AI CUP 2026 課程實作｜程式教學版

瑕疵檢測與根因分析

太陽能板 EO／IR 影像｜PRB-FPN 偵測 ＋ Qwen 成因說明

給國中、高中生的一步一步實作

Windows 與 Linux 都適用

環境安裝 → 重現官方 baseline → 逐支程式講解 → 自己訓練與改進

02　課程地圖

今天會學到什麼

PART 1

比賽在做什麼

輸入、輸出、提交檔、怎麼計分

PART 2

先懂幾個觀念

偵測框、IoU、門檻、資料切分、
語言模型微調

PART 3

準備環境

Windows／Linux 安裝、放資料、
檢查資料

PART 4

快速開始

一行指令重現官方 baseline，
上傳提交檔

PART 5

逐步講解程式

每一支程式在做什麼、
關鍵程式碼逐行看

PART 6

進階與練習

重訓偵測器、改進方向、
常見問題、小練習

03　比賽在做什麼

比賽在做什麼：找出瑕疵，說出原因

輸入
EO／IR 影像
可見光＋熱影像

→

檢測
PRB-FPN 偵測
框出面板、熱點、破裂

→

解釋
Qwen 0.5B
為每個異常寫一句成因

→

輸出
submission.json
上傳到 AIdea 評分

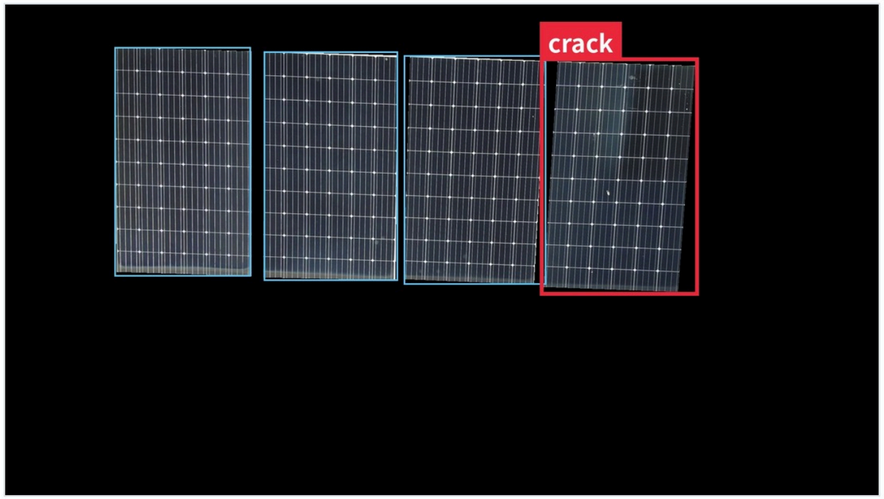
檢測 **瑕疵在哪裡？**　定位分數佔總分 **70%**

解釋 **為什麼會這樣？**　文字分數佔總分 **30%**

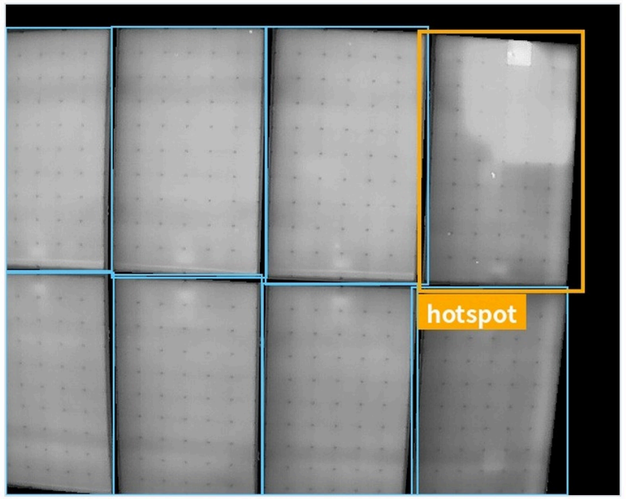
04　比賽在做什麼

兩種相機：可見光 EO 與熱影像 IR

可見光 EO（train_0001）



熱影像 IR（train_0001）



訓練資料 train_0001
框是它的標註（答案）

- 太陽能板
- 破裂
- 熱點

EO　1920×1080
可見光：看得到外觀，例如裂痕

IR　640×512
熱影像：看得到溫度，越亮越熱

同一個編號
EO 與 IR 拍同一處，範圍不同

05　比賽在做什麼

要找的三種東西

`solar_panel`

太陽能板

一片太陽能模組
沒有異常的正常模組

55,810 個訓練集標註

`hotspot`

熱點

IR 上異常發熱的模組
訓練集只出現在 IR

444 個訓練集標註

`crack`

破裂

EO 上看得到破損的模組
訓練集只出現在 EO

352 個訓練集標註

注意

異常的框也要框住整片模組；同一片模組只標一個類別

依官方 evaluation kit 第 4 節「標註慣例」

06　比賽在做什麼

提交檔長什麼樣子

```
{
  "test/EO/image/test_001.jpg": [
    {"label": "solar_panel",
     "points": [[100, 60], [300, 60],
                [300, 400], [100, 400]]},
    {"label": "crack",
     "points": [[320, 60], [520, 60],
                [520, 400], [320, 400]],
     "reason": "這裡寫一句中文的成因說明"}
  ],
  "test/IR/image/test_001.jpg": []
}
```

JSON：存資料的文字格式，寫法很像 Python 的 dict

1. **key**＝影像路徑，必須和 test_images.csv 裡的字串完全相同
2. **label** 三選一：solar_panel、hotspot、crack（全小寫）
3. **points**＝框的角點座標 [x, y]，單位是像素
4. **reason** 只有 hotspot、crack 要寫，用中文寫成因
5. 沒有物件的影像給空陣列 []（像 Python 的空 list）

座標與文字只是示範格式，不是答案；完整規定以 evaluation kit 第 3 節為準

07　比賽在做什麼

怎麼計分

$$總分 = 0.7 \times F1_TIoU + 0.3 \times F1_TEXT$$

項目	白話說明
F1_TIoU（70%）	框得準不準：面板、熱點、破裂三類分開算，框和答案重疊越多分數越高，最後三類平均
沒有信心分數	送出的每一個框都會被算進去，亂框的框會拉低分數
F1_TEXT（30%）	原因寫得好不好：只看和答案配對成功（同類別、IoU ≥ 0.5）的熱點與破裂
文字怎麼比	用 BERTScore 比較你的 reason 和官方的參考描述有多像，參考描述不公開

完整規則以官方 evaluation kit 的 README 為準

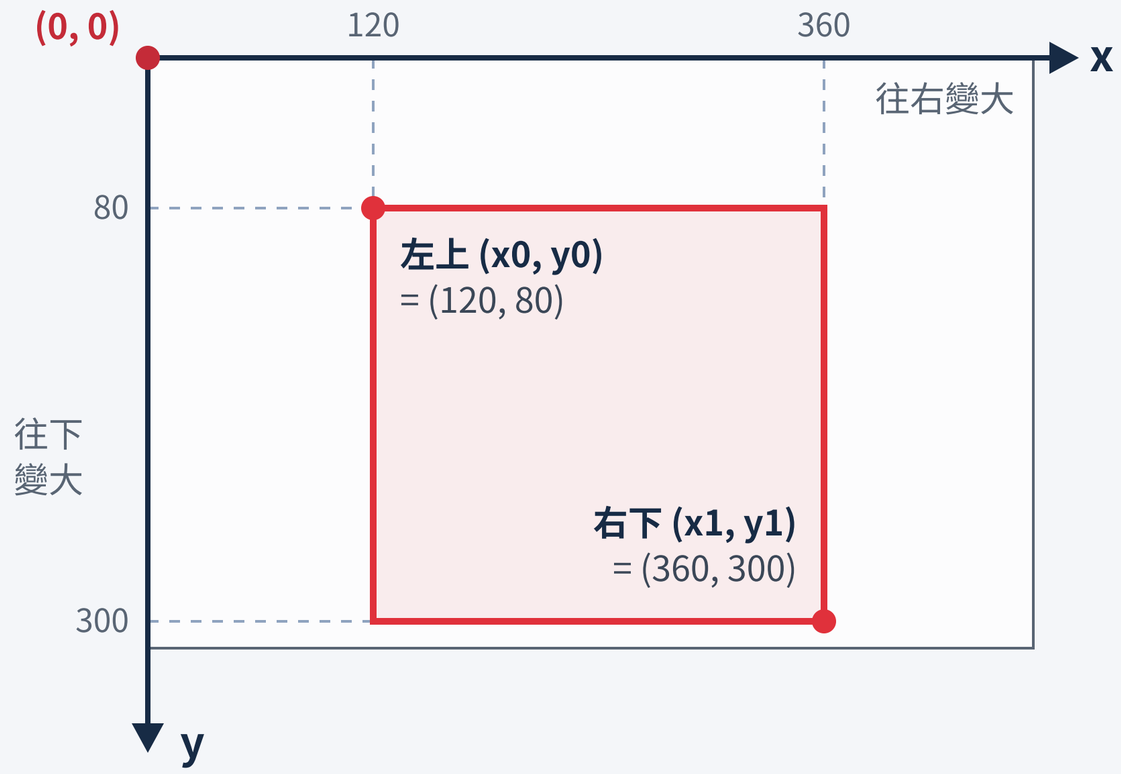
PART 2

先懂幾個觀念

偵測框｜IoU｜信心分數與門檻｜訓練、驗證、測試｜語言模型與微調

09　先懂幾個觀念

偵測框：用四個數字描述一個框

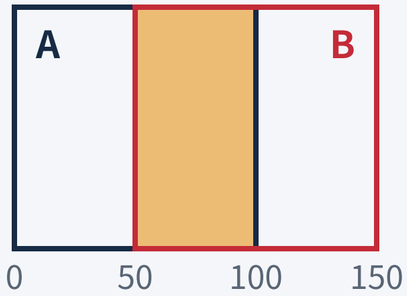


1. **像素**（pixel）＝影像上的一個小點，
   座標就是在數「第幾個點」
2. 原點 (0, 0) 在**左上角**：x 往右變大，**y 往下變大**
   （和數學課的座標相反）
3. 只要左上 (x0, y0) 和右下 (x1, y1) 兩個角，
   就能決定一個框

```
# baseline 寫成 4 個角：左上→右上→右下→左下
[[x0,y0], [x1,y0], [x1,y1], [x0,y1]]
# 左圖的框
[[120,80], [360,80], [360,300], [120,300]]
```

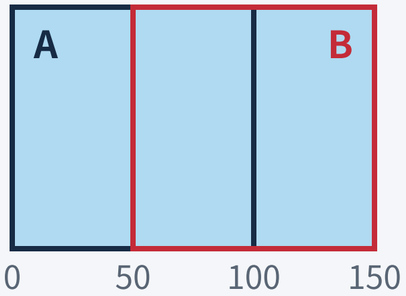
10　先懂幾個觀念

IoU：兩個框重疊了多少



交集＝重疊的部分

÷



聯集＝兩個框合起來

＝

IoU（Intersection over Union）

介於 0～1

完全重合＝1，完全沒碰到＝0
越大代表重疊得越多

算算看：A 是 x 0～100，B 是 x 50～150，高度都是 0～100

交集＝50 × 100＝5,000

聯集＝10,000＋10,000−5,000＝15,000

IoU＝5,000 ÷ 15,000 ≈ 0.33

計分規則

IoU ≥ 0.5 才算配對

文字分數只算和答案配對成功
（同類別、IoU ≥ 0.5）的異常

11　先懂幾個觀念

信心分數與門檻

偵測器替每個框打一個**信心分數**（0～1，越接近 1 越有把握），分數 ≥ **門檻**的框才留下



門檻調高 ↑

框變少，亂框也少

但真的瑕疵可能被漏掉

門檻調低 ↓

框變多，不容易漏掉

但亂框也跟著變多

評分沒有信心分數，送出的每個框都會被算進去，
所以門檻很重要

```
# pipeline.py 的 --conf 預設值
--conf "solar_panel=0.25,hotspot=0.25,crack=0.25"
```

12　先懂幾個觀念

訓練、驗證、測試

訓練集 train

課本練習題

有答案（標註），
模型拿來學

1046 組（同編號的 EO＋IR）

驗證集 validation

模擬考

從訓練集切出一部分，
檢查學得好不好

15% 依編號隨機切出

測試集 test

正式考試

沒有答案，
交卷後由平台評分

696 組（同編號的 EO＋IR）

調參數只能看驗證集

例如門檻、訓練幾輪這些設定，都用驗證集比較；
同編號的 EO 與 IR 一定分在同一邊

13　先懂幾個觀念

語言模型與微調（SFT）

語言模型

Qwen2.5-0.5B-Instruct

約 5 億個參數的小型語言模型
讀一段文字後，接著寫下去

參數＝模型裡面可以被調整的數字

微調 SFT（Supervised Fine-Tuning）

像讀很多範文來學寫作

給它很多「題目 → 標準答案」範例
讓它學會這個比賽要的寫法

標準答案來自訓練集標註裡的 reason

題目（prompt）

瑕疵的位置、大小
同編號另一張影像有沒有異常

→

微調後的

Qwen

→

回答

一句中文的成因說明

14　先懂幾個觀念

整個流程：7 個步驟

1 ir-norm

熱影像轉灰階

`tools/ir_gray.py`

2 detect

偵測面板與瑕疵

`prb/detect.py`

3 dets

過濾與後處理

`reason/yolo2dets.py`

4 sft-data

做 Qwen 的訓練資料

`reason/build_sft_data.py`

5 train-sft

微調 Qwen

`reason/train_sft.py`

6 eval

在驗證集上檢查

`reason/eval_val.py`

7 submit

產生提交檔

`reason/infer_submission.py`

一次跑完 1～7

`python pipeline.py all`

依序跑完全部步驟

各步驟的產出都放在 outputs/ 資料夾，後面的步驟接著讀取（eval 只印在畫面上）

15　先懂幾個觀念

兩種用法

用法	做什麼	時間（RTX 5060 Ti 16 GB 實測）
A　重現官方 baseline	用附的偵測器與 Qwen 權重（訓練好的模型參數） 直接產生提交檔，不訓練任何模型	約 3 分鐘
B　自己訓練成因模型	偵測器沿用附的權重；Qwen 從頭微調，再產生提交檔	約 50 分鐘（預設設定） 改用 --bs 8 約 12 分鐘

```
> python pipeline.py all --reason-model weights/qwen_reason  # 用法 A
> python pipeline.py all                                     # 用法 B
```

建議順序　**先跑 A 確認環境，再進入 B**

PART 3

準備環境

Windows 與 Linux 兩條路｜下載｜放資料｜檢查資料

17　準備環境

需要什麼樣的電腦

項目	需求
顯示卡	NVIDIA 顯示卡。用法 A（重現官方 baseline）實測顯示記憶體峰值約 2.6 GB， 含 Windows 桌面原本佔用的約 1 GB
作業系統	官方在 Windows 11 驗證；Linux 也可使用，但**未經官方驗證**
Python	**3.12**（官方只在 3.12 驗證；requirements.txt 固定的版本需要 3.12）
硬碟空間	專案所在磁碟預留 **15 GB**，C 槽另外預留 **5 GB**（pip 下載快取與 Qwen 基底模型會存在使用者資料夾）
網路	安裝套件時需要；用法 B 第一次訓練會下載 Qwen 基底模型（約 1 GB）

GPU＝顯示卡，模型的計算主要靠它；**顯示記憶體**＝顯示卡上的記憶體；**峰值**＝用量最高的那一刻

官方驗證環境：Windows 11、RTX 5060 Ti 16 GB、Python 3.12.3、torch 2.11.0（CUDA 12.8）

18 準備環境

先下載三樣東西

1

官方 baseline

`aicup_pv_baseline.zip`

程式＋已訓練好的權重
（模型學到的數字），約 950 MB

2

訓練集與測試集

`train/`
`test/`

解壓後的兩個資料夾；train 有標註，test 只有影像

3

evaluation kit

`author-kit_1012.zip`

官方評分工具包，內含
make_submission.py、
test_images.csv、
sample_submission.json

三樣都從競賽頁下載。baseline 請解壓到短路徑，例如 **D:\aicup_pv_baseline**

原因：Windows 的路徑長度上限是 260 字元，路徑太長可能出現「系統找不到指定的路徑」

Windows ①：安裝 Python 3.12

1. 到 **python.org** 下載 Python 3.12 的 Windows 安裝檔（建議 3.12.10）
2. 安裝程式第一頁勾選「**Add python.exe to PATH**」，終端機才找得到 python
3. 裝完把終端機關掉、重新打開
4. 打開 **PowerShell**（終端機＝打指令的視窗），切到專案根目錄，確認版本

```
# 切到專案根目錄（有 pipeline.py 的資料夾）
> cd D:\aicup_pv_baseline
> python --version
Python 3.12.x
```

用 cmd（命令提示字元）要換磁碟機時：
cd /d D:\aicup_pv_baseline

顯示的不是 3.12？只把建立虛擬環境那一行改成
py -3.12 -m venv .venv，其他指令照寫 python

Windows ②：建立並啟用虛擬環境

虛擬環境（venv）＝這個專案自己的「套件房間」，不會和其他專案的套件打架

```
# 1. 建立虛擬環境（只要做一次）
> python -m venv .venv
# 2. 允許這個視窗執行指令碼（不需系統管理員）
> Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass
# 3. 啟用
> .venv\Scripts\Activate.ps1
(.venv) PS D:\aicup_pv_baseline>
```

用 cmd 的話

`.venv\Scripts\activate.bat`

一定要在 cmd 裡執行；
在 PowerShell 裡打不會報錯，但也不會啟用

每次開新終端機都要

切到專案根目錄，再啟用一次
（PowerShell 要再跑第 2、3 步）

21 準備環境

Linux：系統工具與虛擬環境（未經官方實測）

```
# 1. 系統工具（opencv 需要 libgl1、libglib2.0-0）
$ sudo apt-get update
$ sudo apt-get install -y python3.12-venv python3-dev libgl1 libglib2.0-0
# 2. 看得到顯示卡名稱與驅動版本，代表驅動裝好了
$ nvidia-smi
# 3. 在專案根目錄建立並啟用虛擬環境
$ python3.12 -m venv .venv
$ source .venv/bin/activate
```

未經官方實測

以 Ubuntu 24.04 為例；
其他發行版的套件名稱可能不同

名詞

sudo＝用管理員權限執行；
apt-get＝Ubuntu 的軟體安裝工具

啟用成功

前面出現 **(.venv)**；
之後的指令一律打 **python**

安裝 PyTorch：依顯示卡選版本

PyTorch＝跑模型用的套件；**CUDA**＝讓程式使用 NVIDIA 顯示卡計算的平台。網址結尾的 **cu128** 就是 CUDA 12.8 版

你的顯示卡	網址結尾	驗證狀態
RTX 50／40／30 系列、A100 等	**cu128**	官方驗證：RTX 5060 Ti
RTX 20、GTX 16／10、MX、T4、V100 等較舊的卡	**cu126**	未經官方驗證

```
# RTX 50／40／30 系列、A100 等（二選一，只打一行）
> pip install torch==2.11.0 torchvision==0.26.0 --index-url https://download.pytorch.org/whl/cu128
# RTX 20、GTX 16／10、MX、T4、V100 等較舊的卡
> pip install torch==2.11.0 torchvision==0.26.0 --index-url https://download.pytorch.org/whl/cu126
```

pip＝Python 的套件安裝工具；**==** 後面是指定的版本。要先裝 torch，再裝其他套件（下一頁）

安裝其餘套件並確認 GPU

```
# 1. 先升級 pip，再安裝其餘套件（版本都寫在 requirements.txt 裡）
> python -m pip install --upgrade pip
> pip install -r requirements.txt
# 2. 確認 PyTorch 看得到 GPU
> python -c "import torch; print(torch.__version__, torch.cuda.is_available())"
2.11.0+cu128 True
```

1. **python -c "..."**：直接執行引號裡的一小段 Python
2. **torch.__version__**：版本，+cu128 表示 CUDA 12.8 版（舊卡為 +cu126）
3. **torch.cuda.is_available()**：GPU 能不能用，True 可以、False 不行

出現 False？常見三個原因

① 忘了啟用虛擬環境（前面沒有 (.venv)）

② 裝到 CPU 版的 torch：先 **pip uninstall torch torchvision**，再照上一頁重裝

③ 顯示卡驅動太舊：更新 NVIDIA 驅動

把資料放進 data/

```
data/
├── train/
│   ├── EO/image/*.jpg   EO/label/*.json
│   └── IR/image/*.jpg   IR/label/*.json
├── test/
│   ├── EO/image/*.jpg
│   └── IR/image/*.jpg
├── make_submission.py          (evaluation kit)
├── test_images.csv             (evaluation kit)
└── sample_submission.json      (evaluation kit)
```

EO＝一般影像 1920x1080；**IR**＝熱影像 640x512；**label**＝標註檔（用 labelme 工具標的，含 reason）

最常見的錯誤：資料夾多一層

`data\<壓縮檔名>\train\EO`
`data\train\train\EO`

Windows 的「解壓縮全部」會先多建一個以壓縮檔命名的資料夾。請確認 data\train、data\test 底下**直接**就是 EO、IR

訓練集壓縮檔也附了 make_submission.py、sample_submission.json：
請用 **evaluation kit 的版本覆寫掉**

25　準備環境

一行程式檢查資料

```
# README 第 2 節那一行，拆成多行方便看
import os
[print(d,
       len(os.listdir(os.path.join('data', d)))
       if os.path.isdir(os.path.join('data', d))
       else '找不到！請檢查資料夾是否多一層或放錯位置')
 for d in ['train/EO/image', 'train/EO/label',
           'train/IR/image', 'train/IR/label',
           'test/EO/image', 'test/IR/image']]
```

應該看到：train 的 4 個資料夾各 **1046**、
test 的 2 個資料夾各 **696**；
出現「找不到！」就回上一頁檢查

1. **os.path.join('data', d)**：把 data 和資料夾名稱接成一個路徑
2. **os.path.isdir(...)**：檢查資料夾在不在；不在就走 else，印出「找不到！…」
3. **len(os.listdir(...))**：os.listdir 列出資料夾裡的檔案，len 數有幾個
4. **[... for d in [...]]**：list comprehension（串列生成式），把 6 個資料夾逐一做同樣的事

實際要打的是 README 第 2 節的一行版本
python -c "import os; ..."，在專案根目錄執行

PART 4

快速開始

一行指令重現官方 baseline｜看懂輸出｜上傳

27　快速開始

一行指令重現官方 baseline

```
> python pipeline.py all --reason-model weights/qwen_reason
```

all＝依序跑下面 7 個步驟；**--reason-model weights/qwen_reason**＝成因模型改用附的、已微調好的權重

ir-norm IR 偽彩色轉灰階 → **detect** 偵測 → **dets** 轉檔與後處理 → **sft-data** 建 SFT 資料 → **train-sft** 不重訓，略過 → **eval** 驗證 → **submit** 產生 submission

約 3 分鐘

RTX 5060 Ti 16 GB 實測

不訓練任何模型，直接用附的權重

偵測器：`prb/runs/train/pv/weights/best.pt`
成因模型：`weights/qwen_reason/`

28 快速開始

看懂輸出、檢查格式

```
# 跑完之後，最後幾行會看到類似：
已寫入 ...\outputs\submission.json（1392 張影像、65185 個物件）
...
完成：ir-norm → detect → dets → sft-data → train-sft → eval → submit
# 接著用 evaluation kit 檢查格式
> python data/make_submission.py --check outputs/submission.json
格式檢查通過：評分程式會接受這個檔案。
```

英文警告：正常

例如 fix_mistral_regex=True 那行，
不影響結果，不要改程式

「調和平均 0.000」：正常

eval 步驟印的，不代表壞掉；
講 eval 時會解釋

格式檢查通過

代表可以被正常評分，
不代表分數高低

確認重現成功並上傳

```
PowerShell   > Get-FileHash outputs\submission.json
cmd          > certutil -hashfile outputs\submission.json SHA256
Linux        $ sha256sum outputs/submission.json
```

官方 baseline 提交檔的 SHA-256（檔案的「指紋」，英文大小寫不拘）

`7a45f309436cfbe2e3b53760ad799d85a866217afd14efe3809a6f664ab5096e`

一樣＝重現成功

換顯卡、驅動或用 CPU 時不同是正常的，格式檢查通過就能上傳

上傳到 AIdea：outputs/submission.json

上傳規則與每日次數以競賽頁為準

注意：--reason-model 只對這一次指令有效（例：調門檻後重出提交檔，下面兩行要打成同一行）

```
> python pipeline.py dets submit --conf "solar_panel=0.3,hotspot=0.25,crack=0.25"
  --reason-model weights/qwen_reason
```

PART 5

逐步講解程式

轉灰階｜偵測｜過濾與後處理｜做訓練資料｜微調 Qwen｜驗證｜產生提交檔

31 逐步講解程式

專案裡有什麼

```
aicup_pv_baseline/
├── pipeline.py                # 總指揮：依序呼叫其他程式
├── data/                      # 訓練集、測試集、evaluation kit
├── weights/qwen_reason/       # 官方微調好的 Qwen
├── tools/                     # 小工具
│   ├── ir_gray.py             # 偽彩色熱影像 → 灰階
│   └── labelme2yolo.py        # 標註 → YOLO 格式
├── prb/                       # 偵測器（YOLOv7 架構）
│   └── runs/train/pv/weights/best.pt   # 訓練好的權重
├── reason/                    # 成因相關程式（7 支）
│   ├── common.py  yolo2dets.py
│   ├── build_sft_data.py  train_sft.py
│   └── reasoner.py  eval_val.py  infer_submission.py
└── outputs/                   # 所有產出（執行後才出現）
```

資料夾和步驟的對應

tools/	4-1 ir-norm：熱影像轉灰階
prb/	4-2 detect：用偵測器找框
reason/	4-3 dets 到 4-7 submit
outputs/	每一步的產出，重跑會覆寫

pipeline.py 自己不做運算，
只負責**依序呼叫**這些程式（下一頁）

32　逐步講解程式

pipeline.py：整個流程的總指揮

```
# ① 預設要跑的步驟，依序執行
ALL = ["ir-norm", "detect", "dets",
       "sft-data", "train-sft", "eval", "submit"]
# ② 執行另一支程式（例如 detect.py）
def run(cmd, cwd=ROOT):
    subprocess.run(cmd, cwd=cwd, check=True, ...)
# ③ 一個步驟接一個步驟跑
for step in steps:
    ...
    elif step == "ir-norm":
        ...
        run([py, ..., "--images", src, "--out", ir_gray])
    elif step == "detect":
        ...
        run([py, "detect.py", ...], cwd=PRB)
```

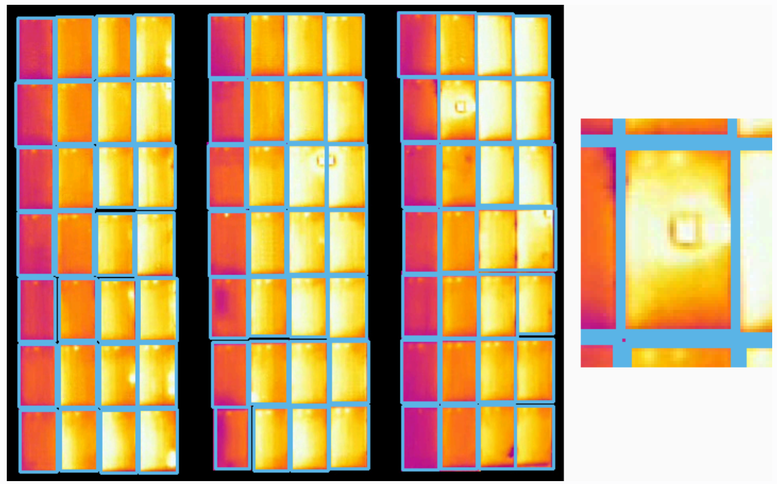
1. 打 **python pipeline.py all** 時，
會依照 ALL 的順序跑 7 個步驟

2. pipeline.py 自己不做運算，
而是用 run() 呼叫其他程式；
check=True 表示那支程式出錯就整個停下來

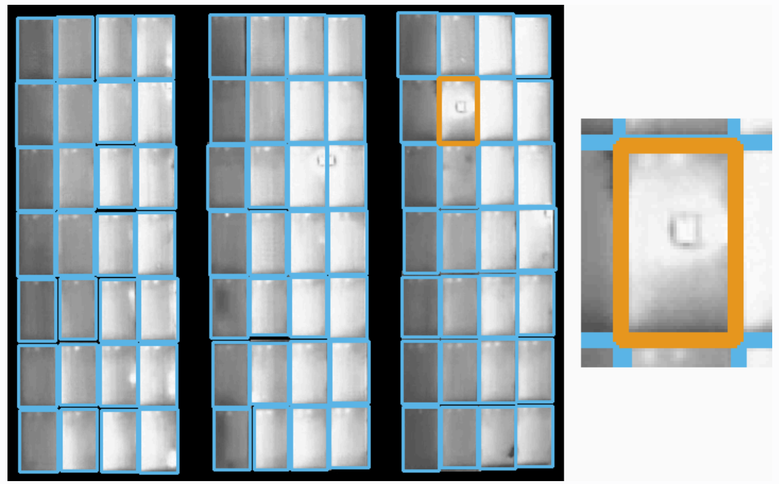
3. 每個步驟對應一段 if／elif，
所以也能只跑其中幾步，例如
python pipeline.py ir-norm detect

4-1 ir-norm：為什麼要把熱影像轉成灰階

訓練集的熱影像全是灰階（R=G=B）；測試集有一部分是偽彩色，用顏色表示溫度



原本的偽彩色：**0 個熱點**；右側是右邊那組第 2 排第 2 片的放大



轉成灰階後：同一個偵測器把這片框成**熱點**，**橘框**框住整片面板

在所有偽彩色的測試熱影像上，轉換前偵測器一個熱點都沒框到，轉換後才開始框出熱點。圖中的框都是模型的預測，不是標準答案，也不是每個可疑的亮點都會被框到

34　逐步講解程式

ir_gray.py 怎麼把顏色變回灰階



上：常見的熱影像配色（示意），越熱越亮

下：同樣的顏色用亮度公式算出的灰階

灰階 = 0.299 × R + 0.587 × G + 0.114 × B

1　判斷是不是彩色

灰階影像的 R、G、B 一樣；
三個值平均差距大於 2 就當成彩色

2　彩色的用亮度公式轉灰階

固定的標準公式，每張影像各自轉換，
不從測試資料學任何參數

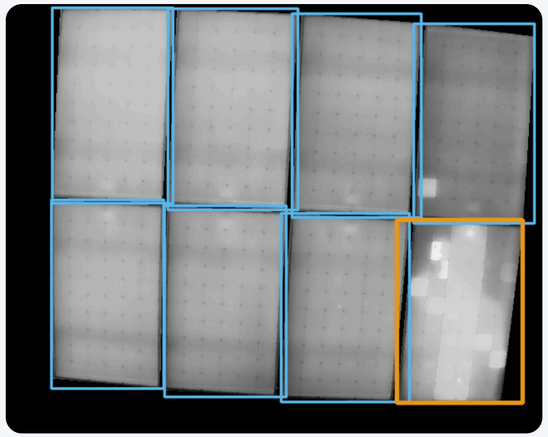
3　灰階的原樣複製

三個通道放一樣的值，和訓練集的灰階熱影像一致

```
# tools/ir_gray.py
def is_color(im):
    px = im.astype(np.int16)
    # 每個像素 R、G、B 最大減最小，再取平均
    return float((px.max(2) - px.min(2)).mean()) > COLOR_TOL

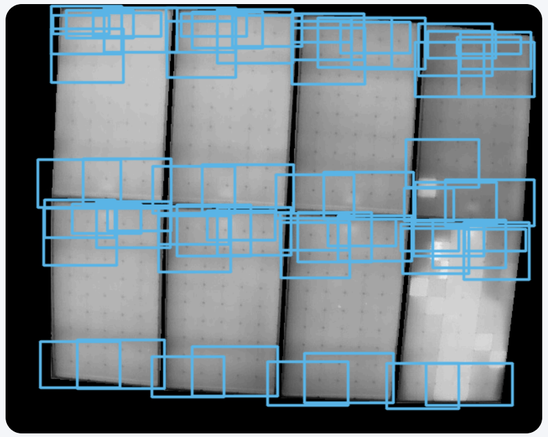
def to_gray(im):
    # 亮度公式（OpenCV 的顏色順序是 BGR）
    g = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
    return cv2.merge([g, g, g])
```

偵測器是怎麼學會框東西的



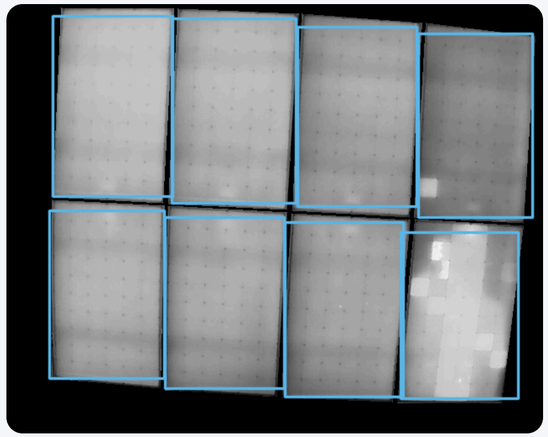
1　標準答案

面板 7、熱點 1，
這就是模型要學的答案



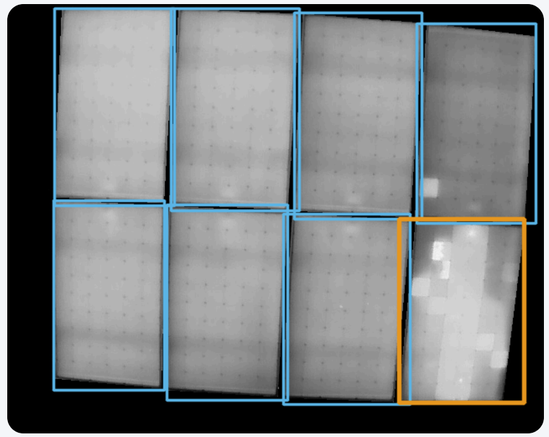
2　只練 1 輪

63 個框擠在一起，
沒有一個和答案對上



3　練 10 輪

面板大致框對了，
但發熱的那片還被當成一般面板



4　附的權重：200 輪

8 個框全部對上，
熱點的 IoU 是 0.99

每一輪：模型先猜框 → 和答案比較算出 **loss（誤差）** → 調整參數。整份訓練資料看一遍＝**1 輪（epoch）**

藍＝面板、橘＝熱點。② ③ 來自我們另外跑的同一次訓練；④ 是附的權重，是另一次訓練

4-2 detect：用偵測器找框

```
# 4-2：偵測測試集的 EO 與 IR（約 1.5 分鐘）
> python pipeline.py detect
```

1. 用附的 **best.pt**（PRB-FPN-CSP、輸入 1280）分別偵測 EO 與 IR
2. IR 用的是 4-1 轉好的灰階圖
3. 先用很低的門檻 **0.05** 把框全部存起來，下一步再過濾，所以之後調門檻**不用重跑這一步**
4. 每張影像的框存成同名的 .txt，放在 **outputs/detect/test_EO/labels/**（IR 放在 test_IR/labels/）

```
# outputs/detect/test_EO/labels/test_001.txt（示意）
2 0.4875 0.312 0.041 0.118 0.8931
```

2	0.4875	0.312	0.041	0.118	0.8931
類別編號	中心 x	中心 y	寬	高	信心分數

類別編號：0＝crack 破裂、1＝hotspot 熱點、2＝solar_panel 面板

中心 x、y 與寬、高：除以影像的寬、高之後的比例，都在 0～1 之間

信心分數：模型覺得這裡真的有東西的把握，也在 0～1 之間

37　逐步講解程式

把一行數字變成框（yolo2dets.py）

```
# ① 影像的寬 W、高 H（像素）
W, H = im.size
...

# ② 一行切成一個個數字；v[1:5] 是 xc, yc, w, h
v = line.split()
...
xc, yc, w, h = (float(x) for x in v[1:5])

# ③ 比例 × 寬高 = 像素；左邊 = 中心 - 寬的一半
x0, x1 = max(0.0, (xc - w / 2) * W), min(W, (xc + w / 2) * W)
y0, y1 = max(0.0, (yc - h / 2) * H), min(H, (yc + h / 2) * H)
```

算算看：IR 寬 W = 640

xc = 0.5、w = 0.1（寬的一半 = 0.05）

x0 = (0.5 − 0.05) × 640 = **288**

x1 = (0.5 + 0.05) × 640 = **352**



max／min：讓框不超出影像

框太靠左時，x0 會算出負數，
max(0.0, …) 把它改成 0；
超過右邊時，min(W, …) 把它改成 W

38 逐步講解程式

4-3 dets：依類別門檻過濾

```
# 4-3：轉檔、依類別門檻過濾、後處理（11 秒）→ outputs/detections.json
> python pipeline.py dets --conf "solar_panel=0.25,hotspot=0.25,crack=0.25"
```

```
# ① 把 --conf 字串拆成每一類的門檻
conf = parse_conf(args.conf)
...
# ② 第 1 個數字 → 類別名稱；第 6 個 → 信心分數
label = names[int(float(v[0]))]
c = float(v[5]) if len(v) > 5 else 1.0
# ③ 查這一類的門檻（沒給就用 0.25），太低就跳過
if c < conf.get(label, 0.25):
    continue
```

評分沒有信心分數，**送出的每個框都算數**，門檻影響很大

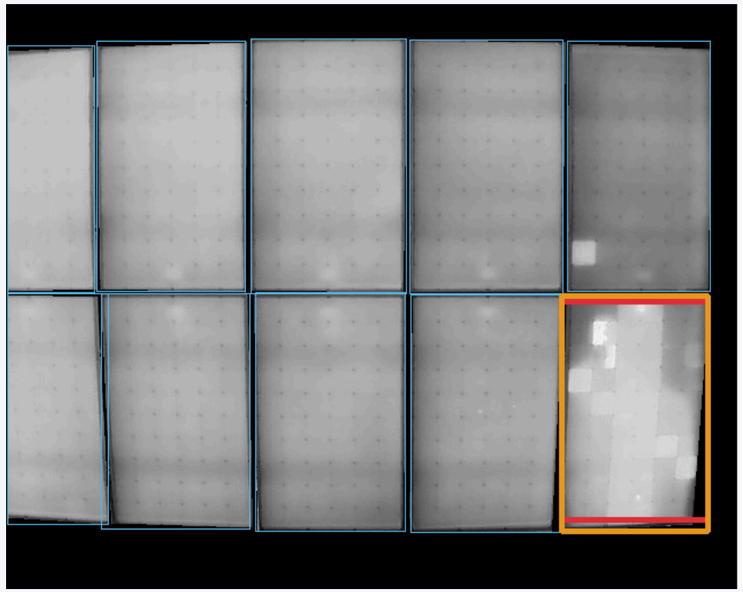
重點：改門檻不用重跑偵測

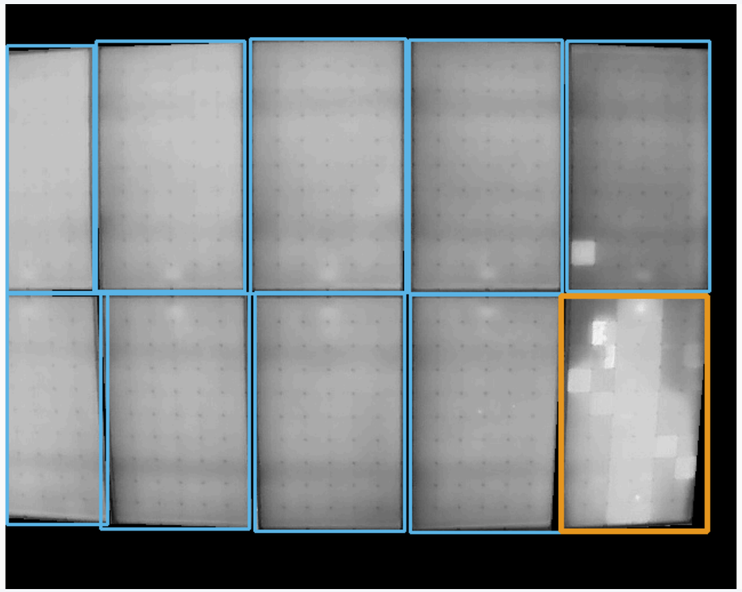
4-2 已經用 0.05 把框都存好了；
但要**再跑 submit**，提交檔才會更新

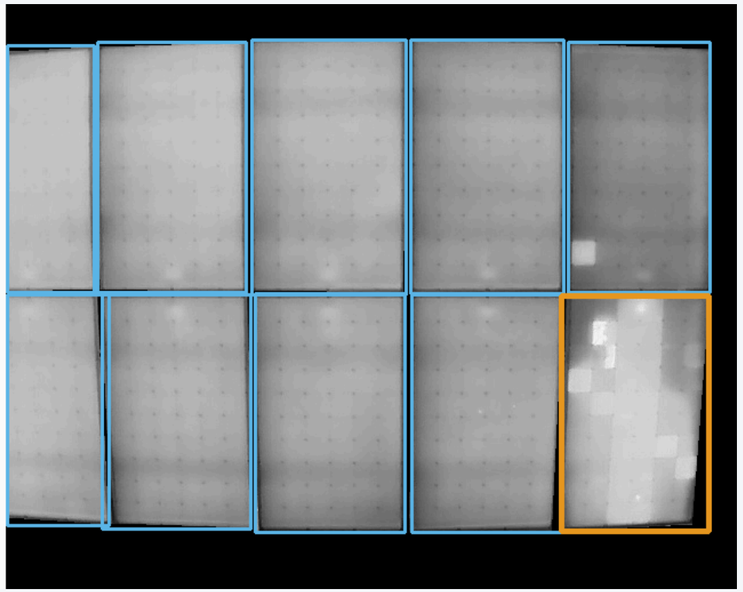
```
python pipeline.py dets submit --conf "..."
```

還沒自己訓練 Qwen 的話，再加
--reason-model weights/qwen_reason

從候選框到最終框：同一張圖的三個階段







① 信心 ≥ 0.05：全部候選

面板 9、熱點 1、破裂 1

② 信心 ≥ 0.25：dets 門檻

面板 9、熱點 1（低信心的破裂框被拿掉）

③ 後處理後＝最終框

面板 9、熱點 1，和答案一樣

後處理：用訓練集的規律清掉不可能的框

```
# 每種影像「可能出現」的類別
ALLOWED = {"EO": {"solar_panel", "crack"},
           "IR": {"solar_panel", "hotspot"}}
...
def postprocess(objs, mod):
    ...
    # ① 只留下 ALLOWED 裡的類別
    objs = [o for o in objs if o["label"] in ALLOWED[mod]]
    anom = [o["_box"] for o in objs if o["label"] in ANOMALY]
    # ② 和異常框重疊（IoU ≥ 0.5）的面板框刪掉
    return [o for o in objs if not (
        o["label"] == "solar_panel" and
        any(_iou(o["_box"], a) >= 0.5 for a in anom))]
```

1 不可能的類別直接拿掉

訓練集裡，**熱點**只出現在 IR（444/444），
破裂只出現在 EO（352/352）

2 一片面板只留一個框

訓練集裡，異常的標註直接取代那片面板：
兩者 IoU ≥ 0.5 的只有 1/796

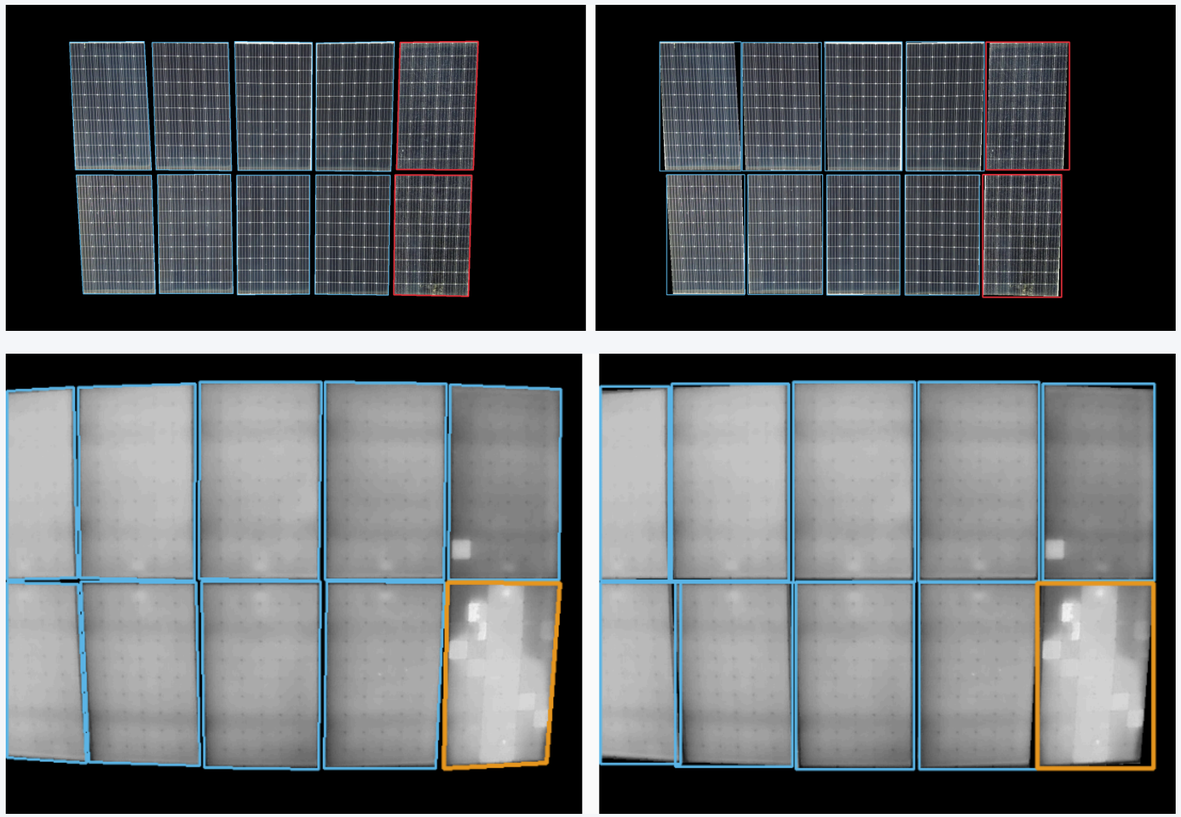
→ 異常框位置上的面板框刪掉，
避免同一片面板被算兩次

ANOMALY 就是 hotspot 和 crack
（定義在 reason/common.py）

41　逐步講解程式

偵測結果 vs 標準答案

左：標準答案（訓練標註）　右：模型預測



藍＝面板、紅＝破裂、橘＝熱點。訓練集影像 train_0419；
訓練資料裡有很多相似場景，這是示範輸出，不是成績

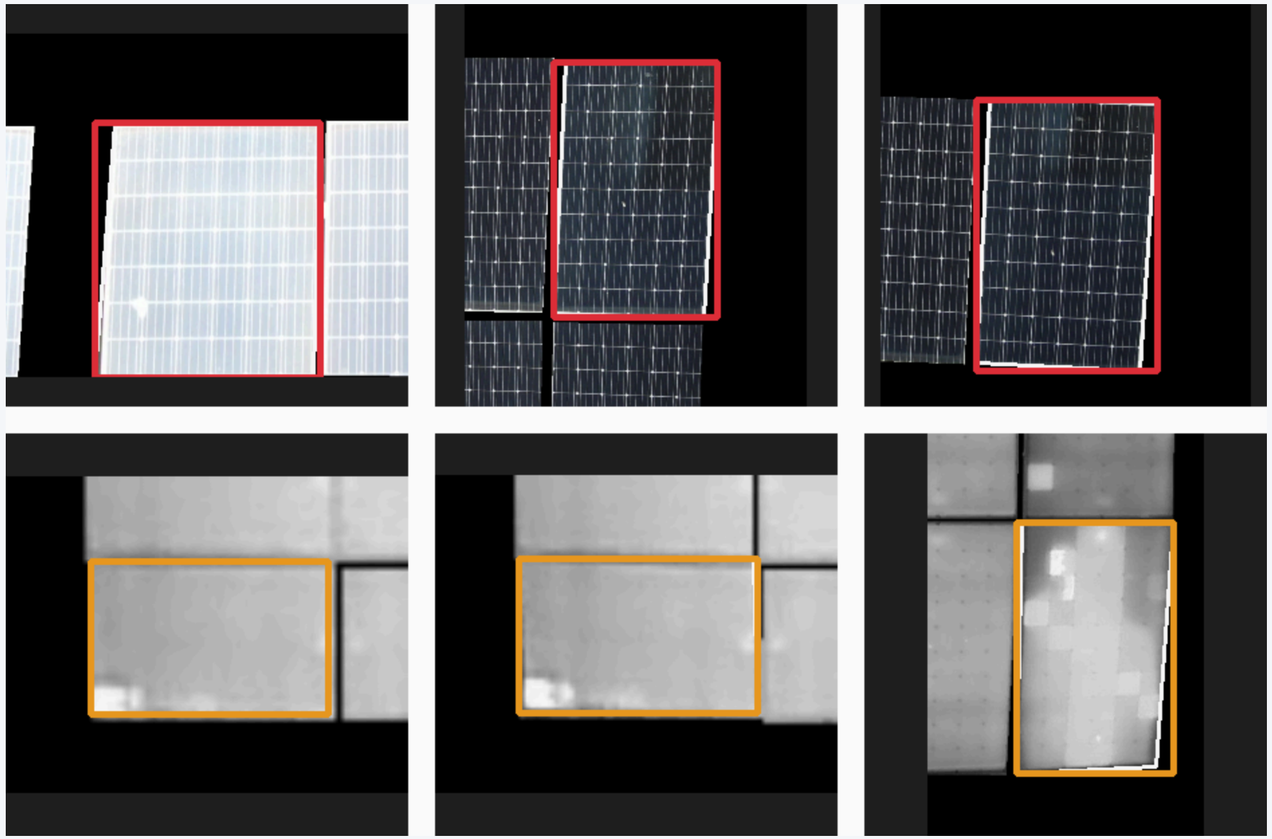
EO（上）

答案 10 個框，模型 10 個框，全部配對成功；
2 個破裂的 IoU 是 0.977、0.973

IR（下）

答案 10 個框，模型 10 個框，全部配對成功；
熱點的 IoU 是 0.995

瑕疵特寫：預測框和答案幾乎重合



白色細線＝標準答案
紅框、橘框＝模型預測

特寫	信心分數	和答案的 IoU
上排破裂	0.75、0.94、0.93	1.00、0.99、0.99
下排熱點	0.92、0.84、0.81	1.00、1.00、1.00

信心分數是**模型自己覺得多確定**；
IoU 是**實際和答案重疊多少**。
左上那個破裂信心只有 0.75，位置卻幾乎完全正確

4-4 sft-data：把標註變成對話

system｜角色設定

你是太陽能電廠的無人機巡檢工程師……用一句繁體中文說明該瑕疵最可能的成因。

user｜題目

由 describe() 產生
（第 45 頁講）

影像類型：熱影像(IR)，640x512
待說明瑕疵：熱點(hotspot)
瑕疵位置：中心(84%, 46%)，寬佔 25.5%、高佔 49.2%，長寬比 1.5
瑕疵面積約為單片面板的 1.65 倍，與 0 片面板重疊
同一張影像：面板 7 片、hotspot 1 個、crack 0 個
同編號可見光(EO)影像：面板 4 片、hotspot 0 個、crack 1 個；異常位置 crack(73%, 49%)
請說明此瑕疵的可能成因。

assistant｜標準答案

推測破裂處導電路徑受損，電阻升高而累積熱量造成熱點。 ← 標註裡的 reason

```
> python pipeline.py sft-data  # 實測約 1 秒
```

每個有 reason 的熱點／破裂標註都變成一筆對話；輸出 **outputs/sft_data/** 裡的 **train.jsonl**、**val.jsonl**、**candidates.json**

44　逐步講解程式

SFT 的訓練資料：題目 → 標準答案

題目裡的關鍵線索	標準答案（訓練集標註的 reason）	近似成因
EO 破裂；同編號 IR 有熱點	面板出現裂損，導電路徑於該處受阻。	破裂影響導電
IR 熱點；同編號 EO 有破裂	模組板破裂造成電流分布不均，局部電阻上升並形成異常發熱區。	模組板破裂
IR 熱點；EO 沒有異常；面板 96 片	可能為內部隱裂、接觸電阻上升或電性異常，進而引發局部過熱。	隱裂或電性異常
EO 破裂；同編號 IR 沒有異常	模組表面出現明顯裂紋，推測為外部物理衝擊所造成。	外力撞擊

模型從 2,340 筆這樣的對話（含擴增）學會：看線索 → 判斷成因 → 寫成一句話

範例取自訓練集 train_0001、0005、0007；近似成因由 approx_group() 分群，非官方分類

describe()：把框變成一段描述

```
lines = [
    "影像類型：{}，{}x{}".format(MOD_DESC[mod], W, H),
    "待說明瑕疵：{}".format(LABEL_DESC[target["label"]]),
    "瑕疵位置：中心{}，寬佔 {:.1f}%、高佔 {:.1f}%，長寬比 {:.1f}".format(
        _pos(target["box"], W, H), w / W * 100, h / H * 100, ...),
]
med = _panel_med(objs)
...
    lines.append("瑕疵面積約為單片面板的 {:.2f} 倍...".format(w * h / med, n_ov))
...
c = _counts(objs)
lines.append("同一張影像：面板 {} 片、hotspot {} 個、crack {} 個".format(...))
...
    s = "同編號{}影像：面板 {} 片、hotspot {} 個、crack {} 個".format(...)
```

1. 位置用百分比：中心、寬、高都除以整張圖的寬高
2. 面積 ÷ med（面板面積的中位數）＝幾片面板大
3. 數同一張影像的面板、熱點、破裂各有幾個
4. 看同編號另一張影像有沒有異常、在哪裡

訓練和推論（用模型產生答案）共用 describe()，
兩邊的描述格式才一樣

為什麼要看另一張影像

另一張影像有沒有異常，是判斷成因的重要線索

情境（訓練集標註統計）	成因（近似群組）	筆數
EO crack，同編號 IR **有 hotspot**	破裂影響導電	285 / 308
IR hotspot，同編號 EO **有 crack**	模組板破裂	191 / 215
IR hotspot，EO **無異常**，面板約 73 片	隱裂或電性異常	190
IR hotspot，多數 EO **無異常**，面板約 17 片	旁路二極體 / 電池串	38
IR hotspot，EO **有 crack**，面板 9 片	電池片裂紋	17
EO crack，IR **無異常**	表層局部破損 35、外力撞擊 8（另 1 筆破裂影響導電）	44

「285 / 308」表示這個情境共 308 筆，其中 285 筆是這個成因。成因群組由 approx_group() 關鍵字規則近似，不是官方分類。資料來源：README 第 9 節

47　逐步講解程式

做訓練資料的三個小技巧

1 切驗證集

依影像編號隨機切 15% 當驗證集，
同編號的 EO 和 IR 一定在同一邊

注意：沒有依 train_groups.csv 分組。
同一段連續拍攝的畫面很像，
驗證數字會偏樂觀

2 模擬偵測誤差

每筆訓練資料多做幾份抖動版：
框座標小幅移動、隨機漏掉少數面板

原因：測試時用的是偵測器的框，不是
人工標註的框，本來就有誤差

3 少數成因過採樣

筆數少的成因群組多做幾份抖動版，
補到約 150 筆

由 build_sft_data.py 的 --min-group
控制（pipeline.py 沒有這個參數）

```
# reason/build_sft_data.py：box_noise＝框座標最多移動框寬高的 2%，drop_panel＝每片面板有 5% 機率被拿掉
def jitter(objs, W, H, rng, box_noise=0.02, drop_panel=0.05):
    """模擬偵測器誤差：框座標小幅抖動、少數面板漏檢。異常框不丟，因為它們是成因判斷的關鍵。"""
```

4-5 train-sft：微調 Qwen

```
> python pipeline.py train-sft
# 產出 outputs/qwen_reason/final/，預設設定約 46 分鐘
```

第一次執行會從 Hugging Face
下載基底模型（約 1 GB），需要網路

項目	預設值	白話說明
基礎模型	Qwen/Qwen2.5-0.5B-Instruct（版本 7ae55760…）	會說中文的小型語言模型，約 5 億個參數
方式	全參數微調（可改 --lora）	所有參數一起調；LoRA 只訓練一小組外加的參數
epoch／batch	4／16	整份資料看 4 遍；每次一起算 16 筆
學習率	2e-5、cosine、warmup 5%	每步改多少（0.00002）；先慢慢調大，再慢慢調小
精度	RTX 30 以後 bf16、較舊的卡 fp16	用比較短的數字格式，省記憶體、算得快
挑模型	每個 epoch 用驗證 loss 挑最好	每看完一遍考一次驗證集，留下 loss（誤差）最低的

49　逐步講解程式

只對回答算 loss

```
# 題目（prompt）和回答分開切成 token 編號，再接起來
p_ids = tok(prompt, add_special_tokens=False)["input_ids"]
c_ids = tok(full[len(prompt):].rstrip("\n"), add_special_tokens=False)["input_ids"]
ids = (p_ids + c_ids)[:max_len]
labels = ([-100] * len(p_ids) + c_ids)[:max_len]
```

	p_ids：題目（system＋user）切成的 token	c_ids：回答切成的 token
ids（輸入）	你是太陽能電廠的……請說明此瑕疵的可能成因。	推測破裂處導電路徑受損，…
labels（答案卷）	-100 -100 -100 … -100　不算分	推測破裂處導電路徑受損，…　要學

就像改考卷：只批改答案，不批改抄題目

token＝模型讀文字時切成的小片段
-100＝PyTorch 約定的「這格不算」記號

顯示記憶體與速度：怎麼選設定

設定	配置峰值	保留峰值	訓練速度	4 個 epoch 約需
預設（batch 16）	16.2 GB	25.9 GB	每秒 3.3 筆	46 分鐘（完整跑過）
--bs 8	9.5 GB	15.7 GB	每秒 19.9 筆	8 分鐘
--bs 4	6.2 GB	11.2 GB	每秒 17.7 筆	9 分鐘
--lora（batch 16）	15.8 GB	24.8 GB	每秒 16.9 筆	9 分鐘
--lora --bs 8	8.4 GB	16.5 GB	每秒 21.5 筆	7 分鐘
--lora --bs 4	4.8 GB	8.8 GB	每秒 16.9 筆	9 分鐘

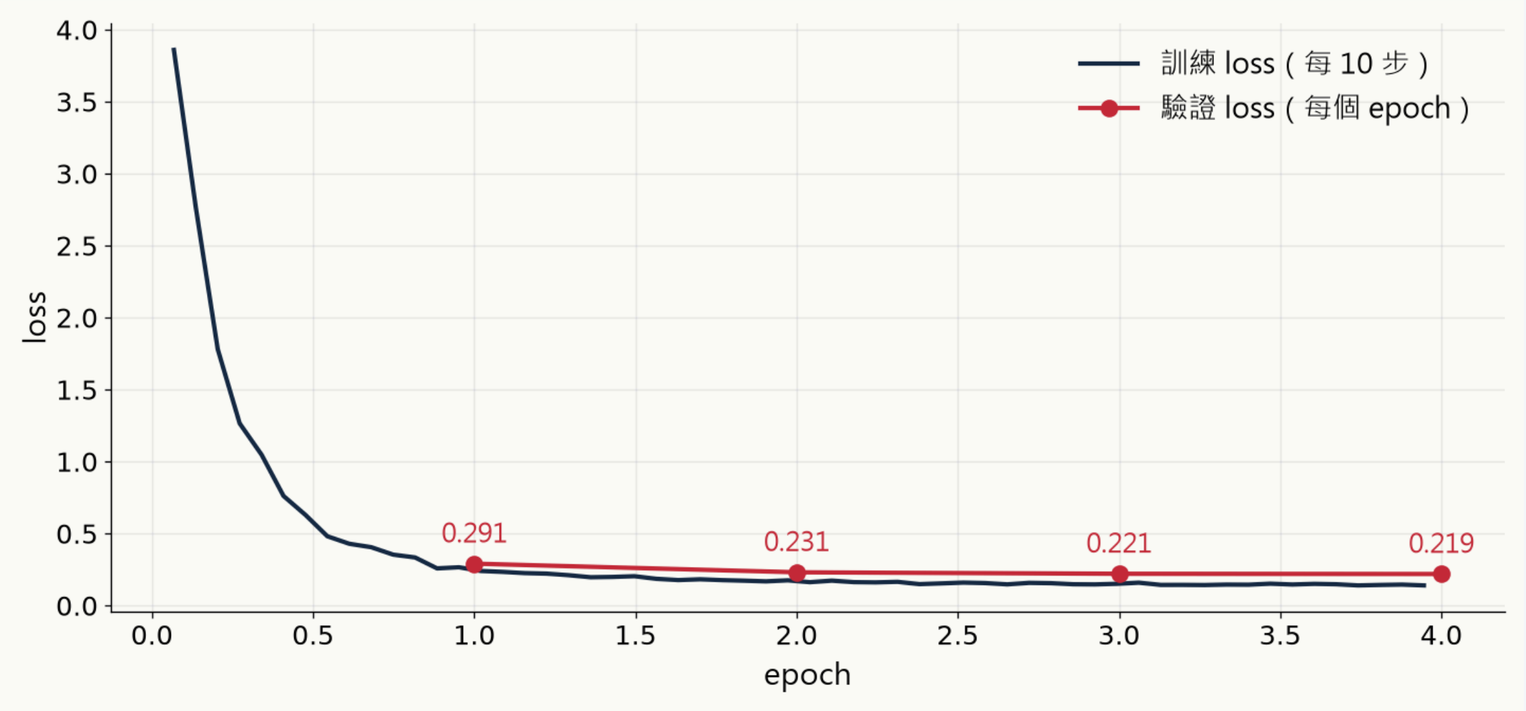
RTX 5060 Ti 16 GB 實測，每種設定試跑約 0.12 個 epoch。配置峰值＝實際用到的量，保留峰值＝再加上 PyTorch 的快取，實際需要介於兩欄之間。
預設 batch 16 在 16 GB 的卡上放不下，Windows 會借用系統記憶體，所以特別慢。

建議：16 GB 用 --bs 8、12 GB 用 --bs 4、
8 GB 試 --lora --bs 4（依上表推估）

注意：改了 --bs、--lora 等設定，
訓練出的模型會和官方不同，分數也會不同

51　逐步講解程式

訓練時怎麼看 loss



官方 baseline 微調 Qwen 時的紀錄：訓練 loss 每 10 步記一次，驗證 loss 每個 epoch 算一次

1. loss＝模型寫的回答和標準答案差多少，**越低越接近**

2. 訓練 loss 一開始很高（3.86），很快往下掉，最後約 0.14

3. 驗證 loss：
   0.291 → 0.231 → 0.221 → 0.219

4. 每個 epoch 結束比一次，最好的一版存到 **final/**

微調前 vs 微調後：同一題，回答差很多

題目：EO 破裂；同編號 IR 有熱點

微調前：這可能是由于材料在使用過程中受到損傷或老化導致的裂紋。熱影像和冷影像顯示出的裂紋大小和位置不一致，……（後面還有一大段）

微調後：模組板結構破裂，電流路徑的完整性受到破壞。

標準答案：面板破裂明顯，損傷已及於內部導電層。

題目：IR 熱點；EO 沒有異常；面板 95 片

微調前：這可能是由于熱點在照片中被過濾了，導致其在照片上不清晰或無法辨識。……（後面還有一大段）

微調後：推測係電池片內部微裂或接觸電阻上升，於運轉時形成異常溫升。

標準答案：研判該處因隱裂或電性異常使電阻上升，運轉時產生明顯熱點。

微調前：答非所問、一大段、還夾雜簡體字

微調後：一句話、用詞像標註，判斷的成因和答案相同

4-6 eval：看成因判斷準不準

```
> python pipeline.py eval  # 實測約 11 秒
# 還沒訓練自己的 Qwen：後面加 --reason-model weights/qwen_reason
```

方法	group acc	旁路二極體或電池串	電池片裂紋	表層局部破損	調和平均
純規則 baseline	0.895	0/7	0/3	6/9	0.000
附的 Qwen	0.902	6/7	3/3	0/9	0.000

group acc＝成因群組猜對的比例。純規則 baseline＝只看「同編號另一張有沒有異常」來猜。用附的權重（用法 A）跑 eval 會印出同樣的數字

1. 只要一個群組全錯，調和平均就是 0，所以 **0.000 不代表壞掉**
2. 成因群組是關鍵字規則近似的，**這不是官方分數**
3. 驗證集只有 133 筆，數字有雜訊，只能當參考

4-7 submit：三種產生 reason 的方式

snap（預設）

Qwen 先寫一句，再換成候選句（訓練集出現過的句子）中最像的一句

rank

不生成，直接在候選句中挑 Qwen 認為機率最高的一句

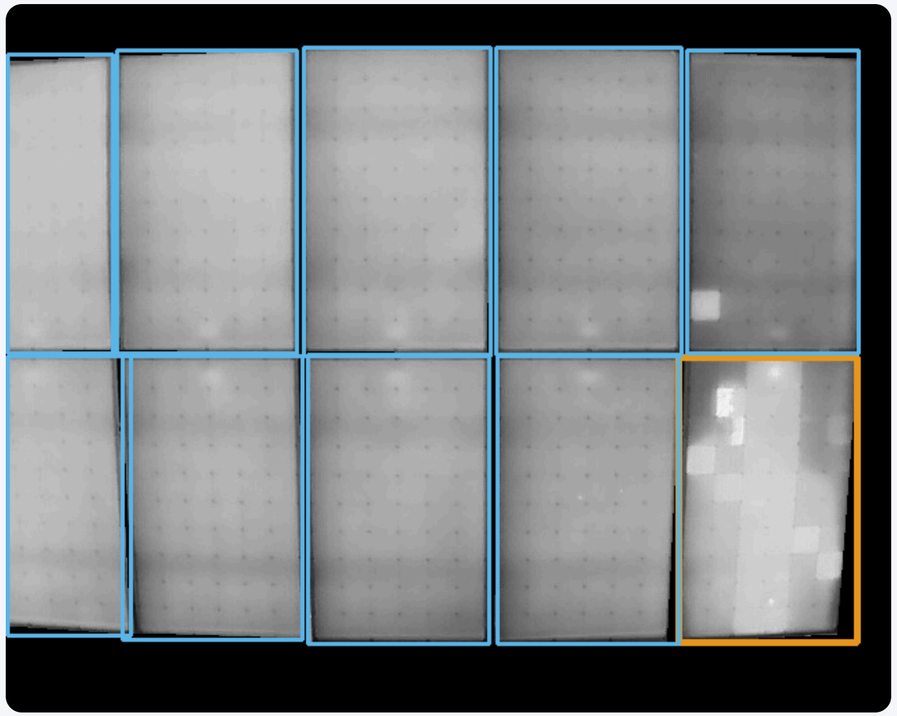
generate

直接用 Qwen 寫的句子，可能是訓練集沒有的新句子

```
def snap(text, candidates):
    """貼齊到字面最相近的候選句。"""
    if not candidates:
        return text
    if text in candidates:
        return text
    return max(candidates, key=lambda c: difflib.SequenceMatcher(None, text, c).ratio())
```

ratio() 兩句話有多像的分數：0 ＝完全不像，1 ＝一模一樣；snap 會挑分數最高的那一句

一個熱點的完整旅程 ①：偵測 → 描述



訓練集驗證切分 train_0419（IR）

1. 偵測：右下角橘框是 hotspot，信心 0.81，和標準答案的 IoU 是 0.995
2. describe() 把這個框和整組 EO／IR 的偵測結果寫成題目：

```
影像類型：熱影像(IR)，640x512
待說明瑕疵：熱點(hotspot)
瑕疵位置：中心(86%, 70%)，寬佔 20.2%、高佔 40.2%，長寬比 1.6
瑕疵面積約為單片面板的 0.98 倍，與 0 片面板重疊
同一張影像：面板 9 片、hotspot 1 個、crack 0 個
同編號可見光(EO)影像：面板 8 片、hotspot 0 個、crack 2 個；異常位置
crack(74%, 31%)、crack(74%, 71%)
請說明此瑕疵的可能成因。
```

一個熱點的完整旅程 ②：寫出原因 → 提交檔

3. Qwen 生成：**模組板破損使電流路徑中斷，局部電阻增加後形成明顯溫升。**
4. snap 對齊：這句本來就是候選句（訓練集出現過的句子），所以不變
5. 寫進 submission.json 的物件：

```
{"label": "hotspot",
 "points": [[486.0, 255.0], [615.0, 255.0], [615.0, 461.0], [486.0, 461.0]],
 "reason": "模組板破損使電流路徑中斷，局部電阻增加後形成明顯溫升。"}
```

標準答案：該處溫度異常偏高，推測與模組板破裂造成的導電路徑損傷有關。

措辭不同，但判斷的成因相同：**EO 上看到破裂 → 模組板破損 → 導電受損 → 發熱**

產生提交檔的程式

```
for k, os_ in objs.items():
    sub[k] = []
    for i, o in enumerate(os_):
        rec = {"label": o["label"], "points": o["points"]}
        if o["label"] in ANOMALY:
            rec["reason"] = rmap[(k, i)] or cands[o["label"]][0]  # 空字串會記 0 分，至少給一句
        sub[k].append(rec)
```

1. **面板**：只有 **label**（類別）和 **points**（框的四個角）
2. **熱點、破裂**（ANOMALY）：多一個 **reason**
3. 空字串時用 **or** 補上候選句第一句（空白記 0 分）

```
> python pipeline.py submit
# 執行 reason/infer_submission.py，約 26 秒
# 寫出 outputs/submission.json
```

注意 程式內建的自我檢查只涵蓋部分規則；上傳前一定要跑官方的 **make_submission.py --check**（第 59 頁）

官方 baseline 的提交檔長這樣（片段）

```
{ ...
  "test/EO/image/test_081.jpg": [
    {"label": "solar_panel", "points": [[1833.0, 0.0], [1920, 0.0], [1920, 723.0], [1833.0, 723.0]]},
    {"label": "crack", "points": [[1463.0, 15.0], [1824.0, 15.0], [1824.0, 671.0], [1463.0, 671.0]],
     "reason": "模組表面出現明顯裂紋，推測為外部物理衝擊所造成。"},
    ...（這張影像共 8 個物件：面板 7、破裂 1）
  ],
  ...
}
```

整份檔案：**1,392 張影像**、**65,185 個物件**

面板只有 label 和 points；
熱點、破裂多一個 reason

點座標是像素；
1920 是 EO 影像的右邊界

這是模型在測試影像上的預測，不代表正確答案；用法 A 會產生一模一樣的檔案

4-8 檢查格式、調門檻、最終版

1 檢查格式

上傳前一定要做

```
> python data/make_submission.py --check outputs/submission.json
格式檢查通過：評分程式會接受這個檔案。
```

2 調門檻後重出

不用重跑偵測

```
> python pipeline.py dets submit --conf "..."
# 還沒自己訓練 Qwen：後面再加 --reason-model weights/qwen_reason
```

3 最終版

驗證集也拿去訓練

```
> python pipeline.py sft-data train-sft submit --val-ratio 0 --bs 8
# 例：調參時用了 --bs 8。調參時用過的參數都要原樣加上
```

加了 **--val-ratio 0** 就沒有驗證集：不挑最佳 epoch，直接用最後一個 epoch（eval 也會略過）

會覆寫 outputs/qwen_reason/ 和 outputs/submission.json，舊檔請先另存；跑完一樣要做第 1 步的檢查

PART 6

進階與練習

重訓偵測器｜改進方向｜常見問題｜小練習

（選做）重新訓練偵測器

```
# 1. 標註轉成 YOLO 格式（outputs/yolo_ds/）
> python pipeline.py yolo-data
# 2. 訓練偵測器（16 GB 的卡用 batch 4）
> python pipeline.py train-det --det-batch 4
```

附的偵測器已經訓練好，一般不需要重訓。

改了 --det-batch 等設定，新偵測器會和附的權重不同；
在 GPU 上重訓，每次結果也會有些微差異。

項目	實測（RTX 5060 Ti 16 GB）
顯示記憶體	用 --det-batch 4，峰值約 15 GB
一個 epoch	約 6 分鐘
200 個 epoch	約 19 小時
硬碟空間	訓練存檔最多約 11.5 GB
新權重	outputs/det_train/pv/weights/best.pt
改用它	從 detect 重跑，並加 --det-weights 指向它
先試跑	加 --det-epochs 10 減少 epoch 數

想改進 baseline？

1 調整各類別門檻

定位佔總分 70%，亂框會拉低分數：用 **--conf** 調整

注意：偵測器看過訓練集，在訓練集上評估會偏樂觀（比實際好看）；baseline 也沒附在驗證集上算官方分數的程式

2 改成因模型

換更大的語言模型（**--llm**）、用 LoRA（**--lora**），或修改 prompt

prompt 由 reason/common.py 的 describe() 產生，訓練與推論共用，改一次兩邊會一起變

3 加入影像資訊

目前的成因模型只看偵測框的位置與數量

它沒有看影像本身。想一想：怎麼讓模型也「看到」畫面？

4 先讀懂評分規則

配對規則、文字分數怎麼算，都寫在官方 evaluation kit 的 README

讀懂再決定優化方向；要精確比較，得自己照規則寫評分程式

常見問題：安裝篇

問題	怎麼辦
torch.cuda.is_available() 是 False	先確認命令列前面有 (.venv)；再依 README 1-3 的表重裝 torch（先 pip uninstall torch torchvision），並更新 NVIDIA 驅動
RTX 50 系列出現 **no kernel image is available**	torch 不是 CUDA 12.8 版：用 README 1-3 表格第一列（cu128）的指令重新安裝
ModuleNotFoundError: No module named 'einops'（或其他套件）	沒在虛擬環境裡裝套件，或忘了啟用：啟用後重新執行 pip install -r requirements.txt
PowerShell 出現「因為這個系統上已停用指令碼執行」	在同一個視窗先執行 Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass，再執行 .venv\Scripts\Activate.ps1；每開一個新視窗都要再做一次
用 **py -3.12 ...** 執行就出錯	py -3.12 會跳過虛擬環境，改用沒裝套件的 Python：啟用虛擬環境後，一律打 python

常見問題：執行篇

問題	怎麼辦
eval 或 submit 出現「找不到成因模型 ...qwen_reason\final」	還沒自己訓練 Qwen：在指令後面加 **--reason-model weights/qwen_reason** 改用附的權重，或先跑 train-sft
訓練 Qwen 時顯示記憶體不足（CUDA out of memory），或非常慢	調小 **--bs**：16 GB 卡用 8、12 GB 卡用 4、8 GB 卡試 --lora --bs 4
Windows 出現「系統找不到指定的路徑」（WinError 3）	多半是資料夾多了一層：用資料檢查指令確認；位置都對還是出錯，就把專案搬到短路徑，例如 D:\aicup_pv_baseline
出現英文警告（warning），要處理嗎？	不用，這是正常現象、不影響結果，**請不要為此修改程式**
中文變成亂碼，或出現 UnicodeEncodeError	先設定環境變數再執行：PowerShell 用 $env:PYTHONUTF8="1"，cmd 用 set PYTHONUTF8=1

65 進階與練習

小練習

```
# 1 改門檻（想更新提交檔要再跑 submit；還沒訓練 Qwen 要加 --reason-model weights/qwen_reason）
> python pipeline.py dets --conf "solar_panel=0.25,hotspot=0.4,crack=0.25"
# 2 讀一筆訓練資料（Linux 用 head -n 1 outputs/sft_data/train.jsonl）
> notepad outputs\sft_data\train.jsonl
# 3 換方式產生 reason（會覆寫 outputs/submission.json，想保留請先另存）
> python pipeline.py submit --mode rank --reason-model weights/qwen_reason
# 4 試跑訓練（會覆寫 outputs/qwen_reason/final；之後 submit 沒加 --reason-model 就會用它）
> python pipeline.py train-sft --bs 8 --epochs 0.2
```

1 改門檻

看印出的「偵測數」：hotspot 變多還是變少？面板呢？

2 讀訓練資料

看第一行的 user（題目）和 assistant（答案）

3 換產生方式

比較最後印出的「前幾名 reason」和 snap 有何不同

4 試跑訓練

看每 10 步印出一次的 loss：數字有沒有越來越小？

重點回顧

1. 框要框準，而且不要亂框：沒有信心分數，每個框都算
2. 原因要和異常配對成功（同類別、IoU ≥ 0.5）才會計分
3. 偽彩色的熱影像要先轉成灰階，偵測器才認得
4. Qwen 的訓練資料，是把訓練集標註轉成一問一答的對話
5. 先跑用法 A 確認環境，再自己訓練、修改

延伸閱讀：專案 README.md｜官方 evaluation kit README｜競賽頁（分數與上傳規則以競賽頁公告為準）