Profiling & Optimization

CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra, IIT Gandhinagar

Why Optimize?

Training is expensive.

- GPT-3 cost ~$4.6M to train.
- Even small models can burn through credits.

Inference is slow.

- Users hate latency (>100ms feels sluggish).
- Real-time applications need <30ms.

Goal: Make code faster and lighter without losing accuracy.

The Optimization Loop

graph LR A[Baseline Model] --> B[Profile]; B --> C{Bottleneck?}; C -- CPU --> D[Dataloader Opt]; C -- GPU Comp --> E[Mixed Precision]; C -- GPU Mem --> F[Gradient Checkpointing]; D --> G[Measure]; E --> G; F --> G; G --> B;

Rule #1: Don't optimize without profiling.

"Premature optimization is the root of all evil." - Donald Knuth

Profiling Tools

1. `nvidia-smi` : The basic dashboard.
    - Volatile GPU-Util: Is the GPU busy? (Aim for >90%)
    - Memory-Usage: Are we maxing out VRAM?
2. **PyTorch Profiler**: The deep dive.
    - Traces execution steps.
    - Identifies CPU vs GPU time.
    - Visualizes the timeline.
3. **Nsight Systems**: System-wide view (OS + CUDA).

Common Bottlenecks

1. Data Loading (CPU Bound)

- GPU is at 0% utilization waiting for data.
- **Fix**: Increase `num_workers`, use `pin_memory=True`, pre-fetch data.

2. Kernel Launch Overhead

- Too many small operations.
- **Fix**: `torch.compile` (Graph fusion).

3. Memory Bandwidth

- Moving data between CPU and GPU too often.
- **Fix**: Keep tensors on GPU.

Automatic Mixed Precision (AMP)

Float32 (Single): 4 bytes. Standard.
Float16 (Half): 2 bytes. Faster, less memory.

Problem: Float16 has low dynamic range (gradients vanish/explode).

Solution: AMP

- Keep master weights in FP32.
- Compute forward/backward in FP16.
- Scale gradients to prevent underflow.

Impact: 2-3x speedup on Tensor Cores (NVIDIA V100/A100/H100).

AMP in PyTorch

```
import torch
from torch.cuda.amp import autocast, GradScaler

model = MyModel().cuda()
optimizer = torch.optim.Adam(model.parameters())
scaler = GradScaler()

for input, target in data_loader:
    optimizer.zero_grad()

    # Run forward pass in FP16
    with autocast():
        output = model(input)
        loss = loss_fn(output, target)

    # Scale gradients and step
    scaler.scale(loss).backward()
    scaler.step(optimizer)
    scaler.update()
```

Model Pruning

Idea: Remove neurons/weights that contribute little.

Unstructured Pruning:

- Zero out individual weights.
- Result: Sparse matrix.
- **Con**: Standard GPUs don't speed up sparse matrices much.

Structured Pruning:

- Remove entire channels/filters.
- Result: Smaller dense matrix.
- **Pro**: Real speedup everywhere.

Knowledge Distillation

Teacher-Student Training:

Train a small "Student" model to mimic a large "Teacher".

$$L = \alpha L_{task} + (1 - \alpha) L_{distill}(y_{teacher}, y_{student})$$

- **Teacher**: ResNet-101 (High acc, slow)
- **Student**: ResNet-18 (Low acc, fast)
- **Result**: Student performs better than if trained alone.

Lab Preview

Today's Mission:

1. **Profile**: Use PyTorch Profiler to find a bottleneck in a ResNet training loop.
2. **Fix Data Loading**: Optimize `num_workers`.
3. **Apply AMP**: Speed up training with `torch.cuda.amp`.
4. **Prune**: Use `torch.nn.utils.prune` to remove 50% of weights.

Metric: Samples per second (Throughput).