Machine Learning Final Project

組別 ： 第一組

組員 ： 蔡東宏、李崇碩

一、 專題目標

利用提供的少量 train data 訓練一個 Model 來分類成人與小孩的圖片。

二、 設計過程

I. Load Data Set

```python
# Define custom dataset
class CustomImageDataset(Dataset):
    def __init__(self, adult_dir, children_dir, transform=None):
        self.adult_dir = adult_dir
        self.children_dir = children_dir
        self.transform = transform

        # Get all image file paths
        self.adult_images = [os.path.join(adult_dir, fname) for fname in os.listdir(adult_dir) if fname.endswith('.jpg')]
        self.children_images = [os.path.join(children_dir, fname) for fname in os.listdir(children_dir) if fname.endswith('.jpg')]

        # Combine adult and children image paths
        self.images = self.adult_images + self.children_images

        # Labels: adults = 0, children = 1
        self.labels = [0] * len(self.adult_images) + [1] * len(self.children_images)

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

    def __getitem__(self, idx):
        img_path = self.images[idx]
        image = Image.open(img_path).convert("RGB")
        label = self.labels[idx]
        if self.transform:
            image = self.transform(image)
        return image, label
```

以上的 code define 了一個 CustomImageDataset 的 class，能處理 adults 人、children 照片 data。首先將這些參數保存為 class 的屬性，然後 go through 這兩個資料夾下所有以.jpg 為結尾的圖片，將其蒐集起來並分別存在 self.adult_images 與 self.children_images 裡。接著，把這兩個 list 合併存到 self.images 裡，並把每張圖片加上標籤 (adults 圖片標籤為 0，children 圖片標籤為 1)。

__len__ function 會 return 圖片總數；__getitem__ function 會根據給的 index 載入對應的圖片再轉為 RGB 格式，同時存下對應的標籤，最後 return 圖片跟標籤，準備給後面的 model training 使用。

II. Data Augmentation

- **Resize**： 由於原本圖片的大小可能不同，將其統一調整至 256 x 256 的大小，確保餵入的資料大小是一致的。同樣地，對 test data 也要進行大小調整至 256 x 256 的處理

```
transforms.Resize((256,256))
```

- **RandomHorizontalFlip、RandomVerticalFlip**：
  以 0.5 的機率在水平或垂直方向隨機翻轉圖片，利用上下左右翻轉圖像，增加 train data 的多樣性，避免 overfitting 的產生。

```
transforms.RandomHorizontalFlip(),  # Random horizontal flip
transforms.RandomVerticalFlip(),    # Random vertical flip
```

- **RandomRotation**：在+-30 度的範圍內隨機地旋轉圖片，讓圖片多樣性增加。

```
transforms.RandomRotation(30),       # Random rotation
```

(旋轉後的照片)

- **RandomErasing：** 隨機擦除圖片中的某個部分，並將擦除的區域填為黑色，這可以訓練 model **在人臉戴口罩或臉部有遮擋**時也能正確判斷出小孩與成人。

```
transforms.RandomErasing(p=0.5, scale=(0.02, 0.33), ratio=(0.3, 3.3), value=0, inplace=False),
```

以下是經過測試但 accuracy 卻沒有顯著提升的方法(故我們沒有採用)：

- **ColorJitter**：隨機調整亮度、對比度、飽和度、色調來模擬不同的光線條件和色調變化。

```
transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.2),
```

- **Normalize**：透過將 image normalize，讓 model 更加快並穩定地收斂。

```
transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
```

III. Model 選擇

首先，我們先上網查詢了關於 CNN 常用的各種 Model，包括 ResNet18、ResNet50、ShuffleNet_v2、EfficientNet 、MobileNet_v2、 VGG16、VGG19、DenseNet121 等，先測試這些模

型的參數量，計算複雜度(FLOPs)以及 Testing Accuracy，比較之後我們發現 ShuffleNet_v2 模型的參數量以及計算複雜度都較小，然而它的準確率卻只有 0.8349(其他 Model 的準確率都來到 0.9 以上)，但在 trade-off 之後的考量下，我們仍然使用 ShuffleNet_v2 模型，並對這個模型做更進一步地優化，以達到更好的 performance。

比較不同的 Model:

Model	Parameter	FLOPS	Testing Accuracy
ResNet18	11.7M	2.382G	0.9266
ResNet50	25.56M	5.4G	0.9266
ShuffleNet_v2	1.367M	57.9M	0.8349
EfficientNet	5.289M	641.64M	0.9083
MobileNet_v2	3.505M	427.346M	0.9174
VGG16	138.36M	20.168G	0.9174
VGG19	143.667M	25.604G	0.9174
DenseNet121	7.98M	3.78G	0.9266

我們在研究 ShuffleNet_v2 的模型時發現，未經過修改的模型參數量為 1.367 M，FLOPs 為 57.9M， Testing Accuracy 為 0.8349，儘管如此，我們認為這個模型還可以進一步優化，除了能夠降低參數量以及運算複雜度，同時並提高準確率。首先，我們移除 ShuffleNet v2 模型的最後兩層，以降低參數量以及運算的複雜度，然而這樣會的修改會導致我們模型的準確率下降，因此我們再添加了兩層全連接層，同時，適當的 dropout 避免過度訓練。這些修改使得模型在減少大量參數和計算複雜度的同時，並且能夠提升準確率。經過修改完之後的模型參數量能壓到 143.528k，FLOPs 壓到 44M 並且 Testing Accuracy 達到 0.8833。

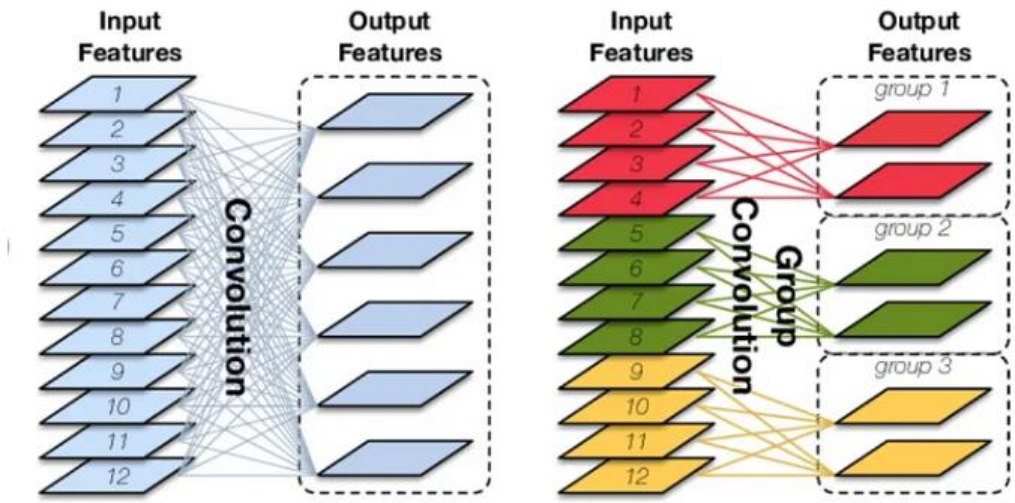
```
================================================================
Total params: 143,528
Trainable params: 143,528
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.75
Forward/backward pass size (MB): 31.22
Params size (MB): 0.55
Estimated Total Size (MB): 32.52
----------------------------------------------------------------
```

- ## ShuffleNet

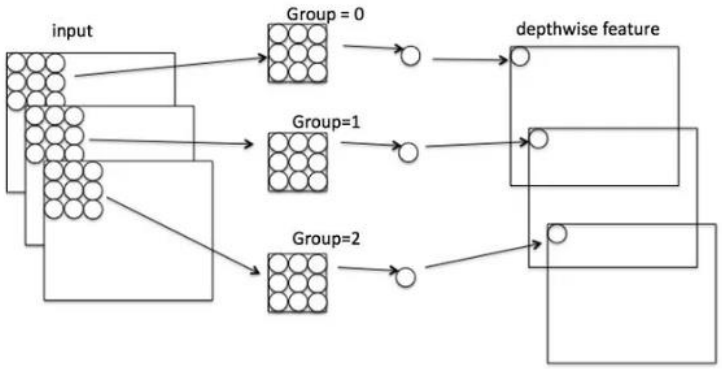
設計概念：

1. Group Convolution：將輸入的 feature 分成多組，再分別進行 Convolution，以達到減少計算量的目的。

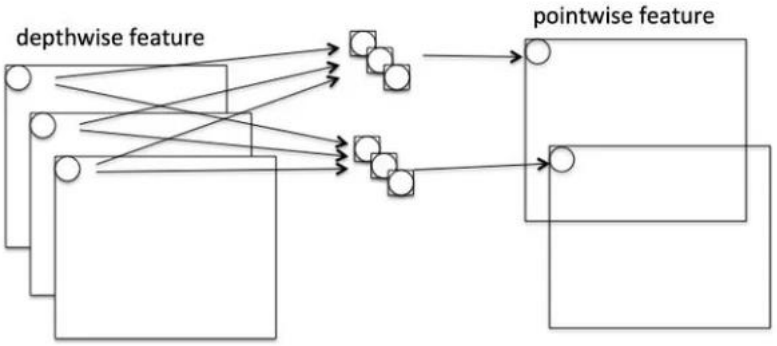
如下圖 input feature 原本有 12 個 channel，現在將其先分為 3 組，每組有 4 個 channel 再進行 Convolution，這樣每組進行 Convolution 時計算量就能減少。



2. Depthwise and Pointwise Convolution：將原本的 Convolution 拆成 Depthwise Convolution+ Pointwise Convolution，詳細的介紹會在 ShuffleNet V1 中。



上圖為 depthwise feature，下圖為 pointwise feature



()

ShuffleNet V1

結構設計 ：ShuffleNet V1 參考 ResNet 的 bottleneck 來做設計的基礎，透過前面提到的 Group Convolution 和 Channel Shuffle 來壓縮計算量與參數量，Channel Shuffle 讓各通道之間能交換訊息，以學習到更複雜的特徵以應付更困難的圖像判斷。

Channel Shuffle 示意圖

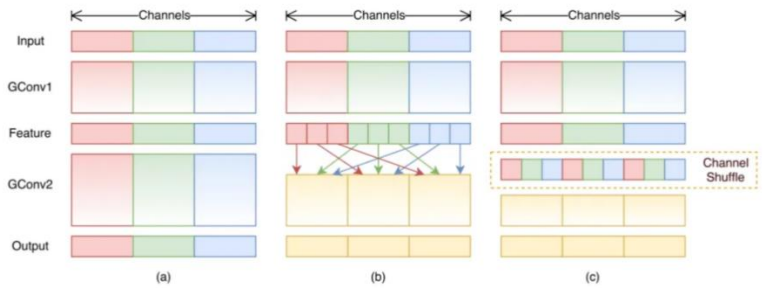
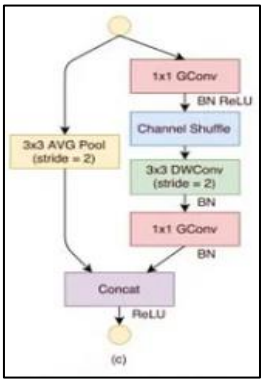


Figure 1. Channel shuffle with two stacked group convolutions. GConv stands for group convolution. a) two stacked convolution layers with the same number of groups. Each output channel only relates to the input channels within the group. No cross talk; b) input and output channels are fully related when GConv2 takes data from different groups after GConv1; c) an equivalent implementation to b) using channel shuffle.

Down-sampling 的 Stride 設為 2 如右圖



(Bottleneck 的 channel 變化是 256D → 64D → 256D，在論文實驗中已證明能有效地提取 image feature)

ShuffleNet V2

設計原則 :在設計高效神經網絡時，不能只考慮 FLOPs，還需考慮 Memory Access Cost (MAC)、GPU 並行度、文件 I/O、平台優化差異

等因素。因此，V2 在設計時採用了以下原則：

- 卷積層前後通道數相同，以達到最小化 MAC
- Group Convolution 組數增加會提升 MAC，因此 V2 的 1x1 卷積不使用 Group Convolution
- 採用 Channel Split 來控制分支數量，避免分支數量過多，導致 parallel 的能力降低。
- 避免耗時的元素級操作，合併分支時使用拼接操作。
- 當需要 downsampling 的時候，以不進行 Channel Split 來達成 channel 數量的加倍。

ShuffleNet_V1 vs ShuffleNet_V2 :

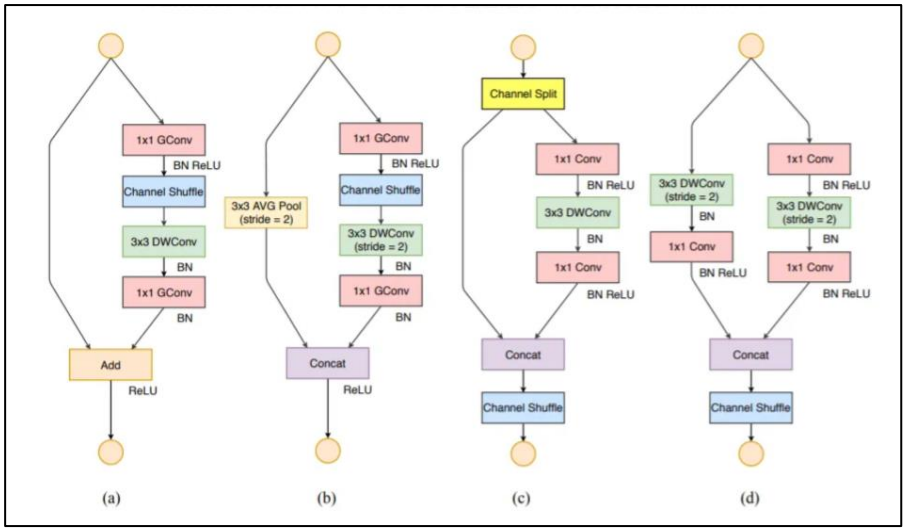
甲、在計算效率上，ShuffleNet_V2 比起 ShuffleNet_V1 更注重計算效率，因為 ShuffleNet_V2 會引入新的單元進行設計，並對整個 Network 進行重新分割和調整，重新分割之後能夠減少連接的參數量，以提升計算效率。

乙、ShuffleNet_V2 比起 ShuffleNet_V1 更簡單實用，因為 ShuffleNet_V1 中包含了 Group Convolution 以及 Channel Shuffle，而 ShuffleNet_V2 去掉了很多不必要的複雜設計。

丙、ShuffleNet_V2 會引入新的計算單元(例如 SE 模塊)，用來提升模型的準確率。

丁、ShuffleNet_V1 的通道數目分配相對固定，而 ShuffleNet_V2 會根據實際計算的特性進行不同的通道分配。

結論: ShuffleNet_V2 在保持高效計算的同時，透過更合理的設計與 Channel 分配，在模型的實際性能和應用效果都提升了。



(ShuffleNet 架構)

IV. 自己嘗試設計的 Model

```python
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=2):
        super(SimpleCNN, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.classifier = nn.Sequential(
            nn.Dropout(p=0.5),
            nn.Linear(128 * 6 * 6, 200),  # Adjusted for 
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(200, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x
```

- **nn.Conv2d：**

  作用：有效地提取圖像中的局部特徵，如邊緣、紋理等。多層卷積層堆疊可以逐步提取更高級的特徵。

  設計原因：每一層卷積層後，channel 數量（特徵數量）逐步增加（從 3(RGB)到 32，32 到 64，再到 128），這是因為希望隨著層數加深，我們能捕捉到更多複雜的特徵。

  Kernel size：3x3 的卷積核是一個常見的選擇，因為它能夠很好地平衡計算複雜度和特徵提取能力。

- **nn.ReLU：**

  作用：ReLU 函數引入非線性，使模型能夠學習更複雜的特徵。

  設計原因：使用 ReLU 來避免線性模型的限制，同時 ReLU 的計算效率高，且能夠解決 vanishing gradient 的問題。

- **nn.MaxPool2d：**

  作用：MaxPool2d 透過取窗口內最大值來減少特徵圖的尺寸，同時保留最重要的特徵。

  設計原因：MaxPool2d 有助於減少計算量和參數數量，並且利用 downsampling 使 feature map 更具不變性（如平移不變性），同時增加 receptive field。

- **nn.Dropout：**

  作用：在訓練過程中隨機丟棄一些神經元，防止過擬合，提高模型的泛化能力。

  設計原因：Dropout 是一種正則化技術，通過隨機丟棄神經元來強迫網絡學習更穩定和泛化的特徵。

- 全連接層（**nn.Linear**）：

  作用：全連接層將提出的 feature 轉換為分類的最後結果。這裡塞了兩層

  設計原因：全連接層的作用是進行 feature 融合和決定分類結果。

  Result :

```
FLOPs: 24146320.0, Params: 1015450.0
```

```
Epoch 53/100, Loss: 0.2989, Accuracy: 0.8786, Test Loss: 0.6106, Test Accuracy: 0.7167
Epoch 54/100, Loss: 0.2693, Accuracy: 0.8946, Test Loss: 0.6460, Test Accuracy: 0.6917
Epoch 55/100, Loss: 0.2715, Accuracy: 0.8911, Test Loss: 0.6245, Test Accuracy: 0.6917
Epoch 56/100, Loss: 0.2708, Accuracy: 0.8911, Test Loss: 0.6230, Test Accuracy: 0.7250
Epoch 57/100, Loss: 0.2830, Accuracy: 0.8804, Test Loss: 0.6079, Test Accuracy: 0.7167
Epoch 58/100, Loss: 0.2778, Accuracy: 0.8821, Test Loss: 0.6180, Test Accuracy: 0.6833
Epoch 59/100, Loss: 0.2763, Accuracy: 0.9000, Test Loss: 0.6277, Test Accuracy: 0.7000
Epoch 60/100, Loss: 0.2850, Accuracy: 0.9036, Test Loss: 0.6211, Test Accuracy: 0.7083
Epoch 61/100, Loss: 0.2715, Accuracy: 0.8946, Test Loss: 0.6094, Test Accuracy: 0.6917
Epoch 62/100, Loss: 0.2768, Accuracy: 0.8946, Test Loss: 0.5993, Test Accuracy: 0.7083
Epoch 63/100, Loss: 0.2744, Accuracy: 0.8982, Test Loss: 0.6210, Test Accuracy: 0.7000
Epoch 64/100, Loss: 0.2795, Accuracy: 0.8857, Test Loss: 0.6162, Test Accuracy: 0.7083
Epoch 65/100, Loss: 0.2695, Accuracy: 0.8946, Test Loss: 0.6280, Test Accuracy: 0.7000
Epoch 66/100, Loss: 0.2693, Accuracy: 0.9036, Test Loss: 0.6373, Test Accuracy: 0.6917
Epoch 67/100, Loss: 0.2821, Accuracy: 0.8911, Test Loss: 0.6073, Test Accuracy: 0.7000
Epoch 68/100, Loss: 0.2933, Accuracy: 0.8821, Test Loss: 0.6103, Test Accuracy: 0.7083
Epoch 69/100, Loss: 0.2852, Accuracy: 0.8839, Test Loss: 0.6318, Test Accuracy: 0.7083
Epoch 70/100, Loss: 0.2856, Accuracy: 0.8768, Test Loss: 0.6266, Test Accuracy: 0.7000
Epoch 71/100, Loss: 0.2726, Accuracy: 0.8911, Test Loss: 0.6260, Test Accuracy: 0.7000
Epoch 72/100, Loss: 0.2694, Accuracy: 0.8893, Test Loss: 0.6320, Test Accuracy: 0.7000
Epoch 73/100, Loss: 0.2616, Accuracy: 0.9000, Test Loss: 0.6227, Test Accuracy: 0.7167
Epoch 74/100, Loss: 0.2653, Accuracy: 0.8982, Test Loss: 0.6032, Test Accuracy: 0.7167
Epoch 75/100, Loss: 0.2762, Accuracy: 0.8929, Test Loss: 0.6217, Test Accuracy: 0.7000
Epoch 76/100, Loss: 0.2660, Accuracy: 0.9018, Test Loss: 0.5976, Test Accuracy: 0.7333
Epoch 77/100, Loss: 0.2811, Accuracy: 0.8982, Test Loss: 0.6120, Test Accuracy: 0.7083
Epoch 78/100, Loss: 0.2861, Accuracy: 0.9089, Test Loss: 0.6007, Test Accuracy: 0.7167
Epoch 79/100, Loss: 0.2730, Accuracy: 0.9000, Test Loss: 0.6171, Test Accuracy: 0.7167
Epoch 80/100, Loss: 0.2798, Accuracy: 0.8946, Test Loss: 0.6179, Test Accuracy: 0.7167
Epoch 81/100, Loss: 0.2676, Accuracy: 0.9000, Test Loss: 0.5936, Test Accuracy: 0.7083
Epoch 82/100, Loss: 0.2839, Accuracy: 0.8821, Test Loss: 0.6056, Test Accuracy: 0.7083
Epoch 83/100, Loss: 0.2705, Accuracy: 0.8964, Test Loss: 0.6238, Test Accuracy: 0.7250
Epoch 84/100, Loss: 0.2702, Accuracy: 0.8946, Test Loss: 0.5975, Test Accuracy: 0.7083
Epoch 85/100, Loss: 0.2748, Accuracy: 0.9036, Test Loss: 0.5957, Test Accuracy: 0.7000
Epoch 86/100, Loss: 0.2780, Accuracy: 0.8982, Test Loss: 0.6026, Test Accuracy: 0.7167
Epoch 87/100, Loss: 0.2757, Accuracy: 0.8964, Test Loss: 0.6325, Test Accuracy: 0.7000
Epoch 88/100, Loss: 0.2872, Accuracy: 0.8911, Test Loss: 0.6124, Test Accuracy: 0.7250
Epoch 89/100, Loss: 0.2577, Accuracy: 0.9107, Test Loss: 0.6264, Test Accuracy: 0.6917
Epoch 90/100, Loss: 0.2671, Accuracy: 0.8893, Test Loss: 0.6187, Test Accuracy: 0.7083
Epoch 91/100, Loss: 0.2709, Accuracy: 0.8982, Test Loss: 0.6158, Test Accuracy: 0.7167
Epoch 92/100, Loss: 0.2562, Accuracy: 0.8929, Test Loss: 0.6230, Test Accuracy: 0.7083
Epoch 93/100, Loss: 0.2725, Accuracy: 0.9071, Test Loss: 0.6152, Test Accuracy: 0.7083
Epoch 94/100, Loss: 0.2688, Accuracy: 0.8964, Test Loss: 0.6153, Test Accuracy: 0.7167
Epoch 95/100, Loss: 0.2740, Accuracy: 0.9000, Test Loss: 0.6295, Test Accuracy: 0.6917
Epoch 96/100, Loss: 0.2821, Accuracy: 0.8982, Test Loss: 0.6078, Test Accuracy: 0.7250
Epoch 97/100, Loss: 0.2834, Accuracy: 0.8804, Test Loss: 0.6199, Test Accuracy: 0.7083
Epoch 98/100, Loss: 0.2771, Accuracy: 0.8875, Test Loss: 0.6018, Test Accuracy: 0.7250
```

結論 ：

觀察自己設計的結果發現層數還沒堆疊上去只是使用很簡單的方法先進行測試，參數量就已經不太理想了，因此就沒有繼續測試下去，使用 ShuffleNet 來取代之。

V. 最終 Model

為了達到較高的 testing accuracy 以及減少參數量，我使用了預訓練的 shufflenet_v2 模型並進行了修改。

首先，我移除 shufflenet_v2 模型裡的最後兩層，並加入 global average pool 來做替代以減少參數量。接著，我們利用 flatten 將三維數據轉換為一維的向量，然後加入一層 Dropout 以 0.2 的機率隨機丟棄一些 Neurons，再加了一個 in_feature 為 192，out_feature 為 2 的 connected layer。最後，我們再加入一層 Dropout 但改以 0.02 的機率隨機丟棄一些 Neurons，並添加一層 in_feature 為 2 (這是為了接上一層的 out_feature = 2)，out_feature 為 2 的 fully connected layer。經過這些調整，model 可以在保留甚至是獲得更好的準確性同時大幅度地減少參數量(1.367 M => 0.143M)。

```python
class Network(nn.Module):
    def __init__(self, num_classes=2):
        super(Network , self).__init__()
        self.model = torchvision.models.shufflenet_v2_x0_5(pretrained=True)

        self.model = nn.Sequential(*list(self.model.children())[:-2])
        self.model.add_module('global_avg_pool', nn.AdaptiveAvgPool2d(1))
        self.model.add_module('flatten', nn.Flatten())
        self.model.add_module('dropout', nn.Dropout(p=0.2))
        self.model.add_module('fc', nn.Linear(in_features=192, out_features=2))


        self.dropout = nn.Dropout(p=0.02)
        self.fc = nn.Linear(in_features=2, out_features=num_classes)

    def forward(self, x):
        x = self.model(x)
        x = self.dropout(x)
        x = self.fc(x)
        return x
```

VI. 計算 FLOPs、parameters 的套件

使用 thop 套件裡的 profile 來協助計算每個 model 的 FLOPs 與 parameters，另外還使用了 torchsummary 套件裡的 summary 來 print 出每一層的參數量、Layer type、Output Shape 以方便我們了解內部的結構，並進一步更改。

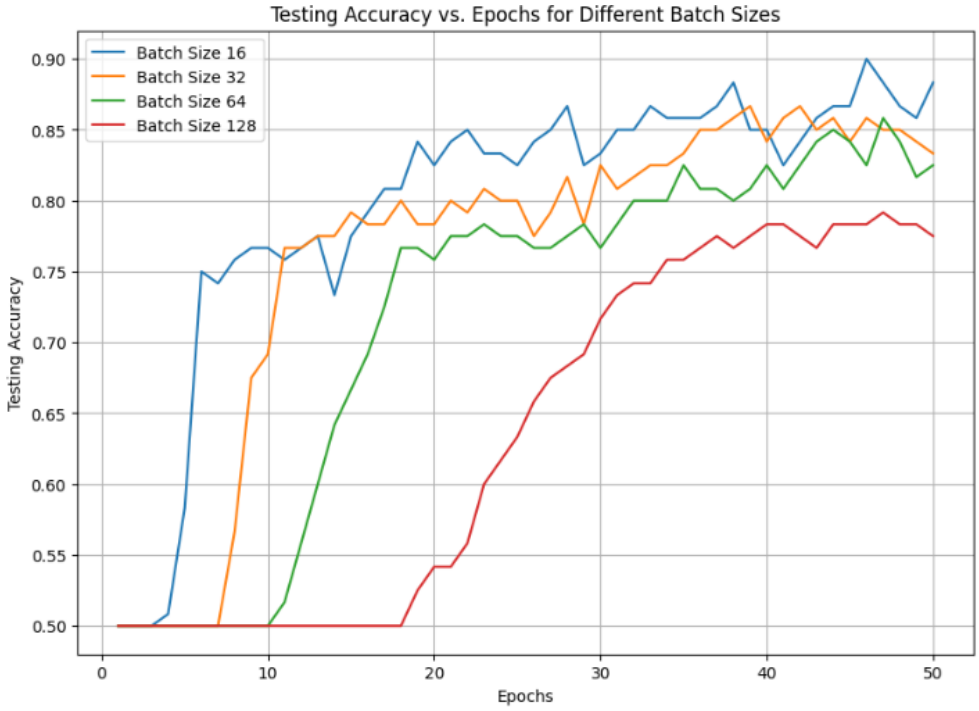
```python
from thop import profile
from torchsummary import summary
model_for_profiling = copy.deepcopy(model)
# Profile the cloned model without modifying the original model's state_dict
model_for_profiling.eval()  # Set the cloned model to evaluation mode
with torch.no_grad():
    flops, params = profile(model_for_profiling, inputs=(random_tensor,), verbose=False)
print(f"FLOPs: {flops}, Params: {params}")
summary(model,(3,256,256))
```

VII. Batch Size

將 data set 分批放到 model 裡訓練，而使用 batch 最大的目的是找出一小部分資料的 loss 就去更新模型參數。

根據實驗的結果我們找到 batch size 在 16 時有最好的 Testing Accuracy，因此選擇 batch_size = 16 這個參數。

觀察趨勢可以發現 batch_size = 128 的收斂速度與 Accuracy 都明顯較其他三者還要來得差，收斂速度慢我認為是每個批次的數量太多導致更新參數的速度較緩慢，而 Accuracy 低是因 batch_size 太大容易記下這個批次共同的特徵導致 overfitting 產生。



VIII. 使用 seed

確保每次測試的環境都一樣，這樣更改 code 進行測試做 optimize 前後比較的結果會比較準確

```
def same_seeds(seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.benchmark = False
    torch.backends.cudnn.deterministic = True

same_seeds(48763)
```

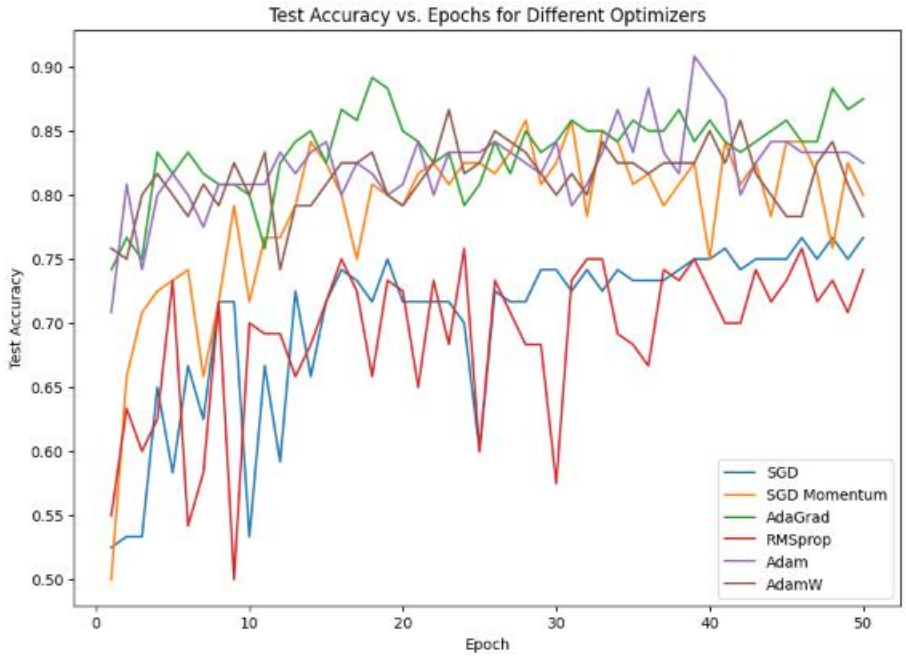
IX. Optimizer 的選擇

Learning rate = 0.0001，經測試此 learning rate 可達到最好的 testing accuracy。

進行 5 種 optimizer 的比較分別是

SGD、SGD Momentum、AdaGrad、RMSprop、Adam、AdamW

Result :



結論 ： 雖然 AdaGrad 在最後幾個 epochs 有超越 Adam，但 Adam 最高的 Testing Accuracy 有達到 90%，因此我們採用 Adam 作 optimizer。

X. Weight_decay

目的是為了讓模型的權重保持在較小的範圍內，使模型更簡單且 Generalization 的能力更好。

調整 Weight decay 測試 L1、L2 Normalization

⇨ **L1**(weight decay = 0.001) : max testing accuracy **82.5 %** (收斂後約在 80%左右震盪)

⇨ **L2**(weight decay = 0.01) : max testing accuracy **88.33 %**(收斂後約在 85%左右震盪)

結論 ：

L2 的結果跟沒有加 weight 前是一樣的，而 L1 的結果比沒有加 weight decay 還要差，故我們最後**採用的方式是不加入 weight decay**。

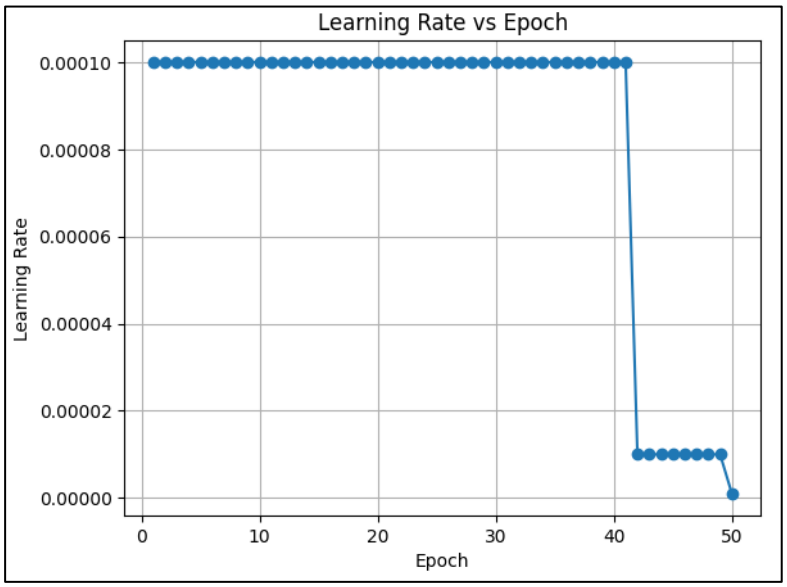
XI. 動態調整 Learning Rate

使用 ReduceLROnPlateau 來動態調整 Learning Rate :

因為在模型最一開始會離 global minimum 比較遠，此時我們可以採用較大的移動距離加速收斂，但是在靠近 minimum 時，過大的 learning rate 可能導致參數難以到達 loss 更低的地方，造成反覆震盪的情形，因此我採用了 ReduceLROnPlateau 來幫助調整 learning

rate。ReduceLROnPlateau 會在偵測到 model 的 testing loss 停滯時開始動態調整 learning rate，優點是可以幫助 model 在訓練過程中找到最佳解的位置。

以下是 Learning Rate 的變化圖 ：



調整前 ：

```
Epoch 40/50, Loss: 0.1276, Accuracy_train: 0.9625, Accuracy_test: 0.8417,
Epoch 41/50, Loss: 0.1223, Accuracy_train: 0.9625, Accuracy_test: 0.8417,
Epoch 42/50, Loss: 0.1410, Accuracy_train: 0.9589, Accuracy_test: 0.8417,
Epoch 43/50, Loss: 0.1193, Accuracy_train: 0.9661, Accuracy_test: 0.8250,
Epoch 44/50, Loss: 0.1028, Accuracy_train: 0.9750, Accuracy_test: 0.8333,
Epoch 45/50, Loss: 0.1113, Accuracy_train: 0.9589, Accuracy_test: 0.8250,
Epoch 46/50, Loss: 0.0831, Accuracy_train: 0.9786, Accuracy_test: 0.8417,
Epoch 47/50, Loss: 0.0933, Accuracy_train: 0.9661, Accuracy_test: 0.8250,
Epoch 48/50, Loss: 0.0842, Accuracy_train: 0.9786, Accuracy_test: 0.8500,
Epoch 49/50, Loss: 0.0785, Accuracy_train: 0.9732, Accuracy_test: 0.8250,
Epoch 50/50, Loss: 0.0778, Accuracy_train: 0.9821, Accuracy_test: 0.8333,
```

調整後 ：

```
Epoch 00041: reducing learning rate of group 0 to 1.0000e-05.
Learning Rate: 0.000010
Epoch 42/50, Loss: 0.2302, Accuracy_train: 0.9393, Accuracy_test: 0.8500
Learning Rate: 0.000010
Epoch 43/50, Loss: 0.2496, Accuracy_train: 0.9107, Accuracy_test: 0.8583
Learning Rate: 0.000010
Epoch 44/50, Loss: 0.2406, Accuracy_train: 0.9286, Accuracy_test: 0.8500
Learning Rate: 0.000010
Epoch 45/50, Loss: 0.2355, Accuracy_train: 0.9304, Accuracy_test: 0.8583
Learning Rate: 0.000010
Epoch 46/50, Loss: 0.2483, Accuracy_train: 0.9214, Accuracy_test: 0.8583
Learning Rate: 0.000010
Epoch 47/50, Loss: 0.2367, Accuracy_train: 0.9268, Accuracy_test: 0.8583
Learning Rate: 0.000010
Epoch 48/50, Loss: 0.2319, Accuracy_train: 0.9321, Accuracy_test: 0.8500
Learning Rate: 0.000010
Epoch 49/50, Loss: 0.2348, Accuracy_train: 0.9321, Accuracy_test: 0.8583
Epoch 00049: reducing learning rate of group 0 to 1.0000e-06.
Learning Rate: 0.000001
Epoch 50/50, Loss: 0.2224, Accuracy_train: 0.9304, Accuracy_test: 0.8583
Training complete
```

結果：

觀察結果可以發現 Learning Rate 被調低後大幅度的不穩定震盪確實減少了，只會在 85%附近徘迴不會再跳回 80%去，確實有達到我們設計這個 dynamic adjusting Learning Rate 的目的。

XII. Train Loop

```python
for epoch in range(num_epochs):
    # print learning rate
    # print(f'{scheduler.get_last_lr()[0]:.6f}')
    running_loss = 0.0
    correct_train = 0
    total_train = 0
    correct_test = 0
    total_test = 0
    model.train()

    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()

        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * inputs.size(0)
        _, predicted = torch.max(outputs, 1)
        total_train += labels.size(0)
        correct_train += (predicted == labels).sum().item()

    scheduler.step(running_loss)
```

圖上的程式是 Train Loop，會在每個 epoch 結束後利用 scheduler.step(running_loss)去更新學習率，每一個周期都經過 feed-forward (outputs = model(inputs)那行)並計算 loss，以及 error back-propagation(loss.backward())再進一步進行參數的優化。

XIII. 紀錄最佳的 Testing Accuracy

將跑過的每個 Testing Accuracy 記下，並 load 最好的 model(Best Testing Accuracy 那組)去 demo，雖然不能保證此 model 在 demo 時一樣會有最好的 Testing Accuracy，但我們能大致確定的是他在 testing 有較好的表現機率較高。

```
model = Network(num_classes=2)
model.load_state_dict(torch.load('best_model.pth'))
model = model.to(device)
model.eval()

correct_test = 0
total_test = 0
with torch.no_grad():
    for inputs, labels in test_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total_test += labels.size(0)
        correct_test += (predicted == labels).sum().item()

    epoch_acc_test = correct_test / total_test
    print(f'Accuracy_test: {epoch_acc_test:.4f}')
```

三、 Reference

https://hackmd.io/@allen108108/H1l4zqtp4

https://medium.com/%E5%AD%B8%E4%BB%A5%E5%BB%A3%E6%89%8D/note-shufflenet-%E5%AD%B8%E7%BF%92%E5%BF%83%E5%BE%97-38f2f715e19a