Deployment on Constrained Devices - Lab

CS 203: Software Tools and Techniques for AI

Duration: 3 hours

Lab Objectives

By the end of this lab, you will:

1. Quantize models to reduce size by 4x
2. Prune models to remove redundant weights
3. Export models to ONNX format
4. Benchmark latency and accuracy trade-offs
5. Deploy optimized models

Prerequisites:

```
pip install torch torchvision onnx onnxruntime numpy matplotlib
```

Part 1: Baseline Measurement

Understanding what we're optimizing

Exercise 1.1: Load and Benchmark Model (20 min)

Goal: Establish baseline metrics

Create `baseline.py`:

```
import torch
import torchvision.models as models
import time
import os

# Load pre-trained ResNet-18
model = models.resnet18(pretrained=True)
model.eval()

# Save model
torch.save(model.state_dict(), 'resnet18_fp32.pth')

# Measure size
size_mb = os.path.getsize('resnet18_fp32.pth') / (1024 * 1024)
print(f"Model size: {size_mb:.2f} MB")

# Count parameters
```

Exercise 1.2: Measure Latency (15 min)

Add to `baseline.py`:

```
import numpy as np

# Prepare dummy input
input_data = torch.randn(1, 3, 224, 224)

# Warmup
for _ in range(10):
    with torch.no_grad():
        _ = 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")
```

Part 1 Checkpoint

Record your baseline:

- [ ] Model size (MB)
- [ ] Mean latency (ms)
- [ ] P95 latency (ms)
- [ ] Total parameters

These are your targets to beat!

Part 2: Quantization

Reduce precision for 4x compression

Exercise 2.1: Dynamic Quantization (25 min)

Goal: Quantize model without calibration

Create `quantize_dynamic.py`:

```
import torch
import torchvision.models as models

# Load model
model = models.resnet18(pretrained=True)
model.eval()

# Apply dynamic quantization
quantized_model = torch.quantization.quantize_dynamic(
    model,
    {torch.nn.Linear, torch.nn.Conv2d},  # Layers to quantize
    dtype=torch.qint8
)

# Save quantized model
torch.save(quantized_model.state_dict(), 'resnet18_int8_dynamic.pth')

# Measure size
import os
size_mb = os.path.getsize('resnet18_int8_dynamic.pth') / (1024 * 1024)
print(f"Quantized model size: {size_mb:.2f} MB")

# Calculate compression ratio
```

Exercise 2.2: Benchmark Quantized Model (15 min)

Add benchmarking:

```
import time
import numpy as np

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

# Warmup
for _ in range(10):
    with torch.no_grad():
        _ = quantized_model(input_data)

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

print(f"Quantized latency: {np.mean(times)*1000:.2f} ms")

# Calculate speedup
baseline_latency = 30  # Replace with your baseline
speedup = baseline_latency / (np.mean(times)*1000)
print(f"Speedup: {speedup:.2f}x")
```

Exercise 2.3: Static Quantization (30 min)

Goal: Better accuracy with calibration

```
from torch.quantization import get_default_qconfig, prepare, convert

# Load model
model = models.resnet18(pretrained=True)
model.eval()

# Set quantization config
model.qconfig = get_default_qconfig('fbgemm')

# Prepare for quantization (insert observers)
model_prepared = prepare(model)

# Calibrate with representative data
calibration_data = [torch.randn(1, 3, 224, 224) for _ in range(100)]

for data in calibration_data:
    model_prepared(data)

# Convert to quantized model
model_quantized = convert(model_prepared)

# Save
torch.jit.save(torch.jit.script(model_quantized), 'resnet18_static_int8.pth')
```

Part 2 Checkpoint

Compare quantization methods:

Method	Size	Latency	Compression	Speedup
Baseline FP32	45MB	30ms	1x	1x
Dynamic INT8	?	?	?	?
Static INT8	?	?	?	?

Fill in your measurements!

Part 3: Pruning

Remove redundant weights

Exercise 3.1: Unstructured Pruning (30 min)

Goal: Prune 30% of weights

Create `prune_model.py`:

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

# Load model
model = models.resnet18(pretrained=True)

# Prune 30% of weights in all Conv2d layers
for name, module in model.named_modules():
    if isinstance(module, torch.nn.Conv2d):
        prune.l1_unstructured(module, name='weight', amount=0.3)

# Make pruning permanent
for name, module in model.named_modules():
    if isinstance(module, torch.nn.Conv2d):
        prune.remove(module, 'weight')

# Count zero weights
total_zeros = 0
total_params = 0

for param in model.parameters():
    total_params += param.numel()
    total_zeros += (param == 0).sum().item()
```

Exercise 3.2: Structured Pruning (25 min)

Goal: Remove entire filters

```
# Structured pruning (remove filters)
for name, module in model.named_modules():
    if isinstance(module, torch.nn.Conv2d):
        prune.ln_structured(
            module,
            name="weight",
            amount=0.3,
            n=2,       # L2 norm
            dim=0      # Filter dimension
        )
        prune.remove(module, 'weight')

# Model is now smaller (fewer filters)
print(f"Pruned parameters: {sum(p.numel() for p in model.parameters()):,}")
```

Note: Structured pruning actually reduces computation, not just parameters.

Exercise 3.3: Pruning + Fine-tuning (Optional, 30 min)

Goal: Recover accuracy after pruning

```
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

# Prepare training data
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

train_dataset = datasets.ImageFolder('path/to/imagenet/train', transform=transform)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)

# Fine-tune for a few epochs
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
criterion = torch.nn.CrossEntropyLoss()

model.train()
for epoch in range(2):  # Just 2 epochs
    for images, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(images)
```

Part 3 Checkpoint

Pruning results:

- Sparsity achieved: ?%
- Model size after pruning: ?MB
- Latency after pruning: ?ms

Question: Why doesn't unstructured pruning reduce latency much?

Answer: Sparse matrix operations need special hardware support.

Part 4: ONNX Export

Universal deployment format

Exercise 4.1: Export to ONNX (20 min)

Goal: Convert PyTorch model to ONNX

Create `export_onnx.py`:

```
import torch
import torchvision.models as models

# Load model
model = models.resnet18(pretrained=True)
model.eval()

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

# Export to ONNX
torch.onnx.export(
    model,
    dummy_input,
    "resnet18.onnx",
    export_params=True,
    opset_version=14,
    do_constant_folding=True,  # Optimize constants
    input_names=['input'],
    output_names=['output'],
    dynamic_axes={
        'input': {0: 'batch_size'},
        'output': {0: 'batch_size'}
    }
)

print("Model exported to resnet18.onnx")

# Check ONNX model
```

Exercise 4.2: ONNX Runtime Inference (25 min)

Goal: Run inference with ONNX Runtime

```
import onnxruntime as ort
import numpy as np

# Load ONNX model
session = ort.InferenceSession("resnet18.onnx")

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

# Warmup
for _ in range(10):
    _ = session.run(None, {'input': input_data})

# Benchmark
import time
times = []
for _ in range(100):
    start = time.time()
    outputs = session.run(None, {'input': input_data})
    times.append(time.time() - start)

print(f"ONNX Runtime latency: {np.mean(times)*1000:.2f} ms")

# Compare to PyTorch
pytorch_latency = 30  # Your baseline
onnx_latency = np.mean(times)*1000
speedup = pytorch_latency / onnx_latency
print(f"ONNX Runtime speedup: {speedup:.2f}x")
```

Exercise 4.3: Quantize ONNX Model (20 min)

Goal: Apply quantization to ONNX model

```
from onnxruntime.quantization import quantize_dynamic, QuantType

# Quantize ONNX model
quantize_dynamic(
    "resnet18.onnx",
    "resnet18_int8.onnx",
    weight_type=QuantType.QInt8
)

# Load quantized model
session_int8 = ort.InferenceSession("resnet18_int8.onnx")

# Benchmark
times = []
for _ in range(100):
    start = time.time()
    _ = session_int8.run(None, {'input': input_data})
    times.append(time.time() - start)

print(f"Quantized ONNX latency: {np.mean(times)*1000:.2f} ms")

# Measure size
import os
fp32_size = os.path.getsize("resnet18.onnx") / (1024*1024)
int8_size = os.path.getsize("resnet18_int8.onnx") / (1024*1024)
print(f"FP32 ONNX: {fp32_size:.2f} MB")
print(f"INT8 ONNX: {int8_size:.2f} MB")
print(f"Compression: {fp32_size/int8_size:.2f}x")
```

Part 4 Checkpoint

ONNX results:

Format	Size	Latency	vs PyTorch
PyTorch FP32	45MB	30ms	1x
ONNX FP32	?MB	?ms	?x faster
ONNX INT8	?MB	?ms	?x faster

Part 5: Comprehensive Comparison

Putting it all together

Exercise 5.1: Create Comparison Dashboard (25 min)

Goal: Visualize all optimizations

Create `compare_all.py`:

```
import matplotlib.pyplot as plt
import numpy as np

# Your measurements (fill in!)
models = ['Baseline\nFP32', 'Dynamic\nINT8', 'Pruned\n30%', 'ONNX\nFP32', 'ONNX\nINT8']
sizes = [45, ?, ?, ?, ?]          # MB
latencies = [30, ?, ?, ?, ?]      # ms
accuracies = [100, 99, 98, 100, 99]  # % relative to baseline

# Create comparison plots
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

# Size comparison
axes[0].bar(models, sizes, color='skyblue')
axes[0].set_ylabel('Model Size (MB)')
axes[0].set_title('Model Size Comparison')
axes[0].axhline(y=sizes[0], color='r', linestyle='--', label='Baseline')
axes[0].legend()

# Latency comparison
axes[1].bar(models, latencies, color='lightcoral')
axes[1].set_ylabel('Latency (ms)')
axes[1].set_title('Inference Latency')
axes[1].axhline(y=latencies[0], color='r', linestyle='--', label='Baseline')
axes[1].legend()

# Accuracy vs Size scatter
axes[2].scatter(sizes, accuracies, s=200)
for i, model in enumerate(models):
    axes[2].annotate(model, (sizes[i], accuracies[i]))
axes[2].set_xlabel('Model Size (MB)')
axes[2].set_ylabel('Relative Accuracy (%)')
axes[2].set_title('Accuracy vs Size Trade-off')
```

Exercise 5.2: Calculate Efficiency Metrics (15 min)

Goal: Quantify improvements

```
# Compression ratio
compression_ratios = [sizes[0] / s for s in sizes]

# Speedup
speedups = [latencies[0] / l for l in latencies]

# Efficiency score (accuracy / latency / size)
efficiency = [acc / (lat * size) for acc, lat, size in zip(accuracies, latencies, sizes)]

# Print results
print("\n=== Optimization Results ===")
for i, model in enumerate(models):
    print(f"\n{model}:")
    print(f"  Compression: {compression_ratios[i]:.2f}x")
    print(f"  Speedup: {speedups[i]:.2f}x")
    print(f"  Efficiency score: {efficiency[i]:.3f}")
    print(f"  Accuracy: {accuracies[i]:.1f}%")
```

Part 5 Checkpoint

Best optimization for:

- Smallest size: ?
- Fastest inference: ?
- Best accuracy: ?
- Best overall efficiency: ?

Bonus: Knowledge Distillation (Optional)

For those who finish early

Bonus Exercise: Train Student Model (30 min)

Goal: Distill ResNet-18 to MobileNet-v2

```python
import torch
import torch.nn as nn
import torchvision.models as models

# Teacher (ResNet-18)
teacher = models.resnet18(pretrained=True)
teacher.eval()

# Student (MobileNet-v2)
student = models.mobilenet_v2(pretrained=False)

# Distillation loss
def distillation_loss(student_logits, teacher_logits, labels, T=3, alpha=0.5):
    hard_loss = nn.CrossEntropyLoss()(student_logits, labels)

    soft_student = nn.functional.log_softmax(student_logits / T, dim=1)
    soft_teacher = nn.functional.softmax(teacher_logits / T, dim=1)
    soft_loss = nn.functional.kl_div(soft_student, soft_teacher, reduction='batchmean') * T*T

    return alpha * hard_loss + (1 - alpha) * soft_loss

# Train student (pseudo-code)
optimizer = torch.optim.Adam(student.parameters(), lr=1e-3)

for epoch in range(10):
    for images, labels in train_loader:
        # Get teacher predictions
        with torch.no_grad():
            teacher_logits = teacher(images)

        # Train student
        student_logits = student(images)
        loss = distillation_loss(student_logits, teacher_logits, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Lab Wrap-up

What you've accomplished:

✅
Measured baseline model performance

✅
Applied dynamic and static quantization (4x compression)

✅
Pruned model weights (30-50% sparsity)

✅
Exported to ONNX format

✅
Benchmarked ONNX Runtime (2-3x speedup)

✅
Created comprehensive comparison

Typical results:

- Quantization: 4x smaller, 2-3x faster, <1% accuracy drop

Deliverables

Submit:

1. **Code**: All Python scripts
2. **Results table**: Size, latency, accuracy for each method
3. **Comparison plot**: `optimization_comparison.png`
4. **Report** (1 page):
    - Which optimization worked best?
    - What trade-offs did you observe?
    - Which would you deploy for a mobile app?

Troubleshooting

Common issues:

Quantization errors:

- Ensure model is in `eval()` mode
- Use `torch.no_grad()` for inference

ONNX export fails:

- Check PyTorch version compatibility
- Try simpler model first

Slow inference:

- Make sure to warmup before benchmarking
- Use appropriate ONNX Runtime providers

Resources

Documentation:

- PyTorch Quantization: https://pytorch.org/docs/stable/quantization.html
- ONNX: https://onnx.ai/
- ONNX Runtime: https://onnxruntime.ai/

Tools:

- Netron (visualize ONNX): https://netron.app/
- ONNX Model Zoo: https://github.com/onnx/models

Next steps:

- Deploy on actual mobile device (Android/iOS)
- Try TensorFlow Lite
- Experiment with TensorRT