



ESP32

ESP-DL 用户指南



Release v1.0.0-68-g3eb27101d0

乐鑫信息科技

2023 年 08 月 21 日




Table of contents

Table of contents	i
1 介绍	3
1.1 概述	3
1.2 入门指南	3
1.3 尝试模型库中的模型	3
1.4 部署模型	3
1.5 反馈	5
2 快速入门	7
2.1 获取 ESP-IDF	7
2.2 获取 ESP-DL 并运行示例	7
2.3 ESP-DL 用作组件	8
3 教程	9
3.1 使用 TVM 自动生成模型部署项目	9
3.1.1 准备	9
3.1.2 步骤 1: 模型量化	10
3.1.3 步骤 2: 部署模型	11
3.1.4 步骤 3: 运行模型	12
3.2 手动部署模型	13
3.2.1 准备	13
3.2.2 部署模型	15
3.3 手动部署已完成量化的模型	17
3.3.1 步骤 1: 保存模型系数	18
3.3.2 步骤 2: 配置模型	19
3.3.3 步骤 3: 转换模型系数	19
3.3.4 步骤 4: 构建模型	19
3.3.5 步骤 5: 运行模型	22
3.4 定制层步骤	23
3.4.1 步骤 1: 从 Layer 类派生层	23
3.4.2 步骤 2: 实现 build()	23
3.4.3 步骤 3: 实现 call()	24
4 工具	25
4.1 量化工具包	25
4.1.1 量化工具包	25
4.1.2 量化规范	28
4.1.3 量化工具包 API	33
4.2 转换工具	36
4.2.1 convert.py 使用说明	36
4.2.2 config.json 配置规范	36
4.3 图片工具	39
4.3.1 图片转换工具 convert_to_u8.py	39
4.3.2 显示工具 display_image.py	40
5 性能	41

5.1	猫脸检测延迟	41
5.2	人脸检测延迟	41
5.3	人脸识别延迟	41
6	词汇表	43
	索引	45
	索引	45

本文档为 ESP-DL 官方文档。如需阅读具体某款芯片的文档，请在页面左上方的下拉菜单中选择您的目标芯片。

ESP-DL 是由乐鑫官方针对乐鑫系列芯片 [ESP32](#)、[ESP32-S2](#)、[ESP32-S3](#) 和 [ESP32-C3](#) 所提供的高性能深度学习开发库。

Chapter 1

介绍

ESP-DL 是由乐鑫官方针对乐鑫系列芯片 [ESP32](#)、[ESP32-S2](#)、[ESP32-S3](#) 和 [ESP32-C3](#) 所提供的高性能深度学习开发库。

1.1 概述

ESP-DL 为 [神经网络推理](#)、[图像处理](#)、[数学运算](#) 以及一些 [深度学习模型](#) 提供 API，通过 ESP-DL 能够快速便捷地将乐鑫各系列芯片产品用于人工智能应用。

ESP-DL 无需借助任何外围设备，因此可作为一些项目的组件，例如可将其作为 [ESP-WHO](#) 的一个组件，该项目包含数个项目级图像应用实例。下图展示了 ESP-DL 的组成及作为组件时在项目中的位置。

1.2 入门指南

安装并入门 ESP-DL，请参考[快速入门](#)。

请使用 ESP-IDF 在 [release/v5.0](#) 分支上的 [最新版本](#)。

1.3 尝试模型库中的模型

ESP-DL 在 [模型库](#) 中提供了一些模型的 API，如人脸检测、人脸识别、猫脸检测等。下表为 ESP-DL 所提供的模型，支持开箱即用。

项目	API 实例
人脸检测	human_face_detect
人脸识别	face_recognition
猫脸检测	cat_face_detect

1.4 部署模型

如果想部署模型，请参考[部署模型的步骤介绍](#)，这一说明中包含三个可运行的实例，有助于迅速设计模型。

阅读上述文档时，可能会用到以下资料：

- [DL API](#)

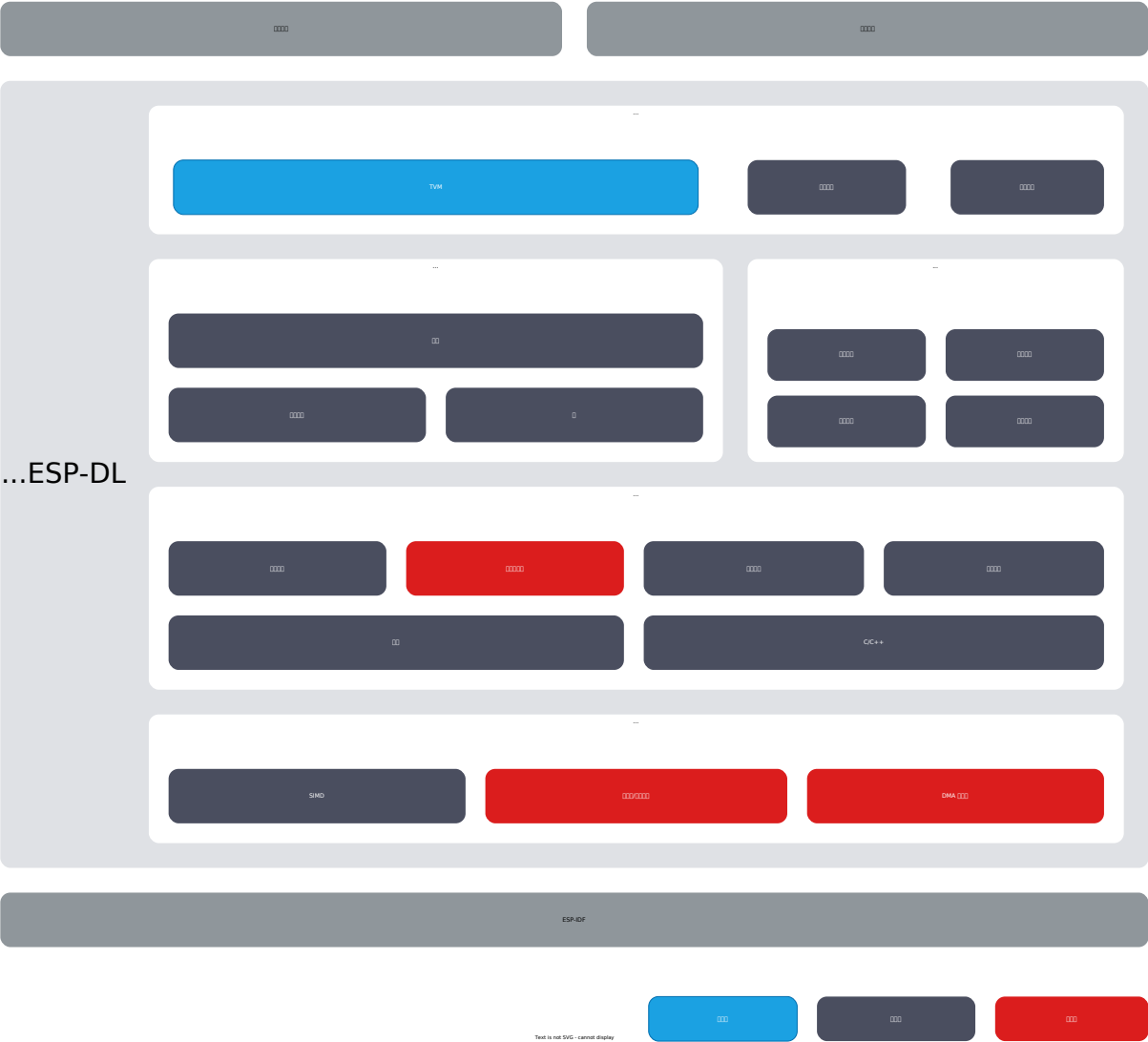


图 1: 架构概览

- [变量与常量](#)：其中提供的信息包括：
 - * 变量：张量
 - * 常量：过滤器、偏差、激活函数
- [定制层](#)：介绍如何定制层。
- [API 文档](#)：关于层、神经网络、数学和工具的 API 指南。
关于 API 的使用说明，请暂时查看头文件注释。
- 平台转换
 - TVM (推荐)：使用 AI 编译器 TVM 来部署模型，TVM 相关内容请参考 [TVM](#)
 - 量化工具：用来量化浮点模型, 并评估定点模型在 ESP SoCs 上的表现
 - * 量化工具：请参考[量化工具包概览](#)
 - * 量化工具 API：请参考[量化工具包 API](#)
 - 转换工具：可对 `coefficient.npy` 进行浮点量化的工具和配置文件。
 - * `config.json`：请参考[config.json 配置规范](#)
 - * `convert.py`：请参考[convert.py 使用说明](#)
`convert.py` 需在 Python 3.7 或更高版本中运行。
- 软硬件加速
 - [量化规范](#)：浮点量化规则

1.5 反馈

如果在使用中发现了错误或者需要新的功能，请提交相关 [issue](#)，我们会优先实现最受期待的功能。

Chapter 2

快速入门

本文描述了如何搭建 ESP-DL 环境，支持使用 [乐鑫](#) 或其他供应商设计的任意 ESP 开发板。

2.1 获取 ESP-IDF

ESP-DL 的运行需要依赖 ESP-IDF。有关 ESP-IDF 的详细安装步骤，请查看 [ESP-IDF 编程指南](#)。

2.2 获取 ESP-DL 并运行示例

1. 使用如下命令下载 ESP-DL：

```
git clone https://github.com/espressif/esp-dl.git
```

2. 打开终端，进入 [tutorial/convert_tool_example](#) 文件夹：

```
cd ~/esp-dl/tutorial/convert_tool_example
```

或是进入 [examples](#) 文件夹下的其他示例项目。

3. 使用以下命令设置目标芯片：

```
idf.py set-target [SoC]
```

将 [SoC] 替换为目标芯片，如 esp32、esp32s2、esp32s3。

注意 ESP32-C3 仅适用于无需 PSRAM 的应用。

4. 烧录固件，打印结果：

```
idf.py flash monitor
```

如果在第二步进入的是 [tutorial/convert_tool_example](#) 文件夹，

- 目标芯片是 ESP32，则

```
MNIST::forward: 37294 µs  
Prediction Result: 9
```

- 目标芯片是 ESP32-S3，则

```
MNIST::forward: 6103 µs  
Prediction Result: 9
```

2.3 ESP-DL 用作组件

ESP-DL 是包含多种深度学习 API 的仓库。我们推荐将 ESP-DL 用作其他项目的组件。

比如, ESP-DL 可以作为 [ESP-WHO](#) 仓库的子模块, 只需将 ESP-DL 加入到 [esp-who/components/](#) 目录即可。

Chapter 3

教程

3.1 使用 TVM 自动生成模型部署项目

本案例介绍了使用 TVM 部署模型的完整流程。

3.1.1 准备

ESP-DL 是适配 ESP 系列芯片的深度学习推理框架。本库无法完成模型的训练，用户可使用 [TensorFlow](#)，[PyTorch](#) 等训练平台来训练模型，然后再通过 ESP-DL 部署模型。

为了帮助您理解本指南中的概念，建议您下载并熟悉以下工具：

- ESP-DL 库：详细了解 ESP-DL，包括量化规范、数据排布格式、支持的加速层。
- ONNX：一种用于表示深度学习模型的开放格式。
- TVM：一个端到端的深度学习编译框架，适用于 CPU、GPU 和各种机器学习加速芯片。

安装 Python 依赖包

环境要求：

- Python == 3.7 or 3.8
- [ONNX](#) == 1.12.0
- [ONNX Runtime](#) == 1.14.0
- [ONNX Optimizer](#) == 0.2.6
- [ONNX Simplifier](#) == 0.4.17
- numpy
- decorator
- attrs
- typing-extensions
- psutil
- scipy

您可以使用 [tools/tvm/requirements.txt](#) 来安装相关 Python 依赖包：

```
pip install -r requirements.txt
```

配置 TVM 包

您可以使用 [tools/tvm/download.sh](#) 来下载我们已经编译好的 TVM 包，请注意：当前编译的 libtvm.so 包仅支持在 Linux 操作系统环境中运行。

```
./download.sh
```

TVM 包将被下载到 `esp-dl/tools/tvm/python/tvm` 中。下载完包后，需要设置环境变量 `PYTHONPATH`，指定 TVM 库的位置。可以在终端运行以下命令，也可以在 `~/.bashrc` 文件中添加以下行。

```
export PYTHONPATH='${PYTHONPATH}:/path-to-esp-dl/esp-dl/tools/tvm/python'
```

3.1.2 步骤 1：模型量化

为了部署的模型在芯片上能快速运行，需要将训练好的浮点模型转换定点模型。

常见的量化手段分为两种：

1. 训练后量化（post-training quantization）：将已有的模型转化为定点数表示。这种方法相对简单，不需要重新训练网络，但在有些情况下会有一定的精度损失。
2. 量化训练（quantization-aware training）：在网络训练过程中考虑量化带来的截断误差和饱和效应。这种方式使用上更复杂，但效果会更好。

ESP-DL 中目前只支持第一种方法。若无法接受量化后的精度损失，请考虑使用第二种方式。

步骤 1.1：转换为 ONNX 格式模型

量化脚本基于开源的 AI 模型格式 [ONNX](#) 运行。其他平台训练得到的模型需要先转换为 ONNX 格式才能使用该工具包。

以 TensorFlow 平台为例，您可在脚本中使用 `tf2onnx` 将训练好的 TensorFlow 模型转换成 ONNX 模型格式，实例代码如下：

```
model_proto, _ = tf2onnx.convert.from_keras(tf_model, input_signature=spec,
↪ opset=13, output_path="mnist_model.onnx")
```

更多平台转换实例可参考 [xxx_to_onnx](#)。

步骤 1.2：预处理

在预处理过程中，将会对 float32 模型进行一系列操作，以便为量化做好准备。

```
python -m onnxruntime.quantization.preprocess --input model.onnx --output model_
↪ opt.onnx
```

参数说明：

- `input`：指定输入的待处理 float32 模型文件路径。
- `output`：指定输出的处理后 float32 模型文件路径。

预处理包括以下可选步骤：

- 符号形状推断（Symbolic Shape Inference）：这个步骤会对输入和输出的张量形状进行推断。符号形状推断可以帮助模型在推理之前确定张量的形状，以便更好地进行后续优化和处理。
- ONNX Runtime 模型优化（ONNX Runtime Model Optimization）：这个步骤使用 ONNX Runtime 来进行模型优化。ONNX Runtime 是一个高性能推理引擎，可以针对特定硬件和平台进行模型优化，以提高推理速度和效率。模型优化包括诸如图优化、内核融合、量化等技术，以优化模型的执行。
- ONNX 形状推断（ONNX Shape Inference）：这个步骤根据 ONNX 格式模型推断张量形状，从而更好地理解 and 优化模型。ONNX 形状推断可以为模型中的张量分配正确的形状，帮助后续的优化和推理。

步骤 1.3: 量化

量化工具接受预处理后的 float32 模型作为输入，并生成一个 int8 量化模型。

```
python esp_quantize_onnx.py --input_model model_opt.onnx --output_model model_
↳ quant.onnx --calibrate_dataset calib_img.npy
```

参数说明：

- **input_model**: 指定输入模型的路径和文件名，应为预处理过的 float32 模型，以 ONNX 格式 (.onnx) 保存。
- **output_model**: 指定输出模型的路径和文件名，将是量化处理后的模型，以 ONNX 格式 (.onnx) 保存。
- **calibrate_dataset**: 指定用于校准的数据集路径和文件名，应为包含校准数据的 NumPy 数组文件 (.npy)，用于生成量化器的校准统计信息。

`tools/tvm/esp_quantize_onnx.py` 中创建了一个用于模型的输入数据读取器，使用这些输入数据来运行模型，以校准每个张量的量化参数，并生成量化模型。具体流程如下：

- **创建输入数据读取器**: 首先，创建一个输入数据读取器，用于从数据源中读取输入的校准数据。用于校准的数据集应保存为 NumPy 数组文件，其中包含输入图片的集合。例如 `model.onnx` 的输入大小为 [32, 32, 3]，`calibe_images.npy` 存储的则是 500 张校准图片的数据，形状为 [500, 32, 32, 3]。
- **运行模型进行校准**: 接下来，代码会使用输入数据读取器提供的数据来运行模型。通过将输入数据传递给模型，模型会进行推断 (inference)，生成输出结果。在这个过程中，代码会根据实际输出结果和预期结果，校准每个张量的量化参数。这个校准过程旨在确定每个张量的量化范围、缩放因子等参数，以便在后续的量化转换中准确地表示数据。
- **生成量化模型**: 校准完量化参数后，代码将使用这些参数对模型进行量化转换。这个转换过程会将模型中的浮点数权重和偏差替换为量化表示，使用较低的位精度来表示数值。生成的量化模型会保留量化参数，以便在后续的部署过程中正确还原数据。请注意，不要在这个量化模型上运行推理过程，可能会与板上运行的结果不一致，具体的调试流程请参考后续章节。

3.1.3 步骤 2: 部署模型

将量化后的 ONNX 模型部署到 ESP 系列芯片上。只有在 ESP32-S3 上运行的部分算子支持 ISA 加速。

支持加速的算子请查看 [include/layer](#)。更多 ISA 相关介绍请查看 《ESP32-S3 技术参考手册》。

步骤 2.1: 准备输入

准备一张输入图像，输入的图像大小应该与得到的 ONNX 模型输入大小一致。模型输入大小可通过 Netron 工具查看。

步骤 2.2: 部署项目生成

使用 TVM 自动生成一个项目，用来运行给定输入的模型推理。

```
python export_onnx_model.py --target_chip esp32s3 --model_path model_quant.onnx --
↳ img_path input_sample.npy --template_path "esp-dl/tools/tvm/template_project_for_
↳ model" --out_path "esp-dl/example"
```

参数说明：

- **target_chip**: 目标芯片的名称。上述命令中目标芯片是 `esp32s3`，表示生成的示例项目将针对 ESP32-S3 芯片进行优化。
- **model_path**: 经过量化的 ONNX 模型的路径。请提供模型的完整路径和文件名。
- **img_path**: 输入图像的路径。请提供输入图像的完整路径和文件名。
- **template_path**: 用于示例项目的模板路径。默认提供的模板程序为 `tools/tvm/template_project_for_model`。
- **out_path**: 生成的示例项目的输出路径。请提供目标目录的路径。

`tools/tvm/export_onnx_model.py` 将量化的 ONNX 模型加载到 TVM 中，并对模型进行布局转换和优化，经过一定的预处理后最终编译成适配 ESP 后端的代码。具体流程如下：

- 通过 `tvm.relay.frontend.from_onnx` 函数将 ONNX 模型转换为 TVM 的中间表示（Relay IR）。
- 将 ONNX 默认的 NCHW 布局转换为 ESP-DL 期望的布局 NHWC。定义 `desired_layouts` 字典，指定要进行布局转换的操作和期望的布局。这里将对模型中的“`qnn.conv2d`”和“`nn.avg_pool2d`”的布局进行转换。转换通过 TVM 的 `transform` 机制来完成。
- 执行针对部署到 ESP 芯片的预处理，包括算子的重写、融合、标注。
- 通过 TVM 的 BYOC（Bring Your Own Codegen）机制编译生成模型的 C 代码，包括支持的加速算子。BYOC 是 TVM 的机制，允许用户自定义代码生成。BYOC 可以将模型的特定部分编译为 ESP-DL 的加速算子，以便在目标硬件上进行加速。使用 TVM 的 `tvm.build` 函数，将 Relay IR 编译为目标硬件上的可执行代码。
- 将生成的模型部分的代码集成到提供的模板工程文件中。

3.1.4 步骤 3：运行模型

步骤 3.1：运行推理

上一步生成的工程文件 `new_project` 结构如下：

```

├── CMakeLists.txt
├── components
│   ├── esp-dl
│   └── tvm_model
│       ├── CMakeLists.txt
│       ├── crt_config
│       └── model
├── main
│   ├── app_main.c
│   ├── input_data.h
│   ├── output_data.h
│   └── CMakeLists.txt
├── partitions.csv
├── sdkconfig.defaults
├── sdkconfig.defaults.esp32
├── sdkconfig.defaults.esp32s2
└── sdkconfig.defaults.esp32s3

```

配置好终端 ESP-IDF（请注意 ESP-IDF 的版本）环境后，即可运行项目：

```
idf.py set-target esp32s3
idf.py flash monitor
```

步骤 3.2：调试

模型的推理过程在 `components/tvm_model/model/codegen/host/src/default_lib1.c` 里的 `tvmgen_default__tvm_main__` 函数中定义。如果想查看板子上运行的模型的输出是否与预期相符，可以参考以下步骤。

模型的第一层为 `conv2d` 算子，从函数体中可以看到 `tvmgen_default_esp_main_0` 调用了 ESP-DL 提供的 `conv2d` 加速算子来实现第一层的卷积操作。添加下列示例代码可以获得该层的结果，示例代码只输出了前 16 个数。

```
int8_t *out = (int8_t *)sid_4_let;
for(int i=0; i<16; i++)
    printf("%d, ", out[i]);
printf("\n");
```

export_onnx_model.py 中的 debug_onnx_model 函数用于调试模型板上运行的结果，验证是否符合预期。请确保模型完成部署、并在板上运行后，再调用 debug_onnx_model 函数。

```
debug_onnx_model(args.target_chip, args.model_path, args.img_path)
```

debug_onnx_model 函数里使用 “evaluate_onnx_for_esp” 函数处理 Relay 使其与板上计算方法一致，请注意这个函数仅适用于调试阶段。

```
mod = evaluate_onnx_for_esp(mod, params)

m = GraphModuleDebug(
    lib["debug_create"]("default", dev),
    [dev],
    lib.graph_json,
    dump_root = os.path.dirname(os.path.abspath(model_path)) + "/tvmdbg",
)
```

通过 TVM 的 GraphModuleDebug 将计算图的全部信息输出到 tvmdbg 目录下，输出的 tvmdbg_graph_dump.json 文件中包含了图中各个运算结点的信息。更多说明可查看 [TVM Debugger 文档](#)。输出文件中第一个卷积输出层的名称为 tvmgen_default_fused_nn_relu，输出的大小为 [1, 32, 32, 16]，输出类型为 int8。

```
tvm_out = tvm.nd.empty((1, 32, 32, 16), dtype="int8")
m.debug_get_output("tvmgen_default_fused_nn_relu", tvml_out)
print(tvm_out.numpy().flatten()[0:16])
```

根据上述信息创建一个变量存储这一层的输出，可以比较这一输出是否与板子上运行得到的结果一致。

3.2 手动部署模型

本案例介绍了如何使用我们提供的[量化工具包](#)来完成模型的部署。

注意，如果模型已通过其他平台量化：

- 若使用的量化方法与 ESP-DL 的[量化规范](#)不同（如 TFLite int8 模型），则无法使用 ESP-DL 进行部署
- 若量化方法一致，则可参考[部署量化模型](#)案例来完成部署。

建议先学习训练后量化 (post-training quantization) 的相关知识。

3.2.1 准备

步骤 1: 模型转换

为了部署模型，必须将训练好的浮点模型转换为 ESP-DL 适配的整型模型格式。由于本库使用的量化方式和参数排列方式与一些平台不同，请使用我们提供的工具[量化工具包](#)来完成转换。

步骤 1.1: 转换为 ONNX 格式模型 量化工具包基于开源的 AI 模型格式 [ONNX](#) 运行。其他平台训练得到的模型需要先转换为 ONNX 格式才能使用该工具包。

以 TensorFlow 平台为例，您可在脚本中使用 [tf2onnx](#) 将训练好的 TensorFlow 模型转换成 ONNX 模型格式，实例代码如下：

```
model_proto, _ = tf2onnx.convert.from_keras(tf_model, input_signature=spec,
↪ opset=13, output_path="mnist_model.onnx")
```

更多平台转换实例可参考 [xxx_to_onnx](#)。

步骤 1.2: 转换为 ESP-DL 适配模型 准备好 ONNX 模型后, 即可使用量化工具包来完成量化。

本小节以 `tools/quantization_tool/examples/mnist_model_example.onnx` 和 `tools/quantization_tool/examples/example.py` 为例。

步骤 1.2.1: 环境准备 环境要求:

- Python == 3.7
- Numba == 0.53.1
- ONNX == 1.9.0
- ONNX Runtime == 1.7.0
- ONNX Optimizer == 0.2.6

您可以使用 `tools/quantization_tool/requirements.txt` 来安装相关 Python 依赖包:

```
pip install -r requirements.txt
```

步骤 1.2.2: 模型优化 量化工具包中的优化器可优化 ONNX 模型图结构:

```
# Optimize the onnx model
model_path = 'mnist_model_example.onnx'
optimized_model_path = optimize_fp_model(model_path)
```

步骤 1.2.3: 模型量化和转换 创建 Python 脚本 `example.py` 来完成转换。

量化工具包中的校准器可将浮点模型量化成可适配 ESP-DL 的整型模型。为了实现训练后量化, 请参考以下实例准备校准集, 该校准集可以是训练集或验证集的子集:

```
# Prepare the calibration dataset
# 'mnist_test_data.pickle': this pickle file stores test images from keras.
↳ datasets.mnist
with open('mnist_test_data.pickle', 'rb') as f:
    (test_images, test_labels) = pickle.load(f)

# Normalize the calibration dataset in the same way as for training
test_images = test_images / 255.0

# Prepare the calibration dataset
calib_dataset = test_images[0:5000:50]
```

```
# Calibration
model_proto = onnx.load(optimized_model_path)
print('Generating the quantization table:')

# Initialize an calibrator to quantize the optimized MNIST model to an int16 model.
↳ using per-tensor minmax quantization method
calib = Calibrator('int16', 'per-tensor', 'minmax')
calib.set_providers(['CPUExecutionProvider'])

# Obtain the quantization parameter
calib.generate_quantization_table(model_proto, calib_dataset, 'mnist_calib.pickle')

# Generate the coefficient files for esp32s3
calib.export_coefficient_to_cpp(model_proto, pickle_file_path, 'esp32s3', '.',
↳ 'mnist_coefficient', True)
```

使用以下命令运行准备好的转换脚本:

```
python example.py
```

然后会看到如下的打印日志，其中包含了模型输入和每层输出的量化指数位，会用于接下来定义模型的步骤中：

```
Generating the quantization table:
Converting coefficient to int16 per-tensor quantization for esp32s3
Exporting finish, the output files are: ./mnist_coefficient.cpp, ./mnist_
↪coefficient.hpp

Quantized model info:
model input name: input, exponent: -15
Reshape layer name: sequential/flatten/Reshape, output_exponent: -15
Gemm layer name: fused_gemm_0, output_exponent: -11
Gemm layer name: fused_gemm_1, output_exponent: -11
Gemm layer name: fused_gemm_2, output_exponent: -9
```

关于工具包中更多 API 的介绍可阅读[量化工具包 API](#)。

3.2.2 部署模型

步骤 2: 构建模型

步骤 2.1: 从 `include/layer/dl_layer_model.hpp` 中的模型类派生一个新类 量化时配置的为 int16 量化，故模型以及之后的层均继承 `<int16_t>` 类型。

```
class MNIST : public Model<int16_t>
{
};
```

步骤 2.2: 将层声明为成员变量

```
class MNIST : public Model<int16_t>
{
private:
    // Declare layers as member variables
    Reshape<int16_t> l1;
    Conv2D<int16_t> l2;
    Conv2D<int16_t> l3;

public:
    Conv2D<int16_t> l4; // Make the l4 public, as the l4.get_output() will be_
↪fetched outside the class.
};
```

步骤 2.3: 用构造函数初始化层 根据模型量化得到的文件和打印日志来初始化层。量化后的模型参数存储在 `tutorial/quantization_tool_example/model/mnist_coefficient.cpp` 中，获取参数的函数头文件为 `tutorial/quantization_tool_example/model/mnist_coefficient.hpp`。

例如定义卷积层“l2”，根据打印得知输出的指数位为“-11”，该层的名称为“fused_gemm_0”。您可调用 `get_fused_gemm_0_filter()` 获取改卷积层权重，调用 `get_fused_gemm_0_bias()` 获取该卷积层偏差，调用 `get_fused_gemm_0_activation()` 获取该卷积层激活参数。同理，配置其他参数，可构造整个 MNIST 模型结构如下：

```
class MNIST : public Model<int16_t>
{
    // ellipsis member variables

    MNIST() : l1(Reshape<int16_t>({1,1,784})),
              l2(Conv2D<int16_t>(-11, get_fused_gemm_0_filter(), get_fused_gemm_0_
↪bias(), get_fused_gemm_0_activation(), PADDING_SAME_END, {}, 1, 1, "l1")),
```

(下页继续)

(续上页)

```

        l3(Conv2D<int16_t>(-11, get_fused_gemm_1_filter(), get_fused_gemm_1_
↪bias(), get_fused_gemm_1_activation(), PADDING_SAME_END, {}, 1, 1, "l2")),
        l4(Conv2D<int16_t>(-9, get_fused_gemm_2_filter(), get_fused_gemm_2_
↪bias(), NULL, PADDING_SAME_END, {}, 1, 1, "l3")){}
};

```

有关如何初始化不同运算层，请查看 [esp-dl/include/layer/](#) 文件夹中相应的.hpp 文件。

步骤 2.4: 实现 `void build(Tensor<input_t> &input)` 为了便于区分 模型 `build()` 和 层 `build()`，现定义：

- 模型 `build()` 为 `Model.build()`；
- 层 `build()` 为 `Layer.build()`。

`Model.build()` 会调用所有 `Layer.build()`。`Model.build()` 仅在输入形状变化时有效。若输入形状没有变化，则 `Model.build()` 不会被调用，从而节省计算时间。

有关 `Model.build()` 何时被调用，请查看 [步骤 3: 运行模型](#)。

有关如何调用每一层的 `Layer.build()`，请查看 [esp-dl/include/layer/](#) 文件夹中相应的.hpp 文件。

```

class MNIST : public Model<int16_t>
{
    // ellipsis member variables
    // ellipsis constructor function

    void build(Tensor<int16_t> &input)
    {
        this->l1.build(input);
        this->l2.build(this->l1.get_output());
        this->l3.build(this->l2.get_output());
        this->l4.build(this->l3.get_output());
    }
};

```

步骤 2.5: 实现 `void call(Tensor<input_t> &input)` `Model.call()` 会调用所有 `Layer.call()`。有关如何调用每一层的 `Layer.call()`，请查看 [esp-dl/include/layer/](#) 文件夹中相应的.hpp 文件。

```

class MNIST : public Model<int16_t>
{
    // ellipsis member variables
    // ellipsis constructor function
    // ellipsis build(...)

    void call(Tensor<int16_t> &input)
    {
        this->l1.call(input);
        input.free_element();

        this->l2.call(this->l1.get_output());
        this->l1.get_output().free_element();

        this->l3.call(this->l2.get_output());
        this->l2.get_output().free_element();

        this->l4.call(this->l3.get_output());
        this->l3.get_output().free_element();
    }
};

```

(下页继续)

```
}
};
```

步骤 3: 运行模型

- 创建模型对象
- 定义输入
 - 输入的图像大小: 与模型输入大小一致 (若原始图像是通过摄像头获取的, 可能需要调整大小)
 - 量化输入: 用训练时相同的方式对输入进行归一化, 并使用[步骤 1.2.3: 模型量化和转换](#)输出日志中的 **input_exponent** 对归一化后的浮点值进行定点化, 设置输入的指数位

```
int input_height = 28;
int input_width = 28;
int input_channel = 1;
int input_exponent = -15;
int16_t *model_input = (int16_t *)dl::tool::malloc_aligned_prefer(input_
    ↪ height*input_width*input_channel, sizeof(int16_t *));
for(int i=0 ; i<input_height*input_width*input_channel; i++){
    float normalized_input = example_element[i] / 255.0; //normalization
    model_input[i] = (int16_t)DL_CLIP(normalized_input * (1 << -input_
    ↪ exponent), -32768, 32767);
}
```

- 定义输入张量

```
Tensor<int16_t> input;
input.set_element((int16_t *)model_input).set_exponent(input_exponent).set_
    ↪ shape({28, 28, 1}).set_auto_free(false);
```

- 运行 `Model.forward()` 进行神经网络推理。 `Model.forward()` 的过程如下:

```
forward()
{
    if (input_shape is changed)
    {
        Model.build();
    }
    Model.call();
}
```

示例: [tutorial/quantization_tool_example/main/app_main.cpp](#) 文件中的 MNIST 对象和 `forward()` 函数。

```
// model forward
MNIST model;
model.forward(input);
```

3.3 手动部署已完成量化的模型

本案例介绍了如何使用我们提供的[转换脚本](#)来完成模型的部署。教程中的示例是可运行的 **MNIST** 分类项目, 以下简称 MNIST。

注意:

- 如果已经通过其他平台对模型进行量化, 若使用的量化方法同 ESP-DL 的[量化规范](#)不同, 则无法使用 ESP-DL 进行部署 (如 TFLite int8 模型);
- 若未进行量化, 则可参考[手动部署模型](#)案例来完成部署。

建议先学习训练后量化 (post-training quantization) 相关知识。

有关如何定制层，请查看[定制层步骤](#)。

本教程的结构如下所示。

```
tutorial/
├── CMakeLists.txt
├── main
│   ├── app_main.cpp
│   └── CMakeLists.txt
├── model
│   ├── mnist_coefficient.cpp    (generated in Step 3)
│   ├── mnist_coefficient.hpp   (generated in Step 3)
│   ├── mnist_model.hpp
│   └── npy
│       ├── config.json
│       ├── l1_bias.npy
│       ├── l1_filter.npy
│       ├── l2_compress_bias.npy
│       ├── l2_compress_filter.npy
│       ├── l2_depth_filter.npy
│       ├── l3_a_compress_bias.npy
│       ├── l3_a_compress_filter.npy
│       ├── l3_a_depth_filter.npy
│       ├── l3_b_compress_bias.npy
│       ├── l3_b_compress_filter.npy
│       ├── l3_b_depth_filter.npy
│       ├── l3_c_compress_bias.npy
│       ├── l3_c_compress_filter.npy
│       ├── l3_c_depth_filter.npy
│       ├── l3_d_compress_bias.npy
│       ├── l3_d_compress_filter.npy
│       ├── l3_d_depth_filter.npy
│       ├── l3_e_compress_bias.npy
│       ├── l3_e_compress_filter.npy
│       ├── l3_e_depth_filter.npy
│       ├── l4_compress_bias.npy
│       ├── l4_compress_filter.npy
│       ├── l4_depth_activation.npy
│       ├── l4_depth_filter.npy
│       ├── l5_compress_bias.npy
│       ├── l5_compress_filter.npy
│       ├── l5_depth_activation.npy
│       └── l5_depth_filter.npy
└── README.md
```

3.3.1 步骤 1: 保存模型系数

使用 `numpy.save()` 函数，保存.npy 格式的模型浮点系数：

```
numpy.save(file=f'{filename}', arr=coefficient)
```

神经网络的每一层都需要有：

- **过滤器**：保存为 '`{layer_name}_filter.npy`'
- **偏差**：保存为 '`{layer_name}_bias.npy`'
- **激活函数**：具有系数的激活函数，如 *LeakyReLU*、*PReLU*，保存为 '`{layer_name}_activation.npy`'

示例： `tutorial/convert_tool_example/model/np/` 文件夹中.npy 文件里的 MNIST 项目系数。

3.3.2 步骤 2：配置模型

根据`config.json` 配置规范，在 `config.json` 文件中配置模型。

示例：`tutorial/convert_tool_example/model/npv/config.json` 文件中 MNIST 项目的配置。

3.3.3 步骤 3：转换模型系数

将 `coefficient.npy` 文件和 `config.json` 准备好且保存在同一文件夹后，使用 `convert.py`（请参考[convert.py 使用说明](#)）把系数转换为 C/C++ 代码。

示例：

运行如下命令

```
python ../convert.py -i ./model/npv/ -n mnist_coefficient -o ./model/
```

然后 `tutorial/convert_tool_example/model` 文件夹中会生成两个文件：`mnist_coefficient.cpp` 和 `mnist_coefficient.hpp`。

之后，调用 `get_{layer_name}_***()` 即可获取每层的系数。比如要获取“l1”的过滤器，可调用 `get_l1_filter()`。

3.3.4 步骤 4：构建模型

步骤 4.1：从 `dl_layer_model.hpp` 中的模型类派生一个新类

```
class MNIST : public Model<int16_t>
{
};
```

步骤 4.2：将层声明为成员变量

```
class MNIST : public Model<int16_t>
{
private:
    Conv2D<int16_t> l1; // a layer named l1
    DepthwiseConv2D<int16_t> l2_depth; // a layer named l2_depth
    Conv2D<int16_t> l2_compress; // a layer named l2_compress
    DepthwiseConv2D<int16_t> l3_a_depth; // a layer named l3_a_depth
    Conv2D<int16_t> l3_a_compress; // a layer named l3_a_compress
    DepthwiseConv2D<int16_t> l3_b_depth; // a layer named l3_b_depth
    Conv2D<int16_t> l3_b_compress; // a layer named l3_b_compress
    DepthwiseConv2D<int16_t> l3_c_depth; // a layer named l3_c_depth
    Conv2D<int16_t> l3_c_compress; // a layer named l3_c_compress
    DepthwiseConv2D<int16_t> l3_d_depth; // a layer named l3_d_depth
    Conv2D<int16_t> l3_d_compress; // a layer named l3_d_compress
    DepthwiseConv2D<int16_t> l3_e_depth; // a layer named l3_e_depth
    Conv2D<int16_t> l3_e_compress; // a layer named l3_e_compress
    Concat2D<int16_t> l3_concat; // a layer named l3_concat
    DepthwiseConv2D<int16_t> l4_depth; // a layer named l4_depth
    Conv2D<int16_t> l4_compress; // a layer named l4_compress
    DepthwiseConv2D<int16_t> l5_depth; // a layer named l5_depth

public:
    Conv2D<int16_t> l5_compress; // a layer named l5_compress. Make the l5_
    ↪compress public, as the l5_compress.get_output() will be fetched outside the_
    ↪class.
};
```

步骤 4.3: 用构造函数初始化层

步骤 3: 转换模型系数 生成的 "mnist_coefficient.hpp" 文件中有层的系数, 用该系数初始化层。有关如何初始化每一层, 请查看 [include/layer/](#) 文件夹中相应的.hpp 文件。

```
class MNIST : public Model<int16_t>
{
    // ellipsis member variables

    MNIST() : l1(Conv2D<int16_t>(-2, get_l1_filter(), get_l1_bias(), get_l1_
↪activation(), PADDING_VALID, {}, 2, 2, "l1")),
              l2_depth(DepthwiseConv2D<int16_t>(-1, get_l2_depth_filter(), NULL, ↪
↪get_l2_depth_activation(), PADDING_SAME_END, {}, 2, 2, "l2_depth")),
              l2_compress(Conv2D<int16_t>(-3, get_l2_compress_filter(), get_l2_
↪compress_bias(), NULL, PADDING_SAME_END, {}, 1, 1, "l2_compress")),
              l3_a_depth(DepthwiseConv2D<int16_t>(-1, get_l3_a_depth_filter(), ↪
↪NULL, get_l3_a_depth_activation(), PADDING_VALID, {}, 1, 1, "l3_a_depth")),
              l3_a_compress(Conv2D<int16_t>(-12, get_l3_a_compress_filter(), get_
↪l3_a_compress_bias(), NULL, PADDING_VALID, {}, 1, 1, "l3_a_compress")),
              l3_b_depth(DepthwiseConv2D<int16_t>(-2, get_l3_b_depth_filter(), ↪
↪NULL, get_l3_b_depth_activation(), PADDING_VALID, {}, 1, 1, "l3_b_depth")),
              l3_b_compress(Conv2D<int16_t>(-12, get_l3_b_compress_filter(), get_
↪l3_b_compress_bias(), NULL, PADDING_VALID, {}, 1, 1, "l3_b_compress")),
              l3_c_depth(DepthwiseConv2D<int16_t>(-12, get_l3_c_depth_filter(), ↪
↪NULL, get_l3_c_depth_activation(), PADDING_SAME_END, {}, 1, 1, "l3_c_depth")),
              l3_c_compress(Conv2D<int16_t>(-12, get_l3_c_compress_filter(), get_
↪l3_c_compress_bias(), NULL, PADDING_SAME_END, {}, 1, 1, "l3_c_compress")),
              l3_d_depth(DepthwiseConv2D<int16_t>(-12, get_l3_d_depth_filter(), ↪
↪NULL, get_l3_d_depth_activation(), PADDING_SAME_END, {}, 1, 1, "l3_d_depth")),
              l3_d_compress(Conv2D<int16_t>(-11, get_l3_d_compress_filter(), get_
↪l3_d_compress_bias(), NULL, PADDING_SAME_END, {}, 1, 1, "l3_d_compress")),
              l3_e_depth(DepthwiseConv2D<int16_t>(-11, get_l3_e_depth_filter(), ↪
↪NULL, get_l3_e_depth_activation(), PADDING_SAME_END, {}, 1, 1, "l3_e_depth")),
              l3_e_compress(Conv2D<int16_t>(-12, get_l3_e_compress_filter(), get_
↪l3_e_compress_bias(), NULL, PADDING_SAME_END, {}, 1, 1, "l3_e_compress")),
              l3_concat(-1, "l3_concat"),
              l4_depth(DepthwiseConv2D<int16_t>(-12, get_l4_depth_filter(), NULL, ↪
↪get_l4_depth_activation(), PADDING_VALID, {}, 1, 1, "l4_depth")),
              l4_compress(Conv2D<int16_t>(-11, get_l4_compress_filter(), get_l4_
↪compress_bias(), NULL, PADDING_VALID, {}, 1, 1, "l4_compress")),
              l5_depth(DepthwiseConv2D<int16_t>(-10, get_l5_depth_filter(), NULL, ↪
↪get_l5_depth_activation(), PADDING_VALID, {}, 1, 1, "l5_depth")),
              l5_compress(Conv2D<int16_t>(-9, get_l5_compress_filter(), get_l5_
↪compress_bias(), NULL, PADDING_VALID, {}, 1, 1, "l5_compress")) {}

};
```

步骤 4.4: 实现 void build(Tensor<input_t> &input)

为了便于区分 模型 build() 和 层 build() , 现定义:

- 模型 build() 为 Model.build() ;
- 层 build() 为 Layer.build() 。

Model.build() 会调用所有 Layer.build() 。Model.build() 仅在输入形状变化时有效。若输入形状没有变化, 则 Model.build() 不会被调用, 从而节省计算时间。

有关 Model.build() 何时被调用, 请查看 [步骤 5: 运行模型](#)。

有关如何调用每一层的 Layer.build() , 请查看 [include/layer/](#) 文件夹中相应的.hpp 文件。

```

class MNIST : public Model<int16_t>
{
    // ellipsis member variables
    // ellipsis constructor function

    void build(Tensor<int16_t> &input)
    {
        this->l1.build(input);
        this->l2_depth.build(this->l1.get_output());
        this->l2_compress.build(this->l2_depth.get_output());
        this->l3_a_depth.build(this->l2_compress.get_output());
        this->l3_a_compress.build(this->l3_a_depth.get_output());
        this->l3_b_depth.build(this->l2_compress.get_output());
        this->l3_b_compress.build(this->l3_b_depth.get_output());
        this->l3_c_depth.build(this->l3_b_compress.get_output());
        this->l3_c_compress.build(this->l3_c_depth.get_output());
        this->l3_d_depth.build(this->l3_b_compress.get_output());
        this->l3_d_compress.build(this->l3_d_depth.get_output());
        this->l3_e_depth.build(this->l3_d_compress.get_output());
        this->l3_e_compress.build(this->l3_e_depth.get_output());
        this->l3_concat.build({&this->l3_a_compress.get_output(), &this->l3_c_
        ↪compress.get_output(), &this->l3_e_compress.get_output()});
        this->l4_depth.build(this->l3_concat.get_output());
        this->l4_compress.build(this->l4_depth.get_output());
        this->l5_depth.build(this->l4_compress.get_output());
        this->l5_compress.build(this->l5_depth.get_output());
    }
};

```

步骤 4.5: 实现 `void call(Tensor<input_t> &input)`

`Model.call()` 会调用所有 `Layer.call()`。有关如何调用每一层的 `Layer.call()`，请查看 [include/layer/](#) 文件夹中相应的 `.hpp` 文件。

```

class MNIST : public Model<int16_t>
{
    // ellipsis member variables
    // ellipsis constructor function
    // ellipsis build(...)

    void call(Tensor<int16_t> &input)
    {
        this->l1.call(input);
        input.free_element();

        this->l2_depth.call(this->l1.get_output());
        this->l1.get_output().free_element();

        this->l2_compress.call(this->l2_depth.get_output());
        this->l2_depth.get_output().free_element();

        this->l3_a_depth.call(this->l2_compress.get_output());
        // this->l2_compress.get_output().free_element();

        this->l3_a_compress.call(this->l3_a_depth.get_output());
        this->l3_a_depth.get_output().free_element();

        this->l3_b_depth.call(this->l2_compress.get_output());
        this->l2_compress.get_output().free_element();
    }
};

```

(下页继续)

(续上页)

```

        this->l3_b_compress.call(this->l3_b_depth.get_output());
        this->l3_b_depth.get_output().free_element();

        this->l3_c_depth.call(this->l3_b_compress.get_output());
        // this->l3_b_compress.get_output().free_element();

        this->l3_c_compress.call(this->l3_c_depth.get_output());
        this->l3_c_depth.get_output().free_element();

        this->l3_d_depth.call(this->l3_b_compress.get_output());
        this->l3_b_compress.get_output().free_element();

        this->l3_d_compress.call(this->l3_d_depth.get_output());
        this->l3_d_depth.get_output().free_element();

        this->l3_e_depth.call(this->l3_d_compress.get_output());
        this->l3_d_compress.get_output().free_element();

        this->l3_e_compress.call(this->l3_e_depth.get_output());
        this->l3_e_depth.get_output().free_element();

        this->l3_concat.call({&this->l3_a_compress.get_output(), &this->l3_c_
→compress.get_output(), &this->l3_e_compress.get_output()}, true);

        this->l4_depth.call(this->l3_concat.get_output());
        this->l3_concat.get_output().free_element();

        this->l4_compress.call(this->l4_depth.get_output());
        this->l4_depth.get_output().free_element();

        this->l5_depth.call(this->l4_compress.get_output());
        this->l4_compress.get_output().free_element();

        this->l5_compress.call(this->l5_depth.get_output());
        this->l5_depth.get_output().free_element();
    }
};

```

3.3.5 步骤 5: 运行模型

- 创建模型对象
- 运行 `Model.forward()` 进行神经网络推理。`Model.forward()` 的过程如下:

```

forward()
{
    if (input_shape is changed)
    {
        Model.build();
    }
    Model.call();
}

```

示例: [tutorial/convert_tool_example/main/app_main.cpp](#) 文件中的 MNIST 对象和 `forward()` 函数。

```

// model forward
MNIST model;
model.forward(input);

```

3.4 定制层步骤

Conv2D、DepthwiseConv2D 等 ESP-DL 实现的层由 `include/layer/dl_layer_base.hpp` 中的基础层 **Layer** 派生而来。Layer 类只有一个成员变量，即名称 `name`。如果没有用到 `name`，可以不必定制 Layer 类的派生层，但为了保持代码一致，我们推荐派生。

本文档中的示例不可运行，仅供参考。如需可运行的示例，请参考 `include/layer` 文件夹中的头文件，其中包括 Conv2D、DepthwiseConv2D、Concat2D 等层。

由于层的输入和输出都是张量，**请务必阅读张量，了解常量的相关内容。**

下面开始定制层吧！

3.4.1 步骤 1：从 Layer 类派生层

从 Layer 类派生一个新层（示例中命名为 MyLayer），并根据要求定义成员变量、构造函数和析构函数。不要忘记初始化基类的构造函数。

```
class MyLayer : public Layer
{
private:
    /* private member variables */
public:
    /* public member variables */
    Tensor<int16_t> output; /*<! output of this layer */

    MyLayer(/* arguments */) : Layer(name)
    {
        // initialize anything frozen
    }

    ~MyLayer()
    {
        // destroy
    }
};
```

3.4.2 步骤 2：实现 build()

通常一层会有一个或多个输入和一个输出。build() 现有如下作用：

- **更新输出形状：**
输出形状由输入形状决定，有时也受系数形状的影响。比如，Conv2D 的输出形状由输入形状、过滤器形状、步幅和扩张决定，但输入形状可能会变化。一旦输入形状改变，输出形状也应有相应改变。build() 的第一个作用是根据输入形状更新输出形状。
- **更新输入填充：**
Conv2D、DepthwiseConv2D 等二维卷积层中，输入张量可能需要填充。正如输出形状一样，输入填充也由输入形状决定，有时受系数形状影响。比如，Conv2D 层的输入填充由输入形状、过滤器形状、步幅、扩张和填充类型决定。build() 的第二个作用是根据待填充输入张量的形状更新输入填充。

build() 不仅限于以上两个作用。**所有根据输入所做的更新都可由 build() 实现。**

```
class MyLayer : public Layer
{
    // ellipsis member variables
    // ellipsis constructor and destructor
}
```

(下页继续)

(续上页)

```
void build(Tensor<int16_t> &input)
{
    /* get output_shape according to input shape and other configuration */
    this->output.set_shape(output_shape); // update output_shape

    /* get padding according to input shape and other configuration */
    input.set_padding(this->padding);
}
};
```

3.4.3 步骤 3: 实现 call()

在 call() 中实现层推理。请注意:

- 在 [include/typedef/dl_variable.hpp](#) 中, Tensor.apply_element()、Tensor.malloc_element() 或 Tensor.calloc_element() 均可为 output.element 分配存储空间;
- [张量](#) 中描述的张量维度顺序, 因为输入和输出均为 [include/typedef/dl_variable.hpp](#)。

```
class MyLayer : public Layer
{
    // ellipsis member variables
    // ellipsis constructor and destructor
    // ellipsis build(...)

    Tensor<feature_t> &call(Tensor<int16_t> &input, /* other arguments */)
    {
        this->output.calloc_element(); // calloc memory for output.element

        /* implement operation */

        return this->output;
    }
};
```

Chapter 4

工具

4.1 量化工具包

4.1.1 量化工具包

量化工具包能够帮助您量化模型，使用乐鑫芯片进行推理。该工具包以开源的 AI 模型格式 [ONNX](#) 运行。该工具包包括三个独立的工具：

- [优化器](#)，用于优化计算图
- [校准器](#)，用于训练后量化，不需要重新训练
- [评估器](#)，用于评估量化后模型的性能

本文档介绍了每个工具的规范。API 介绍请参阅[量化工具包 API](#)。

请确保使用工具包前，您已将模型转换为 ONNX 格式，相关资料请参考[资源](#)。

优化器

计算图优化器 [optimizer.py](#) 可以通过移除多余节点、简化模型结构、模型融合等方式提高模型性能。该优化器基于 [ONNX 优化器](#) 的 [优化传递 \(pass\)](#)，以及我们额外增加的传递。

在量化前开启计算图融合很重要，尤其是对批量归一化 (batch normalization) 的融合。因此我们推荐您在使用校准器和评估器之前，先用优化器优化模型。您可以用 [Netron](#) 查看模型结构。

Python API 示例

```
// load your ONNX model from given path
model_proto = onnx.load('mnsit.onnx')

// fuse batch normalization layers and convolution layers, and fuse biases and
↪convolution layers
model_proto = onnxoptimizer.optimize(model_proto, ['fuse_bn_into_conv', 'fuse_add_
↪bias_into_conv'])

// set input batch size as dynamic
optimized_model = convert_model_batch_to_dynamic(model_proto)
```

(下页继续)

(续上页)

```
// save optimized model to given path
optimized_model_path = 'mnist_optimized.onnx'
onnx.save(new_model, optimized_model_path)
```

校准器

校准器可量化浮点模型，使之符合在乐鑫芯片上进行推理的要求。有关校准器支持的量化形式，请查看[量化规范](#)。

要将一个 32 位浮点 (FP32) 模型转换为一个 8 位整数 (int8) 或 16 位整数 (int16) 模型，工作流程如下：

- 准备 FP32 模型
- 准备校准数据集
- 配置量化
- 获取量化参数

FP32 模型 准备的 FP32 模型必须与 ESP-DL 库适配。如果模型中有库中不支持的操作，校准器会不接受并生成错误信息。

模型的适配性可在获取量化参数时检查，也可提前通过调用 `check_model` 检查。

模型的输入应当是一个归一化后的数据。若您的归一化过程包含在模型的计算图中，为保证量化性能，请删除图中的相关节点并提前做好归一化。

校准数据集 选择适当的校准数据集对量化来说很重要。一个好的校准数据集应具有代表性。您可尝试不同的校准数据集，比较使用不同参数量化后的模型性能。

量化配置 校准器支持 int8 和 int16 量化。int8 和 int16 各自的配置如下：

int8:

- granularity: 'per-tensor'、'per-channel'
- calibration_method: 'entropy'、'minmax'

int16:

- granularity: 'per-tensor'
- calibration_method: 'minmax'

量化参数 如[量化规范](#)所述，ESP-DL 中 8 位或 16 位量化使用以下公式近似表示浮点值：

```
real_value = int_value * 2^exponent
```

其中 2^{exponent} 为尺度。

返回的量化表列出的是模型中所有数据的量化尺度，这些数据包括：

- 常量：权重、偏差和激活函数；
- 变量：张量，如中间层（激活函数）的输入和输出。

Python API 示例

```
// load your ONNX model from given path
model_proto = onnx.load(optimized_model_path)

// initialize an calibrator to quantize the optimized MNIST model to an int8 model.
↳ per channel using entropy method
calib = Calibrator('int8', 'per-channel', 'entropy')
```

(下页继续)

(续上页)

```
// set ONNX Runtime execution provider to CPU
calib.set_providers(['CpuExecutionProvider'])

// use calib_dataset as the calibration dataset, and save quantization parameters.
↳to the pickle file
pickle_file_path = 'mnist_calib.pickle'
calib.generate_quantization_table(model_proto, calib_dataset, pickle_file_path)

// export to quantized coefficient to cpp/hpp file for deploying on ESP SoCs
calib.export_coefficient_to_cpp(model_proto, pickle_file_path, 'esp32s3', '.',
↳'mnist_coefficient', True)
```

评估器

评估器用于模拟乐鑫芯片的量化解决方案，帮助评估量化后模型的性能。

如果模型中有不支持的操作，校准器会不接受并生成错误信息。

如果量化后模型的性能无法满足需求，可考虑量化感知训练。

Python API 示例

```
// initialize an evaluator to generate an MNIST using int8 per-channel.
↳quantization model running on ESP32-S3 SoC
eva = Evaluator('int8', 'per-channel', 'esp32s3')

// use quantization parameters in the pickle file to generate the int8 model
eva.generate_quantized_model(model_proto, pickle_file_path)

// return results in floating-point values
outputs = eva.evaluate_quantized_model(test_images, to_float = True)
res = np.argmax(outputs[0])
```

示例

量化、评估 MNIST 模型的完整代码示例，请参考 [example.py](#)。

将 TensorFlow MNIST 模型转换为 ONNX 模型的代码示例，请参考 [mnist_tf.py](#)。

将 MXNet MNIST 模型转换为 ONNX 模型的代码示例，请参考 [mnist_mxnet.py](#)。

将 PyTorch MNIST 模型转换为 ONNX 模型的代码示例，请参考 [mnist_pytorch.py](#)。

资源

下列工具可帮助您将模型转换为 ONNX 格式。

- TensorFlow、Keras 和 TFLite 转换为 ONNX: [tf2onnx](#)
- MXNet 转换为 ONNX: [MXNet-ONNX](#)
- PyTorch 转换为 ONNX: [torch.onnx](#)

环境要求：

- Python == 3.7
- [Numba](#) == 0.53.1
- [ONNX](#) == 1.9.0
- [ONNX Runtime](#) == 1.7.0
- [ONNX Optimizer](#) == 0.2.6

您可以使用 `requirement.txt` 来安装相关 Python 依赖包：

```
pip install -r requirement.txt
```

4.1.2 量化规范

训练后量化 将浮点模型转换为定点模型。这种转换技术可以缩减模型大小，降低 CPU 和硬件加速器延迟，同时不会降低准确度。

如 ESP32-S3 芯片，存储空间相对有限，在 240 MHz 的情况下每秒乘加累计运算次数 (MACs) 仅达 75 亿次。在这样的芯片上必须要用量化后的模型做推理。您可使用我们提供的 **量化工具包** 量化浮点模型，或根据 [convert.py 使用说明](#) 中的步骤部署定点模型。

全整数量化

模型中的所有数据都需要量化为 8 位或 16 位整数。所有数据包括

- 常量：权重、偏差和激活函数
- 变量：张量，如中间层（激活函数）的输入和输出

8 位或 16 位量化使用以下公式近似表示浮点值：

```
real\_value = int\_value * 2^{\ exponent}
```

有符号整数 8 位量化的 `int_value` 代表一个 **int8** 的有符号数，其范围是 [-128, 127]。16 位量化的 `int_value` 代表一个 **int16** 的有符号数，其范围是 [-32768, 32767]。

对称 所有量化后的数据都是**对称**的，也就是说没有零点（偏差），因此可以节省将零点与其他值相乘的运算时间。

粒度 **按张量（又名按层）量化**的意思是每个完整张量只有一个指数位，该张量内的所有值都按照该指数位量化。

按通道量化的意思是卷积核的每个通道对应着不同的指数位。

与按张量量化相比，按通道量化通常对于部分模型来说精确度会更高，但也会更耗时间。您可使用 **量化工具包** 中的 **评估器**模拟在芯片上进行量化推理的性能，之后再决定使用哪种量化形式。

16 位量化为确保更快的运算速度，目前仅支持按张量量化。8 位量化支持按张量和按通道两种形式，可让您在性能和速度之间折中。

量化算子规范

以下是 API 的量化要求：

```
Add2D
Input 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
Input 1:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
Output 0:
  data_type : int8 / int16
```

(下页继续)

(续上页)

```

    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor

AvgPool2D
Input 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor

Concat
Input ...:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
restriction: Inputs and output must have the same exponent

Conv2D
Input 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
Input 1 (Weight):
    data_type  : int8 / int16
    range      : [-127, 127] / [-32767, 32767]
    granularity: {per-channel / per-tensor for int8} / {per-tensor for int16}
Input 2 (Bias):
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
    restriction: exponent = output_exponent
Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor

DepthwiseConv2D
Input 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
Input 1 (Weight):
    data_type  : int8 / int16
    range      : [-127, 127] / [-32767, 32767]
    granularity: {per-channel / per-tensor for int8} / {per-tensor for int16}
Input 2 (Bias):
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
    restriction: exponent = output_exponent
Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor

```

(下页继续)

(续上页)

ExpandDims

```

Input 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
Output 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
restriction: Input and output must have the same exponent

```

Flatten

```

Input 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
Output 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
restriction: Input and output must have the same exponent

```

FullyConnected

```

Input 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
Input 1 (Weight):
  data_type : int8 / int16
  range     : [-127, 127] / [-32767, 32767]
  granularity: {per-channel / per-tensor for int8} / {per-tensor for int16}
Input 2 (Bias):
  data_type : int8 / int16
  range     : {[-32768, 32767] for int8 per-channel / [-128, 127] for int8 per-
↪ tensor} / {[-32768, 32767] for int16}
  granularity: {per-channel / per-tensor for int8} / {per-tensor for int16}
  restriction: {exponent = input_exponent + weight_exponent + 4 for per-channel /
↪ exponent = output_exponent for per-tensor}
Output 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor

```

GlobalAveragePool2D

```

Input 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
Output 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor

```

GlobalMaxPool2D

```

Input 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
Output 0:
  data_type : int8 / int16

```

(下页继续)

(续上页)

```

    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
    restriction: Input and output must have the same exponent

```

LeakyReLU

```

Input 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
Input 1 (Alpha):
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
Output 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
  restriction: Input and output must have the same exponent

```

Max2D

```

Input 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
Output 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
  restriction: Input and output must have the same exponent

```

MaxPool2D

```

Input 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
Output 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
  restriction: Input and output must have the same exponent

```

Min2D

```

Input 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
Output 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
  restriction: Input and output must have the same exponent

```

Mul2D

```

Input 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
Input 1:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
Output 0:

```

(下页继续)

(续上页)

```

data_type : int8 / int16
range     : [-128, 127] / [-32768, 32767]
granularity: per-tensor

```

PReLU

```

Input 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
Input 1 (Alpha):
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
Output 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
restriction: Input and output must have the same exponent

```

ReLU

```

Input 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
Output 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
restriction: Input and output must have the same exponent

```

Reshape

```

Input 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
Output 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
restriction: Input and output must have the same exponent

```

Squeeze

```

Input 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
Output 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
restriction: Input and output must have the same exponent

```

Sub2D

```

Input 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
Input 1:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
Output 0:

```

(下页继续)

(续上页)

```

data_type : int8 / int16
range     : [-128, 127] / [-32768, 32767]
granularity: per-tensor

Transpose
Input 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
Output 0:
  data_type : int8 / int16
  range     : [-128, 127] / [-32768, 32767]
  granularity: per-tensor
restriction: Input and output must have the same exponent

```

4.1.3 量化工具包 API

校准器类

初始化

```
Calibrator(quantization_bit, granularity='per-tensor', calib_method='minmax')
```

实参

- **quantization_bit** (*string*):
 - ‘int8’ 代表全 8 位整数量化。
 - ‘int16’ 代表全 16 位整数量化。
- **granularity** (*string*):
 - 若 granularity = ‘per-tensor’ (默认), 则整个张量只有一个指数。
 - 若 granularity = ‘per-channel’, 则卷积层的每个通道都有一个指数。
- **calib_method** (*string*):
 - 若 calib_method = ‘minmax’ (默认), 则阈值得自校准数据集每层输出的最小值和最大值。
 - 若 calib_method = ‘entropy’, 则阈值得自 KL 散度。

check_model 方法

```
Calibrator.check_model(model_proto)
```

检查模型的适配性。

实参

- **model_proto** (*ModelProto*): 一个 FP32 ONNX 模型。

返回值

- **-1**: 模型不适配。

set_method 方法

```
Calibrator.set_method(granularity, calib_method)
```

配置量化。

实参

- **granularity** (*string*):

- 若 `granularity = 'per-tensor'`，则整个张量只有一个指数。
- 若 `granularity = 'per-channel'`，则卷积层的每个通道都有一个指数。
- **calib_method** (*string*):
 - 若 `calib_method = 'minmax'`，则阈值得自校准数据集每层输出的最小值和最大值。
 - 若 `calib_method = 'entropy'`，则阈值得自 KL 散度。

set_providers 方法

```
Calibrator.set_providers(providers)
```

配置 ONNX Runtime 的运行环境提供方。

实参

- **providers** (*list of strings*): 列表中的运行环境提供方，如 'CPUExecutionProvider'、'CUDAExecutionProvider'。

generate_quantization_table 方法

```
Calibrator.generate_quantization_table(model_proto, calib_dataset, pickle_file_path)
```

生成量化表。

实参

- **model_proto** (*ModelProto*): 一个 FP32 ONNX 模型。
- **calib_dataset** (*ndarray*): 用于计算阈值的校准数据集。数据集越大，生成量化表的时间越长。
- **pickle_file_path** (*string*): 存储量化参数的 pickle 文件路径。

export_coefficient_to_cpp 方法

```
Calibrator.export_coefficient_to_cpp(model_proto, pickle_file_path, target_chip, output_path, file_name, print_model_info=False)
```

导出模型量化后的权重等系数以用于部署在乐鑫芯片上。

Arguments

- **model_proto** (*ModelProto*): 一个 FP32 ONNX 模型。
- **pickle_file_path** (*string*): 存储量化参数的 pickle 文件路径。
- **target_chip** (*string*): 目前支持 'esp32'、'esp32s2'、'esp32c3'、'esp32s3'。
- **output_path** (*string*): 存储输出文件的路径。
- **file_name** (*string*): 输出文件的名称。
- **print_model_info** (*bool*):
 - False (default): 不打印任何信息。
 - True: 打印模型的相关信息。

评估器类

初始化

```
Evaluator(quantization_bit, granularity, target_chip)
```

实参

- **quantization_bit** (*string*):
 - 'int8' 代表全 8 位整数量化。
 - 'int16' 代表全 16 位整数量化。
- **granularity** (*string*):
 - 若 `granularity = 'per-tensor'`，则整个张量只有一个指数。
 - 若 `granularity = 'per-channel'`，则卷积层的每个通道都有一个指数。

- **target_chip** (*string*): 默认是 ‘esp32s3’。

check_model 方法

```
Evaluator.check_model(model_proto)
```

检查模型的适配性。

实参

- **model_proto** (*ModelProto*): 一个 FP32 ONNX 模型。

Return

- **-1**: 模型不适配。

set_target_chip 方法

```
Evaluator.set_target_chip(target_chip)
```

配置模拟芯片环境。

实参

- **target_chip** (*string*): 目前仅支持 ‘esp32s3’。

set_providers 方法

```
Evaluator.set_providers(providers)
```

配置 ONNX Runtime 的运行环境提供方。

实参

- **providers** (*list of strings*): [列表](#) 中的运行环境提供方，如 ‘CPUExecutionProvider’、‘CUDAExecutionProvider’。

generate_quantized_model 方法

```
Evaluator.generate_quantized_model(model_proto, pickle_file_path)
```

生成量化后的模型。

实参

- **model_proto** (*ModelProto*): 一个 FP32 ONNX 模型。
- **pickle_file_path** (*string*): 存储 FP32 ONNX 模型所有量化参数的 pickle 文件路径。该 pickle 文件必须包含模型计算图所有输入和输出节点的量化参数。

evaluate_quantized_model 方法

```
Evaluator.evaluate_quantized_model(batch_fp_input, to_float=False)
```

获取量化模型的输出。

实参

- **batch_fp_input** (*ndarray*): 批量浮点输入。
- **to_float** (*bool*):
 - False (默认): 直接返回输出。
 - True: 输出转换为浮点值。

返回值

outputs 和 output_names 组成的元组:

- **outputs** (*list of ndarray*): 量化模型的输出。
- **output_names** (*list of strings*): 输出名称。

4.2 转换工具

4.2.1 convert.py 使用说明

[tools/convert_tool/convert.py](#) 脚本将.npy 文件中的浮点系数量化为 C/C++ 代码，存储到.cpp 和.hpp 文件中。该脚本还会转换系数的元素顺序，从而加速操作。

convert.py 根据 config.json 文件运行。该文件是模型必要的配置文件。有关如何写 config.json file 文件，请参考[config.json 配置规范](#)。

注意，convert.py 需在 Python 3.7 或更高版本中运行。

实参描述

运行 convert.py 时需填写以下实参：

实参	值
-t -target_chip	esp32 esp32s2 esp32s3 esp32c3
-i -input_root	npv 文件和 json 文件所在目录
-j -json_file_name	json 文件名 (默认: config.json)
-n -name	输出文件名
-o -output_root	输出文件所在目录
-q -quant	量化颗粒度 0(默认) 代表按层量化, 1 代表按通道量化

示例

假设：

- convert.py 的相对路径为 **./convert.py**
- 目标芯片为 **esp32s3**
- npv 文件和 config.json 文件在 **./my_input_directory** 目录中
- 输出文件名为 **my_coefficient**
- 输出文件将存放在 **./my_output_directory** 目录

运行如下命令：

```
python ./convert.py -t esp32s3 -i ./my_input_directory -n my_coefficient -o ./my_
↪output_directory
```

之后将生成 my_coefficient.cpp 和 my_coefficient.hpp 文件，存放在 ./my_output_directory 目录中。

4.2.2 config.json 配置规范

config.json 用于保存 coefficient.npy 文件中浮点数的量化配置。

配置

config.json 中的每一项代表一层的配置。以下列代码为例：

```
{
  "l1": {"/ * the configuration of layer l1 */},
  "l2": {"/ * the configuration of layer l2 */},
  "l3": {"/ * the configuration of layer l3 */},
  ...
}
```

每项的键(key)是层名。转换工具 convert.py 根据层名搜索相应的 .npv 文件。比如，层名为“l1”，转换工具则会在“l1_filter.npv”文件中搜索 l1 的过滤器系数。**config.json 中的层名需和.npv 文件名中的层名保持一致。**

每项的值是层的配置。请填写表 1 中列出的层配置实参：

表 1: 表 1：层配置实参

键	类型	值
“operation”	string	<ul style="list-style-type: none"> “conv2d” “depthwise_conv2d” “fully_connected”
“feature_type”	string	<ul style="list-style-type: none"> “s16” 代表 16 位整数量化，element_width 为 16 “s8” 代表 8 位整数量化，element_width 为 8
“filter_exponent”	integer	<ul style="list-style-type: none"> 若填写，则过滤器根据公式量化：$\text{value_float} = \text{value_int} * 2^{\text{指数}}$¹ 若空置²，则指数为 $\log_2(\max(\text{abs}(\text{value_float})) / 2^{(\text{element_width} - 1)})$，过滤器会根据公式量化：$\text{value_float} = \text{value_int} * 2^{\text{指数}}$
“bias”	string	<ul style="list-style-type: none"> “True” 代表添加偏差 “False” 和空置代表不使用偏差
“output_exponent”	integer	<p>输出和偏差根据公式量化：$\text{value_float} = \text{value_int} * 2^{\text{指数}}$。</p> <p>目前，“output_exponent” 仅在转换偏差系数时有效。当使用按层量化时，必须提供“output_exponent”。如果特定层没有偏差或使用按通道量化时，“output_exponent” 可空置。</p>
“input_exponent”	integer	<p>当使用按通道量化时，偏差的指数位与输入和过滤器的指数位相关。</p> <p>如果有偏差时必须提供“input_exponent” 用于转换偏差系数。如果特定层没有偏差或使用按层量化时，“input_exponent” 可空置。</p>
“activation”	dict	<ul style="list-style-type: none"> 若填写，详见表 2 若空置，则不使用激活函数

¹ 指数：量化时底数相乘的次数。为能更好理解，请阅读[量化规范](#)。

² 空置：不填写特定实参。

表 2: 表 2: 激活函数配置实参

键	类型	值
"type"	string	<ul style="list-style-type: none"> • "ReLU" • "LeakyReLU" • "PReLU"
"exponent"	integer	<ul style="list-style-type: none"> • 若填写, 则激活函数根据公式量化: $\text{value_float} = \text{value_int} * 2^{\text{指数}}$ • 若空置, 则指数为 $\log_2(\max(\text{abs}(\text{value_float})) / 2^{(\text{element_width} - 1)})$

示例

假设有一个一层模型:

1. 使用 int16 按层量化:

- 层名: "mylayer"
- operation: Conv2D(input, filter) + bias
- output_exponent: -10
- feature_type: s16, 即 16 位整数量化
- 激活函数类型: PReLU

config.json 应写作:

```
{
  "mylayer": {
    "operation": "conv2d",
    "feature_type": "s16",
    "bias": "True",
    "output_exponent": -10,
    "activation": {
      "type": "PReLU"
    }
  }
}
```

"filter_exponent" 和 "activation" 的 "exponent" 空置。必须提供 "output_exponent" 用于转化该层的 bias

2. 使用 int8 按层量化:

- 层名: "mylayer"
- operation: Conv2D(input, filter) + bias
- output_exponent: -7, 该卷积层结果的指数位
- feature_type: s8
- 激活函数类型: PReLU

config.json 应写作:

```
{
  "mylayer": {
    "operation": "conv2d",
    "feature_type": "s8",
    "bias": "True",
    "output_exponent": -7,
    "activation": {
```

(下页继续)

(续上页)

```

        "type": "PReLU"
    }
}

```

必须提供 “output_exponent” 用于转化该层的 bias

3. 使用 int8 按通道量化:

- 层名: “mylayer”
- operation: Conv2D(input, filter) + bias
- input_exponent: -7, 该卷积层输入的指数位
- feature_type: s8
- 激活函数类型: PReLU

config.json 应写作:

```

{
  "mylayer": {
    "operation": "conv2d",
    "feature_type": "s8",
    "bias": "True",
    "input_exponent": -7,
    "activation": {
      "type": "PReLU"
    }
  }
}

```

必须提供 “input_exponent” 用于转化该层的 bias

同时, mylayer_filter.npy、mylayer_bias.npy 和 mylayer_activation.npy 需要准备好。

4.3 图片工具

ESP-DL 是不包含外设驱动的仓库, 在编写模型库的 [示例](#) 时使用了数组保存像素值来表示图片, 运行结果只能显示在终端中。为使您更充分地体验 ESP-DL, 我们提供了以下工具用于图片的转换和显示。

4.3.1 图片转换工具 convert_to_u8.py

该转换工具可将自定义图片转换成 C/C++ 的数组形式。配置说明如下:

参数	类型	值
-i -input	string	输入图片的路径
-o -output	string	输出文件的路径

示例:

假设,

- 自定义图片路径为 my_album/my_image.png
- 输出文件存放至人脸检测项目文件夹 esp-dl/examples/human_face_detect/main 中

则,

```
python convert_to_u8.py -i my_album/my_image.png -o ESP-DL/examples/human_face_
↪ detect/main/image.hpp
```

注意：以上代码只作示例说明，并非有效代码。

4.3.2 显示工具 `display_image.py`

该显示工具可在图片上绘制用于检测的框和点。配置说明如下：

参数	类型	值
<code>-i -image</code>	string	图片路径
<code>-b -box</code>	string	输入依照格式 (x1, y1, x2, y2)，其中 (x1, y1) 表示框的左上角坐标，(x2, y2) 表示框的右下角坐标 空置 ¹ ：不绘制框
<code>-k -key-points</code>	string	输入依照格式 (x1, y1, x2, y2, ..., xn, yn)，其中每对 (x, y) 表示一个点 空置：不绘制点

示例：

假设，

- 图片路径为 `my_album/my_image.jpg`
- 框的左上角坐标：(137, 75)
- 框的右下角坐标：(246, 215)
- 点 1 的坐标：(157, 131)
- 点 2 的坐标：(158, 177)
- 点 3 的坐标：(170, 163)

则，

```
python display_image.py -i my_album/my_image.jpg -b "(137, 75, 246, 215)" -k "(157,
↪ 131, 158, 177, 170, 163)"
```

注意：以上代码只作示例说明，并非有效代码。

¹ 空置：不填写特定实参。

Chapter 5

性能

5.1 猫脸检测延迟

SoC	Latency
ESP32	149,765 us
ESP32-S2	416,590 us
ESP32-S3	18,909 us

5.2 人脸检测延迟

SoC	TWO_STAGE = 1	TWO_STAGE = 0
ESP32	415,246 us	154,687 us
ESP32-S2	1,052,363 us	309,159 us
ESP32-S3	56,303 us	16,614 us

TWO_STAGE 宏可定义目标检测的算法：

- TWO_STAGE = 1：检测器为 two-stage（两阶段），检测结果更加精确（支持人脸关键点），但速度较慢。
- TWO_STAGE = 0：检测器为 one-stage（单阶段），检测结果精确度稍差（不支持人脸关键点），但速度较快。

5.3 人脸识别延迟

SoC	8-bit	16-bit
ESP32	13,301 ms	5,041 ms
ESP32-S3	287 ms	554 ms

Chapter 6

词汇表

张量 张量是矩阵向更高维度的泛化。也就是说，张量可以是：

- 0 维，表示为标量
- 1 维，表示为向量
- 2 维，表示为矩阵
- 难以想象的多维结构

维数和每个维度的大小即为张量的形状。ESP-DL 的主要数据结构就是张量。一个层的所有输入和输出均为张量。

二维操作中，层的输入张量和输出张量均是三维。张量的维度顺序固定，按照 [高度，宽度，通道] 的顺序排序。

假设张量的形状是 [5, 3, 4]，则张量中的元素应如下排列：

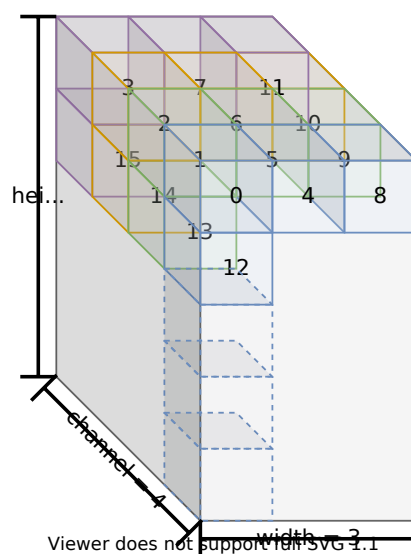


图 1: 3D 张量

过滤器、偏差和激活函数 与张量不同，过滤器、偏差和激活函数无需填充。这三个‘元素’的顺序是灵活的，可根据特定操作调整。

更多细节，可参考 [include/typedef/dl_constant.hpp](#) 或 API 文档。

索引



张量, [43](#)



过滤器、偏差和激活函数, [43](#)