We plan to support quantization in pytorch - enabling fast inference and reduced memory requirements. Quantization in PyTorch supports 8 bit integer tensors that can save 75% of the model size and memory bandwidth. We are rolling out quantization support for x86 CPUs and plan to expand to support a broader range of platforms, including mobile in later releases. We will start with support for quantization in eager mode, with enhanced integration with jit being planned for future releases.

Users have the option to quantize their models using three methods:

Scheme	Post Train	Quantized weights	Quantized Activations	Requires Calibration/Training?	Performance	Accuracy
Dynamic Quantization	Y	Y	N	N	Good	Good
Post training quantization	Y	Y	Y	Y	Better	Good
Quantization aware training	N	Y	Y	Y	Better	Better

Existing models can be converted to 8 bit integer after training (Post Training Quantization) or trained specifically to be executed in quantized form (Quantization Aware Training) — which often results in model accuracy closer to the original floating point model.

Quantized representation

The quantization scheme that is currently supported is **per tensor**** asymmetric linear quantization**, which means that all the values within the tensor are scaled the same way and that the minimum and the maximum of the input data is mapped linearly to the minimum and the maximum of the quantized data type such that zero is represented with no quantization error. We plan to add support for** per channel** linear quantization for weights for specific modules going forward.

The mapping is performed by converting the floating point tensors using

Note that for operators, we restrict support to:

1. 8 bit weights (data_type = qint8)
2. 8 bit activations (data_type = quint8)
3. 32 bit, symmetric quantization for bias (zero_point = 0, data_type = qint32)

Quantized Tensor

In order to do quantization in PyTorch, we need to be able to represent quantized data in Tensors. A quantized Tensor allows for storing quantized data (represented as int8/uint8/int32) along with quantization parameters like scale and zero_point. Quantized tensors allow for many useful operations making quantized arithmetic easy, in addition to allowing for serialization of data in a quantized format.

Here is a brief list of useful functions on quantized Tensor:

- `torch.quantize_linear(x: torch.tensor, scale: float, zero_point: int, dtype: torch.dtype)`: Quantizes input `x` into type `dtype` using the `scale` and `zero_point`
- `qx.dequantize()`: Dequantizes tensor `qx` into floating point
- `qx.__repr__()`: Prints the dequantized values of the quantized tensor `qx`
- `qx.int_repr()`: Prints the raw values in the quantized tensor `qx`.

```
>>> x = torch.rand(100,100)
>>> y = torch.quantize_linear(x, scale = 0.05, zero_point = 0, dtype = torch.qint8)
>>> torch.save(y,'qtz_tensor.pt')
>>> torch.save(x,'float_tensor.pt')
>>> print('Float tensor (bytes)', os.path.getsize('float_tensor.pt'))
Float tensor (bytes) 40344
>>> print('Quantized tensor (bytes)',os.path.getsize('qtz_tensor.pt'))
Quantized tensor (bytes) 10353
>>> print(y.int_repr())
# tensor([[11, 13,  3,  ..., 13, 18,  4],
#         [15, 18,  6,  ..., 10, 15, 17], torch.int8)
```

For a more comprehensive introduction to quantized Tensor please refer to the GitHub wiki.

Tensor operations:

Quantized tensors support a subset of torch Tensor operations, with support for more operations being added. Quantized tensor support is being added to existing methods in an API compatible manner. For example, torch.max() would work for both quantized and float tensors. Quantized tensors are supported by the following operations:

Tensor operations
torch.min
torch.max
torch.argmin
torch.argmax
torch.sort
torch.argsort

Tensor operations
torch.transpose
permute
contiguous
view

```
# Example tensor operations
>>> print(y.size()) # (100,100)
>>> print(torch.max(y)) # prints max
>>> print(torch.argmin(y))
>>> print(torch.transpose(y))
```

The general approach followed is to add support for quantized tensors to an existing torch method (torch.max(), for example). However, in certain cases like add and cat, additional arguments are needed for quantized methods.

In that case, we plan to support quantized operations with default values for the output scale and zero-point. We also plan to support out-variants, which allows us to specify the output scale and zero-point. In addition, we also support quantized versions of the op, which takes in scale and zero-point as arguments.

```
c = torch.add(a,b)
# Works for both quantized and float tensors, scale and zero-point for c
# is set to 0 and 1.
c = torch._empty_affine_quantized(scale = 0.5, zero-point = 32, dtype = torch.qint8)
torch.add(a,b, out = c)
# Can also do:
c = torch.ops.quantized.add(a, b, scale = 0.5, zero-point = 32)
```

Quantized Modules

TORCH.NN.QUANTIZED

We are developing torch.nn.quantized as a namespace for quantized modules. Quantized modules closely parallel their floating point counterparts and follow the following rules:

1. Identical module instantiation API
2. forward method takes in quantized tensors and output quantized tensors.
3. Weights (if present) are represented as quantized tensors.
4. Have a from_float() class method that allows for conversion from a floating point module to a quantized module.

The list of modules currently supported (and growing) are:

- Conv2d
- Linear
- ReLU

- MaxPool2d

TORCH.NN.QUANTIZED.DYNAMIC: Dynamic quantization refers to quantization of activations to int8 dynamically (per batch), performing the computations in reduced precision and returning the output in fp32. Modules that support dynamic quantization are under the torch.nn.quantized.dynamic name-space. These modules follow the same rules as torch.nn.quantized, except that they take in and return floating point activations.

The list of modules that we plan to support are:

- Linear
- RNN
- LSTM
- GRU
- RNNCell
- LSTMCell
- GRUCell
- Conv

FUNCTIONALS:

Functional operators support quantized tensors and are accessible via torch.nn.functional. We also provide functional equivalents of quantized modules at torch.nn.quantized.functional The following functionals support quantized tensors, in addition to existing support for float tensors:

- torch.nn.relu
- torch.nn.maxpool2d
- torch.nn.avgpool2d

```
>>> x = torch.rand(1, 3, 224,224)
>>> y = torch.quantize_linear(x, scale = 0.05, zero_point = 0, dtype = torch.qint8)
>>> v = torch.nn.functional.max_pool2d(y,2)
# v.size() = (1,3,112,112)
>>> z = torch.nn.functional.relu(v)
>>> print(torch.min(z))
# prints the minimum value of the quantized tensor
```

Quantized functionals supported are:

- torch.nn.quantized.functional.conv2d
- torch.nn.quantized.functional.linear

torch.nn.qat

Quantization aware training models quantization during training of both weights and activations. This is done by inserting fake quantization operations. For

modules that do not have weights, inserting fake-quant operation corresponds to applying a fakequant module at the output of the corresponding floating point module. However, for modules that have weights, we are developing torch.nn.qat to provide quantization aware training ready modules: We are currently adding support for:

- Conv2d
- Linear

INTRINSIC MODULES

Since quantization is sensitive to operator fusion, i.e: we can get better quantization accuracy if we quantize activations after operators are fused, we also support fused operations as modules under torch.nn._intrinsic name-space. Note that this name-space is experimental and modules here will be replaced as jit integration occurs.

torch.nn._intrinsic.quantized

- Conv2dRelu
- LinearRelu

For quantization aware training, it is non-trivial to combine batch norm and conv layers and we provide intrinsic modules for this purpose, in addition to providing intrinsic modules that match _intrinsic.quantized modules:

torch.nn._intrinsic.qat

- ConvBn2d
- ConvBnRelu2d
- ConvRelu2d
- LinearRelu

TORCH.QUANTIZATION

We introduce torch.quantization which consists of tools for eager mode quantization. The overall goals are:

1. Simplify the quantization process
2. Allow for easy customization of quantization.

This name-space contains utilities for simplifying eager mode quantization. The figure below provides an overview:

Model quantization in eager mode consists of the following steps:

1. Modify model definition (init and forward methods):
    1. Specify locations where activations should be quantized and dequantized (QuantStub, DeQuantStub

    2. Convert operations that require quantization information into modules (add, cat etc) (ref: make_module)
    3. Use nn._intrinsic modules for fusion.
    4. Specify quantization configuration: This can be done either as a dictionary (qConfigDict) or as a field .qconfig for each module.
    5. Modify forward function to use modules as needed.
2. Call the conversion functions in torch.quantization, three flows are supported with more being planned:
    1. Post training quantization:
        1. Fuse modules: The first step is calling torch.quantization.fuse_modules() to fuse convolution layers with batch norm and optionally Relu operations.
        2. Prepare modules:
            1. Propagate quantization configuration information across modules
            2. Insert observers to collect statistics for activations
        3. Calibrate: Run the model over representative data to collect statistics
        4. Convert: Quantize weights and replace floating point operations with their quantized counterparts.
    2. Dynamic quantization:
        1. Fuse modules: The first step is calling torch.quantization.fuse_modules() to fuse convolution layers with batch norm and optionally Relu operations.
        2. Prepare modules:
            1. Propagate quantization configuration information across modules
        3. Convert: Quantize weights and replace floating point operations with their dynamic quantized counterparts.
    3. Quantization aware training:
        1. Fuse modules: The first step is calling torch.quantization.fuse_modules() to fuse convolution layers with batch norm and optionally Relu operations.
        2. Prepare modules:
            1. Propagate quantization configuration information across modules
            2. Insert fake quantization modules to model quantization of activations
            3. Replace floating point modules with nn.qat modules to model quantization of weights.
        3. Train: Fine tune the model over training data.
        4. Convert: Quantize weights and replace floating point operations with their quantized counterparts.
    4. We also plan to provide support for mixed conversion: allowing for both static and dynamic quantization for different sub modules within a module.

We provide both single line APIs for convenience and more modular APIs for full user control in all cases. To simplify quantization we provide reference implementations of both Observers and FakeQuantization modules.

Observers: Modules that do not modify activations, but collect statistics. Have an API that provides quantization parameters for the tensor being observed.

FakeQuantizationModule: Module that modifies a float tensor to model quantization and returns a float tensor with the same size as the input. Since fake quantization is used in training, this module also supports back propagation. In addition, FakeQuant modules also have an API to provide the quantization parameters of the tensor being quantized.

Note that one can customize both observers and fake-quant modules to easily experiment with different quantization schemes.

Further work:

We are actively planning to add support for more modules and tensor operations with quantized tensors. In addition, we plan to provide multiple reference observer and fake-quantization modules for easy comparison of different quantization methods. In the longer term, we are working on closer integration of quantization with jit IR to further simplify the conversion process.

Code Location

Quantized tensors and kernels: * Core Data Structures for Quantized Tensor: https://github.com/pytorch/pytorch/tree/master/aten/src/ATen/quantized * Native functions for Quantized Tensor: https://github.com/pytorch/pytorch/tree/master/aten/src/ATen/native/
* Quantized CPU ops: https://github.com/pytorch/pytorch/tree/master/aten/src/ATen/native/quantized/cpu
* Tests for quantized tensors: https://github.com/pytorch/pytorch/blob/master/test/test_quantized_tensor.py

Quantized modules: * Modules:https://github.com/pytorch/pytorch/tree/master/torch/nn/quantized/modules
* Examples/tests:https://github.com/pytorch/pytorch/tree/master/test/test_nn_quantized.py

Dynamic quantized modules: * Modules:https://github.com/pytorch/pytorch/tree/master/torch/nn/quantized/d
* Examples/tests:https://github.com/pytorch/pytorch/tree/master/test/test_nn_quantized.py

Quantization * QAT modules:https://github.com/pytorch/pytorch/tree/master/torch/nn/qat/modules
* Examples/tests: : https://github.com/pytorch/pytorch/blob/master/test/test_qat.py

Intrinsic Modules: * Floating point intrinsic modules:https://github.com/pytorch/pytorch/tree/master/torch/nn
* Quantized intrinsic modules: https://github.com/pytorch/pytorch/tree/master/torch/nn/_intrinsic/quantized/
* intrinsic modules for quantization aware training: https://github.com/pytorch/pytorch/tree/master/torch/nn/_

Quantization * Quantization toolchain: https://github.com/pytorch/pytorch/tree/master/torch/quantization(W
* Examples/tests: https://github.com/pytorch/pytorch/blob/master/test/test_quantization.py