Witmem
知存科技

witin_nn V2.1.0 用户手册

修订日期: 2024 年 6 月 17 日

一、witin_nn 概述

1.1、背景介绍

由于知存科技存内计算方案的模拟噪声影响，单纯经过浮点训练的神经网络模型在部署到芯片后往往会出现性能下降，因此有必要引入噪声感知训练，使得神经网络在训练过程中感知到芯片的噪声特性，从而获得部署到芯片的更好性能。

witin_nn 框架是基于 PyTorch 开发的，witin_nn 框架主要实现了适配知存科技芯片的量化感知训练（QAT）和噪声感知训练（NAT）方法，目前支持 Linear、Conv2d、ConvTranspose2d、GruCell 等算子。本框架通过在神经网络的正向传播链路上引入输入、权重、偏置以及输出的噪声，干预神经网络的反向传播（参数更新），从而增强网络的泛化能力。具体来说，witin_nn 模拟神经网络映射到知存科技存内芯片计算的过程，支持输入和输出的 8bits~12bits 位宽量化以及权重的 8bits 量化，实现 QAT，并引入模拟电路噪声，实现 NAT。

从训练效果来看，如果以浮点训练的浮点软跑性能作为 baseline，通常在增加量化感知训练 (QAT)、噪声感知训练 (NAT) 之后，部署到芯片的性能会更加逼近 baseline。

1.2、计算详解

1. 如表 1 所示，展示了各 witin_nn 算子和 torch 算子的对应关系。

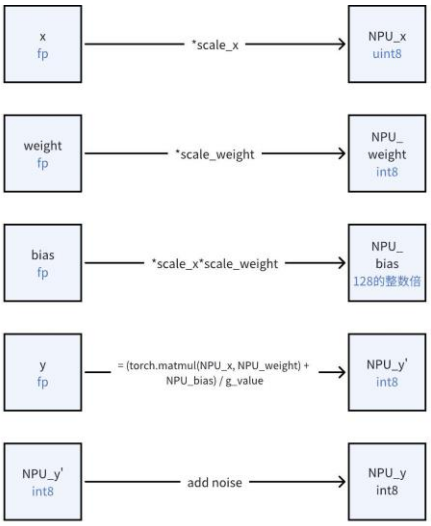
算子类型	**witin_nn 算子**	**对标 pytorch 算子**	**芯片计算公式**
存算	witin_nn.WitinLinear	torch.nn.Linear	output = torch.nn.functional.linear(input, weight, bias) / g_value
存算	witin_nn.WitinConv2d	torch.nn.Conv2d	output = torch.nn.functional.conv2d(input, weight, bias, stride, padding, dilation, groups) / g_value
存	witin_nn.WitinConvTra	torch.nn.ConvTra	output = torch.nn.functional.conv_transpose2d(input,

算	nspose2d	nspose2d	weight, bias, stride, padding, output_padding, groups, dilation) / g_value
存算	witin_nn.WitinGruCell	torch.nn.GRUCell	output = torch._VF.gru_cell(input, hx, weight_ih, weight_hh, bias_ih, bias_hh) / g_value
数字	witin_nn.WitinGELU	torch.nn.GELU	/
数字	witin_nn.WitinSigmoid	torch.nn.Sigmoid	/
数字	witin_nn.WitinTanh	torch.nn.Tanh	/
数字	witin_nn.WitinPReLU	torch.nn.PReLU	/
数字	witin_nn.WitinElementAdd	加法	/
数字	witin_nn.WitinElementDivide	除法	/
数字	witin_nn.WitinElementMul	乘法	/
数字	witin_nn.WitinSqrt	torch.sqrt	/
数字	witin_nn.WitinMean	torch.mean	/

数字	witin_nn.WitinCat	torch.cat	/
数字	witin_nn.WitinBatchNorm2d	torch.nn.BatchNorm2d	/

表 1

2. 下面以 witin_nn.WitinLinear 算子为例，简述 QAT 及 NAT 计算的过程（输入、输出均量化到 8bits）。



如上所示，输入 x 量化为 uint8 的 NPU_x，权重 weight 量化为 int8 的 NPU_weight，偏置 bias 量化为 128 的整数倍，即 NPU_bias，已知 NPU_x，NPU_weight，NPU_bias，可计算出 NPU_y'，其中引入模拟电路噪声，得到 NPU_y，最终量化为 int8。最终，witin_nn.WitinLinear 算子输出为 NPU_y/y_scale（反量化回到浮点域）。

3. 数学等价性分析：

$$y = \frac{(torch.matmul(x, weight) + bias)}{g\_value}$$

<==>

$$y * scale\_y = \frac{(torch.matmul(x * scale\_x, weight * scale\_weight) + (bias * scale\_x * scale\_weight))}{scale\_x * scale\_weight / scale\_y}$$

<==>

$$NPU\_y = \frac{(torch.matmul(NPU\_x, NPU\_weight) + NPU\_bias)}{scale\_x * scale\_weight / scale\_y}$$

其中：

$$NPU\_y = y * scale\_y$$

$$NPU\_x = x * scale\_x$$

$$NPU\_weight = weight * scale\_weight$$

$$NPU\_bias = bias * scale\_x * scale\_weight$$

$$g\_value = scale\_x * scale\_weight / scale\_y$$

二、开发指导

2.1 环境准备

python >= 3.7

torch == 1.13

2.2 算子参数说明

witin_nn 算子是对 torch.nn 对应算子的再次封装，witin_nn 算子保留了 torch.nn 对应算子的所有参数，在 torch.nn 参数列表基础上扩展了 QAT 及 NAT 相关参数。在构建神经网络时，需要将 torch 算子替换为对应的 witin_nn 算子，并为其配置相应参数即可。

保留参数可以参考 pytorch 官方文档，witin_nn 所有算子都包含以下扩展参数，但不是所有参数都可以生效，释义如下：

参数	类型	默认值	含义	适用算子
target_platform	**class TargetPlatform**(Enum):	TargetPlatform.WTM2101	区别不同芯片平台。	全部

	WTM2101 = 1			
hardware	**Class HardwareType** (Enum): ARRAY = 1 VPU = 2	HardwareType.ARRAY	区别不同计算平台。	全部
w_clip	float 或者 None	None	当 w_clip = None 时，将不会对权重做任何操作；反之则会将 weight 限制在 -w_clip~w_clip 之间。	全部存算算子
bias_row_N	int	8	bias 计算所用的 NPU array 行数，仅当 use_quantization = True 时有效。	全部存算算子
use_quantization	bool	False	use_quantization = True 进行量化感知训练。 use_quantization = False 进行浮点训练。	全部
noise_model	**class NoiseModel**(Enum): NORMAL = 1 ARRMDL = 2 MBS = 3 SIMPLE =	NoiseModel.NORMAL	噪声模型类型，目前仅支持 NORMAL 类型噪声模型。	全部存算算子

	4			
noise_level	int	0	noise_level = 0 不加噪声。 0<=noise_level <10 对应 NORMAL 噪声模型的噪声等级，数字越大，噪声越强。	全部存算算子
to_linear	bool	False	是否将 Conv2d、Conv1d、ConvTranspose2d 算子等价替换为 linear 算子进行计算，训练中保持 to_linear = False 即可。	WitinLinear WitinConv2d WitinConvTranspose2d
use_auto_scale	bool	True	是否自动计算 scale_x，scale_y，scale_weight 。	全部
scale_x	int	1	仅当 use_quantization == True 时有效。	全部
scale_y	int	1	仅当 use_quantization == True 时有效。	全部
scale_weight	int	1	仅当 use_quantization == True 时有效。	全部有权重的算子
handle_neg_in	**class HandleNegInType**(Enum): FALSE = 1 #不对负输入做处理	HandleNegInType.FALSE	支持对负输入的处理，仅当 use_quantization == True 时有效，	WitinLinear WitinConv2d WitinConvTranspo

	PN = 2
	#输入符号变换至权重
	Shift = 3
	#对输入整体偏移
				se2d

shift_num	float	1	选择 HandleNegInType.Shift 时，需配置该偏移参数。	WitinLinear WitinConv2d WitinConvTranspose2d
x_quant_bits	int	8	输入量化位宽。	全部
y_quant_bits	int	8	输出量化位宽。	全部
weight_quant_bits	int	8	权重量化位宽。	全部有权重的算子
bias_d	torch.Tensor	torch.tensor(0)	拆出到数字计算的偏置。	WitinLinear WitinConv2d WitinConvTranspose2d
conv2	/	/	预留，暂不开放。	

d_split _N				

表 2 参数列表

2.3 配置文件说明

两种 config 类型:

- WitinGlobalConfig: 全局配置，所有算子的默认配置。
- WitinLayerConfig: 针对某个算子特定的传参设置。

interface/ConfigFactory.py 中定义了几种标准的配置方案。

2.4 使用示例

2.4.1 定义一个简单的 torch 神经网络

```
Python
class DnnNet(nn.Module):
    def __init__(self):
        self.linear1 = torch.nn.Linear(128,128, bias = False)

    def forward(self, _input):
        out = self.linear1(_input)
        return out
```

2.4.2 witin_nn 浮点训练示例

```
Python
class DnnNet(nn.Module):
    def __init__(self):
        config_linear1 = LayerConfigFactory.get_default_config()
        config_linear1.use_quantization = False
```

```
        self.linear1 = WitinLinear(128,128, bias = False,
layer_config=config_linear1)

    def forward(self, _input):
        out = self.linear1(_input)
        return out
```

2.4.3 witin_nn 量化训练示例

```
Python
class DnnNet(nn.Module):
    def __init__(self):
        config_linear1 = LayerConfigFactory.get_default_config()
        config_linear1.use_quantization = True

        config_linear1.x_quant_bit = 8     #输入量化位宽
        config_linear1.y_quant_bit = 8     #输出量化位宽

        config_linear1.scale_x = 16        #输入缩放参数
        config_linear1.scale_y = 16        #输出缩放参数
        config_linear1.scale_weight = 16   #权重缩放参数

        config_linear1.use_auto_scale = True   #是否开启自动计算量化参
数

        config_linear1.handle_neg_in = HandleNegInType.PN #当输入是
有符号数时，需要额外处理，有三种处理方式供选择

        self.linear1 = WitinLinear(128,128, bias = False,
layer_config=config_linear1)

    def forward(self, _input):
        out = self.linear1(_input)
        return out
```

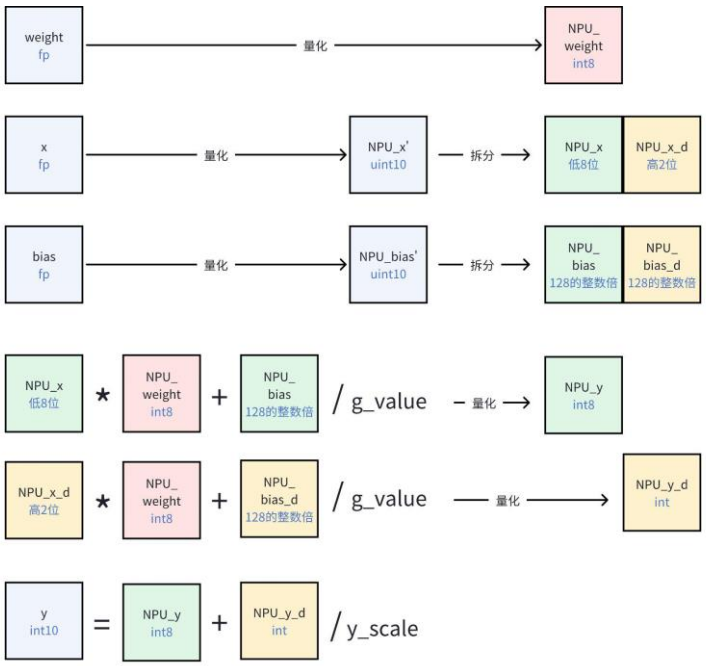
2.4.3 witin_nn 量化及加噪训练示例

```
Python
class DnnNet(nn.Module):
    def __init__(self):
        config_linear1 = LayerConfigFactory.get_default_config()
        config_linear1.use_quantization = True
        config_linear1.noise_level = 4

        config_linear1.x_quant_bit = 8     #输入量化位宽
        config_linear1.y_quant_bit = 8     #输出量化位宽

        config_linear1.scale_x = 16        #输入缩放参数
        config_linear1.scale_y = 16        #输出缩放参数
        config_linear1.scale_weight = 16   #权重缩放参数

        config_linear1.use_auto_scale = True   #是否开启自动计算量化参数

        config_linear1.handle_neg_in = HandleNegInType.PN #当输入是有符号数时，需要额外处理，有三种处理方式供选择

        self.linear1 = WitinLinear(128,128, bias = False, layer_config=config_linear1)

    def forward(self, _input):
        out = self.linear1(_input)
        return out
```

2.5 量化位宽大于 8bit 指导

存算核支持的是 8bits 数据计算，但是为了提高精度，希望量化后输入位宽大于 8bits。witin_nn 将模拟映射到芯片的拆分过程（即低 8 位用模拟计算，高位用数字计算）。需要注意的是，bias 也可能会涉及到拆分以保证映射后模拟计算的输出尽量不出现饱和，在此引入额外参数 bias_d（d 意为 digital）来表示拆出到数字计算的偏置。

下面以 witin_nn.WitinLinear 为例，以 10bits 输入、10bits 输出说明该过程。



如上图所示：

（1） 对输入 x 、权重 weight 分别量化为 uint10（0~1023）、int8（-128~127）的整型；对偏置 bias 量化为 128 的整数倍；

（2）将量化后的 x 拆分为低 8 位 NPU_x 和高 2 位 NPU_x_d、量化后的 bias 拆分为模拟计算部分的偏置 NPU_bias 和数字计算部分的偏置 NPU_bias_d；NPU_weight 为量化后的权重。

（3） 进行计算并得到模拟计算输出 NPU_y、数字计算输出 NPU_y_d；

（4）最终输出 y 先将 NPU_y 与 NPU_y_d 求和并量化为 int10，再除以 y_scale（反量化回到浮点域）。

2.6 auto-scale 策略理解

量化方式为对称量化，按照数据的 min-max 确定量化参数，对算子的输入，输出，权重（如果有）进行量化。

举例如下：量化一组数据，量化位宽为 int8，量化参数按如下方式确定：

```
Python
#量化位宽 int8
```

```
x_quant_bits = 8
x = torch.randn(1,10)
x_max = x.abs().max()
scale_x = 2 ** (x_quant_bits - 1) / 2 ** (torch.log2(x_max).ceil())

'''
x:        tensor([[ 0.1875, -1.3344,  0.5350,  1.5472, -0.9712,
-1.4459,  0.1024, -0.8054,
         -1.7309, -0.8548]])
x_max:    1.7309
scale_x:  128
'''
```

- 在模型训练阶段，配置 use_auto_scale = True，假定训练 M 个 epoch，每个 epoch 包含 N 个 iter。

（1）在训练启动时，会预先训练 n 个 iter，量化参数 data_scale 为用户设置的初始值 (scale_x, scale_weight, scale_y)。训练期间统计数据的绝对值的最大值 data_max，n 由用户自己配置，对应参数 auto_scale_updata_step。

（2）在训练 iter 超过 n 之后，根据 data_max 计算 data_scale，并更新 data_scale，后续的 N-n 个 iter 的训练都将使用该 data_scale。

（3）在下一个 epoch 开始后，重复（1）（2）步，data_max 重新统计，data_scale 重新计算。

（4）训练完成后，在保存的模型文件中，模型的每一层均包含参数 io_max，即该层的 data_max。

- 在模型推理阶段，配置 use_auto_scale = True

witin_nn 自动读取模型中的参数 io_max，并自动计算量化参数。

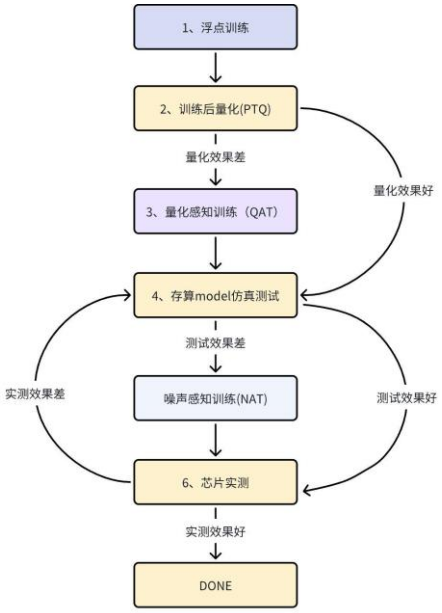
- 如果配置 use_auto_scale = False，量化参数固定，始终为用户配置的 scale_x, scale_weight, scale_y。
- 如果需要提取量化参数，首先要提取 io_max，再手动计算量化参数。
- **启用 auto-scale 时，需要特别注意量化参数初值的选择，过小或者过大会影响最终 scale 的确定。**

2.7 witin_nn 训练建议

下面将对如何应用本框架进行模型训练阐述。

建议在浮点训练模型的基础上逐步引入噪声或限制进行重新训练，所以建议训练顺序:

训练流程如下:



以上三部分的训练精度一般来说满足以下规则:

> step1: use_quantization = False 进行浮点训练（可能需要指定 w_clip 对权重进行限制从而得到比较适合芯片部署的预训练模型）。

在训练结束后，建议分别在测试集上测试三种条件下对应的损失函数值、模型评价指标（取决于具体任务，例如识别率、PSNR 等）。

- use_quantization = False 时，损失函数值记为 Lf1、模型评价指标记为 Pf1。
- use_quantization = True，指定 scale_x、scale_y、scale_weight，指定 bias_row_N (=8) 时，损失函数值记为 Lf2、模型评价指标记为 Pf2。
- use_quantization = True、use_noise = True，指定 scale_x、scale_y、scale_weight，指定 bias_row_N (=8)时，损失函数值记为 Lf3、模型评价指标记为 Pf3。

一般来说，Lf1<Lf2<Lf3、Pf1 好于 Pf2 好于 Pf3，具体差别多少反映了量化、加噪带来的影响。

> step2: use_quantization =True、指定 scale_x、scale_y、scale_weight、指定 bias_row_N (=8) ，加载 step1 浮点模型，进行 QAT 重训练（量化损失不大时可略过）。

在训练结束后，建议分别在测试集上测试三种条件下对应的损失函数值、模型评价指标（取决于具体任务，例如识别率、PSNR 等）。

- use_quantization = False 时，损失函数值记为 Lq1、模型评价指标记为 Pq1
- use_quantization = True、指定 scale_x、scale_y、scale_weight，指定 bias_row_N (=8) 时，损失函数值记为 Lq2、模型评价指标记为 Pq2
- use_quantization = True、use_noise = True，指定 scale_x、scale_y、scale_weight，指定 bias_row_N (=8)时，损失函数值记为 Lq3、模型评价指标记为 Pq3。

量化感知训练之后，我们希望 Lf1≈Lq2<Lq3、Pf1≈Pq2 好于 Pq3，实际情况中，具体问题具体分析。

> step3: use_quantization =True、use_noise = True，指定 scale_x、scale_y、scale_weight、指定 bias_row_N (=8) ，进行 =NAT 重训练。

在训练结束后，建议分别在测试集上测试三种条件下对应的损失函数值、模型评价指标（取决于具体任务，例如识别率、PSNR 等）。

- use_quantization = False 时，损失函数值记为 Ln1、模型评价指标记为 Pn1
- use_quantization = True、指定 scale_x、scale_y、scale_weight，指定 bias_row_N (=8) 时，损失函数值记为 Ln2、模型评价指标记为 Pn2
- use_quantization = True、use_noise = True，指定 scale_x、scale_y、scale_weight，指定 bias_row_N (=8)时，损失函数值记为 Ln3、模型评价指标记为 Pn3。

噪声感知训练之后，我们希望 Lf1≈Lq2≈Ln2≈Ln3、Pf1≈Pq2≈Pn2≈Pn3，实际情况中，具体问题具体分析。