線性變換： `transforms.LinearTransformation()`
- h. 仿射變換： `transforms.RandomAffine`

- i. 依概率  $p$  轉為灰度圖： `transforms.RandomGrayscale`
  - j. 將數據轉換為 PILImage： `transforms.ToPILImage`
  - k. `transforms.Lambda`：Apply a user-defined lambda as a transform
- (4) 對 `transforms` 操作，使數據增強更靈活
- a. `transforms.RandomChoice(transforms)`：從給定的一系列 `transforms` 中選一個操作。
  - b. `transforms.RandomApply(transforms, p=0.5)`：給一個 `transform` 加上概率，以一定的執行該操作。
  - c. `transforms.RandomOrder`：將 `transforms` 中的操作順序隨機打亂中。

## B. Batch size

除了梯度本身，`batchsize` 以及 `learning rate` 直接決定了模型的權重更新，從優化本身來看是影響性能收斂最重要的參數。

1. 大的 `batchsize` 內存利用率提高，大矩陣乘法的並行化效率也有所提升。隨著 `batch size` 增加，跑完一次 `epoch` 的迭代次數減少，對於相同數據量的處理速度較快，並且確定的梯度下降方向較準，引起的訓練震盪較小，但達到相同準確度所需的 `epoch` 會有所增加。

目前已經有多篇公開論文用大的 `batchsize` 在 1 小時內訓練完 ImageNet 數據集。梯度計算更加穩定，模型訓練曲線會更加平滑。並且在微調的時候，大 `batchsize` 可能取得好的結果。但盲目增加 `batchsize` 可能會使模型的泛化能力降低，所以 `batchsize` 的決定是相當重要的。

2. `learning rate` 和 `batchsize` 的關係，通常當我們增加 `batchsize` 為原來的  $N$  倍時，要保證經過同樣的本後更新權重替代，按照線性縮放規則學習率應該增加為原來的  $N$  倍。但是如果保證權重的方差不變，則學習率應該增加為原來的  $\sqrt{N}$  倍，目前這兩種策略都被研究過，使用前者的明顯居多。

## C. imbalanced data

Label 0	20656	73.51%
Label 1	1955	6.96%
Label 2	4210	14.98%
Label 3	698	2.48%
Label 4	581	2.07%

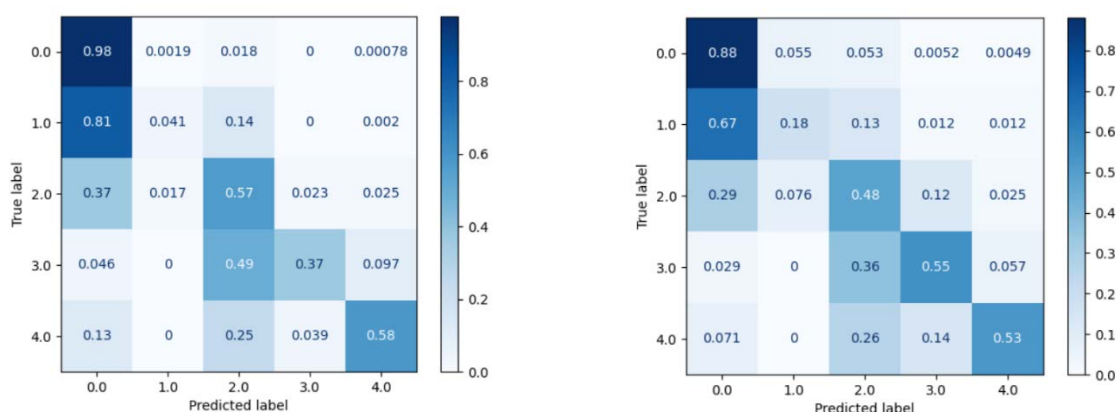


上面是 training dataset 中各個 class 種類及其對應數量，可以觀察到這其實是一組 imbalanced data，大部分都落在 Label 0，為了讓 model 對 Label 0 之外更加敏感，我們可以在 CrossEntropyLoss() 增加 classes weight 參數。

**class weights = [1.0, 10.5652173913043, 4.9061757719714, 29.5916905444126, 35.5507745266781]**

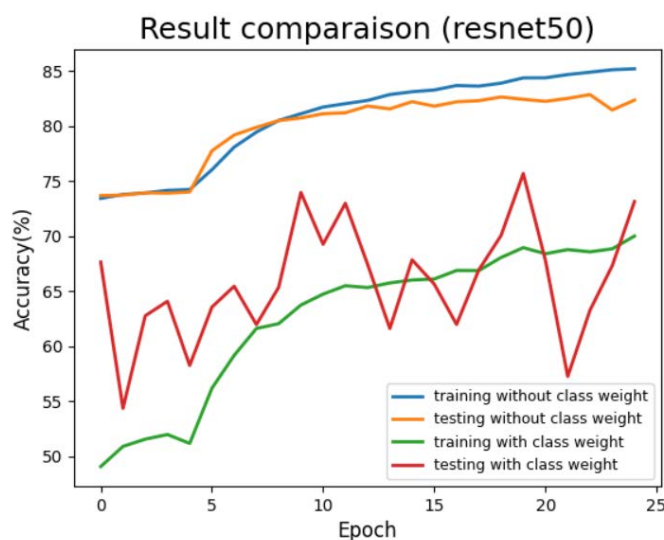
上面的參數值意義為當 label 為 1 時，更新的 loss 量會是 label 0 的 10.56 倍，以此類推。這代表 model 會被 update 成更傾向預測 0 以外的 label。

### Comparison figure of Resnet50 with pretraining



without class weight

with class weight



### Resnet50 with pretraining & class weight: highest testing accuracy (75.6868)

從上比較圖可以看出，我們犧牲了 label 0 的 accuracy，換取了其他 label 更高的 accuracy。但是 label 0 的數量還是太龐大了，因此 total accuracy 有所下降。但 confusion matrix 較為正確，可以看到 label1&label3 的準確度都有提升。若能找到更適當的 class weight 或 hyper parameters，就有可能使 total accuracy 提升。