

# Deployment on Constrained Devices

**CS 203: Software Tools and Techniques for AI**

Prof. Nipun Batra, IIT Gandhinagar



# The "Edge" Challenge

**Scenario:** You trained a ResNet-50. It's 100MB.  
You want to run it on a Raspberry Pi or a Mobile Phone.

## **Constraints:**

1. **Memory:** Device has 2GB RAM, model needs 4GB.
2. **Latency:** Inference takes 5s, user needs <100ms.
3. **Power:** GPU drains battery in 20 mins.
4. **Storage:** App limit is 50MB.

**Solution:** Model Optimization.



# Edge vs Cloud Deployment

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

## Use cases for Edge:

- Real-time (AR/VR, autonomous vehicles)
- Privacy-sensitive (medical, personal data)
- Offline scenarios (rural areas, airplanes)



# Hardware Constraints

## Mobile devices (phones, tablets):

- CPU: ARM-based (different instruction set)
- RAM: 2-8GB
- Storage: Limited app size
- Power: Battery-constrained

## IoT devices (Raspberry Pi, Arduino):

- CPU: Very limited (1-4 cores, < 2GHz)
- RAM: 512MB - 4GB
- No GPU or NPU in many cases

## Edge servers (NVIDIA Jetson):



# Model Optimization Taxonomy

## Size reduction:

1. **Quantization:** Lower precision (FP32 → INT8)
2. **Pruning:** Remove weights
3. **Knowledge Distillation:** Train smaller model
4. **Low-rank factorization:** Decompose weight matrices

## Speed optimization:

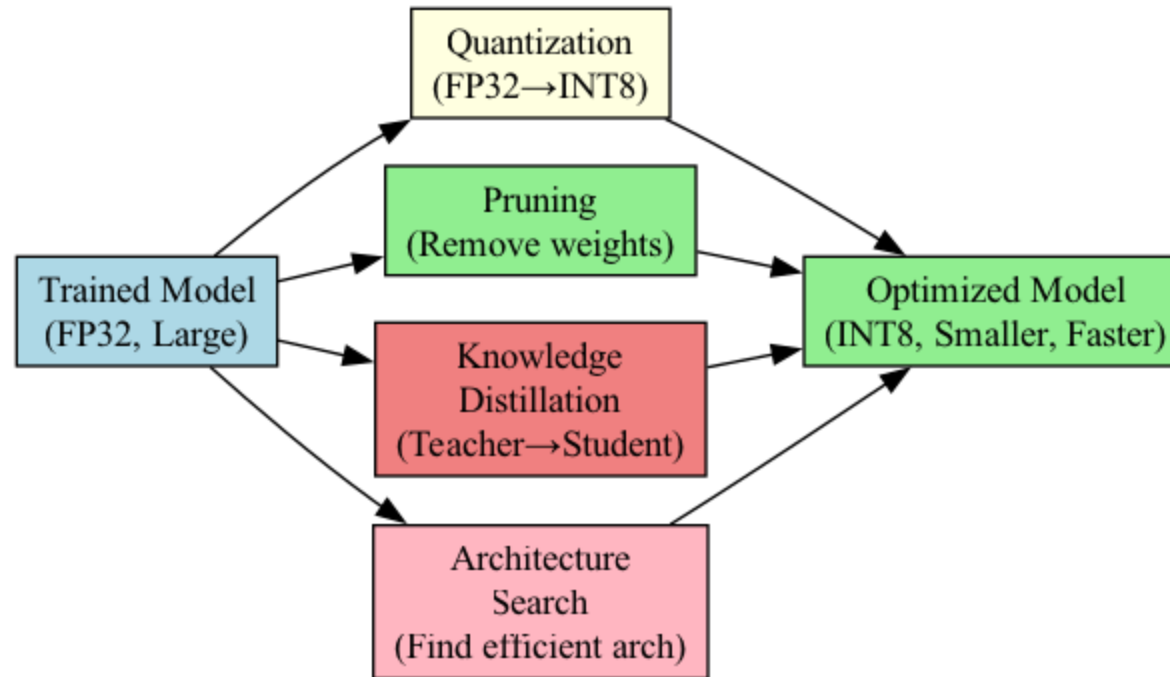
5. **Operator fusion:** Merge operations
6. **Graph optimization:** Simplify computation graph
7. **Hardware acceleration:** Use specialized chips

## Architecture design:

8. **Neural Architecture Search (NAS):** Find efficient architectures



# Techniques Overview



## Today's Focus:

1. **Quantization:** Lower precision math
2. **Pruning:** Removing useless connections
3. **Knowledge Distillation:** Train smaller student model



# Optimization Techniques: Comparison

Technique	Size Reduction	Speed Improvement	Accuracy Impact	Implementation Complexity	When to Use
Quantization (INT8)	4x (FP32→INT8)	2-4x faster	Minimal (<1%)	Low	Always (first step)
Pruning (50%)	2x (half weights)	1.5-2x faster	Small (1-2%)	Medium	After quantization
Knowledge Distillation	Depends on student size	Significant	Can match teacher	High	When you can retrain
NAS	Variable	Variable	Can improve	Very High	Research/complex domains



# Quantization: Theory

**Standard Training:** Float32 (32-bit floating point).

**Quantization:** Convert to Int8 (8-bit integer).

**Formula:**

$$Q(x) = \text{round} \left( \frac{x}{S} + Z \right)$$

- $S$ : Scale (range / 255)
- $Z$ : Zero-point (offset)

**Impact:**

- **Size:** 32 bits  $\rightarrow$  8 bits = **4x reduction**
- **Speed:** Integer math faster on CPUs (2-4x speedup)
- **Accuracy:** Minimal drop (<1%) for robust models



# Quantization: Detailed Example

Float32 weights: `[-2.5, -1.0, 0.0, 1.5, 3.0]`

Quantization process:

```
# 1. Find min/max
min_val, max_val = -2.5, 3.0

# 2. Calculate scale
scale = (max_val - min_val) / 255 # 0.0216

# 3. Calculate zero-point
zero_point = -int(min_val / scale) # 116

# 4. Quantize
quantized = round(weights / scale + zero_point)
# Result: [0, 70, 116, 185, 255]
```

Dequantize: `(quantized - zero_point) * scale`



# Types of Quantization

## 1. Post-Training Quantization (PTQ):

- Train normal FP32 model
- Calibrate with small dataset (~1000 samples)
- Convert to INT8
- **Pros:** Easy, no retraining
- **Cons:** Slight accuracy drop

## 2. Quantization-Aware Training (QAT):

- Simulate quantization during training
- Model learns to adapt to lower precision
- **Pros:** Best accuracy



# Quantization Granularity

## Per-tensor quantization:

- Single scale for entire tensor
- Faster but less accurate

## Per-channel quantization:

- Separate scale per conv channel
- Better accuracy

## Example:

```
import torch

# Per-tensor
model_int8 = torch.quantization.quantize_dynamic(
    model, {torch.nn.Linear}, dtype=torch.qint8
```



# Pruning Theory

**Observation:** Neural networks are over-parameterized.

- Many weights near zero
- Removing them barely affects accuracy

**Magnitude-based pruning:**

1. Rank weights by absolute value
2. Remove smallest X% (e.g., 50%)
3. Fine-tune to recover accuracy

**Structured vs Unstructured:**

- **Unstructured:** Remove individual weights → sparse matrices
- **Structured:** Remove entire channels/neurons → smaller dense matrices



# Pruning Strategies

## One-shot pruning:

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

# Prune 30% of weights in layer
prune.l1_unstructured(module.conv1, name='weight', amount=0.3)
```

## Iterative pruning (better accuracy):

1. Train to convergence
2. Prune small %
3. Fine-tune
4. Repeat

## Lottery Ticket Hypothesis:



# Structured Pruning Example

Remove entire filters:

```
# Prune 40% of filters in conv layer
prune.ln_structured(
    module.conv1,
    name="weight",
    amount=0.4,
    n=2,      # L2 norm
    dim=0     # Filter dimension
)
```

Benefits:

- Actually reduces computation (not just params)
- No special hardware support needed
- Works well with quantization



# Knowledge Distillation

**Idea:** Compress knowledge from large "teacher" to small "student".

**Process:**

1. Train large teacher model (high accuracy)
2. Use teacher's soft outputs as targets
3. Train small student to mimic teacher

**Loss function:**

$$L = \alpha \cdot L_{CE}(y, \hat{y}) + (1 - \alpha) \cdot L_{KD}(T_{teacher}, T_{student})$$

**Why it works:**

- Soft targets contain more information than hard labels
- Student learns nuances from teacher



# Knowledge Distillation Code

```
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)
    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, reduction='batchmean') * T*T

    # Combined loss
    return alpha * hard_loss + (1 - alpha) * soft_loss
```

**Temperature (T):** Higher = softer probabilities.



# Neural Architecture Search (NAS)

**Goal:** Automatically find efficient architectures.

**Search space:**

- Number of layers
- Layer types (conv, pooling, skip)
- Kernel sizes, channels

**Search strategy:**

1. **Random search:** Try random architectures
2. **Reinforcement learning:** RL agent proposes architectures
3. **Gradient-based:** DARTS (differentiable)

**Hardware-aware NAS:** Optimize for specific device constraints.



# Efficient Architecture Families

## MobileNet (Google):

- Depthwise separable convolutions
- 9x fewer parameters than VGG

## EfficientNet (Google):

- Compound scaling (depth + width + resolution)
- State-of-art accuracy/efficiency trade-off

## SqueezeNet:

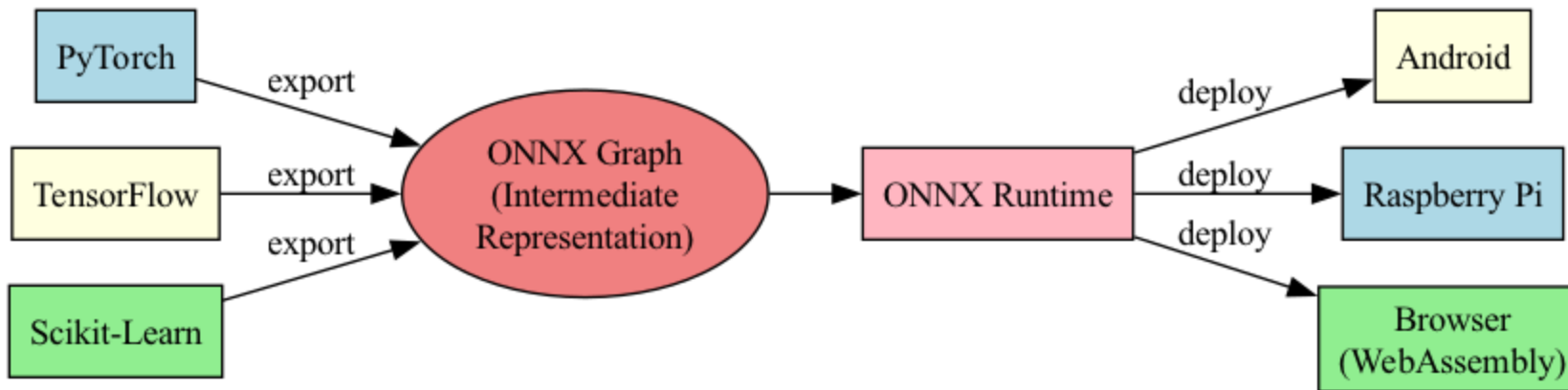
- Fire modules (squeeze + expand)
- 50x smaller than AlexNet

## TinyML architectures:



# ONNX: Open Neural Network Exchange

## The Universal Bridge



## Why use it?

- **Interoperability:** Train in PyTorch, deploy in C++
- **Optimization:** Graph-level optimizations (fusion, constant folding)
- **Hardware support:** CPU, GPU, mobile accelerators



# ONNX Export Example

```
import torch
import torch.onnx

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

# Create dummy input
dummy_input = torch.randn(1, 3, 224, 224)

# Export to ONNX
torch.onnx.export(
    model,
    dummy_input,
    "model.onnx",
    export_params=True,
    opset_version=14,
    input_names=['input'],
    output_names=['output'],
    dynamic_axes={'input': {0: 'batch_size'}} # Variable batch size
)
```



# ONNX Runtime Inference

```
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, # Output names (None = all outputs)
    {'input': input_data}
)

print(outputs[0]) # Prediction
```

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



# ONNX Graph Optimizations

## Operator fusion:

- Conv + BatchNorm + ReLU → Single fused op
- Reduces memory bandwidth

## Constant folding:

- Pre-compute constants at export time

## Dead code elimination:

- Remove unused branches

## Quantization:

- ONNX Runtime supports INT8 quantization

## Graph example:



# Hardware Acceleration Options

## Mobile (iOS/Android):

- **CoreML** (Apple): Optimized for iPhone/iPad
- **TensorFlow Lite**: Cross-platform mobile
- **ONNX Mobile**: ONNX Runtime for mobile

## Edge TPU (Google Coral):

- Hardware accelerator for INT8 models
- 4 TOPS (trillion operations/sec)

## NVIDIA Jetson:

- TensorRT for GPU optimization
- Mixed precision (FP16)



# TensorFlow Lite Conversion

```
import tensorflow as tf

# Convert Keras model to TFLite
converter = tf.lite.TFLiteConverter.from_keras_model(model)

# Enable optimizations
converter.optimizations = [tf.lite.Optimize.DEFAULT]

# Quantize to INT8
converter.representative_dataset = representative_data_gen
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]

# Convert
tflite_model = converter.convert()

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



# TensorRT (NVIDIA): Overview

High-performance inference engine for NVIDIA GPUs

Key optimizations:

- **Layer fusion:** Combine operations (Conv+BN+ReLU → single kernel)
- **Precision calibration:** Automatic FP32 → FP16 → INT8 conversion
- **Kernel auto-tuning:** Select fastest GPU kernels for your model
- **Memory optimization:** Reduce memory footprint

Performance gains:

- Typical speedup: 2-5x over PyTorch
- Latency reduction: 50-80% for large models
- Best for production deployment on NVIDIA GPUs



# TensorRT: Implementation

```
import tensorrt as trt

# 1. Create builder and network
builder = trt.Builder(TRT_LOGGER)
network = builder.create_network()

# 2. Parse ONNX model
parser = trt.OnnxParser(network, TRT_LOGGER)
parser.parse_from_file("model.onnx")

# 3. Configure optimization
config = builder.create_builder_config()
config.set_flag(trt.BuilderFlag.FP16) # Enable FP16

# 4. Build optimized engine
engine = builder.build_engine(network, config)
```

**Next step:** Save engine and load for inference



# Benchmarking Models

## Metrics to measure:

1. **Latency:** Time per inference (ms)
2. **Throughput:** Inferences per second
3. **Memory usage:** Peak RAM (MB)
4. **Model size:** Disk space (MB)
5. **Energy consumption:** Battery drain (mAh)

## Tools:

- `torch.utils.benchmark` (PyTorch)
- `time` module (simple timing)
- `memory_profiler` (RAM usage)



# Latency Benchmarking Code

```
import time
import torch

model.eval()
input_data = torch.randn(1, 3, 224, 224)

# Warmup (JIT compilation, cache warming)
for _ in range(10):
    _ = model(input_data)

# Benchmark
times = []
for _ in range(100):
    start = time.time()
    with torch.no_grad():
        _ = model(input_data)
    times.append(time.time() - start)

print(f"Mean latency: {np.mean(times)*1000:.2f} ms")
print(f"Std latency: {np.std(times)*1000:.2f} ms")
print(f"P95 latency: {np.percentile(times, 95)*1000:.2f} ms")
```



# Accuracy vs Efficiency Trade-off

**Pareto frontier:** No single "best" model.

Model	Size	Latency	Accuracy
ResNet-50	98MB	50ms	76%
+ Pruning 50%	49MB	40ms	75%
+ Quantization INT8	12MB	15ms	74%
MobileNet-v2	14MB	10ms	72%
+ Quantization	3.5MB	5ms	71%

**Choose based on constraints:**

- Strict latency → MobileNet quantized
- High accuracy needed → ResNet pruned



# Deployment Checklist

## Pre-deployment:

- ☐ Model optimized (quantized/pruned)
- ☐ Benchmarked on target device
- ☐ Accuracy validated
- ☐ Error handling implemented

## Deployment:

- ☐ Model packaged (ONNX, TFLite, etc.)
- ☐ Inference code tested
- ☐ Fallback strategy (cloud API)

## Post-deployment:



# Summary

## Key techniques:

1. **Quantization**: 4x smaller, 2-4x faster
2. **Pruning**: Remove redundant weights
3. **Knowledge Distillation**: Train small student model
4. **ONNX**: Universal deployment format
5. **Hardware acceleration**: TensorRT, CoreML, TFLite

## Typical pipeline:

Train (FP32) → Prune → Quantize (INT8) → Export (ONNX) → Deploy

**Lab**: Hands-on optimization and benchmarking!



# Additional Resources

## Libraries:

- PyTorch quantization: <https://pytorch.org/docs/stable/quantization.html>
- ONNX: <https://onnx.ai/>
- TensorFlow Lite: <https://www.tensorflow.org/lite>

## Papers:

- "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference" (Google)
- "The Lottery Ticket Hypothesis" (MIT)
- "Distilling the Knowledge in a Neural Network" (Hinton et al.)

## Hardware: