1. Introduction (20 %)

   In lab3 we want to detect “Diabetic Retinopathy” by the following design.

   (1) Dataset : a large set of high-resolution, i.e., 512x512, retina images with five levels diagnosis, i.e., 0-No DR, 1-Mild, 2-Moderate, 3-Severe, and 4-Proliferative DR. As reality, the dataset is imbalance with large No DR samples while the number of higher level cytopathy(病變) is small. The exactly number of each class is {0: 20655, 1: 1955, 2: 4210, 3: 698, 4: 581} in training dataset and {0: 5153, 1: 488, 2: 1082, 3: 175, 4: 127} in testing dataset. Here, we will build a customized Dataloader to get these data.

   (2) Model : ResNet18 (with/without pre-trained) and ResNet50 (with/without pre-trained)

   (3) Evaluation : Accuracy and Confusion Matrix

2. Experiment setups (30 %)

   A. The details of your model (ResNet)

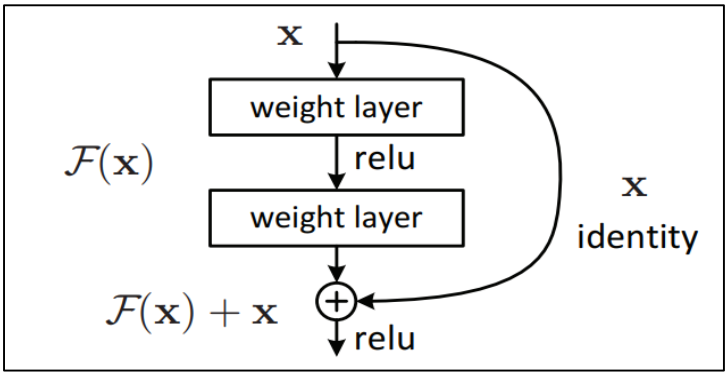



Fig. 1 Single residual block

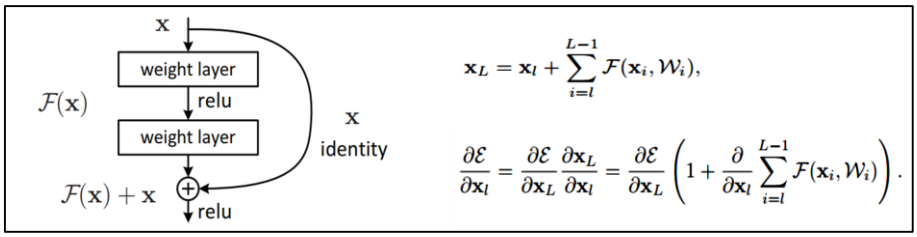
The concept of ResNet (Residual Network) is add a skip / shortcut connection from input to output after few weight layers (Fig. 1). The purpose of this design is to avoid gradient vanishing or exploding problems by changing a series multiplication of gradients to addition operations in one residual block. Fig. 2 shows the traditional back propagation which usually get gradient vanishing or exploding results. On the contrary, back propagation in a residual block (Fig. 3) will solve this problem.

$$x \rightarrow \underset{y_1}{w_1} \rightarrow \underset{y_2}{w_2} \rightarrow \underset{y_3}{w_3} \rightarrow \underset{y_4}{w_4} \rightarrow Loss$$

$$\frac{\partial Loss}{\partial w_1} = \frac{\partial Loss}{\partial y_4}\frac{\partial y_4}{\partial z_4}\frac{\partial z_4}{\partial y_3}\frac{\partial y_3}{\partial z_3}\frac{\partial z_3}{\partial y_2}\frac{\partial y_2}{\partial z_2}\frac{\partial z_2}{\partial y_1}\frac{\partial y_1}{\partial z_1}\frac{\partial z_1}{\partial w_1}$$

$$= \frac{\partial Loss}{\partial y_4}\sigma'(z_4)w_4\sigma'(z_3)w_3\sigma'(z_2)w_2\sigma'(z_1)x_1$$

Fig. 2 Traditional back propagation




$$\mathbf{x}_L = \mathbf{x}_l + \sum_{i=l}^{L-1} \mathcal{F}(\mathbf{x}_i, \mathcal{W}_i),$$

$$\frac{\partial \mathcal{E}}{\partial \mathbf{x}_l} = \frac{\partial \mathcal{E}}{\partial \mathbf{x}_L}\frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_l} = \frac{\partial \mathcal{E}}{\partial \mathbf{x}_L}\left(1 + \frac{\partial}{\partial \mathbf{x}_l}\sum_{i=l}^{L-1}\mathcal{F}(\mathbf{x}_i, \mathcal{W}_i)\right).$$

Fig. 3 Back propagation in a residual block

- ResNet 18

ResNet 18 uses the “Basic block” (shown as Fig. 4) as its components. The overall architecture is shown in Fig. 6 with red box.

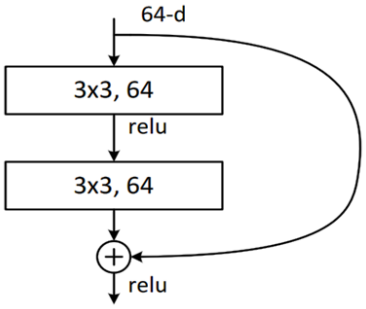



Fig. 4 Basic block

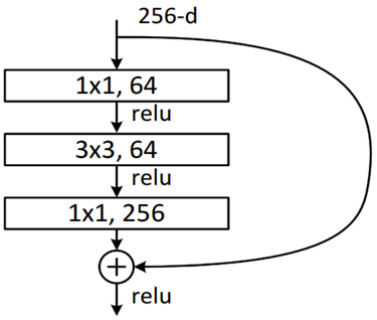



Fig. 5 Bottleneck block

- ResNet 50

ResNet 50 uses the “Bottleneck block” (shown as Fig. 5) as its components. Even if “Bottleneck block” has 3 layers more than “Basic block”, i.e., 2 layers, their size of parameters are almost the same. The overall architecture is shown in Fig. 6 with blue box.

| 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 | $\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$ | $\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$ |
| 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 | | $1.8\times10^9$ | $3.6\times10^9$ | $3.8\times10^9$ | $7.6\times10^9$ | $11.3\times10^9$ |

Fig. 6 The overall architecture of ResNet18 (red box) and ResNet50 (blue box)

- Transfer learning

Because we can use ResNet18 and ResNet50 with pre-trained weights, we can design the number of layers we want to freeze. Here, in the first 5 epochs, we just fine-tune the last fc-layer; in the following 10

epochs, we fine-tune whole networks. The trainable parameters of ResNet18 between fine-tune the last fc-layer and fine-tune whole networks are shown in Fig. 7.

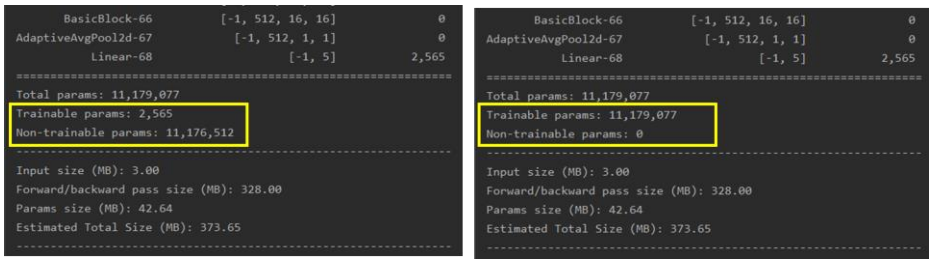



Fig. 7 The trainable parameters of ResNet18 between fine-tune the last fc-layer and fine-tune whole networks.

B. The details of your Dataloader

Step1 : Implement dataset class. In this class, we implemented three functions initial (__init__), get length (__len__), and get a data pair (__getitem__) for the following Dataloader using.

```
class RetinopathyDataset(data.Dataset):
    def __init__(self, root, mode, transform):
        """
        Args:
            root (string): Root path of the dataset.
            mode : Indicate procedure status(training or testing)
            transform : any transform, e.g., random flipping, rotation,
cropping, and normalization.

            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)
        self.mode = mode
        self.transform = transform
        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):
        """retuen a pair of (image, label) which image is changed from
self.tranform """

        img_path = self.root + self.img_name[index] + '.jpeg'
        img = Image.open(img_path).convert('RGB')
        label = int(self.label[index])

        if self.transform is not None:
            img = self.transform(img)

        return img, label
```

Step2 : Create train and validate transforms. Here we used RandomHorizontalFlip, RandomVerticalFlip, ToTensor ([C, H, W] with value 0~1), and Normalize.

```
# create train/val transforms
train_transform = trns.Compose([
                    trns.RandomHorizontalFlip(p=0.4),
                    trns.RandomVerticalFlip(),
                    trns.ToTensor(),
                    trns.Normalize(mean=[0.485, 0.456, 0.406],
                                   std=[0.229, 0.224, 0.225]),
                ])
val_transform = trns.Compose([
                  trns.ToTensor(),
                  trns.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225]),
              ])
```

Step 3 : Create train and validate dataset.

```
# create train/val datasets
trainset = RetinopathyDataset(root='data/',
                              mode='train',
                              transform=train_transform)
valset = RetinopathyDataset(root='data/',
```

```
                        mode='test',
                        transform=val_transform)
```

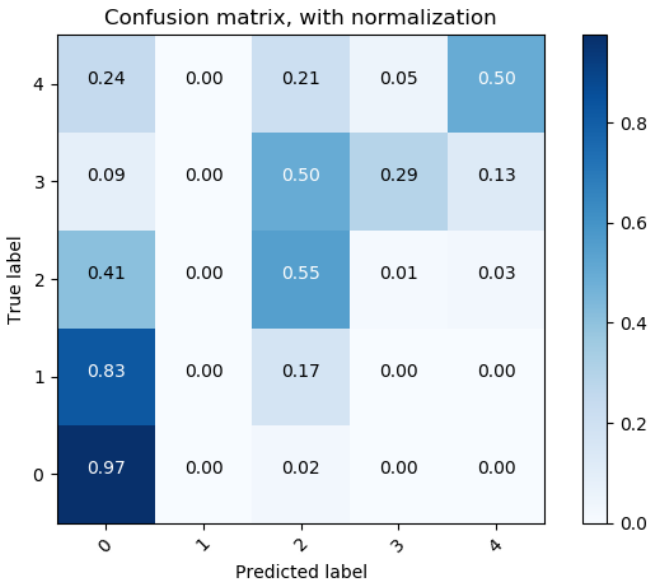
Step 3 : Create train and validate data loaders.

```
# create train/val loaders
train_loader = DataLoader(dataset=trainset,
                          batch_size=BatchSize,
                          shuffle=True,
                          num_workers=0)
val_loader = DataLoader(dataset=valset,
                        batch_size=BatchSize,
                        shuffle=False,
                        num_workers=0)
```

C. Describing your evaluation through the confusion matrix

In the beginning, we still used accuracy as our evaluation function. However, it may cause to false results due to the imbalanced dataset while we observed the confusion matrix. Class with the largest data dominates the loss value, therefore the correct numbers of this class is the most. But, in reality, false alarm of the class with smaller samples may cause the worst results, e.g., 對重度症狀的誤判. In addition, our result shows the classifier cannot distinguish 1-Mild form 0-No DR.

```
# Compute confusion matrix
cnf_matrix = confusion_matrix(true_result, pred_result, labels=[0, 1, 2, 3, 4])
```



3. Experimental results (30 %)
   A. The highest testing accuracy

      BatchSize = 32

      lr = 1e-3

      weight_decay = 5e-4

      momentum = 0.9

      epochs_1 = 5 (fine-tune fc-layer)

      epochs_2 = 10 (fine-tune whole network)

      Augmentation:

         RandomHorizontalFlip(p=0.4)

         RandomVerticalFlip()

         ToTensor()

         Normalize(mean=[0.485, 0.456, 0.406],

                   std=[0.229, 0.224, 0.225])

      ■ Screenshot

      <u>ResNet18</u>

```
Epoch 0, Training Loss: 731.456492, Train Accuracy: 73.646749, Test Accuracy: 73.537367

Epoch 1, Training Loss: 686.746492, Train Accuracy: 74.102281, Test Accuracy: 74.021352

Epoch 2, Training Loss: 678.698461, Train Accuracy: 74.123634, Test Accuracy: 73.978648

Epoch 3, Training Loss: 671.038956, Train Accuracy: 74.408342, Test Accuracy: 74.362989

Epoch 4, Training Loss: 669.284127, Train Accuracy: 75.248229, Test Accuracy: 75.202847
```

```
Epoch 0, Training Loss: 612.727815, Train Accuracy: 78.828428, Test Accuracy: 78.177936

Epoch 1, Training Loss: 543.003266, Train Accuracy: 81.049148, Test Accuracy: 79.672598

Epoch 2, Training Loss: 512.298666, Train Accuracy: 80.938823, Test Accuracy: 80.142349

Epoch 3, Training Loss: 491.663724, Train Accuracy: 82.757394, Test Accuracy: 80.740214

Epoch 4, Training Loss: 476.400256, Train Accuracy: 80.113171, Test Accuracy: 77.893238

Epoch 5, Training Loss: 461.781788, Train Accuracy: 83.205808, Test Accuracy: 80.797153

Epoch 6, Training Loss: 445.600244, Train Accuracy: 84.301932, Test Accuracy: 81.622776

Epoch 7, Training Loss: 427.808387, Train Accuracy: 83.643546, Test Accuracy: 81.138790

Epoch 8, Training Loss: 414.658515, Train Accuracy: 84.099078, Test Accuracy: 81.153025

Epoch 9, Training Loss: 400.031283, Train Accuracy: 85.643617, Test Accuracy: 80.512456
Normalized confusion matrix
[[9.31e-01 6.02e-03 6.13e-02 1.94e-04 9.70e-04]
 [7.36e-01 3.07e-02 2.34e-01 0.00e+00 0.00e+00]
 [2.83e-01 1.29e-02 6.77e-01 1.02e-02 1.76e-02]
 [4.00e-02 5.71e-03 5.89e-01 2.40e-01 1.26e-01]
 [1.02e-01 0.00e+00 3.31e-01 3.94e-02 5.28e-01]]
```

      ■ Anything you want to present

If we set class weight in parameters of CrossEntropy like

$$\text{class\_weights} = \left\{\frac{max}{20655}, \frac{max}{1955}, \frac{max}{4210}, \frac{max}{698}, \frac{max}{581}\right\}, \text{where } \max = 20655$$

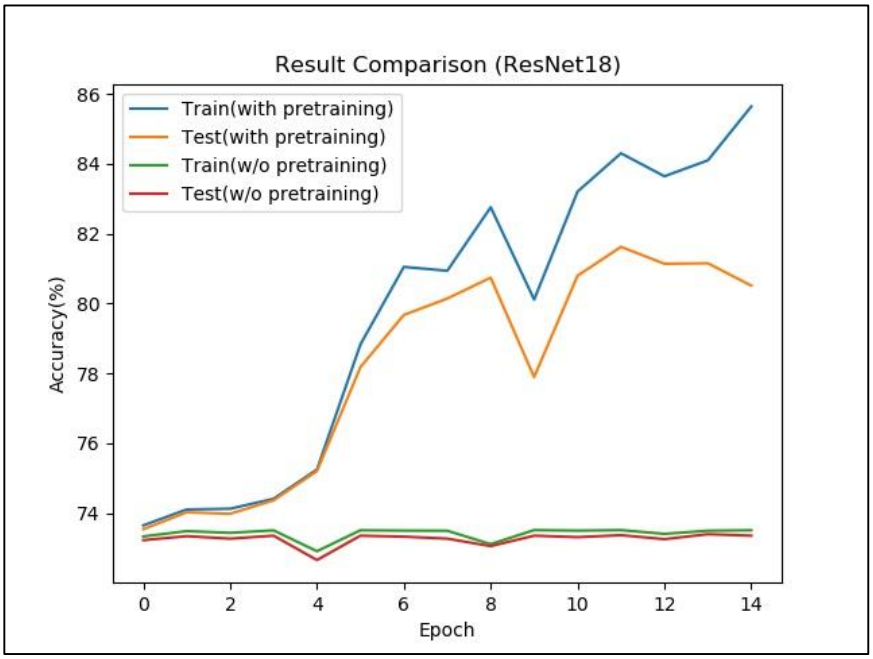
criterion = nn.CrossEntropyLoss(weight=class_weights).to(device)

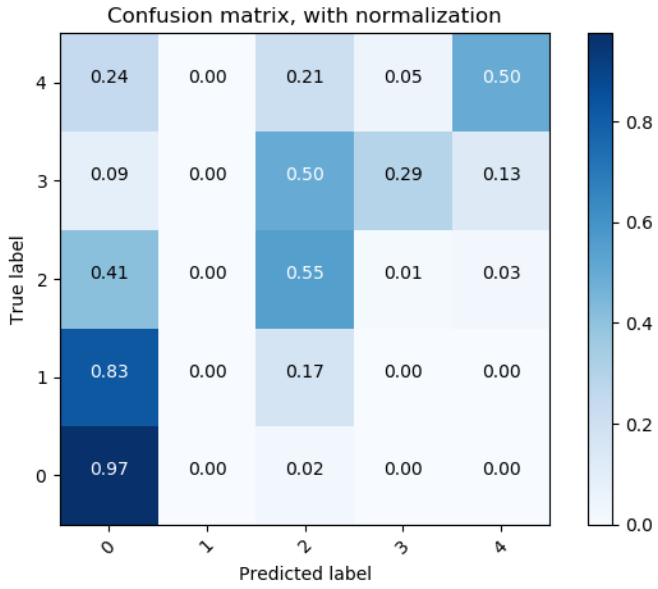
The accuracy will smaller than 73 %. In fact, we got 45 % in experience.

B. Comparison figures

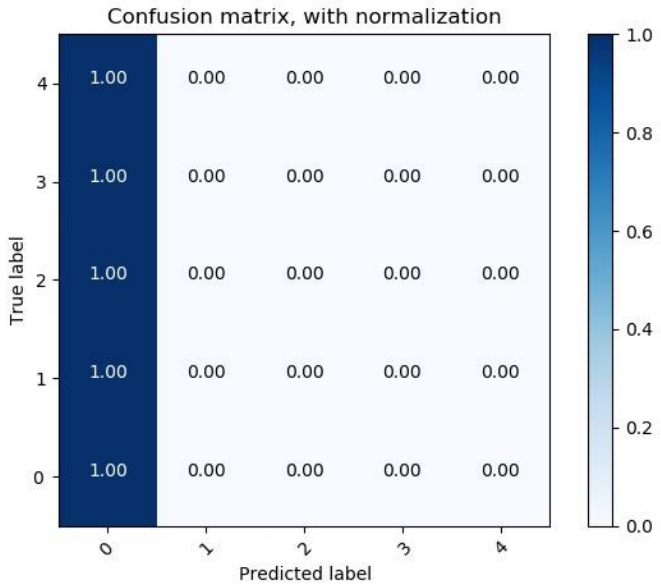
- Plotting the comparison figures (ResNet18/50, with/without pretraining)

(1) ResNet 18



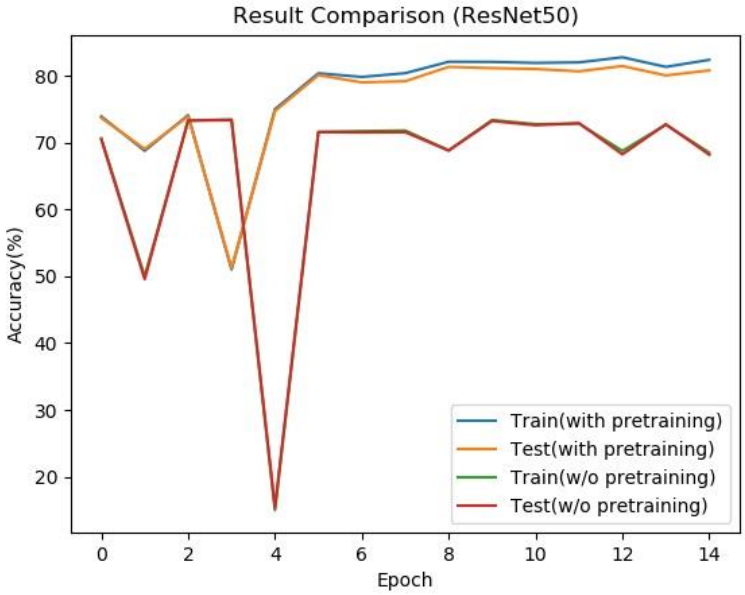



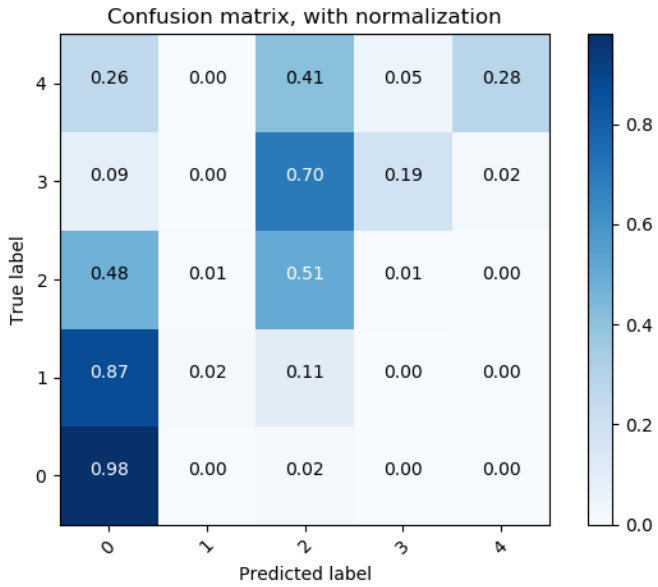

ResNet18 with pre-trained weights



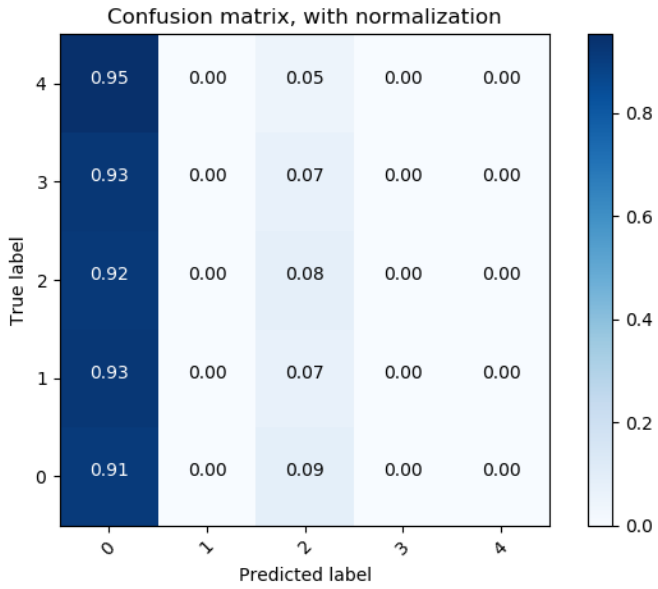

ResNet18 w/o pre-trained weights

(2) ResNet 50






ResNet50 with pre-trained weights




ResNet50 w/o pre-trained weights

(3) Comparison Table

| | with pre-trained | w/o pre-trained |
|---|---|---|
| ResNet 18 | 81.62 % | 73.4 % |
| ResNet 50 | 81.49 % | 73.38 % |

4. Discussion (20 %)

A. Anything you want to share

(1) 在這個作業中，一開始先分析每個類別的資料量，發現不平衡，

依過往經驗，須對資料做處理，不論是減採樣、過採樣、或是對 CrossEntropy 設定 Class weight，但實際執行後發現，準確率會降低，所以放棄這個做法（但實際應用上我覺得應該要使用）

(2) ResNet 的輸入是 224x224，但實際影像是 512x512，比較直接 resize 成 224x224 跟一開始不 resize 直到 AdaptiveAvgPool2d(1)才進行剪裁，發現後者的準確率較高

(3) 在 data augmentation 的部分，因為觀察到原圖的影像明暗不一，有測試對亮度做擴增，實作上包括 pytorch 內建的轉換函式，以及 opencv 的 CLAHE(限制對比度的直方圖均衡)，但實測後效果皆不佳，準確率降至 15 %及 45 %

```
ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5, hue=0)
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
```

(4) 真正讓準確率開始有大幅提升的關鍵是，一開始先只 train 最後一層 fc-layer，之後才全部 fine-tune