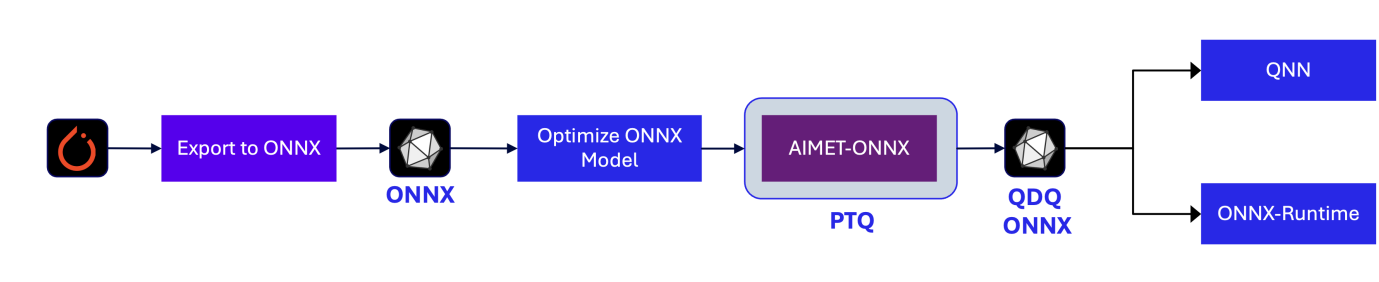
Post Training Quantization

Post-Training Quantization (PTQ) is the process of determining the appropriate scale and offset parameters for the quantizers inserted into a model's computation graph. While quantization parameters for weights can typically be precomputed, determining parameters for activations requires running a small, representative dataset through the model to collect range statistics.

This process of computing scale and offset values is commonly referred to as calibration.

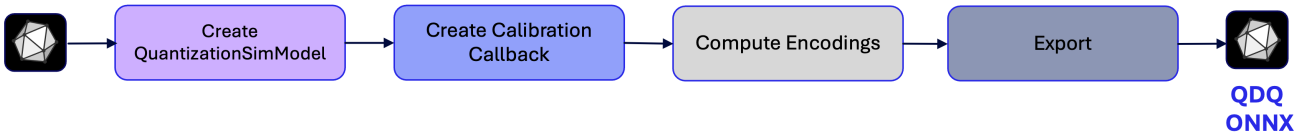
Both aimet-onnx or aimet-torch supports PTQ.



We recommend using aimet-onnx for PTQ for the following reasons:

1. Captured graph
    - Optimize model before quantization
2. Better alignment downstream
    - PyTorch may export certain operation to multiple operations in ONNX leading to missing quantizer information.
    - This missing quantizer information could lead to accuracy difference between on-target and off-target(simulation).

Workflow



Skip to content ıple for calibrating a MobileNetV2 model.

Prerequisites

1. Download ImageNet dataset

```
wget -P ./imagenet_dataset https://image-net.org/data/ILSVRC/2012/ILSVRC2012_devkit_t12.t
wget -P ./imagenet_dataset https://image-net.org/data/ILSVRC/2012/ILSVRC2012_img_val.tar
```

If you already have imagenet dataset locally that you would like to use, simply replace dataset path from *imagenet_dataset* later.

2. Load PyTorch model and dataset

> **Note**
>
> The examples below use a pre-trained MobileNetV2 model. You can also load your model instead.

PyTorch **ONNX**

```
import os
import numpy as np
import onnx
import torch
from tqdm import tqdm
```

Skip to content

```
from torchvision.models import MobileNet_V2_Weights, mobilenet_v2
pt_model = mobilenet_v2(weights=MobileNet_V2_Weights.DEFAULT)
input_shape = (1, 3, 224, 224)
dummy_input = torch.randn(input_shape)

# Modify file_path to save model at a different location
file_path = os.path.join('./', 'mobilenet_v2.onnx')
torch.onnx.export(pt_model,
                  (dummy_input,),
                  file_path,
                  input_names=['input'],
                  output_names=['output'],
                  dynamic_axes={
                      'input': {0: 'batch_size'},
                      'output': {0: 'batch_size'},
                  },
                  dynamo=False,
                  )

# Load exported ONNX model
model = onnx.load_model(file_path)
```

Skip to content

```
from torchvision import transforms, datasets
from torch.utils.data import DataLoader, random_split

BATCH_SIZE = 32
NUM_CALIBRATION_SAMPLES = 1024

def get_calibration_and_eval_data_loaders(path: str, batch_size: int):
    """
    Returns calibration and evaluation data-loader for ImageNet dataset from provided pat
    """
    transform = transforms.Compose(
        [
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
        ]
    )

    dataset = datasets.ImageNet(path, split='val', transform=transform)
    calibration_dataset, eval_dataset = random_split(
        dataset, [.9, 0.1]
    )

    calibration_data_loader = DataLoader(calibration_dataset, shuffle=True, batch_size=ba
    eval_data_loader = DataLoader(eval_dataset, shuffle=True, batch_size=batch_size)
    return calibration_data_loader, eval_data_loader

# Change path here to point to different dataset
PATH_TO_IMAGENET = './imagenet_dataset'
calibration_data_loader, eval_data_loader = get_calibration_and_eval_data_loaders(PATH_TO
```

Optionally simplify the exported onnx graph before quantization. This is not strictly required but may improve accuracy and runtime performance.

```
import onnxsim
model, _ = onnxsim.simplify(model)
```

Step 1: Creating a QuantSim model

Use AIMET to create a `QuantizationSimModel`. AIMET inserts fake quantization operations in the

Skip to content configures them.

PyTorch **ONNX**

```
from aimet_onnx.common.defs import QuantScheme
import aimet_onnx
from aimet_onnx import QuantizationSimModel

# Optionally use ["CUDAExecutionProvider", "CPUExecutionProvider"] to accelerate quantiza
providers = ["CPUExecutionProvider"]
sim = QuantizationSimModel(model,
                           param_type=aimet_onnx.int8,
                           activation_type=aimet_onnx.int16,
                           quant_scheme=QuantScheme.min_max,
                           config_file="default",
                           providers=providers)
```

Step 2: Creating a calibration callback

Before you can use the `QuantizationSimModel` for inference or training, you must compute scale and offset quantization parameters for each 'quantizer' node.

Create a routine to pass small, representative data samples through the model. A quick way to do this is to use the existing train or validation data loader to extract samples and pass them to the model.

500 to 1000 representative data samples are sufficient to compute the quantization parameters.

PyTorch **ONNX**

```
input_name = model.graph.input[0].name
def onnx_data_generator(num_batches):
    """
    Example conversion from torch dataloader to onnx model inputs
    """
    for i, (data, _) in enumerate(calibration_data_loader):
        if i >= num_batches:
            break
        yield {input_name: data.numpy()}
```

Skip to content

Step 3: Computing encodings

Next, call `QuantizationSimModel.compute_encodings()` to use the callback to pass representative data through the quantized model. The quantizers in the quantized model use the observed inputs to initialize their quantization encodings. “Encodings” refers to the scale and offset quantization parameters.

PyTorch **ONNX**

In onnx, calibration can be done through a callback function or by passing calibration data directly as an iterable of model inputs (`Iterable[Dict[str, np.ndarray]]`)

```
sim.compute_encodings(onnx_data_generator(NUM_CALIBRATION_SAMPLES // BATCH_SIZE))
```

Step 4: Evaluation

Next, evaluate the `QuantizationSimModel` to measure the model’s accuracy after quantization.

PyTorch **ONNX**

```
correct_predictions = 0
total_samples = 0
for i, (inputs, labels) in enumerate(tqdm(eval_data_loader)):
    pred_probs, *_ = sim.session.run(None, {input_name: inputs.numpy()})
    pred_labels = np.argmax(pred_probs, axis=1)
    correct_predictions += np.sum(pred_labels == labels.numpy())
    total_samples += labels.shape[0]

accuracy = correct_predictions / total_samples
print(f'Quantized accuracy (W8A16): {accuracy:.4f}')
```

Out:

```
Quantized accuracy (W8A16): 0.7173
```

Step 5: Exporting the model

If the off-target accuracy of the quantized model is within acceptable limits (Step 4), we can

Skip to content

oyment and export the model to ONNX format. During export, all intermediate quantization operations are removed, and the quantization parameters—scale and offset—are serialized into a JSON file.

PyTorch ONNX

```
# Export the model for on-target inference. Saves ONNX model without quantization nodes
# and encodings file with all tensor encodings in JSON format at provided path.
sim.export(path='./', filename_prefix='quantized_mobilenet_v2')
```

API

PyTorch ONNX

Skip to content

Top level APIs

```
class aimet_onnx.QuantizationSimModel(model, *, param_type=int8,
    activation_type=int8, quant_scheme=QuantScheme.min_max, config_file=None,
    dummy_input=None, user_onnx_libs=None, providers=None, path=None)      [source]
```

Class that simulates the quantized model execution on a target hardware backend.

PARAMETERS:

- **model** (*onnx.ModelProto*) – ONNX ModelProto to quantize
- **param_type** (*qtype | str*) – quantized type to use for parameter tensors. Can be { int2, int4, int8, int16, float16, float32 } or `aimet_onnx.qtype`
- **activation_type** (*qtype | str*) – quantized type to use for activation tensors. Can be { int2, int4, int8, int16, float16, float32 } or `aimet_onnx.qtype`
- **quant_scheme** (*QuantScheme | str*) – Quantization scheme to use for calibration. Can be { tf_enhanced, min_max } or `QuantScheme`
- **config_file** (*str, optional*) – File path or alias of the configuration file. Alias can be one of { default, htp_v66, htp_v68, htp_v69, htp_v73, htp_v75, htp_v79, htp_v81 } (Default: *"default"*)
- **dummy_input** (*Dict[str, np.ndarray], optional*) – Sample input to the model. Only needed for non shape-inferable models with parameterized shapes
- **user_onnx_libs** (*List[str], optional*) – List of paths to all compiled ONNX custom ops libraries
- **providers** (*List, optional*) – Onnxruntime execution providers to use when building InferenceSession. If *None*, default provider is "CPUExecutionProvider"
- **path** (*str, optional*) – Directory to save temporary artifacts.

`compute_encodings(*args, **kwargs)` [source]

Computes encodings for all quantizers in the model.

This API will invoke *forward_pass_callback*, a function written by the user that runs forward pass(es) of the quantized model with a small, representative subset of the training dataset. By doing so, the quantizers in the quantized model will observe the inputs and initialize their quantization encodings according to the observed input statistics.

Skip to content

This function is overloaded with the following signatures:

compute_encodings(inputs) [source]

PARAMETERS:

inputs (*Iterable[Dict[str, np.ndarray]]*) – The set of model input samples to use during calibration

compute_encodings(forward_pass_callback) [source]

PARAMETERS:

forward_pass_callback (*Callable[[ort.InferenceSession], Any]*) – A function that takes a quantized model and runs forward passes with a small, representative subset of training dataset

compute_encodings(forward_pass_callback, forward_pass_callback_args) [source]

PARAMETERS:

- **forward_pass_callback** (*Callable[[ort.InferenceSession, T], Any]*) – A function that takes a quantized model and runs forward passes with a small, representative subset of training dataset
- **forward_pass_callback_args** (*T*) – The second argument to *forward_pass_callback*.

EXAMPLE

```
>>> sim = QuantizationSimModel(...)
>>> def run_forward_pass(session: ort.InferenceSession):
...     for input in dataset:
...         _ = sess.run(None, {"input": input})
...
>>> sim.compute_encodings(run_forward_pass)
```

export(path, filename_prefix, export_model=True, *, export_int32_bias=False, encoding_version=None) [source]

Skip to content

Compute encodings and export to files

PARAMETERS:

- **path** – dir to save encoding files
- **filename_prefix** – filename to save encoding files
- **export_model** (*bool, optional*) – If True, then ONNX model is exported. When False, only encodings are exported.
- **export_int32_bias** (*bool, optional*) – If true, generate and export int32 bias encoding on the fly (default: *True*)
- **encoding_version** (*str, optional*) – Version of the encoding format to use. (default: 1.0.0) Supported versions are: ['0.6.1', '1.0.0', '2.0.0']

EXAMPLE

```
>>> sim.export(path=".", filename_prefix="model", encoding_version="2.0.0")
```

Skip to content

Note

- We recommend you use onnx-simplifier before creating the QuantSim model.
- Since ONNX Runtime is used for optimized inference only, ONNX framework supports Post Training Quantization schemes (such as TF or TF-enhanced) to compute the encodings.

`aimet_onnx.quantsim.load_encodings_to_sim(quant_sim_model, onnx_encoding_path, strict=True, *, allow_overwrite=True, disable_missing_quantizers=True)` [source]

Loads the saved encodings to quant sim model. The encoding filename to load should end in .encodings, generated as part of quantsim export.

PARAMETERS:

- **quant_sim_model** (`QuantizationSimModel`) – Quantized model to load encodings for. Note: The model configuration should be the same as when encodings were exported.
- **onnx_encoding_path** (`str` | `dict`) – Path of the encodings file to load.
- **strict** – If set to True and encoding settings between encodings to load do not line up with Quantsim initialized settings, an assertion will be thrown. If set to False, quantizer settings will update to align with encodings to load.
- **allow_overwrite** – If true, loaded encodings will be overwritten by subsequent compute_encodings calls If false, loaded quantizer encodings will be frozen.
- **diable_missing_quantizers** – If true, quantizers which do not have encodings will be disabled.

RETURN TYPE:

`List[_EncodingMismatchInfo]`

RETURNS:

List of EncodingMismatchInfo objects containing quantizer names and mismatched settings

Quant Scheme Enum

`class aimet_onnx.common.defs.QuantScheme(value)` [source]

Quantization schemes

`classmethod from_str(alias)` [source]

Returns QuantScheme object from string alias

RETURN TYPE:

`ntScheme`

Skip to content