智慧家居-垃圾分類

張幃翔

2024/11/19

Outline

- 陸、智慧居家-垃圾分類
  - 一、案例介紹
  - 二、資料集與模組訓練
  - 三、在PC上使用Anaconda環境進行訓練
  - 四、 C++軟體流程
  - 五、DEMO影片

一、案例介紹

- (一) 案例介紹 – 垃圾分類
- (二) 案例介紹 – 預計結果

(一) 案例介紹 – 垃圾分類

- 簡介及應用 –
  - 透過nuvoTon開發的M55M1開發板來實作影像辨識垃圾分類，透過鏡頭模組可以有效的辨識垃圾種類、快速的判斷並即時的顯示垃圾種類。
  - 使用YOLOX_nano來訓練模型，透過深度學習來訓練出一個可以辨識多個垃圾種類的模型，並將其燒錄至M55M1中。
  - 在家中，對小孩子來說要分類垃圾有一定的困難度，因為有的顏色相同但材質不同、大小不同，導致小孩無法有效地去做分類，所以透過垃圾分類這個實例，可以使小孩只需要拿到鏡頭前，螢幕即可顯示此垃圾種類，並即時告知孩童，以便達到準確分類的目的。

(二) 案例介紹 – 預計結果

- 對於四種垃圾辨識準確率達到90.1%，分別為紙類、塑膠頻、鋁罐、紙杯。
- 模型大小為2MB以下，以達到燒錄開發板標準。
- 同時辨識多種垃圾，並且精準的分類，下圖為預計結果照片。

二、資料集與模組訓練

資料集

- LabelImg
- 資料集
  - Train set : 378 Images
  - Valid set : 36 Images
  - Test set : 19 Images

模型訓練

- 模型: YOLOX-Nano(Light Model)
- 虛擬環境: Anaconda
- 訓練框架: PyTorch
- 資料集格式: COCO JSON

二、資料集與模組訓練

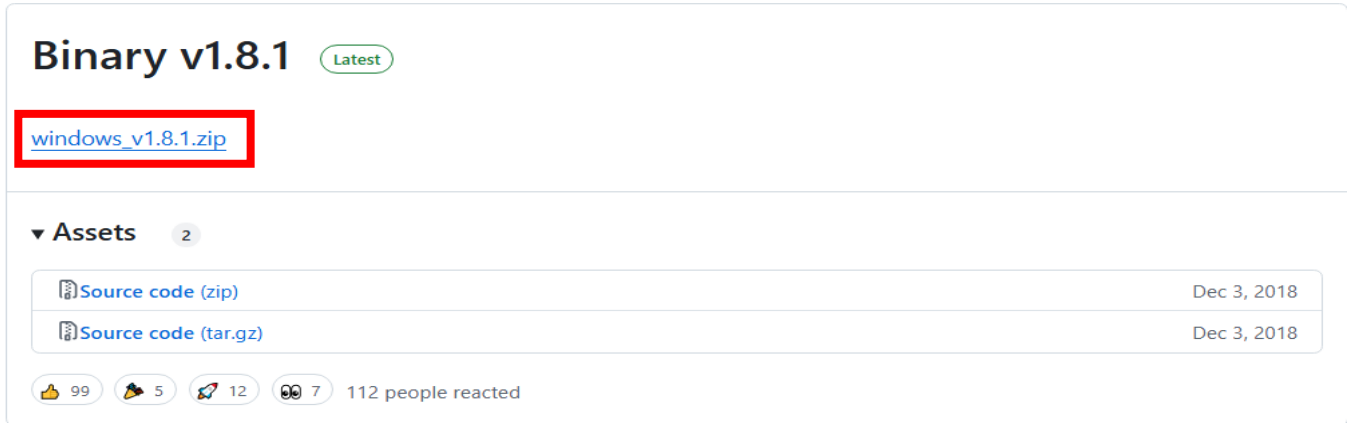
- (一) 資料集蒐集
- (二) LabelImg教學
- (三) 訓練設備
- (四) 訓練環境
- (五) 安裝環境
- (六) YOLOX_Nano模型訓練
- (七) Train dataset format
- (八) 訓練的自定義參數(train.py檔)
- (九)pytorch to Tflite
- (十) Vela Compiler and Convert to Deplyment Format
- (十一) Evaluate TFlite int8/float Model
- (十二) Test Singl/All Validation Images

(一) 資料集蒐集

- 資料集透過自行蒐集的照片來進行訓練
  - STEP1 : 蒐集足夠的照片(我使用約400張)，越多訓練效果會較佳
  - STEP2 : 進行Label，透過網路上可以下載的labelimg來進行
  - STEP3 : 確認資料集與要訓練的格式相同(本次使用 COCO JSON)
  - STEP4 : 把原先yolo格式轉為coco json格式

(二) LabelImg教學

- STEP1：下載此連結，點擊連結內為labelImg.exe
  - https://github.com/tzutalin/labelImg/releases



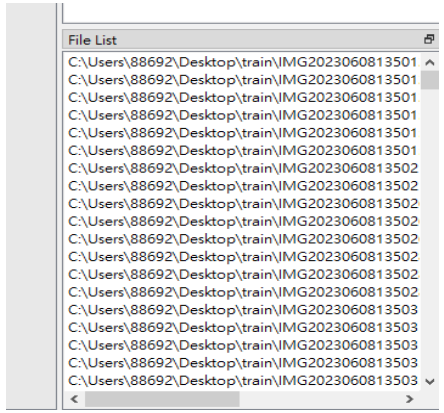
(二) LabelImg教學

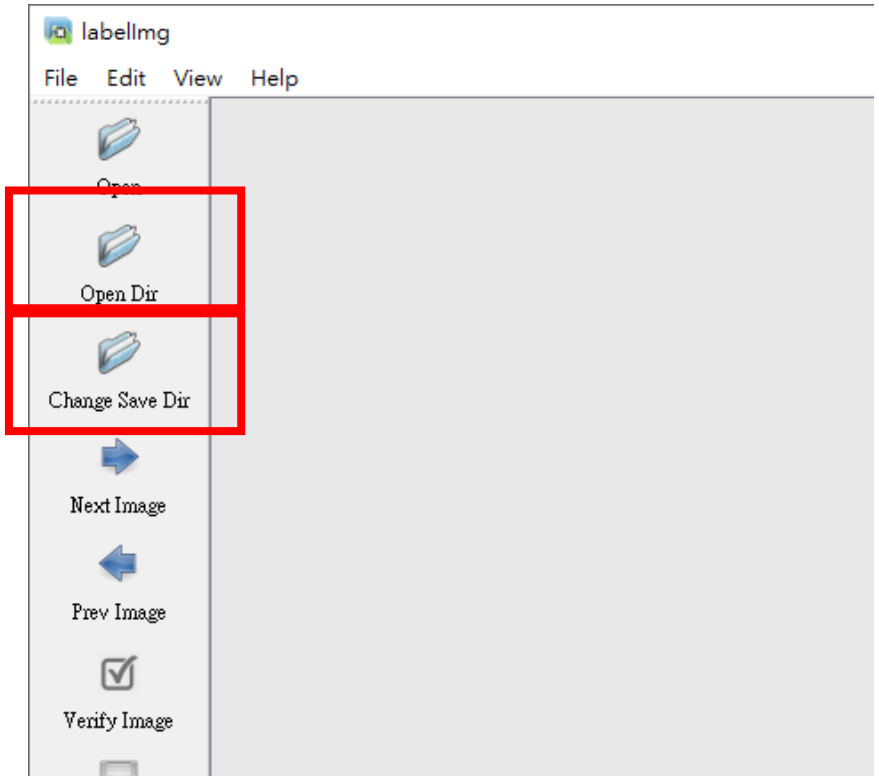
- STEP2 : 打開後要選取資料集位址，以及要儲存的位址

Step1:
需要標註的資料集位址

Step2:
儲存的資料集位址

Step3:
設定好後，右下角會出現所有要標註的照片





(二) LabelImg教學

- STEP3:選取要儲存的labelimg格式，分別為PascalVOC、YOLO、CreateML，依照要訓練的模型不同(yolo版本不相同)，來選擇要使用的格式

- PascalVOC : XML 文件來保存標註信息，圖片的檔案名、尺寸（高度、寬度）、以及每個標註框的類別和座標，用於TensorFlow、Keras。

- YOLO (You Only Look Once) : 純文字，每行表示一個標註框(本次使用)
  - <class_id> <x_center> <y_center> <width> <height>
  - 類別id 、 標註框中心座標 、 標註框寬以及高

- CreateML : JSON 文件保存標註信息，JSON 文件描述一張圖片，包含標註框的位置、類別以及相關元數據

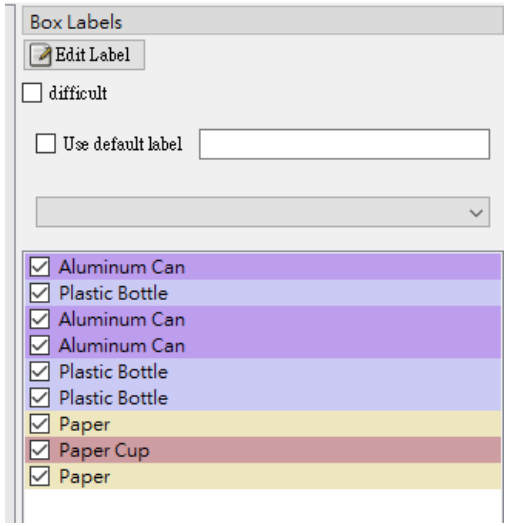
(二) LabelImg教學

- STEP4: 設定標註的種類並進行標註
  - 創建一個label.txt，裡面填寫你要的標註種類

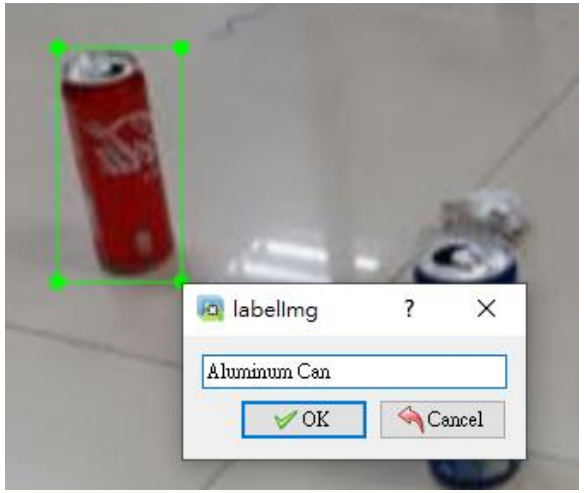
```
1  Aluminum Can
2  Paper
3  Paper Cup
4  Plastic Bottle
```

  - 開始進行標註，點選Create RectBox後(快捷鍵為W)，長案左鍵框住要標註的物品，標註完整張照片後可以看到所有標註的物品

所有標註的物品



標註框並給class



(二) LabelImg教學

- STEP5:使用yolo格式全部標註完以後，透過一段python程式將原本的yolo格式轉成COCO JSON檔，我們要使用的是COCO JSON檔。
- 因為COCO JSON較適合用來做深度學習的任務。
  - 下面為詳細的yolo2cocojson.py檔的連結，為一個python檔跟隨步驟即可變更，其餘所有都無須更動
  - https://drive.google.com/file/d/15cSq0MiBjgwUIwJn0Z_0t2c51vUY2jh6/view?usp=sharing

STEP1 : 加入自己的種類，需按照順序

```
# 類別名稱與對應的 ID
#categories可以增加到需要的數量
categories = [
    {"id": 0, "name": "class1"},
    {"id": 1, "name": "class2"},
    {"id": 2, "name": "class3"},
    {"id": 3, "name": "class4"}
]
```

STEP2 : 設定自己資料集的路徑，
資料夾名稱必須為Images、labels，
裡面的照片以及標註檔名稱必須相同。

```
# 定義資料夾路徑
images_dir = "/images"  # 圖片資料夾
labels_dir = "/lables"  # YOLO 標註檔案資料夾
```

(三) 訓練設備

- GPU => Nvidia GTX 1080 ti(可以透過指令來得知自身顯卡規格)
  - 方法一 => 在終端輸入 nvidia-smi
  - 方法二 => 打開工作管理員，選擇效能即可看到自身的GPU規格
- 透過GPU 來找相對應的CUDA、Torch版本
  - 使用下列網站可以看到適合自身顯卡的cuda版本
  - https://pytorch.org/get-started/previous-versions/
- CUDA => 11.8
- Torch => 2.0.0
- Python => 3.10
- 訓練時長 => 約 1.5hr
- 訓練數據 => 400張圖像

(四) 訓練環境

- 安裝anaconda3
  - https://www.anaconda.com/download
- 建立新的環境
  - 先在Anaconda Prompt(終端)中建立環境，我們使用python 3.10
  - conda create --name yolox_nu python=3.10
  - conda activate yolox_nu
- 下載yolox_nano github上的zip檔案
  - gh repo clone MaxCYCHEN/yolox-ti-lite_tflite_int8
- 更新pip
  - python -m pip install --upgrade pip setuptools

(五) 安裝環境

- 安裝Pytorch, CUDA, MMCV version需要相互配合，如果無使用cpu來training，則無需使用CUDA。
- 使用GPU訓練要根據自行的GPU的等級來選用，下列為範例
  - 使用cuda 118版本, torch 2.0版本
  - python -m pip install mmcv==2.0.1 -f https://download.openmmlab.com/mmcv/dist/cu118/torch2.0/index.html
- 安裝其他python所需套件，透過requirements.txt來下載
  - python -m pip install --no-input -r requirements.txt
- 安裝YOLOX
  - python setup.py develop

(六) YOLOX_Nano模型訓練

- 因為模型大小限制，在不外接Hyper RAM的情況下，只能使用2MB，因為YOLOX_Nano_ti_lite_nu僅僅0.91MB，為最適合使用在M55M1的Model。
- 訓練 –
  - $ python tools/train.py -f exps/default/yolox_nano_ti_lite_nu.py -d 1 -b 64 --fp16 -o -c pretrain/tflite_yolox_nano_ti/320_DW/yolox_nano_320_DW_ti_lite.pth
  - 對應的功能為
    $ python tools/train.py -f <MODEL_CONFIG_FILE> -d 1 -b <BATCH_SIZE> --fp16 -o -c <PRETRAIN MODEL PATH>
  - 對上面放置檔案進行解釋
    - MODEL_CONFIG_FILE => 將下載的yolox_nano中的yolox-nano-ti-nu model，連同位址一起放入
    - BATCH_SIZE => 設定一次訓練所要的batch大小，一般設定64，若電腦設備無法承受，可自行往下調(-> 32 -> 16)
    - PRETRAIN MODEL PATH => 放置預先訓練好的權重，亦在下載的yolox_nano中

(七) Train dataset format

- 資料結構按照下列分為
  - Annotations(coco.json檔)、train2017、val2017

```
datasets/<dataset_name>
           |
           |----annotations
           |        |----------<train_annotation_json_file>
           |        |----------<val_annotation_json_file>
           |
           |----train2017
           |        |---------train img
           |
           |----val2017
                    |---------validation img
```

- 設定dataset路徑

```
self.data_dir = "datasets/hagrid_coco"
self.train_ann = "hagrid_train.json"
self.val_ann = "hagrid_val.json"
```

(八) 訓練的自定義參數(train.py檔)

- 主要調整圖片大小、class數量、執行的epoch次數。
  - Class : 標註的種類數量
  - Epoch : 指訓練數據集通過模型一次的完整過程。
    - Epoch次數越多不一定效果越好，通常預設在150-200，超過此數量容易overfitting。
    - Overfitting : 當訓練的epoch太多，反而導致準確率下降。

```
self.input_size = (320, 320) # resolution
self.test_size = (320, 320) # resolution
self.num_classes = 11 # number of classes
self.max_epoch = 150 # training epoch
```

Ex:總共為11個類別

Ex:執行150個epoch

(九) Pytorch to ONNX

- 轉換成ONNX後續再轉換成要使用的模型框架
  - $ python tools/export_onnx.py -f <MODEL_CONFIG_FILE> -c <TRAINED_PYTORCH_MODEL> --output-name <ONNX_MODEL_PATH>
  - $ python tools/export_onnx.py -f exps/default/yolox_nano_ti_lite_nu.py -c YOLOX_outputs/yolox_nano_ti_lite_nu/latest_ckpt.pth --output-name YOLOX_outputs/yolox_nano_ti_lite_nu/yolox_nano_nu_medicine.onnx
    - MODEL_CONFIG_FILE =>用於描述模型結構與參數，也是從下載的yolox_nano當中給予位址
    - TRAINED_PYTORCH_MODEL => 指定已訓練完成的 PyTorch 模型（通常是 .pth 文件），作為轉換的來源
    - ONNX_MODEL_PATH =>轉換後的 ONNX 模型儲存路徑與名稱

(九) ONNX to TFlite

- 透過ONNX轉成要使用的模型框架(TFlite)
  - $onnx2tf -i <ONNX_MODEL_PATH> -oiqt -qcind images <CALI_DATA_NPY_FILE> "[[[[0,0,0]]]]" "[[[[1,1,1]]]]“
  - $ onnx2tf -i YOLOX_outputs/yolox_nano_ti_lite_nu/yolox_nano_nu_medicine.onnx -oiqt -qcind images YOLOX_outputs\yolox_nano_ti_lite_nu\calib_data_320x320_n200.npy "[[[[0,0,0]]]]" "[[[[1,1,1]]]]“
    - ONNX_MODEL_PATH =>轉換完ONNX的檔案的路徑
    - CALI_DATA_NPY_FILE=>校準資料檔案的路徑，通常是 .npy 格式的 NumPy 檔案

(十) Vela Compiler and Convert to Deplyment Format

- 使用vela complier
  - $ set MODEL_SRC_FILE=<your tflite model>
  - $ set MODEL_OPTIMISE_FILE=<output vela model>
  - $ set MODEL_SRC_FILE=yolox_nano_ti_lite_nu_hg_150_full_integer_quant.tflite
  - $ Set MODEL_OPTIMISE_FILE=yolox_nano_ti_lite_nu_hg_150_full_integer_quant_vela.tflite
    - your tflite model => tflite 的模型(.tflite檔)
    - output vela model => 輸出vela的模型(.tflite檔)
- Output file(用於後續處理)
  - $ vela\generated\yolox_nano_ti_lite_nu_full_integer_quant_vela.tflite.cc

(十一) Evaluate TFlite int8/float Model

- 看訓練出來每一個class的效果，包含各個class的準確率(AP，Average Precision)，訓練的每個IOU的準確率
  - IOU :衡量模型預測框與真實框（Ground Truth）重疊程度的指標。

$$\text{IoU} = \frac{\text{預測框與真實框的交集面積}}{\text{預測框與真實框的聯集面積}}$$

  - $ python demo\TFLite\tflite_inference.py -m <FULL_INTEGER_QUANT_TFLITE> -s <SCORE_THR> -i <PATH_OF_IMAGE> -a <PATH_OF_VAL_ANNOTATION_FILE>
  - $ python demo\TFLite\tflite_inference.py -m YOLOX_outputs\yolox_nano_ti_lite_nu\yolox_nano_ti_lite_nu_hg_full_integer_quant.tflite -s 0.6 -i datasets/hagrid_coco/val2017/0001.jpg -a hagrid_coco/annotations/hagrid_val.json
    - FULL_INTEGER_QUANT_TFLITE =>需要評估的全整數量化 TFLite 模型（通常是 .tflite 格式）。
    - SCORE_THR =>分數閾值，指定物體檢測的置信度閾值，只有高於此分數的檢測結果才會被保留。
    - PATH_OF_IMAGE =>影像路徑：指定要進行推論的圖片文件路徑
    - PATH_OF_VAL_ANNOTATION_FILE =>驗證標註檔案，指定包含圖片真實標註的檔案路徑（.json 文件）

(十二) Test Singl/All Validation Images

- 測試所有的照片集(單張亦可)
  - $ python demo\TFLite\tflite_inference.py -m <FULL_INTEGER_QUANT_TFLITE> -s <SCORE_THR> -i <PATH_OF_IMAGE> -a <PATH_OF_VAL_ANNOTATION_FILE>
  - $ python demo\TFLite\tflite_inference.py -m YOLOX_outputs\yolox_nano_ti_lite_nu\yolox_nano_ti_lite_nu_hg_full_integer_quant.tflite -s 0.6 -i datasets/hagrid_coco/val2017/0001.jpg -a hagrid_coco/annotations/hagrid_val.json
    - FULL_INTEGER_QUANT_TFLITE =>模型檔案：指定需要執行推論的全整數量化 TFLite 模型（.tflite 格式）。
    - SCORE_THR =>分數閾值：指定物體檢測的置信度閾值，只有高於此分數的檢測結果才會被保留。
    - PATH_OF_IMAGE =>影像路徑：指定要進行推論的圖片文件路徑。
    - PATH_OF_VAL_ANNOTATION_FILE =>驗證標註檔案：指定包含圖片真實標註的檔案路徑（.json 文件）

四、C++軟體流程

- (一) 軟體流程
- (二) 系統初始化程式解析(BoardInit.cpp)
- (三) 模型推理程式解析(inferenceTask.cpp)
- (四) 後處理程式解析(DetectorProcessing.cpp)
- (五) Main.cpp關鍵程式解析

(一) 軟體流程

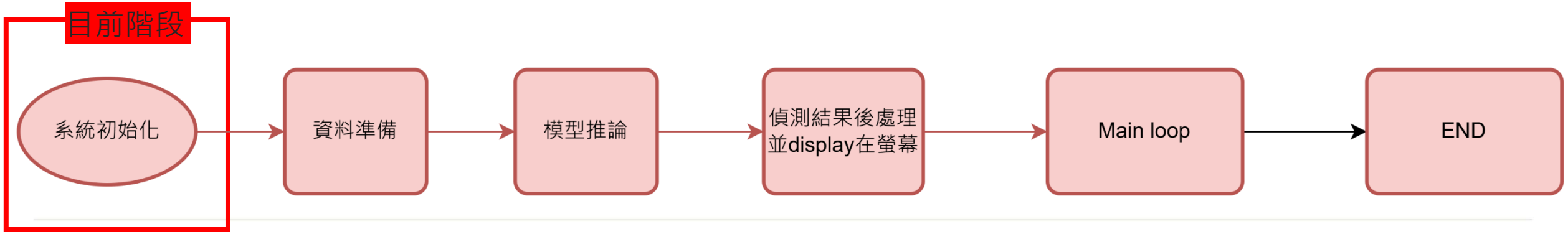
- 1. 系統初始化 (System Initialization)
  - 位置: BoardInit.cpp 和 mpu_config_M55M1.h
  - 設定硬體資源，包括時鐘、UART、記憶體和NPU。
  - 配置記憶體保護單元（MPU），分配記憶體區域和訪問屬性(mpu_config_M55M1)(BoardInit)。
- 2. 資料準備 (Data Preparation)
  - 加載圖片資料，進行必要的預處理如尺寸縮放或格式轉換。
  - InferenceTask.cpp 的任務排程負責接收資料(InferenceTask)。

(一) 軟體流程

- 3. 模型推論 (Model Inference)
  - 位置: InferenceTask.cpp使用模型執行推論 (m_model->RunInference())，並獲取模型輸出張量 (GetOutputTensor)。
  - 推論階段處理了模型運算核心的執行過程，計算出每個分類的機率和預測框(InferenceTask)。
- 4. 結果輸出及後處理 (Post-processing)
  - 位置: DetectorPostProcessing.cpp執行非極大值抑制（NMS）過濾低相關預測。
  - 調整預測框尺寸，將模型輸出的結果轉換為原始圖像空間的座標。
  - 使用分類閾值篩選最終的檢測結果(DetectorPostProcessing)。
- 5. Main loop
  - 主函數可能包含循環控制，保持系統處於運行狀態，直到觸發退出條件（如接收到退出信號）。
  - 確保所有資源在退出時正確釋放。

(二) 系統初始化程式解析(BoardInit.cpp)

- 重點功能概述(BoardInit.cpp & mpu_config_M55M1.h) :
  - 系統時鐘與模組時鐘初始化(SYS_Init)
  - UART初始化
  - HyperRAM初始化與模式配置
  - Arm Ethos-U NPU 初始化
  - 確保硬體保護與設定完成



(二) 系統初始化程式解析(BoardInit.cpp)

- 1. 系統時鐘與模組時鐘初始化
  - 啟用內部與外部振盪器，並等待其穩定。
  - 設置 APLL 鎖相迴路為系統時鐘源，頻率為 180 MHz。
  - 開啟必要的硬體模組時鐘。

```
/* 開啟內建 RC 振盪器與外部晶體振盪器 */
CLK_EnableXtalRC(CLK_SRCCTL_HIRCEN_Msk);
CLK_WaitClockReady(CLK_STATUS_HIRCSTB_Msk);
CLK_EnableXtalRC(CLK_SRCCTL_HXTEN_Msk);
CLK_WaitClockReady(CLK_STATUS_HXTSTB_Msk);

/* 設置系統時鐘來源與更新核心時鐘 */
CLK_SetBusClock(CLK_SCLKSEL_SCLKSEL_APLL0, CLK_APLLCTL_APLLSRC_HIRC, FREQ_180MHZ);
SystemCoreClockUpdate();
```

(二) 系統初始化程式解析(BoardInit.cpp)

- 2. UART 初始化：
  - 設定 UART6 作為調試用串列埠，允許使用 printf 進行標準輸出。
  - 配置 UART 的時鐘來源與多功能引腳。

```
/* 設置 UART 時鐘為 HIRC 並啟用 */
SetDebugUartCLK();
SetDebugUartMFP();
InitDebugUart();  // 初始化 UART，支持標準輸出
```

- 3. HyperRAM 初始化與模式配置：
  - 初始化 HyperRAM 的通訊端口，並進入直接映射模式以提升存取效率。

```
/* 配置 HyperRAM 腳位與初始化 */
HyperRAM_PinConfig(HYPERRAM_SPIM_PORT);
HyperRAM_Init(HYPERRAM_SPIM_PORT);

/* 切換至 Direct Map 模式運行應用 */
SPIM_HYPER_EnterDirectMapMode(HYPERRAM_SPIM_PORT);
```

(二) 系統初始化程式解析(BoardInit.cpp)

- 4. Arm Ethos-U NPU 初始化：
  - 如果啟用了 Arm NPU，進行初始化，並檢查狀態是否成功。
  - 確保嵌入式 AI 模型能正常運行。

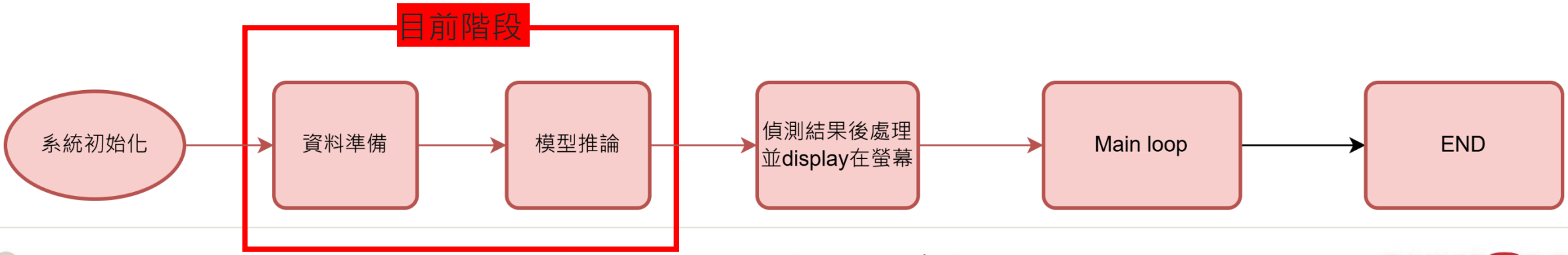
```
#if defined(ARM_NPU)
int state;

/* 初始化 Arm Ethos-U NPU，若失敗則返回錯誤碼 */
if (0 != (state = arm_ethosu_npu_init())) {
    return state;
}
#endif
```

- 5. 確保硬體保護與設定完成：
  - 在初始化完成前解鎖受保護寄存器，最後重新鎖定以確保安全。
  - 使用 info 宏輸出初始化完成訊息。

```
/* 解鎖受保護寄存器 */
SYS_UnlockReg();

/* 鎖定寄存器保護 */
SYS_LockReg();

/* 初始化完成訊息輸出 */
info("%s: complete\n", __FUNCTION__);
```

(三) 模型推理程式解析(inferenceTask.cpp)

- 重點功能概述(inferenceTask.cpp)
  - 推論執行（Inference Execution）：
    - 呼叫模型的推論函式，處理輸入資料並產生結果。
    - 使用後處理功能（DetectorPostprocessing）來解析模型輸出的 Tensor 資料。
  - 任務處理（Task Handling）：
    - 提供 FreeRTOS 任務的執行函式（inferenceProcessTask），負責處理推論任務佇列（Queue）。
    - 任務間使用訊號量（Semaphore）與互斥鎖（Mutex）進行同步。



(三) 模型推理程式解析(inferenceTask.cpp)

- 1. 推論以及執行
  - m_model->RunInference()：執行 YOLO 模型的推論。
  - pPostProc->RunPostProcessing(...)：將模型輸出的 Tensor 轉換為可讀的物體偵測結果，包含座標框與標籤。

```
20  bool InferenceProcess::RunJob(
21      object_detection::DetectorPostprocessing *pPostProc,
22      int modelCols,
23      int mode1Rows,
24      int srcImgWidth,
25      int srcImgHeight,
26      std::vector<object_detection::DetectionResult> *results
27  )
28  {
29      //info("Inference process task run job...\n");
30
31  #if defined(__PROFILE__)
32      uint64_t u64StartCycle;
33      uint64_t u64EndCycle;
34
35      profiler.StartProfiling("Inference");
36  #endif
37
38      bool runInf = m_model->RunInference(); // 執行模型推論
39
40  #if defined(__PROFILE__)
41      profiler.StopProfiling();
42      profiler.PrintProfilingResult();
43  #endif
44
45      TfLiteTensor *modelOutput0 = m_model->GetOutputTensor(0); // 獲取模型輸出
46
47  #if defined(__PROFILE__)
48      u64StartCycle = pmu_get_systick_Count();
49  #endif
```

```
50
51      pPostProc->RunPostProcessing(
52          mode1Rows,
53          modelCols,
54          srcImgHeight,
55          srcImgWidth,
56          modelOutput0,
57          *results); // 執行後處理
```

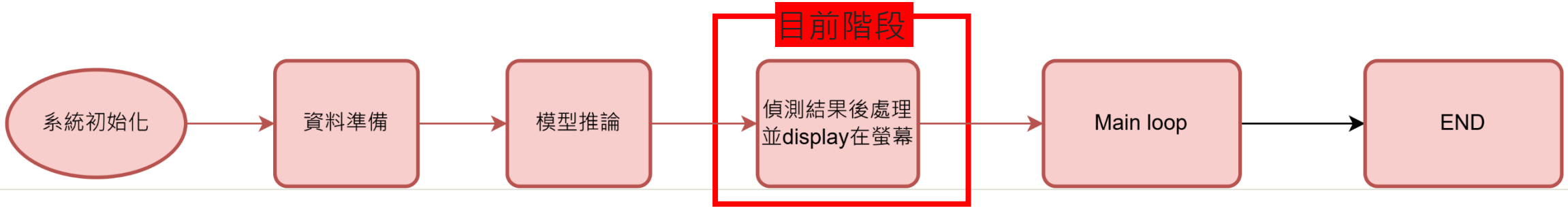
(三) 模型推理程式解析(inferenceTask.cpp)

- 2. FreeRTOS 的多任務架構
  - 功能用法：
    - 從任務佇列中接收推論請求（xQueueReceive）。
    - 執行推論與後處理。將推論結果傳回給發送者（xQueueSend）。

```
void inferenceProcessTask(void *pvParameters)
{
    struct ProcessTaskParams params = *reinterpret_cast<struct ProcessTaskParams *>(pvParameters);

    InferenceProcess::InferenceProcess inferenceProcess(params.model);

    for (;;)
    {
        xInferenceJob *xJob;

        xQueueReceive(params.queueHandle, &xJob, portMAX_DELAY);  // 等待推論任務

        inferenceProcess.RunJob(
            xJob->pPostProc,
            xJob->modelCols,
            xJob->mode1Rows,
            xJob->srcImgWidth,
            xJob->srcImgHeight,
            xJob->results
        );

        xQueueSend(xJob->responseQueue, &xJob, portMAX_DELAY);  // 回傳結果
    }
}
```

(四) 後處理程式解析(DetectorProcessing.cpp)

- 重點功能介紹
  - 初始化後處理參數：
    - 設定偵測門檻值、非極大值抑制（NMS）閾值、分類數量及 Top-N 選項。
  - 處理模型輸出：
    - 解析 YOLO 模型的輸出 Tensor，計算偵測框的位置與大小，並篩選符合門檻值的物體。
  - 非極大值抑制（NMS）：
    - 移除重疊過高的偵測框，保留最相關的結果。
  - 轉換至原始影像尺寸：
    - 將模型輸出的座標框比例轉換至原始影像大小，得到真實的物體位置。



(四) 後處理程式解析(DetectorProcessing.cpp)

- 1. 後處理初始化
  - 參數意義：
    - threshold：物體偵測的置信度門檻值，低於此值的偵測框會被忽略。
    - nms：非極大值抑制的 IoU 門檻值，用來去除重疊過多的框。
    - numClasses：模型支援的分類數量。
    - topN：返回的最高置信度結果數量。

```
DetectorPostprocessing::DetectorPostprocessing(
    const float threshold,
    const float nms,
    int numClasses,
    int topN)
    :   m_threshold(threshold),
        m_nms(nms),//IOU
        m_numClasses(numClasses),//分類的數量
        m_topN(topN)
{}
```

(四) 後處理程式解析(DetectorProcessing.cpp)

- 2. 執行後處理
  - 功能：
    - 初始化 YOLO 模型輸出的結構化資料（Network）。
    - modelOutput0：模型輸出 Tensor，包含邊框座標與分類資訊。
    - scale 與 zeroPoint：量化模型輸出的相關參數，用於將 Tensor 值轉換回浮點數格式。

```
void DetectorPostprocessing::RunPostProcessing(
    uint32_t imgNetRows,
    uint32_t imgNetCols,
    uint32_t imgSrcRows,
    uint32_t imgSrcCols,
    TfLiteTensor *modelOutput0,
    std::vector<DetectionResult> &resultsOut    /* init postprocessing */
)
```

```
Network net
{
    .inputWidth = static_cast<int>(imgNetCols),
    .inputHeight = static_cast<int>(imgNetRows),
    .numClasses = m_numClasses,
    .branches = {
        Branch {
            .resolution = modelOutput0->dims->data[1],
            .modelOutput = modelOutput0->data.int8,
            .scale = ((TfLiteAffineQuantization *)(modelOutput0->quantization.params))->scale->data[0],
            .zeroPoint = ((TfLiteAffineQuantization *)(modelOutput0->quantization.params))->zero_point->data[0],
            .size = modelOutput0->bytes
        }
    },
    .topN = m_topN
```

(四) 後處理程式解析(DetectorProcessing.cpp)

- 3. 偵測框解析與篩選
  - 邏輯：
    - 從模型輸出中解析每個偵測框的座標、大小和置信度。
    - 篩選置信度高於 threshold 的框，並將座標從模型輸入比例轉換至原始影像比例。

```
GetNetworkBoxes(net, originalImageWidth, originalImageHeight, m_threshold, detections);
```

- 4. 非極大值抑制（NMS）
  - 目的：
    - 移除重疊率（IoU）高於 nms 門檻值的重複框，只保留最高置信度的框。
    - 確保每個物體僅對應一個框。

```
/* Do nms */
CalculateNMS(detections, net.numClasses, m_nms);
```

(四) 後處理程式解析(DetectorProcessing.cpp)

- 5. 轉換至原始影像尺寸
  - 轉換邏輯：
    - 根據模型輸出的框座標與模型輸入影像的比例，計算對應原始影像的框位置與大小。

```
for (auto &it : detections)
{

    // transfer to original img size (base on resize ratio)
    it.bbox.x = (it.bbox.x * originalImageWidth) / net.inputWidth;
    it.bbox.y = (it.bbox.y * originalImageHeight) / net.inputHeight;
    it.bbox.w = (it.bbox.w * originalImageWidth) / net.inputWidth;
    it.bbox.h = (it.bbox.h * originalImageHeight) / net.inputHeight;
```

- 6. 最終結果生成
  - 生成結構化的偵測結果，包含：
    - 框的起點座標（m_x0、m_y0）和大小（m_w、m_h）。
    - 偵測置信度（m_normalisedVal）。
    - 物體分類（m_cls）。

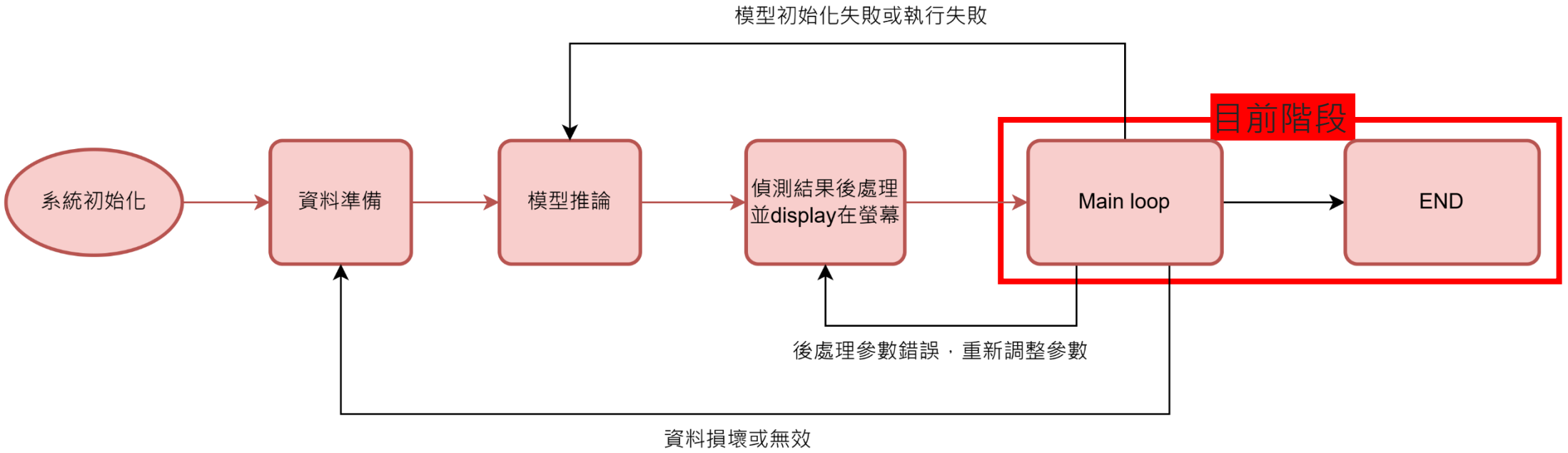
```
{

    DetectionResult tmpResult = {};
    tmpResult.m_normalisedVal = it.prob[j];
    tmpResult.m_x0 = (int)boxX;
    tmpResult.m_y0 = (int)boxY;
    tmpResult.m_w = (int)boxWidth;
    tmpResult.m_h = (int)boxHeight;
    tmpResult.m_cls = j;

    resultsOut.push_back(tmpResult);
}
```

(五) Main.cpp關鍵程式解析

- 流程與主循環的對應位置
- 接收數據(回到 資料準備 (Data Preparation) 階段)
  - 主循環檢查是否有新的數據需要處理，並嘗試從緩衝區獲取完整的幀圖像 (frame buffer)。
  - 若緩衝區中存在數據，則進行進一步的處理：
  - 獲取空閒緩衝區 (get_empty_framebuf())。
  - 獲取完整緩衝區 (get_full_framebuf())。
- 推論執行(回到 模型推論 (Model Inference) 階段)
  - 當緩衝區的數據準備就緒，將數據發送給推論任務：使用 TensorFlow Lite 模型執行推論 (RunInference() 方法)。
  - 主循環等待推論結果。
- 結果後處理(回到 後處理 (Post-processing) 階段)
  - 推論完成後，對輸出數據進行後處理，執行：
  - 非極大值抑制（NMS）。
  - 將座標從模型輸出空間轉換到原始圖像空間。
  - 篩選有效的檢測結果。
- 顯示與結果輸出(對應流程：回到 結果處理與輸出 (Result Handling & Output) 階段)
  - 將後處理完成的檢測結果顯示在輸出設備（如LCD屏幕或串口）。
  - 更新結果緩衝區狀態（如設置為 eFRAMEBUF_EMPTY 表示處理完成）。

(五) Main.cpp關鍵程式解析

- **總結**
- **主循環的錯誤會導致流程重啟，但它應該根據問題的性質回到以下位置：**
- **資料準備：** 若幀數據損壞或無效，需重新嘗試獲取數據。
- **模型推論：** 若模型初始化或執行失敗，需重設模型並重啟推論任務。
- **後處理：** 若後處理參數錯誤，需調整輸入參數或模型輸出配置。



(五) Main.cpp關鍵程式解析

- 1. 影像緩衝區管理：
  - 這些函式用於取得不同狀態的緩衝區，以便進行影像填充、推論或結果處理。

```
static S_FRAMEBUF *get_empty_framebuf();
static S_FRAMEBUF *get_full_framebuf();
static S_FRAMEBUF *get_inf_framebuf();
```

    - get_empty_framebuf()：尋找可用於新影像填充的緩衝區（狀態為 eFRAMEBUF_EMPTY）。
    - get_full_framebuf()：尋找已填充影像且可用於推論的緩衝區（狀態為 eFRAMEBUF_FULL）。
    - get_inf_framebuf()：尋找正在推論中的緩衝區（狀態為 eFRAMEBUF_INF）。
- 2. 模型初始化：
  - 初始化 YOLO 模型，分配 Tensor 緩衝區。
  - 從模型指標與長度資訊中載入模型。

```
arm::app::YoloXnanoNu model;
model.Init(arm::app::tensorArena, sizeof(arm::app::tensorArena),
           arm::app::yoloxnanonu::GetModelPointer(), arm::app::yoloxnanonu::GetModelLen());
```

(五) Main.cpp關鍵程式解析

- 3. 輸入影像處理
  - 使用 imlib_nvt_scale() 將影像縮放到模型要求的尺寸。
  - 若模型需要有符號資料（ IsDataSigned() ），將影像資料轉換為 int8 格式。

```
imlib_nvt_scale(&fullFramebuf->frameImage, &resizeImg, &roi);
if (model.IsDataSigned()) {
    arm::app::image::ConvertImgToInt8(inputTensor->data.data, inputTensor->bytes);
}
```

- 4. 推論觸發
  - 推論任務會根據當前影像進行物體檢測。
  - 使用 FreeRTOS 隊列 (Queue) 傳遞推論任務。

```
xQueueSend(inferenceProcessQueue, &inferenceJob, portMAX_DELAY);
```

(五) Main.cpp關鍵程式解析

- 5. 後處理與結果呈現
  - DrawImageDetectionBoxes()：在影像上繪製檢測結果（矩形框和標籤）
  - PresentInferenceResult()：顯示結果資訊（例如物件類別與座標）。

```
DrawImageDetectionBoxes(infFramebuf->results, &infFramebuf->frameImage, labels);
PresentInferenceResult(infFramebuf->results, labels);
```

- 6. 主循環邏輯
  - 主循環中不斷檢查不同狀態的緩衝區並執行相應處理。
  - 每次循環間加入延遲 (vTaskDelay(1)) 以避免佔用過多 CPU 資源。

```
while (1) {
    infFramebuf = get_inf_framebuf();
    fullFramebuf = get_full_framebuf();
    emptyFramebuf = get_empty_framebuf();
    // 推論、後處理與顯示邏輯...
    vTaskDelay(1);
}
```

五、DEMO影片

- Link

Joy of innovation

NUVOTON

谢谢

謝謝

Děkuji

Bedankt

Thank you

Kiitos

Merci

Danke

Grazie

ありがとう

감사합니다

Dziękujemy

Obrigado

Спасибо

Gracias

Teşekkür ederim

Cảm ơn