

# 智慧居家2-人員追蹤

Joy of innovation  
**nuvoTon**

# | 大綱

- 柒、智慧居家2 – 人員追蹤
  - 一、 案例介紹
  - 二、 模型比較
  - 三、 labelImg
  - 四、 在PC上使用Anaconda進行模型訓練
  - 五、 開發版上的推論程式系統流程步驟
  - 六、 DEMO

# I 一、案例介紹

- (一)專案摘要
- (二)需求及使用場景舉例
- (三) 案例呈現結果

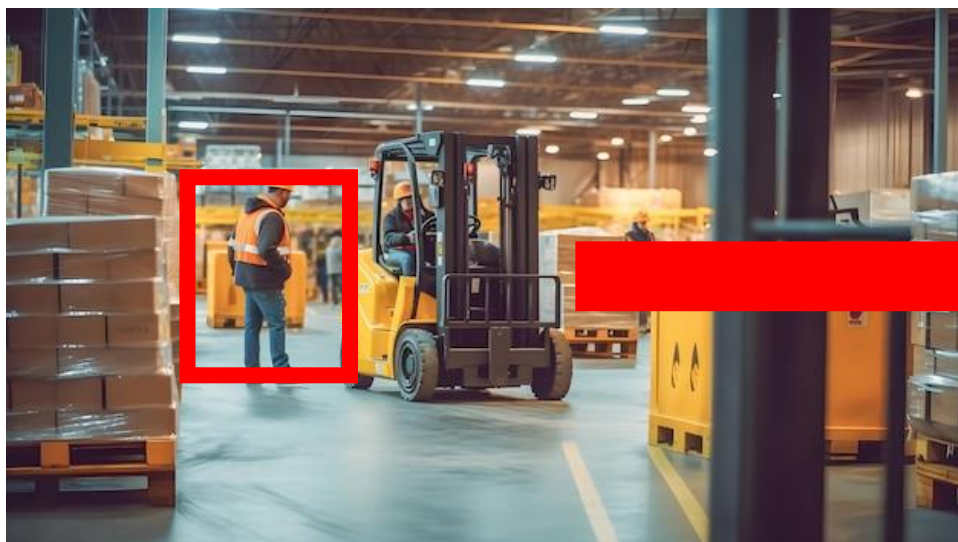
# | (一) 專案摘要

- 本專案旨在探討利用機器學習技術於新唐M55M1嵌入式平台上實現人員檢測的可行性與應用。隨著智慧化技術的快速發展，人員檢測在安防、智慧建築、工業自動化等領域中逐漸成為重要的應用場景。本研究選擇新唐M55M1作為硬體平台，因其具備高效能的處理器和低功耗特性，適合應用於物聯網（IoT）和智能設備中。



## I (二)需求及使用場景舉例

- 安全性：應用於危險區域（如工廠、施工現場）時，需監控人員進入或離開特定區域的情況。
- 在工業生產線間禁止人員進入的作業區域安裝檢測設備，監測是否有人員進入

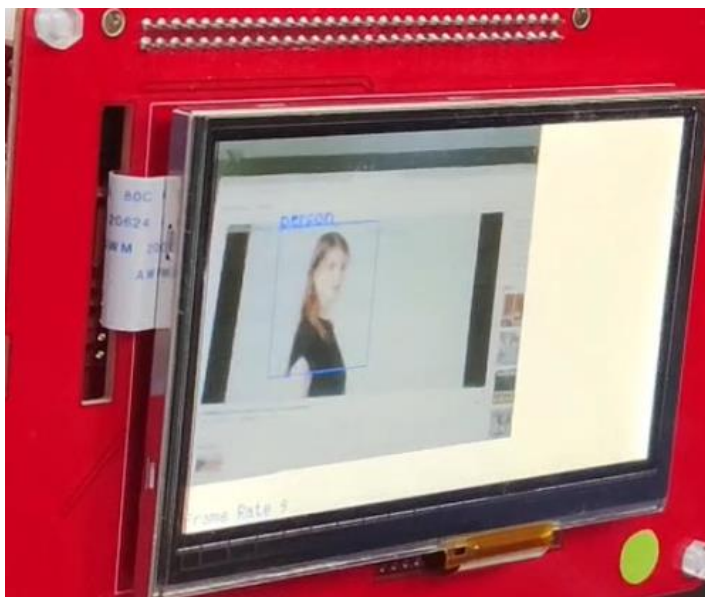


Source:[https://jp.freepik.com/premium-ai-image/workers-with-safety-vests-hard-hats-working-with-forklift-machines-warehouse-xaworking-with-forklift-machines-warehouse-with-pallets-boxes-generative-ai\\_46152009.htm](https://jp.freepik.com/premium-ai-image/workers-with-safety-vests-hard-hats-working-with-forklift-machines-warehouse-xaworking-with-forklift-machines-warehouse-with-pallets-boxes-generative-ai_46152009.htm)



## I (三) 案例呈現結果

- 圖中框選部分為人物出現的部分，該人員檢測系統達到了92.5%的正確率（136/148），顯示出在嵌入式平台上實現高效、準確的檢測。本研究針對模型進行了量化和裁剪，確保能夠在資源受限的環境中高效運行。



TinyML 訓練

輕量化模型  
與轉換

模型Vela  
編譯與配置

C語言  
程式撰寫

部署至開發板

## I 二、模型比較

(一)模型大小與推論速度

(二)訓練效率

(三)應用場景

# I (一)模型大小與推論速度

- **YOLOX Nano (本專案使用)**
  - 極小的模型大小，約 **0.91M** 參數數量。
  - 高效能針對低算力設備進行優化，適用於即時應用，如智慧攝影機、無人機等。
  - 推論速度非常快，在 NUVOTON M55M1 等邊緣運算平台上表現出色。
- **其他 YOLO 模型**
  - **YOLOv4 / YOLOv5**：針對高效能 GPU 設計，模型大小從小到大分層次。
  - **YOLOX 標準版本**：支援不同尺寸的模型（YOLOX-S、YOLOX-M 等），適用於更高的準確率要求，但推論速度相對較慢，尤其是大模型



## I (二)訓練效率

- **YOLOX Nano (本專案使用)**

- 使用 Anchor-free 結構，減少了訓練中對 anchor 的依賴，簡化了超參數調整。
- 相較於其他 YOLO 模型更容易適配新的資料集。

- **其他 YOLO 模型**

- **YOLOv4 / YOLOv5**：使用 Anchor-based 方法，對於資料標註格式和 anchor 設置有較高的要求，可能需要更多調參。
- **YOLOX 標準版本**：同樣使用 Anchor-free 方法，與 YOLOX Nano 的訓練流程一致，但需要更高的算力。

## I (三)應用場景

- **YOLOX Nano (本專案使用)**

- 適用於**邊緣運算**或**即時性**要求高的場景：
  - 智慧家庭（例如智慧門鎖、人員偵測）。
  - 移動設備（手機 App 的物體偵測）。
  - 工業邊緣計算（小型攝影機）。

- **其他 YOLO 模型**

- **YOLOv4 / YOLOv5**：適用於伺服器或高效能 GPU，常用於大規模的影像分析或實時視訊流分析。
- **YOLOX 標準版**：可用於專業應用，如白駕車感測、商業監控系統等。

## 三、labellImg

- 使用LabelImg 標記1000張圖片框線，將COCO檔存放於Annotations資料夾以利後續模型訓練



## I 三、labelImg

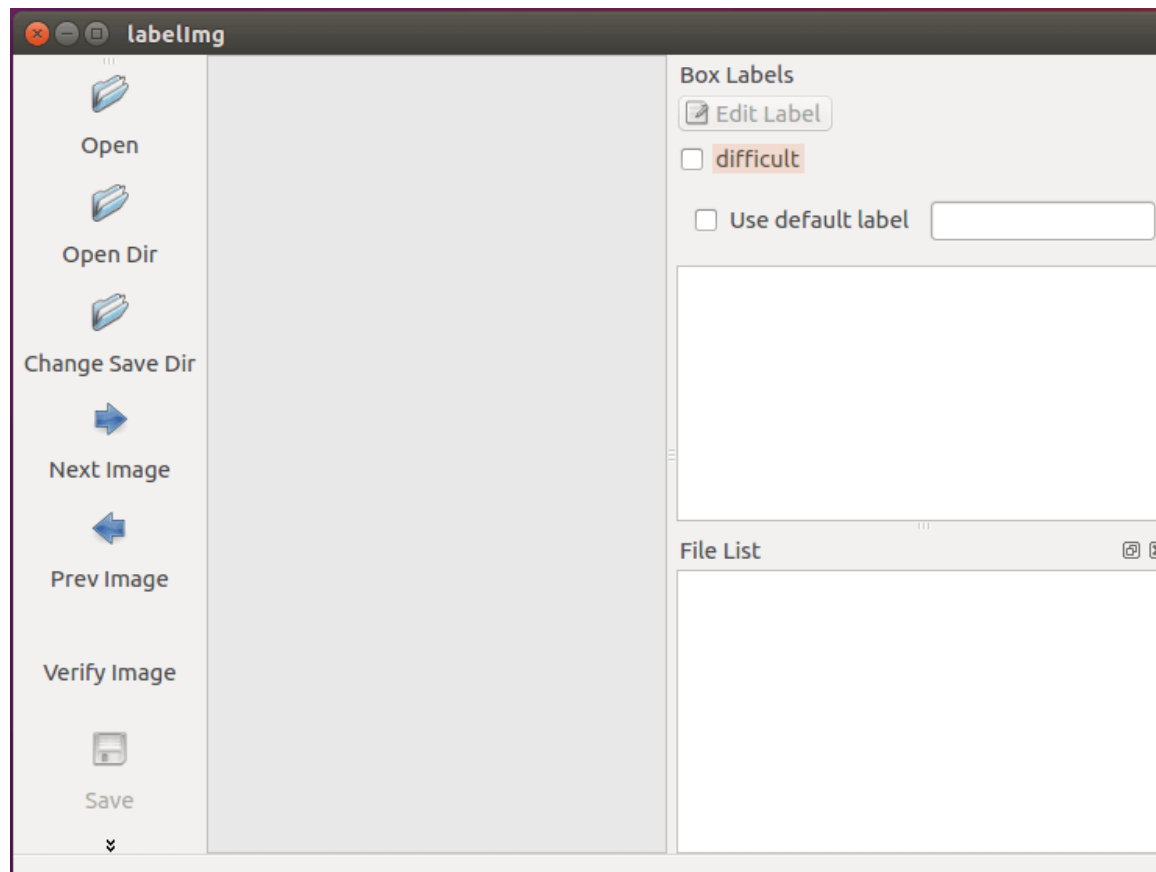
於終端輸入以下指令

- 下載 labelImg 套件
  - `$pip install labelImg`
- 執行 labelImg
  - `$labelImg`

參考教學步驟：<https://blog.csdn.net/knighthood2001/article/details/125883343>

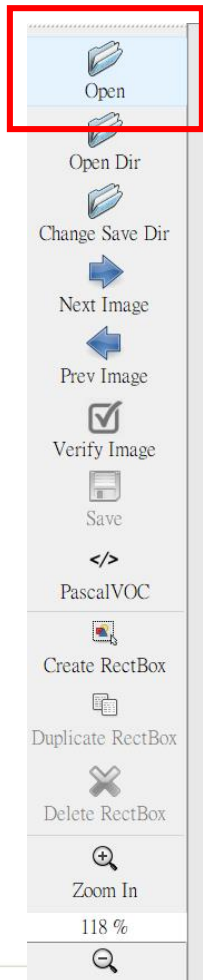
## 三、labelImg

- labelImg執行視窗



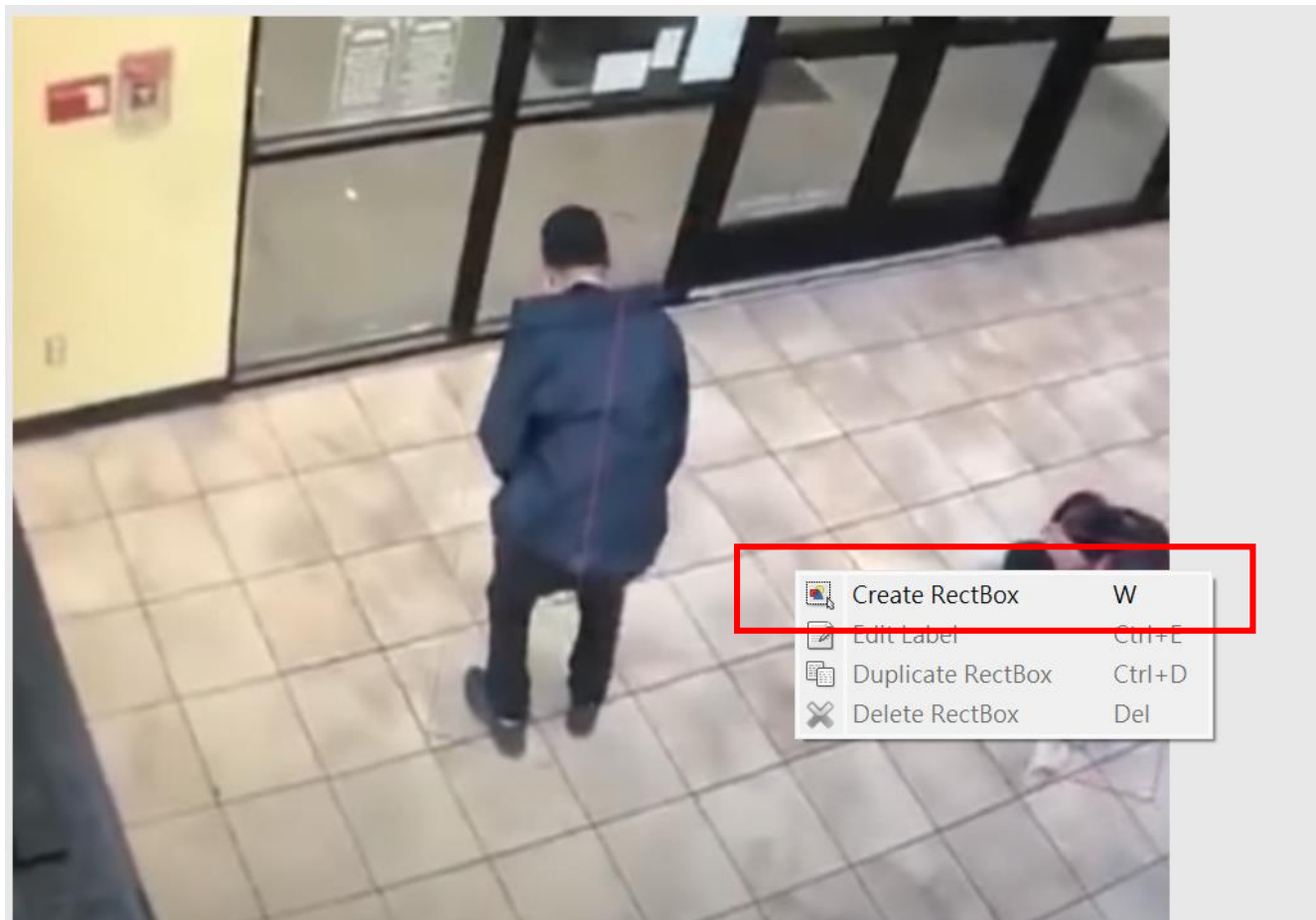
# 三、labellmg

- 選擇open開啟圖片



## 三、labellmg

- 右鍵選擇 create Rectbox





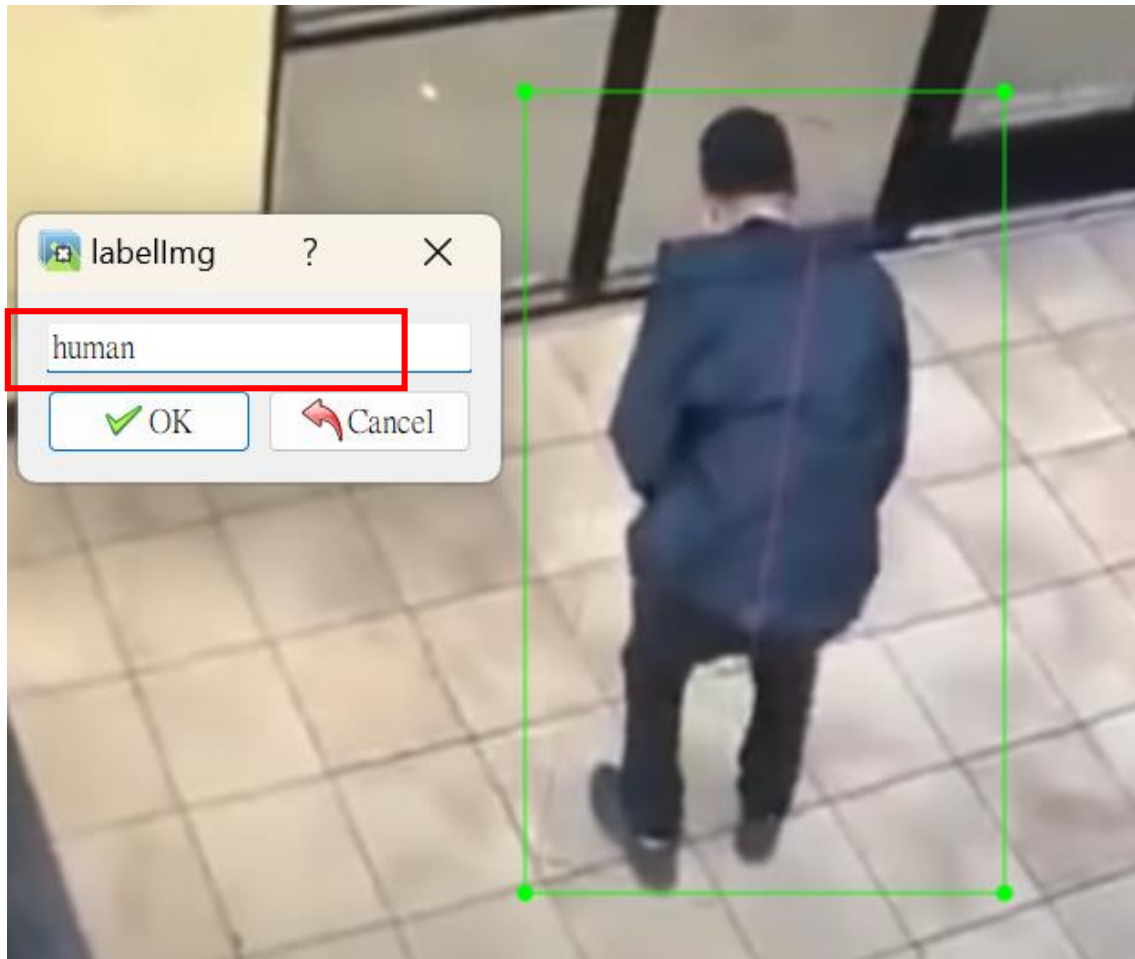
## 三、labellmg

- 將想框選部分框起來



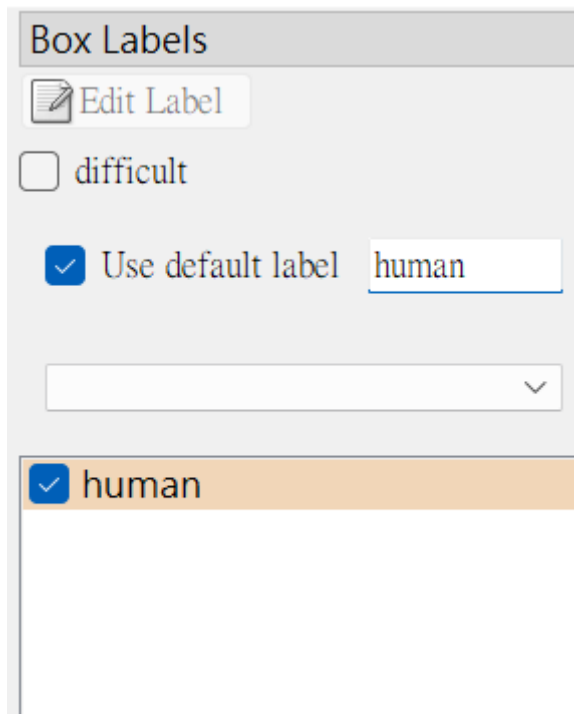
## 三、labellmg

- 輸入物件名稱



## 三、labellmg

- 在標示多個物件時，Labellmg 會自動列出已經標示過的名稱，所以如果有重複的名稱，就可以直接用滑鼠在選單上點選



The screenshot shows the 'Box Labels' panel in the Labellmg application. It contains an 'Edit Label' button, a checkbox for 'difficult', and a checked checkbox for 'Use default label' with a text input field containing 'human'. Below these is a dropdown menu. At the bottom, a list of labels is shown, with 'human' selected and highlighted in orange.

## 四、在PC上使用Anaconda環境進行模型訓練

- (一)本專案使用之硬體及模型規格
- (二)資料集準備
- (三) 建立Anaconda環境
- (四)Yolox nano模型訓練
- (五) ONNX
- (六) Vela compiler

# I (一)本專案使用之硬體及模型規格

- GPU: RTX 3060 Ti
- CUDA 版本: 11.8
- Pytorch版本: 2.0.0
- 模型參數: EPOCH 200, BATCH SIZE 64
- 模型訓練時間: 1.5 hours

## I (二)資料集準備

### 資料集準備

- **LabelImg**
- **kaggle**

### 模型訓練

- 選擇輕量級的模型
  - **yolox nano**

### 模型框架轉換

- 利用**ONNX**實現**Pytorch => Tensorflow lite**

## I (二) 資料集準備

於kaggle下載資料集





## I (二)資料集準備

- 訓練框架：PyTorch
- 訓練模型：Yolox-nano
- 資料集：Human detection dataset
- 標註方式: labelImg
- 格式: COCO JSON
  - Train Set : 343 images
  - Valid Set : 148 images
- 模型資源:[https://github.com/MaxCYCHEN/yolox-ti-lite\\_tflite\\_int8](https://github.com/MaxCYCHEN/yolox-ti-lite_tflite_int8)
- 資料集:[Human detection](#)

## I (三) 建立Anaconda環境

- 環境建立步驟:
  - 建立python環境
    - `$conda create --name yolox_nu python=3.10`
    - `$conda activate yolox_nu`
  - upgrade pip
    - `$python -m pip install --upgrade pip setuptools`
  - 安裝CUDA,PyTorch,MMCV (本案使用CUDA11.8,torch2.0)
    - `$python -m pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118`

## I (三) 建立Anaconda環境

- 環境建立步驟:
  - 根據您的硬體配置安裝 mmcv (本案使用2.0.1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`

## I (四) Yolo nano 模型訓練

- 使用預訓練好的模型訓練
  - `exps/default/yolox_nano_ti_lite_nu.py`
- 模型訓練:
  - 準備好自己訓練資料並整理成下列形式並放在dataset目錄中
  - `Datasets/<your_datasets_name>/`
    - `annotations/`
      - `train_annotation_json_file`
      - `val_annotation_json_file`
    - `train2017/`
      - `train_img`
    - `val2017/`
      - `validation_img`

```
# Define yourself dataset path
self.data_dir = "datasets/coco128"
self.train_ann = "train_annotations.coco.json"
self.val_ann = "val_annotations.coco.json"
```



設定自己的資料集路徑

## (四) YoloX nano模型訓練

- 更改yolox\_nano\_ti\_lite\_nu.py之參數

```
# ----- model config ----- #  
self.num_classes = 6  
self.depth = 0.33  
self.width = 0.25  
self.input_size = (320, 320)  
self.random_size = (10, 20)  
self.mosaic_scale = (0.5, 1.5)  
  
# ----- training config ----- #  
self.warmup_epochs = 5  
self.max_epoch = 200  
# ----- testing config ----- #  
self.test_size = (320, 320)
```

根據自己的資料集種類  
設定class數

設定epoch為200

- 開始訓練  
輸入指令：

\$python tools/train.py -f <MODEL\_CONFIG\_FILE> -d 1 -b <BATCH\_SIZE> --fp16 -o -c  
<PRETRAIN MODEL PATH>

## I (五) ONNX

- **ONNX格式(Open Neural Network Exchange)**
  - 跨框架模型轉換：允許不同的深度學習框架間自由轉換模型，解決框架之間的兼容性問題。
- **TensorFlow Lite**
  - 提供模型量化、硬體加速等技術，提升模型在設備上的推理速度和效率。
  - 適合移動端與嵌入式設備的模型與低功耗設備部署。

### 模型框架轉換流程



# | (五)ONNX

- 模型框架轉換

- 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]]]]"`



## I (六) Vela compiler

- Vela 是一款專為 Arm 微型神經網絡處理單元（microNPU）設計的編譯器。
- 主要功能：
  - 優化 TensorFlow Lite 模型，生成經過優化的 TFLite 文件
  - 將模型中適合的運算操作分配到 NPU，藉此加速模型推理
  - 記憶體訪問模式優化、和指令流優化

## I (六) Vela compiler

- Vela編譯步驟:
  - 將輕量化完的模型放到 vela\generated\
    - 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、開發版上的推論程式系統流程圖

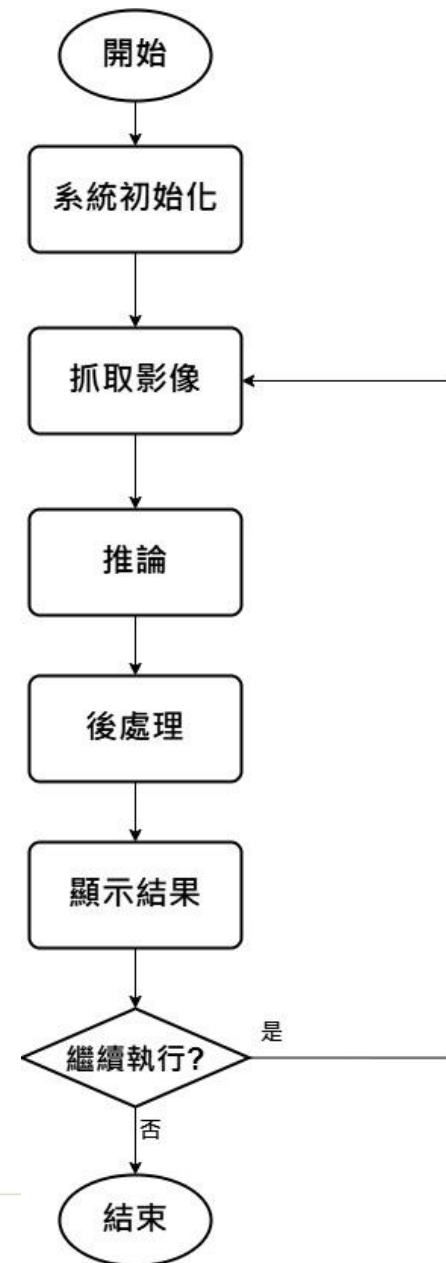
(一)系統初始化 (System Initialization)

(二)捕捉影像 (Capture Image)

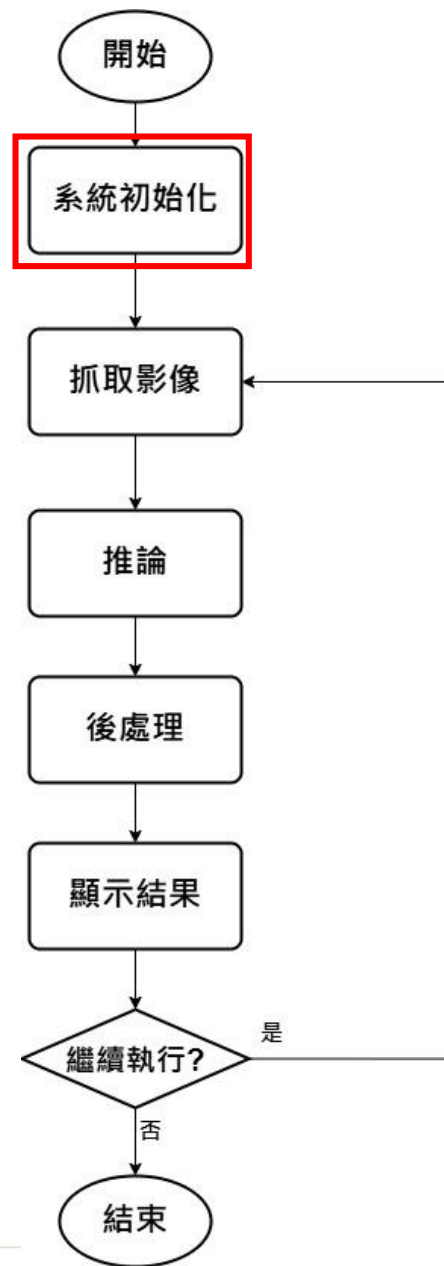
(三)推論 (Inference)

(四)後處理與繪製偵測結果 (Post-Processing & Draw Results)

(五)顯示結果 (Display Results)



# I (一)系統初始化



# | (一)系統初始化

- **BoardInit() 函數：**

- 負責執行硬體相關的初始化操作。確保基本硬體資源準備完成，
- 執行的目的：確保整個系統其他模組（如影像處理、神經網路推理）的運行有穩定的CLK與可靠的通訊機制。提供基本的輸入/輸出功能。

```
static void SYS_Init(void)
```

```
{
```

```
/*-----主程式CLK初始化-----*/
/* Init System Clock
/*-----*/

/* 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. */
```

```
int BoardInit(void)
```

```
{
```

```
/* Unlock protected registers */
SYS_UnlockReg();

SYS_Init();

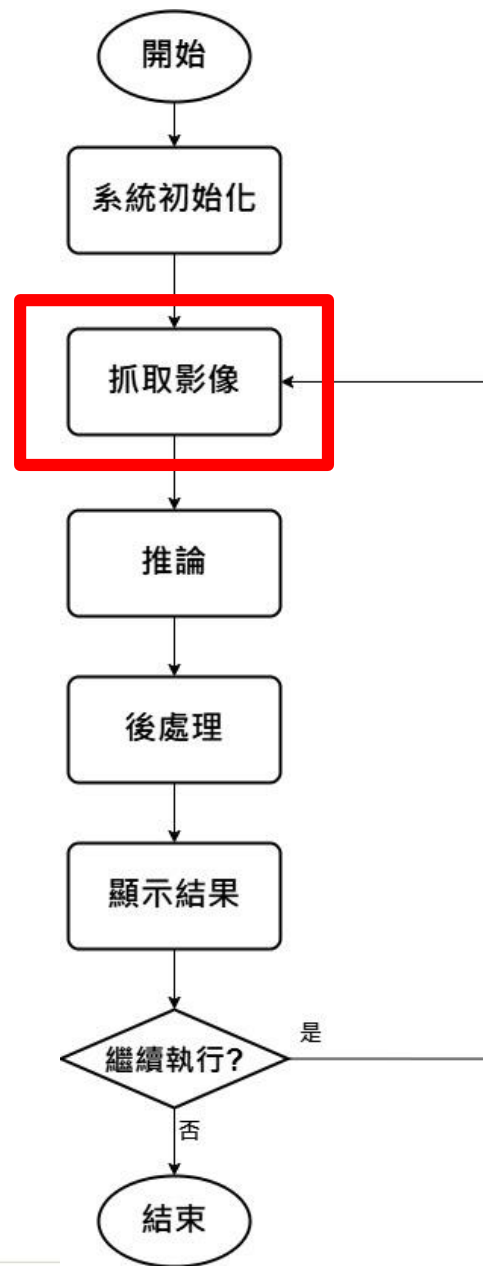
/* UART init - will enable valid use of printf (stdout
/* re-directed at this UART (UART6) */
InitDebugUart();

SYS_LockReg(); /* Unlock register lock protect */

HyperRAM_Init(HYPERRAM_SPIM_PORT);
/* Enter direct-mapped mode to run new applications */
SPIM_HYPER_EnterDirectMapMode(HYPERRAM_SPIM_PORT);
```

開發版周邊硬體初始化

## I (二) 捕捉影像



## I (二)捕捉影像

- 緩衝區 ( frame buffer ) : 根據緩衝區的狀態來查找符合條件的緩衝
- get\_empty\_framebuf() : 用於準備捕捉影像時，系統需要一個空的緩衝區來存儲新影像數據。

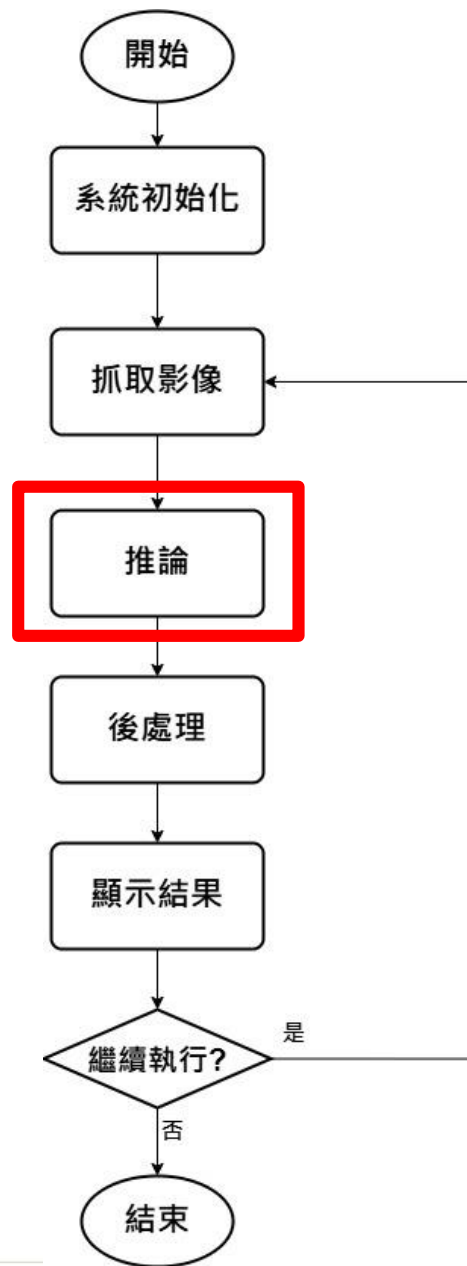
```
//frame buffer managemnet function
static S_FRAMEBUF *get_empty_framebuf()
{
    int i;

    for (i = 0; i < NUM_FRAMEBUF; i ++ )
    {
        if (s_asFramebuf[i].eState == eFRAMEBUF_EMPTY)
            return &s_asFramebuf[i];
    }

    return NULL;
}
```



## I (三)推論



## I (三)推論

- get\_full\_framebuf() :
- 從已填滿的緩衝區取得一個數據，然後對其進行推論。

人員檢測模型推論程式碼

```
//trigger inference
inferenceJob->responseQueue = inferenceResponseQueue;
inferenceJob->pPostProc = &postProcess;
inferenceJob->modelCols = inputImgCols;
inferenceJob->modelRows = inputImgRows;
inferenceJob->srcImgWidth = fullFramebuf->frameImage.w;
inferenceJob->srcImgHeight = fullFramebuf->frameImage.h;
inferenceJob->results = &fullFramebuf->results;

xQueueSend(inferenceProcessQueue, &inferenceJob, portMAX_DELAY);
fullFramebuf->eState = eFRAMEBUF_INF;
```

函式宣告

```
int i;

for (i = 0; i < NUM_FRAMEBUF; i ++){
    if (s_asFramebuf[i].eState == eFRAMEBUF_FULL){
        return &s_asFramebuf[i];
    }
}

return NULL;
```

## I (三)推論

- PresentInferenceResult 函式

- 人員推論中間結果：

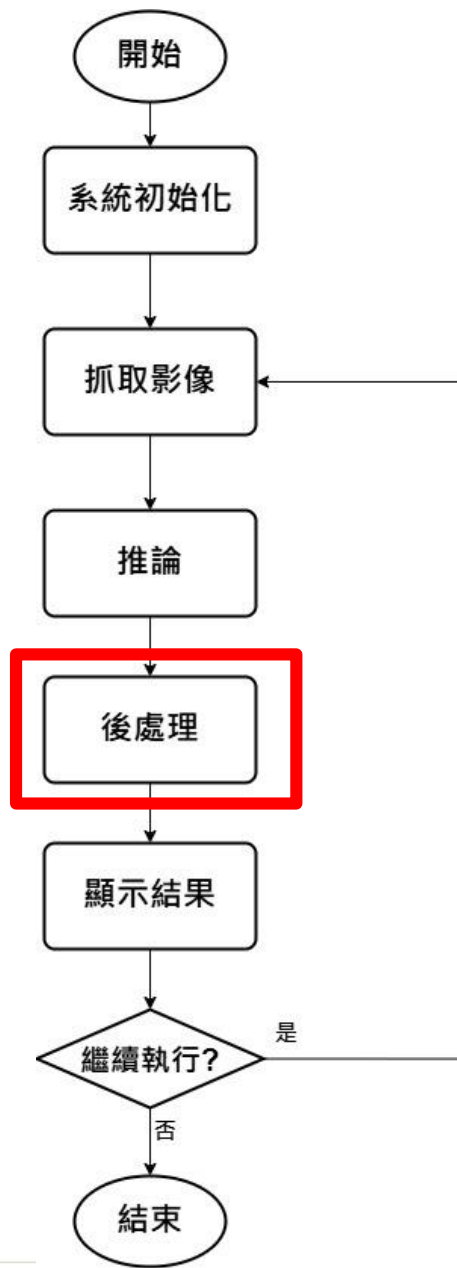
- 程式會將每一個偵測到的物件標註其類別及在影像中的位置。

```
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;
}
```

## I (四)後處理



## I (四)後處理

後處理流程有以下步驟：

1. **縮放偵測框**：從推理網路輸出轉換回原始影像大小，確保偵測框對應到原始圖像。
2. **提取偵測結果**：從模型的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 來移除重疊的偵測框，只保留最有可能的框。
4. **結果輸出**：將最終的偵測結果儲存到 resultsOut 向量中。

## (四)後處理

- InsertTopNDetection:將新的檢測結果插入到指定的前N個檢測匡列表中，根據objectness分數進行排序
- 參數Std::forward\_list<image::Detection> &detection:引用類型的單向鏈表，存儲檢測結果。image::Detection &det:新檢測結果，嘗試插入到列表中。

```
void DetectorPostprocessing::InsertTopNDetections(std::forward_list<image::Detection> &detections, image::Detection &det)
{
    std::forward_list<image::Detection>::iterator it;
    std::forward_list<image::Detection>::iterator last_it;

    for (it = detections.begin(); it != detections.end(); ++it)
    {
        if (it->objectness > det.objectness)
            break;

        last_it = it;
    }

    if (it != detections.begin())
    {
        detections.emplace_after(last_it, det);
        detections.pop_front();
    }
}
```

## I (四)後處理

- DetectorPostprocessing的功能是實現 YOLO 類型神經網路推理的後處理邏輯，從網路輸出的特徵圖中提取檢測框（ bounding boxes ），
- 並根據「物件性分數」（ objectness score ）過濾出前 **Top-N** 的檢測結果。

```
void DetectorPostprocessing::GetNetworkBoxes(Network &net, int imageWidth, int imageHeight, float threshold,
{
    int numClasses = net.numClasses;
    int num = 0;
    auto det_objectness_comparator = [](image::Detection & pa, image::Detection & pb)
    {
        return pa.objectness < pb.objectness;
    };
};
```

## I (四)後處理

```
DetectionResult(double normalisedVal, int x0, int y0, int w, int h, int cls) :
    m_normalisedVal(normalisedVal),
    m_x0(x0),
    m_y0(y0),
    m_w(w),
    m_h(h),
    m_cls(cls)
{
}

DetectionResult() = default;
~DetectionResult() = default;

double m_normalisedVal{0.0};
int m_x0{0};
int m_y0{0};
int m_w{0};
int m_h{0};
int m_cls{0};
};
```

- DetectionResult主要用於示物件偵測的結果，例如偵測框的位置、大小，以及對應的分類
- m\_normalizedVal:  
偵測結果信心值，表示該框屬於某一類別的機率
- m\_x0, m\_y0:  
偵測框的左上角座標
- m\_w, m\_h:  
框的寬度及高度，用於定義框的尺寸
- m\_cls:  
偵測結果的類別標籤(class label)



## (四)後處理

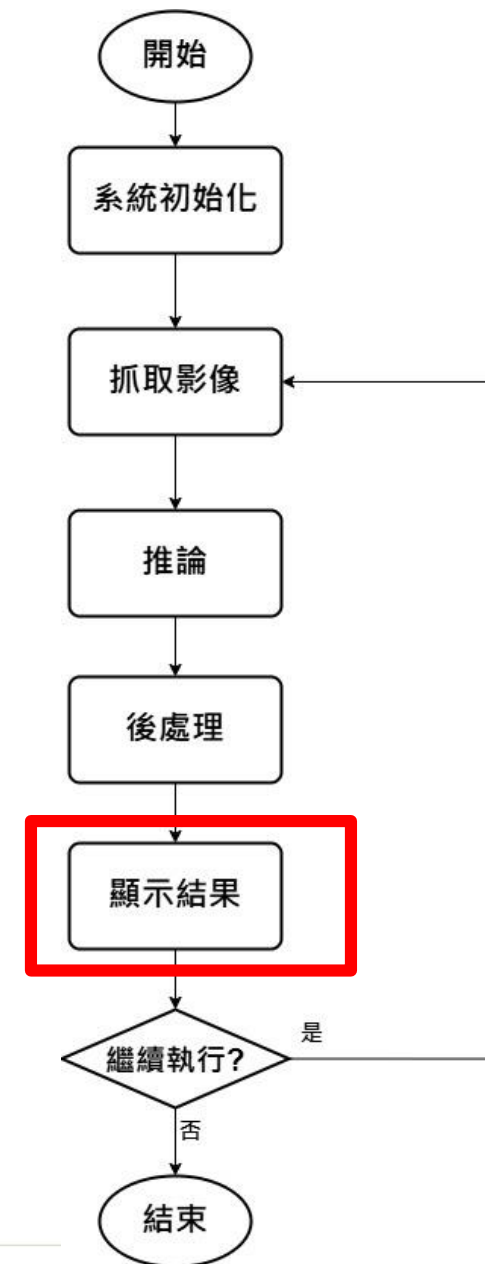
- DrawImageDetectionBoxes 函式
  - 函式會在偵測到的物件位置上繪製偵測框，並顯示標籤，將物件偵測的結果以視覺化的方式呈現出來。

```
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);  
    }  
}
```

## I (五)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);  
    }  
}
```



## | 六、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