# 智慧工廠 – 火災偵測

# Outline

- 貳、智慧工廠2 – 火災偵測
  - 一、 案例介紹
  - 二、 資料集蒐集
  - 三、 在PC上使用Anaconda環境進行模型訓練
  - 四、 開發版上的推論程式系統流程步驟
  - 五、 DEMO影片

# 一、案例介紹 – 火災偵測

- **專案摘要**
  - 本專案利用 Roboflow 開源資料庫的數據集，通過機器學習方式在 YOLOX Nano 上進行訓練。接著經由 ONNX 進行模型框架轉換，轉換成 TensorFlow Lite 框架，並應用 Full-INT8 輕量化技術達到開發板模型大小限制。最後，通過 Vela 編譯器最佳化，確保運算全部在 NPU (Neural Processing Unit) 上執行實現一套即時、高效且節能的火災偵測系統。

# (一) 實際應用說明

- 圖中框選部分為火光偵測區域，系統標註出FIRE，能即時反應火災情況，目前使用火災圖片測試100張，正確率為98%。



發現火災
發出警報

# 二、資料集與訓練環境

| | |
|---|---|
| 資料集準備 | • Roboflow |
| 模型訓練 | • 選擇符合開發板限制的輕量型模型<br>  • yolox nano |
| 模型框架轉換 | • 透過onnx進行模型轉換<br>• Pytorch => onnx =>Tensorflow lite |

# (一) 資料集蒐集

- 虛擬環境 : Anaconda
- 訓練框架 : PyTorch
- 訓練模型 : Yolox-nano
- 資料集 : fire detection(Roboflow)
- 格式: COCO JSON
  - Train Set : 1210 images
  - Valid Set : 347 images
  - Test Set : 164 images

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

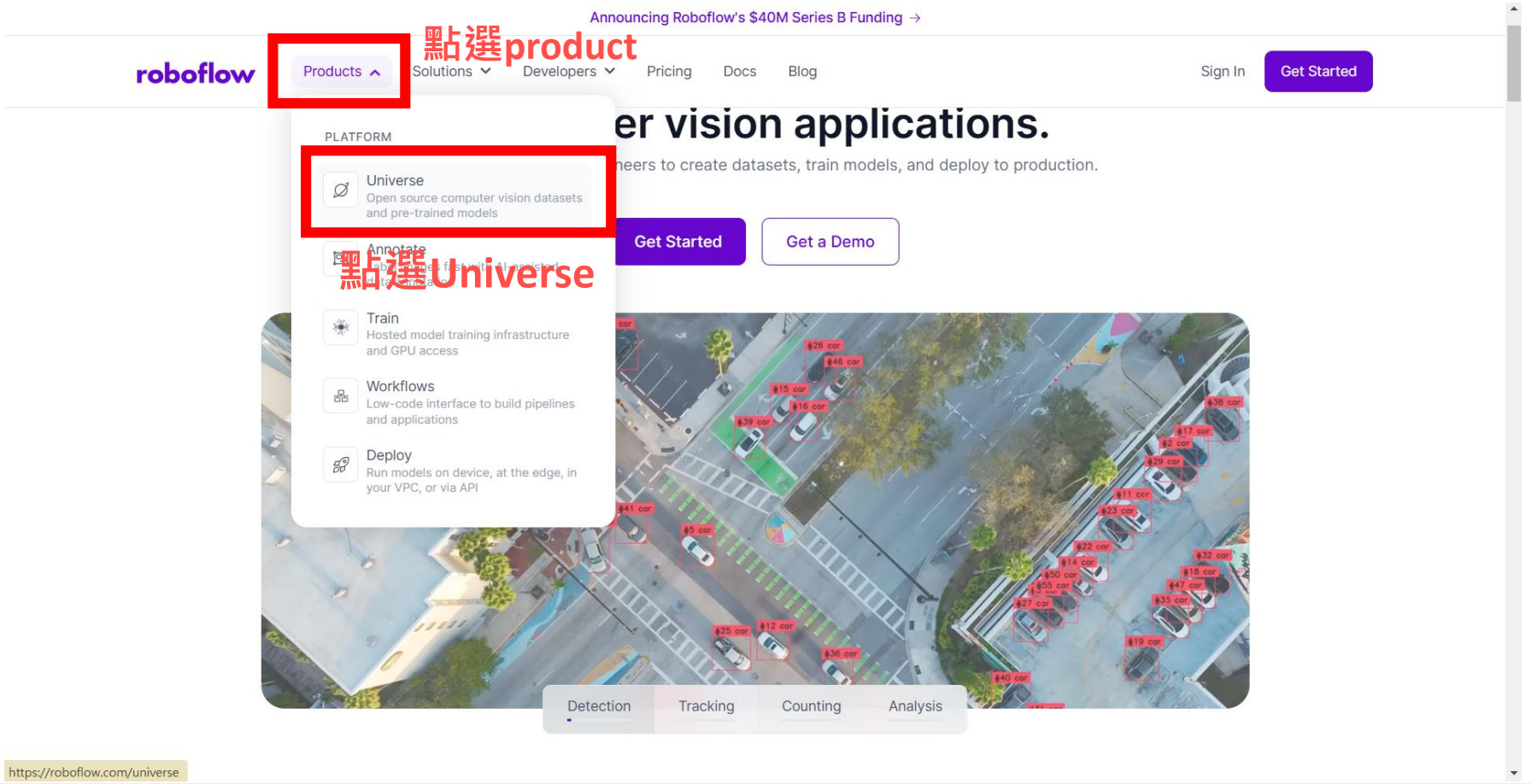
# (二) 資料集來源

- **開源資料集網站 - Roboflow**
  - Roboflow 是一個專門管理影像數據的平台，目標是幫助使用者更有效地管理和處理圖像數據。主要功能包括數據標註、數據清理、數據轉換和數據管理。
  - Roboflow 平台還提供了許多不同用戶所公開的資料集。使用者可以透過 Roboflow 平台輕鬆地瀏覽這些資料集，找到符合自己需求的資料集。


圖片來源：https://roboflow.com

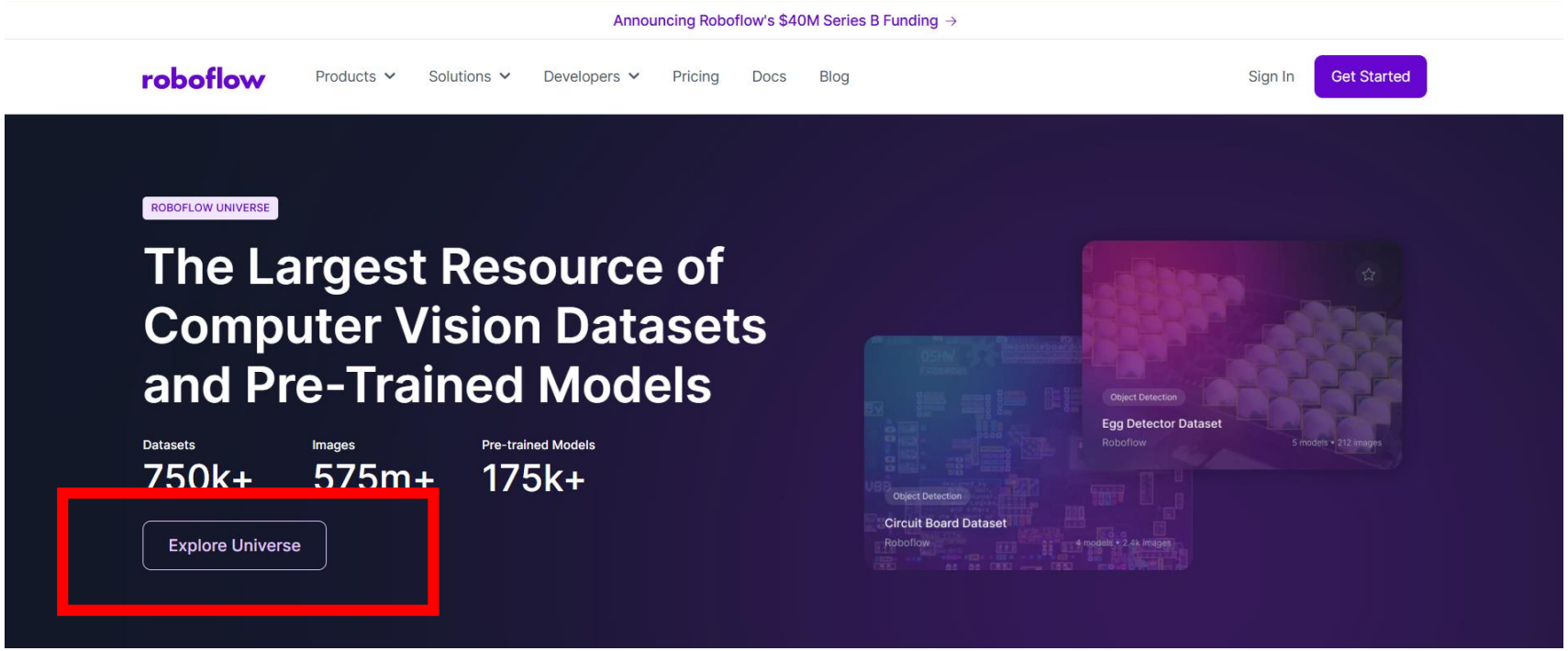
# (三) Roboflow取得資料集步驟

- Step1： 進入Roboflow官網　　網站來源：https://roboflow.com



# (三) Roboflow取得資料集步驟

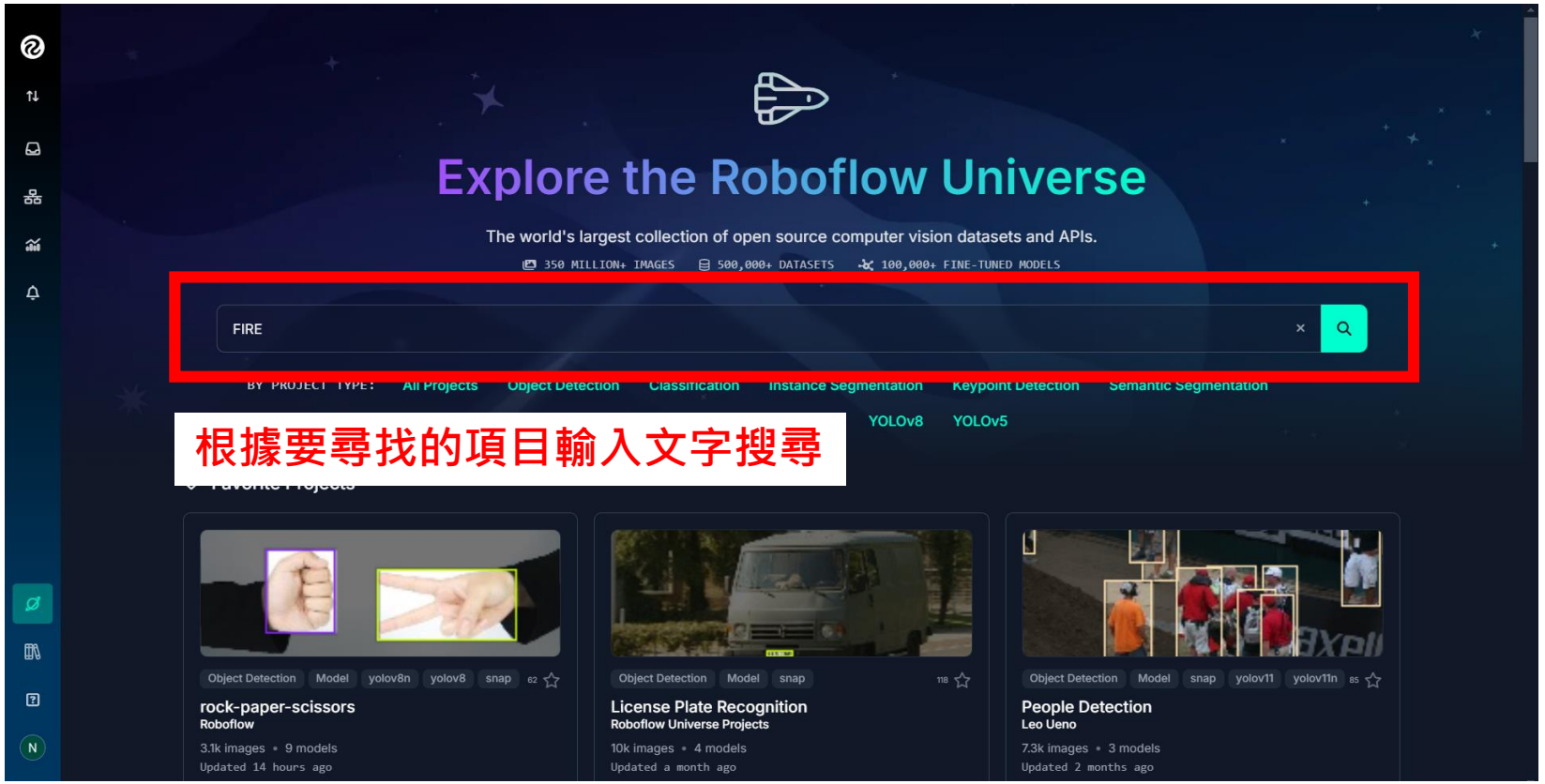
- Step2：點選進入開源資料庫　　網站來源：https://roboflow.com




點選Explore Universe即可開始瀏覽open source的datasets

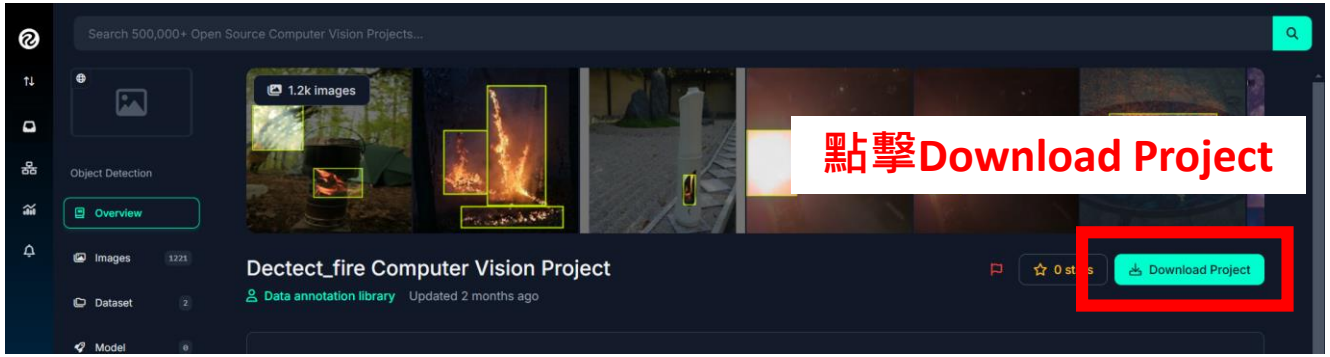
# (三) Roboflow取得資料集步驟

- Step3： 根據需求搜尋需要的DATASET(本案例關鍵字為fire)
  - 網站來源：https://roboflow.com



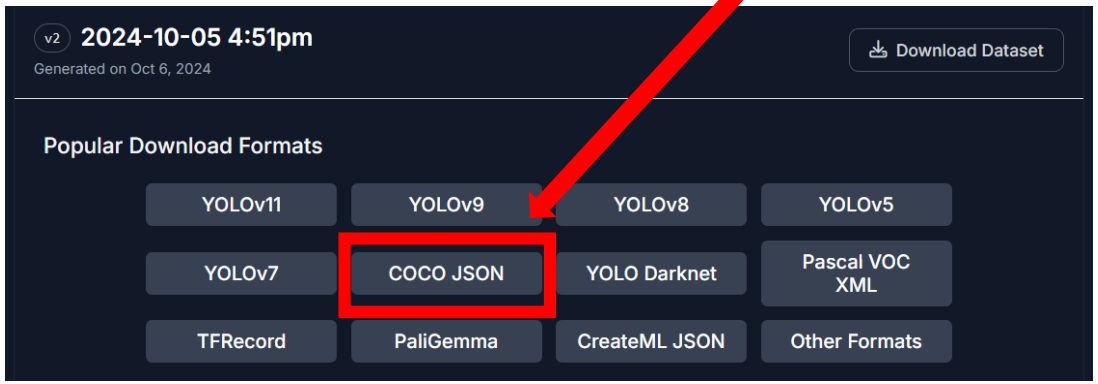
# (三) Roboflow取得資料集步驟

- Step4： 選定項目後可Download Project之dataset




- Step5：選定格式即可下載Dataset

(本案因使用yolox nano訓練故選擇COCO JSON格式)




網站來源：https://roboflow.com

# 三、在PC上使用Anaconda環境進行模型訓練

- **訓練設備：**
  - 硬體設備 : NVDIA GeForce RTX4060
  - CUDA版本 : 11.8
  - Pytorch版本 : 2.0.0
  - 訓練數據： 1210張圖像
  - 訓練參數 : Epoch = 200, Batch size = 64
  - 訓練時長 : 約 1 hr

# (一) 建立Anaconda環境

- 環境建立步驟:
  - 1.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
  - 2.upgrade pip
    - $python -m pip install --upgrade pip setuptools
  - 3.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

# (一) 建立Anaconda環境

- 環境建立步驟:
  - 4.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
  - 5.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
  - 6.Installing the YOLOX
    - $python setup.py develop

# (二) Yolox 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>/
    - annotatios/
      - 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

# (二) 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)
```

根據自己的資料集種類設定class數

```
# --------------  training config ---------------------- #
self.warmup_epochs = 5
self.max_epoch = 200
# -----------------  testing config ------------------ #
```

設定epoch為200

- 開始訓練
  - 指令：

$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

# (二) Yolox nano模型訓練

- 機器學習模型框架轉換
  - 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]]]]"

# (三) 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

# 四、開發版上的推論程式系統流程圖

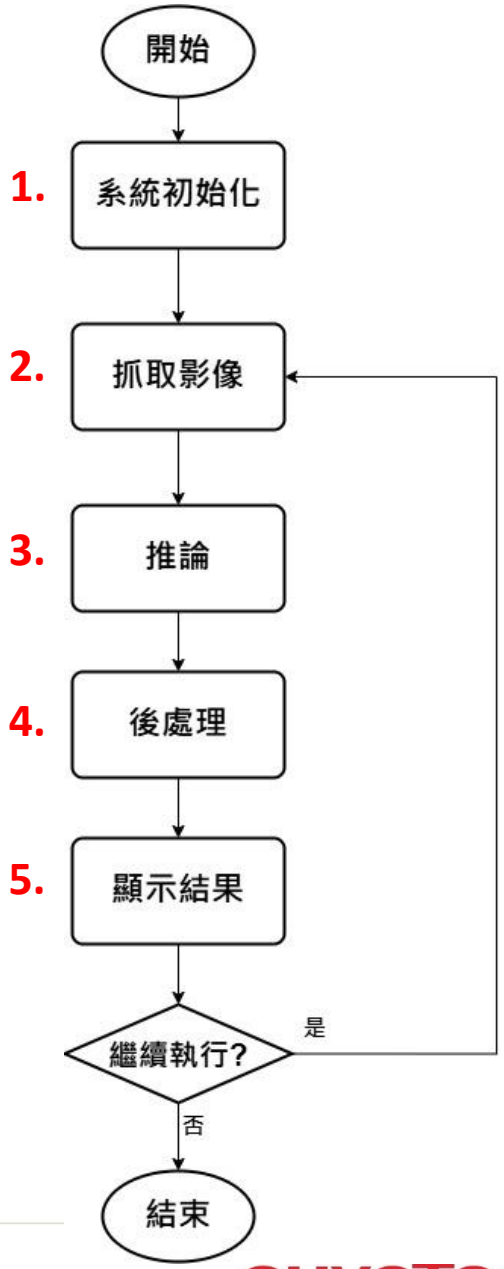
(一)、系統初始化 (System Initialization)

(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)



# (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。

**主程式CLK初始化**

```
static void SYS_Init(void)
{
    /*---------------------------------------------------------------------------------------------------------*/
    /* 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. */
    SystemCoreClockUpdate();
```

**開發版周邊硬體初始化**

```
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);

    info("%s: complete\n", __FUNCTION__);
```

# (二) 捕捉影像

- **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。

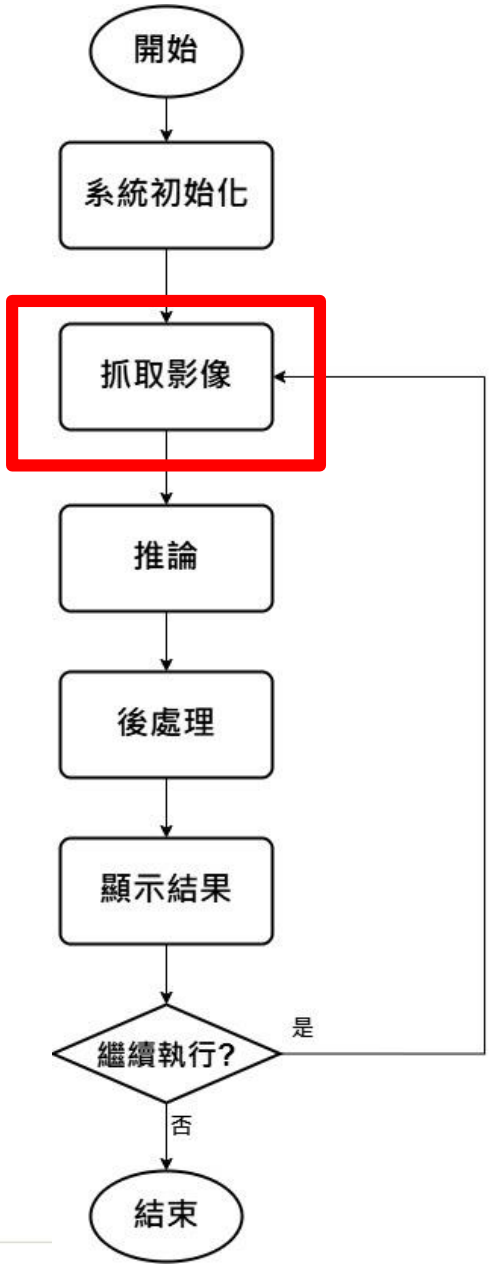
```
//frame buffer managemnet function
static S_FRAMEBUF *get_empty_framebuf()
{
    int i;

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

    return NULL;
}
```

```
emptyFramebuf = get_empty_framebuf();
```

確認是否有資源可供使用，
避免在捕捉影像時覆蓋現有數據。

```
ImageSensor_Capture((uint32_t)(emptyFramebuf->frameImage.data));
    emptyFramebuf = get_empty_framebuf();

    if (emptyFramebuf)
    {
        const uint8_t *pu8ImgSrc = get_img_array(u8ImgIdx);

        if (nullptr == pu8ImgSrc)
        {
            printf_err("Failed to get image index %" PRIu32 " (max: %u)\n", u8ImgIdx,
                       NUMBER_OF_FILES - 1);
            vTaskDelete(nullptr);
            return;
        }

        u8ImgIdx ++;

        if (u8ImgIdx >= NUMBER_OF_FILES)
            u8ImgIdx = 0;
```

實際執行影像捕捉操作，
將影像數據填充到剛剛分配的緩衝區中。



# (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。

函式宣告

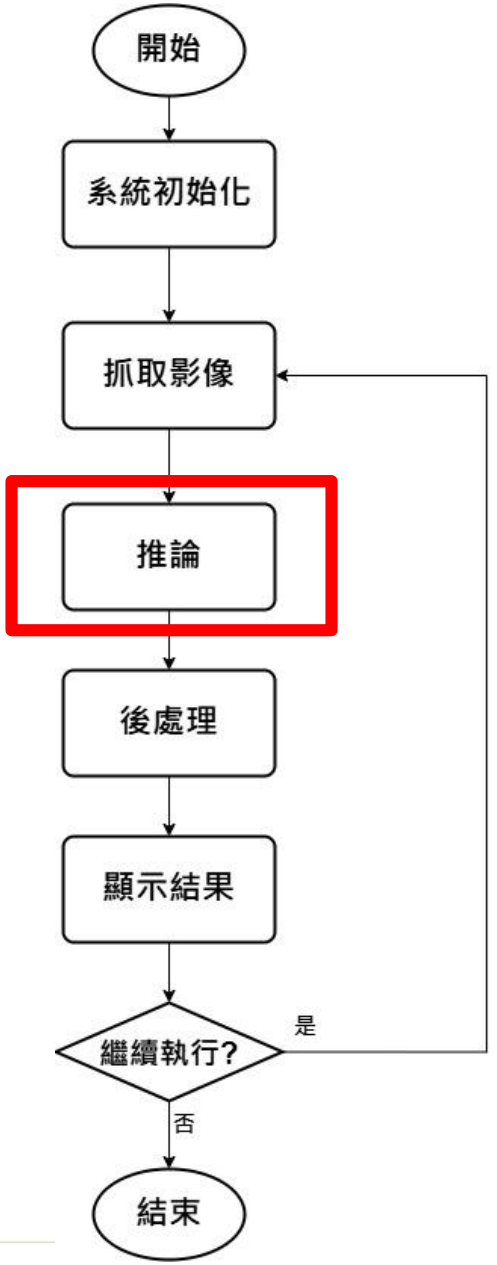
```
static S_FRAMEBUF *get_full_framebuf()
{
    int i;

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

    return NULL;
}
```

函式調用

```
fullFramebuf = get_full_framebuf();
```

火災模型推論程式碼

```
    //trigger inference
    inferenceJob->responseQueue = inferenceResponseQueue;
    inferenceJob->pPostProc = &postProcess;
    inferenceJob->modelCols = inputImgCols;
    inferenceJob->mode1Rows = inputImgRows;
    inferenceJob->srcImgWidth = fullFramebuf->frameImage.w;
    inferenceJob->srcImgHeight = fullFramebuf->frameImage.h;
    inferenceJob->results = &fullFramebuf->results;

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



# (三) 推論

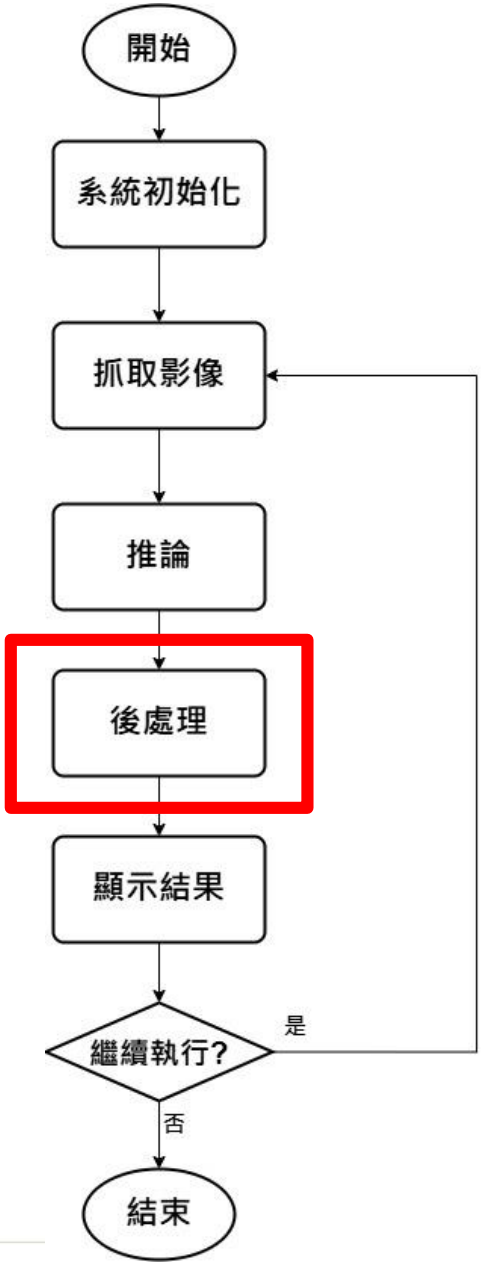
- PresentInferenceResult 函式
  - 火災推論中間結果：
    - 程式會將每一個偵測到的物件標註其類別及在影像中的位置。

```
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;
}
```

# (四) 後處理

## 後處理流程有以下步驟：

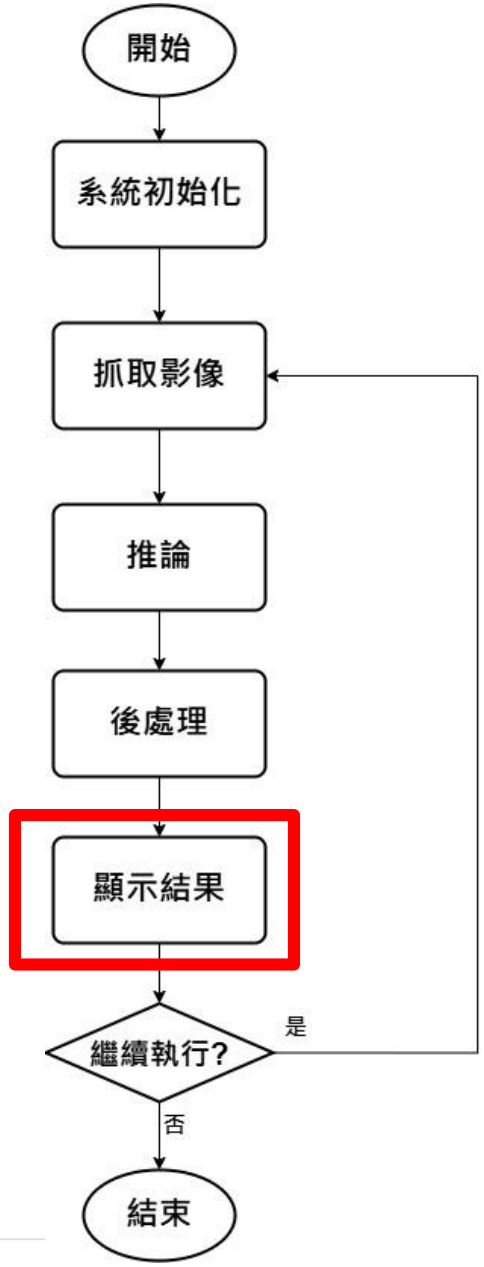
1. **縮放偵測框**：從推理網路輸出轉換回原始影像大小，確保偵測框對應到原始圖像。
2. **提取偵測結果**：從模型的輸出張量中提取出物件的邊界框和類別機率。
3. **執行 NMS**：應用 NMS 來移除重疊的偵測框，只保留最有可能的框。
4. **結果輸出**：將最終的偵測結果儲存到 resultsOut 向量中。



# (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);
    }
}
```



# 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