# 基于 ExecuTorch 的图像分类应用

NuMicro® 32 位系列微控制器范例代码介绍

## 文件信息

| | |
|---|---|
| 应用简述 | 本范例程序使用 NuMaker-X-M55M1D 开发板，实作基于 ExecuTorch 的图像分类推论应用。 |
| BSP 版本 | M55M1 Series BSP CMSIS V3.01.003<br>ExecuTorch v1.0.1 |
| 开发平台 | 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® 推出的轻量级 PyTorch 推论执行框架，专门用于边缘装置（Edge / Embedded / Mobile）。它的目标是让 PyTorch 训练好的模型，能高效、安全、可控地部署到资源受限的装置上，它的设计目标为:

- 可部署性
  - 不依赖 Python
  - C++ Runtime
  - 可在 Linux/Android/RTOS/MCU 执行
- 可裁剪性
  - 编译实际使用到的运算符 (operator / kernel)
  - 适合 SRAM / Flash 受限系统
- 可控性
  - 静态图
  - 移除原始模型图的动态行为（control flow / guards）
- 可扩展性
  - 后端/委托 (Backend / Delegate) 机制
  - 可支持 NPU / DSP

NuMicro® M55M1 系列微控制器搭载高达 220 MHz 运行的 Arm® Cortex®-M55 处理器，除了拥有 Cortex-M55 核心外，还整合了 Arm® Ethos™-U55 神经处理单元（NPU）作为协处理器，用于加速神经网络操作。另外具备 HyperBus 接口，方便扩展模型运算时对 RAM 的需求。

ExecuTorch 支持 Arm Ethos-U 委托机制，因此适合将神经网络模型使用 ExecuTorch 框架布署在 M55M1，在推论时经 NPU 运算加速神经网络运行。

## 1.1 原理

本范例的核心在于利用 ExecuTorch 把模型委托 (delegate) 到支持 Arm Ethos-U (NPU) 的 PTE (PyTorch ExecuTorch format) 推论档案，使得该模型在推论时能利用 M55M1 上的 Ethos-U 做推论加速。整个工作流程如图 1-1 会有底下三步骤:

1. 模型导出 (export)：将动态 PyTorch 程序 (program) 转化成静态可分析图，在这静态可分析图里，不会有控制流运算符 (control flow operator (if/loop))，以及模型形状/数据型态 (shape/dtype) 都要静态化。

2. 编译出 ExecuTorch program：将静态分析图转化为支持目标硬件的 ExecuTorch program，ExecuTorch program 会以 PTE 静态推论文件格式储存。以 Ethos-U 来说，会有几个部份：

   - 量化(quantization)：由于 Ethos-U 是整数处理器，量化是必须的。因此利用了 executorch.backends.arm.quantizer 量化器的功能进行量化。
   - 进行分区 (partitioning)：决定模型的哪些部分由 Ethos-U 加速执行，哪些部分可能回退到 CPU 。ARM 的分区策略目标是分区整个图形以在 Ethos-U 上执行。
   - 进行 Ethos-U 后端编译：针对 Ethos-U 分区，将 IR (Intermediate Representation) 转换为 TOSA IR (ARM 的机器学习应用标准) 后，送入 Vela 编译程序，生成 Ethos-U 的命令比特流 (command stream)。
   - 进行降级 (lowering)：将中间表示图形转换为运行时使用的格式。这包括创建委托 (delegates) (在运行时加载到 Ethos-U 的工作负载)、内存规划以及将整个程序串行化 (serialization) 到一个单一的 PTE (ExecuTorch Program) 档案中。

3. 在装置上执行 ExecuTorch program:在装置上使用 ExecuTorch runtime 加载 ExecuTorch Program，并且依 ExecuTorch Program 的指令进行模型推论。当 runtime 执行到模型图里的 Ethos-U backend 运算符，则会调用 Ethos-U driver 进行 Ethos-U 加速器运算。

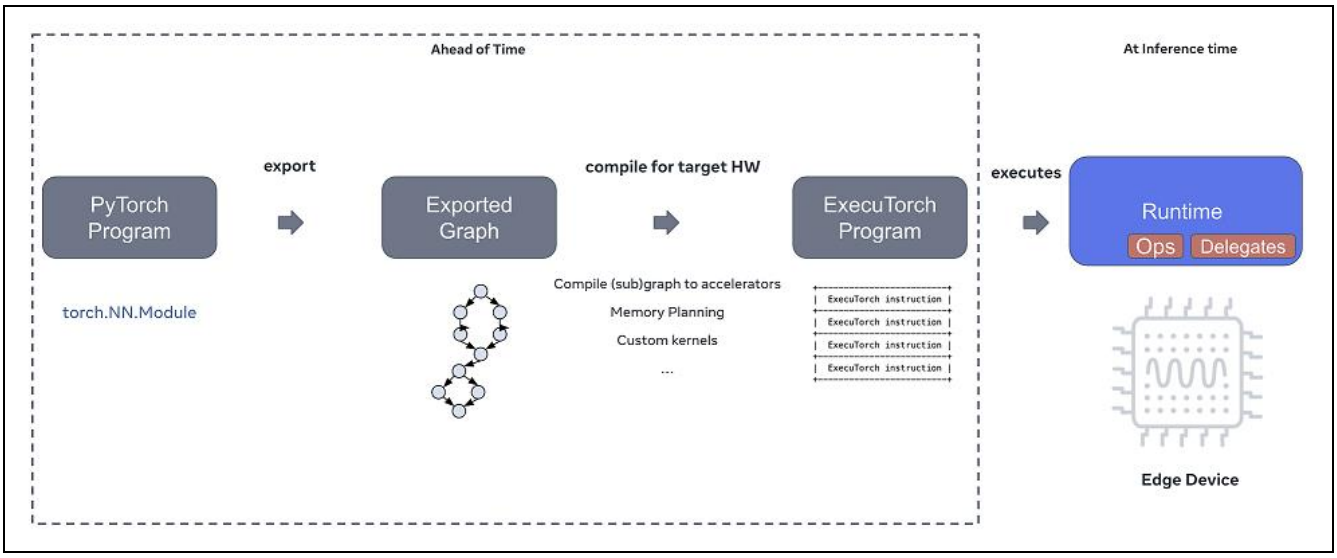


图 1-1 ExecuTorch 工作流程 (来源: How ExecuTorch Works)

上述步骤 1 和步骤 2 称之 Ahead-of-Time(AOT) 编译，是 off-line 流程。因此需在 PC 端架设 ExecuTorch 环境并执行编译脚本，导出模型 PTE 档案。相关的环境搭建请参考 GitHub 上 ExecuTorch 的 README.md 说明。本文件将专注在编译脚本和 runtime 环境的说明。

## 1.2 MobileNetV2 介绍

MobileNetV2 是由 Google® 在 2018 年提出的一种高效、轻量化的卷积神经网络（CNN）架构。它在 MobileNetV1 的基础上进行了重大改进，目的在为行动端（如手机、嵌入式设备）提供兼具低延迟与高精度的视觉模型。其核心架构为「具有线性瓶颈的倒置残差模块 (Inverted Residuals with Linear Bottlenecks)」。

### 1.2.1 核心架构

MobileNetV2 采用了倒置残差模块，这是与传统残差网络 (如 ResNet) 最大的不同之处。

- 传统残差网络 (RestNet)：采用「压缩 → 卷积 → 扩张」的结构 (沙漏型)。先将输入维度降低以减少计算量，处理后再恢复高维度。
- 倒置残差模块：采用 「扩张 → 卷积 → 压缩」 的结构 (橄榄型)。
  - Expansion：先用 1x1 卷积将低维通道数扩张（通常扩张 6 倍），进入高维特征空间。
  - Depthwise：在高维度下进行轻量化的深度卷积，这能让模型捕捉到更丰富的特征。

- Projection/Bottleneck：将特征投射回低维通道。

### 1.2.2 关键技术

- 线性瓶颈 (Linear Bottlenecks)

  研究团队发现，在低维空间（瓶颈层）使用非线性激活函数（如 ReLU）会导致严重的信息丢失。因此在倒置残差模块的最后一层 1x1 卷积后，不使用 ReLU，直接输出线性特征。这确保了模型在压缩数据时能保留关键的信息。

- ReLU6 激活函数

  MobileNetV2 广泛使用 ReLU6，这将输出限制在 [0, 6] 之间。这在移动端使用低精度（如 Float16 或 Quantized INT8）计算时，能防止数值溢出并提高模型的强健性。

- 深度可分离卷积 (Depthwise Separable Convolution)

  继承自 V1，将标准卷积拆分为 Depthwise（逐通道处理）与 Pointwise（逐像素融合）。这让计算量降低了约 8 至 9 倍，而精度仅有些微下降。

### 1.2.3 网络架构与效能

MobileNetV2 总共由一个初始的全卷积层和 19 个残差瓶颈层组成，其特性如表 1-1。

| 特性 | 描述 |
|---|---|
| 输入尺寸 | 224 x 224 x 3 |
| 输出尺寸 | 1000 (ImageNet 类别数) |
| 参数数量 | 约 3.4 M（比 V1 更少） |
| ImageNet 准确率 | Top-1 约 72.0%（优于 V1 的 70.6%） |
| 计算量 (MACs) | 约 300 M |

表 1-1 MobileNetV2 特性

## 1.3 执行结果

### 1.3.1 产生模型 PTE 档案

这一阶段是在 PC 上执行模型编译脚本，将 MobileNetV2 模型转成 ExecuTorch runtime 可执行的 PTE 档案。环境是采用 Ubuntu 24.04 ，并在 Ubuntu 上安装 ExecuTorch 及其相依 python 套件。

执行 MobileNetV2 编译脚本时，会自动下载预训练 MobileNetV2 模型，因此请保持网络的联机。执行结束后会产生「mv2_arm_delegate_ethos-u55-256.pte」档案，如图 1-2。使用 Netron 可视化工具检视这档案，可看到转化后的模型图，如图 1-3。

转化后的模型图会有一个 EthosUBackend 运算符和两个需由 CPU 处理的运算符 (quantize_per_tensor 和 dequantize_per_tensor) 。在程序代码的部份需特别将 CPU 处理的运算符注册到 kernel 表。

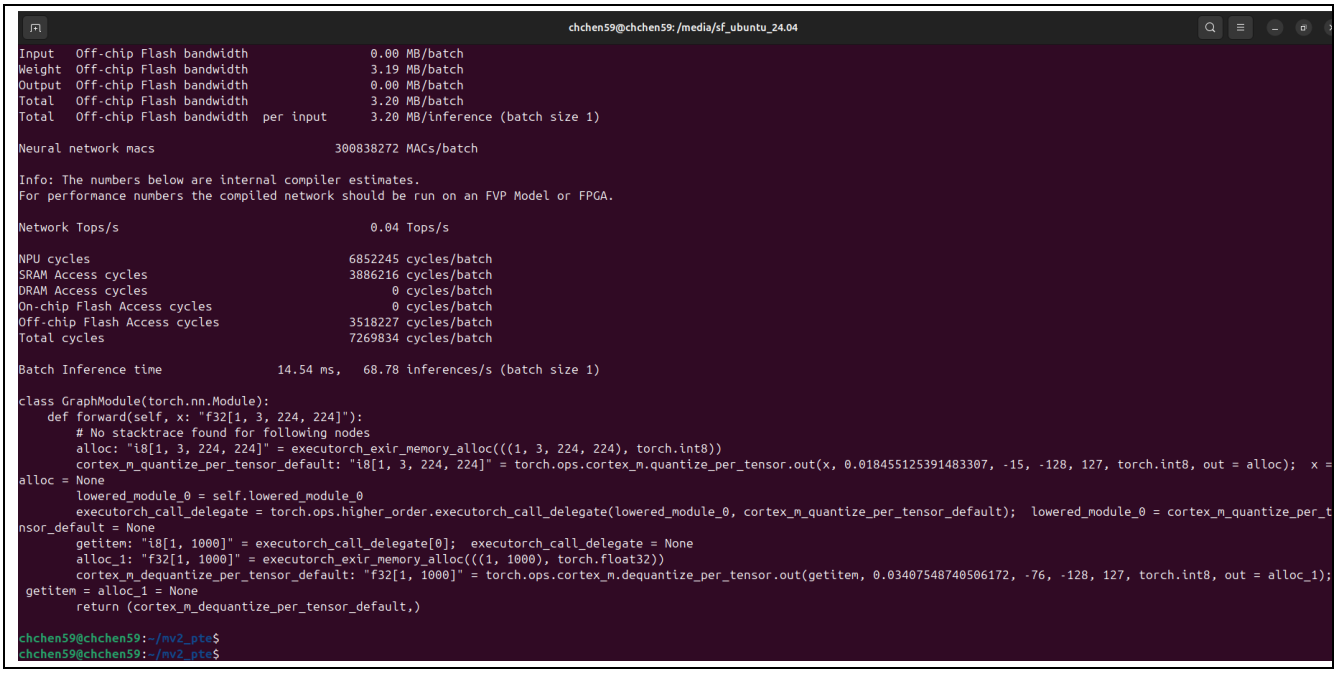


图 1-2 MobileNetV2 编译结果

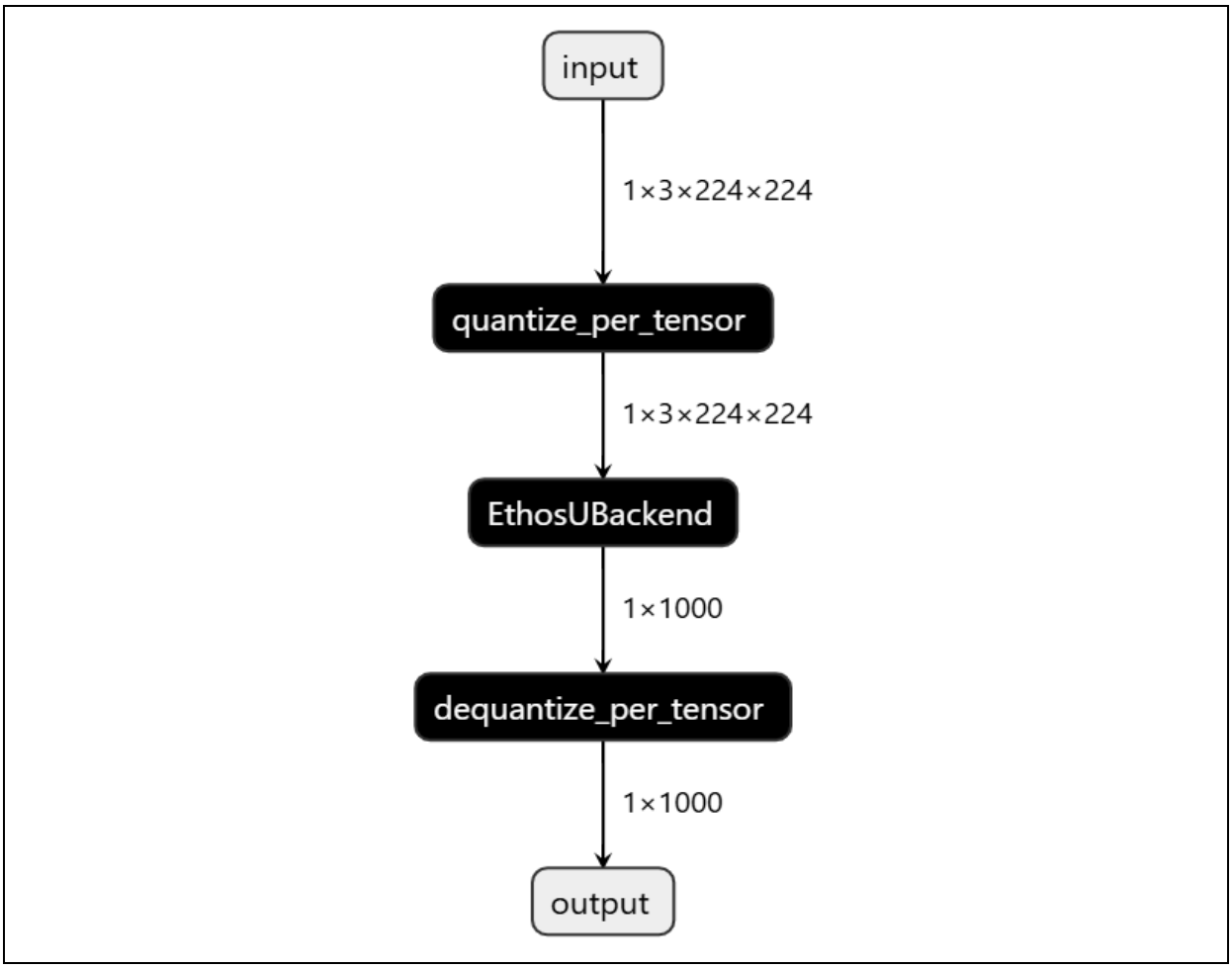


图 1-3 模型图

### 1.3.2 执行

由于 MobileNetV2 PTE 档案超过内建 Flash 容量，因此需将档案放入 SD card ，并将档案重新命名为「mv2.pte」，后续由 runtime 主程序从 SD card 加载到 HyperRAM 执行。

范例程序执行编译和刻录后，运行结果，如图 1-4。

图 1-4 范例程序执行结果

# 2. 代码介绍

## 2.1 MobileNetV2 编译脚本 (mv2_pte.py)

MobileNetV2 编译脚本程序代码在 Script/mv2_pte.py。这份 mv2_pte.py 脚本主要用于将 PyTorch 的 MobileNetV2 模型导出、量化，并针对 Arm Ethos-U55 与 Cortex-M 后端进行优化与转换，最终生成可在嵌入式设备上运行的 ExecuTorch 格式模型。以下为完整说明：

1. 主要功能:

   - 下载并加载 MobileNetV2 预训练模型。
   - 准备范例输入（可用随机张量或真实图片）。
   - 将模型导出为 PyTorch Exported Program。
   - 进行后训练量化（PTQ），并再次导出量化后模型。
   - 针对 Arm Ethos-U55 进行分割、编译与优化。
   - 进行 Cortex-M 后端相关转换。
   - 转换为 ExecuTorch 格式并储存为「.pte」档案。

2. 主要步骤说明:

   1) 加载模型与准备输入:

```
# 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 下载 MobileNetV2 预训练权重。
- get_example_inputs 函数可选择加载随机张量或下载真实图片（dog.jpg）并进行预处理。

2) 导出模型:

```
# 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,
)
```

- 将模型设为 eval 模式。
- 使用 torch.export.export 将模型与范例输入导出为 Exported Program。
- 使用 torch.export.save 储存导出的模型，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,
)
```

- 汇入 Ethos-U 相关 quantizer 与 config。
- 建立 EthosUCompileSpec，描述目标硬件与编译选项。
- 建立 quantizer 并设定全局对称量化参数。
- 使用 prepare_pt2e 进行量化前准备，并用范例输入进行校准。

- 使用 convert_pt2e 完成量化，并再次导出量化后的 Exported Program。

4) Ethos-U 后端分割与优化：

```
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，根据 compile spec 进行分割。
- 使用 to_edge_transform_and_lower 进行分割、优化与 IR 降级。

5) Cortex-M 后端转换:

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

- 执行 transform_for_cortex_m_backend，将部分运算转换为 Cortex-M 支持的运算。

6) 转换为 ExecuTorch 格式并储存:

```
# 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 转换为 ExecuTorch 格式。

- 使用 save_pte_program 储存为 *.pte* 档案，供嵌入式设备部署。

## 2.2 Runtime 主程序 (main.cpp)

### 2.2.1 主要步骤说明:

1. 定义区段:

```
#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 中的存放地址。
- __PROFILE__: 启用效能分析功能。
- MV2_*_TENSOR_INDEX: MobileNetV2 模型的输入/输出张量索引。
- IMAGE_PATTERN_INDEX: 选择测试影像（0: cat, 1: dog, 2: tiger）。

2. 内存池配置:

```
// 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];
```

- 预先分配两个大型内存池，放置在 SRAM/HyperRAM 中。
- s_au8TempAllocationPool：用于 Ethos-U NPU 的暂存缓冲区（1.5 MB）。
- s_au8MethodAllocationPool：用于方法执行的内存池（1.4 MB）。

3. 日志输出函数:

```
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 平台层的日志输出实作。
- 将 ExecuTorch 的内部日志导向到标准错误输出（stderr）。
- 格式化显示日志等级、文件名、行号和信息内容。

4. 辅助函数群:

1) 模型输入张量信息显示:

```
// 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]);
        }
    }
}
```

- 分析并显示模型所有输入张量的详细信息。包括：输入数量、每个张量的内存大小、数据型态、维度顺序和大小。

2) 模型加载到 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;
}
```

- 从 SD 卡读取「mv2.pte」模型档案。
- 以 512 字节为单位逐步加载到 HyperRAM 指定地址。
- 包含档案完整性检查机制（#if 0 批注掉的验证代码）。
- 返回实际读取的档案大小，失败则返回负值。

3) 自定义内存分配器:

```
// 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_;
};
```

- 继承 ExecuTorch 的基础内存分配器。
- 新增内存使用量追踪功能。
- 提供 used_size() 和 free_size() 方法来监控内存使用状况。
- 考虑内存对齐的实际使用量计算。

4) 影像正规化处理:

```
// 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++;
                }
            }
        }
    }
}
```

- 将 RGB888 格式的影像数据转换为模型输入张量。
- 支持三种数据型态：Float (标准化)、Char (减 128)、Byte (直接复制)。
- Float 型态使用 ImageNet 的均值和标准偏差进行标准化。
- 按照 channel-first 格式 (C, H, W) 排列资料。

5) 准备输入张量:

```
// 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});
}
```

- 为 MobileNetV2 模型准备输入张量。
- 从 method 的 meta 数据采集输入张量规格。

- 分配适当的内存并建立张量对象。
- 调用 NormalizeImageToInputTensor() 填入影像数据。

6) Softmax 实作:

```
// 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;
    }
}
```

- 对浮点数向量执行 Softmax 运算。
- 先减去最大值防止数值溢出（数值稳定性）。
- 计算指数和并且正规化，得到机率分布。
- 确保所有输出值总和为 1。

5. 主程序 main() 说明:

主程序流程如图 2-1。

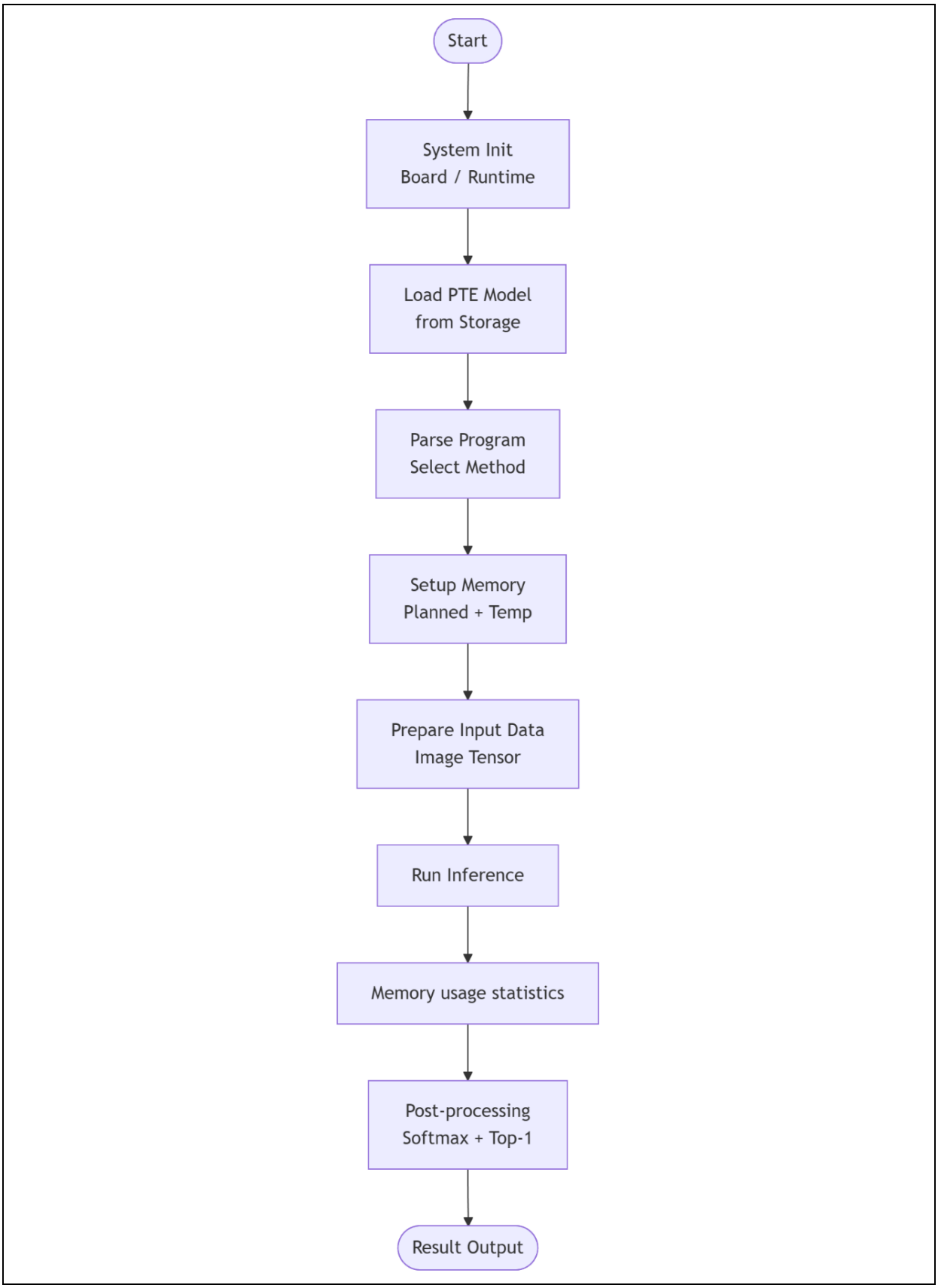
Start
System Init
Board / Runtime
Load PTE Model
from Storage
Parse Program
Select Method
Setup Memory
Planned + Temp
Prepare Input Data
Image Tensor
Run Inference
Memory usage statistics
Post-processing
Softmax + Top-1
Result Output

图 2-1 主程序流程

1) 初始化阶段:

```
    /* 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 等硬件资源。
- 注册 ExecuTorch 的所有运算核心（卷积、全连接层等）。
- 初始化 ExecuTorch 执行环境。

2) 模型加载与解析:

```
    // 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());
```

- 调用 PrepareModelToHyperRAM() 从 SD 卡加载 PTE 模型档案。
- 建立 BufferDataLoader 包装内存中的模型数据。
- 解析模型档案建立 pProgram 对象，包含模型的所有 method 和 meta 数据。

3) 方法 (method) 选择与内存规划:

```
    // 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});

    }
```

- 选择模型的第一个方法进行执行。
- 取得方法的 meta 数据，包含内存需求信息。
- 建立自定义的内存分配器。
- 根据 meta 数据分配模型执行所需的计划缓冲区。

4) 内存管理器建立:

```
    // 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：管理预先分配的计划缓冲区。
- objTempAllocator：提供临时内存（用于 Ethos-U NPU）。
- objMemoryManager：统一管理三种类型的内存分配器。

5) 加载方法 (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.");
```

- 使用内存管理器加载指定的方法。
- 方法加载后即可执行推论。

6) 输入数据准备:

```
    // 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());
    }
```

- 加载分类标签向量（ImageNet 1000 类）。
- 显示模型输入张量的详细信息。
- 根据 IMAGE_PATTERN_INDEX 取得预先编译的测试影像。

- 调用 PrepareInputTensorsFromImage() 准备模型输入。

7) 模型执行:

```
    // 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
```

- 开始效能分析（如果启用）。
- 记录执行前的内存使用量。
- 执行模型推论。
- 停止效能分析并显示结果。

8) 内存使用统计:

```
    // 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());
    }
```

- 详细统计各阶段的内存使用情况。
- 包含：总使用量、计划缓冲区、方法加载、输入准备、运行时间。
- 计算使用率百分比。

9) 结果处理:

```
    // 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;
        }
    }
```

- 从执行完成的方法中取得输出结果。
- 支持多种输出数据型态（Float、Byte、Char）。
- 统一转换为 float 向量以便后续分类处理。

10) 分类结果输出:

```
    // 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);
```

- 对模型原始输出执行 Softmax，转换为机率分布。
- 遍历所有类别找出最高机率的预测结果。
- 输出分类结果：类别名称和对应机率。

## 2.3 注册核心运算符 (RegisterNativeKernels.cpp)

前面章节提到编译完后的 MobileNetV2 PTE 模型图里有两个需由 CPU 运算的运算符。为了缩减程序代码 ROM 空间，由 CPU 运算的运算符需要注册其相对应的运算符调用程序代码到核心注册表中。

以 MobileNetV2 PTE 模型图来说明，使用 Netron 来检视，分别有「cortex_m::quantize_per_tensor.out」和「cortex_m::dequantize_per_tensor.out」两个 CPU 运算的运算符，其前缀为「cortex_m」，因此我们从 ThirdParty\executorch_prebuilt\backends\cortex_m\RegisterCodegenUnboxedKernelsEverything.cpp 中，找到相对应的运算符调用程序代码，并将其复制到 SampleCode\ExampleCode\ExecuTorch_ImageClassification\RegisterNativeKernels.cpp 里的 kernels_to_register 表中。

总结一下各个前缀其相对应的运算符调用源代码位置：

- 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. 程序说明:

1) 头文件引入:

```
#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 类别，用于封装不同类型的数据。
- exec_aten.h: ExecuTorch 的 ATen（A Tensor library）实作。
- tensor_util.h: 张量操作工具。
- span.h: 类似 std::span 的容器视图。
- operator_registry.h: 运算符注册系统。
- profiler.h: 性能分析工具。

2) 类型别名定义:

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

- 定义了一个类型别名，用于表示 Kernel 的 Span（类似数组视图）。

3) 核心数组表定义:

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

- kernels_to_register 变量，它是一个静态数组，包含要注册的所有运算核心。

4) 第一个 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) 第二个 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));
```

建立了一个 KernelSpan 对象，用于包装 kernels_to_register 数组，使其能够被 ExecuTorch 的注册系统使用。

7) 注册函数:

```
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;
}
```

- 此函数负责注册所有 kernel 到 ExecuTorch 系统中。

# 3. 软件与硬件需求

## 3.1 软件需求

- 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 硬件需求

- 电路组件
  - NuMaker-X-M55M1D
- 线路示意图
  - 将 UART0 TX (PB.13) pin 脚连接到 PC UART RX，以显示范例代码的执行结果。

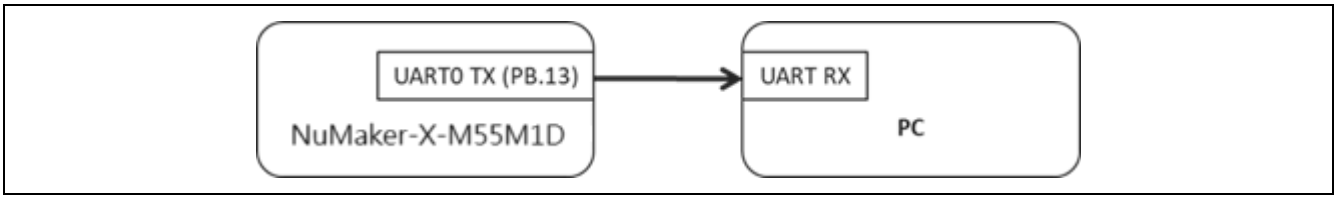


图 3-1 线路示意图

# 4. 目录信息

目录结构如下图所示。

| | |
|---|---|
| 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 |

图 4-1 目录信息

# 5. 范例程序执行

根据目录信息章节进入 ExampleCode 路径中 Model 文件夹，将预编译的「mv2.pte」复制到 micro SD 卡，然后将 micro SD 卡置入到 NuMaker-X-M55M1D。

## 5.1 Keil MDK

1. 根据目录信息章节进入 ExampleCode 路径中的 KEIL 文件夹，双击 ExecuTorch_ImageClassification.uvprojx。
2. 进入编译模式接口
   - 编译
   - 下载代码至内存
   - 进入 / 离开除错模式
3. 进入除错模式接口
   - 执行代码

## 5.2 VS Code

1. 根据目录信息章节，在 VS Code 中开启 ExampleCode 路径中的 VSCode 文件夹。
2. 点击 VS Code 窗口侧边栏的 CMSIS 图标，显示 CMSIS view。
3. 进入 CMSIS 模式接口
   - 编译
   - 下载程序代码至内存且执行
   - 下载程序代码至内存且进入除错模式
4. 进入除错模式接口
   - 执行程序代码

# 6. 修订纪录

| Date | Revision | Description |
|---|---|---|
| 2026.01.15 | 1.00 | 初始发布。 |

## 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.*