312553040_郭晉維_Lab2_Report

1. Introduction

電腦視覺中的圖片分類是指利用電腦技術，將輸入的圖像分類為不同的類別。這包括使用機器學習和深度學習技術，通過訓練模型來識別圖像中的特徵，然後將其歸類到預定的分類中。例如，將圖像分為動物、車輛、食物等不同的類別。這個過程涉及圖像特徵提取、模型訓練和測試驗證等步驟，通常使用的算法包括卷積神經網絡。圖片分類在圖像識別、自動化監控、醫學影像分析等領域有廣泛的應用。這次作業要在使用 PyTorch Framework 來實現兩個十分知名的圖片分類網路，分別是 VGGNet 以及 ResNet50。

VGG (Visual Geometry Group Network)是 ILSVRC 2014 年的第二名，是當時第一個超過 10 層的神經網路，VGG 的第一個特色是只使用 3x3 的 Conv 和 2x2 的 Maxpool，目的是用多層的較小的 filter 達到一個大的 filter 包含的資訊，例如 2 個 3x3 的 Conv 可以和 1 個 5x5 的 Conv 涵蓋一樣的資訊量，但參數量卻比較少(3*3*2 < 5*5*1)。VGG 的第二個特色是做了很多次 Conv 才接 Pooling，跟以前一個 Conv 後面就接一個 Pooling 的方法不同，因為這樣可以透過 activation function 使 data 有更多 non-linear 的變化。

ResNet (Residual Neural Network)是 ILSVRC 2015 年的第一名，主要是為了解決深度神經網絡中的梯度消失和梯度爆炸問題而設計，也讓他可以達到超深的 152 層，ResNet 通過引入殘差塊（Residual Blocks）來解決這個問題。每個殘差塊都包含一個捷徑連接（Shortcut Connection），也就是一個將輸入直接連接到輸出的路徑，也就是兩三層卷積後就會跟原始的輸入相加，從而優化整體的效果。這樣即使網絡的其他部分在訓練過程中出現梯度消失或梯度爆炸，捷徑連接也能保證梯度能夠順利地反向傳播。

這次的作業讓我更清楚瞭解了深度學習在圖像分類上的應用，並用 PyTorch 實際重現了如 VGGNet, ResNet 等大型神經網路的架構，讓我獲益良多。

2. Implementation Details

A. The details of model (VGG19, ResNet50)

VGG19 我主要參考了 https://ithelp.ithome.com.tw/articles/10332866 網站，其中前 16 層 layers 中每一個 layer 都是由 convolution, batch normalization, activate function 組成，在 PyTorch 的寫法如下：

nn.Conv2d(*channel_in*, *channel_out*, kernel_size=3, stride=1, padding=1**)**

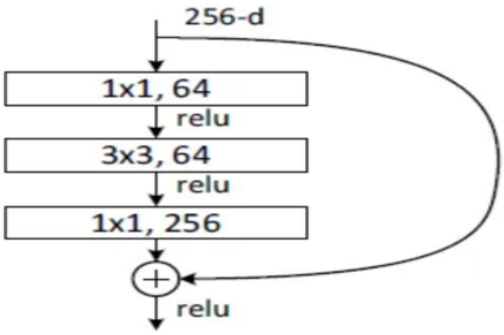
nn.BatchNorm2d(*channel_out*)

nn.ReLU( )

其中在 layer 2, 4, 8, 12, 16 後會進行 maxpool 來將圖片尺寸減半，最後在 layer 16 後的 output.shape 為(512, 7, 7)，然後 layer 17 是 (512, 7, 7) to 4096 的 full connection layer，layer 18 是 4096 => 4096 的 fully connected layer，layer 19 是 4096 => 100 的 fully connected layer。我完整的 implementation 如下圖所示

```
class VGG(nn.Module):
    def __init__(self):
        super(VGG, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1), # layer_1
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1,padding=1), # layer_2
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1), # layer_3
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1), # layer_4
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1), # layer_5
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1), # layer_6
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1), # layer_7
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1), # layer_8
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1), # layer_9
            nn.BatchNorm2d(512),
            nn.ReLU(),
        nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1), # layer_11
        nn.BatchNorm2d(512),
        nn.ReLU(),
        nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1), # layer_12
        nn.BatchNorm2d(512),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1), # layer_13
        nn.BatchNorm2d(512),
        nn.ReLU(),
        nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1), # layer_14
        nn.BatchNorm2d(512),
        nn.ReLU(),
        nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1), # layer_15
        nn.BatchNorm2d(512),
        nn.ReLU(),
        nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1), # layer_16
        nn.BatchNorm2d(512),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
    )
    self.avgpool = nn.AdaptiveAvgPool2d((7, 7))
    self.classifier = nn.Sequential(
        nn.Dropout(0.5),
        nn.Linear(512 * 7 * 7, 4096), # layer_17
        nn.ReLU(),
        nn.Dropout(0.5),
        nn.Linear(4096, 4096), # layer_18
        nn.ReLU(),
        nn.Dropout(0.5),
        nn.Linear(4096, 100), # layer_19
    )
def forward(self, x):
    x = self.features(x)
    x = self.avgpool(x)
    x = x.view(x.size(0), -1)
    x = self.classifier(x)
    return x
```

ResNet50 的部分我主要參考了 https://zhuanlan.zhihu.com/p/353235794 網站，針對右圖的殘差網路架構(residual bottleneck block)分別設計了 BTNK1 跟 BTNK2 兩個較小的網路，分別處理 input_channel 跟 output_channel 相同及 input_channel 跟 output_channel 不同的兩種情況，下面的兩張圖是我對於 BTNK1 跟 BTNK2 的 implementation：




```python
class BTNK1(nn.Module):
    def __init__(self, input_size, output_size, s):
        super(BTNK1, self).__init__()
        self.feature = nn.Sequential(
            nn.Conv2d(input_size, int(output_size/4), kernel_size=1, stride=s),
            nn.BatchNorm2d(int(output_size/4)),
            nn.ReLU(),
            nn.Conv2d(int(output_size/4), int(output_size/4), kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(int(output_size/4)),
            nn.ReLU(),
            nn.Conv2d(int(output_size/4), output_size, kernel_size=1, stride=1),
            nn.BatchNorm2d(output_size),
        )
        self.shortcut = nn.Sequential(
            nn.Conv2d(input_size, output_size, kernel_size=1, stride=s),
            nn.BatchNorm2d(output_size),
        )
        self.relu = nn.ReLU()

    def forward(self, x) :
        fx = self.feature(x)
        x = self.shortcut(x)
        x = fx + x
        x = self.relu(x)
        return x
```

```python
class BTNK2(nn.Module):
    def __init__(self, size):
        super(BTNK2, self).__init__()
        self.feature = nn.Sequential(
            nn.Conv2d(size, int(size/4), kernel_size=1, stride=1),
            nn.BatchNorm2d(int(size/4)),
            nn.ReLU(),
            nn.Conv2d(int(size/4), int(size/4), kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(int(size/4)),
            nn.ReLU(),
            nn.Conv2d(int(size/4), size, kernel_size=1, stride=1),
            nn.BatchNorm2d(size),
        )
        self.shortcut = nn.Sequential(
            nn.Conv2d(size, size, kernel_size=1, stride=1),
            nn.BatchNorm2d(size),
        )
        self.relu = nn.ReLU()

    def forward(self, x) :
        fx = self.feature(x)
        x = self.shortcut(x)
        x = fx + x
        x = self.relu(x)
        return x
```

在 ResNet50 中我主要分成 stage 0 ~ stage 4 跟 classifier 六個部分，首先 stage 0 是將原本的輸入圖片 (3, 224, 224) 經過 convolution 跟 maxpool 後變成 (64, 56, 56)的 features，接著 stage 1 ~ stage 4 就是利用 BTNK1 跟 BTNK2 不斷將 features 送入 residual block，增加 channel 數量並減少圖片尺寸，流程大致如下：(64, 56, 56) => stage 1 => (256, 56, 56) => stage 2 => (512, 28, 28) => stage 3 => (1024, 14,14) => stage 4 => (2048,7,7)，接著進行 average pooling 讓 features 從 (2048,7,7) => (2048,2,2)，最後的 classifier 為一個 8192 => 100 的 fully connected layer，即可得到輸入圖片對應 100 種 classes 的機率分佈。我對於 ResNet50 完整的 implementation 如下圖所示

```python
class ResNet(nn.Module):
    def __init__(self):
        super(ResNet, self).__init__()
        self.stage0 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
        )
        self.stage1 = nn.Sequential(
            BTNK1(64, 256, 1),
            BTNK2(256),
            BTNK2(256),
        )
        self.stage2 = nn.Sequential(
            BTNK1(256, 512, 2),
            BTNK2(512),
            BTNK2(512),
            BTNK2(512),
        )
        self.stage3 = nn.Sequential(
            BTNK1(512, 1024, 2),
            BTNK2(1024),
            BTNK2(1024),
            BTNK2(1024),
            BTNK2(1024),
            BTNK2(1024),
        )
        self.stage4 = nn.Sequential(
            BTNK1(1024, 2048, 2),
            BTNK2(2048),
            BTNK2(2048),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((2, 2))
        self.classifier = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(2048 * 2 * 2, 100),
        )

    def forward(self, x):
        x = self.stage0(x) # layer_1
        x = self.stage1(x) # layer_2 - layer_10
        x = self.stage2(x) # layer_11 - layer_22
        x = self.stage3(x) # layer_23 - layer_40
        x = self.stage4(x) # layer_41 - layer_49
        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x) # layer_50
        return x
```

B. The details of Dataloader

在 Dataloader 的部分我分成 step 1 ~ step 4，首先 step 1 是從 dataset 中讀取圖片，step 2 是得到圖片相對應的 label，step 3 是先對圖片進行 data preprocessing 之後將圖片先轉成 np.array 的格式，再將 np.array 轉成 tensor 的格式以便 PyTorch 使用，最後 step 4 則是 return　img, label。

```
# step 1.
img = Image.open(str(self.root) + str(self.img_name[index]))

# step 2.
label = self.label[index]

# step3. random flipping & random rotation
flipping = random.random()
if(flipping < 0.5) :
    if(flipping < 0.25) :
        img = img.transpose(Image.Transpose.FLIP_LEFT_RIGHT)
    else :
        img = img.transpose(Image.Transpose.FLIP_TOP_BOTTOM)
rotation = random.random()
if(rotation < 0.45) :
    if(rotation < 0.15) :
        img = img.transpose(Image.Transpose.ROTATE_90)
    elif(rotation < 0.3) :
        img = img.transpose(Image.Transpose.ROTATE_180)
    else :
        img = img.transpose(Image.Transpose.ROTATE_270)

# step 3. jpg -> np.array -> tensor
img = np.array(img)
img = img / 255.0
img = np.transpose(img, (2, 0, 1))
img = torch.Tensor(img)

# step 4.
return img, label
```

3. Data Preprocessing

在 data preprocessing 的部分我讓圖片有 50% 的機率進行隨機的水平或垂直翻轉，然後讓圖片有 45% 的機率進行逆時針旋轉 90, 180, 270 度，我認為這樣可以讓網路更好的去學習到位於圖片中不同區塊的特徵，也能避免 training data 的拍攝視角太固定的問題，實驗結果顯示不論是在 VGG19 還是 ResNet50 的架構下，有進行 data preprocessing 所訓練出來的 neural network，在 test data 的 accuracy 都可以較沒有進行 data preprocessing 的 neural network 來得出色，詳細的實驗結果我會放在 **5. Discussion** 的部分。

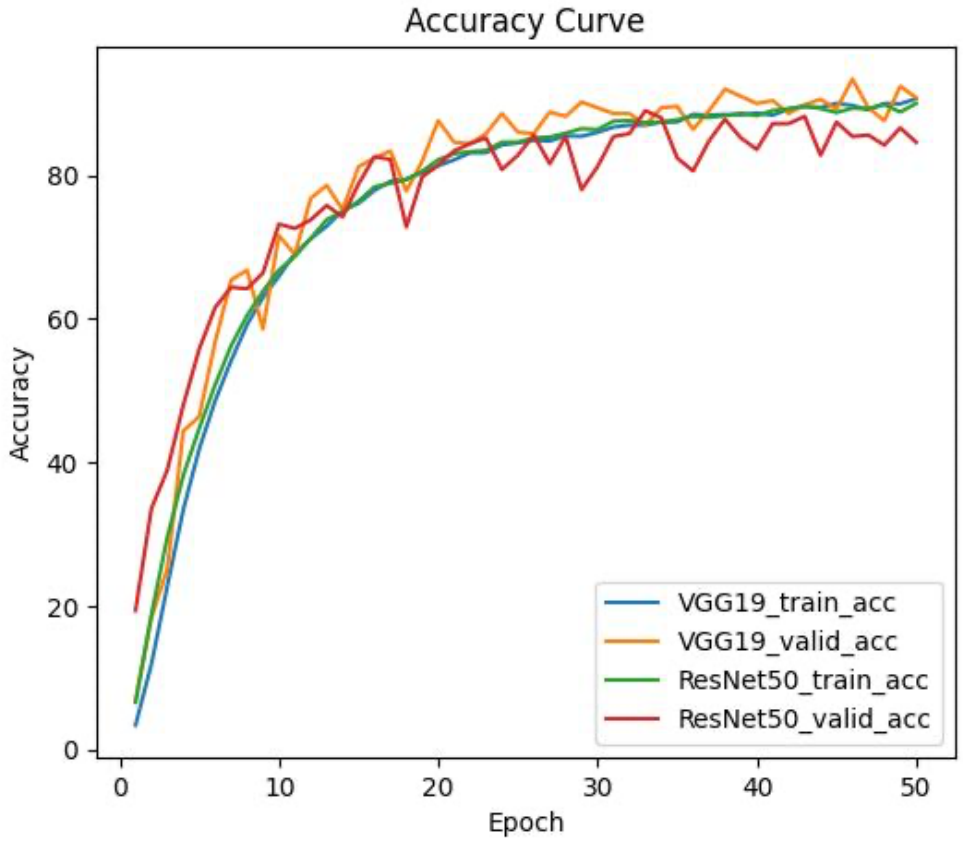
4. Experimental results

在實驗中我設定的參數分別為 Learning rate = 0.001,　Epoch = 50, Loss function = Cross Entropy,　Optimizer = SGD。

A. The highest testing accuracy

```
(base) ibm@ibm:/media/ibm/F/mark/DLP/lab2$ python demo.py
> Found 500 images...
VGG19 Test Data Accuracy :  94.6 %
ResNet50 Test Data Accuracy :  91.8 %
```

B. Comparison figures

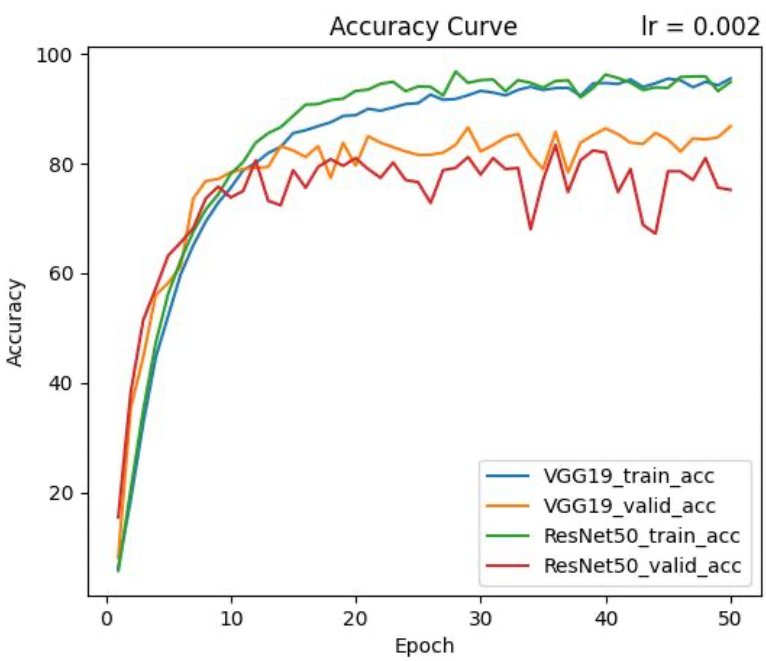


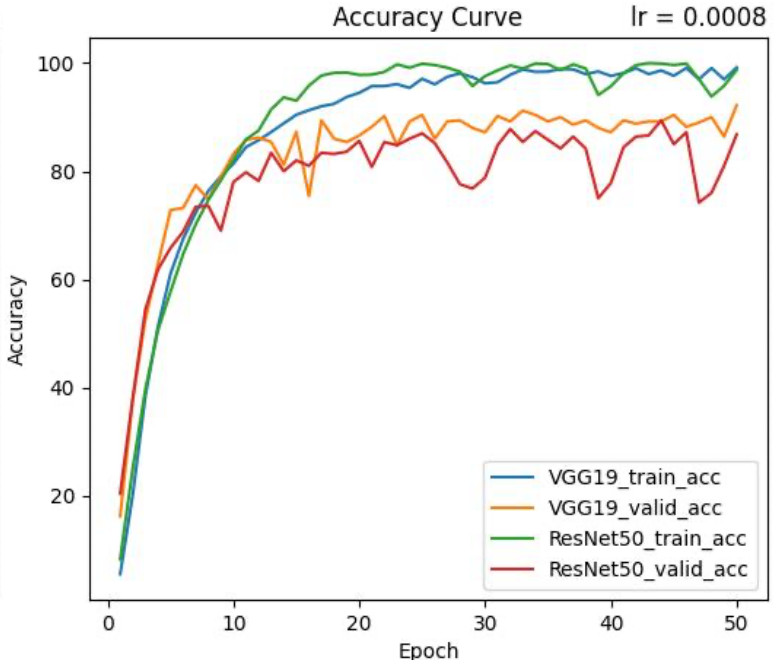
5. Discussion

根據 spec 上的 accuracy curve，Resnet50 的表現應該會比 VGG19 來得好，但我的結果卻是 VGG19 會稍微優於 Resnet50，我想可能是因為我的 VGG19 網路在最後的 fully connected layer 加入了三次 drop out 的步驟，使得出現 overfitting 的情況較低，泛化能力也較好，因此表現得比 Resnet50 好。除了原本的參數設定外，我還針對 Learning rate, Optimizer, without data preprocessing 三種情況進行了實驗，以下將分別進行討論及分析。

A. Different Learning rate

除了原本的 Learning rate = 0.001 之外，我還額外試了 Learning rate = 0.002, 0.0008 兩種不同的參數，accuracy curve 跟比較表格如下頁所示，可以看到兩種情況都有 overfitting 的情況，而 Learning rate = 0.001 時的表現會是最好的。





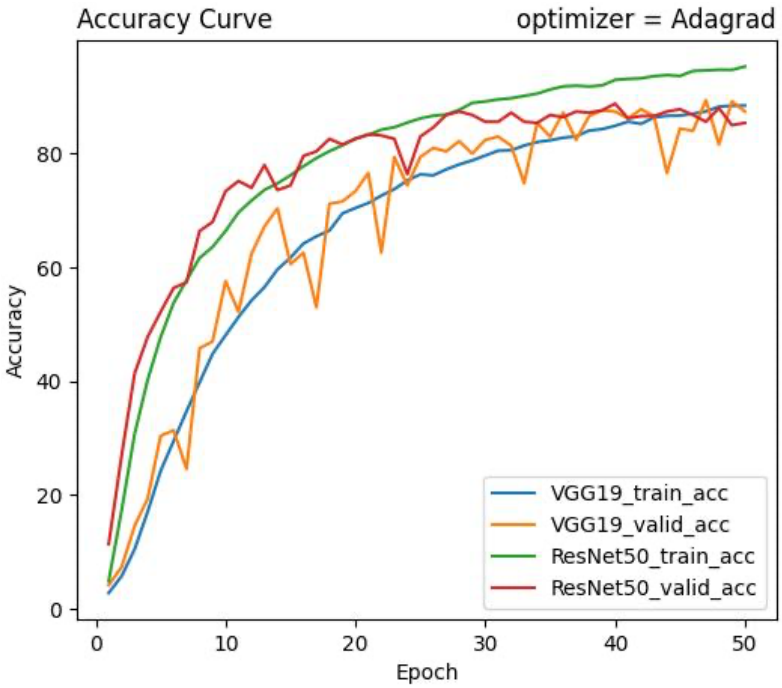
Best Test Accuracy

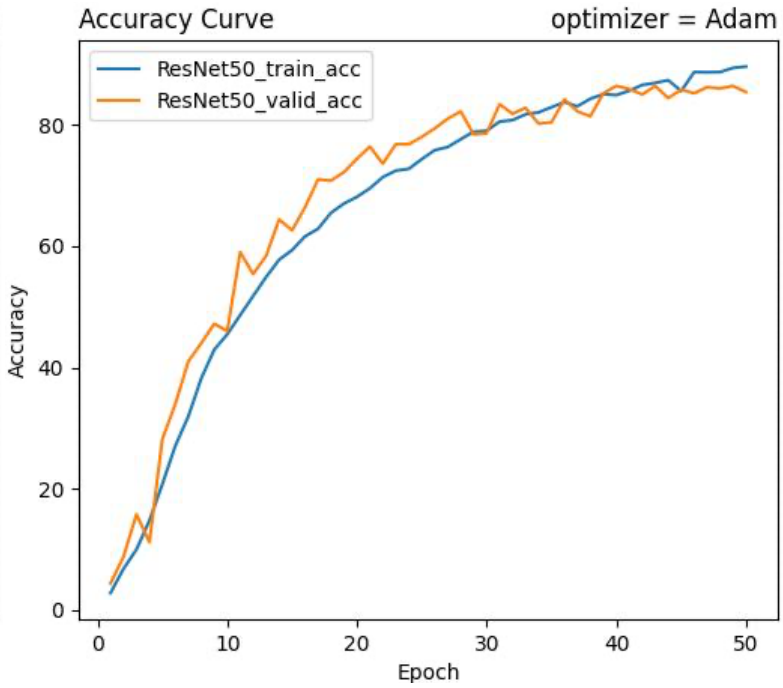
Learning rate	VGG19	ResNet50
0.002	88.6 %	85.0 %
0.001	**94.6 %**	**91.8 %**
0.0008	92.6 %	89.4 %

B. Different Optimizer

除了原本的 SGD optimizer 之外，我還額外試了 Adagrad 跟 Adam 兩種不同的 optimizer，SGD 又稱隨機梯度下降法，可以被視為一種近似梯度下降的方法，通常會更快達到收斂，因為只需考慮一個 example，只對一個 example 的 loss 做計算。Adagrad 可以獨立地適應所有模型參數的學習率，當 gradient 越大，參數 update 越多，當 gradient 越大，參數 update 越少。Adam 結合了 Adagrad、RMSprop 及 Momentum 的優點，對所有不同的參數都有新舊之間的權衡調節，各種狀況均適用，是目前較常使用的優化方式。

我在使用 Adam optimizer 時，發現在 VGG19 網路上無法成功 train 起來，不論是 train_accc 還是 valid_acc 始終都只有 1%左右，我認為可能是因為 VGG19 容易發生梯度消失的情況，而 ResNet50 就沒有這種情況，可以順利進行 train 的過程。Accuracy curve 跟比較表格如下頁所示，可以看到使用 Adam optimizer 時，ResNet50 的 train_acc 較 VGG19 好，但 valid_acc 卻差不多，test_acc 也還是 VGG19 較佳，Adam optimizer 雖然可以套用在 ResNet50，但結果也比其他兩種 optimize 差一些，最好的還是 SGD optimizer。





Best Test Accuracy

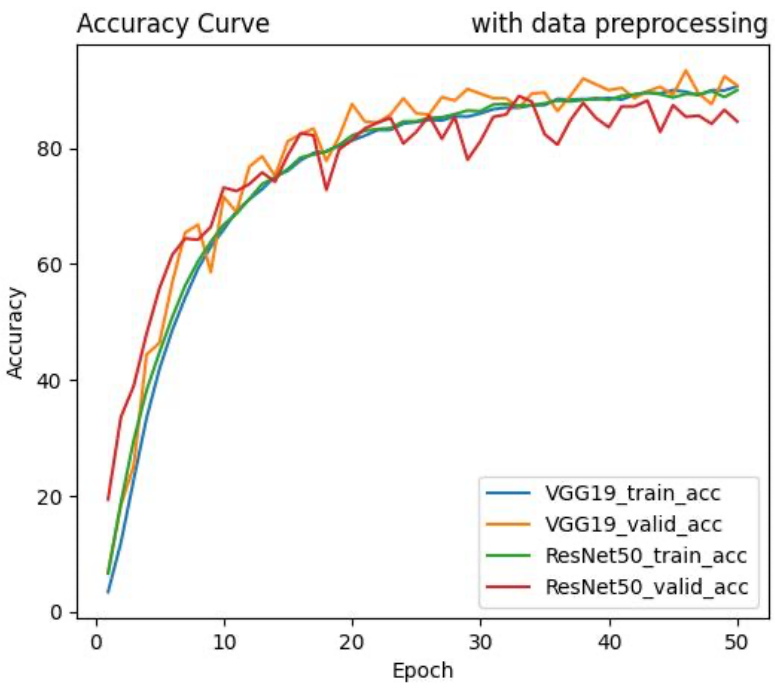
Optimizer	VGG19	ResNet50
SGD	**94.6 %**	**91.8 %**
Adagrad	90.8 %	88.2 %
Adam	X	86.0 %

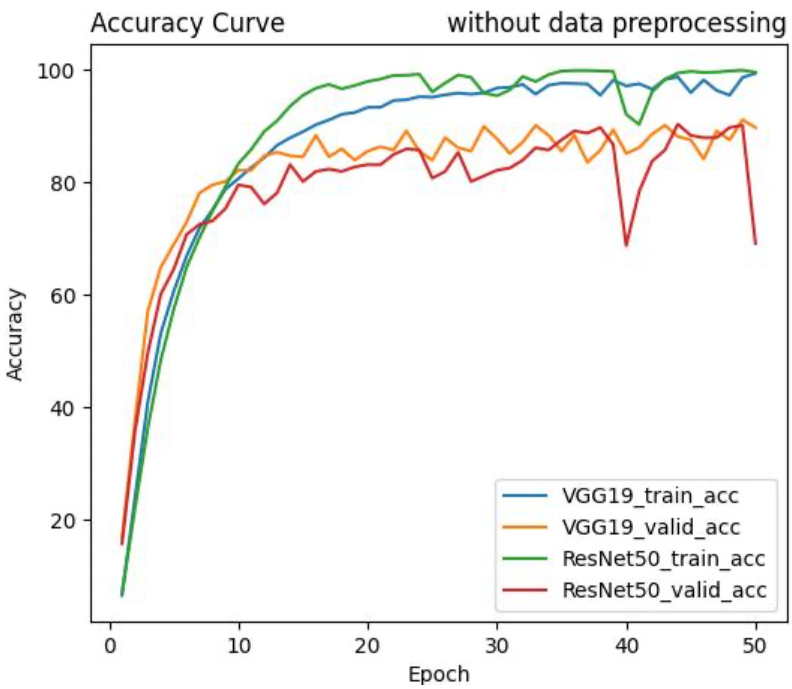
C. With or without Data Preprocessing

我嘗試了不將圖片進行隨機的翻轉或旋轉就直接進行訓練，比較表格與 Accuracy curve 如下所示，如同 **3. Data Preprocessing** 所說的，若沒有進行 data preprocessing，較容易導致 overfitting 的結果，而經過 data preprocessing 過後，可以讓訓練的網路有較好的泛化性，在 test_acc 的表現也比較優異。

Best Test Accuracy

	VGG19	ResNet50
with data preprocessing	**94.6 %**	**91.8 %**
without data preprocessing	92.2 %	89.2 %


Accuracy Curve
with data preprocessing
Accuracy
Epoch
VGG19_train_acc
VGG19_valid_acc
ResNet50_train_acc
ResNet50_valid_acc


Accuracy Curve
without data preprocessing
Accuracy
Epoch
VGG19_train_acc
VGG19_valid_acc
ResNet50_train_acc
ResNet50_valid_acc