Week 13 Lab: Profiling & Optimization

CS 203: Software Tools and Techniques for AI

Duration: 3 hours

Lab Overview

Objective: Optimize a ResNet-18 training loop from baseline to 2-3x faster.

Structure:

- **Part 1**: Baseline measurement and profiling (30 min)
- **Part 2**: Data loading optimization (30 min)
- **Part 3**: Mixed precision training (AMP) (40 min)
- **Part 4**: Memory optimization (40 min)
- **Part 5**: torch.compile optimization (30 min)
- **Part 6**: Comprehensive comparison (20 min)

Prerequisites:

- GPU runtime (Google Colab or Kaggle recommended)

Setup and Installation

Create a new notebook:

```
# Install required packages
!pip install torch torchvision tensorboard torch-tb-profiler line_profiler memory_profiler matplotlib pandas seaborn

# Imports
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms, models
from torch.utils.data import DataLoader
from torch.profiler import profile, record_function, ProfilerActivity
from torch.cuda.amp import autocast, GradScaler
import torch.utils.checkpoint as checkpoint
import time
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from pathlib import Path

# Check GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")
if torch.cuda.is_available():
    print(f"GPU: {torch.cuda.get_device_name(0)}")
```

Part 1: Baseline Measurement (30 min)

Goal: Establish performance baseline and profile the training loop.

Exercise 1.1: Create baseline training script (10 min)

```
# Define hyperparameters
BATCH_SIZE = 32
NUM_EPOCHS = 1
NUM_WORKERS = 0  # Start with 0 for baseline
IMG_SIZE = 224

# Data preprocessing
transform = transforms.Compose([
    transforms.RandomResizedCrop(IMG_SIZE),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])
```

Exercise 1.1 (continued)

```
# Load CIFAR-10 (small for quick iteration)
# Note: We'll use transforms designed for ImageNet for realistic overhead
train_dataset = datasets.CIFAR10(
    root='./data',
    train=True,
    download=True,
    transform=transform
)

train_loader = DataLoader(
    train_dataset,
    batch_size=BATCH_SIZE,
    shuffle=True,
    num_workers=NUM_WORKERS,
    pin_memory=False  # Baseline without optimization
)

# Load pretrained ResNet-18
model = models.resnet18(pretrained=True)
model.fc = nn.Linear(model.fc.in_features, 10)  # CIFAR-10 has 10 classes
model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-4)
```

Exercise 1.2: Baseline training loop with timing (10 min)

```
def train_epoch(model, dataloader, criterion, optimizer, device):
    """Train for one epoch and return metrics."""
    model.train()
    total_loss = 0
    correct = 0
    total = 0
    batch_times = []

    for batch_idx, (images, labels) in enumerate(dataloader):
        batch_start = time.perf_counter()

        images, labels = images.to(device), labels.to(device)

        # Forward pass
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward pass
        loss.backward()
        optimizer.step()

        # Metrics
        total_loss += loss.item()
        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()

        batch_time = time.perf_counter() - batch_start
        batch_times.append(batch_time)

        if batch_idx % 50 == 0:
            print(f"Batch {batch_idx}/{len(dataloader)}, "
                  f"Loss: {loss.item():.4f}, "
                  f"Acc: {100.*correct/total:.2f}%, "
                  f"Batch time: {batch_time*1000:.2f}ms")

    avg_loss = total_loss / len(dataloader)
    accuracy = 100. * correct / total
    throughput = total / sum(batch_times)  # samples/sec

    return {
        'loss': avg_loss,
```

Exercise 1.2 (continued)

```
# Run baseline training
print("="*50)
print("BASELINE TRAINING")
print("="*50)

# Warmup
print("Warming up...")
for i, batch in enumerate(train_loader):
    if i >= 5:
        break
    images, labels = batch
    images = images.to(device)
    _ = model(images)

# Measure
torch.cuda.reset_peak_memory_stats() if torch.cuda.is_available() else None
start_time = time.time()

baseline_metrics = train_epoch(model, train_loader, criterion, optimizer, device)

end_time = time.time()
baseline_metrics['epoch_time'] = end_time - start_time
baseline_metrics['peak_memory_mb'] = torch.cuda.max_memory_allocated() / 1e6 if torch.cuda.is_available() else 0

print(f"\nBaseline Results:")
print(f"  Epoch time: {baseline_metrics['epoch_time']:.2f}s")
print(f"  Throughput: {baseline_metrics['throughput']:.2f} samples/sec")
print(f"  Peak memory: {baseline_metrics['peak_memory_mb']:.2f} MB")
print(f"  Final accuracy: {baseline_metrics['accuracy']:.2f}%")
```

Exercise 1.3: Profile with PyTorch Profiler (10 min)

```
# Profile the training loop
print("\nProfiling baseline...")

with profile(
    activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA] if torch.cuda.is_available() else [ProfilerActivity.CPU],
    record_shapes=True,
    profile_memory=True,
    with_stack=True
) as prof:
    with record_function("baseline_training"):
        for batch_idx, (images, labels) in enumerate(train_loader):
            if batch_idx >= 10:  # Profile first 10 batches
                break

            images, labels = images.to(device), labels.to(device)

            with record_function("forward"):
                optimizer.zero_grad()
                outputs = model(images)
                loss = criterion(outputs, labels)

            with record_function("backward"):
                loss.backward()

            with record_function("optimizer_step"):
                optimizer.step()

            prof.step()

# Print profiling results
print("\nTop 10 operations by CUDA time:")
print(prof.key_averages().table(
    sort_by="cuda_time_total" if torch.cuda.is_available() else "cpu_time_total",
    row_limit=10
))
```

Exercise 1.3 (continued)

```
# Save trace for TensorBoard visualization
prof.export_chrome_trace("baseline_trace.json")
print("\nTrace saved to baseline_trace.json")
print("To visualize: chrome://tracing in Chrome browser")

# Questions to answer:
print("\n" + "="*50)
print("ANALYSIS QUESTIONS:")
print("="*50)
print("1. What % of time is spent on data loading vs computation?")
print("2. Is the GPU utilization high (>80%) or low (<50%)?")
print("3. How much time is spent on memory transfers (HtoD)?")
print("4. What are the top 3 slowest operations?")
print("="*50)
```

Expected observations:

- Low GPU utilization if `num_workers=0`

Part 2: Data Loading Optimization (30 min)

Goal: Optimize CPU-bound data loading bottleneck.

Exercise 2.1: Experiment with num_workers (15 min)

```
# Test different num_workers configurations
worker_configs = [0, 2, 4, 8]
worker_results = []

for num_workers in worker_configs:
    print(f"\n{'='*50}")
    print(f"Testing num_workers={num_workers}")
    print(f"{'='*50}")

    # Create dataloader with new config
    train_loader_test = DataLoader(
        train_dataset,
        batch_size=BATCH_SIZE,
        shuffle=True,
        num_workers=num_workers,
        pin_memory=False  # Will add later
    )

    # Warmup
    for i, batch in enumerate(train_loader_test):
        if i >= 3:
            break
        images, _ = batch
        images = images.to(device)

    # Measure
    start_time = time.time()
    metrics = train_epoch(model, train_loader_test, criterion, optimizer, device)
    epoch_time = time.time() - start_time
```

Exercise 2.1 (continued)

```
# Visualize results
df_workers = pd.DataFrame(worker_results)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

ax1.plot(df_workers['num_workers'], df_workers['throughput'], marker='o')
ax1.set_xlabel('num_workers')
ax1.set_ylabel('Throughput (samples/sec)')
ax1.set_title('Throughput vs num_workers')
ax1.grid(True)

ax2.plot(df_workers['num_workers'], df_workers['epoch_time'], marker='o', color='orange')
ax2.set_xlabel('num_workers')
ax2.set_ylabel('Epoch time (s)')
ax2.set_title('Epoch Time vs num_workers')
ax2.grid(True)

plt.tight_layout()
plt.savefig('num_workers_comparison.png', dpi=150)
plt.show()

print(f"\nBest configuration: num_workers={df_workers.loc[df_workers['throughput'].idxmax(), 'num_workers']:.0f}")
print(f"Speedup: {df_workers['throughput'].max() / df_workers['throughput'].min():.2f}x")
```

Exercise 2.2: Add pin_memory and persistent_workers (15 min)

```
# Find optimal num_workers from previous exercise
optimal_workers = int(df_workers.loc[df_workers['throughput'].idxmax(), 'num_workers'])

# Compare configurations
configs = [
    {'name': 'Baseline', 'num_workers': 0, 'pin_memory': False, 'persistent_workers': False},
    {'name': 'Optimal workers', 'num_workers': optimal_workers, 'pin_memory': False, 'persistent_workers': False},
    {'name': '+ pin_memory', 'num_workers': optimal_workers, 'pin_memory': True, 'persistent_workers': False},
    {'name': '+ persistent', 'num_workers': optimal_workers, 'pin_memory': True, 'persistent_workers': True},
]

dataloader_results = []

for config in configs:
    print(f"\n{'='*50}")
    print(f"Testing: {config['name']}")
    print(f"{'='*50}")

    train_loader_test = DataLoader(
        train_dataset,
        batch_size=BATCH_SIZE,
        shuffle=True,
        num_workers=config['num_workers'],
        pin_memory=config['pin_memory'],
        persistent_workers=config['persistent_workers'] if config['num_workers'] > 0 else False
    )

    # Warmup
    for i, batch in enumerate(train_loader_test):
        if i >= 3:
            break
        images, _ = batch
        images = images.to(device)

    # Measure
    start_time = time.time()
    metrics = train_epoch(model, train_loader_test, criterion, optimizer, device)
    epoch_time = time.time() - start_time

    dataloader_results.append({
        'config': config['name'],
        'throughput': metrics['throughput'],
        'epoch_time': epoch_time,
        'speedup': metrics['throughput'] / baseline_metrics['throughput']
```

Exercise 2.2 (continued)

```
# Visualize comparison
df_loaders = pd.DataFrame(dataloader_results)

plt.figure(figsize=(10, 6))
bars = plt.bar(df_loaders['config'], df_loaders['speedup'], color=['gray', 'blue', 'green', 'orange'])
plt.axhline(y=1, color='red', linestyle='--', label='Baseline')
plt.ylabel('Speedup vs Baseline', fontsize=12)
plt.title('DataLoader Optimization Impact', fontsize=14)
plt.xticks(rotation=15, ha='right')
plt.legend()
plt.grid(axis='y', alpha=0.3)

for bar in bars:
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2., height,
             f'{height:.2f}x',
             ha='center', va='bottom', fontsize=10)

plt.tight_layout()
plt.savefig('dataloader_optimization.png', dpi=150)
plt.show()

print(f"\nFinal speedup from data loading optimizations: {df_loaders['speedup'].max():.2f}x")
```

Expected result: 1.3-1.8x speedup from optimized data loading.

Part 3: Mixed Precision Training (40 min)

Goal: Enable automatic mixed precision (AMP) for faster training.

Exercise 3.1: Implement AMP (20 min)

```
# Create optimized dataloader for remaining experiments
train_loader_optimized = DataLoader(
    train_dataset,
    batch_size=BATCH_SIZE,
    shuffle=True,
    num_workers=optimal_workers,
    pin_memory=True,
    persistent_workers=True
)

# Reload model to reset weights
model = models.resnet18(pretrained=True)
model.fc = nn.Linear(model.fc.in_features, 10)
model = model.to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-4)

# Training function with AMP support
def train_epoch_amp(model, dataloader, criterion, optimizer, device, use_amp=False):
    """Train for one epoch with optional AMP."""
    model.train()
    scaler = GradScaler() if use_amp else None

    total_loss = 0
    correct = 0
    total = 0
    batch_times = []

    for batch_idx, (images, labels) in enumerate(dataloader):
        batch_start = time.perf_counter()

        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()

        # Forward pass with optional autocast
        if use_amp:
            with autocast():
                outputs = model(images)
                loss = criterion(outputs, labels)
        else:
            outputs = model(images)
            loss = criterion(outputs, labels)

        # Backward pass with optional gradient scaling
        if use_amp:
            scaler.scale(loss).backward()
            scaler.step(optimizer)
            scaler.update()
        else:
            loss.backward()
            optimizer.step()

        # Metrics
        total_loss += loss.item()
```

Exercise 3.1 (continued)

```
# Compare FP32 vs AMP
precision_results = []

for use_amp in [False, True]:
    precision_name = "AMP (FP16)" if use_amp else "FP32"
    print(f"\n{'='*50}")
    print(f"Training with {precision_name}")
    print(f"{'='*50}")

    # Reset model
    model = models.resnet18(pretrained=True)
    model.fc = nn.Linear(model.fc.in_features, 10)
    model = model.to(device)
    optimizer = optim.Adam(model.parameters(), lr=1e-4)

    # Warmup
    for i, batch in enumerate(train_loader_optimized):
        if i >= 3:
            break
        images, labels = batch
        images, labels = images.to(device), labels.to(device)
        with autocast() if use_amp else torch.no_grad():
            _ = model(images)

    # Measure
    torch.cuda.reset_peak_memory_stats() if torch.cuda.is_available() else None
    start_time = time.time()

    metrics = train_epoch_amp(model, train_loader_optimized, criterion, optimizer, device, use_amp)

    epoch_time = time.time() - start_time
    peak_memory = torch.cuda.max_memory_allocated() / 1e6 if torch.cuda.is_available() else 0

    precision_results.append({
        'precision': precision_name,
        'throughput': metrics['throughput'],
        'epoch_time': epoch_time,
        'peak_memory_mb': peak_memory,
        'accuracy': metrics['accuracy'],
        'speedup': metrics['throughput'] / baseline_metrics['throughput']
    })

    print(f"Throughput: {metrics['throughput']:.2f} samples/sec")
    print(f"Peak memory: {peak_memory:.2f} MB")
    print(f"Accuracy: {metrics['accuracy']:.2f}%")
    print(f"Speedup vs baseline: {precision_results[-1]['speedup']:.2f}x")
```

Exercise 3.2: Analyze AMP impact (20 min)

```
# Visualize AMP benefits
df_precision = pd.DataFrame(precision_results)

fig, axes = plt.subplots(1, 3, figsize=(15, 4))

# Throughput comparison
axes[0].bar(df_precision['precision'], df_precision['throughput'], color=['blue', 'green'])
axes[0].set_ylabel('Throughput (samples/sec)')
axes[0].set_title('Training Throughput')
axes[0].grid(axis='y', alpha=0.3)
for i, v in enumerate(df_precision['throughput']):
    axes[0].text(i, v, f'{v:.1f}', ha='center', va='bottom')

# Memory comparison
axes[1].bar(df_precision['precision'], df_precision['peak_memory_mb'], color=['blue', 'green'])
axes[1].set_ylabel('Peak Memory (MB)')
axes[1].set_title('Memory Usage')
axes[1].grid(axis='y', alpha=0.3)
for i, v in enumerate(df_precision['peak_memory_mb']):
    axes[1].text(i, v, f'{v:.0f}', ha='center', va='bottom')

# Speedup comparison
axes[2].bar(df_precision['precision'], df_precision['speedup'], color=['blue', 'green'])
axes[2].axhline(y=1, color='red', linestyle='--', alpha=0.5)
axes[2].set_ylabel('Speedup vs Baseline')
axes[2].set_title('Overall Speedup')
axes[2].grid(axis='y', alpha=0.3)
for i, v in enumerate(df_precision['speedup']):
    axes[2].text(i, v, f'{v:.2f}x', ha='center', va='bottom')

plt.tight_layout()
plt.savefig('amp_comparison.png', dpi=150)
plt.show()

print(f"\nAMP Speedup: {df_precision.iloc[1]['throughput'] / df_precision.iloc[0]['throughput']:.2f}x")
print(f"Memory savings: {(1 - df_precision.iloc[1]['peak_memory_mb'] / df_precision.iloc[0]['peak_memory_mb']) * 100:.1f}%")
```

Part 4: Memory Optimization (40 min)

Goal: Use gradient checkpointing and accumulation to fit larger batches.

Exercise 4.1: Gradient accumulation (20 min)

```
def train_epoch_with_accumulation(model, dataloader, criterion, optimizer, device,
                                  accumulation_steps=1, use_amp=False):
    """Train with gradient accumulation."""
    model.train()
    scaler = GradScaler() if use_amp else None

    total_loss = 0
    correct = 0
    total = 0
    batch_times = []

    optimizer.zero_grad()

    for batch_idx, (images, labels) in enumerate(dataloader):
        batch_start = time.perf_counter()

        images, labels = images.to(device), labels.to(device)

        # Forward pass
        if use_amp:
            with autocast():
                outputs = model(images)
                loss = criterion(outputs, labels) / accumulation_steps
        else:
            outputs = model(images)
            loss = criterion(outputs, labels) / accumulation_steps

        # Backward pass
        if use_amp:
            scaler.scale(loss).backward()
        else:
            loss.backward()

        # Update weights every accumulation_steps
        if (batch_idx + 1) % accumulation_steps == 0:
            if use_amp:
                scaler.step(optimizer)
                scaler.update()
            else:
                optimizer.step()
            optimizer.zero_grad()

        # Metrics
```

Exercise 4.1 (continued)

```
# Test gradient accumulation with different steps
accumulation_configs = [1, 2, 4, 8]
accumulation_results = []

for accum_steps in accumulation_configs:
    effective_batch = BATCH_SIZE * accum_steps
    print(f"\n{'='*50}")
    print(f"Gradient Accumulation: {accum_steps} steps (effective batch: {effective_batch})")
    print(f"{'='*50}")

    # Reset model
    model = models.resnet18(pretrained=True)
    model.fc = nn.Linear(model.fc.in_features, 10)
    model = model.to(device)
    optimizer = optim.Adam(model.parameters(), lr=1e-4)

    # Measure
    torch.cuda.reset_peak_memory_stats() if torch.cuda.is_available() else None
    start_time = time.time()

    metrics = train_epoch_with_accumulation(
        model, train_loader_optimized, criterion, optimizer, device,
        accumulation_steps=accum_steps, use_amp=True
    )

    epoch_time = time.time() - start_time
    peak_memory = torch.cuda.max_memory_allocated() / 1e6 if torch.cuda.is_available() else 0

    accumulation_results.append({
        'accumulation_steps': accum_steps,
        'effective_batch': effective_batch,
        'throughput': metrics['throughput'],
        'peak_memory_mb': peak_memory,
        'epoch_time': epoch_time
    })

    print(f"Throughput: {metrics['throughput']:.2f} samples/sec")
    print(f"Peak memory: {peak_memory:.2f} MB")
    print(f"Epoch time: {epoch_time:.2f}s")
```

Exercise 4.2: Gradient checkpointing (20 min)

```
# Create a model with gradient checkpointing
class ResNet18Checkpointed(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        resnet = models.resnet18(pretrained=True)

        # Split into checkpointed segments
        self.layer0 = nn.Sequential(resnet.conv1, resnet.bn1, resnet.relu, resnet.maxpool)
        self.layer1 = resnet.layer1
        self.layer2 = resnet.layer2
        self.layer3 = resnet.layer3
        self.layer4 = resnet.layer4
        self.avgpool = resnet.avgpool
        self.fc = nn.Linear(resnet.fc.in_features, num_classes)

    def forward(self, x):
        # Use gradient checkpointing for each layer
        x = checkpoint.checkpoint(self.layer0, x, use_reentrant=False)
        x = checkpoint.checkpoint(self.layer1, x, use_reentrant=False)
        x = checkpoint.checkpoint(self.layer2, x, use_reentrant=False)
        x = checkpoint.checkpoint(self.layer3, x, use_reentrant=False)
        x = checkpoint.checkpoint(self.layer4, x, use_reentrant=False)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)
        return x

# Compare standard vs checkpointed model
memory_results = []

for use_checkpointing in [False, True]:
    model_name = "With Checkpointing" if use_checkpointing else "Standard"
    print(f"\n{'='*50}")
    print(f"Model: {model_name}")
    print(f"{'='*50}")

    if use_checkpointing:
        model = ResNet18Checkpointed(num_classes=10).to(device)
    else:
        model = models.resnet18(pretrained=True)
        model.fc = nn.Linear(model.fc.in_features, 10)
        model = model.to(device)

    optimizer = optim.Adam(model.parameters(), lr=1e-4)

    # Measure
    torch.cuda.reset_peak_memory_stats() if torch.cuda.is_available() else None
    start_time = time.time()

    metrics = train_epoch_amp(model, train_loader_optimized, criterion, optimizer, device, use_amp=True)

    epoch_time = time.time() - start_time
    peak_memory = torch.cuda.max_memory_allocated() / 1e6 if torch.cuda.is_available() else 0

    memory_results.append({
        'model': model_name,
        'peak_memory_mb': peak_memory,
        'throughput': metrics['throughput'],
        'epoch_time': epoch_time
    })

    print(f"Peak memory: {peak_memory:.2f} MB")
    print(f"Throughput: {metrics['throughput']:.2f} samples/sec")
    print(f"Epoch time: {epoch_time:.2f}s")
```

Exercise 4.2 (continued)

```
# Visualize memory optimization results
df_memory = pd.DataFrame(memory_results)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Memory comparison
ax1.bar(df_memory['model'], df_memory['peak_memory_mb'], color=['blue', 'green'])
ax1.set_ylabel('Peak Memory (MB)')
ax1.set_title('Memory Usage Comparison')
ax1.grid(axis='y', alpha=0.3)
for i, v in enumerate(df_memory['peak_memory_mb']):
    ax1.text(i, v, f'{v:.0f} MB', ha='center', va='bottom')

# Speed comparison
ax2.bar(df_memory['model'], df_memory['throughput'], color=['blue', 'green'])
ax2.set_ylabel('Throughput (samples/sec)')
ax2.set_title('Training Speed Comparison')
ax2.grid(axis='y', alpha=0.3)
for i, v in enumerate(df_memory['throughput']):
    ax2.text(i, v, f'{v:.1f}', ha='center', va='bottom')

plt.tight_layout()
plt.savefig('checkpointing_comparison.png', dpi=150)
plt.show()

print(f"\nMemory savings: {(1 - df_memory.iloc[1]['peak_memory_mb'] / df_memory.iloc[0]['peak_memory_mb']) * 100:.1f}%")
print(f"Speed impact: {(df_memory.iloc[1]['throughput'] / df_memory.iloc[0]['throughput'] - 1) * 100:.1f}%")
```

Expected results: 30-50% memory reduction, 10-20% slower training.

Part 5: torch.compile Optimization (30 min)

Goal: Use PyTorch 2.0's compilation for additional speedup.

Exercise 5.1: Compare compiled vs eager mode (30 min)

```
# Check PyTorch version
print(f"PyTorch version: {torch.__version__}")
if not hasattr(torch, 'compile'):
    print("Warning: torch.compile requires PyTorch 2.0+")
    print("Skipping this exercise or upgrade PyTorch")
else:
    compile_results = []

    for use_compile in [False, True]:
        mode_name = "Compiled" if use_compile else "Eager"
        print(f"\n{'='*50}")
        print(f"Mode: {mode_name}")
        print(f"{'='*50}")

        # Create model
        model = models.resnet18(pretrained=True)
        model.fc = nn.Linear(model.fc.in_features, 10)
        model = model.to(device)

        # Compile if requested
        if use_compile:
            model = torch.compile(model, mode='default')

        optimizer = optim.Adam(model.parameters(), lr=1e-4)

        # Warmup (important for compiled models!)
        print("Warming up...")
        for i, batch in enumerate(train_loader_optimized):
            if i >= 5:
                break
            images, labels = batch
            images, labels = images.to(device), labels.to(device)
            with autocast():
                _ = model(images)

        # Measure
        start_time = time.time()
        metrics = train_epoch_amp(model, train_loader_optimized, criterion, optimizer, device, use_amp=True)
        epoch_time = time.time() - start_time
```

Exercise 5.1 (continued)

```
    # Visualize torch.compile impact
    df_compile = pd.DataFrame(compile_results)

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

    # Throughput
    ax1.bar(df_compile['mode'], df_compile['throughput'], color=['blue', 'purple'])
    ax1.set_ylabel('Throughput (samples/sec)')
    ax1.set_title('Training Throughput')
    ax1.grid(axis='y', alpha=0.3)
    for i, v in enumerate(df_compile['throughput']):
        ax1.text(i, v, f'{v:.1f}', ha='center', va='bottom')

    # Speedup
    ax2.bar(df_compile['mode'], df_compile['speedup_vs_baseline'], color=['blue', 'purple'])
    ax2.axhline(y=1, color='red', linestyle='--', alpha=0.5, label='Baseline')
    ax2.set_ylabel('Speedup vs Baseline')
    ax2.set_title('Overall Speedup')
    ax2.legend()
    ax2.grid(axis='y', alpha=0.3)
    for i, v in enumerate(df_compile['speedup_vs_baseline']):
        ax2.text(i, v, f'{v:.2f}x', ha='center', va='bottom')

    plt.tight_layout()
    plt.savefig('compile_comparison.png', dpi=150)
    plt.show()

    if len(df_compile) > 1:
        print(f"\ntorch.compile speedup: {df_compile.iloc[1]['throughput'] / df_compile.iloc[0]['throughput']:.2f}x")
```

Part 6: Comprehensive Comparison (20 min)

Goal: Compare all optimizations and create final report.

Exercise 6.1: Final benchmark of all configurations (20 min)

```
# Define all configurations to test
final_configs = [
    {
        'name': 'Baseline',
        'num_workers': 0,
        'pin_memory': False,
        'use_amp': False,
        'use_compile': False
    },
    {
        'name': 'Optimized DataLoader',
        'num_workers': optimal_workers,
        'pin_memory': True,
        'use_amp': False,
        'use_compile': False
    },
    {
        'name': '+ AMP',
        'num_workers': optimal_workers,
        'pin_memory': True,
        'use_amp': True,
        'use_compile': False
    },
    {
        'name': '+ torch.compile',
        'num_workers': optimal_workers,
        'pin_memory': True,
        'use_amp': True,
        'use_compile': hasattr(torch, 'compile')
    }
]

final_results = []

for config in final_configs:
    print(f"\n{'='*60}")
    print(f"Configuration: {config['name']}")
    print(f"{'='*60}")

    # Create dataloader
    dataloader = DataLoader(
        train_dataset,
        batch_size=BATCH_SIZE,
        shuffle=True,
        num_workers=config['num_workers'],
        pin_memory=config['pin_memory'],
        persistent_workers=config['num_workers'] > 0
    )

    # Create model
    model = models.resnet18(pretrained=True)
    model.fc = nn.Linear(model.fc.in_features, 10)
    model = model.to(device)

    if config['use_compile']:
        model = torch.compile(model)

    optimizer = optim.Adam(model.parameters(), lr=1e-4)

    # Warmup
    for i, batch in enumerate(dataloader):
        if i >= 5:
            break
        images, labels = batch
        images, labels = images.to(device), labels.to(device)
        if config['use_amp']:
            with autocast():
                _ = model(images)
        else:
            _ = model(images)

    # Measure
```

Exercise 6.1 (continued)

```
# Create comprehensive comparison table
df_final = pd.DataFrame(final_results)

print("\n" + "="*80)
print("FINAL OPTIMIZATION SUMMARY")
print("="*80)
print(df_final.to_string(index=False))
print("="*80)

# Save to CSV
df_final.to_csv('optimization_results.csv', index=False)
print("\nResults saved to optimization_results.csv")
```

Exercise 6.2: Create visualization dashboard

```
# Extract numeric values for plotting
df_plot = pd.DataFrame({
    'config': [r['Configuration'] for r in final_results],
    'throughput': [float(r['Throughput (samples/sec)']) for r in final_results],
    'memory': [float(r['Peak Memory (MB)']) for r in final_results],
    'speedup': [float(r['Speedup'].replace('x', '')) for r in final_results]
})

# Create comprehensive visualization
fig = plt.figure(figsize=(16, 10))
gs = fig.add_gridspec(3, 2, hspace=0.3, wspace=0.3)

# 1. Throughput comparison
ax1 = fig.add_subplot(gs[0, :])
bars = ax1.bar(df_plot['config'], df_plot['throughput'], color=['gray', 'blue', 'green', 'purple'])
ax1.set_ylabel('Throughput (samples/sec)', fontsize=12)
ax1.set_title('Training Throughput Comparison', fontsize=14, fontweight='bold')
ax1.grid(axis='y', alpha=0.3)
ax1.tick_params(axis='x', rotation=15)
for bar in bars:
    height = bar.get_height()
    ax1.text(bar.get_x() + bar.get_width()/2., height,
             f'{height:.1f}',
             ha='center', va='bottom', fontsize=10)

# 2. Speedup progression
ax2 = fig.add_subplot(gs[1, 0])
ax2.plot(range(len(df_plot)), df_plot['speedup'], marker='o', linewidth=2, markersize=8, color='green')
ax2.axhline(y=1, color='red', linestyle='--', alpha=0.5, label='Baseline')
ax2.set_xticks(range(len(df_plot)))
ax2.set_xticklabels(df_plot['config'], rotation=15, ha='right')
ax2.set_ylabel('Cumulative Speedup', fontsize=12)
ax2.set_title('Optimization Progress', fontsize=12, fontweight='bold')
ax2.legend()
ax2.grid(True, alpha=0.3)
for i, v in enumerate(df_plot['speedup']):
    ax2.text(i, v + 0.05, f'{v:.2f}x', ha='center', fontsize=9)

# 3. Memory usage
ax3 = fig.add_subplot(gs[1, 1])
bars = ax3.bar(df_plot['config'], df_plot['memory'], color=['gray', 'blue', 'green', 'purple'])
ax3.set_ylabel('Peak Memory (MB)', fontsize=12)
ax3.set_title('Memory Usage', fontsize=12, fontweight='bold')
ax3.grid(axis='y', alpha=0.3)
ax3.tick_params(axis='x', rotation=15)
for bar in bars:
    height = bar.get_height()
    ax3.text(bar.get_x() + bar.get_width()/2., height,
             f'{height:.0f}',
             ha='center', va='bottom', fontsize=9)

# 4. Incremental speedup
ax4 = fig.add_subplot(gs[2, :])
incremental_speedup = [1.0]
for i in range(1, len(df_plot)):
    incremental_speedup.append(df_plot['speedup'].iloc[i] / df_plot['speedup'].iloc[i-1])

bars = ax4.bar(df_plot['config'], incremental_speedup, color=['gray', 'blue', 'green', 'purple'])
ax4.axhline(y=1, color='red', linestyle='--', alpha=0.5, label='No change')
ax4.set_ylabel('Incremental Speedup', fontsize=12)
ax4.set_title('Speedup from Each Optimization Step', fontsize=12, fontweight='bold')
ax4.grid(axis='y', alpha=0.3)
ax4.legend()
ax4.tick_params(axis='x', rotation=15)
for bar in bars:
    height = bar.get_height()
    ax4.text(bar.get_x() + bar.get_width()/2., height,
             f'{height:.2f}x',
             ha='center', va='bottom', fontsize=9)

plt.suptitle('PyTorch Training Optimization Dashboard', fontsize=16, fontweight='bold', y=0.995)
plt.savefig('optimization_dashboard.png', dpi=150, bbox_inches='tight')
plt.show()

print(f"\nFinal speedup: {df_plot['speedup'].iloc[-1]:.2f}x")
print(f"Total time saved per epoch: {baseline_metrics['epoch_time'] - (baseline_metrics['epoch_time'] / df_plot['speedup'].iloc[-1]):.2f}s")
```

Deliverables

Required files to submit:

1. `optimization_report.md`:
    - Summary of findings
    - Table of all configurations tested
    - Analysis of bottlenecks identified
    - Recommendations for production deployment
2. `optimized_training.py`:
    - Clean, production-ready training script with all optimizations
3. **Visualizations**:
    - `optimization_dashboard.png`

Report Template

```
# Profiling & Optimization Report

## Executive Summary
- Baseline throughput: X samples/sec
- Optimized throughput: Y samples/sec
- **Final speedup: Zx**
- Total optimization time: N hours

## Bottlenecks Identified
1. **Data Loading**: [Description and evidence]
2. **GPU Compute**: [Description and evidence]
3. **Memory**: [Description and evidence]

## Optimizations Applied

### 1. DataLoader Optimization
- Configuration: num_workers=X, pin_memory=True
- Speedup: Xx
- Impact: [High/Medium/Low]

### 2. Mixed Precision (AMP)
- Speedup: Xx
- Memory savings: X%
- Impact: [High/Medium/Low]

### 3. torch.compile
- Speedup: Xx
- Impact: [High/Medium/Low]

## Production Recommendations
1. [Recommendation 1]
2. [Recommendation 2]
3. [Recommendation 3]

## Lessons Learned
- [Lesson 1]
- [Lesson 2]
```

Bonus Challenges (Optional)

Challenge 1: Profile on CPU (30 min)

- Run all experiments on CPU
- Compare optimization strategies (some may not help on CPU!)
- Identify CPU-specific bottlenecks

Challenge 2: Larger model (45 min)

- Try ResNet-50 instead of ResNet-18
- Will you run out of memory?
- Which optimizations are most critical?

Challenge 3: Custom profiling (60 min)

- Use `cProfile` to profile Python code

Key Takeaways

1. **Profile first, optimize second** - Don't guess bottlenecks!
2. **Quick wins matter** - AMP and num_workers give major speedups with minimal effort
3. **Memory vs speed trade-offs** - Checkpointing saves memory but costs speed
4. **Measure everything** - Use proper benchmarking (warmup, multiple runs, statistics)
5. **Cumulative gains** - Small optimizations compound to major improvements
6. **Production readiness** - Optimized code should be clean and maintainable

Expected final speedup: 2-3x with all optimizations combined!

Additional Resources

PyTorch Documentation:

- Profiler: https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html
- AMP: https://pytorch.org/docs/stable/amp.html
- torch.compile: https://pytorch.org/tutorials/intermediate/torch_compile_tutorial.html

Profiling Tools:

- TensorBoard: https://www.tensorflow.org/tensorboard
- Chrome Tracing: chrome://tracing
- NVIDIA Nsight Systems: https://developer.nvidia.com/nsight-systems

Further Reading:

- "Mixed Precision Training" (Micikevicius et al., 2018)