地平线
Horizon Robotics

天工开物 J6算法工具链

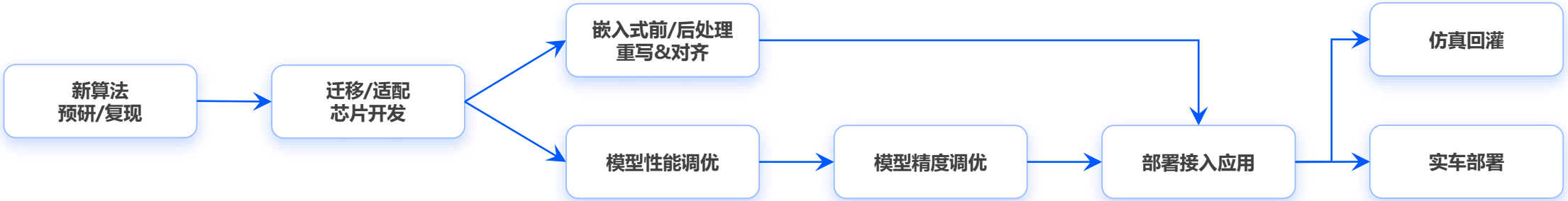
J6“芯”征程，打造领先高效率生产平台

OpenExplorer®：算法模型端侧部署开发套件

基于简单易用的使用链路和丰富的辅助工具，Jx 工具链确保用户算法准确、高效地完成部署

智驾算法开发的普遍痛点

针对以下普遍移植痛点，J6 工具链引入新特性进行优化



01

基础算子支持能力

容易导致模型转换编译不通过，降低新算法快速验证的效率

02

Transformer部署效率

当下先进智驾算法大量使用 Transformer，是否能高效支持 Transformer 将直接影响智驾算法方案的先进性

03

生产调试工具能力

调试信息、开放度不够，将影响开发者自主调优效率

04

部署精度调优

量化是高效部署的常见策略，但也意味着一定程度的精度损失，调优策略是否丰富直接影响调优难度和部署精度

05

算法效果验证

算法在应用中的指标验证依赖工程实现，周期长，迭代慢，并且往往存在许多一致性对齐工作

06

大规模仿真回灌

大规模仿真回灌处理的数据量大，仿真的效率直接影响整个应用开发/迭代周期的效率

J6算法工具链新特性

旨在解决智驾算法开发的普遍痛点，打造领先的高效率生产平台

极致性能优化

- 依托 BPU®Nash 架构，CNN 和 Transformer 提供高效支持
- 算子丰富度大幅提升，且由 BPU 直接加速
- 提供丰富参考算法，结合业务方案提出 J6 的高效模型结构

统一异构计算框架

- 强化 Host 仿真效率，提供 GPU 加速能力
- Host 仿真与芯片部署接口统一，降低维护代价
- 开放 BPU 自定义灵活计算，更适应复杂算法部署

混合精度量化

- 新增定点&浮点混合精度方案，浮点兜底可大幅降低精度调优难度
- 使用智驾模型库强化自动校准策略，大幅提升自动调优的应用范围
- 内置精度调优模板，为典型的算法提供预置的最佳量化配置

生产调试工具优化

- 性能&精度 Debug 工具，经过高/中/低阶全场景打磨，能力完备
- 开发 BPU 内部细粒度 Profiling 信息，开发者可更精准把握算法优化方向
- 新增运行时 Profiling，直接观测应用中动态调度带来的性能偏差

BPU®Nash架构

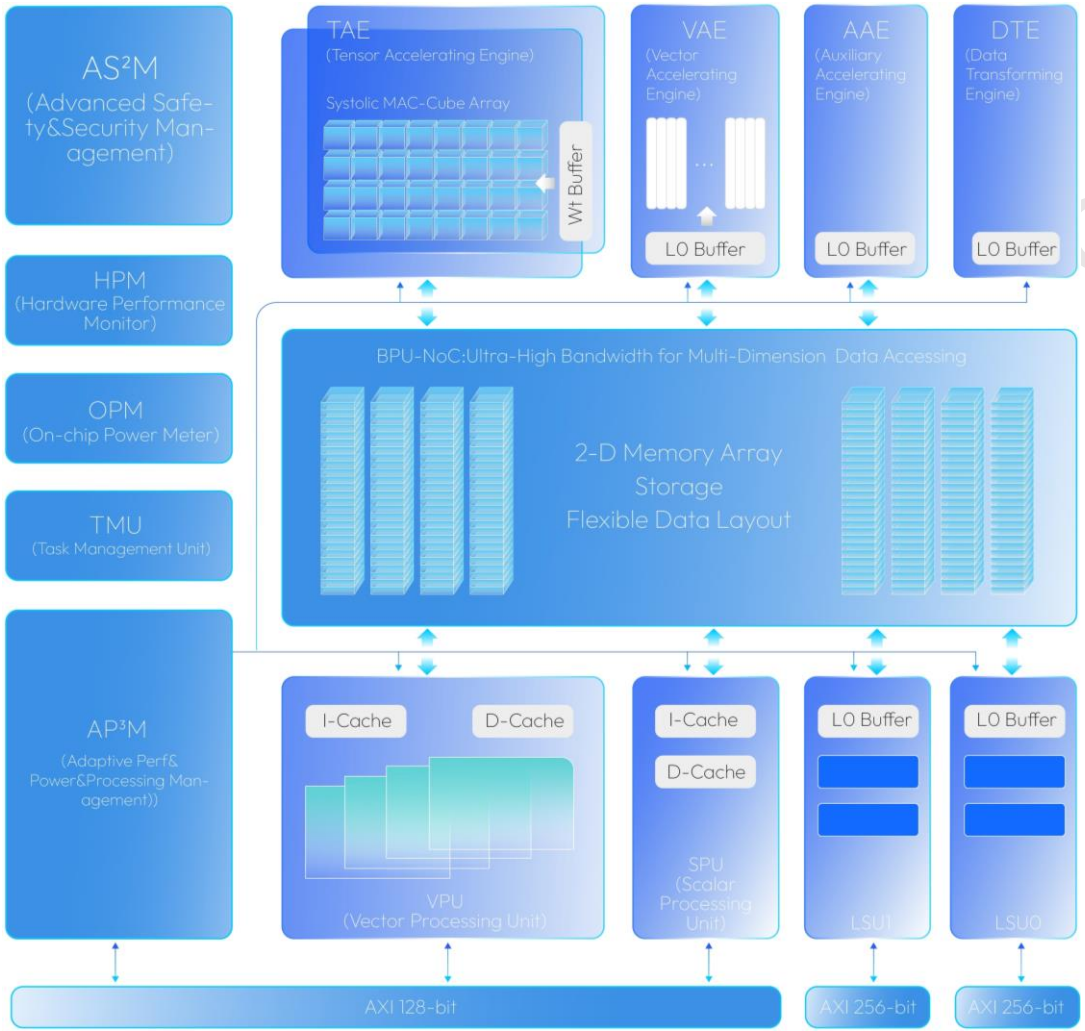
地平线 Horizon Robotics

三级存储架构
极致优化大参数下的带宽瓶颈
核间高效协同

多脉动立方加速引擎
灵活的引擎间数据流动
高能效、低宽带占用

紧耦合异构计算
高效加速不同类型数据处理

虚拟化Virtualization
透明式提升多任务并行处理能力



数据变换引擎
灵活支持Transformer细小算子

浮点向量加速单元
通用、灵活
关键算子精度需求

多向数据流动
核内、核间高效灵活的数据流动
动态调度，灵活调优

数据驱动功耗优化
针对神经网络数据动态范围特性
降低功耗30%

J6工具链算子范围

地平线 Horizon Robotics

160+

- 标准 ONNX 算子
- opset 版本支持 ≤20
- 算子范围可完整覆盖智驾常见算法

100%

- BPU 硬件加速

FP16

- 所有向量/标量 计算均可提供 FP16 高精度计算

算子

- 约束进一步放宽，无四维限制
- 支持定点模型在 GPU 环境加速
- 使用 DSP 补充非 NN 计算能力

算子支持

NN算子（BPU）

Conv	ConvTranspose	MatMul	Gemm	Mul	Add	Relu	Softmax
LayerNorm	MultiHeadAttention	Deformable Attention	ArgMax	AveragePool	BatchNormalization	Sigmoid	Transpose
Reshape	Resize	Abs	Atan	Ceil	Clip	Concat	Constant
ConstantOfShape	Cos	DepthToSpace	Div	InstanceNormalization	Elu	Equal	Exp
Expand	Flatten	Floor	Gather	GatherElements	GlobalAveragePool	GlobalMaxPool	Greater
LSTM	LeakyRelu	Less	Log	Max	MaxPool	Min	Pad
Pow	Reciprocal	ReduceMax	ReduceMean	ReduceSum	Shape	Sin	Size
Slice	Softplus	SpaceToDepth	Split	Sqrt	Squeeze	Sub	Sum
Tanh	Identity	Tile	TopK	Unsqueeze	Upsample	GridSample	Deformable conv
Shape	GlobalLpPool	voxel_pooling	bev_pooling	Dynamic Conv	Acos	Acosh	And
ArgMin	Asin	Asinh	Atanh	Cast	Cosh	CumSum	DequantizeLinear
Erf	Eyelike	GRU	HardSigmoid	...			

非NN算子（DSP）

FFT	IFFT	fir	Dilate	EqualizeHist	2D FFT	Lidar点云前处理	Erode
remap	resize	Filter2d	Flip	bosfilter	gausfilter	Iplacian_filter	BilateralFilter
GaussianBlur	Integral	粒子滤波	MedianBlur	BoxFilter	Canny	CvtColor	CornerHarris
RoiResize	Rotate	SepFilter2d	WarpPerspective	Sobel	Threshold	WarpAffine	...

J6工具链参考算法

系列化参考算法覆盖多场景，提供高效、易用、灵活、开放的使用体验

高效模型结构参考

提供适配征程芯片特性的高效算法模型结构设计

开箱即用

参考算法随工具链(OE)开发包释放，提供 Docker 开发环境，简单易用

加速算法设计

提供多个高效 Backbone 和场景算法加速自研效率

持续迭代更新

面向自动驾驶场景，跟踪前沿算法，不断增加参考

灵活可扩展

算法源码释放，灵活可修改、扩展、重训练

多传感器支持

覆盖视觉感知、激光雷达感知的多任务、BEV 参考

性能指标有验证

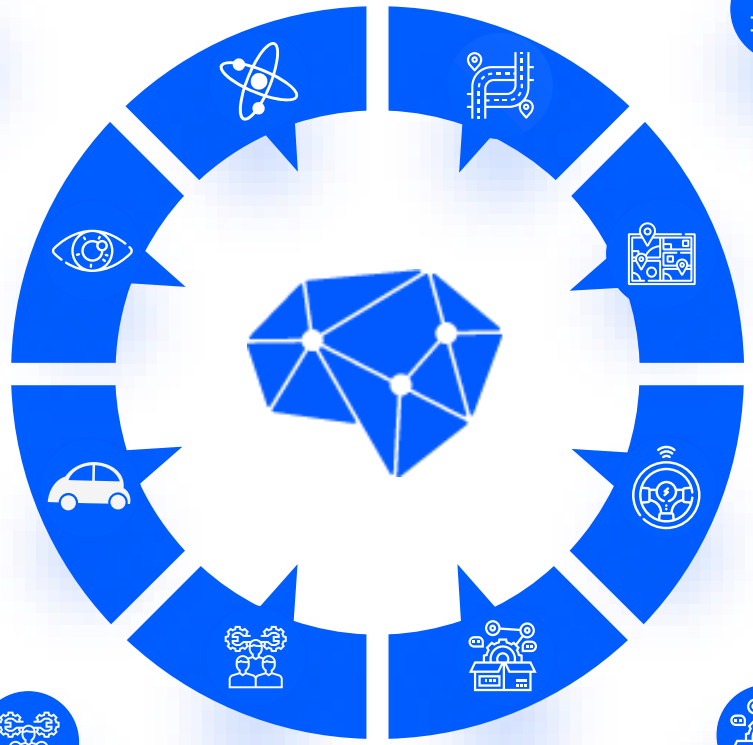
基于公开数据集，算法性能经过严格测试与验证

开放生态

所有参考算法在地平线开发者社区全面开放

J6工具链参考算法

结合智驾主流算法技术，持续积累在J6优化的高效率版本



基础Backbone

Vargnet、 MixVarGENet、
EfficientNet、HENET...

Transformer基础算法

SwinT、 ViT、DETR、
Deformable DETR、 ...

单/双目算法

FCOS、FCOS_3D、Unet、MOTR、
FasterSCNN、Deeplabv3+、
PWCNet、GANet、StereoNet、 ...

Lidar算法

Pointpillars、CenterPoint
Lidar_multitask、...

BEV检测

BEVFormer 、PETR、Sparse4D、DETR3D、
LSS、GKT、IPM、IPM_4D、...

BEV道路拓扑

MapTR、vectorMapNet、...

BEV空间占用

flashocc,、SurroundOcc,、TPVFormer、...

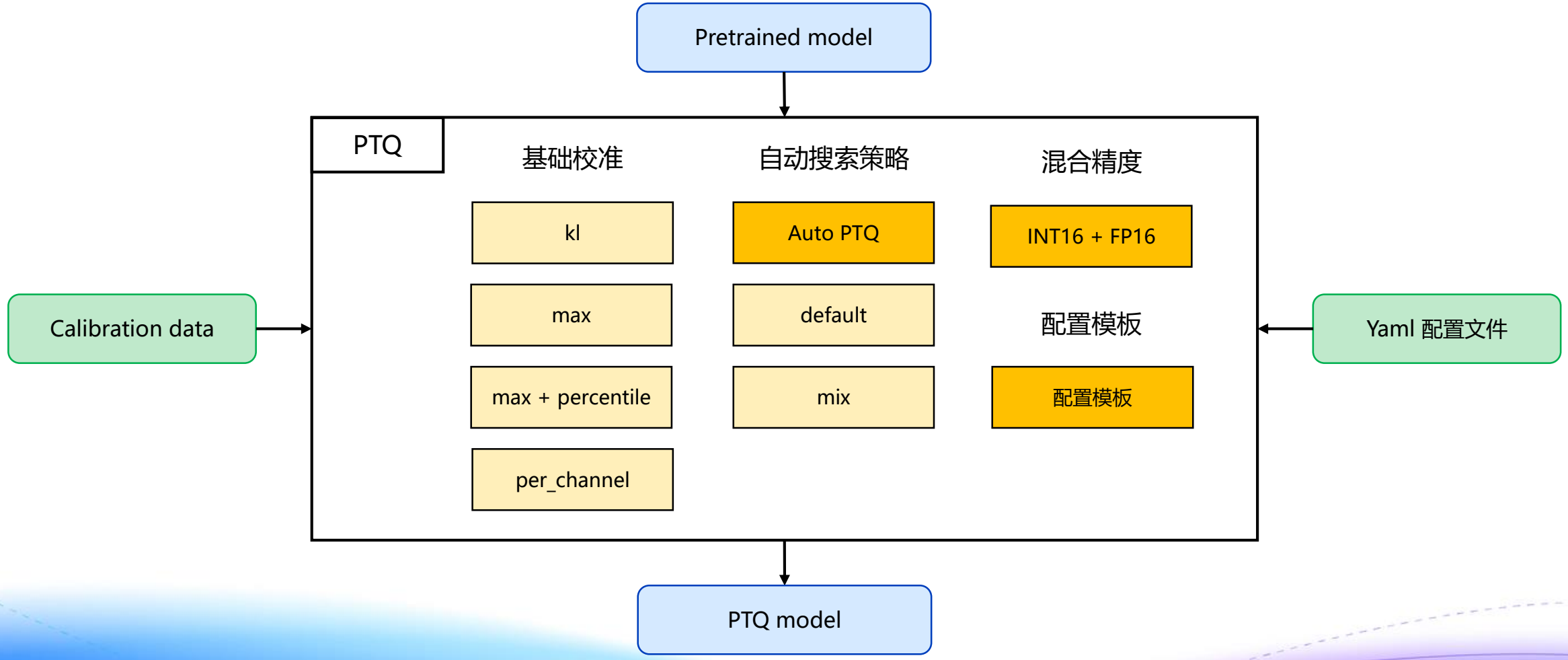
轨迹预测

MTR、QCNET、DenseTNT、...

** 仅做参考实现，非量产算法*

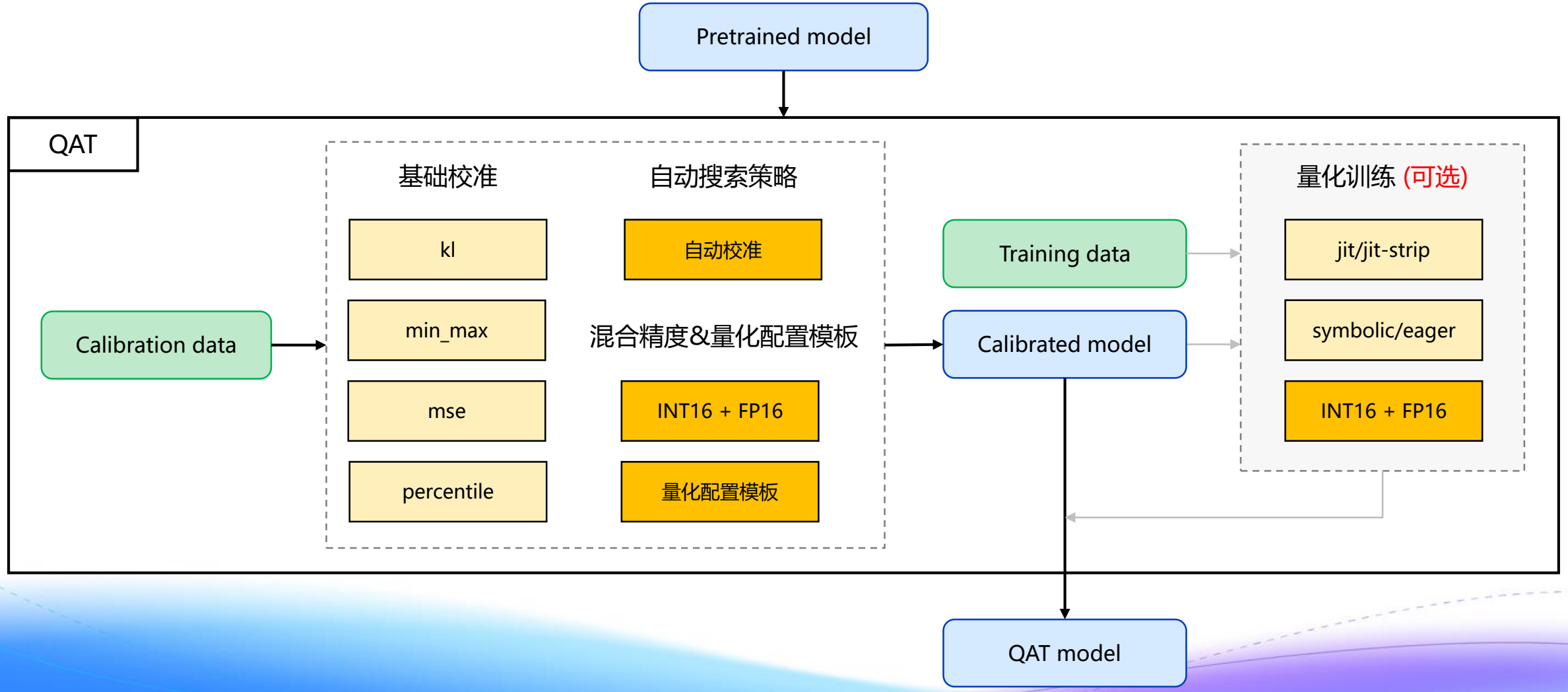
PTQ后量化路径

基础校准方式之上，提供易用的 default、mix 校准，以及 Auto PTQ、INT16+FP16 混合精度等



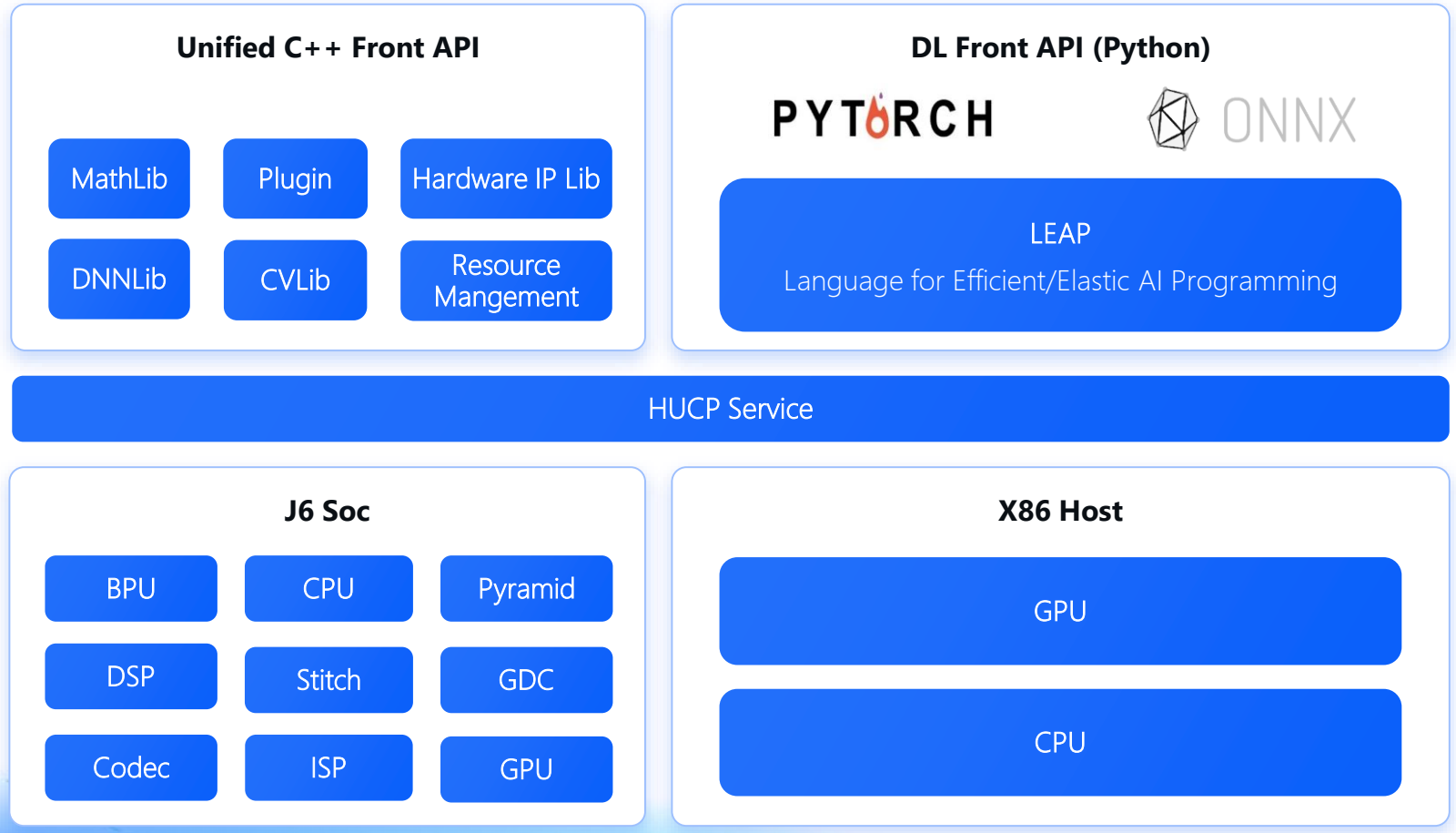
QAT量化训练路径

新增自动校准策略，支持自定义基础校准搜索范围，并支持 INT16+FP16 混合精度



统一异构计算框架

Horizon Unify Compute Platform

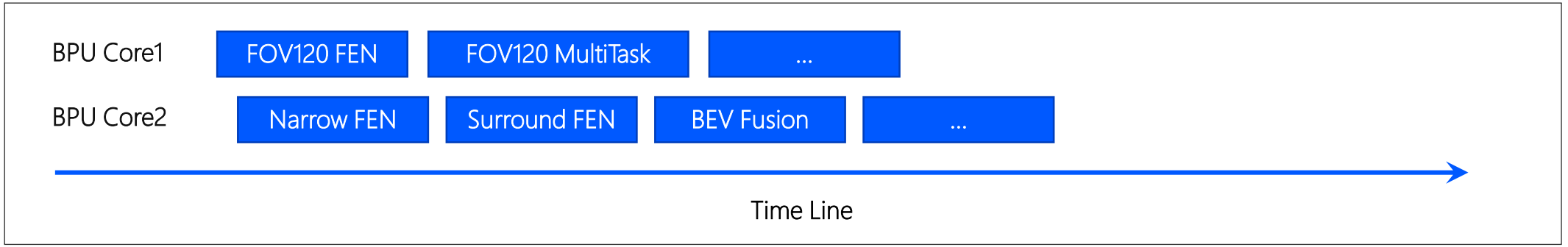


- HUCP Service 提供跨平台的能力抽象，封装了 BPU、CPU、DSP、Pyramid、GDC 等计算 IP，并通过统一的 C++ 前端接口提供 x86 Host 端与 SoC 端数值一致的计算能力
- 模型推理兼容 J5 平台 Runtime 预测库 LibDNN，同时支持仿真环境下的 GPU 加速
- DL Front API 开放 IR 层面的计算图控制能力和硬 IP 仿真能力

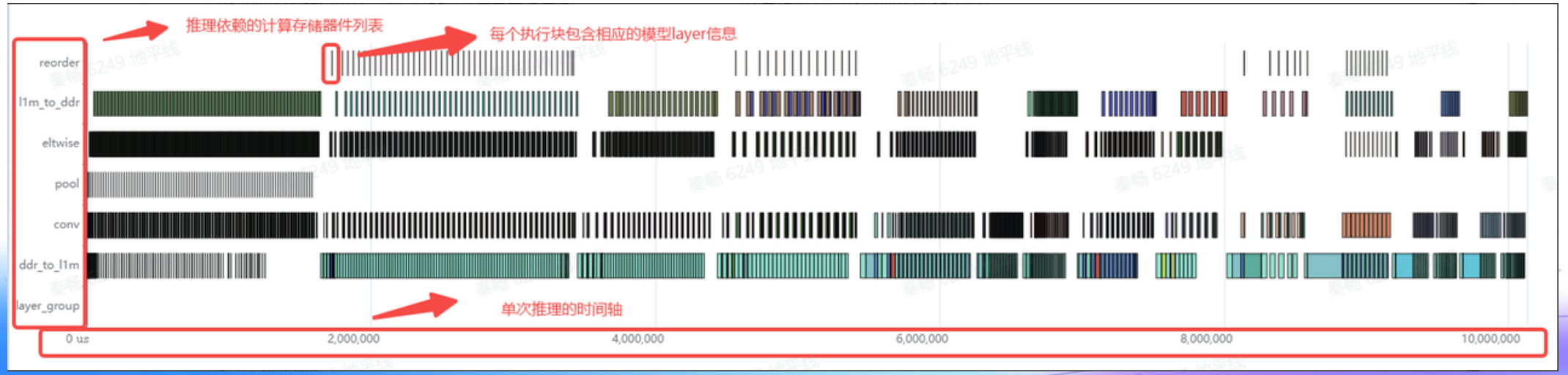
性能分析工具

J6 新增任务推理的时序信息，提供更佳的调试调优手段

- 可跟踪运行时状态中，每个模型推理发生的具体时序



- 可跟踪运行时状态中，每个模型推理在 BPU 内部发生的具体时序



J6 OE开发包目录

```
horizon_j6_open_explorer_v3.0.17-py310_20240705/
├── package
│   ├── board # 板端推理环境依赖包和工具
│   │   ├── hrt_model_exec -> ../../samples/ucp_tutorial/tools/hrt_model_exec
│   │   └── install.sh
│   └── host # x86端编译环境依赖包
│       ├── ai_toolchain
│       ├── install.sh
│       └── resolve.sh
├── README-CN
├── README-EN
├── resolve_all.sh # 一键下载开发包示例依赖的数据、模型
├── run_docker.sh # 一键起docker
└── samples
    ├── ai_toolchain
    │   ├── horizon_model_convert_sample # PTQ示例
    │   ├── horizon_model_train_sample # QAT示例
    │   └── model_zoo
    ├── model_zoo -> ai_toolchain/model_zoo
    └── ucp_tutorial # UCP示例
        ├── deps_aarch64
        ├── deps_x86
        ├── dnn # 板端模型推理示例等
        ├── hpl
        ├── tools # hrt源码
        └── vp
```

注：当前为试用版本内容，正式版本将会有更丰富内容。

J6 vs J5 环境版本差异

地平线 Horizon Robotics

开发环境

J6 Alpha版本	
python	3.10
torch	2.1.0+cu118
torchvision	0.16.0+cu118
ONNX	1.15.0
X86 GCC	11.4
Linaro GCC	11.4

J5 CS版本（1.1.74）	
python	3.8
torch	1.13
cuda	11.6
ONNX	1.8.0
X86 GCC	5.4
Linaro GCC	9.3

前端DL框架

J6 Alpha版本		
onnx	推荐版本	1.15.0
	opset	10-19
	ir_version	不高于9
torch	2.1.0+cu118	
caffe	1.0	

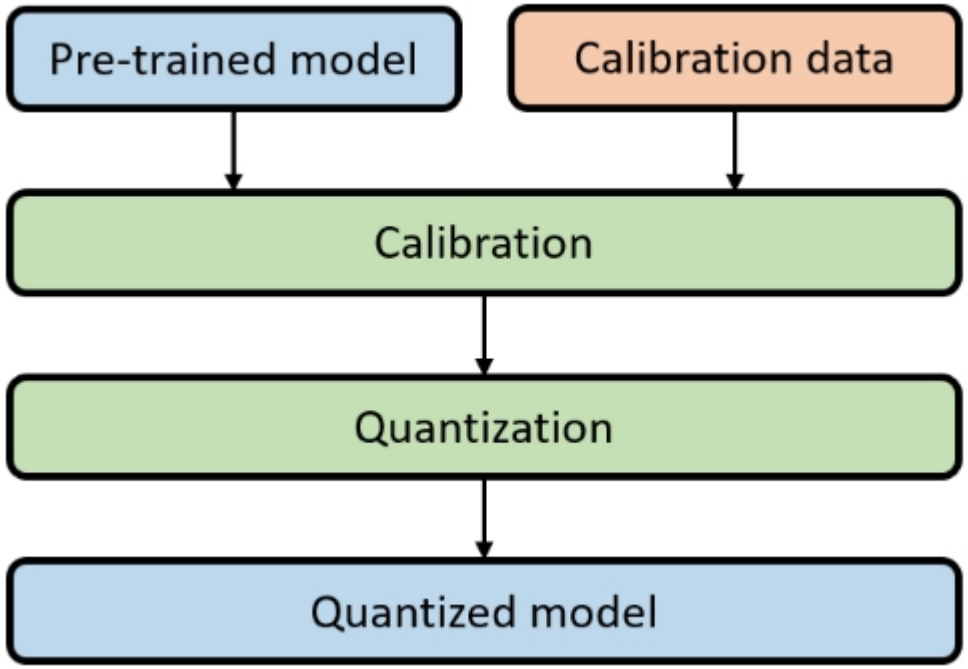
J5 CS版本（1.1.74）		
onnx	推荐版本	1.8.0
	opset	10、11
	ir_version	不高于7
torch	1.13.0+cu116	
caffe	1.0	

地平线
Horizon Robotics

天工开物 J6算法工具链

模型转换详解-PTQ

PTQ原理

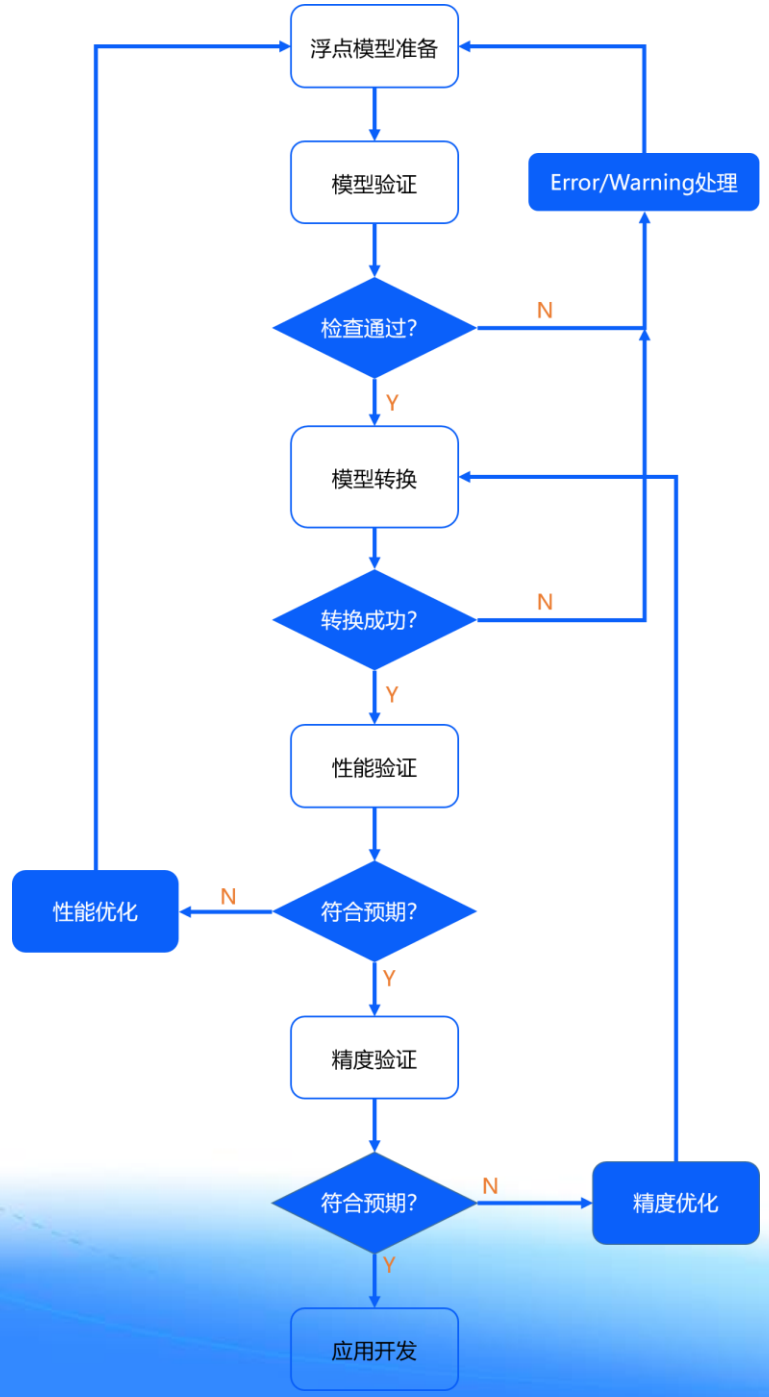


训练后量化 PTQ：

使用一批校准数据对训练好的模型进行校准，将训练过的FP32模型直接转换为定点计算的模型，过程中无需对原始模型进行任何训练。只对几个超参数调整就可完成量化过程， 且过程简单快速，无需训练，因此此方法已被广泛应用于大量的端侧和云侧部署场景， 我们优先推荐您尝试PTQ方法来查看是否满足您的部署精度和性能要求

模型转换详解-PTQ

转换流程 | 主要工具 | 精度验证&调优 | 性能评测



转换基本流程

地平线 Horizon Robotics

浮点模型准备：

samples/ai_toolchain/horizon_model_convert_sample路径下丰富示例，运行00_init.sh脚本获取模型以及校准数据；私有模型请导出Caffe或ONNX格式

模型验证：

```
hb_compile --march ${march} \
           --proto ${caffe_proto} \
           --model ${caffe_model/onnx_model} \
           --input-shape input.0 1x1x224x224
           --input-shape input.1 1x1x224x224
           --input-shape input.2 1x1x224x224
```

(可选参数)

模型转换：

```
hb_compile --fast-perf --model ${caffe_model/onnx_model} \
           --proto ${caffe_proto} \
           --march ${march} \
           --input-shape ${input_node_name} ${input_shape}
```

快速性能评测

(可选参数)

```
hb_compile --config ${config_file}
```

传统转换编译

转换配置文件生成

```
hb_compile --config ${config_file}
```

1. 传统转换编译，手动配置文件

1. 生成最简yaml配置文件：

```
hb_config_generator --simple-yaml
```

2. 生成基于模型信息的最简yaml配置文件：

```
hb_config_generator --simple-yaml --model model.onnx --march nash-e
```

3. 包含全部参数默认值的yaml配置文件：

```
hb_config_generator --full-yaml
```

4. 生成基于模型信息的全部参数默认值yaml配置文件：

```
hb_config_generator --full-yaml --model model.onnx --march nash-e
```

2. 使用工具生成配置文件模板

```
hb_compile --fast-perf --model ${caffe_model/onnx_model} \
           --proto ${caffe_proto} \
           --march ${march} \
           --input-shape ${input_node_name} ${input_shape}
```

3. 可以使用hb_compile --fast-perf模式下生成的模板yaml，保存在.fast_perf文件夹下

转换配置文件解读

yaml**参数** 除列出的参数外，其他参数配置与J5完全一致	**新增参数**	seperate_batch：沿第一维将输入节点做拆分，用于片上部署时不同batch的数据可以来源于不同的地址空间的场景。目前只支持onnx模型本身是单输入且第一维为1时，结合input_batch使用。配置取值True/False
		quant_config：支持通过json方式从模型、op type、以及单个op三个维度去灵活配置量化bit。配置取值json文件路径
	变动参数	march：nash-e或nash-m依次对应J6E和J6M处理器
		input_layout_rt：J6编译出的模型输入输出节点shape将保持与浮点完全一致，不再支持通过该参数做输入节点layout的修改
		calibration_type：该参数不再支持“load”模式，即J6不再支持load QAT训练后导出的onnx模型
		optimize_level：优化等级可选范围为 O0 ~ O2

注：J6 工具链 PTQ 链路的校准数据和原始 onnx 模型的推理数据保持完全一致，
因此需要用户自行完成数据的预处理，无论是否配置norm_type参数。

转换配置文件解读-quant_config

地平线 Horizon Robotics

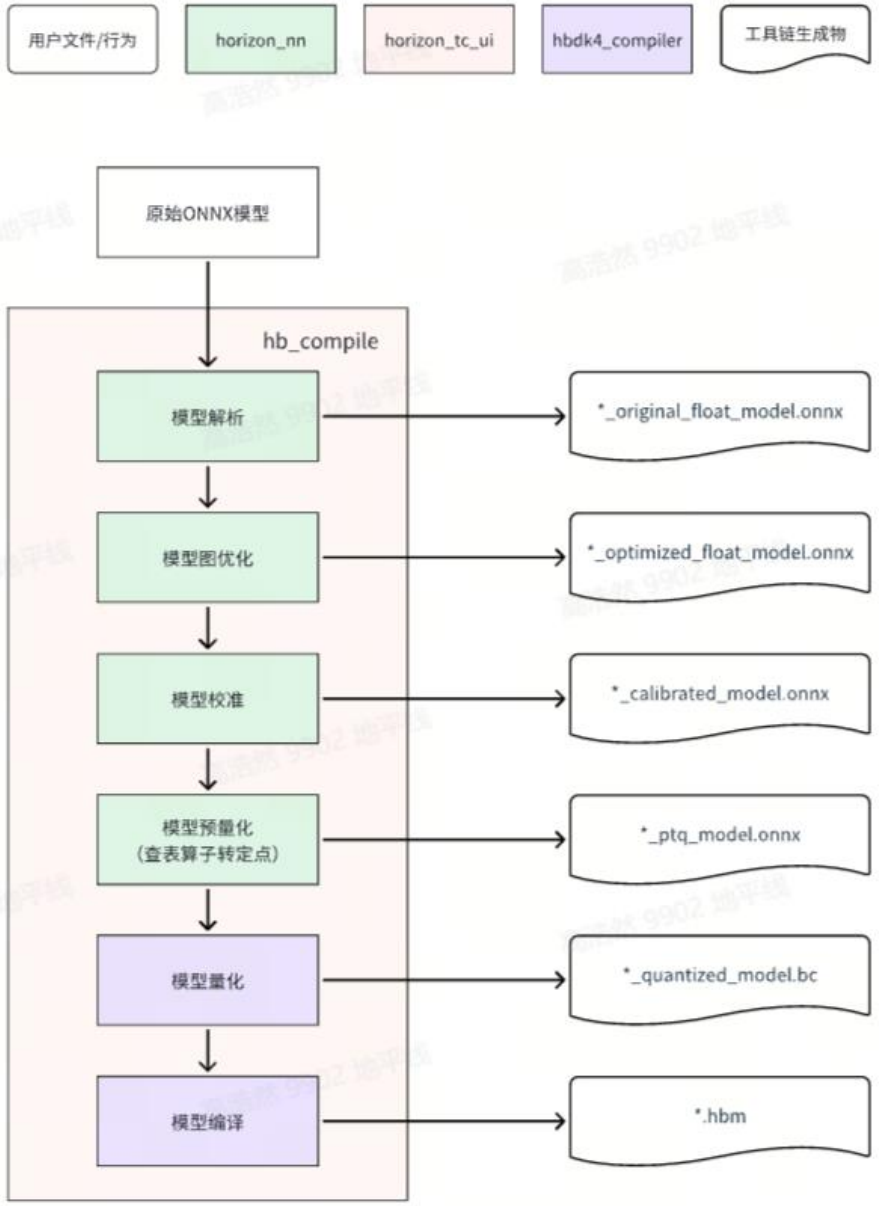
配置参数说明：

一级参数	二级参数	三级参数	是否必填	说明
model_config	all_node_type	无	optional	一次性将模型中所有节点的输入设为指定类型，可选配置int16/float16。
	model_output_type	无	optional	将模型的输出tensor设为指定类型，可选配置int8/int16。
op_config	NodeKind	type	optional	配置某一类型的节点的输入数据类型，可选配置int8/int16/float16。
node_config	NodeName	type	optional	配置指定名称的节点的输入数据类型，可选配置int8/int16/float16。

配置示例：

```
{
    // 配置模型层面的参数
    "model_config":{
        // 一次性配置所有节点的输入数据类型
        "all_node_type":"int16/float16",
        // 配置模型输出的数据类型
        "model_output_type":"int8/int16"
    },
    // 配置某一类型节点的参数，将op_name修改为节点类型名，例如"Conv","Add","Softmax"
    "op_config":{
        // 配置某一类型节点的输入数据类型
        "op_name1":{"type":"int8/int16/float16"},
        "op_name2":{"type":"int8/int16/float16"}
    },
    // 配置某个节点的参数,,将node_name修改为节点名称，例如"Conv_0","Add_1"...
    "node_config":{
        // 配置某个节点的输入数据类型
        "node_name1":{"type":"int8/int16/float16"},
        "node_name2":{"type":"int8/int16/float16"}
    }
}
```

转换生成物解读



- *.html：模型的静态性能评估文件（可读性更好）；
- *.json：模型的静态性能评估文件；
- *_node_info.csv：文件内保存了转换成功后算子的余弦相似度等信息的结果，与hb_compile成功执行后控制台输出的相似度信息相同；
- *_quant_info.json：文件内记录了算子的校准量化参数信息；
- *_advice.json：文件内保存了模型编译过程中，地平线模型编译器op checker打印的结果，包含算子的后端信息。

Node	NodeType	ON	Threshold	Calibrated Cosine	Quantized Cosine	Output Data Type
Conv_0+Relu_1	Conv+Relu	BPU	1.0	0.999958	0.999428	si8
MaxPool_2	MaxPool	BPU	1.11039	0.999858	0.999575	si8
Conv_3+Relu_4	Conv+Relu	BPU	1.11039	0.999572	0.998684	si8
Conv_5+Relu_6	Conv+Relu	BPU	0.653097	0.998671	0.995936	si8
Conv_8+Relu_9	Conv+Relu	BPU	1.11039	0.999907	0.999137	si8
Conv_12+Relu_13	Conv+Relu	BPU	0.388216	0.997999	0.997073	si8
Conv_14+Relu_15	Conv+Relu	BPU	0.43385	0.989948	0.981367	si8

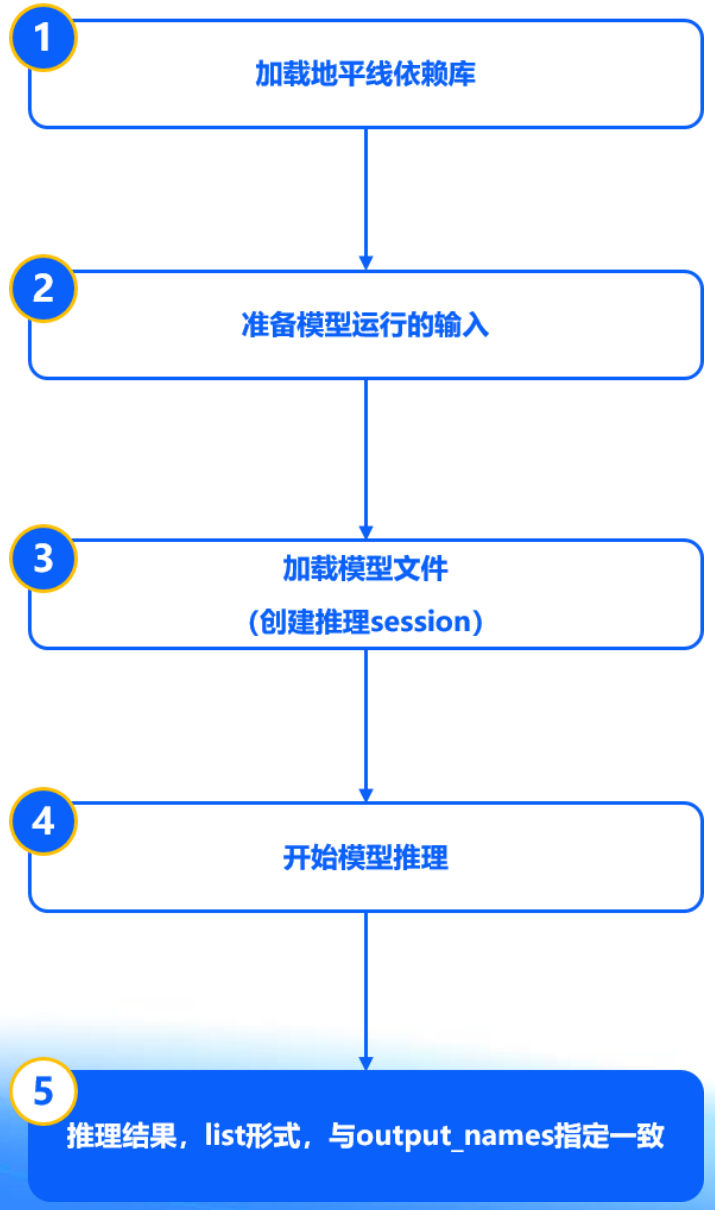
模型转换详解-PTQ

转换流程 | 主要工具 | 精度验证&调优 | 性能评测

主要工具

工具	功能	使用
hb_compile	将浮点模型映射为量化模型并附带验证、编译以及模型修改功能的工具	fast-perf模式和传统模式
hb_config_generator	用于获取最简yaml配置文件、包含全部参数默认值的yaml配置文件的工具	见转换配置文件生成
hb_model_info	用于解析hbm和bc编译时的依赖及参数信息并可视化、onnx模型基本信息，同时支持对bc可删除节点进行查询的工具	hb_model_info *.bc/*.hbm –v可视化bc模型和hbm模型
hb_verifier	用于进行模型逐层算子输出余弦相似度对比的工具，支持进行onnx模型之间、onnx模型与hbir (bc)模型之间的对比	hb_verifier –m *.onnx,*.onnx/*.bc –i input.npy
hb_eval_preprocess	用于对AI Benchmark模型精度进行评估时，在x86环境下对图片数据进行预处理	hb_eval_preprocess -h
HBRuntime	用于推理PTQ各阶段生成物模型	from horizon_tc_ui.hb_runtime import HBRuntime
精度Debug工具	用于定位模型量化过程中产生的精度问题的工具，该工具能够协助您对校准模型进行节点粒度的量化误差分析，最终帮助您快速定位出现精度异常的节点	见精度验证&调优

推理工具HBRuntime



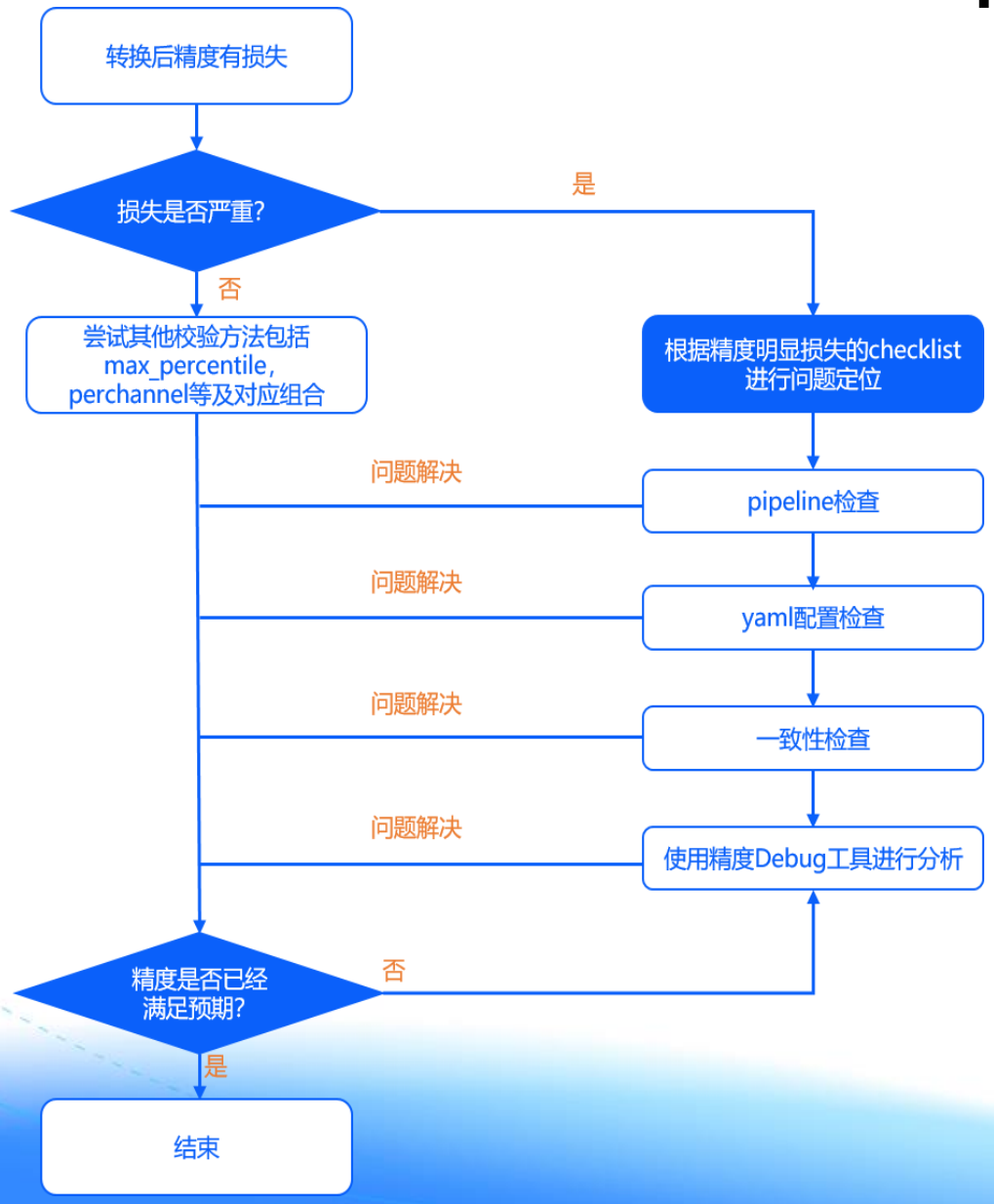

```python
import numpy as np
# 加载地平线依赖库
from horizon_tc_ui.hb_runtime import HBRuntime

# 准备模型运行的输入，此处`input.npy`为处理好的数据
data = np.load("input.npy")
# 加载模型文件，根据实际模型进行设置
# ONNX模型
sess = HBRuntime("model.onnx")
# HBIR模型
sess = HBRuntime("model.bc")
# 获取输入&输出节点名称
input_names = sess.input_names
output_names = sess.output_names

# 准备输入数据，根据实际输入类型和layout进行准备，配置格式要求为字典形式，输入名称和输入数据组
# 如模型仅有一个输入
input_feed = {input_names[0]: data}
# 如模型有多个输入
input_feed = {input_names[0]: data1, input_names[1]: data2}
# 进行模型推理，推理的返回值是一个list，依次与output_names指定名称一一对应
output = sess.run(output_names, input_feed)
```

支持ONNX和HBIR模型推理（实际代码中二选一）

模型转换详解-PTQ

转换流程 | 主要工具 | 精度验证&调优 | 性能评测

精度验证流程



推理各阶段生成物模型：

用户原始onnx模型、original_float模型和optimized_float模型三者精度应严格对齐，精度问题往往从calibrated_model开始

1. 掉点幅度大（超过4%）：

对齐原始onnx模型、original_float模型和optimized_float模型可以排除大部分因使用错误导致的大幅度掉点问题，如果掉点幅度仍然较大，请排查模型和校准数据集是否匹配

2. 掉点幅度较大（1.5%-3%）：

调整calibration_type，尝试开启per_channel，尝试optimization提供的asymmetric与bias_correction等量化tricks；尝试调整校准数据集，使校准数据集更均衡

3. 掉点幅度小：

使用精度Debug工具进行更精细地算子层面定位

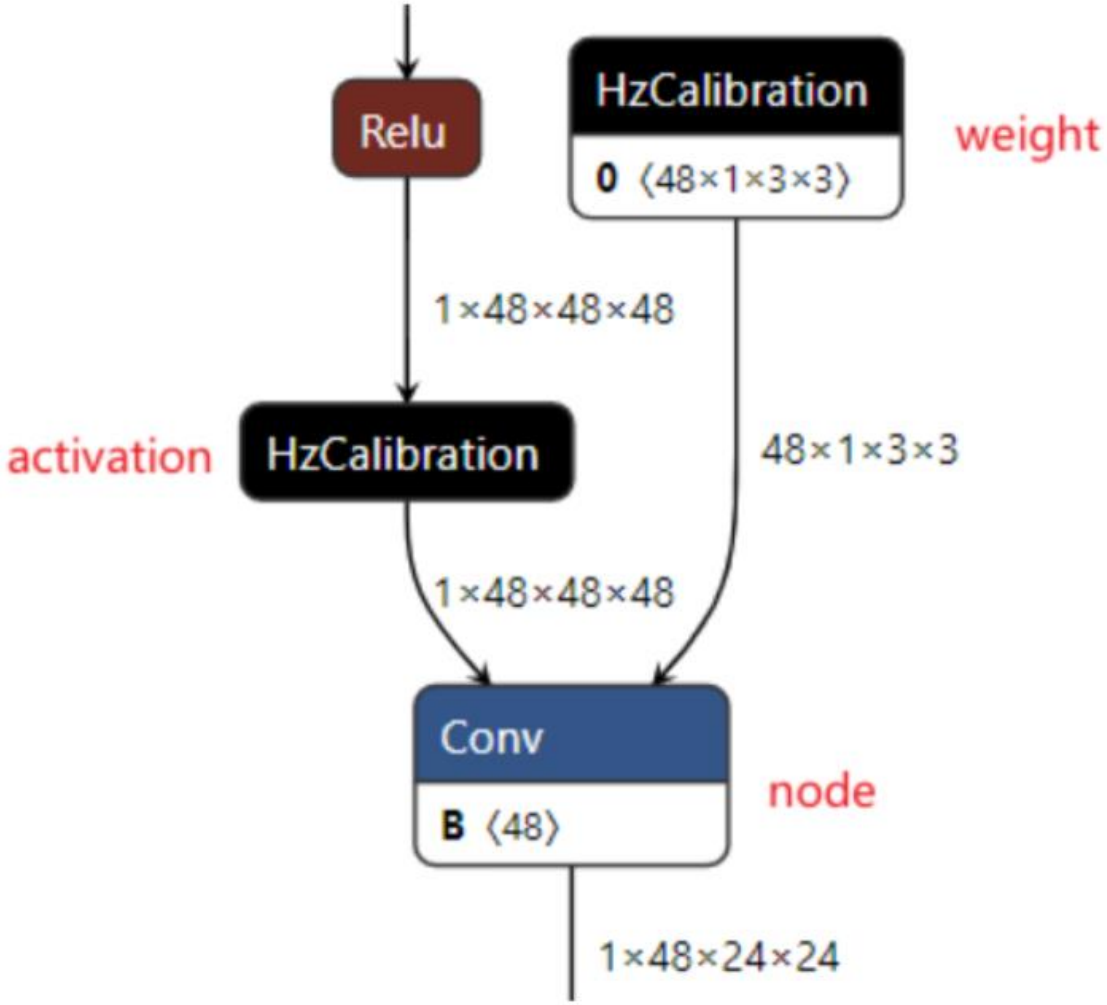
精度Debug工具-校准数据

```
model_parameters:
  debug_mode: "dump_calibration_data"
```

```
|-- calibration_data  #校准数据
    |---- input1  #文件夹名为模型的输入节点并保存对应的输入数据
          |-------- 0.npy
          |-------- 1.npy
          |-------- ...
    |---- input2  #对于多输入模型将保存多个文件夹
          |-------- 0.npy
          |-------- 1.npy
          |-------- ...
    |---- ...
```

- 模型通过对校准数据（Calibration Data）进行前向推理来获取每个被量化节点的量化参数如缩放因子和阈值。可通过左图的配置获得精度Debug工具使用的校准数据
- 此处保存的校准数据和转换流程里用到的一致，需注意文件夹结构参考左图

精度Debug工具-校准模型



- 校准模型（calibrated_model.onnx）是将在校准阶段计算得到的每个被量化节点的量化参数保存在校准节点中，从而得到的模型，也即生成物中calibrated_model.onnx

- 校准节点的节点类型为HzCalibration，主要分为两类：激活（activation）校准节点和权重（weight）校准节点。校准模型中的其他节点称为普通节点(node)，例如左图中的Conv节点。

精度Debug工具-使用流程

累积误差曲线

获取累积
误差曲线

调用接口：
plot_acc_error

API使用方法：

```python
# 导入debug模块
import horizon_nn.debug as dbg

dbg.plot_acc_error(
        save_dir='./',
        calibrated_data='./calibration_data/',
        model_or_file='./calibrated_model.onnx',
        quantize_node=['weight', 'activation'],
        metric='cosine-similarity',
        average_mode=False)
```

命令行使用方法：

```
# 例如
hmct-debugger plot-acc-error calibrated_model.onnx calibration_data -q ['weight', 'activation']
```

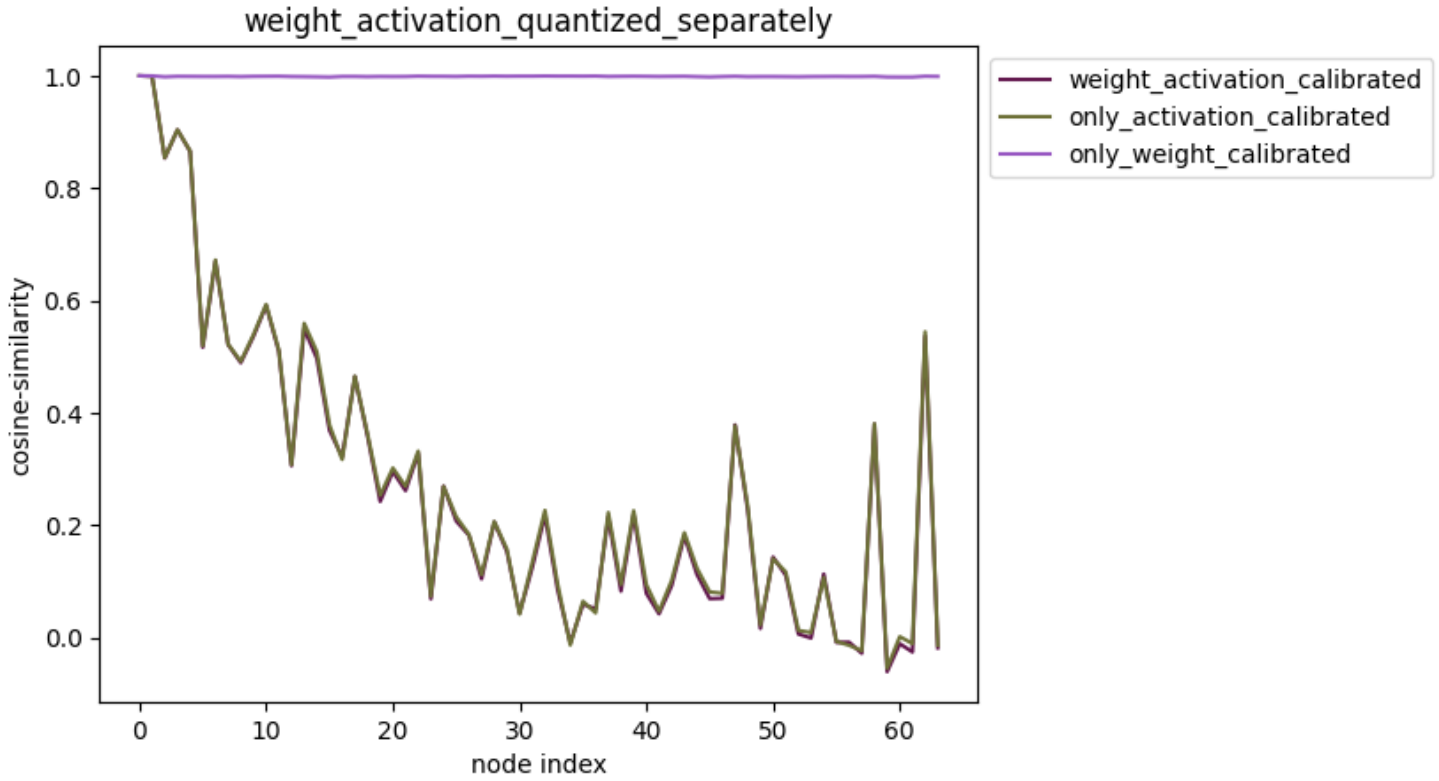
注：可通过hmct-debugger plot-acc-error -h/--help查看更多参数

精度Debug工具-使用流程

累积误差曲线

获取累积
误差曲线

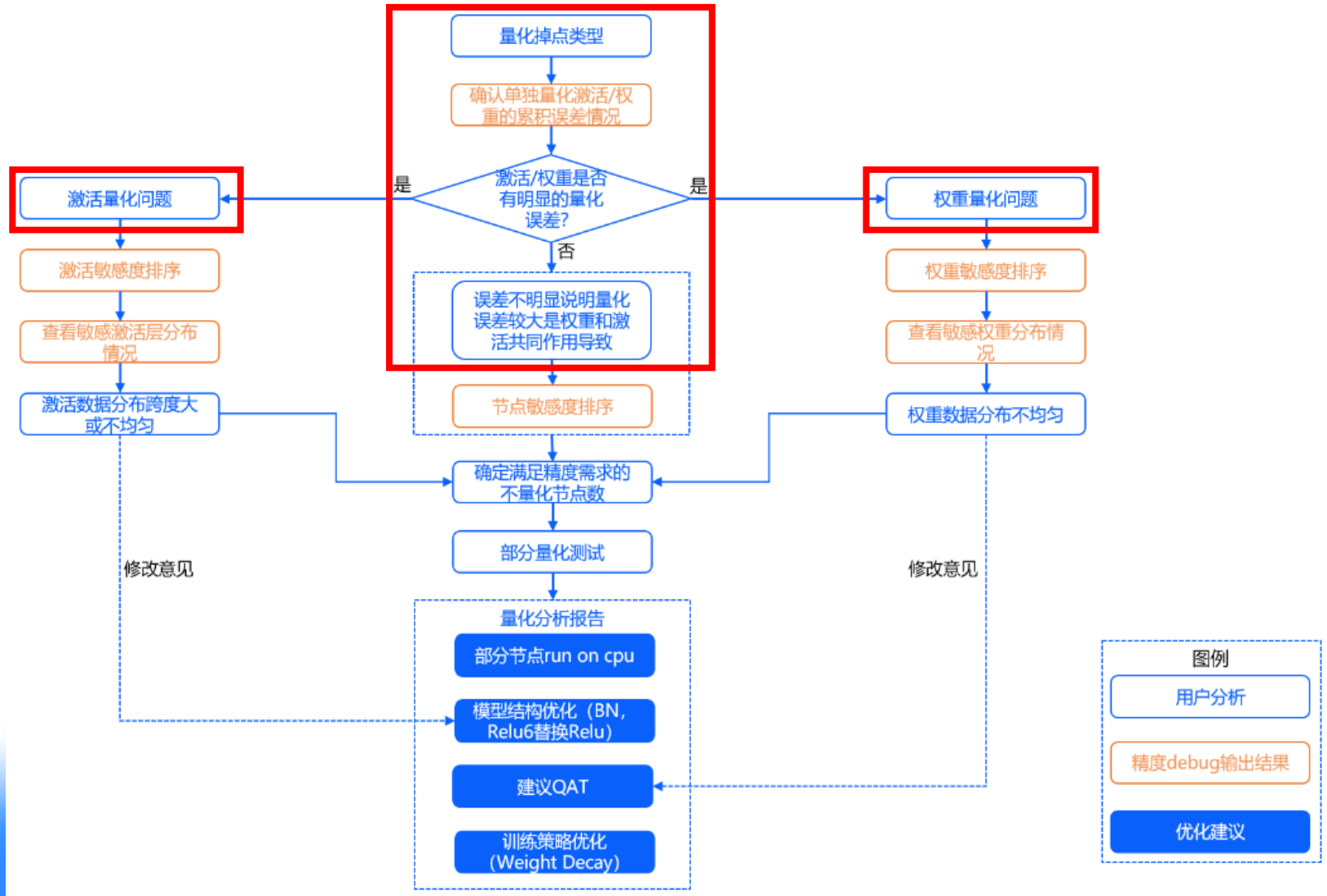
调用接口：
plot_acc_error



注：可通过hmct-debugger plot-acc-error -h/--help查看更多参数

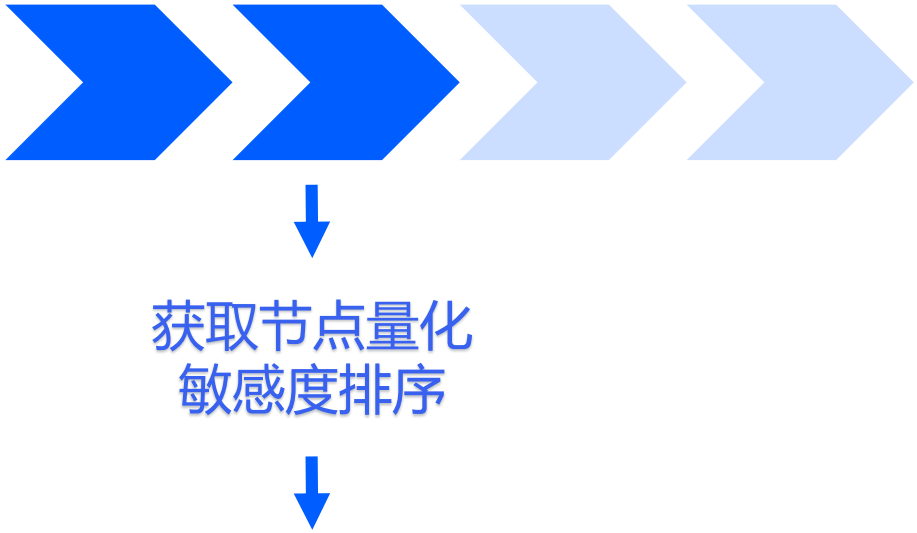
精度Debug工具-使用流程

累积误差曲线



- 首先根据累积误差曲线判断模型量化掉点是由激活量化导致的还是由权重量化导致的，确定是激活量化问题还是权重量化问题。如果激活和权重的单独量化误差不明显，可以判断是两者共同作用导致的。

精度Debug工具-使用流程



节点量化敏感度

API使用方法：

```
# 导入debug模块
import horizon_nn.debug as dbg
# 导入log日志模块
import logging

# 若verbose=True时，需要先设置log level为INFO
logging.getLogger().setLevel(logging.INFO)
# 获取节点量化敏感度
node_message = dbg.get_sensitivity_of_nodes(
        model_or_file='./calibrated_model.onnx',
        metrics=['cosine-similarity', 'mse'],
        calibrated_data='./calibration_data/',
        output_node=None,
        node_type='node',
        data_num=None,
        verbose=True,
        interested_nodes=None)
```

命令行使用方法：

```
# 例如
hmct-debugger get-sensitivity-of-nodes calibrated_model.onnx calibration_data -m ['cosine-similarity','mse']
```

注：可通过hmct-debugger get-sensitivity-of-nodes -h/--help查看更多参数

精度Debug工具-使用流程

获取节点量化敏感度排序

调用接口：
get_sensitivity_of_nodes

节点量化敏感度

```
=================node sensitivity=================
node                    cosine-similarity  mse
--------------------------------------------------
Conv_60                 0.77795            68.02103
Conv_48                 0.78428            64.36318
Conv_82                 0.80394            61.09268
Conv_94                 0.80499            65.05224
Conv_42                 0.83787            49.4949
Conv_88                 0.84614            49.81132
Conv_54                 0.86602            41.69972
Conv_71                 0.87148            39.96296
Conv_65                 0.87495            40.45997
Conv_25                 0.89214            34.30351
Conv_20                 0.89829            32.35053
Conv_77                 0.89916            31.9907
Conv_14                 0.90058            32.40179
Conv_9                  0.90107            34.08191
Conv_37                 0.91162            28.21194
Conv_31                 0.91637            28.79291
```

- node_type为node，且verbose=True时打印结果

注：可通过hmct-debugger get-sensitivity-of-nodes -h/--help查看更多参数

精度Debug工具-使用流程

节点量化敏感度

获取节点量化敏感度排序

调用接口：
get_sensitivity_of_nodes

针对量化敏感节点，分别分析测试单独量化以及部分量化这些节点后的模型精度

调用接口：sensitivity_analysis

```python
# 导入debug模块
import horizon_nn.debug as dbg


dbg.sensitivity_analysis(model_or_file='calibrated_model.onnx',
                         calibrated_data='calibration_data',
                         pick_threshold=0.9999,
                         data_num=1,
                         sensitive_nodes=[])
```

```
hmct-debugger sensitivity-analysis calibrated_model.onnx calibration_data
```

注：可通过hmct-debugger sensitivity-analysis -h/--help查看更多参数

精度Debug工具-使用流程

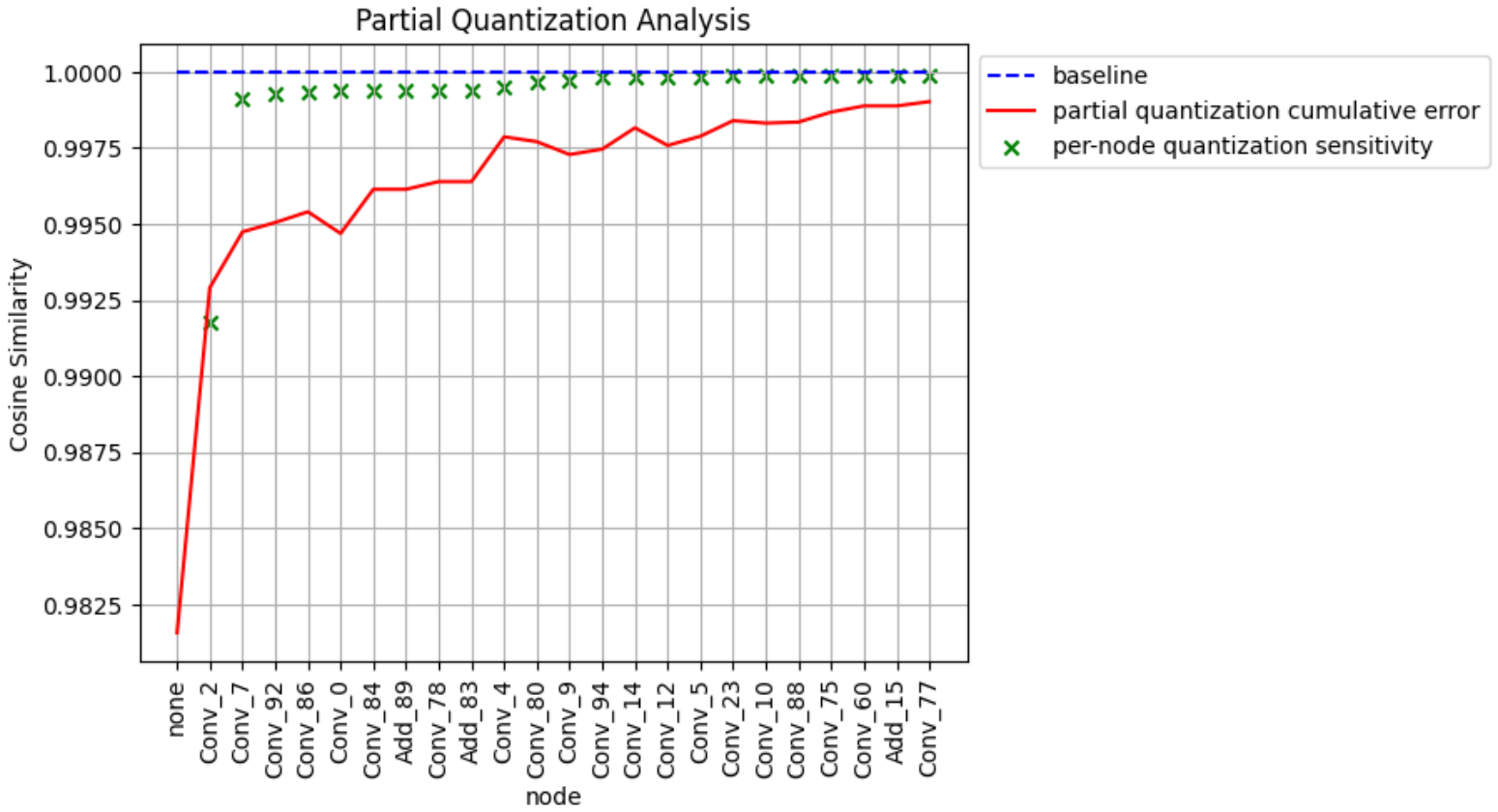
节点量化敏感度

获取节点量化敏感度排序

调用接口：
get_sensitivity_of_nodes

针对量化敏感节点，分别分析测试单独量化以及部分量化这些节点后的模型精度

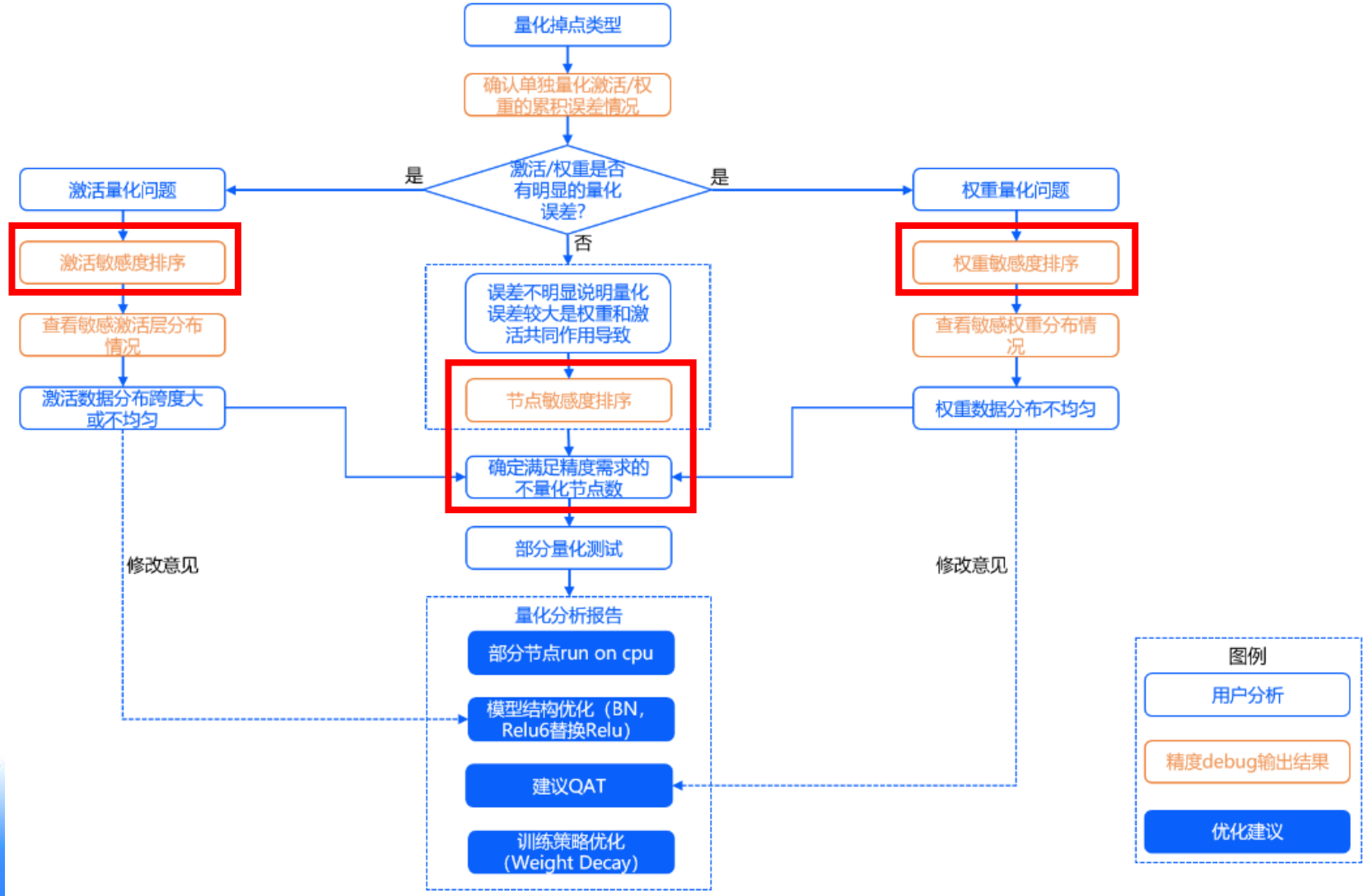
调用接口：sensitivity_analysis



注：可通过hmct-debugger sensitivity-analysis -h/--help查看更多参数

精度Debug工具-使用流程

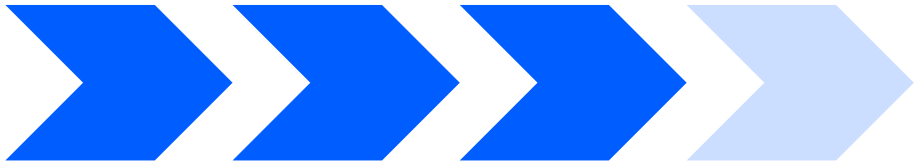
节点量化敏感度



- 权重和激活校准敏感节点最终还需对应到他们所属的普通节点，从而确定普通节点的量化问题是权重量化敏感还是激活量化敏感。我们对敏感节点可以采取不量化测试或高精度量化解决

精度Debug工具-使用流程

数据分布



获取敏感节点数据分布

调用接口：
plot_distribution

API使用方法：

```
# 导入debug模块
import horizon_nn.debug as dbg

dbg.plot_distribution(
        save_dir='./',
        model_or_file='./calibrated_model.onnx',
        calibrated_data='./calibration_data',
        nodes_list=['317_HzCalibration', #激活节点
                    '471_HzCalibration', #权重节点
                    'Conv_2']) #普通节点
```

命令行使用方法：

```
hmct-debugger plot-distribution calibrated_model.onnx calibration_data -n ['317_HzCalibration','471_HzCalibration','Conv_2'
```

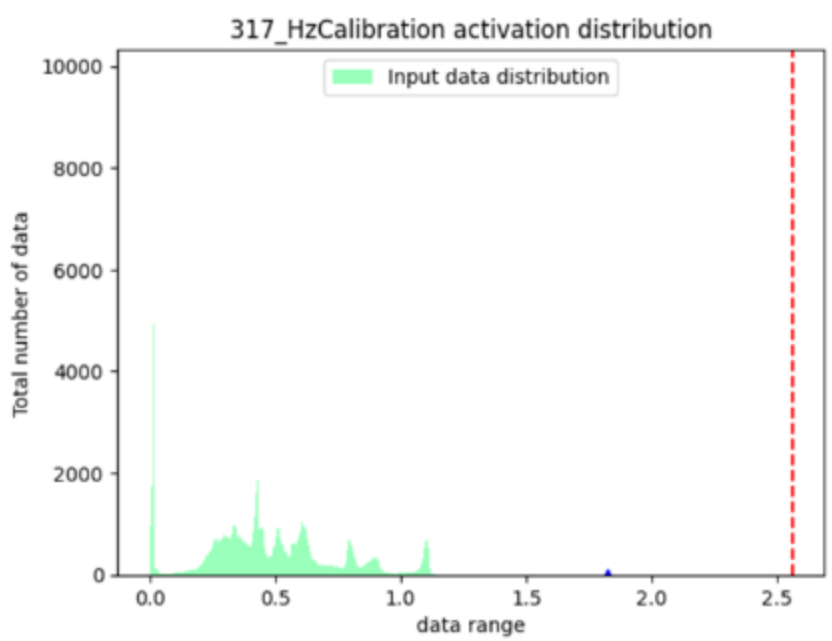
注：可通过hmct-debugger plot-distribution -h/--help查看更多参数

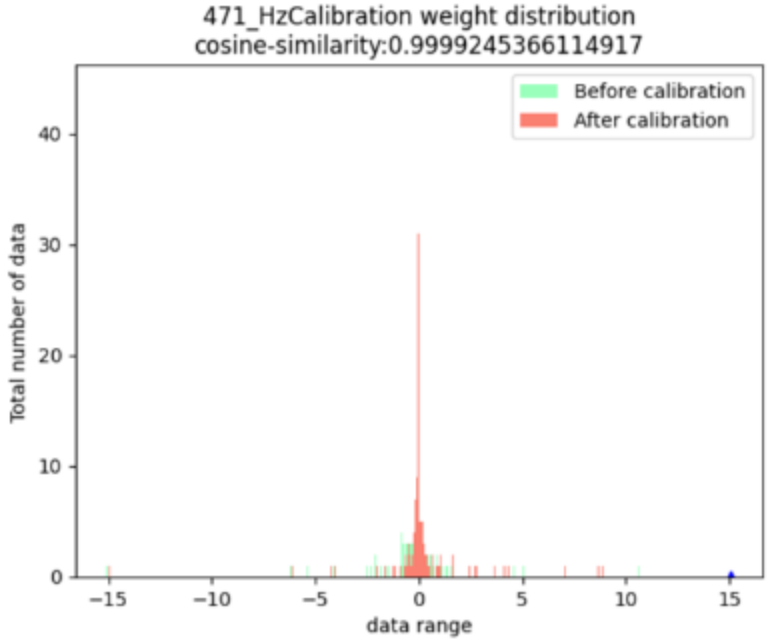
精度Debug工具-使用流程

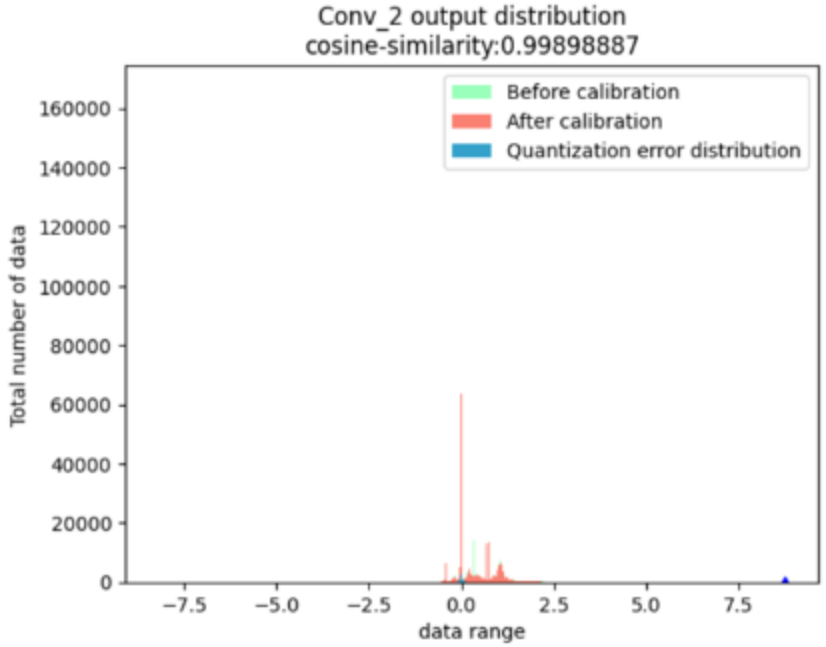
获取敏感节点数据分布

调用接口：
plot_distribution

数据分布







注：可通过hmct-debugger plot-distribution -h/--help查看更多参数

精度Debug工具-使用流程

通道间数据分布

获取敏感节点数据分布

调用接口：get_channelwise_data_distribution

API使用方法：

```
# 导入debug模块
import horizon_nn.debug as dbg

dbg.get_channelwise_data_distribution(
        save_dir='./',
        model_or_file='./calibrated_model.onnx',
        calibrated_data='./calibration_data',
        nodes_list=['317_HzCalibration'],
        axis=None)
```

命令行使用方法：

```
hmct-debugger get-channelwise-data-distribution calibrated_model.onnx calibration_data -n ['317_HzCalibration']
```

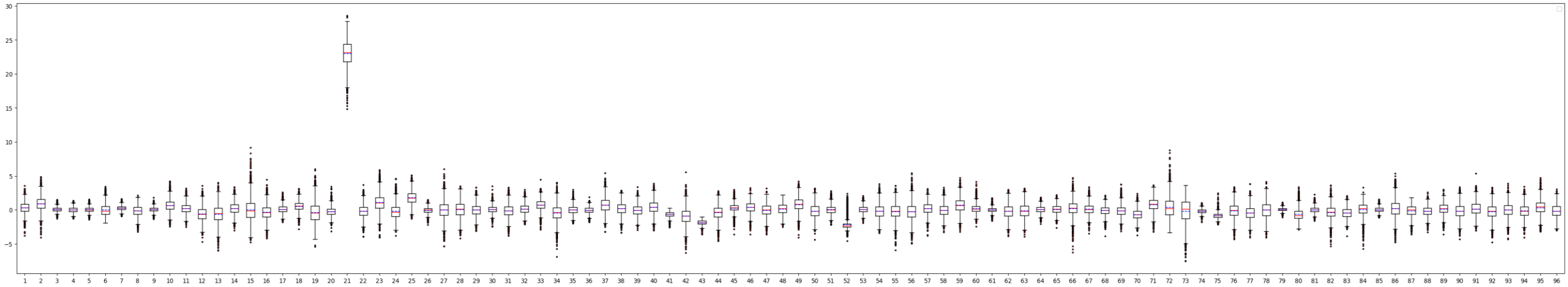
注：可通过hmct-debugger get-channelwise-data-distribution -h/--help查看更多参数

精度Debug工具-使用流程

获取敏感节点数据分布

调用接口：
get_channelwise_data_distribution

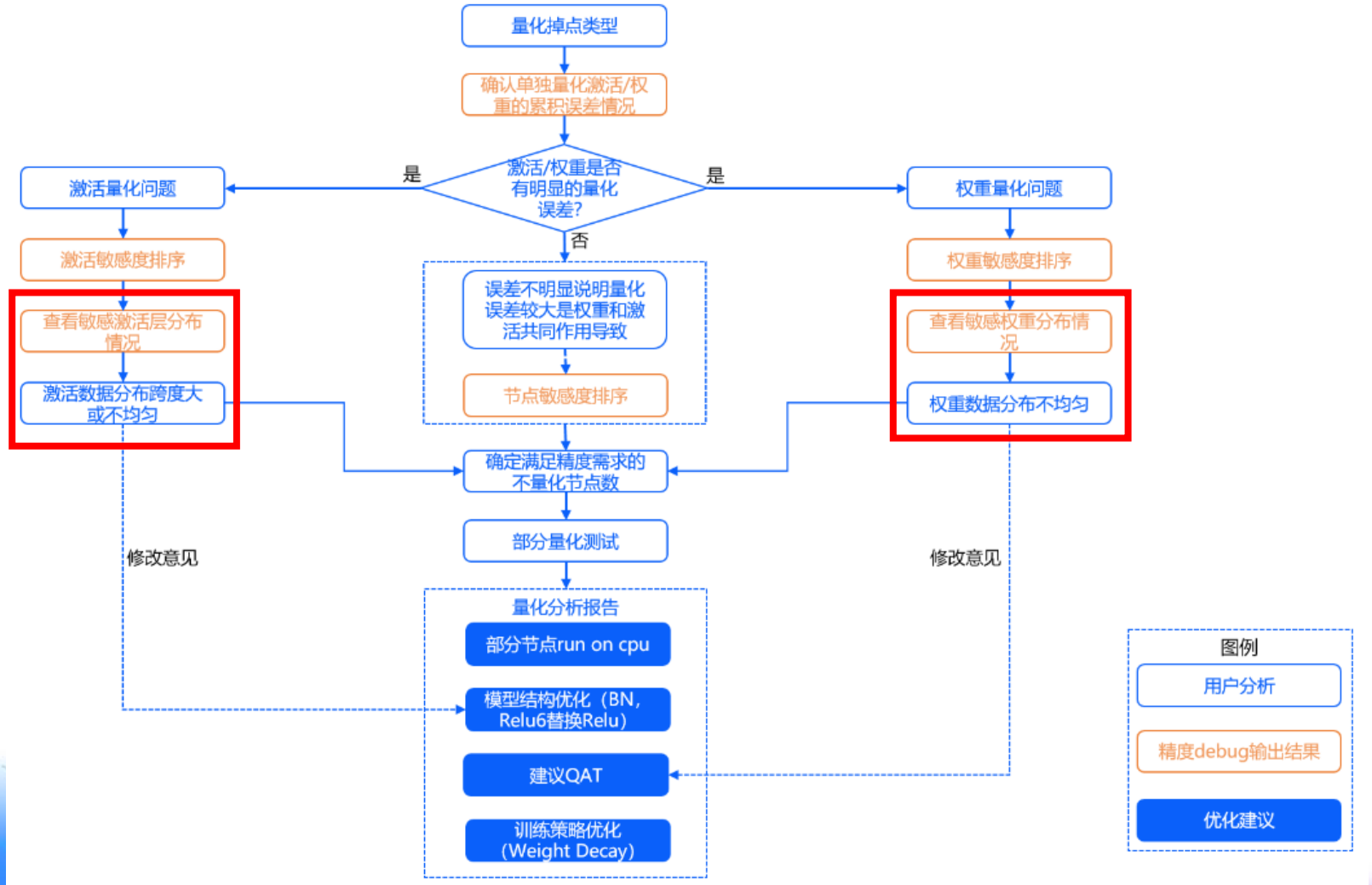
通道间数据分布



注：可通过hmct-debugger get-channelwise-data-distribution -h/--help查看更多参数

精度Debug工具-使用流程

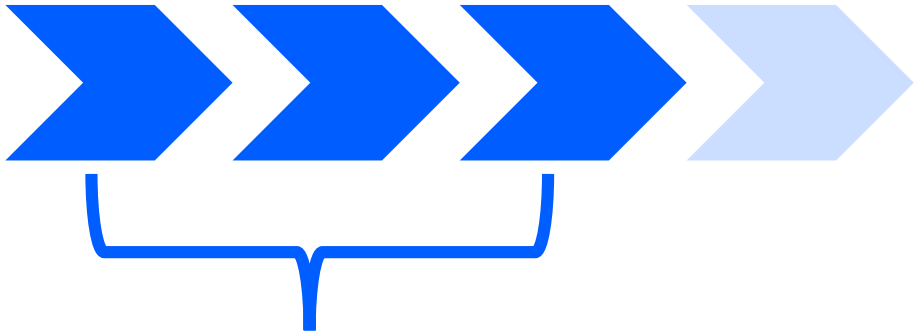
数据分布&通道间数据分布



- 对于量化敏感的节点，根据敏感节点数据分布直方图以及通道间数据分布的箱线图可以进一步分析节点的量化问题，例如是否是数据分布范围过大或通道间数据分布差异大，进一步确定模型的调优方向。

精度Debug工具-使用流程

一键运行



一键运行

调用接口：runall

1、API使用

```
# 导入debug模块
import horizon_nn.debug as dbg

dbg.runall(model_or_file='./calibrated_model.onnx',
           calibrated_data='./calibration_data',)
```

2、命令行使用

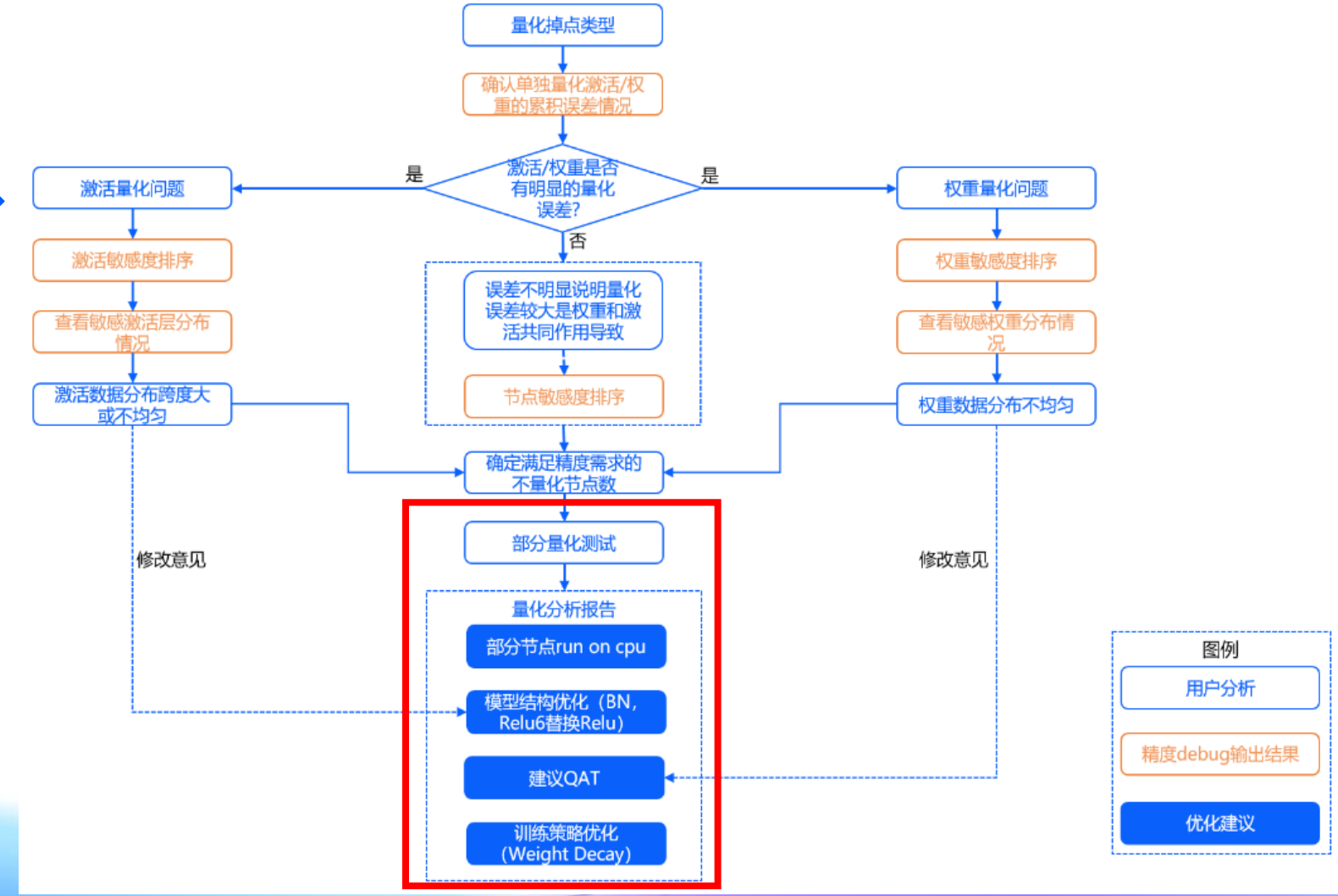
```
# 例如
hmct-debugger runall calibrated_model.onnx calibration_data
```

注：可通过hmct-debugger runall -h/--help查看更多参数

精度Debug工具-使用流程

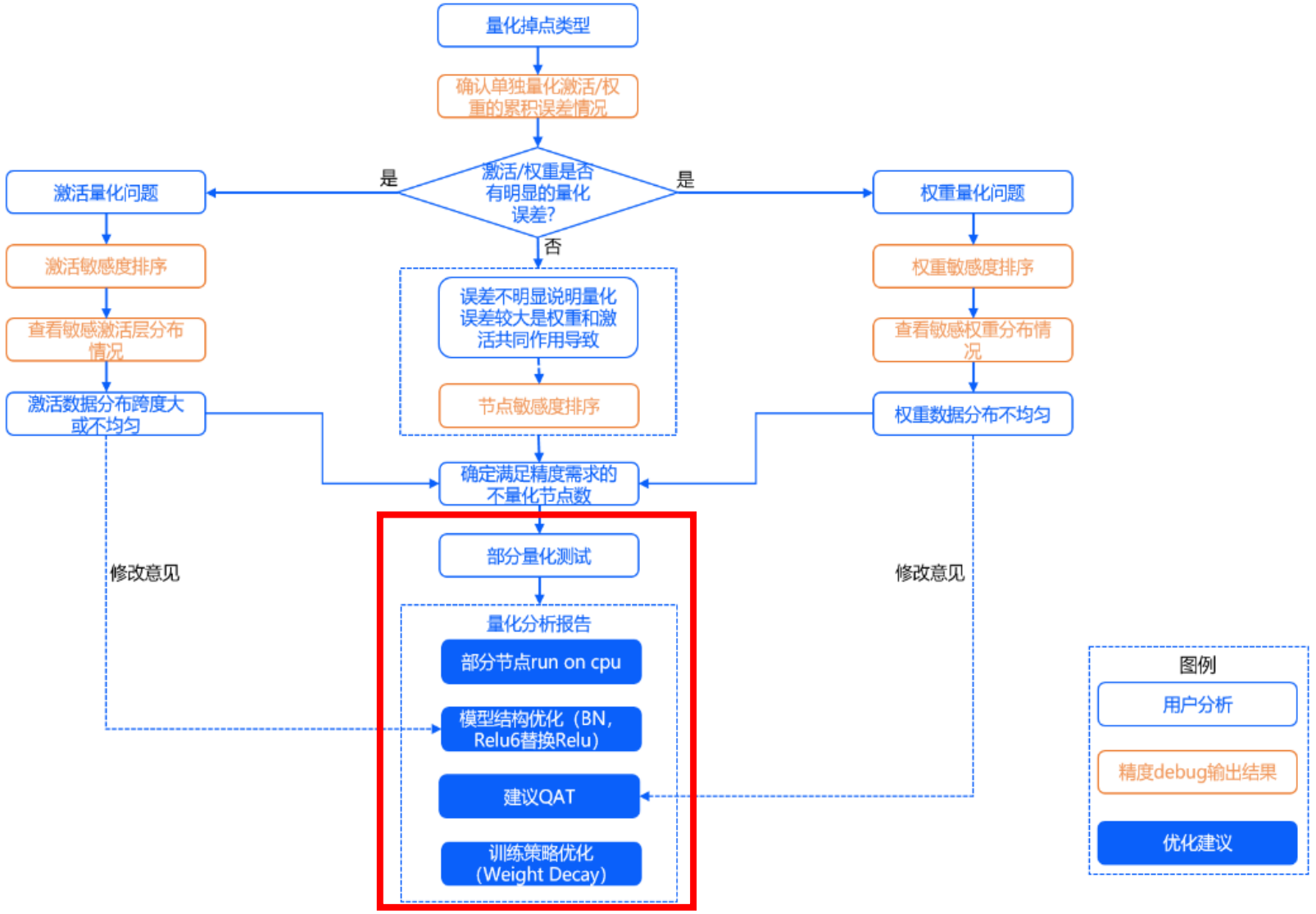
量化调优

量化调优



精度Debug工具-使用流程

量化调优



- 对于量化敏感的节点，进一步确定是激活或权重量化敏感后可对应设置激活或权重以Int16量化（optimization参数提供set_all_nodes_int16功能，对模型中所有激活节点自动设置Int16量化），甚至设置CPU运行

- 其他的调优方向包括但不限于激活敏感节点数据分布不均匀可以通过优化模型结构来改善（增加BN层），权重敏感节点数据分布不均匀则建议使用QAT等；通道间数据分布差异大可以使用per_channel量化


地平线
Horizon Robotics
Break Time

模型转换详解-PTQ

转换流程 | 主要工具 | 精度验证&调优 | 性能评测

性能评测

Latency和FPS作为模型性能评测的主要指标，Latency表示在资源充足的情况下单线程推理一帧的平均耗时，FPS表示在资源充足和多线程并发的情况下模型的吞吐率，使用地平线平台评测模型性能主要有两种方式：静态性能评测和动态性能评测。静态性能评测指编译器针对模型编译生成的二进制指令预估模型的整体性能，其中只包括BPU部分的耗时而不包含CPU部分的性能；动态性能评测则是指在开发板板端实测模型性能，同时包括BPU和CPU两部分耗时。

总体来说，算子在CPU的计算效率远不如BPU，所以当模型出现性能问题时，优先排查模型中是否有CPU算子，我们提供的hb_model_info工具可以可视化生成的定点HBIR模型，从而检查模型中是否存在CPU算子。

此外，部分yaml配置参数可能影响模型性能，也建议排查：

- **layer_out_dump**：指定模型转换过程中是否输出模型的中间结果，一般仅用于调试功能。如果将其配置为 True，则会为每个卷积算子增加一个反量化输出节点，它会显著的降低模型上板后的性能。所以在性能评测时，务必要将该参数配置为 False；
- **compile_mode**：该参数用于选择模型编译时的优化方向为带宽还是时延，关注性能时请配置为 latency；
- **optimize_level**：该参数用于选择编译器的优化等级，实际生产中应配置为 O2 获取最佳性能；
- **max_time_per_fc**：该参数用于控制编译后的模型数据指令的function-call的执行时长，从而实现模型优先级抢占功能。设置此参数更改被抢占模型的function-call执行时长会影响该模型的上板性能

静态性能评测

执行命令hb_compile --fast-perf --model {onnx_model} --march {target_platform}，进行PTQ快速模型转换，获取可部署的性能最优模型（不可用作精度验证），完成后目录结构如下：

```
├── .fast_perf
│   └── resnet50_config.yaml fast-perf模式的配置文件，可参考并修改使用
├── hb_compile.log 转换编译的log文件
├── model_output
│   ├── resnet50.hbm 板端部署模型
│   ├── resnet50.html
│   ├── resnet50.json 静态性能文件
│   ├── resnet50_advice.json 编译器op checker打印结果，带算子后端信息
│   ├── resnet50_calibrated_model.onnx 校准模型
│   ├── resnet50_optimized_float_model.onnx 经过图优化后的原始浮点模型
│   ├── resnet50_original_float_model.onnx 原始浮点模型，相比原始onnx模型，增加每层的shape信息，查阅友好
│   ├── resnet50_ptq_model.onnx 校准后部分量化的模型
│   ├── resnet50_quant_info.json 逐层量化信息
│   └── resnet50_quantized_model.bc 量化后定点的hbir模型，remove_node_type/name前保存的模型
└── resnet50.onnx 原始onnx模型
```

静态性能解读

地平线 Horizon Robotics

Summary:

FPS (1 core): 1291.11 单核FPS

latency: 0.77 ms (774.5 us) 单核Latency

BPU conv original OPs per frame: 8,179,271,680 模型计算量

HBDK version: 4.0.25 编译器版本

Temporal Statistics:

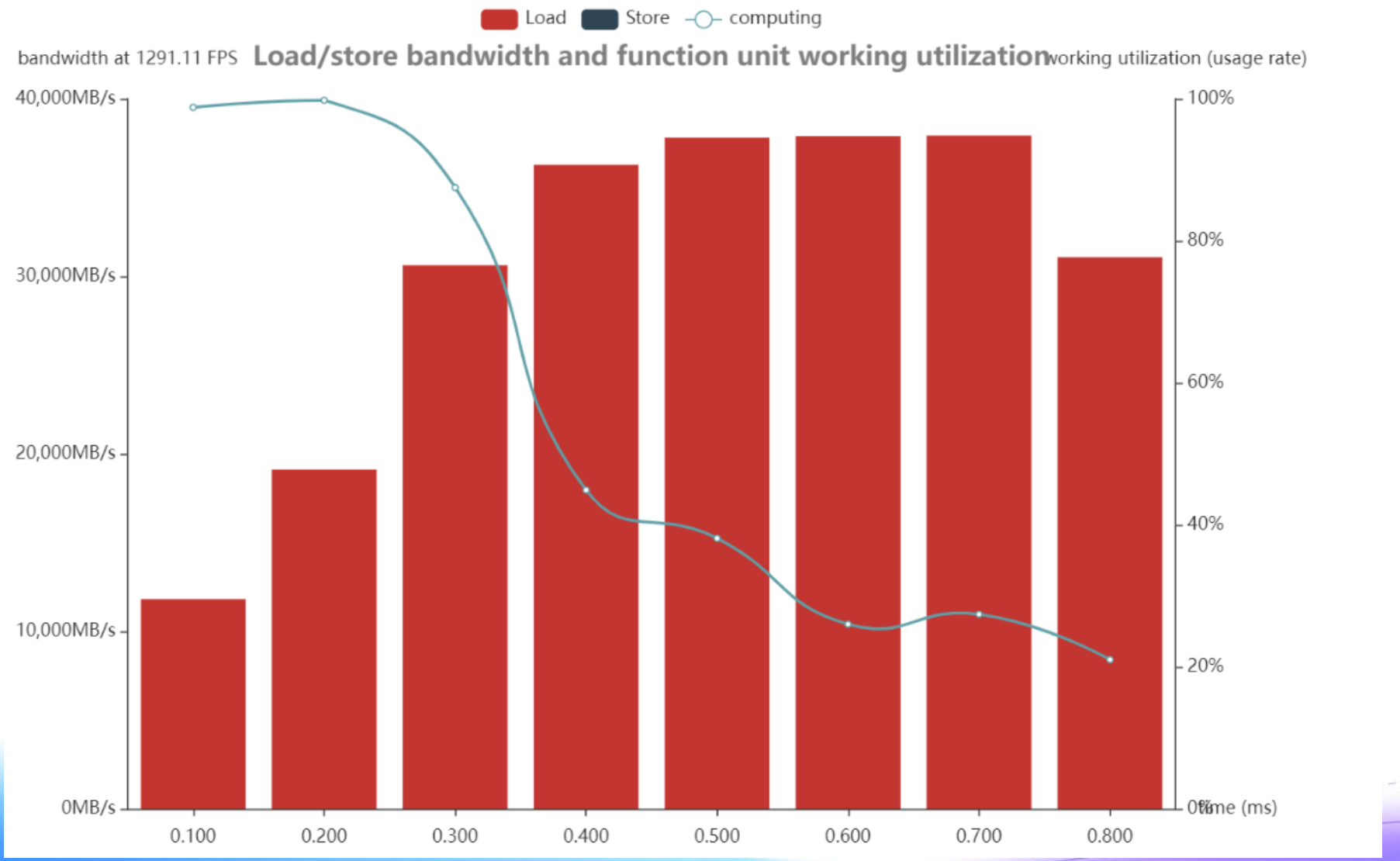
loaded bytes per frame: 26,082,560 单帧推理，BPU从DDR读取的数据量

stored bytes per frame: 1,024 单帧推理，BPU向DDR存储的数据量

DDR (loaded + stored) bytes per frame: 26,083,584 上述两者总和，BPU和DDR交互的总数据量

DDR bytes per second (for 1291.11 FPS): 33,676,704,164

静态性能解读


Load
Store
computing
bandwidth at 1291.11 FPS
Load/store bandwidth and function unit working utilization
working utilization (usage rate)
40,000MB/s
30,000MB/s
20,000MB/s
10,000MB/s
0MB/s
100%
80%
60%
40%
20%
0%
0.100
0.200
0.300
0.400
0.500
0.600
0.700
0.800
Time (ms)

静态性能解读

地平线 Horizon Robotics

Layer Details:

算子活跃时段

layer	ops	computing cost (no DDR)	load cost	store cost	active period of time
input_id_1	0	0	3 us (0.4% of model)	0	0 ~ 6 us (6)
input_id_4	903168	2 us (0.3% of model)	<1 us	0	0 ~ 6 us (6)
Conv_0_id_7	236027904	23 us (2.9% of model)	<1 us	0	1 ~ 22 us (21)
MaxPool_2_id_8	0	5 us (0.6% of model)	0	0	13 ~ 24 us (11)
Conv_3_id_11	25690112	1 us (0.1% of model)	<1 us	0	16 ~ 25 us (9)
Conv_5_id_14	231211008	7 us (0.9% of model)	1 us (0.1% of model)	0	16 ~ 33 us (17)
Conv_7_id_17	102760448	3 us (0.4% of model)	1 us (0% of model)	0	25 ~ 36 us (11)
Add_9_id_20	102760448	3 us (0.4% of model)	1 us (0% of model)	0	25 ~ 40 us (15)
Conv_11_id_23	102760448	3 us (0.4% of model)	1 us (0% of model)	0	26 ~ 44 us (18)
Conv_13_id_26	231211008	7 us (0.9% of model)	1 us (0.1% of model)	0	27 ~ 52 us (25)
Add_16_id_29	102760448	3 us (0.4% of model)	1 us (0% of model)	0	28 ~ 55 us (27)
Conv_18_id_32	102760448	3 us (0.4% of model)	1 us (0% of model)	0	29 ~ 59 us (30)
Conv_20_id_35	231211008	7 us (0.9% of model)	1 us (0.1% of model)	0	29 ~ 67 us (38)
Add_23_id_38	102760448	3 us (0.4% of model)	1 us (0% of model)	0	30 ~ 71 us (41)

- 从算子活跃的时段存在重叠这一现象，可以反映编译器Layer Group和Tiling优化

动态性能评测与解读

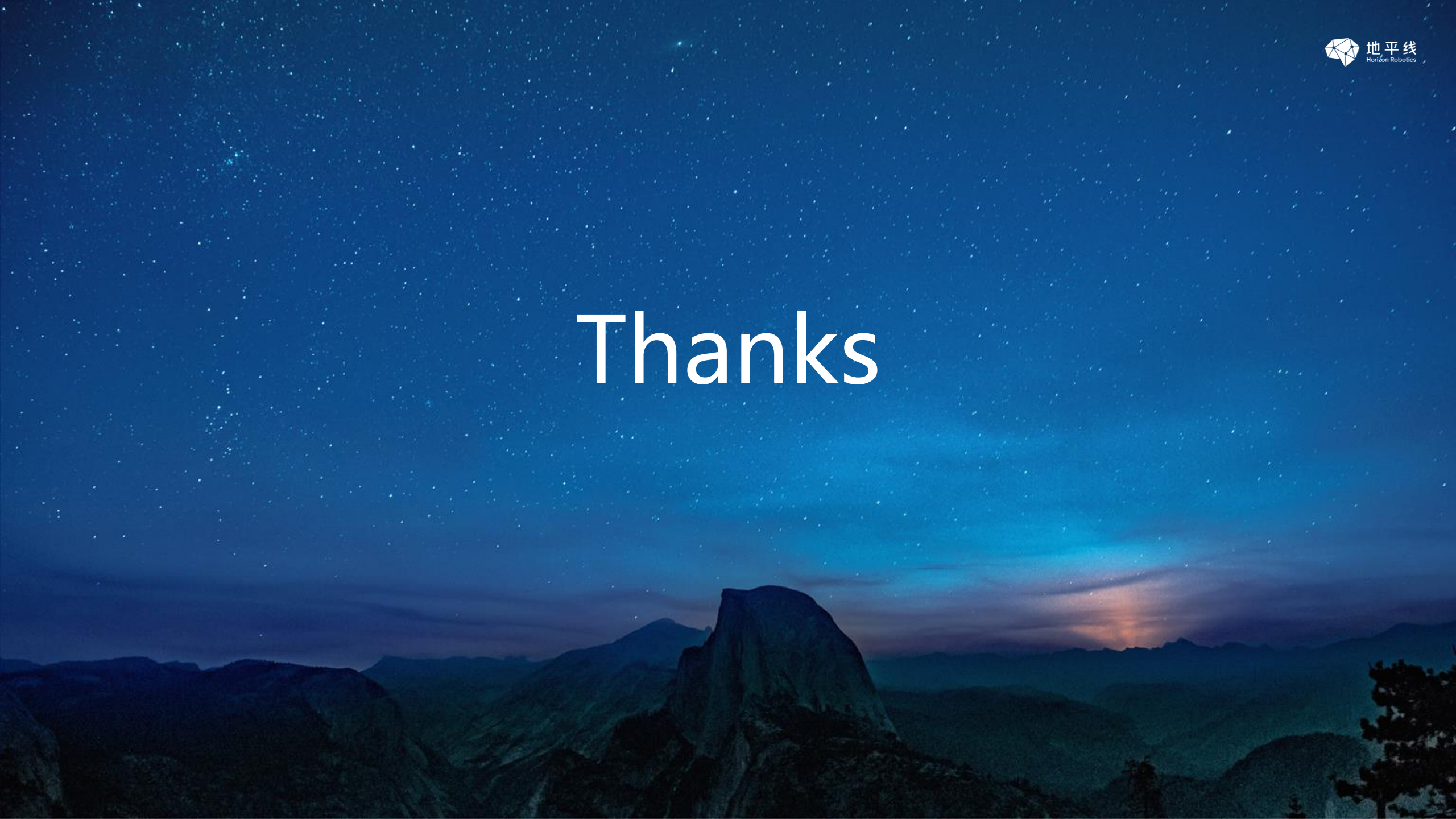
快速执行板端动态Perf的命令为hrt_model_exec perf --model_file {hbm_model} --thread_num {1/8} --frame_count 200 --internal_use

使用hrt_model_exec完成模型性能Perf后会在当前目录下生成BPU和CPU部分的耗时统计（需配置环境变量），保存在profiler.log和profiler.csv文件中，可通过该信息排查性能异常模型（需要控制模型CPU耗时尽可能地短）

如下左图为hrt_model_exec perf工具在终端的打印信息，右图为profiler.log文件信息：

```
Running condition:
  Thread number is: 8
  Frame count   is: 200
  Program run time: 135.578000 ms
Perf result:
  Frame totally latency is: 1048.064819 ms
  Average    latency    is: 5.240324 ms
  Frame      rate       is: 1475.165587 FPS
```


```
{
  "model_latency": {
    "Node-0-BPU-_hb_resnet50_bpu_segment_0": {
      "avg_time": 2.021915,
      "max_time": 2.477,
      "min_time": 1.355
    }
  },
  "processor_latency": {
    "BPU_inference_time_cost": {
      "avg_time": 2.021915,
      "max_time": 2.477,
      "min_time": 1.355
    },
    "CPU_inference_time_cost": {
      "avg_time": 0.0,
      "max_time": 0.0,
      "min_time": 0.0
    }
  },
  "task_latency": {
    "TaskRunningTime": {
      "avg_time": 5.16833,
      "max_time": 5.538,
      "min_time": 1.529
    }
  }
```


地平线
Horizon Robotics
Thanks