TLDR: Quantized Tensor is here. In this note I'll introduce some core concepts for quantized Tensor and list the current user facing API in Python. This means now you can play with the quantized Tensor in PyTorch, write quantized operators and quantized Modules. Support for developing full quantized models either through post training quantization or quantization aware training will come soon.

## What is a Quantized Tensor?

Quantized Tensor is a Tensor that is quantized from a float Tensor, it stores quantization parameters like scale and zero_point and the data will be integers, and we can call quantized operators on it. A quantized operator is an operator that takes quantized Tensors and produces quantized Tensors.

Quantized Tensor holds a Quantizer object which can be shared among multiple Tensors and it has special quantized data types. Right now it supports qint8, quint8 and qint32 data types which corresponds to int8, uint8 and int32 respectively, we will add support later for qint16 as well.

## What is Quantizer

A Quantizer is a class that stores necessary information for quantization and has quantize and dequantize methods that can convert between Tensor and quantized Tensor. We have a base Quantizer class and different types of Quantizer will inherit from the same Quantizer. All Quantizer will store a enum called QScheme, which describes which quantization scheme it is, for example PerTensorAffine, PerChannelAffine etc. It also stores a ScalarType because we can quantize a given float Tensor to different data types.

Currently supported quantization schemes are: PerTensorAffine and PerChannelAffine, currently supported data types are qint8, quint8 and qint32. ## Quantized Tensor APIs I'll use Python API as an example, C++ APIs are similar.

### CREATING A QUANTIZED TENSOR

Right now we support three ways of creating a quantized Tensor:

```
# 1. Get a quantized Tensor by quantizing unquantized float Tensors
float_tensor = torch.randn(2, 2, 3)

scale, zero_point = 1e-4, 2
dtype = torch.qint32
q_per_tensor = torch.quantize_per_tensor(float_tensor, scale, zero_point, dtype)

# we also support per channel quantization
scales = torch.tensor([1e-1, 1e-2, 1e-3])
zero_points = torch.tensor([-1, 0, 1])
```

```
channel_axis = 2
q_per_channel = torch.quantize_per_channel(float_tensor, scales, zero_points, axis=channel_a

# 2. Create a quantized Tensor directly from empty_quantized functions
# Note that _empty_affine_quantized is a private API, we will replace it
# something like torch.empty_quantized_tensor(sizes, quantizer) in the future
q = torch._empty_affine_quantized([10], scale=scale, zero_point=zero_point, dtype=dtype)

# 3. Create a quantized Tensor by assembling int Tensors and quantization parameters
# Note that _per_tensor_affine_qtensor is a private API, we will replace it with
# something like torch.form_tensor(int_tensor, quantizer) in the future
int_tensor = torch.randint(0, 100, size=(10,), dtype=torch.uint8)

# The data type will be torch.quint8, which is the corresponding type
# of torch.uint8, we have following correspondance between torch int types and
# torch quantized int types:
# - torch.uint8 -> torch.quint8
# - torch.int8 -> torch.qint8
# - torch.int32 -> torch.qint32
q = torch._make_per_tensor_quantized_tensor(int_tensor, scale, zero_point)  # Note no `dtyp
```

With the current API, we'll have to specialize the function for each quantization scheme, for example, if we want to quantize a Tensor, we'll have `quantize_per_tensor` and `quantize_per_channel`. Similarly for `q_scale` and `q_zero_point`, we should have a single quantize function which takes a `Quantizer` as argument. For inspecting the quantization parameters, we should have quantized Tensor return a `Quantizer` object so that we can inspect the quantization parameters on the `Quantizer` object rather than putting everything in the Tensor API. Current infra is not ready for this support, and is currently under development.

### OPERATIONS ON QUANTIZED TENSOR

```
# Dequantize
dequantized_tensor = q.dequantize()

# Quantized Tensor supports slicing like usual Tensors do
s = q[2] # a quantized Tensor of with same scale and zero_point
                          # that contains the values of the 2nd row of the original quantize
                          # same as q_made_per_tensor[2, :]

# Assignment
q[0] = 3.5 # quantize 3.5 and store the int value in quantized Tensor

# Copy
# we can copy from a quantized Tensor of the same size and dtype
# but different scale and zero_point
```

```
scale1, zero_point1 = 1e-1, 0
scale2, zero_point2 = 1, -1
q1 = torch._empty_affine_quantized([2, 3], scale=scale1, zero_point=zero_point1, dtype=torch
q2 = torch._empty_affine_quantized([2, 3], scale=scale2, zero_point=zero_point2, dtype=torch
q2.copy_(q1)

# Permutation
q1.transpose(0, 1)  # see https://pytorch.org/docs/stable/torch.html#torch.transpose
q1.permute([1, 0])  # https://pytorch.org/docs/stable/tensors.html#torch.Tensor.permute
q1.contiguous()  # Convert to contiguous Tensor

# Serialization and Deserialization
import tempfile
with tempfile.NamedTemporaryFile() as f:
    torch.save(q2, f)
    f.seek(0)
    q3 = torch.load(f)
```

**INSPECTING A QUANTIZED TENSOR**

```
# Check size of Tensor
q.numel(), q.size()

# Check whether the tensor is quantized
q.is_quantized

# Get the scale of the quantized Tensor, only works for affine quantized tensor
q.q_scale()

# Get the zero_point of quantized Tensor
q.q_zero_point()

# get the underlying integer representation of the quantized Tensor
# int_repr() returns a Tensor of the corresponding data type of the quantized data type
# e.g.for quint8 Tensor it returns a uint8 Tensor while preserving the MemoryFormat when pos
q.int_repr()

# If a quantized Tensor is a scalar we can print the value:
# item() will dequantize the current tensor and return a Scalar of float
q[0].item()

# printing
print(q)
# tensor([0.0026, 0.0059, 0.0063, 0.0084, 0.0009, 0.0020, 0.0094, 0.0019, 0.0079,
#         0.0094], size=(10,), dtype=torch.quint8,
#        quantization_scheme=torch.per_tensor_affine, scale=0.0001, zero_point=2)
```

```
# indexing
print(q[0]) # q[0] is a quantized Tensor with one value
# tensor(0.0026, size=(), dtype=torch.quint8,
#        quantization_scheme=torch.per_tensor_affine, scale=0.0001, zero_point=2)
```

For more up to date examples about Tensor API please refer to tests

## Quantized Operators/Kernels

We are also working on quantized operators like quantized QRelu, QAdd, QCat, QLinear, QConv etc. We either have naive implementation of an operator or wrap around fbgemm implementations in the operator. All operators are registered in C10 and they are only in CPU right now. We also have instructions on how to write quantized ops/kernels.

## Quantized Modules

We also have quantized modules that wraps these kernel implementations which live in torch.nn.quantized namespace and will be used in model development. We will provide utility functions to replace torch.nn.Module to torch.nn.quantized.Module but users are free to use them directly as well. We will try to match APIs of quantized module with the its counterpart in torch.nn.Module as much as possible.

## Code Location

- Core Data Structures for Quantized Tensor:https://github.com/pytorch/pytorch/blob/master/aten/src/AT
- Native Functions for Quantized Tensor: https://github.com/pytorch/pytorch/tree/master/aten/src/ATen/
- Quantized CPU Ops: https://github.com/pytorch/pytorch/tree/master/aten/src/ATen/native/quantized/
- Python Tests: https://github.com/pytorch/pytorch/blob/master/test/test_quantized_tensor.py
- CPP Tests: https://github.com/pytorch/pytorch/blob/master/aten/src/ATen/test/quantized_test.cpp
- Quantized Modules: https://github.com/pytorch/pytorch/blob/master/torch/nn/quantized/modules
- Quantization Utilities: https://github.com/pytorch/pytorch/blob/master/torch/nn/quantization

## What's Next

- Exposing Quantizer in Python Frontend
  - Right now we don't have a common API for creating a quantized Tensor for different quantization schemes, instead we have a set of APIs we need to write for each quantization scheme. This feature needs some infra support from the JIT team, we'll implement it when that is ready.
- Supporting Other Quantization Schemes and DataTypes
  - Right now we only support PerTensorAffine and PerChannelAffine quantization, we might add other quantization schemes in the future.
- Supporting Quantized Tensor on other devices

  - Right now we only have quantized Tensor on CPU, we might add quantized Tensor support in other devices like CUDA and HIP if needed.
- More Quantized Ops/Kernels and Quantized Modules
  - We will add more quantized operators and modules as we on-boarding more models to use quantization.

## Related Documents

- Quantization master design doc: https://github.com/pytorch/pytorch/issues/18318
- PyTorch Quantization Design Proposal: https://github.com/pytorch/pytorch/wiki/torch_quantization_de