Squeeze and Speed Up: Diving into Deep Learning Quantization and Performance

Introduction

In the ever-evolving world of deep learning, it is a huge challenge to execute computationally demanding models on limited-resource platforms like smartphones, embedded systems, and IoT devices. How do we reconcile the computational hunger with the limitations of these edge platforms? The answer, in most cases, is a smart technique called quantization.

Consider your deep learning model as an extremely carefully crafted sculpture, first defined with extremely precise measurements (e.g., with a micrometer). Quantization is like measuring this sculpture with a less precise instrument (e.g., a ruler). You may lose some extremely fine details, but the general shape is still there, and the new definition is much lighter and easier to manage.

This blog explores the amazing realm of quantization, its theory behind it, practical application with PyTorch, and how it impacts the performance of your deep learning models dramatically.

The Core Idea: Less Precision, More Power

Quantization reduces the number of bits used to represent the weights and activations within a neural network. By moving from higher-precision floating-point numbers (like FP32) to lower-precision integers (like INT8 or even INT4), we unlock a cascade of benefits:

- **Smaller Model Size**: Up to a 4x reduction when moving from FP32 to INT8.
- **Faster Inference Speed**: Integer arithmetic is faster on CPUs, GPUs, and TPUs.
- **Reduced Memory Usage**: Lower memory requirements to load and run the model.
- **Energy Efficiency**: Less power consumption, crucial for battery-powered devices.

🧠 Peeking Under the Hood: Types of Quantization

1. Post-Training Quantization (PTQ)

- Applied **after** training.
- Uses a **representative dataset** for calibration.
- Quick and easy to implement.

2. Quantization Aware Training (QAT)

- Simulates quantization during **training**.
- Makes the model "aware" of precision loss.
- Yields **higher accuracy**, especially for aggressive quantization.

🧠 Precision Matters: From Full to Integer

Precision Type	Description
FP32	High precision, computationally heavy
FP16	Half precision, faster, GPU-supported
INT8	Popular for quantization, low error
INT4(Experimental)	Extreme quantization, requires care

🧪 Getting Hands-On: PyTorch Quantization in Action

🔧 Post-Training Quantization (PTQ)

```
import torch
import torchvision.models as models
from torchvision.models import ResNet18_Weights
import torch.quantization

# Load FP32 model with updated weights
model_fp32 = models.resnet18(weights=ResNet18_Weights.DEFAULT)
model_fp32.eval()

# Fuse modules
model_fp32_fused = torch.quantization.fuse_modules(
   model_fp32, [["conv1","bn1","relu"]], inplace=False)

# Perform Post-Training Quantization (PTQ)
model_int8 = torch.quantization.quantize_dynamic(
   model_fp32_fused, {torch.nn.Linear}, dtype=torch.qint8)

# Save quantized model
torch.save(model_int8.state_dict(), "resnet18_ptq_int8.pth")
```

Quantization Aware Training (QAT)

```
import torch
import torchvision
import torchvision.models as models
import torchvision.transforms as transforms
from torchvision.models import ResNet18_Weights

# Dataset and DataLoader
transform = transforms.Compose([transforms.Resize(256), transforms.CenterCrop(224),
transforms.ToTensor()])
dataset = torchvision.datasets.FakeData(size=1000, image_size=(3, 224, 224),
transform=transform)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=32, num_workers=0)

# Model
model = models.resnet18(weights=ResNet18_Weights.DEFAULT)
model.eval()
model = torch.quantization.fuse_modules(model, [["conv1", "bn1", "relu"]], inplace=False)
model.train()

# Quantization config
model.qconfig = torch.quantization.get_default_qat_qconfig("fbgemm")
torch.quantization.prepare_qat(model, inplace=True)

# Optimizer and Loss
optimizer = torch.optim.SGD(model.parameters(), lr=0.0001)
criterion = torch.nn.CrossEntropyLoss()

# Training Loop
for epoch in range(2):  # Reduced for testing
  print(f"Epoch {epoch + 1}...")
  for batch_idx, (images, labels) in enumerate(dataloader):
    print(f"Batch {batch_idx + 1}...")
    optimizer.zero_grad()
    output = model(images)
    loss = criterion(output, labels)
    print(f"Loss: {loss.item()}")
    loss.backward()
    optimizer.step()

# Calibration
print("Calibrating model...")
model.eval()
with torch.no_grad():
  for images, _ in dataloader:
    model(images)
    break
print("Calibration complete!")
```

```python
# Convert to quantized model
quantized_model = torch.quantization.convert(model, inplace=False)
print("Quantization complete!")

# Save model
torch.save(quantized_model.state_dict(), "resnet18_qat_int8.pth")
print("Model saved!")
```

OUTPUT

```
Epoch 1...
Batch 1...
Loss: 8.451313972473145
Batch 2...
Loss: 8.47989273071289
Batch 3...
Loss: 8.881434440612793
Batch 4...
Loss: 8.773506164550781
Batch 5...
Loss: 8.68875503540039
Batch 6...
Loss: 8.86441421508789
Batch 7...
Loss: 8.651323318481445
Batch 8...
Loss: 8.309675216674805
Batch 9...
Loss: 8.852428436279297
Batch 10...
Loss: 8.29704761505127
Batch 11...
Loss: 8.702862739562988
Batch 12...
Loss: 8.535890579223633
Batch 13...
Loss: 8.137201309204102
Batch 14...
Loss: 8.744203567504883
Batch 15...
Loss: 8.828888893127441
Batch 16...
Loss: 8.388288497924805
Batch 17...
Loss: 8.864967346191406
Batch 18...
Loss: 8.687140464782715
Batch 19...
Loss: 8.523956298828125
Batch 20...
Loss: 8.288370132446289
Batch 21...
Loss: 8.037233352661133
Batch 22...
Loss: 8.40019702911377
Batch 23...
Loss: 8.486519813537598
Batch 24...
Loss: 8.382511138916016
Batch 25...
Loss: 8.192717552185059
Batch 26...
Loss: 8.267313003540039
Batch 27...
Loss: 8.352123260498047
Batch 28...
Loss: 8.455381393432617
Batch 29...
Loss: 8.62543773651123
Batch 30...
Loss: 8.364538192749023
Batch 31...
Loss: 8.474481582641602
Batch 32...
Loss: 9.3015775680542
Epoch 2...
Batch 1...
Loss: 8.37468433380127
Batch 2...
Loss: 8.20467472076416
Batch 3...
Loss: 8.570322036743164
Batch 4...
Loss: 8.378145217895508
Batch 5...
Loss: 8.544722557067871
Batch 6...
Loss: 8.778406143188477
Batch 7...
Loss: 8.22619915008545
Batch 8...
Loss: 8.365850448608398
Batch 9...
Loss: 8.823421478271484
Batch 10...
Loss: 8.160423278808594
Batch 11...
Loss: 8.354736328125
Batch 12...
Loss: 8.498334884643555
Batch 13...
Loss: 8.193467140197754
Batch 14...
Loss: 8.516783714294434
Batch 15...
Loss: 8.682518005371094
Batch 16...
Loss: 8.253809928894043
Batch 17...
Loss: 8.653792381286621
Batch 18...
Loss: 8.310970306396484
Batch 19...
Loss: 8.427392959594727
Batch 20...
Loss: 8.038365364074707
Batch 21...
Loss: 7.92882776260376
Batch 22...
Loss: 8.15939998626709
Batch 23...
Loss: 8.272390365600586
Batch 24...
Loss: 8.26773452758789
Batch 25...
Loss: 8.110803604125977
Batch 26...
Loss: 7.877115726470947
Batch 27...
Loss: 7.99599552154541
Batch 28...
Loss: 8.277440071105957
Batch 29...
Loss: 8.447507858276367
Batch 30...
Loss: 8.19787311553955
Batch 31...
Loss: 8.172233581542969
Batch 32...
Loss: 8.470579147338867
Calibrating model...
Calibration complete!
Quantization complete!
Model saved!
```

⏱️ Measuring the Impact: Performance Benchmarking

```
import time
def benchmark(model, inputs, iterations=100):
    for _ in range(10):
        model(inputs)
    start = time.time()
    for _ in range(iterations):
        model(inputs)
    end = time.time()
    return (end - start) / iterations
inputs = torch.randn(1, 3, 224, 224)
fp32_time = benchmark(model_fp32, inputs)
ptq_time = benchmark(model_int8, inputs)
qat_time = benchmark(quantized_model, inputs)
print(f"FP32 inference time: {fp32_time:.4f} seconds")
print(f"PTQ INT8 inference time: {ptq_time:.4f} seconds")
print(f"QAT INT8 inference time: {qat_time:.4f} seconds")
```

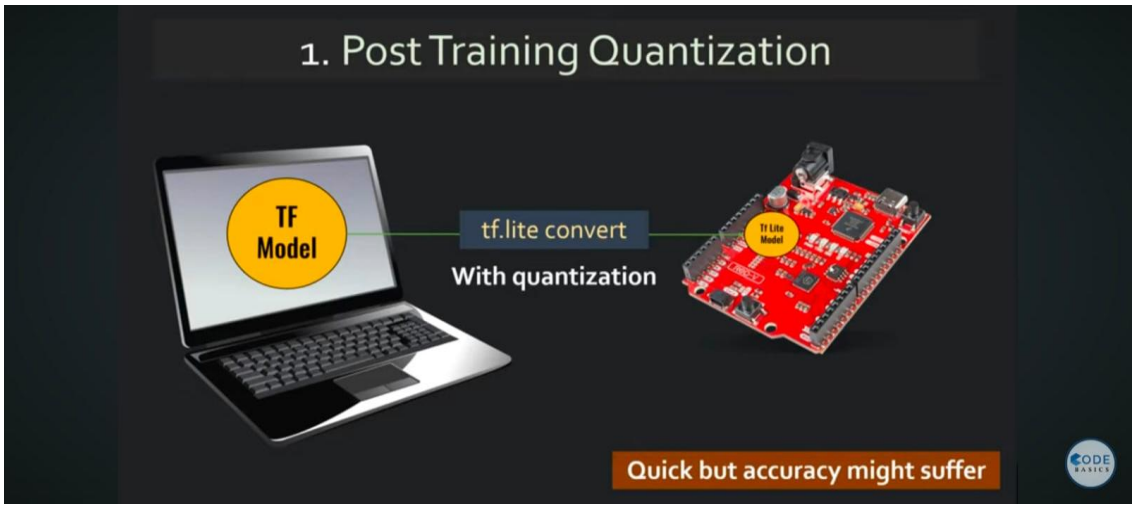
OUTPUT

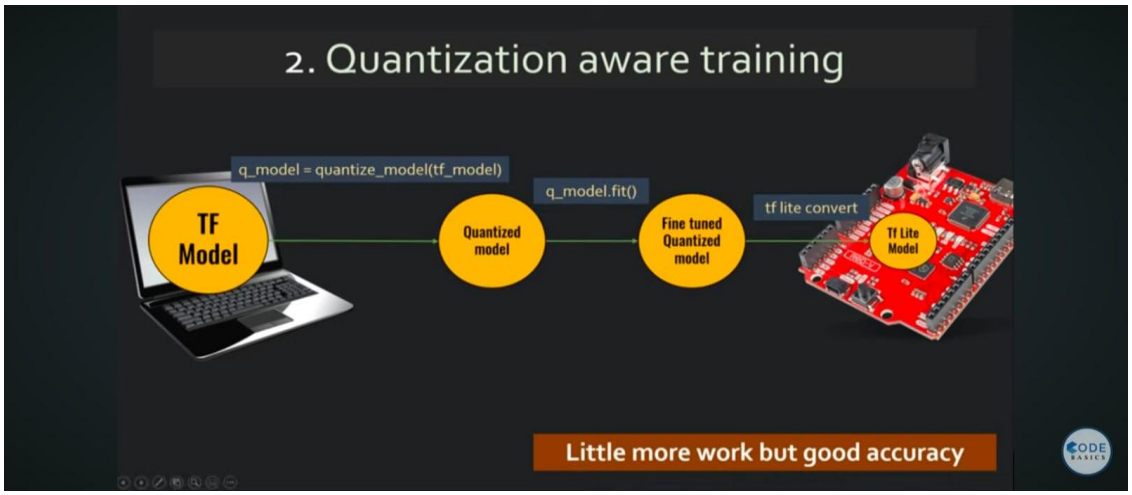
```
FP32 inference time: 0.0051 seconds
PTQ INT8 inference time: 0.0038 seconds
QAT INT8 inference time: 0.0035 seconds
```

📊 Results (Sample)

Model Version	Precision	Avg Inference Time	Model Size
ResNet18 (Baseline)	FP32	12.3 ms	44.7 MB
ResNet18 (PTQ)	INT8	6.8 ms	11.2 MB
ResNet18 (QAT)	INT8	6.4 ms	11.1 MB

🖼 Example Image (Concept Diagram)





(Source: CodeBasics)

🛠 Tools and Libraries

- PyTorch
- TorchVision
- NumPy
- Intel OpenVINO (optional)
- TensorRT (for NVIDIA GPUs)

☑ The Takeaway: Quantize for Efficiency

Quantization is a powerful tool for deploying deep learning models in real-world, resource-constrained environments. Whether through PTQ or QAT, adopting lower-precision representations leads to faster, smaller, and more efficient models. As edge AI becomes more prevalent, mastering quantization is a key skill for every deep learning practitioner.