NVIDIA

DEPLOYING QUANTIZATION-AWARE TRAINED NETWORKS USING TENSORRT

Dheeraj Peri, Jhalak Patel, Josh Park

AGENDA

QUANTIZATION IN NEURAL NETWORKS

Post Training Quantization (PTQ)
Quantization Aware Training (QAT)

DESIGNING QUANTIZED NETWORKS

Train QAT network in Tensorflow
Transforming QAT network to ONNX

ACCELERATE QUANTIZED NETWORKS WITH TENSORRT

Optimize QAT networks with TensorRT
Inference and evaluation

INTRODUCTION

- State of the art neural networks have seen tremendous success on computer vision, natural language processing, robotics tasks.
- With millions of floating-point operations, deployment of AI models in real time is challenging.
- Some of the techniques for making neural networks faster and lighter

1) Architectural improvements

2) Designing new and efficient layers which can replace traditional layers

3) Neural network pruning which removes unimportant weights

4) Software and hardware optimizations

5) Quantization techniques

QUANTIZATION IN NEURAL NETWORKS

- Quantization is the process of converting continuous values to discrete set of values using linear/non-linear scaling techniques.
- Dequantized FP32 tensors should not deviate too much from the pre-quantized FP32 tensor.
- Quantization parameters are essential for minimizing information loss when converting from higher precision to lower precision values.

0.34	3.75	5.64
1.12	2.7	-0.9
-4.7	0.68	1.43

FP32
(pre-quantized)

quantize →

64	134	217
76	119	21
3	81	99

INT8
(quantized)

dequantize →

0.41	3.62	5.29
1.3	2.8	-0.92
-4.5	0.71	1.39

FP32
(dequantized)

QUANTIZATION SCHEMES

- Floating point tensors can be converted to lower precision tensors using a variety of quantization schemes.

  e.g., R = ***s****(*Q - ***z****)* where R is the real number, Q is the quantized value

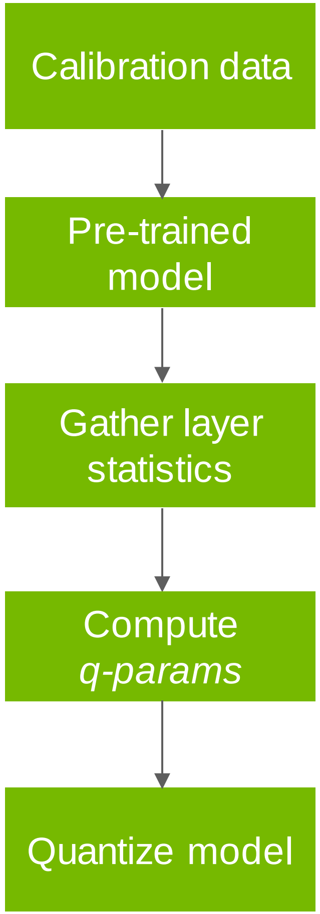
  s and ***z*** are scale and zero point which are the quantization parameters (***q-params***) to be determined.

- For symmetric quantization, zero point is set to 0. This indicates the real value of 0.0 is equivalent to a quantized value of 0.

- ***q-params*** can be determined from either **post training quantization** or **quantization aware training** schemes.

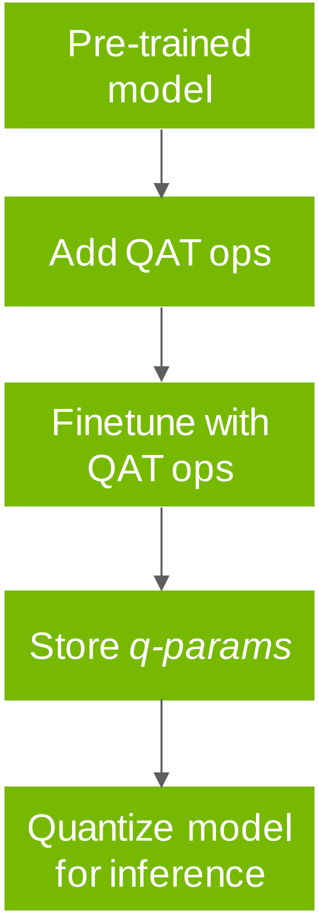
POST TRAINING QUANTIZATION (PTQ)

- Start with a pre-trained model and evaluate it on a calibration dataset.
- Calibration data is used to calibrate the model. It can be a subset of training data.
- Calculate dynamic ranges of weights and activations in the network to compute quantization parameters (*q-params*).
- Quantize the network using *q-params* and run inference.



QUANTIZATION AWARE TRAINING (QAT)

- Start with a pre-trained model and introduce quantization ops at various layers.
- Finetune it for a small number of epochs.
- Simulates the quantization process that occurs during inference.
- The goal is to learn the *q-params* which can help to reduce the accuracy drop between the quantized model and pre-trained model.

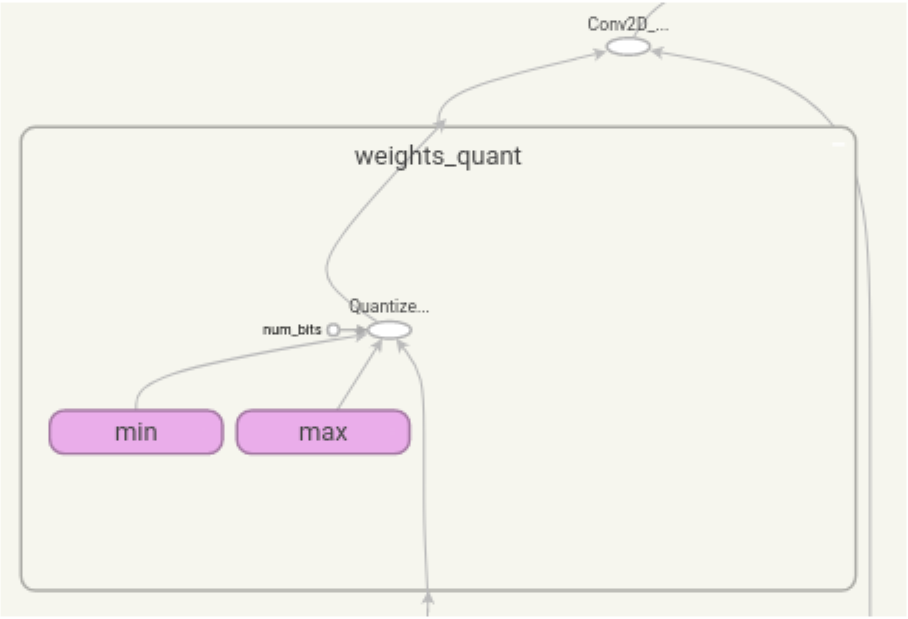


PTQ VS QAT

PTQ	QAT
Usually fast	Slow
No re-training of the model	Model needs to be trained/finetuned
Plug and play of quantization schemes	Plug and play of quantization schemes (requires re-training)
Less control over final accuracy of the model	More control over final accuracy since *q-params* are learned during training.

QAT IN TENSORFLOW

- TF has a quantization API which automatically adds quantization ops to a given graph.
- ***tf.contrib.quantize.create_training_graph()***
  tf.contrib.quantize.create_eval_graph()
- Provides tools to rewrite the original graph and adds quantization ops for weights and activations.
- Additional arguments need to be provided for configuring the type of quantization.
- We use ***tf.quantization.quantize_and_dequantize (QDQ)*** operation for symmetric quantization.
  Output = round(input *scale) * inverse_scale



TOOLKIT

- Deep Learning examples toolkit open sourced by NVIDIA.
- NGC container support with latest features from different frameworks.
- End-End Workflow for deploying Resnet-50 with QAT in TensorRT

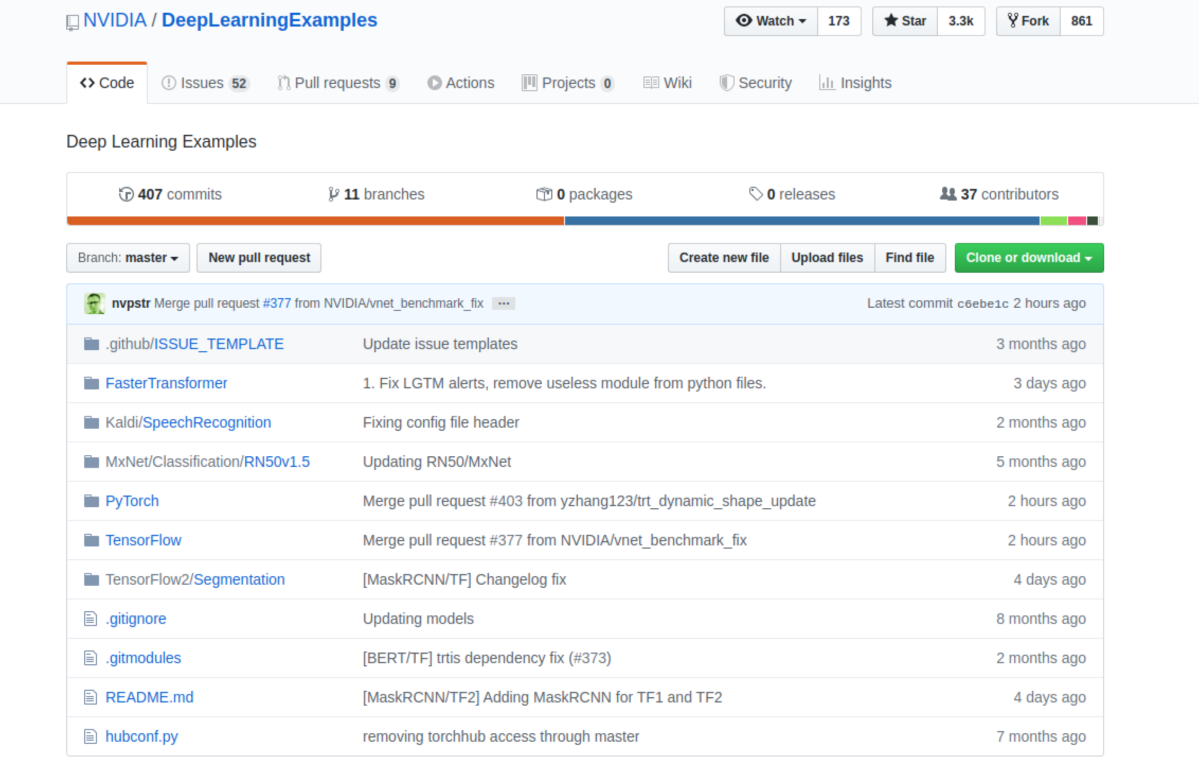
  1) Finetuning RN-50 QAT

  2) Post processing

  3) Exporting frozen graph

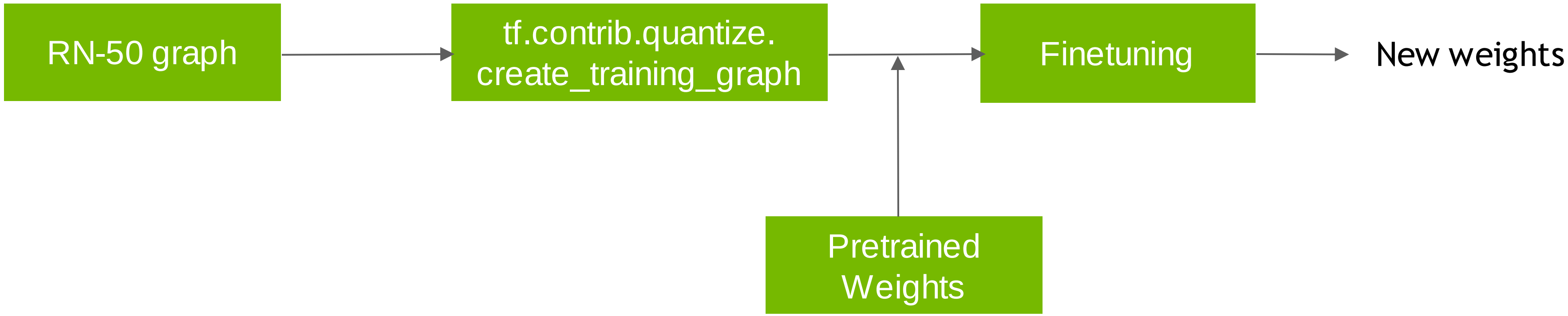
  4) TF2ONNX conversion

  5) TensorRT Inference



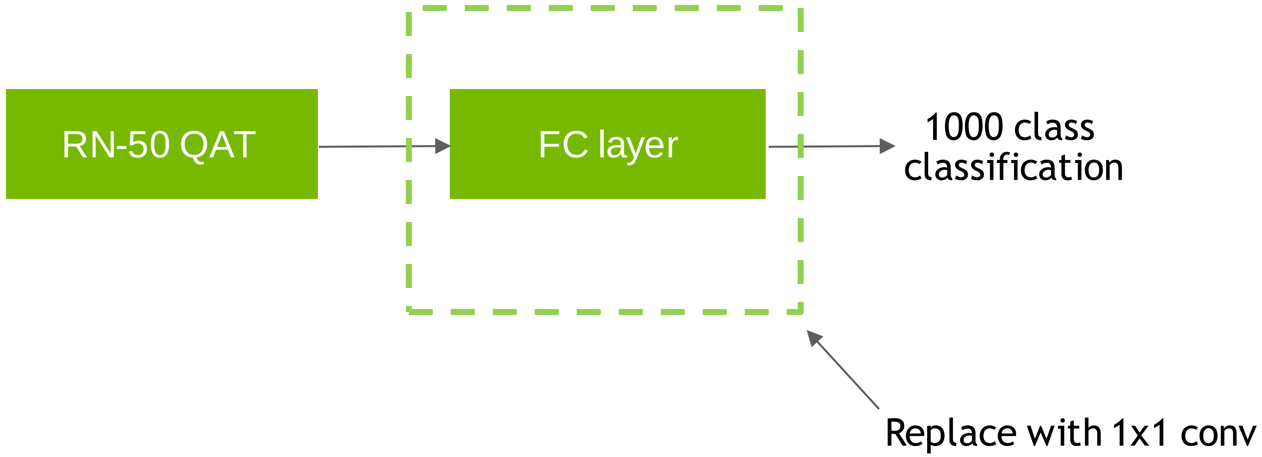
STEP 1: FINETUNING RN50 WITH QAT

- tf.contrib.quantize.create_training_graph adds quantization nodes in the RN50 graph.
- Quantization nodes are added at weights (conv/FC layers) and activation layers in the network.
- Load the pre-trained weights, finetune the QAT model and save the new weights.



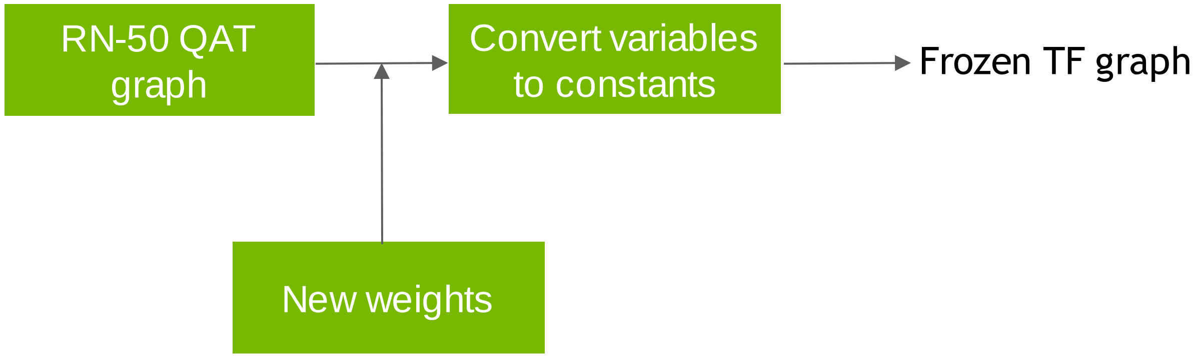
STEP 2: POST PROCESSING

- This step is required to ensure TensorRT builds successfully on RN50 QAT graph.
- After finetuning, convert the final fully connected (FC) layer into a 1x1 convolution layer preserving the same weights.



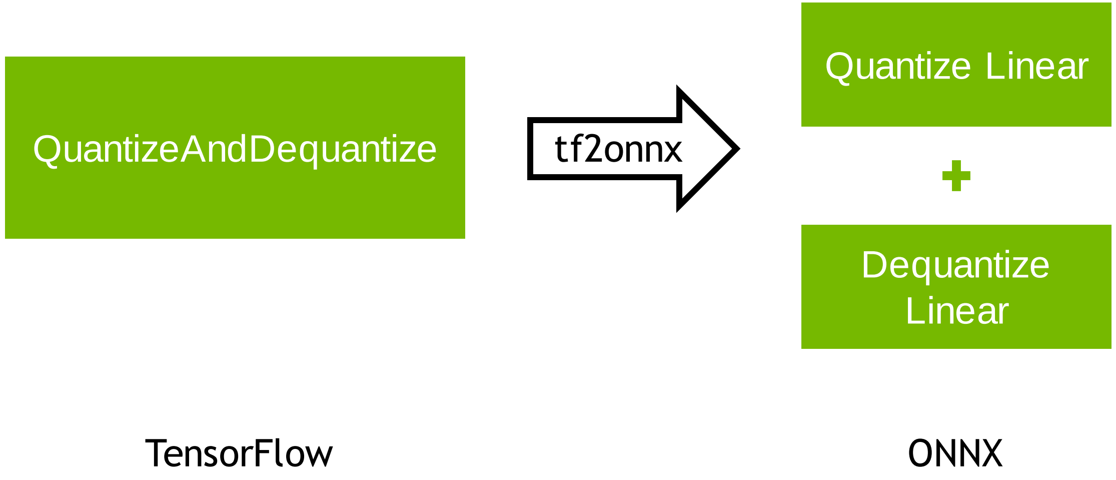
STEP 3: EXPORTING FROZEN GRAPHS

- Generate a frozen graph using the RN-50 QAT graph and the new weights from finetuning stage.
- This step converts the variables in the graph to constants by using the weights in the checkpoints.
- Both data formats (NCHW and NHWC) can be used, although NCHW is recommended for the final graph.



STEP 4: TF2ONNX CONVERSION

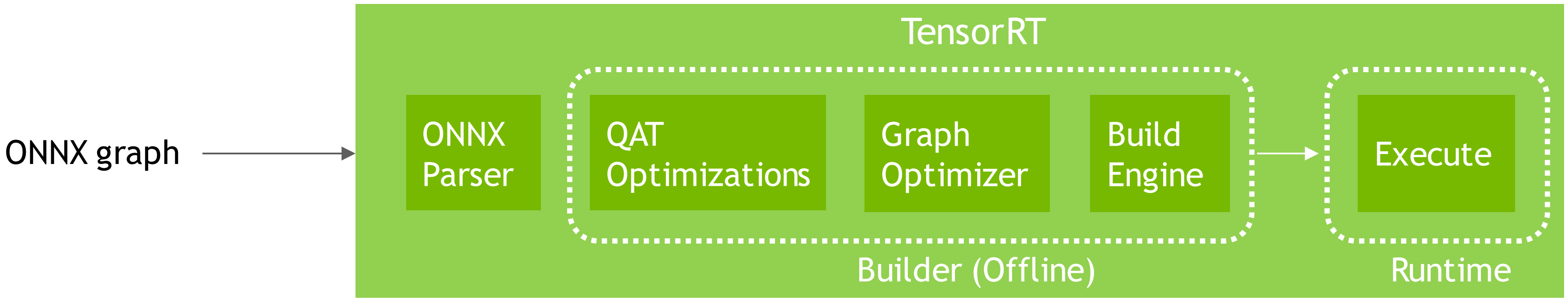
- TF2ONNX converter (https://github.com/onnx/tensorflow-onnx) transforms a TensorFlow pb file to ONNX. It has conversion support for all common deep learning layers.
- Support for QDQ layers in TF2ONNX converter has been added for the following conversion.
- QDQ ops store information about dynamic ranges of the tensors. This is converted as scale and zero-point parameters during ONNX conversion.



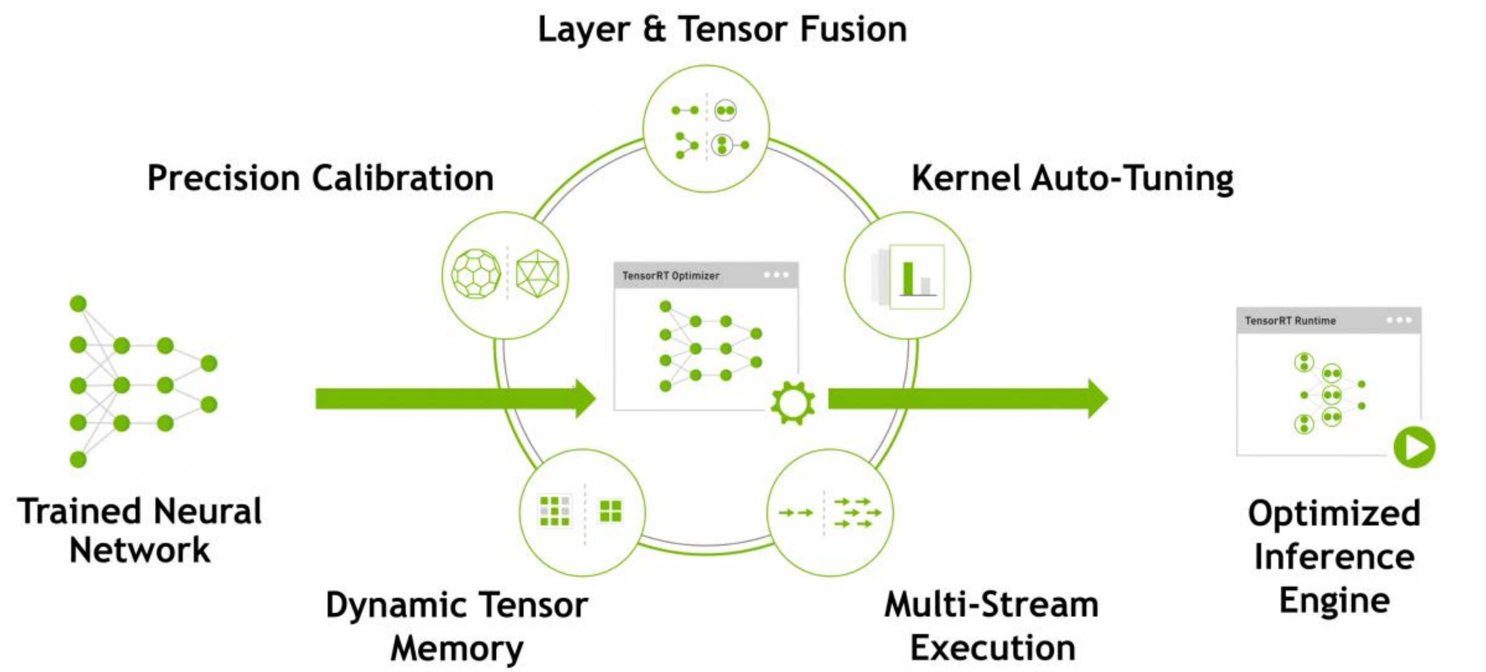
Support for QDQ: https://github.com/jhalakpatel/tensorflow-onnx/tree/fake_quant_ops_rewriter/tf2onnx

STEP 5: TENSORRT INFERENCE

- Generated ONNX graph with QuantizeLinear and DequantizeLinear ops is parsed using ONNX parser available in TensorRT.
- TensorRT performs several optimizations on this graph and builds an optimized engine for the specific GPU.



TENSORRT INFERENCE ACCELERATOR



QUANTIZATION

$$Q(x,\ scale,\ zero\_point) = round(\frac{x}{scale} + zero\_point)$$ - affine

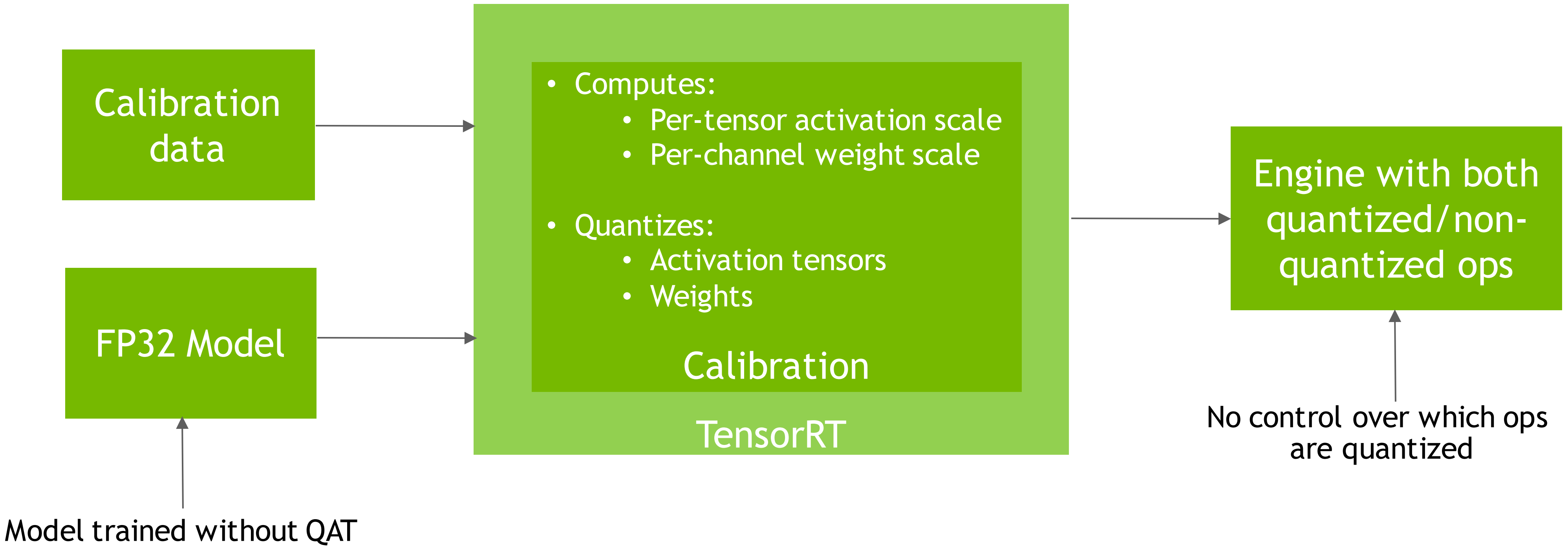
$$Q(x, scale) = round(\frac{x}{scale})$$ - symmetric*

Non-Quantized

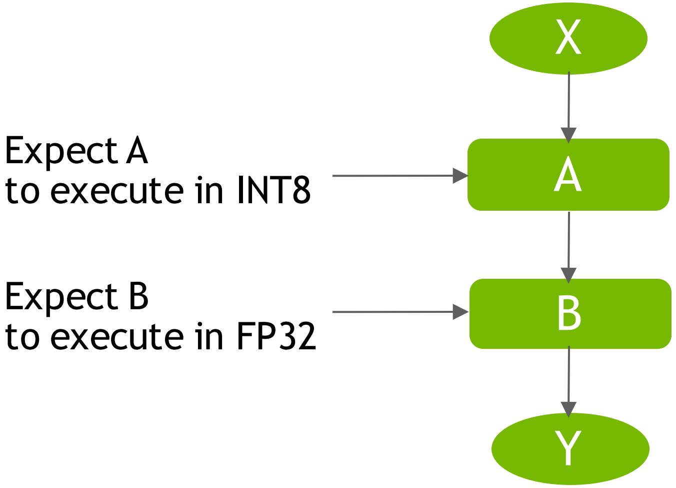
Quantized Op

* TensorRT only supports symmetric quantization

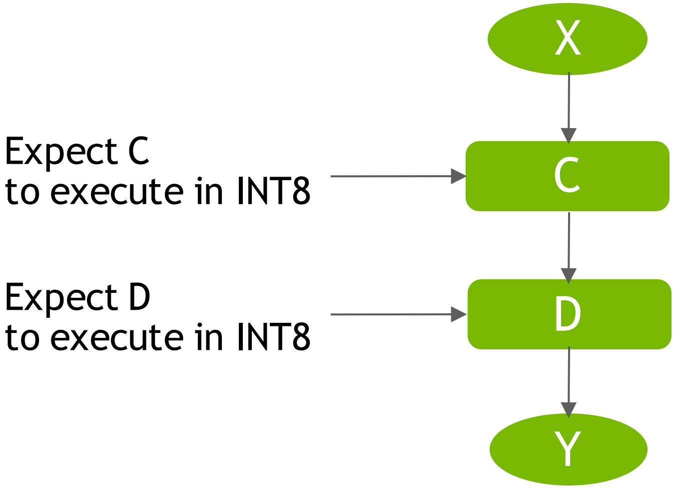
PTQ MODEL INFERENCE



PTQ LIMITATIONS



Quantized GEMM followed by **high** precision activation for **accuracy** eg. LSTM

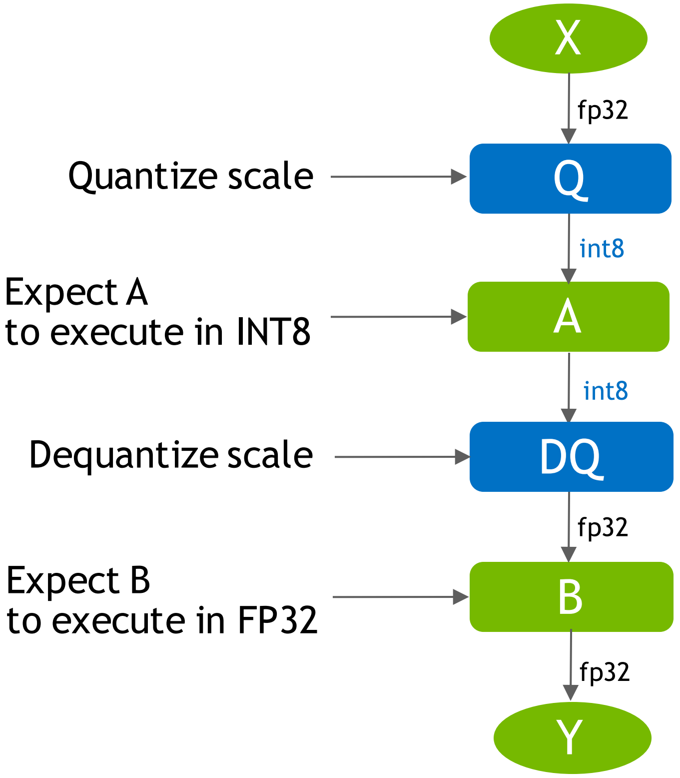


Quantized GEMM followed by **low** precision activation for **speed** eg. Image classification

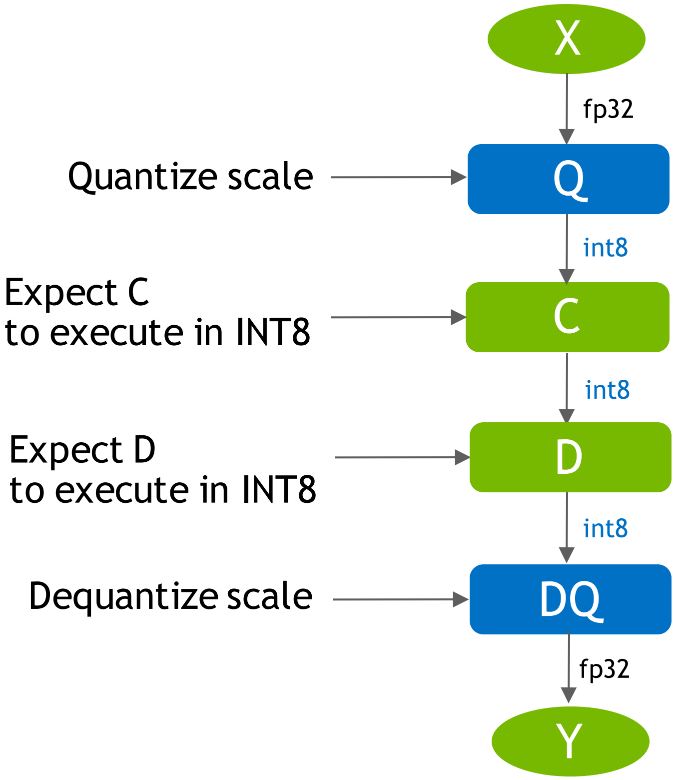
For best results, the network must:

- specify where quantization and dequantization take place.
- learn the best quantization scales .

QAT MODEL INFERENCE



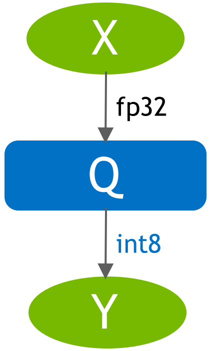
Quantized GEMM followed by **high** precision activation for **accuracy** eg. LSTM



Quantized GEMM followed by **low** precision activation for **speed** eg. Image classification

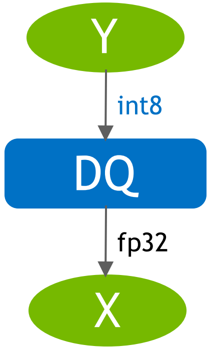
QUANTIZATION OPS

ONNX::QuantizeLinear



$$y = saturate(round(\frac{x}{scale_k} + zero\_point_k)$$

ONNX::DequantizeLinear



$$x = (y - zero\_point_k) * scale_k$$

- Zero point must be 0. Symmetric scaling.
- Per-tensor scaling.
- Per-channel scaling with arbitrary scaling axis (k).

QDQ OPS INSERTIONS: RECOMMENDATION

- Recommend QDQ ops insertion at Inputs of quantizable ops
- Matches QLinear/QConv semantics i.e. low precision input, high precision output.
- No complexity in deciding whether to quantize output or not. Just Don't.
- Let the ops decide what precision input they want.

QDQ OPS INSERTIONS: RECOMMENDATION

- Inserting QDQ ops at inputs (recommended)
  - Makes life easy for frameworks quantization tools
    - No special logic for Conv-BN or Conv-ReLU
    - Just insert QDQ in front of quantizable ops. Leave the rest to the back end (TensorRT).
  - Makes life easy for back end optimizers (TensorRT)
    - Explicit quantization. No implicit rule eg. "Quantize operator input if output is quantized".

- Inserting QDQ ops at outputs (not recommended, but supported)
  - Some frameworks quantization tools have this behavior by default.
  - Sub-optimal performance when network is "partial quantization" i.e. not all ops are quantized.
  - Optimal performance when network is "fully quantized" i.e. all ops in network are quantized.

QDQ OPS INSERTIONS: AT INPUTS

- Some ops require high precision input form QConv/QLinear.
  - Don't insert QDQ at inputs.
  - Eg. LayerNorm (BERT), Sigmoid, TanH (LSTM), Swish (EfficientNet)
- Some ops can handle low precision input without accuracy drop.
  - Insert QDQ at inputs.
  - Eg. GeLU (BERT), Softmax (BERT).

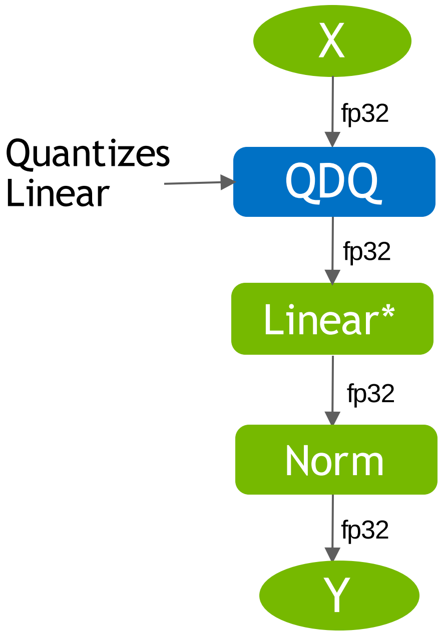
BERT large finetuned for squad v1.1 (91.01 F1 in fp32)

Ops with quantized input	F1
Baseline: Linear, MM, BMM	90.66
BaseLine + GeLU	90.28
BaseLine + LayerNorm after Linear	5.98

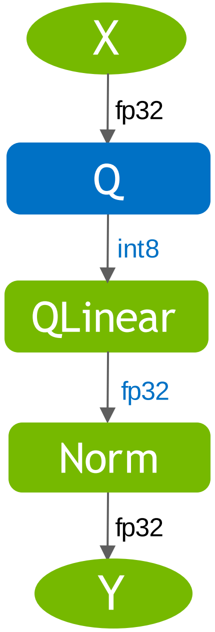
EfficientNet b3 (81.61 top-1 in fp32)

Ops with quantized input	Top-1
Conv	80.28
Conv + Swish	78.37

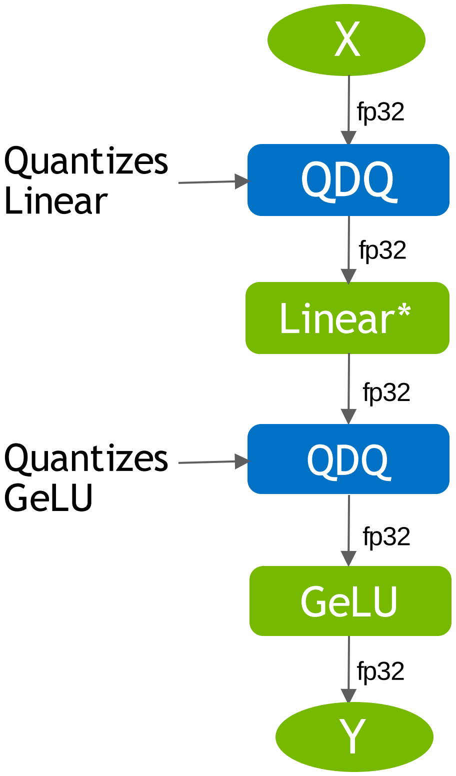
QDQ OPS INSERTIONS: EXAMPLE



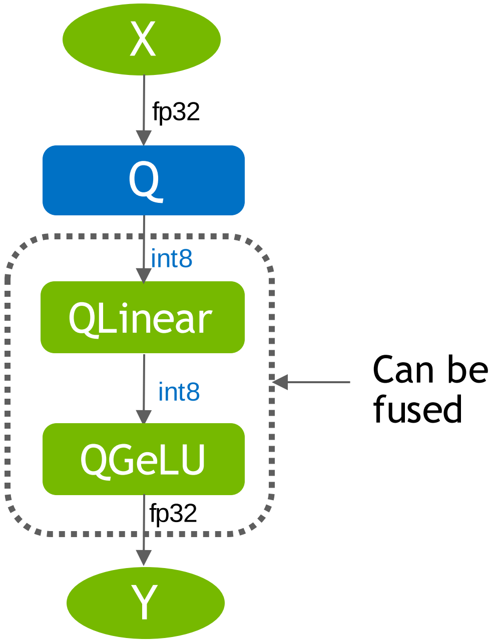
QAT Model



Quantized Inference Model



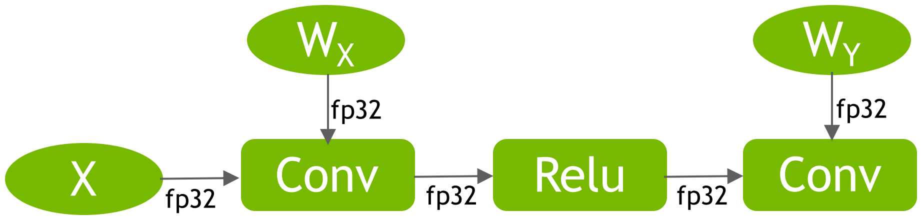
QAT Model



Quantized Inference Model

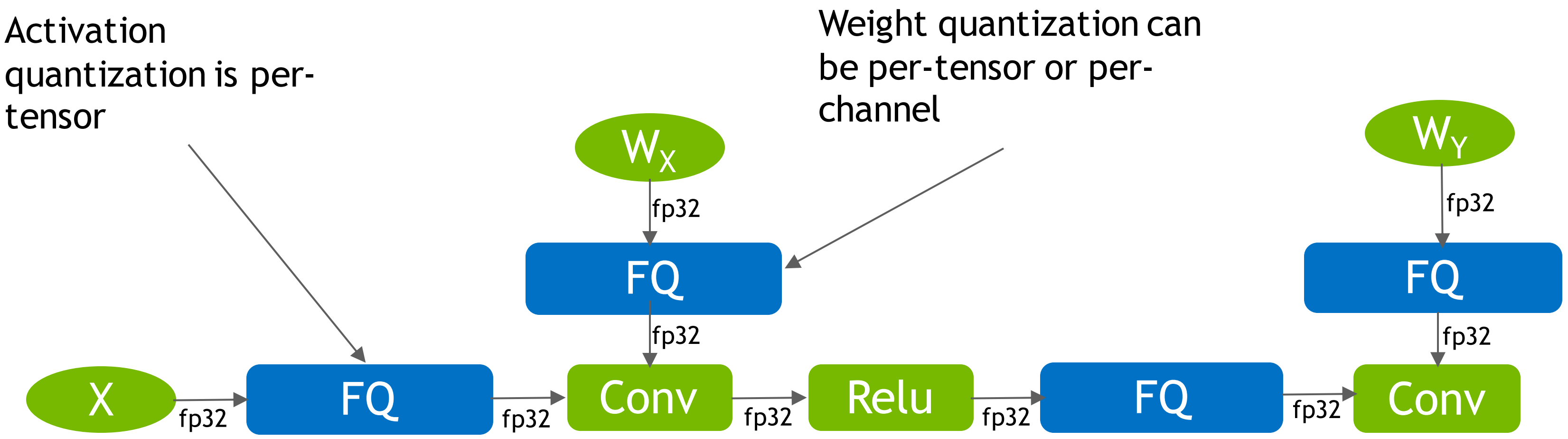
* Omitting weights QDQ for Linear op for simplifying diagram

EXAMPLE: QAT MODEL INFERENCE



Model trained without QAT

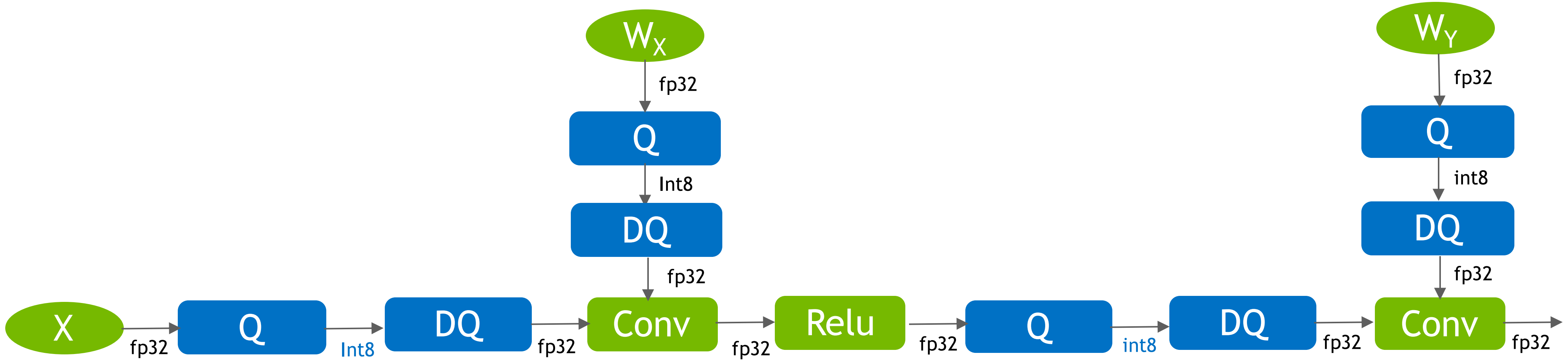
FINE-TUNED TF GRAPH: WITH FAKE QUANT OPS



Fake Quant ops are inserted **before** quantizable ops

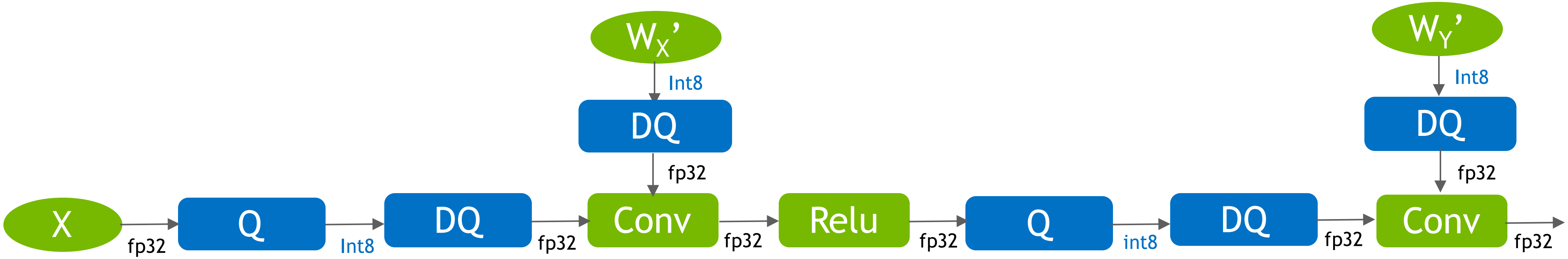
WLOG FQ can be FakeQuant*, QDQV2, QDQV3

FINE-TUNED ONNX GRAPH: WITH QDQ OPS



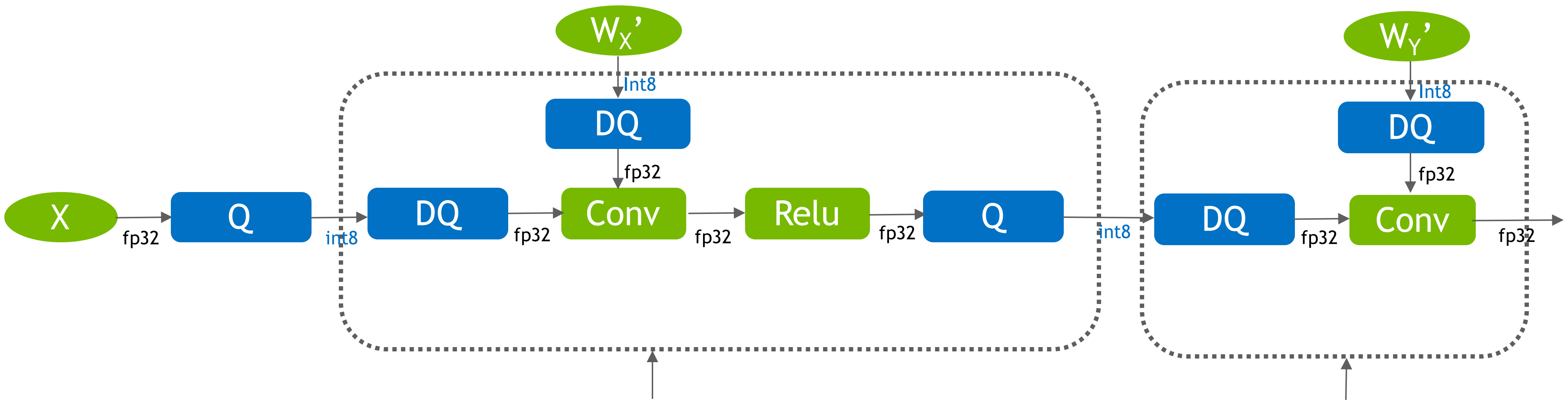
QDQ rewriter in TF2ONNX converter replaces Fake Quant ops with QDQ pairs

QDQ GRAPH OPTIMIZER: FOLD CONSTANTS



Note: QDQ graph optimizer is part of generic TensorRT graph optimizer

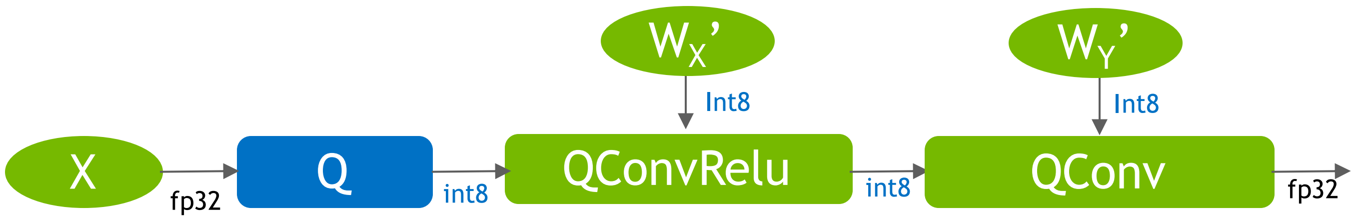
QDQ GRAPH OPTIMIZER: MATCH QUANTIZED OP AND FUSE



We fuse DQ ops with Conv, Conv with Relu, and Q op with ConvRelu to create QConvRelu with INT8 inputs and INT8 output

If there is no Q op available for epilog fusion, this will fuse into QConv with FP32 output

QDQ GRAPH OPTIMIZER: QUANTIZED INFERENCE GRAPH



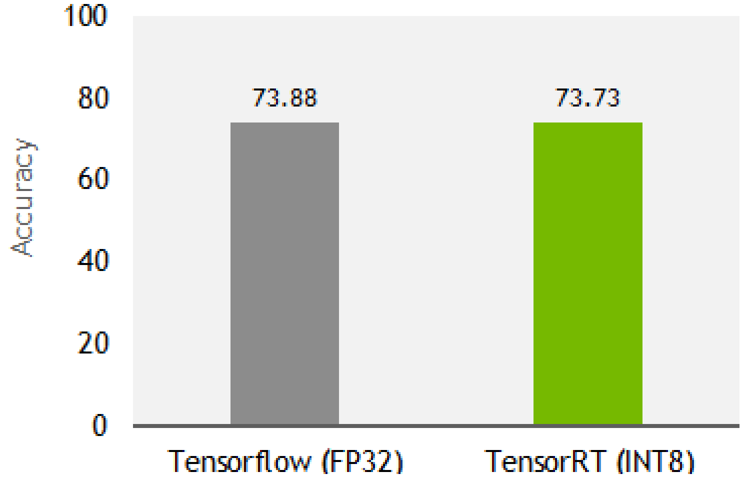
INFERENCE PIPELINE

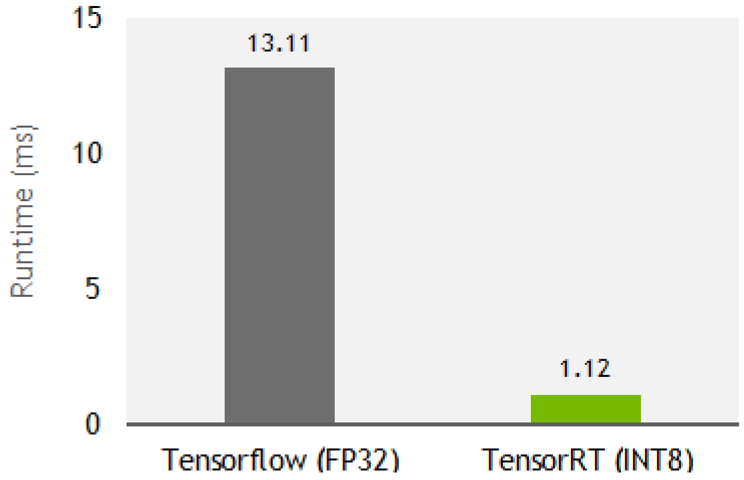
- Create network with ***kEXPLICIT_PRECISION*** flag.
- Set *trt.Builderflag.INT8* to enable INT8 precision.
- Parse Resnet-50 ONNX graph using ONNX parser available in TensorRT and build TensorRT engine.
- Setup the test data pipeline and perform input pre-processing and resizing operations.
- Run the engine on the input data. Copy the outputs of the model back to the host.

```python
"""
Parse the model file through TensorRT, build TRT engine and run inference
"""
# Create builder and network
TRT_LOGGER = trt.Logger(trt.Logger.VERBOSE)
network_flags = 1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)
network_flags = network_flags | (1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_PRECISION))
builder = trt.Builder(TRT_LOGGER)
network = builder.create_network(flags=network_flags)
# Parse the onnx model
with trt.OnnxParser(network, TRT_LOGGER) as parser:
    parser.parse(model.read())
# Set builder config
config = builder.create_builder_config()
config.max_workspace_size = 1 << 30
config.flags = config.flags | 1 << int(trt.BuilderFlag.INT8)
# Build TensorRT engine
engine = builder.build_engine(network, config)
# Inference
with engine.create_execution_context() as context:
    trt_outputs = execute(context, bindings, inputs, outputs, stream, batch_size=1)
```

EVALUATION OF RESNET-50 QAT NETWORK

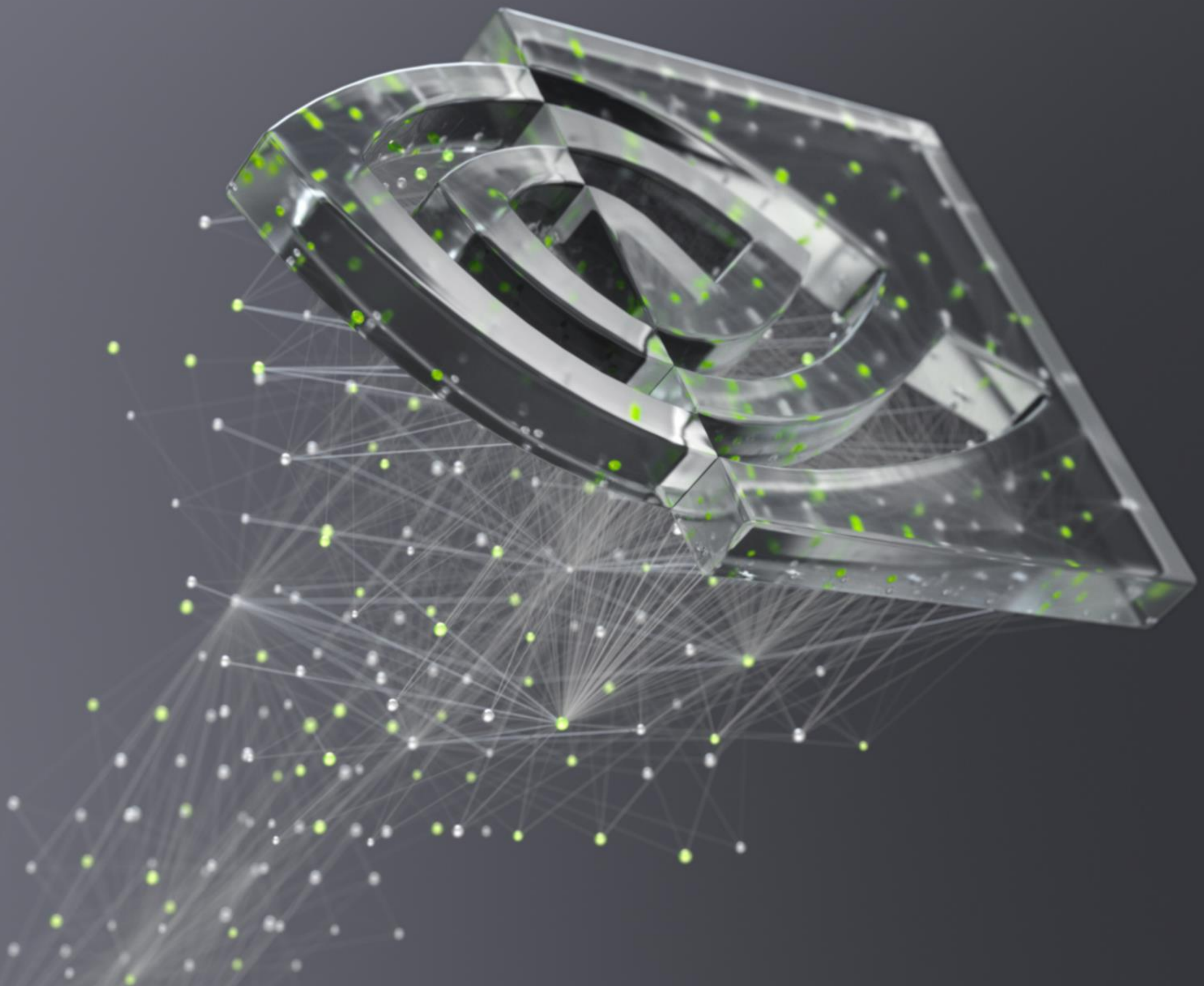
- The evaluation has been performed on RTX 2080 Ti GPU and Tensorflow 1.15.
- TF network is running in FP32 whereas TensorRT inference is in INT8 precision.
- **Slight drop in accuracy (0.15 %).**
- Preprocessing of input images influences the final accuracy.
- Runtime is significantly improved by TensorRT. Around **12x speed up**.





CONCLUSION

- Quantization aware training provides a new alternative to deploy networks in lower precision.
- Since quantization scales are computed during training, QAT models might be less prone to accuracy drop during inference compared to PTQ networks in some cases.
- We have demonstrated an end to end workflow of Resnet-50 QAT model and show that the INT8 accuracy is close to FP32 model.



NVIDIA