DSC 204A: Scalable Data Systems, Fall 2025
Lecture 13: Machine Learning Systems II

Lecturer: Hao Zhang
Scriber: Aritra Das, Sirui Cao, Yanghe Sun, Xuanwen Hua
Halıcıoğlu Data Science Institute
University of California, San Diego
La Jolla, CA
ardas@ucsd.edu, x2hua@ucsd.edu, sic038@ucsd.edu, yas029@ucsd.edu

1 Introduction

This lecture continues the discussion on Machine Learning Systems (MLSys), focusing on bridging the gap between high-level model definitions and efficient hardware execution. The discussion covers the evolution of deep learning frameworks, the mechanics of automatic differentiation, the layers of the MLSys stack (graph optimization, parallelization, runtime, and operators), and an introduction to Large Language Models (LLMs) from a systems perspective.

2 Deep Learning Frameworks and Compilation

2.1 Imperative vs. Symbolic Programming

Deep learning frameworks have historically fallen into two categories:

- **Imperative (Define-by-Run):** Represented by PyTorch. The computation graph is constructed dynamically as the code is executed. This approach offers excellent debugging capabilities and flexibility (e.g., using Python control flow) but often sacrifices performance due to the lack of a global graph view for optimization.
- **Symbolic (Define-and-Run):** Represented by TensorFlow (v1). The user defines the entire computation graph first, which is then compiled and executed. This allows for extensive optimizations (operator fusion, memory planning) but makes debugging difficult and the learning curve steeper.

2.2 Just-In-Time (JIT) Compilation

Modern frameworks aim to combine the best of both worlds: the ease of imperative programming with the performance of symbolic execution. This is achieved through Just-In-Time (JIT) compilation, a technique central to frameworks like PyTorch and JAX.

In PyTorch 2.0, this is realized via `torch.compile()`. A user writes standard Python code (imperative), and by decorating a function with `@torch.compile()`, the framework extracts the code, translates it into a symbolic graph, optimizes it (graph lowering), and compiles it into efficient machine code. This transition moves the paradigm from "Define-by-Run" to "Define-then-Run" during deployment, while maintaining "Define-by-Run" during development. The compiler has a global view of the entire program, enabling significant optimizations before launching the efficient binary code to the device.

Limitation: JIT compilation relies on static graphs. Models with dynamic control flow (where the graph structure changes based on input data, e.g., the decode phase of LLM inference, where the graph changes with each generated token) present challenges for static optimization because the graph

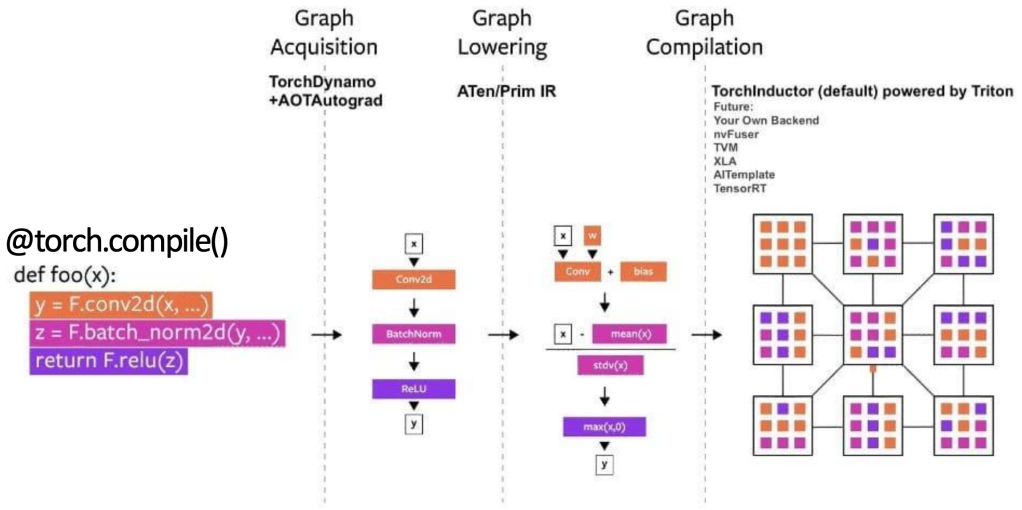


Figure 1: JIT Compilation

must be re-optimized/re-compiled for changing conditions. A direct consequence of this compilation process is that the first iteration is slow, because it is optimizing a graph, which takes a lot of CPU time. This initial overhead is known as the "warm-up" cost. Subsequent iterations then benefit from the optimized code

3 Automatic Differentiation (Autodiff)

To train models, frameworks must compute gradients via backpropagation. Modern systems explicitly construct a **backward computation graph** that complements the forward graph.

3.1 Mathematical Formulation

Given a function $y = f(x_1, x_2, \dots)$, we compute gradients using the chain rule. In Reverse Mode Autodiff, we calculate the **adjoint** ($\bar{v}_i$) for each node v_i, defined as the partial derivative of the output y with respect to v_i:

$$\bar{v}_i = \frac{\partial y}{\partial v_i}$$

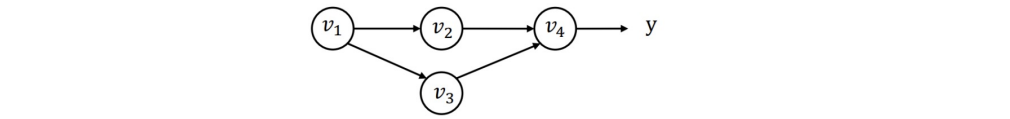
For a node v_i that acts as an input to multiple consumers $v_j \in \text{next}(i)$, the adjoint is the sum of the partial adjoints from all consumers:

$$\bar{v}_i = \sum_{j \in \text{next}(i)} \bar{v}_j \frac{\partial v_j}{\partial v_i}$$

3.2 Implementation Loop

Frameworks implement autodiff by traversing the forward graph in reverse topological order. The algorithm conceptually follows these steps:

1. Initialize the adjoint of the output node to 1 (since $\frac{\partial y}{\partial y} = 1$).
2. Iterate through nodes in reverse topological order.
3. For the current node, sum all accumulated partial adjoints to get the full adjoint.
4. For each input (producer) of the current node, compute the partial adjoint using the local derivative (e.g., if $z = x \cdot y$, $\frac{\partial z}{\partial x} = y$) and the current node's adjoint.



How to derive the gradient of v_1

$$\overline{v_1} = \frac{\partial y}{\partial v_1} = \frac{\partial f(v_2,v_3)}{\partial v_2}\frac{\partial v_2}{\partial v_1} + \frac{\partial f(v_2,v_3)}{\partial v_3}\frac{\partial v_3}{\partial v_1} = \overline{v_2}\frac{\partial v_2}{\partial v_1} + \overline{v_3}\frac{\partial v_3}{\partial v_1}$$

Figure 2: an example of reverse-mode automatic differentiation using chain rule

5. Propagate this partial adjoint to the producer's list of gradients.

This process constructs a backward graph where nodes represent gradient computations (e.g., matrix multiplication transpose, element-wise derivative) and edges represent the flow of gradient data. Finally, optimizer nodes (e.g., SGD update: $\theta \leftarrow \theta - \eta\nabla$) are appended to the graph.

3.3 Explicit Graph Construction vs. On-the-Fly Execution

A key distinction discussed in the lecture is the evolution of autodiff implementation. Early frameworks like **Caffe** performed backpropagation "on the fly"—they took the forward graph and executed backward steps numerically without generating a new graph structure. Modern frameworks (TensorFlow, PyTorch) explicitly construct a materialized backward graph. This separation allows the system to apply the same optimization techniques (kernel fusion, scheduling) to the backward pass as it does to the forward pass.

4 The MLSys Stack

The "Grand Problem" of MLSys is to take a user-defined dataflow graph and execute it on a cluster of diverse hardware (GPUs, TPUs, etc.) while maximizing speed, scalability, and memory efficiency. This is solved via a layered optimization stack:

4.1 Graph Optimization

The system rewrites the computation graph G into an equivalent but more efficient graph G'.

- **Kernel Fusion:** Combining multiple operations into a single kernel to improve Arithmetic Intensity (operations per byte of memory transfer).
- **Example:** In Multi-Head Attention, computing Q, K, V involves three separate matrix multiplications: XW_Q, XW_K, XW_V. A graph optimizer can fuse these into a single matrix multiplication XW_{QKV} followed by a split, reducing kernel launch overhead and memory I/O.
- **Methods of Optimization:** The lecture highlighted two primary approaches to finding efficient graph rewrites:
  1. **Rule-Based (Templates):** Experts write specific templates (e.g., "If you see A + B + C, replace it with D"). This is the classical compiler approach.
  2. **Auto-Discovery (Search):** The system iteratively tries merging nodes or altering the graph structure and benchmarking the results to discover new optimizations automatically (conceptually similar to Neural Architecture Search).
- **Mega Kernels:** An extreme case of fusion discussed was the "Mega Kernel" approach (e.g., by researchers at Stanford), where the entire computation graph of a model (like Llama) is fused into a single massive operator. While efficient, this makes the model impossible to debug.

4.2 Parallelization

Partitioning the graph across a cluster of devices.

- **Data Parallelism:** Replicating the model on every worker and splitting the data.
- **Model Parallelism:** Partitioning the graph itself when the model is too large for a single GPU.
- The goal is to maximize computation on high-bandwidth intra-node connections (e.g., NVLink) and minimize communication over slower inter-node connections (e.g., Ethernet/InfiniBand).

4.3 Runtime and Scheduling

Orchestrating execution to minimize idle time ("bubbles") and manage memory. This involves scheduling compute and communication to overlap whenever possible and managing tensor allocation to prevent Out-Of-Memory (OOM) errors.

4.4 Operator Optimization

The lowest level involves writing highly optimized kernels (e.g., in CUDA) for specific hardware.

- Optimization depends on hardware characteristics: tensor core availability, precision support (FP16, FP8, INT8), and memory hierarchy.
- Libraries like cuDNN or Triton provide optimized implementations for standard shapes (e.g., matrix multiplication of specific sizes), but custom shapes or operators may require bespoke kernel implementations.

5 Q & A part

Q1: Just-in-Time Compilation

Question:
Ideally, we want *define-and-run* during _____.
We want *define-then-run* during _____.
How can we combine the best of both worlds?

Answer:
We want define-and-run during **development** and define-then-run during **deployment**.
We can combine them using **Just-in-Time (JIT) compilation** (e.g., PyTorch's `torch.compile()`). This allows us to write models imperatively (flexible debugging) while compiling them into an optimized graph for efficient deployment.

Q2: The Problem with JIT

Question: What is the main limitation of JIT?

Answer: It requires (almost) **static computation graphs**.

Explanation:
JIT compilers work best when shapes, control flow, and operations are fixed. Typical PyTorch code uses dynamic control flow (Python `if` loops) and dynamic shapes. To use JIT, we must often rewrite or restrict code to fit a static graph, making the models less flexible and harder to debug.

Q3: Reverse-mode AD Calculation

Question: Compute $\frac{\partial y}{\partial x_1}$ for the function below using reverse-mode AD:

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$$

(Assuming inputs derived from context: $v_1 = 2, v_2 = 5$)

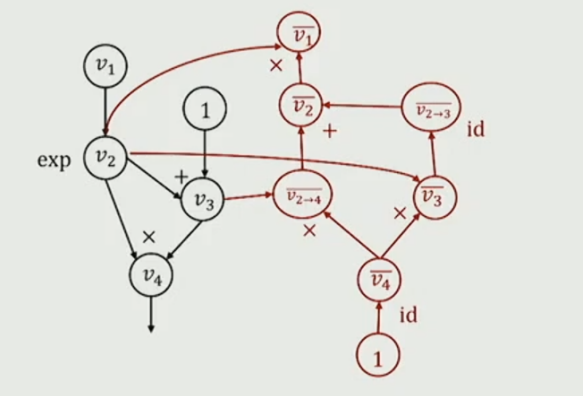
Answer: Define adjoints $\bar{v}_i = \frac{\partial y}{\partial v_i}$ and compute in reverse topological order starting from output $v_7 = y$.

$$\begin{aligned}
\bar{v}_7 &= \frac{\partial y}{\partial v_7} = 1 \\
\bar{v}_6 &= \bar{v}_7 \cdot \frac{\partial v_7}{\partial v_6} = 1 \cdot 1 = 1 \\
\bar{v}_5 &= \bar{v}_7 \cdot \frac{\partial v_7}{\partial v_5} = 1 \cdot (-1) = -1 \\
\bar{v}_4 &= \bar{v}_6 \cdot \frac{\partial v_6}{\partial v_4} = 1 \cdot 1 = 1 \\
\bar{v}_3 &= \bar{v}_6 \cdot \frac{\partial v_6}{\partial v_3} = 1 \cdot 1 = 1 \\
\bar{v}_2 &= \bar{v}_5 \frac{\partial v_5}{\partial v_2} + \bar{v}_4 \frac{\partial v_4}{\partial v_2} = \bar{v}_5 \cos(v_2) + \bar{v}_4 v_1 \\
&= (-1)\cos(5) + 1(2) \approx 1.716 \\
\bar{v}_1 &= \bar{v}_4 \frac{\partial v_4}{\partial v_1} + \bar{v}_3 \frac{\partial v_3}{\partial v_1} = \bar{v}_4 v_2 + \bar{v}_3 \frac{1}{v_1} \\
&= 1(5) + 1\left(\frac{1}{2}\right) = 5.5
\end{aligned}$$

$$\therefore \quad \frac{\partial y}{\partial x_1} = \bar{v}_1 = 5.5$$

Q4: Computation Graphs in ML

Question: What is missing from the computational graph below regarding ML training?



Answer: The **parameter update step** (the optimizer).

Explanation:
The graph likely shows the Forward and Backward pass to compute gradients. However, ML training requires updating the weights θ based on those gradients:

$$\theta \leftarrow \theta - \eta \nabla_\theta L(\theta)$$

Without this optimization step (e.g., SGD), the model parameters never change.