

密级状态：绝密() 秘密() 内部() 公开(√)

RKNN-Toolkit 用户使用指南

(技术部，图形计算平台中心)

文件状态： [] 正在修改 [√] 正式发布	当前版本：	V1.7.1
	作 者：	饶洪
	完成日期：	2021-11-20
	审 核：	熊伟
	完成日期：	2021-11-20

瑞芯微电子股份有限公司

Rockchip Electronics Co., Ltd

(版本所有,翻版必究)

更新记录

版本	修改人	修改日期	修改说明	核定人
V0.1	杨华聪	2018-08-25	初始版本	熊伟
V0.9.1	饶洪	2018-09-29	增加 RKNN-Toolkit 工具使用说明,包括主要功能、系统依赖、安装方式、使用场景及各 API 接口的详细说明。	熊伟
V0.9.2	卓鸿添	2018-10-12	优化性能预估方式	熊伟
V0.9.3	杨华聪	2018-10-24	添加连接开发板硬件说明	熊伟
V0.9.4	杨华聪	2018-11-03	添加 docker 镜像使用说明	熊伟
V0.9.5	饶洪	2018-11-19	1. 添加 npy 文件作为量化校正数据的使用说明; 2. build 接口 pre_compile 参数说明; 3. 完善 config 接口 reorder_channel 参数的使用说明	熊伟
V0.9.6	饶洪	2018-11-24	1. 新增接口 get_perf_detail_on_hardware 和 get_run_duration 的使用说明; 2. 更新 RKNN 初始化接口使用说明。	熊伟
V0.9.7	饶洪	2018-12-29	1. 接口优化: 删除 get_run_duration、get_perf_detail_on_hardware 使用说明, 重写 eval_perf 接口使用说明; 2. 重写 RKNN()接口使用说明; 3. 新增接口 init_runtime 的使用说明。	熊伟
V0.9.7.1	饶洪	2019-01-11	1. 解决多次调用 inference 后程序可能挂起的 BUG; 2. 接口调整: init_runtime 时不需要再指定 host, 工具会自动判断。	熊伟
V0.9.8	饶洪	2019-01-30	1. 新增 verbose 选项, 开启后可以打印模型加载、构建等阶段的日志信息, 并写到指定文件中。	熊伟

版本	修改人	修改日期	修改说明	核定人
V0.9.9	饶洪	2019-03-06	<ol style="list-style-type: none"> 1. 新增 eval_memory 接口，查看模型运行时的内存占用情况。 2. 优化 inference 接口；优化错误信息提示。 3. 新增 get_sdk_version 接口使用说明 	熊伟
V1.0.0	饶洪	2019-05-08	<ol style="list-style-type: none"> 1. 初始化运行时环境接口新增异步模式。 2. 推理接口新增输入透传模式。 3. 新功能：混合量化。 4. 优化 pre-compile 模型的加载时间。新版本工具生成的预编译模型无法在 NPU 驱动版本号小于 0.9.6 的设备上运行；旧版本生成的预编译模型也无法在新版本驱动上运行。 5. 调整模型推理结果的排列顺序：在 1.0.0 以前，如果原始模型输出的结果是按"NHWC"排列（如 TensorFlow），则工具会把结果转成"NCHW"；从 1.0.0 版本开始，将不做这个转换，而是保持跟原始模型的排列一致。 	熊伟
V1.1.0	饶洪	2019-06-28	<ol style="list-style-type: none"> 1. 新增对 TB-RK1808 AI 计算棒的支持。 2. 新增接口 list_devices，用来查询已连接设备信息。 3. 支持使用 Python 3.5 的 ARM64 Linux 平台。 4. 支持 Windows / Mac OS X 操作系统。 	熊伟
V1.2.0	饶洪	2019-08-21	<ol style="list-style-type: none"> 1. 新增对多 input 模型的支持。 2. 新增批量推理功能。 3. 新增模型分段功能，实现多模型同时运行。 4. 新增自定义算子功能。 	熊伟

版本	修改人	修改日期	修改说明	核定人
V1.2.1	饶洪	2019-09-26	<ol style="list-style-type: none"> 1. 解决使用 Python logging 模块报错的问题。 2. 解决多线程推理时在获取输出阶段发生段错误的问题。 3. 修复 dataset.txt 里所有文件路径都非法时构建量化模型会卡死的问题。 4. 调整 config 接口 batch_size 和 epochs 参数的默认值，修复 dataset.txt 中的数据未被充分利用的问题。 5. 新功能，load_rknn 接口支持直接加载 NPU 中的 rknn 模型。 	熊伟
V1.3.0	饶洪	2019-12-23	<ol style="list-style-type: none"> 1. 解决创建 RKNN 对象时间过长的的问题。 2. 新增加载 PyTorch 模型功能。 3. 新增加载 MXNet 模型功能。 4. 新增对 4 通道输入的支持。 5. 新增误差分析功能。 6. 新增可视化功能。 7. 新增模型优化等级功能。 8. 优化混合量化功能。 	熊伟

版本	修改人	修改日期	修改说明	核定人
V1.3.2	饶洪	2020-04-03	<ol style="list-style-type: none"> 1. 增加对 RV1109、RV1126 的支持。 2. 完善 eval_perf 功能，不再需要填输入参数。 3. TensorFlow: 增加对 reducemax 的支持；完善对 dilated convolution 的支持。 TFLite: 增加对 dilated convolution 的支持。 Caffe: 增加对 CRNN 的支持。 ONNX: 增加对 Gather 和 Cast 的支持；完善对 avg_pool 的支持。 PyTorch: 增加对 upsample_nearest2d, contiguous, softmax, permute, leaky_relu, prelu, log, deconv 和 sub 的支持；完善对 Reshape, Constant 的支持。 MXNet: 增加对 Crop, UpSampling, SoftmaxActivation, _minus_scalar, log 的支持。 RKNN: 完善对 reshape, concat, split 的支持。 4. 修复已知 bug。 	熊伟

版本	修改人	修改日期	修改说明	核定人
V1.4.0	饶洪	2020-08-13	<ol style="list-style-type: none"> 1. 新功能：增加逐层量化分析子功能；输入预处理支持多个 std_value；支持从开发板导出预编译模型。 2. 功能优化：优化 channel_mean_value 参数，改成 mean_values/std_values；移除 load_tensorflow 接口中的 mean_values 和 std_values；可视化完善对多输入的支持，增加对 RK1806/RV1109/RV1126 的支持；精度分析功能增加非归一化的余弦距离和欧式距离。 3. TensorFlow：增加对 dense 子图的支持。 TFLite：增加对 split_v 的支持；完善对 pad 的支持。 ONNX：完善对 prelu / deconvolution / avg_pool / clip 的支持。 PyTorch：增加对 pixel_shuffle, unsqueeze, sum, select, hardtanh, elu, slice, squeeze, exp, relu6, threshold_, matmul, exp, pad 的支持；完善对 adaptive_avg_pool2d, upsample_bilinear, relu6 的支持。 MXNet：完善对 fc 的支持。 Darknet：增加对 mish 的支持；完善对 route 的支持。 4. 修复已知 bug。 	熊伟

版本	修改人	修改日期	修改说明	核定人
V1.6.0	饶洪 洪启飞	2020-12-31	<ol style="list-style-type: none"> 1. 新功能：支持 Keras 框架，并且支持 TF 2.0 导出的 h5 模型；支持 PyTorch 1.6.0；支持 ONNX 1.6.0；增加模型加密功能；支持通过命令行输出已连接设备；离线预编译支持多输入模型。 2. 功能优化：优化精度分析功能，完整模型比对时，遇到 Conv+Relu 等情况时，不再需要跳层比较；优化 RKNN 模型预处理，提高模型推理性能；优化性能分析功能，打印详情时，精简每层 OP；模型分段功能限制在 RK1806/RK1808/RV1109/RV1126 芯片范围；docker 镜像系统升级为 Ubuntu 18.04，Python 升级到 3.6。 3. 完善对各框架 OP 的支持。 4. 修复已知问题。 	熊伟
V1.6.1	饶洪	2021-05-21	<ol style="list-style-type: none"> 1. 新功能：增加量化参数优化方法 MMSE；支持导出每一层 weight / bias 和输出数据的分布直方图，用于分析精度问题。 2. 功能优化：完善 mean_values / std_values 设置不当时的提示信息。 3. 完善对各框架 OP 的支持。 4. 修复已知 Bug。 	熊伟

版本	修改人	修改日期	修改说明	核定人
V1.7.0	饶洪	2021-08-08	<ol style="list-style-type: none"> 1. 新功能：支持 ONNX 量化模型（对应 onnxruntime 版本为 1.5.2）；加载 ONNX 模型时支持指定输入、输出节点。 2. 功能优化：完善混合量化接口；完善 MMSE 量化参数优化功能；完善精度分析接口；完善可视化相关功能。 3. 完善各框架 op 的支持。 4. 增加推理接口 pass_through 参数的示例；增加精度分析接口使用示例；优化 yolov3 示例。 5. 修复已知 bug。 	熊伟
V1.7.1	饶洪	2021-11-20	<ol style="list-style-type: none"> 1. 新功能：支持 PyTorch 量化模型。 2. 功能优化：优化精度分析接口的性能；预处理层优化为卷积网络，使用 npu 处理加速；优化日志功能。 3. 完善各框架 op 的支持。 4. 增加 ONNX/PyTorch/TFLite 量化模型转换示例；新增 yolov5 PyTorch 模型转换示例。 5. 修复已知 bug。 	熊伟

目 录

1	概述.....	1
1.1	主要功能说明.....	1
1.2	适用芯片	1
1.3	适用系统	2
1.4	适用的深度学习框架.....	2
2	开发环境搭建.....	4
2.1	参考系统配置.....	4
2.2	系统依赖说明.....	4
2.3	工具安装	6
2.3.1	通过 <i>pip install</i> 命令安装.....	6
2.3.2	使用 <i>Docker</i> 镜像.....	7
2.3.3	常见问题.....	8
2.4	连接 ROCKCHIP NPU 设备	8
2.4.1	通过 <i>USB-OTG</i> 口连接设备.....	9
2.4.2	通过 <i>DEBUG</i> 口连接设备.....	10
2.4.3	确认设备连接正确.....	12
2.4.4	常见问题.....	13
3	使用流程.....	14
3.1	基本使用流程.....	14
3.2	模型转换	14
3.2.1	模型转换流程.....	14
3.2.2	模型量化.....	15
3.2.3	模型转换示例.....	18
3.2.4	常见问题.....	19

3.3	模型评估	19
3.4	模型部署	20
3.4.1	准备工程文件	20
3.4.2	应用程序示例	21
3.4.3	应用程序运行	24
3.4.4	模型预编译	24
3.4.5	模型加密	25
3.4.6	多模型调度	26
3.4.7	应用部署常见问题	27
4	准确性评估	28
4.1	评估方法	28
4.2	准确性问题排查	29
4.2.1	浮点模型精度问题排查步骤	30
4.2.2	量化模型精度问题排查步骤	31
4.3	精度分析	32
4.3.1	功能介绍	32
4.3.2	使用流程	32
4.3.3	输出说明	33
4.4	优化量化参数	35
4.5	混合量化	35
4.5.1	混合量化功能用法	35
4.5.2	混合量化配置文件	35
4.5.3	混合量化使用流程	37
4.6	常见问题	38
5	性能评估	39
5.1	性能评估方法	39

5.2	性能评估示例.....	39
5.3	性能评估结果说明.....	41
5.3.1	模拟器性能评估结果说明.....	41
5.3.2	开发板性能评估结果说明.....	42
5.4	常见性能优化方法.....	43
6	内存评估.....	45
6.1	评估方法	45
6.2	评估示例	45
6.3	内存评估结果说明.....	46
7	API 详细说明.....	47
7.1	RKNN 初始化及对象释放	47
7.2	RKNN 模型配置	48
7.3	模型加载	50
7.3.1	Caffe 模型加载接口.....	50
7.3.2	Darknet 模型加载接口.....	51
7.3.3	Keras 模型加载接口.....	52
7.3.4	MXNet 模型加载接口.....	52
7.3.5	ONNX 模型加载.....	53
7.3.6	PyTorch 模型加载接口.....	54
7.3.7	TensorFlow 模型加载接口.....	55
7.3.8	TensorFlow Lite 模型加载接口.....	56
7.4	构建 RKNN 模型	57
7.5	导出 RKNN 模型	58
7.6	加载 RKNN 模型	59
7.7	初始化运行时环境.....	59
7.8	模型推理	61

7.9	模型性能评估.....	63
7.10	获取内存使用情况.....	64
7.11	混合量化	65
7.11.1	<i>hybrid_quantization_step1</i>	65
7.11.2	<i>hybrid_quantization_step2</i>	65
7.12	量化精度分析.....	68
7.13	注册自定义算子.....	69
7.14	导出预编译模型（在线预编译）	70
7.15	导出分段模型.....	72
7.16	导出加密模型.....	73
7.17	查询 SDK 版本.....	74
7.18	获取设备列表.....	75
7.19	查询模型可运行平台.....	76
8	附录.....	77
8.1	参考文档	77
8.2	问题反馈渠道.....	77

1 概述

1.1 主要功能说明

RKNN-Toolkit 是为用户提供在 PC、Rockchip NPU 平台上进行模型转换、推理和性能评估的开发套件，用户通过该工具提供的 Python 接口可以便捷地完成以下功能：

- 1) 模型转换：支持 Caffe、TensorFlow、TensorFlow Lite、ONNX、Darknet、PyTorch、MXNet 和 Keras 模型转为 RKNN 模型。

从 1.2.0 版本开始支持多输入模型的转换。

从 1.3.0 版本开始支持 PyTorch 和 MXNet 框架。

从 1.6.0 版本开始支持 Keras 框架模型，并支持 TensorFlow 2.0 导出的 H5 模型。

- 2) 量化功能：支持将浮点模型量化为定点模型，目前支持的量化方法为非对称量化（`asymmetric_quantized-u8`），动态定点量化（`dynamic_fixed_point-i8` 和 `dynamic_fixed_point-i16`）。

从 1.0.0 版本开始，RKNN-Toolkit 开始支持混合量化功能，该功能的详细说明请参考第 4.5 章节。

从 1.6.1 版本开始，RKNN-Toolkit 提供量化参数优化算法 MMSE 和 KL 散度。

从 1.7.0 版本开始，支持加载已量化的 ONNX 模型。从 1.7.1 版本开始，支持加载已量化的 PyTorch 模型。

- 3) 模型推理：支持在 PC（Linux x86 平台）上模拟 Rockchip NPU 运行 RKNN 模型并获取推理结果；也支持将 RKNN 模型分发到指定的 NPU 设备上运行推理。
- 4) 性能评估：支持在 PC（Linux x86 平台）上模拟 Rockchip NPU 运行 RKNN 模型，并评估模型性能（包括总耗时和每一层的耗时）；也支持将 RKNN 模型分发到指定 NPU 设备上运行，以评估模型在实际设备上运行时的性能。
- 5) 内存评估：评估模型运行时系统内存的使用情况。支持将 RKNN 模型分发到 NPU 设备中运行，并调用相关接口获取内存使用信息。从 0.9.9 版本开始支持该功能。

-
- 6) 模型预编译：通过预编译技术生成的 RKNN 模型可以减少 NPU 加载模型的时间。通过预编译技术生成的 RKNN 模型只能在 NPU 硬件上运行，不能在模拟器中运行。当前只有 x86_64 Ubuntu 平台支持直接从原始模型生成预编译 RKNN 模型。RKNN-Toolkit 从 0.9.5 版本开始支持模型预编译功能，并在 1.0.0 版本中对预编译方法进行了升级，升级后的预编译模型无法与旧驱动兼容。

从 1.4.0 版本开始，支持通过 NPU 设备将普通 RKNN 模型转为预编译 RKNN 模型，详情请参考 7.14 章节接口 `export_rknn_precompile_model` 的使用说明。

- 7) 模型分段：该功能用于多模型同时运行的场景。将单个模型分成多段在 NPU 上执行，借此来调节多个模型占用 NPU 的时间，避免因为一个模型占用太多 NPU 时间，而使其他模型无法及时执行。RKNN-Toolkit 从 1.2.0 版本开始支持该功能。目前，只有 RK1806/RK1808/RV1109/RV1126 芯片支持该功能，且 NPU 驱动版本要大于 0.9.8。

- 8) 自定义算子功能：当模型含有 RKNN-Toolkit 不支持的算子 (operator)，模型转换将失败。针对这种情况，RKNN Toolkit 提供自定义算子功能，允许用户自行实现相应算子，从而使模型能正常转换和运行。RKNN-Toolkit 从 1.2.0 版本开始支持该功能。自定义算子的使用 and 开发请参考《Rockchip_Developer_Guide_RKNN_Toolkit_Custom_OP_CN》文档。自定义算子目前只支持 TensorFlow 框架。

- 9) 量化精度分析功能：RKNN Toolkit 精度分析功能可以保存浮点模型、量化模型推理时每一层的中间结果，并用欧式距离和余弦距离评估它们的相似度。该功能从 1.3.0 版本开始支持。

1.4.0 版本增加逐层量化精度分析子功能，上一层的浮点结果会被记录并输入到下一个被量化的网络层，避免逐层误差积累，能够更准确的反映每一层自身受量化的影响。

- 10) 可视化功能：该功能以图形界面的形式呈现 RKNN-Toolkit 的各项功能，简化用户操作步骤。允许通过填写表单、点击功能按钮的形式完成模型的转换和推理等功能，无需手动编写脚本。有关可视化功能的具体使用方法请参考

《Rockchip_User_Guide_RKNN_Toolkit_Visualization_CN》文档。1.3.0 版本开始支持该功能。

1.4.0 版本完善了对多输入模型的支持，并且支持 RK1806, RV1109, RV1126 等新的 RK NPU 设备。

1.6.0 版本增加对 Keras 框架的支持。

11) 模型优化等级功能：RKNN-Toolkit 在模型转换过程中会对模型进行优化，默认的优化选项可能会对模型精度或性能产生一些影响。通过设置优化等级，可以关闭部分或全部优化选项。有关优化等级的具体使用方法请参考 config 接口中 optimization_level 参数的说明。该功能从 1.3.0 版本开始支持。

12) 模型加密功能：使用指定的加密等级将 RKNN 模型整体加密。RKNN-Toolkit 从 1.6.0 版本开始支持模型加密功能。因为 RKNN 模型的加密是在 NPU 驱动中完成的，使用加密模型时，与普通 RKNN 模型一样加载即可，NPU 驱动会自动对其进行解密。

注：部分功能受限于对操作系统或芯片平台的依赖，在某些操作系统或平台上无法使用。当前版本对各操作系统（平台）的功能支持情况列表如下：

表 1-1-1 RKNN Toolkit 各平台功能支持详情

	Ubuntu 16.04/18.04	Windows 7/10	Debian 9/10 (aarch64)	MacOS Mojave / Catalina
模型转换	支持	支持	支持	支持
量化/混合量化	支持	支持	部分支持（不支持量化参数优化方法 MMSE）	支持
模型推理	支持	支持	支持	支持
性能评估	支持	支持	支持	支持
内存评估	支持	支持	支持	支持
模型预编译	支持	部分支持（只支持在线预编译）	部分支持（只支持在线预编译）	部分支持（只支持在线预编译）
模型分段	支持	支持	支持	支持
自定义算子	支持	不支持	不支持	不支持
多输入	支持	支持	支持	支持
批量推理	支持	支持	支持	支持
设备查询	支持	支持	支持	支持
SDK 版本查询	支持	支持	支持	支持
量化精度分析	支持	支持	支持	支持
可视化功能	支持	支持	不支持	支持
模型优化开关	支持	支持	支持	支持
模型加密	支持	支持	支持	支持

1.2 适用芯片

RKNN-Toolkit 当前版本所支持芯片的型号如下：

- RK1806

-
- RK1808
 - RK3399Pro
 - RV1109
 - RV1126

注：RK3566 和 RK3568 需要使用 RKNN-Toolkit2，相关资料请参考以下工程：

<https://github.com/rockchip-linux/rknn-toolkit2>

1.3 适用系统

RKNN-Toolkit 是一个跨平台的开发套件，已支持的操作系统如下：

- Ubuntu: 16.04 (x64) 及以上
- Windows: 7 (x64) 及以上
- MacOS: 10.13.5 (x64) 及以上
- Debian: 9.8 (aarch64) 及以上

1.4 适用的深度学习框架

RKNN Toolkit 支持的深度学习框架包括 TensorFlow, TensorFlow Lite, Caffe, ONNX, Darknet 和 Keras。

它和各深度学习框架的版本对应关系如下：

表 1-4-1 RKNN Toolkit 深度学习框架支持情况

RKNN Toolkit	TensorFlow	TF Lite	Caffe	ONNX	Darknet	Pytorch	MXNet	Keras
1.0.0	>=1.10.0, <=1.13.2	Schema version = 3	1.0	Release version 1.3.0	Commit 号 : 810d7f7	不支持	不支持	不支持
1.1.0	>=1.10.0, <=1.13.2	Schema version = 3	1.0	Release version 1.3.0	Commit 号 : 810d7f7	不支持	不支持	不支持
1.2.0	>=1.10.0, <=1.13.2	Schema version = 3	1.0	Release version 1.4.1	Commit 号 : 810d7f7	不支持	不支持	不支持
1.2.1	>=1.10.0, <=1.13.2	Schema version = 3	1.0	Release version 1.4.1	Commit 号 : 810d7f7	不支持	不支持	不支持
1.3.0	>=1.10.0, <=1.13.2	Schema version = 3	1.0	Release version 1.4.1	Commit 号 : 810d7f7	>=1.0.0, <=1.2.0	>=1.4.0, <=1.5.1	不支持
1.3.2	>=1.10.0, <=1.13.2	Schema version = 3	1.0	Release version 1.4.1	Commit 号 : 810d7f7	>=1.0.0, <=1.2.0	>=1.4.0, <=1.5.1	不支持
1.4.0	>=1.10.0, <=1.13.2	Schema version = 3	1.0	Release version 1.4.1	Commit 号 : 810d7f7	>=1.0.0, <=1.2.0	>=1.4.0, <=1.5.1	不支持

							1	
1.6.0	>=1.10.0, <=2.0.0	Schema version = 3	1.0	Release version 1.6.0	Commit 号 : 810d7f7	>=1.0.0, <=1.6.0	>=1.4. 0, <=1.5. 1	>=2.1.6- tf
1.6.1	>=1.10.0, <=2.0.0	Schema version = 3	1.0	Release version 1.6.0	Commit 号 : 810d7f7	>=1.0.0, <=1.6.0	>=1.4. 0, <=1.5. 1	>=2.1.6- tf
1.7.0	>=1.10.0, <=2.0.0	Schema version = 3	1.0	Release version 1.6.0	Commit 号 : 810d7f7	>=1.0.0, <=1.6.0	>=1.4. 0, <=1.5. 1	>=2.1.6- tf
1.7.1	>=1.10.0, <=2.0.0	Schema version = 3	1.0	Release version 1.6.0	Commit 号 : 810d7f7	>=1.5.1, <=1.9.0	>=1.4. 0, <=1.5. 1	>=2.1.6- tf

注：

1. 依照语义版本，用某一版本 TensorFlow 写出的任何图或检查点，都可以通过相同主要版本中更高（次要或补丁）版本的 TensorFlow 来进行加载和评估，所以理论上，1.14.0 之前版本的 TensorFlow 生成的 pb 文件，RKNN Toolkit 1.0.0 及之后的版本都是支持的。关于 TensorFlow 版本兼容性的更多信息，可以参考官方资料：
https://www.tensorflow.org/guide/version_compat?hl=zh-CN
2. 因为 tflite 不同版本的 schema 之间是互不兼容的，所以构建 tflite 模型时使用与 RKNN Toolkit 不同版本的 schema 可能导致加载失败。目前 RKNN Toolkit 使用的 tflite schema 是基于 TensorFlow Lite 官方 GitHub master 分支上的如下提交：

0c4f5dfea4ceb3d7c0b46fc04828420a344f7598 。 具 体 的 schema 链 接 如 下 :

<https://github.com/tensorflow/tensorflow/commits/master/tensorflow/lite/schema/schema.fbs>

3. RKNN Toolkit 所使用的 caffe protocol 有两种, 一种是基于 berkeley 官方修改的 protocol, 一种是包含 LSTM 层的 protocol。其中基于 berkeley 官方修改的 protocol 来自:
<https://github.com/BVLC/caffe/tree/master/src/caffe/proto>, commit 值为 21d0608, RKNN Toolkit 在这个基础上新增了一些 OP。而包含 LSTM 层的 protocol 参考以下链接:
<https://github.com/xmfbit/warpctc-caffe/tree/master/src/caffe/proto>, commit 值为 bd6181b。这两种 protocol 通过 load_caffe 接口中的 proto 参数指定。

4. ONNX release version 和 opset version、IR version 之间的关系参考官网说明:

<https://github.com/microsoft/onnxruntime/blob/master/docs/Versioning.md>

ONNX release version	ONNX opset version	Supported ONNX IR version
1.3.0	8	3
1.4.1	9	3
1.6.0	11	6

5. Darknet 官方 Github 链接: <https://github.com/pjreddie/darknet>. RKNN Toolkit 现在的转换规则是基于 master 分支的最新提交 (commit 值: 810d7f7) 制定的。
6. 加载 Pytorch 模型 (torchscript 模型) 时, 推荐使用相同版本的 Pytorch 导出模型并转为 RKNN 模型, 前后版本不一致时可能会导致转 RKNN 模型失败。
7. RKNN Toolkit 目前主要支持以 TensorFlow 为 backend 的 Keras 版本, 所测 Keras 版本均为 TensorFlow 自带的 Keras。

2 开发环境搭建

本章节介绍在使用 RKNN Toolkit 前的开发环境准备工作。对于环境部署可能遇到的问题，将在 2.4 小节常见问题中给出。

2.1 参考系统配置

RKNN Toolkit 建议的系统配置如下：

硬件/操作系统	推荐配置
CPU	Intel Core i3 以上或同级别至强 / AMD 处理器
内存	16G 或以上
操作系统	Linux: Ubuntu 16.04 64bit 或 Ubuntu 18.04 64bit MacOS: 10.13.5 Windows: 7 或 10

注：模型转换阶段建议使用以上推荐配置的 PC 完成。在 RK3399Pro、RK1808 计算卡等设备上使用指定固件时也可以完成模型转换，但因为这些设备的 CPU 和内存资源有限，并不建议使用。

2.2 系统依赖说明

使用本开发套件时需要满足以下运行环境要求：

表 2-2-1 运行环境

操作系统版本	Ubuntu16.04（x64）及以上 Windows 7（x64）及以上 Mac OS X 10.13.5（x64）及以上 Debian 9.8（aarch64）及以上
Python 版本	3.5/3.6/3.7
Python 库依赖	<pre> 'numpy == 1.16.3' 'scipy == 1.3.0' 'Pillow == 5.3.0' 'h5py == 2.8.0' 'lmdb == 0.93' 'networkx == 1.11' 'flatbuffers == 1.10', 'protobuf == 3.11.2' 'onnx == 1.6.0' 'onnx-tf == 1.2.1' 'flask == 1.0.2' 'tensorflow == 1.11.0' or 'tensorflow-gpu' 'dill==0.2.8.2' 'ruamel.yaml == 0.15.81' 'psutils == 5.6.2' 'ply == 3.11' 'requests == 2.22.0' 'torch == 1.2.0' or 'torch == 1.5.1' or 'torch==1.6.0' 'mxnet == 1.5.0' 'sklearn == 0.0' 'opencv-python == 4.0.1.23' 'Jinja2 == 2.10' </pre>

注：

1. Windows 只提供 Python3.6 的安装包。
2. MacOS 提供 python3.6 和 python3.7 的安装包。
3. ARM64 平台（安装 Debian 9 或 10 操作系统）提供 Python3.5（Debian 9）和 Python3.7（Debian10）的安装包。
4. 因为 PyTorch / TensorFlow 等逐渐停止对 Python3.5 的支持，RKNN-Toolkit 下一个大版本将移除 Linux x86 平台上 Python3.5 的安装包，转而提供 Python3.6 和 Python3.7 的安装包。

5. 除 MacOS 平台外，其他平台的 scipy 依赖为 $\geq 1.1.0$ 。
6. ARM64 平台不需要依赖 sklearn 和 opencv-python。
7. Jinja2 只在使用自定义 OP 时用到。

2.3 工具安装

目前提供两种方式安装 RKNN-Toolkit：一是通过 Python 包安装与管理工具 pip 进行安装（本文的所有安装操作均基于 Ubuntu 16.04 Python3.5 环境，其他操作系统/平台请参考快速上手指南：Rockchip_Quick_Start_RKNN_Toolkit_CN.pdf）；二是运行带完整 RKNN-Toolkit 开发环境的 docker 镜像。下面分别介绍这两种安装方式的具体步骤。

注：Toybrick 设备的安装流程请参考以下链接：

<http://t.rock-chips.com/wiki.php?mod=view&id=36>

2.3.1 通过 pip install 命令安装

1. 创建 virtualenv 环境（如果系统中同时有多个版本的 Python 环境，建议使用 virtualenv 管理 Python 环境）

```
sudo apt install virtualenv
sudo apt-get install libpython3.5-dev
sudo apt install python3-tk

virtualenv -p /usr/bin/python3 venv
source venv/bin/activate
```

2. 安装 TensorFlow、python-opencv 等依赖库：

```
# 如果要使用 TensorFlow GPU 版本，请执行以下命令
pip3 install tensorflow-gpu==1.14.0
# 如果要使用 TensorFlow CPU 版本，请执行以下命令
pip3 install tensorflow==1.14.0
# 执行以下命令安装 PyTorch(ubuntu16.04 python3.5 官方只提供到 1.5.1
# 的安装包)和 torchvision
pip3 install torch==1.5.1 torchvision==0.4.0
# 执行以下命令安装 mxnet
pip3 install mxnet==1.5.0
```

注：因为 examples/pytorch 中的例子无法在 torchvision 0.6.1 上运行，所以这里安装 torchvision 0.4.0。请根据自行需要安装相应版本的 torchvision。

3. 安装 RKNN-Toolkit

```
pip3 install package/rknn_toolkit-1.7.1-cp35-cp35m-linux_x86_64.whl
```

请根据不同的 python 版本及处理器架构，选择不同的安装包文件（位于 package/目录）：

- **Python3.5 for x86_64:** rknn_toolkit-1.7.1-cp35-cp35m-linux_x86_64.whl
- **Python3.5 for arm_x64:** rknn_toolkit-1.7.1-cp35-cp35m-linux_aarch64.whl
- **Python3.6 for x86_64:** rknn_toolkit-1.7.1-cp36-cp36m-linux_x86_64.whl
- **Python3.6 for Windows x86_64:** rknn_toolkit-1.7.1-cp36-cp36m-win_amd64.whl
- **Python3.6 for Mac OS X:** rknn_toolkit-1.7.1-cp36-cp36m-macosx_10_15_x86_64.whl
- **Python3.7 for Mac OS X:** rknn_toolkit-1.7.1-cp37-cp37m-macosx_10_15_x86_64.whl
- **Python3.7 for arm_x64:** rknn_toolkit-1.7.1-cp37-cp37m-linux_aarch64.whl

2.3.2 使用 Docker 镜像

在 docker 文件夹下提供了一个已打包所有开发环境的 Docker 镜像，用户只需要加载该镜像即可直接上手使用 RKNN-Toolkit，使用方法如下：

1、安装 Docker

请根据官方手册安装 Docker（<https://docs.docker.com/install/linux/docker-ce/ubuntu/>）。

2、加载镜像

执行以下命令加载镜像：

```
docker load --input rknn-toolkit-1.7.1-docker.tar.gz
```

加载成功后，执行“docker images”命令能够看到 rknn-toolkit 的镜像，如下所示：

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
rknn-toolkit	1.7.1	50105e3a4110	1 hours ago	3.54GB

3、运行镜像

执行以下命令运行 docker 镜像，运行后将进入镜像的 bash 环境。

```
docker run -t -i --privileged -v /dev/bus/usb:/dev/bus/usb rknn-toolkit:1.7.1 /bin/bash
```

将文件夹映射进 Docker 环境可通过附加 “-v <host src folder>:<image dst folder>” 参数实现，例如：

```
docker run -t -i --privileged -v /dev/bus/usb:/dev/bus/usb -v /home/rk/test:/test rknn-toolkit:1.7.1 /bin/bash
```

4、运行 demo

```
cd /example/tflite/mobilenet_v1  
python test.py
```

注：从 RKNN-Toolkit V1.6.0 版本开始，提供的 Docker 镜像基于 Ubuntu 18.04 和 Python3.6。该镜像只能用于 x86 Linux 平台。

2.3.3 常见问题

工具安装常见问题请参考《Rockchip_Trouble_Shooting_RKNN_Toolkit_CN.pdf》第 1.1 章节。

2.4 连接 Rockchip NPU 设备

模型评估或部署时，需要连接 Rockchip NPU 设备。本节将给出 RK3399Pro, RK1808, RV1109, RV1126 开发板的连接方式，以及如何确认设备是否成功连接。

注：

1. 如果使用的是 Toybrick 开发板，请参考以下链接中的开机启动和串口调试内容：

<https://t.rock-chips.com/wiki.php?filename=%E7%BD%91%E7%AB%99%E5%AF%BC%E8%88%AA%E9%A6%96%E9%A1%B5>。

2. 使用模拟器进行模型评估，只需要一台装有 RKNN Toolkit 的 x86_64 Linux PC 即可。考虑到模拟器与开发板存在差异且耗时比较久，建议最终的评估工作通过实际使用的开发板进行。

2.4.1 通过 USB-OTG 口连接设备

RKNN Toolkit 与 Rockchip NPU 开发板之间的通信是通过 USB-OTG 接口进行的。在模型评估或模型部署阶段，需要通过 USB-OTG 接口连接 PC 和 Rockchip NPU 设备，以完成相关工作。

RK3399Pro USB-OTG 接口位置如下图红圈所示，通过 USB Type-C 线与 PC 连接：

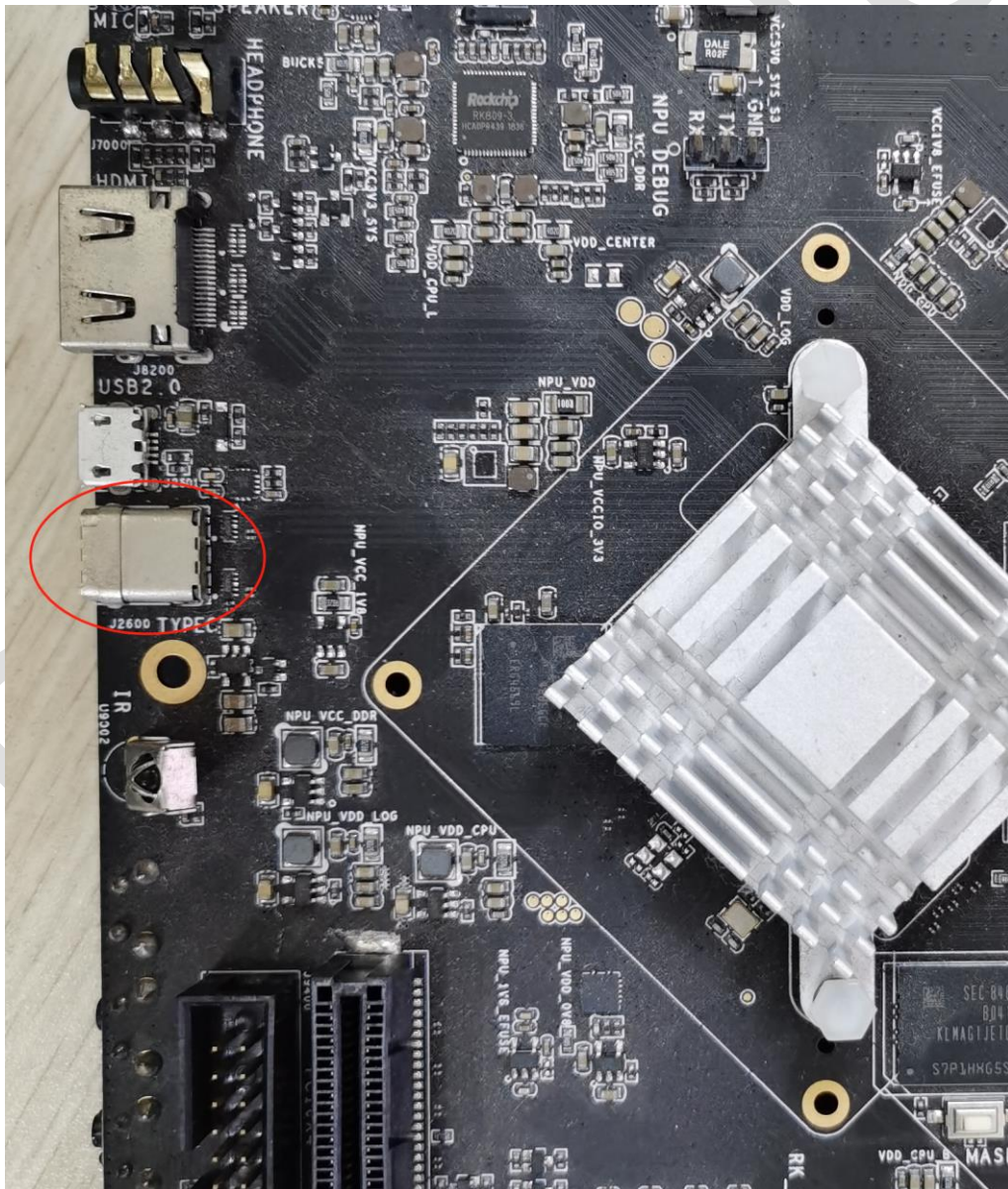


图 2-4-1-1 RK3399Pro USB-OTG 接口位置

RK1808 USB-OTG 接口位置如下图所示，通过 Micro-USB 线与 PC 连接：

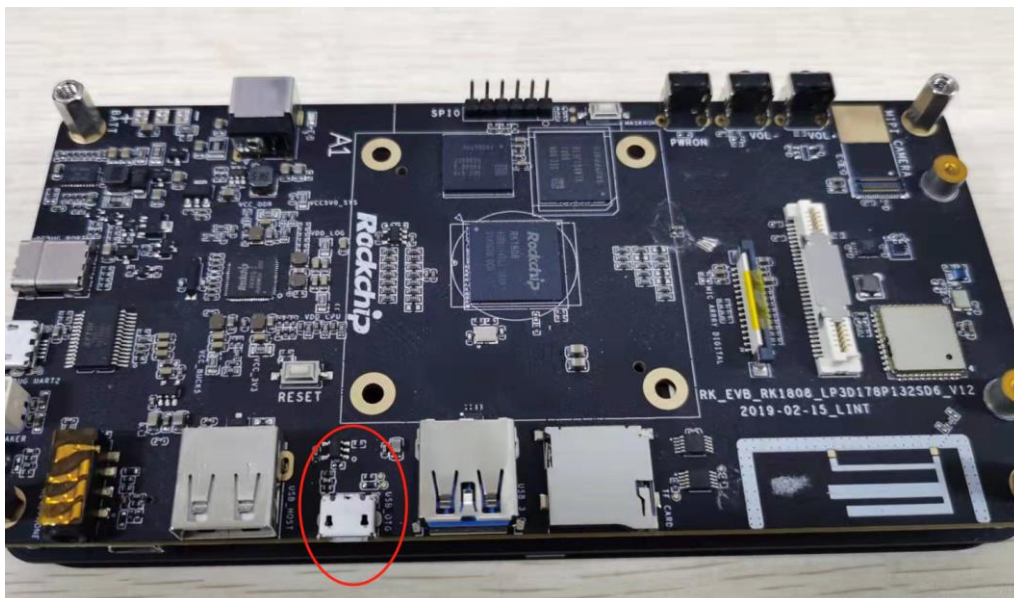


图 2-4-1-2 RK1808 USB-OTG 接口位置

RV1109/RV1126 的 USB-OTG 接口位置如下图所示，通过 Micro-USB 线与 PC 连接：



图 2-4-1-3 RV1109/RV1126 USB-OTG 口位置

2.4.2 通过 DEBUG 口连接设备

在模型评估或模型部署阶段，如果遇到开发板上的错误，需要通过 DEBUG 口连接 NPU 设备，抓取上面的日志。

RK3399Pro NPU DEBUG 接口位置如下图所示，通过 USB-串口转换小板与 PC 连接，PC 端通过 Minicom 或 Putty 等软件与开发板通信：

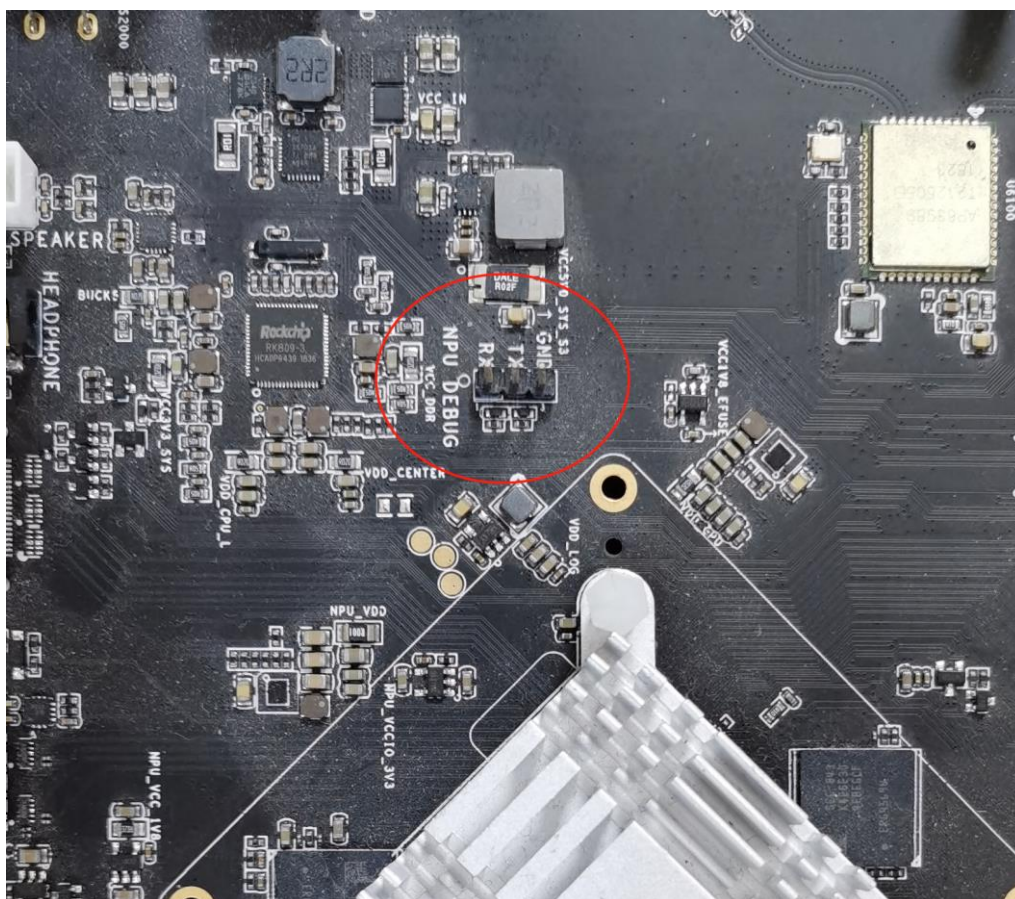


图 2-4-2-1 RK3399Pro DEBUG 接口位置

RK1808 DEBUG 接口位置如下图红圈所示，通过 Micro-USB 线与 PC 连接：

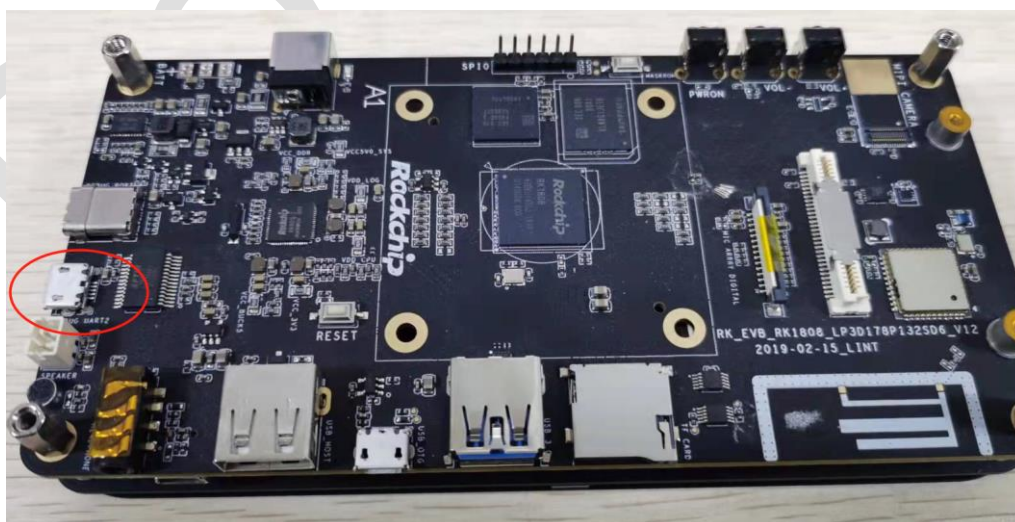


图 2-4-2-2 RK1808 DEBUG 接口位置

RV1109/RV1126 DEBUG 接口位置如下图红圈所示：

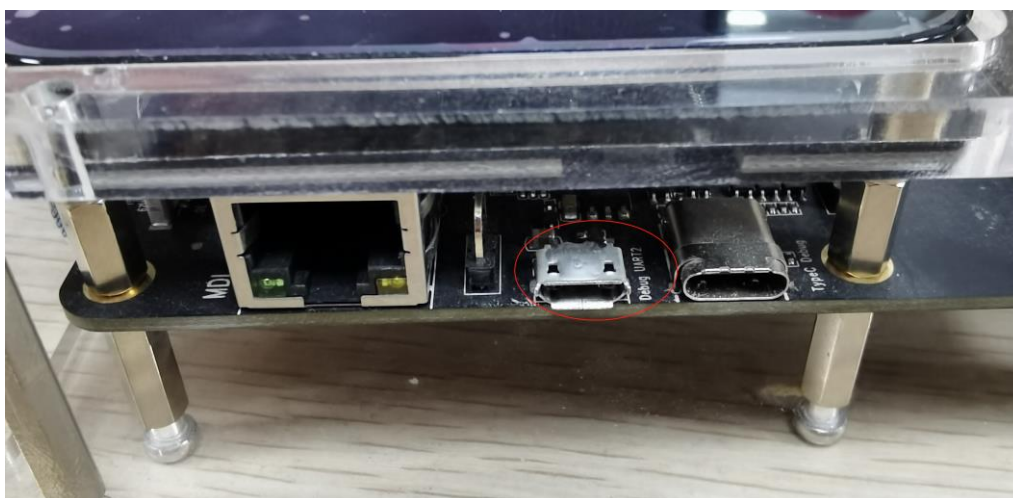


图 2-4-2-3 RV1109/RV1126 DEBUG 接口位置

2.4.3 确认设备连接正确

如果设备连接正确，在 PC 端的命令行窗口执行“python3 -m rknn.bin.list_devices”命令将列出所连接设备的 ID。参考输出如下：

```
*****
all device(s) with adb mode:
cf8f01ae745b49ce
all device(s) with ntb mode:
1126
*****
```

从这个输出可以看到，PC 目前连着两块开发板，第一块开发板的设备编号是 cf8f01ae745b49ce，第二块开发板的设备编号是 1126。

注：

1. 如果使用的是 x86_64 Linux 系统，第一次使用 Rockchip NPU 设备时可能需要更新 USB 设备权限，否则系统对这些设备可能没有读写权限。更新方法是执行 platform-tools/update_rk_usb_rule/linux/目录下的 update_rk1808_usb_rule.sh 脚本，执行完后需要重启系统。
2. 如果是在 Windows 上使用 Rockchip NPU 设备，需要先开启 NTB 通信模式（RK1808 和 Toybrick 开发板默认已开启），开启方法是通过 adb 进入开发板系统，修改文件

/etc/init.d/usb_config，在这个文件中增加一行：usb_ntb_en，然后重启开发板。原来的usb_adb_en要保留，否则无法通过adb进入开发板系统。

2.4.4 常见问题

设备连接常见问题请参考《Rockchip_Trouble_Shooting_RKNN_Toolkit_CN.pdf》文档 1.5 章节。

Rockchip

3 使用流程

3.1 基本使用流程

RKNN-Toolkit 的基本使用流程可以分为三个阶段：

1. 模型转换：将开源深度学习框架（Caffe、Darknet 等）训练导出的模型转成 RKNPU 能识别的 RKNN 模型。
2. 模型评估：通过 inference, eval_perf, eval_memory 等接口对 RKNN 模型推理结果的准确性、性能、内存使用情况进行评估。
3. 模型部署：通过模型评估确认 RKNN 模型符合部署要求后，可以通过 RKNN Toolkit Python API 进行模型部署，也可以通过 RKNN C API 进行模型部署。有关使用 RKNN Toolkit Python API 进行模型部署的更详细介绍，请参考 3.4 章节；有关使用 RKNN C API 进行模型部署的更详细介绍，请参考 RKNN C API 相关文档：

<https://github.com/rockchip-linux/rknpu/tree/master/rknn/doc>

3.2 模型转换

模型转换阶段的主要任务是将各类深度学习框架导出的模型转换成 Rockchip NPU 能处理的 RKNN 模型。RKNN Toolkit 对各深度学习框架的支持情况请参考 1.4 章节。

注：因为模型转换阶段需要消耗较多的 CPU（或 GPU），内存等资源，建议在满足 2.1 章节系统配置要求的 PC 上进行该操作。不建议在开发板上转换模型。

3.2.1 模型转换流程

模型转换的基本工作流程如下图所示：



图 3-2-1-1 RKNN 模型转换流程

成功执行完上述步骤后，RKNN 模型将保存到指定目录中。后续的评估和部署都将基于该 RKNN 模型进行。

注：上述流程中所用各接口的详细说明请参考第 7 章。

3.2.2 模型量化

3.2.2.1 模型量化简介

量化模型使用较低精度（如 `int8/uint8/int16`）保存模型的权重信息，在部署时可以使用更少的存储空间，获得更快的推理速度。但各深度学习框架训练、保存模型时，通常使用浮点数据，所以模型量化是模型转换过程中非常重要的一环。

RKNN Toolkit 目前对量化模型的支持主要有以下两种形式：

- RKNN Toolkit 根据用户提供的量化数据集，对加载的浮点模型进行量化，生成量化的 RKNN 模型。
 - 支持的量化精度类型：int16, int8, uint8
 - 量化方式：训练后静态量化
 - 支持的量化粒度：per-tensor（或 per-layer），**不支持 per-channel 量化**
- 由深度学习框架导出量化模型，RKNN Toolkit 加载并利用已有的量化信息,生成量化 RKNN 模型。
 - 支持的深度学习框架（括号内为主要支持版本，请尽量使用对应版本生成的量化模型）：PyTorch(v1.9.0)、ONNX(Onnxruntime v1.5.1)、Tensorflow、TFLite
 - 支持的量化精度类型：int8, uint8
 - 量化方式：训练后静态量化，量化感知训练(QAT)

3.2.2.2模型量化细节

- 训练后静态量化

使用这种方式时，RKNN Toolkit 加载用户训练好的浮点模型，然后根据 config 接口指定的量化方法和用户提供的校准数据集（训练数据或验证数据的一个小子集，大约 100~500 张）估算模型中所有浮点数据的范围（最小值，最大值）。RKNN Toolkit 目前支持 3 种量化方法：

- asymmetric_quantized-u8(默认量化方法)

这是 TensorFlow 支持的训练后量化算法，也是 Google 推荐的。根据论文”Quantizing deep convolutional networks for efficient inference: A whitepaper”的描述，这种量化方式对精度的损失最小。

其计算公式如下：

$$quant = round\left(\frac{float_num}{scale}\right) + zero_point$$

$$quant = cast_to_bw$$

其中 quant 代表量化后的数，float_num 代表浮点数，scale 表示缩放系数（float32 类型），

zero-points 代表实数为 0 时对应的量化值（int32 类型），最后把 quant 饱和到[range_min, range_max]，目前只支持 uint8 类型，所以 range_max 等于 255，range_min 等于 0

对应的反量化公式如下：

$$\text{float_num} = \text{scale}(\text{quant} - \text{zero_point})$$

■ dynamic_fixed_point-i8

对于有些量化模型而言，dynamic_fixed_point-i8 量化的精度比 asymmetric_quantized-u8 高。

其计算公式如下：

$$\begin{aligned}\text{quant} &= \text{round}(\text{float_num} * 2^{\text{fl}}) \\ \text{quant} &= \text{cast_to_bw}\end{aligned}$$

其中 quant 代表量化后的数，float_num 代表浮点数，fl 是左移的位数，最后把量化后的数饱和到[range_min, range_max]。如果位宽 bw 等于 8，则范围是[-127, 127]。

■ dynamic_fixed_point-i16

dynamic_fixed_point-i16 的量化公式与 dynamic_fixed_point-i8 一样，只不过它的位宽 bw 是 16。RK3399Pro 或 RK1808 的 NPU 自带 300Gops int16 计算单元，对于某些量化到 8 位后精度损失较大的模型，可以考虑使用此量化方式。

● 量化感知训练

通过量化感知训练可以得到一个带量化权重的模型。RKNN Toolkit 目前支持 TensorFlow 和 PyTorch 这两种框架量化感知训练得到的模型。量化感知训练技术细节请参考如下链接：

TensorFlow: https://www.tensorflow.org/model_optimization/guide/quantization/training

PyTorch: <https://pytorch.org/blog/introduction-to-quantization-on-pytorch/>

这种方法要求用户使用原始框架训练（或 fine tune）得到一个量化模型，接着使用 RKNN Toolkit 导入这个量化模型（模型转换时需要在 build 接口设置 do_quantization=False）。

此时 RKNN Toolkit 将使用模型自身的量化参数，因此理论上几乎不会有精度损失。

注：RKNN Toolkit 也支持 ONNX, PyTorch, TensorFlow, TensorFlow Lite 的训练后量化模型，模型转换方法与量化感知训练模型一样，调用 build 接口时 do_quantization 要设置成 False。如果要

使用 ONNX 量化模型，请更新 RKNN Toolkit 到 1.7.0 或更新的版本；如果要使用 PyTorch 量化模型，请更新 RKNN Toolkit 到 1.7.1。

3.2.3 模型转换示例

以 PyTorch 框架为例，模型转换示例如下：

```
import torchvision.models as models
import torch
from rknn.api import RKNN

PT_PATH = './resnet18.pt'

def export_pytorch_model():
    net = models.resnet18(pretrained=True)
    net.eval()
    trace_model = torch.jit.trace(net, torch.Tensor(1,3,224,224))
    trace_model.save(PT_PATH)

if __name__ == '__main__':

    # 导出 PT 模型
    export_pytorch_model()

    # 创建 RKNN 对象
    rknn = RKNN()

    # 设置模型输入预处理参数
    rknn.config(mean_values=[[123.675, 116.28, 103.53]], std_values=[[58.395, 58.395,
58.395]], reorder_channel='0 1 2')

    # 加载 PyTorch 模型
    ret = rknn.load_pytorch(model=PT_PATH, input_size_list=[[3, 224, 224]])
    if ret != 0:
        print('Load Pytorch model failed!')
        exit(ret)

    # 构建 RKNN 量化模型
    ret = rknn.build(do_quantization=True, dataset='./dataset.txt')
    if ret != 0:
        print('Build model failed!')
        exit(ret)

    # 导出 RKNN 模型到指定路径
    ret = rknn.export_rknn('./resnet18.rknn')
    if ret != 0:
```

```
print('Export resnet18.rknn failed!')
exit(ret)

rknn.release()
```

其他框架请参考 SDK/examples 目录下各框架的示例。

转换已量化模型的例子请参考 SDK/examples/common_function_demos/目录下的示例。

3.2.4 常见问题

模型转换常见问题请参考《Rockchip_Trouble_Shooting_RKNN_Toolkit_CN.pdf》文档 1.3 章节

3.3 模型评估

通过模型转换得到 RKNN 模型后，可以使用 RKNN Toolkit 提供的 Python 接口在 Rockchip NPU 开发板（或模拟器，仅 x86_64 Linux 支持）上对模型的准确性、性能、内存使用情况进行评估。

注：

1. 使用模拟器进行评估时，不需要连接 Rockchip NPU 开发板，但要求 RKNN Toolkit 安装在 x86_64 Linux 系统上，且使用模拟器评估耗时会比较长。
2. 使用 Rockchip NPU 开发板进行评估前，请参考 2.4 章节，正确连接 PC 和开发板，确保 RKNN Toolkit 能识别到该设备。
3. 如果是在 RK3399Pro 开发板上评估，因为其自带 NPU，不需要额外连接 Rockchip NPU 开发板。RKNN Toolkit Python 接口只支持在 Debian 固件的 RK3399Pro 上运行，如果开发板的固件是 Android 系统，请使用 PC 接 RK3399Pro 的方式进行评估。

典型的评估流程如下图所示：

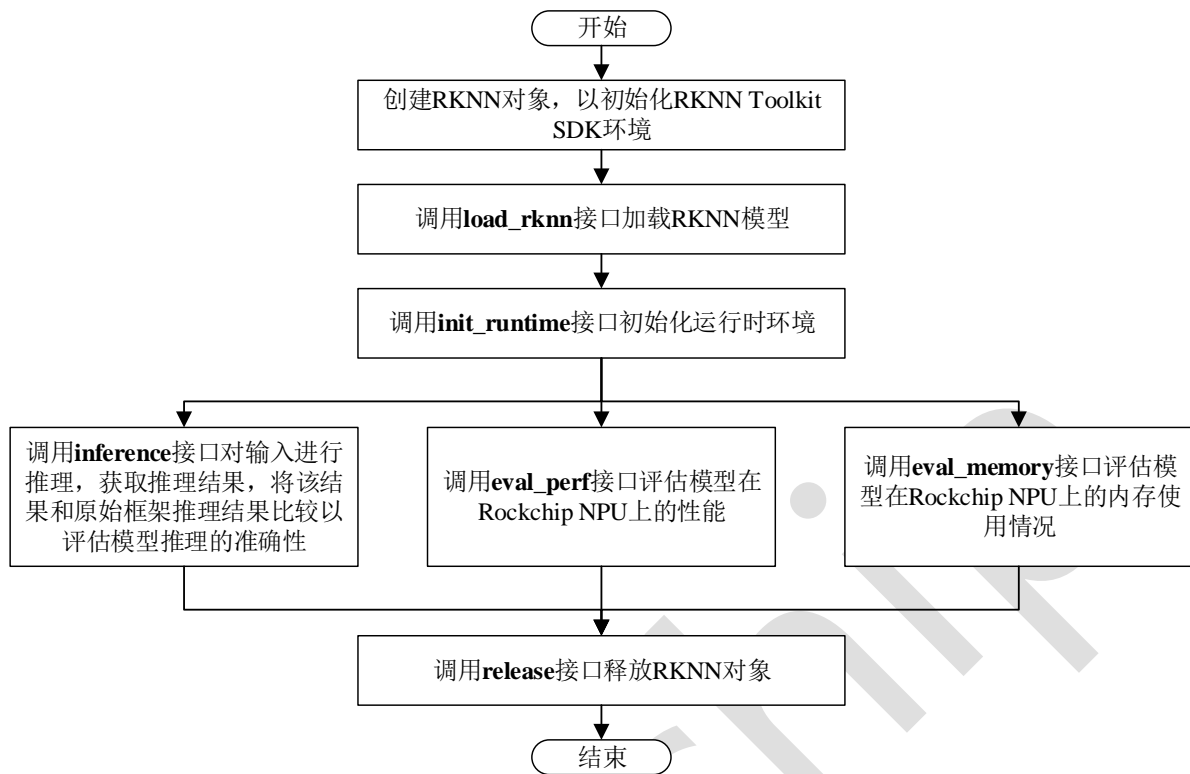


图 3-3-1 RKNN 模型评估流程

模型评估主要分三部分进行：准确性评估、性能评估和内存评估。

- 准确性评估：评估 RKNN 模型推理结果的准确性。
- 性能评估：评估 RKNN 模型在指定开发板上推理时的耗时。
- 内存评估：评估 RKNN 模型推理时在 Rockchip NPU 上的内存使用情况。

准确性评估相关内容将在第 4 章中详细展开，性能评估相关内容将在第 5 章中详细展开，内存评估相关内容将在第 6 章中详细展开。

3.4 模型部署

本节主要介绍如何利用 RKNN Toolkit 提供的 Python 接口进行应用开发，将前一步评估好的模型部署到 Rockchip NPU 上。

3.4.1 准备工程文件

最简单的应用只需要包含输入数据，RKNN 模型和应用程序脚本。

这里以一个基于 resnet18 模型的 ImageNet 图像分类器为例，该工程包含如下文件：

```
ai_demo/
├── imagenet_classes.txt
├── imgs
│   └── space_shuttle_224.jpg
├── resnet18_classifier.py
└── resnet18.rknn
```

文件说明如下：

- **imagenet_classes.txt**: 记录图片类别的文件。
- **imgs** 目录: 存放待分类的图片，这里只放了一张图片 `space_shuttle_224.jpg`，可以根据需要存放更多图片。
- **resnet18_classifier.py**: 图像分类主程序，该程序的详细说明将在下一节中给出。
- **resnet18.rknn**: 根据第 3.2 章节转换得到的 RKNN 模型（注：该模型的目标设备是 RK1808，只能部署在 RK1808 或 RK3399Pro 上，如果要部署在 RV1109 或 RV1126 上，需要在转换模型时修改 config 接口的 `target_platform` 参数，具体说明请参考第 8.2 章节）。

3.4.2 应用程序示例

最简单的应用程序运行流程如下图所示：

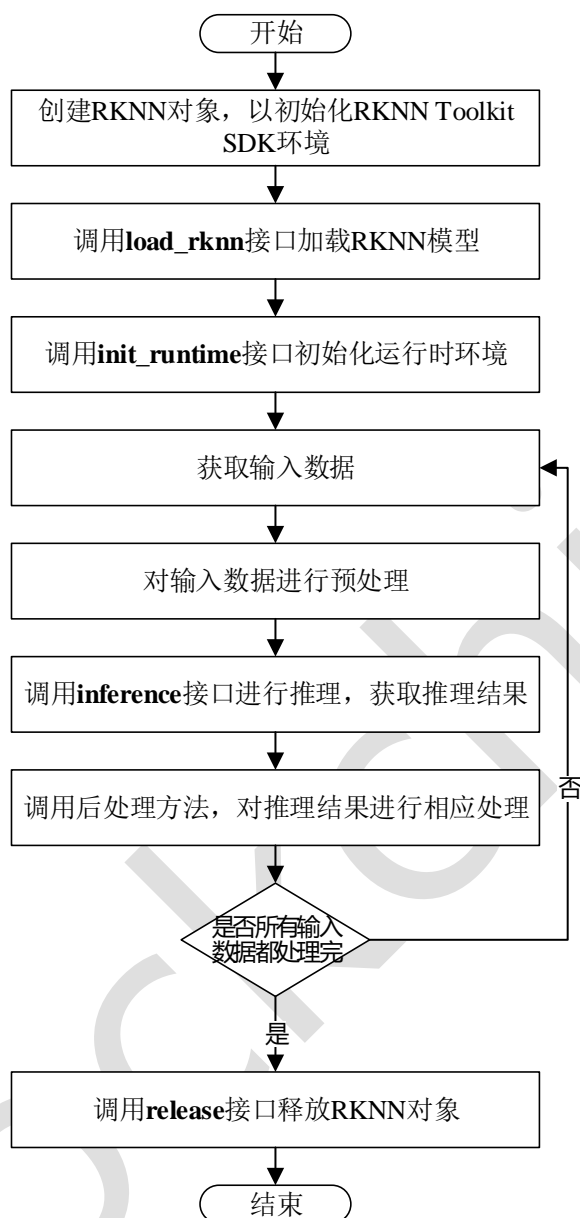


图 3-4-2-1 简单的应用程序流程图

在这个流程中，核心步骤是加载 RKNN 模型、初始化运行时环境、准备输入数据、模型推理和推理结果的后处理。

这里以图像分类为例，展示如何将分类模型 `resnet18.rknn` 部署到 RK1808 开发板上。

工程目录：

```

import os
import cv2
import torch
import numpy as np
from rknn.api import RKNN

RKNN_PATH = './resnet18.rknn'
IMGS_DIR = './imgs'

def classification_post_process(output, categories):
    output = output.reshape(-1)
    probabilitites = np.exp(output)/sum(np.exp(output))
    reverse_sort_index = np.argsort(output)[::-1]
    print('-----TOP 1-----')
    for i in range(1):
        print(categories[reverse_sort_index[i]], ':', probabilitites[reverse_sort_index[i]])

if __name__ == '__main__':
    # 创建 RKNN 对象
    rknn = RKNN()

    # 从当前目录加载 RKNN 模型 resnet_18
    ret = rknn.load_rknn(path=RKNN_PATH)
    if ret != 0:
        print('Load Pytorch model failed!')
        exit(ret)

    # 初始化运行时环境，设置目标开发板为 RK1808
    # 如果只有一个设备，device_id 可以不填
    ret = rknn.init_runtime(target='rk1808', device_id='1808')
    if ret != 0:
        print('Init runtime environment failed')
        exit(ret)

    # 读取 imagenet_classes.txt, 获取类别
    with open("imagenet_classes.txt", "r") as f:
        categories = [s.strip() for s in f.readlines()]

    for img_file in os.listdir(IMGS_DIR):
        img_path = os.path.join(IMGS_DIR, img_file)

        # 读取测试图像, 缩放到指定尺寸, 将 BGR 转成 RGB
        img = cv2.imread(img_path)
        img = cv2.resize(img, (224, 224), interpolation=cv2.INTER_CUBIC)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        # 调用 inference 接口进行推理

```



```
outputs = rknn.inference(inputs=[img])

# 调用后处理方法得到图片的类别,打印类别和概率
classification_post_process(outputs[0], categories)

# 所有待分类图片处理完后,释放 RKNN 对象
rknn.release()
```

目标检测模型的部署,可以参考 SDK/examples/caffe/vgg-ssd, SDK/examples/darknet/yolov3, SDK/examples/onnx/yolov5 或者 SDK/examples/tensorflow/ssd_mobilenet_v1 等。

图像分割模型的部署,可以参考 SDK/examples/mxnet/fcn_resnet101 等。

其他分类模型的部署,可以参考 SDK/examples/caffe/mobilenet_v2, SDK/examples/keras/xception, SDK/examples/mxnet/resnext50, SDK/examples/onnx/resnet50v2, SDK/examples/tflite/mobilenet_v1 等。

注:

1. 如果最终产品部署时使用 Python 接口,建议使用 RKNN Toolkit Lite, 它将 RKNN Toolkit 的推理功能剥离出来,可以大幅减少 SDK 的空间占用。有关 RKNN Toolkit Lite 的详细介绍,请参考文档: Rockchip_User_Guide_RKNN_Toolkit_Lite_CN.pdf
2. 如果最终产品部署时使用 RKNN C API, 请参考 RKNN C API 的使用文档和示例:
 - RK1808 / RV1109 / RV1126: <https://github.com/rockchip-linux/rknpu/tree/master/rknn>
 - RK3399Pro: <https://github.com/rockchip/linux/RKNPUTools/tree/rk33/mid/8.1/develop/rknn-api>

3.4.3 应用程序运行

如 3.4.2 章节示例, 在安装有 RKNN Toolkit 的 python 环境中直接执行”python resnet18_classifier.py”命令即可。

注: 请先按照 2.4 章节说明连接好 Rockchip NPU 开发板。

3.4.4 模型预编译

Rockchip NPU 加载 RKNN 模型时, 根据网络结构构建一个运行时的图, 接着转成 NPU 上运行的命令, 对于网络结构比较大的模型, 这一步会消耗比较多的时间。为了解决这个问题, RKNN

Toolkit 提供模型预编译功能，让模型初始化时间降到 1 秒以内。

RKNN Toolkit 提供两种模型预编译方法：

- 离线预编译：在调用 `build` 接口时设置 `pre_compile` 参数为 `True` 即可。该方法只有 `x86_64 Linux` 平台支持。
- 在线预编译：通过在线预编译接口 `export_rknn_precompile_model` 完成。使用这种预编译方法时需要连接 Rockchip NPU 开发板。

使用在线预编译时，可以参考以下示例：

SDK/examples/common_function_demos/export_rknn_precompile_model/

3.4.5 模型加密

为了避免模型的结构、权重等信息泄漏，RKNN Toolkit 提供模型加密功能。

RKNN Toolkit 提供 3 个加密等级，等级越高，安全性越高，解密越耗时；反之，安全性越低，解密越快。

模型加密流程如下图所示：

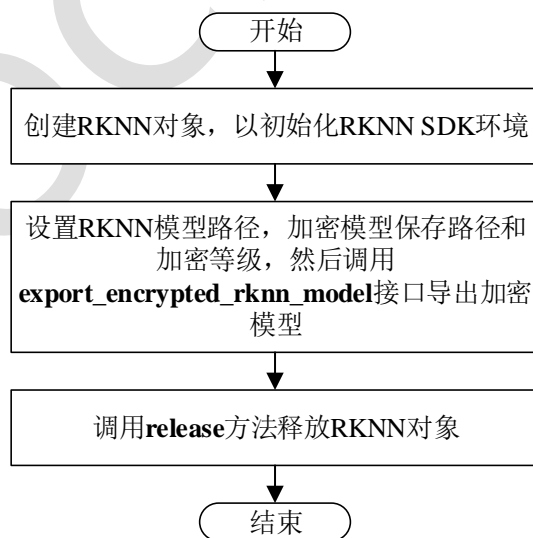


图 3-4-5-1 模型加密流程

加密模型的部署流程和普通一样，不需要解密等额外操作。

3.4.6 多模型调度

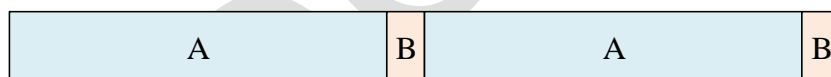
多 RKNN 模型场景中，每个 RKNN 模型都需要使用有限的 NPU 资源完成推理任务，这就导致了 NPU 资源抢占的问题。RK1808, RV1109 等 NPU 本身不具备任务抢占功能，一个模型一旦开始推理，就只能等这个模型推理结束才会让出 NPU 资源，其他模型只能等待。这就很容易出现 NPU 资源被大模型长期占用，优先级较高的小模型没有机会运行的问题。为了解决这个问题，RKNN Toolkit 提供模型分段的功能，从软件层面提供简单可行的调度方法。

3.4.6.1 模型分段功能介绍

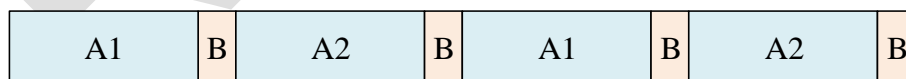
RKNN-Toolkit 通过 `export_rknn_sync_model` 接口将普通的 RKNN 模型分成多段。每个分段(未启用模型分段功能的模型默认就是一个分段)抢占 NPU 的机会是均等的，在一个分段执行完成后，会主动让出 NPU（如果该模型还有下一分段，则会将该分段再次加入到命令队列尾部），此时如果有其他的模型的分段在等待执行，则会按命令队列先后顺序执行其他模型的分段。

举例如下：

当前应用程序使用了两个模型，模型 A 推理耗时 200ms，模型 B 推理耗时 20ms。初始化时先初始化 A，后初始化 B，如果不进行模型分段，此时推理队列如下：



模型 B 推理一次，必须等待 200ms。但如果我们将模型 A 分成 2 段，每段 100ms，此时推理队列如下：



此时模型 B 推理一次，只需要等待 100ms。在实际使用时可以根据自己的需要对模型进行分段。

注：开启模型分段功能会降低单模型推理的效率，所以单模型场景下，建议关闭该功能（不使用模型分段功能即可）。

3.4.6.2 模型分段使用流程

模型分段流程如下图所示：

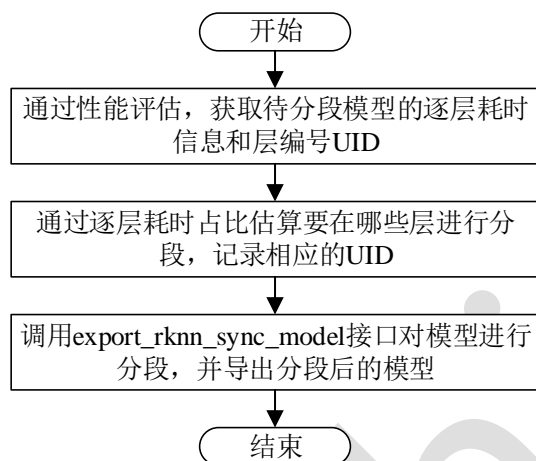


图 3-4-6-2-1 模型分段使用流程

注：

1. 模型性能评估时获取的每一层耗时可能会偏长，所以计算时建议用比例换算，比如模型有三层，性能评估时 A 层耗时 10ms，B 层耗时 20ms，C 层耗时 10ms，但实际模型推理时间只有 24ms，按比例换算，实际耗时可能是 A 层 7ms，B 层 14ms，C 层 7ms。
2. 性能评估时有两个 ID，Layer ID 和 Uid，模型分段时要填写的是 Uid。

3.4.7 应用部署常见问题

应用部署常见问题请参考文档：《Rockchip_Trouble_Shooting_RKNN_Toolkit_CN》相关章节。

4 准确性评估

本章将详细说明 RKNN 模型准确性评估的方法、常见问题和常用的精度排查方法。

4.1 评估方法

最简单的评估方法是將 RKNN 模型的推理结果与原始框架的推理结果进行比较，用欧式距离或余弦距离评估这两个结果的相似性。也可以使用后处理验证结果的准确性，或者直接在数据集上进行精度测试。

这里以最简单的计算余弦距离评估推理结果正确性为例，代码如下所示：

```
import cv2
import torch
import numpy as np
from rknn.api import RKNN

PT_PATH = './resnet18.pt'
RKNN_PATH = './resnet18.rknn'
MEANS = [123.675, 116.28, 103.53]
STDS = [58.395, 58.395, 58.395]

def inference_with_rknn(target, device_id, inputs):
    # 创建 RKNN 对象
    rknn = RKNN()

    # 从当前目录加载 RKNN 模型 resnet_18
    ret = rknn.load_rknn(path=RKNN_PATH)
    if ret != 0:
        print('Load Pytorch model failed!')
        exit(ret)

    # 初始化运行时环境
    ret = rknn.init_runtime(target=target, device_id=device_id)
    if ret != 0:
        print('Init runtime environment failed')
        exit(ret)

    # 调用 inference 接口进行推理
    outputs = rknn.inference(inputs=inputs)

    # 释放 RKNN 对象
    rknn.release()
```

```

return outputs

def inference_with_torch(img):
    # 输入图像预处理
    img = (img - MEANS) / STDS
    ## OpenCV 读取的图像是按 HWC 排列的,Torch 要求 NCHW,需要做下转换
    img = img.reshape((1, 224, 224, 3))
    img = img.transpose((0, 3, 1, 2))
    img = img.astype(np.float32)
    torch_inputs = [torch.from_numpy(img)]

    # 加载 PyTorch 模型
    net = torch.load(PT_PATH)

    # 调用 forward 方法推理
    outputs = net.forward(*torch_inputs)
    return outputs

def compute_cos_dis(x, y):
    cos_dist= (x* y)/(np.linalg.norm(x)*(np.linalg.norm(y)))
    return cos_dist.sum()

if __name__ == '__main__':

    # 读取测试图像
    img = cv2.imread('./space_shuttle_224.jpg')
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

    # 使用 RKNN Toolkit 推理接口在 RK1808 开发板上推理
    rknn_outs = inference_with_rknn(target='rk1808', device_id='1808', inputs=[img])

    # 使用 torch 进行推理
    torch_outs = inference_with_torch(img=img)

    # 计算并打印两个结果的余弦相似度
    cos_dis = compute_cos_dis(rknn_outs[0], torch_outs[0].cpu().detach().numpy())
    print("Cosine distance of RKNN output and Torch output: {}".format(cos_dis))

```

4.2 准确性问题排查

如果准确性评估结果不理想，RKNN 模型精度达不到预期指标，请参考如下步骤排查：

- 首先确保浮点 RKNN 模型的精度和原始框架测试结果相近；

- 使用 build 接口构建 RKNN 模型时，设置 do_quantization 参数的值为 False;
- 正确设置 config 接口中的 mean_values, std_values, reorder_channel 等参数，确保这些参数与模型训练时一致;
- 如果模型输入是图片，确保测试图片的通道顺序是 RGB(使用 OpenCV 读取的图像，默认通道顺序是 BGR)，不论训练时使用的图像通道顺序如何，使用 RKNN 模型测试时都按 RGB 输入;
- 如果模型输入是 4 维的，确保测试图片的数据排列格式和 inference 接口中的 data_format 参数保持一致。OpenCV 读取图像的格式是 HWC，如果是单张图片，此时排列顺序与 data_format 中的 NHWC 一致。
- 模型量化时，校准数据集里的输入数据建议多放一些有代表性的数据(数量在 100~500 张之间)，不要放与测试场景无关的数据。校准数据集数据比较多时，使用 build 接口构建 RKNN 模型时可能因为内存不足而失败，此时建议将 config 接口中的 batch_size 参数设小一点，例如 8, 16 等。
- 如果是用数据集进行评估，尽量使用较大数据集进行评估。分类网络比较 top1, top5 精度，检测网络比较数据集的 mAP, Recall 等。

排查上述因素后，RKNN 模型的推理结果仍不对，针对浮点模型和量化模型，可以做如下分析以找出引起误差的算子。考虑耗时等因素，进一步的分析建议先用一组输入数据。

4.2.1 浮点模型精度问题排查步骤

RKNN Toolkit (x86_64 Linux) 自带的模拟器或者 Rockchip NPU 通过设置，都可以输出中间层的结果。将中间层的结果与原始框架对比，可以定位哪个算子出现问题。如果推理结果出错的算子可以用其他算子替换，请尝试替换该算子，然后重新训练模型再转成 RKNN 进行评估(建议先少量训练几次进行验证，通过后再进行完整训练); 如果无法找到替换算子，可以将该算子的参数、输入输出等信息或包含该算子的最小复现模型反馈给瑞芯微 NPU 团队进行进一步分析。

输出中间层结果的具体方法如下:

1. 如果是模拟器，执行如下命令设置环境变量 NN_LAYER_DUMP 后再执行模型推理脚本:

`export NN_LAYER_DUMP=1`。推理时每一层的结果都将保存在当前目录下。

2. 如果是 Rockchip NPU 开发板，分两种情况，一种是使用 RKNN Toolkit Python 接口，此时需要通过串口连接到开发板，在开发板上找一个存放中间层结果的目录，并在该目录设置环境变量 `NN_LAYER_DUMP`，且需要重启 `rknn_server` 进程，具体命令为“`export NN_LAYER_DUMP=1 && restart_rknn.sh`”；第二种是使用 RKNN C API 接口进行评估，此时只要在开发板上设置 `NN_LAYER_DUMP` 环境变量即可，中间层结果将保存在执行推理程序的目录中。

注：如果开发板是 RK3399Pro，需要在自带的 NPU 上执行上述操作。

得到中间层结果后将这些结果与原始框架中间层的结果进行比较，原始框架获取中间层结果的方法请查阅各框架相关资料。

对比数据时，如果发现第一层结果差距很大（余弦距离小于 0.98），这通常是输入预处理与原始框架不一致导致。请首先检查 `config` 中的 `mean_values`, `std_values`, `reorder_channel` 等参数是否正确设置，或者推理时传给 `inference` 接口的数据是否正确。

如果是其他情况，请将具体的算子或者最小复现模型反馈给瑞芯微 NPU 团队进行进一步分析。

4.2.2 量化模型精度问题排查步骤

- 首先检查浮点模型推理结果是否正确，不正确，请先按照上一小节进行排查；如果正确，则进行下一步。
- 接着可以使用精度分析接口 `accuracy_analysis` 进行精度分析，也可以参考上一节的方法，手动导出每一层推理结果，与原始框架推理结果进行比较，找出造成精度损失的层。
- 对精度下降的层进行分析。

量化后精度下降可能存在以下三种情况：一是 RKNN Toolkit 对该层算子的匹配有问题，或者驱动对该算子的实现有问题（此时该层的推理结果通常会和浮点模型的结果差距巨大，余弦距离小于 0.5）；二是该层数据分布对量化不友好（例如数据分布范围比较广，用低精度表达后精度损失严重）；三是 RKNN Toolkit 或 NPU 驱动的某些图优化可能导致精度下降，例如用 `conv` 替换 `add` 或者 `average pool` 等。对于**第一种场景**，可以尝试用混合量化的方式进行规

避。对于**第二种场景**，可以尝试用 MMSE 或 KL 散度方法对量化参数进行调优。还无法提高精度的情况下，考虑使用混合量化，用更高精度的方法去计算对量化不友好的算子；或者使用 TensorFlow，PyTorch 进行量化感知训练，得到一个量化模型，然后直接将这个量化模型转成 RKNN 模型。对于**第三种场景**，可以尝试将优化等级下调（下调的方法是在 `config` 接口将 `optimization_level` 设成 2 或 1）。如果使用以上方法后模型精度还是无法满足要求，请将相关模型、精度分析结果和精度测试方法反馈给瑞芯微 NPU 团队进行进一步分析。

注：

1. 精度分析功能的详细用法请参考 4.3 章节；
2. MMSE 量化参数优化算法的详细说明请参考 4.4 章节；
3. 混合量化相关接口的用法请参考 4.5 章节。

4.3 精度分析

4.3.1 功能介绍

RKNN Toolkit 精度分析功能可以保存浮点模型、量化模型推理时每一层的中间结果，并用欧式距离和余弦距离评估它们的相似度。通过观察每一层相似度的变化情况，可以初步定位哪些算子计算结果不正确，或者对量化不友好。

4.3.2 使用流程

量化精度分析功能使用流程如下：

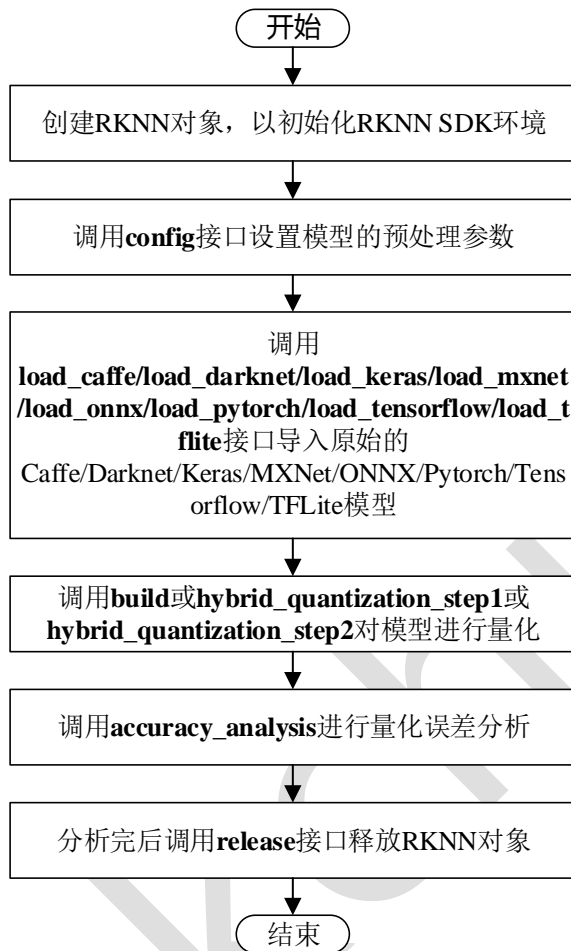


图 4-3-2-1 量化精度分析使用流程

注：

1. accuracy_analysis 中指定的 inputs 只能包含一组输入。
2. 上述接口的调用顺序不能调换。

4.3.3 输出说明

精度分析输出的目录结构如下：

```

├── entire_qnt
├── entire_qnt_error_analysis.txt
├── fp32
├── individual_qnt
├── individual_qnt_error_analysis.txt
├── individual_qnt_error_analysis_on_npu.txt
└── qnt_npu_dump
  
```

各文件/目录含义如下：

- `entire_qnt` 目录：保存整个量化模型完整运行时每一层的结果（已转成 `float32`）；
- `entire_qnt_error_analysis.txt`：记录量化模型完整运算时每一层结果与浮点模型结果的余弦距离/欧式距离、归一化后的余弦距离/欧式距离。余弦距离越小或欧式距离越大，表明量化后的精度下降得越厉害。
- `fp32` 目录：保存整个浮点模型完整跑下来时每一层的结果，可以根据该目录中 `order.txt` 记录的顺序与原始模型进行对应。如果浮点模型本身结果不对，可以将该目录中每一层的结果与原始框架推理时每一层的结果进行对比，从而确定出问题的是哪一层，然后反馈给瑞芯微 NPU 团队。
- `individual_qnt` 目录：将量化模型拆成一层一层，逐层运行。每一层在推理时的输入为上一层用浮点模型推理的结果。该目录保存每层单独运行时的结果（已转成 `float32`）。这样做可以避免累积误差。
- `individual_qnt_error_analysis.txt` 文件：记录量化模型逐层运行时每一层的结果与浮点模型结果的余弦距离/欧式距离、归一化后的余弦距离/欧式距离。余弦距离越小或欧式距离越大，表明量化后的精度下降得越厉害。

如果设置了 `target`，该目录下会多出以下内容：

```
├── individual_qnt_error_analysis_on_npu.txt
└── qnt_npu_dump
```

- `individual_qnt_error_analysis_on_npu.txt` 文件：记录量化模型逐层在硬件设备上运行时每一层结果与浮点模型结果的余弦距离/欧式距离、归一化后的余弦距离/欧式距离。余弦距离越小或欧式距离越大，表明量化后的精度下降得越厉害。
- `qnt_npu_dump` 目录：将量化模型拆成一层一层后逐个放到 NPU 设备上运行，所用的输入为浮点模型上一层的结果，该目录保存量化模型逐层在 NPU 上实际运行时的结果（`dump` 结果时会自动转成 `float32`，方便比较）。

4.4 优化量化参数

RKNN Toolkit 提供两种量化参数优化方法：MMSE 和 KL 散度，这两种方法会根据各自算法去搜索最优的量化参数组合，以提高精度。

在 config 接口指定 `quantized_algorithm` 参数值为"mmse"可启用 MMSE 量化参数优化方式，指定该参数值为"kl_divergence"，可以启用 KL 散度量化的参数优化算法。

4.5 混合量化

RKNN-Toolkit 提供的量化功能可以在提高模型推理速度的基础上尽量保证模型精度，但是仍有某些特殊模型在量化后出现精度下降较多的情况。为了在性能和精度之间做更好的平衡，RKNN-Toolkit 从 1.0.0 版本开始提供混合量化功能，用户可以指定各层是否量化，也可以手动指定量化参数。

注：

1. `examples/common_function_demos/hybrid_quantization` 目录下提供有两个混合量化的例子，可以参考该例子进行混合量化的实践。

4.5.1 混合量化功能用法

目前混合量化功能支持如下三种用法：

1. 将指定的量化层改成非量化层（如用 float32 进行计算）。因 NPU 的浮点数算力较低，推理性能会下降。
2. 将指定的非量化层改成量化层。
3. 修改指定量化层的量化参数。

4.5.2 混合量化配置文件

在使用混合量化功能时，第一步是生成混合量化配置文件，本节对该配置文件进行简要介绍。

当调用混合量化接口 `hybrid_quantization_step1` 后，会在当前目录下生成名为

{model_name}.quantization.cfg 的配置文件，该配置使用 YAML 语法。配置文件格式如下：

```
%YAML 1.2
# add layer name and corresponding quantized_dtype to customized_quantize_layers, e.g
conv2_3: float32
customized_quantize_layers: {}
quantize_parameters:
  '@attach_concat_1/out0_0:out0':
    dtype: asymmetric_affine
    method: layer
    max_value:
      - 10.097497940063477
    min_value:
      - -52.340476989746094
    zero_point:
      - 214
    scale:
      - 0.24485479295253754
    qtype: u8

.....

  '@FeatureExtractor/MobilenetV2/Conv/Conv2D_230:bias':
    dtype: asymmetric_affine
    method: layer
    max_value:
    min_value:
    zero_point: 0
    scale:
      - 0.00026041566161438823
    qtype: i32
```

第一行是 YAML 的版本，第二行是一个分隔符，第三行是注释。后面是配置文件的主要内容。

配置文件正文第一行是一个自定义量化层的字典，修改时将层名和相应的量化类型（当前版本可选值为 **asymmetric_affine-u8**, **dynamic_fixed_point-i8**, **dynamic_fixed_point-i16**, **float32**）填写到这个字典中，作为自定义的量化层。从 1.6.0 版本开始，混合量化第一步会根据一定的规则给出可能可以提高精度的层，并将量化方法指定成 **dynamic_fixed_point-i16**，这些层仅做参考。如果只想修改后面的量化参数，而不使用其他的量化方法，则这一行后面需要加上 {}。

之后是模型每层的量化参数，每一层都是一个字典。每个字典的 key 由 @{layer_name}_{layer_id} 组成，其中 layer_name 是层名，layer_id 是层 id。字典的 value 即量化参数，如果没有经过量化，则 Value 里只有 dtype 项，且值为 None。

4.5.3 混合量化使用流程

使用混合量化功能时，具体分四步进行。

第一步，加载原始模型，生成量化配置文件和模型结构文件和模型配置文件。具体的接口调用流程如下：



图 4-5-3-1 混合量化第一步接口调用流程

第二步，修改第一步生成的量化配置文件。

- 如果是将某些量化层改成非量化层，则找到不要量化的层，将这些层名加到 `customized_quantize_layers` 字典中，值为 `float32`，例如 `"<layer_name>: float32"`。也可以使用其他量化方式，例如原来是用 `asymmetric_affine-u8` 的，这里也可以改成 `dynamic_fixed_point-i8` 或 `dynamic_fixed_point-i16`。但一个模型最多同时只能存在两种量化方式。层名最好用双引号括起来，避免因为特殊字符导致解析失败。层名和量化类型之间的冒号后面要加上空格，否则 `yaml` 解析时可能会出错。
- 如果是将某些层从非量化改成量化，同样找到这些层，将它的层名加到

customized_quantize_layers 字典中，例如 “<layername>: asymmetric_affine-u8”。

- 如果是要修改量化参数，直接修改指定层的量化参数即可，customized_quantize_layers 置成空。

注：从 RKNN-Toolkit 1.6.0 版本开始，第一步生成的量化配置文件会给出一些混合量化建议，仅供参考。

第三步，生成 RKNN 模型。具体的接口调用流程如下：

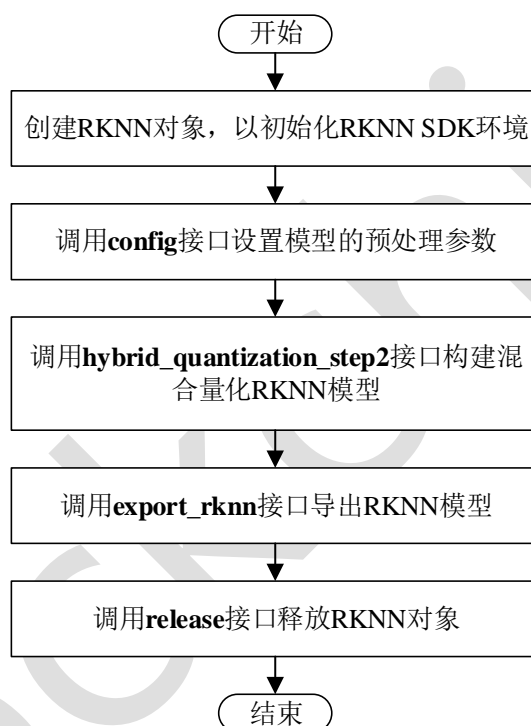


图 4-5-3-2 混合量化第三步接口调用流程

第四步，使用上一步生成的 RKNN 模型进行推理。

4.6 常见问题

准确性评估常见问题请参考文档：《Rockchip_Trouble_Shooting_RKNN_Toolkit_CN》相关章节。

5 性能评估

本节将详细说明 RKNN 模型的性能评估方法，优化方法和常见问题。

5.1 性能评估方法

RKNN Toolkit 提供 RKNN 模型性能评估接口 `eval_perf`。该接口可以评估 RKNN 模型在 Rockchip NPU 上推理时模型运行的耗时，也可以评估模型运行时每一层的耗时信息。

性能评估时涉及的主要接口和参数如下：

- `init_runtime`: 初始化运行时环境。初始化运行时环境指定的 `target` 和 `perf_debug` 参数决定了性能评估时的目标设备和是否评估每一层的耗时。
- `eval_perf`: 调用该接口，将根据 `init_runtime` 接口设置的参数，返回 RKNN 模型在指定设备上运行时的耗时，如果将 `perf_debug` 设置成 `True`，还将返回每一层的耗时信息。该接口的 `loop_cnt` 参数可以指定循环运行的次数。

注：

1. 如果是在模拟器上评估，则无论是否将 `perf_debug` 设置成 `True`，都会给出每一层的耗时信息。模拟器评估的性能可能与实际开发板的性能有差距，建议最终的模型性能评估在开发板上完成。
2. 如果 `perf_debug` 参数设置成 `True`，为了收集每一层的耗时信息，NPU 驱动会在每一层推理前后插入一些代码，导致总耗时比实际耗时高，这是正常现象。
3. 有关 `init_runtime` 和 `eval_perf` 接口的详细说明，请参考 7.7 章节和 7.9 章节。

5.2 性能评估示例

请参考如下代码使用 RKNN Toolkit 接口完成性能评估：


```

from rknn.api import RKNN

RKNN_PATH = './resnet18.rknn'

def eval_perf_with_simulator():
    # 创建 RKNN 对象
    rknn = RKNN()

    # 从当前目录加载 RKNN 模型 resnet_18
    ret = rknn.load_rknn(path=RKNN_PATH)
    if ret != 0:
        print('Load Pytorch model failed!')
        exit(ret)

    # 初始化运行时环境
    ret = rknn.init_runtime()
    if ret != 0:
        print('Init runtime environment failed')
        exit(ret)

    # 调用 eval_perf 接口进行性能评估
    rknn.eval_perf()

    # 释放 RKNN 对象
    rknn.release()

def eval_perf_with_rk1808():
    # 创建 RKNN 对象
    rknn = RKNN()

    # 从当前目录加载 RKNN 模型 resnet_18
    ret = rknn.load_rknn(path=RKNN_PATH)
    if ret != 0:
        print('Load Pytorch model failed!')
        exit(ret)

    # 初始化运行时环境
    ## 默认 perf_debug 为 False,如果要打印每一层耗时,设置该参数的值为 True
    ret = rknn.init_runtime(target='rk1808', device_id='1808', perf_debug=True)
    if ret != 0:
        print('Init runtime environment failed')
        exit(ret)

    # 调用 eval_perf 接口统计模型运行耗时,loop_cnt 指定循环次数,返回的是每帧的平
    均耗时
    rknn.eval_perf(loop_cnt=100)

    # 释放 RKNN 对象
    rknn.release()

```

```

if __name__ == '__main__':

    # 使用模拟器进行评估(只能在 x86_64 Linux 上使用)
    # eval_perf_with_simulator()

    # 使用 RK1808 进行性能评估
    eval_perf_with_rk1808()

```

5.3 性能评估结果说明

模拟器评估结果和开发板上的评估结果略有区别，以下小节分别对它们的评估结果进行说明。

5.3.1 模拟器性能评估结果说明

模拟器性能评估结果示例如下：

Layer ID	Name	Time(us)
3	convolution.relu.pooling.layer2_2	238
4	pooling.layer2_3	240
7	convolution.relu.pooling.layer2_2	126
8	convolution.relu.pooling.layer2_2	140
11	convolution.relu.pooling.layer2_2	84
14	convolution.relu.pooling.layer2_2	141
15	convolution.relu.pooling.layer2_2	141
18	convolution.relu.pooling.layer2_2	84
21	convolution.relu.pooling.layer2_2	192
22	convolution.relu.pooling.layer2_2	131
24	convolution.relu.pooling.layer2_2	37
27	convolution.relu.pooling.layer2_2	52
30	convolution.relu.pooling.layer2_2	131
31	convolution.relu.pooling.layer2_2	131
34	convolution.relu.pooling.layer2_2	52
37	convolution.relu.pooling.layer2_2	220
38	convolution.relu.pooling.layer2_2	184
40	convolution.relu.pooling.layer2_2	32
43	convolution.relu.pooling.layer2_2	38
46	convolution.relu.pooling.layer2_2	184
47	convolution.relu.pooling.layer2_2	184
50	convolution.relu.pooling.layer2_2	38
53	convolution.relu.pooling.layer2_2	430
54	convolution.relu.pooling.layer2_2	710
56	convolution.relu.pooling.layer2_2	51
59	convolution.relu.pooling.layer2_2	63
62	convolution.relu.pooling.layer2_2	710

63	convolution.relu.pooling.layer2_2	710
66	convolution.relu.pooling.layer2_2	63
67	pooling.layer2	17
69	fullyconnected.relu.layer_3	57
Total Time(us): 5611		
FPS(600MHz): 133.67		
FPS(800MHz): 178.22		
Note: Time of each layer is converted according to 800MHz!		

模拟器性能评估结果由以下内容组成：

- 逐层耗时：逐层耗时包括层的 ID，层名和该层耗时（按 800MHz NPU 频率换算）组成；
- Total Time：总耗时，单位为微秒；
- FPS(600MHz)：根据 600MHz NPU 频率换算得到的帧率；
- FPS(800MHz)：根据 800MHz NPU 频率换算得到的帧率。

注：

1. 使用模拟器进行性能评估时，模拟的 NPU 是由模型的 `target_platform` 决定的，该字段在模型转换阶段由 `config` 的 `target_platform` 指定。
2. 使用模拟器评估的性能可能和实际性能有偏差，建议用开发板进行评估。

5.3.2 开发板性能评估结果说明

当 `perf_debug` 设为 `False` 时，只打印平均推理耗时（Average inference Time, 单位为 us）和对应帧率。

举例如下：

=====
Average inference Time(us): 5438.98
FPS: 183.86
=====

当 `perf_debug` 设为 `True` 时，性能评估结果由以下内容组成：

- 逐层耗时：逐层耗时包括层 ID，层名，算子名称，UID 和该层耗时组成
- 总耗时：逐层耗时累加的结果
- FPS：帧率

`perf_debug` 设为 `True` 时，开发板性能评估结果举例如下：

Layer ID	Name	Operator	Uid	Time(us)
2	convolution_at_input0.1_1_1_2	CONVOLUTION	1	2739
0	max_pooling_at_input.10_4_4_0	POOLING	4	482
3	convolution_at_input.14_5_5_2	CONVOLUTION	5	315
4	convolution_at_input.13_8_8_2	CONVOLUTION	8	263
23	add_at_input.15_10_10_2	CONVOLUTION	10	236
5	convolution_at_input.17_12_12_2	CONVOLUTION	12	252
6	convolution_at_input.19_15_15_2	CONVOLUTION	15	238
24	add_at_input.20_17_17_2	CONVOLUTION	17	234
7	convolution_at_input.21_19_19_2	CONVOLUTION	19	336
9	convolution_at_input.23_22_22_2	CONVOLUTION	22	239
8	convolution_at_input.24_24_24_0	RESHUFFLE	24	444
		CONVOLUTION		
25	add_at_input.25_26_26_2	CONVOLUTION	26	186
10	convolution_at_input.26_28_28_2	CONVOLUTION	28	238
11	convolution_at_input.28_31_31_2	CONVOLUTION	31	236
26	add_at_input.29_33_33_2	CONVOLUTION	33	181
12	convolution_at_input.30_35_35_2	CONVOLUTION	35	310
14	convolution_at_input.32_38_38_2	CONVOLUTION	38	261
13	convolution_at_input.33_40_40_0	RESHUFFLE	40	447
		CONVOLUTION		
27	add_at_input.34_42_42_2	CONVOLUTION	42	184
15	convolution_at_input.35_44_44_2	CONVOLUTION	44	260
16	convolution_at_input.37_47_47_2	CONVOLUTION	47	255
28	add_at_input.38_49_49_2	CONVOLUTION	49	122
17	convolution_at_input.6_51_51_2	CONVOLUTION	51	359
19	convolution_at_input.5_54_54_2	CONVOLUTION	54	556
18	convolution_at_input.8_56_56_0	RESHUFFLE	56	387
		CONVOLUTION		
29	add_at_input.9_58_58_2	CONVOLUTION	58	487
20	convolution_at_input.3_60_60_2	CONVOLUTION	60	739
21	convolution_at_input.1_63_63_2	CONVOLUTION	63	666
30	add_at_input.2_65_65_2	CONVOLUTION	65	202
1	avg_pooling_at_x.1_67_67_1	POOLING	67	417
22	linear_at_539_70_69_0	FULLYCONNECTED	69	388
Total Time(us): 12659				
FPS: 79.00				

5.4 常见性能优化方法

1. 如果模型初始化时间太长，请参考 3.4.4 章节，对 RKNN 模型进行预编译。
2. 如果是用 RKNN C API 进行部署，发现设置输入或获取推理结果时耗时占比过长，请参

考文档《Rockchip_User_Guide_RKNN_API_CN.pdf》进行优化。

3. 还可以参考《Rockchip_Trouble_Shooting_RKNN_Toolkit》文档中关于卷积神经网络的设计建议，对网络结构进行优化，提高模型在 Rockchip NPU 上的性能。

Rockchip

6 内存评估

本节详细说明如何利用 RKNN Toolkit 接口获取 RKNN 运行时的内存消耗情况。

6.1 评估方法

RKNN Toolkit 提供 RKNN 模型内存评估接口 `eval_memory`。该接口可以评估 RKNN 模型在 Rockchip NPU 上推理时内存的使用情况。

注：内存评估时，必须连接 Rockchip NPU 开发板。

6.2 评估示例

```
from rknn.api import RKNN

RKNN_PATH = './resnet18.rknn'

def eval_mem_with_rk1808():
    # 创建 RKNN 对象
    rknn = RKNN()

    # 从当前目录加载 RKNN 模型 resnet18
    ret = rknn.load_rknn(path=RKNN_PATH)
    if ret != 0:
        print('Load Pytorch model failed!')
        exit(ret)

    # 初始化运行时环境
    ## 设置 eval_mem 为 True,进入内存评估模式
    ret = rknn.init_runtime(target='rk1808', device_id='1808', eval_mem=True)
    if ret != 0:
        print('Init runtime environment failed')
        exit(ret)

    # 调用 eval_memory 接口统计模型运行时内存使用情况
    rknn.eval_memory()

    # 释放 RKNN 对象
    rknn.release()

if __name__ == '__main__':
```

```
# 使用 RK1808 进行内存评估
eval_mem_with_rk1808()
```

6.3 内存评估结果说明

内存评估接口 `eval_memory` 返回 RKNN 模型运行时的内存使用情况，封装在字典 `memory_detail` 中。内存使用情况举例如下：

```
{
    'system_memory', {
        'maximum_allocation': 128000000,
        'total_allocation': 152000000
    },
    'npu_memory', {
        'maximum_allocation': 30000000,
        'total_allocation': 40000000
    },
    'total_memory', {
        'maximum_allocation': 158000000,
        'total_allocation': 192000000
    }
}
```

内存评估结果说明：

- **system_memory**：非 NPU 驱动分配的系统内存，包括为模型，输入数据等在系统中分配的内存。
- **npu_memory**：表示 NPU 驱动在运行期间使用的内存。
- **total_memory**：**system_memory** 和 **npu_memory** 的总和。
- **maximum_allocation**：内存使用的峰值，单位是 **Byte**。表示从模型运行开始到结束内存的最大分配值。
- **total_allocation**：表示 RKNN 模型运行期间分配的所有内存之和。

7 API 详细说明

本章详细说明 RKNN Toolkit 各个接口的使用方法。

7.1 RKNN 初始化及对象释放

在使用 RKNN-Toolkit 的所有 API 接口时，都需要先调用 `RKNN()` 方法初始化 RKNN 对象，并
不再使用该对象时，调用该对象的 `release()` 方法进行释放。

初始化 RKNN 对象时，可以设置 `verbose` 和 `verbose_file` 参数，以打印详细的日志信息。其中
`verbose` 参数指定是否要在屏幕上打印详细日志信息；如果设置了 `verbose_file` 参数，且 `verbose` 参
数值为 `True`，日志信息还将写到该参数指定的文件中。如果出现 Error 级别的错误，而 `verbose_file`
又被设为 `None`，则错误日志将自动写到 `log_feedback_to_the_rknn_toolkit_dev_team.log` 文件中。

反馈错误信息给 Rockchip NPU 团队时，建议反馈完整的错误日志。

举例如下：

```
# 将详细的日志信息输出到屏幕，并写到 mobilenet_build.log 文件中
rknn = RKNN(verbose=True, verbose_file='./mobilenet_build.log')
# 只在屏幕打印详细的日志信息
rknn = RKNN(verbose=True)
...
rknn.release()
```


7.2 RKNN 模型配置

在构建 RKNN 模型之前，需要先对模型进行通道均值、通道顺序、量化类型等的配置，这些操作可以通过 `config` 接口进行配置。

API	config
描述	设置模型参数
参数	batch_size : 批处理大小，默认值为 100。量化时将根据该参数决定每一批次参与运算的数据量，以校正量化结果。如果 <code>dataset</code> 中的数据量小于 <code>batch_size</code> ，则该参数值将自动调整为 <code>dataset</code> 中的数据量。如果量化时出现内存不足的问题，建议将这个值设小一点，例如 8。
	mean_values : 输入的均值。该参数与 <code>channel_mean_value</code> 参数不能同时设置。参数格式是一个列表，列表中包含一个或多个均值子列表，多输入模型对应多个子列表，每个子列表的长度与该输入的通道数一致，例如 <code>[[128,128,128]]</code> ，表示一个输入的三个通道的值减去 128。如果 <code>reorder_channel</code> 设置成 <code>'2 1 0'</code> ，则优先做通道调整，再做减均值。
	std_values : 输入的归一化值。该参数与 <code>channel_mean_value</code> 参数不能同时设置。参数格式是一个列表，列表中包含一个或多个归一化值子列表，多输入模型对应多个子列表，每个子列表的长度与该输入的通道数一致，例如 <code>[[128,128,128]]</code> ，表示设置一个输入的三个通道的值减去均值后再除以 128。如果 <code>reorder_channel</code> 设置成 <code>'2 1 0'</code> ，则优先做通道调整，再减均值和除以归一化值。
	epochs : 量化时的迭代次数，每迭代一次，就选择 <code>batch_size</code> 指定数量的图片进行量化校正。默认值为 -1，此时 RKNN-Toolkit 会根据 <code>dataset</code> 中的图片数量自动计算迭代次数以最大化利用数据集中的数据。
	reorder_channel : 表示是否需要对图像通道顺序进行调整，只对三通道输入有效。 <code>'0 1 2'</code> 表示按照输入的通道顺序来推理，比如图片输入时是 RGB，那推理的时候就根据 RGB 顺序传给输入层； <code>'2 1 0'</code> 表示会对输入做通道转换，比如输入时通道顺序是 RGB，

	<p>推理时会将其转成 BGR，再传给输入层，同样的，输入时通道的顺序为 BGR 的话，会被转成 RGB 后再传给输入层。如果有多个输入，每个输入的参数以“#”进行分隔，如 '0 1 2#0 1 2'。该参数的默认值是 None，对于 Caffe 框架的三通道输入模型，表示需要做通道顺序的调整，其他框架的三通道输入模型，默认不做通道顺序调整。</p>
	<p>need_horizontal_merge: 是否需要进行水平合并，默认值为 False。如果模型是 inception v1/v3/v4，建议开启该选项，可以提高推理时的性能。</p>
	<p>quantized_dtype: 量化类型，目前支持的量化类型有 asymmetric_quantized-u8、dynamic_fixed_point-i8、dynamic_fixed_point-i16，默认值为 asymmetric_quantized-u8。</p>
	<p>quantized_algorithm: 量化参数优化算法。当前版本支持的算法有：normal，mmse 和 kl_divergence，默认值为 normal。其中 normal 算法的特点是速度较快。而 mmse 算法，因为需要对量化参数进行多次调整，其速度会慢很多，但通常能得到比 normal 算法更高的精度；kl_divergence 所用时间会比 normal 多一些，但比 mmse 会少很多，在某些场景下可以得到较好的改善效果。</p>
	<p>mmse_epoch: mmse 量化算法的迭代次数，默认值为 3。通常情况下，迭代次数越多，精度往往越高。</p>
	<p>optimization_level: 模型优化等级。通过修改模型优化等级，可以关掉部分或全部模型转换过程中使用到的优化规则。该参数的默认值为 3，打开所有优化选项。值为 2 或 1 时关闭一部分可能会对部分模型精度产生影响的优化选项，值为 0 时关闭所有优化选项。</p>
	<p>target_platform: 指定 RKNN 模型目标运行平台。目前支持 RK1806、RK1808、RK3399Pro、RV1109 和 RV1126。其中基于 RK1806、RK1808 或 RK3399Pro 生成的 RKNN 模型可以在这三个平台上通用，基于 RV1109 或 RV1126 生成的 RKNN 模型可以在这两个平台通用。如果模型要在 RK1806、RK1808 或 RK3399Pro 上运行，该参数的值可以是 ["rk1806"], ["rk1808"], ["rk3399pro"] 或 ["rk1806", "rk1808", "rk3399pro"] 等；如果模型要在 RV1109 或 RV1126 上运行，该参数的值可以是 ["rv1126"], ["rv1109"] 或 ["rv1109", "rv1126"] 等。这个参数的值不可以是类似 ["rk1808",</p>

	<p>“rv1126”]这样的组合，因为这两款芯片互不兼容。如果不填该参数，则默认是[“rk1808”]，生成的 RKNN 模型可以在 RK1806、RK1808 和 RK3399Pro 平台上运行。该参数的值大小写不敏感。</p>
	<p>quantize_input_node: 开启后无论模型是否量化，均强制对模型的输入节点进行量化。输入节点被量化的模型，在部署时会有性能优势，rknn_input_set 接口的耗时更少。当 RKNN-Toolkit 量化没有按理想情况对输入节点进行量化（仅支持输入为图片的模型）、或用户选择加载深度学习框架已生成的量化模型时可以启用（这种情况下，第一层的 quantize_layer 会被合并到输入节点）。默认值为 False。</p>
	<p>merge_dequant_layer_and_output_node: 将模型输出节点与上一层的 dequantize_layer，合并成一个被量化的输出节点，允许模型在部署时返回 uint8 或 float 类型的推理结果。此配置仅对加载深度学习框架已生成的量化模型有效。默认为 False。</p>
返回值	无

举例如下：

```
# model config
rknn.config(mean_values=[[103.94, 116.78, 123.68]],
            std_values=[[58.82, 58.82, 58.82]],
            reorder_channel='0 1 2',
            need_horizontal_merge=True,
            target_platform=['rk1808', 'rk3399pro'])
```

7.3 模型加载

RKNN-Toolkit 目前支持 Caffe, Darknet, Keras, MXNet, ONNX, PyTorch, TensorFlow, 和 TensorFlow Lite 等模型的加载转换，这些模型在加载时需调用对应的接口，以下为这些接口的详细说明。

7.3.1 Caffe 模型加载接口

API	load_caffe
描述	加载 caffe 模型
参数	model: caffe 模型文件（.prototxt 后缀文件）所在路径。
	proto: caffe 模型的格式（可选值为'caffe'或'lstm_caffe'）。为了支持 RNN 模型，增加了相关网络层的支持，此时需要设置 caffe 格式为'lstm_caffe'。
	blobs: caffe 模型的二进制数据文件（.caffemodel 后缀文件）所在路径。该参数值可以为 None，RKNN-Toolkit 将随机生成权重等参数。
返回值	0: 导入成功
	-1: 导入失败

举例如下：

```
# 从当前路径加载 mobilenet_v2 模型
ret = rknn.load_caffe(model='./mobilenet_v2.prototxt',
                      proto='caffe',
                      blobs='./mobilenet_v2.caffemodel')
```

7.3.2 Darknet 模型加载接口

API	load_darknet
描述	加载 Darknet 模型
参数	model: Darknet 模型文件（.cfg 后缀）所在路径。
	weight: 权重文件（.weights 后缀）所在路径
返回值	0: 导入成功
	-1: 导入失败

举例如下：

```
# 从当前目录加载 yolov3-tiny 模型
ret = rknn.load_darknet(model='./yolov3-tiny.cfg',
                        weight='./yolov3.weights')
```

7.3.3 Keras 模型加载接口

API	load_keras
描述	加载 Keras 模型
参数	model: Keras 模型文件（后缀为.h5）。必填参数。
	convert_engine: 转换引擎，可以是'Keras'或'tflite'。默认转换引擎为 Keras。可选参数。
返回值	0: 导入成功
	-1: 导入失败

举例如下：

```
# 从当前目录加载 xception 模型
ret = rknn.load_keras(model='./xception_v3.h5')
```

7.3.4 MXNet 模型加载接口

API	load_mxnet
描述	加载 MXNet 模型
参数	symbol: MXNet 模型的网络结构文件，后缀是 json。必填参数。
	params: MXnet 模型的参数文件，后缀是 params。必填参数。
	input_size_list: 每个输入节点对应的图片的尺寸和通道数。例如 [[1,224,224],[3,224,224]]表示有两个输入，其中一个输入的 shape 是[1,224,224]，另外一个输入的 shape 是[3,224,224]。必填参数。
返回值	0: 导入成功； -1: 导入失败

举例如下：

```
# 从当前目录加载 resnext50 模型
ret = rknn.load_mxnet(symbol='resnext50_32x4d-symbol.json',
                        params='resnext50_32x4d-4ecf62e2.params',
                        input_size_list=[[3,224,224]] )
```

7.3.5 ONNX 模型加载

API	load_onnx
描述	加载 ONNX 模型
参数	model: ONNX 模型文件 (.onnx 后缀) 所在路径。
	inputs: 指定模型的输入节点, 数据类型为列表。例如示例中的 resnet50v2 模型, 其输入节点是['data']。默认值是 None, 此时工具自动从模型中查找输入节点。可选参数。
	input_size_list: 每个输入节点对应的数据形状。例如示例中的 resnet50v2 模型, 其输入节点对应的输入尺寸是[[3, 224, 224]]。可选参数。 注: 1. 填写输入数据形状时不要填 batch 维。如果要批量推理, 请使用 build 接口的 rknn_batch_size 参数。 2. 如果指定了 inputs 节点, 则该参数必须填写。
	outputs: 指定模型的输出节点, 数据类型为列表。例如示例中的 resnet50v2 模型, 其输出节点是['resnetv24_dense0_fwd']。默认值是 None, 此时工具将自动从模型中搜索输出节点。可选参数。
返回值	0: 导入成功
	-1: 导入失败

举例如下:

```
# 从当前目录加载 resnet50v2 模型
ret = rknn.load_onnx(model = './resnet50v2.onnx',
                    inputs = ['data'],
                    input_size_list = [[3, 224, 224]],
                    outputs=['resnetv24_dense0_fwd'])
```

7.3.6 PyTorch 模型加载接口

API	load_pytorch
描述	加载 PyTorch 模型
参数	model: PyTorch 模型文件 (.pt 后缀) 所在路径, 而且需要是 torchscript 格式的模型。必填参数。
	input_size_list: 每个输入节点对应的图片的尺寸和通道数。例如 [[1,224,224],[3,224,224]]表示有两个输入, 其中一个输入的 shape 是[1,224,224], 另外一个输入的 shape 是[3,224,224]。必填参数。
返回值	0: 导入成功
	-1: 导入失败

举例如下:

```
# 从当前目录加载 resnet18 模型
ret = rknn.Load_pytorch(model = './resnet18.pt',
                        input_size_list=[[3,224,224]])
```

7.3.7 TensorFlow 模型加载接口

API	load_tensorflow
描述	加载 TensorFlow 模型
参数	tf_pb: TensorFlow 模型文件 (.pb 后缀) 所在路径。
	inputs: 模型输入节点, 支持多个输入节点。所有输入节点名放在一个列表中。
	input_size_list: 每个输入节点对应的数据形状。如示例中的 mobilenet-v1 模型, 其输入节点对应的输入尺寸是[[224, 224, 3]]。
	outputs: 模型的输出节点, 支持多个输出节点。所有输出节点名放在一个列表中。
	predef_file: 为了支持一些控制逻辑, 需要提供一个 npz 格式的预定义文件。可以通过以下方法生成预定义文件: np.savez('prd.npz', placeholder_name=prd_value)。如果“placeholder_name”中包含',', 请用'#'替换。
返回值	0: 导入成功
	-1: 导入失败

举例如下:

```
# 从当前目录加载 ssd_mobilenet_v1_coco_2017_11_17 模型
ret = rknn.load_tensorflow(
    tf_pb='./ssd_mobilenet_v1_coco_2017_11_17.pb',
    inputs=['FeatureExtractor/MobilenetV1/MobilenetV1/Conv2d_0
           /BatchNorm/batchnorm/mul_1'],
    outputs=['concat', 'concat_1'],
    input_size_list=[[300, 300, 3]])
```


7.3.8 TensorFlow Lite 模型加载接口

API	load_tflite
描述	<p>加载 TensorFlow Lite 模型。</p> <p>注：</p> <p>因为 tflite 不同版本的 schema 之间是互不兼容的,所以构建的 tflite 模型使用与 RKNN-Toolkit 不同版本的 schema 可能导致加载失败。目前 RKNN-Toolkit 使用的 tflite schema 是基于官网 master 分支上的提交:0c4f5dfea4ceb3d7c0b46fc04828420a344f7598。官网地址如下:</p> <p>https://github.com/tensorflow/tensorflow/commits/master/tensorflow/lite/schema/schema.ubs</p>
参数	model: TensorFlow Lite 模型文件 (.tflite 后缀) 所在路径
返回值	0: 导入成功
	-1: 导入失败

举例如下:

```
# 从当前目录加载 mobilenet_v1 模型
ret = rknn.load_tflite(model = './mobilenet_v1.tflite')
```

7.4 构建 RKNN 模型

API	build
描述	依照加载的模型结构及权重数据，构建对应的 RKNN 模型。
参数	<p>do_quantization: 是否对模型进行量化，值为 True 或 False。</p> <p>dataset: 量化校正数据的数据集。目前支持文本文件格式，用户可以把用于校正的图片（jpg 或 png 格式）或 npy 文件路径放到一个.txt 文件中。文本文件里每一行一条路径信息。如：</p> <p>a.jpg b.jpg 或 a.npy b.npy</p> <p>如有多个输入，则每个输入对应的文件用空格隔开，如：</p> <p>a.jpg a2.jpg b.jpg b2.jpg 或 a.npy a2.npy b.npy b2.npy</p> <p>pre_compile: 模型预编译开关。预编译 RKNN 模型可以减少模型初始化时间，但是无法通过模拟器进行推理或性能评估。如果 NPU 驱动有更新，预编译模型通常也需要重新构建。</p> <p>注：</p> <ol style="list-style-type: none">1. 该选项只在 Linux x86_64 平台上有效。2. RKNN-Toolkit-V1.0.0 及以上版本生成的预编译模型不能在 NPU 驱动版本小于 0.9.6 的设备上运行；RKNN-Toolkit-V1.0.0 以前版本生成的预编译模型不能在

	NPU 驱动版本大于等于 0.9.6 的设备上运行。驱动版本号可以通过 <code>get_sdk_version</code> 接口查询。
	<p><code>rknn_batch_size</code>: 模型的输入 Batch 参数调整, 默认值为 1。如果大于 1, 则可以在一次推理中同时推理多帧输入图像或输入数据, 如 MobileNet 模型的原始 input 维度为 [1, 224, 224, 3], output 维度为 [1, 1001], 当 <code>rknn_batch_size</code> 设为 4 时, input 的维度变为 [4, 224, 224, 3], output 维度变为 [4, 1001]。</p> <p>注:</p> <ol style="list-style-type: none"> 1. <code>rknn_batch_size</code> 的调整并不会提高一般模型在 NPU 上的执行性能, 但却会显著增加内存消耗以及增加单帧的延迟。 2. <code>rknn_batch_size</code> 的调整可以降低超小模型在 CPU 上的消耗, 提高超小模型的平均帧率。(适用于模型太小, CPU 的开销大于 NPU 的开销)。 3. <code>rknn_batch_size</code> 的值建议小于 32, 避免内存占用太大而导致推理失败。 4. <code>rknn_batch_size</code> 修改后, 模型的 input/output 维度会被修改, 使用 inference 推理模型时需要设置相应的 input 的大小, 后处理时, 也需要对返回的 outputs 进行处理。
返回值	0: 构建成功
	-1: 构建失败

注: 如果在执行脚本前设置 `RKNN_DRAW_DATA_DISTRIBUTE` 环境变量的值为 1, 则 RKNN-Toolkit 会将每一层的 `weight/bias` (如果有) 和输出数据的直方图保存在当前目录下的 `dump_data_distribute` 文件夹中。使用该功能时推荐只在 `dataset.txt` 中存放单独一个 (组) 输入。

举例如下:

```
# 构建 RKNN 模型, 并且做量化
ret = rknn.build(do_quantization=True, dataset='./dataset.txt')
```

7.5 导出 RKNN 模型

通过该接口导出 RKNN 模型文件, 用于模型部署。

API	export_rknn
描述	将 RKNN 模型保存到指定文件中（.rknn 后缀）。
参数	export_path: 导出模型文件的路径。
返回值	0: 导出成功
	-1: 导出失败

举例如下：

```
.....
# 将构建好的 RKNN 模型保存到当前路径的 mobilenet_v1.rknn 文件中
ret = rknn.export_rknn(export_path = './mobilenet_v1.rknn')
.....
```

7.6 加载 RKNN 模型

API	load_rknn
描述	加载 RKNN 模型。
参数	path: RKNN 模型文件路径。
	load_model_in_npu: 是否直接加载 npu 中的 rknn 模型。其中 path 为 rknn 模型在 npu 中的路径。只有当 RKNN-Toolkit 运行在 RK3399Pro Linux 开发板或连有 NPU 设备的 PC 上时才可以设为 True。默认值为 False。
返回值	0: 加载成功
	-1: 加载失败

举例如下：

```
# 从当前路径加载 mobilenet_v1.rknn 模型
ret = rknn.load_rknn(path='./mobilenet_v1.rknn')
```

7.7 初始化运行时环境

在模型推理或性能评估之前，必须先初始化运行时环境，明确模型在的运行平台（具体的目标

硬件平台或软件模拟器）。

API	init_runtime
描述	初始化运行时环境。确定模型运行的设备信息（硬件平台信息、设备 ID）；性能评估时是否启用 debug 模式，以获取更详细的性能信息。
参数	target: 目标硬件平台，目前支持“rk3399pro”、“rk1806”、“rk1808”、“rv1109”、“rv1126”。默认为 None，即在 PC 使用工具时，模型在模拟器上运行，在 RK3399Pro Linux 开发板运行时，模型在 RK3399Pro 自带 NPU 上运行，否则在设定的 target 上运行。其中“rk1808”包含了 TB-RK1808 AI 计算棒。
	device_id: 设备编号，如果 PC 连接多台设备时，需要指定该参数，设备编号可以通过“list_devices”接口查看。默认值为 None。 注：MAC OS X 系统当前版本还不支持多个设备。
	perf_debug: 进行性能评估时是否开启 debug 模式。在 debug 模式下，可以获取到每一层的运行时间，否则只能获取模型运行的总时间。默认值为 False。
	eval_mem: 是否进入内存评估模式。进入内存评估模式后，可以调用 eval_memory 接口获取模型运行时的内存使用情况。默认值为 False。
	async_mode: 是否使用异步模式。调用推理接口时，涉及设置输入图片、模型推理、获取推理结果三个阶段。如果开启了异步模式，设置当前帧的输入将与推理上一帧同时进行，所以除第一帧外，之后的每一帧都可以隐藏设置输入的时间，从而提升性能。在异步模式下，每次返回的推理结果都是上一帧的。该参数的默认值为 False。
返回值	0：初始化运行时环境成功。
	-1：初始化运行时环境失败。

举例如下：

```
# 初始化运行时环境
ret = rknn.init_runtime(target='rk1808', device_id='012345789AB')
if ret != 0:
    print('Init runtime environment failed')
    exit(ret)
```

7.8 模型推理

在进行模型推理前，必须先构建或加载一个 RKNN 模型。

API	inference
描述	<p>对当前模型进行推理，返回推理结果。</p> <p>如果 RKNN-Toolkit 运行在 PC 上，且初始化运行环境时设置 target 为 Rockchip NPU 设备，得到的是模型在硬件平台上的推理结果。</p> <p>如果 RKNN-Toolkit 运行在 PC 上，且初始化运行环境时没有设置 target，得到的是模型在模拟器上的推理结果。模拟器可以模拟 RK1808,也可以模拟 RV1126,具体模拟哪款芯片,取决于 RKNN 模型的 target_platform 参数值。</p> <p>如果 RKNN-Toolkit 运行在 RK3399Pro Linux 开发板上，得到的是模型在实际硬件上的推理结果。</p>
参数	<p>inputs: 待推理的输入，如经过 cv2 处理的图片。格式是 ndarray list。</p> <p>data_type: 输入数据的类型，可填以下值：'float32', 'float16', 'int8', 'uint8', 'int16'。默认值为'uint8'。</p> <p>data_format: 数据模式，可以填以下值："nchw", "nhwc"。默认值为'nhwc'。这两个的不同之处在于 channel 放置的位置。</p> <p>inputs_pass_through: 将输入透传给 NPU 驱动。非透传模式下，在将输入传给 NPU 驱动之前，工具会对输入进行减均值、除方差等操作；而透传模式下，不会做这些操作。这个参数的值是一个数组，比如要透传 input0，不透传 input1，则这个参数的值为[1, 0]。默认值为 None，即对所有输入都不透传。</p>
返回值	results: 推理结果，类型是 ndarray list。

对于分类模型,如 `mobilenet_v1`,模型推理代码如下(完整代码参考 `example/tflite/mobilenet_v1`):

```
# 使用模型对图片进行推理, 输出 TOP5
.....
outputs = rknn.inference(inputs=[img])
show_outputs(outputs)
.....
```

输出的 TOP5 结果如下:

```
-----TOP 5-----
[156]: 0.85107421875
[155]: 0.09173583984375
[205]: 0.01358795166015625
[284]: 0.006465911865234375
[194]: 0.002239227294921875
```

7.9 模型性能评估

API	eval_perf
描述	<p>评估模型性能。</p> <p>模型运行在 PC 上，初始化运行环境时不指定 <code>target</code>，得到的是模型在模拟器上运行的性能数据，包含逐层的运行时间及模型完整运行一次需要的时间。模拟器可以模拟 RK1808,也可以模拟 RV1126,具体模拟哪款芯片,取决于 RKNN 模型的 <code>target_platform</code> 参数值。</p> <p>模型运行在与 PC 连接的 Rockchip NPU 上，且初始化运行环境时设置 <code>perf_debug</code> 为 <code>False</code>，则获得的是模型在硬件上运行的总时间；如果设置 <code>perf_debug</code> 为 <code>True</code>，除了返回总时间外，还将返回每一层的耗时情况。</p> <p>模型运行在 RK3399Pro Linux 开发板上时，如果初始化运行环境时设置 <code>perf_debug</code> 为 <code>False</code>，获得的也是模型在硬件上运行的总时间；如果设置 <code>perf_debug</code> 为 <code>True</code>，返回总时间及每一层的耗时情况</p>
参数	<p><code>loop_cnt</code>: 指定 RKNN 模型推理次数,用于计算平均推理时间。该参数只在 <code>init_runtime</code> 中的 <code>target</code> 为非模拟器，且 <code>perf_debug</code> 设成 <code>False</code> 时生效。该参数数据类型为整型，默认值为 1。</p>
返回值	<p><code>perf_result</code>: 性能评估结果，详细说明请参考 5.3 章节。</p>

举例如下：

```
# 对模型性能进行评估
.....
rknn.eval_perf(inputs=[image], is_print=True)
.....
```


7.10 获取内存使用情况

API	eval_memory
描述	<p>获取模型在硬件平台运行时的内存使用情况。</p> <p>模型必须运行在与 PC 连接的 Rockchip NPU 设备上, 或直接运行在 RK3399Pro Linux 开发板上。</p> <p>注:</p> <p>1. 使用该功能时, 对应的驱动版本必须要大于等于 0.9.4。驱动版本可以通过 <code>get_sdk_version</code> 接口查询。</p>
参数	<code>is_print</code> : 是否以规范格式打印内存使用情况。默认值为 <code>True</code> 。
返回值	<ul style="list-style-type: none">● <code>memory_detail</code>: 内存使用情况。详细说明请参考 6.3 章节。

举例如下:

```
# 对模型内存使用情况进行评估
.....
memory_detail = rknn.eval_memory()
.....
```

如 `example/tflite` 中的 `mobilenet_v1`, 它在 RK1808 上运行时内存占用情况如下:

```
=====
                        Memory Profile Info Dump
=====

System memory:
    maximum allocation : 22.65 MiB
    total allocation   : 72.06 MiB
NPU memory:
    maximum allocation : 33.26 MiB
    total allocation   : 34.57 MiB

Total memory:
    maximum allocation : 55.92 MiB
    total allocation   : 106.63 MiB

INFO: When evaluating memory usage, we need consider
the size of model, current model size is: 4.10 MiB
=====
```

7.11 混合量化

7.11.1 hybrid_quantization_step1

使用混合量化功能时，第一阶段调用的主要接口是 `hybrid_quantization_step1`，用于生成模型结构文件（`{model_name}.json`）、权重文件（`{model_name}.data`）和量化配置文件（`{model_name}.quantization.cfg`）。接口详情如下：

API	hybrid_quantization_step1
描述	根据加载的原始模型，生成对应的模型结构文件、权重文件和量化配置文件。
参数	<p>dataset: 量化校正数据的数据集。目前支持文本文件格式，用户可以把用于校正的图片（jpg 或 png 格式）或 npy 文件路径放到一个.txt 文件中。文本文件里每一行一条路径信息。如：</p> <p>a.jpg b.jpg 或 a.npy b.npy</p>
返回值	0: 成功
	-1: 失败

举例如下：

```
# Call hybrid_quantization_step1 to generate quantization config
.....
ret = rknn.hybrid_quantization_step1(dataset='./dataset.txt')
.....
```

7.11.2 hybrid_quantization_step2

使用混合量化功能时，生成混合量化 RKNN 模型阶段调用的主要接口是 `hybrid_quantization_step2`。接口详情如下：

API	hybrid_quantization_step2
描述	接收模型结构文件、权重文件、量化配置文件、校正数据集作为输入，生成混合量化后的 RKNN 模型。
参数	model_input : 第一步生成的模型结构文件，例如 “{model_name}.json”。数据类型为字符串。必填参数。
	data_input : 第一步生成的权重数据文件，例如 “{model_name}.data”。数据类型为字符串。必填参数。
	model_quantization_cfg : 经过修改后的模型量化配置文件，例如 “{model_name}.quantization.cfg”。数据类型为字符串。必填参数。
	dataset : 量化校正数据的数据集。目前支持文本文件格式，用户可以把用于校正的图片（jpg 或 png 格式）或 npy 文件路径放到一个.txt 文件中。文本文件里每一行一条路径信息。如： a.jpg b.jpg 或 a.npy b.npy
	pre_compile : 模型预编译开关。预编译 RKNN 模型可以减少模型初始化时间，但是无法通过模拟器进行推理或性能评估。如果 NPU 驱动有更新，预编译模型通常也需要重新构建。 注： 1. 该选项不能在 RK3399Pro Linux 开发板 / Windows PC / Mac OS X PC 上使用。 2. RKNN-Toolkit-V1.0.0 及以上版本生成的预编译模型不能在 NPU 驱动版本小于 0.9.6 的设备上运行；RKNN-Toolkit-V1.0.0 以前版本生成的预编译模型不能在 NPU 驱动版本大于等于 0.9.6 的设备上运行。驱动版本号可以通过 get_sdk_version 接口查询。

返回值	0: 成功
	-1: 失败

举例如下：

```
# Call hybrid_quantization_step2 to generate hybrid quantized RKNN model
.....
ret = rknn.hybrid_quantization_step2(
    model_input='./ssd_mobilenet_v2.json',
    data_input='./ssd_mobilenet_v2.data',
    model_quantization_cfg='./ssd_mobilenet_v2.quantization.cfg',
    dataset='./dataset.txt')
.....
```

7.12 量化精度分析

API	accuracy_analysis
描述	<p>逐层对比浮点模型和量化模型的输出结果，输出余弦距离和欧式距离，用于分析量化模型精度下降原因。</p> <p>注：</p> <ol style="list-style-type: none">该接口在 build 或 hybrid_quantization_step1 或 hybrid_quantization_step2 之后调用，并且原始模型应该为浮点模型，否则会调用失败。该接口使用的量化方式与 config 中指定的一致。
参数	inputs : 包含输入图像或数据的数据集文本文件（与量化校正数据集 dataset 格式相同，但只能包含一组输入）。
	output_dir : 输出目录，所有快照都保存在该目录。该目录内容的详细说明见 4.3.3 章节。
	calc_qnt_error : 是否计算量化误差（默认为 True ）。
	target : 指定设备类型。如果指定 target ，在逐层量化误差分析时，将连接到 NPU 上获取每一层的真实结果，跟浮点模型结果相比较。这样可以更准确的反映实际运行时的误差。
	device_id : 如果 PC 连接了多个 NPU 设备，需要指定具体的设备编号。
返回值	dump_file_type : 精度分析过程中会输出模型每一层的结果，这个参数指定了输出文件的类型。有效值为 'tensor' 和 'numpy' ，默认值是 'tensor' 。如果指定数据类型为 'numpy' ，可以减少精度分析的耗时。
	0 : 成功
	-1 : 失败

注：如果在执行脚本前设置 **RKNN_DRAW_DATA_DISTRIBUTE** 环境变量的值为 1，则 RKNN-Toolkit 会将每一层的 **weight/bias**（如果有）和输出数据的直方图保存在当前目录下的 **dump_data_distribute** 文件夹中。

举例如下：

```
.....

print('--> config model')
rknn.config(channel_mean_value='0 0 0 1', reorder_channel='0 1 2')
print('done')

print('--> Loading model')
ret = rknn.load_onnx(model='./mobilenetv2-1.0.onnx')
if ret != 0:
    print('Load model failed! Ret = {}'.format(ret))
    exit(ret)
print('done')

# Build model
print('--> Building model')
ret = rknn.build(do_quantization=True, dataset='./dataset.txt')
if ret != 0:
    print('Build rknn failed!')
    exit(ret)
print('done')

print('--> Accuracy analysis')
rknn.accuracy_analysis(inputs='./dataset.txt', target='rk1808')
print('done')

.....
```

7.13 注册自定义算子

API	register_op
描述	注册自定义算子。
参数	op_path: 算子编译生成的 rknnop 文件的路径
返回值	无

参考代码如下所示。注意，该接口要在模型转换前调用。自定义算子的使用和开发请参考《Rockchip_Developer_Guide_RKNN_Toolkit_Custom_OP_CN》文档。

```
rknn.register_op('./resize_area/ResizeArea.rknnop')
```

```
rknn.load_tensorflow(...)
```

7.14 导出预编译模型（在线预编译）

构建 RKNN 模型时，可以指定预编译选项以导出预编译模型，这被称为离线编译。同样，RKNN-Toolkit 也提供在线编译的接口：`export_rknn_precompile_model`。使用该接口，可以将普通 RKNN 模型转成预编译模型。

API	<code>export_rknn_precompile_model</code>
描述	<p>经过在线编译后导出预编译模型。</p> <p>注：</p> <ol style="list-style-type: none">1. 使用该接口前必须先调用 <code>load_rknn</code> 接口加载普通 rknn 模型；2. 使用该接口前必须调用 <code>init_runtime</code> 接口初始化模型运行环境，<code>target</code> 必须是 RK NPU 设备，不能是模拟器；而且要设置 <code>rknn2precompile</code> 参数为 <code>True</code>。
参数	<code>export_path</code> : 导出模型路径。必填参数。
返回值	0: 成功
	-1: 失败

举例如下：

```
from rknn.api import RKNN

if __name__ == '__main__':
    # Create RKNN object
    rknn = RKNN()

    # Load rknn model
    ret = rknn.load_rknn('./test.rknn')
    if ret != 0:
        print('Load RKNN model failed.')
        exit(ret)

    # init runtime
    ret = rknn.init_runtime(target='rk1808', rknn2precompile=True)
    if ret != 0:
        print('Init runtime failed.')
        exit(ret)

    # Note: the rknn2precompile must be set True when call init_runtime
    ret = rknn.export_rknn_precompile_model('./test_pre_compile.rknn')
    if ret != 0:
        print('export pre-compile model failed.')
        exit(ret)

    rknn.release()
```


7.15 导出分段模型

该接口的功能是将普通的 RKNN 模型转成分段模型，分段的位置由用户指定。

API	export_rknn_sync_model
描述	在用户指定的模型层后面插入 sync 层，用于将模型分段，并导出分段后的模型。
参数	<p>input_model: 待分段的 rknn 模型路径。数据类型为字符串。必填参数。</p> <p>sync_uids: 待插入 sync 节点层的层 uid 列表, RKNN-Toolkit 将在这些层后面插入 sync 层。</p> <p>注:</p> <ol style="list-style-type: none">uid 只能通过 eval_perf 接口获取，且需要在 init_runtime 时设置 perf_debug 为 True, target 不能是模拟器。uid 的值不可以随意填写，一定需要是在 eval_perf 获取性能详情的 uid 列表中，否则可能出现不可预知的结果。 <p>output_model: 导出模型的保存路径。数据类型：字符串。默认值为 None，如果不填该参数，导出的模型将保存在 input_model 指定的文件中。</p>
返回值	<p>0: 成功</p> <p>-1: 失败</p>

举例如下：

```
from rknn.api import RKNN

if __name__ == '__main__':
    rknn = RKNN()
    ret = rknn.export_rknn_sync_model(
        input_model='./ssd_inception_v2.rknn',
        sync_uids=[206, 186, 152, 101, 96, 67, 18, 17],
        output_model='./ssd_inception_v2_sync.rknn')
    if ret != 0:
        print('export sync model failed.')
        exit(ret)
    rknn.release()
```

7.16 导出加密模型

该接口的功能是将普通的 RKNN 模型进行加密，得到加密后的模型。

API	export_encrypted_rknn_model
描述	根据用户指定的加密等级对普通的 RKNN 模型进行加密。
参数	input_model: 待加密的 RKNN 模型路径。数据类型为字符串。必填参数。
	output_model: 模型加密后的保存路径。数据类型为字符串。选填参数，如果不填该参数，则使用 {original_model_name}.crypt.rknn 作为加密后的模型的名字。
	crypt_level: 加密等级。等级越高，安全性越高，解密越耗时；反之，安全性越低，解密越快。数据类型为整型，默认值为 1，目前安全等级有 3 个等级，1，2 和 3。
返回值	0: 成功
	-1: 失败

举例如下：

```
from rknn.api import RKNN

if __name__ == '__main__':
    rknn = RKNN()
    ret = rknn.export_encrypted_rknn_model('test.rknn')
    if ret != 0:
        print('Encrypt RKNN model failed.')
        exit(ret)
    rknn.release()
```

7.17 查询 SDK 版本

API	get_sdk_version
描述	获取 SDK API 和驱动的版本号。 注：使用该接口前必须完成模型加载和初始化运行环境。且该接口只有在 target 是 Rockchip NPU 或 RKNN-Toolkit 运行在 RK3399Pro Linux 开发板才可以使用。
参数	无
返回值	sdk_version: API 和驱动版本信息。类型为字符串。

举例如下：

```
# 获取 SDK 版本信息
.....
sdk_version = rknn.get_sdk_version()
.....
```

返回的 SDK 信息如下：

```
=====
RKNN VERSION:
  API: 1.7.1 (566a9b6 build: 2021-10-28 14:53:41)
  DRV: 1.7.1 (566a9b6 build: 2021-11-12 20:24:57)
=====
```

7.18 获取设备列表

API	list_devices
描述	列出已连接的 RK3399PRO/RK1808/RV1109/RV1126 或 TB-RK1808 AI 计算棒。 注：目前设备连接模式有两种：ADB 和 NTB。其中 RK3399PRO 目前只支持 ADB 模式，TB-RK1808 AI 计算棒只支持 NTB 模式，RK1808/RV1109 支持 ADB/NTB 模式。 多设备连接时请确保他们的模式都是一样的。
参数	无。
返回值	返回 adb_devices 列表和 ntb_devices 列表，如果设备为空，则返回空列表。 例如我们的环境里插了两个 TB-RK1808 AI 计算棒，得到的结果如下： adb_devices = [] ntb_devices = ['TB-RK1808S0', '515e9b401c060c0b']

举例如下：

```
from rknn.api import RKNN

if __name__ == '__main__':
    rknn = RKNN()
    rknn.list_devices()
    rknn.release()
```

返回的设备列表信息如下（这里有两个 RK1808 开发板，它们的连接模式都是 adb）：

```
*****
all device(s) with adb mode:
['515e9b401c060c0b', 'XGOR2N4EZR']
*****
```

注：使用多设备时，需要保证它们的连接模式都是一致的，否则会引起冲突，导致设备通信失败。

7.19 查询模型可运行平台

API	list_support_target_platform
描述	查询给定 RKNN 模型可运行的芯片平台。
参数	rknn_model: RKNN 模型路径。如果不指定模型路径，则按类别打印 RKNN-Toolkit 当前支持的芯片平台。
返回值	support_target_platform: Returns runnable chip platforms, or None if the model path is empty.

参考代码如下所示：

```
rknn.list_support_target_platform(rknn_model='mobilenet_v1.rknn')
```

参考结果如下：

```
*****
Target platforms filled in RKNN model:      []
Target platforms supported by this RKNN model: ['RK1806', 'RK1808', 'RK3399PRO']
*****
```

8 附录

8.1 参考文档

OP 支持列表: 《RKNN_OP_Support.md》

快速上手指南: 《Rockchip_Quick_Start_RKNN_Toolkit_CN.pdf》

问题排查手册: 《Rockchip_Trouble_Shooting_RKNN_Toolkit_CN.pdf》

自定义 OP 使用指南: 《Rockchip_Developer_Guide_RKNN_Toolkit_Custom_OP_CN.pdf》

可视化功能使用指南: 《Rockchip_User_Guide_RKNN_Toolkit_Visualization_CN.pdf》

RKNN Toolkit Lite 使用指南: 《Rockchip_User_Guide_RKNN_Toolkit_Lite_CN.pdf》

以上文档均存放在 SDK/doc 目录中, 也可以访问以下链接查阅:

<https://github.com/rockchip-linux/rknn-toolkit/tree/master/doc>

8.2 问题反馈渠道

请通过 RKNN QQ 交流群, Github Issue 或瑞芯微 redmine 将问题反馈给 Rockchip NPU 团队。

RKNN QQ 交流群: 1025468710

Github issue: <https://github.com/rockchip-linux/rknn-toolkit/issues>

Rockchip Redmine: <https://redmine.rock-chips.com/>

注: Redmine 账号需要通过销售或业务人员开通。如果是第三方开发板, 请先找第三方厂商反馈问题。