基於 ExecuTorch 的圖像分類應用

NuMicro® 32 位元系列微控制器範例代碼介紹

文件資訊

應用簡述	本範例程式使用 NuMaker-X-M55M1D 開發板，實作基於 ExecuTorch 的圖像分類推論應用。
BSP 版本	M55M1 Series BSP CMSIS V3.01.003 ExecuTorch v1.0.1
開發平台	NuMaker-X-M55M1D

The information described in this document is the exclusive intellectual property of Nuvoton Technology Corporation and shall not be reproduced without permission from Nuvoton.

Nuvoton is providing this document only for reference purposes of NuMicro microcontroller and microprocessor based system design. Nuvoton assumes no responsibility for errors or omissions.

All data and specifications are subject to change without notice.

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

1. 概述

ExecuTorch 是 Meta® 推出的輕量級 PyTorch 推論執行框架，專門用於邊緣裝置（Edge / Embedded / Mobile）。它的目標是讓 PyTorch 訓練好的模型，能高效、安全、可控地部署到資源受限的裝置上，它的設計目標為：

- 可部署性
  - 不依賴 Python
  - C++ Runtime
  - 可在 Linux/Android/RTOS/MCU 執行
- 可裁剪性
  - 編譯實際使用到的運算子 (operator / kernel)
  - 適合 SRAM / Flash 受限系統
- 可控性
  - 靜態圖
  - 移除原始模型圖的動態行為（control flow / guards）
- 可擴展性
  - 後端/委託 (Backend / Delegate) 機制
  - 可支援 NPU / DSP

NuMicro® M55M1 系列微控制器搭載高達 220 MHz 運行的 Arm® Cortex®-M55 處理器，除了擁有 Cortex-M55 核心外，還整合了 Arm® Ethos™-U55 神經處理單元（NPU）作為輔助處理器，用於加速神經網路操作。另外具備 HyperBus 介面，方便擴展模型運算時對 RAM 的需求。

ExecuTorch 支援 Arm Ethos-U 委託機制，因此適合將神經網路模型使用 ExecuTorch 框架佈署在 M55M1，在推論時經 NPU 運算加速神經網路運行。

1.1 原理

本範例的核心在於利用 ExecuTorch 把模型委託 (delegate) 到支援 Arm Ethos-U (NPU) 的 PTE (PyTorch ExecuTorch format) 推論檔案，使得該模型在推論時能利用 M55M1 上的 Ethos-U 做推論加速。整個工作流程如圖 1-1 會有底下三步驟：

1. 模型導出 (export) : 將動態 PyTorch 程序 (program) 轉化成靜態可分析圖，在這靜態可分析圖裡，不會有控制流運算子 (control flow operator (if/loop))，以及模型形狀/資料型態 (shape/dtype) 都要靜態化。

2. 編譯出 ExecuTorch program ： 將靜態分析圖轉化為支援目標硬體的 ExecuTorch program ，ExecuTorch program 會以 PTE 靜態推論檔案格式儲存。以 Ethos-U 來說，會有幾個部份：

   - 量化(quantization) ： 由於 Ethos-U 是整數處理器，量化是必須的。因此利用了 executorch.backends.arm.quantizer 量化器的功能進行量化。
   - 進行分區 (partitioning) ： 決定模型的哪些部分由 Ethos-U 加速執行，哪些部分可能回退到 CPU 。ARM 的分區策略目標是分區整個圖形以在 Ethos-U 上執行。
   - 進行 Ethos-U 後端編譯：針對 Ethos-U 分區，將 IR (Intermediate Representation) 轉換為 TOSA IR (ARM 的機器學習應用標準) 後，送入 Vela 編譯器，生成 Ethos-U 的命令位元流 (command stream)。
   - 進行降階 (lowering) ： 將中間表示圖形轉換為運行時使用的格式。這包括創建委託 (delegates) (在運行時加載到 Ethos-U 的工作負載)、記憶體規劃以及將整個程序序列化 (serialization) 到一個單一的 PTE (ExecuTorch Program) 檔案中。

3. 在裝置上執行 ExecuTorch program：在裝置上使用 ExecuTorch runtime 載入 ExecuTorch Program，並且依 ExecuTorch Program 的指令進行模型推論。當 runtime 執行到模型圖裡的 Ethos-U backend 運算子，則會調用 Ethos-U driver 進行 Ethos-U 加速器運算。

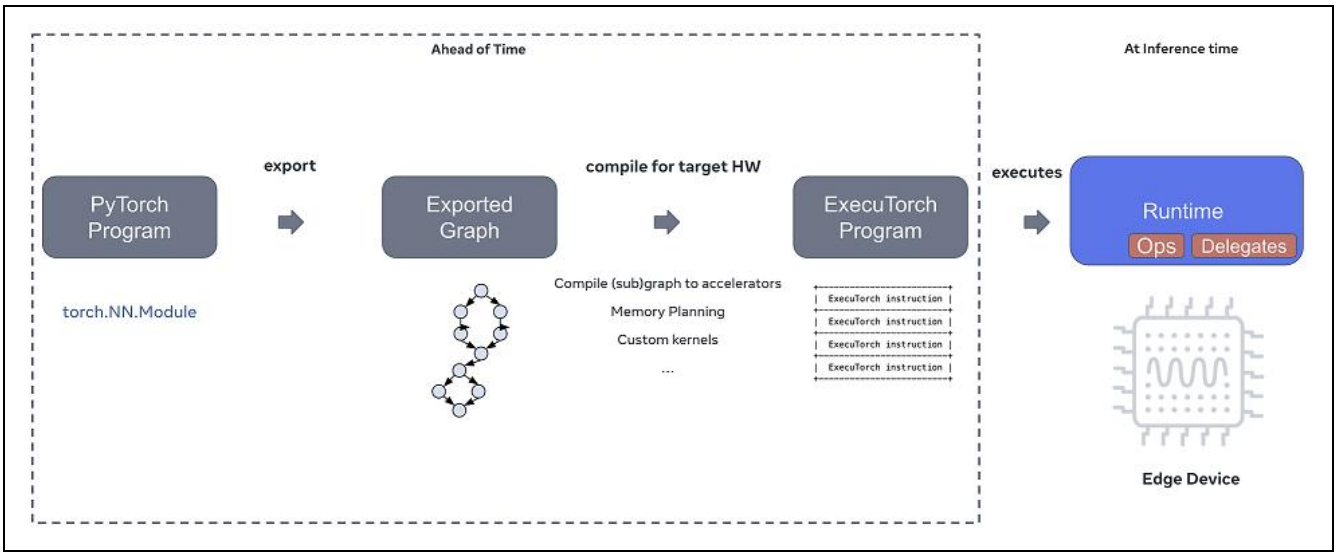

圖 1-1 ExecuTorch 工作流程 (來源: How ExecuTorch Works)

上述步驟 1 和步驟 2 稱之 Ahead-of-Time(AOT) 編譯，是 off-line 流程。因此需在 PC 端架設 ExecuTorch 環境並執行編譯腳本，導出模型 PTE 檔案。相關的環境建置請參考 GitHub 上 ExecuTorch 的 README.md 說明。本文件將專注在編譯腳本和 runtime 環境的說明。

1.2 MobileNetV2 介紹

MobileNetV2 是由 Google® 在 2018 年提出的一種高效、輕量化的卷積神經網絡（CNN）架構。它在 MobileNetV1 的基礎上進行了重大改進，目的在為行動端（如手機、嵌入式設備）提供兼具低延遲與高精度的視覺模型。其核心架構為「具有線性瓶頸的倒置殘差模組 (Inverted Residuals with Linear Bottlenecks)」。

1.2.1 核心架構

MobileNetV2 採用了倒置殘差模組，這是與傳統殘差網路 (如 ResNet) 最大的不同之處。

- 傳統殘差網路 (RestNet) ： 採用「壓縮 → 卷積 → 擴張」的結構 (沙漏型)。先將輸入維度降低以減少計算量，處理後再恢復高維度。
- 倒置殘差模組 ： 採用 「擴張 → 卷積 → 壓縮」 的結構 (橄欖型)。
  - Expansion ： 先用 1x1 卷積將低維通道數擴張（通常擴張 6 倍），進入高維特徵空間。
  - Depthwise ： 在高維度下進行輕量化的深度卷積，這能讓模型捕捉到更豐富的特徵。
  - Projection/Bottleneck ： 將特徵投射回低維通道。

1.2.2 關鍵技術

- 線性瓶頸 (Linear Bottlenecks)

  研究團隊發現，在低維空間（瓶頸層）使用非線性激活函數（如 ReLU）會導致嚴重的資訊丟失。因此在倒置殘差模組的最後一層 1x1 卷積後，不使用 ReLU，直接輸出線性特徵。這確保了模型在壓縮資料時能保留關鍵的資訊。

- ReLU6 激活函數

  MobileNetV2 廣泛使用 ReLU6，這將輸出限制在 [0, 6] 之間。這在移動端使用低精度（如 Float16 或 Quantized INT8）計算時，能防止數值溢位並提高模型的強健性。

- 深度可分離卷積 (Depthwise Separable Convolution)

  繼承自 V1，將標準卷積拆分為 Depthwise（逐通道處理）與 Pointwise（逐像素融合）。這讓計算量降低了約 8 至 9 倍，而精度僅有些微下降。

1.2.3 網路架構與效能

MobileNetV2 總共由一個初始的全卷積層和 19 個殘差瓶頸層組成，其特性如表 1-1。

特性	描述
輸入尺寸	224 x 224 x 3
輸出尺寸	1000 (ImageNet 類別數)
參數數量	約 3.4 M（比 V1 更少）
ImageNet 準確率	Top-1 約 72.0%（優於 V1 的 70.6%）
計算量 (MACs)	約 300 M

表 1-1 MobileNetV2 特性

1.3 執行結果

1.3.1 產生模型 PTE 檔案

這一階段是在 PC 上執行模型編譯腳本，將 MobileNetV2 模型轉成 ExecuTorch runtime 可執行的 PTE 檔案。環境是採用 Ubuntu 24.04 ，並在 Ubuntu 上安裝 ExecuTorch 及其相依 python 套件。

執行 MobileNetV2 編譯腳本時，會自動下載預訓練 MobileNetV2 模型，因此請保持網路的連線。執行結束後會產生「mv2_arm_delegate_ethos-u55-256.pte」檔案，如圖 1-2。使用 Netron 可視化工具檢視這檔案，可看到轉化後的模型圖，如圖 1-3。

轉化後的模型圖會有一個 EthosUBackend 運算子和兩個需由 CPU 處理的運算子 (quantize_per_tensor 和 dequantize_per_tensor) 。在程式碼的部份需特別將 CPU 處理的運算子註冊到 kernel 表。

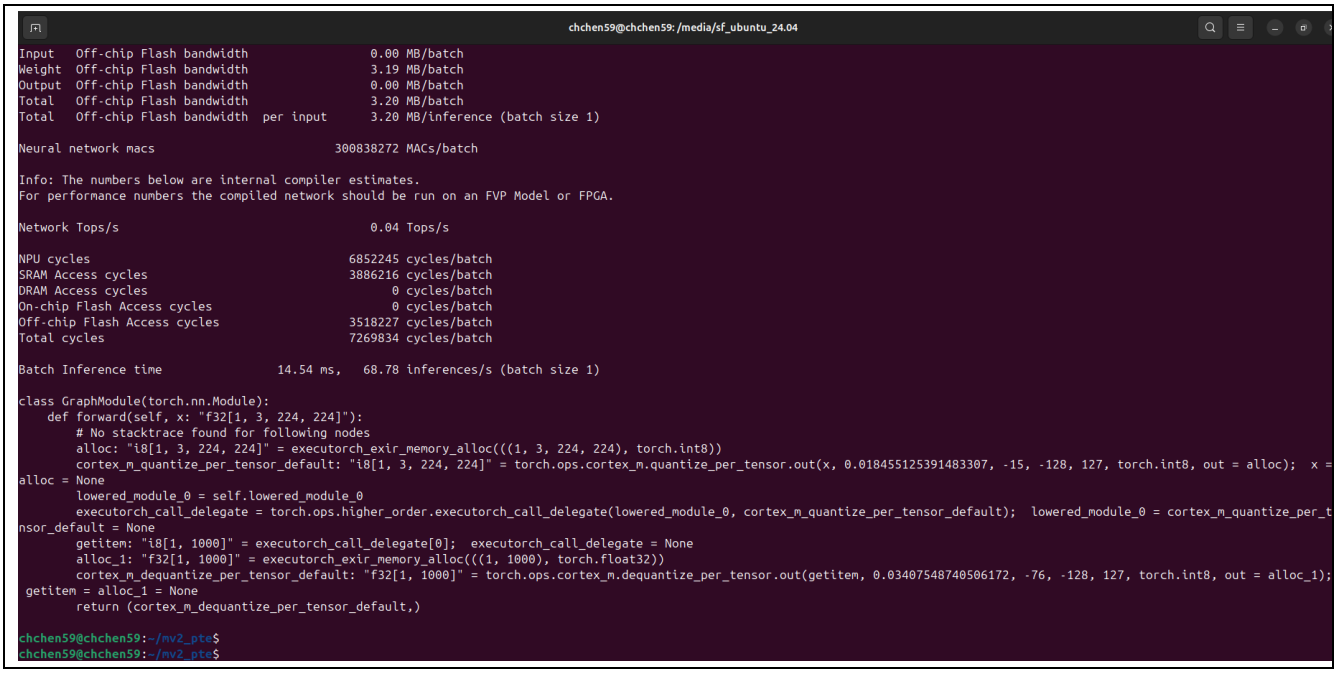

圖 1-2 MobileNetV2 編譯結果

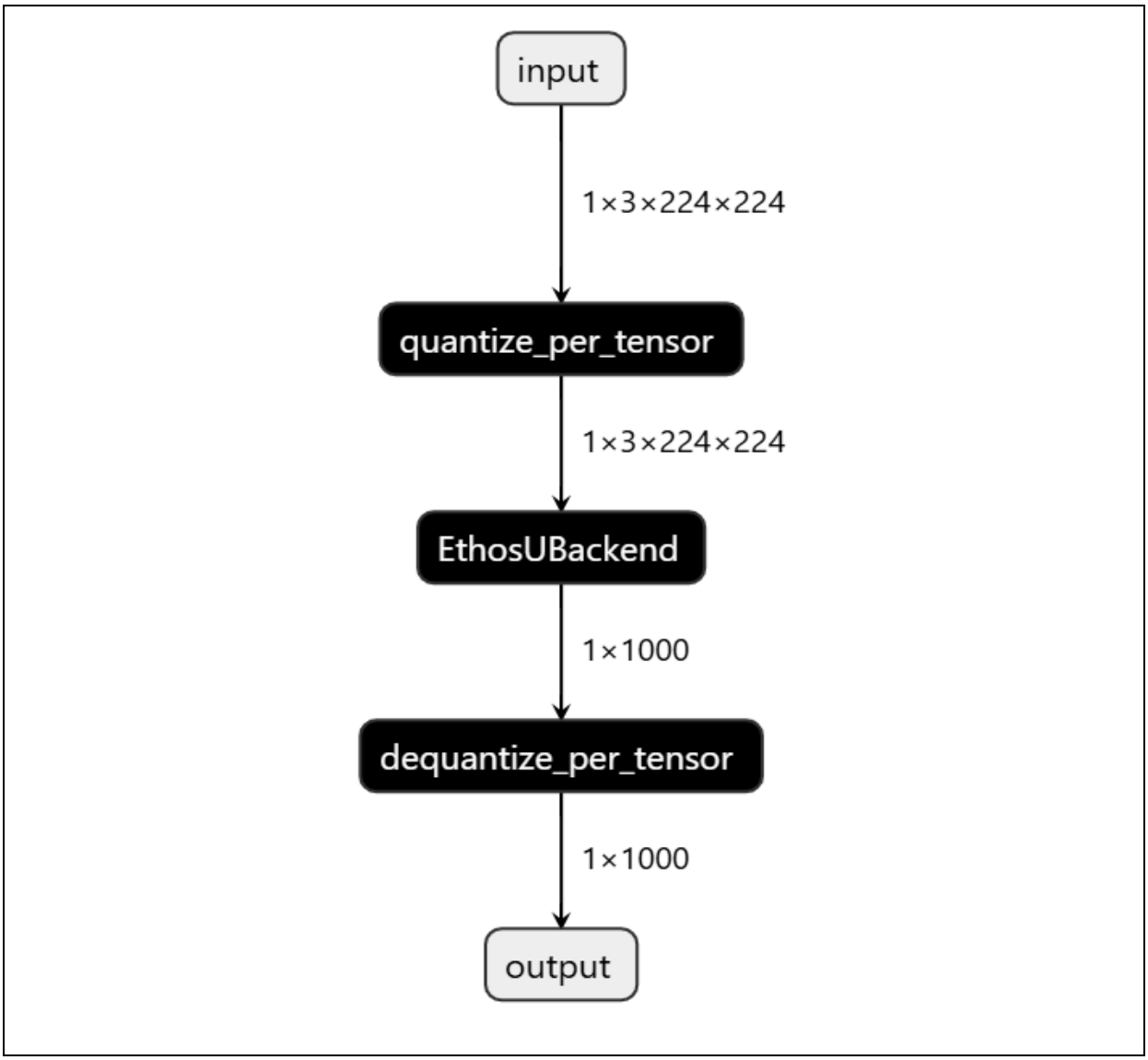

圖 1-3 模型圖

1.3.2 執行

由於 MobileNetV2 PTE 檔案超過內建 Flash 容量，因此需將檔案放入 SD card ，並將檔案重新命名為「mv2.pte」，後續由 runtime 主程式從 SD card 載入到 HyperRAM 執行。

範例程式執行編譯和燒錄後，運行結果，如圖 1-4。

圖 1-4 範例程式執行結果

2. 代碼介紹

2.1 MobileNetV2 編譯腳本 (mv2_pte.py)

MobileNetV2 編譯腳本程式碼在 Script/mv2_pte.py。這份 mv2_pte.py 腳本主要用於將 PyTorch 的 MobileNetV2 模型導出、量化，並針對 Arm Ethos-U55 與 Cortex-M 後端進行優化與轉換，最終生成可在嵌入式設備上運行的 ExecuTorch 格式模型。以下為完整說明：

1. 主要功能：

- 下載並載入 MobileNetV2 預訓練模型。
- 準備範例輸入（可用隨機張量或真實圖片）。
- 將模型導出為 PyTorch Exported Program。
- 進行後訓練量化（PTQ），並再次導出量化後模型。
- 針對 Arm Ethos-U55 進行分割、編譯與優化。
- 進行 Cortex-M 後端相關轉換。
- 轉換為 ExecuTorch 格式並儲存為「.pte」檔案。

2. 主要步驟說明：

1) 載入模型與準備輸入：

```
# Get example input
def get_example_inputs(use_real_input):
    tensor_size = (1, 3, 224, 224)
    input_batch = (torch.randn(tensor_size),)
    if use_real_input == True:
        print("Loaded real input image dog.jpg")
        import urllib

        url, filename = (
            "https://github.com/pytorch/hub/raw/master/images/dog.jpg",
            "dog.jpg",
        )
        try:
            urllib.URLopener().retrieve(url, filename)
        except:
            urllib.request.urlretrieve(url, filename)
        from PIL import Image
        from torchvision import transforms
```

```
        input_image = Image.open(filename)
        preprocess = transforms.Compose(
            [
                transforms.Resize(256),
                transforms.CenterCrop(224),
                transforms.ToTensor(),
                transforms.Normalize(
                    mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
                ),
            ]
        )
        input_tensor = preprocess(input_image)
        input_batch = input_tensor.unsqueeze(0)
        input_batch = (input_batch,)
    return input_batch
```

- 使用 torchvision 下載 MobileNetV2 預訓練權重。
- get_example_inputs 函數可選擇載入隨機張量或下載真實圖片（dog.jpg）並進行預處理。

2) 導出模型：

```
# Get mobilenetv2 form torchvision
original_model = mobilenet_v2(weights=MobileNet_V2_Weights.DEFAULT)
print(type(original_model))

example_inputs = get_example_inputs(True)

model = original_model.eval()

exported_program = torch.export.export(model, example_inputs, strict=False)
graph_module = exported_program.module(check_guards=False)
#_ = graph_module.print_readable()

# We only support Python3.10 and above, so use a later pickle protocol
torch.export.save(
    exported_program,
    f"mv2_exported_program.pt2",
    pickle_protocol=5,
)
```

- 將模型設為 eval 模式。
- 使用 torch.export.export 將模型與範例輸入導出為 Exported Program。
- 使用 torch.export.save 儲存導出的模型，指定 pickle protocol 5（需 Python 3.10+）。

3) 量化流程

```
from executorch.backends.arm.ethosu import EthosUCompileSpec
from executorch.backends.arm.util._factory import create_partitioner, create_quantizer

from executorch.backends.arm.quantizer import (
    EthosUQuantizer,
    get_symmetric_quantization_config,
    get_symmetric_a16w8_quantization_config,
)
from torchao.quantization.pt2e.quantize_pt2e import convert_pt2e, prepare_pt2e

# Create a compilation spec describing the target for configuring the quantizer
# Some args are used by the Arm Vela graph compiler later in the example. Refer to Arm
Vela documentation for an
# explanation of its flags: https://gitlab.arm.com/artificial-intelligence/ethos-u/ethos-
u-vela/-/blob/main/OPTIONS.md
compile_spec = EthosUCompileSpec(
            target="ethos-u55-256",
            system_config="Ethos_U55_High_End_Embedded",
            memory_mode="Shared_Sram",
            extra_flags=["--verbose-operators", "--verbose-cycle-estimate"],
            config_ini="Arm/vela.ini"
        )

# Create and configure quantizer to use a symmetric quantization config globally on all
nodes
print(compile_spec)
quantizer = create_quantizer(compile_spec)

operator_config = get_symmetric_quantization_config(is_per_channel=True)
quantizer.set_global(operator_config)

# Post training quantization
quantized_graph_module = prepare_pt2e(graph_module, quantizer)
quantized_graph_module(*example_inputs) # Calibrate the graph module with the example
input
quantized_graph_module = convert_pt2e(quantized_graph_module)
#print(quantized_graph_module)

# Create a new exported program using the quantized_graph_module
quantized_exported_program = torch.export.export(quantized_graph_module, example_inputs)

torch.export.save(
    quantized_exported_program,
    f"mv2_quantize_exported_program.pt2",
    pickle_protocol=5,
)
```

- 匯入 Ethos-U 相關 quantizer 與 config。
- 建立 EthosUCompileSpec，描述目標硬體與編譯選項。
- 建立 quantizer 並設定全域對稱量化參數。

- 使用 prepare_pt2e 進行量化前準備，並用範例輸入進行校準。
- 使用 convert_pt2e 完成量化，並再次導出量化後的 Exported Program。

4) Ethos-U 後端分割與優化：

```
from executorch.backends.arm.ethosu import EthosUPartitioner
from executorch.exir import (
    EdgeCompileConfig,
    ExecutorchBackendConfig,
    to_edge_transform_and_lower,
)
from executorch.extension.export_util.utils import save_pte_program

# Create partitioner from compile spec
partitioner = EthosUPartitioner(compile_spec)

# Lower the exported program to the Ethos-U backend
edge_program_manager = to_edge_transform_and_lower(
            quantized_exported_program,
            partitioner=[partitioner],
            compile_config=EdgeCompileConfig(
                _check_ir_validity=False,
            ),
        )
```

- 建立 EthosUPartitioner，根據 compile spec 進行分割。
- 使用 to_edge_transform_and_lower 進行分割、優化與 IR 降階。

5) Cortex-M 後端轉換：

```
# Transform so we can use ops from the Cortex M backend
edge_program_manager = transform_for_cortex_m_backend(edge_program_manager)
```

- 執行 transform_for_cortex_m_backend，將部分運算轉換為 Cortex-M 支援的運算。

6) 轉換為 ExecuTorch 格式並儲存：

```
# Convert edge program to executorch
executorch_program_manager = edge_program_manager.to_executorch(
            config=ExecutorchBackendConfig(extract_delegate_segments=False)
        )

_ = executorch_program_manager.exported_program().graph_module.print_readable()

# Save pte file
save_pte_program(executorch_program_manager, "mv2_arm_delegate_ethos-u55-256.pte")
```

- 使用 to_executorch 轉換為 ExecuTorch 格式。
- 使用 save_pte_program 儲存為 *.pte* 檔案，供嵌入式設備部署。

2.2 Runtime 主程式 (main.cpp)

2.2.1 主要步驟說明：

1. 定義區段：

```
#define MODEL_AT_HYPERRAM_ADDR (0x82300000)
#define __PROFILE__

#define MV2_INPUT_TENSOR_INDEX      0
#define MV2_OUTPUT_TENSOR_INDEX     0

//select image pattern: 0: cat, 1: dog, 2: tiger
#define IMAGE_PATTERN_INDEX         0
```

- MODEL_AT_HYPERRAM_ADDR: 模型在 HyperRAM 中的存放位址。
- __PROFILE__: 啟用效能分析功能。
- MV2_*_TENSOR_INDEX: MobileNetV2 模型的輸入/輸出張量索引。
- IMAGE_PATTERN_INDEX: 選擇測試影像（0: cat, 1: dog, 2: tiger）。

2. 記憶體池配置：

```
// To speed up inference time. Using temp_allocation_pool for EthosU's scratch buffer
__attribute__((section(".bss.sram.data"), aligned(32)))
uint8_t s_au8TempAllocationPool[1500 * 1024U];

__attribute__((section(".bss.sram.data"), aligned(32)))
uint8_t s_au8MethodAllocationPool[1400 * 1024U];
```

- 預先分配兩個大型記憶體池，放置在 SRAM/HyperRAM 中。
- s_au8TempAllocationPool：用於 Ethos-U NPU 的暫存緩衝區（1.5 MB）。
- s_au8MethodAllocationPool：用於方法執行的記憶體池（1.4 MB）。

3. 日誌輸出函式：

```
void et_pal_emit_log_message(
    __ET_UNUSED et_timestamp_t timestamp,
    et_pal_log_level_t level,
    const char* filename,
```

```
    __ET_UNUSED const char* function,
    size_t line,
    const char* message,
    __ET_UNUSED size_t length) {
    fprintf(stderr, "%c executorch:%s:%zu] %s\n", level, filename, line, message);
}
```

- ExecuTorch 平台層的日誌輸出實作。
- 將 ExecuTorch 的內部日誌導向到標準錯誤輸出（stderr）。
- 格式化顯示日誌等級、檔案名、行號和訊息內容。

4. 輔助函式群：

1) 模型輸入張量資訊顯示：

```
// Print model input tensor information
static void PrintModelInputTensorInfo(Method& objMethod)
{
    MethodMeta objMethodMeta = objMethod.method_meta();
    size_t nNumInputs = objMethodMeta.num_inputs();
    ET_LOG(Info, "Total inputs %d ", nNumInputs);
    for (size_t i = 0; i < nNumInputs; i++) {
        ET_LOG(Info, "===== Inputs number %d =====", i);
        Result<TensorInfo> objTensorMeta = objMethodMeta.input_tensor_meta(i);
        ET_LOG(Info, "Input tensor buffer %d bytes", objTensorMeta->nbytes());
        ET_LOG(Info, "Input tensor data type %s ",
torch::executor::toString(objTensorMeta->scalar_type()));

        size_t nNumDims = objTensorMeta.get().sizes().size();
        ET_LOG(Info, "Input dimensions %d ", nNumDims);
        Tensor::DimOrderType *pDimOrder = const_cast<TensorImpl::DimOrderType
*>(objTensorMeta->dim_order().data());
        Tensor::SizesType *pDimSize = const_cast<TensorImpl::SizesType*>(objTensorMeta-
>sizes().data());

        for(int32_t j = 0; j < nNumDims; j ++)
        {
            ET_LOG(Info, "Input dim_order %d and size %d", pDimOrder[j], pDimSize[j]);
        }
    }
}
```

分析並顯示模型所有輸入張量的詳細資訊。包括：輸入數量、每個張量的記憶體大小、資料型態、維度順序和大小。

2) 模型載入到 HyperRAM：

```
// Load model from SD card to HyperRAM
static int32_t PrepareModelToHyperRAM(void)
{
```

```
#define MODEL_FILE "0:\\mv2.pte"
#define EACH_READ_SIZE 512

    TCHAR achSDPath[] = { '0', ':', 0 };    /* SD drive started from 0 */
    f_chdrive(achSDPath);          /* set default path */

    int32_t i32FileSize;
    int32_t i32FileReadIndex = 0;
    int32_t i32Read;

    if(!ModelFileReader_Initialize(MODEL_FILE))
    {
        printf("Unable open model %s\n", MODEL_FILE);
        return -1;
    }

    i32FileSize = ModelFileReader_FileSize();
    printf("Model file size %i \n", i32FileSize);

    while(i32FileReadIndex < i32FileSize)
    {
        i32Read = ModelFileReader_ReadData((BYTE *)(MODEL_AT_HYPERRAM_ADDR +
i32FileReadIndex), EACH_READ_SIZE);
        if(i32Read < 0)
            break;
        i32FileReadIndex += i32Read;
    }

    if(i32FileReadIndex < i32FileSize)
    {
        printf("Read Model file size is not enough\n");
        return -2;
    }

#if 0
    /* verify */
    i32FileReadIndex = 0;
    ModelFileReader_Rewind();
    BYTE au8TempBuf[EACH_READ_SIZE];

    while(i32FileReadIndex < i32FileSize)
    {
        i32Read = ModelFileReader_ReadData((BYTE *)au8TempBuf, EACH_READ_SIZE);
        if(i32Read < 0)
            break;

        if(std::memcmp(au8TempBuf, (void *)(MODEL_AT_HYPERRAM_ADDR + i32FileReadIndex),
i32Read)!= 0)
        {
            printf("verify the model file content is incorrect at %i \n",
i32FileReadIndex);
            return -3;
        }
        i32FileReadIndex += i32Read;
    }
```

```
#endif
    ModelFileReader_Finish();

    return i32FileSize;
}
```

- 從 SD 卡讀取「mv2.pte」模型檔案。
- 以 512 位元組為單位逐步載入到 HyperRAM 指定位址。
- 包含檔案完整性檢查機制（#if 0 註解掉的驗證代碼）。
- 返回實際讀取的檔案大小，失敗則返回負值。

3) 自訂記憶體分配器：

```
// Setup our own allocator that can show some extra stuff like used and free
// memory info
class ArmMemoryAllocator : public executorch::runtime::MemoryAllocator {
public:
    ArmMemoryAllocator(uint32_t u32Size, uint8_t* pu8BaseAddress)
        : MemoryAllocator(u32Size, pu8BaseAddress), used_(0) {}

    void* allocate(size_t nSize, size_t nAlignment = kDefaultAlignment) override {
        void* pvRet = executorch::runtime::MemoryAllocator::allocate(nSize, nAlignment);
        if (pvRet != nullptr) {
            // Align with the same code as in MemoryAllocator::allocate() to keep
            // used_ "in sync" As alignment is expected to be power of 2 (checked by
            // MemoryAllocator::allocate()) we can check it the lower bits
            // (same as alignment - 1) is zero or not.
            if ((nSize & (nAlignment - 1)) == 0) {
                // Already aligned.
                used_ += nSize;
            } else {
                used_ = (used_ | (nAlignment - 1)) + 1 + nSize;
            }
        }
        return pvRet;
    }

    // Returns the used size of the allocator's memory buffer.
    size_t used_size() const {
        return used_;
    }

    // Returns the free size of the allocator's memory buffer.
    size_t free_size() const {
        return executorch::runtime::MemoryAllocator::size() - used_;
    }

private:
    size_t used_;
};
```

- 繼承 ExecuTorch 的基礎記憶體分配器。
- 新增記憶體使用量追蹤功能。
- 提供 used_size() 和 free_size() 方法來監控記憶體使用狀況。
- 考慮記憶體對齊的實際使用量計算。

4) 影像正規化處理：

```
// ImageNet mean/std
static const float s_afImagenetMean[3] = {0.485f, 0.456f, 0.406f};
static const float s_afImagenetStd[3]  = {0.229f, 0.224f, 0.225f};

// Normalize RGB888 image data to input tensor
static void NormalizeImageToInputTensor(
    Tensor *pInputTensor,
    uint8_t *pu8ImageData
)
{
    Tensor::SizesType *pnDimSize = const_cast<TensorImpl::SizesType*>(pInputTensor-
>sizes().data());

    int32_t C = pnDimSize[1];          //channel
    int32_t H = pnDimSize[2];          //height
    int32_t W = pnDimSize[3];          //width

    auto *pfTensorData = pInputTensor->mutable_data_ptr<float>();
    auto *pi8TensorData = pInputTensor->mutable_data_ptr<int8_t>();
    auto *pu8TensorData = pInputTensor->mutable_data_ptr<uint8_t>();

    for(int32_t c = 0; c < C; c ++)
    {
        for(int32_t h = 0; h < H; h++ )
        {
            for(int32_t w = 0; w < W; w++ )
            {
                uint8_t u8Color = pu8ImageData[(((h * W) + w) * 3) + c];

                if(pInputTensor->scalar_type() ==  ScalarType::Float)
                {
                    // Normalize to [0,1], then standardize
                    float fColor = u8Color / 255.0f;
                    // Standardize
                    *pfTensorData = (fColor - s_afImagenetMean[c]) / s_afImagenetStd[c];
                    pfTensorData ++;
                }
                else if(pInputTensor->scalar_type() ==  ScalarType::Char)
                {
                    int16_t ui16color = u8Color;
                    *pi8TensorData = (int8_t)(ui16color - 128);
                    pi8TensorData++;
                }
                else if(pInputTensor->scalar_type() ==  ScalarType::Byte)
                {
```

```
                    *pu8TensorData = u8Color;
                    pu8TensorData++;
                }
            }
        }
    }
}
```

- 將 RGB888 格式的影像資料轉換為模型輸入張量。
- 支援三種資料型態：Float (標準化)、Char (減 128)、Byte (直接複製)。
- Float 型態使用 ImageNet 的均值和標準差進行標準化。
- 按照 channel-first 格式 (C, H, W) 排列資料。

5) 準備輸入張量：

```
// Prepare input tensors from image data
static Result<executorch::extension::BufferCleanup> PrepareInputTensorsFromImage(
    Method& objMethod,
    executorch::runtime::MemoryAllocator& objAllocator,
    std::vector<std::pair<char*, size_t>>& vecInputBuffers)
{
    // Get method meta to setup input tensors
    MethodMeta method_meta = objMethod.method_meta();
    size_t nNumInputs = method_meta.num_inputs();
    size_t nNumAllocated = 0;
    ET_CHECK_OR_RETURN_ERROR(
        nNumInputs == 1,
        InvalidArgument,
        "MobileNetV2 input should be 1");

    void** vppInputs = static_cast<void**>(objAllocator.allocate(nNumInputs *
sizeof(void*)));

    Result<TensorInfo> objTensorMeta = method_meta.input_tensor_meta(0);
    ET_CHECK_OK_OR_RETURN_ERROR(objTensorMeta.error());

    // Input is a tensor. Allocate a buffer for it.
    void* vpData = objAllocator.allocate(objTensorMeta->nbytes());
    ET_CHECK_OR_RETURN_ERROR(
        vpData != nullptr,
        MemoryAllocationFailed,
        "Could not allocate memory for input buffers.");
    vppInputs[nNumAllocated++] = vpData;

    // Create the input tensor
    TensorImpl objTensorImpl = TensorImpl(
                                    objTensorMeta.get().scalar_type(),
                                    objTensorMeta.get().sizes().size(),

const_cast<TensorImpl::SizesType*>(objTensorMeta.get().sizes().data()),
                                    vpData,
```

```
                                    const_cast<TensorImpl::DimOrderType*>(
                                        objTensorMeta.get().dim_order().data()));
    Tensor objTensor(&objTensorImpl);

    Error err = Error::Ok;
    if (vecInputBuffers.size() > 0) {
        auto [image_data_ptr, image_data_size] = vecInputBuffers.at(0);
        //Normalize RGB888 image data to float
        NormalizeImageToInputTensor(&objTensor, (uint8_t *)image_data_ptr);
    }
    else
    {
        // If input_buffers.size <= 0, we don't have any input, fill t with 1's.
        for (size_t j = 0; j < objTensor.numel(); j++) {
            switch (objTensor.scalar_type()) {
            case ScalarType::Float:
                objTensor.mutable_data_ptr<float>()[j] = 1.;
                break;
            case ScalarType::Char:
                objTensor.mutable_data_ptr<int8_t>()[j] = 1;
                break;
            case ScalarType::Byte:
                objTensor.mutable_data_ptr<uint8_t>()[j] = 1;
                break;
            default:
                ET_LOG(
                    Error, "Failed to handle input data type");
            }
            break;
        }
    }

    // Set the input tensor to the method
    err = objMethod.set_input(objTensor, 0);

    if (err != Error::Ok) {
        ET_LOG(
            Error, "Failed to prepare input %zu: 0x%" PRIx32, 0, (uint32_t)err);
        // The BufferCleanup will free the inputs when it goes out of scope.
        executorch::extension::BufferCleanup cleanup({vppInputs, nNumAllocated});
        return err;
    }

    return  executorch::extension::BufferCleanup({vppInputs, nNumAllocated});
}
```

- 為 MobileNetV2 模型準備輸入張量。
- 從 method 的 meta 資料獲取輸入張量規格。
- 分配適當的記憶體並建立張量物件。
- 呼叫 NormalizeImageToInputTensor() 填入影像資料。

6) Softmax 實作：

```
// Softmax for float vector
static void SoftmaxF32(std::vector<float>& vecData)
{
    /* Fix for numerical stability and apply exp. */
    auto nStart = vecData.begin();
    auto nEnd = vecData.end();

    float maxValue = *std::max_element(nStart, nEnd);
    for (auto it = nStart; it != nEnd; ++it) {
        *it = std::exp((*it) - maxValue);
    }

    float fSumExp = std::accumulate(nStart, nEnd, 0.0f);

    for (auto it = nStart; it != nEnd; ++it) {
        *it = (*it)/fSumExp;
    }
}
```

- 對浮點數向量執行 Softmax 運算。
- 先減去最大值防止數值溢位（數值穩定性）。
- 計算指數和並且正規化，得到機率分佈。
- 確保所有輸出值總和為 1。

5. 主程式 main() 說明：

主程式流程如圖 2-1。

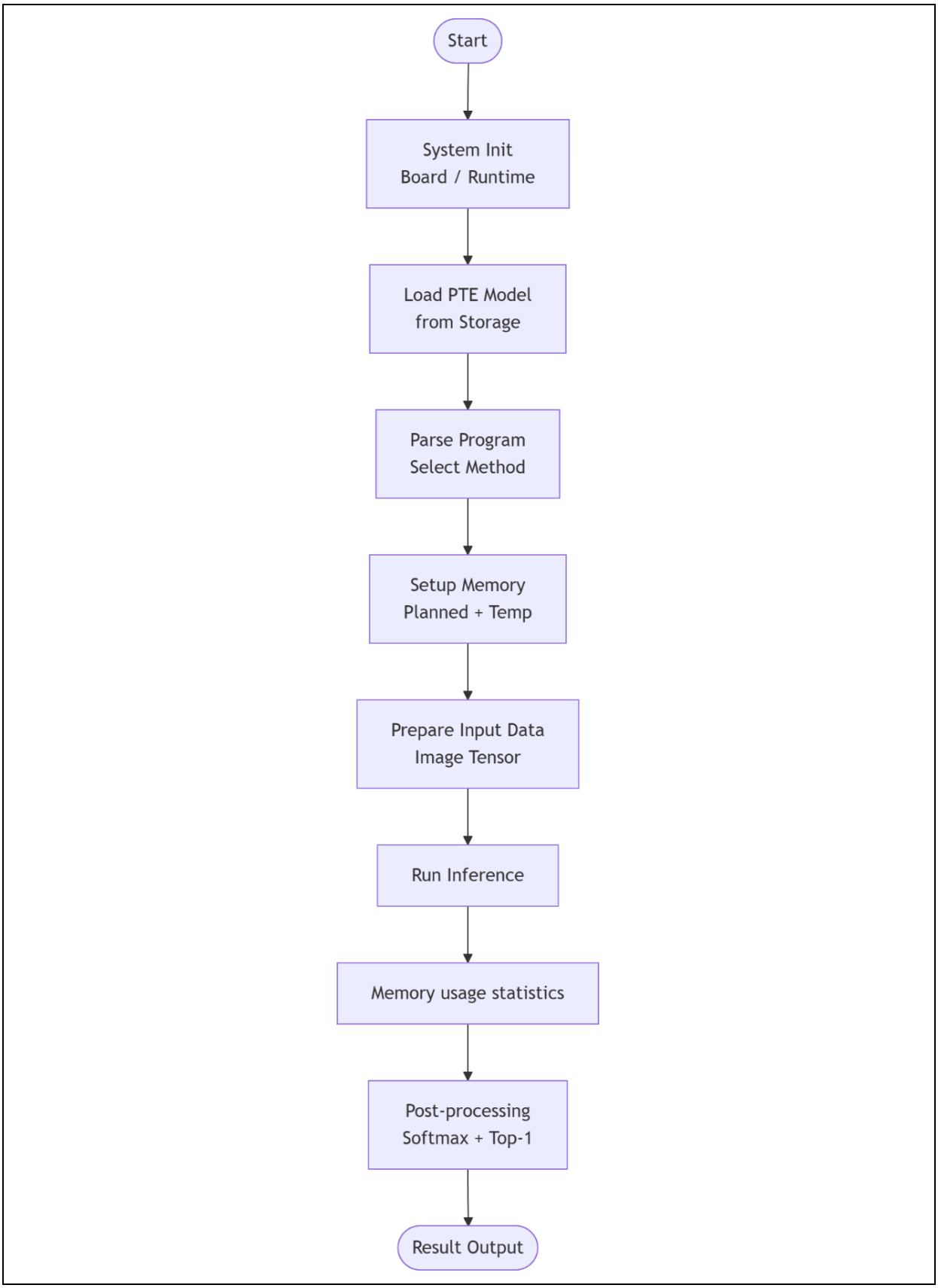
Start
System Init
Board / Runtime
Load PTE Model
from Storage
Parse Program
Select Method
Setup Memory
Planned + Temp
Prepare Input Data
Image Tensor
Run Inference
Memory usage statistics
Post-processing
Softmax + Top-1
Result Output

圖 2-1 主程式流程

1) 初始化階段：

```
    /* Initialise the hardware resource (UART, NPU, HyperRAM) */
    BoardInit();

#if defined(__PROFILE__)
    arm::app::Profiler objProfiler;
#endif

    torch::executor::register_all_kernels();
    torch::executor::runtime_init();
```

- BoardInit()：初始化 UART、NPU、HyperRAM 等硬體資源。
- 註冊 ExecuTorch 的所有運算核心（卷積、全連接層等）。
- 初始化 ExecuTorch 執行環境。

2) 模型載入與解析：

```
    // Load model form SD card to HyperRAM
    uint8_t *pu8ModelPTE = (uint8_t *)MODEL_AT_HYPERRAM_ADDR;
    int i32PTESize;

    i32PTESize = PrepareModelToHyperRAM();

    if(i32PTESize <= 0)
    {
        ET_LOG(
            Info,
            "Model loading failed @ 0x%p" PRIx32,
            pu8ModelPTE);
        return -1;
    }

    ET_LOG(
        Info, "PTE in %p Size: %d bytes", pu8ModelPTE, i32PTESize);

    // Create a data loader for the model file in memory
    auto objLoader =
        torch::executor::util::BufferDataLoader(pu8ModelPTE, i32PTESize);
    ET_LOG(Info, "Model PTE file loaded. Size: %u bytes.", i32PTESize);

    // Parse the program file. This is immutable, and can also be reused
    // between multiple execution invocations across multiple threads.
    Result<torch::executor::Program> pProgram =
        torch::executor::Program::load(&objLoader);
    if (!pProgram.ok()) {
        ET_LOG(
            Info,
            "Program loading failed @ 0x%p: 0x%" PRIx32,
            pu8ModelPTE,
           pProgram.error());
        return -2;
```

```
    }

    ET_LOG(Info, "Model buffer loaded, has %zu methods", pProgram->num_methods());
```

- 呼叫 PrepareModelToHyperRAM() 從 SD 卡載入 PTE 模型檔案。
- 建立 BufferDataLoader 包裝記憶體中的模型資料。
- 解析模型檔案建立 pProgram 物件，包含模型的所有 method 和 meta 資料。

3) 方法 (method) 選擇與記憶體規劃：

```
    // For this example we just run the first method found in the program.
    const char* pchMethodName = nullptr;
    {
        const auto pchMethodNameResult = pProgram->get_method_name(0);
        ET_CHECK_MSG(pchMethodNameResult.ok(), "Program has no methods");
        pchMethodName = *pchMethodNameResult;
    }
    ET_LOG(Info, "Running method %s", pchMethodName);

    // Get method meta to setup memory planned buffers
    Result<torch::executor::MethodMeta> pMethodMeta =
        pProgram->method_meta(pchMethodName);
    if (!pMethodMeta.ok()) {
        ET_LOG(
            Info,
            "Failed to get method_meta for %s: 0x%x",
            pchMethodName,
            (unsigned int)pMethodMeta.error());
    }

    // Setup memory allocators
    ArmMemoryAllocator objMethodAllocator(
        sizeof(s_au8MethodAllocationPool), s_au8MethodAllocationPool);

    std::vector<uint8_t*> vecPlannedBuffers; // Owns the memory
    std::vector<torch::executor::Span<uint8_t>> vecPlannedSpans; // Passed to the
allocator

    size_t nNumMemoryPlannedBuffers = pMethodMeta->num_memory_planned_buffers();

    size_t nPlannedBufferMembase = objMethodAllocator.used_size();

    // Allocate memory planned buffers
    for (size_t id = 0; id < nNumMemoryPlannedBuffers; ++id) {
        size_t nBufferSize =
            static_cast<size_t>(pMethodMeta->memory_planned_buffer_size(id).get());
        ET_LOG(Info, "Setting up planned buffer %zu, size %zu.", id, nBufferSize);

        /* Move to it's own allocator when MemoryPlanner is in place. */
        uint8_t* pu8Buffer =
            reinterpret_cast<uint8_t*>(objMethodAllocator.allocate(nBufferSize));
        ET_CHECK_MSG(
```

```
            pu8Buffer != nullptr,
            "Could not allocate memory for memory planned buffer size %zu",
            nBufferSize);
        vecPlannedBuffers.push_back(pu8Buffer);
        vecPlannedSpans.push_back({vecPlannedBuffers.back(), nBufferSize});

    }
```

- 選擇模型的第一個方法進行執行。
- 取得方法的 meta 資料，包含記憶體需求資訊。
- 建立自訂的記憶體分配器。
- 根據 meta 資料分配模型執行所需的計劃緩衝區。

4) 記憶體管理器建立：

```
    // Create the hierarchical allocator for planned buffers
    torch::executor::HierarchicalAllocator objPlannedMemory(
    {vecPlannedSpans.data(), vecPlannedSpans.size()});

    ArmMemoryAllocator objTempAllocator(
        sizeof(s_au8TempAllocationPool), s_au8TempAllocationPool);

    // Create the memory manager
    torch::executor::MemoryManager objMemoryManager(
        &objMethodAllocator, &objPlannedMemory, &objTempAllocator);
```

- HierarchicalAllocator：管理預先分配的計劃緩衝區。
- objTempAllocator：提供臨時記憶體（用於 Ethos-U NPU）。
- objMemoryManager：統一管理三種類型的記憶體分配器。

5) 載入方法 (method)：

```
    // Load the method from the program
    Result<torch::executor::Method> pMethod =
        pProgram->load_method(pchMethodName, &objMemoryManager, pEventTracer);
    if (!pMethod.ok()) {
        ET_LOG(
            Info,
            "Loading of method %s failed with status 0x%" PRIx32,
            pchMethodName,
            pMethod.error());
    }
    ET_LOG(Info, "Method loaded.");
```

- 使用記憶體管理器載入指定的方法。

- 方法載入後即可執行推論。

6) 輸入資料準備：

```
    // label information
    ET_LOG(Info, "Preparing labels...");
    std::vector <std::string> vecLabels;
    GetLabelsVector(vecLabels);

    ET_LOG(Info, "Preparing inputs...");
    size_t nInputMembase = objMethodAllocator.used_size();

    // Print model input tensor information
    PrintModelInputTensorInfo(*pMethod);

    // Get input image data
    uint8_t u8ImgIdx = IMAGE_PATTERN_INDEX;
    const uint8_t *pu8ImgSrc = get_img_array(u8ImgIdx);

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

    // Prepare input tensors from image data
    std::vector<std::pair<char*, size_t>> vecInputBuffers = {
        {(char *)pu8ImgSrc, IMAGE_DATA_SIZE}
    };

    // Prepare the input tensors
    auto pPreparedInputs = ::PrepareInputTensorsFromImage(*pMethod, objMethodAllocator,
vecInputBuffers);

    if (!pPreparedInputs.ok()) {
        ET_LOG(
            Info,
            "Preparing inputs tensors for method %s failed with status 0x%" PRIx32,
            pchMethodName,
            pPreparedInputs.error());
    }
```

- 載入分類標籤向量（ImageNet 1000 類）。
- 顯示模型輸入張量的詳細資訊。
- 根據 IMAGE_PATTERN_INDEX 取得預先編譯的測試影像。
- 呼叫 PrepareInputTensorsFromImage() 準備模型輸入。

7) 模型執行：

```
    // Execute the method
```

```
#if defined(__PROFILE__)
    objProfiler.StartProfiling("Inference");
#endif

    ET_LOG(Info, "Starting the model execution...");
    size_t nExecutorMembase = objMethodAllocator.used_size();
    Error eStatus = pMethod->execute();

#if defined(__PROFILE__)
    objProfiler.StopProfiling();
    objProfiler.PrintProfilingResult();
#endif
```

- 開始效能分析（如果啟用）。
- 記錄執行前的記憶體使用量。
- 執行模型推論。
- 停止效能分析並顯示結果。

8) 記憶體使用統計：

```
    // Calculate executor memory usage
    size_t nExecutorMemsize = objMethodAllocator.used_size() - nExecutorMembase;

    // Print memory usage information
    if (objMethodAllocator.size() != 0) {
        size_t nMethodAllocatorUsed = objMethodAllocator.used_size();
        ET_LOG(
            Info,
            "method_allocator_used:     %zu / %zu  free: %zu ( used: %zu %% ) ",
            nMethodAllocatorUsed,
            objMethodAllocator.size(),
            objMethodAllocator.free_size(),
            100 * nMethodAllocatorUsed / objMethodAllocator.size());
        ET_LOG(
            Info, "method_allocator_planned:  %zu bytes", nPlannedBufferMemsize);
        ET_LOG(Info, "method_allocator_loaded:   %zu bytes", nMethodLoadedMemsize);
        ET_LOG(Info, "method_allocator_input:    %zu bytes", nInputMemsize);
        ET_LOG(Info, "method_allocator_executor: %zu bytes", nExecutorMemsize);
    }
    if (objTempAllocator.size() > 0) {
        ET_LOG(
            Info,
            "temp_allocator_used:       %zu / %zu free: %zu ( used: %zu %% ) ",
            objTempAllocator.used_size(),
            objTempAllocator.size(),
            objTempAllocator.free_size(),
            100 * objTempAllocator.used_size() / objTempAllocator.size());
    }
```

- 詳細統計各階段的記憶體使用情況。

- 包含：總使用量、計劃緩衝區、方法載入、輸入準備、執行階段。
- 計算使用率百分比。

9) 結果處理：

```
// Get and process the outputs
std::vector<torch::executor::EValue> vsOutputs(pMethod->outputs_size());
ET_LOG(Info, "%zu outputs: ", vsOutputs.size());
eStatus = pMethod->get_outputs(vsOutputs.data(), vsOutputs.size());
ET_CHECK(eStatus == Error::Ok);

// Process the output tensor
Tensor objOutputTensor = vsOutputs[MV2_OUTPUT_TENSOR_INDEX].toTensor();
std::vector<float> vecOutputTensorData(objOutputTensor.numel());

for (int j = 0; j < objOutputTensor.numel(); j++) {
    if (objOutputTensor.scalar_type() == ScalarType::Float) {
        auto fProb = objOutputTensor.const_data_ptr<float>()[j];
        vecOutputTensorData[j] = fProb;
    } else if (objOutputTensor.scalar_type() == ScalarType::Byte) {
        auto u8Prob = objOutputTensor.const_data_ptr<uint8_t>()[j];
        vecOutputTensorData[j] = u8Prob;
    } else if (objOutputTensor.scalar_type() == ScalarType::Char) {
        auto i8Prob = objOutputTensor.const_data_ptr<int8_t>()[j];
        vecOutputTensorData[j] = i8Prob;
    }
}
```

- 從執行完成的方法中取得輸出結果。
- 支援多種輸出資料型態（Float、Byte、Char）。
- 統一轉換為 float 向量以便後續分類處理。

10) 分類結果輸出：

```
// Apply softmax to get probabilities
SoftmaxF32(vecOutputTensorData);

// Find the class with the highest probability
int i32MaxCLS = 0;
float fMaxProb = -1;

for (int j = 0; j < objOutputTensor.numel(); j++) {
    if(vecOutputTensorData[j] > fMaxProb)
    {
        fMaxProb = vecOutputTensorData[j];
        i32MaxCLS = j;
    }
}
```

```
    printf("Image classification result: %s(%f) \n", vecLabels[i32MaxCLS].c_str(),
fMaxProb);
```

- 對模型原始輸出執行 Softmax，轉換為機率分佈。
- 遍歷所有類別找出最高機率的預測結果。
- 輸出分類結果：類別名稱和對應機率。

2.3 註冊核心運算子 (RegisterNativeKernels.cpp)

前面章節提到編譯完後的 MobileNetV2 PTE 模型圖裡有兩個需由 CPU 運算的運算子。為了縮減程式碼 ROM 空間，由 CPU 運算的運算子需要註冊其相對應的運算子呼叫程式碼到核心註冊表中。

以 MobileNetV2 PTE 模型圖來說明，使用 Netron 來檢視，分別有「cortex_m::quantize_per_tensor.out」和「cortex_m::dequantize_per_tensor.out」兩個 CPU 運算的運算子，其前綴為「cortex_m」，因此我們從 ThirdParty\executorch_prebuilt\backends\cortex_m\RegisterCodegenUnboxedKernelsEverything.cpp 中，找到相對應的運算子呼叫程式碼，並將其複製到 SampleCode\ExampleCode\ExecuTorch_ImageClassification\RegisterNativeKernels.cpp 裡的 kernels_to_register 表中。

總結一下各個前綴其相對應的運算子呼叫原始程式碼位置：

- cortex_m：

  ThirdParty\executorch_prebuilt\backends\cortex_m\RegisterCodegenUnboxedKernelsEverything.cpp

- quantized_decomposed：

  ThirdParty\executorch_prebuilt\kernels\quantized\RegisterCodegenUnboxedKernelsEverything.cpp

- aten 以及 dim_order_ops：

  ThirdParty\executorch_prebuilt\kernels\portable\RegisterCodegenUnboxedKernelsEverything.cpp

1. 程式說明：

1) 標頭檔引入：

```
#include <executorch/runtime/core/evalue.h>
#include <executorch/runtime/core/exec_aten/exec_aten.h>
#include <executorch/runtime/core/exec_aten/util/tensor_util.h>
#include <executorch/runtime/core/span.h>
#include <executorch/runtime/kernel/operator_registry.h>
#include <executorch/runtime/platform/profiler.h>
```

- evalue.h：提供 EValue 類別，用於封裝不同類型的數據。
- exec_aten.h： ExecuTorch 的 ATen（A Tensor library）實作。
- tensor_util.h： 張量操作工具。
- span.h： 類似 std::span 的容器視圖。
- operator_registry.h： 運算子註冊系統。
- profiler.h： 性能分析工具。

2) 類型別名定義：

```
using KernelSpan = ::executorch::runtime::Span<
                   const ::executorch::ET_RUNTIME_NAMESPACE::Kernel>;
```

- 定義了一個類型別名，用於表示 Kernel 的 Span（類似陣列視圖）。

3) 核心陣列表定義：

```
static Kernel kernels_to_register[] = {
```

- kernels_to_register 變數，它是一個靜態陣列，包含要註冊的所有運算核心。

4) 第一個 Kernel：量化操作。

```
    Kernel(
    "cortex_m::quantize_per_tensor.out",
    [](torch::executor::KernelRuntimeContext & context, Span<EValue*> stack) {
        ET_KERNEL_CHECK_MSG(context, stack.size() == 8, InvalidProgram, /*void*/,
"Expected %" ET_PRIsize_t "args received %" ET_PRIsize_t, (size_t)8, stack.size());
        EValue& input = *stack[0];
        EValue& scale = *stack[1];
        EValue& zero_point = *stack[2];
        EValue& quant_min = *stack[3];
        EValue& quant_max = *stack[4];
        EValue& dtype = *stack[5];
        EValue& out = *stack[6];
        const torch::executor::Tensor & input_base = input.to<torch::executor::Tensor>();
        double scale_base = scale.to<double>();
        int64_t zero_point_base = zero_point.to<int64_t>();
```

```
        int64_t quant_min_base = quant_min.to<int64_t>();
        int64_t quant_max_base = quant_max.to<int64_t>();
        torch::executor::ScalarType dtype_base = dtype.to<torch::executor::ScalarType>();
        torch::executor::Tensor & out_base = out.to<torch::executor::Tensor>();

        internal::EventTracerProfileOpScope
event_tracer_op_scope(context.internal_event_tracer(),
"native_call_quantize_per_tensor.out");
        EXECUTORCH_SCOPE_PROF("native_call_quantize_per_tensor.out");
        cortex_m::native::quantize_per_tensor_out(context, input_base, scale_base,
zero_point_base, quant_min_base, quant_max_base, dtype_base, out_base);
        internal::event_tracer_log_evalue(context.internal_event_tracer(), *stack[6]);

    }
    ),
```

5) 第二個 Kernel：反量化操作。

```
    Kernel(
        "cortex_m::dequantize_per_tensor.out",
    [](torch::executor::KernelRuntimeContext & context, Span<EValue*> stack) {
        ET_KERNEL_CHECK_MSG(context, stack.size() == 8, InvalidProgram, /*void*/,
"Expected %" ET_PRIsize_t "args received %" ET_PRIsize_t, (size_t)8, stack.size());
        EValue& input = *stack[0];
        EValue& scale = *stack[1];
        EValue& zero_point = *stack[2];
        EValue& quant_min = *stack[3];
        EValue& quant_max = *stack[4];
        EValue& dtype = *stack[5];
        EValue& out = *stack[6];
        const torch::executor::Tensor & input_base = input.to<torch::executor::Tensor>();
        double scale_base = scale.to<double>();
        int64_t zero_point_base = zero_point.to<int64_t>();
        int64_t quant_min_base = quant_min.to<int64_t>();
        int64_t quant_max_base = quant_max.to<int64_t>();
        torch::executor::ScalarType dtype_base = dtype.to<torch::executor::ScalarType>();
        torch::executor::Tensor & out_base = out.to<torch::executor::Tensor>();

        internal::EventTracerProfileOpScope
event_tracer_op_scope(context.internal_event_tracer(),
"native_call_dequantize_per_tensor.out");
        EXECUTORCH_SCOPE_PROF("native_call_dequantize_per_tensor.out");
        cortex_m::native::dequantize_per_tensor_out(context, input_base, scale_base,
zero_point_base, quant_min_base, quant_max_base, dtype_base, out_base);
        internal::event_tracer_log_evalue(context.internal_event_tracer(), *stack[6]);

    }
    ),
```

6) Kernel Span 建立：

```
static KernelSpan kernel_span(
    kernels_to_register,
    kernels_to_register + sizeof(kernels_to_register) / sizeof(Kernel));
```

- 建立了一個 KernelSpan 物件，用於包裝 kernels_to_register 陣列，使其能夠被 ExecuTorch 的註冊系統使用。

7) 註冊函數：

```
Error register_all_kernels() {
    Error success_with_kernel_reg = register_kernels(kernel_span);

    if(success_with_kernel_reg != Error::Ok) {
        ET_LOG(Error, "Failed register all kernels");
    }
    return success_with_kernel_reg;
}
```

- 此函數負責註冊所有 kernel 到 ExecuTorch 系統中。

3. 軟體與硬體需求

3.1 軟體需求

- BSP 版本
  - M55M1_Series_BSP_CMSIS_V3.01.003
- ExecuTorch 版本
  - ExecuTorch v1.0.1
- IDE 版本
  - Keil uVision 5.43

3.2 硬體需求

- 電路元件
  - NuMaker-X-M55M1D
- 線路示意圖
  - 將 UART0 TX (PB.13) pin 腳連接到 PC UART RX，以顯示範例代碼的執行結果。

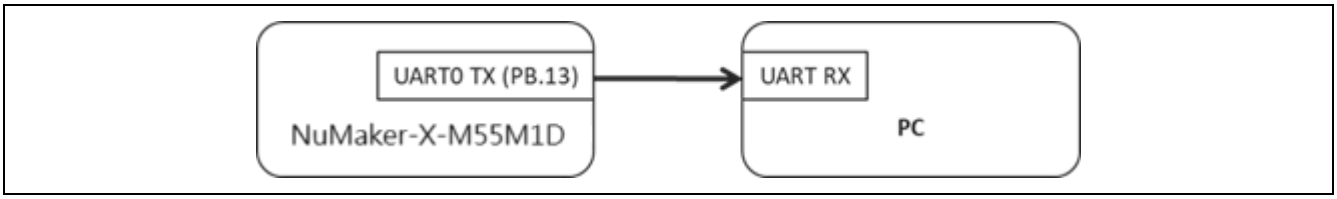

圖 3-1 線路示意圖

4. 目錄資訊

目錄結構如下圖所示。

EC_M55M1_ExecuTorch_ImgClass_V1.00	
Library	Sample code header and source files
CMSIS	Cortex® Microcontroller Software Interface Standard (CMSIS) by Arm® Corp.
Device	CMSIS compliant device header file
StdDriver	All peripheral driver header and source files
Storage	SDH disk IO source file
SampleCode	
ExampleCode	
ExecuTorch_ImageClassification	Source file of example code
Script	Script file
ThirdParty	Third party libraries
executorch	ExecuTorch library source files
FatFs	FAT file system library source files

圖 4-1 目錄資訊

5. 範例程式執行

根據目錄資訊章節進入 ExampleCode 路徑中 Model 資料夾，將預編譯的「mv2.pte」複製到 micro SD 卡，然後將 micro SD 卡置入到 NuMaker-X-M55M1D。

5.1 Keil MDK

1. 根據目錄資訊章節進入 ExampleCode 路徑中的 KEIL 資料夾，雙擊 ExecuTorch_ImageClassification.uvprojx。
2. 進入編譯模式介面
   - 編譯
   - 下載代碼至記憶體
   - 進入 / 離開除錯模式
3. 進入除錯模式介面
   - 執行代碼

5.2 VS Code

1. 根據目錄資訊章節，在 VS Code 中開啟 ExampleCode 路徑中的 VSCode 資料夾。
2. 點擊 VS Code 視窗側邊欄的 CMSIS 圖示，顯示 CMSIS view。
3. 進入 CMSIS 模式介面
   - 編譯
   - 下載程式碼至記憶體且執行
   - 下載程式碼至記憶體且進入除錯模式
4. 進入除錯模式介面
   - 執行程式碼

6. 修訂紀錄

Date	Revision	Description
2026.01.15	1.00	初始發布。

Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.

Please note that all data and specifications are subject to change without notice.
All the trademarks of products and companies mentioned in this datasheet belong to their respective owners.