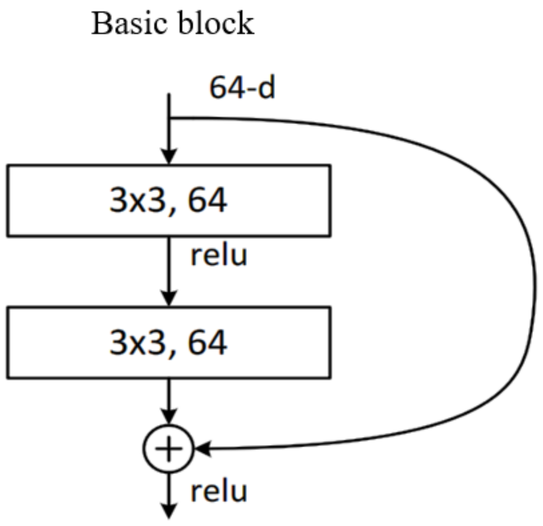
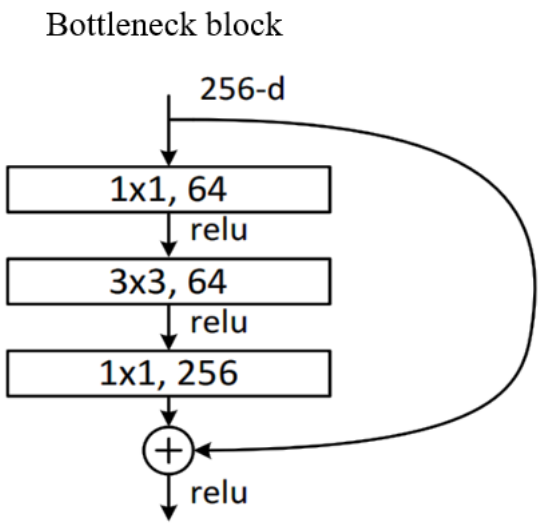
1. Introduction

In this lab, I will analysis a sequence of high-resolution retina images, and classify each image based on the level of ***diabetic retinopathy*** via the ***ResNet*** architectures. **Diabetic retinopathy** is the leading cause of blindness in the working-age population of the developed world. Thus, it is crucial that we come up with an accurate classification model, to help doctors make diagnosis. Talking about prominent image classification model, **ResNet (Residual Network)** is a well-known one that solves the problem of *vanishing/exploding gradients* by *skip-connections*, which adds the inputs to the outputs after few weight layers. By such a mechanism, ResNet does not directly learn the plain transformed values, rather it learns the **“residual”** of the transformed values. This, is the so-called *residual-learning*.

In this implementation, we are going to try 2 different structures: ***ResNet18*** and ***ResNet50***, and try the models ***with pretrained weights*** and ***without pretrained weigths*** for both models.
In **ResNet18**, only basic residual blocks are used:



while in **ResNet50**, ***bottleneck blocks*** are used:



Bottleneck block uses *1x1 convolution* to reduce the channels of the input before performing the expensive 3x3 convolution, then using another 1x1 convolution to project it back into the original shape. Hence, the advantage of the bottleneck is that by doing so, it *saves a lot of memory in GPU*, and thus we can append **more layers** compared to the ResNet with ordinary residual blocks.

As for the dataset, it contains **28099** training data and **7025** testing data. Each image has the resolution of 512×512 and has been preprocessed. Each image belongs to one of the **5** levels of diabetic retinopathy: **0 = No DR**, **1 = mild DR**, **2 = moderate DR**, **3 = severe DR**, **4 = proliferative DR**.

Finally, I will calculate and draw the confusion matrixes for these 4 combinations: $(ResNet18,\ w/\ pretraining)$, $(ResNet18,\ w/o\ pretraining)$, $(ResNet50,\ w/\ pretraining)$, $(ResNet50,\ w/o\ pretraining)$, all examined on the testing dataset. The models having the weights with the best testing accuracy for these 4 combinations will be used to draw the confusion matrixes.
Confusion matrix is a table that is often used to describe the performance of a classification model on a set of test data for which the true values are known (i.e. *supervised*

learning). I will use the `confusion_matrix()` for the package `sklearn.metrics` to compute the *normalized* confusion matrixes. Also, `ConfusionMatrixDisplay()` from the same package will be used to visualize the confusion matrixes.

2. Experiment Setups

A. The Details of my ResNet Models

Here's my code scripts for **ResNet18** and **ResNet50**. Both are practically the same logics, so I will describe them altogether.

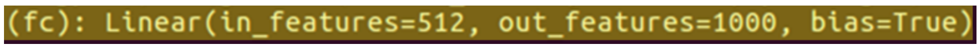
```
class ResNet18(nn.Module):
    def __init__(self, num_classes = 5, pretrained = True):
        super().__init__()
        self.model = models.resnet18(pretrained = pretrained)
        if pretrained:
            for param in self.model.parameters():
                param.require_grads = False
        num_of_fc_input_neurons = self.model.fc.in_features # 512
        self.model.fc = nn.Linear(num_of_fc_input_neurons, num_classes)
    def forward(self, X):
        outputs = self.model(X)
        return outputs
```

```
class ResNet50(nn.Module):
    def __init__(self, num_classes = 5, pretrained = True):
        super().__init__()
        self.model = models.resnet50(pretrained = pretrained)
        if pretrained:
            for param in self.model.parameters():
                param.require_grads = False
        num_of_fc_input_neurons = self.model.fc.in_features # 512
        self.model.fc = nn.Linear(num_of_fc_input_neurons, num_classes)
    def forward(self, X):
        outputs = self.model(X)
        return outputs
```

Firstly, we load the models from the **torchvision** library. Both ResNet18 and ResNet50 can be directly load in from `models` like so.

```
from torchvision import models
class ResNet18(nn.Module): #(or ResNet50(nn.Module))
    def __init__(self, num_classes = 5, pretrained = True):
        ......
        self.model = models.resnet18(pretrained = pretrained) # or models.resnet50
```

One can set whether he/she wants to use the **pretrained** ResNet model, or train the ResNet from scratch, by setting the boolean variable `pretrained` .



```
(fc): Linear(in_features=512, out_features=1000, bias=True)
```

Note that the original model structure has a linear layer as the last layer, and the layer has **512** input neurons and **1000** output neurons, since the ResNet was trained on **ImageNet** dataset which contains 1000 classes of objects. However, we only have **5** classes here, and these 5 classes are *special* things, far from daily-life objects. Therefore, we need to **reinitialize** and **train** this specific layer, *no matter we are using the pretrained weights or not*!

```
class ResNet18(nn.Module): #(or ResNet50(nn.Module))
    def __init__(self, num_classes = 5, pretrained = True):
        ......
        num_of_fc_input_neurons = self.model.fc.in_features # 512
        self.model.fc = nn.Linear(num_of_fc_input_neurons, num_classes)
```

Under the `pretrained = True` mode, the ResNet models are loaded in **with the pretrained weights**. Aside from the to-be-trained linear layer at the end, **all the weights from previous layers are frozen**. They are kept non-trainable at first few epochs.

```
class ResNet18(nn.Module): #(or ResNet50(nn.Module))
    def __init__(self, num_classes = 5, pretrained = True):
        ......
        if pretrained:
            for param in self.model.parameters():
                param.require_grads = False
```

Note that these lines are positioned before the part where we interchange the final linear layer, thus **the replaced new linear layer is still trainable.**
The reason why I do this is that I basically split the training session into 2 steps: the ***feature extraction (FE)*** step and the ***finetuning (FT)*** step.
Under the `pretrained = True` mode, the first few epochs will be used to train *only* the linear classifier at the end. Meanwhile, the previous layers are all frozen and act like a *feature extractor* for now, so that the inputs for the last linear layer are actually a kind of *feature representations* of the input images. These feature representations are good representations coming from the pretrained ResNet. The linear classifier can thus learn to classify on these "good" representations.
Now, nevertheless, we should still unfreeze these previous

layers, and ***finetune*** in the later epochs. The goal of finetuning is to slightly adjust so that the model can work with the new dataset better, rather than overwrite the generic learning.
Hence, under the pretrained mode, the training processes look like so (taking ResNet50 for example):

```
ResNet50_with_pretrained = ResNet50()
'''Feature Extraction Step'''
params_FC_layer = []
for param_name, param in ResNet50_with_pretrained.named_parameters():
    if param.requires_grad:
        params_FC_layer.append(param)
optimizer = optim.SGD(params_FC_layer, lr = lr, momentum = momentum,
                      weight_decay = weight_decay)
train_acc1, test_acc1 = train(ResNet50_with_pretrained, train_dataloader,
                              test_dataloader, loss_fn, optimizer, epochs50_FE,
                              device, "ResNet50_with_pretrained")
'''Fine-tuning step'''
for param in ResNet50_with_pretrained.parameters():
    param.requires_grad = True
optimizer = optim.SGD(ResNet50_with_pretrained.parameters(), lr = lr,
                      momentum = momentum, weight_decay = weight_decay)
train_acc2, test_acc2 = train(ResNet50_with_pretrained, train_dataloader,
                              test_dataloader, loss_fn, optimizer, epochs50_FT,
                              device, "ResNet50_with_pretrained")
'''concat'''
train_acc = np.append(train_acc1, train_acc2)
test_acc = np.append(test_acc1, test_acc2)
```

In ***feature extraction*** step, we pick out those trainable parameters, which are in fact those in the linear classifier, and train the model by only updating them. The first few epochs will be used for feature extraction.
Later in ***finetuning*** step, we reset all the parameters to the *trainable* state, and then train the whole model. Since the original ResNet model is good enough, what the training processes do are probably making slight adaption to the current dataset.

As for the `pretrained = False` mode, all the parameters are randomly initialized, meaning that the models are trained ***from scratch***.
Similary, taking the ResNet50 for example, the code implementation of the training processes look like so,

```
ResNet50_without_pretrained = ResNet50(pretrained = False)
optimizer = optim.SGD(ResNet50_without_pretrained.parameters(), lr = lr,
                      momentum = momentum, weight_decay = weight_decay)
```

A.1 The hyper-parameters settings

- ***ResNet18***
    - ***Without Pretraining***
        - batch size: **32**
        - epochs: **15**
        - optimizer: **SGD**
            - learning rate: 10^{-3}
            - momentum: **0.9**
            - weight decay: 5×10^{-4}
        - loss function: **cross entropy loss**
    - ***With Pretraining***
        - batch size: **32**
        - epochs:
            - feature extraction step: **5**
            - finetuning step: **10**
        - optimizer: **SGD**
            - learning rate: 10^{-3}
            - momentum: **0.9**
            - weight decay: 5×10^{-4}
        - loss function: **cross entropy loss**
- ***ResNet50***
    - ***Without Pretraining***
        - batch size: **8**
        - epochs: **10**
        - optimizer: **SGD**
            - learning rate: 10^{-3}
            - momentum: **0.9**
            - weight decay: 5×10^{-4}
        - loss function: **cross entropy loss**
    - ***With Pretraining***
        - batch size: **8**
        - epochs:
            - feature extraction step: **3**
            - finetuning step: **7**
        - optimizer: **SGD**
            - learning rate: 10^{-3}

        - momentum: **0.9**
        - weight decay: 5×10^{-4}
    - loss function: **cross entropy loss**

B. The Details of my Dataloader

Before making my own customized dataset, I first define a function called `getData()` to load in all the file names of the image data as well as their ground truth labels, no matter we're retrieving training data or testing data. The following code snippet has been provided in the lab spec.

```python
def getData(mode):
    if mode == 'train':
        img = pd.read_csv('train_img.csv')
        label = pd.read_csv('train_label.csv')
        return np.squeeze(img.values), np.squeeze(label.values)
    else:
        img = pd.read_csv('test_img.csv')
        label = pd.read_csv('test_label.csv')
        return np.squeeze(img.values), np.squeeze(label.values)
```

Now, in respect of my customized dataset ***RetinopathyDataset()***, it is inherited from the `Dataset` class from `torch.utils.data` package. We need to define 3 functions inside the customized dataset, namely `__init__()` , `__len__()` and `__getitem__()` .
I will describe what I've done inside each of them one by one at the below section.

```python
class RetinopathyDataset(Dataset):
    def __init__(self, root, mode, mean, std):
        self.root = root
        self.mode = mode
        self.img_names, self.labels = getData(self.mode)
        self.length = len(self.img_names)
        self.transformations = transforms.Compose([
            transforms.RandomHorizontalFlip(),
            transforms.RandomVerticalFlip(),
            transforms.ToTensor(),
            transforms.Normalize(mean, std)
        ])
        print("> Found %d images..." % (self.length))
    def __len__(self):
        return self.length
    def __getitem__(self, index):
        img_name = os.path.join(self.root, self.img_names[index] + '.jpeg')
        img = Image.open(img_name)
        img = self.transformations(img)
        label = self.labels[index]
        return img, label
```

The **__init__** function is where the initial logic happens. Here we define several *attributes* which are available later.

1. `self.root` records the **folder path to all data**, including both training data and testing data.
2. `self.mode` indicates whether we are retrieving training data or testing data, the value can either be **“train”** or **“test”**.
3. `self.img_names`, `self.labels` store the file names of all the image data and their ground truth labels, respectively. The data are gathered by the aforementioned function `getData()`, with `self.mode` specifying if we are collecting trainind data, or testing data.
4. `self.length` stores the length of the dataset, which happens to be the length of the numpy array of the file names in `self.img_names`.
5. `self.transforms` is an attribute composed of several transformations, wrapped in the `torchvision.transforms.Compose()` method. The transformations I implemented in my code include randomly **flipping** the images **horizontally** and/or **vertically**, turning the *Image* objects into Pytorch tensors, and then **normalizing** them by the mean and the standard deviation of **each of the 3 channels**.

The reason why we applied the transformations such as random flipping, is that these data are retina photos taken under the same situation. They are too similar. Randomly Flipping these images somehow equivalents to applying regularizations, consequently **lowering the possibility of overfitting to some frequent patterns**. Also, this will help the CNN-based models enhance the learning on spatial variability.
As for the reason why I applied the image normalization step, is that the normalization can actually **accelerate the convergence speed of the training processes**. Since, if we don’t normalize the data, the value of each pixel falls in the range $[0, 255]$ and can have larger variances, which makes the converging step more challenging. Another benefit is that many algorithms perform better with

standardrized data as they **assume features to have a standard Gaussian form** (mean $\mu = 0$ and standard deviation $\sigma = 1$).
Both training data and testing data are standardrized by the **mean and standard deviation of training data**, which makes more sense since during inference, *the image is likely to be input one-by-one but not a bunch*. Here, I use the **original** data (without any random flips) to compute the mean and standard deviation tensors, by replacing this part

```
self.transformations = transforms.Compose([
            transforms.RandomHorizontalFlip(),
            transforms.RandomVerticalFlip(),
            transforms.ToTensor(),
            transforms.Normalize(mean, std)
        ])
```

with this single line

```
self.transformations = transforms.ToTensor()
```

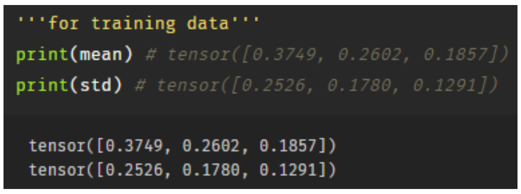
And then, I wrote a function `get_stat()` to compute the statistics.

```
def get_stat(dataloader):
    mean, std = 0.0, 0.0
    for imgs, _ in dataloader:
        imgs = imgs.view(imgs.size(0), imgs.size(1), -1)
        mean += imgs.mean(2).sum(0)
        std += imgs.std(2).sum(0)
    mean /= len(dataloader.dataset)
    std /= len(dataloader.dataset)
    return mean, std
```

So, the mean as well as the standard deviation for each channel are calculated as:

1. Taking the mean/standard deviation over the whole $512 \times 512 = 262144$ pixels for each of the 3 channels, respectively.
2. Taking the average of the means/standard deviations over the whole dataset (**28099** images for **training** data and **7025** images for **testing** data).

The results of the mean tensor and the standard deviation tensor for the 3 channels are:



```
'''for training data'''
print(mean) # tensor([0.3749, 0.2602, 0.1857])
print(std) # tensor([0.2526, 0.1780, 0.1291])

 tensor([0.3749, 0.2602, 0.1857])
 tensor([0.2526, 0.1780, 0.1291])
```

The **__len__** function returns the length of the dataset. Since we've defined the attribute `self.length` which indicates the length of the dataset in the __init__ function, we will directly return it here.

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

Finally, the **__getitem__** function return a single image and its label by the index number specified in the `index` argument.

```
def __getitem__(self, index):
    img_name = os.path.join(self.root, self.img_names[index] + '.jpeg')
    img = Image.open(img_name)
    img = self.transformations(img)
    label = self.labels[index]
    return img, label
```

The function retrieve a single image and its label from the root folder, open the image into an *Image* object by `open()` method in `PIL.Image` class, and then conduct a series of transformations defined in `self.transformations()` beforehand. Finally, return both the transformed image and its corresponding label.

C. Describing my Evaluation Through the Confusion Matrix

In order to plot the confusion matrixes, we need to get the predictions of the testing data out of the model. For this, I have written a function called `get_predictions()` which uses the model passed in to output the predictions, and returns these predictions as well as the ground truths.

```
def get_predictions(model, test_dataloader, device):
    model.eval()
    predictions = torch.tensor([], dtype = torch.float, device = device)
    ground_truths = torch.tensor([], dtype = torch.long, device = device)
    with torch.no_grad():
        for X_batched, y_batched in test_dataloader:
            X_batched = X_batched.to(device, dtype = torch.float)
            y_batched = y_batched.to(device, dtype = torch.long)
            ground_truths = torch.cat((ground_truths, y_batched))
            outputs = model(X_batched)
            predictions = torch.cat((predictions, outputs.max(dim = 1)[1]))
    return predictions.detach().cpu().numpy(), ground_truths.detach().cpu().numpy(
```

As we mention before, there are totally 4 models we have tried: $(ResNet18,\ w/\ pretraining)$, $(ResNet18,\ w/o\ pretraining)$,

$(ResNet50,\ w/\ pretraining)$,
$(ResNet50,\ w/o\ pretraining)$.

```
all_models = [ResNet18(pretrained = False), ResNet18(pretrained = True),
              ResNet50(pretrained = False), ResNet50(pretrained = True)]
name = ["ResNet18_without_pretrained", "ResNet18_with_pretrained",
        "ResNet50_without_pretrained", "ResNet50_with_pretrained"]
```

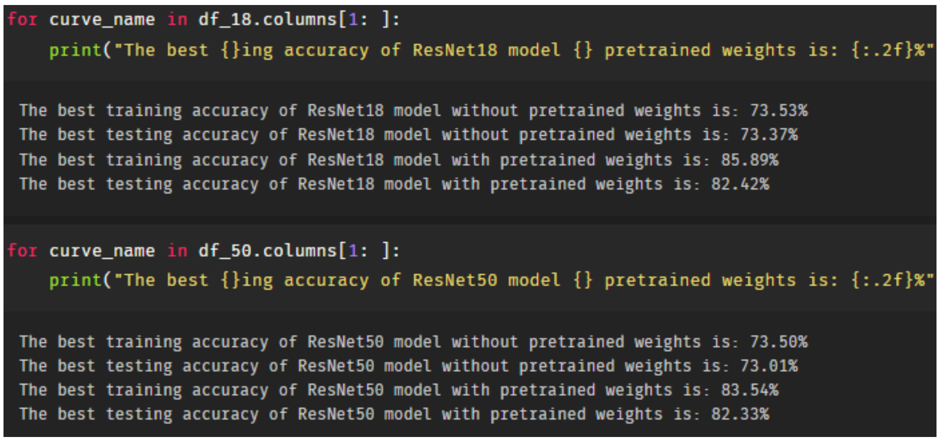
We will load the weights with the best accuracy of each of these 4 models in one by one, and then get the predictions via the `get_predictions()` function we just explained.

```
for i in range(4):
    model = all_models[i]
    model.load_state_dict(torch.load(os.path.join("D:/DL_lab3_params",
                                name[i] + ".pt"), map_location = device))
    model.to(device)
    predictions, ground_truths = get_predictions(model, test_dataloader, device)
    cm = confusion_matrix(ground_truths, predictions, normalize = 'true')
    cm_plot = ConfusionMatrixDisplay(cm)
    fig, ax = plt.subplots(figsize = (8, 8))
    cm_plot.plot(cmap = 'coolwarm', values_format = '.2f', ax = ax)
    plt.savefig(os.path.join("C:\\Users\\GL75\\OneDrive\\桌面\\深度學習\\Lab3",
                         name[i] + ".png"))
```

After both the predictions and the ground truths are retrieved, these 2 will be used to compute a ***normalized*** confusion matrix by the `confusion_matrix` function from the `sklearn.metrics` package. And then, we will wrap them inside a `ConfusionMatrixDisplay` object, and use the **plot()** method to visualize the confusion matrixes.
The confusion matrixes are **normalized over the ground true conditions**, and are rounded to the second decimal place. For the results, please refer to the later ***Experimental Results*** section.

3. Experimental Results

A. The Highest Testing Accuracy



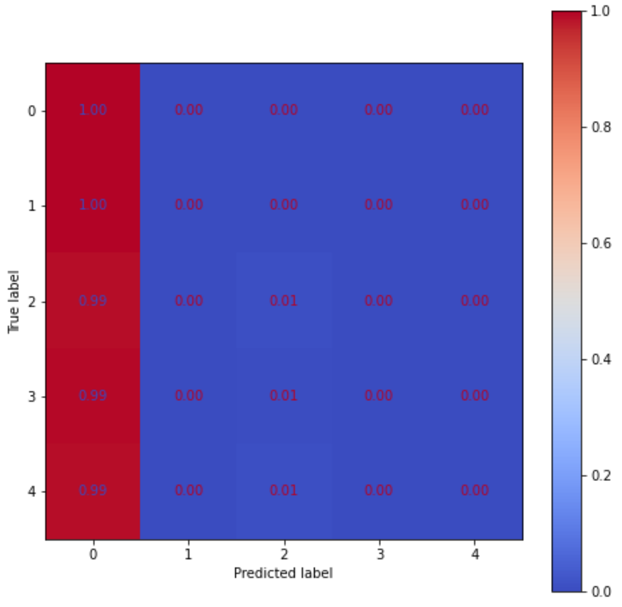
```
for curve_name in df_18.columns[1: ]:
    print("The best {}ing accuracy of ResNet18 model {} pretrained weights is: {:.2f}%"

The best training accuracy of ResNet18 model without pretrained weights is: 73.53%
The best testing accuracy of ResNet18 model without pretrained weights is: 73.37%
The best training accuracy of ResNet18 model with pretrained weights is: 85.89%
The best testing accuracy of ResNet18 model with pretrained weights is: 82.42%
```

```
for curve_name in df_50.columns[1: ]:
    print("The best {}ing accuracy of ResNet50 model {} pretrained weights is: {:.2f}%"

The best training accuracy of ResNet50 model without pretrained weights is: 73.50%
The best testing accuracy of ResNet50 model without pretrained weights is: 73.01%
The best training accuracy of ResNet50 model with pretrained weights is: 83.54%
The best testing accuracy of ResNet50 model with pretrained weights is: 82.33%
```

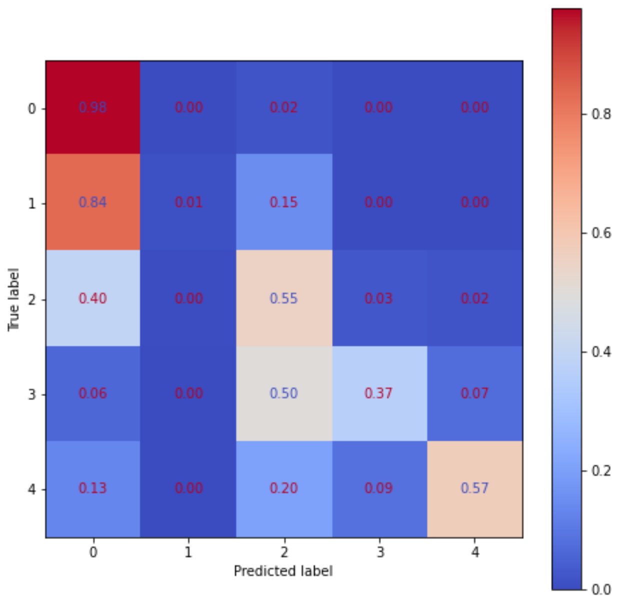
The best **training** accuracy is **85.89%**, and the best **testing** accuracy is **82.42%**, both coming from the **ResNet18** model **with** pretraining.
Also, one can apparently see that no matter ResNet18 or ResNet50, and no matter in training session or in testing session, models **with** pretrained weights outperform those **without** pretrained weights (trained from scratch) (around 73% vs. around 82%).
I reckon that this phenomenon might be contributed to the fact that models with pretrained weights already have **a better initialization of parameters**, compared to those models trained from scratch. To train a model without any pretraining is like climbing a mountain without any map, which will take a great effort to summit. On the other hand, finetuning a pretrained model is like trying to summit a mountain from a spot close to the top of the mountain. The model can easily reach a relatively good performance within short epochs.
As for why the ResNet50 models do not have better performances than the ResNet18, I assume that the reason might comes from **shorter epochs** and **smaller batch sizes**. In this implementation, the settings for **ResNet18** are **epochs = 15** and **batch sizes = 32**, whereas the settings for **ResNet50** are **epochs = 10** and **batch sizes = 8**. I believe that if we extend the epochs and batch sizes to the same degree as those in ResNet18, then ResNet50 will likely surpass ResNet18 under the same experimental conditions. However, I did not and could not try this, due to my *GPU memory limit* and *lengthy training time*. It takes so long even to train an epoch.

Below are confusion matrixes of the classification from the 4 models.

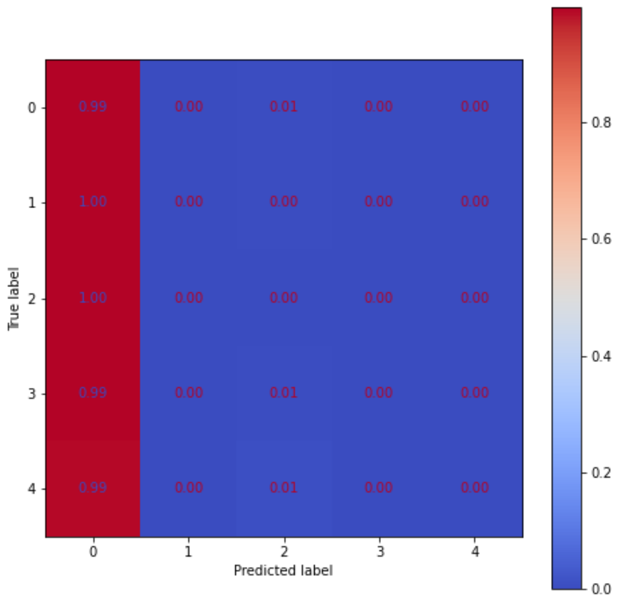
ResNet18 without pretraining



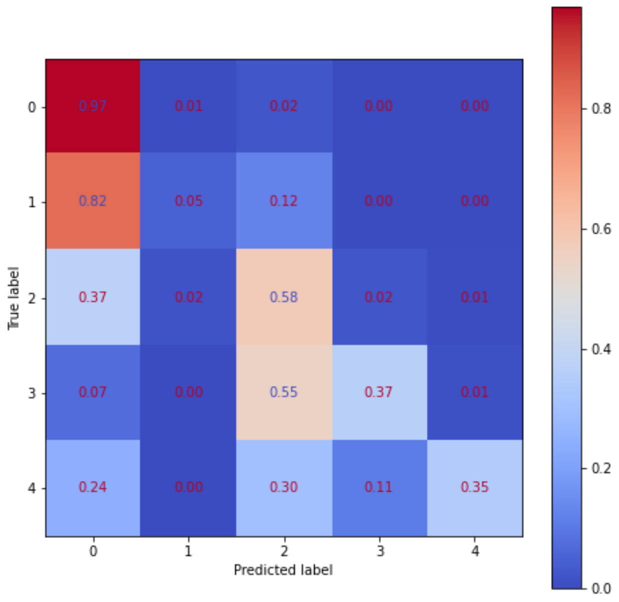
ResNet18 with pretraining



ResNet50 without pretraining



ResNet50 with pretraining



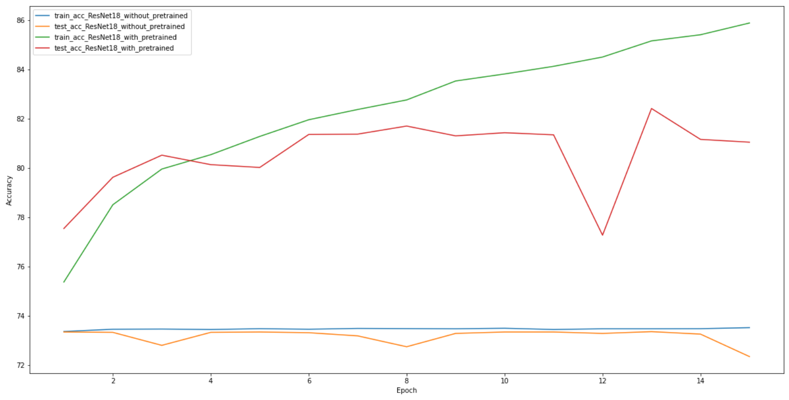
As the above figures illustrate, the models **without** any pretraining will probably learn to **guess all outcomes** to be class **0**. But even doing so, the accuracy of this kind of stupid models can still reach around **73%**! This gives a hint that the datasets might be severely **imbalanced**!!! We will check them out, and try to overcome this problem later in the following **Discussion** section.

As for the models with pretraining, the confusion matrixes look much better now. However, most of **ground truth label = 1 (mild DR)** were misclassified into **label = 0 (No DR)**. I think this is excusable, since patients with mild diabetic retinopathy might merely have any symptom, even doctors might sometimes neglect the lesion. Another often misclassified portion is **ground truth = 3 (severe DR)**. Although around 37% are correctly classified, there are still 55% erroneously classified into **label = 2 (moderate DR)**. This is somehow understandable, since the boundary between “severe” and “moderate” might subtle. This boundary is human-defined after all.

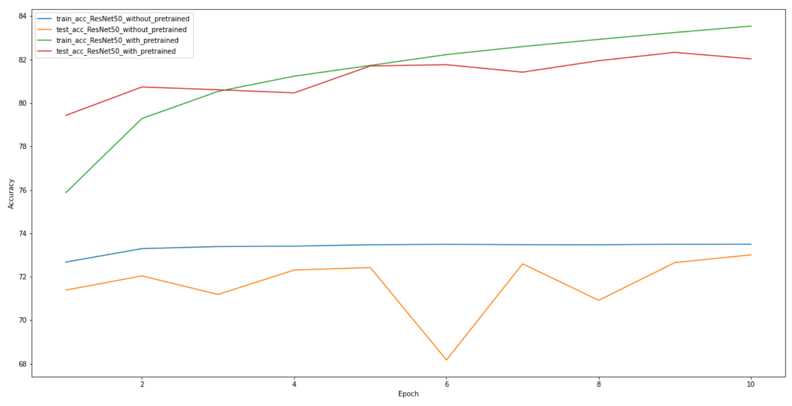
B. Comparison Figures

```python
def plot_acc_curves(df, save_name):
    fig = plt.figure(figsize = (20, 10))
    for curve_name in df.columns[1: ]:
        plt.plot("epoch", curve_name, data = df)
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.legend()
    plt.savefig(os.path.join("C:\\Users\\GL75\\OneDrive\\桌面\\深度學習\\Lab3",
                             save_name + ".png"))
```

ResNet18



ResNet50



Due to the bad resolution resulted from large images, I will manually reproduce what the legends recorded below:

1. **blue** lines are **training** accuracy of models **without** pretraining
2. **yellow** lines are **testing** accuracy of models **without** pretraining
3. **green** lines are **training** accuracy of models **with** pretraining
4. **red** lines are **testing** accuracy of models **with** pretraining

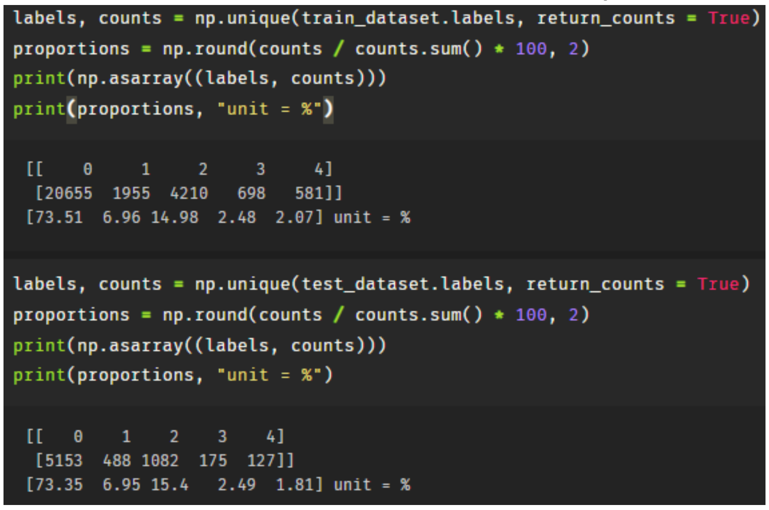
One can see that no matter ResNet18 or ResNet50 structure is used, the models **without** pretrained weights ***barely improve*** during the whole training session. The training accuracy reached 73% at the beginning, and stay like that the whole training. This might be owing to:

1. The models learned to lazily guess all data to be **label = 0**. Even doing so, the models can reach 73% accuracy with little penalty, so there is no motivation for it to improve.
2. 10~15 epochs might be too few, far less than the epochs needed to converge.

On the other hand, the models with pretraining not only reached quite a good performance at the beginning, but also kept improving during the training session.

4. Discussion

From the confusion matrixes of those models without pretraining, one can observe that these models blindly predict all the data to be in class **0**. I suspect that this phenomenon might be by cause of ***severely imbalanced dataset***. So, I conducted a little analysis



```
labels, counts = np.unique(train_dataset.labels, return_counts = True)
proportions = np.round(counts / counts.sum() * 100, 2)
print(np.asarray((labels, counts)))
print(proportions, "unit = %")

 [[    0     1     2     3     4]
  [20655  1955  4210   698   581]]
 [73.51  6.96 14.98  2.48  2.07] unit = %
```

```
labels, counts = np.unique(test_dataset.labels, return_counts = True)
proportions = np.round(counts / counts.sum() * 100, 2)
print(np.asarray((labels, counts)))
print(proportions, "unit = %")

 [[   0    1    2    3    4]
  [5153  488 1082  175  127]]
 [73.35  6.95 15.4   2.49  1.81] unit = %
```

and found out that no matter the training dataset or the testing dataset, the counts for all 5 classes are indeed **severely imbalanced**, with most of the data (around **73%** for both datasets) belonging to **class 0 (no DR)**. This can explain the behavior of these models. Models do have "idleness" like human being does. If most of the time, the ground truths are belonging to class 0 with no surprise, and the *risk* of a wrong guess is relatively small (since most of the data belong to class 0), then the models are likely to guess **0** since this is a **safe guess**.
As a consequence, one way to solve this problem is to employ a ***weighted*** losses. We need to make sure that the models **"focus"** on the hard samples, not be satisfied with having high accuracy on easy samples.
Focal loss was initially introduced in ***RetinaNet***. This kind of loss function is a variant of cross entropy loss and used in binary classification task originally in the paper. We can further extend it to be *multi-class* focal loss.
Let p_t be the predicted probability of the class t by the model, then the traditional cross entropy loss will be:

$$CE(p_t) = -\log(p_t)$$

A modified version of this is the **α-balanced cross entropy**

$$ACE(p_t) = -\alpha \log(p_t)$$

However, this version does not solve the problem that the summation of several losses of easy samples would still be greater than the loss of a hard sample, yet.
Thus, ***focal loss*** introduces another thing called **modulating factor** $(1 - p_t)^\gamma$. So, the focal loss is defined as below:

$$FL(p_t) = -\alpha(1 - p_t)^\gamma \log(p_t)$$

Note that the ordinary cross entropy can be deemed as a special case of focal loss, with

$$\alpha = 1, \gamma = 0$$

Following is my implementation of focal loss:

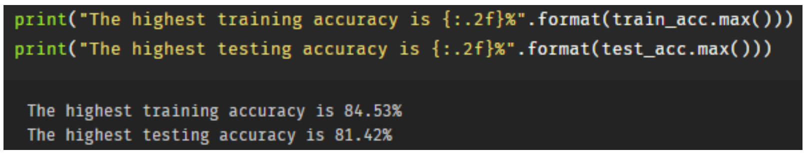
```
import torch.nn as nn
import torch.nn.functional as F

class FocalLoss(nn.modules.loss._WeightedLoss):
    def __init__(self, weight = None, gamma = 2, reduction = 'mean'):
        super(FocalLoss, self).__init__(weight, reduction = reduction)
        self.gamma = gamma
        self.reduction = reduction
        self.weight = weight # the weight to balance the classes.
    def forward(self, inputs, targets):
        CEloss = F.cross_entropy(inputs, targets, reduction = self.reduction,
        weight = self.weight)
        Pt = torch.exp(-CEloss)
        focal_loss = ((1 - Pt) ** self.gamma * CEloss).mean()
        return focal_loss
```

(reference:
https://github.com/gokulprasadthekkel/pytorch-multi-class-focal-loss/blob/master/focal_loss.py
(https://github.com/gokulprasadthekkel/pytorch-multi-class-focal-loss/blob/master/focal_loss.py))
`CEloss` is the ordinary cross entropy, which equals to $-\log(p_t)$. So, we can reversely obtained p_t, like so:

$$CE(p_t) = -\log(p_t)$$
$$\therefore\ p_t = e^{-CE(p_t)}$$

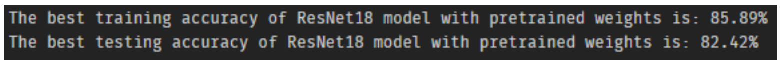
```
CEloss = F.cross_entropy(inputs, targets, reduction = self.reduction,
        weight = self.weight)
Pt = torch.exp(-CEloss)
```

In the original paper of ***RetinaNet***, the parameters are set to be $\gamma = 2$ and $\alpha = 1$. We will leave the settings unchanged.
Also, since the training takes way too long even with GPU, we will only try the **focal loss** on the model with the highest accuracy, namely ***ResNet18 with pretraining***.
Here are the results



```
print("The highest training accuracy is {:.2f}%".format(train_acc.max()))
print("The highest testing accuracy is {:.2f}%".format(test_acc.max()))

 The highest training accuracy is 84.53%
 The highest testing accuracy is 81.42%
```

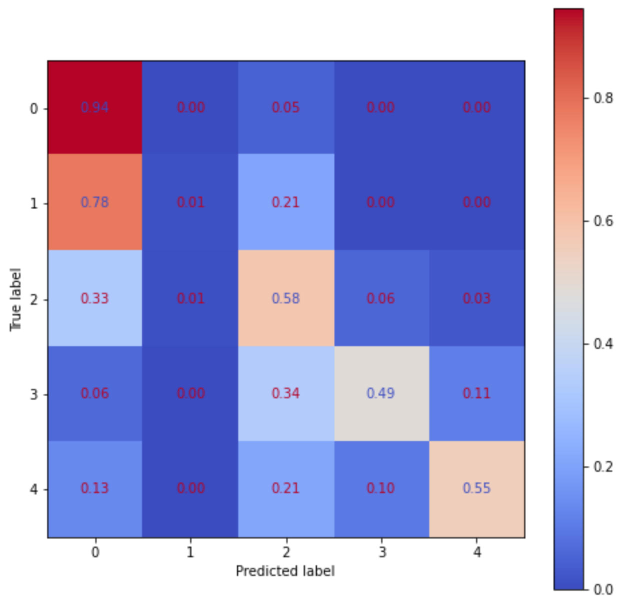
The highest **training** accuracy is **84.53%**, and the highest **testing** accuracy is **81.42%**. Both are slightly worse than the model with standard cross entropy loss:



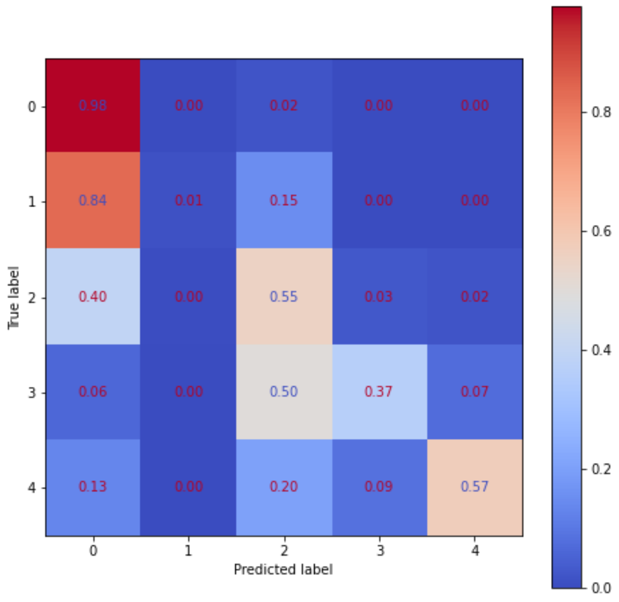
```
The best training accuracy of ResNet18 model with pretrained weights is: 85.89%
The best testing accuracy of ResNet18 model with pretrained weights is: 82.42%
```

Nonetheless, if we focus on the **confusion matrix**, we can find that the model with the **focal loss** does have some improvement.

The model with focal loss



The model with standard cross entropy



Although lots of **class-1** data are still classified into **class-0**, the misclassification concerning **class-3** data being classified into **class-2** are alleviated with the application of *focal loss*!