Deep Learning and Practice
Lab4-2: Diabetic Retinopathy Detection

310605009
吳公耀

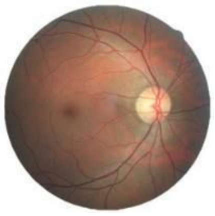
I. Introduction

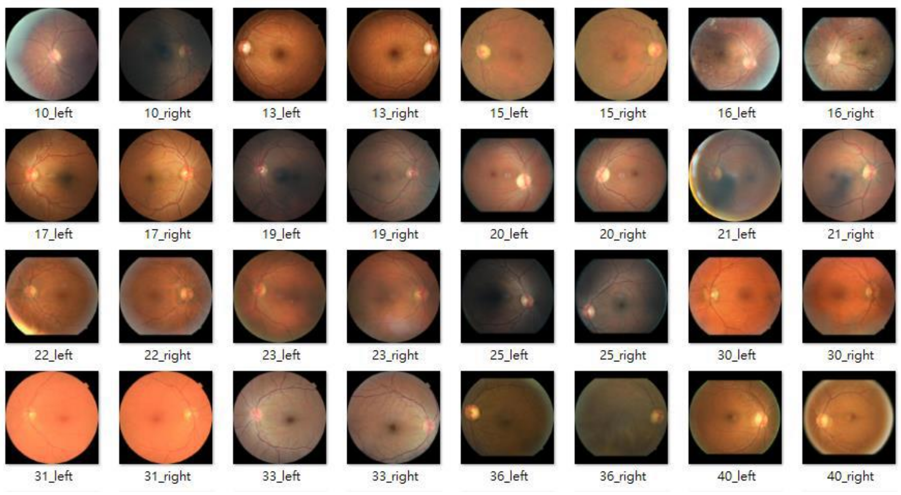
A. Lab Objective

- 分析糖尿病引起的變化
- 實現自己的DataLoade
- 使用Pytorch 構建網絡ResNet18 和ResNet50 有/沒有預訓練模型來訓練和測試數據集
- 計算混淆矩陣來計算性能

B. Dataset

diabetic retinopathy (糖尿病所引發視網膜變 糖尿病所引發視網膜變)是一種眼部疾病，可導致糖尿病患者視力喪失和失明。它會影響視網膜中的血管（眼睛後部的感光組織層）。且是發達國家工作年齡人口失明的主要原因。據估計，它將影響超過 9300 萬人。該數據集提供了在各種成像條件下拍攝的大量高分辨率視網膜圖像

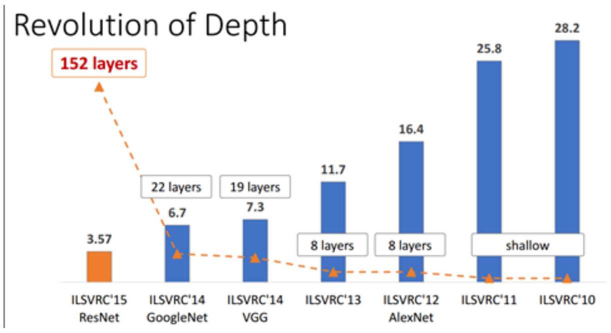




Prepare Data The dataset contains 35124 images, we divided dataset into 28,099 training data and 7025 testing data. The image resolution is 512x512 and has been preprocessed.
Download link:
https://drive.google.com/open?id=1RTmrk7Qu9IBjQYLczaYKOvXaHWBS0o72

II. Experiment setups

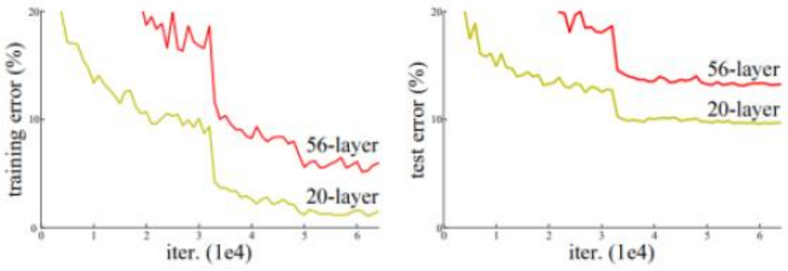
ResNet



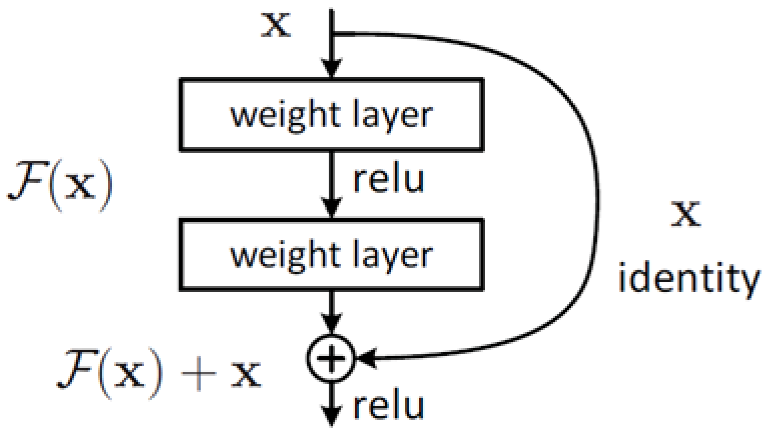
ResNet是解決了深度CNN模型難訓練的問題，從上層圖中可以看到VGG才19層，而 ResNet多達152層，這在網絡深度完全不是一個量級上，所以如果是第一眼看這個圖的話，肯定會覺得ResNet是只是因為深度取勝。事實當然是這樣，但是即使只是單純加深模型並不會用遠帶來正面的效果，深的網路比較容易訓練不起來，使得更深的網路有時反而帶來更差的效果。而ResNet提出有構上的trick，這才使得網絡的深度發揮出作用，這個trick就是殘差學習（Residual learning）。

Degradation problem

上面所說的更深的網路有時反而帶來更差的效果現實驗發現深度網絡出現了退化問題：網絡深度增加時，網絡準確度出現飽和，甚至出現下降。這個現象可以在下圖中直觀看出來：56層的網絡比20層網絡效果還要差。這不會是過擬合問題，因為56層網絡的訓練誤差同樣高。我們知道深層網絡存在著梯度消失或者爆炸的問題，這使得深度學習模型很難訓練。



Skip/Shortcut connection: 為了解決梯度消失/爆炸的問題，添加了一個跳過/快捷連接以在幾個權重層之後將輸入 x 添加到輸出，如下所示


x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

殘差學習residual learning

$$y_l = h(x_l) + F(x_l, W_l)$$
$$x_{l+1} = f(y_l)$$

其中 xl 和 xl+1 分別表示的是第 ℓ 個殘差單 元的輸入和出，每元一般包含多層結構。 F是殘差函數，表示學習到的而 是殘差函數，表示學習到的而 $h(xl)=xl$ 表 示恆等映射， f是 ReLU激活函數。 因此 求得從淺層到深的學習特徵為 ：

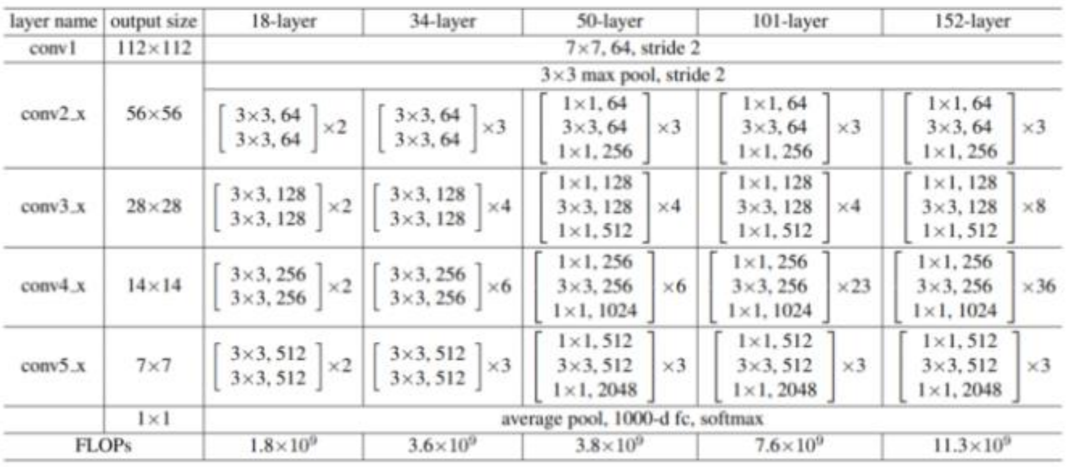
$$\mathbf{x} = x + \sum_{i=l}^{L-1} F(x_i, W_i)$$

利用Chain rule，可以求得反向過程的梯度：

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

式子的第一個因子$\partial loss/\partial xl$ 表示的損失函數到達L的梯度，小括號中的1表明短路機制可以無損地傳播梯度，而另外一項殘差梯度則需要經過帶有weights的層，梯度不是直接傳遞過來的。殘差梯度不會那麼巧全為-1，而且就算其比較小，有1的存在也不會導致梯度消失。所以殘差學習會更容易。

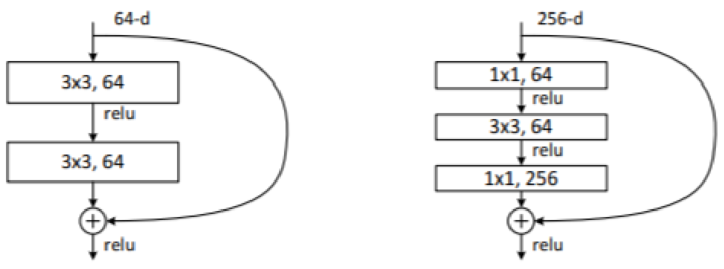
不同深度的**ResNet**架構



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		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

不同深度的ResNet的差別有兩個：

- ResNet18、ResNet34使用一般的residual block，而ResNet50、ResNet101、ResNet152使用了expansion為4的bottleneck block。
- 剩下的差異就在於每個stage堆疊的building block層數不同，詳細差異可以參考下面的表格。



疊更深的網路時，ResNet設計了bottleneck (building) block，降低3x3 convolution的寬度，大幅減少了所需的運算量。

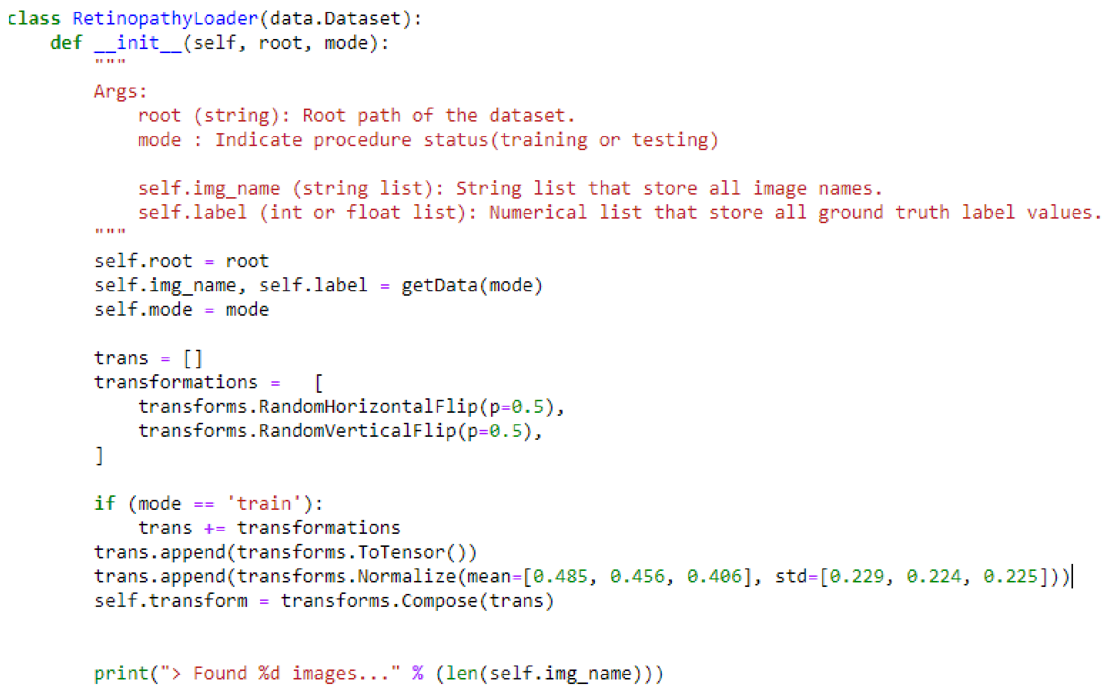
A. The details of your model (ResNet)

利用Pytorch套件，創建ResNet18以及ResNet50

```
#load model
if models[choose_model] == "resnet18":
    model = torchvision.models.resnet18(pretrained = pretrain)
elif models[choose_model] == "resnet50":
    model = torchvision.models.resnet50(pretrained = pretrain)

fc_features = model.fc.in_features
model.fc = nn.Linear(fc_features, 5) # change output layer
model.to(device)

Loss = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr = lr, momentum = 0.9, weight_decay = 5e-4)
```

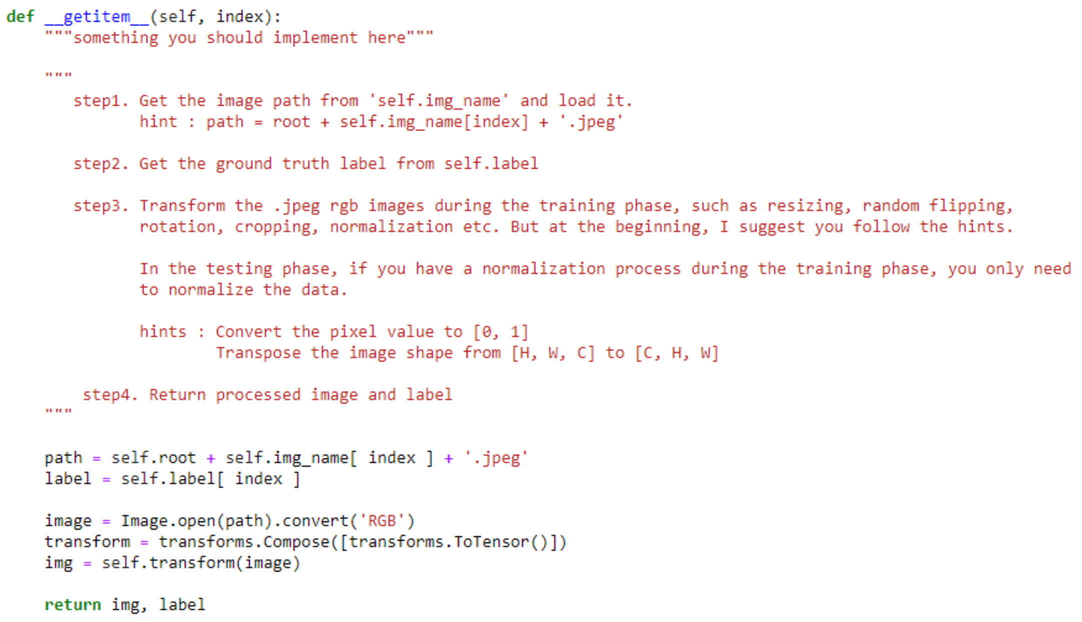
B. The details of your Dataloader

對訓練資料進行預處理和資料增強，逐 channel 地對影像進行標準化還有對圖片的隨機水平垂直翻轉得到最好的準確率。



```
class RetinopathyLoader(data.Dataset):
    def __init__(self, root, mode):
        """
        Args:
            root (string): Root path of the dataset.
            mode : Indicate procedure status(training or testing)

            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

        trans = []
        transformations =   [
            transforms.RandomHorizontalFlip(p=0.5),
            transforms.RandomVerticalFlip(p=0.5),
        ]

        if (mode == 'train'):
            trans += transformations
        trans.append(transforms.ToTensor())
        trans.append(transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]))
        self.transform = transforms.Compose(trans)


        print("> Found %d images..." % (len(self.img_name)))
```

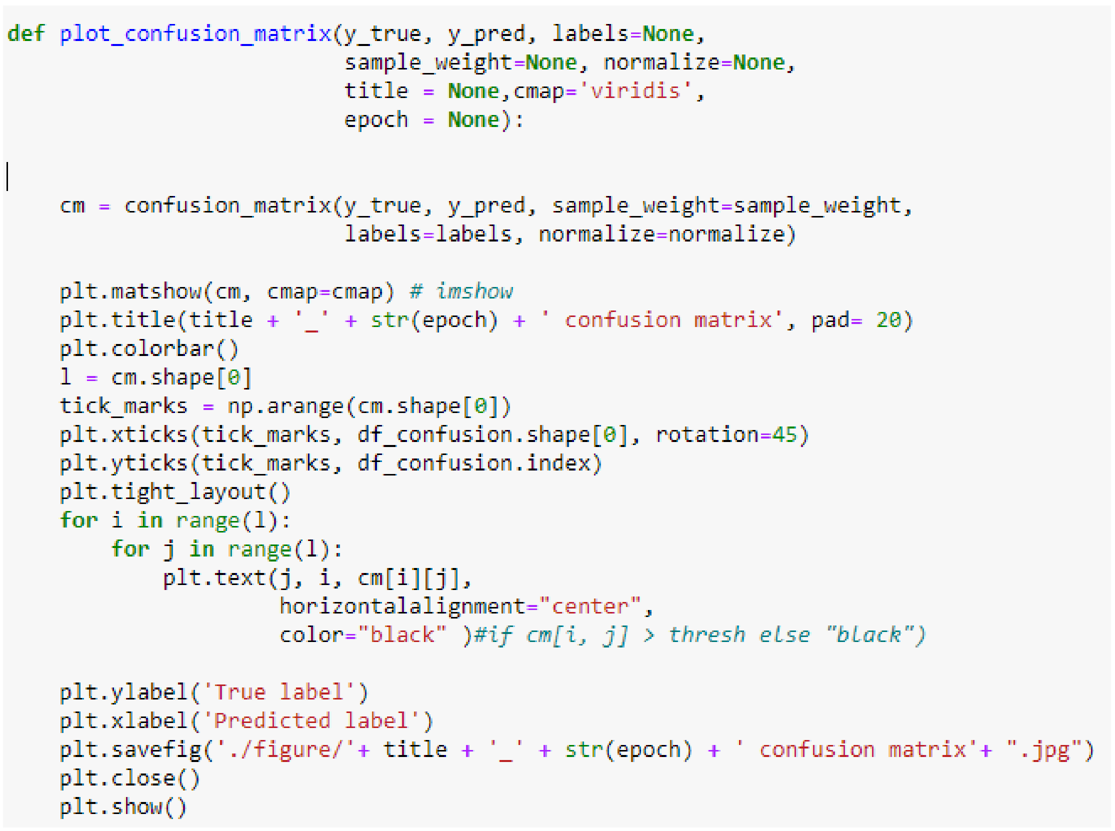
利用助教給的csv檔，進行檔名及副檔名的配對讀取



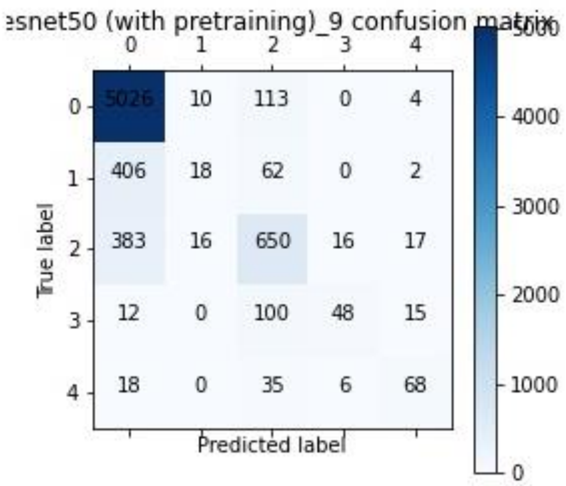
```
    def __getitem__(self, index):
        """something you should implement here"""

        """
           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
        """

        path = self.root + self.img_name[ index ] + '.jpeg'
        label = self.label[ index ]

        image = Image.open(path).convert('RGB')
        transform = transforms.Compose([transforms.ToTensor()])
        img = self.transform(image)

        return img, label
```

C. Describing your evaluation through the confusion matrix

混淆矩陣(Confusion matrix) 是一種可視化工具，用來評估哪些類別容易搞錯，而同一列代表同一個標籤，搭配每一行，代表那個列的標籤被判斷成哪個標籤(行)。



```python
def plot_confusion_matrix(y_true, y_pred, labels=None,
                          sample_weight=None, normalize=None,
                          title = None,cmap='viridis',
                          epoch = None):

    cm = confusion_matrix(y_true, y_pred, sample_weight=sample_weight,
                          labels=labels, normalize=normalize)

    plt.matshow(cm, cmap=cmap) # imshow
    plt.title(title + '_' + str(epoch) + ' confusion matrix', pad= 20)
    plt.colorbar()
    l = cm.shape[0]
    tick_marks = np.arange(cm.shape[0])
    plt.xticks(tick_marks, df_confusion.shape[0], rotation=45)
    plt.yticks(tick_marks, df_confusion.index)
    plt.tight_layout()
    for i in range(l):
        for j in range(l):
            plt.text(j, i, cm[i][j],
                     horizontalalignment="center",
                     color="black" )#if cm[i, j] > thresh else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.savefig('./figure/'+ title + '_' + str(epoch) + ' confusion matrix'+ ".jpg")
    plt.close()
    plt.show()
```



所以在左上到右下的對角線代表預測正確的結果，準確率也就是預測正確的結果除以預測的總量，以上圖為例是5026+18+650+48+68/ 7026 = 82.7%。程式碼的部分使用sklearn幫忙把預測結果轉為confusion matrix的格式，在用另外用plt來畫出confusion matrix。

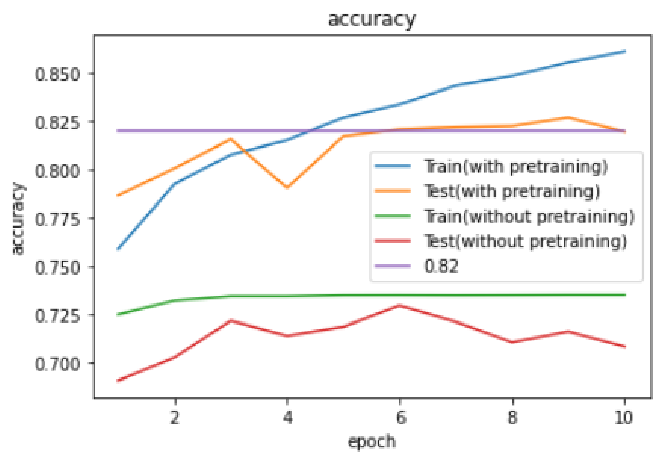
III. Experimental results

A. The highest testing accuracy

- Screenshot

  model: resnet50 (with pretraining), learning rate: 0.01, Batch size: 8, epochs: 10

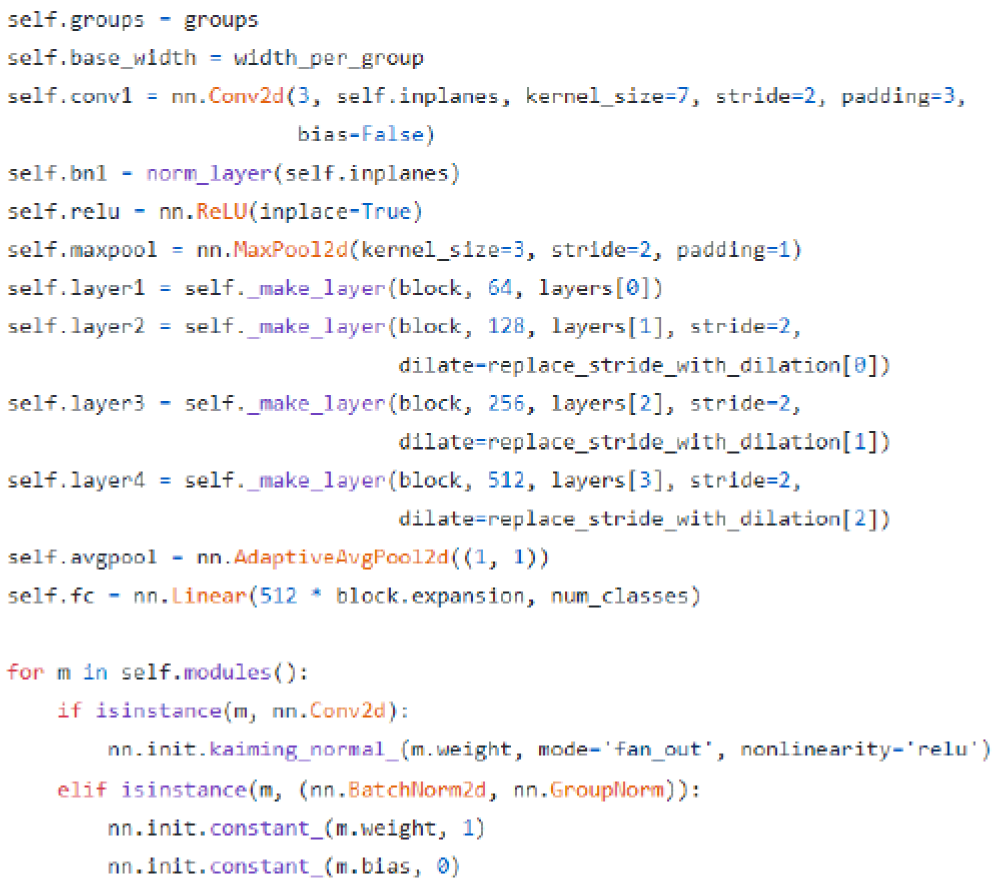
```
Best accuracy: 0.8270462633451957
```



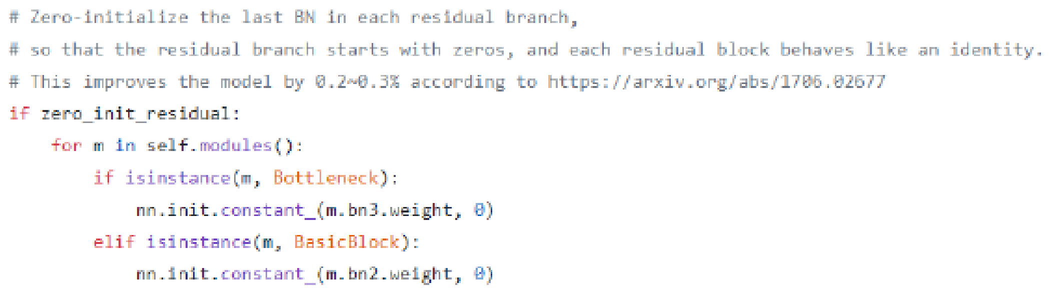
- Anything you want to present

ResNet架構

```
class ResNet(nn.Module):

    def __init__(self, block, layers, num_classes=1000, zero_init_residual=False,
                 groups=1, width_per_group=64, replace_stride_with_dilation=None,
                 norm_layer=None):
        super(ResNet, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        self._norm_layer = norm_layer

        self.inplanes = 64
        self.dilation = 1
        if replace_stride_with_dilation is None:
            # each element in the tuple indicates if we should replace
            # the 2x2 stride with a dilated convolution instead
            replace_stride_with_dilation = [False, False, False]
        if len(replace_stride_with_dilation) != 3:
            raise ValueError("replace_stride_with_dilation should be None "
                             "or a 3-element tuple, got {}".format(replace_stride_with_dilation))
```



```
self.groups = groups
self.base_width = width_per_group
self.conv1 = nn.Conv2d(3, self.inplanes, kernel_size=7, stride=2, padding=3,
                       bias=False)
self.bn1 = norm_layer(self.inplanes)
self.relu = nn.ReLU(inplace=True)
self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
self.layer1 = self._make_layer(block, 64, layers[0])
self.layer2 = self._make_layer(block, 128, layers[1], stride=2,
                               dilate=replace_stride_with_dilation[0])
self.layer3 = self._make_layer(block, 256, layers[2], stride=2,
                               dilate=replace_stride_with_dilation[1])
self.layer4 = self._make_layer(block, 512, layers[3], stride=2,
                               dilate=replace_stride_with_dilation[2])
self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
self.fc = nn.Linear(512 * block.expansion, num_classes)

for m in self.modules():
    if isinstance(m, nn.Conv2d):
        nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
    elif isinstance(m, (nn.BatchNorm2d, nn.GroupNorm)):
        nn.init.constant_(m.weight, 1)
        nn.init.constant_(m.bias, 0)
```

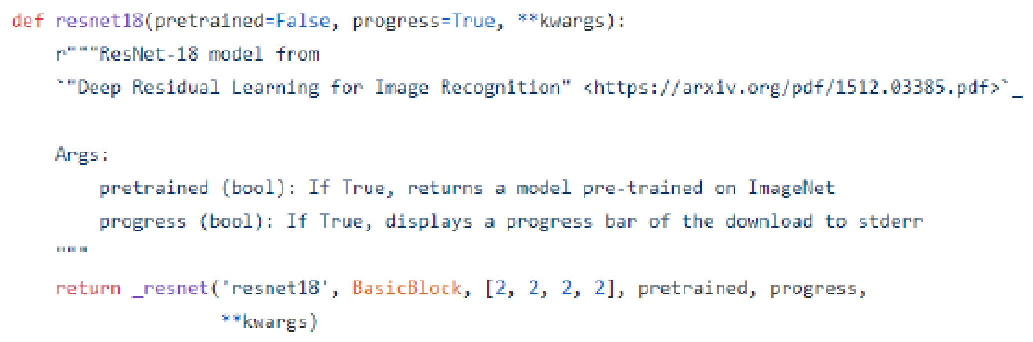


```
# Zero-initialize the last BN in each residual branch,
# so that the residual branch starts with zeros, and each residual block behaves like an identity.
# This improves the model by 0.2~0.3% according to https://arxiv.org/abs/1706.02677
if zero_init_residual:
    for m in self.modules():
        if isinstance(m, Bottleneck):
            nn.init.constant_(m.bn3.weight, 0)
        elif isinstance(m, BasicBlock):
            nn.init.constant_(m.bn2.weight, 0)
```

```python
def _make_layer(self, block, planes, blocks, stride=1, dilate=False):
    norm_layer = self._norm_layer
    downsample = None
    previous_dilation = self.dilation
    if dilate:
        self.dilation *= stride
        stride = 1
    if stride != 1 or self.inplanes != planes * block.expansion:
        downsample = nn.Sequential(
            conv1x1(self.inplanes, planes * block.expansion, stride),
            norm_layer(planes * block.expansion),
        )

    layers = []
    layers.append(block(self.inplanes, planes, stride, downsample, self.groups,
                        self.base_width, previous_dilation, norm_layer))
    self.inplanes = planes * block.expansion
    for _ in range(1, blocks):
        layers.append(block(self.inplanes, planes, groups=self.groups,
                            base_width=self.base_width, dilation=self.dilation,
                            norm_layer=norm_layer))

    return nn.Sequential(*layers)
```

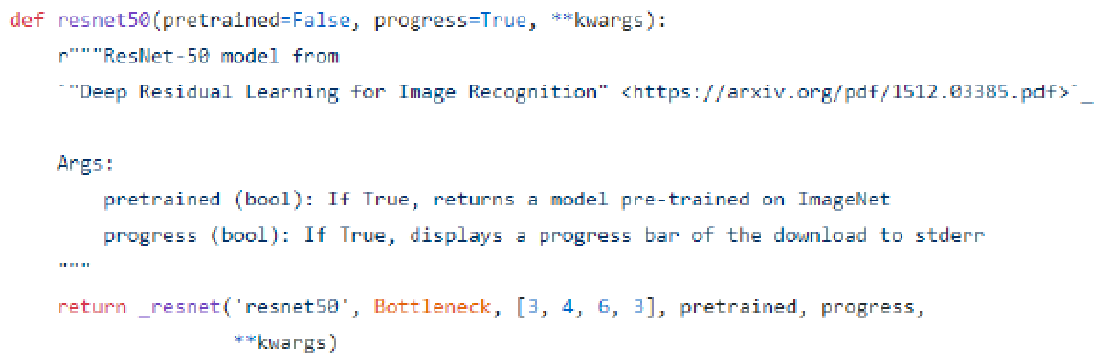
```python
def _forward_impl(self, x):
    # See note [TorchScript super()]
    x = self.conv1(x)
    x = self.bn1(x)
    x = self.relu(x)
    x = self.maxpool(x)

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

    x = self.avgpool(x)
    x = torch.flatten(x, 1)
    x = self.fc(x)

    return x


def forward(self, x):
    return self._forward_impl(x)
```

Resnet18



```
def resnet18(pretrained=False, progress=True, **kwargs):
    r"""ResNet-18 model from
    `"Deep Residual Learning for Image Recognition" <https://arxiv.org/pdf/1512.03385.pdf>`_

    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
        progress (bool): If True, displays a progress bar of the download to stderr
    """
    return _resnet('resnet18', BasicBlock, [2, 2, 2, 2], pretrained, progress,
                   **kwargs)
```

Resnet50

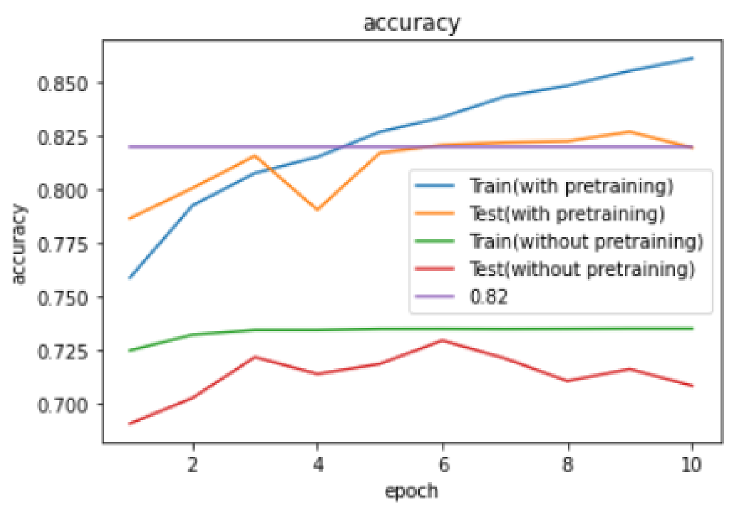


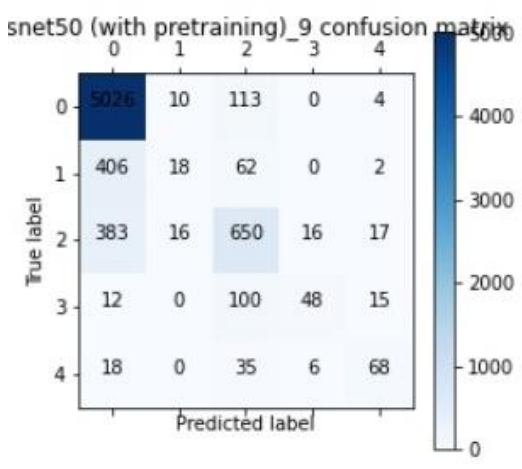
```
def resnet50(pretrained=False, progress=True, **kwargs):
    r"""ResNet-50 model from
    `"Deep Residual Learning for Image Recognition" <https://arxiv.org/pdf/1512.03385.pdf>`_

    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
        progress (bool): If True, displays a progress bar of the download to stderr
    """
    return _resnet('resnet50', Bottleneck, [3, 4, 6, 3], pretrained, progress,
                   **kwargs)
```

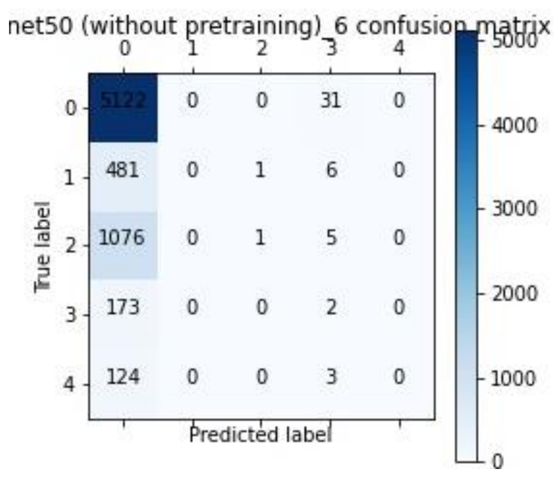
B. Comparison figures

● Plotting the comparison figures (RseNet18/50, with/without pretraining)

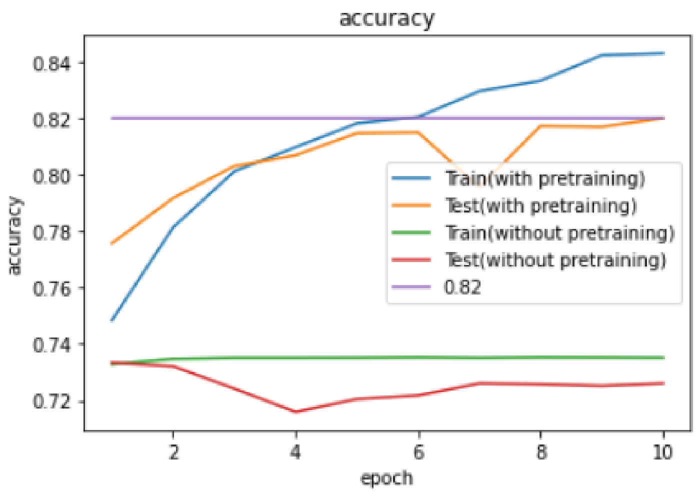
Resnet50

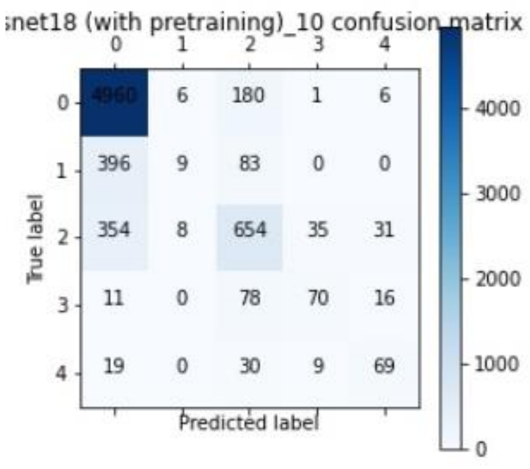


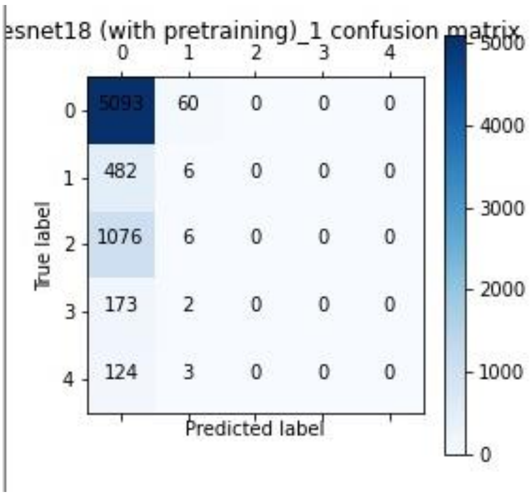




Resnet18







可以從confusion matrix發現，大部分的data都集中在class 0，在training data也有相同情形，所以有pretrained的模型會把大部分的data預測為class 0，少部分為class 2，其他則占了少數。而這樣的結果在without pretrained的模型上就導致模型會把所有data都預測為class 0，我想是因為模型沒有pretrain，generalization的能力不足，只是由一堆隨機數字構成，遇到class 0占多數的

資料集，不能很好的抓出每個類別之間的差異，所以在預測時就會把所有data都預測為class 0，準確率也就無法提升。這也顯示出了pretrained
模型的重要性，有pretrained的模型其實也是相當有更多的data訓練過了，雖然可能與目前的工作的資料集沒有相關，但是可以提高模型的能力，而不是只是一堆由隨機數字組成的模型，在這樣的data下可能就容易overfitting或是學不到資料集的重點。

	pretrained	Without pretrained
Resnet18	82.7	73
Resnet50	82	73.3

IV. Discussion

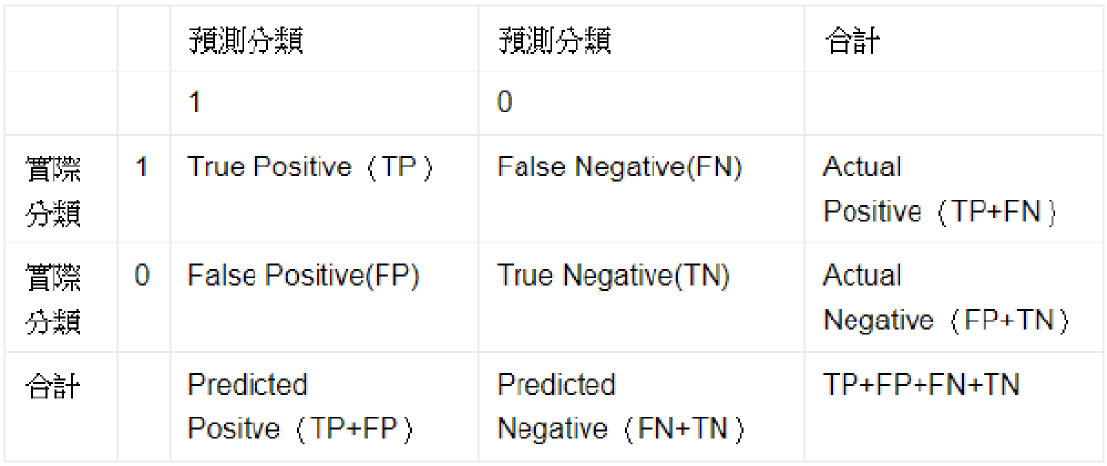
Anything you want to share

A. Pretrained model

Pre-training的優點可以減少訓練時間，因為網路已經預先訓練在相似的資料上，可以加快模型的收斂時間。也可以減少target domain所需資料數量pre-training可以幫助模型認識general的知識，因此可能可以使用較少的target domain資料，訓練出泛化性好的模型。由結果也可以觀察到pretrained model很重要，有沒有pretrained的結果差異相當大，pretrain增加了模型generalization的能力，也算變相增加了training data。

B. 混淆矩陣的四個元素(TP,TN,FP,FN)

TP(True Positive): 正確預測成功的正樣本，例如在一個預測是不是貓的圖像分類器，成功的把一張貓的照片標示成貓，即為TP
TN(True Negative): 正確預測成功的負樣本，以上述例子，成功的把一張狗的照片標示成不是貓，即為TN
FP(False Positive): 錯誤預測成正樣本，實際上為負樣本，例如：錯誤的把一張狗的照片標示成貓
FN(False Negative): 錯誤預測成負樣本(或者說沒能預測出來的正樣本)，例如：錯誤的把一張貓的照片標示成不是貓
以下為二分類的混淆矩陣



		預測分類	預測分類	合計
		1	0	
實際分類	1	True Positive（TP）	False Negative(FN)	Actual Positive（TP+FN）
實際分類	0	False Positive(FP)	True Negative(TN)	Actual Negative（FP+TN）
合計		Predicted Positve（TP+FP）	Predicted Negative（FN+TN）	TP+FP+FN+TN

C.　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）
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— 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

PyTorch不僅可設置對圖片的操作，還以這些進行隨機選擇、組合 不僅可設置對圖片的操作，還以這些進行隨機選擇、組合

a..transforms.RandomChoice(transforms)：從給定的一系列 transforms中選一個 操作， randomly picked from a list

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