Define-by-run quantization

Note: this code is an early prototype and the API may change at any time.

Define-by-run quantization is a prototype of automated quantization syntax transforms for PyTorch Eager mode. High level algorithm:

1. take a user model and an example input
2. trace the model with example input and record the subgraphs of seen quantizeable ops
3. define quantization syntax transforms over the seen subgraphs
4. during execution of user code, dynamically call into the subgraphs from (3) when necessary

User API overview

```
from torch.ao.quantization._quantize_dbr import prepare, convert

m = M(...)
mp = prepare(m, example_input)
# calibration (not shown)
mq = convert(mp)
```

Framework concepts overview

The framework is defined with two major concepts:

1. Each non-leaf module has a child of type `AutoQuantizationState`, stored under the `_auto_quant_state` attribute name. This child contains the quantization related state such as captured subgraphs and observers.
2. During program execution, the framework overrides each module and quantizeable op to call hooks on the objects defined in (1), to implement the quantization syntax transforms.

For example, imagine a user module such as this one:

```
class Child(torch.nn.Module):
    def forward(self, x):
        return x + x

class Parent(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = torch.nn.Conv2d(1, 1, 1)
        self.child = Child()

    def forward(self, x):
        x = self.conv(x)
        x = self.child(x)
```

```
        return x

m = Parent().eval()
```

After model creation, the model hierarchy looks like

```
m (Parent)
|- conv (torch.nn.Conv2d)
|- child (Child)
```

After adding auto-observe, the model hierarchy looks like

```
m (Parent)
|- conv (torch.nn.Conv2d)
|- _auto_quant_state (AutoQuantizationState)
|- child (Child)
   |- _auto_quant_state (AutoQuantizationState)
```

Each `AutoQuantizationState` instance stores (a) subgraphs of seen quantizeable ops and (b) observers. Here is an example (in pseudocode, actual variable names many change):

```
AutoQuantizationState(
  (seen_op_infos): {
    0: (type): <built-in method add of type object at 0x112a7fe50>
       (input_tensor_infos): [
         QTensorInfo(id=0, inf_dtype=torch.float32),
         QTensorInfo(id=1, inf_dtype=torch.float32),
       ],
       (output_tensor_infos): [
         QTensorInfo(id=2, inf_dtype=torch.quint8),
       ],
  }
  (tensor_id_to_observer): ModuleDict(
    (0): MinMaxObserver(...),
    ...
  )
)
```

During program execution, the following overrides will be called, in order: 1. m.___call___ override start 2. m.conv.___call___ override start, end 3. m.child.___call___ override start 4. m.child::add.___torch_function___ override start, end 5. m.child.___call___ override end 6. m.___call___ override end

There are three flavors of hooks:

1. op hooks. These are called from the parent module on individual ops, and used to implement quantization of op subgraphs.
2. module I/O hooks. These are called from the parent module on child modules only, and used to enforce dtype of module outputs, if needed. For

example, if the model needs to output a fp32 tensor and the last op is int8, the conversion to fp32 will happen in this hook (and not in the op hook).
3. arg dequant hooks. These are called when the current op requires `torch.float` tensors. Any non-quantizeable op will go through these, and have any quantized tensor arguments dequantized.

Code overview

auto_trace.py

This file contains the logic for partial program capture. We override `__torch_function__` and `torch.nn.Module.__call__` to define the interception points. During tracing, calibration and inference, we dynamically execute quantization transforms from these interception points. The following pseudocode illustrates how both `add_auto_observation` and `add_auto_convert` call into quantization hooks:

```
# override of `__torch_function__`
def __torch_function__(cls, func, types, args, kwargs):
    ...

    # the framework provides `cur_module` of the current function
    # `quant_state` is the quantization state of the current module
    quant_state = cur_module._auto_quant_state

    # only some ops are quantizeable, the following line allows us to terminate
    # early for unquantizeable ones
    hook_type = get_torch_function_hook_type(parent_module, func)

    if hook_type is HookType.OP_HOOKS:

        # this line will throw an exception if control flow over quantizeable ops
        # is detected
        qstate.validate_cur_op(func)

        # "before" hook
        args, kwargs = qstate.op_prepare_before_hook(func, args, kwargs, ...)

        # run original function
        output = super().__torch_function__(func, types, args, kwargs)

        # "after" hook
        output = qstate.op_prepare_after_hook(func, output, args, kwargs, ...)

    else:
        output = super().__torch_function(func, types, args, kwargs)
```

```
    ...

    return output

# override of `torch.nn.Module.__call__`
def record_module(self, *args, **kwargs):
    # the framework keeps track of parent module of the current module
    parent_module = get_parent_module(...)
    parent_qstate = parent_module._auto_quant_state
    cur_module = self
    cur_qstate = self._auto_quant_state


    # the framework calculates when the current module needs op_hooks, io hooks
    # or arg_dequants
    hook_type = get_module_hook_type(parent_module, func)

    if hook_type is HookType.OP_HOOKS:
        # call before, during and after hooks on parent_qstate with the current op
        # execute original forward
        ...

    elif hook_type is HookType.MODULE_IO_HOOKS:
        # execute original forward
        # call hook on outputs of module for dtype transitions
        ...

    elif hook_type is HookType.ARG_DEQUANTS:
        # dequantize incoming arguments, if they are quantized
        # execute original forward
        ...

    else:
        # execute original forward

    ...

    return output
```

In detail:

calibration This happens in the `add_auto_observation` function.

1. For each non-leaf module in the model, we create a new `AutoQuantizationState` module and attach it as a child. This contains the quantization state

(subgraphs and observers).
2. For each `__torch_function__` and `torch.nn.Module.__call__` override, call quantization hooks if necessary. If `first_call` is true, this captures the subgraphs. Otherwise, this performs observation.

inference This happens in the `add_auto_convert` function.

1. For each `__torch_function__` and `torch.nn.Module.__call__` override, call quantization hooks if necessary. This performs the quantization inference syntax transforms.

quantization_state.py

This file defines `AutoQuantizationState`. This is an object which stores quantization state for its parent module. It contains the following state:

1. all captured quantization op subgraphs
2. all dynamically created observers and fake_quants

It also contains the following hooks:

1. module before and after hooks (used for dtype transitions)
2. function before and after hooks (used for dtype transitions and observation)
3. function replacement hooks (used for substiting quantized kernels)

auto_trace_rewriter.py

This file defines a custom FX tracer which can encode the transforms captured by `AutoQuantizationState` into an FX graph. This is useful because it provides a path to `torch.jit.script`.

TODO(future PR): write up more.

fusion.py

This file has a function which finds all potential module fusions of a model. This is implemented by tracing the model with the machinery in `auto_trace.py`, and traversing the found subgraphs to look for fusions. A list of module fqns to fuse is returned, which can then be plugged in to the Eager mode fusion APIs.

mappings.py

This file defines the quantization mappings (which fp32 ops are related to which int8 ops, which ops are quantizeable, etc.

TODO(future PR): delete this file and use existing mappings instead.

utils.py

This file contains various stateless utilities.