CS336 Language Modeling from Scratch

Lecture 2: PyTorch Primitives and Resource Accounting

Stanford University, Spring 2025

Abstract

This lecture covers fundamental PyTorch primitives and resource accounting for building large language models from scratch. Key topics include memory accounting with different floating-point representations, compute optimization strategies, tensor operations and storage mechanisms, GPU memory management, and practical resource estimation techniques. The lecture provides essential foundations for efficient deep learning implementation and understanding computational resource requirements in modern language model training.

Enhanced Summary by:
GitHub: HtmMhmd — LinkedIn: Hatem Mohamed

Contents

1 Introduction and Course Context

1.1 Lecture Overview

This lecture focuses on the fundamental building blocks for training large language models, emphasizing PyTorch primitives and resource accounting. The goal is to understand how to efficiently use computational resources (memory and compute) when building models from scratch.

Key Point: Efficiency Imperative

When training large language models, understanding resource utilization directly translates to cost optimization. Inefficient resource usage can result in significant financial overhead.

Learning Objectives:

1. Master PyTorch primitives and tensor operations
2. Understand memory and compute accounting principles
3. Apply resource estimation techniques for model planning
4. Optimize GPU memory usage and data movement

1.2 Napkin Math: Practical Resource Estimation

Algorithm: Training Time Calculation

Problem: How long to train a 70B parameter dense transformer on 15T tokens using 1,024 H100s?
Solution Steps:

1. Total FLOPs needed: $6 \times$ parameters $\times$ tokens
2. H100 peak performance with MFU $= 0.5$
3. Daily FLOP capacity: $1,024 \times$ H100 FLOPS $\times\ 86,400s$
4. Training time: $\frac{\text{Total FLOPs}}{\text{Daily FLOP capacity}}$

Result: Approximately 144 days of training

Algorithm: Maximum Model Size Estimation

Problem: Largest model trainable on 8 H100s (80GB each) using Adam optimizer?
Memory Breakdown per Parameter:

- Parameters (BF16): 2 bytes
- Gradients (BF16): 2 bytes
- Master weights (FP32): 4 bytes
- Adam first moments: 4 bytes
- Adam second moments: 4 bytes
- **Total: 16 bytes per parameter**

Calculation: $\frac{8\times 80 \text{ GB}}{16 \text{ bytes}} \approx 40$ billion parameters

2 Memory Accounting Fundamentals

2.1 Tensor Foundations

Definition: Tensors in Deep Learning

Tensors are the fundamental building blocks for storing all data in deep learning: parameters, gradients, optimizer states, input data, and activations. Understanding their memory footprint is crucial for resource planning.

Key Point: Memory Calculation Formula

For any tensor, memory usage follows the simple relationship:

$$\text{Memory Usage} = \text{Number of Elements} \times \text{Element Size} \tag{1}$$

This fundamental principle governs all memory accounting calculations.

2.2 Floating Point Representations

Understanding different numerical precisions is essential for memory optimization:

2.2.1 Float32 (FP32)

Definition: FP32 Standard

32-bit floating point with 1 sign bit + 8 exponent bits + 23 fraction bits. This is the gold standard for scientific computing with excellent numerical stability.

Characteristics:

- **Memory:** 4 bytes per element
- **Range:** Approximately 10^{-38} to 10^{38}
- **Precision:** 7 decimal digits

- **Usage:** Master weights, critical computations

2.2.2 Float16 (FP16)

Definition: FP16 Half Precision

16-bit floating point with 1 sign bit + 5 exponent bits + 10 fraction bits. Provides 2× memory savings but with limited dynamic range.

Warning: FP16 Precision Limitations

Values like 1×10^{-8} round to zero in FP16, making it unsuitable for gradient accumulation and other precision-critical operations.

2.2.3 BFloat16 (BF16)

Definition: BFloat16 Brain Float

16-bit floating point with 1 sign bit + 8 exponent bits + 7 fraction bits. Maintains FP32's dynamic range while halving memory usage.

Key Point: BF16 Advantages

BF16 is preferred for modern training because it:

- Maintains FP32's exponent range (preventing underflow/overflow)
- Provides 2× memory savings over FP32
- Supports efficient mixed-precision training
- Has native hardware support on modern GPUs

2.2.4 Float8 (FP8)

Definition: FP8 Ultra-Low Precision

8-bit floating point introduced by NVIDIA in 2022 with two variants balancing resolution vs. dynamic range for extreme memory efficiency.

2.3 Memory Usage Examples

Algorithm: Practical Memory Calculation

Example: Memory usage of a 4×8 matrix with FP32
Calculation:

$$\text{Elements} = 4 \times 8 = 32 \tag{2}$$

$$\text{Element Size} = 4 \text{ bytes (FP32)} \tag{3}$$

$$\text{Total Memory} = 32 \times 4 = 128 \text{ bytes} \tag{4}$$

Performance: Real-World Scale Example

A single matrix in GPT-3's FFN layer:

- Dimensions: $12,288 \times 49,152$ (approximately)
- Elements: ~ 603 million
- Memory (FP32): $603M \times 4 = 2.3$ GB per matrix
- Memory (BF16): $603M \times 2 = 1.15$ GB per matrix

2.4 Mixed Precision Training Strategy

Key Point: Precision Allocation Strategy

Use different precisions strategically based on numerical requirements:

- **Parameters & Optimizer States:** FP32 for numerical stability
- **Forward/Backward Pass:** BF16 for memory efficiency
- **Loss Scaling:** Dynamic scaling to prevent gradient underflow

3 Compute and Hardware

3.1 GPU Memory Management

Key Principle: Always know where your tensors reside (CPU vs GPU).

- **Default:** Tensors created on CPU
- **GPU Transfer:** Use `.to(device)` or create directly on GPU
- **Data Movement Cost:** CPU to GPU transfers are expensive

Example Hardware (H100):

- Memory: 80GB High Bandwidth Memory (HBM)
- Performance: 312 teraFLOPs/s peak

3.2 Tensor Operations and Views

3.2.1 Tensor Storage Structure

Definition: PyTorch tensors are pointers to allocated memory with metadata specifying access patterns.

[Insert diagram showing tensor storage layout with strides]

Stride Calculation: For a 4×4 matrix stored row-major:

- **Stride 0:** 4 (elements to skip for next row)
- **Stride 1:** 1 (elements to skip for next column)

Element Access: Element at position (i, j) located at:

$$\text{Index} = i \times \text{stride}_0 + j \times \text{stride}_1 \tag{5}$$

3.2.2 Views vs Copies

View Operations (No Memory Allocation):

- Row/column slicing: `x[0]` or `x[:, 1]`
- Reshape: `x.view(new_shape)`
- Transpose: `x.transpose()`

Copy Operations (New Memory):

- `contiguous()`: Creates copy for non-contiguous tensors
- Element-wise operations: `x + y`, `torch.relu(x)`

Contiguity: Tensors are contiguous if elements can be accessed by incrementing through memory sequentially.

4 Matrix Operations

4.1 Basic Matrix Multiplication

Standard Case: $A_{16\times32} \times B_{32\times2} = C_{16\times2}$

Batched Operations: Common pattern in deep learning

$$X_{[B,S,D_1]} \times W_{[D_1,D_2]} = Y_{[B,S,D_2]} \tag{6}$$

where B = batch size, S = sequence length, D = feature dimension.

4.2 Einstein Summation (Einops)

Motivation: Replace error-prone index notation (e.g., `x[..., -2, -1]`) with explicit dimension naming.

Basic Syntax:

- Name all dimensions explicitly
- Dimensions not in output are summed over
- Use `...` for broadcasting over arbitrary dimensions

Examples:

Matrix Multiplication:

```
# Traditional: torch.matmul(x, y.transpose(-2, -1))
# Einops: einsum(x, y, "batch seq1 hidden, batch seq2 hidden -> batch seq1 seq2")
```

Reduction:

```
# Traditional: x.mean(dim=-1)
# Einops: reduce(x, "batch seq hidden -> batch seq", "mean")
```

Rearrange:

```
# Unpack flattened dimensions
rearrange(x, "batch seq (heads hidden) -> batch seq heads hidden", heads=8)
```

5 Computational Cost Analysis

5.1 Floating Point Operations (FLOPs)

Definition: A floating-point operation includes:

- Addition: $a + b$
- Multiplication: $a \times b$
- Multiply-accumulate: $a \times b + c$ (counts as 2 FLOPs)

Terminology Clarification:

- **FLOPs (lowercase s):** Number of floating-point operations (measure of work)
- **FLOP/s:** Floating-point operations per second (measure of speed)

5.2 Training Scale Examples

Model	Training FLOPs
GPT-3	3.23×10^{23}
GPT-4	$\sim 2 \times 10^{25}$ (estimated)

Table 1: Training compute requirements for major language models

5.3 Hardware Performance

A100 Peak Performance: 312 teraFLOP/s

Regulatory Context:

- **US Executive Order (revoked):** Models ¿ 10^{26} FLOPs must be reported
- **EU AI Act:** Models ¿ 10^{25} FLOPs subject to regulation

6 Resource Accounting Framework

6.1 Memory Budget Components

For each parameter, account for:

1. **Parameter weights:** FP32 (4 bytes)
2. **Gradients:** FP32 (4 bytes)
3. **Optimizer state:** Adam requires 2 additional FP32 values (8 bytes)
4. **Total:** 16 bytes per parameter

6.2 Training Cost Estimation

FLOPs per Token Formula:

$$\text{FLOPs per token} = 6 \times N \tag{7}$$

where N is the number of parameters.

Total Training FLOPs:

$$\text{Total FLOPs} = 6 \times N \times T \tag{8}$$

where T is the number of training tokens.

Training Time Estimation:

$$\text{Training Time} = \frac{\text{Total FLOPs}}{\text{Hardware FLOP/s} \times \text{MFU} \times \text{Number of GPUs}} \tag{9}$$

where MFU (Model FLOPs Utilization) is typically 0.3-0.5.

7 Best Practices

7.1 Memory Efficiency

- Use views instead of copies when possible
- Be explicit about tensor device placement
- Monitor memory allocation patterns
- Consider mixed precision training

7.2 Code Organization

- Use einops for clear dimension handling
- Add assertions for tensor shapes and devices
- Document tensor dimensions in comments or type hints
- Profile memory and compute usage regularly

8 PyTorch Resource Accounting Mind Map

8.1 Mind Map Structure

Central Concept: PyTorch Resource Accounting

Main Branches:

1. **Memory Management (Green):**
   - Data type considerations (FP32, BF16)
   - Mixed precision strategies
   - Parameter memory budgeting (16 bytes/param)
2. **Tensor Operations (Yellow):**
   - View vs copy operations

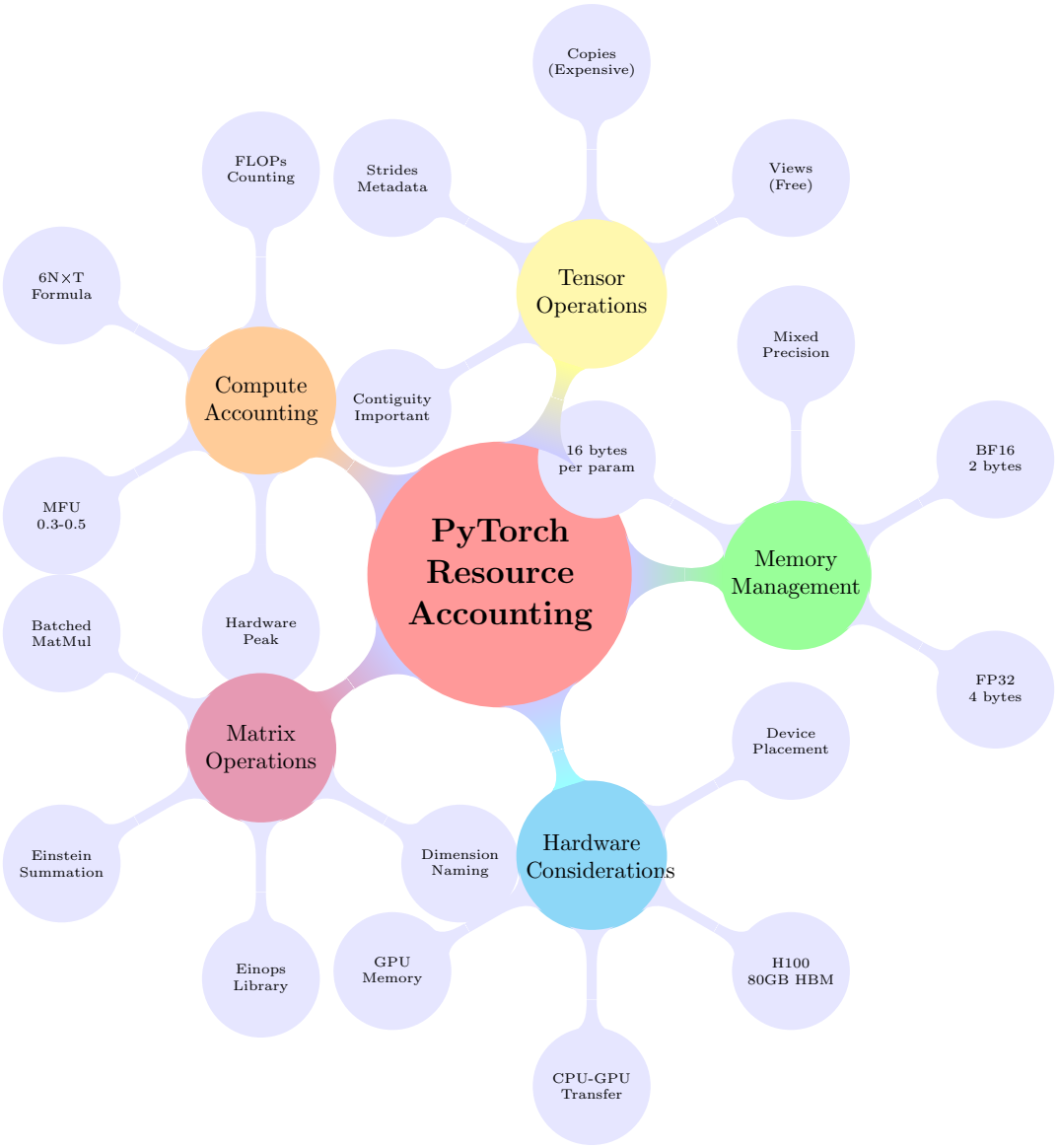

Figure 1: Mind Map of PyTorch Resource Accounting Concepts

- Stride and storage metadata
- Contiguity requirements

3. **Compute Accounting (Orange):**
   - FLOP counting methodology
   - 6N×T training formula
   - Model FLOP utilization (MFU)
   - Hardware performance metrics
4. **Matrix Operations (Purple):**
   - Batched matrix multiplication patterns
   - Einstein summation notation
   - Einops library for dimension management

5. **Hardware Considerations (Cyan):**
    - GPU memory constraints
    - CPU-GPU data transfer costs
    - Specific hardware capabilities (H100)
    - Device placement strategies

Key Interconnections:

- Memory and compute are fundamentally linked
- Tensor operations affect both memory usage and computational efficiency
- Hardware constraints drive precision and batching decisions
- Proper resource accounting enables scaling predictions