Vitis AI User Guide (UG1414)

PyTorch Version (vai_q_pytorch)

Installing vai_q_pytorch
Inspect Float Model Before Quantization
Running vai_q_pytorch
Hardware-Aware Quantization Strategy
Module Partial Quantization
Register Custom Operation
vai_q_pytorch Fast Finetuning
Configuration of Quantization Strategy
vai_q_pytorch QAT
vai_q_pytorch Usage
vai_q_pytorch message

PyTorch Version (vai_q_pytorch)

Installing vai_q_pytorch

vai_q_pytorch has GPU and CPU versions. It supports PyTorch version 1.2~1.12but does not support PyTorch data parallelism. There are two ways to install vai_q_pytorch:

Install Using Docker Containers

The Vitis AI provides a Docker container for quantization tools, including vai_q_pytorch. After running a GPU/CPU container, activate the Conda environment, vitis-ai-pytorch.

```
conda activate vitis-ai-pytorch
```

✎ **Note:**In some cases, if you want to install some packages in the Conda environment and encounter permission problems, you can create a separate Conda environment based on `vitis-ai-pytorch` instead of using `vitis-ai-pytorch` directly. The pt_pointpillars_kitti_12000_100_10.8G_1.3 model in Xilinx Model Zoo is an example of this.

A new Conda environment with a specified PyTorch version (1.2~1.12) can be created using the https://github.com/Xilinx/Vitis-AI/blob/v3.0/docker/common/replace_pytorch.sh script. This script clones a Conda environment from vitis-ai-pytorch, uninstalls the original PyTorch, Torchvision and vai_q_pytorch packages, and then installs the specified version of PyTorch, Torchvision, and re-installs vai_q_pytorch from source code. The following is the command line to create a new Conda environment with the script:

```
replace_pytorch.sh new_conda_env_name
```

✎ **Note:**Before running the script, you must check the version of Python, PyTorch, and cuda-toolkit version in the replace_pytorch.sh script and edit them according to your requirement. When choosing PyTorch version and editing the command line, it needs to follow the instructions on pytorch official webpage.

Install from the Source Code

vai_q_pytorch is a Python package designed to work as a PyTorch plugin. It is an open source in Vitis_AI_Quantizer. It is recommended to install vai_q_pytorch in the Conda environment. To do so, follow these steps:

1. Add the CUDA_HOME environment variable in .bashrc.
   For the GPU version, if the CUDA library is installed in /usr/local/cuda, add the following line into .bashrc. If CUDA is in other directory, change the line accordingly.

   ```
   export CUDA_HOME=/usr/local/cuda
   ```

   For the CPU version, remove all CUDA_HOME environment variable setting in your .bashrc. It is recommended to cleanup it in command line of a shell window by running the following command:

   ```
   unset CUDA_HOME
   ```

2. Install PyTorch (1.2~1.12) and Torchvision.
   The following code takes PyTorch 1.7.1 and torchvision 0.8.2 as an example. You can find detailed instructions for other versions on the PyTorch website.

   ```
   pip install torch==1.7.1 torchvision==0.8.2
   ```

3. Install other dependencies.

   ```
   pip install -r requirements.txt
   ```

4. Install vai_q_pytorch.

   ```
   cd ./pytorch_binding
   python setup.py install
   ```

5. Verify the installation.

   ```
   python -c "import pytorch_nndct"
   ```

✎ **Note:**If the installed PyTorch version is 1.4 or higher, import pytorch_nndct before importing torch in your script. This is caused by a PyTorch bug in versions prior to 1.4. Refer to PyTorch GitHub issue 28536 and 19668 for details.

```
import pytorch_nndct
import torch
```

Inspect Float Model Before Quantization

Vai_q_pytorch provides a function called inspector to help you diagnose neural network (NN) models under different device architectures. The inspector can predict target device assignments based on hardware constraints. The generated inspection report can be used to guide users to modify or optimize the NN model, greatly reducing the difficulty and time of deployment. It is recommended to inspect float models before quantization.
Take resnet18_quant.py to demonstrate how to edit model code and apply this feature:

1. Import vai_q_pytorch module

    ```
    from pytorch_nndct.apis import Inspector
    ```

2. Create a inspector with target name or fingerprint

    ```
    inspector = Inspector("0x603000b16013831") # by target
    fingerprint
    or
    inspector = Inspector("DPUCAHX8L_ISA0_SP") # by target
    name
    ```

3. Inspect float model

    ```
    input = torch.randn([batch_size, 3, 224, 224])
    inspector.inspect(model, input)
    ```

Run the following command line to inspect model:

```
python resnet18_quant.py --quant_mode float --inspect
```

Inspector will display some special messages on screen with special color and special keyword prefix "VAIQ_*" according to the verbose_level setting. Note the

messages displayed between "[VAIQ_NOTE]: =>Start to inspect model..." and "[VAIQ_NOTE]: =>Finish inspecting."
If the inspector runs successfully, three important files are usually generated under the output directory "./quantize_result".

```
inspect_{target}.txt: Target information and all the details
of operations in float model
inspect_{target}.svg: If image_format is not None. A
visualization of inspection result is generated
inspect_{target}.gv: If image_format is not None. Dot source
code of inspetion result is generated
```

✎ **Note:**

- The inspector relies on 'xcompiler' package. In conda env vitis-ai-pytorch in Vitis-AI docker, xcompiler is ready. But if vai_q_pytorch is installed by source code, it needs to install xcompiler in advance.
- Visualization of inspection results relies on the dot engine. If you don't install dot successfully, set 'image_format = None' when inspecting.
- If you need more detailed guidance, you can refer to example/jupyter_notebook/inspector/inspector_tutorial.ipynb. Install jupyter notebook in advance. Run the following command:

  ```
  jupyter notebook
  example/jupyter_notebook/inspector/inspector_tutorial.ipy
  nb
  ```

Running vai_q_pytorch

vai_q_pytorch is designed to work as a PyTorch plugin. Xilinx provides the simplest APIs to introduce the FPGA-friendly quantization feature. For a well-defined model, you only need to add a few lines to get a quantize model object. To do so, follow these steps:

Preparing Files for vai_q_pytorch

Prepare the following files for vai_q_pytorch.

Table: Input Files for vai_q_pytorch

No.	Name	Description
1	model.pth	Pre-trained PyTorch model, generally pth file.
2	model.py	A Python script including float model definition.
3	calibration dataset	A subset of the training dataset containing 100 to 1000 images.

Modifying the Model Definition

To make a PyTorch model quantizable, it is necessary to modify the model definition to make sure the modified model meets the following conditions. An example is available in Vitis AI GitHub.

1. The model to be quantized should include forward method only. All other functions should be moved outside or move to a derived class. These functions usually work as pre-processing and post-processing. If they are not moved outside, the API removes them in the quantized module, which causes unexpected behavior when forwarding the quantized module.
2. The float model should pass the jit trace test. Set the float module to evaluation status, then use the `torch.jit.trace` function to test the float model. For more details, please refer to example/jupyter_notebook/jit_trace_test/jit_trace_test.ipynb.
3. The most common operators in pytorch are supported in vai_q_pytorch. For more information, go to doc/support_op.md.

Adding vai_q_pytorch APIs to Float Scripts

If there is a trained float model and some Python scripts to evaluate accuracy/mAP of the model before quantization, the Quantizer API replaces the float module with a quantized module. The normal evaluate function encourages quantized module forwarding. Quantize calibration determines quantization steps of tensors in evaluation process if flag quant_mode is set to "calib". After calibration, evaluate the quantized model by setting quant_mode to "test".

1. Import the vai_q_pytorch module.

```
from pytorch_nndct.apis import torch_quantizer,
dump_xmodel
```

2. Generate a quantizer with quantization needed input and get the converted model.

```
input = torch.randn([batch_size, 3, 224, 224])
quantizer = torch_quantizer(quant_mode, model, (input))
quant_model = quantizer.quant_model
```

3. Forward a neural network with the converted model.

```
acc1_gen, acc5_gen, loss_gen = evaluate(quant_model,
val_loader, loss_fn)
```

4. Output the quantization result and deploy the model.

```
if quant_mode == 'calib':
    quantizer.export_quant_config()
if deploy:

    quantizer.export_torch_script()
    quantizer.export_onnx_model()
    quantizer.export_xmodel(deploy_check=False)
```

Running Quantization and Getting the Result

✎ **Note:**vai_q_pytorch log messages have special colors and a special keyword prefix, "VAI_Q_*.". vai_q_pytorch log message types include "error", "warning", and "note." Pay attention to vai_q_pytorch log messages to check the flow status.

---

1. Run command with "--quant_mode calib" to quantize model.

```
python resnet18_quant.py --quant_mode calib --subset_len
200
```

   When calibrating forward, borrow the float evaluation flow to minimize code change from float script. If you encounter loss and accuracy messages displayed in the end, you can ignore them.
   It is important to control iteration numbers during quantization and evaluation. Generally, 100-1000 images are enough for quantization and the whole validation set is required for evaluation. The iteration numbers can be controlled in the data loading part. In this case, the `subset_len` argument controls the number of images that are used for network forwarding. If the float

evaluation script does not have an argument with a similar role, you must add one.
If this quantization command runs successfully, two important files are generated in the output directory ./quantize_result.

ResNet.py
: Converted vai_q_pytorch format model.

Quant_info.json
: Quantization steps of tensors. Retain this file for evaluating quantized models.

2. To evaluate the quantized model, run the following command:

```
python resnet18_quant.py --quant_mode test
```

   The accuracy displayed after the command has executed successfully is the accuracy for the quantized model.
3. To generate the XMODEL for compilation (and ONNX format quantized model) , the batch size should be 1. Set `subset_len=1` to avoid redundant iterations and run the following command:

```
python resnet18_quant.py --quant_mode test --subset_len 1
--batch_size=1 --deploy
```

   Skip loss and accuracy displayed in the log during running. The xmodel file for the Vitis AI compiler is generated in the output directory, ./quantize_result. It is further used to deploy to the FPGA.

```
ResNet_int.xmodel: deployed XIR format model
ResNet_int.onnx:   deployed onnx format model
ResNet_int.pt:     deployed torch script format model
```

✎ **Note:**XIR is ready in "vitis-ai-pytorch" conda environment in the Vitis AI docker but if vai_q_pytorch is installed from the source code, you have to install XIR in advance. If XIR is not installed, the xmodel file cannot be generated and the command will return an error. However, you can still check the accuracy in the output log.

Hardware-Aware Quantization Strategy

Inspector provides device assignments to operators in the neural network based on the target device. vai_q_pytorch can use the power of inspector to perform hardware-aware quantization.
Example code in example/resnet18_quant.py:

```
quantizer = torch_quantizer(quant_mode=quant_mode,
                            module=model,
                            input_args=(input),
                            device=device,
                            quant_config_file=config_file,
                            target=target)
```

For example/resnet18_quant.py, command line to do hardware-aware calibration:

```
python resnet18_quant.py --quant_mode calib --target
DPUCAHX8L_ISA0_SP
```

command line to test hardware-aware quantized model accuracy:

```
python resnet18_quant.py --quant_mode test --target
DPUCAHX8L_ISA0_SP
```

command line to deploy quantized model:

```
python resnet18_quant.py --quant_mode test --target
DPUCAHX8L_ISA0_SP --subset_len 1 --batch_size 1 --deploy
```

Module Partial Quantization

You can use module partial quantization if not all the sub-modules in a model need to be quantized. Besides using general vai_q_pytorch APIs, the `QuantStub/DeQuantStub` operator pair can be used to realize it. The following example demonstrates how to quantize `subm0` and `subm2`, but not quantize `subm1`.

```
from pytorch_nndct.nn import QuantStub, DeQuantStub
```

```
class WholeModule(torch.nn.module):
    def __init__(self,...):
        self.subm0 = ...
        self.subm1 = ...
        self.subm2 = ...

        # define QuantStub/DeQuantStub submodules
        self.quant = QuantStub()
        self.dequant = DeQuantStub()

    def forward(self, input):
        input = self.quant(input) # begin of part to be
quantized
        output0 = self.subm0(input)
        output0 = self.dequant(output0) # end of part to be
quantized

        output1 = self.subm1(output0)

        output1 = self.quant(output1) # begin of part to be
quantized
        output2 = self.subm2(output1)
        output2 = self.dequant(output2) # end of part to be
quantized
```

Register Custom Operation

In order to convert a quantized model to an xmodel，vai_q_pytorch provides a decorator to register an operation or a group of operations as a custom operation which is unknown for XIR.

```
# Decorator API
def register_custom_op(op_type: str, attrs_list:
Optional[List[str]] = None):
  """The decorator is used to register the function as a
custom operation.
  Args:
  op_type(str) - the operator type registered into quantizer.
  The type should not conflict with pytorch_nndct

  attrs_list(Optional[List[str]], optional) -
```

```
  the name list of attributes that define operation flavor.
  For example, Convolution operation has such attributes as
padding, dilation, stride and groups.
  The order of name in attrs_list should be consistent with
that of the arguments list.
  Default: None

  """
```

Perform the following steps:

1. Aggregate some operations as a function. The first argument name of this function should be ctx. The meaning of ctx is the same as that in torch.autograd.Function
2. Decorate this function with the decorator described above.

```
from pytorch_nndct.utils import register_custom_op

@register_custom_op(op_type="MyOp", attrs_list=["scale_1",
"scale_2"])
def custom_op(ctx, x: torch.Tensor, y:torch.Tensor,
scale_1:float, scale_2:float) -> torch.Tensor:
  return scale_1 * x + scale_2 * y

class MyModule(torch.nn.Module):
  def __init__(self):
  ...

  def forward(self, x, y):
    return custom_op(x, y, scale_1=2.0, scale_2=1.0)
```

Limitations:

1. Loop operation is not allowed in a custom operation.
2. The number of return values for a custom operation can only be one.

vai_q_pytorch Fast Finetuning

Generally, there is a small accuracy loss after quantization, but for some networks such as MobileNets, the accuracy loss can be large. In this situation, first try fast finetune. If fast finetune still does not yield satisfactory results, QAT can be used to further improve the accuracy of the quantized models.

The AdaQuant algorithm [1] uses a small set of unlabeled data. It not only calibrates the activations but also finetunes the weights. The Vitis AI quantizer implements this algorithm and under the alias "fast finetuning". Though slightly slower, fast finetuning can achieve better performance than quantize calibration. Similar to QAT, each run of fast finetuning may produce a different result.

Fast finetuning does not train the model, and only needs a limited number of iterations. For classification models on the Imagenet dataset, 5120 images are enough in experiment. Data annotation information is not needed in fast finetuning flow, so data without annotation can be input and it still works fine. Fast finetuning only needs some modification based on the model evaluation script. There is no need to set up the optimizer for training. To use fast finetuning, a function for model forwarding iteration is needed and will be called during fast finetuning. Re-calibration with the original inference code is recommended.

You can find a complete example in the open source example.

```
# fast finetune model or load finetuned parameter before test
  if fast_finetune == True:
      ft_loader, _ = load_data(
          subset_len=5120,
          train=False,
          batch_size=batch_size,
          sample_method='random',
          data_dir=args.data_dir,
          model_name=model_name)
      if quant_mode == 'calib':
          quantizer.fast_finetune(evaluate, (quant_model,
ft_loader, loss_fn))
      elif quant_mode == 'test':
          quantizer.load_ft_param()
```

For parameter finetuning and re-calibration of this ResNet18 example, run the following command:

```
python resnet18_quant.py --quant_mode calib --fast_finetune
```

To test the finetuned quantized model accuracy, run the following command:

```
python resnet18_quant.py --quant_mode test --fast_finetune
```

To deploy the finetuned quantized model, run the following command:

```
python resnet18_quant.py --quant_mode test --fast_finetune --
subset_len 1 --batch_size 1 --deploy
```

✎ **Note:**

1. Itay Hubara et.al., Improving Post Training Neural Quantization: Layer-wise Calibration and Integer Programming, arXiv:2006.10518, 2020.

Configuration of Quantization Strategy

```
For multiple quantization strategy configurations, vai_q_pytorch supports quantization configuration
file in JSON format.
```

1. **Usage**

   In order to make the customized configuration take effect, you only need to pass the configuration file to torch_quantizer API.

   ```
   config_file = "./pytorch_quantize_config.json"
   quantizer = torch_quantizer(quant_mode=quant_mode,
                               module=model,
                               input_args=(input),
                               device=device,

   quant_config_file=config_file)
   ```

   There is example code in example/resnet18_quant.py, which could use the file example/pytorch_quantize_config.json as its configuration file. Run command with "--config_file pytorch_quantize_config.json" to quantize model.

   ```
   python resnet18_quant.py --quant_mode calib --config_file
   pytorch_quantize_config.json
   python resnet18_quant.py --quant_mode test --config_file
   pytorch_quantize_config.json
   ```

   In the example configuration file, the model configuration in "overall_quantizer_config" is set to entropy calibration method and per_tensor quantization.

```
"overall_quantize_config": {
  ...
  "method": "entropy",
  ...
  "per_channel": false,
  ...
},
```

And the configuration of weights in "tensor_quantize_config" is maxmin calibration method and per_tensor quantization, which means weights use different quantization method from model configuration.

```
"tensor_quantize_config": {
  ...
  "weights": {
    ...
    "method": "maxmin",
    ...
    "per_channel": false,
    ...
    }
```

Besides, there is one layer quantization configuration in "layer_quantize_config" list. The configuration is based on layer_type, and set torch.nn.Conv2d layer to per_channel quantization.

```
"layer_quantize_config": [
  {
    "layer_type": "torch.nn.Conv2d",
    ...
    "overall_quantize_config": {
      ...
      "per_channel": false,
```

2. **The configurations that can be set in the file:**

convert_relu6_to_relu

(Global quantizer setting) Whether to convert ReLU6 to ReLU. Options: True or False.

include_cle

(Global quantizer setting) Whether to use cross layer equalization. Options: True or False.

include_bias_corr

(Global quantizer setting) Whether to use bias correction. Options: True or False

target_device

(Global quantizer setting) Device to deploy quantized model, options: DPU, CPU, GPU

quantizable_data_type

(Global quantizer setting) tensor types to be quantized in model

bit_width

(Tensor quantization setting)Bit width used in quantization

method

(Tensor quantization setting)Method used to calibrate the quantization scale. Options: Maxmin, Percentile, Entropy, MSE, diffs.

round_mode

(Tensor quantization setting)Rounding method in quantization process. Options: half_even, half_up, half_down, std_round

symmetry

(Tensor quantization setting)Whether to use symmetric quantization. Options: True or False

per_channel

(Tensor quantization setting)Whether to use per_channel quantization. Options: True or False

signed

(Tensor quantization setting)Whether to use signed quantization. Options: True or False

narrow_range

(Tensor quantization setting)Whether to use symmetric integer range for signed quantization. Options: True or False

scale_type

(Tensor quantization setting)Scale type used in quantization process. Options: Float, poweroftwo

calib_statistic_method

(Tensor quantization setting)Method to choose one optimal quantization scale if got different scales using multiple batch data. Options: modal, max, mean, median

3. **Hierarchical Configuration**

   Quantization configuration is in hierarchical structure.

   - If configuration file is not provided in the torch_quantizer API, the default configuration will be used, which is adapted to DPU device and uses poweroftwo quantization method.
   - If configuration file is provided, model configuration, including global quantizer settings and global tensor quantization settings are required.
   - If only model configuration is provided in the configuration file, all tensors in the model will use the same configuration.
   - Layer configuration could be used to set some layers to specific configuration parameters.

   a. **Default Configurations**

      Details of default configuration are shown below.

```
"convert_relu6_to_relu": false,
"include_cle": true,
"include_bias_corr": true,
"target_device": "DPU",
"quantizable_data_type": [
  "input",
  "weights",
  "bias",
  "activation"],
"bit_width": 8,
"method": "diffs",
"round_mode": "std_round",
"symmetry": true,
"per_channel": false,
"signed": true,
"narrow_range": false,
"scale_type": "poweroftwo",
"calib_statistic_method": "modal"
```

   b. **Model Configurations**

      In the example configuration file "example/pytorch_quantize_config.json", the global quantizer settings are set under their respective keywords. And global quantization parameters must be set under the "overall_quantize_config" keyword. As shown below.

```
    "convert_relu6_to_relu": false,
    "include_cle": false,
    "keep_first_last_layer_accuracy": false,
    "keep_add_layer_accuracy": false,
    "include_bias_corr": false,
    "target_device": "CPU",
    "quantizable_data_type": [
      "input",
      "weights",
      "bias",
      "activation"],
  "overall_quantize_config": {
      "bit_width": 8,
      "method": "maxmin",
      "round_mode": "half_even",
      "symmetry": true,
      "per_channel": false,
      "signed": true,
      "narrow_range": false,
      "scale_type": "float",
      "calib_statistic_method": "max"
  }
```

Optionally, the quantization configuration of different tensors in the model can be set separately. And the configurations must be set in "tensor_quantize_config" keyword. And in the example configuration file, just change the quantization method of activation to "mse". The rest of the parameters are used the same as the global parameters.

```
  "tensor_quantize_config": {

      "activation": {

          "method": "mse",

      }
  }
```

c. **Layer Configurations**

Layer quantization configurations must be added in the "layer_quantize_config" list. And two parameter configuration methods, layer type and layer name, are supported. There are five notes to do layer configuration.

- Each individual layer configuration must be in dictionary format.
- In each layer configuration, the "quantizable_data_type" and "overall_quantize_config" parameter are required. And in "overall_quantize_config" parameter, all quantization parameters for this layer need to be included.
- If the setting is based on layer type, the “layer_name” parameter should be null.
- If the setting is based on layer name, the model needs to run the calibration process firstly, then pick the required layer name from the generated python file in quantized_result directory. Besides, the “layer_type” parameter should be null.
- Same as the model configuration, the quantization configuration of different tensors in the layer can be set separately. And they must be set in "tensor_quantize_config" keywords.

In the example configuration file, there are two layer configurations. One is based on layer type, the other is based on layer name. In the layer configuration based on layer type, torch.nn.Conv2d layer need to set to specific quantization parameters. And the "per_channel" parameter of weight is set to "true", "method" parameter of activation is set to "entropy".

```
{
  "layer_type": "torch.nn.Conv2d",
  "layer_name": null,
  "quantizable_data_type": [
    "weights",
    "bias",
    "activation"],
  "overall_quantize_config": {
    "bit_width": 8,
    "method": "maxmin",
    "round_mode": "half_even",
    "symmetry": true,
    "per_channel": false,
    "signed": true,
    "narrow_range": false,
```

```
        "scale_type": "float",
        "calib_statistic_method": "max"
      },
      "tensor_quantize_config": {
        "weights": {
          "per_channel": true
        },
        "activation": {
          "method": "entropy"
        }
      }
    }
```

In the layer configuration based on layer name, the layer named "ResNet::ResNet/Conv2d[conv1]/input.2" need to set to specific quantization parameters. And the round_mode of activation in this layer is set to "half_up".

```
    {
      "layer_type": null,
      "layer_name":
    "ResNet::ResNet/Conv2d[conv1]/input.2",
      "quantizable_data_type": [
        "weights",
        "bias",
        "activation"],
      "overall_quantize_config": {
        "bit_width": 8,
        "method": "maxmin",
        "round_mode": "half_even",
        "symmetry": true,
        "per_channel": false,
        "signed": true,
        "narrow_range": false,
        "scale_type": "float",
        "calib_statistic_method": "max"
      },
      "tensor_quantize_config": {
        "activation": {
          "round_mode": "half_up"
        }
```

```
    }
  }
```

The layer name "ResNet::ResNet/Conv2d[conv1]/input.2" is picked from generated file "quantize_result/ResNet.py" of example code "example/resnet18_quant.py".

- Run the example code with command "python resnet18_quant.py --subset_len 100". The quantize_result/ResNet.py file is generated.
- In the file, the name of first convolution layer is "ResNet::ResNet/Conv2d[conv1]/input.2".
- Copy the layer name to quantization configuration file if this layer is set to specific configuration.

```
import torch
import pytorch_nndct as py_nndct
class ResNet(torch.nn.Module):
  def __init__(self):
    super(ResNet, self).__init__()
    self.module_0 = py_nndct.nn.Input()
#ResNet::input_0
    self.module_1 = py_nndct.nn.Conv2d(in_channels=3,
out_channels=64, kernel_size=[7, 7], stride=[2, 2],
padding=[3, 3], dilation=[1, 1], groups= 1,
bias=True) #ResNet::ResNet/Conv2d[conv1]/input.2
```

d. **Configuration Restrictions**

Due to the restriction of DPU device design, if quantized models need to be deployed in DPU device, the quantization configuration should meet the restrictions as below:

```
method: diffs or maxmin
round_mode: std_round for weights, bias, and input;
half_up for activation.
symmetry: true
per_channel: false
signed: true
narrow_range: true
scale_type: poweroftwo
calib_statistic_method: modal.
```

And for CPU and GPU device, there is no restriction as DPU device. However, there are some conflicts when using different configurations.

For example, if calibration method is ‘maxmin’, ‘percentile’, ‘mse’ or ‘entropy’, the calibration statistic method ‘modal’ is not supported. If symmetry mode is asymmetry, the calibration method ‘mse’ and ‘entropy’ are not supported. Quantization tool will give error message if there are configuration conflicts.

vai_q_pytorch QAT

Assuming that there is a pre-defined model architecture, use the following steps to do quantization aware training. Take the ResNet18 model from Torchvision as an example. The complete model definition is here.

1. Check if there are non-module operations to be quantized. ResNet18 uses `‘+’` to add two tensors. Replace them with `pytorch_nndct.nn.modules.functional.Add`.
2. Check if there are modules to be called multiple times. Usually such modules have no weights; the most common one is the `torch.nn.ReLu` module. Define multiple such modules and then call them separately in a forward pass. The revised definition that meets the requirements is as follows:

```
class BasicBlock(nn.Module):
  expansion = 1

  def __init__(self,
               inplanes,
               planes,
               stride=1,
               downsample=None,
               groups=1,
               base_width=64,
               dilation=1,
               norm_layer=None):
    super(BasicBlock, self).__init__()
    if norm_layer is None:
      norm_layer = nn.BatchNorm2d
    if groups != 1 or base_width != 64:
      raise ValueError('BasicBlock only supports groups=1
and base_width=64')
    if dilation > 1:
      raise NotImplementedError("Dilation > 1 not
```

```
supported in BasicBlock")
        # Both self.conv1 and self.downsample layers
downsample the input when stride != 1
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = norm_layer(planes)
        self.relu1 = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride

        # Use a functional module to replace '+'
        self.skip_add = functional.Add()

        # Additional defined module
        self.relu2 = nn.ReLU(inplace=True)

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu1(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
          identity = self.downsample(x)

        # Use function module instead of '+'
        # out += identity
        out = self.skip_add(out, identity)
        out = self.relu2(out)

        return out
```

3. Insert `QuantStub` and `DeQuantStub`.
   Use `QuantStub` to quantize the inputs of the network and `DeQuantStub` to de-quantize the outputs of the network. Any sub-network from `QuantStub` to `DeQuantStub` in a forward pass will be quantized. Multiple QuantStub-DeQuantStub pairs are allowed.

```
class ResNet(nn.Module):

  def __init__(self,
               block,
               layers,
               num_classes=1000,
               zero_init_residual=False,
               groups=1,
               width_per_group=64,
               replace_stride_with_dilation=None,
               norm_layer=None):
    super(ResNet, self).__init__()
    if norm_layer is None:
      norm_layer = nn.BatchNorm2d
    self._norm_layer = norm_layer

    self.inplanes = 64
    self.dilation = 1
    if replace_stride_with_dilation is None:
      # each element in the tuple indicates if we should
replace
      # the 2x2 stride with a dilated convolution instead
      replace_stride_with_dilation = [False, False,
False]
    if len(replace_stride_with_dilation) != 3:
      raise ValueError(
          "replace_stride_with_dilation should be None "
          "or a 3-element tuple, got
{}".format(replace_stride_with_dilation))
    self.groups = groups
    self.base_width = width_per_group
    self.conv1 = nn.Conv2d(
        3, self.inplanes, kernel_size=7, stride=2,
padding=3, bias=False)
    self.bn1 = norm_layer(self.inplanes)
    self.relu = nn.ReLU(inplace=True)
    self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2,
padding=1)
    self.layer1 = self._make_layer(block, 64, layers[0])
    self.layer2 = self._make_layer(
        block, 128, layers[1], stride=2,
```

```
dilate=replace_stride_with_dilation[0])
    self.layer3 = self._make_layer(
        block, 256, layers[2], stride=2,
dilate=replace_stride_with_dilation[1])
    self.layer4 = self._make_layer(
        block, 512, layers[3], stride=2,
dilate=replace_stride_with_dilation[2])
    self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
    self.fc = nn.Linear(512 * block.expansion,
num_classes)

    self.quant_stub = nndct_nn.QuantStub()
    self.dequant_stub = nndct_nn.DeQuantStub()

    for m in self.modules():
      if isinstance(m, nn.Conv2d):
        nn.init.kaiming_normal_(m.weight, mode='fan_out',
nonlinearity='relu')
      elif isinstance(m, (nn.BatchNorm2d, nn.GroupNorm)):
        nn.init.constant_(m.weight, 1)
        nn.init.constant_(m.bias, 0)

    # Zero-initialize the last BN in each residual
branch,
    # so that the residual branch starts with zeros, and
each residual block behaves like an identity.
    # This improves the model by 0.2~0.3% according to
https://arxiv.org/abs/1706.02677
    if zero_init_residual:
      for m in self.modules():
        if isinstance(m, Bottleneck):
          nn.init.constant_(m.bn3.weight, 0)
        elif isinstance(m, BasicBlock):
          nn.init.constant_(m.bn2.weight, 0)

  def forward(self, x):
    x = self.quant_stub(x)

    x = self.conv1(x)
    x = self.bn1(x)
    x = self.relu(x)
    x = self.maxpool(x)
```

```
    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)
    x = self.layer4(x)

    x = self.avgpool(x)
    x = torch.flatten(x, 1)
    x = self.fc(x)
    x = self.dequant_stub(x)
    return x
```

4. Use QAT APIs to create the quantizer and train the model.

```
def _resnet(arch, block, layers, pretrained, progress,
**kwargs):
  model = ResNet(block, layers, **kwargs)
  if pretrained:
    #state_dict =
load_state_dict_from_url(model_urls[arch],
progress=progress)
    state_dict = torch.load(model_urls[arch])
    model.load_state_dict(state_dict)
  return model

def resnet18(pretrained=False, progress=True, **kwargs):
  r"""ResNet-18 model from
    `"Deep Residual Learning for Image Recognition"
<https://arxiv.org/pdf/1512.03385.pdf>'_

    Args:
        pretrained (bool): If True, returns a model pre-
trained on ImageNet
        progress (bool): If True, displays a progress bar
of the download to stderr
    """
  return _resnet('resnet18', BasicBlock, [2, 2, 2, 2],
pretrained, progress,
                 **kwargs)

model = resnet18(pretrained=True)
```

```
# Generate dummy inputs.
input = torch.randn([batch_size, 3, 224, 224],
dtype=torch.float32)

# Create a quantizer
from pytorch_nndct import QatProcessor
qat_processor = QatProcessor(model, inputs, bitwidth=8)
quantized_model =
qat_processor.trainable_model()optimizer =
torch.optim.Adam(
        quantized_model.parameters(),
        lr,
        weight_decay=weight_decay)

# Use the optimizer to train the model, just like a
normal float model.
…
```

5. Get the deployable model and test it.
   Convert the quantized model to a deployable model after training is complete. The accuracy of the deployable model may differ slightly from the accuracy of the quantized model.

```
output_dir = 'qat_result'
deployable_model =
qat_processor.to_deployable(quantized_model, output_dir)
validate(val_loader, deployable_model, criterion, gpu)
```

6. Export xmodel from the deployable model.
   `batch size=1` is a must for the compilation of xmodel.

```
# Use cpu mode to export xmodel.
deployable_model.cpu()
val_subset = torch.utils.data.Subset(val_dataset,
list(range(1)))
subset_loader = torch.utils.data.DataLoader(
    val_subset,
    batch_size=1,
    shuffle=False,
    num_workers=8,
    pin_memory=True)
# Must forward deployable model at least 1 iteration with
```

```
batch_size=1
for images, _ in subset_loader:
  deployable_model(images)
qat_processor.export_xmodel(output_dir)
```

vai_q_pytorch QAT Requirements

Generally, there is a small accuracy loss after quantization, but for some networks such as MobileNets, the accuracy loss can be large. In this situation, first try fast finetune. If fast finetune does not yield satisfactory results, QAT can be used to further improve the accuracy of the quantized models.
The QAT APIs have some requirements for the model to be trained.

1. All operations to be quantized must be instances of the `torch.nn.Module` object, rather than Torch functions or Python operators. For example, it is common to use `'+'` to add two tensors in PyTorch. However, this is not supported in QAT. Thus, replace `'+'` with `pytorch_nndct.nn.modules.functional.Add`. Operations that need replacement are listed in the following table.

   Table: Operation-Replacement Mapping

Operation	Replacement
`+`	`pytorch_nndct.nn.modules.functional.Add`
`-`	`pytorch_nndct.nn.modules.functional.Sub`
`torch.add`	`pytorch_nndct.nn.modules.functional.Add`
`torch.sub`	`pytorch_nndct.nn.modules.functional.Sub`

   !! Important: A module to be quantized cannot be called multiple times in the forward pass.

2. Use `pytorch_nndct.nn.QuantStub` and `pytorch_nndct.nn.DeQuantStub` at the beginning and end of the network to be quantized. The network can be the complete network or a partial sub-network.

Guidelines for Better Training Results

The following are some tips for getting better training results:

- Load the pre-trained floating-point weights as initial values to start the quantization aware training if possible. It is possible to train from scratch with random initial values, but this will make training more difficult and long.
- If pre-trained floating-point weights are loaded, then different initial learning rates and learning rate decrease strategies need to be used for the network parameters and quantizer parameters, respectively. In general, the learning rate of network parameters needs to be set small, while the learning rate of quantizer parameters needs to be larger.

```
model = qat_processor.trainable_model()
param_groups = [{
    'params': model.quantizer_parameters(),
    'lr': 1e-2,
    'name': 'quantizer'
}, {
    'params': model.non_quantizer_parameters(),
    'lr': 1e-5,
    'name': 'weight'
}]
optimizer = torch.optim.Adam(param_groups)
```

- For the choice of optimizer, avoid using torch.optim.SGD, as this optimizer may prevent the training from converging. Xilinx recommends using torch.optim.Adam or torch.optim.RMSprop and their variants.

vai_q_pytorch Usage

This section introduces the usage of execution tools and APIs to implement quantization and generate a model to be deployed on the target hardware. The APIs in the module `pytorch_binding/pytorch_nndct/apis/quant_api.py` are as follows:

class torch_quantizer()

Class `torch_quantizer` creates a quantizer object.

```
class torch_quantizer():
  def __init__(self,
               quant_mode: str, # ['calib', 'test']
               module: torch.nn.Module,
```

```
                    input_args: Union[torch.Tensor, Sequence[Any]]
 = None,
                    state_dict_file: Optional[str] = None,
                    output_dir: str = "quantize_result",
                    bitwidth: int = 8,
                    device: torch.device = torch.device("cuda"),
                    quant_config_file: Optional[str] = None,
                    target: Optional[str]=None):
```

Arguments

Quant_mode

An integer that indicates which quantization mode the process is using. "calib" for calibration of quantization, and "test" for evaluation of quantized model.

Module

Float module to be quantized.

Input_args

Input tensor with the same shape as real input of float module to be quantized, but the values can be random numbers.

State_dict_file

Float module pretrained parameters file. If float module has read parameters in, the parameter is not needed to be set.

Output_dir

Directory for quantization result and intermediate files. Default is “quantize_result”.

Bitwidth

Global quantization bit width. Default is 8.

Device

Run model on GPU or CPU.

Quant_config_file

Json file path for quantization strategy configuration.

Target

If target device is specified, the hardware-aware quantization is on. Default is None.

def export_quant_config(self)

This function exports information related to the quantization steps.

```
def export_quant_config(self):
```

def export_xmodel(self, output_dir, deploy_check)

This function exports the xmodel and dumps the output data of the operators for detailed data comparison.

```
def export_xmodel(self, output_dir, deploy_check):
```

Arguments

Output_dir
: Directory for quantization result and intermediate files. Default is “quantize_result.”

Deploy_check
: Flags to control dump of data for detailed data comparison. Default is FALSE. If it is set to TRUE, binary format data is dumped in the output_dir/deploy_check_data_int/ location.

def export_onnx_model(self, output_dir, verbose)

The function is to export onnx format quantized model

```
def export_onnx_model(self, output_dir, verbose):
```

Arguments

Output_dir
: Directory for quantization result and intermediate files. The default value is “quantize_result”

Verbose
: Flag to control the display of verbose log.

def export_torch_script(self, output_dir, verbose)

The function is to export torch script format quantized model

```
def export_torch_script(self, output_dir, verbose):
```

Arguments

Output_dir

Directory for quantization result and intermediate files. The default value is “quantize_result”

Verbose

Flag to control the display of verbose log.

Class Inspector

Class Inspector creates a inspector object as follows:

```
class Inspector():
def __init__(self, name_or_fingerprint: str):
```

Arguments

name_or_fingerprint

Specify the hardware target name or fingerprint

def inspect(…)

The function is to inspect a float model

```
def inspect(self,
            module: torch.nn.Module,
            input_args: Union[torch.Tensor, Tuple[Any]],
            device: torch.device = torch.device("cuda"),
            output_dir: str = "quantize_result",
            verbose_level: int = 1,
            image_format: Optional[str] = None):
```

Arguments

module

Float module to be depolyed

input_args

Input tensor with the same shape as real input of float module, but the value can be random number

device

Trace model on GPU or CPU

output_dir

Directory for inspection results

verbose_level

Control the level of detail of the inspection results displayed on the screen. The default value is 1.

0: turn off printing inspection results

1: print summary report of operations assigned to CPU

2: print summary report of device allocation of all operations

image_format

Export visualized inspection result. Supports SVG and PNG image formats.

vai_q_pytorch message

In this part, some important messages are listed and can be searched by message ID. For every message, it can help users to identify the issues among their model deployment, and gives possible solution for the issue.

VAIQ_WARN

Vai_q_pytorch prints warning message on screen when there is issue may causing the quantization result has problem or incomplete (check according to the message text), but the process can be performed to its end, the format of this kind of message is "[VAIQ_WARN][MESSAGE_ID]: message text"
List important warning messages in the following table:

Table: Vai_q_pytorch warning message table

Message ID	Description
QUANTIZER_TORCH_BATCHNORM_AFFINE	BatchNorm OP attribute affine=False has been replaced by affine=True when parsing the model.
QUANTIZER_TORCH_BITWIDTH_MISMATCH	Bit width setting in configuration file is conflict with that from torch_quantizer API, will use that in configuration file.
QUANTIZER_TORCH_CONVERT_XMODEL	Convert to xmodel failed. Check message text to locate the reason.
QUANTIZER_TORCH_CUDA_UNAVAILABLE	CUDA (HIP) is not available, change device to CPU
QUANTIZER_TORCH_DATA_PARALLEL	Data parallel is not supported. The wrapper 'torch.nn.DataParallel' has been removed in vai_q_pytorch.
QUANTIZER_TORCH_DEPLOY_MODEL	Only quantization aware training process has deployable model.
QUANTIZER_TORCH_DEVICE_MISMATCH	The device of input arguments mismatch with quantizer device type.
QUANTIZER_TORCH_EXPORT_XMODEL	Failed to generate xmodel due to some reasons. Refer to the message text.
QUANTIZER_TORCH_FINETUNE_IGNORED	Fast finetune function will be ignored in test mode!
QUANTIZER_TORCH_FLOAT_OP	vai_q_pytorch recognize the list OP as a float operator by default.
QUANTIZER_TORCH_INSPECTOR_PATTERN	The OP may be fused by compiler and will be assigned to DPU.
QUANTIZER_TORCH_LEAKYRELU	Force to change negative_slope of LeakyReLU to 0.1015625 because DPU only supports this value. It is recommended to change all negative_slope of LeakyReLU to 0.1015625 and re-train the float model for better deployed model accuracy.
QUANTIZER_TORCH_MATPLOTLIB	matplotlib is needed for visualization but not found. It needs to be installed.

Message ID	Description
QUANTIZER_TORCH_MEMORY_SHORTAGE	There is not enough memory for fast fine-tune and this process will be ignored!. Try to use a smaller calibration dataset.
QUANTIZER_TORCH_NO_XIR	Can't find XIR package in environment. It needs to be installed.
QUANTIZER_TORCH_REPLACE_RELU6	ReLU6 has been replaced by ReLU.
QUANTIZER_TORCH_REPLACE_SIGMOID	Sigmoid has been replaced by Hardsigmoid.
QUANTIZER_TORCH_REPLACE_SILU	SiLU has been replaced by Hardswish.
QUANTIZER_TORCH_SHIFT_CHECK	Quantization scale is too large or too small.
QUANTIZER_TORCH_TENSOR_NOT_QUANTIZED	Some tensors are not quantized, please check their particularity.
QUANTIZER_TORCH_TENSOR_TYPE_NOT_QUANTIZABLE	The tensor type of the node cannot be quantized. Only support float32/double/float16 quantization.
QUANTIZER_TORCH_TENSOR_VALUE_INVALID	The tensor has "inf" or "nan" value. Quantization for this tensor is ignored.
QUANTIZER_TORCH_TORCH_VERSION	Only support exporting torch script with pytorch 1.10 and later version.
QUANTIZER_TORCH_XIR_MISMATCH	XIR version does not match current vai_q_pytorch.
QUANTIZER_TORCH_XMODEL_DEVICE	Not support to dump xmodel when target device is not DPU.
QUANTIZER_TORCH_REUSED_MODULE	Reused module may lead to low accuracy of QAT, make sure this is what you expect. Refer to the message text to locate the module with issue.
QUANTIZER_TORCH_DEPRECATED_ARGUMENT	The argument "device" is no longer needed. Device information is get from the model directly.
QUANTIZER_TORCH_SCALE_VALUE	Exported scale values are not trained.

VAIQ_ERROR

Vai_q_pytorch prints error message on screen when there is issue causing the process cannot be performed anymore (check and solve the problem according to the message text), the format of this kind of message is "[VAIQ_ERROR][MESSAGE_ID]: message text"
List important error messages in the following table:

Table: Vai_q_pytorch error message table

Message ID	Description
QUANTIZER_TORCH_BIAS_CORRECTION	Bias correction file in quantization result directory does not match current model.
QUANTIZER_TORCH_CALIB_RESULT_MISMATCH	Node name mismatch is found when loading quantization steps of tensors. Please make sure vai_q_pytorch version and pytorch version for test mode are the same as those in calibration (or QAT training) mode.
QUANTIZER_TORCH_EXPORT_ONNX	The quantized module, which is based pytorch traced model, can not be exported to ONNX due to pytorch internal failure. The pytorch internal failure reason is listed in message text. May needs adjust float model code.
QUANTIZER_TORCH_EXPORT_XMODEL	Fail to convert graph to xmodel. Needs check the reasons in message text.
QUANTIZER_TORCH_FAST_FINETINE	Fast fine-tuned parameter file does not exist. Call load_ft_param in model code to load them.
QUANTIZER_TORCH_FIX_INPUT_TYPE	Data type or value is illegal in arguments of quantization OP when exporting ONNX format model.
QUANTIZER_TORCH_ILLEGAL_BITWIDTH	The configuration of tensors quantization is illegal. It should be integer, and in range given in message text.

Message ID	Description
QUANTIZER_TORCH_IMPORT_KERNEL	Importing vai_q_pytorch library file error. Check pytorch version matching vai_q_pytorch version (pytorch_nndct.__version__) or not.
QUANTIZER_TORCH_NO_CALIB_RESULT	Quantization result file does not exist. Please check calibration is done or not.
QUANTIZER_TORCH_NO_CALIBRATION	Quantization calibration is not performed completely, check if module FORWARD function is called! FORWARD function of torch_quantizer.quant_model needs to be called in user code explicitly. Please refer to the example code at https://github.com/Xilinx/Vitis-AI/blob/master/src/Vitis-AI-Quantizer/vai_q_pytorch/example/resnet18_quant.py.
QUANTIZER_TORCH_NO_FORWARD	torch_quantizer.quant_model FORWARD function must be called before exporting quantization result. Please refer to example code at https://github.com/Xilinx/Vitis-AI/blob/master/src/Vitis-AI-Quantizer/vai_q_pytorch/example/resnet18_quant.py.
QUANTIZER_TORCH_OP_REGIST	The type of OP can't be registered multiple times.
QUANTIZER_TORCH_PYTORCH_TRACE	Failed to get pytorch traced graph from model and input arguments. The pytorch internal failure reason is reported in message text. May needs adjust float model code.
QUANTIZER_TORCH_QUANT_CONFIG	Quantization configuration items are illegal. Refer to the message text.
QUANTIZER_TORCH_SHAPE_MISMATCH	Tensors shape are mismatch. Refer to the message text.
QUANTIZER_TORCH_TORCH_VERSION	PyTorch version is not supported for the function or does not match vai_q_pytorch version

Message ID	Description
	(pytorch_nndct.__version__). Refer to the message text.
QUANTIZER_TORCH_XMODEL_BATCHSIZE	Batch size must be 1 when exporting xmodel.
QUANTIZER_TORCH_INSPECTOR_OUTPUT_FORMAT	Inspector only support dump svg or png format.
QUANTIZER_TORCH_INSPECTOR_INPUT_FORMAT	Inspector no longer support fingerprint. Please provide architecture name instead.
QUANTIZER_TORCH_UNSUPPORTED_OPS	The quantization of the op is not supported.
QUANTIZER_TORCH_TRACED_NOT_SUPPORT	The model produced by 'torch.jit.script' is not supported in vai_q_pytorch.
QUANTIZER_TORCH_NO_SCRIPT_MODEL	vai_q_pytorch does not find any script model.
QUANTIZER_TORCH_REUSED_MODULE	The quantized module has been called multiple times in forward pass. If you want to share quantized parameters in multiple calls, call trainable_model with "allow_reused_module=True"
QUANTIZER_TORCH_DATA_PARALLEL_NOT_ALLOWED	torch.nn.DataParallel object is not allowed.
QUANTIZER_TORCH_INPUT_NOT_QUANTIZED	Input is not quantized. Please use QuantStub/DeQuantStub to define quantization scope.
QUANTIZER_TORCH_NOT_A_MODULE	Quantized operation must be instance of "torch.nn.Module", please replace the function by a "torch.nn.Module" object. Original source range is indicated in message text.
QUANTIZER_TORCH_QAT_PROCESS_ERROR	Must call "trainable_model" first before getting deployable model.
QUANTIZER_TORCH_QAT_DEPLOYABLE_MODEL_ERROR	The given trained model has BN fused to CONV and cannot be converted to a deployable model. Make sure model.fuse_conv_bn() is not called.
QUANTIZER_TORCH_XMODEL_DEVICE	XModel can only be exported in CPU mode, use deployable_model(src_dir, used_for_xmodel=True)

Message ID	Description
	to get a CPU model.