4-bit Quantization Model on Image Classification

COMS 6998
Practical Deep Learning System Performance
Final Project

Group:
Praditya Raudi Avinanto (pra2118)
Rifqi Luthfan (rl3154)

Date Submitted: Dec 16, 2021

Contents

EXECUTIVE SUMMARY

Executive Summary

1. Quantization is one of the techniques to reduce model size and computational complexity which can then be implemented in edge devices (Mobile Phones, IoT devices)
   - ➢ Currently, PyTorch and Tensorflow supports only 8-bit integer quantization
2. In this project, we explore converting a 32-bit float neural network (NN) model into a precision lower than 8-bit integer NN model
   - ➢ We experimented using 8,7,6,5,4 bits quantization for two models (ResNet-18 and ResNet-50) for two datasets (CIFAR10 and ImageNette)
   - ➢ We experimented both Post Training Quantization and Quantization Aware Training
3. We found the different effect of each model responding to different bitwidth quantization

PROBLEM MOTIVATION & BACKGROUND WORK

Problem Motivation

1. Neural networks have evolved without regard to model complexity and computational efficiency
   - ➢ Requires powerful computation machine with large storage capacity
   - ➢ **Not well suited for edge devices** with lower computation power
2. Quantization, a method to perform computations and storing tensors at lower bitwidths [2], can be utilized to solve this problem, however:
   - ➢ We usually see a significant drop in accuracy performance when moving to a lower precision bits
   - ➢ The current implementation of quantization only support 8 bits quantization

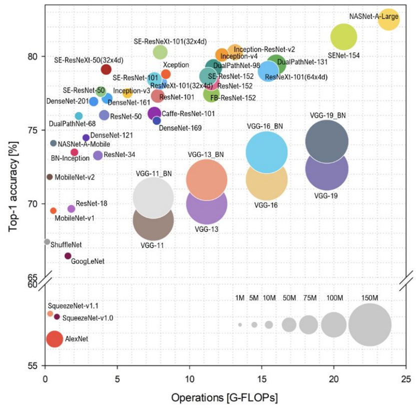
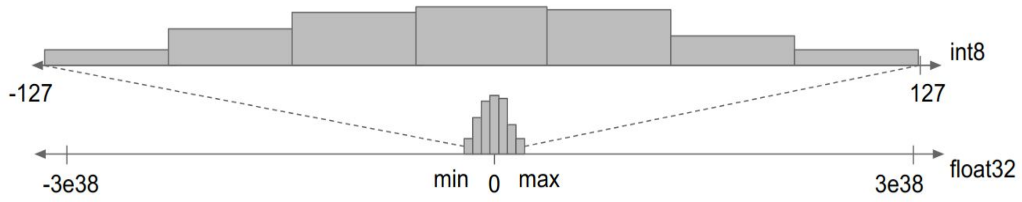


Image from [1] indicating higher accuracy implies higher computation & memory

[1] Benchmark Analysis of Representative Deep Neural Network Architectures
[2] Quantization - PyTorch

COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

What is Quantization?



1. Neural Networks models usually run arithmetic operations for the weights and activations in 32 bits floating point
2. Question arise: can neural networks run in lower bit precision?
   - ➢ This will allow model to be more compact
   - ➢ The concept behind quantization is mapping the floating point weights and activations into a different integer bit scheme
     1. Get zero_point and scale of the float32 values
     2. Do quantize/dequantize based on the formula

        float_value = (int_value - zero_point) * scale

        int_value = float_value/scale - zero_point

[3] Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference

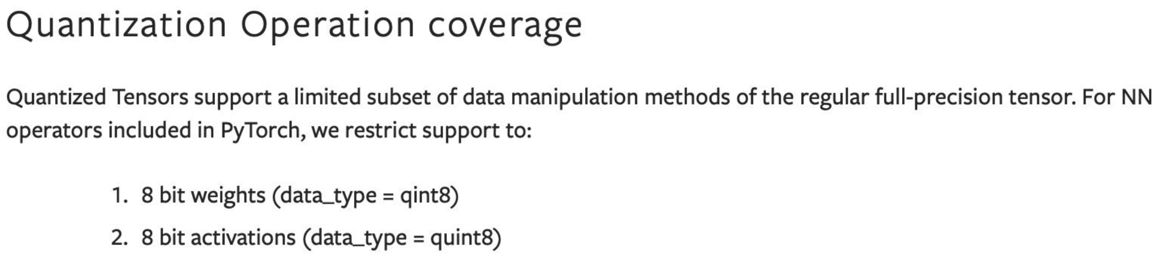
Types of Quantization

1. Post Training Quantization (PTQ)
   - ➢ Model training is done using normal float32 weights and activations, and then quantization is applied after the training is completed
     - i. Simple to apply
     - ii. Accuracy downgrade might be big
2. Quantization Aware Training (QAT)
   - ➢ Simulates the quantization of weights and activations during training's forward pass process
     - i. More robust accuracy performance because quantization error is taken into account
     - ii. More involved than other option (involving fine-tuning and access to full dataset)

[3] Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference

TECHNICAL CHALLENGES

Frameworks only support 8 bit quantization



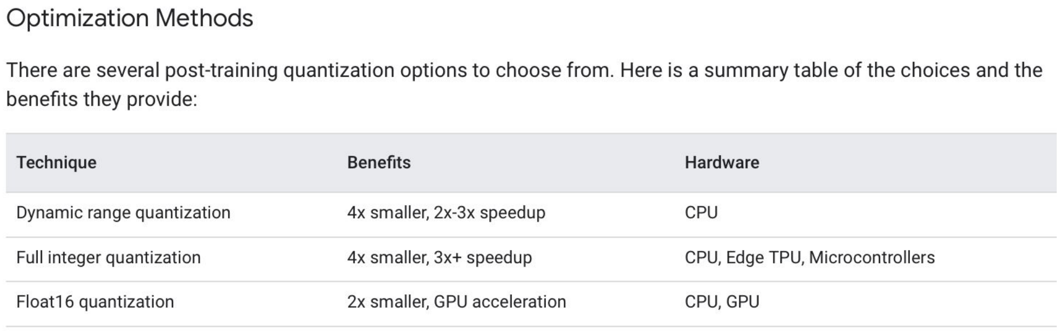
Quantization Operation coverage

Quantized Tensors support a limited subset of data manipulation methods of the regular full-precision tensor. For NN operators included in PyTorch, we restrict support to:

1. 8 bit weights (data_type = qint8)
2. 8 bit activations (data_type = quint8)

Source: https://pytorch.org/docs/stable/quantization.html

Source: https://www.tensorflow.org/lite/performance/post_training_quantization



Optimization Methods

There are several post-training quantization options to choose from. Here is a summary table of the choices and the benefits they provide:

Technique	Benefits	Hardware
Dynamic range quantization	4x smaller, 2x-3x speedup	CPU
Full integer quantization	4x smaller, 3x+ speedup	CPU, Edge TPU, Microcontrollers
Float16 quantization	2x smaller, GPU acceleration	CPU, GPU

Quantization affects all Neural Networks operations

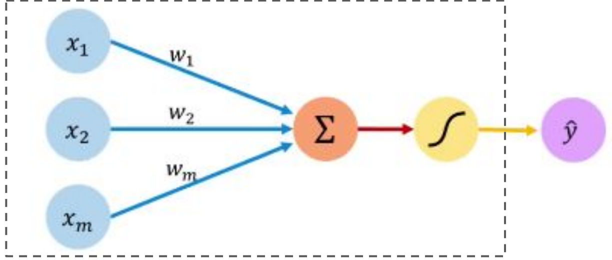
All of the operations are executed on integer tensors rather than floating point values on Quantized model

- ➢ We need to implement custom modules and functions that supports quantization lower than 8 bits from vanilla framework

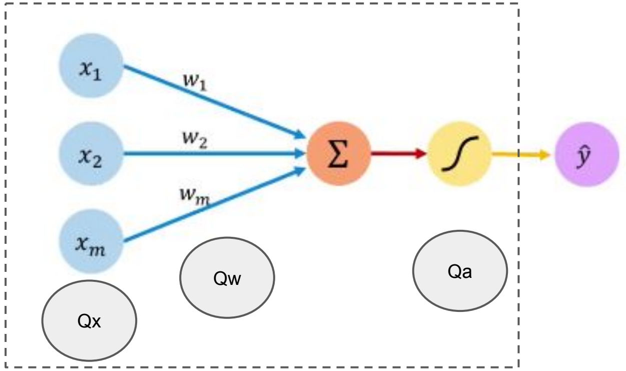
[3] Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference

APPROACH & SOLUTION DIAGRAM / ARCHITECTURE

Neural Network vs Quantized Neural Network



Source: https://medium.com/analytics-vidhya/neural-network-part1-inside-a-single-neuron-fee5e44f1e



Qx = {x_scale, x_zero_point, x_max, x_min, q_max, q_min}
Qw = {w_scale, w_zero_point, w_max, w_min, q_max, q_min}
Qa = {a_scale, a_zero_point, a_max, a_min, q_max, q_min}

QUANT & DEQUANT OP

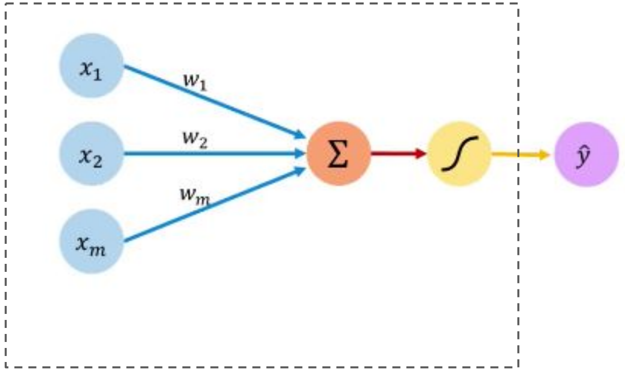
Scale = (v_max - v_min) / (q_max - q_min)

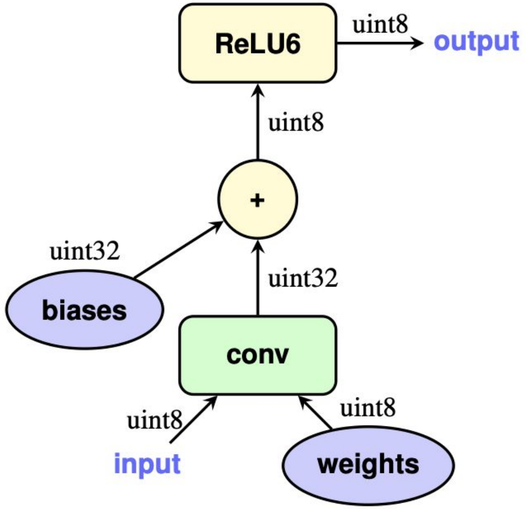
Zero_point = q_max - v_max / scale

float_value = (int_value - zero_point) * scale

int_value = float_value/scale - zero_point

Post Training Quantization





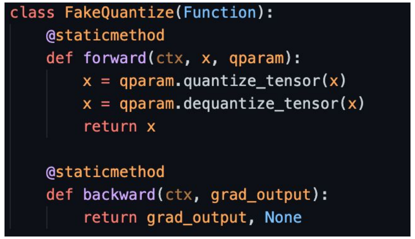
Qx = {x_scale, x_zero_point, x_max, x_min, q_max, q_min}
Qw = {w_scale, w_zero_point, w_max, w_min, q_max, q_min}
Qa = {a_scale, a_zero_point, a_max, a_min, q_max, q_min}

QUANT & DEQUANT OP
Scale = (v_max - v_min) / (q_max - q_min)
Zero_point = q_max - v_max / scale
float_value = (int_value - zero_point) * scale
int_value = float_value/scale - zero_point

Image from [3] Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference

Quantization Aware Training

1. We implemented "Fake" Quantization Function that quantize and dequantize the weights and activations of the network during forward pass to simulate precision loss.
2. We also implemented QParam as a module responsible for updating the quantization parameters during training and also quantize the tensor if needed.



```python
class FakeQuantize(Function):
    @staticmethod
    def forward(ctx, x, qparam):
        x = qparam.quantize_tensor(x)
        x = qparam.dequantize_tensor(x)
        return x

    @staticmethod
    def backward(ctx, grad_output):
        return grad_output, None
```

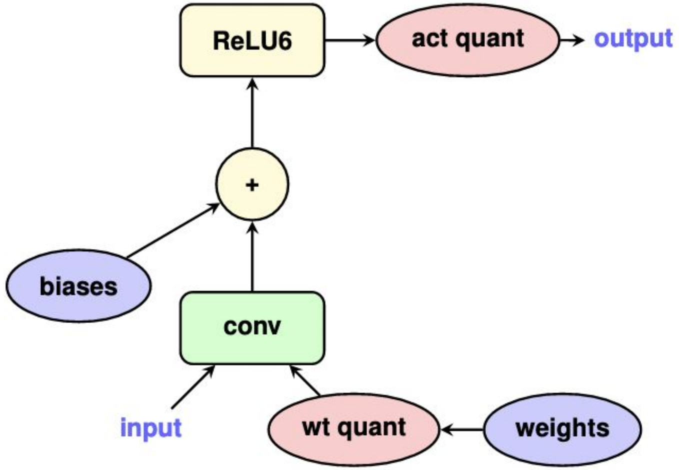
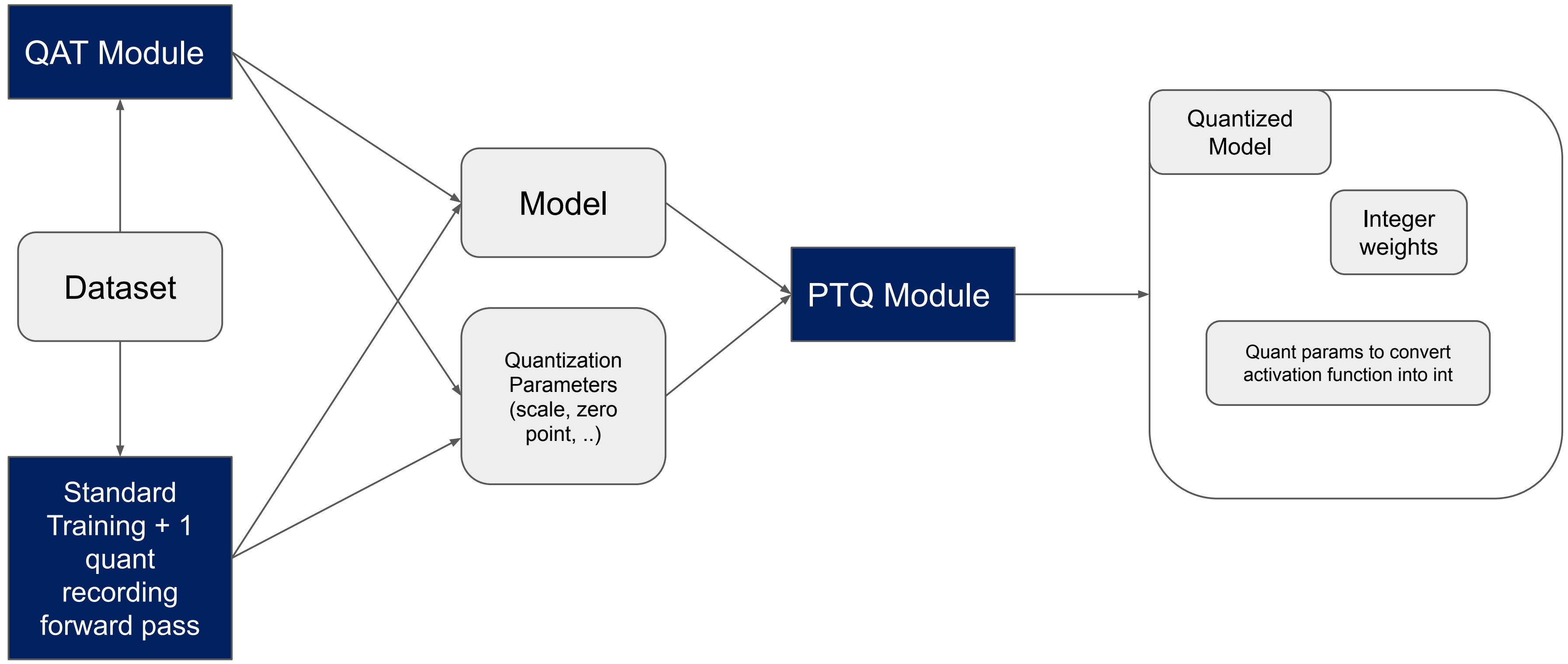
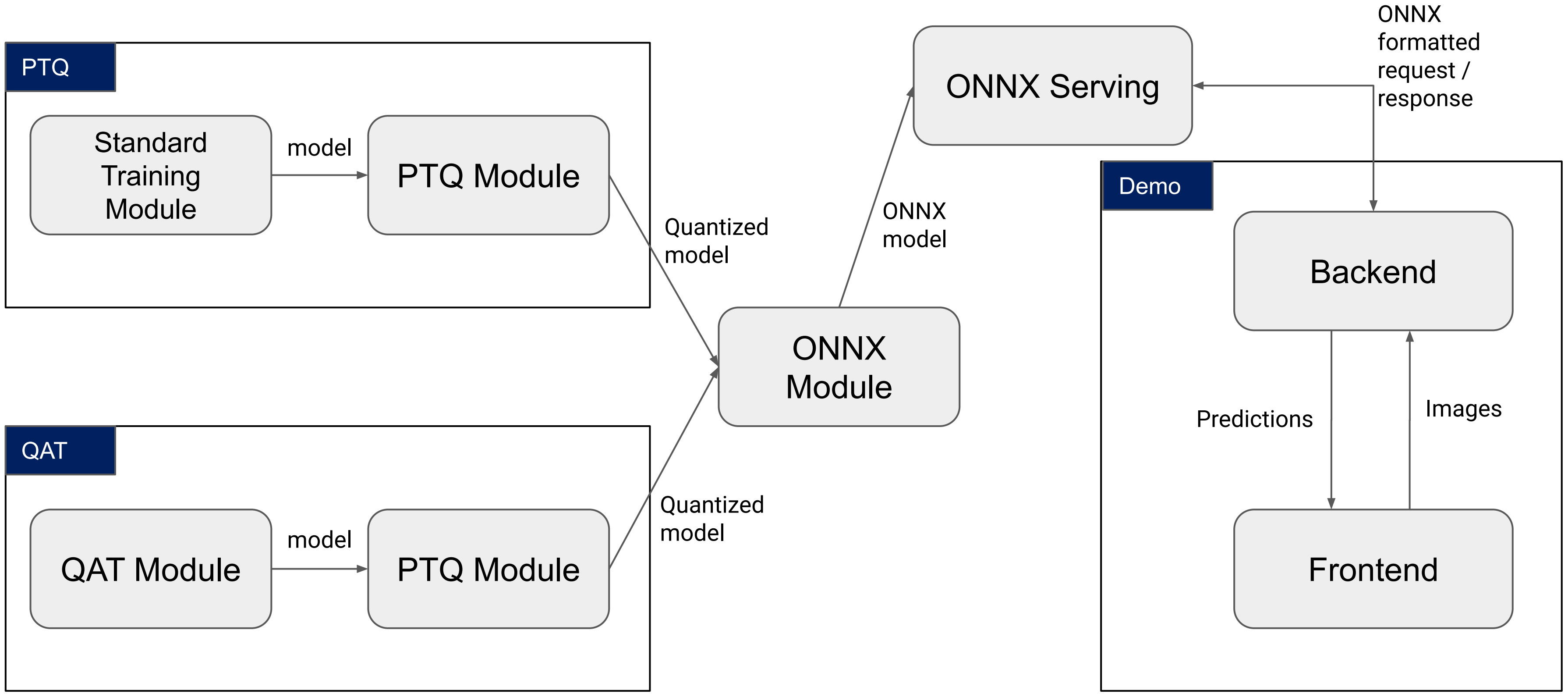


Image from [3] Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference

Quantization Flow


QAT Module
Dataset
Standard Training + 1 quant recording forward pass
Model
Quantization Parameters (scale, zero point, ..)
PTQ Module
Quantized Model
Integer weights
Quant params to convert activation function into int

Solution Diagram / Architecture



IMPLEMENTATION DETAILS & EXPERIMENT DESIGN

Experiment Design

Datasets:

- Imagenette
    - subset of 10 easily classified classes from Imagenet
- CIFAR10

Model Architectures (pretrained on Imagenet):

- Resnet18
- Resnet50

Quantization Methods:

- Post Training Quantization
- Quantization Aware Training

Hardware:

- NVIDIA Tesla V100
- 8 vCPU 30gb RAM (n1-standard-8 GCP)

Bitwidths:

- Full precision: 32 bit
- Low precision: 8 bit, 7 bit, 6 bit, 5 bit, 4 bit

Metrics:

- Accuracy
- Model Size
- Inference Time

Framework:

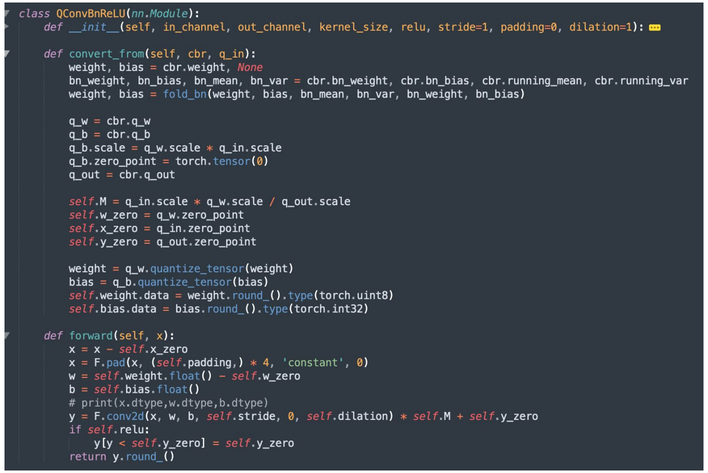
- Pytorch
- Fastai (for dataloader)
- ONNX (for inference runtime)

Implementation Details

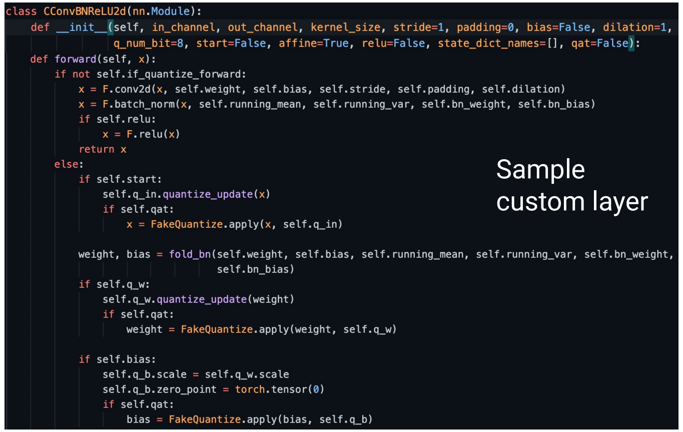
1. Implement quantization module from scratch
    - ➢ Use Pytorch module register buffer to keep quantization parameters
        - i. zero point, scale, vmin, vmax, and num bit
    - ➢ Implement quantization-compatible layer for conv, batchnorm, relu, maxpool, average pool, linear, and add layer
    - ➢ Conv, BN, and relu fusion function implementation to speedup runtime
    - ➢ These layers is used to implement quantization-compatible basic and bottleneck block for Resnet
2. QAT implementation
    - ➢ Fake quantization Function
    - ➢ Quantization parameters updates on training
3. PTQ implementation
    - ➢ Quantization / model conversion.

Custom Layers and Models

1. We implemented custom Convolutional, Batch Normalization, Relu, Linear, and Addition Layers differently for PTQ and QAT. All PTQ weights & activation outputs are already in integers while QAT has FakeQuant and QParam.
2. Those layers are then used to create a custom ResNet Model then we can do Post Training Quantization and Quantization Aware Training



```python
class QConvBnReLU(nn.Module):
    def __init__(self, in_channel, out_channel, kernel_size, relu, stride=1, padding=0, dilation=1):

    def convert_from(self, cbr, q_in):
        weight, bias = cbr.weight, None
        bn_weight, bn_bias, bn_mean, bn_var = cbr.bn_weight, cbr.bn_bias, cbr.running_mean, cbr.running_var
        weight, bias = fold_bn(weight, bias, bn_mean, bn_var, bn_weight, bn_bias)

        q_w = cbr.q_w
        q_b = cbr.q_b
        q_b.scale = q_w.scale * q_in.scale
        q_b.zero_point = torch.tensor(0)
        q_out = cbr.q_out

        self.M = q_in.scale * q_w.scale / q_out.scale
        self.w_zero = q_w.zero_point
        self.x_zero = q_in.zero_point
        self.y_zero = q_out.zero_point

        weight = q_w.quantize_tensor(weight)
        bias = q_b.quantize_tensor(bias)
        self.weight.data = weight.round_().type(torch.uint8)
        self.bias.data = bias.round_().type(torch.int32)

    def forward(self, x):
        x = x - self.x_zero
        x = F.pad(x, (self.padding,) * 4, 'constant', 0)
        w = self.weight.float() - self.w_zero
        b = self.bias.float()
        # print(x.dtype,w.dtype,b.dtype)
        y = F.conv2d(x, w, b, self.stride, 0, self.dilation) * self.M + self.y_zero
        if self.relu:
            y[y < self.y_zero] = self.y_zero
        return y.round_()
```

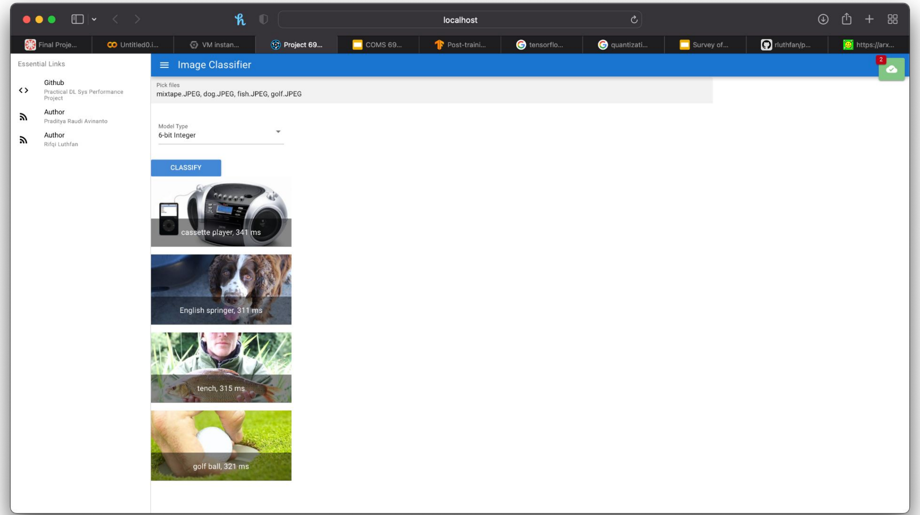


```python
class CConvBNReLU2d(nn.Module):
    def __init__(self, in_channel, out_channel, kernel_size, stride=1, padding=0, bias=False, dilation=1,
                 q_num_bit=8, start=False, affine=True, relu=False, state_dict_names=[], qat=False):
    def forward(self, x):
        if not self.if_quantize_forward:
            x = F.conv2d(x, self.weight, self.bias, self.stride, self.padding, self.dilation)
            x = F.batch_norm(x, self.running_mean, self.running_var, self.bn_weight, self.bn_bias)
            if self.relu:
                x = F.relu(x)
            return x
        else:
            if self.start:
                self.q_in.quantize_update(x)
                if self.qat:
                    x = FakeQuantize.apply(x, self.q_in)

            weight, bias = fold_bn(self.weight, self.bias, self.running_mean, self.running_var, self.bn_weight,
                                   self.bn_bias)
            if self.q_w:
                self.q_w.quantize_update(weight)
                if self.qat:
                    weight = FakeQuantize.apply(weight, self.q_w)

            if self.bias:
                self.q_b.scale = self.q_w.scale
                self.q_b.zero_point = torch.tensor(0)
                if self.qat:
                    bias = FakeQuantize.apply(bias, self.q_b)
```

Sample custom layer

Web Application

1. Four ONNXRuntime servers (Resnet50 32, 8, 6, 4 bit), 1 vCPU, 3.75 GB RAM each
2. Backend Flask
3. Frontend VueJS



Essential Links

Github
Practical DL Sys Performance Project

Author
Praditya Raudi Avinanto

Author
Rifqi Luthfan

Pick files
mixtape.JPEG, dog.JPEG, fish.JPEG, golf.JPEG

Model Type
6-bit Integer

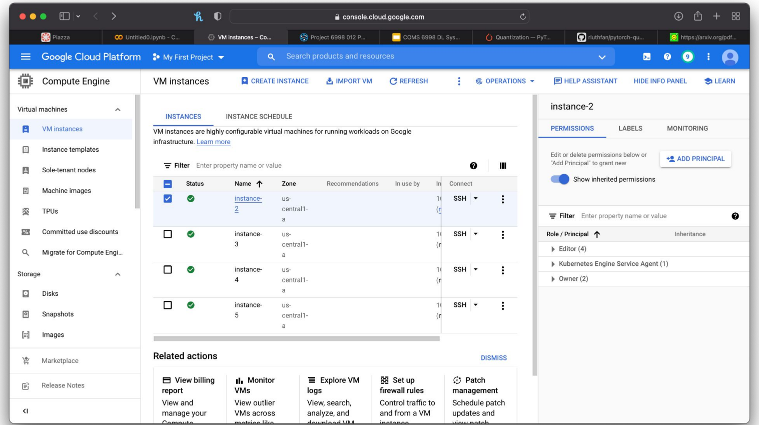
CLASSIFY

cassette player, 341 ms

English springer, 311 ms

tench, 315 ms

golf ball, 321 ms



VM instances

VM instances are highly configurable virtual machines for running workloads on Google infrastructure. Learn more

Filter Enter property name or value

Status	Name	Zone	Recommendations	In use by	In	Connect
✓	instance-2	us-central1-a			1	SSH
✓	instance-3	us-central1-a			1	SSH
✓	instance-4	us-central1-a			1	SSH
✓	instance-5	us-central1-a			1	SSH

Related actions DISMISS

View billing report — View and manage your Compute
Monitor VMs — View outlier VMs across metrics like
Explore VM logs — View, search, analyze, and download VM
Set up firewall rules — Control traffic to and from a VM instance
Patch management — Schedule patch updates and view patch

instance-2

Edit or delete permissions below or "Add Principal" to grant new

Show inherited permissions

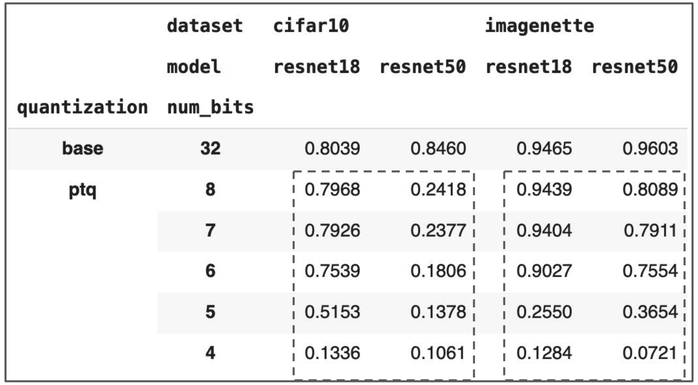
Filter Enter property name or value

Role / Principal	Inheritance
Editor (4)	
Kubernetes Engine Service Agent (1)	
Owner (2)	

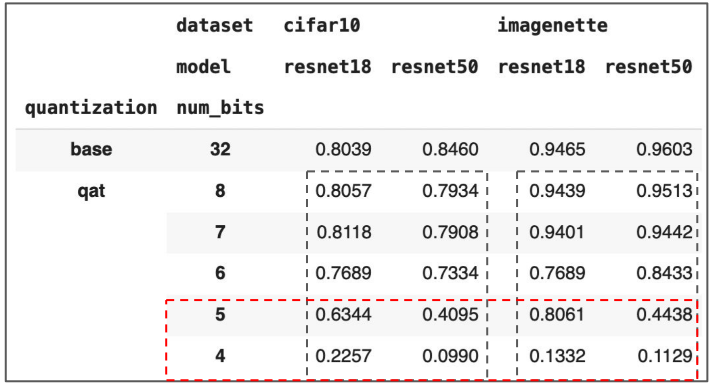
DEMO

EXPERIMENT EVALUATION

Validation Accuracy of Quantized model



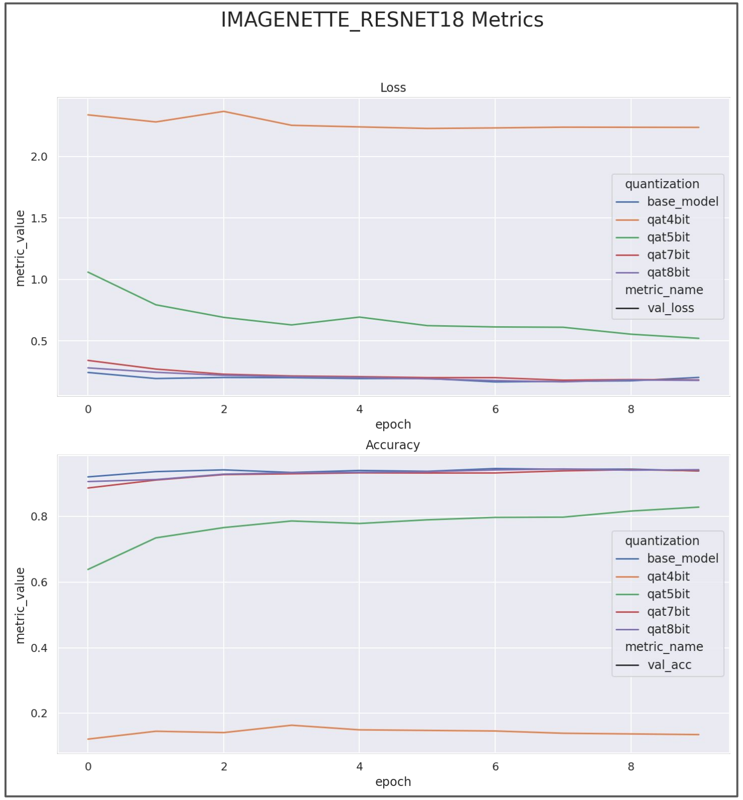
	dataset	cifar10		imagenette	
	model	resnet18	resnet50	resnet18	resnet50
quantization	num_bits				
base	32	0.8039	0.8460	0.9465	0.9603
ptq	8	0.7968	0.2418	0.9439	0.8089
	7	0.7926	0.2377	0.9404	0.7911
	6	0.7539	0.1806	0.9027	0.7554
	5	0.5153	0.1378	0.2550	0.3654
	4	0.1336	0.1061	0.1284	0.0721

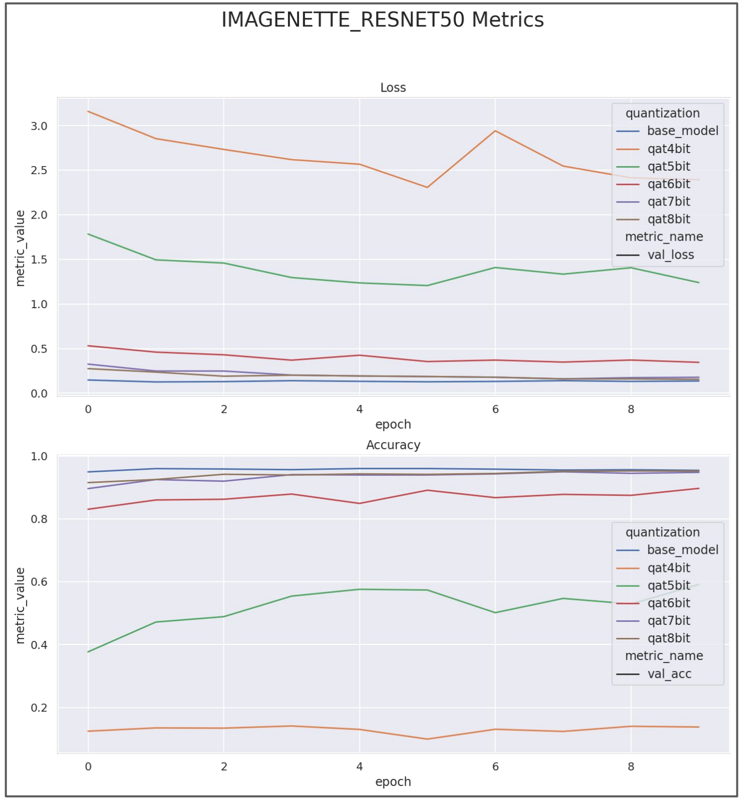


	dataset	cifar10		imagenette	
	model	resnet18	resnet50	resnet18	resnet50
quantization	num_bits				
base	32	0.8039	0.8460	0.9465	0.9603
qat	8	0.8057	0.7934	0.9439	0.9513
	7	0.8118	0.7908	0.9401	0.9442
	6	0.7689	0.7334	0.7689	0.8433
	5	0.6344	0.4095	0.8061	0.4438
	4	0.2257	0.0990	0.1332	0.1129

1. Post Training Quantization (left) will benefit more from a similar source dataset
2. Quantization Aware Training (right) has lower accuracy drop compared to Post Training Quantization (left)
   ➢ Quantization error is taken into account during learning process
3. Even with QAT, the more complex the model is (ResNet-50), the worse the resulting accuracy drop when doing lower bitwidths quantization (true for both dataset)
   ➢ This is possibly caused by the more complex float32 values required in deeper model

Model lose capacity to learn with 4 bits quantization





Model Size

	Resnet18	Resnet50
32 bit	44.9 MB	94.8 MB
8 bit	11.2 MB	23.7 MB
7 bit	11.2 MB	23.7 MB
6 bit	11.2 MB	23.7 MB
5 bit	11.2 MB	23.7 MB
4 bit	11.2 MB	23.7 MB

Most current frameworks still only support 8 bit integers

Web Inference Time

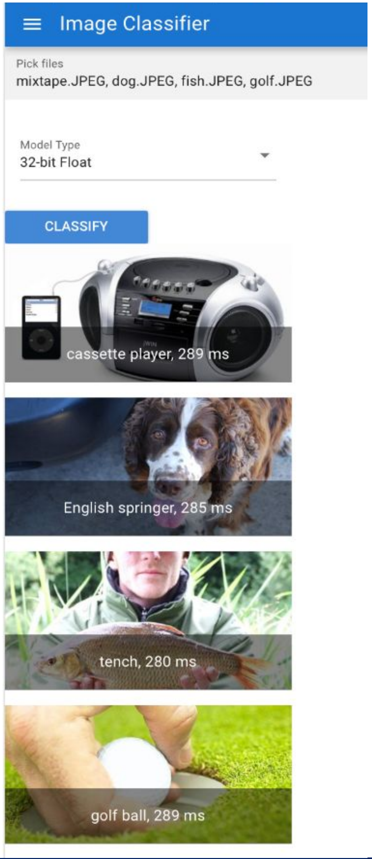


Image Classifier

Pick files
mixtape.JPEG, dog.JPEG, fish.JPEG, golf.JPEG

Model Type
32-bit Float

CLASSIFY

cassette player, 289 ms
English springer, 285 ms
tench, 280 ms
golf ball, 289 ms

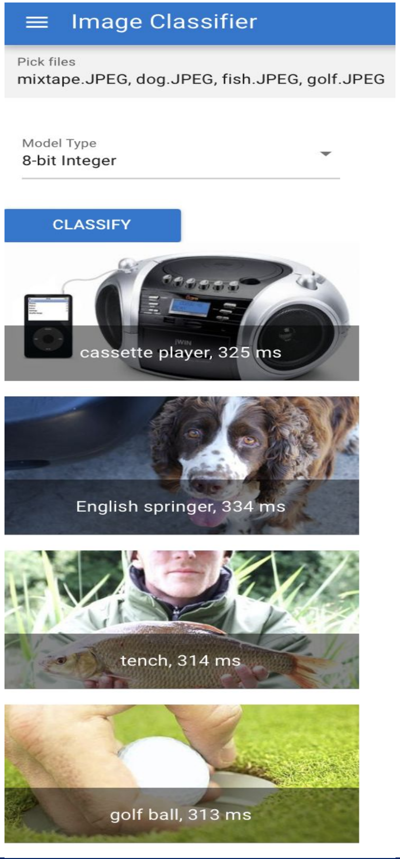


Image Classifier

Pick files
mixtape.JPEG, dog.JPEG, fish.JPEG, golf.JPEG

Model Type
8-bit Integer

CLASSIFY

cassette player, 325 ms
English springer, 334 ms
tench, 314 ms
golf ball, 313 ms

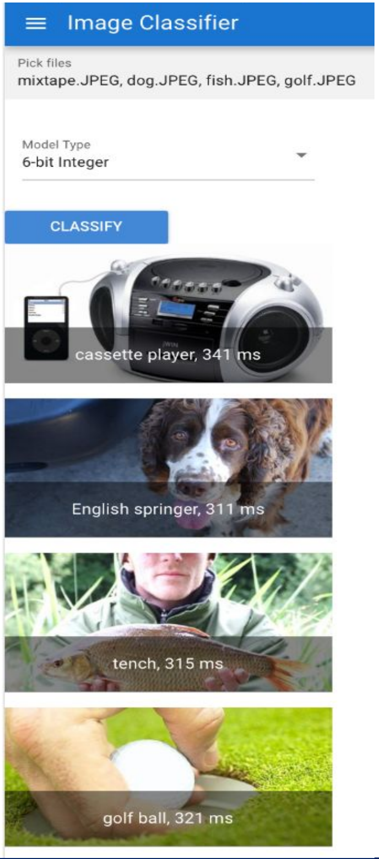


Image Classifier

Pick files
mixtape.JPEG, dog.JPEG, fish.JPEG, golf.JPEG

Model Type
6-bit Integer

CLASSIFY

cassette player, 341 ms
English springer, 311 ms
tench, 315 ms
golf ball, 321 ms

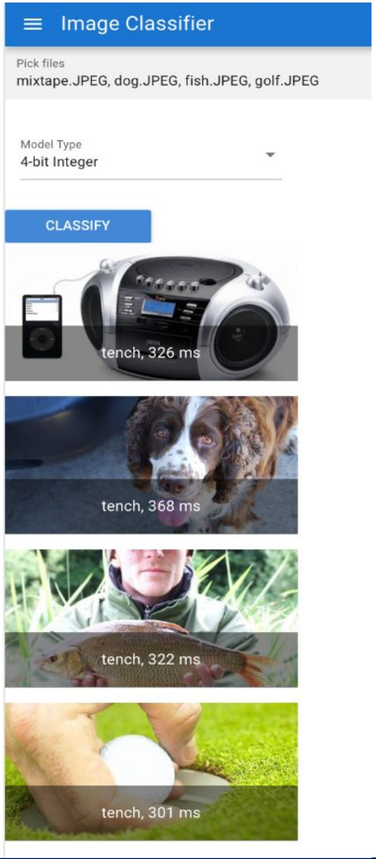


Image Classifier

Pick files
mixtape.JPEG, dog.JPEG, fish.JPEG, golf.JPEG

Model Type
4-bit Integer

CLASSIFY

tench, 326 ms
tench, 368 ms
tench, 322 ms
tench, 301 ms

CONCLUSION

Conclusion

1. Quantization Aware Training performs better than Post Training Quantization since it incorporates quantization loss during the training.
2. Compressing the model to lower precision than 8-bit is possible and the model can still achieve acceptable accuracy on 5-bit precision.
3. Model is unable to learn or losing the capacity when trained on 4-bit precision.
4. For x86-based processor, 4-bit, 5-bit, 6-bit, 7-bit and 8-bit quantization don’t make any difference in size and memory usage. Empirically, the lowest possible representation of integer in python is 8-bit.
5. ONNX can’t leverage PyTorch quantization.
   - ➢ Theoretically, the advantage of 4-bit quantization is 2x reduction in bandwidth, which can increase throughput even though model size is the same as 8 bit