智慧工廠 – 藥物辨識

大綱

- 肆、智慧醫療1 - 藥物辨識
  - 一、案例介紹 藥物辨識
  - 二、使用資料集與模型介紹
  - 三、訓練環境與硬體介紹
  - 四、模型訓練
  - 五、C++ 軟體實現
  - 六、模型設計介紹 – 模型推理

一、案例介紹 - 藥物辨識

- 藥物辨識摘要:
  - 製藥工廠或醫療領域中，快速且準確地辨識藥物是非常重要的。
  - 傳統人工方法可能耗時且容易出錯，因此透過一個嵌入式解決方案，能夠即時辨識藥物，能夠解決問題。
- 使用場景舉例:
  - 1. 避免病人用藥錯誤，提供它隨時自我確認。
  - 2. 監控製藥廠生產過程，提升效率與降低人工出錯率。

一、案例介紹 - 藥物辨識

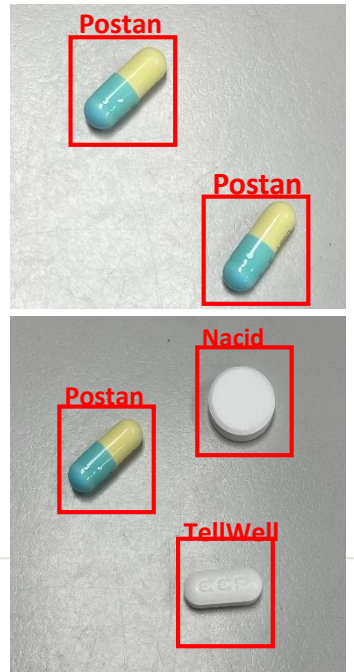
- **技術背景與需求**：
  - 藥物辨識技術利用人工智慧和影像處理技術，對藥物進行高效的分類與識別。
  - 對於醫療、製藥與監控領域而言，準確且快速的藥物辨識是一項關鍵需求。
- **核心技術**：
  - **機器學習與深度學習**：結合先進的物件檢測模型（如 YOLO）進行藥物特徵分析。
  - **影像處理與分類**：透過影像數據的前處理與後處理，確保辨識結果準確度。
- **應用價值**：
  - **醫療場景**：協助醫護人員快速核對藥物，降低用藥錯誤風險。
  - **製藥工廠**：自動化生產監控，提升效率並降低人工成本。
- **主要挑戰**：
  - **辨識準確度**：面對相似外觀的藥物，提升分類模型的辨識能力。
  - **實時性要求**：保證高效率的處理速度以滿足應用需求。

一、預期成果- 藥物辨識

- 案例預期效果
  - 支援複數、多種與不同類型(膠囊、圓形或半顆劑量)的藥物
  - 精度高且速度快，適用醫療與製藥生產環境
  - 辨識準確率達95%

測試拍攝物

預期開發版LCD顯示結果

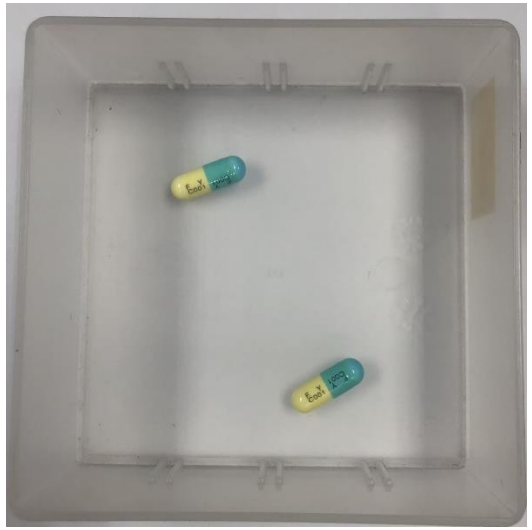


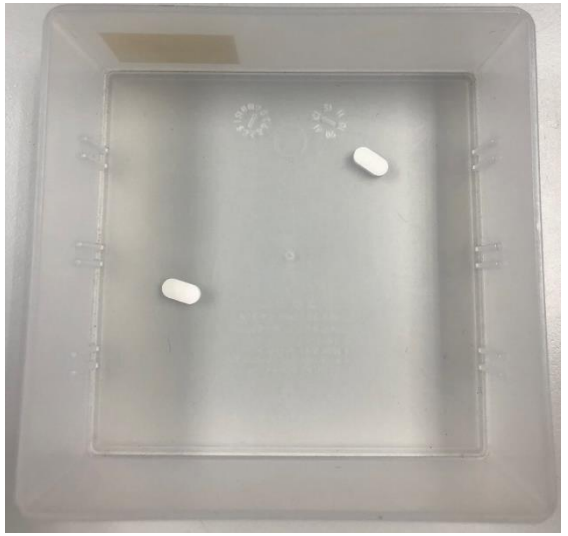
二、使用資料集與模型介紹

資料集準備	• **LabelImg** • **Roboflow**
模型訓練	• **選擇輕量級的模型** • **yolox nano**
模型框架轉換	• **Pytorch => Tensorflow lite**

二、資料集與使用模型介紹

- 訓練框架 : PyTorch
- 訓練模型 : Yolox-nano
- 資料集 : 醫院藥物資料集
- 標註方式: labelImg
- 格式: COCO JSON
  - Train Set : 1500images
  - Valid Set : 300 images





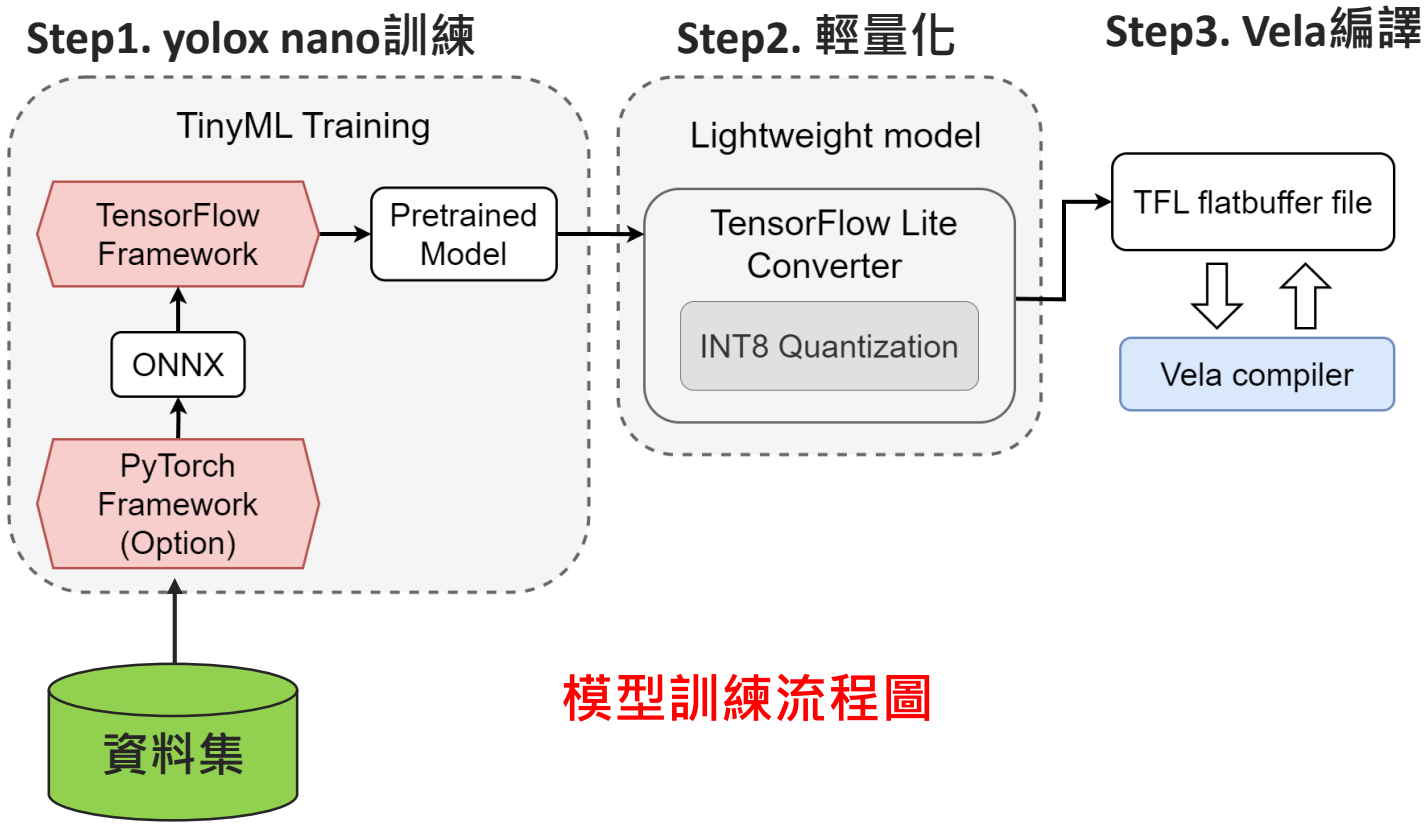
- 模型資源:https://github.com/MaxCYCHEN/yolox-ti-lite_tflite_int8

三、訓練環境與硬體介紹

- 本專案硬體規格與耗時
  - 硬體設備:NVIDIA RTX 3060 TI
  - CUDA版本:11.8
  - Pytorch版本:2.4.1
  - 訓練數據:1800張圖片
  - 訓練參數: Epoch = 200 , Batch size = 32
  - 訓練時間: 1.5小時

四、模型訓練

- 模型訓練步驟
  - 環境建立
  - 資料集準備
  - TinyML Training
  - 模型輕量化
  - Vela編譯



模型訓練流程圖

(一)、Anaconda3 環境建立 (1/2)

- 環境建立步驟:
  - 建立python環境
    - conda create --name yolox_nu python=3.10
    - conda activate yolox_nu
      - 透過上述指令在anaconda上建立環境
  - upgrade pip
    - python -m pip install --upgrade pip setuptools
  - 依照系統類型選擇不同CUDA版本、PyTorch版本、MMCV版本安裝pytorch
    - python -m pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118　(舉例使用cu118版本)

(一)、Anaconda3 環境建立 (2/2)

- 環境建立步驟:
  - 根據您的硬體配置安裝 mmcv
    - $python -m pip install mmcv==2.0.1 -f https://download.openmmlab.com/mmcv/dist/cu118/torch2.0/index.html
  - 安裝其它需求套件:
    - 開啟安裝好的檔案目錄並透過下面指令下載
    - $python -m pip install --no-input -r requirements.txt
  - Installing the YOLOX
    - python setup.py develop

(二)、Yoloxnano 模型的資料集整理

- Yoloxnano模型資料集需求
  - 使用COCO JSON格式
  - 準備好訓練資料並整理成下列形式並放在Dataset目錄中
    - Datasets/<your_datasets_name>/
      - annotatios/
        - train_annotation_json_file
        - val_annotation_json_file
      - train2017/
        - train_img
      - val2017/
        - validation_img

(三)、Yoloxnano模型訓練步驟

- 模型訓練設定檔
  - exps/default/yolox_nano_ti_lite_nu.py
- 更新設定檔內部檔案路徑:
  - self.data_dir = "datasets/your_coco"
  - self.train_ann = “your_train.json"
  - self.val_ann = “your_val.json"
- 預訓練模型路徑
  - pretrain/tflite_yolox_nano_ti/320_DW/yolox_nano_320_DW_ti_lite.pth
- 開始訓練
  - $python tools/train.py -f <MODEL_CONFIG_FILE> -d 1 -b <BATCH_SIZE> --fp16 -o -c <PRETRAIN MODEL PATH>

模型訓練設定檔

預訓練模型路徑

(四)、模型轉換

- 模型轉換
  - **Pytorch to ONNX**
    - $python tools/export_onnx.py -f <MODEL_CONFIG_FILE> -c <TRAINED_PYTORCH_MODEL> --output-name <ONNX_MODEL_PATH>
- 輕量化
  - Create calibration data
    - $python demo/TFLite/generate_calib_data.py --img-size <IMG_SIZE> --n-img <NUMBER_IMG_FOR_CALI> -o <CALI_DATA_NPY_FILE> --img-dir <PATH_OF_TRAIN_IMAGE_DIR>
  - **Convert ONNX to Tflite**
    - $onnx2tf -i <ONNX_MODEL_PATH> -oiqt -qcind images <CALI_DATA_NPY_FILE> "[[[[0,0,0]]]]" "[[[[1,1,1]]]]"

(五)、Vela編譯

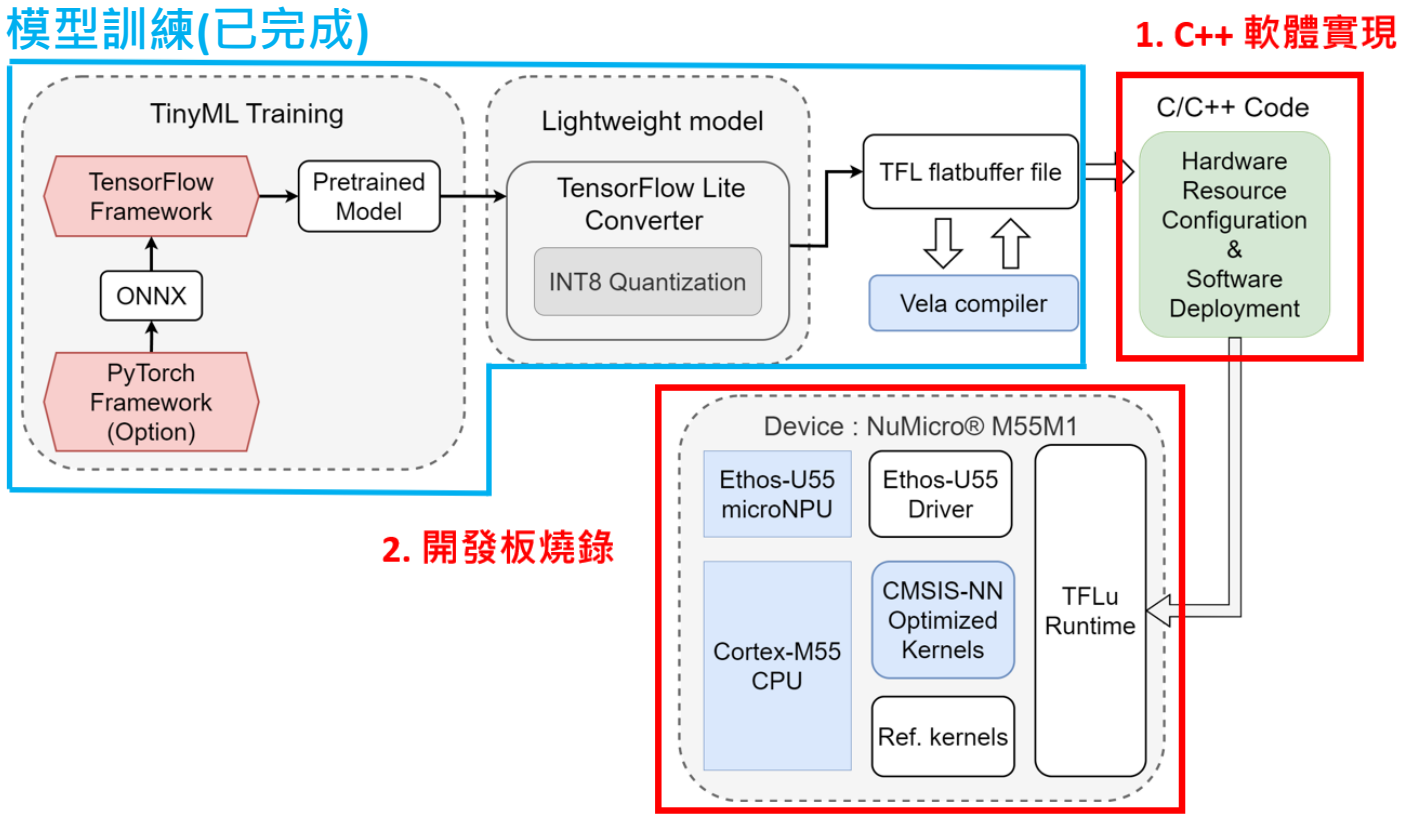
- Vela編譯步驟:
  - 將輕量化完的模型放到 vela\generated\
  - 並且修改vela目錄下的 variables.bat
    - set MODEL_SRC_FILE=<your tflite model>
    - set MODEL_OPTIMISE_FILE=<output vela model>
  - 執行gen_modle_cpp檔案
    - 執行結果會出現在 vela\generated\yolox_nano_ti_lite_nu_full_integer_quant_vela.tflite.cc

五、C++軟體實現

- 系統流程圖
- 確認開發版硬體需求
- C++ 軟體功能設計
  - C++ Sample code簡介
  - C++軟體流程圖
  - C++軟體設計介紹
- 設定Label檔
- DEMO影片

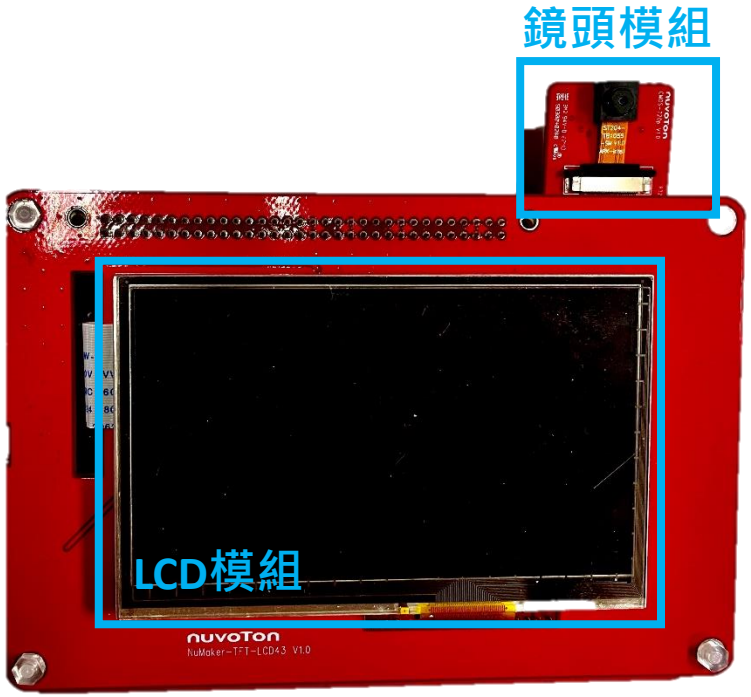
(一)、系統流程圖

- 模型訓練(藍色部分)在前面已完成並且會得到一個C++的模型檔，再來才能進行軟體實現與開發板燒錄(紅色部分)



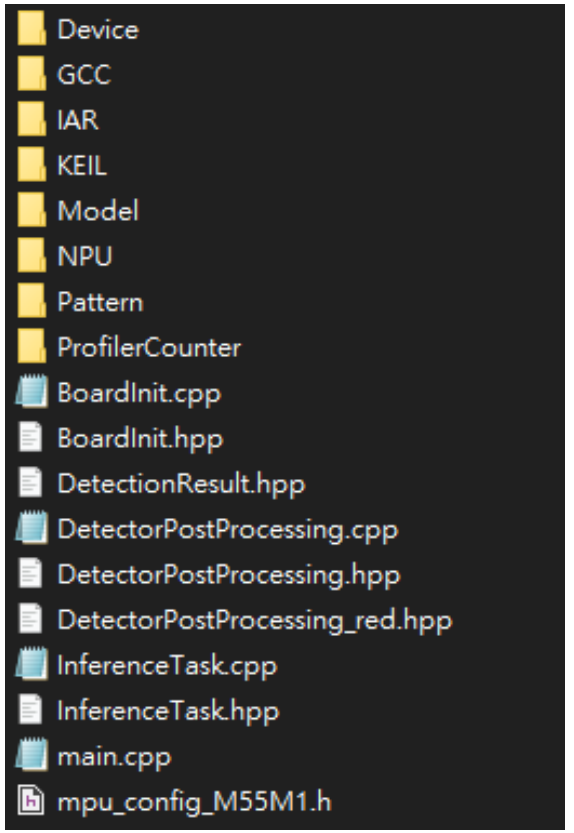
(二)、確認開發版硬體需求

- 實現藥物辨識所需模塊
  - 開發版硬體硬體配置:
    - LCD
      - 顯示介面的設置與EBI通訊介面的軟體配置
    - Image sensor
      - 設置解析度屬性與I2C通訊介面的軟體配置



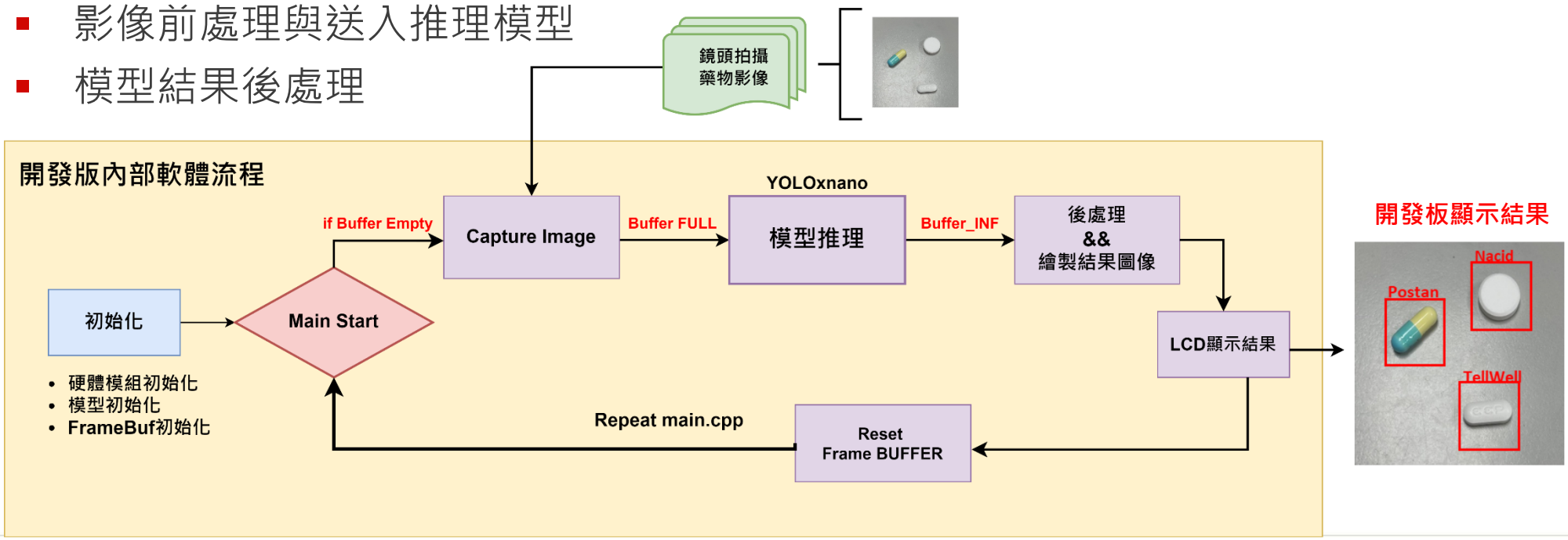
(三)、C++ Sample code簡介

- Sample code 檔案介紹
  - Device:
    - 關於開發板模組: LCD、ImageSensor與HyperRam等
  - KEIL:
    - 燒錄軟體相關檔案
  - Model:
    - 將訓練好的C++模型檔放入這
  - NPU:
    - 開發版運算單元相關檔案



(三)、C++軟體執行流程圖

- C++設計模塊包含
  - 硬體、模型與Frambuf初始化設定
  - 設計Frambuf 狀態 對應3種功能:
    - 擷取影像
    - 影像前處理與送入推理模型
    - 模型結果後處理

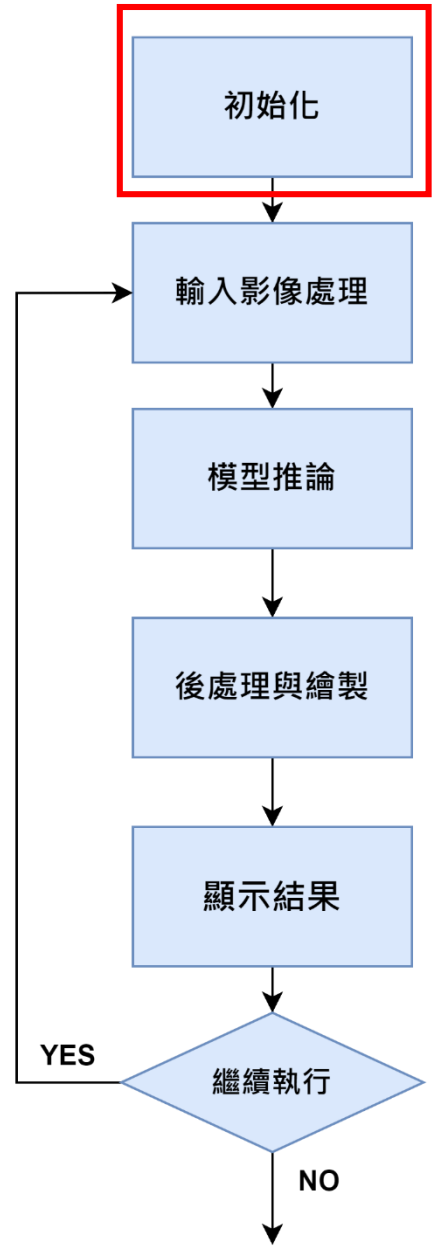


硬體模組初始化

- BoardInit.cpp
  - 系統時鐘初始化 (SYS_Init)：
    - 啟用內部 RC 12 MHz 時鐘和外部 12 MHz 時鐘。
    - 切換系統時鐘（SCLK）來源至 APLL0 並啟用 180 MHz APLL0 時鐘。
    - 更新系統核心時鐘。

BoardInit.cpp

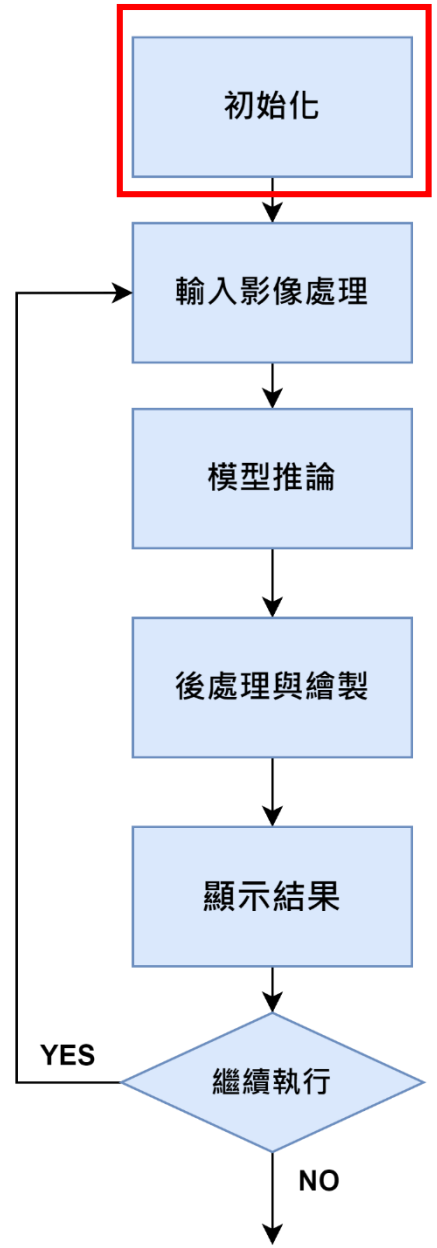
```
/* Enable Internal RC 12MHz clock */
CLK_EnableXtalRC(CLK_SRCCTL_HIRCEN_Msk);

/* Waiting for Internal RC clock ready */
CLK_WaitClockReady(CLK_STATUS_HIRCSTB_Msk);

/* Enable HXT clock */
CLK_EnableXtalRC(CLK_SRCCTL_HXTEN_Msk);

/* Waiting for HXT clock ready */
CLK_WaitClockReady(CLK_STATUS_HXTSTB_Msk);

/* Switch SCLK clock source to APLL0 and Enable APLL0 180MHz clock */
CLK_SetBusClock(CLK_SCLKSEL_SCLKSEL_APLL0, CLK_APLLCTL_APLLSRC_HIRC, FREQ_180MHZ);

/* Update System Core Clock */
/* User can use SystemCoreClockUpdate() to calculate SystemCoreClock. */
SystemCoreClockUpdate();
```



1.模型建立與初始化

- 初始化 YOLOxnano模型
  - 提供張量緩衝區作為推論空間
  - 使用函式 GetModelPointer() 與 GetModelLen() 載入模型數據。

```
arm::app::YoloXnanoNu model;

if (!model.Init(arm::app::tensorArena,
                sizeof(arm::app::tensorArena),
                arm::app::yoloxnanonu::GetModelPointer(),
                arm::app::yoloxnanonu::GetModelLen()))
{
    printf_err("Failed to initialise model\n");
    vTaskDelete(nullptr);
    return;
}
```



2.硬體模組初始化

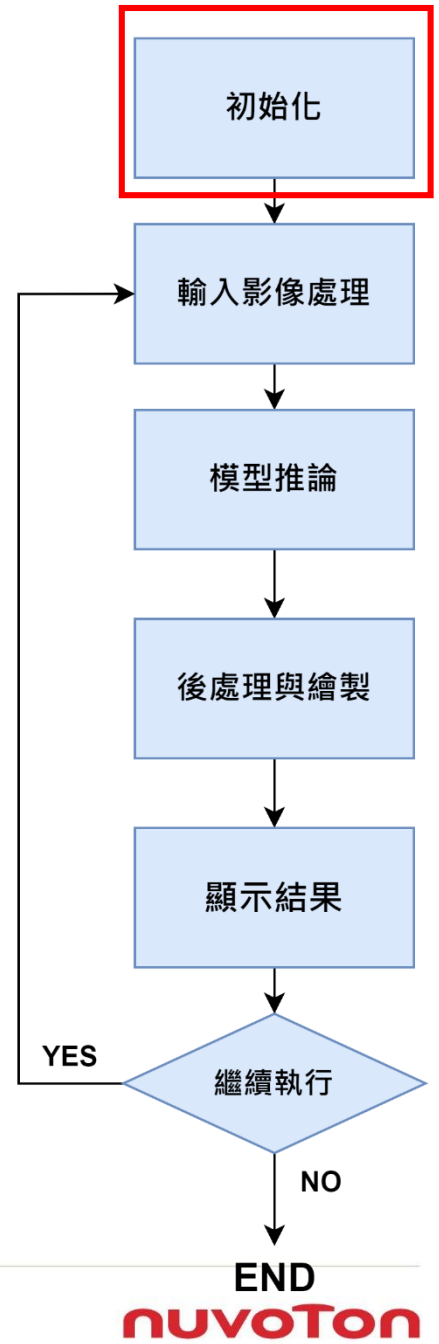
- 常數定義: main.cpp()
  - USE_CCAP &USE_DISPLAY 定義是否使用影像感測器 (CCAP)和LCD顯示模組
- Code:
  - 在main.cpp宣告藥物辨識需要用到的模塊的常數
  - 便能使用.h file內對應功能function

Main.cpp

```
29  #define __USE_CCAP__
30  #define __USE_DISPLAY__
```

Main.cpp

```
34  #if defined (__USE_CCAP__)
35      #include "ImageSensor.h"
36  #endif
37  #if defined (__USE_DISPLAY__)
38      #include "Display.h"
39  #endif
```



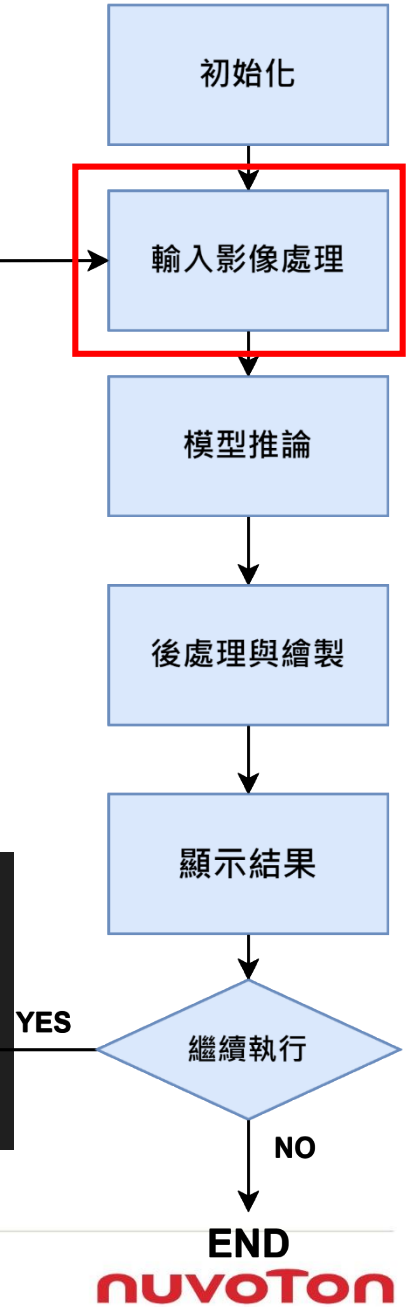
3.影像處理function介紹

- 鏡頭模組設置(Image Sensor)
  - 在main.cpp 呼叫到ImageSensor.h後，便可以去使用.h內部function
- ImageSensor function
  - int ImageSensor_Init()
    - ImageSensor_Init() 函式會初始化圖像感測器，設置感測器的頻率和時序

      並初始化對應的硬體（ CCAP 影像擷取控制器 ）

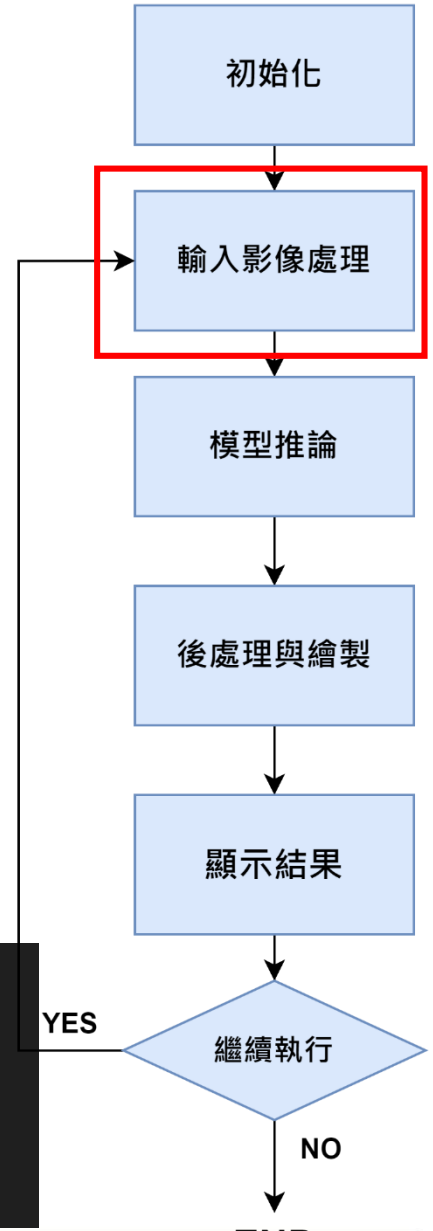
ImageSensor.h

```
int ImageSensor_Init(void);
int ImageSensor_Capture(uint32_t u32FrameBufAddr);
int ImageSensor_Config(E_IMAGE_FMT eImgFmt, uint32_t u32ImgWidth, uint32_t u32ImgHeight);
int ImageSensor_TriggerCapture(uint32_t u32FrameBufAddr);
int ImageSensor_WaitCaptureDone(void);
```



3.影像處理function介紹

- ImageSensor function
  - int ImageSensor_Capture()
    - 開始一次影像捕捉（Capture），將捕捉到的影像存入指定的緩衝區（由 u32FrameBufAddr 指定）
  - int ImageSensor_Config()
    - 配置影像感測器的影像參數：
      - eImgFmt: 影像格式（例如 RGB、YUV、灰階等）。
      - u32ImgWidth: 影像的寬度。
      - u32ImgHeight: 影像的高度。



ImageSensor.h

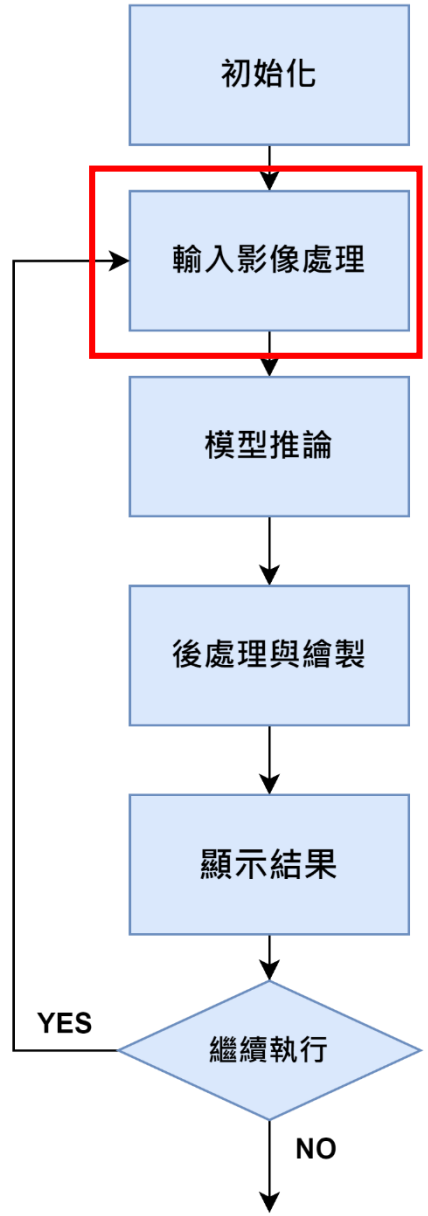
```
int ImageSensor_Init(void);
int ImageSensor_Capture(uint32_t u32FrameBufAddr);
int ImageSensor_Config(E_IMAGE_FMT eImgFmt, uint32_t u32ImgWidth, uint32_t u32ImgHeight);
int ImageSensor_TriggerCapture(uint32_t u32FrameBufAddr);
int ImageSensor_WaitCaptureDone(void);
```

3.影像處理function介紹

- Imagesensor - **捕捉影像** (Capture Image)：
  - ImageSensor_Capture() 用於啟動影像擷取並將擷取到的影像存入指定的記憶體地址。
  - ImageSensor_TriggerCapture() 用於觸發圖像擷取，並設置快門訊號。
  - ImageSensor_WaitCaptureDone() 用來等待影像擷取完成，並處理超時情況。

ImageSensor.h

```
int ImageSensor_Init(void);
int ImageSensor_Capture(uint32_t u32FrameBufAddr);
int ImageSensor_Config(E_IMAGE_FMT eImgFmt, uint32_t u32ImgWidth, uint32_t u32ImgHeight);
int ImageSensor_TriggerCapture(uint32_t u32FrameBufAddr);
int ImageSensor_WaitCaptureDone(void);
```

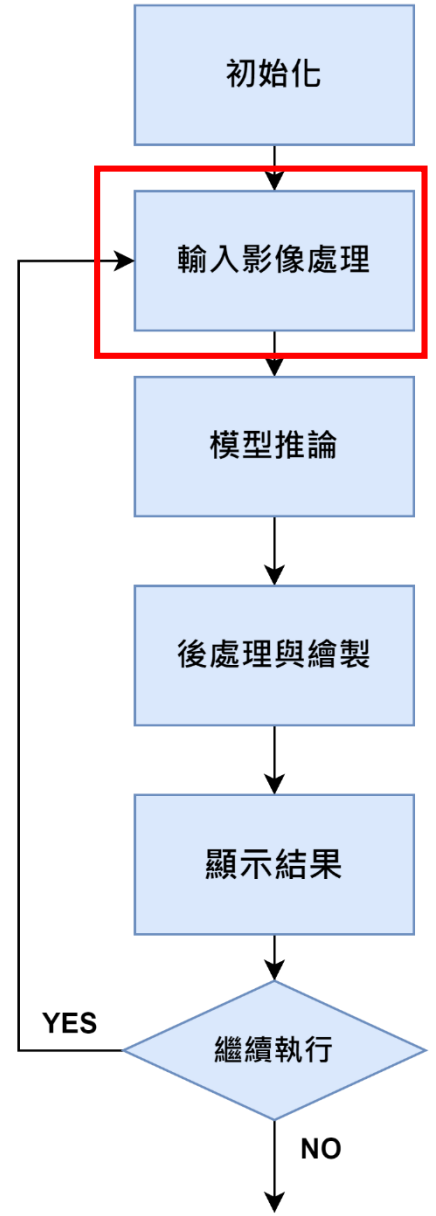


4.輸入影像處理

- Main.cpp - **framebuf**（幀緩衝區）
  - 在 main_task() 函數的主循環中，程式不斷地從相機獲取新幀
  - 因此需要Framebuffer 是用來暫存從相機捕捉的影像幀，然後再將其送入 YOLOX_nano 模型進行推理。
  - S_FRAMEBUF 來管理影像數據與檢測結果。
- Code
  - 呼叫framebuffer.h
  - 定義buffer狀態: Empty、FULL與INF
    - 根據不同的狀態，會做不同的動作

```
24  #include "framebuffer.h"
```

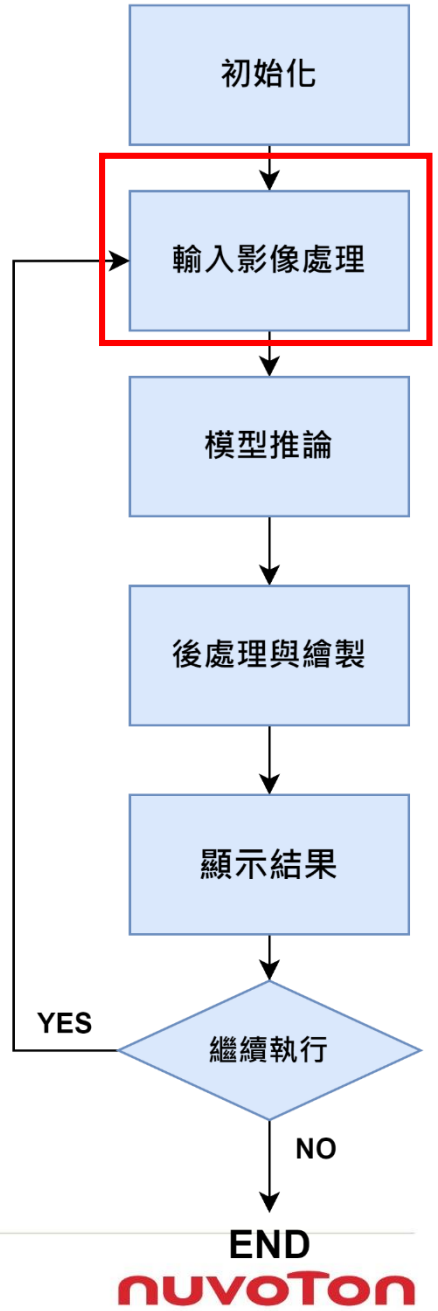
```
46  typedef enum
47  {
48      eFRAMEBUF_EMPTY,
49      eFRAMEBUF_FULL,
50      eFRAMEBUF_INF
51  } E_FRAMEBUF_STATE;
```



5.輸入影像處理 – Frame Buffer用途介紹

- Frame Buffer 資料狀態遷移邏輯

狀態	功能	Main.cpp對應處理部分
eFRAMEBUF_EMPTY	接收新的圖像數據	ImageSensor_Capture() 捕捉圖像，更新狀態為 FULL
eFRAMEBUF_FULL	圖像前處理與推論	imlib_nvt_scale() 圖像縮放 推論完後更新狀態為 INF
eFRAMEBUF_INF	繪製檢測框並顯示結果，準備下一輪處理	DrawImageDetectionBoxes() 繪製框，重設狀態為 EMPTY

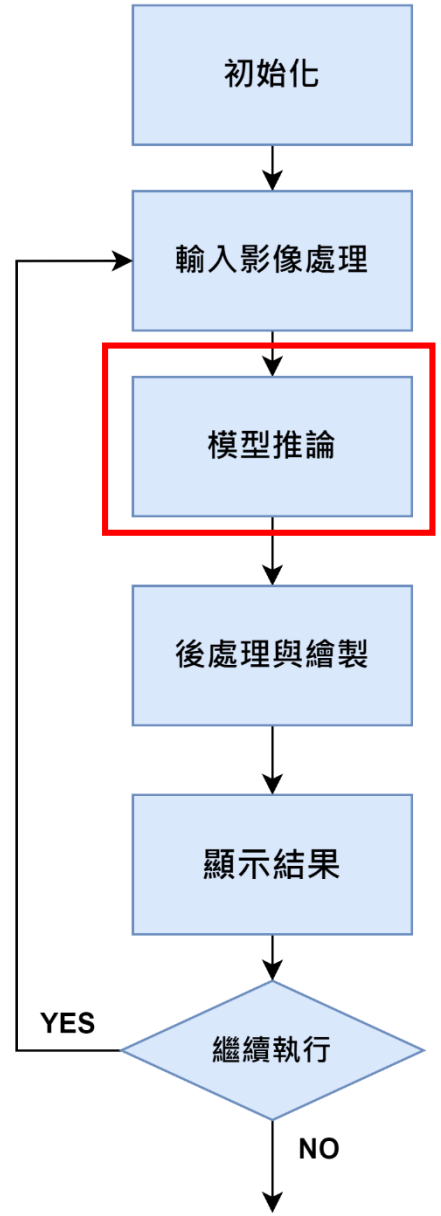


六、C++軟體設計介紹– 模型推理

- 推理任務創建：
  - 使用 FreeRTOS 的 xTaskCreate 函式創建推理任務，處理推理過程所需的影像資料。
- 模型推理:
  - 在eFRAMEBUF_FULL時會執行推理

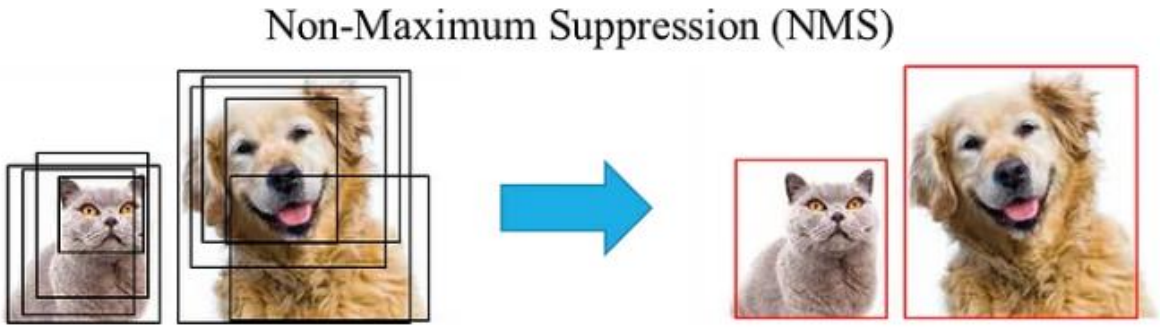
```
434             //trigger inference
435             inferenceJob->responseQueue = inferenceResponseQueue;
436             inferenceJob->pPostProc = &postProcess;
437             inferenceJob->modelCols = inputImgCols;
438             inferenceJob->mode1Rows = inputImgRows;
439             inferenceJob->srcImgWidth = fullFramebuf->frameImage.w;
440             inferenceJob->srcImgHeight = fullFramebuf->frameImage.h;
441             inferenceJob->results = &fullFramebuf->results;
442
443             xQueueSend(inferenceProcessQueue, &inferenceJob, portMAX_DELAY);
444             fullFramebuf->eState = eFRAMEBUF_INF;
445         }
446
```

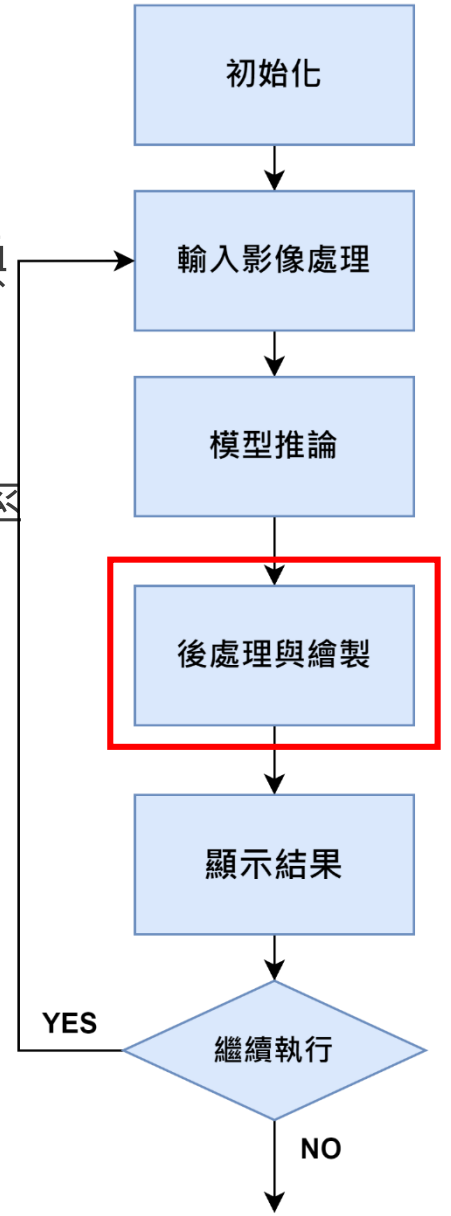
  - 並在推理完成後buf的eState 設成eFRAMEBUF_INF



(一)、後處理與繪製

1. **縮放偵測框**：從推理網路輸出大小 (imgNetRows, imgNetCols) 轉換回原始大小 (imgSrcRows, imgSrcCols)，確保偵測框對應原始圖像
2. **提取偵測結果**：從模型輸出張量中提取出物件的邊界框和類別機率
3. **執行 NMS**：應用 NMS 來移除高度重疊的偵測框，
4. **結果輸出**：將最終的偵測結果儲存到 resultsOut 向量中。

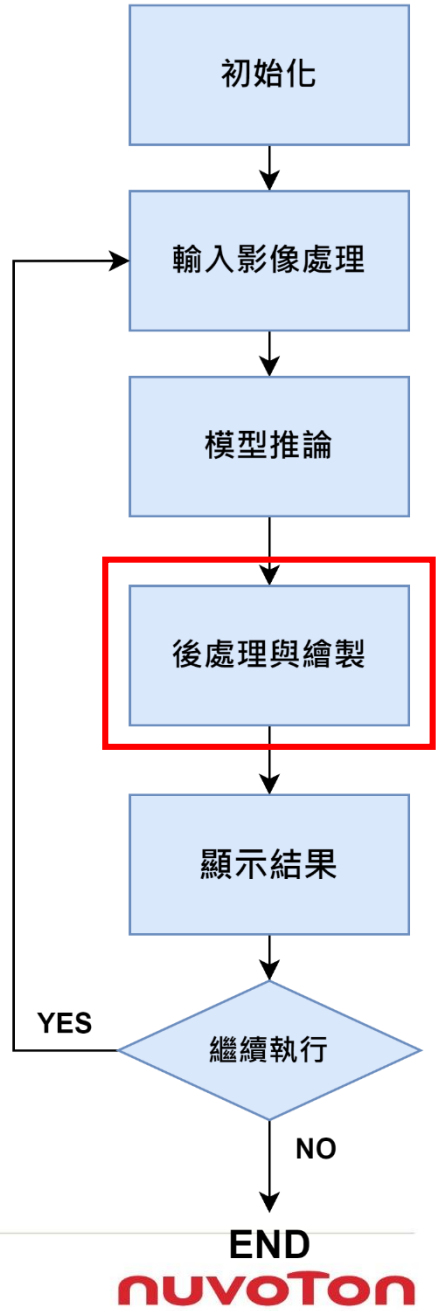




(一)、後處理與繪製

- PresentInferenceResult 函式
  - 顯示推論結果：程式會將每一個偵測到的物件標註其類別及在影像中的位置

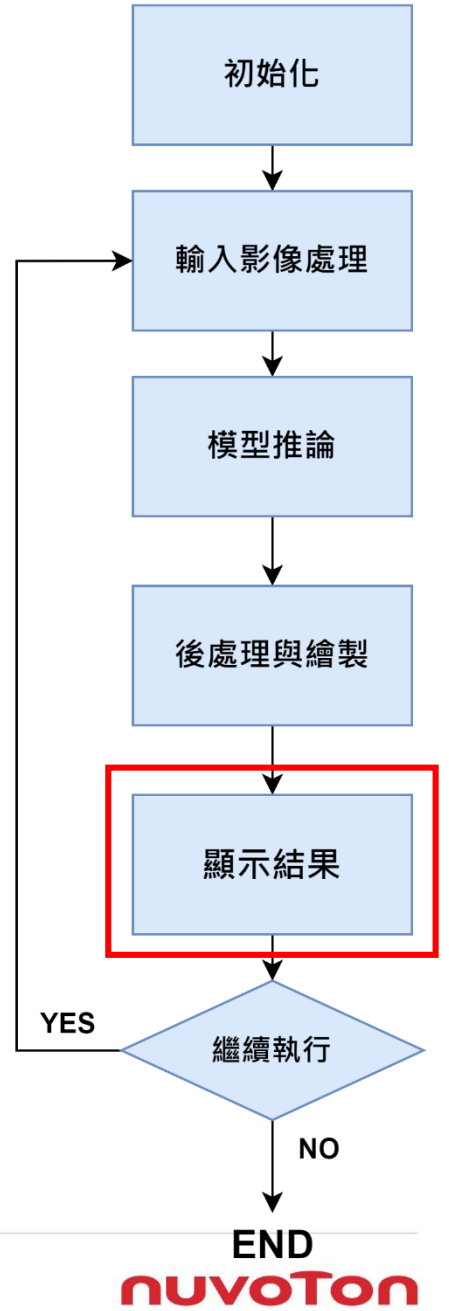
```
static bool PresentInferenceResult(const std::vector<arm::app::object_detection::DetectionResult> &results,
                                   std::vector<std::string> &labels)
{
    /* If profiling is enabled, and the time is valid. */
    info("Final results:\n");

    for (uint32_t i = 0; i < results.size(); ++i)
    {
        info("%" PRIu32 ") %s(%f) -> %s {x=%d,y=%d,w=%d,h=%d}\n", i,
             labels[results[i].m_cls].c_str(),
             results[i].m_normalisedVal, "Detection box:",
             results[i].m_x0, results[i].m_y0, results[i].m_w, results[i].m_h);
    }

    return true;
}
```



(二)、顯示結果

- DrawImageDetectionBoxes 函式
  - 函式會在偵測到的物件位置上繪製偵測框，並顯示標籤，將物件偵測的結果以視覺化的方式呈現出來

```
static void DrawImageDetectionBoxes(
    const std::vector<arm::app::object_detection::DetectionResult> &results,
    image_t *drawImg,
    std::vector<std::string> &labels)
{
    for (const auto &result : results)
    {
        imlib_draw_rectangle(drawImg, result.m_x0, result.m_y0, result.m_w, result.m_h, COLOR_B5_MAX, 1, false);
        imlib_draw_string(drawImg, result.m_x0, result.m_y0 - 16, labels[result.m_cls].c_str(), COLOR_B5_MAX, 2, 0, 0, false,
                          false, false, false, 0, false, false);
    }
}
```



(三)、設定Label檔

- 在Model資料夾內更新Label.cpp
  - 根據自己的訓練集(藥物資料集)
    - 分別是TellWell、Postan與Nacid 將所有class列在labelVec[] 內部

Label.cpp

```
7   static const char *labelsVec[] LABELS_ATTRIBUTE =
8   {
9           "Tellwell",
10          "Postan"
11          "Nacid",
12  };
13
```

(四)、DEMO

- 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