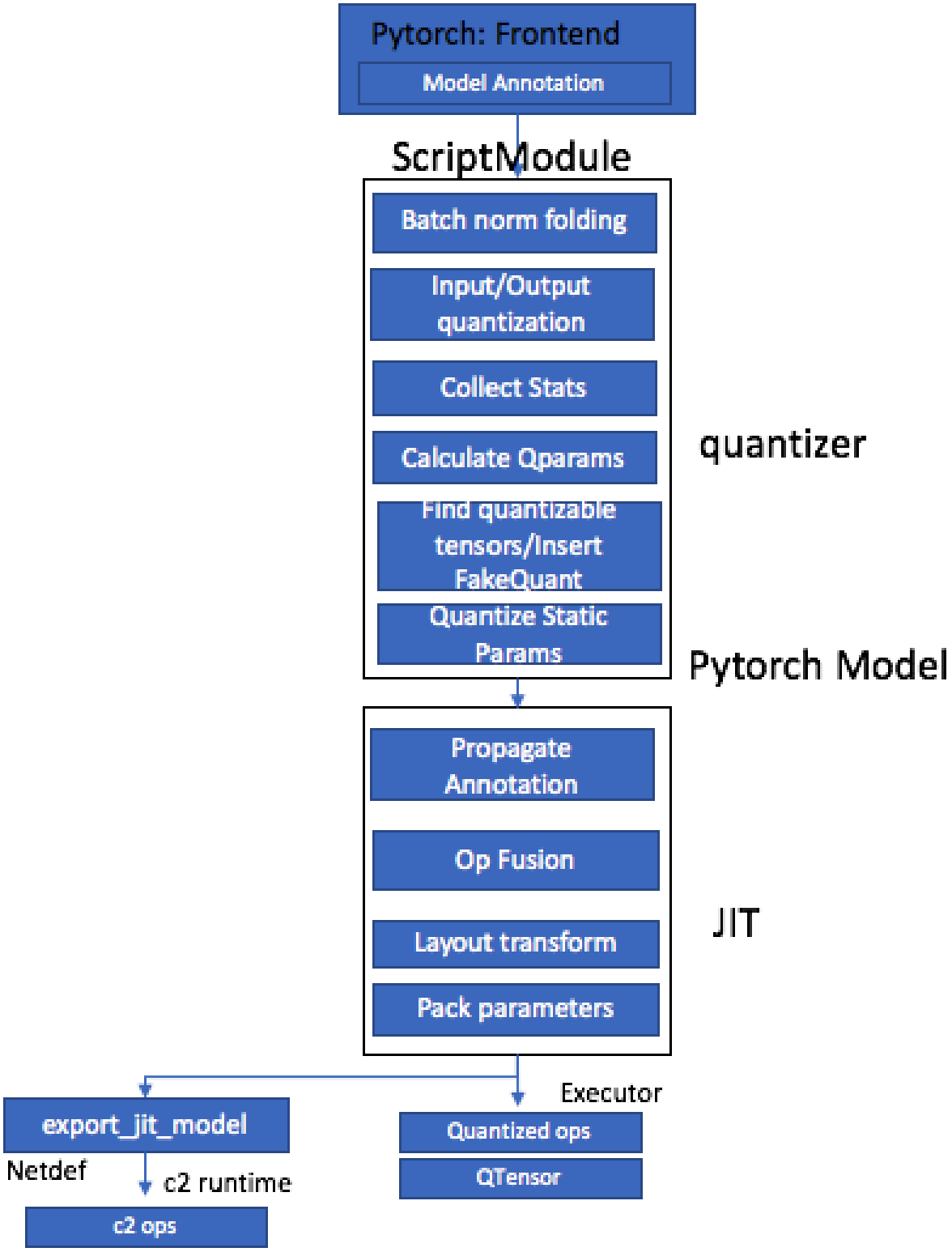
Model Quantization for Pytorch | Cleaned up for GitHub

TL;DR:
We review a proposal for graph mode quantization in pytorch (model_quantizer) that provides end to end post training quantization support for both mobile and server backends. Model quantization supports fp32 and int8 precisions as a starting point and will expand to support other precision types based on customer needs.

The figure below provides an overview of model quantization. In the rest of the document, we will discuss the end to end flow with an example



Design Choices:

One of the key design choices for model quantization is determining the ordering of graph transformations for quantization and determining the split between explicit model quantization and compilation. The split between the

converter and the compiler is determined based on the following principles:

1. Compiler performs transformations that do not change the numerics
2. The design does not assume a fixed set of fused kernels. The kernels can be generated dynamically (say using TVM) and the compiler is the source of truth for operator fusion and quantized operator support.
3. The interface between the converter and the jit compiler is chosen to support inter-operability: This means that the model format is independent of op fusion/layout/packing requirements. Note that for mobile deployment this is not possible and we address this separately.

We anticipate quantization support to evolve over time and our design is intended to be robust:

- Any op that does not have quantization support will be compiled with warnings and will execute, but with poorer performance.
- Fake-quant op insertion by the converter will always be honored by the compiler and an error will be returned if the contract is violated.
- We propose having a QuantLinting tool that will provide useful debug info on
    - Parts of the graph that were not compiled to quantized ops, even though this was requested by the user.
    - Incorrect fake quant annotations specified by the user.

Model Annotation (WIP)

We aim to provide simple annotation tools to users so that they can easily experiment with quantizing different parts of the model and try out different precision choices easily.

Towards this goal, we allow for out of band specification of per module quantization information. It is more desirable to allow for specifying quantization choices during model creation rather than provide it as an argument when we actually perform the transformation.

WITH SEMANTICS FOR QUANTIZATION PARAMETERS

We considered using `with` statement. `with` captures scoping, that is, everything within the scope of `with` should have same quantization parameter. The module will have different quantization parameters depending on the actual `with` statement it is in. This is convenient when we want to use the same submodule but different quantization parameters to construct a new module. In the scoping semantics, in case we call some function we need to propagate the same quantization parameters into these functions, which we can not do. Another drawback for `with` is the semantics in eager mode and script mode does not match, in eager mode, `with` will be ignored, and in script mode it will be used in a later stage for transformation. Introducing this inconsistency is not a good idea for the future decisions we might make and make `with` work with existing or future constructs of the language might be even more challenging. Therefore we decide not to pursue this path.

MODULE SEMANTICS FOR QUANTIZATION PARAMETERS

An alternative is consider quantization parameters being a module level concept, this is more stable and natural than `with` since module is a first class object. Suppose we have a module1 with quantization parameter p1 and module2 with quantization parameter p2 and module1 invokes module2, module1 should be quantized with p1 and module2 should be quantized with p2.

Since quantization parameter is a property of module, we could pass in a dictionary to quantizer specifying quantization parameters for each module, however, this keeps the quantization parameters and the actual code that defines the model apart. In order to have a more user-friendly API, we can put the quantization parameters inside

the `__init__` function and the quantizer function can construct the dictionary for user by traversing through the submodules. We also keep the dictionary API since we might not have direct access to the source code that defines the module.

```
class SubModule(nn.Module):
    def __init__(self):
        self.conv2=nn.Conv(3,3)
        # Quantization parameters for SubModule
        self.__quant_params__ = QuantParams(weights=[8,PER_CHANNEL_QUANT], activa

    def forward(self, y):
        y=self.conv2(y)
        y=nn.functional.relu(y)
        return y

class MyModule(nn.Module):

    def __init__(self):
        self.conv1=nn.Conv(3,3,3)
        self.submodule=SubModule()
        self.submodule2 = SubModule()
        # Quantization parameters for MyModule
        self.__quant_params__ = QuantParams(weights=[8,PER_CHANNEL_QUANT], bias=.

    def forward(self,x):
        y = self.conv1(x)
        y = nn.functional.relu(y)
        # Submodule is required in order to specify quantization subset
        # Settings apply to both weights and activations within a module
        # This is more powerful than pure eager mode quantization as it
        # applies also to functional calls (like nn.functional.relu) that
        # don't have module identity
        y = self.submodule(y)
        return y

        # These annotations should be captured by IR to provide the
        # desired precision and quantizer type for each tensor within scope
        # Annotation parameters are captured from with statement

# If we don't have extra quantization parameters to specify
m = quantizer(MyModule())

# Or:
# We still keep this API in case we don't have access to the
# source code of module
m = quantizer(MyModule(), {
  "": QuantParams(weights=[8,PER_CHANNEL_QUANT], bias=...),
  # 8 bit precisions for weights, with per-channel quant
  # 8 bit precisions for bias with per-layer quant
  #16 bit precisions for activations, with per-layer quant
  "submodule": QuantParams(weights=[8,PER_CHANNEL_QUANT], activations=[16,PER_LAY
})
```

Model quantization passes

PROPOSED USER FACING API

```
my_script_module = torch.jit.trace(MyModule,input_data) # or use ScriptModule
script_module = quantizer(my_script_module, qtz_config, eval_fn, data_iterator)


# For server
# Save representation containing fake-quant ops,
# prior to JIT compilation to file. saves to model.pt file.
# Also need option to save to netdef
script_module.save('mymodule.pt')

result = script_module.forward(test_data)
# For mobile:
# Need to do JIT compilation prior to export.
# Output format is netdef.
torch.utils.mobile_export(script_module)
```

PROPOSED HIGH-LEVEL DESIGN FOR `QUANTIZER` FUNCTION

```
# Private member of quantizer class
def _transformPreparation(scriptModule, qtz_config):
    # Script Module should also capture which parts of the module
    # require quantization
    # Constant fold batch norm layers in floating point module
    FoldBatchNormLayers(script_module)
    # Constant fold input/output de-quant ops
    FoldDeQuantOps(script_module)
    # Insert stats collection function at desired locations. This may not be needed
    # for dynamic quantization but we will refine it later
    stats_script_module = InsertStatsOps(script_module)
    return StatsScriptModule

# Public API for quantization
def quantizer(script_module, qtz_config, eval_fn, data_iterator):
    stats_script_module = _transformPreparation(script_module)
    # Call the eval function - until data_iterator is exhausted
    eval_fn(stats_script_module, data_iterator)

    # Extract statistics collected
    tensor_stats = extractStatistics(stats_script_module)
    # Calculate Q params
    qparams = CalcQParams(tensor_stats, qtz_config)

    # Transformation Pass (Quantizer - corresponding to steps 1 and 2)
    script_module = InsertFakeQuantOps(script_module, qparams, qtz_config)

    # Quantize weights, biases to quantized tensors.
    script_module = QuantizeConstants(script_module,qtz_config)
    return script_module
```

Most of the model transformations described in this design document will be implemented as JIT compiler passes. Examples of such passes are quant-linting pass, propagation of qparams, constant-folding etc. It's important to note that all compiler passes statelessly operate on IR, which contains all the required information, and are generally agnostic to what passes have been run before (if any). Such design allows to decompose a bigger problem (e.g. perform quantization) into a set of simple independent subproblems (e.g. expand FakeQuant nodes, fold constants, perform fusion, etc.).

As a side benefit, this design also provides a fully-manual way for user to perform quantization: a user can insert FakeQuant nodes manually and skipping our implementation of InsertFakeQuantNodes pass - all other passes will handle this case without any change and no effort from our side to support such use-case.

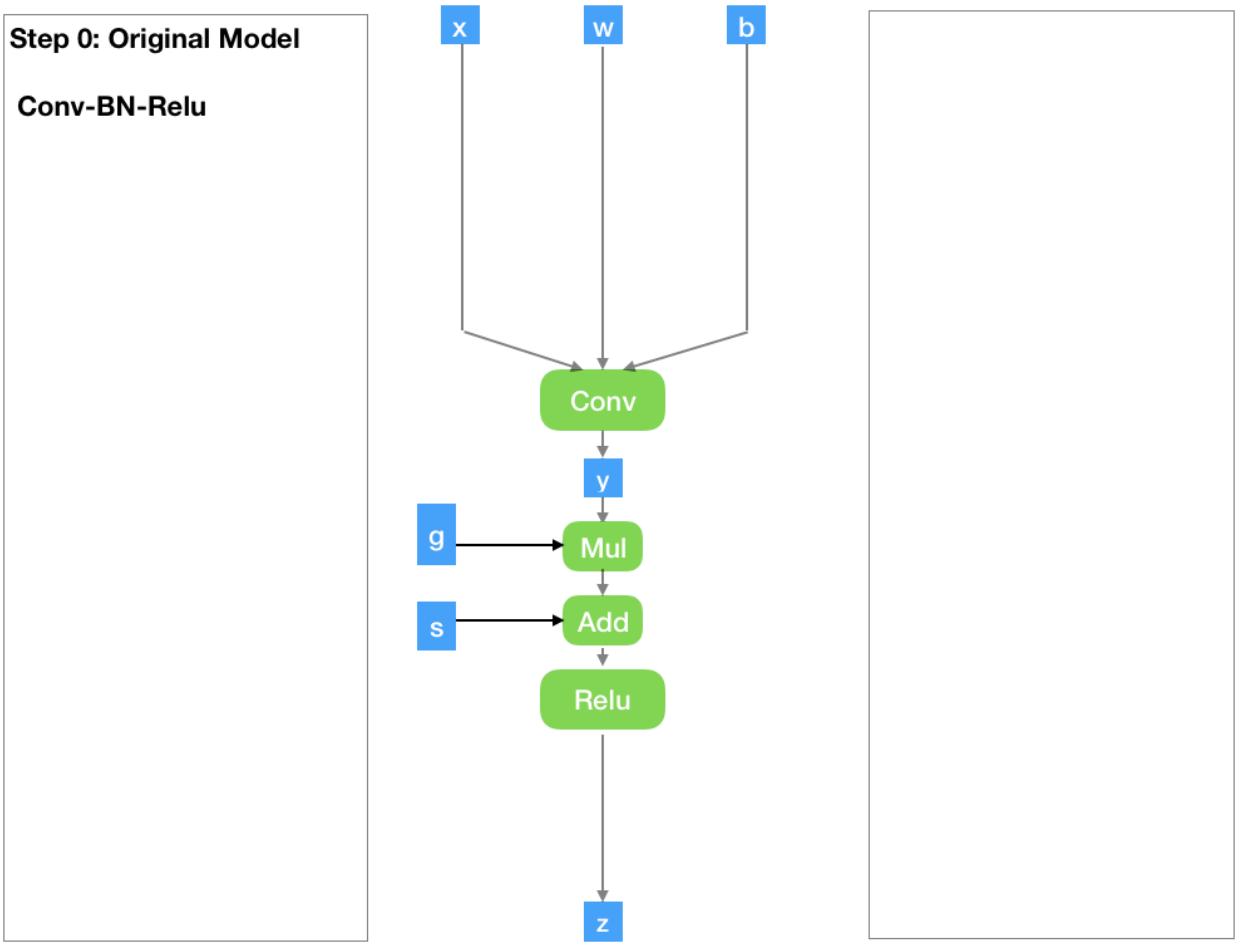
PASSES INVOCATION

As a part of this effort we will also implement a user facing API for invoking passes or a group of passes and a necessary infrastructure for testing and debugging.

The user facing API will be used, for instance, for invocation of `InsertObservers` and `InsertFakeQuantNodes` passes - those will be called by user individually and explicitly. Semantic preserving passes, such as `Fusion`, will be invoked along with other optimizations at some later point, and user will not necessarily call them explicitly. For example, they can be called on export or on load of the model, but we are also considering an option for enabling explicit invocation of the entire optimization pipeline. We also plan to provide a way for users to entirely or selectively disable compiler optimizations - similar to how in traditional compilers a user can choose `Debug` or `Release` mode.

The testing and debugging infrastructure will allow invocation of an individual pass, printing its output and verifying the correctness (similar to `opt` tool in LLVM toolchain). Most of the passes are expected to not change numerical result of the model, except when they are explicitly used for the opposite. E.g. `InsertFakeQuant` pass is explicitly changing semantics of the model. Compiler will guarantee that those “semantic-preserving” passes actually keep the numerical results the same, and we don't expect users to verify that.

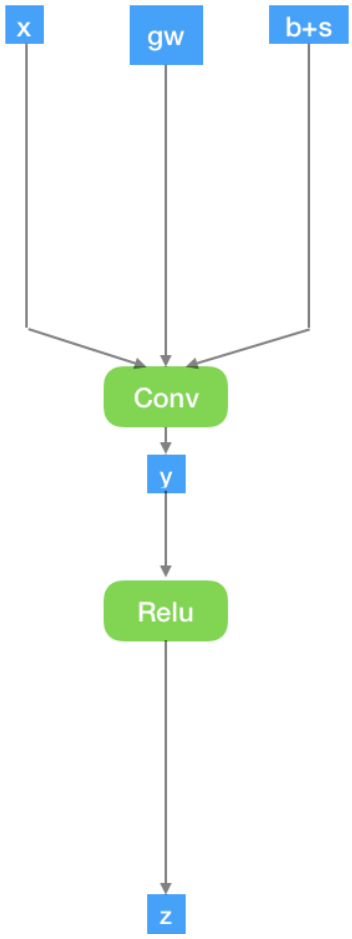
We start with a simple model, after conversion to a script module:



The passes done by the quantizer are:

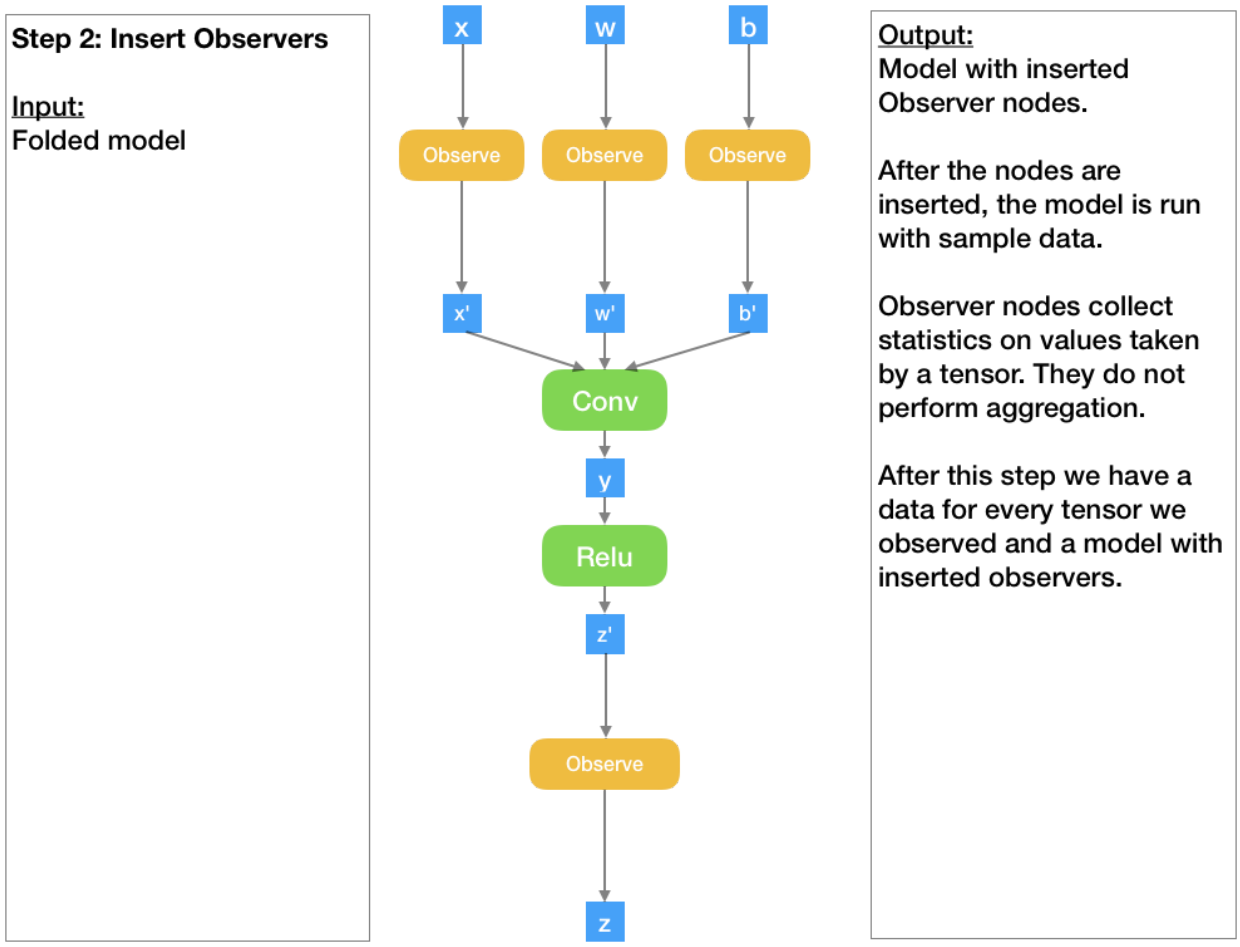
Batch norm constant folding:
Fold constants into weights and biases of conv/fc layers. This is needed to ensure that quantization is performed on the folded constants.

Step 1: Fold Batch Norm
Input:
Original model (IR).



Output:
Model with inserted constants folded into static parameters (weights and biases)

InsertObservers
The pass will go through the graph and insert an observer node for every input/output tensor. We might choose not to insert observers for some nodes if we believe we won't be quantizing them (this will be based on some heuristics - e.g. we might want to skip an output of `Conv` it's only used as an argument of a subsequent `Relu`).



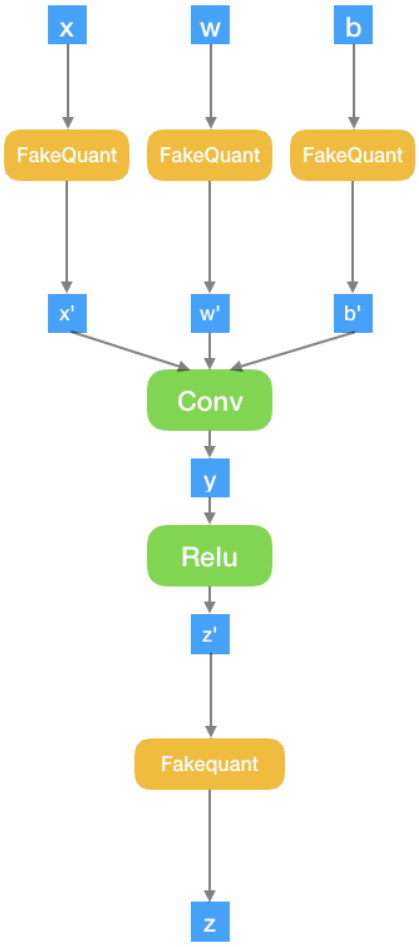
After this step, we calculate the quantizer params based on the observer statistics.

InsertFakeQuantNodes

The pass will go through the graph and according to some heuristics insert FakeQuant nodes to the graph. These nodes will designate tensors we will be quantizing. This pass will take an additional input: some sort of a map from a tensor to its qparams.

Step 3: Insert FakeQ nodes

Input:
1. Model
2. Stats from observer



Output:
Statistics collected by the observer nodes are used to calculate quantization parameters.

FakeQuant nodes are inserted into the model, observer nodes are removed.

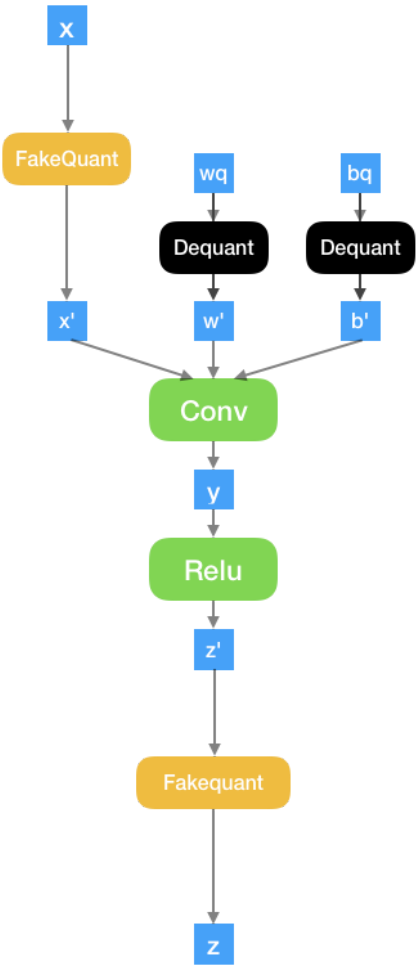
FakeQ-nodes take a tensor and quantization params as input and return a floating point tensor modeling the loss in precision due to quantization.

Quantize static parameters:

Constants followed by quant nodes are quantized. This is the last step that is performed explicitly. At this point the model can be serialized with quantized weights. The rest of the quantization occurs as JIT compiler passes

Step 4: Quantize static parameters

Input:
1. Model with fake quant nodes



Output:
Model with quantized weights

This is the final step of quantization as seen from a user point-of-view. The rest happens in graph-optimizer under the hood.

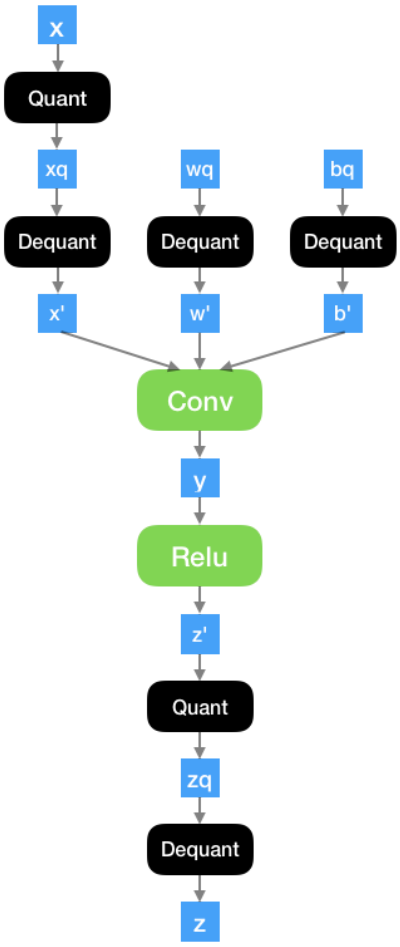
Note: graph-optimizer does not perform any transformations that change semantics (changing accuracy is changing semantics).

JIT passes:

Expanding FakeQuant Nodes

Step 5: GraphOpt/ Expanding

Input:
Model with FakeQ nodes and quantized parameters



Output:
FakeQ nodes are expanded into Quant and Dequant nodes.

The model is ready for graph-optimizations.

A pass will go through all nodes in the graph and replace FakeQuant node with its body:

```
Y  = FakeQuant(X, QParam)
              ↓
Xq = Quant(X, QParam)
Y  = Dequant(Xq, QParam)
```

Note: this pass might not be needed if we implement FakeQuant op directly or if we decide to insert explicit `Quant-Dequant` pairs from the very beginning.

Propagate QParam Info

The pass will go through all nodes and propagate QParam through nodes that are not supposed change that. An example of such node is a `Split` node - even though we might observe different values on its output, we want to avoid requantizing as we go through `Split` and thus we would replace `QParams` of the outputs with `QParams` of the inputs.

```
X  = FakeQuant(A, QParam_A)
Y  = Split(X, 2)
Z  = FakeQuant(Y, QParam_Y)
              ↓
X  = FakeQuant(A, QParam_A)
Y  = Split(X, 2)
Z  = FakeQuant(Y, QParam_A)
```

Fusion pass

The pass will go through the graph and replace some subgraphs with equivalent but faster nodes.

Example 1:

```
X  = Add(A, B)
Y  = Conv(X, W)
Z  = Relu(Y)
O  = Sigmoid(Z)
```

↓

```
X  = Add(A, B)
Z  = ConvRelu(X, W)
O  = Sigmoid(Z)
```

Example 2:

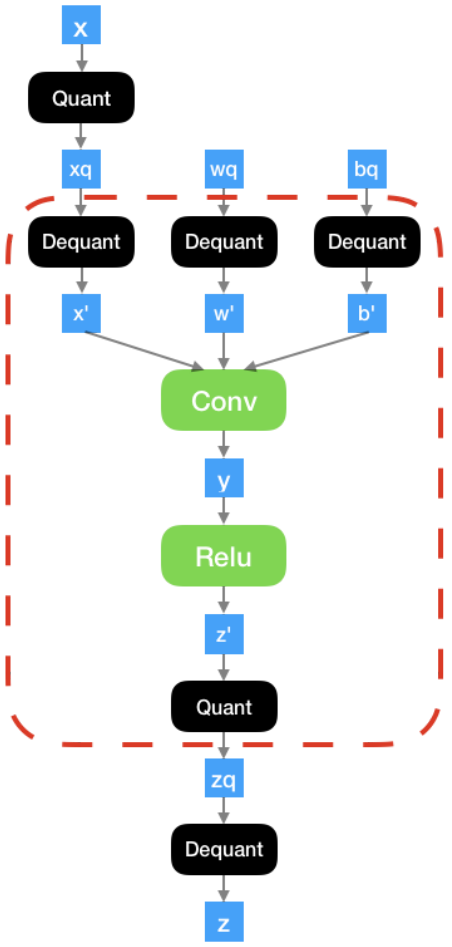
```
Xq = Quant(A, Q1)
X  = Dequant(Xq, Q1)
Y  = Conv(X, W)
Z' = Relu(Y)
Zq = Quant(Z, Q2)
Z  = Dequant(Zq, Q2)
```

↓

```
Xq = Quant(A, Q1)
Zq = QuantConvRelu(Xq, Q1, Q2)
Z  = Dequant(Zq, Q2)
```

Step 6: GraphOpt/Fusion

Input:
Model with FakeQ nodes expanded.

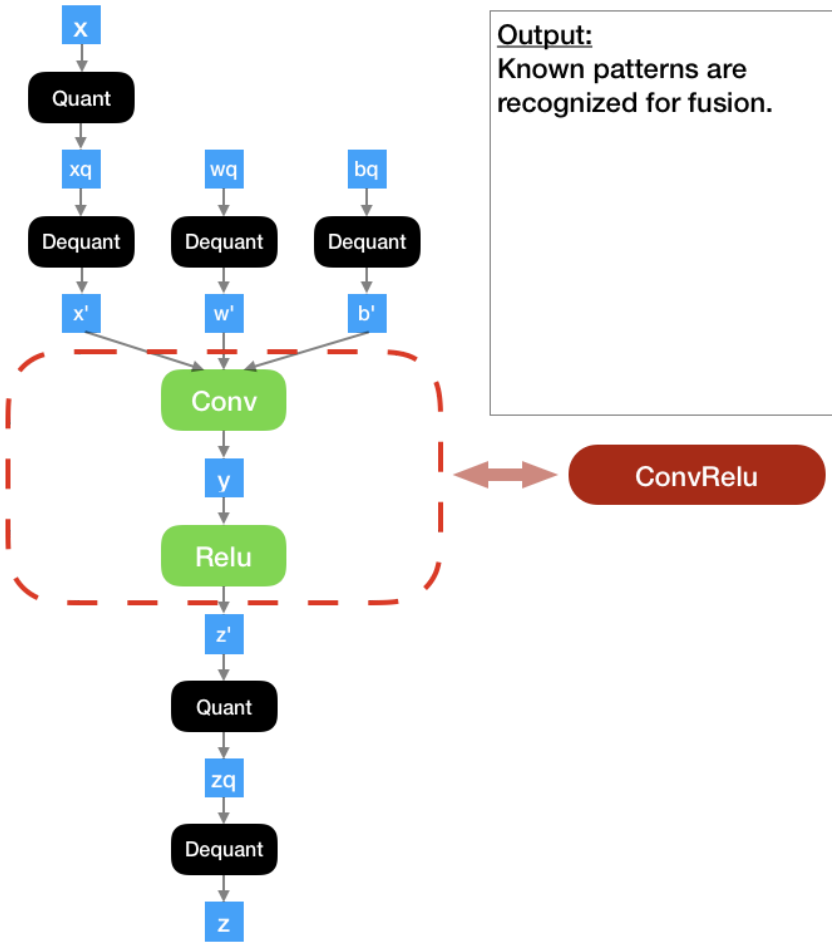


Output:
Known patterns are recognized for fusion.

QuantConvRelu

Step 6: GraphOpt/Fusion (if QuantConvRelu isn't supported)

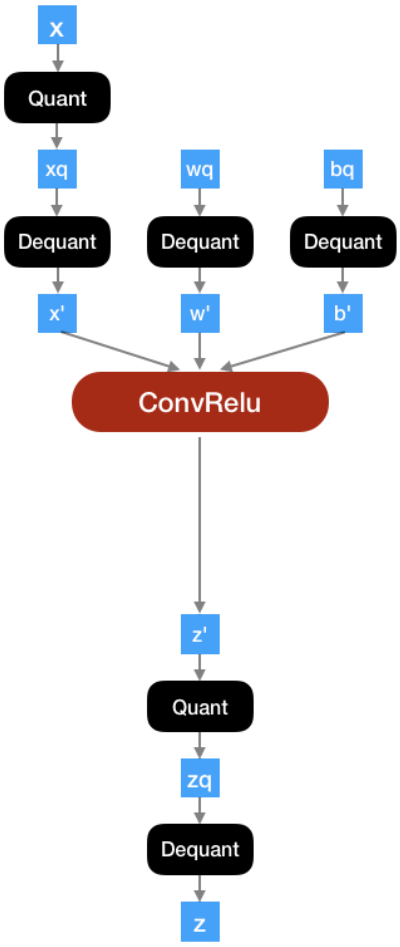
Input:
Model with FakeQ nodes expanded.



Output:
Known patterns are recognized for fusion.

Step 6: GraphOpt/Fusion

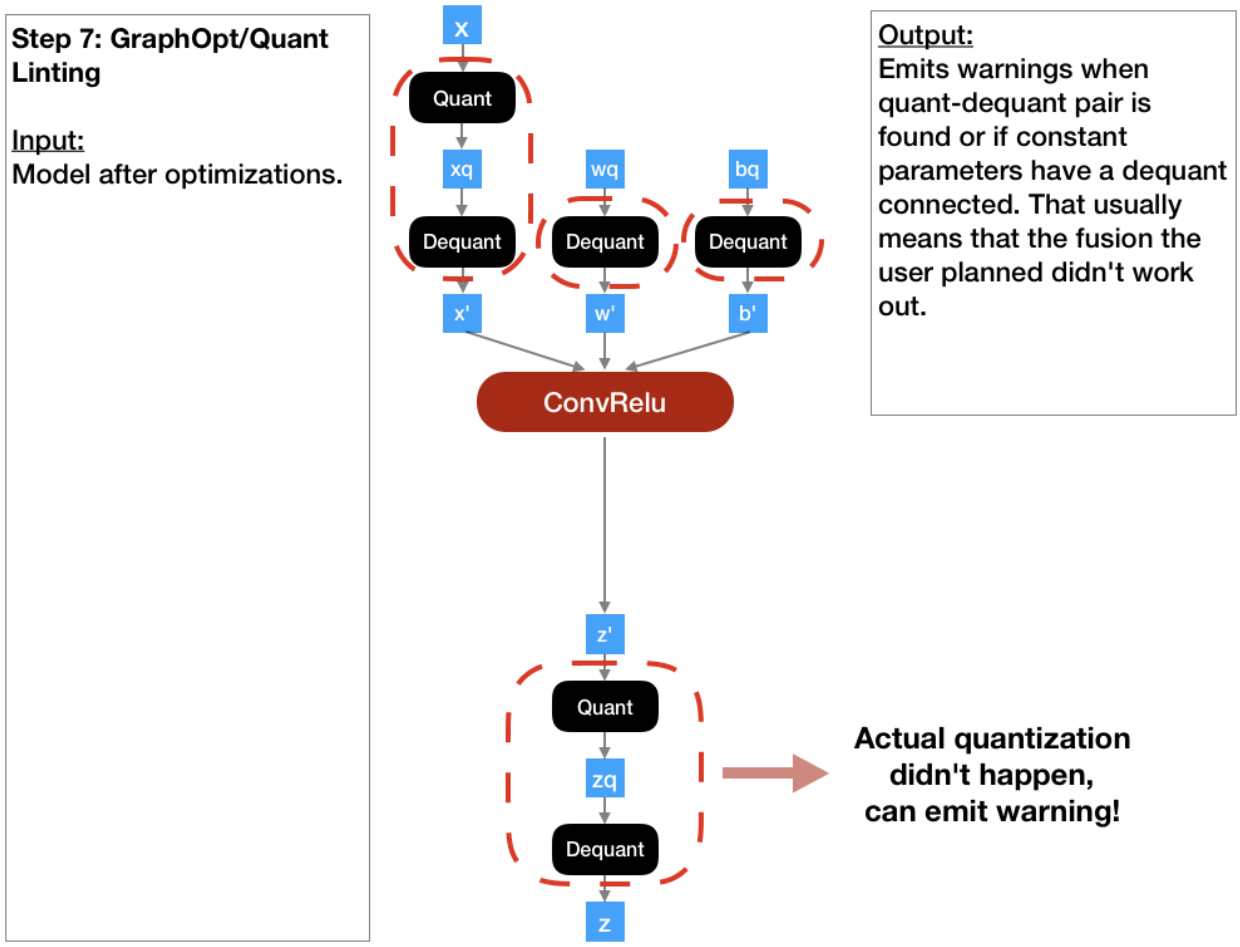
Input:
Model with FakeQ nodes expanded.



Output:
Known patterns are fused into a single fused node.

QuantLinting pass

The pass will go through all nodes in the graph and report any remaining `Quant`- `Dequant` pairs. This will be considered as an indication that quantization did not go as planned - e.g. because an expected fusion did not happen for some reason. The pass will emit warning to notify user that the quantization strategy they use does not work for a specific part of the graph. They can then either disable quantization for that part or perform it manually.



IMPLEMENTATION OF PASSES

Passes will be implemented on top of PyTorch JIT IR, which provides a basic API for inspecting and mutating graphs. We consider adding Python bindings for this API to enable quick prototyping, but in the long-term we expect passes to be implemented in C++ similarly to already existing ones.

QTensor and Quantized ops

The result of the compiler pass is a series of fused ops in C++ that is executed by the interpreter. The fundamental building block for representing quantized computation is a QTensor, which is currently being designed.

A critical pole in the entire effort is implementing quantized ops in pytorch. These ops need to be implemented in c10 and a design goal is to ensure identical op numerics and semantics across pytorch and c2. In addition, for mobile deployment, c2 ops for mobile need to be numerically consistent with quantized pytorch ops in the server. We plan to leverage support from the share Asyc team to help with porting of ops to pytorch.

We also plan to support benchmarking of quantized model implementations in pytorch with c2 deployments to ensure that pt models are competitive in performance.

Exporting quantized models

For server, storing a scriptmodel is sufficient, and the additional work item is to support serializing qtensors. For mobile, due to constraints on code size, we anticipate storing a model after all jit passes are completed after conversion to netdef.

Eager Mode support:

We plan to support fully eager quantization in pytorch without any jit dependencies. This is intended to be a DIY style approach to quantization where the user manually performs calibration, qparam calculation and module substitution. Eager mode quantization does not provide operator fusion support. For this mode, we plan to implement quantized modules for a few modules to demonstrate the benefits of model quantization and are targeting ResNext 101 classification as an example use case.

Open Items for Further investigation

- Quantization aware training and APIs for that
- How to export to mobile