1. Introduction (20%)

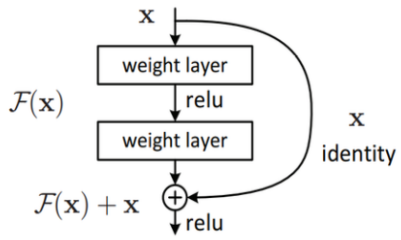
   本次實驗希望利用在 Pytorch 的基礎上建立的 DataLoader 來讀資料， 並分別透過 pretrained ResNet18 、 pretrained ResNet50、non-pretrained ResNet18、non-pretrained ResNet 50 來分類該資料的糖尿病所引發視網膜病變的嚴重程度會落在哪，嚴重程度有五個等級，分別為 0-No DR、1-Mild、2-Moderate、3-Severe、4-Proliferative DR。

   關於遷移學習(transfer learning)、預訓練模型 (pretrained model)、微調(fine tune)

   > **遷移學習**會透過某種問題 A 所對應的大量資料集來將某種神經網路結構的參數都訓練好，並將該模型結構與模型參數，也就是**預訓練模型**，一起放在網路上給其他人透過**微調**修改、重新訓練模型的部分架構或部分參數，來應用於其他不同的特定問題 B,C,D,E,... 上，這樣就不用每個相似的模型應用到不同的問題時都得重新訓練一次，相當省時；另外，通常預訓練模型應用的場景 B,C,D,E,... 需跟原先的目標問題 A 相近，否則應用效果也不會太好。
   >
   > 微調的方式有很多種，包含 1)特徵擷取(feature extraction)，凍結前面幾層的參數只訓練後面特定幾層以取得新的參數，前面幾層的網路就很像特徵提取機，後面幾層再針對提取出來的特徵去做處理，2)採用預訓練模型的架構，將預訓練模型的參數當作初始值，並在這樣的基礎上去對全部或者部分模型繼續做訓練。
   >
   > 預訓練模型可以參考 Pytorch 官網的 torchvision.models 來使用。

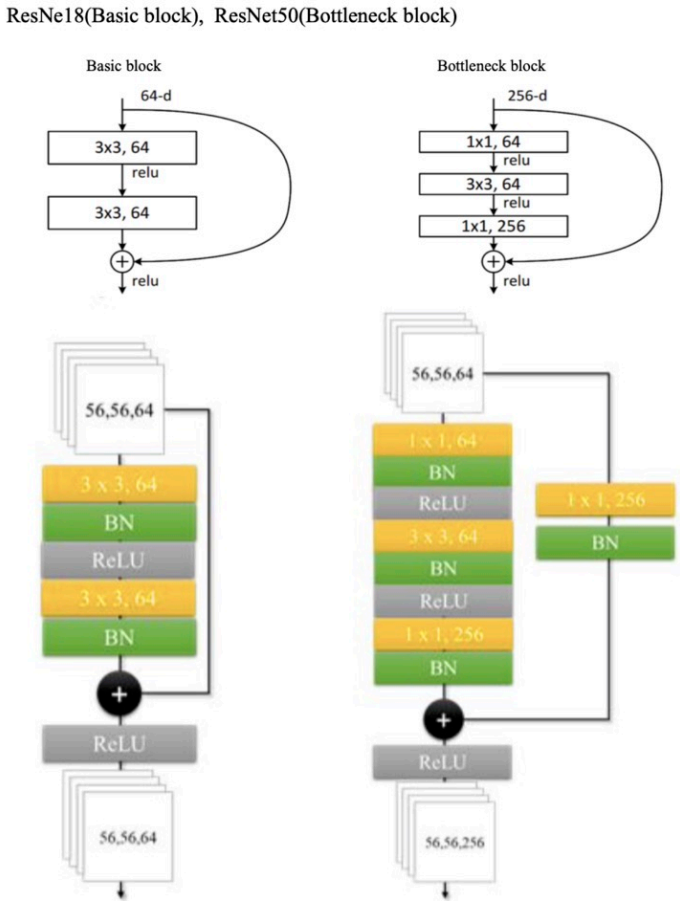
2. Experiment setups (30%)

   A. The details of your model (ResNet)

   > ResNet (殘差學習, residual learning) 概念
   >
   > > 透過 batch normalization 可以有效解決避免了梯度消失/爆炸 (gradient vanishing/exploding) 的問題並讓模型可以成功收斂，但隨著模型層數的增加到合適的一定值之後，訓練得到的準確率也會從上升開始轉變成下降，過多的模型層數導致了訓練錯誤率過高，這跟我們預期越深的網路應該要達到更精確的準確率的目標不符。而**殘差學習**則成功於合理時間內在層數越來越深的網路架構上訓練出越來越好的準確率。
   > >
   > > 假設在某一已成功學出恆等映射(identity mapping)函式 $H(x)$ 的淺層網路模型上做訓練，現在希望在增加網路層數之後，該模型仍然能夠學到恆等映射函式，也就是該模型可以使得輸入 x 經過多個非線性層之後輸出的仍是 x，但實驗結果表明一般的網路模型增加過多層數之後就沒辦法學到這樣的性質。
   > >
   > > 不過對於殘差學習就不一樣了，它的學習目標從直接學恆等映射函式 $H(x)$ 轉變成學習殘差 $F(x) = H(x) - x$，目標是希望能讓殘差 $F(x)$ 變為 0，成功學完之後，再將 x 透過 Shortcut connection(會跳過一層或多層來指向下一層，避免增加額外的參數) 移項回來得到 $H(x) = F(x) + x$，這讓訓練過程變得容易許多，舉例來說，假設輸入 $x = 9$，模型第一層學出 $H_1(x) = 10$，第二層學出 $H_2(x) = 11$，則第一層殘差為 $F_1(x) = 10 - 9 = 1$，第二層殘差為 $F_2(x) = 11 - 9 = 2$，對於一般網路模型 (學習$H(x)$) 來說，誤差變化率為 $H_2(x) - H_1(x) = (11 - 10)/10 = 10\%$，對於殘差網路模型 (學習$F(x)$) 來說，誤差變化率為 $F_2(x) - F_1(x) = (2 - 1)/1 = 100\%$，可以看出學習殘差 $F(x)$ 比起直接學習原來的目標函式 $H(x)$ 更能凸顯微小誤差的變化程度，因此殘差學習才得以在越來越深的網路層數中得到越來越好的準確率。
   > >
   > > 
   > >
   > >
   > > 另外，在訓練淺層殘差網路模型的時候會使用左下圖的 basic (building) block，訓練深層(大於等於50層)殘差網路模型時會利用右下圖的 bottleneck (building) block（圖參考自给妹纸的深度学习教学(4)——同Residual玩耍），如此一來無論是訓練淺層還是深層的殘差網路模型所需計算的參數量就會一樣，不會隨著網路層數增加而增加。其計算過程如下：
   > >
   > > Basic block 所需的參數量為 $64 * (3 * 3 * 64) + 64 * (3 * 3 * 64) = (18 * 64) * 64 = 73728$。

Bottleneck block 所需的參數量為
$64 * (1 * 1 * 64) + 64 * (3 * 3 * 64) + 64 * (1 * 1 * 256) + 64 * (1 * 1 * 256) = (18 * 64) * 64 = 73728$
。

註：$b * (k_1 * k_2 * c)$ means batchsize $*$ (kernel size $*$ number of channels i.e. filters)



torchvision 定義的 ResNet18 架構同樣遵從以上 basic block 的定義。

```
In [14]: resnet18 = models.resnet18(pretrained=False)
                    ↖import torchvision.models as models
In [15]: resnet18
Out[15]:
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
```

torchvision 定義的 ResNet50 架構同樣遵從以上 bottleneck block 的定義。

```
In [6]: resnet50 = models.resnet50(pretrained=False)
                  ↖import torchvision.models as models
In [7]: resnet50
Out[7]:
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

參考資料

> [直觀理解ResNet —簡介、 觀念及實作(Python Keras)]
>
> [Residual Leaning: 認識ResNet與他的冠名後繼者ResNeXt、ResNeSt]
>
> [ResNet学习笔记]
>
> [给妹纸的深度学习教学(4)——同Residual玩耍]

ResNet 模型建設

> 利用 torchvision.models 將 resnet model 引入，並在宣告該 model 的時候定義好針對該 model 所需的 number of layers、epochsize 與 batchsize，而這裡的 pretrained model 並沒有把某些 layer凍結才去做訓練，而是在該 pretrained model 的基礎上，對「整個」網路（不是部分網路）再去做訓練。在本次實驗中，對於 ResNet18 with/without pretrained model 採用 15 個 epochsize、32 個 batchsize 去做訓練，平均 train 一個 epoch 需要花 10 分鐘，test 一個 epoch 需要花 1 分鐘，對於 ResNet50 with/without pretrained model 則採用 15 個 epochsize、8 個 batchsize 去做訓練，平均 train 一個 epoch 需要花 30 分鐘，test 一個 epoch 需要花 3 分鐘。
>
> ```
> class ResNet(nn.Module):
>  def __init__(self,pretrained=False,num_layers=18,epochsize=10,batchsize=4):
>      super(ResNet, self).__init__()
>      #model information
>      self.name = f'ResNet{num_layers}' #model name, for plot
>      self.pretrained_status = 'withPretraining' if pretrained else
> 'withoutPretraining'
>      self.epochsize = epochsize
>      self.batchsize = batchsize
>      self.model = torchvision.models.__dict__[f'resnet{num_layers}'](pretrained)
>      self.model.fc = nn.Linear(in_features=self.model.fc.in_features, out_features=5)
>  def forward(self,x):
>      y = self.model(x)
>      return y
> #ResNet18 with/without pretrained model
> resnet18_pretrained =
> ResNet(pretrained=True,num_layers=18,epochsize=15,batchsize=32).to(device)
> resnet18_curtrained =
> ResNet(pretrained=False,num_layers=18,epochsize=15,batchsize=32).to(device)
> #ResNet50 with/without pretrained model
> ```

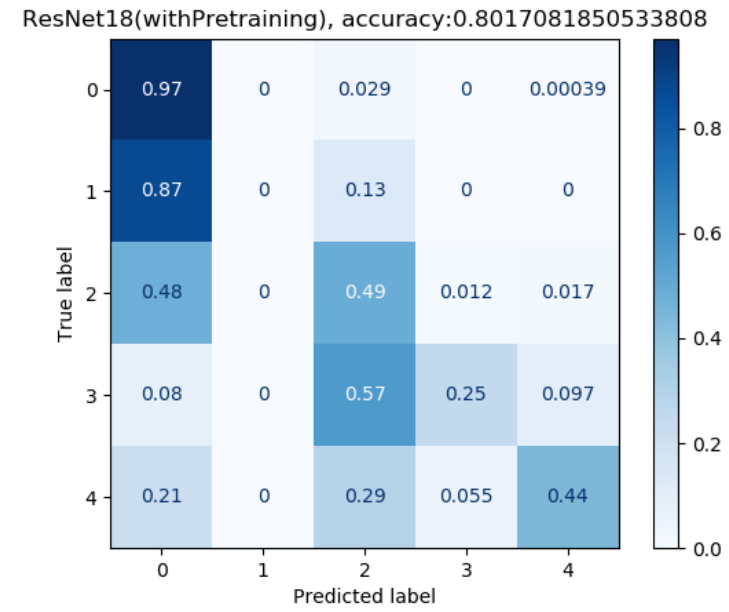
```
resnet50_pretrained =
ResNet(pretrained=True,num_layers=50,epochsize=15,batchsize=8).to(device)
resnet50_curtrained =
ResNet(pretrained=False,num_layers=50,epochsize=15,batchsize=8).to(device)
```

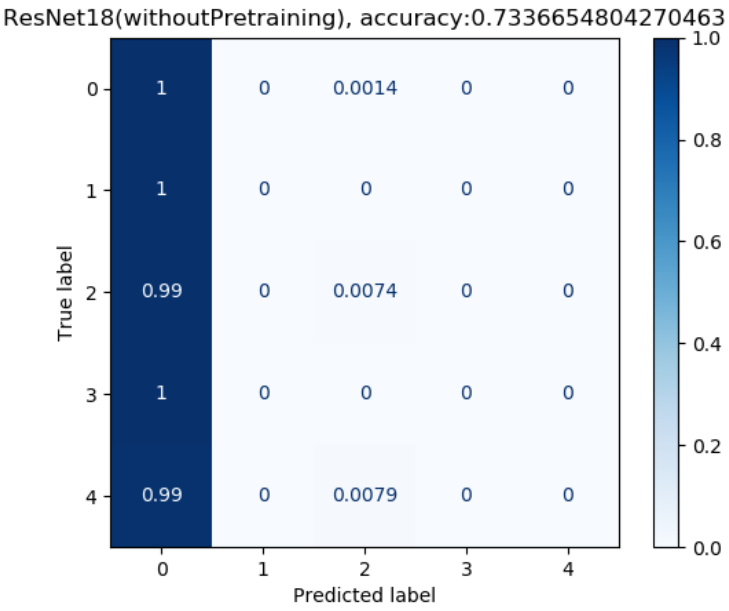
B.The details of your Dataloader

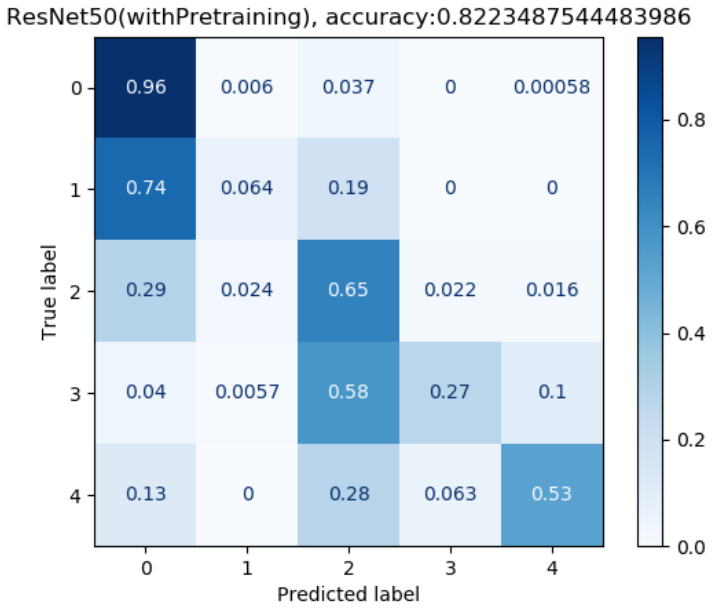
> Dataloader 有更動的地方為 __init__ function 與 __getitem__ function，在 __init__ function 中定義了針對個別圖片該如何處置，包含利用 transforms.ToTensor() 將值標準化至 0~1 之間、利用 transforms.RandomVerticalFlip() 與 transforms.RandomHorizontalFlip() 隨機將圖片翻轉來做 data augmentation，並在 __getitem__ function 實作剛剛定義好的圖片處理方式並回傳處理過的圖片與對應的 label。

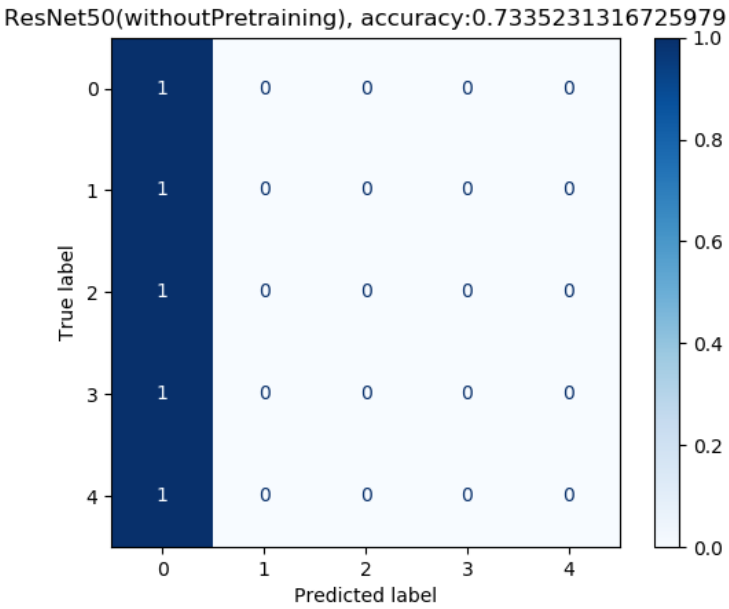
```
import pandas as pd
from torch.utils.data import Dataset
from torchvision import transforms
import numpy as np
import os
from PIL import Image

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)
elif mode == 'test':
  img = pd.read_csv('test_img.csv')
  label = pd.read_csv('test_label.csv')
  return np.squeeze(img.values), np.squeeze(label.values)

class RetinopathyDataset(Dataset):
def __init__(self, root, mode):
  self.root = root
  self.img_name, self.label = getData(mode)
  self.mode = mode
  if mode == 'test':
       self.transforms=transforms.ToTensor() #without data augmentation
  elif mode == 'train':
       self.transforms = transforms.Compose( [transforms.RandomHorizontalFlip(),
transforms.RandomVerticalFlip(), transforms.ToTensor()] ) #with data augmentation
  print("> Found %d images..." % (len(self.img_name)))

def __len__(self):
  return len(self.img_name)

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

C. Describing your evaluation through the confusion matrix


ResNet18(withPretraining), accuracy:0.8017081850533808
0.97 0 0.029 0 0.00039
0.87 0 0.13 0 0
0.48 0 0.49 0.012 0.017
0.08 0 0.57 0.25 0.097
0.21 0 0.29 0.055 0.44
True label
Predicted label
0 1 2 3 4
0.0 0.2 0.4 0.6 0.8


ResNet18(withoutPretraining), accuracy:0.7336654804270463
1 0 0.0014 0 0
1 0 0 0 0
0.99 0 0.0074 0 0
1 0 0 0 0
0.99 0 0.0079 0 0
True label
Predicted label
0 1 2 3 4
0.0 0.2 0.4 0.6 0.8 1.0





由圖可知，這四個 model 都會偏向去預測 label 0 ，尤其 without pretrained model 的模型都會非常集中於預測 label 0 的選項，這是因為我們的 data 是 imbalance (label 0 較多) 的，所以導致就算模型全部猜 label 0 也會有 70% 的準確率，在沒有前人的幫助如 pretrained model 下，不利學習。而對於從 with pretrained model 開始去做 fine-tune 的模型，無論是 ResNet18 還是 ResNet50，在 testing set 得到的準確率都會比較好，預測的 label 分佈也相對比較分散。

3. Experimental results (30%)

A. The highest testing accuracy

- Screenshot

  如圖，最高的準確率為 0.8223。

```
100%|████████████████████████████████████████| 879/879 [02:27<00:00,  5.97it/s]
model:ResNet50 withPretraining, accuracy:0.82235
100%|████████████████████████████████████████| 879/879 [02:27<00:00,  5.95it/s]
model:ResNet50 withoutPretraining, accuracy:0.73352
100%|████████████████████████████████████████| 220/220 [00:54<00:00,  4.00it/s]
model:ResNet18 withPretraining, accuracy:0.80171
100%|████████████████████████████████████████| 220/220 [00:55<00:00,  3.97it/s]
model:ResNet18 withoutPretraining, accuracy:0.73367
                     ResNet18  ResNet50
withPretraining      0.801708  0.822349
withoutPretraining   0.733665  0.733523
```

- Anything you want to present

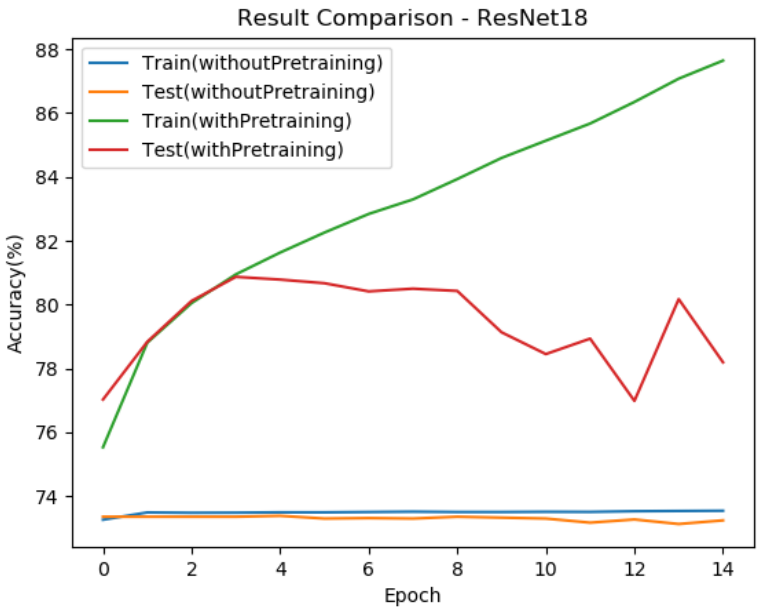
  關於模型訓練的其他參數如下：

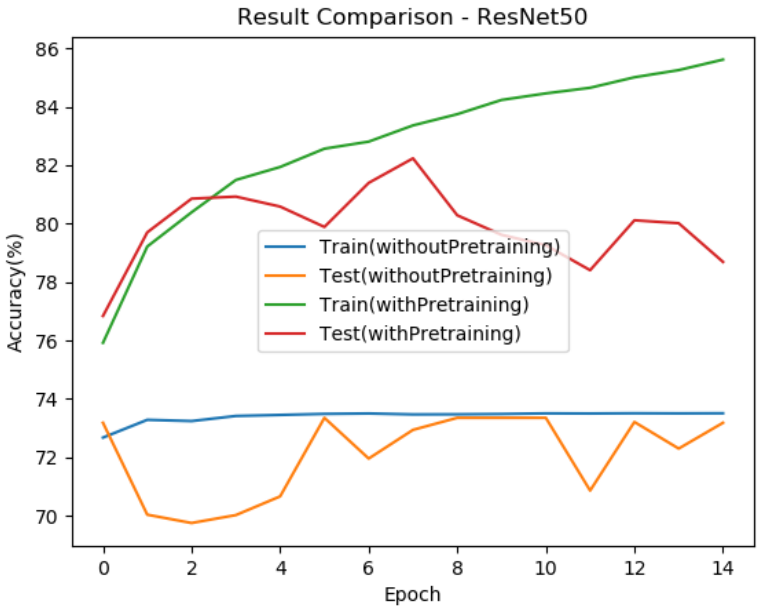
  > learning rate : 1e-3
  >
  > optimizer : SGD
  >
  > optimizer's parameter - momentum : 0.9
  >
  > optimizer's parameter - weight_decay : 5e-4

B. Comparison figures

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

  可以看出在 ResNet 的架構下，深層網路（ResNet50）的確比淺層網路（ResNet18）來得容易取得較高準確率。





4. Discussion (20%)
   A. Anything you want to share

   在做訓練的時候，尤其常在程式終於快跑完了的時候，常常會突然出現這個錯誤訊息而中止。

   > RuntimeError: CUDA out of memory. Tried to allocate 256.00 MiB (GPU 0; 5.94 GiB total capacity; 4.99 GiB already allocated; 85.25 MiB free; 5.22 GiB reserved in total by PyTorch)

   又或者是連錯誤訊息都沒有，整個 process 直接被 killed 掉，這種跑了一整天後卻什麼產出都沒有，或是得在程式裡設置許多 checkpoint 來避免這種情況，非常惱人！這種錯誤訊息是因為快取記憶體不足所致，若整個網路裡需要算 gradient 的參數太多，每個參數當下的 gradient 又沒有適時的清掉並釋出記憶體的話，這些參數就會漸漸地將可用的快取記憶體佔滿，最終整個程式崩潰 (out of memory) 而被強致終止，解決辦法有以下幾種：

   - 縮減 model 的大小，看要減少 layer 數量還是其他，盡可能減少需要計算的參數的數量。
   - 減少 batchsize 的大小，縮小一次計算需要計算的參數量，在我被分配到的機器上，ResNet18 一次最多只能計算 batchsize 為 32 的資料，ResNet50 一次最多則計算 batchsize 為 8 的資料，再多就超出快取記憶體能負荷的範圍了。
   - 利用 with torch.no_grad() 或 detach() 來將 gradient 的資訊屏蔽而不會因此佔用記憶體，常常在 evaluation (testing phase) 的時候我們會不小心把 gradient 的參數也帶進去計算，但在 evaluation 的階段並不需要做 backward，只需要做 forward，根本不需要 gradient 的資訊，所以在這個階段，最好把 gradient 的參數先拿掉再進來做 forward，減輕記憶體負擔。
   - 利用 .float() 或者 .item() 來將帶有 gradient 的參數轉成 scalar，常常在利用 +=、/=、-=、*= 之類的簡寫運算來算 total loss 和 total accuracy 的時候，會不小心把帶有 gradient 的參數一起拿去做加減乘除的計算，導致原本不應該佔用快取記憶體的 total loss 和 total accuracy 也開始越吃越多 GPU 的記憶體，因此就需要將這些帶有 gradient 的參數轉成簡單的 scalar，使之不會使吃到記憶體的空間。
   - 利用 del 來將用不到的記憶體刪掉釋出，並搭配 torch.cuda.empty_cache() 讓這個快取記憶體釋出的資訊可以利用 nvidia-smi 查看，例如 train 完某個 model 要 train 下一個 model 前，就可以透過 del model 把現在這個 model 佔用的記憶體都釋出；又或者要 train 下一個 epoch 前，可以先把計算本次 epoch 的 minibatch 資料 backward 資訊的 loss 刪掉，因為下一個 epoch 就會有新的 loss 需要去計算 gradient，而不會使用到本次 epoch 的 loss。
   - 在使用 pretrained model 的時候，除了把 model 前面幾層 layers 的 requires_grad 設成 False 來凍結前面幾層 layer，還可以只餵那些沒有被凍結的參數給 optimizer 做更新，減少需要計算 gradient 的參數量。

   另外，檢查這種 GPU memory 佔用的方法，是要在 command line 輸入 nvdia-smi 的指令來看 GPU memory usage 那欄在不同情況下是否有異常上升，所謂異常是指下列情況：

   - 做 eval 或 test 的時候，GPU memory 的用量不應該上升，因為用不到神經網路。

- 做 train 的下一個 epoch 時，GPU memory 的用量也不應該上升，因為是同一個神經網路同一個 model。

- 用下一個 model 做 train 的之前, GPU memory 的用量應該要清空成初始值而不應該上升。