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

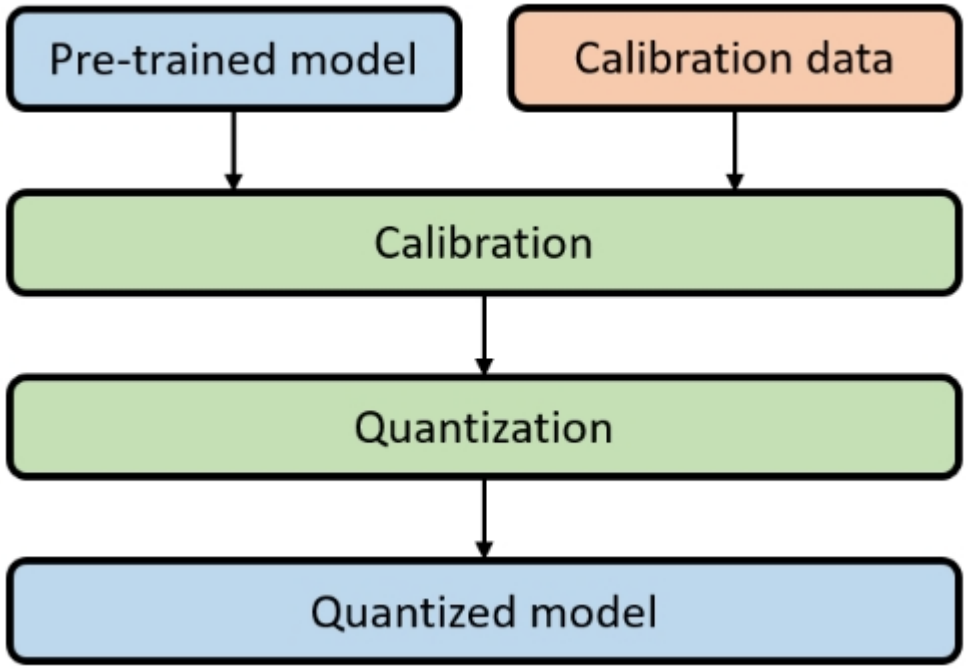
OpenExplorer®：算法模型端侧部署开发套件

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

天工开物 J6算法工具链

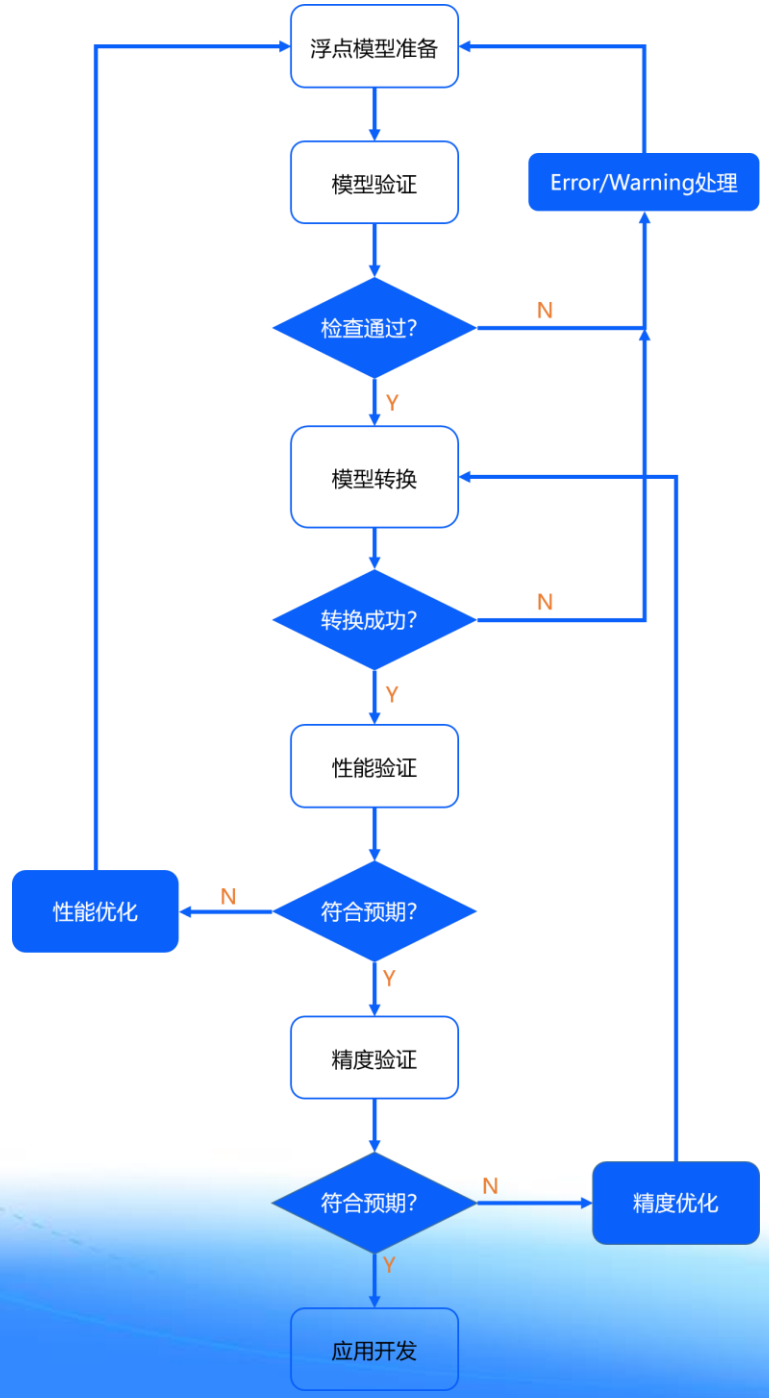
模型转换-PTQ

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



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：

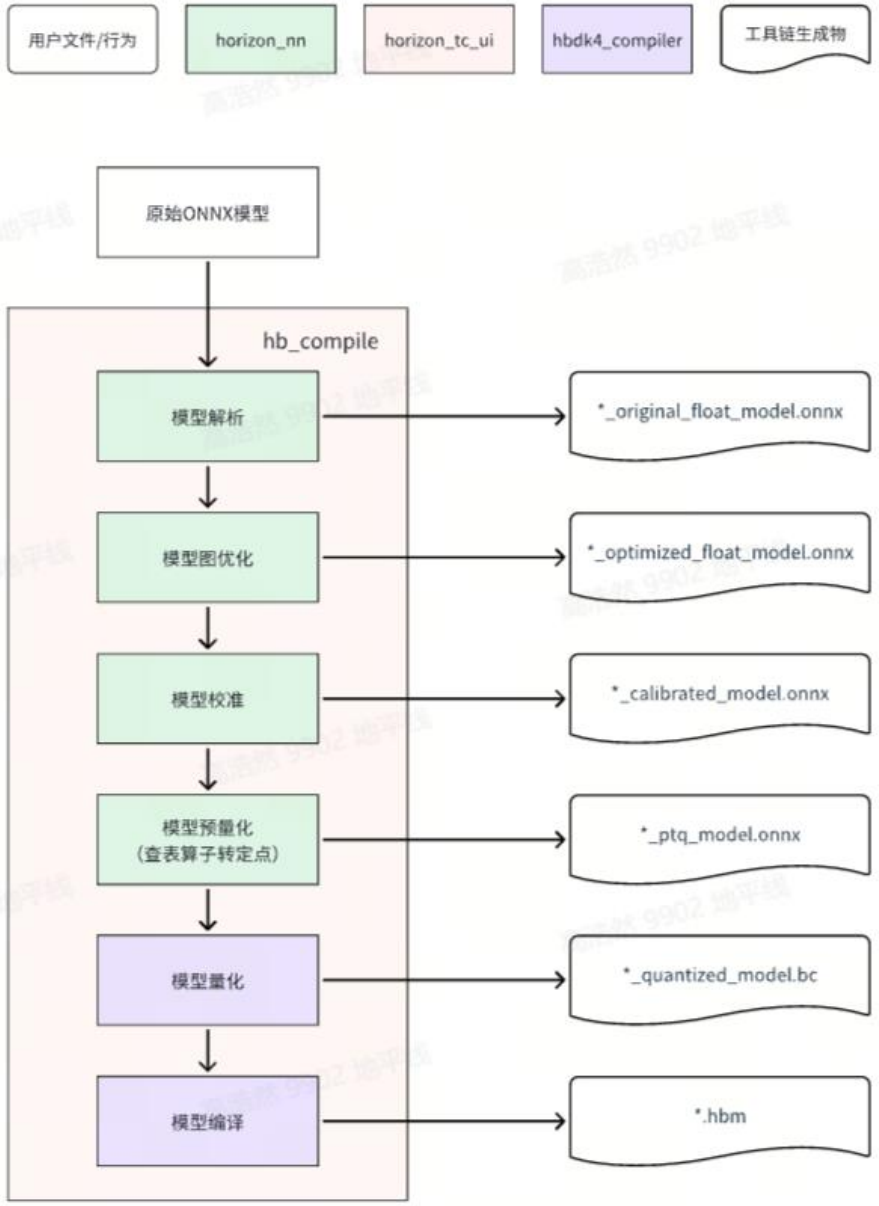
```
hb_config_generator --full-yaml --model model.onnx --march nash-e
```

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



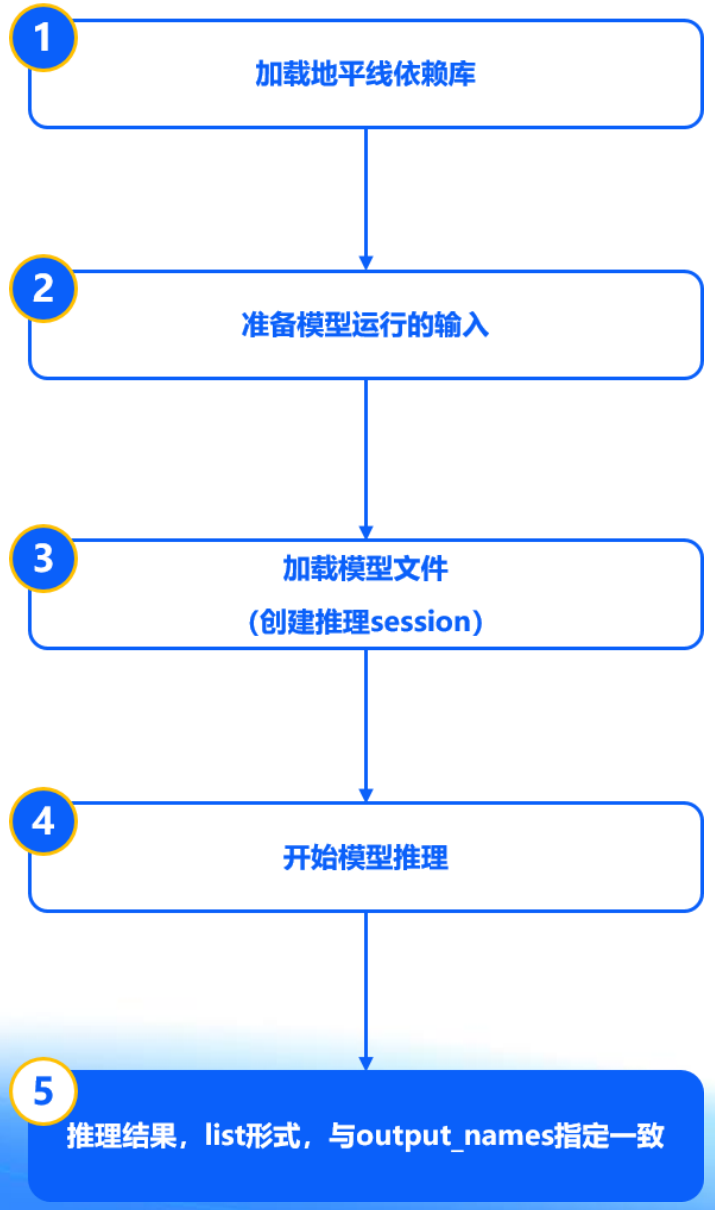
- ***.html**：模型的静态性能评估文件（可读性更好）；
- ***.json**：模型的静态性能评估文件；
- ***_node_info.csv**：文件内保存了转换成功后算子的余弦相似度等信息的结果，与hb_compile成功执行后控制台输出的相似度信息相同；
- ***_quant_info.json**：文件内记录了算子的校准量化参数信息；
- ***_advice.json**：文件内保存了模型编译过程中，地平线模型编译器op checker打印的结果，包含算子的后端信息。

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

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
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



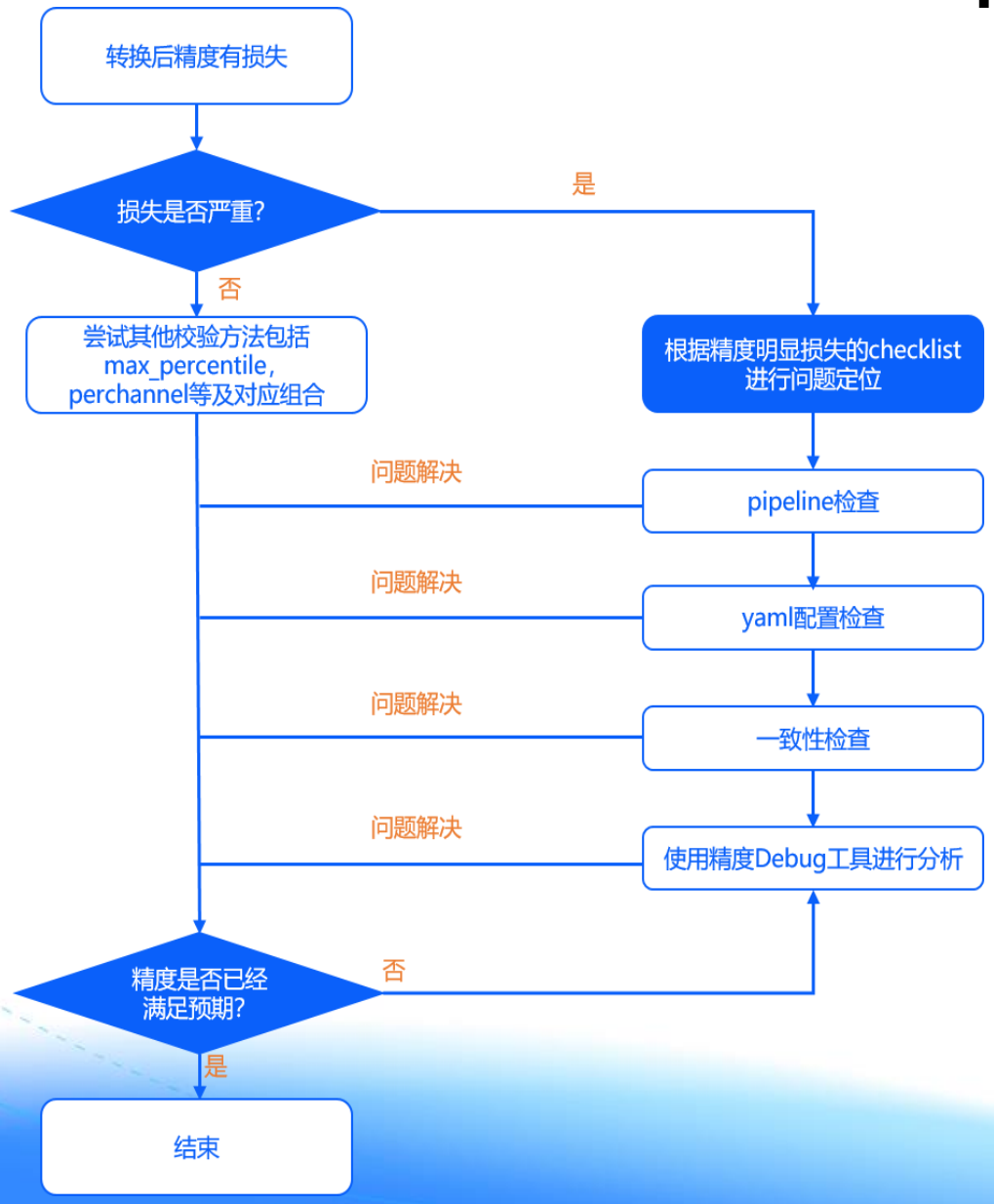

```python
import numpy as np
# 加载地平线依赖库
from horizon_tc_ui.hb_runtime import HBRuntime

# 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）

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

转换精度调优-quant_config参数

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"}
    }
}
```

性能评测

Latency和**FPS**作为模型性能评测的主要指标，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。**

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，**优先通过动态性能评测工具排查模型中是否有CPU算子。**如存在CPU算子，我们提供hb_model_info工具可以可视化生成的定点HBIR模型，定位到具体算子。

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

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

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export _HB_UCP_PROFILER_LOG_PATH_=./），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

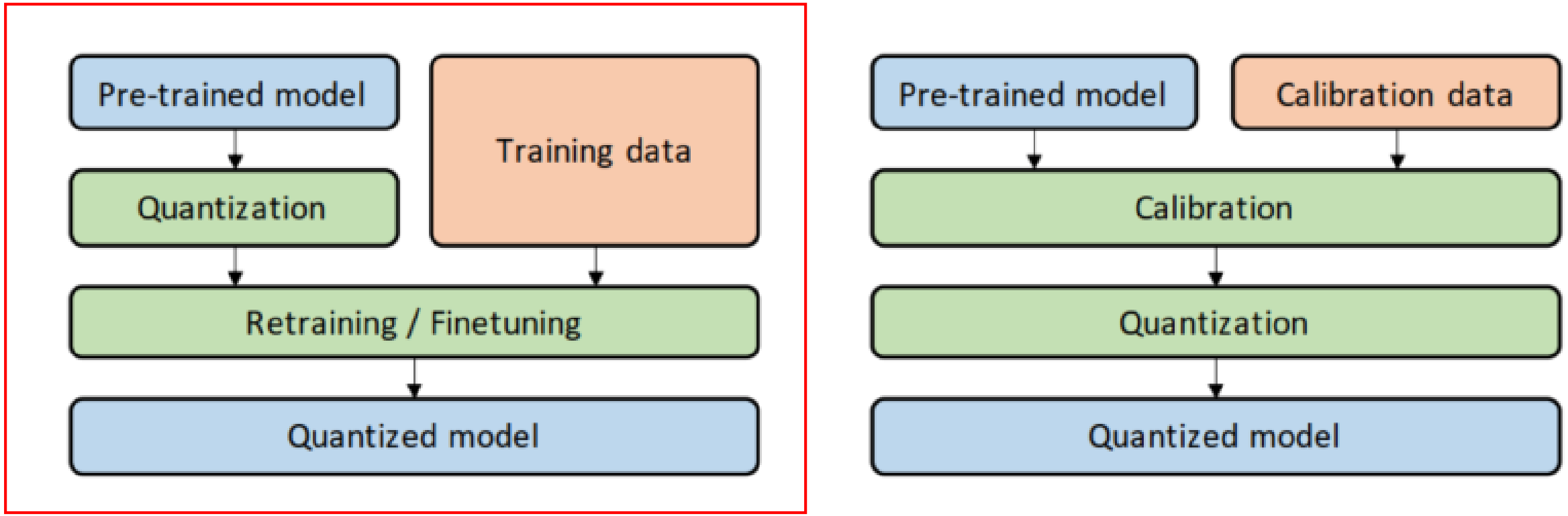
```
Running condition:
  Thread number is: 8
  Frame count   is: 200
  Program run time: 135.578000 ms
Perf result:
  Frame totally latency is: 1048.064819 ms
  Average    latency    is: 5.240324 ms
  Frame      rate       is: 1475.165587 FPS
```


```
"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
  }
}
```

天工开物 J6 算法工具链

模型量化-QAT

QAT量化



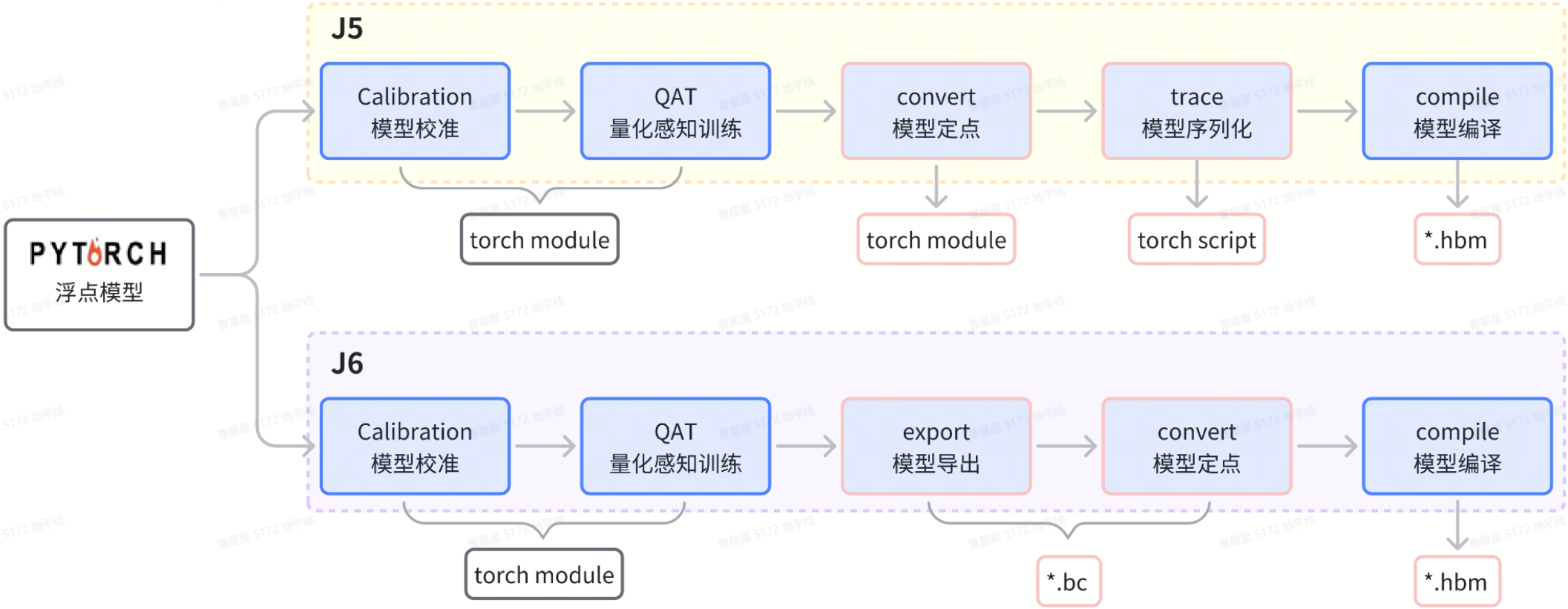
QAT（左）和PTQ（右）

QAT量化

量化训练工具目前支持的量化方式有Eager、Symbolic Trace、Jit Trace三种

量化方式	Eager	Symbolic Trace	Jit Trace
算子替换	手动	自动	自动
算子融合	手动	自动	自动
语法限制	无	较多	无
动态控制流	支持	不支持	支持
Debug难度	低	高	低

QAT流程图


J5
Calibration
模型校准
QAT
量化感知训练
convert
模型定点
trace
模型序列化
compile
模型编译
torch module
torch module
torch script
*.hbm
PYTORCH
浮点模型
J6
Calibration
模型校准
QAT
量化感知训练
export
模型导出
convert
模型定点
compile
模型编译
torch module
*.bc
*.hbm

浮点模型改造

对浮点模型做必要的改造，以支持量化相关操作。模型改造必要的操作有：

•在模型输入前插入 QuantStub。

•在模型输出后插入 DequantStub。

```python
class FxQATReadyMobileNetV2(MobileNetV2):
    def __init__(
        self,
    ):
        self.quant = QuantStub(scale=1 / 128)
        self.dequant = DeQuantStub()

    def forward(self, x: Tensor) -> Tensor:
        x = self.quant(x)
        x = super().forward(x)
        x = self.dequant(x)

        return x
```

$$x_{int} = clamp(round\left(\frac{x_{float}}{\boldsymbol{scale}} + zero_{point}\right), -2^{b-1}, 2^{b-1} - 1)$$

from horizon_plugin_pytorch.quantization import QuantStub

from torch.quantization import DeQuantStub

•插入的 QuantStub 和 DequantStub 必须注册为模型的子模块，否则将无法正确处理它们的量化状态。

•多个输入仅在 scale 相同时可以共享 QuantStub，否则请为每个输入定义单独的 QuantStub。

•可以使用 torch.quantization.QuantStub插入量化，手动固定scale时需要使用 horizon_plugin_pytorch.quantization.QuantStub。

Calibration

此过程通过在模型中插入 Observer 的方式，在 forward 过程中统计各处的数据分布情况，从而计算出合理的量化参数。

```python
from horizon_plugin_pytorch.quantization.qconfig_template import(
    default_calibration_qconfig_setter
)
# 输出模型会共享输入模型的 attributes，为不影响 float_model 的后续使用，
# 此处进行了 deepcopy
calib_model = copy.deepcopy(float_model)

# 将模型转化为 Calibration 状态，以统计各处数据的数值分布特征
calib_model = prepare(
    calib_model,
    example_inputs=example_input,
    qconfig_setter=default_calibration_qconfig_setter,
    method=PrepareMethod.JIT,
    )

# 执行 Calibration 过程（不需要 backward）
# 注意此处对模型状态的控制，模型需要处于 eval 状态以使 Bn 的行为符合要求
calib_model.eval()
set_fake_quantize(calib_model, FakeQuantState.CALIBRATION)
with torch.no_grad():
    cnt = 0
    for image, target in calib_data_loader:
        image, target = image.to(device), target.to(device)
        calib_model(image)
        cnt += image.size(0)
        if cnt >= num_examples:
            break
# 测试伪量化精度
# 注意此处对模型状态的控制
calib_model.eval()
set_fake_quantize(calib_model, FakeQuantState.VALIDATION)

top1, top5 = evaluate(
    calib_model,
    eval_data_loader,
    device,
)
```

- 模型为上板部署模型。
- 前处理使用infer前处理， 校准数据集使用训练数据集
- Batch Size尽可能设置最大，step_num>100

- 对于部分模型，仅通过 Calibration 便可使精度达到要求，不必进行比较耗时的量化感知训练。
- 即使模型经过量化校准后无法满足精度要求，此过程也可降低后续量化感知训练的难度，缩短训练时间，提升最终的训练精度。
- 若精度和浮点相差较大，则需要做debug，避免在QAT阶段的错误训练成本

Qat训练

量化感知训练通过在模型中插入伪量化节点的方式，在训练过程中使模型感知到量化带来的影响，在这种情况下对模型参数进行微调，以提升量化后的精度。

```python
from horizon_plugin_pytorch.quantization.qconfig_template import (
    default_qat_qconfig_setter
    )
qat_model = copy.deepcopy(float_model)

# 将模型转为 QAT 状态
qat_model = prepare(
    qat_model,
    example_inputs=example_input,
    qconfig_setter=default_qat_qconfig_setter,
    method=PrepareMethod.JIT
    )

# 加载 Calibration 模型中的量化参数
qat_model.load_state_dict(calib_model.state_dict())

# 进行量化感知训练
# 作为一个 filetune 过程，量化感知训练一般需要设定较小的学习率
optimizer = torch.optim.Adam(
    qat_model.parameters(), lr=1e-3, weight_decay=1e-4
)
```

```python
for nepoch in range(epoch_num):
    # 注意此处对 QAT 模型 training 状态的控制方法
    qat_model.train()
    set_fake_quantize(qat_model, FakeQuantState.QAT)

    train_one_epoch(
        ...
    )

    # 注意此处对 QAT 模型 eval 状态的控制方法
    qat_model.eval()
    set_fake_quantize(qat_model, FakeQuantState.VALIDATION)

    top1, top5 = evaluate(
        qat_model,
        eval_data_loader,
        device,
    )
```

训练策略配置：

- 可复用浮点训练阶段的优化器，epoch一般为浮点的1/10
- 选择较小的lr做finetune

固定scale：若calib和浮点精度差距较小，可以固定住激活的量化scale，仅做模型参数的finetune。

```python
qat_model = prepare(
    model,
    example_inputs=example_input,
    qconfig_setter=(get_qconfig(fix_scale=True)),
)
```

定点转换与编译

伪量化精度达标后，便可执行模型的定点化。

```python
# 使用哪个模型作为流程的输入，可以选择 calib_model 或 qat_model
base_model = qat_model
######################################################################
from horizon_plugin_pytorch.quantization.hbdk4 import export
from hbdk4.compiler import convert, save

hbir_qat_model = export(base_model, (example_input,))
save(hbir_qat_model,"./qat.bc")
# 将模型转为定点状态，注意此处的 march 需要区分 march

hbir_quantized_model = convert(
    hbir_qat_model,
    March.NASH_E,
)
save(hbir_quantized_model,"./quantized.bc")
```

模型定点化

```python
def evaluate_hbir(
    model: hb4.Module, data_loader: data.DataLoader
) -> Tuple[AverageMeter, AverageMeter]:
    top1 = AverageMeter("Acc@1", ":6.2f")
    top5 = AverageMeter("Acc@5", ":6.2f")

    for image, target in data_loader:
        image, target = image.cpu(), target.cpu()
        output = model.functions[0](image)[0]
        acc1, acc5 = accuracy(output, target, topk=(1, 5))
        top1.update(acc1, image.size(0))
        top5.update(acc5, image.size(0))

    return top1, top5

# 测试定点模型精度
top1, top5 = evaluate_hbir(
    hbir_quantized_model,
    eval_hbir_data_loader,
)
```

模型推理（DDR输入）

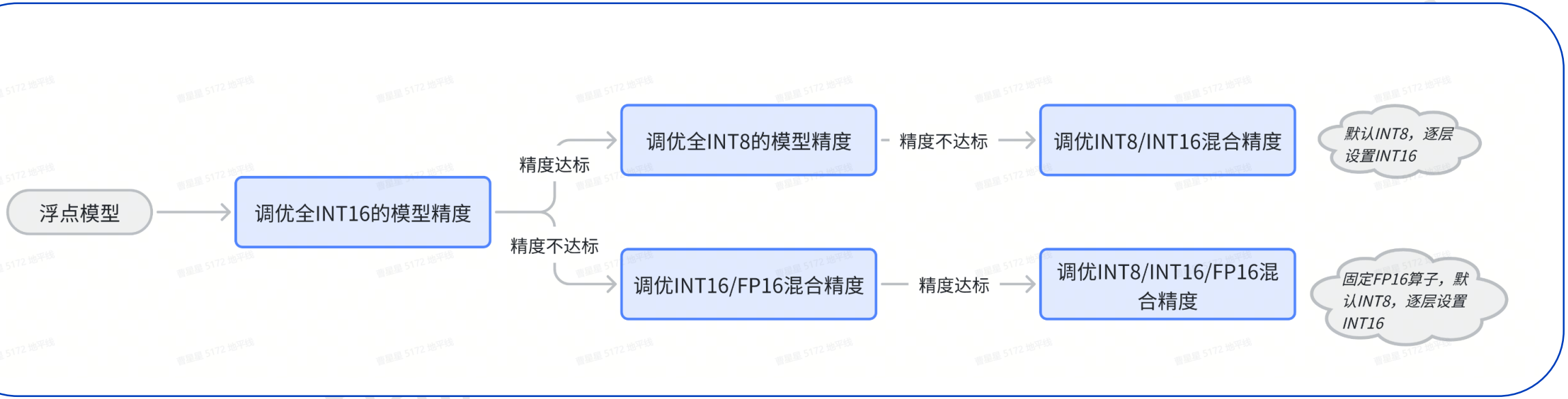
```python
from hbdk4.compiler import compile, hbm_perf
# 模型编译
compile(
    hb_quantized_model,
    march="nash-e",
    path=os.path.join(model_path, "model.hbm",
    opt=2, #J6仅支持O0-O2
    jobs=16,
    progress_bar=True
)

# 性能评测
hbm_perf(os.path.join(model_path, "model.hbm"))
```

模型编译

- 本阶段产生的hbir_qat_model和hbir_quantized_model建议使用save接口保存，该bc模型会被用于做部署适配时需要修改的基准模型

精度调优流程



整个流程首先进行全 int16 精度调优，此阶段用于确认模型的精度上限，排查工具使用问题和量化不友好模块。

1. 在确认全 int16 精度满足需求后，进行全 int8 精度调优。
2. 如果全int16达标，全int8不达标则进行 int8 / int16 混合精度调优。即在全 int8 的基础上逐步增加 int16 的比例，在满足精度的前提下，找出性能尽可能好的量化配置。
3. 如果全 int16 精度不满足需求，则进行 int16 / fp16 混合精度调优。理想情况下 int16 / fp16 混合精度调优可以解决所有精度问题。
4. 在3的基础上进行 int8 / int16 / fp16 混合精度调优，固定所有 fp16 算子的配置，按照 2 中 int8 / int16 混合精度调优的方法调整 int16 算子比例。

基础调优手段

地平线 Horizon Robotics

Calibration阶段

- 调整校准 step。校准数据越多越好，但因为边际效应的存在，当数据量大到一定程度后，对精度的提升将非常有限。

  如果训练集较小，可以全部用来 calibration，如果训练集较大，可以结合 calibration 耗时挑选大小合适的子集，

  建议至少进行 10 - 100 个 step 的校准。

- 调整 batch size。一般 batch size 要尽可能大，如果数据噪声较大或模型离群点较多，可以适当减小。

- 使用推理阶段的前处理 + 训练数据进行校准。校准数据应接近真实分布，可以使用翻转这类数据增强，不要使用旋

  转，马赛克等会改变真实分布的数据增强方法。

基础调优手段

地平线 Horizon Robotics

Qat阶段

- 调整学习率。

初始学习率：取消 warmup，取消 decay 策略，使用不同的固定学习率（1e-3, 1e-4, ...） finetune 少量 step，并观察评测指标，取效果最好的作为初始学习率。如果浮点模型不同模块学习率不同，那么这里也要做对应尝试 。

Scheduler：学习率 decay 策略与浮点对齐，但需要确保没有 warmup 类的策略。

- 固定scale 。

一般来说，校准模型精度较好时，固定 input / output scale 进行 QAT 训练可以取得更好的效果，精度较差时，则不能固定。具体使用哪种策略，没有明确指标可以参考，需要分别进行尝试。

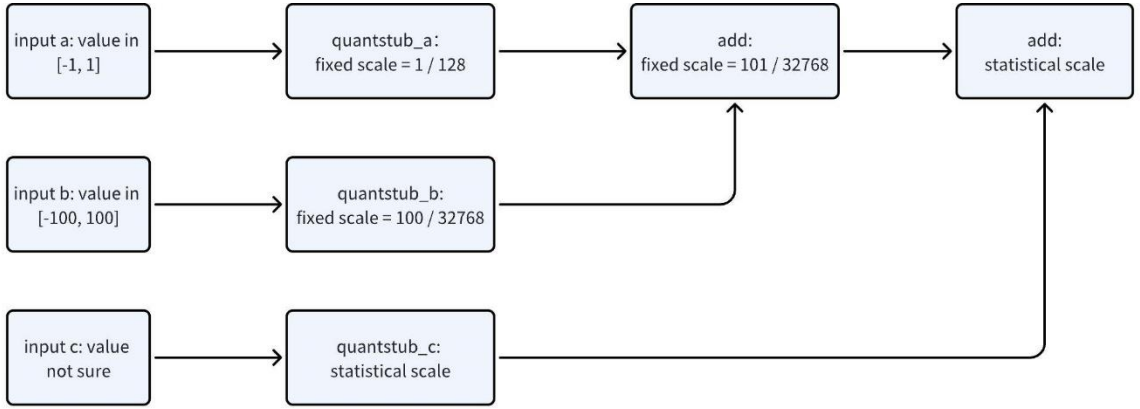
其他Tips

设置fixed scale

模型中的某些地方很难依靠统计的方式获得最佳的量化 scale。需要对于算子的输出值域确定的设置 fixed scale 。

例如: 输入数据为速度 km / h，值域为 [0, 200], 对于 quantstub 而言，输出值域是固定的，需要将 scale 设置为 (200 / 量化数值范围)。

在上面输入数据为速度的例子中，如果不设置 fixed scale，那么统计出来的最大速度可能是大多数车辆的平均速度，导致所有超过这个速度的样本在输入时就产生较大的精度损失。在精度 debug 的逐层比较步骤中，将很容易发现需要fixsale的问题。



输入 a 和 b 值域确定，输入 c 值域不确定，除 quantstub_c 和后一个 add 以外，其余算子均需要设置 fixed scale。

其他Tips

calibration

- 调整average_constant 。 average_constant表示每个 step 对最大值最小值的影响，average_constant 越小，当前 step 的影响越小，历史滑动均值的影响越大。该参数需要结合数据量在 0.01 ~ 0.5 之间调整。当数据量充足时（step > 100），average_constant 取 0.01，数据量不足时，average_constant 酌情增加。此参数应当在尝试 min max 方法时确定，之后其他方法都沿用此参数。

- 固定scale。calibration 模型精度较好时，固定 feature map 的量化参数进行 QAT 训练可以取得更好的效果，精度较差时，则不能固定 calibration 得到的量化参数。比如：某模型精度为 100，如果 calibration 精度为 50，则需要做debug查看当前calib模型是否存在异常；若精度达到90，可以尝试使用QAT找回损失的精度；如果 calibration 精度为 95，可以固定量化参数的情况下做QAT，由于该流程较短可以根据精度情况尝试不同方式。

其他Tips

calibration

- 调整校准方式。优先尝试 min max 方法（速度最快），用来跑通 calibration 流程，调整并确定 batch_size 和 average_constant 两个参数，接着分别尝试 percentile、kl、mse 和 mix 四种方法并选取效果最好的方法。

qat

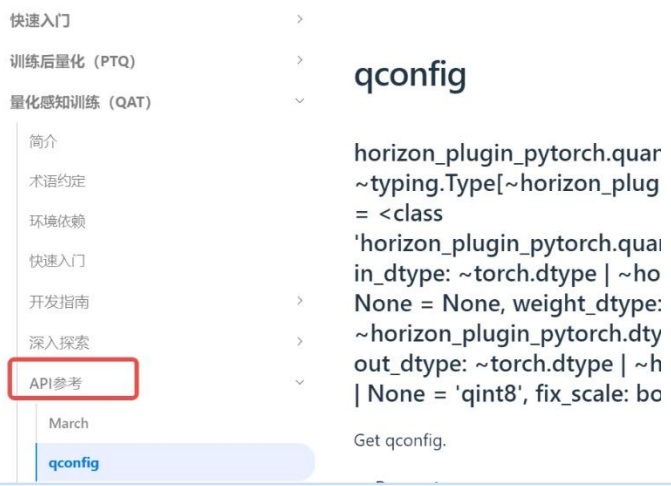
- 调整 weight decay。weight decay 会影响模型中权重的数值范围，更小的数值范围更加量化友好。有时，只调整 qat 阶段的 weight decay 还不足以解决问题，需要调整浮点训练阶段的 weight decay。

- 调整数据增强。量化模型比浮点模型的学习能力更差，太强的数据增强会影响 qat 模型收敛，一般需要适当减弱数据增强。

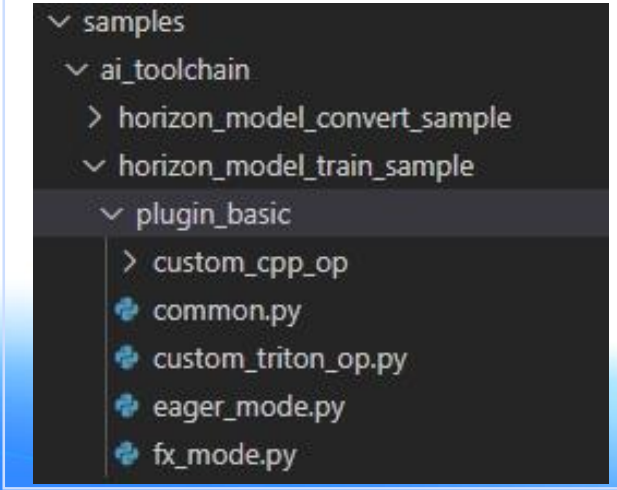
附录

用户手册

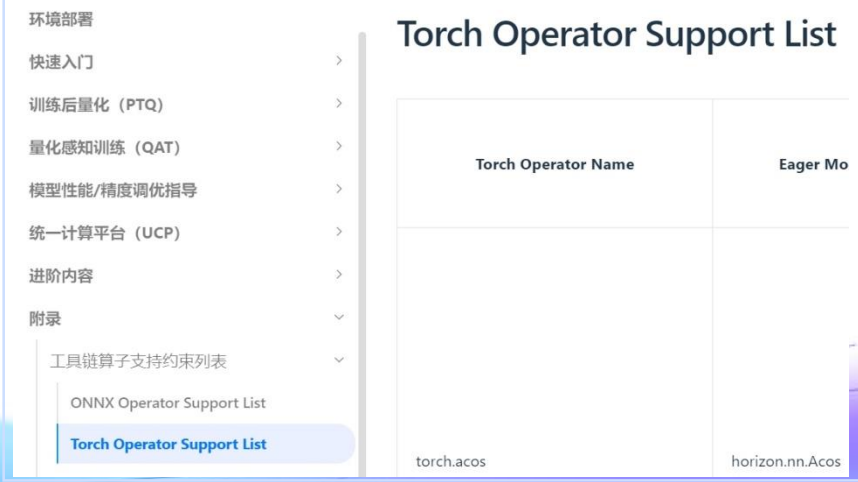
API：



Plugin sample：



算子支持列表：



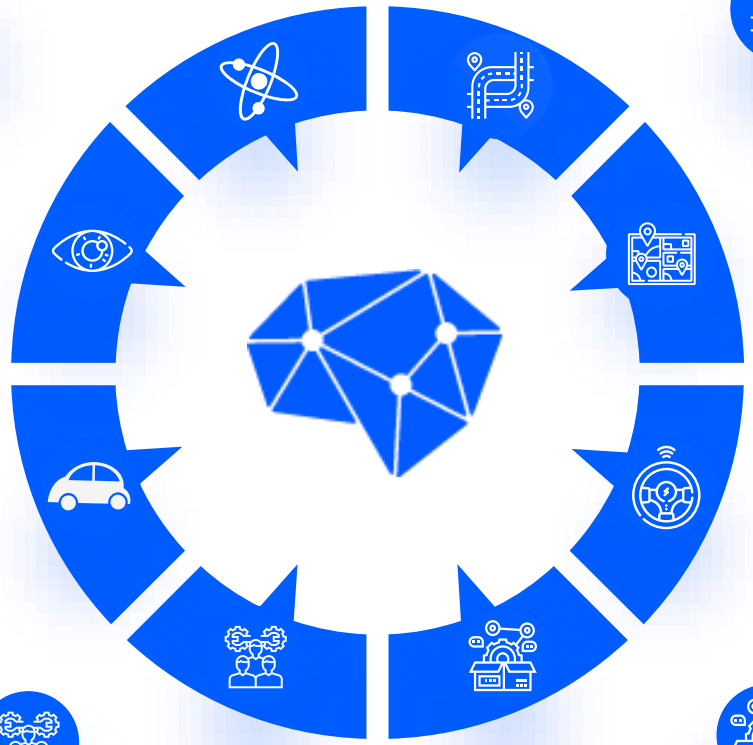
地平线
Horizon Robotics

天工开物 J6 算法工具链

模型设计-参考算法

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、...

BEV空间占用

FlashOcc,、SurroundOcc,、
TPVFormer、...

轨迹预测

QCNET、DenseTNT、...

配置文件：samples/ai_toolchain/horizon_model_train_sample/scripts/
源码（docker环境）：/usr/local/lib/python3.10/dist-packages/hat/

** 仅做参考实现，非量产算法*

高效Backbone-HENET

为了 提供针对 J6 系列芯片专门设计的高效 backbone，我们充分利用了 J6 芯片的硬件特性，设计了高效模型 HENet (Hybrid Efficient Network)

模型	输入大小	J6E 帧率 (FPS)	J6M 帧率 (FPS)	浮点精度	量化精度	数据集
HENet_TinyE	1x3x224x224	2637	3268	77.67	76.92	ImageNet
	1x3x704x1280	534	737	/	/	ImageNet
HENet_TinyM	1x3x224x224	2467	3114	78.38	77.62	ImageNet
	1x3x704x1280	444	605	/	/	ImageNet

高效Backbone-HENET

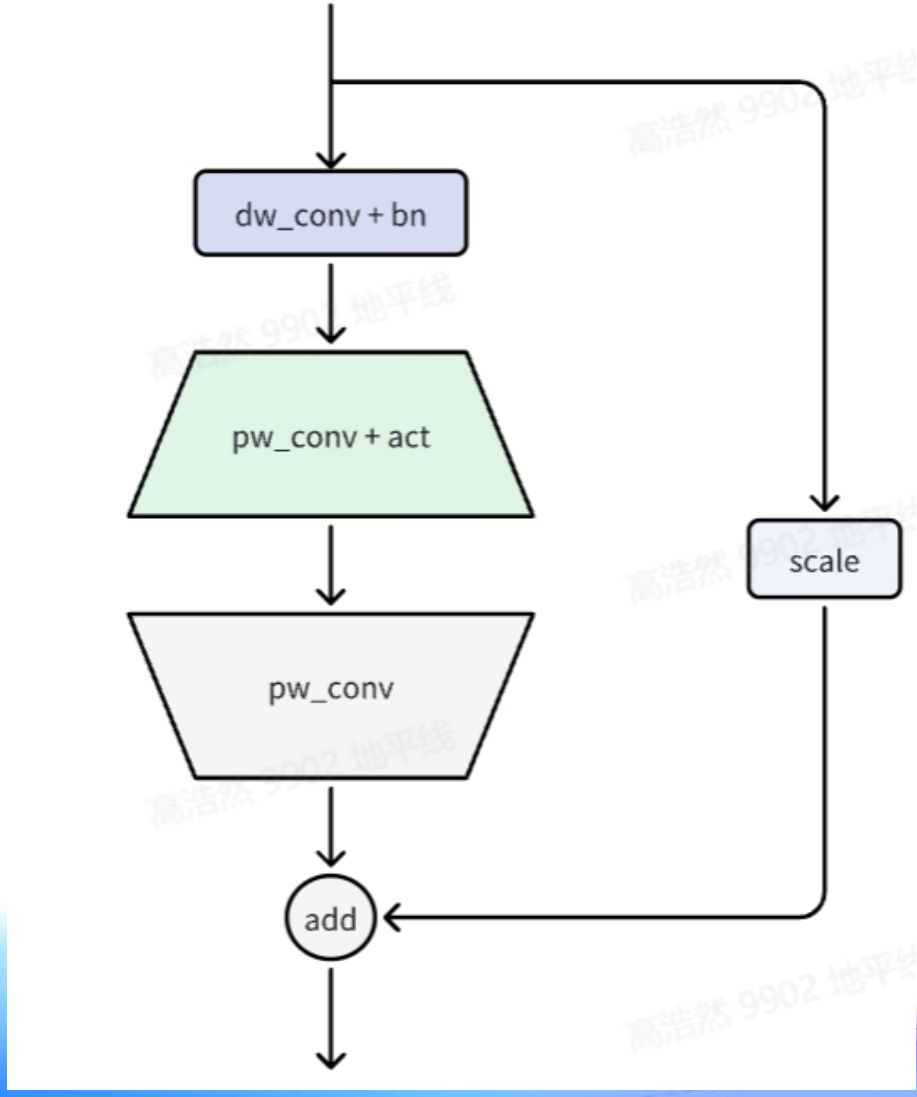
以HENet_TinyM (Hybrid Efficient Network, Tiny for J6M)为例，采用了纯 CNN 架构，总体分为四个 stage，每个 stage 会进行一次 2 倍下采样。以下是总体的结构配置：

```
# ---------------------- TinyM ----------------------
depth = [4, 3, 8, 6]
block_cls = ["GroupDWCB", "GroupDWCB", "AltDWCB", "DWCB"]
width = [64, 128, 192, 384]
attention_block_num = [0, 0, 0, 0]
mlp_ratios, mlp_ratio_attn = [2, 2, 2, 3], 2
act_layer = ["nn.GELU", "nn.GELU", "nn.GELU", "nn.GELU"]
use_layer_scale = [True, True, True, True]
final_expand_channel, feature_mix_channel = 0, 1024
down_cls = ["S2DDown", "S2DDown", "S2DDown", "None"]
```

- **depth**：每个 stage 包含的 block 数量；
- **block_cls**：每个 stage 使用的基础 block 类型；
- **width**：每个 stage 中 block 的输出 channel 数；
- **attention_block_num**：每个 stage 中的 attention_block 数量，将用在 stage 的尾部（TinyM 中没有用到）；
- **act_layer**：每个 stage 使用的激活函数；
- **use_layer_scale**：是否对 residual 分支进行可学习的缩放
- **final_expand_channel**：在网络尾部的 pooling 之前进行 channel 扩增的数量，0 代表不使用扩增；
- **feature_mix_channel**：在分类 head 之前进行 channel 扩增的数量；
- **down_cls**：每个 stage 对应的下采样类型

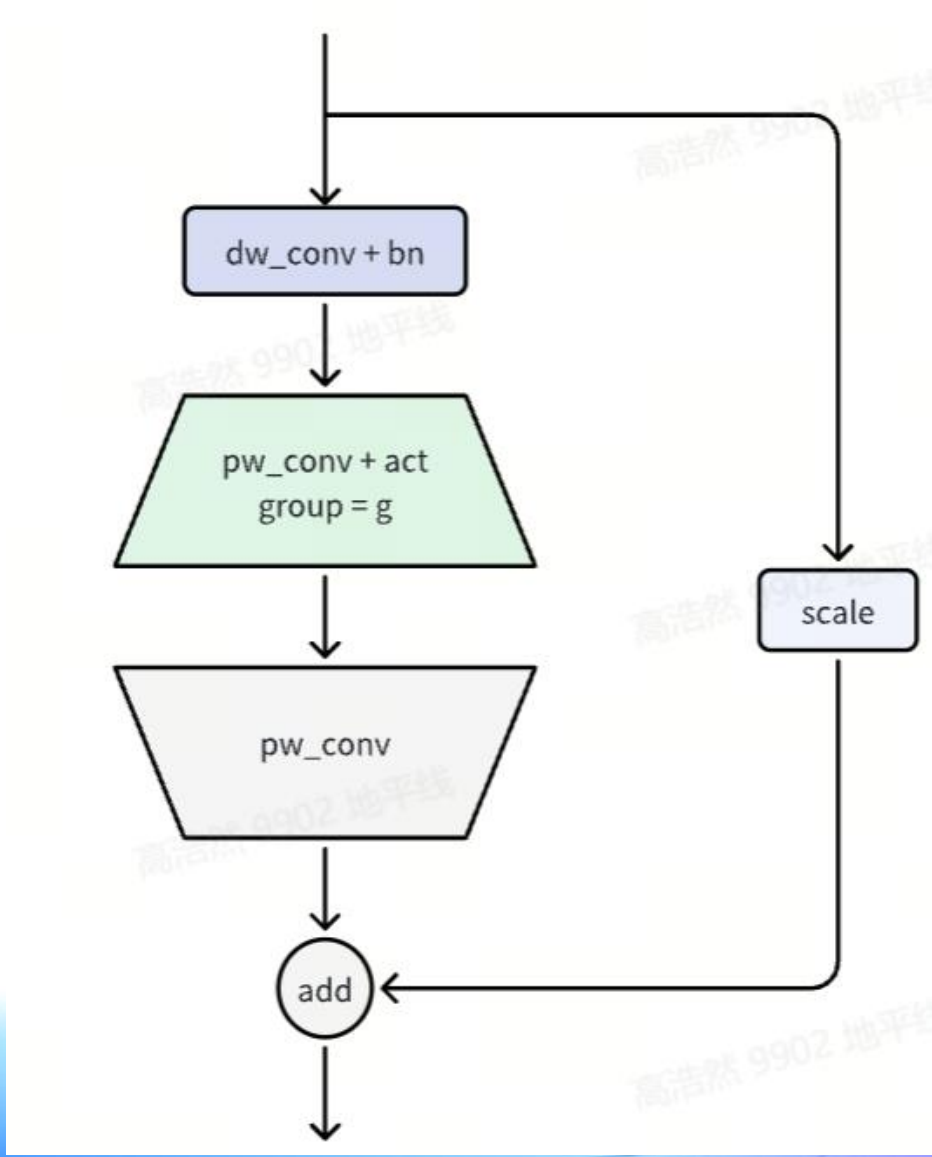
高效Backbone-HENET

基础Bock：**DWCB**



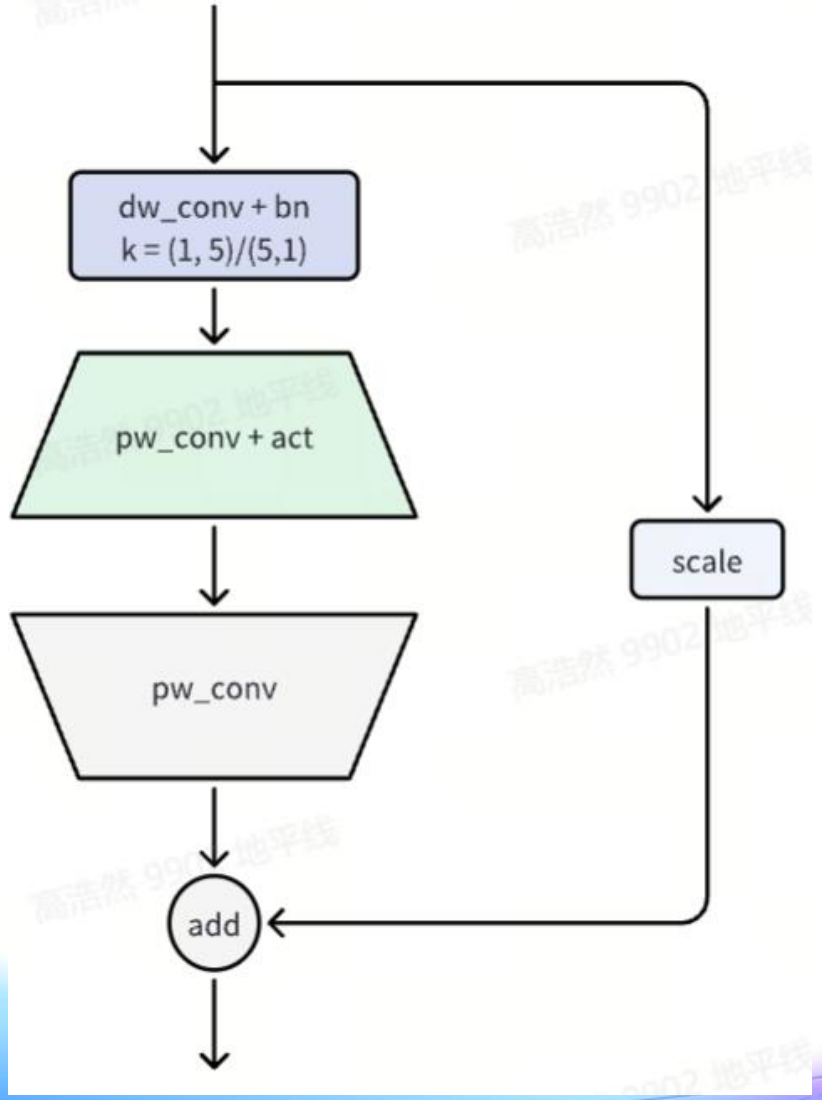
高效Backbone-HENET

基础Bock：**GroupDWCB**



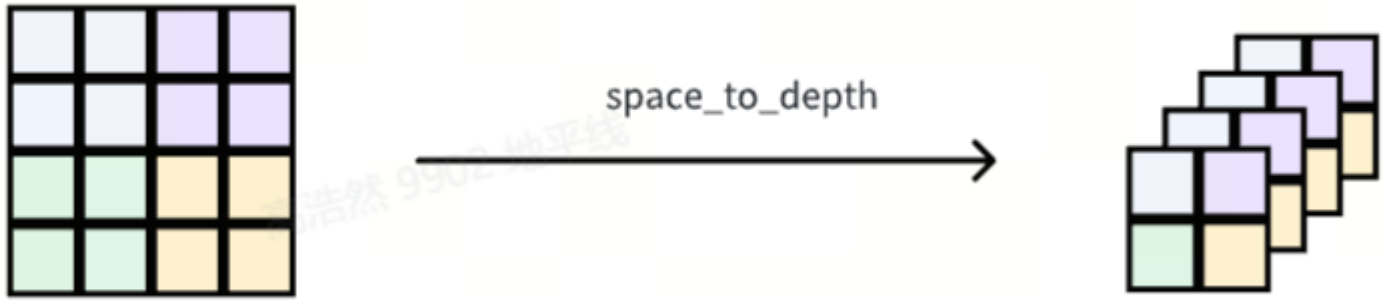
高效Backbone-HENET

基础Bock：**AltDWCB**



高效Backbone-HENET

基础Bock：**S2DDown**



S2DDown (Space2Depth Downsample) 使用一个 space to depth 操作进行降采样，以 stride = 2 为例，降采样后，feature 的空间维度将变为原来的 1/2，channel 维度将变为原来的 4 倍。对应的代码实现为：

```
N, C, H, W = x.size()  # self.bs = 2
x = x.view(N, C, H // self.bs, self.bs, W // self.bs, self.bs)  # (N, C, H//bs, bs, W//bs,
bs)
x = x.permute(0, 3, 5, 1, 2, 4).reshape(N, C * (self.bs ** 2), H // self.bs, W // self.bs)
# (N, C*bs^2, H//bs, W//bs)
```

高效Backbone-HENET使用建议

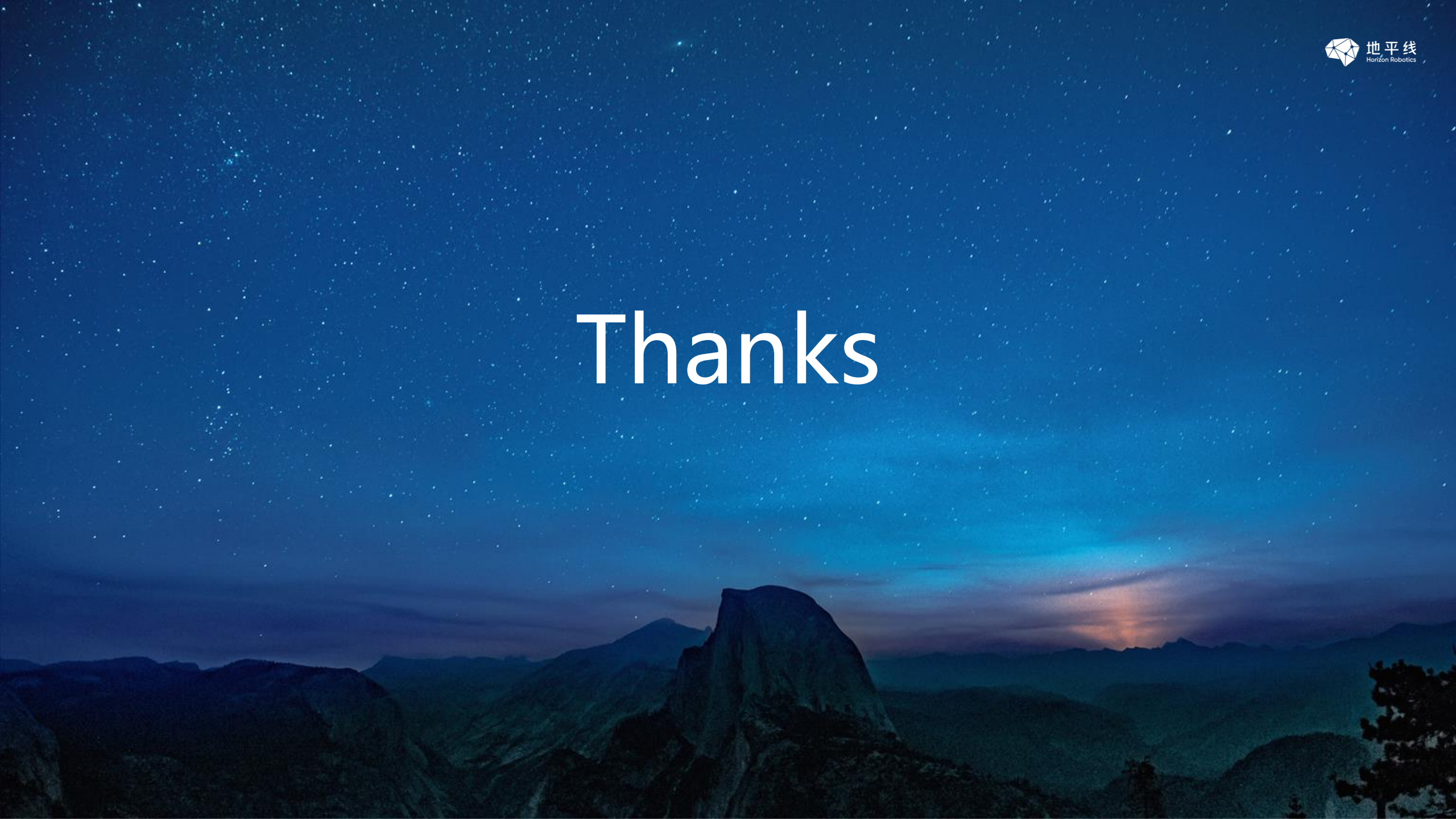
地平线 Horizon Robotics

主体结构：

1. 针对输入输出相近的场景，或者作为主 backbone 使用的场景，我们推荐直接使用 TinyM / TinyE 原生结构。针对其他场景，我们建议参考 TinyM 的基础 block 结构，灵活构建模型。
2. 对于感知模型中的多 camera backbone，针对不同类型的 camera 使用不同量级的 backbone，比如对于 front/rear camera，推荐使用较强的结构，对于 side camera，推荐使用较轻量的结构

基础Block：

1. 如需快速构建 baseline，建议先全局使用 DWCB，然后尝试 GroupDWCB、AltDWCB 等结构提升速度精度
2. 建议在满足 ① channel 数量不太小 ② 较浅的位层 两个条件的 stage 尝试使用 GroupDWCB（例如 stage 2）
3. 建议在层数较多的 stage 尝试交替使用 AltDWCB
4. 建议在 backbone 最开始的降采样中谨慎使用S2DDown，优先尝试带有 overlap 的降采样方法（例如 k=2，s=3 的 conv）


地平线
Horizon Robotics
Thanks