

# Edge Deployment & Model Optimization

Week 12 · CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra

*IIT Gandhinagar*

# The Problem: Models Are Too Big

## Your trained model:

- ResNet-50: 100 MB
- BERT: 440 MB
- GPT-2: 1.5 GB

## Target devices:

- Smartphone: Limited memory, battery
- Raspberry Pi: 2-4 GB RAM
- Web browser: Size limits

**Challenge:** How do we run models on constrained devices?

# Edge vs Cloud Deployment

Aspect	Cloud	Edge
Compute	Unlimited	Limited
Latency	Network + Processing	Processing only
Privacy	Data leaves device	Data stays local
Cost	Per-request pricing	One-time hardware
Connectivity	Requires internet	Works offline

**Edge examples:** Phone apps, IoT sensors, in-car systems

# Why Deploy on Edge?

## 1. Speed

- No network latency
- Real-time predictions

## 2. Privacy

- Data never leaves the device
- GDPR/HIPAA compliance

## 3. Reliability

- Works without internet
- No server downtime issues

## 4. Cost

- No cloud compute costs
- Scales with devices, not users

# The Speed of Light Problem

**Physics limits cloud AI.** Light travels 300km per millisecond. A round trip to a server and back takes time no algorithm can reduce. For a self-driving car needing to react in 10ms, a 100ms network round-trip is fatal. Edge AI isn't just convenient - for some applications, it's the only option.

```
Cloud AI:  You → Network (50ms) → Server → Network (50ms) → Response
           Total: 100+ ms minimum
```

```
Edge AI:   You → Local Device → Response
           Total: 10 ms
```

```
Self-driving car at 60mph:
```

```
100ms = 2.7 meters traveled (too late to brake!)
```

```
10ms  = 0.27 meters (can still react)
```

# Model Optimization Techniques

Technique	Size Reduction	Speed Up	Accuracy Loss
Quantization	4x smaller	2 - 4x faster	< 1%
Pruning	2x smaller	1.5 - 2x faster	1 - 2%
Distillation	Varies	Varies	Can match original

**The good news:** You can often get 4x smaller AND faster with minimal accuracy loss!

# Quantization: The Big Idea

## Normal models use 32-bit floats:

- Each weight: 32 bits
- High precision, but large

## Quantized models use 8-bit integers:

- Each weight: 8 bits
- 4x smaller, faster on CPUs

Float32: 32 bits per weight

Int8: 8 bits per weight → 4x compression!

# The Precision Intuition

**Do you really need 9 decimal places?** A weight of 0.234567891 vs 0.23 - does it matter? For most neural networks, the answer is "barely." The model learned approximate patterns, not exact numbers. Quantization exploits this: we trade precision we don't need for speed and size we do need.

Precision	Example Value	Use Case
Float32 (full)	0.234567891...	Training
Float16	0.2346	GPU inference
Int8	$60/255 \approx 0.24$	Edge deployment

**The key insight:** Neural networks are surprisingly robust to reduced precision.



# Quantization Example

## Before (Float32):

```
weights = [0.234, -0.567, 0.891, ...] # 32 bits each  
model_size = 100 MB
```

## After (Int8):

```
weights = [45, -127, 95, ...] # 8 bits each  
model_size = 25 MB # 4x smaller!
```

## The math:

- Find min/max of weights
- Scale to 0-255 range
- Store as integers

# Types of Quantization

## 1. Post-Training Quantization (PTQ)

- Train model normally (Float32)
- Convert to Int8 after training
- Quick and easy

## 2. Quantization-Aware Training (QAT)

- Simulate quantization during training
- Model learns to handle lower precision
- Better accuracy, more effort

**For most cases:** PTQ is good enough!

# Quantization in PyTorch

## Dynamic quantization (easiest):

```
import torch

# Original model
model = MyModel()
model.load_state_dict(torch.load("model.pth"))
model.eval()

# Quantize
quantized_model = torch.quantization.quantize_dynamic(
    model,
    {torch.nn.Linear}, # Layers to quantize
    dtype=torch.qint8
)

# Save
torch.save(quantized_model.state_dict(), "model_quantized.pth")
```

# Checking Model Size

```
import os

def get_model_size(path):
    """Get model size in MB."""
    size = os.path.getsize(path) / (1024 * 1024)
    return f"{size:.1f} MB"

print(f"Original: {get_model_size('model.pth')}")
print(f"Quantized: {get_model_size('model_quantized.pth')}")

# Output:
# Original: 100.0 MB
# Quantized: 25.2 MB
```

# Pruning: Remove Useless Weights

**Observation:** Many weights in neural networks are close to zero.

**Idea:** Remove them!

```
Before pruning: [0.9, 0.01, -0.8, 0.001, 0.7]
```

```
After 40% pruning: [0.9, 0, -0.8, 0, 0.7]
```

**Benefits:**

- Smaller model
- Faster inference (fewer multiplications)

# Pruning in PyTorch

```
import torch.nn.utils.prune as prune

# Prune 30% of weights (smallest magnitudes)
prune.l1_unstructured(
    model.fc1,          # Layer to prune
    name='weight',
    amount=0.3          # Remove 30%
)

# Make pruning permanent
prune.remove(model.fc1, 'weight')

# Check sparsity
zeros = (model.fc1.weight == 0).sum()
total = model.fc1.weight.numel()
print(f"Sparsity: {zeros/total:.1%}")
```

# Knowledge Distillation

**Idea:** Train a small "student" model to mimic a large "teacher" model.

```
Teacher (Large): 100 MB, 95% accuracy
    ↓ Knowledge Transfer
Student (Small): 10 MB, 93% accuracy
```

## Why it works:

- Student learns from teacher's "soft" outputs
- More information than hard labels
- Can get near-teacher accuracy with smaller model

# The Teacher's Soft Knowledge

**Hard labels throw away information.** A label saying "cat" tells you nothing about how cat-like vs dog-like an image is. But a teacher saying "90% cat, 8% dog, 2% fox" reveals structure - cats and dogs are similar, cats and airplanes aren't. The student learns these relationships, not just the final answer.

	Hard Label	Soft Label (Teacher)
Image of fluffy cat:	"cat"	cat:0.90, dog:0.08, fox:0.02
	↑	↑
	No nuance!	"This looks a bit dog-like too"

Student learns: Cats and dogs have similar features (fur, ears, etc.)  
Cats and airplanes are nothing alike



# Distillation: Simple Example

```
import torch.nn.functional as F

def distillation_loss(student_logits, teacher_logits, labels, T=3, alpha=0.5):
    # Hard loss: student vs true labels
    hard_loss = F.cross_entropy(student_logits, labels)

    # Soft loss: student vs teacher (with temperature)
    soft_student = F.log_softmax(student_logits / T, dim=1)
    soft_teacher = F.softmax(teacher_logits / T, dim=1)
    soft_loss = F.kl_div(soft_student, soft_teacher)

    # Combine
    return alpha * hard_loss + (1 - alpha) * soft_loss * T * T
```

# ONNX: Universal Model Format

**Problem:** You trained in PyTorch, but want to deploy on mobile/web.

**Solution:** ONNX (Open Neural Network Exchange)

- Standard format for neural networks
- Export from PyTorch, TensorFlow, etc.
- Run on any platform

PyTorch Model → ONNX → ONNX Runtime → Any Device

# Exporting to ONNX

```
import torch

# Load model
model = MyModel()
model.load_state_dict(torch.load("model.pth"))
model.eval()

# Dummy input (same shape as real input)
dummy_input = torch.randn(1, 3, 224, 224)

# Export
torch.onnx.export(
    model,
    dummy_input,
    "model.onnx",
    input_names=['image'],
    output_names=['prediction'],
    dynamic_axes={'image': {0: 'batch_size'}} # Variable batch
)

print("Exported to model.onnx")
```

# Running with ONNX Runtime

```
import onnxruntime as ort
import numpy as np

# Load ONNX model
session = ort.InferenceSession("model.onnx")

# Prepare input
input_data = np.random.randn(1, 3, 224, 224).astype(np.float32)

# Run inference
outputs = session.run(
    None, # Get all outputs
    {'image': input_data}
)

print(f"Prediction: {outputs[0]}")
```

**Benefits:** 2-3x faster than PyTorch on CPU!

# ONNX Optimizations

ONNX Runtime automatically applies:

1. **Operator fusion**: Combine Conv + BatchNorm + ReLU into one
2. **Constant folding**: Pre-compute constants
3. **Memory optimization**: Reuse buffers

```
Before: Conv → BatchNorm → ReLU (3 operations)
After:  ConvBNReLU (1 operation)
```

# TensorFlow Lite (TFLite)

For mobile deployment (Android/iOS):

```
import tensorflow as tf

# Convert to TFLite
converter = tf.lite.TFLiteConverter.from_saved_model('model')
converter.optimizations = [tf.lite.Optimize.DEFAULT] # Quantize
tflite_model = converter.convert()

# Save
with open('model.tflite', 'wb') as f:
    f.write(tflite_model)
```

TFLite is optimized for:

- ARM processors (phones)
- Edge TPU accelerators
- Microcontrollers

# Choosing the Right Approach

Scenario	Recommended Approach
Quick optimization	Quantization (PTQ)
Maximum compression	Quantization + Pruning
Best accuracy	Knowledge distillation
Mobile app	TensorFlow Lite
Cross-platform	ONNX Runtime
Web browser	ONNX + WebAssembly

**Start with quantization** - it's the easiest and most effective!

# Benchmarking Your Model

```
import time
import numpy as np

def benchmark(model, input_data, n_runs=100):
    """Measure average inference time."""
    # Warmup
    for _ in range(10):
        _ = model(input_data)

    # Benchmark
    times = []
    for _ in range(n_runs):
        start = time.perf_counter()
        _ = model(input_data)
        times.append(time.perf_counter() - start)

    avg_time = np.mean(times) * 1000 # ms
    print(f"Average: {avg_time:.2f} ms")
    print(f"Throughput: {1000/avg_time:.1f} samples/sec")
```



# Before vs After Optimization

Metric	Original	Optimized
Size	100 MB	25 MB
Latency	50 ms	12 ms
Memory	400 MB	100 MB
Accuracy	95.0%	94.5%

**Trade-off:** 0.5% accuracy for 4x smaller and 4x faster!

# Deployment Pipeline

```
Train Model (Float32)
  ↓
Prune (optional)
  ↓
Quantize (Int8)
  ↓
Export (ONNX/TFLite)
  ↓
Benchmark on target device
  ↓
Deploy
```

# Common Deployment Targets

## 1. Mobile Apps

- Use TensorFlow Lite or Core ML (iOS)
- Optimize for ARM processors
- Consider battery usage

## 2. Web Browser

- Use ONNX.js or TensorFlow.js
- Models must be small (< 10 MB)
- Use WebGL for acceleration

## 3. Embedded/IoT

- Use TensorFlow Lite Micro
- Very limited memory (KB, not MB)
- May need specialized models

# Real-World Example: Mobile App

**Original model:** ResNet-50

- Size: 98 MB
- Latency: 200 ms

**Optimization steps:**

1. Replace with MobileNet-v2 (smaller architecture)
2. Quantize to Int8
3. Export to TFLite

**Optimized model:**

- Size: 3.4 MB
- Latency: 30 ms
- Accuracy: 71% (vs 76% for ResNet)

# Efficient Model Architectures

## Designed for mobile/edge:

Model	Size	Top - 1 Accuracy	Latency
MobileNet - v2	3.4 MB	71.8%	30 ms
EfficientNet - B0	5.3 MB	77.1%	45 ms
SqueezeNet	1.2 MB	57.5%	25 ms

## vs. Desktop models:

| ResNet-50 | 98 MB | 76.1% | 200 ms |  
| VGG-16 | 528 MB | 71.5% | 400 ms |

# Tips for Edge Deployment

## 1. Start with a smaller model

- MobileNet instead of ResNet
- DistilBERT instead of BERT

## 2. Always quantize

- Easy 4x size reduction
- Often 2-4x speed improvement

## 3. Profile on target device

- Desktop performance  $\neq$  Mobile performance
- Test on actual hardware

## 4. Consider accuracy trade-offs

- 1-2% accuracy loss is usually acceptable
- Test with your specific use case

# Summary

Technique	What it does	When to use
Quantization	32-bit → 8-bit	Always (first step)
Pruning	Remove small weights	Need more compression
Distillation	Train smaller model	Can afford retraining
ONNX	Cross-platform format	Non-Python deployment
TFLite	Mobile format	Android/iOS apps

# Lab Preview

## This week you'll:

1. Benchmark your model's size and speed
2. Apply quantization and measure improvement
3. Try pruning and compare results
4. Export to ONNX format
5. Run with ONNX Runtime
6. Compare all approaches

**Result:** An optimized model ready for edge deployment!



# Questions?

## Key takeaways:

- Quantization is the easiest win (4x smaller, 2-4x faster)
- ONNX enables cross-platform deployment
- Start with efficient architectures when possible
- Always benchmark on target hardware

**Next week:** Profiling & Performance