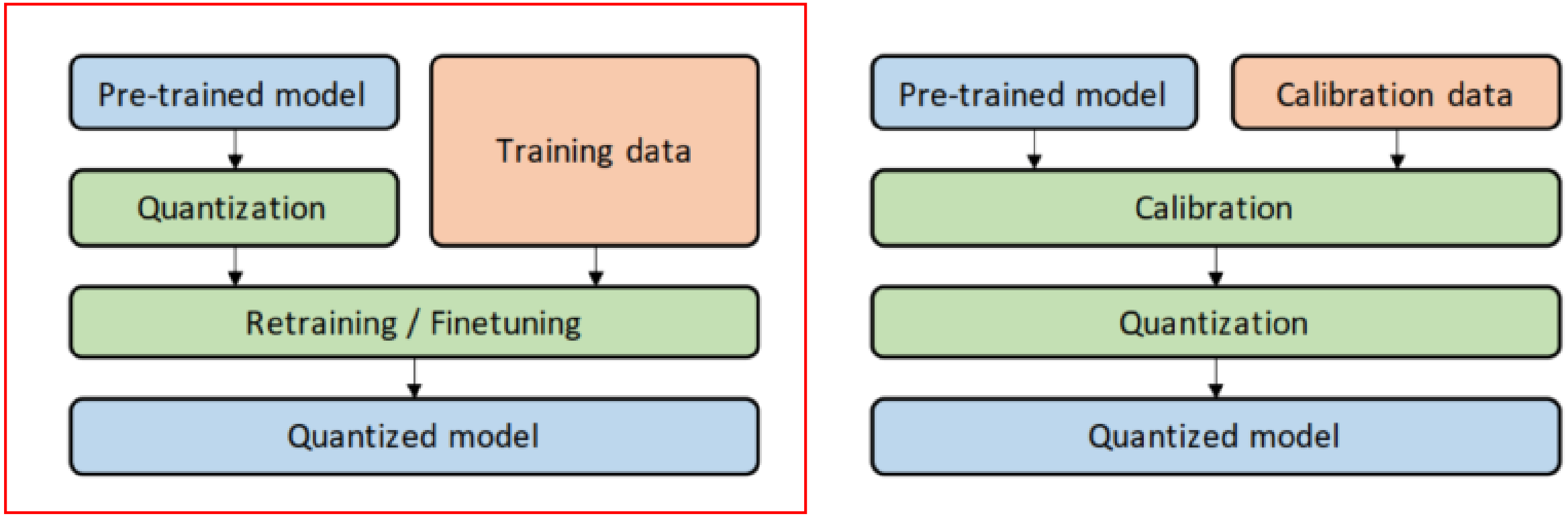
天工开物 J6 算法工具链

QAT量化与精度调试

QAT量化



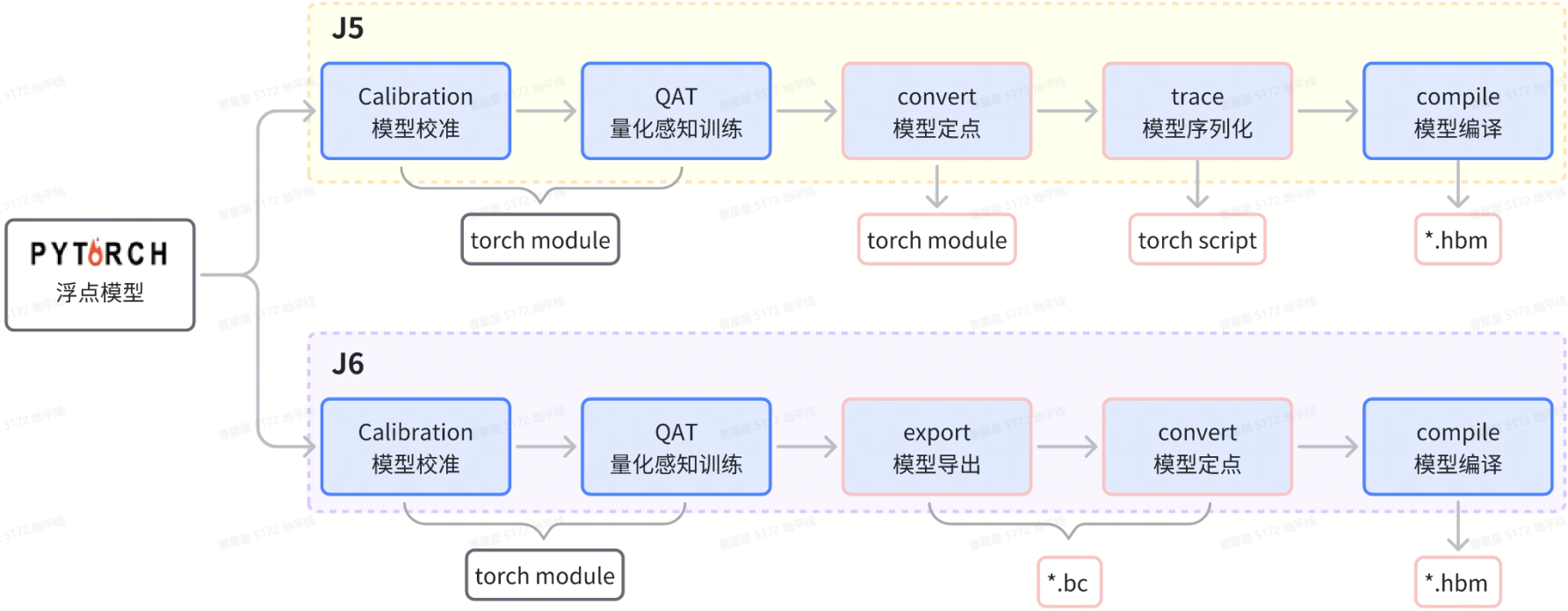
QAT（左）和PTQ（右）

QAT量化

量化训练工具目前支持的量化方式有Eager、SymbolicTrace、JitTrace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



.bc为PC端的验证模型　　.hbm为板端部署模型

QAT部署流程

Calibration	Qat	Quantized	Compile

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
- Batchsize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部署流程

Calibration	Qat	Quantized	Compile

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，可以固定住量化的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)),
)
```

QAT部署流程

Calibration | Qat | Quantized | Compile

定点转换

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）

- 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

QAT部署流程

Calibration | Qat | Quantized | Compile

部署适配-指定模型输入输出节点名

地平线 Horizon Robotics

指定模型输入输出节点名

在qat.bc上执行，指定 J6 hbm 模型的输入输出 tensor 的 name 属性。

```
from horizon_plugin_pytorch.quantization.hbdk4 import export

hbir_qat_model = export(
    base_model, (example_input,),
    name="my_model",
    input_names=("input_name1","input_name2"),
    output_names=("output_name1","output_name2")
    )
# ps: export的name参数，可用于指定模型名称，即工程部署时的 model name字段
```

方式1：在 export 的时候通过 input_names 以及 output_names 参数指定

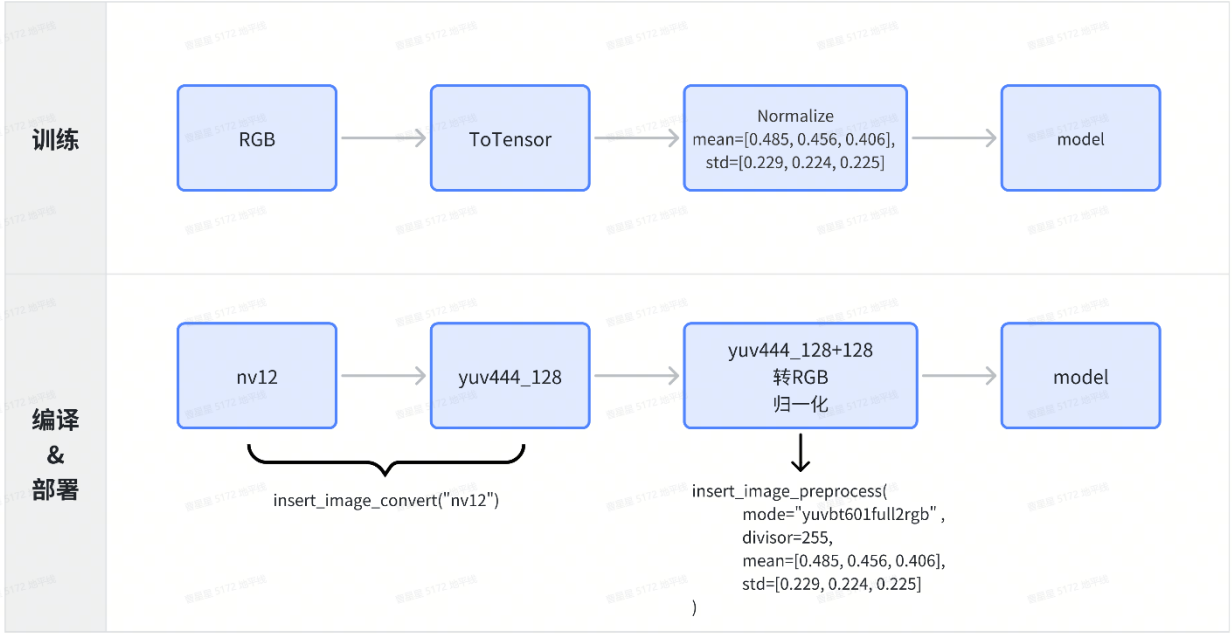
方式2：修改 *.bc 文件的输入输出节点 name 属性

```
from hbdk4.compiler import load,compile

model = load("qat.bc")
func = model.functions[0]
func.inputs[0].name = "input"
func.outputs[0].name = "output"

quantized_bc = convert(model, "nash-e")
compile(
  quantized_bc,
  march="nash-e",
  path="./*.hbm"
)
```

部署适配-插入前处理节点

插入前处理节点

在qat.bc上执行，插入前处理节点，完成训练和部署图像格式之间的转换，以及数据归一化操作。



前处理流程

```python
from hbdk4.compiler import load,convert, compile,

model = load("qat.bc")
func = model.functions[0]

resizer_input = ["resize"]      # 部署时数据来源于resizer的输入节点名称列表
pyramid_input = ["pym"]         # 部署时数据来源于pyramid的输入节点名称列表

for name in pyramid_input[::-1]:
    for input in func.inputs[::-1]:
        if name == input.name:
            # pyramid&resizer 目前只支持 NHWC 的 input layout
            input.insert_transpose(permutes=[0, 3, 1, 2])
            # 插入前处理节点，具体可参考下一节的说明
            input.insert_image_preprocess(
                mode=None,
                divisor=1,
                mean=[128, 128, 128],
                std=[128, 128, 128]
                )
            input.insert_image_convert("nv12")
```

```python
for name in resizer_input[::-1]:
    for input in func.inputs[::-1]:
        if name == input.name:
            # pyramid&resizer 目前只支持 NHWC 的 input layout
            input.insert_transpose(permutes=[0, 3, 1, 2])
            # 插入前处理节点，具体可参考下一节的说明
            input.insert_image_preprocess(
                mode=None,
                divisor=1,
                mean=[128, 128, 128],
                std=[128, 128, 128]
                )
            input.insert_roi_resize("nv12")
quantized_bc = convert(model, "nash-e")
compile(quantized_bc,march="nash-e",path="./*.hbm")
```

Yuv444训练

部署适配-删除指定节点

删除指定节点

在quantized.bc上执行，去除量化反量化节点以及尾部的reshape、transpose等可以融入后处理的节点。

```python
import os
from hbdk4.compiler import load, compile, save

# 删除节点
node_type_mapping = {
    "qnt.quantize": "Quantize",
    "qnt.dequantize": "Dequantize",
    "hbir.transpose": "Transpose",
    "hbtl.call::quant::qcast": "Quantize",
    "hbtl.call::quant::dcast": "Dequantize",
    "hbtl.call::native::Transpose": "Transpose",
    "hbir.cast_type": "Cast",
    "hbir.reshape": "Reshape",
    "hbtl.call::native::Cast": "Cast",
    "hbtl.call::native::Reshape": "Reshape",
}

quantized_bc = load("quantized.bc")
func = quantized_bc.functions[0]
# 删除argmax后的类型转换节点
remove_op(func, op_type="Reshape")
remove_op(func, op_type="Cast")
# 删除量化反量化节点
remove_op(func, op_type="Dequantize")
remove_op(func, op_type="Quantize")
# 删除max后的类型转换节点
remove_op(func, op_type="Reshape")

save(quantized_bc, "./quantized_model_remove.bc")
```

```python
def get_type_for_hbtl_call(attached_op):
    schema = attached_op.schema
    node_type = attached_op.type + "::" + \
        schema.namespace + "::" + schema.signature
    return node_type
def remove_op(func, op_type=None, op_name=None):
    for loc in func.inputs + func.outputs:
        if not loc.is_removable[0]:
            continue
        attached_op = loc.get_attached_op[0]
        removed = None
        # 目前hbir模型中的op name格式还未完全确定，暂建议使用op type来删除节点
        attached_op_name = attached_op.name
        if op_name and attached_op.name in op_name:
            removed, diagnostic = loc.remove_attached_op()
        elif op_type and attached_op.type in node_type_mapping.keys() \
                and node_type_mapping[attached_op.type] in op_type:
            removed, diagnostic = loc.remove_attached_op()
        elif attached_op.type == "hbtl.call":
            # 由于同一type的op在后端可能对应多种实现，因此采用"签名"的方式确认具体类型
            node_type = get_type_for_hbtl_call(attached_op)
            if op_type and node_type in node_type_mapping.keys() \
                    and node_type_mapping[node_type] in op_type:
                removed, diagnostic = loc.remove_attached_op()
        if removed is True:
            print(f'Remove node', op_type, "successfully")
        if removed is False:
            raise ValueError(f'Remove node type', op_type,
                f"Failed when deleting {attached_op.name} operator,"
                f"error: {diagnostic}")
```

部署适配-Batch拆分

多batch拆分

在qat.bc上执行，J6 不支持 Batch>1 的 Pyramid/Resizer 模型进行推理，因此需要将 batch 输入显式地拆分成 多个batch1输入。

```python
from hbdk4.compiler import load

model = load("qat_model.bc")
func = model.functions[0]
# 依据部署需要，维护一个独立地址部署的输入节点名称列表，或index列表
# 输入节点名采用上一节的方式进行了自定义修改
batch_input = ["input_name1"]
for input in func.inputs[::-1]:
    for name in batch_input[::-1]:
        if name in input.name:
            input.insert_split(dim=0)
# ps:
#     1.insert_split会导致节点数变多，因此建议从后往前拆
#     2.就算batch=1，该接口也会尝试拆分，会导致输入节点的name增加“_0”后缀
```

```
input[0]:
name: _input_0_0_y
input source: HB_DNN_INPUT_FROM_DDR
valid shape: (1,512,960,1,)
aligned shape: (0,0,0,0,)
aligned byte size: -1
tensor type: HB_DNN_TENSOR_TYPE_U8
tensor layout: HB_DNN_LAYOUT_NONE
quanti type: NONE
stride: (-1,-1,1,1,)

input[1]:
name: _input_0_0_uv
input source: HB_DNN_INPUT_FROM_DDR
valid shape: (1,256,480,2,)
aligned shape: (0,0,0,0,)
aligned byte size: -1
tensor type: HB_DNN_TENSOR_TYPE_U8
tensor layout: HB_DNN_LAYOUT_NONE
quanti type: NONE
stride: (-1,-1,2,1,)
```

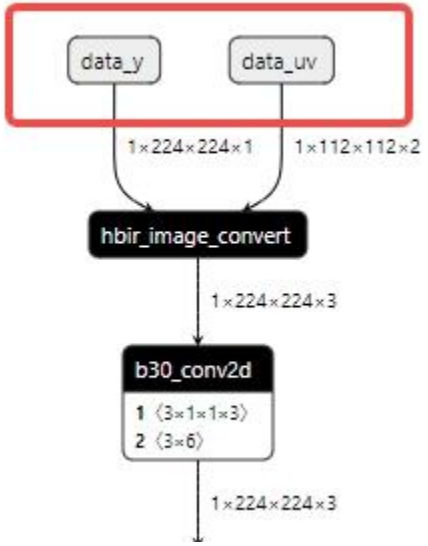
batch6输入会被拆分为6个batch1的y、uv输入

目的：由于在图像通路上，通常Y、UV是独立的地址（Pyramid输出结果即为Y和UV两个独立地址），特别是对于在原图上Crop ROI区域时，Y、UV一定是非连续的，因此J6工具链将输入来源为Pyramid的模型输入直接定义为Y和UV两个Tensor，不再区分NV12和NV12_SEP两种类型。

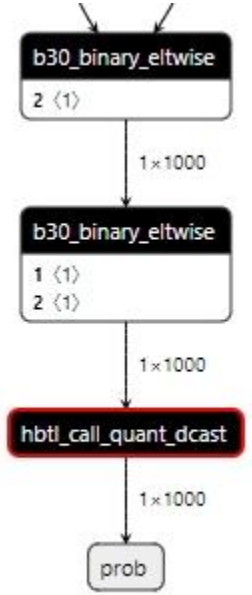
部署适配-模型检查与编译

模型检查与编译

查看前处理节点是否插入



查看模型是否包含cpu算子



反量化节点

可视化工具：hb_model_info –v quantized.bc

符合要求

```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00-02
    jobs=16,
    progress_bar=True
)

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

模型编译

bc模型推理

Pyramid推理

```python
hbir = load("./model_output/mobilenetv1_224x224_nv12_quantized_model.bc")
img = "../../01_common/test_data/cls_images/zebra_cls.jpg"
# 准备输入数据
image = cv2.imread(img)
inputs = hbir[0].inputs
model_shape = inputs[0].type.shape
image= cv2.resize(image, (model_shape[2],model_shape[1]))
Y, UV = image2nv12(image)

# 支持通过model_name以及index来获取模型（与部署侧保持一致）
# 推理方式 1，输入输出均为 dict
# 准备输入（支持dict of np.array 以及 torch.tensor）
inputs = {inputs[0].name: Y, inputs[1].name: UV}
hbir_outputs = hbir[0].feed(inputs)
# hbir_outputs = hbir["mobilenetv1_224x224_resizer"].feed(inputs)
from horizon_tc_ui.data.imagenet_val import imagenet_val
def postprocess(model_output):
    prob = np.squeeze(model_output[hbir[0].outputs[0].name])
    ...
```

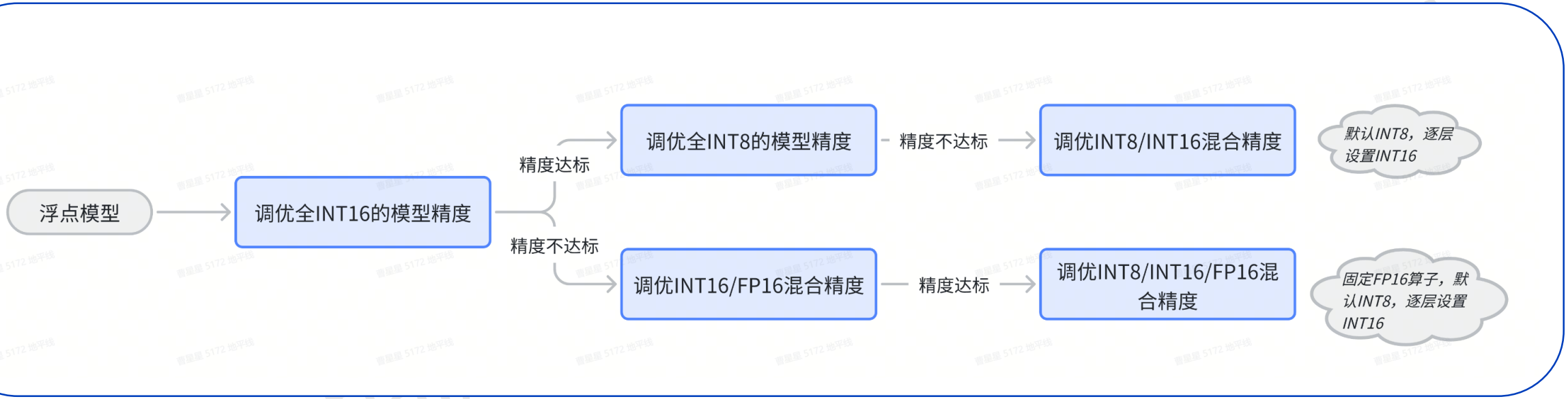
```python
def image2nv12(image):
    image = image.astype(np.uint8)
    height, width = image.shape[0], image.shape[1]
    yuv420p = cv2.cvtColor(
        image,
        cv2.COLOR_BGR2YUV_I420
        ).reshape((height * width * 3 // 2, ))
    y = yuv420p[:height * width].reshape(1, height, width, 1)
    uv_planar = yuv420p[
        height * width:
        ].reshape((1, 2, height // 2, width // 2))
    uv = uv_planar.transpose((0, 2, 3, 1))

    return y, uv
```

Resizer推理

```python
roi = torch.tensor([[0, 0, 128, 128]], dtype=inputs[2].type.torch_dtype)
inputs = {inputs[0].name: Y, inputs[1].name: UV, inputs[2].name: roi}
hbir_outputs = hbir[0].feed(inputs)
```

QAT精度调优

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。

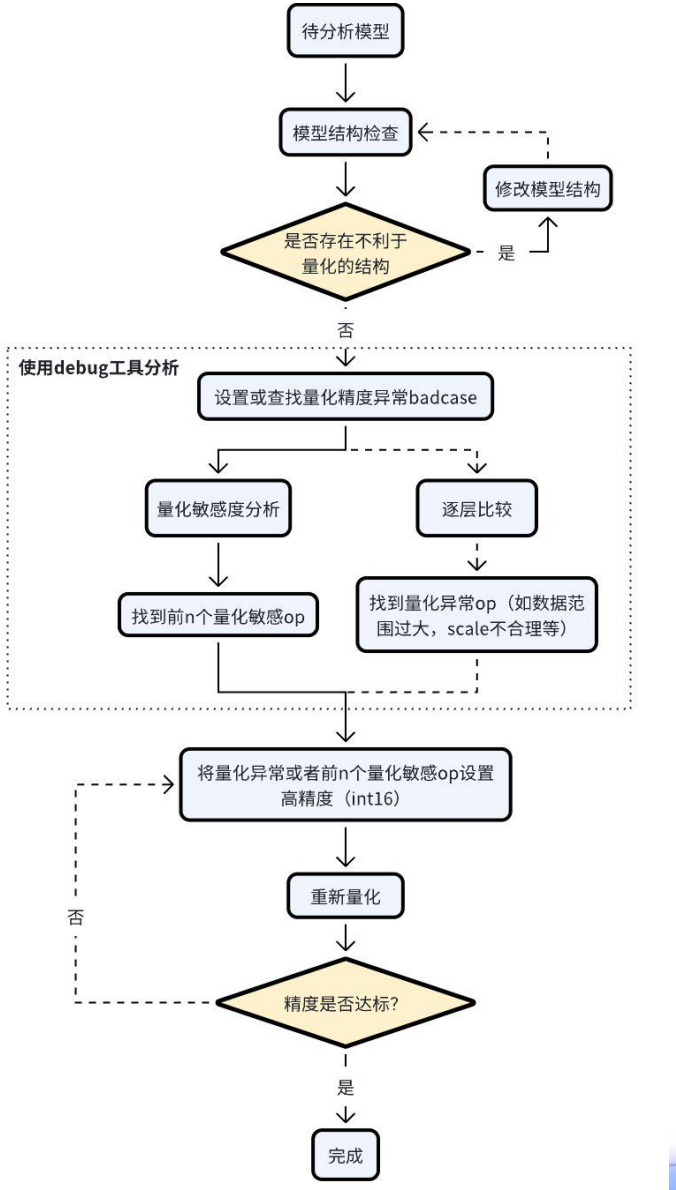
初始学习率：取消 warmup，取消 decay 策略，使用不同的固定学习率（1e-3, 1e-4, ...） finetune 少量 step，并观察评测指标，取效果最好的作为初始学习率。如果浮点模型不同模块学习率不同，那么这里也要做对应尝试 。

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。

精度Debug


待分析模型
模型结构检查
修改模型结构
是否存在不利于量化的结构
是
否
使用debug工具分析
设置或查找量化精度异常badcase
量化敏感度分析
逐层比较
找到前n个量化敏感op
找到量化异常op（如数据范围过大，scale不合理等）
将量化异常或者前n个量化敏感op设置高精度（int16）
重新量化
否
精度是否达标?
是
完成

模型检查类

```
def check_qat_model(
    model: torch.nn.Module,
    example_inputs: Any,
    save_results: bool = False,
    out_dir: Optional[str] = None,
):
```

功能说明

在 save_results = True 时生成model_check_result.txt。主要由5部分组成：

1. 未 fuse 的 pattern。
2. 每个 module 的调用次数。正常每个 op 仅调用 1 次，0 表示未被调用，超过 1 次则表示被共享了多次。
3. 每个 op 输出的 qconfig 配置。
4. 每个 op weight（如果有的话）的 qconfig 配置。
5. 异常 qconfig 提示（如果有的话）。

```
Fusable modules are listed below:
name    type
------  ------------------------------------------------------
conv    <class 'horizon_plugin_pytorch.nn.qat.conv2d.Conv2d'>
relu    <class 'horizon_plugin_pytorch.nn.qat.relu.ReLU'>
```

融合模块检查

```
Each module called times:
name        called times
-------  --------------
conv                  1
relu                  1
quant                 1
dequant               1
```

共享模块检查

model_check_result.txt

模型检查类

```
def check_qat_model(
    model: torch.nn.Module,
    example_inputs: Any,
    save_results: bool = False,
    out_dir: Optional[str] = None,
):
```

功能说明

在 save_results = True 时生成model_check_result.txt。主要由5部分组成：

1. 未 fuse 的 pattern。
2. 每个 module 的调用次数。正常每个 op 仅调用 1 次，0 表示未被调用，超过 1 次则表示被共享了多次。
3. 每个 op 输出的 qconfig 配置。
4. 每个 op weight 的 qconfig 配置。
5. 异常 qconfig 提示。

```
Each layer out qconfig:
+--------------+-----------------------------------------------------------+--------------+--------------+---------------+-----------------------------+
| Module Name  | Module Type                                               | Input dtype  | out dtype    | ch_axis       | observer                    |
|--------------+-----------------------------------------------------------+--------------+--------------+---------------+-----------------------------|
| quant        | <class 'horizon_plugin_pytorch.nn.qat.stubs.QuantStub'>   | torch.float32 | qint8        | -1            | MovingAverageMinMaxObserver |
| conv         | <class 'horizon_plugin_pytorch.nn.qat.conv2d.Conv2d'>     | qint8        | qint8        | -1            | MovingAverageMinMaxObserver |
| relu         | <class 'horizon_plugin_pytorch.nn.qat.relu.ReLU'>         | qint8        | qint8        | qconfig = None |                             |
| dequant      | <class 'horizon_plugin_pytorch.nn.qat.stubs.DeQuantStub'> | qint8        | torch.float32 | qconfig = None |                             |
+--------------+-----------------------------------------------------------+--------------+--------------+---------------+-----------------------------+

Weight qconfig:
+--------------+-----------------------------------------------------+---------------+-----------+-------------------------------------+
| Module Name  | Module Type                                         | weight dtype  |   ch_axis | observer                            |
|--------------+-----------------------------------------------------+---------------+-----------+-------------------------------------|
| conv         | <class 'horizon_plugin_pytorch.nn.qat.conv2d.Conv2d'> | qint8         |         0 | MovingAveragePerChannelMinMaxObserver |
+--------------+-----------------------------------------------------+---------------+-----------+-------------------------------------+
```

model_check_result.txt

QuantAnalysis类-自动搜索badcase

```
class QuantAnalysis(object):
    def __init__(
        self,
        baseline_model: Union[torch.nn.Module, HbirModule],
        analysis_model: Union[torch.nn.Module, HbirModule],
        analysis_model_type: str,
        device_ids: Union[List[int], int] = None,
        post_process: Optional[Callable] = None,
        out_dir: Optional[str] = None,
    )
```

功能说明

compare_per_layer：比较两个模型中每一层的结果。

sensitivity：模型中各个节点的敏感度排序。适用于 float 转 calibration/qat 的精度掉点问题。

auto_find_bad_case：自动寻找导致两个模型输出最差的 badcase。

```
from horizon_plugin_profiler.model_profilerv2 import QuantAnalysis
qa = QuantAnalysis(float_net, calibration_model, "fake_quant")
qa.auto_find_bad_case(dataloader)#量化精度较差的样本集
qa.run()
```

The bad case input index of each output:

Name/Index	Cosine	MSE	L1	KL	SQNR
box0	16	0	0	97	16
box1	77	32	32	81	77
box2	66	46	56	79	66
scores	61	96	100	96	60
centerness	0	0	0	0	0
yawness	17	76	18	18	39

badcase.txt

The metric results of each badcase:

Name/Index	Cosine	MSE	L1	KL	SQNR
box0	0.906329	84.7319	2.97311	0.323674	7.40183
box1	0.968623	38.602	2.98777	0.268769	11.4491
box2	0.799405	237.053	4.37895	0.0386395	4.28581
scores	0.388762	0.0080612	0.0395061	1.43469e-05	-0.749675
centerness	0.904206	0.0456813	0.178009	4.34062e-05	5.28536
yawness	-0.325684	4.87353	1.27329	0.141269	-3.66645

The bad case input index of the worst output:

metric	dataloader index
Cosine	17
MSE	46
L1	56
KL	97
SQNR	39

QuantAnalysis类-逐层相似度对比

```
class QuantAnalysis(object):
    def __init__(
        self,
        baseline_model: Union[torch.nn.Module, HbirModule],
        analysis_model: Union[torch.nn.Module, HbirModule],
        analysis_model_type: str,
        device_ids: Union[List[int], int] = None,
        post_process: Optional[Callable] = None,
        out_dir: Optional[str] = None,
    )
```

功能说明

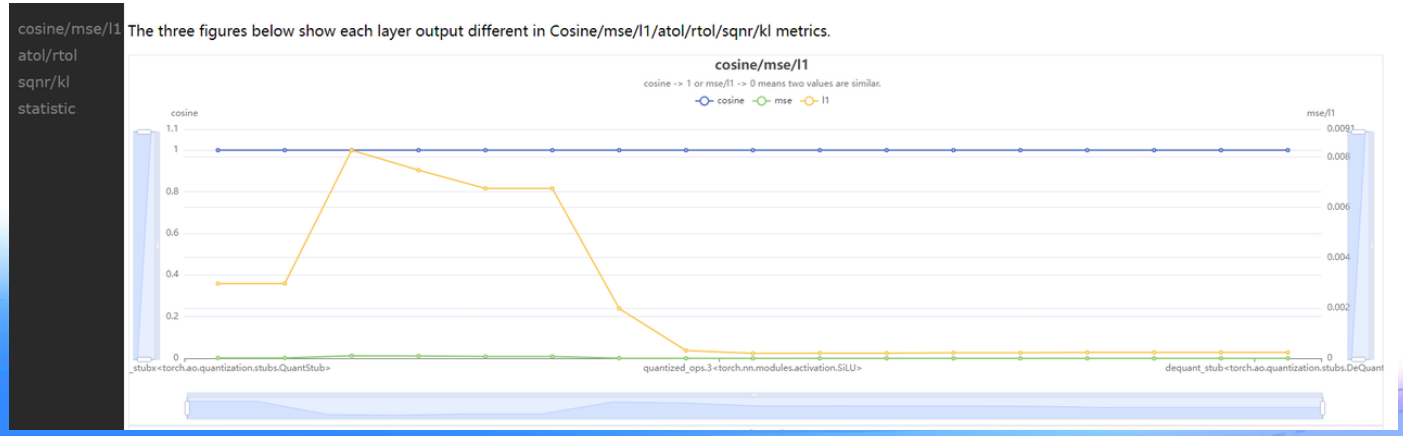
compare_per_layer：比较两个模型中每一层的结果。

sensitivity：模型中各个节点的敏感度排序。适用于 float 转 calibration/qat 的精度掉点问题。

auto_find_bad_case：自动寻找导致两个模型输出最差的 badcase。

```
from horizon_plugin_profiler.model_profilerv2 import QuantAnalysis
qa = QuantAnalysis(float_net, calibration_model, "fake_quant")
#可在此步骤插入auto_find_bad_case
qa.compare_per_layer()
```

analy_op_type	shape	quant_dtype	qscale	Cosine	MSE	L1	KL	SQNR	Atol
horizon_plugin_pytorch.nn.qat.stubs.QuantStub	torch.Size([1, 3, 32, 32])	qint8	0.0078354	0.9999924	0.0000052	0.0019757	0.0000006	48.1179886	0.0039178
horizon_plugin_pytorch.nn.qat.conv2d.Conv2d	torch.Size([1, 3, 32, 32])	qint8	0.0060428	0.9999037	0.0000085	0.0023614	0.0000012	37.1519432	0.0096008
horizon_plugin_pytorch.nn.qat.relu.ReLU	torch.Size([1, 3, 32, 32])	qint8	0.0060428	0.9998640	0.0000037	0.0010231	0.0000004	35.5429153	0.0093980
horizon_plugin_pytorch.nn.interpolate.autocasted_interpolate_outer	torch.Size([1, 3, 41, 41])	qint8	0.0060428	0.9234583	0.0012933	0.0245362	0.0001882	8.1621437	0.1928777	3
horizon_plugin_pytorch.nn.qat.stubs.DeQuantStub	torch.Size([1, 3, 41, 41])	torch.float32		0.9234583	0.0012933	0.0245362	0.0001882	8.1621437	0.1928777	3



compare_per_layer_out.txt:

profiler.html

QuantAnalysis类-敏感节点排序

```
class QuantAnalysis(object):
    def __init__(
        self,
        baseline_model: Union[torch.nn.Module, HbirModule],
        analysis_model: Union[torch.nn.Module, HbirModule],
        analysis_model_type: str,
        device_ids: Union[List[int], int] = None,
        post_process: Optional[Callable] = None,
        out_dir: Optional[str] = None,
    )
```

功能说明

compare_per_layer：比较两个模型中每一层的结果。

sensitivity：模型中各个节点的敏感度排序。适用于 float 转 calibration/qat 的精度掉点问题。

auto_find_bad_case：自动寻找导致两个模型输出最差的 badcase。

```
from horizon_plugin_profiler.model_profilerv2 import QuantAnalysis
qa = QuantAnalysis(float_net, calibration_model, "fake_quant")
#可在此步骤插入auto_find_bad_case
qa.sensitivity()
```

```
op_name    sensitive_type    op_type                                                    L1        quant_dtype
---------  ----------------  ---------------------------------------------------------  --------  -------------
quant      activation        <class 'horizon_plugin_pytorch.nn.qat.stubs.QuantStub'>    0.0245567  qint8
conv       activation        <class 'horizon_plugin_pytorch.nn.qat.conv2d.Conv2d'>      0.0245275  qint8
conv       weight            <class 'horizon_plugin_pytorch.nn.qat.conv2d.Conv2d'>      0.024501   qint8
```

sensitive_ops.pt

```
qat_model = prepare(
    model,
    example_inputs=example_inputs, # 用来感知图结构
    qconfig_setter=( # qconfig 模板，支持传入多个模板，优先级从高到低。
        sensitive_op_qat_8bit_weight_16bit_act_qconfig_setter(table, ratio=0.2),
        default_qat_qconfig_setter,
    ),
)
```

敏感节点配置使用示例

ModelProfiler类

```
class ModelProfiler(object):
    def __init__(
        self,
        model: torch.nn.Module,
        out_dir: str,
    )
```

功能说明

统计模型 forward 过程中，每一层算子的输入输出等信息。

```
with ModelProfiler(net, "./profiler_dir") as p:
    net(data)

p.get_info_manager.table()
p.get_info_manager.tensorboard()
```

Index	Op Name	Mod Name	Attr	Dtype	Scale	Min	Max	Mean	Var	Shape
0	horizon_plugin_pytorch.nn.qat.stubs.QuantStub	quant	input	torch.float32		0.0003164	0.9990171	0.5015678	0.0846284	torch.Size([1, 3, 32, 32])
0	horizon_plugin_pytorch.nn.qat.stubs.QuantStub	quant	output	qint8	0.0078354	0.0000000	0.9950994	0.5014852	0.0846521	torch.Size([1, 3, 32, 32])
1	horizon_plugin_pytorch.nn.qat.conv2d.Conv2d	conv	input	qint8	0.0078354	0.0000000	0.9950994	0.5014852	0.0846521	torch.Size([1, 3, 32, 32])
1	horizon_plugin_pytorch.nn.qat.conv2d.Conv2d	conv	weight	torch.float32		-0.5315086	0.5750652	0.0269936	0.1615299	torch.Size([3, 3, 1, 1])
1	horizon_plugin_pytorch.nn.qat.conv2d.Conv2d	conv	bias	torch.float32		-0.4963555	0.4448483	-0.0851902	0.2320642	torch.Size([3])
1	horizon_plugin_pytorch.nn.qat.conv2d.Conv2d	conv	output	qint8	0.0060428	-0.7674332	0.4652941	-0.0412943	0.0422743	torch.Size([1, 3, 32, 32])
2	horizon_plugin_pytorch.nn.qat.relu.ReLU	relu	input	qint8	0.0060428	-0.7674332	0.4652941	-0.0412943	0.0422743	torch.Size([1, 3, 32, 32])
2	horizon_plugin_pytorch.nn.qat.relu.ReLU	relu	output	qint8	0.0060428	0.0000000	0.4652941	0.0639115	0.0089839	torch.Size([1, 3, 32, 32])
3	horizon_plugin_pytorch.nn.interpolate.autocasted_interpolate_outer		input	qint8	0.0060428	0.0000000	0.4652941	0.0639115	0.0089839	torch.Size([1, 3, 32, 32])
3	horizon_plugin_pytorch.nn.interpolate.autocasted_interpolate_outer		output	qint8	0.0060428	0.0000000	0.3504813	0.0639483	0.0043366	torch.Size([1, 3, 41, 41])
4	horizon_plugin_pytorch.nn.qat.stubs.DeQuantStub	dequant	input	qint8	0.0060428	0.0000000	0.3504813	0.0639483	0.0043366	torch.Size([1, 3, 41, 41])
4	horizon_plugin_pytorch.nn.qat.stubs.DeQuantStub	dequant	output	torch.float32		0.0000000	0.3504813	0.0639483	0.0043366	torch.Size([1, 3, 41, 41])

statistic.txt

可通过统计量来确认op的量化精度是否足够覆盖，避免值域差异较大导致大数吃小数问题

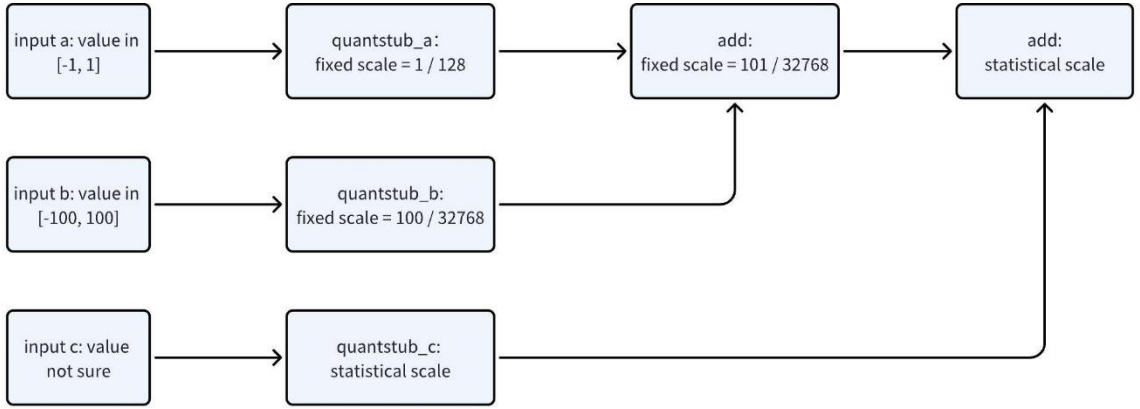
其他Tips

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

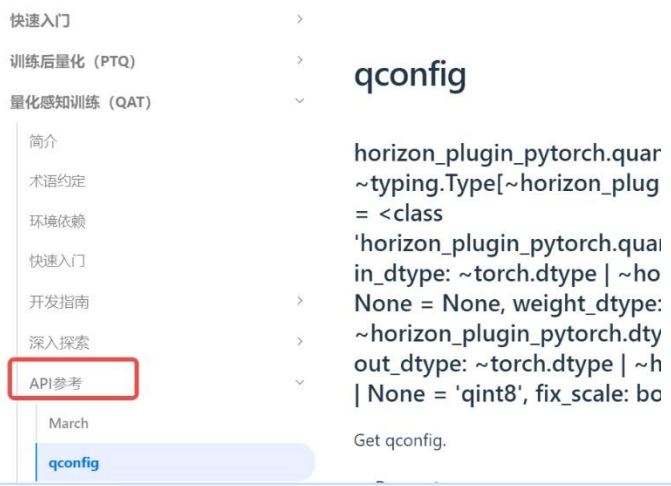
- 调整 weight decay。weight decay 会影响模型中权重的数值范围，更小的数值范围更加量化友好。有时，只调整 qat 阶段的 weight decay 还不足以解决问题，需要调整浮点训练阶段的 weight decay。

- 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。

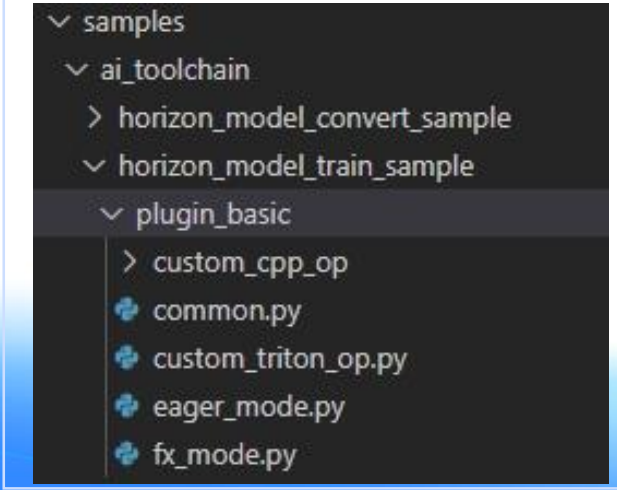
附录

用户手册

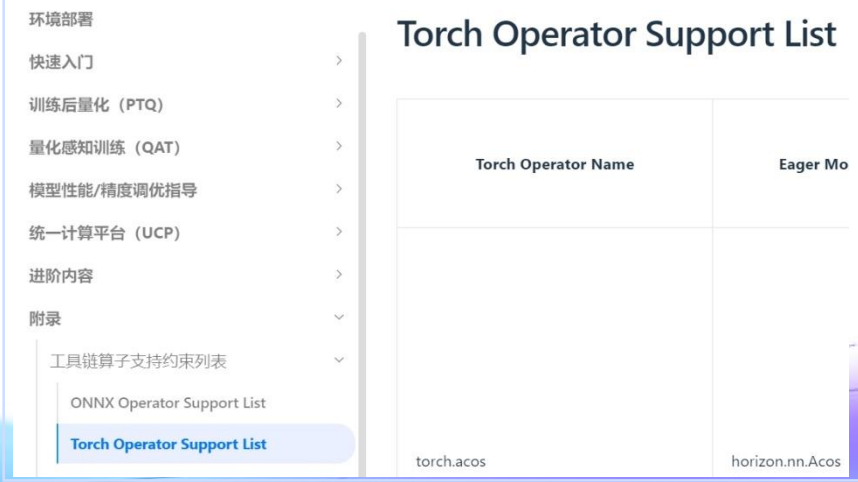
API：

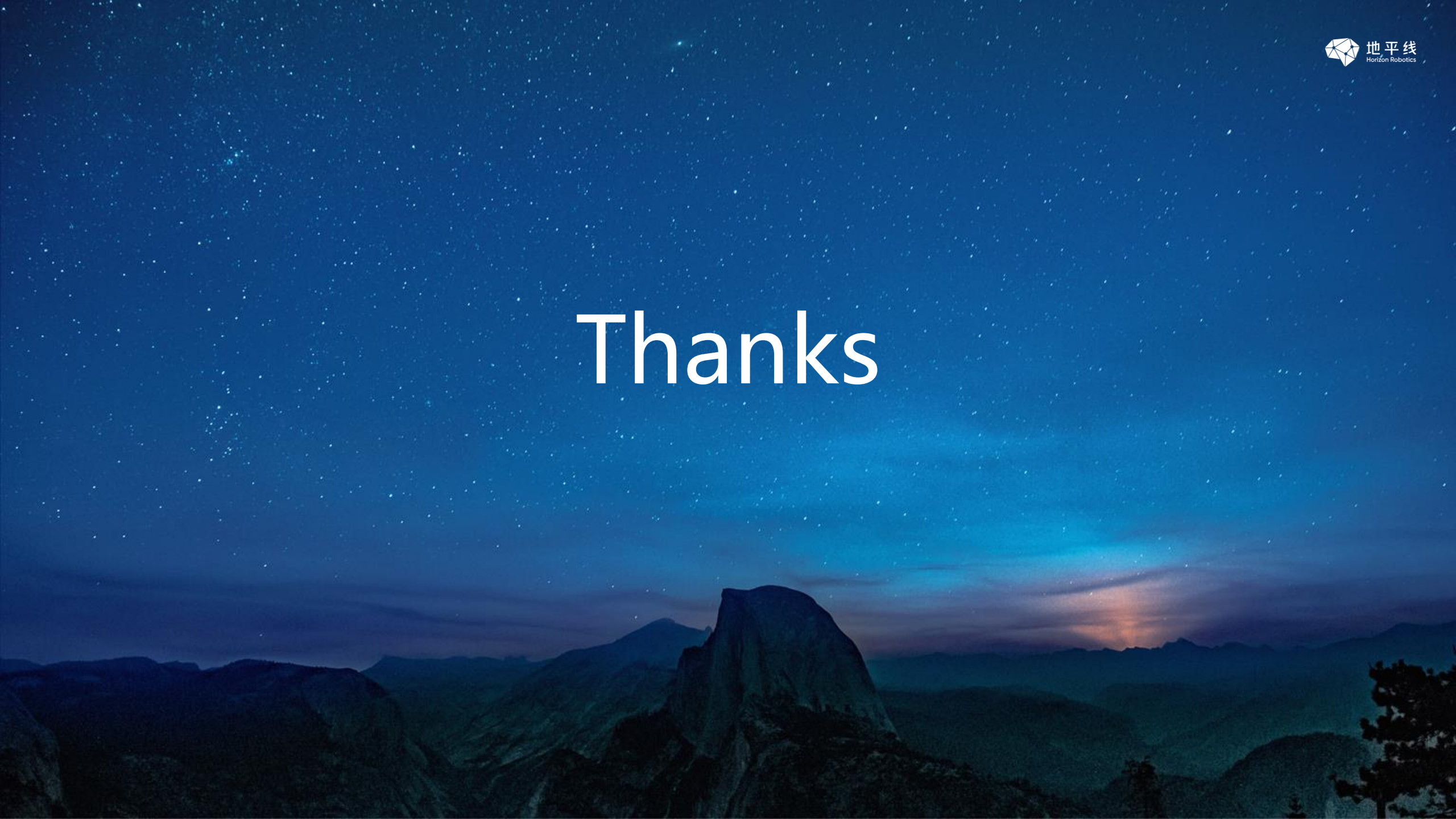


Plugin sample：



算子支持列表：




地平线
Horizon Robotics
Thanks