Lab3: Leukemia classification

Student id / name: A113599 / 楊淨富

I. Introduction:

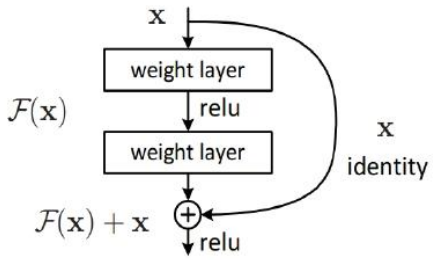
訓練 3 個 model，分別為 ResNet18, ResNet50, ResNet152 來判斷給定的圖片是否為急性淋巴性白血病。此外，該 lab 禁止使用 pre-trained model，即必須自行 train 出參數，並且需上傳對於 test data(unlabeled)的預測結果至 Kaggle 上進行競賽。

II. Implementation Details:

A. The detail of your model

- ResNet

- ResNet can avoid vanishing gradient problem



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

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

在標準的神經網路中，每一層對輸入數據進行轉換，並將一層的輸出作為下一層的輸入。在反向傳播過程中，計算並沿著這些層傳播梯度。然而，在非常深的網路中，隨著梯度通過每一層，它們可能呈指數級遞減，使得初始層難以學習有意義的表示。這個問題稱為「梯度消失」，在具有許多層的網路中特別突出， 這會使得神經網路難以有效地更新權重，導致訓練緩慢或停滯。

ResNet 是透過 residual connections(or skip connection)來避免梯度消失的問題。它並非在一層中學習從輸入到輸出的直接映射(identity mapping)，ResNet 學習殘差映射(residual mapping)—即輸入和輸出之間的差異。這個過程可以表示為：

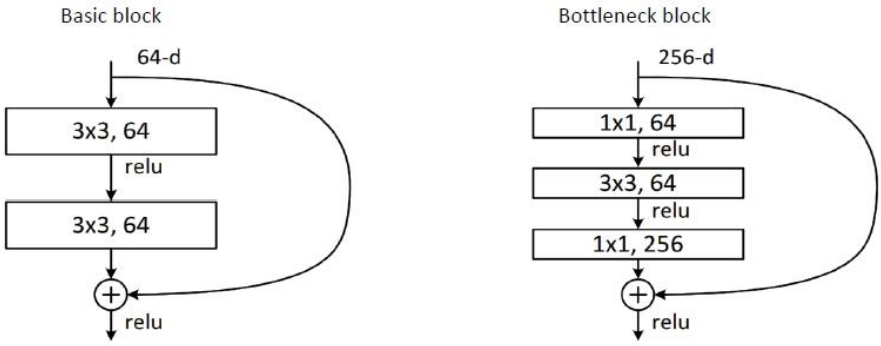
輸出 = 輸入 + F(輸入)

這種 residual mapping 使得 gradient 在 back propagation 更容易流動。此外，如果 F(x) = x(恆等函數)，則 gradient 可以直接通過 skip connection 流動，輸入的梯度，即∂輸出/∂輸入是固定的，不受層數增加而改變，輸出的梯度不會消失（變得過小）或爆炸（變得過大），而是保持恆定值，有效避免了梯度消失。

若 F(x)非恆等函數，梯度仍然可以通過跳躍連接流動，有助於學習過程。這些跳躍連接的存在確保梯度在網路中具有更短的傳播路徑，在訓練過程中有助於保持資訊的良好流動，減輕了梯度消失問題，並成功地實現了非常深的網路的訓練。

因此，ResNet 架構可以更深（例如 ResNet-101、ResNet-152 等）而不會嚴重降低性能，並且已被廣泛應用於各種電腦視覺任務，例如：圖像分類、目標檢測等任務中。

- ResNe18 (Basic block), ResNet50 & ResNet152 (Bottleneck block)



- Basic block 應用在 ResNet18, ResNet34
  - 由兩層 CNN 為主組成，kernel_size = (3, 3)，並且在第一個 CNN 搭配 ReLU，並且會將 residual 和最後的結果相加並再次經過 ReLU。每次 ReLU 前都可以先 batch normalization。
- Bottleneck block 應用在 ResNet50, ResNet101, ResNet152
  - 由三層 CNN，kernel_size 分別為(1, 1), (3, 3)和(1, 1)為主所組成，主要目的是：

1. 降低維度: 使用 1x1 卷積層可以減少 input 的 channel。因為 1x1 卷積在深度(通道)維度上進行計算，相當於通道間的線性組合。
2. 減少計算量: 相比於直接使用 3x3 卷積，Bottleneck block 中的 1x1 卷積可以大幅度地減少計算量。1x1 卷積在通道數上的計算成本要遠小於 3x3 卷積，因為它涉及的參數數量更少。

3. 增加非線性：通過在 1x1 卷積和 3x3 卷積之間添加 ReLU，Bottleneck block 可以增加非線性，有助於網路學習更複雜的特徵表示。

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
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

接下來解釋 ResNet18, ResNet50, ResNet152 的配置: (參考原 paper，如上圖)

- ResNet18: 5 層 layer
  - 第一層(conv1): kernel_size=(7, 7) stride=2，之後可再搭配 batch normalization 和 ReLU。
  - 第二層(con2_x): 先透過 max pooling，再使用兩個 Residual block。
  - 第三至第五層(conv3_x, conv4_x, conv5_x): 和第二層最大的差別為，basic block 的 stride 改為 2，為了減少資料量，且為了使輸入和輸出的尺寸一致，所以需要加上 down sample 的動作。此外，不使用 max pooling。

此外，ResNet18 有一些 Hyper parameters，以下是我 tune 的參數:

- Batch size = 64
- Learning rate = 0.0001
- Epochs = 201
- Optimizer: torch.optim.SGD()
- Loss function: torch.nn.CrossEntropyLoss()

- ResNet50/ResNet152: 5 層 layer
  - 第一層(conv1): 和 ResNet18 相同。
  - 第二層(conv2_x): 先透過 max pooling，再使用了三個 bottleneck block。
  - 第三至第五層(conv3_x, conv4_x, conv5_x): 差別在於使用的 bottleneck 數，以及第二個 CNN 所使用的 stride 由 1 改為 2。此外，不使用 max pooling。
  - ResNet50 和 ResNet152 僅差在第二至第四層的 bottleneck 數，ResNet50 為[3, 4, 6, 3]，ResNet152 為[3, 8, 36, 3]。

ResNet50 的 Hyper parameters:

- Batch size = 32
- Learning rate = 0.0001
- Epochs = 201
- Optimizer: torch.optim.SGD()
- Loss function: torch.nn.CrossEntropyLoss()

ResNet152 的 Hyper parameters:

- Batch size = 32
- Learning rate = 0.0001
- Epochs = 31
- Optimizer: torch.optim.SGD()
- Loss function: torch.nn.CrossEntropyLoss()

B. The details of your Dataloader

```python
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.mode = mode

        if mode == 'train' or mode == 'valid':
            self.img_name, self.label = getData(mode)
        else:  # mode == 'test'
            self.img_name = getData(mode)
            self.label = None  # No labels for test data

        # 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):

        """
           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
        """

        img_path = os.path.join(self.root, self.img_name[index]) # + '.jpeg'
        img = Image.open(img_path)

        if self.mode == 'train':
            transform=transforms.Compose([
                # transforms.CenterCrop(450), # crops the center region of the image with a square size of height
                # transforms.Resize(150), # (h, w) 512x512 pixels
                transforms.RandomHorizontalFlip(), #  randomly flips the image horizontally with a 50% chance
                transforms.RandomRotation(degrees=15), # randomly rotates the image by a maximum of 15 degree
                transforms.ToTensor(),
                transforms.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
                # Add an additional transformation to rescale pixel values to [0, 1]
                # transforms.Lambda(lambda x: (x + 1.0) / 2.0)
            ])
        elif self.mode == 'valid':
            transform=transforms.Compose([
                # transforms.CenterCrop(450),
                # transforms.Resize(150),
                transforms.ToTensor(),
                transforms.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
                # Add an additional transformation to rescale pixel values to [0, 1]
                # transforms.Lambda(lambda x: (x + 1.0) / 2.0)
            ])
        else:
            transform = transforms.Compose([
                # transforms.CenterCrop(450),
                transforms.ToTensor(),
                transforms.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
            ])

        img = transform(img)

        # print(img[0].shape)

        if self.label is not None:
            label = self.label[index]
            return img, label
        else:
            return img
```

- __init__(self, root, mode): 初始化，root(string)為 dataset 所在的路徑，mode 則根據對應的 mode(train/ valid/ test)來取得 csv 檔中的 image path 跟 label。
- __len__(self): 回傳 dataset 的 size
- __getitem()__(self, index): 根據 index 去取得 image 的 path，並 import PIL 並使用它的 Image.open()來開啟圖片，再使用 transforms.Compose() 這個 function 來將圖片轉換成 tensor，轉換時可以搭配 transforms.CenterCrop(),transforms.Resize(),transforms.RandomRotation() transforms.RandomHorizontalFlip(),等等 function 來對圖片進行處理。最後再根據 mode 傳回 img(以及 label)的 tensor。

C. Describing your evaluation through the confusion matrix

混淆矩陣（Confusion Matrix）是在機器學習和統計學中用來評估分類模型的性能的一種方法。它是一個矩陣，用於展示模型在測試集上對不同類別的預測結果。

混淆矩陣的行代表實際的類別，列代表預測的類別。對於一個二分類問題，混淆矩陣如下

	實際正例	實際反例
預測正例	TP	FP
預測反例	FN	TN

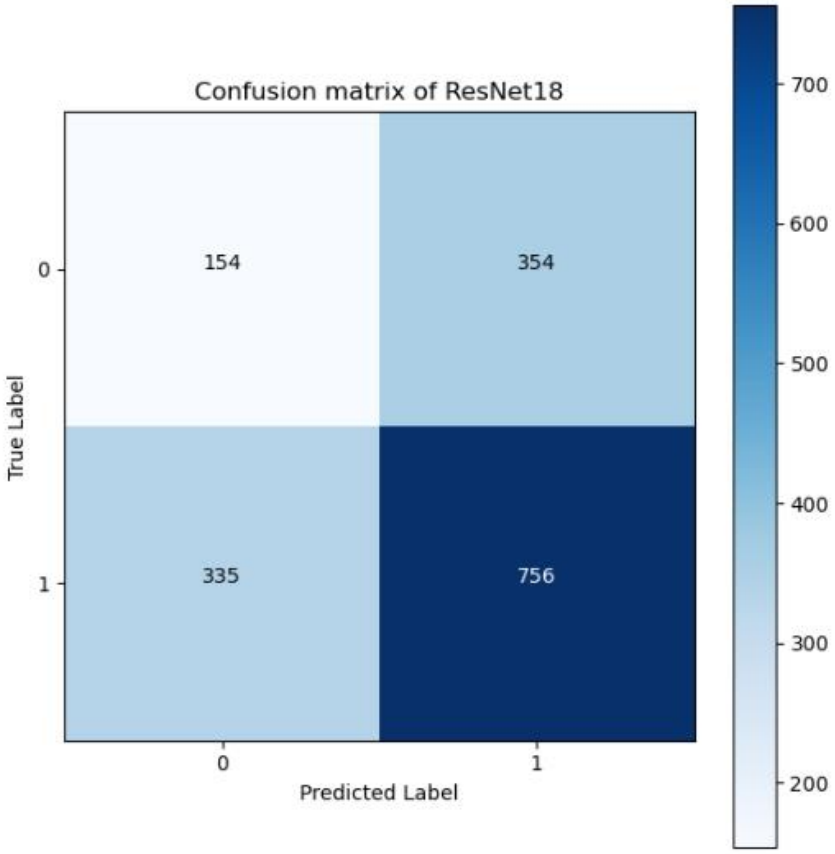
- TP (True Positive)：真正例，表示實際為正例且被正確預測為正例的數量。
- TN (True Negative)：真反例，表示實際為反例且被正確預測為反例的數量。
- FP (False Positive)：假正例，表示實際為反例但被錯誤預測為正例的數量。
- FN (False Negative)：假反例，表示實際為正例但被錯誤預測為反例的數量。

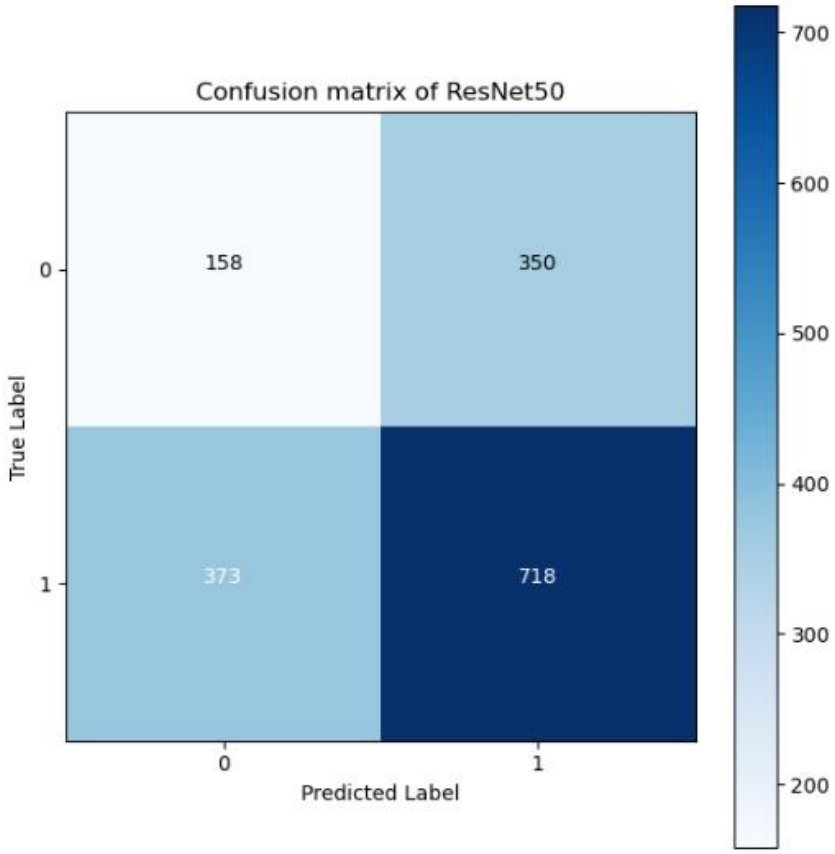
混淆矩陣也可以用於多分類問題，其中類別數目超過兩個。在這種情況下，混淆矩陣的大小會隨著類別數目的增加而增加，但基本原理保持不變。

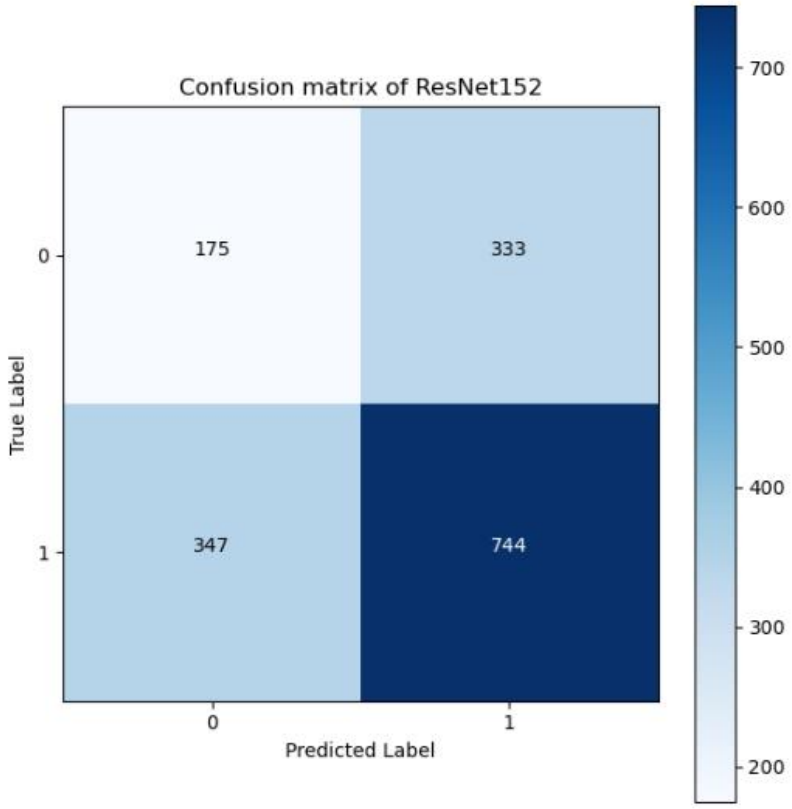
通過分析混淆矩陣，我們可以獲得模型在不同類別上的表現情況，找出模型在哪些類別上出現了錯誤，並進一步優化模型的設計。混淆矩陣是評估和理解分類模型性能的重要工具之一。

```python
def plot_confusion_matrix(true_label, predicted_label, classes, normalize = False, title = None, cmap = plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    # Compute confusion matrix
    cm = confusion_matrix(true_label, predicted_label)

    # Calculate class-wise accuracy if normalization is enabled
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    plt.figure(figsize=(6, 6))
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()

    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.tight_layout()
    plt.show()
```

- Normalize 預設為 False，即顯示對應的分類圖片數量，所以區域數字總和為圖片總數量。
- Normalize 若設為 True 則改為顯示分類圖片的比例，所以每一列(row)總和應為 1(100%)。且對角線為預測相應結果正確的對應比例。
- 以下三張圖分別為 ResNet18, ResNet50, ResNet152 的 confusion matrix。資料集為 valid set 的 1599 張圖片 。
- 三個模型 train 出來後的 confusion matrix 結果相似。在 FP/FN 的數量差不多。也可以發現我在預測 valid set 中非急性淋巴性白血病的圖片有較高的準確率。


Confusion matrix of ResNet18
154
354
335
756
0
1
True Label
0
1
Predicted Label
700
600
500
400
300
200


Confusion matrix of ResNet50
158
350
373
718
0
1
True Label
0
1
Predicted Label
700
600
500
400
300
200


Confusion matrix of ResNet152
175
333
347
744
0
1
True Label
0
1
Predicted Label
700
600
500
400
300
200

III. Data Preprocessing

A. How you preprocessed your data?

- 利用 transform.CenterCrop()來擷取圖片中心，可以加快運算速度。
- 為了讓訓練時可以看過更多圖片類型，增加訓練的彈性，即 data augmentation，加入了 transform.RandomHorizontalFlip()跟 transform.RandomRotation()來讓圖片有隨機水平翻轉和旋轉一個小角度的設定。
- 加入 transform.Normalize()來使得圖片的資訊更均勻。

B. What makes your method special?

- 根據 A 的描述，由於使用 CenterCrop，可以使圖片大小變小，但又不至於失去太多圖片的特徵，因為圖片主要是集中在中央，可以大幅提升運算速度。
- 可以使用 resize 來進一步減少圖片大小，也可以加快運算速度，但測試後發現會讓結果變差太多，所以最後沒有使用。
- 另外有關 random 的函數，可以在 training 時，看過更多種圖片，增加 model 彈性，豐富 data set，有助於 model generalization。
- transforms.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225))這組數字是從 ImageNet 訓練資料集中抽樣算出來的，經過正規化後，可以加快模型的收斂速度。

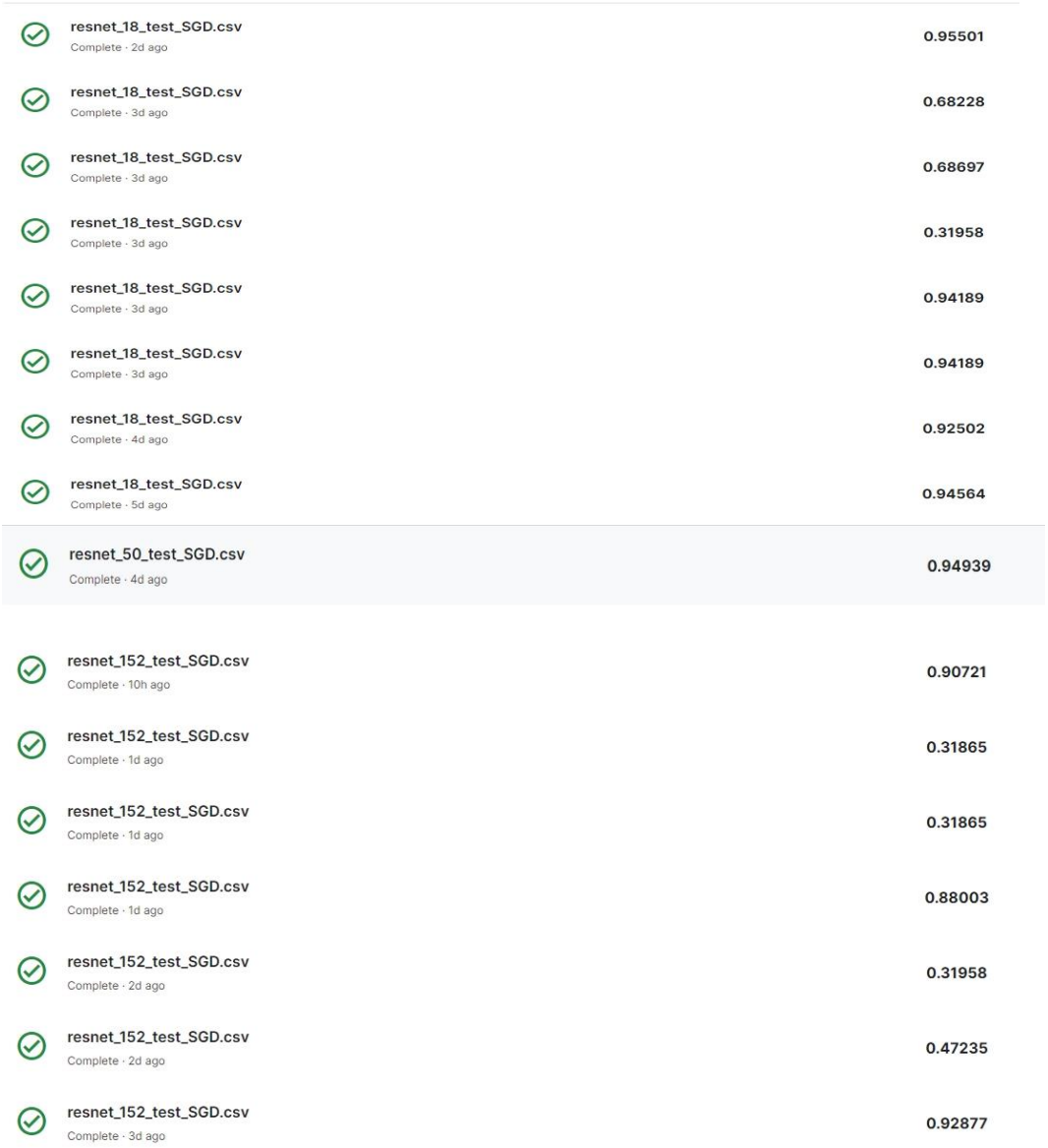
IV. Experimental results

A. The highest testing accuracy

- Screenshot

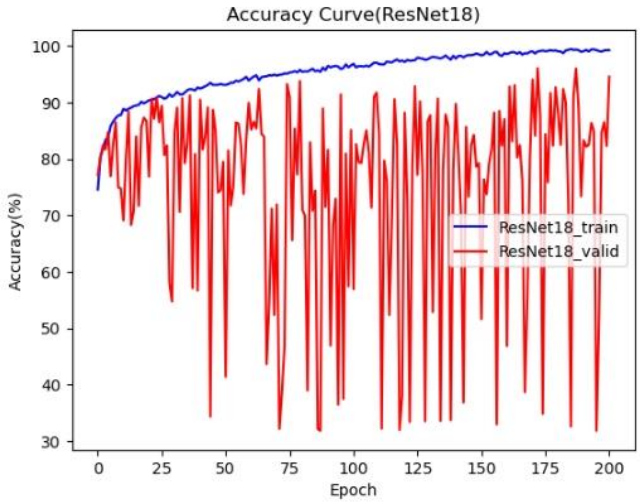
	ResNet18	ResNet50	ResNet152
Train accuracy	99.46%	99.29%	92.95%
Valid accuracy	96.06%	94.56%	89.74%
Test accuracy	95.50%	94.93%	92.87%

- Anything you want to present

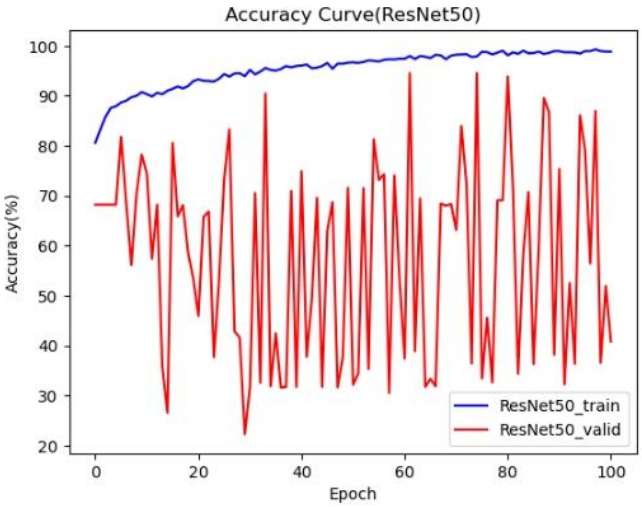


B. Comparison figures

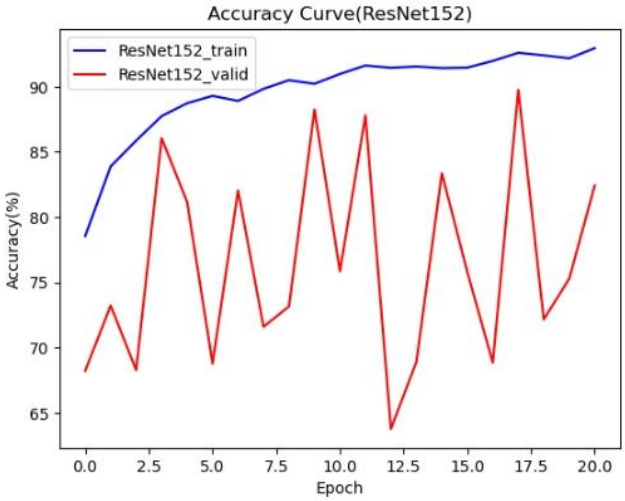
- Plotting the comparison figures(RseNet18, ResNet50, ResNet152)




```
Best train acc: 99.46%, Best valid acc: 96.06%
```



-
```
Best train acc: 99.29%, Best valid acc: 94.56%
```



-
```
Best train acc: 92.95%, Best valid acc: 89.74%
```

V. Discussion

A. Anything you want to share

- 一開始我在 train ResNet18 時，發現 epoch=201 的結果比 epoch=101 上傳到 Kaggle 的結果好，所以我覺得可以硬 train 一發 epoch 數非常大的 training。而且畫出 accuracy 的圖片進行觀察時，accuracy 上下震盪的幅度很大，突然就有可能找到一組很不錯的參數。但我一開始並沒有把 training 的 code 寫好(即每次都是重新 train，而不是從上次 train 好的 model 繼續 train)，加上前面花了很多時間不斷嘗試圖片裁切的參數，導致花了很多時間 train 出一堆垃圾。
- ResNet152 的 model state(.pth)非常大，一開始沒寫進 gitignore，又不小心 push 到 github，然後就出錯，要用 LFS 傳，但最好還是 model 都不要 push。
- Kaggle 比賽很好玩，前面排名的人很厲害，很想知道他們是用什麼方法做出來的，希望以後課程，助教可以請這些大神分享是如何利用自己獨門的技巧增加 accuracy。
- 一開始 train ResNet152 時，發現算超級久，然後又沒寫存一個 check point 的 code，導致我有時候 interrupt kernel 就沒辦法從上次的地方繼續 train，超浪費時間。也是很後面才意識到這個問題，變成最後只 train 了 21 個 epoch，成績也不是很好看，但後面寫好這部分的 code 時已經很接近 deadline 加上又有三個 model 要 train，結論就是，以後 training 應該是最後面的事，不要急著 train，而是應該把 code 架構搞好(即存 model 或是可以往下繼續 train)。
- 一開始看不太懂 Basic(Residual) block 跟 Bottleneck block 的內部結構，導致一些 channel 數一直跳 error(即 matrix size 不對)，花了一些時間進行推導。
- 我有換 Adam 來 train，但結果似乎跟 SGD 差不多，所以後面就都用 SGD 了。