DL HW5

109610025 陳品妍

結果一: 初步寫出架構並驗證是否可以訓練再去針對訓練過程進行改進

Best accuracy while training:83.35%

```
Epoch 66/100
Train Loss: 0.0025, Train Accuracy: 100.00%
Validation Loss: 1.8311, Validation Accuracy: 83.08%
Saved Best Model
Learning Rate: 0.001000
Epoch 67/100
Train Loss: 0.0020, Train Accuracy: 100.00%
Validation Loss: 1.8420, Validation Accuracy: 83.26%
Saved Best Model
Learning Rate: 0.001000
Epoch 68/100
Train Loss: 0.0016, Train Accuracy: 100.00%
Validation Loss: 1.8553, Validation Accuracy: 83.27%
Saved Best Model
Learning Rate: 0.001000
Epoch 69/100
Train Loss: 0.0013, Train Accuracy: 100.00%
Validation Loss: 1.8714, Validation Accuracy: 83.35%
Saved Best Model
Learning Rate: 0.001000
```

Pytest 結果:

```
flops = 11776320
Model parameter size = 3814.391 kB
Accuracy = 83.35 %
```

Performance:

發現到後期結果會有一點 overfitting 的現象，因此使用 learning rate scheduler 來幫助其

```
Epoch 39/100
Train Loss: 0.6358, Train Accuracy: 79.11%
Validation Loss: 1.9892, Validation Accuracy: 58.54%
Learning Rate: 0.010000
Epoch 40/100
Train Loss: 0.6117, Train Accuracy: 79.48%
Validation Loss: 2.0226, Validation Accuracy: 58.98%
Learning Rate: 0.001000
Epoch 41/100
Train Loss: 0.5795, Train Accuracy: 81.55%
Validation Loss: 1.1895, Validation Accuracy: 74.92%
Saved Best Model
Learning Rate: 0.001000
Epoch 42/100
Train Loss: 0.3392, Train Accuracy: 89.32%
Validation Loss: 1.1360, Validation Accuracy: 77.19%
Saved Best Model
Learning Rate: 0.001000
Epoch 43/100
Train Loss: 0.2539, Train Accuracy: 92.93%
Validation Loss: 1.1250, Validation Accuracy: 78.27%
Saved Best Model
Learning Rate: 0.001000
Epoch 44/100
Train Loss: 0.1983, Train Accuracy: 95.37%
Validation Loss: 1.1320, Validation Accuracy: 78.72%
Saved Best Model
```

結果二: epoch 提升到 100 個以及設定每 35 個 epoch 將 lr 再往下調整(0.01-0.001-0.0001)

Best accuracy: 84.13%

```
Epoch 96/100
Train Loss: 0.0000, Train Accuracy: 100.00%
Validation Loss: 2.8560, Validation Accuracy: 84.12%
Saved Best Model
Learning Rate: 0.000100
Epoch 97/100
Train Loss: 0.0000, Train Accuracy: 100.00%
Validation Loss: 2.8730, Validation Accuracy: 84.07%
Learning Rate: 0.000100
Epoch 98/100
Train Loss: 0.0000, Train Accuracy: 100.00%
Validation Loss: 2.8898, Validation Accuracy: 84.13%
Saved Best Model
Learning Rate: 0.000100
Epoch 99/100
Train Loss: 0.0000, Train Accuracy: 100.00%
Validation Loss: 2.9066, Validation Accuracy: 84.11%
Learning Rate: 0.000100
Epoch 100/100
Train Loss: 0.0000, Train Accuracy: 100.00%
Validation Loss: 2.9234, Validation Accuracy: 84.09%
Learning Rate: 0.000100
```

Pytest 結果:

```
flops = 11776320
Model parameter size = 3814.391 kB
Accuracy = 84.13 %
```

Performance:

結果三:結果發現 train 跟 validation 之間的差異還是有點太大，因此加上 dropout = 0.2 在架構中來解決 overfitting 的問題。

```
Epoch 97/100
Train Loss: 2.0487, Train Accuracy: 49.97%
Validation Loss: 1.3704, Validation Accuracy: 62.25%
Learning Rate: 0.000100
Epoch 98/100
Train Loss: 2.0362, Train Accuracy: 50.30%
Validation Loss: 1.3753, Validation Accuracy: 62.32%
Learning Rate: 0.000100
Epoch 99/100
Train Loss: 2.0393, Train Accuracy: 50.05%
Validation Loss: 1.3732, Validation Accuracy: 61.98%
Learning Rate: 0.000100
Epoch 100/100
Train Loss: 2.0206, Train Accuracy: 50.51%
Validation Loss: 1.3744, Validation Accuracy: 62.02%
Learning Rate: 0.000100
```

結果發現 train 時並沒有辦法好好的使用資料，因此降低 dropout_prob。

結果四: 改成 DROPOUT = 0.1

```
Epoch 96/100
Train Loss: 1.0617, Train Accuracy: 71.21%
Validation Loss: 0.8447, Validation Accuracy: 78.72%
Learning Rate: 0.000100
Epoch 97/100
Train Loss: 1.0611, Train Accuracy: 71.31%
Validation Loss: 0.8439, Validation Accuracy: 78.51%
Learning Rate: 0.000100
Epoch 98/100
Train Loss: 1.0619, Train Accuracy: 71.11%
Validation Loss: 0.8448, Validation Accuracy: 78.64%
Learning Rate: 0.000100
Epoch 99/100
Train Loss: 1.0550, Train Accuracy: 71.55%
Validation Loss: 0.8451, Validation Accuracy: 78.60%
Learning Rate: 0.000100
Epoch 100/100
Train Loss: 1.0510, Train Accuracy: 71.69%
Validation Loss: 0.8425, Validation Accuracy: 78.77%
Learning Rate: 0.000100
```

Dropout 似乎並不太適合這個架構，因此改回原本的架構。

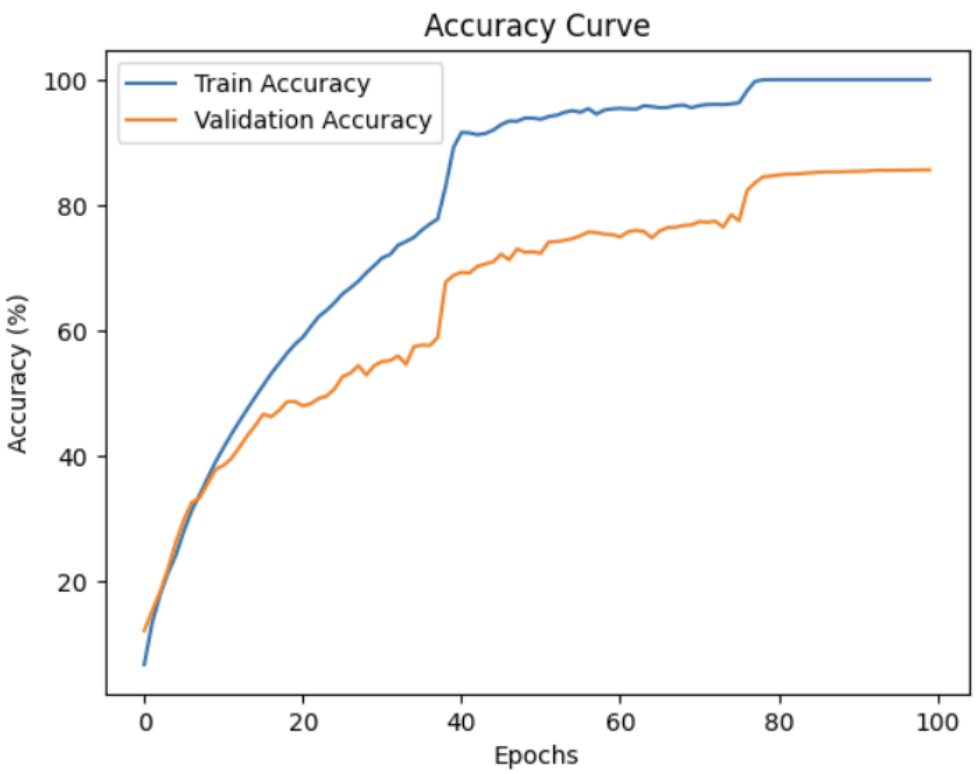
結果五:

```
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-2, weight_decay=1e-4)


# Define scheduler
scheduler = StepLR(optimizer, step_size=38,gamma=0.5)  # 每隔 40 个 epoch 学习率乘以 0.1
```

```
Epoch 97/100
Train Loss: 0.0000, Train Accuracy: 100.00%
Validation Loss: 2.0804, Validation Accuracy: 85.55%
Learning Rate: 0.002500
Epoch 98/100
Train Loss: 0.0000, Train Accuracy: 100.00%
Validation Loss: 2.1224, Validation Accuracy: 85.59%
Saved Best Model
Learning Rate: 0.002500
Epoch 99/100
Train Loss: 0.0000, Train Accuracy: 100.00%
Validation Loss: 2.1646, Validation Accuracy: 85.60%
Saved Best Model
Learning Rate: 0.002500
Epoch 100/100
Train Loss: 0.0000, Train Accuracy: 100.00%
Validation Loss: 2.2071, Validation Accuracy: 85.62%
Saved Best Model
Learning Rate: 0.002500
```

```
flops = 11776320
Model parameter size = 3814.391 kB
Accuracy = 85.62 %
```



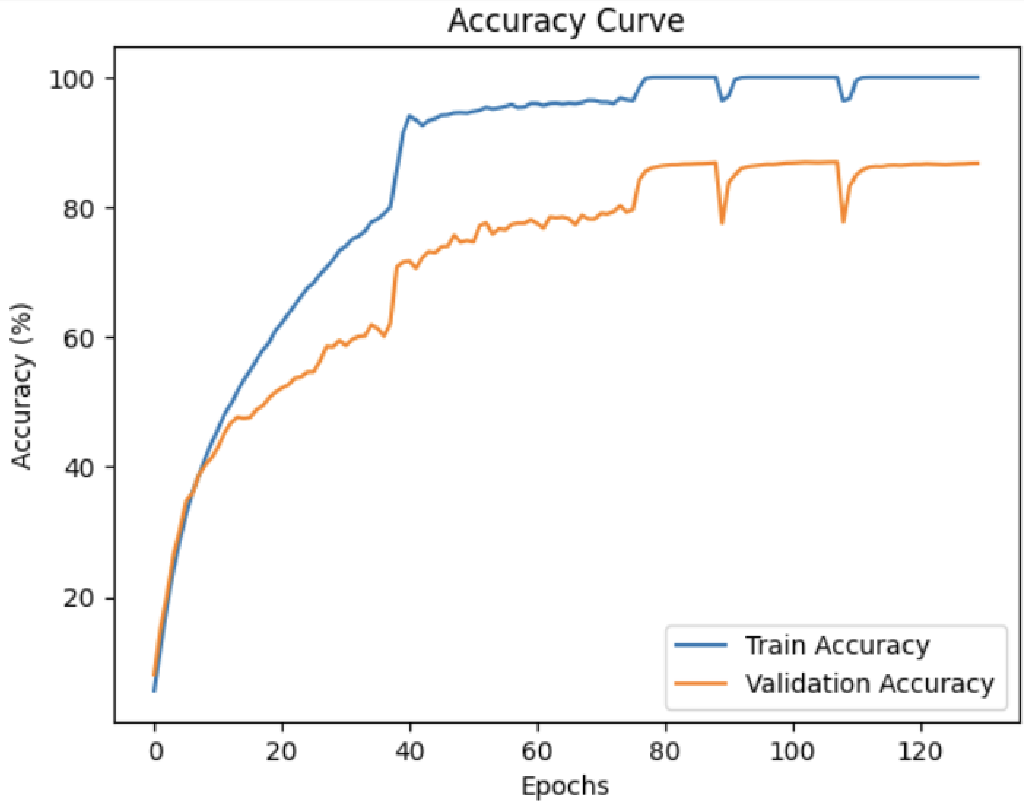
原本在 35epoch 時降低 lr(0.1)，但是發現降低之後，結果無法收斂太多(約到 81%)。因此，將降低 epoch 拉到 38，使模型可以再多訓練一點再降低 learning rate，且降低的幅度不要太大，使之可以保持一定的收斂速度。

結果六: 加大訓練 epoch (最佳結果)

Best accuracy while training: 86.94

```
Epoch 107/130
Train Loss: 0.0000, Train Accuracy: 100.00%
Validation Loss: 1.7754, Validation Accuracy: 86.94%
Saved Best Model
Learning Rate: 0.002500
```

```
flops = 11776320
Model parameter size = 3814.391 kB
Accuracy = 86.95 %
```



參考 MobileNetV3

架構:

方法使用&how :

1. Efficient squeeze and excitation modules:

使用 squeeze-and-excitation 模組，透過 shrink feature map 來增加取得 global information 以及透過 excitation 可以學習 channel-wise weights 來結合通道的資訊。架構上使用的是 bottleneck structure(先縮小再放大)然後再使用 sigmoid activation 來製造出一個 channel-wise attention map。這個方法是要選擇的因為 squeeze and excitation 會導致計算效率下降(引入額外的 cost from pooling and fc)，而淺層的 block 通常處理較低階的特徵(例如邊緣紋理等)，使 squeeze and excitation(整合通道們調整權重)的效益降低。且前面的 block 設計沒有進行擴展，使得通道數比後面的 channel 少，能提取的特徵就相對少，較浪費。因此當發展到後面的 block (use_se = True)才會使用這個技巧。(下方有架構圖供參考)

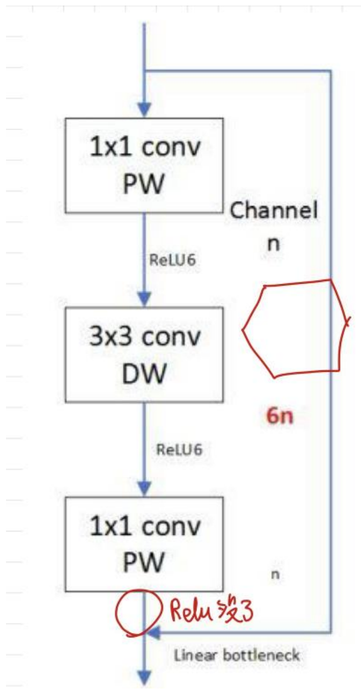
實現:

self.se = SEBlock(hidden_dim) if use_se else nn.Identity()

2. Inverted residual and linear bottleneck:

為了要減少訓練的 computational cost 跟 mac。原本的 ResNet 作法是將通道壓所 ➔ conv.來提取特徵 ➔ 擴張(還原)，然而這種方法使 acc 下降因為 parameters 下降導致能提取的資訊減少而且降 channel 後會有一個 relu 所以會損失許多資料。因此 linear bottleneck 減少 relu 帶來的資料損失(因為 relu 會把幾乎一半的資料都設成零，尤其對於 depthwise conv.不是一個好的方法)。

在 MobileVetV3Block 當中，先擴展後進行 depthwise conv.然後再使用同一個線性 conv.來將其映射到低維當中。



實現:

self.use_res_connect = self.stride == 1 and in_channels == out_channels

3. Hardswish activation:

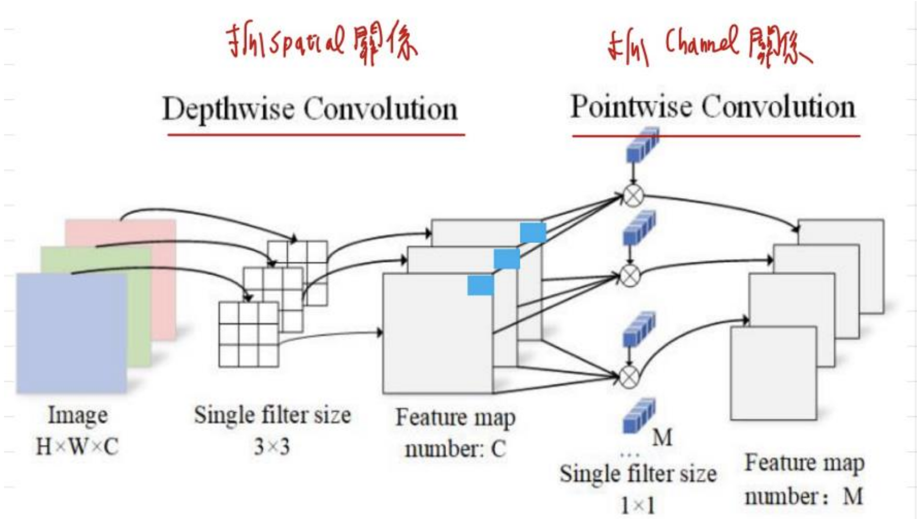
Hardswich 可以代替 swish 所帶來的計算量龐大的缺點但同時又可以有效提溝網路的精度。在 key layers 的 activation()就是使用 hardswish，特別是在處理非線性的 block。

實現:

activation = nn.Hardswish

4. Depthwise separable conv.:

主要是可以減少餐數量以及計算成本。透過 depthwise conv. 對於每個 channel 的 spatial feature 獨立處理然後再透過 pointwise conv.在 channel 間混和訊息。在架構中的 dwconv 有使用。



Depthwise and pointwise conv. 如何去減少整體計算量(相較於傳統 conv.而言)

$$HWNK^2\ (depthwise) + HWNM\ (pointwise) = HWN(K^2 + M)$$

$$\frac{depthwise+pointwise}{conv} = \frac{(K^2+M)HWN}{K^2MHWN} = \frac{1}{M} + \frac{1}{K^2} \simeq \frac{1}{K^2}$$

For 3x3 convolution, complexity reduction ~1/9

實現:

self.dwconv = nn.Conv2d(hidden_dim, hidden_dim, kernel_size=kernel_size, stride=stride,
padding=kernel_size // 2, groups=hidden_dim, bias=False)

5. Progessive reduction of spatial dimensions:

為了減少 spatial resolution 當在增加 channel dimensions(tradeoff between feature representation and computational overhead)。在 mobilenetv3block 層中將 stride 設為 2 的 conv.進行 sample。

整體架構: (cifar-100 輸入 32*32)

input = B×3×32×32　　B: batch size

init_conv (stride=1, padding=1): $(B,3,32,32) \rightarrow (B,16,32,32)$

不使用 Squeeze & excitation, ReLU as activation function

Block 1: $(B,16,32,32) \rightarrow (B,16,32,32)$

Block 2: $(B,16,32,32) \xrightarrow{\text{擴展}} (B,64,32,32) \xrightarrow[\text{stride}=2]{\text{Conv.}} (B,64,16,16)$

$\xrightarrow{\text{壓縮}} (B,24,16,16)$

Block 3: $(B,24,16,16) \xrightarrow{\text{擴展}} (B,72,16,16) \xrightarrow{\text{Conv}} (B,72,16,16)$

$\xrightarrow{\text{壓縮}} (B,24,16,16)$

※第一個 block 沒有 expand ∵

① 避免淺層特徵被過度表示

② 減少後續層負擔

∵ $FLOPS = C_{in} \cdot C_{out} \cdot k \cdot k \cdot H \cdot W$

C_{in} 過早 expand ⇒ FLOPS ↑

使用 squeeze & excitation, 使用 hardswish as activation function

Block 4: $(B,24,16,16) \xrightarrow{\text{擴展}} (B,72,16,16) \xrightarrow{\text{conv. w/ stride}=2} (B,72,8,8) \xrightarrow{\text{S\&E blocks}} (B,72,1,1)$

$\xrightarrow[\text{降}]{} (B,18,1,1) \xrightarrow[\text{升}]{} (B,72,1,1) \longrightarrow (B,72,8,8) \xrightarrow{\text{壓縮}} (B,40,8,8)$

Block 5: $(B,40,8,8) \longrightarrow (B,40,8,8)$

block 6: $(B,40,8,8) \longrightarrow (B,80,4,4)$

Block 7: $(B,80,4,4) \longrightarrow (B,112,4,4)$

Block 8: $(B,112,4,4) \longrightarrow (B,160,2,2)$

final_conv.

$(B,160,2,2) \xrightarrow[\text{B.N. \& Hardswish}]{\text{pointwise conv.}} (B,960,2,2)$

Self pooling & FC & flatten

Self pooling: $(B,960,2,2) \longrightarrow (B,960,1,1)$

flatten: $(B,960,1,1) \longrightarrow (B,960)$

Fc: $(B,960) \longrightarrow (B,100)$

Reference:

講義

https://blog.csdn.net/baidu_36913330/article/details/120079096

https://blog.csdn.net/weixin_43334693/article/details/130772823