Witin_NN V2.1.0

G u i l d l i n e

一、witin_nn

1.1、Introduction

Due to the impact of analog noise in the in-memory computing solution of Zhicun Technology, neural network models that have only been trained with floating points often experience a decline in performance when deployed on chips. Therefore, it is necessary to introduce noise-aware training, which allows the neural network to perceive the noise characteristics of the chip during the training process, thereby achieving better performance when deployed on the chip.

The "witin_nn" framework is developed based on PyTorch and mainly implements the quantization-aware training (QAT) and noise-aware training (NAT) methods adapted to Zhicun Technology's chips. It currently supports operators such as Linear, Conv2d, ConvTranspose2d, and GruCell. This framework introduces noise to the input, weights, biases, and output in the forward propagation chain of the neural network, intervening in the backward propagation (parameter update) of the neural network, thereby enhancing the network's generalization ability. Specifically, "witin_nn" simulates the process of mapping neural networks to the in-memory chip computation of Zhicun Technology, supporting 8-bit to 12-bit quantization for inputs and outputs and 8-bit quantization for weights, to achieve QAT, and introduces analog circuit noise to achieve NAT.

In terms of training effects, if the performance of the floating-point software running after floating-point training is taken as the baseline, the performance deployed on the chip will usually be closer to the baseline after adding quantization-aware training (QAT) and noise-aware training (NAT).

1.2、In detail

1. As table 1, show the realtionship of witin_nn operator and torch

Type	witin_nn	Pytorch	Computational formula for chip
CIM	witin_nn.WitinLinear	torch.nn.Linear	output = torch.nn.functional.linear(input, weight, bias) / g_value
CIM	witin_nn.WitinConv2d	torch.nn.Conv2d	output = torch.nn.functional.conv2d(input, weight, bias, stride, padding, dilation, groups) / g_value
CIM	witin_nn.WitinConvTranspose2d	torch.nn.ConvTranspose2d	output = torch.nn.functional.conv_transpose2d(input, weight, bias, stride, padding, output_padding, groups, dilation) / g_value

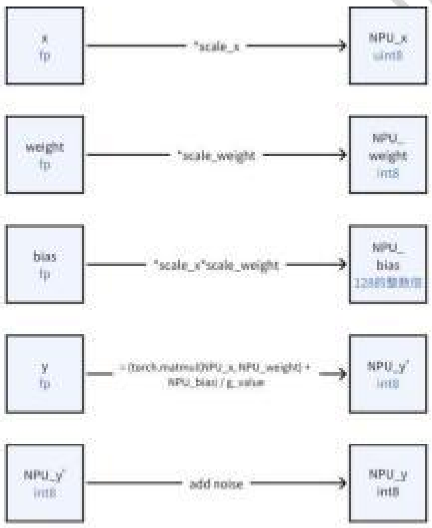
CIM	witin_nn.WitinGruCell	torch.nn.GRUCell	output = torch._VF.gru_cell(input, hx, weight_ih, weight_hh, bias_ih, bias_hh) / g_value
Digital	witin_nn.WitinGELU	torch.nn.GELU	/
Digital	witin_nn.WitinSigmoid	torch.nn.Sigmoid	/
Digital	witin_nn.WitinTanh	torch.nn.Tanh	/
Digital	witin_nn.WitinPReLU	torch.nn.PReLU	/
Digital	witin_nn.WitinElement Add	add	/
Digital	witin_nn.WitinElement Divide	Divison	/
Digital	witin_nn.WitinElement Mul	mix	/

Digital	witin_nn.WitinSqrt	torch.sqrt	/
Digital	witin_nn.WitinMean	torch.mean	/

digital	witin_nn.WitinCat	torch.cat	/
Digital	witin_nn.WitinBatchNorm2d	torch.nn.BatchNorm2d	/

Table 1

2. Below, taking the witin_nn.WitinLinear operator as an example, we briefly describe the process of QAT (Quantization Aware Training) and NAT (Noise Aware Training) computations (where both inputs and outputs are quantized to 8 bits).



As shown above, the input x is quantized to a uint8 NPU_x, the weight weight is quantized to an int8 NPU_weight, and the bias bias is quantized to an integer multiple of 128, i.e., NPU_bias. Given NPU_x, NPU_weight, and NPU_bias, one can calculate the preliminary output NPU_y'. Introducing simulated circuit noise to NPU_y' yields NPU_y, which is then quantized to an int8. Ultimately, the witin_nn.WitinLinear operator's output is NPU_y de-quantized back to the floating-point domain, represented as NPU_y/y_scale.

3. Mathematical Equivalence Analysis:

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

<==>

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

<==>

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

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、Operate Guildline

2.1 Arounding Prepare

python >= 3.7
torch == 1.13

2.2 Operators Parameters Guildline

The "witin_nn" operators are re-encapsulated versions of the corresponding "torch.nn" operators. These "witin_nn" operators retain all parameters of the "torch.nn" operators and extend them with additional parameters related to QAT (Quantization Aware Training) and NAT (Noise Aware Training) on top of the "torch.nn" parameter list. When constructing a neural network, you simply need to replace the "torch" operators with the corresponding "witin_nn" operators and configure the appropriate parameters.
The preserved parameters can be referenced from the official PyTorch documentation. All "witin_nn" operators include the following extended parameters, although not all parameters may take effect. The descriptions are as follows:

Paremeters	Type	Sign	Means	Ava operators
target_ platform	**class TargetPlatform**(Enum): WTM2101 = 1	TargetPlatform. WTM2101	**Differ to different operators platform**	All
hardware	**Class HardwareTyp**(Enum): ARRAY = 1 VPU = 2	HardwareType.ARRAY		
w_clip	float or None	None	**When** w_clip = None, no action to weight,or its true; weight limit from -w_clip to w_clip	All CIM operator

bias_ row_N	int	8	The number of rows of the NPU array used for bias calculation is only effective when use_quantization is set to True	All CIM operator
use_ quantization	bool	False	use_quantization = True going QAT training use_quantization = False ,going floating training。	All
noise_ model	**class** **NoiseModel** (E num): NORMAL=1 ARRMDL = 2 MBS = 3 SIMPLE=	NoiseModel.NORMAL	Noise model type，now only support NORMAL noise model。	All CIM operator

noise_ level	int	0	noise_level = 0 No noise。 0<=noise_level <10 NORMAL equal to noise level， data more high, noise more high	All CIM Operators
to_line ar	bool	False	eep the original convolutional operators such as Conv2d, Conv1d, and ConvTranspose2d without replacing them with a linear operator, simply set the to_linear parameter to False.	WitinLine ar WitinCon v2d WitinCon vTranspo se2d
use_a uto_sc ale	bool	True	calculate automaticly scale_x, scale_y， scale_weight 。	all
scale_ x	int	1	Only available use_quantization == True	all
scale_ y	int	1	Only available use_quantization == True	all
scale_ weight	int	1	Only available use_quantization == True	All weight operators

handle _neg_i n	PN = 2 #Input symbol transformation to weights. Shift = 3 #Overall offset of the input	**class HandleNegInType**(Enum): FALSE = 1 #No solution for negative input	HandleNegInType.FALSE	Support negative input, if use_quantization == True , its available WitinLinear WitinConv2d WitinConvTranspose2d

shift_num	float	1	Choose HandleNegInType.Shift,The offset parameter needs to be configured.	WitinLinear WitinConv2d WitinConvTranspose2d
x_quant_bits	int	8	Input Quantization Bit Width	ALL
y_quant_bits	int	8	Output Quantization Bit Width	ALL
weight_quant_bits	int	8	Weight Quantization Bit Width	All weight operators
bias_d	torch.Tensor	torch.tensor(0)	Bias Extracted for Digital Computation。	WitinLinear WitinConv2d WitinConvTranspose2d

conv2d_split_N	/	/	Reserved, Not Open for Use at the Moment	

2.3 Configuration File Description

There are two types of configuration classes:

- WitinGlobalConfig: Global configuration, the default settings for all operators.

- WitinLayerConfig: Specific parameter settings for a particular operator. Several standard configuration schemes are defined in interface/ConfigFactory.py.

2.4 Usage Case

2.4.1 Definite a simple torch neural network

```
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 Floating Training Case

```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 QFA Training Case:

```
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      #Input Quantization Bit Width
        config_linear1.y_quant_bit = 8      #Output Quantization Bit Width

        config_linear1.scale_x = 16         #Input Scaling Parameter
        config_linear1.scale_y = 16         #Output Scaling Parameter
        config_linear1.scale_weight = 16    #Weight Scaling Parameter


       #if open automaticly  quantizate parametres


        config_linear1.use_auto_scale = True



        config_linear1.handle_neg_in = HandleNegInType.PN #When the input is a signed number, additional processing is required, and there are three methods available for selection.
        self.linear1 = WitinLinear(128,128, bias = False, layer_config=config_linear1)

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

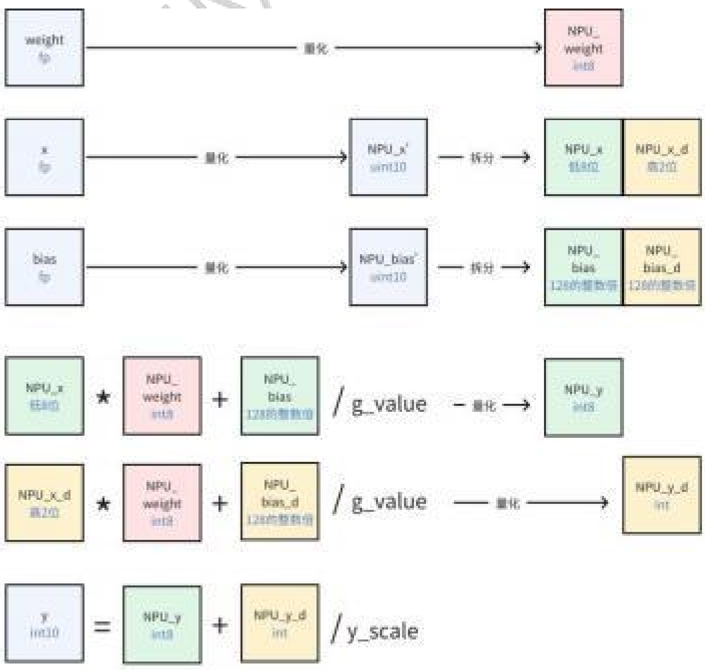
2.4.3 witin_nn 量化及加噪训练示例

```
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   #Input Quantization Bit Width
        config_linear1.y_quant_bit = 8    #Output Quantization Bit
                                              Width

        config_linear1.scale_x = 16          #Input Scaling Parameter
        config_linear1.scale_y = 16         #Output Scaling Parameter
        config_linear1.scale_weight = 16    #Weight Scaling Parameter

    config_linear1.use_auto_scale = True
                                 #if open automaticly   quantizate
                                 parametres



        config_linear1.handle_neg_in = HandleNegInType.PN #When the input
is a signed number, additional processing is required, and there are three methods
available for selection.


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


    def forward(self, _input):

        out  =  self.linear1(_input)  return
        out
```

2.5 Guidance on quantization bit-widths greater than 8 bits

For guidance on quantization bit-widths greater than 8 bits, the storage and computation core supports 8-bit data computation. However, to improve accuracy, there is a desire for the quantized input bit-width to be greater than 8 bits. The witin_nn simulates the mapping process to the chip, which involves splitting the process (i.e., the lower 8 bits are computed using analog computation, and the higher bits are computed digitally). It should be noted that the bias may also be involved in the splitting to ensure that the output of the analog computation after mapping does not saturate, introducing an additional parameter bias_d (where d stands for digital) to represent the bias extracted for digital computation.

Below is an example using witin_nn.WitinLinear with 10-bit input and 10-bit output to illustrate this process.



As shown in the figure above:

1,The input x and weight weight are quantized to uint10 (0 to 1023) and int8 (-128 to 127) integers, respectively; the bias bias is quantized to an integer multiple of 128.
2,The quantized x is split into the lower 8 bits NPU_x and the higher 2 bits NPU_x_d; the quantized bias is split into the analog computation part of the bias NPU_bias and the digital computation part of the bias NPU_bias_d.
NPU_weight is the quantized weight.
3,Computations are carried out to obtain the analog computation output NPU_y and the digital computation output NPU_y_d.
4,The final output y is obtained by summing NPU_y and NPU_y_d, quantizing to int10, and then dividing by y_scale (de-quantizing back to the floating-point domain).

2.6 Understanding the auto-scale strategy:

The quantization method is symmetric quantization, which determines the quantization parameters based on the min-max of the data, and quantizes the operator's input, output, and weight (if any). Here is an example: Quantize a set of data with a quantization bit-width of int8. The quantization parameters are determined as follows:。

```
Python
#The width of quantization for  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
'''
```

•During the model training phase, set use_auto_scale = True, assuming that the training consists of M epochs, each containing N iterations.

1,At the start of training, n iterations are pre-trained, and the quantization parameter data_scale is set to the initial value set by the user (scale_x, scale_weight, scale_y). During training, the maximum absolute value of the data data_max is statistically analyzed; n is configured by the user and corresponds to the parameter auto_scale_update_step.

2,After the training iterations exceed n, data_scale is calculated based on data_max and updated. The subsequent N-n iterations of training will use this data_scale.

3,At the beginning of the next epoch, steps (1) and (2) are repeated, data_max is re-calculated, and data_scale is re-calculated.

4,After training is completed, in the saved model file, each layer of the model contains the parameter io_max, which is the data_max of that layer.

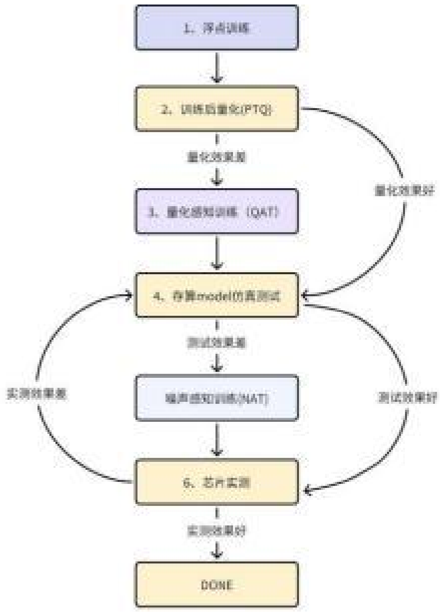
- During the model inference phase, set use_auto_scale = True. witin_nn automatically reads the model's io_max parameters and automatically calculates the quantization parameters.
- If use_auto_scale is set to False, the quantization parameters are fixed and always the user-configured scale_x, scale_weight, scale_y.
- If you need to extract the quantization parameters, first extract io_max and then manually calculate the quantization parameters.
- When enabling auto-scale, special attention should be paid to the selection of the initial value of the quantization parameters; too small or too large will affect the final determination of the scale.

2.7 witin_nn Training Recommendations

The following outlines how to apply this framework for model training.

so the suggested training order is:

The training process is as follows:



The training accuracies of the above three parts generally satisfy the following rules:

step1: use_quantization = False for floating-point training (it may be necessary to specify w_clip to restrict the weights to obtain a pre-trained model more suitable for chip deployment). After training, it is recommended to test the loss function values and model evaluation metrics on the test set under three conditions (depending on the specific task, such as recognition rate, PSNR, etc.).

- When use_quantization = False, the loss function value is recorded as Lf1, and the model evaluation metric is recorded as Pf1.
- use_quantization = True, specify scale_x, scale_y, scale_weight, specify bias_row_N (=8), the loss function value is recorded as Lf2, and the model evaluation metric is recorded as Pf2.
- use_quantization = True, use_noise = True, specify scale_x, scale_y, scale_weight, specify bias_row_N (=8), the loss function value is recorded as Lf3, and the model evaluation metric is recorded as Pf3. Generally, Lf1 < Lf2 < Lf3, Pf1 is better than Pf2 which is better than Pf3. The specific difference reflects the impact of quantization and noise addition.

Step 2: use_quantization = True, specify scale_x, scale_y, scale_weight, specify bias_row_N (=8), load the floating-point model from Step 1, and retrain with QAT (if the quantization loss is not significant, this step can be skipped). After training, it is recommended to test the loss function values and model evaluation metrics on the test set under three conditions (depending on the specific task, such as recognition rate, PSNR, etc.).

- When use_quantization = False, the loss function value is recorded as Lq1, and the model evaluation metric is recorded as Pq1.
- use_quantization = True, specify scale_x, scale_y, scale_weight, specify bias_row_N (=8), the loss function value is recorded as Lq2, and the model evaluation metric is recorded as Pq2.
- use_quantization = True, use_noise = True, specify scale_x, scale_y, scale_weight, specify bias_row_N (=8), the loss function value is recorded as Lq3, and the model evaluation metric is recorded as Pq3. After quantization-aware training, we hope Lf1 ≈ Lq2 < Lq3, Pf1 ≈ Pq2 is better than Pq3. In practical situations, specific issues should be analyzed specifically

Step 3: use_quantization = True, use_noise = True, specify scale_x, scale_y, scale_weight, specify bias_row_N (=8), retrain with NAT. After training, it is recommended to test the loss function values and model evaluation metrics on the test set under three conditions (depending on the specific task, such as recognition rate, PSNR, etc.).

- When use_quantization = False, the loss function value is recorded as Ln1, and the model evaluation metric is recorded as Pn1.
- use_quantization = True, specify scale_x, scale_y, scale_weight, specify bias_row_N (=8), the loss function value is recorded as Ln2, and the model evaluation metric is recorded as Pn2.
- use_quantization = True, use_noise = True, specify scale_x, scale_y, scale_weight, specify bias_row_N (=8), the loss function value is recorded as Ln3, and the model evaluation metric is recorded as Pn3. After noise-aware training, we hope Lf1 ≈ Lq2 ≈ Ln2 ≈ Ln3, Pf1 ≈ Pq2 ≈ Pn2 ≈ Pn3. In practical situations, specific issues should be analyzed specifically.