

DLBOI Homework 4

國立清華大學 109003807 跨院博士學程 四年級 鄒長志

這次作業是 Lab 6: Build a CNN，主要是練習利用 torchvision.models 函式庫選擇 2 種預先訓練好的電腦視覺模型，並透過之前作業用到的 X-ray dataset 進行 transfer learning，來比較兩者模型表現上的不同，以及跟之前 lab 4 單純建立並訓練 CNN from scratch 的結果差異性。

1.1 Task A: Model Selection (20%)

In this task, you will explore various pre-trained models available in torchvision.models, select two models, and justify their choices. The focus will be on understanding the architecture, their pre-trained performances, and predicting their adaptability to new tasks via transfer learning.

Model Choice (5%): List the two pre-trained models you have selected for the transfer learning tasks.

Explanation (15%): The explanation should encompass a range of aspects including, but not limited to, the complexity of the architecture, the performance of the models when pretrained, and their computation time for the transfer learning tasks.

1.1 Task A Answer

第一個任務是利用 PyTorch 的 torchvision.models 函式庫，挑選兩個模型，然後解釋為什麼選這兩個模型。

torchvision.models subpackage

Torchvision 是一個專門處理電腦視覺任務的 PyTorch 函式庫，裡面提供了不同用途的模型 (Models) 和其預訓練權重 (pre-trained weights)。

Docs > Models and pre-trained weights



Models and pre-trained weights

The `torchvision.models` subpackage contains definitions of models for addressing different tasks, including: image classification, pixelwise semantic segmentation, object detection, instance segmentation, person keypoint detection, video classification, and optical flow.

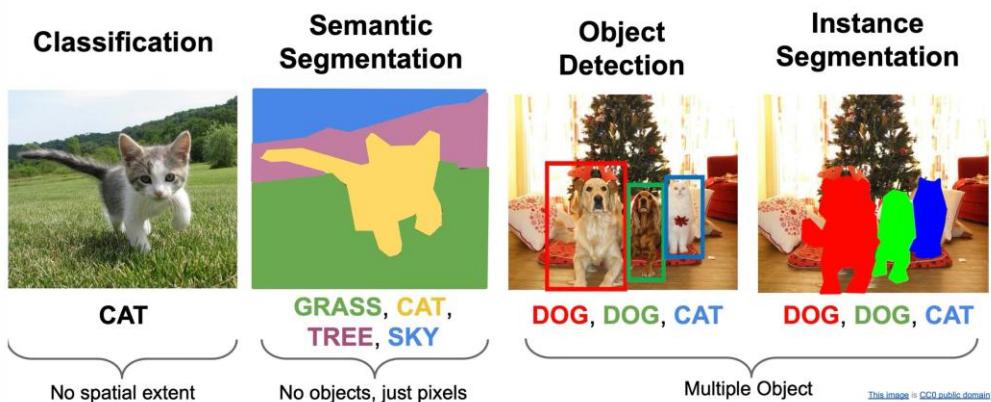
General information on pre-trained weights

TorchVision offers pre-trained weights for every provided architecture, using the PyTorch `torch.hub`. Instancing a pre-trained model will download its weights to a cache directory. This directory can be set using the `TORCH_HOME` environment variable. See `torch.hub.load_state_dict_from_url()` for details.

多種 torchvision 模型依照用途分類：

- **圖像分類 (image classification)**：識別圖像中的主要對象（例如：這是一隻貓）。**這也是我們做作業會用到的類別**
- **像素級語義分割 (pixelwise semantic segmentation)**：為圖像中的每個像素標註其所屬的類別（例如：哪些像素屬於天空，哪些屬於樹木）。
- **目標檢測 (object detection)**：在圖像中找出特定對象並用邊界框標示出來（例如：標出圖像中所有汽車的位置）。
- **實例分割 (instance segmentation)**：結合目標檢測和語義分割，為每個檢測到的對象實例進行像素級的分割（例如：區分圖像中的兩隻不同的貓）。
- **人物關鍵點檢測 (person keypoint detection)**：找出圖像中人物的身體關節位置（例如：肩、肘、膝蓋等）。
- **影片分類 (video classification)**：識別影片的內容類別。
- **光流 (optical flow)**：計算影片中不同幀之間像素的運動情況。

Fundamental Image-Level Vision Tasks



HW GROUP Reference: http://cs231n.stanford.edu/slides/2022/lecture_9_jiajun.pdf

3

預訓練權重的資訊：

預訓練權重是模型在大型資料集（如 ImageNet）上訓練後得到的參數。使用這些權重可以從一個已經「學會」基礎特徵的模型開始，而不用從零開始訓練，大大加快在新任務上的訓練速度和效果。

Classification 和 Quantized models 的區別

接著要看 `torchvision.models` subpackage 中 for classification 總共有哪些 models and weights。這邊要注意有兩個表格，分別是 Classification 的標準模型和 Quantized models (量化模型)，標準模型的權重 (weights) 通常使用 FP32 (32

位元浮點數) 來儲存，準確度最高，是學術研究和模型訓練時的標準。

Quantized models (量化模型)的權重被量化成了 INT8 (8 位元整數)，準確度會比 FP32 版本下降。我們要選的是 Classification 的標準模型。

List of all available classification models and weights

這張表格列出了 torchvision.models subpackage 中 for classification 的所有標準模型和 weights。

Table of all available classification weights

Accuracies are reported on ImageNet-1K using single crops:

Weight	Acc@1	Acc@5	Params	GFLOPS	Recipe
AlexNet_Weights.IMAGENET1K_V1	56.522	79.066	61.1M	0.71	link
ConvNeXt_Base_Weights.IMAGENET1K_V1	84.062	96.87	88.6M	15.36	link
ConvNeXt_Large_Weights.IMAGENET1K_V1	84.414	96.976	197.8M	34.36	link
ConvNeXt_Small_Weights.IMAGENET1K_V1	83.616	96.65	50.2M	8.68	link
ConvNeXt_Tiny_Weights.IMAGENET1K_V1	82.52	96.146	28.6M	4.46	link
DenseNet121_Weights.IMAGENET1K_V1	74.434	91.972	8.0M	2.83	link
DenseNet161_Weights.IMAGENET1K_V1	77.138	93.56	28.7M	7.73	link
DenseNet169_Weights.IMAGENET1K_V1	75.6	92.806	14.1M	3.36	link
DenseNet201_Weights.IMAGENET1K_V1	76.896	93.37	20.0M	4.29	link

- Weight 的名稱含意是模型名稱+weight 訓練於哪個資料庫，例如 AlexNet_Weights.IMAGENET1K_V1 代表模型名稱是 AlexNet，weight 訓練於 IMAGENET1K 資料庫。
- ImageNet-1K 是指 ImageNet 大型視覺辨識挑戰 (ILSVRC) 中所使用的標準資料集。它包含：1,000 個物體類別，約 128 萬張圖片的訓練集，5 萬張圖片驗證集，10 萬張圖片測試集。
- 所有的模型都是透過 ImageNet-1K 訓練，因為 ImageNet 測試集中的圖片大小不一，而且通常比 224x224 大，所以所有的測試圖片都會經過 Single crop，的意思是：先把圖片等比例縮小（例如，縮到 256x256），然後只從圖片的「正中央」裁切 (crop) 出一張 224x224 的圖片，模型只根據這一張中央裁切的圖片來進行預測，代表這是一個公平、標準化的評估基準，沒有使用那些會讓分數膨脹的複雜技巧。
- 有三種衡量模型的指標，分別是 Acc@1 (Top-1 準確率) 或 Acc@5 (Top-5 準確率)，Params (參數) ，GFLOPS (Giga Floating-point Operations)。
- Acc@1 (Top-1 準確率) 或 Acc@5 (Top-5 準確率): 代表模型的預訓練效能 (Performance when pre-trained)。通常會以 Acc@1 (Top-1 準確率)作為評估模型「預訓練效能」的最佳指標。模型會為 1000 個類別都輸出一個

機率，Acc@1 是指模型預測機率最高的「那一個」類別，剛好就是正確答案的機率，Acc@5 是指模型預測機率「前五高」的類別中，有包含正確答案的機率。Acc@5 總會高於或等於 Acc@1。

- **Params (參數)**: 代表架構的複雜性 (Complexity of the architecture)。這是衡量模型大小和複雜性最直接的指標。「Params」代表模型的參數數量（例如 $M = \text{百萬}$ ）。所以也可以說，Params (Parameters) 是衡量模型「大小」或「複雜度」的指標，Params 越高，模型越吃資源，尤其是 GPU 記憶體 (VRAM)。因為載入一個模型時，所有的參數（權重）都必須被載入到 VRAM 中。如果模型太大而 VRAM 不足（例如 Colab K80 只有 12GB），可能連載入模型都會失敗，更不用說訓練了。
- **GFLOPS (Giga Floating-point Operations)**: 代表計算時間 (Computation time)，也就是模型進行一次預測所需的計算量。GFLOPS 越高，代表模型越耗費計算資源，在學習時訓練和預測所需的時間就越長。如果說 Params 決定了模型佔用多少空間 (VRAM)，那麼 GFLOPS 就決定了模型跑起來有多耗時 (Speed)。GFLOPS 越高，代表模型做一次預測（或訓練一次）所需的數學運算量越大。這會直接影響訓練速度。GFLOPS 高的模型，在 Colab 上訓練一個 Epoch (一輪) 所需的時間就會顯著更長。
- 因此，由上述的討論可以知道，理論上三者之間的關係是: **Params (參數) 越大，模型「大小」或「複雜度」越大 → 理論上 GFLOPS 越大，代表所需的數學運算量越大，所需時間更長 → 理論上 Acc@1 (Top-1 準確率) 或 Acc@5 (Top-5 準確率) 越高**。但是，考量到 colab 系統環境的限制，因此可於無法選擇過大的模型。
- 所以我選擇兩種模型，以方便做為比較，其一是選擇較小的模型，其二是選擇準確度較高的模型。考量到 colab 系統環境的限制，挑選原則是盡量選擇 GFLOPS 較低（例如接近 1.0 或更低）的模型，避免受到 colab 系統算力與用量的限制。
- 第一個模型是

模型一	ShuffleNet_V2_X0_5_Weights.IMGNET1K_V1
選擇理由	1. GFLOPS 是所有模型最小 (0.04) 2. Parameter 是所有模型第二少(1.4M) 3. Acc@1 (Top-1 準確率) 或 Acc@5 (Top-5 準確率)尚可接受 (60.552) and (81.746) 4. 想要用這最小的模型，來看看 CNN 的 transfer learning 可以多快

	5. 跟較大的模型來做對比，看準確度還有時間會差多少				
● 第二個模型是	<table border="1"> <tr> <td>模型二</td> <td>EfficientNet_B2_Weights.IMGNET1K_V1</td> </tr> <tr> <td>選擇理由</td> <td> <ol style="list-style-type: none"> 1. 在 $\text{Acc}@1 > 80\%$ 的模型當中，GFLOPS 是所有模型最小 (1.09) 2. 在 $\text{Acc}@1 > 80\%$ 的模型當中，Parameter 是所有模型最少 (9.1M) 3. Acc@1 (Top-1 準確率) 或 Acc@5 (Top-5 準確率) 尚可接受 (80.608) and (95.31) 4. 想要用這相對較大的模型，來看看 CNN 的 transfer learning 可以多準確 5. 跟最小的模型來做對比，看準確度還有時間會差多少 </td> </tr> </table>	模型二	EfficientNet_B2_Weights.IMGNET1K_V1	選擇理由	<ol style="list-style-type: none"> 1. 在 $\text{Acc}@1 > 80\%$ 的模型當中，GFLOPS 是所有模型最小 (1.09) 2. 在 $\text{Acc}@1 > 80\%$ 的模型當中，Parameter 是所有模型最少 (9.1M) 3. Acc@1 (Top-1 準確率) 或 Acc@5 (Top-5 準確率) 尚可接受 (80.608) and (95.31) 4. 想要用這相對較大的模型，來看看 CNN 的 transfer learning 可以多準確 5. 跟最小的模型來做對比，看準確度還有時間會差多少
模型二	EfficientNet_B2_Weights.IMGNET1K_V1				
選擇理由	<ol style="list-style-type: none"> 1. 在 $\text{Acc}@1 > 80\%$ 的模型當中，GFLOPS 是所有模型最小 (1.09) 2. 在 $\text{Acc}@1 > 80\%$ 的模型當中，Parameter 是所有模型最少 (9.1M) 3. Acc@1 (Top-1 準確率) 或 Acc@5 (Top-5 準確率) 尚可接受 (80.608) and (95.31) 4. 想要用這相對較大的模型，來看看 CNN 的 transfer learning 可以多準確 5. 跟最小的模型來做對比，看準確度還有時間會差多少 				

1.2 Task B: Fine-tuning the ConvNet (30%)

You will fine-tune the selected pre-trained models on our 2-class x-ray dataset. They need to modify the models to fit the new dataset and analyze the effectiveness of fine-tuning, considering the architecture and depth of the networks.

Discussion (30%, 15% for each): Analyze the performances of the fine-tuned models. Include a comparative evaluation, focusing on how effectively fine-tuning facilitated their adaptation to the new task.

1.2 Task B Answer

Task A 是挑選模型 (ShuffleNet V2 和 EfficientNet B2)，Task B 則是要實際使用這兩個模型，並比較它們的表現。核心任務是做 Fine-tuning，再討論與分析。

- Fine-tuning (微調)：是「遷移學習 (Transfer Learning)」的核心步驟。
- Task B 基本上有三步驟，資料是上次作業原本的資料集 2-class x-ray dataset(判斷 X 光片是正常還是肺炎，這是一個 2 分類問題)。**第一步是要先 import**，在 Task A 選的模型 (ShuffleNet, EfficientNet) 是在 ImageNet (1000 個類別) 上預先訓練的。**第二步是修改(Modify)**，因為不能直接用，模型的「頭」(最後的分類層) 被設計為輸出 1000 個答案，所以需要砍掉原本的 1000 類別輸出，然後換上一個只輸出 2 個答案的新輸出。**第三步是 Fine-tuning**，用 X 光片的資料集來訓練這個「新模型」。記住這裡並沒

有凍結 parameter。

- 我們先來看看原先上次作業使用的 sequential model

```
model = nn.Sequential(  
    nn.Flatten(),  
    nn.Linear(256 * 256 * 1, 64),  
    nn.BatchNorm1d(64),  
    nn.ReLU(),  
    nn.Dropout(0.5),  
  
    nn.Linear(64, 64),  
    nn.BatchNorm1d(64),  
    nn.ReLU(),  
    nn.Dropout(0.5),  
  
    nn.Linear(64, 64),  
    nn.BatchNorm1d(64),  
    nn.ReLU(),  
    nn.Dropout(0.5),  
  
    nn.Linear(64, 1),  
    nn.Sigmoid()  
)  
  
model.to(device)  
print(model)
```

```
Sequential(  
(...)  
    (0): Flatten(start_dim=1, end_dim=-1)  
    (1): Linear(in_features=65536, out_features=64, bias=True)  
    (2): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (3): ReLU()  
    (4): Dropout(p=0.5, inplace=False)  
    (5): Linear(in_features=64, out_features=64, bias=True)  
    (6): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (7): ReLU()  
    (8): Dropout(p=0.5, inplace=False)  
    (9): Linear(in_features=64, out_features=64, bias=True)  
    (10): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (11): ReLU()  
    (12): Dropout(p=0.5, inplace=False)  
    (13): Linear(in_features=64, out_features=1, bias=True)  
    (14): Sigmoid()  
)
```

- 筆記：載入欲訓練模型的不同語法

model = models.ShuffleNet_V2_X0_5(pretrained=True)	是舊的寫法，但目前為了向後相容仍可使用，會載入預設的預訓練權重
model = models.ShuffleNet_V2_X0_5(weights=None)	用於不載入任何預訓練權重，僅建立模型的結構（從零開始訓練），隨機初始化權重。
<u>models.ShuffleNet_V2_X0_5(weights= ShuffleNet_V2_X0_5_Weights.IMAGENET 1K_V1)</u>	用於精確指定要載入的權重版本
models.ShuffleNet_V2_X0_5(weights=	載入該模型的最佳預設權重

ShuffleNet_V2_X0_5_Weights.DEFAULT	(通常等同於 IMAGENET1K_V1)
------------------------------------	--------------------------

- 筆記: 每種模型只會有一種 Call 最後一層的方法

對於 TorchVision 官方提供的單一模型系列（例如所有版本的 ResNet、所有版本的 VGG 等），它們會遵循一個統一的命名慣例來指代其最終分類層。這是為了讓使用者在進行遷移學習時，不必為每個模型系列都查閱其源碼。

存取方式	屬於哪一類模型？(TorchVision 慣例)
model.fc	這是單一線性層。 ResNet, ShuffleNet
model.head.0	現代 Transformer 模型。head 模塊是一個 Sequential 容器。 Vision Transformer (ViT)
model.classifier[-1]	classifier 是一個 nn.Sequential 容器，[-1] 存取其最後一個元素。 VGG, AlexNet, DenseNet, SqueezeNet, EfficientNet
都不適用	任何自定義或非標準模型，或者模型設計者選擇了完全不同的命名。

- 在 PyTorch 的設計哲學中，沒有一個「絕對命名，沒有例外」的屬性可以通用於所有模型，來指代某一層。PyTorch 賦予了設計者極大的自由度來定義模型結構和命名模塊。**模型多樣性**：不同的模型架構（如 ResNet, Transformer, GANs 等）有著截然不同的尾部結構。有些模型的分類頭是單一 FC 層，有些是多個 Conv 層，有些是多個 Block。**模塊容器**：PyTorch 提供了 nn.Sequential、nn.ModuleList、nn.ModuleDict 等容器。這些容器內的層是通過索引而非命名屬性來訪問的。命名慣例的範圍：像 model.fc 這樣的命名只是 TorchVision 為了方便遷移學習而建立的特定慣例，只適用於其官方提供的圖像分類模型。
- 注意問題：**損失函數 (Loss Function) 與模型輸出不匹配**。我們在修改最後一層時，會改成 linear function (model.fc = nn.Linear(num_ftrs, 2))，它會輸出兩個原始的數值，但原始的損失函數是 criterion = **nn.BCELoss()**。
nn.BCELoss (二元交叉熵) 預期模型的輸出是單一的機率值（介於 0 和 1 之間，通常是經過 Sigmoid 函數）。它無法處理來自 Linear(..., 2) 的兩個輸出。如果標籤 (label) 是 0 或 1，改用 **nn.CrossEntropyLoss()**。這是 PyTorch 中用於分類問題最標準的損失函數。它會自動在內部處理 Softmax 並接受 Linear(..., 2) 的原始 logits 輸出。也可以用另外一種處理方式，使用 **nn.BCEWithLogitsLoss()**，它在功能上等同於 **Sigmoid + nn.BCELoss()**。

BCELoss，這樣就不用更改 BCELoss 相關的計算方式。它就是設計來接收 `nn.Linear(..., 1)` 所輸出的原始 logits，且數值計算上更穩定。

```
# hyperparameter
lr = 0.01
weight_decay = 0.001
# 修正：只將 fc 層的參數傳遞給 optimizer
optimizer = optims.Adam(model.fc.parameters(), lr=lr, weight_decay = weight_decay)
#optimizer = optims.Adam(model.parameters(), lr=lr, weight_decay = weight_decay)
lr_scheduler = optims.lr_scheduler.ReduceLROnPlateau(optimizer, factor=0.1, patience=5,
epochs = 10
#criterion = nn.BCELoss()
#criterion = nn.CrossEntropyLoss()
# 這是專門用來處理 "logits" 的 BCELoss
criterion = nn.BCEWithLogitsLoss()

# save checkpoint
save = 'model'
```

- 注意問題：輸入影像的通道數 (Channels) 不符。我們原本使用的是 1-channel (單通道，例如灰階) 影像。但是包括 ShuffleNet_V2_X0_5 以及幾乎所有在 ImageNet 上預訓練的模型都是使用 3-channel (RGB) 影像訓練的。因此模型的第一層 (model.conv1) 期望收到 3 個輸入通道。當您傳入 1 個通道的資料時，程式會立刻拋出 RuntimeError (維度不匹配)。我們不應該修改模型的第一層 (這樣會失去預訓練的意義)，而應該在您的資料預處理 (transforms) 步驟中，將單通道影像轉換為三通道。

```
# define transformation
train_transform = transforms.Compose([
    transforms.Resize(256, 256), interpolation=InterpolationMode.BICUBIC),
    #transforms.Grayscale(num_output_channels=1),
    transforms.Grayscale(num_output_channels=3), # 轉換為 3 通道
    transforms.ToTensor(),
])

common_transform = transforms.Compose([
    transforms.Resize(256, 256), interpolation=InterpolationMode.BICUBIC),
    #transforms.Grayscale(num_output_channels=1),
    transforms.Grayscale(num_output_channels=3), # 將灰階影像複製成3通道
    transforms.ToTensor(),
])
```

- 注意問題：最佳化器 (Optimizer) 的參數設定，取決於訓練的目的。
`optimizer = optims.Adam(model.parameters(), ...)` 會將 `model.parameters()` (模型的所有參數) 都傳給了 Adam。雖然您在之後設定了 `requires_grad = False`，但更標準且有效率的做法是只將您要訓練的參數傳給最佳化器。如果只要訓練最後一層，可以直接使最佳化器只接收「解凍」的最後一層

(model.fc) 的參數。

```
# 修正：只將 fc 層的參數傳遞給 optimizer
optimizer = optims.Adam(model.fc.parameters(), lr=lr, weight_decay = weight_decay)
#optimizer = optims.Adam(model.parameters(), lr=lr, weight_decay = weight_decay)
```

- 注意問題：(所有 Loop 中) 準確率計算：閾值從 0.5 改為 0.0。因為原本模型輸出是 sigmoid function，模型現在輸出的是 preds (logits)。Sigmoid 函數會將 logits 轉換為機率 ($\text{Sigmoid}(0.0) = 0.5$)。因此， $\text{logit} > 0.0$ 就等同於機率 > 0.5 。準確率計算是在 logits 上進行的，所以閾值應該是 0.0。

```
# Calculating Metrics
#predicts = (preds > 0.5).float()
predicts = (preds > 0.0).float()
predicts = predicts.view(-1)
predicts = predicts.detach().cpu().numpy()
labels = labels.detach().cpu().numpy()
acc = accuracy_score(labels, predicts)
```

模型一：ShuffleNet V2_X0_5_Weights.IMAGENET1K_V1

- 第一步是要先 import，在 Task A 選的模型 (ShuffleNet) 是在 ImageNet (1000 個類別) 上預先訓練的，載入模型+預訓練的權重

帶入模型1

(ShuffleNet_V2_X0_5_Weights.IMAGENET1K_V1)

```
import torchvision.models as models
import torch.nn as nn

from torchvision.models import ShuffleNet_V2_X0_5_Weights
model = models.shufflenet_v2_x0_5(weights=ShuffleNet_V2_X0_5_Weights.IMAGENET1K_V1)
#print(model)

...
ShuffleNetV2(
    (conv1): Sequential(
        (0): Conv2d(3, 24, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
    )
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    (stage2): Sequential(
        ...
        (conv5): Sequential(
            (0): Conv2d(192, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): ReLU(inplace=True)
        )
        (fc): Linear(in_features=1024, out_features=1000, bias=True)
    )
    'nmodel = models.ShuffleNet_V2_X0_5(pretrained=True)\nmodel\nnmodel = models.ShuffleNet_V2_X0_5(weights=None)\nmodel\n'
)
```

● 接著我們觀察模型的整體架構

```
from torchinfo import summary
import torch

# 假設的輸入張量形狀: (Batch Size, Channels, Height, Width)
# 您的模型中 nn.Flatten() 在 Linear(256 * 256 * 1, 64) 之前,
# 說明輸入資料在經過 Flatten 前可能是 (Batch Size, 1, 256, 256)
input_size = (64, 3, 256, 256)

# 執行 summary
summary(model, input_size=input_size)
# --- 修改這行 ---
# 我們要求它顯示 'mult_adds' (MAdds)
summary(model,
         input_size=input_size,
         col_names=["input_size", "output_size", "num_params", "mult_adds"], #
         verbose=1)
```

...

Layer (type:depth-idx)	Input Shape	Output Shape
ShuffleNetV2	[64, 3, 256, 256]	[64, 1000]
└ Sequential: 1-1	[64, 3, 256, 256]	[64, 24, 128, 128]
└ Conv2d: 2-1	[64, 3, 256, 256]	[64, 24, 128, 128]
└ BatchNorm2d: 2-2	[64, 24, 128, 128]	[64, 24, 128, 128]
└ ReLU: 2-3	[64, 24, 128, 128]	[64, 24, 128, 128]
└ MaxPool2d: 1-2	[64, 24, 128, 128]	[64, 24, 64, 64]
└ Sequential: 1-3	[64, 24, 64, 64]	[64, 48, 32, 32]
└ InvertedResidual: 2-4	[64, 24, 64, 64]	[64, 48, 32, 32]
└ Sequential: 2-1	[64, 24, 64, 64]	[64, 24, 64, 64]

Total params: 1,366,792
Trainable params: 1,366,792
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 3.36

Input size (MB): 50.33
Forward/backward pass size (MB): 1369.95
Params size (MB): 5.47
Estimated Total Size (MB): 1425.75

Total params: 1,366,792
Trainable params: 1,366,792
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 3.36

Input size (MB): 50.33
Forward/backward pass size (MB): 1369.95
Params size (MB): 5.47
Estimated Total Size (MB): 1425.75

GFLOPS (Giga Floating-point Operations):

「MAdds」(Multiply-Adds，乘法-加法 運算)，這個值約為 FLOPS 的一半，是目前通用的計算量指標。這個數字的單位通常是 G (Giga)，例如 1.2G。這代表模型進行一次前向傳播 (forward pass) 需要 1.2 Giga-MAdds ($1.2 * 10^9$ 次乘加運算)。

- 接著我們要修改結構，更換最後一層，使其輸出為 2 類。

```
[12] 0 秒
▶ # 2. 獲取最後一個全連接層 (fc 層) 的輸入特徵數量 (In Features)
# 這是 ShuffleNet V2 模型的標準做法
num_ftrs = model.fc.in_features

# 3. 替換最後一個全連接層 (Linear Layer)
# 將輸出類別數 (out_features) 設定為 2
model.fc = nn.Linear(num_ftrs, 2)
# 只需要 1 個輸出 (logit) for BCE loss
model.fc = nn.Linear(num_ftrs, 1)

# 可選：將模型移動到 GPU (如果您有設定 device 的話)
model.to(device)

# 4. 檢查修改後的模型結構
print(model.fc)
```

```
... Linear(in_features=1024, out_features=1, bias=True)
```

- 設定好需要訓練的權重，是否需要凍結，以及 loss 定義。

```
[14] 0 秒
▶ # hyperparameter
lr = 0.01
weight_decay = 0.001
optimizer = optims.Adam(model.parameters(), lr=lr, weight_decay = weight_decay)
lr_scheduler = optims.lr_scheduler.ReduceLROnPlateau(optimizer, factor=0.1, patience=5,
epochs = 10
#optimizer = optims.Adam(model.parameters(), lr=lr, weight_decay = weight_decay)
#criterion = nn.BCELoss()
#criterion = nn.CrossEntropyLoss()
# 這是專門用來處理 "logits" 的 BCELoss
criterion = nn.BCEWithLogitsLoss()

# save checkpoint
save = 'model'
```

- 在 X 光資料集上進行訓練 (fine-tuning)。

這裡要注意要把 `model.eval()` 註銷掉。

這幾乎是 PyTorch 遷移學習中最常遇到的陷阱。這一切都跟一個叫做 Batch Normalization (BN) 層的元件有關。可以把 `model.eval()` 想像成一個「總開關」，會改變模型中所有 BN 層的行為。

`1.model.train()` 模式 (訓練時)，BN 層會品嚐每一批送來的 (X-ray batch)，並即時調整(計算當前批次的 mean/std)。

結果：`(fc 層)` 學會了如何品嚐這種組合。他學得很好，`train_acc` 達到

100%。

2. `model.eval()` 模式 (驗證時)，呼叫了 `model.eval()`。這等於是命令 (BN 層)：嚴格按照那本舊的 (ImageNet 統計數據) 來調整，現在拿到 (X-ray)，卻用完全錯誤的 (ImageNet mean/std)。變成了無法辨識的「垃圾特徵」。(fc 層)完全無法理解。他只能亂猜，準確率就掉到了 50% (二元分類的亂猜機率)。

```
... Epoch 1/10 - loss: 0.2589 - train_acc: 94.44% - val_loss: 1.5142 - val_acc: 50.00% - time: 56.78s  
Epoch 2/10 - loss: 0.1284 - train_acc: 95.48% - val_loss: 1.6791 - val_acc: 50.00% - time: 56.31s  
Epoch 3/10 - loss: 0.0619 - train_acc: 97.84% - val_loss: 1.4130 - val_acc: 50.00% - time: 56.05s  
Epoch 4/10 - loss: 0.0579 - train_acc: 97.83% - val_loss: 1.2384 - val_acc: 50.00% - time: 56.31s  
Epoch 5/10 - loss: 0.0315 - train_acc: 99.09% - val_loss: 0.9351 - val_acc: 50.00% - time: 55.88s  
Epoch 6/10 - loss: 0.0214 - train_acc: 99.36% - val_loss: 0.8064 - val_acc: 50.00% - time: 59.33s  
Epoch 7/10 - loss: 0.0091 - train_acc: 99.95% - val_loss: 0.7344 - val_acc: 50.00% - time: 56.01s  
Epoch 8/10 - loss: 0.0055 - train_acc: 99.90% - val_loss: 0.7212 - val_acc: 50.00% - time: 55.69s  
Epoch 9/10 - loss: 0.0029 - train_acc: 100.00% - val_loss: 0.7225 - val_acc: 50.00% - time: 55.88s  
Epoch 10/10 - loss: 0.0022 - train_acc: 100.00% - val_loss: 0.7349 - val_acc: 50.00% - time: 56.69s
```

做修改 (註解掉 `#model.eval()`)，等於是告訴 (BN 層)：在驗證 (validation) 的時候，請繼續使用訓練時的行為，這樣一來，(fc 層) 在訓練和驗證時方式保持一致，他才能正確地評估 (`val_acc` 於是就上升了)。

總結：`model.eval()` 的關鍵作用是凍結 BN 層的統計數據，並強制使用它在預訓練時儲存的 `running_mean` 和 `running_var`。當您的新資料 (X 光片) 與舊資料 (ImageNet) 分佈差異太大時，這就成了一個災難。

為了獲得一致且有意義的測試結果，也應該將 `evaluate` 函數中的 `model.eval()` 註解掉。

```
▶ model_path = 'model.pth'  
avg_test_loss, avg_test_acc = evaluate(model, device, model_path, test_loader)  
  
... Model weights loaded successfully.  
Test Accuracy: 77.15%  
Test Loss: 0.7165
```

Model weights loaded successfully.

Test Accuracy: 77.15%

Test Loss: 0.7165

正確版本的數值：

```
# validation loop  
def val_one_epoch(model, device, criterion, val_data_loader, best_acc, save):  
    epoch_loss = []  
    epoch_acc = []  
    start_time = time.time()  
  
    #model.eval()
```

```

# evaluate loop (test loop)
def evaluate(model, device, model_path, test_loader):
    try:
        model.load_state_dict(torch.load(model_path))
        print("Model weights loaded successfully.")
    except Exception as e:
        print("Warning: Failed to load model weights. Using randomly initialized weights instead.")
        print(e)

    model.to(device)
    #model.eval()

[28] 9 分鐘
    lr_scheduler.step(val_acc)
...
    Epoch 1/10 - loss: 0.5869 - train_acc: 79.83% - val_loss: 0.3761 - val_acc: 87.50% - time: 57.03s
    Epoch 2/10 - loss: 0.2275 - train_acc: 93.35% - val_loss: 0.2046 - val_acc: 93.75% - time: 55.92s
    Epoch 3/10 - loss: 0.1217 - train_acc: 95.83% - val_loss: 0.0760 - val_acc: 100.00% - time: 56.09s
    Epoch 4/10 - loss: 0.0513 - train_acc: 97.91% - val_loss: 0.0414 - val_acc: 100.00% - time: 56.51s
    Epoch 5/10 - loss: 0.0372 - train_acc: 98.69% - val_loss: 0.1614 - val_acc: 93.75% - time: 55.89s
    Epoch 6/10 - loss: 0.0240 - train_acc: 99.11% - val_loss: 0.1463 - val_acc: 93.75% - time: 55.82s
    Epoch 7/10 - loss: 0.0183 - train_acc: 99.41% - val_loss: 0.0278 - val_acc: 100.00% - time: 56.47s
    Epoch 8/10 - loss: 0.0122 - train_acc: 99.65% - val_loss: 0.2948 - val_acc: 93.75% - time: 55.81s
    Epoch 9/10 - loss: 0.0074 - train_acc: 99.89% - val_loss: 0.0161 - val_acc: 100.00% - time: 55.83s
    Epoch 10/10 - loss: 0.0142 - train_acc: 99.61% - val_loss: 0.0054 - val_acc: 100.00% - time: 55.94s

```

Epoch 1/10 - loss: 0.5869 - train_acc: 79.83% - val_loss: 0.3761 - val_acc: 87.50% - time: 57.03s
 Epoch 2/10 - loss: 0.2275 - train_acc: 93.35% - val_loss: 0.2046 - val_acc: 93.75% - time: 55.92s
 Epoch 3/10 - loss: 0.1217 - train_acc: 95.83% - val_loss: 0.0760 - val_acc: 100.00% - time: 56.09s
 Epoch 4/10 - loss: 0.0513 - train_acc: 97.91% - val_loss: 0.0414 - val_acc: 100.00% - time: 56.51s
 Epoch 5/10 - loss: 0.0372 - train_acc: 98.69% - val_loss: 0.1614 - val_acc: 93.75% - time: 55.89s
 Epoch 6/10 - loss: 0.0240 - train_acc: 99.11% - val_loss: 0.1463 - val_acc: 93.75% - time: 55.82s
 Epoch 7/10 - loss: 0.0183 - train_acc: 99.41% - val_loss: 0.0278 - val_acc: 100.00% - time: 56.47s
 Epoch 8/10 - loss: 0.0122 - train_acc: 99.65% - val_loss: 0.2948 - val_acc: 93.75% - time: 55.81s
 Epoch 9/10 - loss: 0.0074 - train_acc: 99.89% - val_loss: 0.0161 - val_acc: 100.00% - time: 55.83s
 Epoch 10/10 - loss: 0.0142 - train_acc: 99.61% - val_loss: 0.0054 - val_acc: 100.00% - time: 55.94s

- 記錄最終的準確率。

```

    model_path = 'model.pth'
    avg_test_loss, avg_test_acc = evaluate(model, device, model_path, test_loader)
...
    Model weights loaded successfully.
    Test Accuracy: 73.26%
    Test Loss: 1.1689

```

Model weights loaded successfully.

Test Accuracy: 73.26%

Test Loss: 1.1689

- 問題: val acc (100%) 這麼高且不合理？驗證集 (validation set) 太小了。證據：val_acc 數字是 87.50%、93.75%、100.00%。val_dataset 總共只有 16 張圖片。模型 (train_dataset 非常大) 要「背誦」或「記住」這 16 張圖片的答案相對容易。val_acc 達到 100% 並不代表模型學會了「如

何看 X 光片」，只代表它過度擬合 (overfit) 了這 16 張特定的驗證圖片。

- 為什麼 val_acc 會大於 train_acc？這發生在 Epoch 1 (train_acc: 79.83% vs val_acc: 87.50%)。val_dataset 太小太簡單（只有 16 張圖）。模型在一個大訓練集上（2000 張圖）達到 79.83% 的平均準確率，但在一個 16 張圖的小考上拿到 87.5%（答對 14 張）是完全合理的。train_acc 是整個 epoch 的平均值。模型在 epoch 開始時很差，在 epoch 結束時較好。而 **val acc** 是在 epoch 結束後才測的（此時模型處於最佳狀態）。
- 為什麼 val_acc (100%) 和 test_acc (73%) 差這麼多？這是最關鍵的問題，它總結了上述所有情況：訓練迴圈 (for epoch in range(epochs)) 的目標是找到 **best acc (最佳驗證準確率)** 並儲存模型。由於驗證集只有 16 張圖，模型在第 3 個 epoch 就輕易達到了 100% 並儲存了 model.pth。這個「最佳模型」的唯一技能，就是完美地回答那 16 張驗證圖片。接著，執行 evaluate 函數。test_loader 會載入一個完全不同且規模大得多的測試集。
- 結論：val_acc (100%) 是誤導性的、無意義的數字，因為驗證集太小。test_acc (73.26%) 才是您模型真正的準確率。模型有「過度擬合」，它在訓練資料上表現完美 (train_acc: 99.61%)，但在從未見過的測試資料上表現不佳 (test_acc: 73.26%)。
- 根據 val_one_epoch 函數中的儲存邏輯，73.26% 的測試結果是來自 **Epoch 3 的權重**。模型儲存的邏輯（在 val_one_epoch 函數中）是模型檔案 (**model.pth**) 只會在當前的 val_acc 嚴格大於 (>) 過去的 best_acc 時，才會被「覆蓋」。

Epoch 1: val_acc: 87.50%。 (87.50 > 0.0)。best_acc 更新為 87.50。模型儲存 (Epoch 1 權重)。

Epoch 2: val_acc: 93.75%。 (93.75 > 87.50)。best_acc 更新為 93.75。模型覆蓋 (Epoch 2 權重)。

Epoch 3: val_acc: 100.00%。 (100.00 > 93.75)。best_acc 更新為 100.00。模型覆蓋 (Epoch 3 權重)。

Epoch 4: val_acc: 100.00%。 (100.00 並不大於 100.00)。if 條件為 False。模型未儲存。

Epoch 5: val_acc: 93.75%。 (93.75 並不大於 100.00)。if 條件為 False。模型未儲存。

Epoch 7: val_acc: 100.00%。 (100.00 並不大於 100.00)。if 條件為 False。模型未儲存。

Epoch 10: val_acc: 100.00%。 (100.00 並不大於 100.00)。if 條件為 False。模型未儲存。

結論：evaluate 函數載入了 model.pth，而這個檔案最後一次被更新是在 Epoch 3 結束時。雖然 Epoch 10 的模型在（那 16 張）驗證集上表現更好 (loss 僅 0.0054)，但因為儲存標準是 val_acc 嚴格遞增，所以程式並沒有儲存 Epoch 10 的權重。val_acc 達到 100% 是一個由「**驗證集太小**」所引起的假象。您

73.26% 的 Test Accuracy 才是對 Epoch 3 模型真實泛化能力的評估。

模型二：EfficientNet_B2_Weights.IMAGENET1K_V1

- 第一步是要先 import，在 Task A 選的模型是在 ImageNet (1000 個類別) 上預先訓練的，載入模型+預訓練的權重

模型2帶入模型2

(EfficientNet_B2_Weights.IMAGENET1K_V1)

```
[8] 0 秒
● import torchvision.models as models
    import torch.nn as nn

    from torchvision.models import EfficientNet_B2_Weights
    model = models.efficientnet_b2(weights=EfficientNet_B2_Weights.IMAGENET1K_V1)
    print(model)
```

... Downloading: "https://download.pytorch.org/models/efficientnet_b2_rwightman-c35c14" 100%|██████████| 35.2M/35.2M [00:00<00:00, 186MB/s]

```
EfficientNet(
    (features): Sequential(
        (0): Conv2dNormActivation(
            (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
            (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
        )
        (1): Sequential(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (classifier): Sequential(
                (0): Dropout(p=0.3, inplace=True)
                (1): Linear(in_features=1408, out_features=1000, bias=True)
            )
        )
    )
    \nmodel = models.ShuffleNet_V2_X0_5(pretrained=True)\nmodel\n\nmodel = models.ShuffleNet_V2_X0_5(weights=None)\nmodel\n\n
```

- 接著我們觀察模型的整體架構

```
summary(model,
        input_size=input_size,
        col_names=["input_size", "output_size", "num_params", "mult_adds"], # 加入 "mult_adds"
        verbose=1)
```

Layer (type:depth-idx)	Input Shape	Output Shape	Param #	Mul
EfficientNet	[64, 3, 256, 256]	[64, 1000]	--	--
└─Sequential: 1-1	[64, 3, 256, 256]	[64, 1408, 8, 8]	--	-
└─Conv2dNormActivation: 2-1	[64, 3, 256, 256]	[64, 32, 128, 128]	--	
└─Conv2d: 3-1	[64, 3, 256, 256]	[64, 32, 128, 128]	864	
└─BatchNorm2d: 3-2	[64, 32, 128, 128]	[64, 32, 128, 128]	64	
└─SiLU: 3-3	[64, 32, 128, 128]	[64, 32, 128, 128]	--	
└─Sequential: 2-2	[64, 32, 128, 128]	[64, 16, 128, 128]	--	
└─MBConv: 3-4	[64, 32, 128, 128]	[64, 16, 128, 128]	1,448	

```

...
    |   └── Sequential: 2-8
    |       |   └── MBConv: 3-25
    |       |   └── MBConv: 3-26
    |   └── Conv2dNormActivation: 2-9
    |       |   └── Conv2d: 3-27
    |       |   └── BatchNorm2d: 3-28
    |       |   └── SILU: 3-29
    |   └── AdaptiveAvgPool2d: 1-2
    |   └── Sequential: 1-3
    |       |   └── Dropout: 2-10
    |           |   └── Linear: 2-11
    ...
[64, 208, 8, 8] [64, 352, 8, 8] 164, 352, 8, 8] 846,900
[64, 208, 8, 8] [64, 352, 8, 8] [64, 352, 8, 8] 1,888,920
[64, 352, 8, 8] [64, 352, 8, 8] [64, 1408, 8, 8] --
[64, 352, 8, 8] [64, 1408, 8, 8] [64, 1408, 8, 8] 495,616
[64, 1408, 8, 8] [64, 1408, 8, 8] [64, 1408, 8, 8] 2,816
[64, 1408, 8, 8] [64, 1408, 8, 8] [64, 1408, 8, 8] --
[64, 1408, 1, 1] [64, 1000] [64, 1000] --
[64, 1408] [64, 1408] [64, 1408] --
[64, 1408] [64, 1000] [64, 1000] 1,409,000
=====

Total params: 9,109,994
Trainable params: 9,109,994
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 55.04
=====
Input size (MB): 50.33
Forward/backward pass size (MB): 13105.12
Params size (MB): 36.44
Estimated Total Size (MB): 13191.89
=====
```

Total params: 9,109,994

Trainable params: 9,109,994

Non-trainable params: 0

Total mult-adds (Units.GIGABYTES): 55.04

Input size (MB): 50.33

Forward/backward pass size (MB): 13105.12

Params size (MB): 36.44

Estimated Total Size (MB): 13191.89

- 接著我們要修改結構，更換最後一層，使其輸出為 2 類。

```

[12]
✓ 0 秒
▶ # 1. 存取 'classifier' 容器中的最後一個元素 (即 Linear 層) ↑ ↓ ⌂ ⌄ ⌅ ⌋
final_linear_layer = model.classifier[-1]
num_ftrs = final_linear_layer.in_features

# 2. 替換最終層 (從 1000 類換成 1 類)
# 直接替換 nn.Sequential 容器中的最後一個元素
model.classifier[-1] = nn.Linear(num_ftrs, 1)

# 3可選: 將模型移動到 GPU (如果您有設定 device 的話)
model.to(device)

# 4. 檢查修改後的模型結構
print(model.classifier[-1])
```

... Linear(in_features=1408, out_features=1, bias=True)

```
[1408]      Dropout_2_tv
... |    └─Linear: 2-11
1]           1,409
90,176
=====
Total params: 7,702,403
Trainable params: 7,702,403
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 54.95
=====
Input size (MB): 50.33
Forward/backward pass size (MB): 13104.60
Params size (MB): 30.81
Estimated Total Size (MB): 13185.75
=====
```

Total params: 7,702,403

Trainable params: 7,702,403

Non-trainable params: 0

Total mult-adds (Units.GIGABYTES): 54.95

Input size (MB): 50.33

Forward/backward pass size (MB): 13104.60

Params size (MB): 30.81

Estimated Total Size (MB): 13185.75

- 設定好需要訓練的權重，是否需要凍結，以及 loss 定義。

```
[14]
✓ 0 秒
▶ # hyperparameter
lr = 0.01
weight_decay = 0.001
optimizer = optims.Adam(model.parameters(), lr=lr, weight_decay = weight_decay)
lr_scheduler = optims.lr_scheduler.ReduceLROnPlateau(optimizer, factor=0.1, patience=5,
epochs = 10
#optimizer = optims.Adam(model.parameters(), lr=lr, weight_decay = weight_decay)
#criterion = nn.BCELoss()
#criterion = nn.CrossEntropyLoss()
# 這是專門用來處理 "logits" 的 BCELoss
criterion = nn.BCEWithLogitsLoss()

# save checkpoint
save = 'model'
```

- 在 X 光資料集上進行訓練 (fine-tuning)。

```
...  
 8  
 9     for epoch in range(epochs):  
---> 10         train_loss, train_acc, train_time = train_one_epoch(model, device,  
criterion, optimizer, train_loader)  
 11         val_loss, val_acc, val_time, best_acc = val_one_epoch(model, device,  
criterion, val_loader, best_acc, save)  
 12
```

23 frames

```
...  
 2815     _verify_batch_size(input.size())  
 2816  
-> 2817     return torch.batch_norm(  
 2818         input,  
 2819         weight,
```

OutOfMemoryError: CUDA out of memory. Tried to allocate 3.00 GiB. GPU 0 has a total capacity of 14.74 GiB of which 74.12 MiB is free. Process 5356 has 14.67 GiB memory in use. Of the allocated memory 14.42 GiB is allocated by PyTorch, and 126.48 MiB is reserved by PyTorch but unallocated. If reserved but unallocated memory is large try setting PYTORCH_CUDA_ALLOC_CONF=expandable_segments=True to avoid fragmentation. See documentation for Memory Management (<https://pytorch.org/docs/stable/notes/cuda.html#environment-variables>)

EfficientNet B2 過大，必須縮小 batch size 到 32 才能執行

```
[33] 10 分鐘  
Epoch 1/10 - loss: 0.1241 - train_acc: 94.99% - val_loss: 0.1450 - val_acc: 93.75% - time: 62.96s  
Epoch 2/10 - loss: 0.1251 - train_acc: 95.24% - val_loss: 0.3232 - val_acc: 75.00% - time: 61.86s  
Epoch 3/10 - loss: 0.1035 - train_acc: 96.38% - val_loss: 0.4407 - val_acc: 81.25% - time: 62.53s  
Epoch 4/10 - loss: 0.1189 - train_acc: 95.34% - val_loss: 0.1089 - val_acc: 100.00% - time: 62.71s  
Epoch 5/10 - loss: 0.1415 - train_acc: 94.74% - val_loss: 0.2936 - val_acc: 87.50% - time: 62.70s  
Epoch 6/10 - loss: 0.1292 - train_acc: 94.69% - val_loss: 0.3509 - val_acc: 87.50% - time: 61.46s  
Epoch 7/10 - loss: 0.1034 - train_acc: 96.38% - val_loss: 0.1727 - val_acc: 93.75% - time: 62.66s  
Epoch 8/10 - loss: 0.1142 - train_acc: 95.44% - val_loss: 0.1537 - val_acc: 93.75% - time: 62.61s  
Epoch 9/10 - loss: 0.1031 - train_acc: 96.58% - val_loss: 0.1207 - val_acc: 93.75% - time: 63.28s  
Epoch 10/10 - loss: 0.1067 - train_acc: 95.98% - val_loss: 0.2945 - val_acc: 93.75% - time: 62.51s
```

Epoch 1/10 - loss: 0.1241 - train_acc: 94.99% - val_loss: 0.1450 - val_acc: 93.75% - time: 62.96s

Epoch 2/10 - loss: 0.1251 - train_acc: 95.24% - val_loss: 0.3232 - val_acc: 75.00% - time: 61.86s

Epoch 3/10 - loss: 0.1035 - train_acc: 96.38% - val_loss: 0.4407 - val_acc: 81.25% - time: 62.53s

Epoch 4/10 - loss: 0.1189 - train_acc: 95.34% - val_loss: 0.1089 - val_acc: 100.00% - time: 62.71s

Epoch 5/10 - loss: 0.1415 - train_acc: 94.74% - val_loss: 0.2936 - val_acc: 87.50% - time: 62.70s

Epoch 6/10 - loss: 0.1292 - train_acc: 94.69% - val_loss: 0.3509 - val_acc: 87.50% - time: 61.46s

Epoch 7/10 - loss: 0.1034 - train_acc: 96.38% - val_loss: 0.1727 - val_acc: 93.75% - time: 62.66s

Epoch 8/10 - loss: 0.1142 - train_acc: 95.44% - val_loss: 0.1537 - val_acc: 93.75% - time: 62.61s

Epoch 9/10 - loss: 0.1031 - train_acc: 96.58% - val_loss: 0.1207 - val_acc: 93.75% - time: 63.28s

Epoch 10/10 - loss: 0.1067 - train_acc: 95.98% - val_loss: 0.2945 - val_acc: 93.75% - time: 62.51s

- 記錄最終的準確率和訓練時間。

▼ Evaluate

```
[34] ✓ 15 秒
    model_path = 'model.pth'
    avg_test_loss, avg_test_acc = evaluate(model, device, model_path, test_loader)

    Model weights loaded successfully.
    Test Accuracy: 52.19%
    Test Loss: 1.6434
```

Model weights loaded successfully.

Test Accuracy: 52.19%

Test Loss: 1.6434

 討論與分析：比較兩個模型（ShuffleNet V2 和 EfficientNet B2）在 X 光資料集上的表現。

Pre-trained model 數據

	Acc@1	Params	GFLOPS
ShuffleNet_V2_X0_5_Weights.IMAGENET1K_V1	60.552	1.4M	0.04
EfficientNet_B2_Weights.IMAGENET1K_V1	80.608	9.1M	1.09

Task B 在 X 光資料集上的表現

	ShuffleNet_V2_X0_5	EfficientNet_B2
Task A		
Pre-trained model (修改 output 前)		
Total params	1,366,792	9,109,994
Trainable params	1,366,792	9,109,994
Non-trainable params	0	0
Total mult-adds	3.36	55.04
Acc@1 on imagenet	60.552%	80.608%
Task B		
Pre-trained model (修改 output 後) (no freezing weight)		
Total params	342,817	7,702,403
Trainable params	342,817	7,702,403
Non-trainable params	0	0
Total mult-adds	3.30	54.95

Acc@1 on x-ray	73.26%	52.19%
Task C Pre-trained model (修改 output 後) (freezing weight)		
Total params	342,817	7,702,403
Trainable params	1,025	1,409
Non-trainable params	341,792	7,700,994
Total mult-adds	3.30	54.95
Acc@1 on x-ray	???	???

- 原本假設 for 比較性評估 (comparative evaluation): 結果 (Task B) 當初的分析 (Task A) 應該會有一致表現，也就是說，Task A 上，EfficientNet B2 (ImageNet 準確率 80.6%) 和 ShuffleNet V2 (ImageNet 準確率 60.5%) 相比，EfficientNet B2 預訓練時學到的特徵 (Params 9.1M, GFLOPS 1.09) 更複雜、更強大，因此應該能更有效地適應 (adaptation) 新的醫學影像任務。
- 反向結果: 然而，ShuffleNet V2(X-ray 準確率 73.26%) 在 transfer learning 上的訓練結果反而比 EfficientNet B2(X-ray 準確率 52.19%) 來的好。
- 反向結果可能的原因: 模型過於複雜。因為 EfficientNet_B2 太複雜了。EfficientNet_B2 有大約 920 萬個參數。ShuffleNet_V2_X0.5 只有大約 130 萬個參數。當 X-ray 資料集不夠大時，擁有 920 萬個參數的 EfficientNet 就像一個「記性太好的學生」，他不去理解公式，而是把練習題的「答案」全部背下來。一到考試 (Test Set)，題目稍微變形，他就完全不會寫。相反，ShuffleNet 參數較少，記性沒那麼好，它被迫要去學習「真正判斷肺炎的特徵」(比如肺部的紋理、浸潤情況)，所以它在考試中還能拿到 73 分。

1.3 Task C: ConvNet as Fixed Feature Extractor (30%)

You will transform the selected models into fixed feature extractors by freezing all layers except the final one and evaluate their performance.

Discussion (30%, 15% for each): Similar to Task B, provides a comprehensive analysis of the models' performances. The focus should again be on a comparative

evaluation of their effectiveness in the new task, though this time as fixed feature extractors.

1.3 Task C Answer

Task A 是挑選模型 (ShuffleNet V2 和 EfficientNet B2)，Task B 則是要實際使用這兩個模型，並比較它們的表現。核心任務是做 Fine-tuning，再討論與分析。Task C 則是延續 Task B 的做法，差別在於，將最後一層以外的 weight凍結，來實際使用這兩個模型，並比較它們的表現。核心任務是使用預訓練的卷積神經網路（ConvNet）模型，並將其作為一個「固定特徵提取器」來解決一個新的任務（New Task）。

模型一: ShuffleNet V2 X0_5 Weights.IMAGENET1K_V1

- 第一步是要先 import，在 Task A 選的模型是在 ImageNet (1000 個類別) 上預先訓練的，載入模型+預訓練的權重

模型1帶入模型1

(ShuffleNet_V2_X0_5_Weights.IMAGENET1K_V1)

```
[7] ✓ 0 秒
    import torchvision.models as models
    import torch.nn as nn

    from torchvision.models import ShuffleNet_V2_X0_5_Weights
    model = models.shufflenet_v2_x0_5(weights=ShuffleNet_V2_X0_5_Weights.IMAGENET1K_V1)
    #model = models.ShuffleNet_V2_X0_5(weights=ShuffleNet_V2_X0_5_Weights.IMAGENET1K_V1)
    print(model)

...
  Downloading: "https://download.pytorch.org/models/shufflenetv2_x0.5-f707e7126e.pth"
  100%[██████████] 5.28M/5.28M [00:00<00:00, 82.6MB/s]ShuffleNetV2(
    (conv1): Sequential(
        (0): Conv2d(3, 24, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
    )
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    (stage2): Sequential(
        (0): InvertedResidual(
            (branch1): Sequential(
                (conv5): Sequential(
                    (0): Conv2d(192, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
                    (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
                    (2): ReLU(inplace=True)
                )
                (fc): Linear(in_features=1024, out_features=1000, bias=True)
            )
        )
    )
  )
  '\nmodel = models.ShuffleNet_V2_X0_5(pretrained=True)\nmodel\nmodel = models.ShuffleNet_V2_X0_5(weights=None)\nmodel\n'
```

● 接著我們觀察模型的整體架構

```
...  
|---Linear: 1-7 [64, 1024] [64, 1000]  
1,025,000 65,600,000  
=====  
Total params: 1,366,792  
Trainable params: 1,366,792  
Non-trainable params: 0  
Total mult-adds (Units.GIGABYTES): 3.36  
=====  
Input size (MB): 50.33  
Forward/backward pass size (MB): 1369.95  
Params size (MB): 5.47  
Estimated Total Size (MB): 1425.75  
=====
```

Total params: 1,366,792

Trainable params: 1,366,792

Non-trainable params: 0

Total mult-adds (Units.GIGABYTES): 3.36

● 接著我們要修改結構，更換最後一層，使其輸出為 2 類。

```
[10] 0 秒  
# 2. 獲取最後一個全連接層 (fc 層) 的輸入特徵數量 (In Features)  
# 這是 ShuffleNet V2 模型的標準做法  
num_ftrs = model.fc.in_features  
  
# 3. 替換最後一個全連接層 (Linear Layer)  
# 將輸出類別數 (out_features) 設定為 2  
#model.fc = nn.Linear(num_ftrs, 2)  
# 只需要 1 個輸出 (logit) for BCE loss  
model.fc = nn.Linear(num_ftrs, 1)  
  
# 可選: 將模型移動到 GPU (如果您有設定 device 的話)  
model.to(device)  
  
# 4. 檢查修改後的模型結構  
print(model.fc)
```

```
...  
Linear(in_features=1024, out_features=1, bias=True)
```

● 設定好需要訓練的權重，需要凍結，以及 loss 定義。

```
[11] 0 秒  
# 凍結所有模型參數  
for param in model.parameters():  
    param.requires_grad = False  
  
# 只有新替換的分類器層的參數需要訓練  
for param in model.fc.parameters():  
    param.requires_grad = True  
  
# hyperparameter  
lr = 0.01  
weight_decay = 0.001  
# 修正: 只將 fc 層的參數傳遞給 optimizer  
optimizer = optims.Adam(model.fc.parameters(), lr=lr, weight_decay = weight_decay)  
#optimizer = optims.Adam(model.parameters(), lr=lr, weight_decay = weight_decay)
```

```

    |---Linear: 1-7           [64, 1024]          [64, 1]
1,025           65,600
=====
Total params: 342,817
Trainable params: 1,025
Non-trainable params: 341,792
Total mult-adds (Units.GIGABYTES): 3.30
=====
Input size (MB): 50.33
Forward/backward pass size (MB): 1369.44
Params size (MB): 1.37
Estimated Total Size (MB): 1421.14
=====
```

Total params: 342,817

Trainable params: 1,025

Non-trainable params: 341,792

Total mult-adds (Units.GIGABYTES): 3.30

- 在 X 光資料集上進行訓練 (fine-tuning)。

```

[25] 9 分鐘
    output_list.append(output_str)
    print(output_str)
    lr_scheduler.step(val_acc)

    ▾
    Epoch 1/10 - loss: 0.4835 - train_acc: 86.31% - val_loss: 0.4893 - val_acc: 75.00% - time: 59.18s
    Epoch 2/10 - loss: 0.3333 - train_acc: 90.62% - val_loss: 0.4385 - val_acc: 81.25% - time: 57.05s
    Epoch 3/10 - loss: 0.2888 - train_acc: 92.06% - val_loss: 0.4246 - val_acc: 87.50% - time: 58.33s
    Epoch 4/10 - loss: 0.2849 - train_acc: 91.57% - val_loss: 0.4218 - val_acc: 68.75% - time: 56.53s
    Epoch 5/10 - loss: 0.2747 - train_acc: 91.96% - val_loss: 0.4088 - val_acc: 87.50% - time: 56.11s
    Epoch 6/10 - loss: 0.2723 - train_acc: 92.66% - val_loss: 0.4068 - val_acc: 87.50% - time: 55.94s
    Epoch 7/10 - loss: 0.2720 - train_acc: 91.96% - val_loss: 0.4081 - val_acc: 87.50% - time: 57.64s
    Epoch 8/10 - loss: 0.2719 - train_acc: 91.72% - val_loss: 0.4052 - val_acc: 87.50% - time: 55.72s
    Epoch 9/10 - loss: 0.2669 - train_acc: 92.81% - val_loss: 0.4061 - val_acc: 87.50% - time: 55.66s
    Epoch 10/10 - loss: 0.2633 - train_acc: 93.06% - val_loss: 0.4038 - val_acc: 87.50% - time: 56.02s
```

Epoch 1/10 - loss: 0.4835 - train_acc: 86.31% - val_loss: 0.4893 - val_acc: 75.00% - time: 59.18s

Epoch 2/10 - loss: 0.3333 - train_acc: 90.62% - val_loss: 0.4385 - val_acc: 81.25% - time: 57.05s

Epoch 3/10 - loss: 0.2888 - train_acc: 92.06% - val_loss: 0.4246 - val_acc: 87.50% - time: 58.33s

Epoch 4/10 - loss: 0.2849 - train_acc: 91.57% - val_loss: 0.4218 - val_acc: 68.75% - time: 56.53s

Epoch 5/10 - loss: 0.2747 - train_acc: 91.96% - val_loss: 0.4088 - val_acc: 87.50% - time: 56.11s

Epoch 6/10 - loss: 0.2723 - train_acc: 92.66% - val_loss: 0.4068 - val_acc: 87.50% - time: 55.94s

Epoch 7/10 - loss: 0.2720 - train_acc: 91.96% - val_loss: 0.4081 - val_acc: 87.50% - time: 57.64s

Epoch 8/10 - loss: 0.2719 - train_acc: 91.72% - val_loss: 0.4052 - val_acc: 87.50% - time: 55.72s

Epoch 9/10 - loss: 0.2669 - train_acc: 92.81% - val_loss: 0.4061 - val_acc: 87.50% - time: 55.66s

Epoch 10/10 - loss: 0.2633 - train_acc: 93.06% - val_loss: 0.4038 - val_acc: 87.50% - time: 56.02s

- 記錄最終的準確率和訓練時間。

▼ Evaluate

```
[26] 13 秒
    model_path = 'model.pth'
    avg_test_loss, avg_test_acc = evaluate(model, device, model_path, test_loader)

    Model weights loaded successfully.
    Test Accuracy: 55.00%
    Test Loss: 0.7733
```

Model weights loaded successfully.

Test Accuracy: 55.00%

Test Loss: 0.7733

模型二：EfficientNet_B2_Weights.IMAGENET1K_V1

- 第一步是要先 import，在 Task A 選的模型是在 ImageNet (1000 個類別) 上預先訓練的，載入模型+預訓練的權重

模型2帶入模型2

(EfficientNet_B2_Weights.IMAGENET1K_V1)

```
[7] 0 秒
    import torchvision.models as models
    import torch.nn as nn

    from torchvision.models import EfficientNet_B2_Weights
    model = models.efficientnet_b2(weights=EfficientNet_B2_Weights.IMAGENET1K_V1)
    print(model)

... Downloading: "https://download.pytorch.org/models/efficientnet_b2_rwightman-c35c14"
100% [██████████] | 35.2M/35.2M [00:00<00:00, 179MB/s]EfficientNet(
    features): Sequential(
        (0): Conv2dNormActivation(
            (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
            (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): SiLU(inplace=True)
        )
        (1): Sequential(
            (0): MBConv(
                (block): Sequential(
                    (avgpool): AdaptiveAvgPool2d(output_size=1)
                    (classifier): Sequential(
                        (0): Dropout(p=0.3, inplace=True)
                        (1): Linear(in_features=1408, out_features=1000, bias=True)
                    )
                )
            )
        )
    )
    'model = models.ShuffleNet_V2_X0_5(pretrained=True)\nmodel\nmodel = models.ShuffleNet_V2_X0_5(weights=None)\nmodel\n'
```

- 接著我們觀察模型的整體架構

Layer (type:depth-idx)	Input Shape	Output Shape	Param #	Mul
EfficientNet	[64, 3, 256, 256]	[64, 1000]	--	--
Sequential: 1-1	[64, 3, 256, 256]	[64, 1408, 8, 8]	--	--
Conv2dNormActivation: 2-1	[64, 3, 256, 256]	[64, 32, 128, 128]	--	--
Conv2d: 3-1	[64, 3, 256, 256]	[64, 32, 128, 128]	864	
BatchNorm2d: 3-2	[64, 32, 128, 128]	[64, 32, 128, 128]	64	
SiLU: 3-3	[64, 32, 128, 128]	[64, 32, 128, 128]	--	--
Sequential: 2-2	[64, 32, 128, 128]	[64, 16, 128, 128]	--	--
MBConv: 3-4	[64, 32, 128, 128]	[64, 16, 128, 128]	1,448	
MBConv: 3-5	[64, 16, 128, 128]	[64, 16, 128, 128]	612	
Sequential: 2-3	[64, 16, 128, 128]	[64, 24, 64, 64]	--	--
MBConv: 3-6	[64, 16, 128, 128]	[64, 24, 64, 64]	6,004	
MBConv: 3-7	[64, 24, 64, 64]	[64, 24, 64, 64]	10,710	
AdaptiveAvgPool2d: 1-2	[64, 1408, 8, 8]	[64, 1408, 1, 1]	--	--
Sequential: 1-3	[64, 1408]	[64, 1000]	--	--
Dropout: 2-10	[64, 1408]	[64, 1408]	--	--
Linear: 2-11	[64, 1408]	[64, 1000]	1,409,000	
90,176,000				
Total params: 9,109,994				
Trainable params: 9,109,994				
Non-trainable params: 0				
Total mult-adds (Units.GIGABYTES): 55.04				
Input size (MB): 50.33				
Forward/backward pass size (MB): 13105.12				
Params size (MB): 36.44				
Estimated Total Size (MB): 13191.89				

Total params: 9,109,994

Trainable params: 9,109,994

Non-trainable params: 0

Total mult-adds (Units.GIGABYTES): 55.04

Input size (MB): 50.33

Forward/backward pass size (MB): 13105.12

Params size (MB): 36.44

Estimated Total Size (MB): 13191.89

- 接著我們要修改結構，更換最後一層，使其輸出為 2 類。

```

# 1. 存取 'classifier' 容器中的最後一個元素 (即 Linear 層)
final_linear_layer = model.classifier[-1]
num_ftrs = final_linear_layer.in_features

# 2. 替換最終層 (從 1000 類換成 1 類)
# 直接替換 nn.Sequential 容器中的最後一個元素
model.classifier[-1] = nn.Linear(num_ftrs, 1)

# Fix: Clear GPU memory before attempting to move the model
import torch
import gc
torch.cuda.empty_cache()
gc.collect()

```

... Linear(in_features=1408, out_features=1, bias=True)

```

... └─Sequential: 1-3
    └─Dropout: 2-10
        └─Linear: 2-11
90,176
=====
Total params: 7,702,403
Trainable params: 7,702,403
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 54.95
=====
Input size (MB): 50.33
Forward/backward pass size (MB): 13104.60
Params size (MB): 30.81
Estimated Total Size (MB): 13185.75
=====
```

Total params: 7,702,403

Trainable params: 7,702,403

Non-trainable params: 0

Total mult-adds (Units.GIGABYTES): 54.95

- 設定好需要訓練的權重，**需要凍結**，以及 loss 定義。

```

[12] 0 秒
▶ # 凍結所有模型參數
for param in model.parameters():
    param.requires_grad = False

# 只有新替換的分類器層的參數需要訓練
for param in model.classifier[-1].parameters():
    param.requires_grad = True

# hyperparameter
lr = 0.01
weight_decay = 0.001
# 修正：只將 fc 層的參數傳遞給 optimizer
optimizer = optims.Adam(model.classifier[-1].parameters(), lr=lr, weight_decay = weight_decay)
#optimizer = optims.Adam(model.parameters(), lr=lr, weight_decay = weight_decay)
lr_scheduler = optims.lr_scheduler.ReduceLROnPlateau(optimizer, factor=0.1, patience=5, mode='max')
epochs = 10

```

```

...
  -- Sequential: 1-3 [64, 1408] [64, 1]
  |   └─ Dropout: 2-10 [64, 1408] [64, 1408] --
  |   └─ Linear: 2-11 [64, 1408] [64, 1] 1,409
90, 176
=====
Total params: 7,702,403
Trainable params: 1,409
Non-trainable params: 7,700,994
Total mult-adds (Units.GIGABYTES): 54.95
=====
Input size (MB): 50.33
Forward/backward pass size (MB): 13104.60
Params size (MB): 30.81
Estimated Total Size (MB): 13185.75
=====
```

Total params: 7,702,403

Trainable params: 1,409

Non-trainable params: 7,700,994

Total mult-adds (Units.GIGABYTES): 54.95

- 在 X 光資料集上進行訓練 (fine-tuning)。

```

[14] 10
✓ 分鐘
[14] 10
✓ 分鐘
  print(output_str)
  lr_scheduler.step(val_acc)
...
*** Epoch 1/10 - loss: 0.2583 - train_acc: 88.64% - val_loss: 0.2284 - val_acc: 81.25% - time: 65.02s
Epoch 2/10 - loss: 0.1618 - train_acc: 94.05% - val_loss: 0.3662 - val_acc: 87.50% - time: 62.75s
Epoch 3/10 - loss: 0.1646 - train_acc: 93.70% - val_loss: 0.5867 - val_acc: 68.75% - time: 62.44s
Epoch 4/10 - loss: 0.1464 - train_acc: 94.64% - val_loss: 0.3421 - val_acc: 81.25% - time: 61.75s
Epoch 5/10 - loss: 0.1585 - train_acc: 94.54% - val_loss: 0.0998 - val_acc: 93.75% - time: 62.72s
Epoch 6/10 - loss: 0.1308 - train_acc: 95.19% - val_loss: 0.1678 - val_acc: 93.75% - time: 62.40s
Epoch 7/10 - loss: 0.1424 - train_acc: 94.69% - val_loss: 0.3165 - val_acc: 75.00% - time: 62.37s
Epoch 8/10 - loss: 0.1148 - train_acc: 95.68% - val_loss: 0.3150 - val_acc: 93.75% - time: 61.12s
Epoch 9/10 - loss: 0.1725 - train_acc: 94.00% - val_loss: 0.1357 - val_acc: 93.75% - time: 62.40s
Epoch 10/10 - loss: 0.1617 - train_acc: 94.25% - val_loss: 0.3249 - val_acc: 93.75% - time: 62.48s
```

Epoch 1/10 - loss: 0.2583 - train_acc: 88.64% - val_loss: 0.2284 - val_acc: 81.25% - time: 65.02s

Epoch 2/10 - loss: 0.1618 - train_acc: 94.05% - val_loss: 0.3662 - val_acc: 87.50% - time: 62.75s

Epoch 3/10 - loss: 0.1646 - train_acc: 93.70% - val_loss: 0.5867 - val_acc: 68.75% - time: 62.44s

Epoch 4/10 - loss: 0.1464 - train_acc: 94.64% - val_loss: 0.3421 - val_acc: 81.25% - time: 61.75s

Epoch 5/10 - loss: 0.1585 - train_acc: 94.54% - val_loss: 0.0998 - val_acc: 93.75% - time: 62.72s

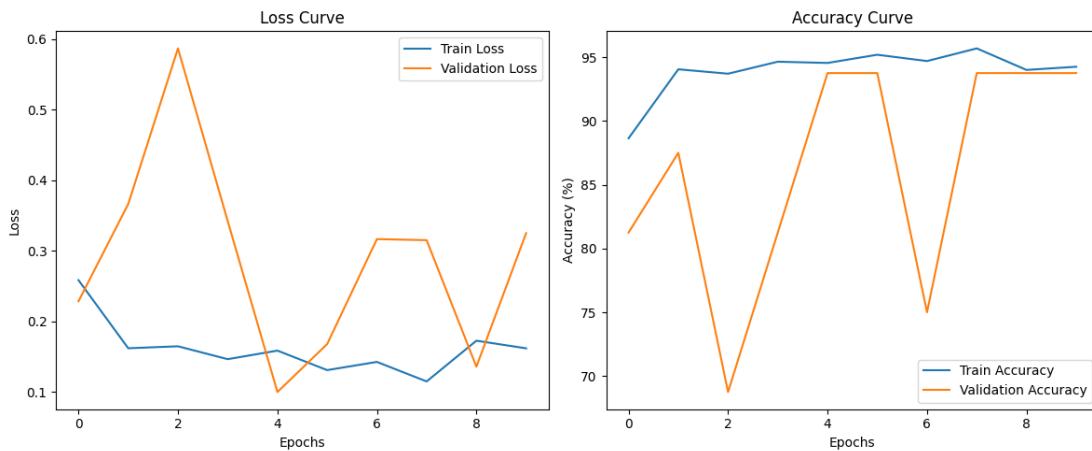
Epoch 6/10 - loss: 0.1308 - train_acc: 95.19% - val_loss: 0.1678 - val_acc: 93.75% - time: 62.40s

Epoch 7/10 - loss: 0.1424 - train_acc: 94.69% - val_loss: 0.3165 - val_acc: 75.00% - time: 62.37s

Epoch 8/10 - loss: 0.1148 - train_acc: 95.68% - val_loss: 0.3150 - val_acc: 93.75% - time: 61.12s

Epoch 9/10 - loss: 0.1725 - train_acc: 94.00% - val_loss: 0.1357 - val_acc: 93.75% - time: 62.40s

Epoch 10/10 - loss: 0.1617 - train_acc: 94.25% - val_loss: 0.3249 - val_acc: 93.75% - time: 62.48s



- 記錄最終的準確率和訓練時間。

▼ Evaluate

```
[16] ✓ 16 秒
model_path = 'model.pth'
avg_test_loss, avg_test_acc = evaluate(model, device, model_path, test_loader)

Model weights loaded successfully.
Test Accuracy: 52.19%
Test Loss: 1.3310
```

Model weights loaded successfully.

Test Accuracy: 52.19%

Test Loss: 1.3310

討論與分析：比較兩個模型（ShuffleNet V2 和 EfficientNet B2）在 X 光資料集上的表現。

Pre-trained model 數據

	Acc@1	Params	GFLOPS
ShuffleNet_V2_X0_5_Weights.IMAGENET1K_V1	60.552	1.4M	0.04
EfficientNet_B2_Weights.IMAGENET1K_V1	80.608	9.1M	1.09

Task C 在 X 光資料集上的表現

	ShuffleNet_V2_X0_5	EfficientNet_B2
Task A		
Pre-trained model (修改 output 前)		
Total params	1,366,792	9,109,994
Trainable params	1,366,792	9,109,994

Non-trainable params	0	0
Total mult-adds	3.36	55.04
Acc@1 on imagenet	60.552%	80.608%
Task B Pre-trained model (修改 output 後) (no freezing weight)		
Total params	342,817	7,702,403
Trainable params	342,817	7,702,403
Non-trainable params	0	0
Total mult-adds	3.30	54.95
Acc@1 on x-ray	73.26%	52.19%
Task C Pre-trained model (修改 output 後) (freezing weight)		
Total params	342,817	7,702,403
Trainable params	1,025	1,409
Non-trainable params	341,792	7,700,994
Total mult-adds	3.30	54.95
Acc@1 on x-ray	55.00%	52.19%

- 原本假設 for 比較性評估 (comparative evaluation): Task B 的問題可能是參數過多難以學習，Task C 只學習最後一層，須調整參數量小應該不會有這個問題，所以預期跟 Task A 應該會有一致表現，也就是說，Task A 上，EfficientNet B2 (ImageNet 準確率 80.6%)和 ShuffleNet V2 (ImageNet 準確率 60.5%)相比，EfficientNet B2 預訓練時學到的特徵 (Params 9.1M, GFLOPS 1.09) 更複雜、更強大，因此應該能更有效地適應 (adaptation) 新的醫學影像任務。
- **反向結果:** 然而，ShuffleNet V2(**X-ray 準確率 55.00%**) 在 transfer learning 上的訓練結果反而比 EfficientNet B2(**X-ray 準確率 52.19%**)來的好。
- **反向結果可能的原因:** 模型過於複雜→因為參數固定，等於是只能用學習到的特徵來學習辨識 X 光片。特徵的「複雜度」不代表「通用性」 ShuffleNet (小模型) 由於模型較小，它學到的特徵比較基礎和通用，例如「邊緣」、「角落」、「簡單紋理」。這些簡單特徵恰好對 X 光片上模糊的「一團東西」也有點用。EfficientNet (大模型) 由於模型更深更複雜，它學

到的特徵非常複雜且高度特化，例如「狗鼻子上的濕潤反光」、「特定種類花朵的花蕊」。因此，**當「凍結」所有層時**，你等於是拿著「辨識狗鼻子的工具」去「辨識肺炎」。**這些高度特化的工具完全派不上用場，對新任務來說，它們產生的特徵就像雜訊一樣，反而干擾了最後一層分類器的學習。**

- **嚴格來說，兩個模型皆可視為失敗，無統計顯著性**: 52.19% 和 55.00% 都等於「亂猜」。這兩者之間的差異很難有統計意義。與其解讀為「55% > 52%」，倒不如解讀為「兩者都 $\approx 50\%$ 」，即兩個模型都失敗了。
- **失敗的可能原因: 領域差異 (Domain Gap)**。模型是在 ImageNet (自然影像) 上預訓練的。它學會辨識的是「毛皮紋理」、「耳朵輪廓」、「車輪形狀」。X 光片的特徵是「浸潤 (infiltrates)」、「毛玻璃樣 (ground-glass opacity)」。這兩種特徵截然不同。

1.4 Task D: Comparison and Analysis (10%)

After completing Task B and C, contrast the performance outcomes and adaptability of the models when subjected to the two distinct transfer learning approaches.

Discussion (10%): Offer a succinct analysis that highlights the differences in performance and adaptability observed when the models are fine-tuned versus when used as fixed feature extractors.

1.4 Task D Answer

- 我們先來比較 Task A, Task B, Task C 分別是在做什麼

Task	Goal	Description
Task A	Pre-trained model	是挑選模型 (ShuffleNet V2 和 EfficientNet B2)
Task B	Transform the Models with Fine-tuning (微調)	Task B: Fine-tuning (微調) 解凍模型的多個層 (所有層)，並重新訓練它們。
Task C	Transform the Models with Fixed Feature Extractor (固定特徵提取器)	Task C: Fixed Feature Extractor (固定特徵提取器) 凍結了所有層，只訓練了最後一層。

- 接著我們來比較 Task A, Task B, Task C 的結果

Pre-trained model 數據

	Acc@1	Params	GFLOPS
ShuffleNet_V2_X0_5_Weights.IMAGENET1K_V1	60.552	1.4M	0.04
EfficientNet_B2_Weights.IMAGENET1K_V1	80.608	9.1M	1.09

Task A, Task B, Task C 的在資料集上的表現

	ShuffleNet_V2_X0_5	EfficientNet_B2
Task A		
Pre-trained model (修改 output 前)		
Total params	1,366,792	9,109,994
Trainable params	1,366,792	9,109,994
Non-trainable params	0	0
Total mult-adds	3.36	55.04
Acc@1 on imagenet	60.552%	80.608%
Task B		
Pre-trained model (修改 output 後) (no freezing weight)		
Total params	342,817	7,702,403
Trainable params	342,817	7,702,403
Non-trainable params	0	0
Total mult-adds	3.30	54.95
Acc@1 on x-ray	73.26%	52.19%
Task C		
Pre-trained model (修改 output 後) (freezing weight)		
Total params	342,817	7,702,403
Trainable params	1,025	1,409
Non-trainable params	341,792	7,700,994
Total mult-adds	3.30	54.95
Acc@1 on x-ray	55.00%	52.19%

- 為何 Task B(Transform the Models with Fine-tuning (微調))的性能會優於 Task C(Transform the Models with Fixed Feature Extractor (固定特徵提取器))？適應新任務的能力更好？Task C (固定特徵提取器) 的結果非常糟糕 (ShuffleNet_V2 Test Acc 只有 55.00%，EfficientNet_B2 Test Acc 只有 52.19%)。Task B (微調) 的準確率比較好一點 (ShuffleNet_V2 Test Acc 為 73.26%，EfficientNet_B2 Test Acc 仍只有 52.19%)。
- Task C 的失敗：在 Task C (固定特徵提取器) 中，模型表現極差，幾乎等於隨機猜測。這證明了模型在 ImageNet 上學到的「固定特徵」(用來看貓、狗、車子) 無法直接被用來辨識肺炎。模型的「適應性」(Adaptability)幾乎為零，因為「領域差異 (Domain Gap)」太大了。
- Task B 的成功：在 Task B (微調) 中，透過「解凍」模型的所有層，我們允許模型去調整 (adapt) 它原有的特徵。模型得以將原本「看貓毛紋理」的知識，微調成「看 X 光片上浸潤紋理」的新知識。這顯示了「微調」具有高得多的適應性(Adaptability)。
- 結論 1：對於 X 光片分類這種「新任務」與「預訓練任務 (ImageNet)」差異巨大的情況下，Fine-tuning (微調) 是一種遠比 Fixed Feature Extractor (固定特徵提取器) 更有效、適應性更強的遷移學習策略。
- 為何不論是 Task B(Transform the Models with Fine-tuning (微調))或是 Task C(Transform the Models with Fixed Feature Extractor (固定特徵提取器))，ShuffleNet_V2 的表現都優於 EfficientNet_B2 Test Acc？
實驗數據：
Task B (微調)： ShuffleNet (73.26%) > EfficientNet (52.19%)
Task C (固定)： ShuffleNet (55.00%) > EfficientNet (52.19%)
- 因為模型過於複雜，複雜性不等於適用性。EfficientNet_B2 太複雜了。EfficientNet_B2 有大約 920 萬個參數。ShuffleNet_V2_X0.5 只有大約 130 萬個參數。For task B: 當 X-ray 資料集不夠大時，難以完整訓練擁有 920 萬個參數的 EfficientNet。For task C: 若參數固定，也等於是只能用學習到的特徵來學習辨識 X 光片，EfficientNet (大模型)由於模型更深更複雜，它學到的特徵非常複雜且高度特化，當「凍結」所有層時，高度特化的工具完全派不上用場，對新任務來說，它們產生的特徵就像雜訊一樣，反而干擾了最後一層分類器的學習。
- 結論 2：Domain Gap 大且目標資料集有限的情況下，「複雜性不等於適用性」。選擇一個較小、更通用的模型（如 ShuffleNet）並採用微調策略，可

能是最穩健有效的方法。

1.5 Task E: Test Dataset Analysis (10%)

In the original Lab 4's code, you may have encountered challenges in enhancing the performance on the test dataset.

Discussion (10%): Elucidate your perspective on this phenomenon. Provide an analysis explaining the reasons behind the difficulty in improving the test dataset performance.

1.5 Task E Answer

- Task E 是回顧分析上一次在 Lab 4 做的那個 4 層 FC (Fully-Connected) 網路，並分析一下當時遇到的問題。我們先來回顧當時的模型：

```
Sequential(  
    (0): Flatten(start_dim=1, end_dim=-1)  
    (1): Linear(in_features=65536, out_features=64, bias=True)  
    (2): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (3): ReLU()  
    (4): Dropout(p=0.5, inplace=False)  
    (5): Linear(in_features=64, out_features=64, bias=True)  
    (6): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (7): ReLU()  
    (8): Dropout(p=0.5, inplace=False)  
    (9): Linear(in_features=64, out_features=64, bias=True)  
    (10): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (11): ReLU()  
    (12): Dropout(p=0.5, inplace=False)  
    (13): Linear(in_features=64, out_features=1, bias=True)  
    (14): Sigmoid()  
)
```

Layer (type:depth-idx)	Output Shape	Param #
Sequential	[64, 1]	--
└─Flatten: 1-1	[64, 65536]	--
└─Linear: 1-2	[64, 64]	4, 194, 368
└─BatchNorm1d: 1-3	[64, 64]	128
└─ReLU: 1-4	[64, 64]	--
└─Dropout: 1-5	[64, 64]	--
└─Linear: 1-6	[64, 64]	4, 160
└─BatchNorm1d: 1-7	[64, 64]	128
└─ReLU: 1-8	[64, 64]	--
└─Dropout: 1-9	[64, 64]	--
└─Linear: 1-10	[64, 64]	4, 160
└─BatchNorm1d: 1-11	[64, 64]	128
└─ReLU: 1-12	[64, 64]	--
└─Dropout: 1-13	[64, 64]	--
└─Linear: 1-14	[64, 1]	65
└─Sigmoid: 1-15	[64, 1]	--

```
=====
Total params: 4,203,137
Trainable params: 4,203,137
Non-trainable params: 0
Total mult-adds (Units. MEGABYTES): 269.00
=====
```

```
=====
Input size (MB): 16.78
Forward/backward pass size (MB): 0.20
Params size (MB): 16.81
Estimated Total Size (MB): 33.79
=====
```

Total params: 4,203,137

Trainable params: 4,203,137

Non-trainable params: 0

Total mult-adds (Units.MEGABYTES): 269.00

- 為什麼在 **Lab 4** 中，單純用 **4 層 FC 網路**來跑 **X-ray 分類的模型**，很難在『**測試資料集 (Test dataset)**』上持續提升效能？其實用單純 FC 網路來做圖像分類本來就會遇到諸多問題與限制。
- **問題 1：破壞空間結構 (Spatial Structure)**。FC 網路看不懂圖片結構，因為 FC 網路在處理圖片時，會先把 2D 的圖片「**攤平 (Flatten)**」成一個很長的一維向量。這個動作會完全破壞圖片的「**空間結構 (Spatial Structure)**」。對 FC 網路來說， $(x=10, y=10)$ 的像素，跟 $(x=11, y=10)$ 的像素是完全獨立的輸入；它無法理解「這兩個像素是相鄰的」。但對 X-ray 來說，「**浸潤 (infiltrates)** 的形狀」、「**毛玻璃的紋理**」才是判斷肺炎的關鍵，這些都需要空間結構。
- **問題 2：缺乏空間不變性 (Lack of Spatial Invariance)**。像 Task B/C 的 CNN 模型) 是用「**卷積核 (Kernel)**」來掃描圖片。如果它學會了一個「邊緣偵測器」，這個偵測器可以在圖片的**任何位置**偵測到邊緣。FC 網路沒有這個能力。如果它在訓練集中看到「肺炎在左上角」，它只會學會「當左上角的像素有反應時，分類為肺炎」。如果測試集圖片的「肺炎在右下角」，它就完全無法辨識。
- **問題 3：參數爆炸與嚴重過擬合 (Parameter Explosion & Overfitting)**。在我們的 FC 模型中，只是一個簡單的 4 層 FC 網路，但是參數已經高達 4,203,137。FC 模型跟 CNN 模型相比，很容易參數爆炸。用巨量的參數去訓練（可能只有幾千張的）X-ray 資料集，模型唯一能做的事就是「死記硬背」(Memorization) 訓練集。這會導致 Train Accuracy 很高，但 Test Accuracy 極低的典型「嚴重過擬合」現象。這完美地解釋了「為什麼很難提升 test dataset performance」。

- 那為什麼 Task B, Task C 透過 CNN 做 transfer learning 的結果，反而比不上單純 4 層網路的 fc model 訓練的結果？

模型	學習方法	Accuracy on test set of X-ray
4 Fc layers	From scratch	76.61%
ShuffleNet/EfficientNet	Transfer learning with Fine Tunning	ShuffleNet (73.26%) > EfficientNet (52.19%)
ShuffleNet/EfficientNet	Transfer learning with Fixed Feature Extractor	ShuffleNet (55.00%) > EfficientNet (52.19%)

- 最主要的原因，還是在於領域差異 (Domain Gap)。模型是在 ImageNet (自然影像) 上預訓練的。它學會辨識的是「毛皮紋理」、「耳朵輪廓」、「車輪形狀」。X 光片的特徵是「浸潤 (infiltrates)」、「毛玻璃樣 (ground-glass opacity)」。這兩種特徵截然不同。這些高度特化的工具完全派不上用場，對新任務來說，它們產生的特徵就像雜訊一樣，反而干擾了分類器的學習。在 Task B (微調) 中，透過「解凍」模型的所有層，至少允許模型去調整 (adapt) 它原有的特徵，所以結果會稍微好一點。
- 其他可能造成學習障礙的理由: dataset 不足(training 2000, test 624)與 validation set 過小(validation 16)。
- Test Accuracy 與 Validation Accuracy 差異巨大，暗示我們的驗證 (Validation) 過程本身是不可靠的。可能原因 1 (資料洩漏 Data Leakage)：Train/Val 的資料切分可能沒有按「病人 ID」執行，導致同一病人的 X 光片同時出現在訓練集和驗證集。模型學會了「辨識病人特徵」，而不是「辨識肺炎」。可能原因 2 (分佈不一致)：Validation set 和 Test set 來自截然不同的資料源（例如：Val 是 A 醫院的清晰影像，Test 是 B 醫院的模糊影像；或 Val 的肺炎比例為 50%，Test 的肺炎比例為 80%）。可能原因 3 (Val Set 太小)：93.75% 的準確率可能是由非常小的樣本數（例如 16 張答對 15 張）得出的，這不具統計代表性。
- 一個極小且不具代表性的驗證集 (16 張圖) 會徹底摧毀遷移學習的訓練過程。它會導致更強大的模型 (ShuffleNet) 嚴重提早停止 (Early Stopping)，使其儲存一個訓練不足的「半成品」(Epoch 3 的權重)，導致其測試準確率低落。相比之下，較弱的模型 (FC Net) 因為無法過度擬合這 16 張圖，反而被允許訓練更久，從而找到一個更具泛化能力的權重，取得了更高的測試準確率。
- 總結

問題來源	例子	解決方法
資料本身	dataset 不足(training 2000, test 624, validation 16)與 validation set 過小(validation 16)	1. 增加樣本數 2. 修正資料分割。例如從 train 資料夾中，手動移動 500 張圖片到 val 資料夾中。
模型本身	FC 網路的設計本質不適合處理影像。 1.攤平 (Flatten) 影像，破壞了所有關鍵的空間結構與紋理。 2.缺乏空間不變性，無法泛化 (generalize) 物件在不同位置時的情況。 3.巨量參數，導致在有限的 X-ray 資料集上嚴重過擬合。	改用 CNN 模型架構
Transfer learning 本身	領域差異 (Domain Gap)過大。 1.模型是在 ImageNet (自然影像) 上預訓練的。要學習的是 X 光片的特徵。這兩種特徵截然不同。 2.對新任務來說，它們產生的特徵就像雜訊一樣，反而干擾了分類器的學習。 3.在 Task B (微調) 中，透過「解凍」模型的所有層，至少允許模型去調整 (adapt) 它原有的特徵，所以結果會稍微好一點。	Domain Gap 大且目標資料集有限的情況下，「複雜性不等於適用性」。選擇一個較小、更通用的模型並採用 微調策略 ，可能是最穩健有效的方法。