infocusp Innovations

Quantization

An Introduction

Table of Contents

1 What is Quantization

2 Float 32 in Memory

3 Symmetric Quantization

4 Asymmetric Quantization

5 Modes of Quantization

6 Post Training Dynamic Quantization

7 Post Training Static Quantization

8 Quantization Aware Training

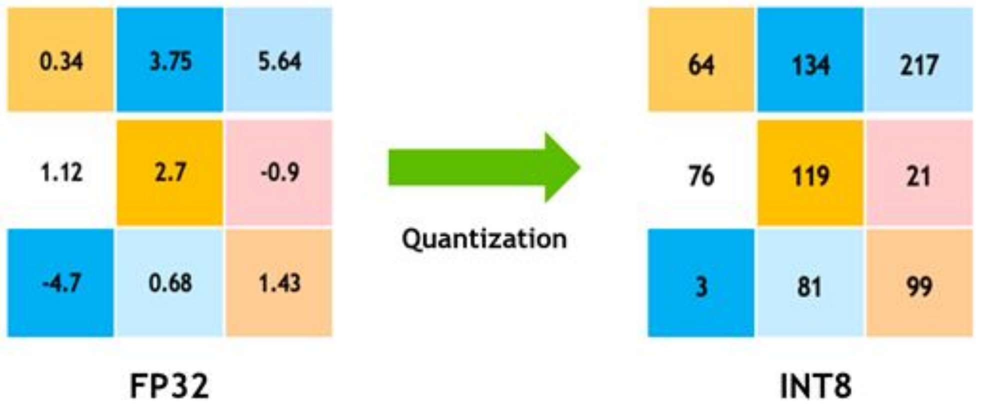
9 ONNX Model Quantization

10 Bits and Bytes for Quantization

What is Quantization

- ***Edge devices*** like cell phones, microcontrollers, have ***less memory*** in MBs
- When we want to deploy ML models in these devices, we require to optimise them such that they can run on these devices.
- An option for Optimisation is ***Quantization***.
- Reduces memory footprint by reducing model size.
- Inference becomes ***faster***.
- Both PyTorch and TensorFlow support quantization, more details in next section.
- There are some tools which can quantize ONXX models and API like NVIDIA TensorRT which might ease this quantization process.

What is Quantization

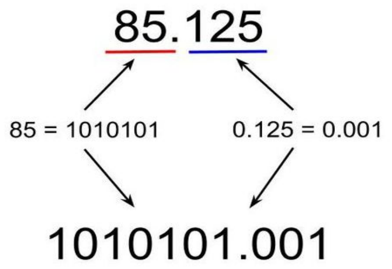


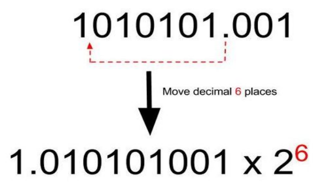
Quantization is the process of reducing the model size and do fast computation by converting the model parameter from float32 to int8 or lower bits representation.

Float32 To INT8:

- 4x reduction in size
- 2-4x reduction in memory bandwidth
- 2-4x faster inference

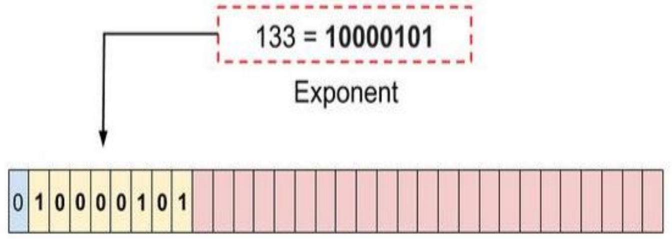
Float 32 representation in memory

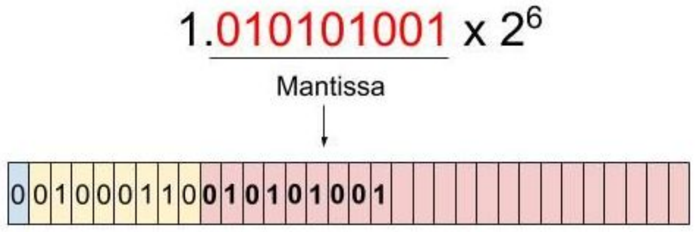




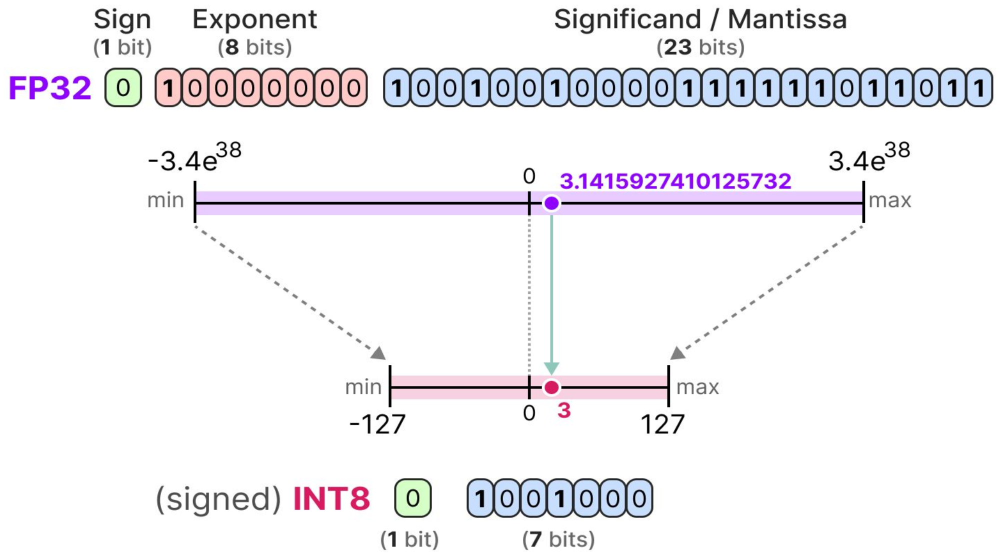
1.010101001×2^{6}

$127 + 6 = 133$





Int 8 representation in Memory



Types of quantization

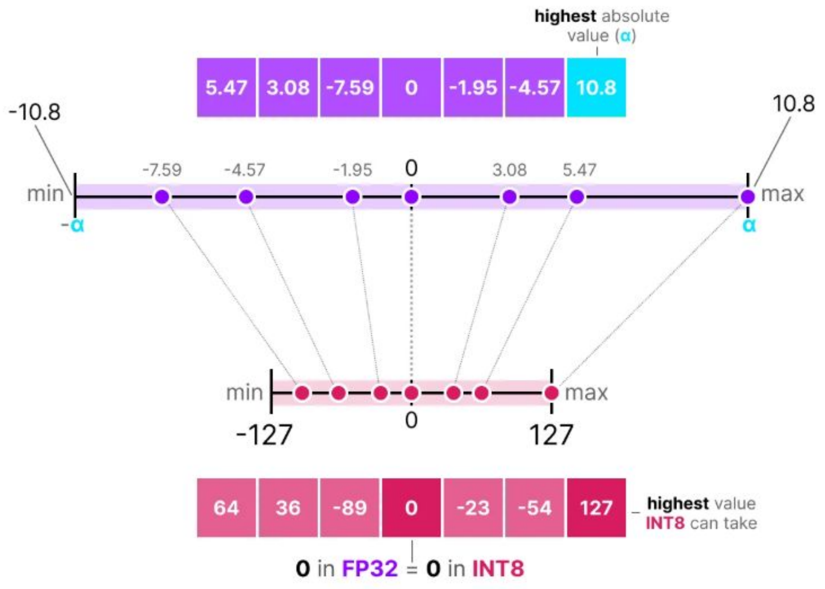
Symmetric Quantization

- Use a zero-centered scale where values are mapped symmetrically around zero
- Range : -127 to 127 (for 8 bit)
- Good for balance dataset distribution.

Asymmetric Quantization

- Different scale and zero point, allowing a shift in the representation.
- Range : 0 to 255 (for 8 bit)
- Good for unbalanced data distribution.

What is Symmetric Quantization



Note the [-127, 127] range of values represents the restricted range. The unrestricted range is [-128, 127] and depends on the quantization method.

Symmetric Quantization

1. **Compute Scale (S):**

$$S = \frac{\alpha}{2^{n-1} - 1}$$

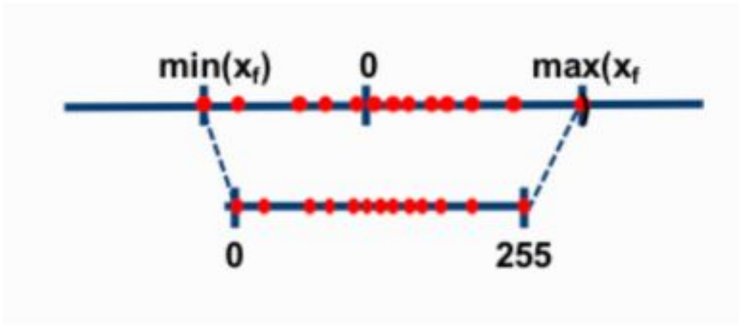
2. **Quantize Values:**

$$x_q = \text{clip}\left(\text{round}\left(\frac{x}{S}\right), -(2^{n-1} - 1), 2^{n-1} - 1\right)$$

```
- alpha (max value): The absolute maximum value of the input data.
- S (scale): The factor that maps floating-point values to integer values.
- Z (zero-point): The offset applied to align the quantized values.
```

The process of choosing these clipping values alpha and beta and hence the clipping range is called **calibration**.

What is Asymmetric Quantization



Asymmetric Quantization

1. **Compute Scale (S):**

$$S = \frac{\alpha - \beta}{2^n - 1}$$

2. **Compute Zero-Point (Z):**

$$Z = \text{round}\left(-\frac{\beta}{S}\right)$$

3. **Quantize Values:**

$$x_q = \text{clip}\left(\text{round}\left(\frac{x}{S}\right) + Z, 0, 2^n - 1\right)$$

```
- alpha (max value): The maximum value of the input data.
- beta (min value): The minimum value of the input data.
- S (scale): The factor that maps floating-point values to integer values.
- Z (zero-point): The offset applied to align the quantized values.
- n is 8 in case of int8 conversion.
```

3 Modes of Quantization

01 **Post Training Dynamic Quantization**

- Weights are quantized ahead of inference, but activations are quantized dynamically during execution.
- Suitable for models with lots of matrix multiplications.

02 **Post Training Static Quantization**

- Both weight and activations are quantized before inference.
- Require a representative dataset for calibration.
- Less accuracy but faster as compared to dynamic quantization.

03 **Quantization Aware Training**

- The model is trained with quantization in mind.
- Simulated the effects of quantization during training to reduce accuracy loss.
- Provides the highest accuracy among the three methods.

Dynamic Quantization

What is Dynamic Quantization?

- We select layers to quantize (like Linear layers)
- Not all layers are compatible for dynamic quantization. (check pytorch docs)
- The weights of these layers are quantized to int8 before inference
- The model continues to take float32 inputs

Input Handling

- When float32 input hits a quantized layer
- The input is temporarily quantized to int8 just for the matrix multiplication
- This is done because int8 × int8 is much faster than float32 × float32 on CPU
- This temporary quantization uses dynamic scaling factors calculated on-the-fly

Dynamic Quantization - PyTorch

```python
# Define the network:
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Quantization API

```python
quantized_model = torch.quantization.quantize_dynamic(
    model, {torch.nn.Linear}, dtype=torch.qint8
)
```

Quantized Network

```
Net(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): DynamicQuantizedLinear(in_features=400, out_features=120, dtype=torch.qint8,
qscheme=torch.per_tensor_affine)
  (fc2): DynamicQuantizedLinear(in_features=120, out_features=84, dtype=torch.qint8,
qscheme=torch.per_tensor_affine)
  (fc3): DynamicQuantizedLinear(in_features=84, out_features=10, dtype=torch.qint8,
qscheme=torch.per_tensor_affine)
)
```

https://github.com/infocusp/model_quantization

Static Quantization

1. Unlike dynamic quantization, static quantization does not calculate the *zero point* (***z***) and scale factor (***s***) during inference but beforehand.
2. To find those values, a **calibration dataset** is used and given to the model to collect these potential distributions.
3. Once we calculate s and z, we perform quantization to the model.
4. Same s and z values are kept for every batch which can lead to model become less accurate as compare do dynamic quantization.
5. Static quantization is less accuracy but faster than dynamic as it don’t need to calculate s and z per every layer.

Static Quantization

```python
class QNet(nn.Module):
    def __init__(self):
        super(QNet, self).__init__()
        self.quant = torch.quantization.QuantStub()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
        self.dequant = torch.quantization.DeQuantStub()

    def forward(self, x):
        x = self.quant(x)
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.reshape(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        x = self.dequant(x)
        return x
```

https://github.com/infocusp/model_quantization

Quantization API

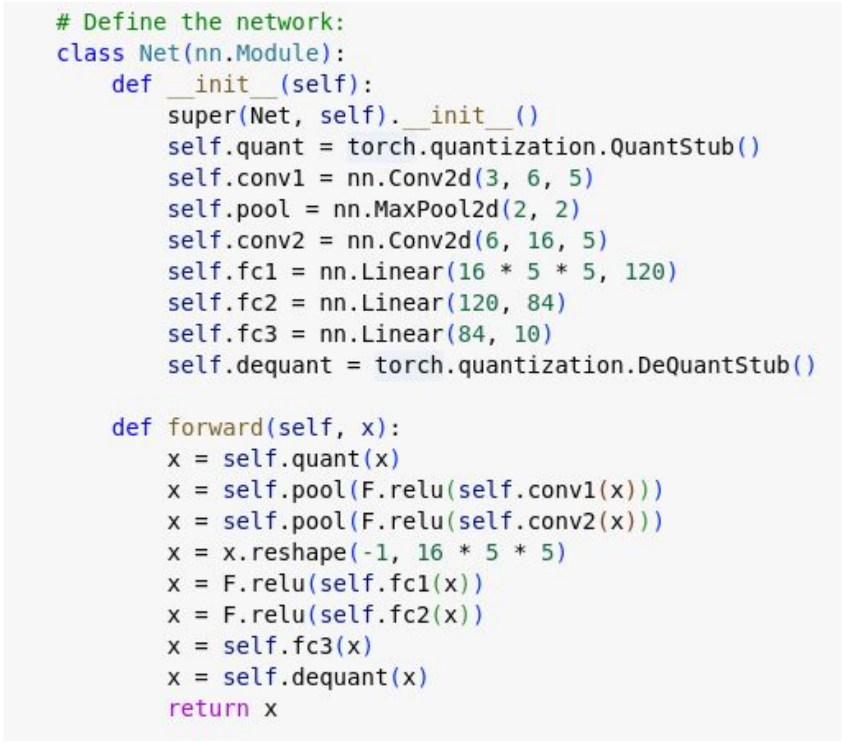
```python
# Static quantization pytorch needs to happen on CPU
qmodel = QNet().to('cpu')
qmodel.load_state_dict(model.state_dict())
qmodel.eval()
# Set qconfig for static quantization
qmodel.qconfig = torch.ao.quantization.get_default_qconfig('fbgemm')
# Prepare model for quantization (adds observers)
qmodel = torch.ao.quantization.prepare(qmodel)
# Calibrate the model
calibrate(qmodel, train_loader_subset)
# Convert to quantized model
qmodel = torch.ao.quantization.convert(qmodel)
```

Quantized Network

model summary

```
QNet(
  (quant): Quantize(scale=tensor([0.0157]), zero_point=tensor([64]), dtype=torch.quint8)
  (conv1): QuantizedConv2d(3, 6, kernel_size=(5, 5), stride=(1, 1), scale=0.08214379101991653,
zero_point=65)
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): QuantizedConv2d(6, 16, kernel_size=(5, 5), stride=(1, 1), scale=0.09786629676818848,
zero_point=37)
  (fc1): QuantizedLinear(in_features=400, out_features=120, scale=0.10066696256399155,
zero_point=48, qscheme=torch.per_channel_affine)
  (fc2): QuantizedLinear(in_features=120, out_features=84, scale=0.07960764318704605, zero_point=46,
qscheme=torch.per_channel_affine)
  (fc3): QuantizedLinear(in_features=84, out_features=10, scale=0.10699643939733505, zero_point=58,
qscheme=torch.per_channel_affine)
  (dequant): DeQuantize()
)
```

Quantization Aware Training



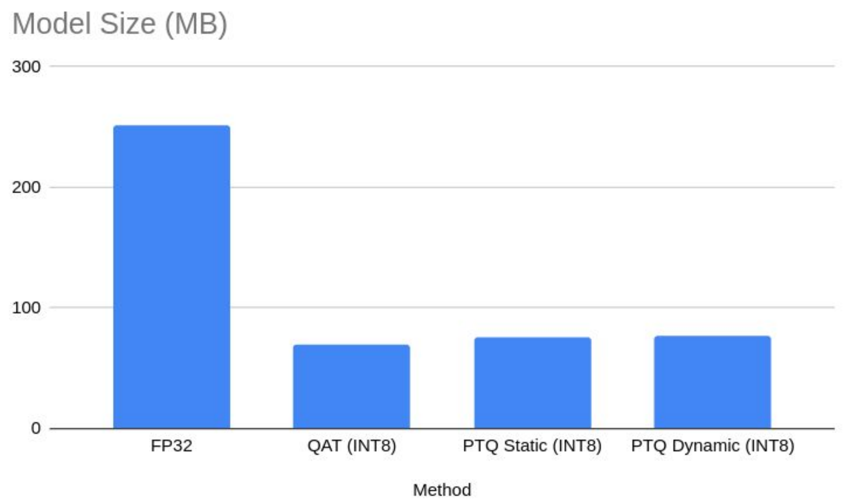
```python
# Define the network:
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.quant = torch.quantization.QuantStub()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
        self.dequant = torch.quantization.DeQuantStub()

    def forward(self, x):
        x = self.quant(x)
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.reshape(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        x = self.dequant(x)
        return x
```

Quantization API

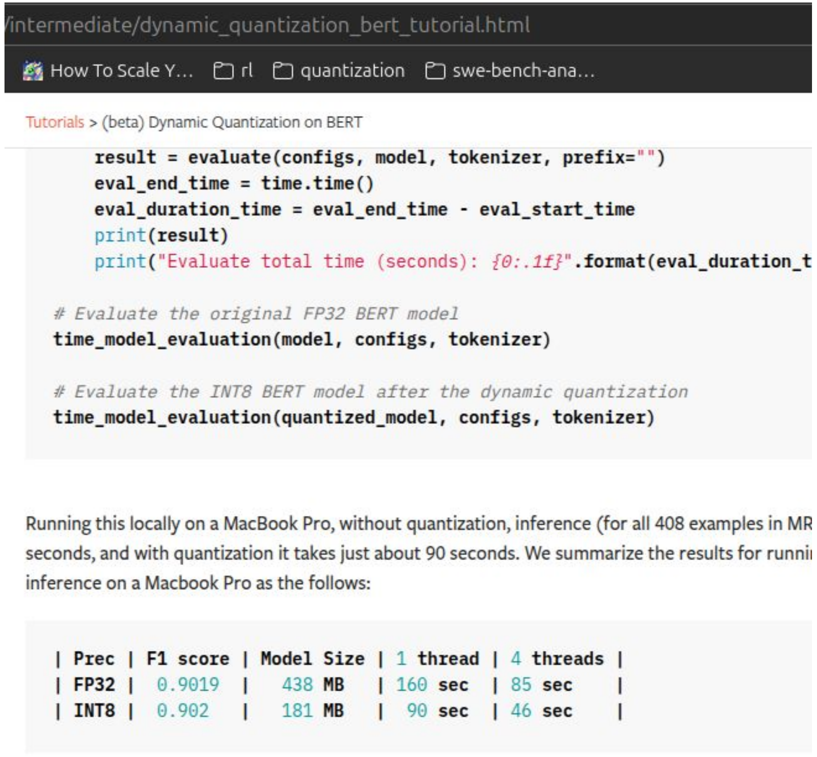
```python
# before train
model = Net().to(device)
model.train()
model.qconfig = torch.ao.quantization.default_qconfig
model = torch.ao.quantization.prepare_qat(model)
# After Training
model.eval()
q_model = torch.ao.quantization.convert(model)
```

https://github.com/infocusp/model_quantization

Results



Model Name	Method	Accuracy	Model Size (MB)
FP32(Original)	Non Quantized Model	0.4731	251.618
QAT (INT8)	Quantization Aware Training	0.486	69.922
PTQ Static (INT8)	Post Training Static Quantization	0.4719	76.002
PTQ Dynamic (INT8)	Post Training Dynamic Quantization	0.4719	76.634



/intermediate/dynamic_quantization_bert_tutorial.html
How To Scale Y...
rl
quantization
swe-bench-ana...
Tutorials > (beta) Dynamic Quantization on BERT

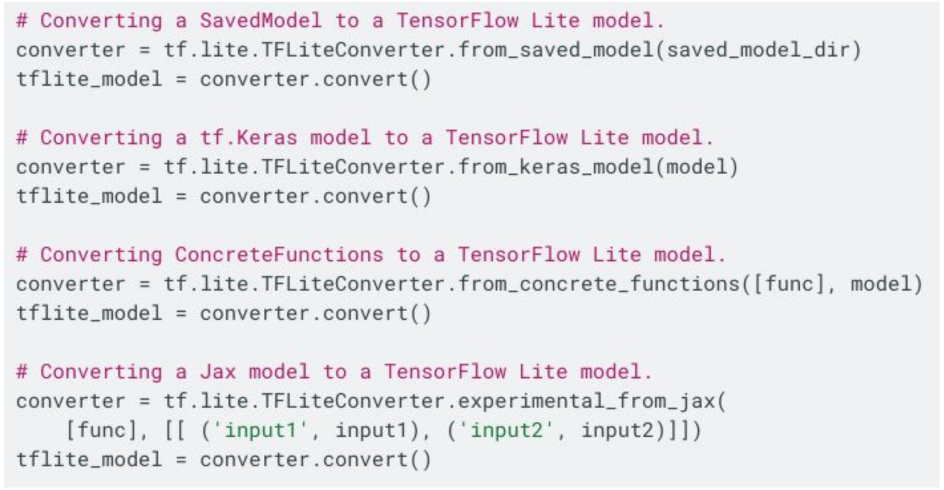
```
        result = evaluate(configs, model, tokenizer, prefix="")
        eval_end_time = time.time()
        eval_duration_time = eval_end_time - eval_start_time
        print(result)
        print("Evaluate total time (seconds): {0:.1f}".format(eval_duration_t

    # Evaluate the original FP32 BERT model
    time_model_evaluation(model, configs, tokenizer)

    # Evaluate the INT8 BERT model after the dynamic quantization
    time_model_evaluation(quantized_model, configs, tokenizer)
```

Running this locally on a MacBook Pro, without quantization, inference (for all 408 examples in MR
seconds, and with quantization it takes just about 90 seconds. We summarize the results for runni
inference on a Macbook Pro as the follows:

```
| Prec | F1 score | Model Size | 1 thread | 4 threads |
| FP32 |  0.9019  |  438 MB    | 160 sec  | 85 sec    |
| INT8 |  0.902   |  181 MB    |  90 sec  | 46 sec    |
```

TFLiTe Framework

- TFLite is an open source tensorflow framework format for on device inference.
- TFLite converter converts the tf model to tflite model which is efficient in terms of memory footprint and accuracy.
- Designed for edges devices deployment.
- Cross-platform support like android, ios, et. cetera is available.
- Post quantization, the model is stored as tflite model which saves the memory footprint even more.
- More details please check - https://www.tensorflow.org/api_docs/python/tf/lite



```
# Converting a SavedModel to a TensorFlow Lite model.
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
tflite_model = converter.convert()

# Converting a tf.Keras model to a TensorFlow Lite model.
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

# Converting ConcreteFunctions to a TensorFlow Lite model.
converter = tf.lite.TFLiteConverter.from_concrete_functions([func], model)
tflite_model = converter.convert()

# Converting a Jax model to a TensorFlow Lite model.
converter = tf.lite.TFLiteConverter.experimental_from_jax(
    [func], [[ ('input1', input1), ('input2', input2)]])
tflite_model = converter.convert()
```

Post Training Quantization - TensorFlow

- ***Dynamic range quantization***
  - The weights and bias of model are stored in INT8, while activations are still expected to be FP32.
  - During inference, the activations, model inputs are converted to INT8 and back to FP32 after operation completion.
  - Little slower as there is a computation overhead, but more accurate than Full integer quantization

```
import tensorflow as tf
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_quant_model = converter.convert()
```

- ***FP16 Quantization***

```
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_types = [tf.float16]
```

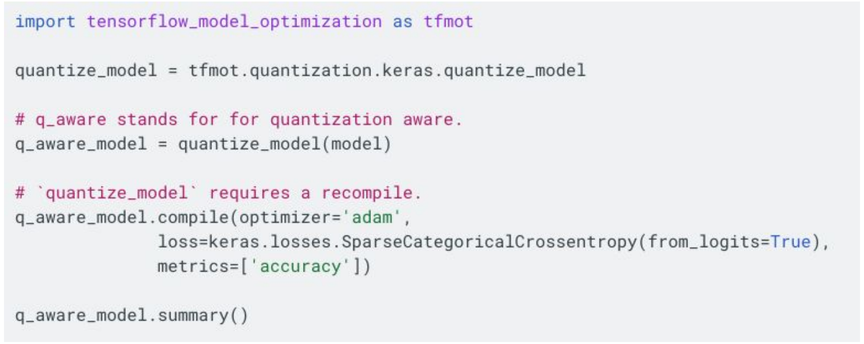
- ***Full integer quantization***
  - The activations, inputs are also quantized to INT 8.
  - So we require some representative dataset to run few inference cycles like 100-200 samples.

```
import tensorflow as tf

def representative_dataset_gen():
  for _ in range(num_calibration_steps):
    # Get sample input data as a numpy array in a method of your choosing.
    yield [input]

converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset_gen
tflite_quant_model = converter.convert()
```

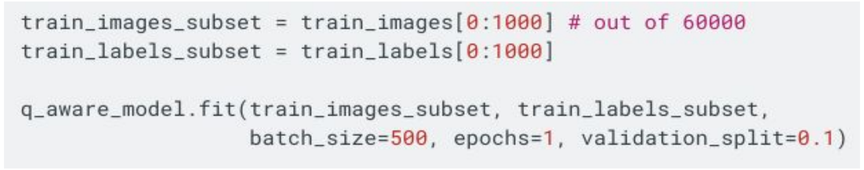
- References

Quantization aware training - TensorFlow

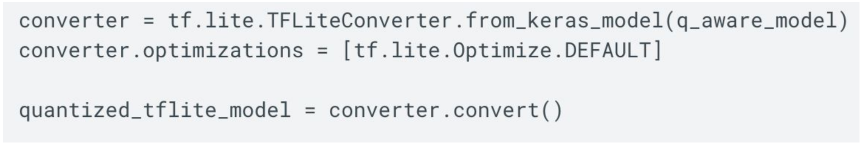
- Non-quantized model is trained first.
- Then this model is converted to quantized model using tfmot library.
- Still the model is not actually quantised, just made aware of quantization.
- This quantised model is compiled, trained for a epoch. This is when actual quantization happens and then the model is saved in tflite format.



```
import tensorflow_model_optimization as tfmot

quantize_model = tfmot.quantization.keras.quantize_model

# q_aware stands for for quantization aware.
q_aware_model = quantize_model(model)

# `quantize_model` requires a recompile.
q_aware_model.compile(optimizer='adam',
              loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

q_aware_model.summary()
```



```
train_images_subset = train_images[0:1000] # out of 60000
train_labels_subset = train_labels[0:1000]

q_aware_model.fit(train_images_subset, train_labels_subset,
                  batch_size=500, epochs=1, validation_split=0.1)
```

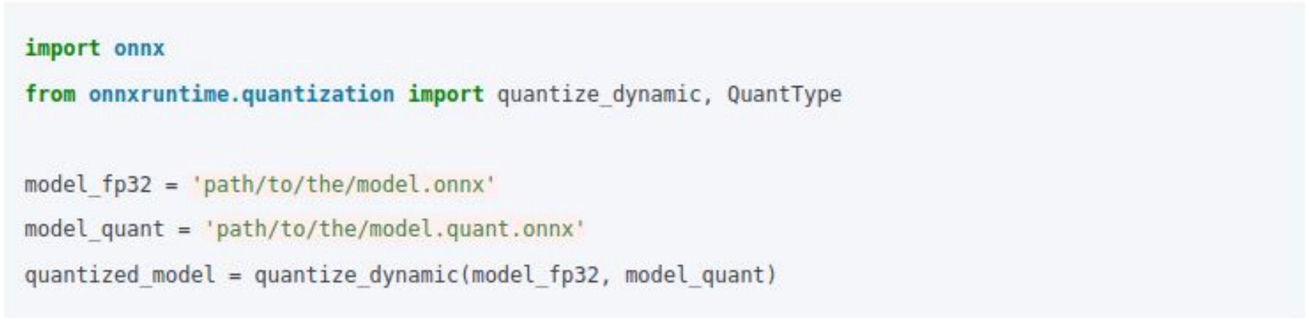


```
converter = tf.lite.TFLiteConverter.from_keras_model(q_aware_model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]

quantized_tflite_model = converter.convert()
```

- References
  - Quantization aware training comprehensive guide

Quantize ONNX models

- Dynamic quantization



```python
import onnx
from onnxruntime.quantization import quantize_dynamic, QuantType

model_fp32 = 'path/to/the/model.onnx'
model_quant = 'path/to/the/model.quant.onnx'
quantized_model = quantize_dynamic(model_fp32, model_quant)
```

References

- Quantize ONNX models | onnxruntime
- Static Quantization

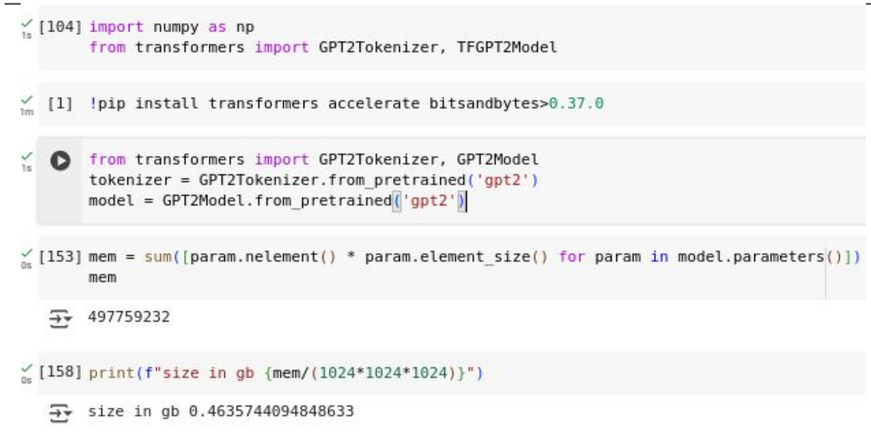
Bits and Bytes for Quantization

- Efficient 8-bit and 4-bit quantization of the models.
- Enables training large models on smaller GPUs.
- Commonly used with QLoRA to finetune quantized models.

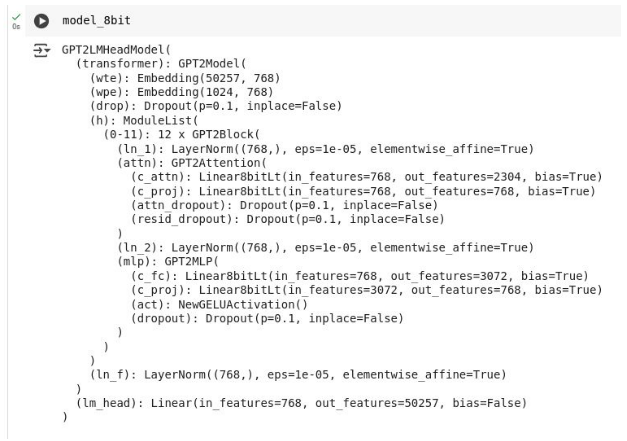
```
from transformers import AutoModelForCausalLM, BitsAndBytesConfig
quantization_config = BitsAndBytesConfig(load_in_8bit=True) # (load_in_4bit=True)
model_8bit = AutoModelForCausalLM.from_pretrained(
   "openai-community/gpt2",
   quantization_config=quantization_config
)
```

https://github.com/infocusp/model_quantization

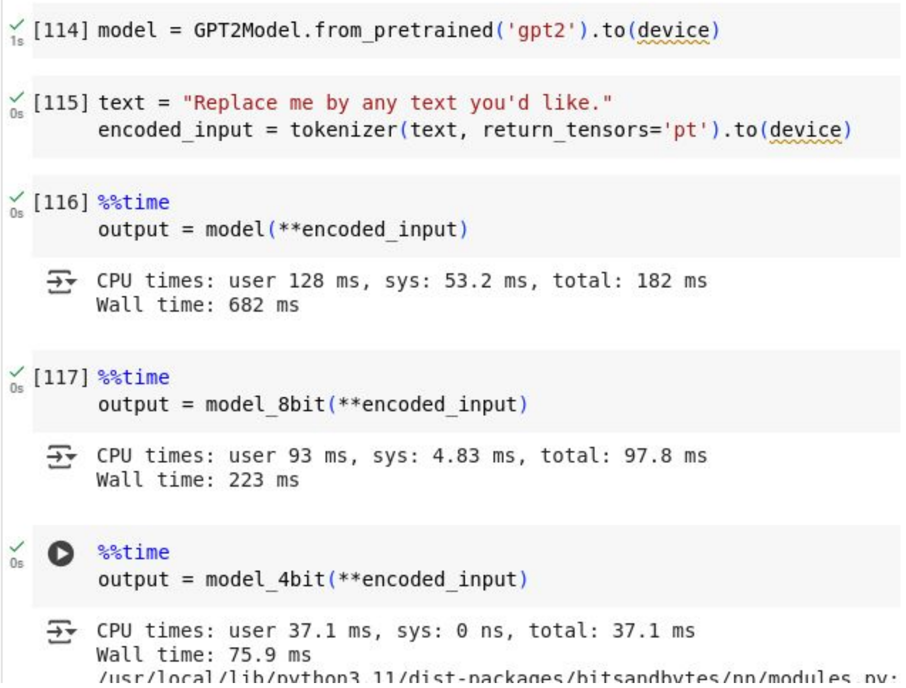
Bits and Bytes for Quantization



```
[104] import numpy as np
      from transformers import GPT2Tokenizer, TFGPT2Model

[1]  !pip install transformers accelerate bitsandbytes>0.37.0

      from transformers import GPT2Tokenizer, GPT2Model
      tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
      model = GPT2Model.from_pretrained('gpt2')

[153] mem = sum([param.nelement() * param.element_size() for param in model.parameters()])
      mem

      497759232

[158] print(f"size in gb {mem/(1024*1024*1024)}")

      size in gb 0.4635744094848633
```



```
model_8bit

GPT2LMHeadModel(
  (transformer): GPT2Model(
    (wte): Embedding(50257, 768)
    (wpe): Embedding(1024, 768)
    (drop): Dropout(p=0.1, inplace=False)
    (h): ModuleList(
      (0-11): 12 x GPT2Block(
        (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (attn): GPT2Attention(
          (c_attn): Linear8bitLt(in_features=768, out_features=2304, bias=True)
          (c_proj): Linear8bitLt(in_features=768, out_features=768, bias=True)
          (attn_dropout): Dropout(p=0.1, inplace=False)
          (resid_dropout): Dropout(p=0.1, inplace=False)
        )
        (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (mlp): GPT2MLP(
          (c_fc): Linear8bitLt(in_features=768, out_features=3072, bias=True)
          (c_proj): Linear8bitLt(in_features=3072, out_features=768, bias=True)
          (act): NewGELUActivation()
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
    )
    (ln_f): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
  )
  (lm_head): Linear(in_features=768, out_features=50257, bias=False)
)
```



```
[114] model = GPT2Model.from_pretrained('gpt2').to(device)

[115] text = "Replace me by any text you'd like."
      encoded_input = tokenizer(text, return_tensors='pt').to(device)

[116] %%time
      output = model(**encoded_input)

      CPU times: user 128 ms, sys: 53.2 ms, total: 182 ms
      Wall time: 682 ms

[117] %%time
      output = model_8bit(**encoded_input)

      CPU times: user 93 ms, sys: 4.83 ms, total: 97.8 ms
      Wall time: 223 ms

      %%time
      output = model_4bit(**encoded_input)

      CPU times: user 37.1 ms, sys: 0 ns, total: 37.1 ms
      Wall time: 75.9 ms
      /usr/local/lib/python3.11/dist-packages/bitsandbytes/nn/modules.py:
```

https://github.com/infocusp/model_quantization

Thank you