機器學習導論
辨識動物

資科三B 陳妙音 11173228

目錄

資料預處理

01

Step1 載入資料

只有train.npy
沒有test.npy資料
➡使用
train_test_split

```
[ ]  X_train  =  np.load("/content/drive/MyDrive/224/224train/X_train.npy")
     y_train  =  np.load("/content/drive/MyDrive/224/y_train.npy")
```

```
[ ]  X_train_split,  X_test_split,  y_train_split,  y_test_split  =  train_test_split(
            X_train,  y_train,  test_size=0.2,  random_state=42)


     print("X_train_split  shape:",  X_train_split.shape)
     print("X_test_split  shape:",  X_test_split.shape)
     print("y_train_split  shape:",  y_train_split.shape)
     print("y_test_split  shape:",  y_test_split.shape)

     del  X_train,  y_train
     gc.collect()
```

因為資料太大記憶體不夠,
所以用不到的變數先刪掉

```
X_train_split shape: (12316, 224, 224, 3)
X_test_split shape: (3080, 224, 224, 3)
y_train_split shape: (12316,)
y_test_split shape: (3080,)
7331
```

Step2 正規化

但資料真的太大RAM不足

➡使用批次寫入file的方式，下次執行也可以直接load即可

```
#  分批正規化並儲存到新檔案
batch_size  =  50    #  根據記憶體調整
n_train  =  X_train_split.shape[0]
n_test  =  X_test_split.shape[0]

#  創建空的  .npy  檔案用於儲存正規化結果
X_train_split_normalized_file  =  '/content/drive/MyDrive/224/X_train_split_normalized.npy'
X_test_split_normalized_file  =  '/content/drive/MyDrive/224/X_test_split_normalized.npy'

#  初始化檔案（只需要指定形狀和類型）
np.save(X_train_split_normalized_file,  np.zeros(X_train_split.shape,  dtype='float32'))
np.save(X_test_split_normalized_file,  np.zeros(X_test_split.shape,  dtype='float32'))

#  使用  memmap  打開檔案進行寫入
X_train_split_normalized  =  np.memmap(X_train_split_normalized_file,  dtype='float32',
                                       mode='r+',  shape=X_train_split.shape)
X_test_split_normalized  =  np.memmap(X_test_split_normalized_file,  dtype='float32',
                                      mode='r+',  shape=X_test_split.shape)

#  分批正規化並寫入
for  i  in  range(0,  n_train,  batch_size):
        batch  =  X_train_split[i:i+batch_size].astype('float32')  /  255.0
        X_train_split_normalized[i:i+batch_size]  =  batch
        X_train_split_normalized.flush()
        del  batch    #  釋放臨時批次
        gc.collect()

for  i  in  range(0,  n_test,  batch_size):
        batch  =  X_test_split[i:i+batch_size].astype('float32')  /  255.0
        X_test_split_normalized[i:i+batch_size]  =  batch
        X_test_split_normalized.flush()
        del  batch
        gc.collect()
```

寫入檔案存在雲端

批次正規化

刪除用不到的變數，釋放記憶體

Step2 正規化

➡下次執行直接load，不用再花時間重新執行

```
y_train_split = np.load(os.path.join(load_dir, 'y_train_split.npy'))
y_test_split = np.load(os.path.join(load_dir, 'y_test_split.npy'))
X_train_split_normalized = np.memmap('/content/drive/MyDrive/224/X_train_split_normalized.npy',
                          dtype='float32', mode='r', shape=(12316, 224, 224, 3))
X_test_split_normalized = np.memmap('/content/drive/MyDrive/224/X_test_split_normalized.npy',
                          dtype='float32', mode='r', shape=(3080, 224, 224, 3))
```

Step3 one-hot 編碼

```
n_classes = len(np.unique(y_train))
n_classes
```

```
8
```

→ 資料有8個類別

```
n_classes = 8
if y_train_split.shape[-1] != n_classes or len(y_train_split.shape) > 2:
    print("修正 y_train_split 形狀...")
    y_train_split = to_categorical(y_train_split, n_classes)
if y_test_split.shape[-1] != n_classes or len(y_test_split.shape) > 2:
    print("修正 y_test_split 形狀...")
    y_test_split = to_categorical(y_test_split, n_classes)
```

→ One-hot encoding

為之後的模型做準備

02

Step1 批次輸入資料

```
#  創建  tf.data.Dataset
batch_size  =  16
#  創建  tf.data.Dataset  使用  output_signature
train_dataset  =  tf.data.Dataset.from_generator(
        lambda:  data_generator(X_train_split_normalized,  y_train_split,  batch_size),
        output_signature=(
                tf.TensorSpec(shape=(None,  224,  224,  3),  dtype=tf.float32),    #  資料形狀和型別
                tf.TensorSpec(shape=(None,  n_classes),  dtype=tf.float32)            #  標籤形狀和型別
        )
).prefetch(tf.data.AUTOTUNE)

test_dataset  =  tf.data.Dataset.from_generator(
        lambda:  data_generator(X_test_split_normalized,  y_test_split,  batch_size),
        output_signature=(
                tf.TensorSpec(shape=(None,  224,  224,  3),  dtype=tf.float32),
                tf.TensorSpec(shape=(None,  n_classes),  dtype=tf.float32)
        )
).prefetch(tf.data.AUTOTUNE)
```

目的是構建高效的數據輸入管道，將正規化的圖像數據和標籤轉換為 tf.data.Dataset 格式，以便在 TensorFlow 中進行模型訓練和測試。它通過明確的數據形狀定義、分批處理和預取優化，確保數據能夠高效、穩定地傳遞到模型中，特別適合圖像分類等需要處理大量數據的任務。

Step1 批次輸入資料

```
#  創建  tf.data.Dataset
batch_size  =  16 ──→ 一次處理16筆
#  創建  tf.data.Dataset  使用  output_signature
train_dataset  =  tf.data.Dataset.from_generator(
        lambda:  data_generator(X_train_split_normalized,  y_train_split,  batch_size),  ──→ 下一頁解釋
        output_signature=(
                tf.TensorSpec(shape=(None,  224,  224,  3),  dtype=tf.float32),    #  資料形狀和型別
                tf.TensorSpec(shape=(None,  n_classes),  dtype=tf.float32)            #  標籤形狀和型別
        )
).prefetch(tf.data.AUTOTUNE)

test_dataset  =  tf.data.Dataset.from_generator(
        lambda:  data_generator(X_test_split_normalized,  y_test_split,  batch_size),
        output_signature=(
                tf.TensorSpec(shape=(None,  224,  224,  3),  dtype=tf.float32),
                tf.TensorSpec(shape=(None,  n_classes),  dtype=tf.float32)
        )
).prefetch(tf.data.AUTOTUNE)
```

之後模型訓練時:

```
model.fit(train_dataset,  epochs=1,  verbose=1,  validation_data=test_dataset)
```

Step1 批次輸入資料

```
#  定義生成器函數，從  memmap  逐批讀取資料
def  data_generator(memmap_data,  labels,  batch_size):
        n_samples  =  memmap_data.shape[0]    #  直接從  memmap_data  獲取形狀
        for  i  in  range(0,  n_samples,  batch_size):
                batch_data  =  memmap_data[i:i+batch_size].astype('float32')
                batch_labels  =  labels[i:i+batch_size]
                yield  batch_data,  batch_labels
```

data_generator()用於從記憶體映射數據 (memmap_data) 和對應標籤 (labels) 中按批次 (batch_size) 動態生成數據和標籤對。它的主要目的是將數據分批提供給 tf.data.Dataset，以便在 TensorFlow 中進行高效的模型訓練或測試。

Step2 混和精度

```
from tensorflow.keras.mixed_precision import Policy, set_global_policy, LossScaleOptimizer

policy = Policy('mixed_float16')
set_global_policy(policy)
```

通過利用 GPU（如 NVIDIA 的 Tensor Core）的硬體加速，減少記憶體使用並加快訓練速度，同時保持模型精度。特別適合大模型或大批量數據。

Step3 Early Stopping

```
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
lr_scheduler = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5, min_lr=1e-6)
epoch_logger = EpochLogger()
```

```
class EpochLogger(Callback):
    def on_epoch_end(self, epoch, logs=None):
        self.current_epoch = epoch + 1  # 記錄當前 epoch（從 1 開始計數）
    def on_train_end(self, logs=None):
        print(f"最終運行了 {self.current_epoch} 個 epoch")
```

監控驗證集損失 (val_loss)，如果損失在一定次數的 epoch（patience=10）內沒有改善，則提前終止訓練，以避免過擬合或浪費計算資源。並在訓練結束時輸出最終運行的 epoch 數量。

模型一

02

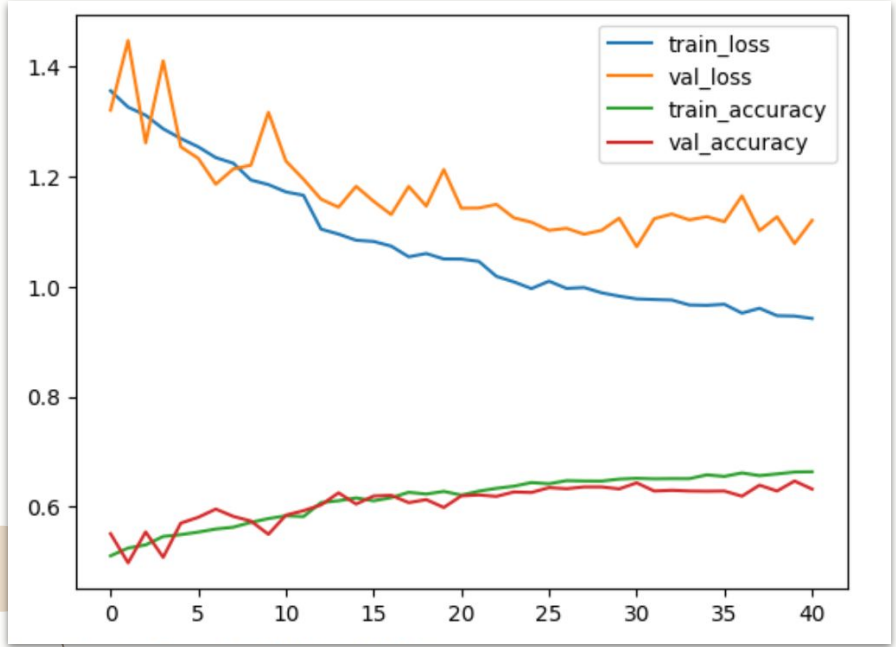
```
model = Sequential([
        # 資料增強層
        RandomFlip('horizontal', input_shape=(224, 224, 3)),
        RandomRotation(0.1),
        RandomZoom(0.1),

        # 卷積層
        Conv2D(32, kernel_size=(3, 3), activation='relu', padding='same'),
        MaxPooling2D(pool_size=(2, 2)),
        Conv2D(64, kernel_size=(3, 3), activation='relu', padding='same'),
        MaxPooling2D(pool_size=(2, 2)),
        Conv2D(128, kernel_size=(3, 3), activation='relu', padding='same'),
        MaxPooling2D(pool_size=(2, 2)),
        Dropout(0.3),

        # 全連接層
        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.5),

        # 輸出層
        Dense(n_classes, activation='softmax', dtype='float32')
])
```

增加一些讓圖片更多樣化的方法
看看會不會得到好的準確率

```
model.compile(loss='categorical_crossentropy',
                              optimizer=Adam(learning_rate=0.0001),
                              metrics=['accuracy'])
```

```
#  訓練模型
history  =  model.fit(train_dataset,  epochs=50,  validation_data=test_dataset,
                                          callbacks=[early_stopping,  lr_scheduler,  epoch_logger],  verbose=1)
```

```
770/770 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 67s 87ms/step - accuracy: 0.6040 - loss: 1.0885 - val_accuracy: 0.6393 - val_loss: 1.1018
Epoch 39/50
770/770 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 70s 90ms/step - accuracy: 0.6091 - loss: 1.0648 - val_accuracy: 0.6286 - val_loss: 1.1271
Epoch 40/50
770/770 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 69s 90ms/step - accuracy: 0.6164 - loss: 1.0648 - val_accuracy: 0.6468 - val_loss: 1.0783
Epoch 41/50
770/770 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 68s 89ms/step - accuracy: 0.6093 - loss: 1.0771 - val_accuracy: 0.6321 - val_loss: 1.1205
最終運行了 41 個 epoch → Early stopping發揮作用
```




```
Final train loss: 0.942183792591095
Final val loss: 1.120490312576294
Final train accuracy: 0.6636083126068115
Final val accuracy: 0.6321428418159485
```

模型一結論➡

1. Val loss有點太大了，模型的預測與真實標籤的差異較為顯著，需要更接近0
2. 準確率 63.21%，表現一般
3. 訓練後期，線條依舊很抖，並非正常現象

模型二

03

```
model = Sequential([
        # 增強資料增強層
        RandomFlip('horizontal', input_shape=(224, 224, 3)),
        RandomRotation(0.2),     # 提高旋轉幅度
        RandomZoom(0.2),         # 提高縮放幅度
        RandomContrast(0.2),  # 保持對比度增強
        RandomBrightness(0.2),  # 保持亮度增強

        # 卷積層 1
        Conv2D(32, kernel_size=(3, 3), activation='relu', padding='same'),
        MaxPooling2D(pool_size=(2, 2)),

        # 卷積層 2
        Conv2D(64, kernel_size=(3, 3), activation='relu', padding='same'),
        MaxPooling2D(pool_size=(2, 2)),

        # 卷積層 3
        Conv2D(128, kernel_size=(3, 3), activation='relu', padding='same'),
        MaxPooling2D(pool_size=(2, 2)),
        Dropout(0.3),     # 提高 Dropout 比率

        # 全連接層
        Flatten(),
        Dense(256, activation='relu'),
        Dropout(0.5),     # 提高 Dropout 比率
        Dense(128, activation='relu'),
        Dropout(0.5),

        # 輸出層
        Dense(8, activation='softmax', dtype='float32')
])
```

增加更多資料增強的程式碼
讓圖片更多樣化
讓模型學習更多種樣本

比模型一多了這一段
嘗試看看會不會效果更好

```
optimizer = AdamW(learning_rate=0.001, weight_decay=1e-4, clipnorm=1.0)
model.compile(loss='categorical_crossentropy',
              optimizer=optimizer,
              metrics=['accuracy'])
```

更改使用AdamW深度學習優化器

以下為Grok解釋Adam和AdamW的差異

- **Adam（Adaptive Moment Estimation）**：
  - Adam 是一種結合了動量法和 RMSProp 的優化器，通過計算梯度的一階動量（均值）和二階動量（方差的未中心化估計）來適應性地調整學習率。
  - 它使用以下公式更新參數：
    - 一階動量（均值）：$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$
    - 二階動量（方差）：$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
    - 偏差校正：$\hat{m}_t = m_t / (1 - \beta_1^t), \hat{v}_t = v_t / (1 - \beta_2^t)$
    - 參數更新：$\theta_t = \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$
  - Adam 的 L2 正則化（如果啟用）直接包含在損失函數中，通過在梯度更新中添加正則化項（$\lambda\theta$）實現。
- **AdamW（Adam with Weight Decay）**：
  - AdamW 是 Adam 的改進版本，專門針對權重衰減（weight decay）進行優化，它將正則化從損失函數中解耦，改為直接應用到參數更新中。
  - 參數更新公式為：
    - 動量和方差計算與 Adam 相同。
    - 參數更新：$\theta_t = \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon) - \alpha \cdot \lambda \theta_{t-1}$
    - 其中，λ 是權重衰減係數，α 是學習率，權重衰減直接應用於參數 θ。

```
optimizer = AdamW(learning_rate=0.001, weight_decay=1e-4, clipnorm=1.0)
model.compile(loss='categorical_crossentropy',
              optimizer=optimizer,
              metrics=['accuracy'])
```

為何我使用AdamW?
我想到老師在上課提及這個資料集容易過擬和

4. 何時選擇 Adam 或 AdamW

- **選擇 Adam**：
  - 當你的模型不需要強正則化（例如，數據量充足，過擬合風險低）。
  - 如果你已經在損失函數或層中顯式添加了 L2 正則化，且對當前設置滿意。
  - 適合快速實驗或對泛化性能要求不高的場景。
- **選擇 AdamW**：
  - 當你希望提高模型的泛化性能，特別是在測試集表現不佳或過擬合的情況。
  - 當數據集較小或模型較複雜，容易過擬合時，AdamW 的權重衰減能有效控制參數範數。
  - 適合大多數現代深度學習任務，特別是圖像分類（例如你的 224x224 RGB 圖像分類任務）和自然語言處理。

```
optimizer = AdamW(learning_rate=0.001, weight_decay=1e-4, clipnorm=1.0)
model.compile(loss='categorical_crossentropy',
              optimizer=optimizer,
              metrics=['accuracy'])
```

```
history = model.fit(train_dataset, epochs=60, validation_data=test_dataset,
                    callbacks=[early_stopping, lr_scheduler, epoch_logger], verbose=1)
```

```
770/770 ━━━━━━━━━━━━━━━━━━━━ 32s 41ms/step - accuracy: 0.2507 - loss: 1.9977 - val_accuracy: 0.2617 - val_loss: 1.9906
Epoch 39/60
770/770 ━━━━━━━━━━━━━━━━━━━━ 32s 41ms/step - accuracy: 0.2507 - loss: 1.9977 - val_accuracy: 0.2617 - val_loss: 1.9906
Epoch 40/60
770/770 ━━━━━━━━━━━━━━━━━━━━ 32s 41ms/step - accuracy: 0.2507 - loss: 1.9978 - val_accuracy: 0.2617 - val_loss: 1.9906
Epoch 41/60
770/770 ━━━━━━━━━━━━━━━━━━━━ 32s 41ms/step - accuracy: 0.2507 - loss: 1.9977 - val_accuracy: 0.2617 - val_loss: 1.9906
Epoch 42/60
770/770 ━━━━━━━━━━━━━━━━━━━━ 32s 41ms/step - accuracy: 0.2507 - loss: 1.9976 - val_accuracy: 0.2617 - val_loss: 1.9906
Epoch 43/60
770/770 ━━━━━━━━━━━━━━━━━━━━ 32s 41ms/step - accuracy: 0.2507 - loss: 1.9975 - val_accuracy: 0.2617 - val_loss: 1.9906
最終運行了 43 個 epoch → Early stopping
```

從Epoch 4 到 Epoch 43 完全沒有進步，停滯不前
這個模型更失敗了，學習率過低，可能是模型結構過簡單或梯度問題

如果一個模型訓練發生從第四個Epoch到第43個epoch，都顯示accuracy: 0.2507 - loss: 1.9975 - val_accuracy: 0.2617 - val_loss: 1.9906 - learning_rate: 3.1250e-05，數字完全沒有浮動，這象徵甚麼

當模型在訓練過程中，從第 4 個 epoch 到第 43 個 epoch 的訓練指標（`accuracy: 0.2507`，`loss: 1.9975`）和驗證指標（`val_accuracy: 0.2617`，`val_loss: 1.9906`）完全沒有任何浮動，且學習率（`learning_rate: 3.1250e-05`）保持不變，這是一個異常現象，通常表明模型訓練存在嚴重問題。以下是對這種現象的分析、可能原因以及解決方法：

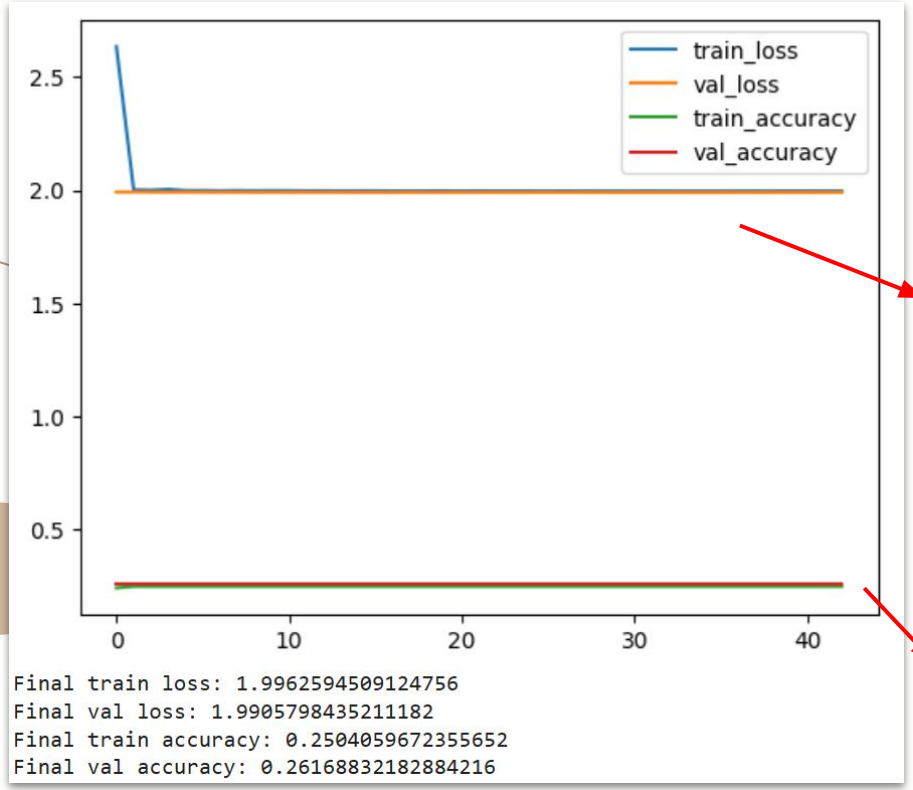
1. 現象的象徵

這種完全靜止的指標（損失和準確率在多個 epoch 不變）通常意味著模型的參數在訓練過程中幾乎沒有更新，導致模型無法學習數據中的模式。具體來說，這可能反映以下問題之一：

- **模型無法有效優化**：梯度更新無效，導致參數無法改變。
- **數據或輸入管道問題**：數據輸入可能存在錯誤，導致模型接收到無效或重複的數據。
- **學習率過低**：學習率過小，導致參數更新幅度太小，無法改變損失或準確率。
- **模型或損失函數設計問題**：模型結構、損失函數或優化器設置可能不適合當前任務。

(3) 模型或梯度問題

- **現象**：如果模型結構不適合任務（例如，過於簡單或初始化不當），或者梯度消失/爆炸，模型可能無法有效更新參數。
- **可能原因**：
  - **模型過於簡單**：如果你的模型是一個淺層網絡（例如，幾層簡單的 CNN），它可能無法捕捉 224x224 RGB 圖像的複雜特徵，導致訓練停滯。
  - **梯度消失**：如果網絡層數較深且激活函數（如 sigmoid）不當，可能導致梯度消失，參數無法更新。
  - **參數初始化問題**：如果權重初始化不當（例如，初始值過小或過大），可能導致梯度更新無效。
- **證據**：
  - 準確率固定在 0.2507，損失固定在 1.9975，表明模型的輸出分佈可能接近隨機猜測（例如，4 類分類問題的隨機準確率為 25%）。
  - 損失值 1.9975 對於交叉熵損失來說，接近於隨機預測的損失（例如，4 類問題的隨機預測損失約為 $\ln(4) \approx 1.386$, 但你的損失更高，可能表明模型輸出分佈不均勻或標籤有問題）。



模型二結論➡

1. 訓練損失=2
   是一個超級高數字
   代表這個模型非常不好
   訓練損失應愈接近0愈好

2. 準確率=26.16%非常低
   也表示這個模型表現超級差

04

🏆最終選擇🎉
模型三

ChatGPT建議我，由於
1.我的colab是免費版，有RAM的限制
2.之前處理這個Dataset有RAM爆掉的經驗
3.模型一和二表現都不好

➡推薦使用預訓練模型MobileNetV2

```
base_model = MobileNetV2(
        weights='imagenet',
        include_top=False,
        input_shape=(224, 224, 3)
)
base_model.trainable = False    # 凍結預訓練參數

model = Sequential([
        base_model,
        GlobalAveragePooling2D(),
        Dropout(0.3),
        Dense(128, activation='relu'),
        Dropout(0.5),
        Dense(n_classes, activation='softmax', dtype='float32')
])
```

MobileNetV2 是 Google 團隊於 2018 年提出的一種輕量級卷積神經網路（CNN）架構，主要針對移動端和嵌入式設備設計，強調高效能與低運算量，廣泛應用於影像分類、物件偵測、語意分割等任務 [2] [4] [6]。

主要特點

- **深度可分離卷積（Depthwise Separable Convolution）**
  延續 MobileNetV1 的設計，將傳統卷積拆分為 Depthwise 和 Pointwise 兩步，大幅減少參數量與運算量 [2] [4]。
- **Inverted Residual Block（反向殘差結構）**
  MobileNetV2 的核心創新是「反向殘差結構」：先用 1x1 卷積升維（通道數擴張），再用 3x3 深度卷積提取特徵，最後再用 1x1 卷積降維，並在 bottleneck（通道較少）層之間做 shortcut 連接。這種設計既保留了殘差結構提升準確率的優點，又兼顧速度與低運算量 [2] [4] [5] [6]。
- **Linear Bottleneck（線性瓶頸）**
  在 bottleneck 層去除非線性激活（如 ReLU），以避免特徵信息損失，進一步提升表達能力 [4] [6]。

實際應用與優勢

- 運算量與記憶體需求極低，非常適合終端設備部署。
- 在 ImageNet 等分類任務中，MobileNetV2 兼具高準確度與高速度，優於 MobileNetV1，且在運算量提升時可超越 NasNet 準確率 [2]。
- 廣泛用於物件偵測（如 SSDLite）、語意分割（如 Mobile DeepLabv3）等多種任務 [5] [6]。

小結

MobileNetV2 結合了深度可分離卷積、反向殘差結構與線性瓶頸，實現了低運算、高效能的目標，是目前移動端與嵌入式應用的主流選擇之一 [2] [4] [6]。

❗遇到狀況❗訓練時，train_loss和val_loss為nan

🔥 你 `train_loss: nan` 的關鍵原因是：你使用了 MobileNetV2，但輸入只正規化到 0~1，而不是它預期的 -1 到 1！

✅ 你目前的做法：

python 複製 編輯

```
X = X.astype('float32') / 255.0
```

會讓圖片值變成 `0~1`

❗ 但 `MobileNetV2` 預期的是：

python 複製 編輯

```
[-1.0, 1.0]
```

所以你必須使用：

python 複製 編輯

```
from tensorflow.keras.applications.mobilenet_v2 import preprocess_input

X_train_split_normalized = preprocess_input(X_train_split_normalized)
X_test_split_normalized = preprocess_input(X_test_split_normalized)
```

這個函式會把 0~255 的 RGB 值轉換為符合 MobileNetV2 要求的範圍（大約 -1 到 1）

```
X_train_split = X_train_split.astype('float32')
X_test_split = X_test_split.astype('float32')

X_train_split_normalized = preprocess_input(X_train_split)
X_test_split_normalized = preprocess_input(X_test_split)
```

新加入: 修正train_loss和val_loss為nan的錯誤

```
base_optimizer = Adam(learning_rate=0.001)
optimizer = LossScaleOptimizer(base_optimizer)

model.compile(
        optimizer=optimizer,
        loss='categorical_crossentropy',
        metrics=['accuracy']
)
```

改回Adam

```
history = model.fit(train_dataset, epochs=60, validation_data=test_dataset,
                    callbacks=[early_stopping, lr_scheduler, epoch_logger], verbose=1)
```

```
247/247 ━━━━━━━━━━━━━━━━━━━━ 8s 33ms/step - accuracy: 0.9626 - loss: 0.1064 - val_accuracy: 0.9552 - val_loss: 0.1515
Epoch 15/60
247/247 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9625 - loss: 0.1090 - val_accuracy: 0.9545 - val_loss: 0.1526
Epoch 16/60
247/247 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9648 - loss: 0.1059 - val_accuracy: 0.9536 - val_loss: 0.1597
Epoch 17/60
247/247 ━━━━━━━━━━━━━━━━━━━━ 9s 35ms/step - accuracy: 0.9684 - loss: 0.0946 - val_accuracy: 0.9532 - val_loss: 0.1540
最終運行了 17 個 epoch
```

被early_stopping提前停止訓練

❓只訓練17個epochs是否正常

✅ 為什麼早停是好現象？

`EarlyStopping` 的目的就是：

> 避免模型過度訓練、浪費資源，並減少過擬合風險。

如果在第 17 epoch 之後，模型的 **驗證集（val）準確率或損失沒有進步**，就代表：

- 模型已經學到足夠的資訊
- 再繼續訓練只會記住雜訊（導致過擬合）

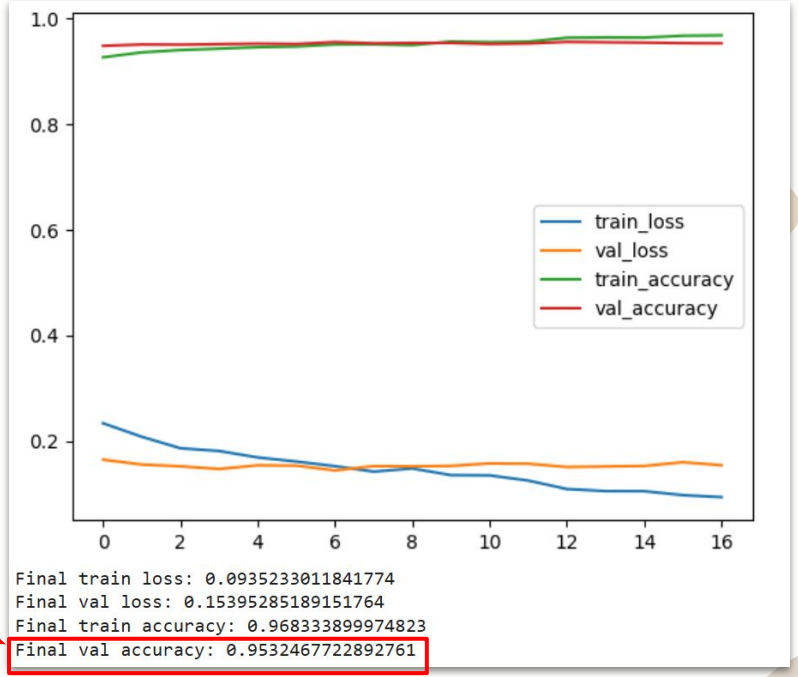
因此提早停止訓練是合理且有效的。

❓只訓練17個epochs是否正常

✅ 判斷 EarlyStopping 效果是否「夠好」的方法：

項目	判斷方式
val accuracy 高嗎？	若驗證準確率超過 85%、90% 或達到你的任務標準，就不用硬追求更多 epoch
train vs val 差異大嗎？	若訓練 accuracy 99%、val accuracy 70%，代表過擬合（就算訓練再久也無法改善）
loss 或 accuracy 曲線平穩了嗎？	若兩者已趨於收斂，代表訓練已完成

Ans:高



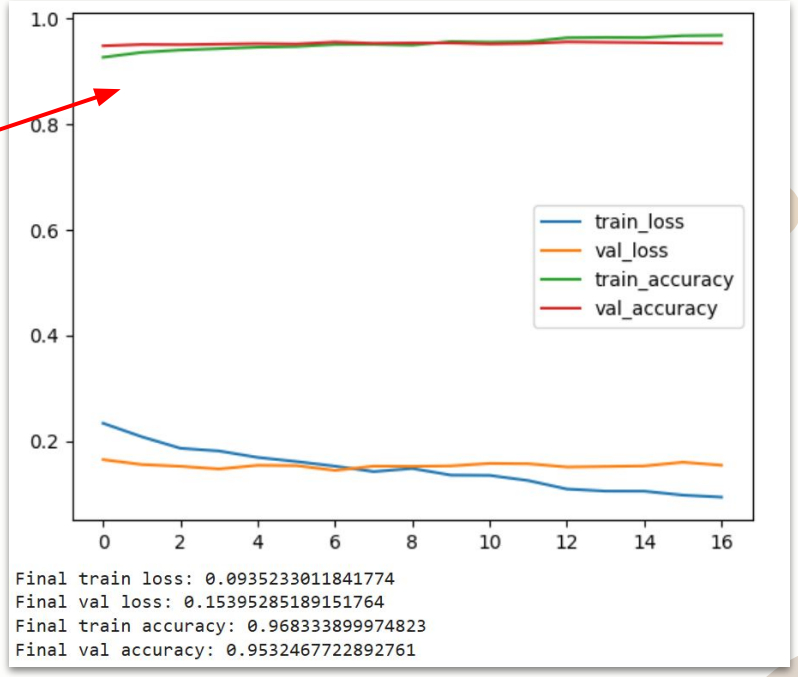
❓只訓練17個epochs是否正常

train_accuracy和val_accuracy相近

✅ **判斷 EarlyStopping 效果是否「夠好」的方法：**

項目	判斷方式
val accuracy 高嗎？	若驗證準確率超過 85%、90% 或達到你的任務標準，就不用硬追求更多 epoch
train vs val 差異大嗎？	若訓練 accuracy 99%、val accuracy 70%，代表過擬合（就算訓練再久也無法改善）
loss 或 accuracy 曲線平穩了嗎？	若兩者已趨於收斂，代表訓練已完成

Ans:相差小，代表沒有過擬和

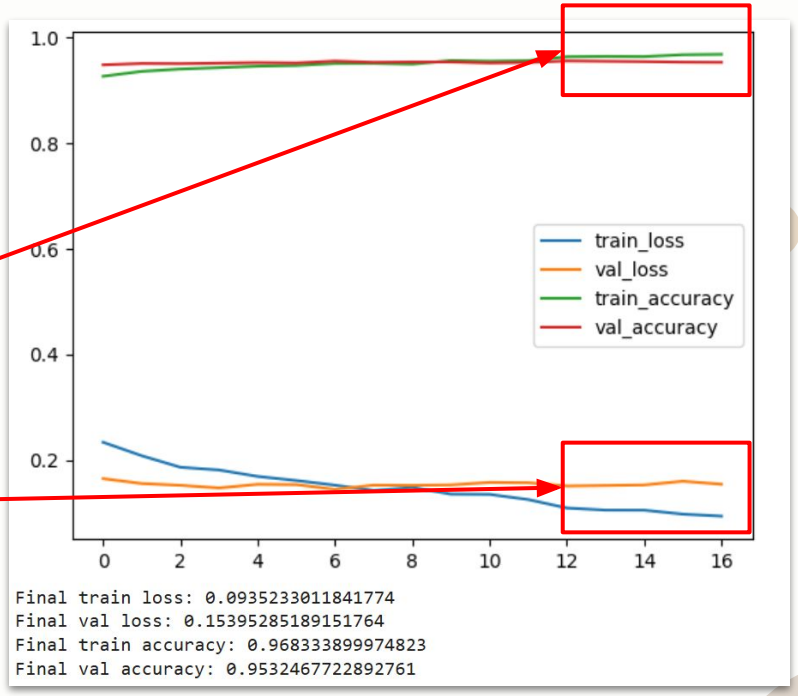


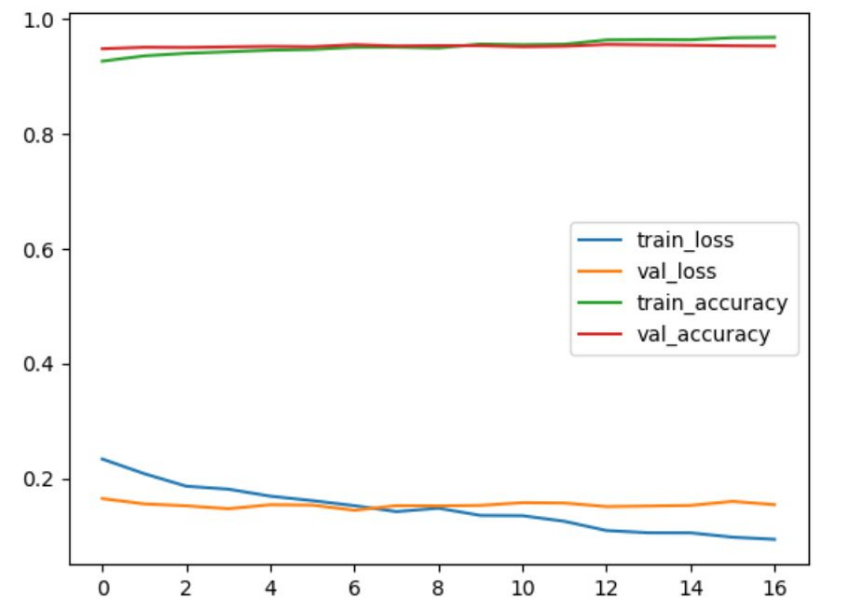
❓只訓練17個epochs是否正常

✅ 判斷 EarlyStopping 效果是否「夠好」的方法：

項目	判斷方式
val accuracy 高嗎？	若驗證準確率超過 85%、90% 或達到你的任務標準，就不用硬追求更多 epoch
train vs val 差異大嗎？	若訓練 accuracy 99%、val accuracy 70%，代表過擬合（就算訓練再久也無法改善）
loss 或 accuracy 曲線平穩了嗎？	若兩者已趨於收斂，代表訓練已完成

Ans:平穩





模型三結論➡

1. 訓練損失接近0
   符合我的期待
   這是一個不錯的模型

2. 準確率=95.32%
   好像挺高的
   所以我認為這個模型很適合
   此次作業

視覺化預測效果

```
def show_images_labels_predictions(images, labels, predictions, start_id, num=10):
    plt.gcf().set_size_inches(12, 14)
    if num > 25: num = 25
    for i in range(0, num):
        ax = plt.subplot(5, 5, 1+i)
        # 顯示彩色圖片
        ax.imshow(images[start_id])

        # 有 AI 預測結果資料，才在標題顯示預測結果
        if (len(predictions) > 0):
            title = 'ai = ' + str(predictions[start_id])
            # 預測正確顯示(o)，錯誤顯示(x)
            title += ('(o)' if predictions[start_id] == labels[start_id] else '(x)')
            title += '\nlabel = ' + str(labels[start_id])
        # 沒有 AI 預測結果資料，只在標題顯示真實數值
        else:
            title = 'label = ' + str(labels[start_id])

        # X, Y 軸不顯示刻度
        ax.set_title(title, fontsize=12)
        ax.set_xticks([])
        ax.set_yticks([])
        start_id += 1
    plt.show()
prediction = np.argmax(model.predict(X_test_split_normalized), axis = 1)
X_test_split = X_test_split[:,:,:,[2,1,0]]
show_images_labels_predictions(X_test_split, y_test_split, prediction, 0)
```

```
97/97 ━━━━━━━━━━━━━━━━━━━━ 2s 18ms/step
```