DSC-204A: Scalable Data Systems, Fall 2025

12: Machine Learning Systems - I

Lecturer: Hao Zhang *Scribe: Junda Su, Peng Wang, Shreejith Suthraye Gokulnath, Xiaotian Shao*

1 Overview

This lecture introduces the evolution of Machine Learning (ML) Systems, discussing how ideas from large-scale data systems (e.g., Spark) inspired new architectures for distributed ML training. The design is always focusing on specification over ML algorithms, from granular to finer tuned models, representing by MapReduce, followed by Spark, till Pytorch and Tensorflow, fulfilling the different requirements in computation.

2 ML System History

The ML era began roughly around 2008, before Spark took off. Early ML workloads were extremely diverse, covering graphical models, topic models, random forests, and linear models. They require different computation paradigms and optimization methods, which offered great opportunities for researchers at time. The disadvantage is that there was no unified programming abstraction or runtime that could efficiently support all these diverse workloads. So it was extremely hard for system researchers to work out a general solution for efficient machine learning frameworks.

Gradually, the ML ecosystem evolved from diversity to unification with a few dominant models and training paradigms as in Figure 1:

- Iterative-Convergent Algorithms: repeatedly update parameters until convergence (e.g., gradient descent, EM, coordinate descent).
- Neural Networks (Especially Transformer and LLM): Unified model architecture trained using stochastic gradient descent (SGD).

3 Gradient / backward computation

Most ML models can be expressed using the iterative update rule:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t, x)$$

where parameters θ are updated via gradients of the loss L. These updates must be distributed across multiple devices to scale training.

Challenges in expressing in Spark:

- ML is too diverse; hard to express their computation in coarse-grained data transformations.

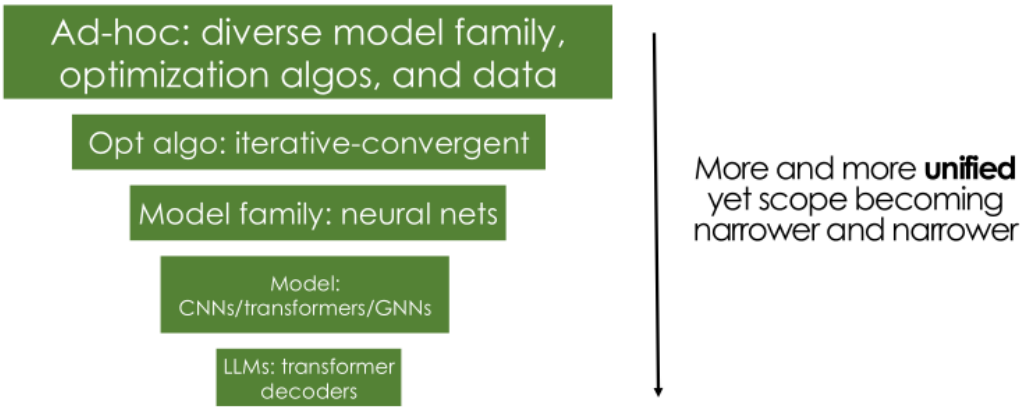

Figure 1: Evolution of ML systems: from diverse models to unified architectures

- Very heavy communication per iteration
- Compute time : communication time = 1:10 in the era of 2012

4 Consistency in ML Systems

When distributing gradient descent across devices, a decision must be made on how and when devices synchronize their parameters.

4.1 Strict Consistency: Bulk Synchronous Parallel (BSP)

Represented by MapReduce, Spark, and early distributed ML systems. All workers compute updates in lock-step and synchronize at iteration boundaries.

4.2 Asynchronous Communication

Removes global barriers entirely so that workers proceed independently and communicate whenever it's ready. The advantage is maximize throughput while the disadvantage is there is no consistency guarantees. There might be divergence of model parameters.

4.3 Bounded Consistency: Stale Synchronous Parallel (SSP)

It was initially proposed as a middle ground between BSP and full asynchrony. Workers are allowed to drift up to s iterations with "Lazy" communication - only sync when staleness violated

- Reduces communication cost.
- Trades off strict convergence guarantees for higher system throughput.
- When $s = 0$, equivalent to BSP; when $s \to \infty$, equivalent to fully asynchronous.

5 Parameter Server Architecture

The Parameter Server (PS) is proposed to implement distributed iterative-convergent algorithms efficiently while supporting flexible consistency.

It's motivated to solve the following problems:

- Heavy communication per iteration in synchronous training.
- Scalable and fault-tolerant parameter aggregation.

It's designed with Sharded Key-Value Store and has advantages such as:

- Handles iterative-convergent algorithms efficiently.
- Reduces communication bottlenecks.
- Scales to large models and clusters.

But it also has disadvantages:

- CPU-based; difficult to extend to GPU clusters.
- Convergence slows with excessive staleness or poor synchronization.

6 Deep Learning Era: The Second Unification

With the rise of neural networks (2012–2015), ML entered the deep learning era.

- Still iterative-convergent (via SGD).
- GPUs became mainframe for training.
- Models (CNNs, RNNs, Transformers) required expressing computation as fine-grained tensor operations.

Spark's RDD operators were too coarse-grained for this workload, leading to the development of specialized deep learning frameworks.

7 Deep Learning Libraries and Computational Graphs

The core design of deep learning frameworks (e.g., TensorFlow, Pytorch) are based on dataflow graph and Auto-differentiable Libraries.

And the key components includle:

- Model architecture: connecting math primitives to define the forward computation.

- Objective function: Defines loss (e.g., MSE, cross-entropy).
- Optimizer: Algorithm that updates parameters (e.g., SGD).
- Data: Training inputs and labels.

A typical example is TensorFlow v1 on Logistic Regression

8 After here is Xiaotian's version of last 20 min lecture

To recap, the process involves building up a **data flow graph** (Figure 2) through the following steps:

1. We generate the **model prediction**.
2. We compute the **cross entropy loss**.
3. We perform **automatic differentiation (Autodiff)**. Note that this part builds the **backward graph** precisely and will be discussed in a later lecture.
4. Lastly, `sess.run` gets the graph to **execute in the most optimal way**.

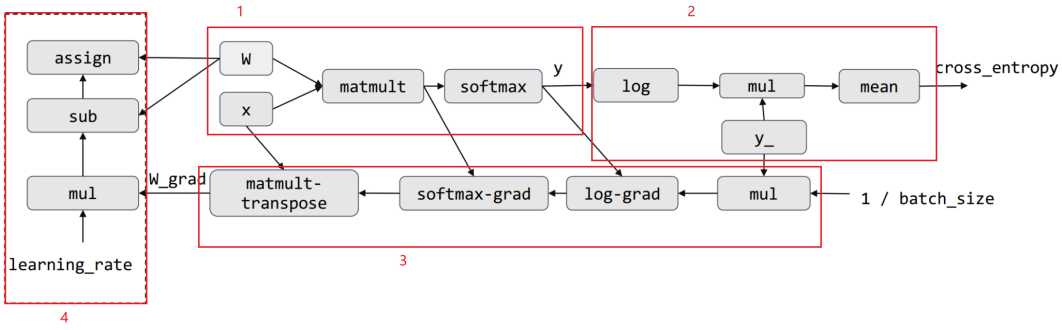

Figure 2: Data Flow Graph Behind the Code

Dissucsion

Q: What are the benefits of computational graph abstraction?

A:

- You can automatically obtain the **backward pass (gradient computation)**, which is difficult to implement manually.
- It is **universal**, regardless of how the neural network architecture changes.

Q: What are possible implementations and optimizations on this graph?

A:

- For example, you can make each **operator run faster**.
- You can **fuse one path** (combine sequential operations) to **reduce I/O overhead**.
- You can also **distribute the computation** across many devices, but you must also consider the **communication problem**.

Q: What are the cons of computational graph abstraction?

A:

- **No intermediate values** makes **debugging difficult**.
- The graph is **static**, but neural networks are often not. However, there are ways to improve this, which will be discussed later.

9 PyTorch, Another flavor

TensorFlow learned a lot from Spark's design, but nowadays we prefer another framework, **PyTorch**, which does not use symbolic execution. For example, we directly get the result from `torch.mm`. Instead, PyTorch **dynamically creates the graph** and copes well with the **imperative programming style** used by Python.

Symbolic vs. Imperative

Here, we simply summarize the key characteristic of symbolic and imperative programming:

- **Symbolic Style**: **Define-then-Run**.
- **Imperative Style**: **Define-and-Run**.
- **Symbolic Programming**:
  - **Pros: Easy to optimize**. Users get a global view of the computation, making it clear where to optimize. As a result, it can be significantly **more efficient** (e.g., 10 times) than imperative programming.
  - **Cons: Counter-intuitive**, as Python itself is imperative. It is also **hard to debug** and **less flexible**, as you need to define the entire computation carefully beforehand.
- **Imperative Programming**:
  - **Pros: More flexible** and **easy to program and debug**.
  - **Cons: Less efficient** and **more difficult to optimize globally**.

In-class Quiz

Here we categorize some famous programming languages:

- **Symbolic:** C++, SQL
- **Imperative:** Python

9.1 Something Interesting

An interesting point is that TensorFlow used a symbolic style while having Python as the primary interface language. What happens behind the scenes is that you are essentially using a **Domain-Specific Language (DSL)** built on top of Python. The PyTorch DSL is considered **more "Pythonic"** than the original TensorFlow DSL.

10 History of ML Frameworks

In history, people built different kinds of ML frameworks (Figure 3).

- **Torch** was an early version using Lua.
- **PyTorch** borrowed many ideas from DyNet and Chainer.
- **DMLC MXNet** once had a large community but was acquired by Amazon and died in the end.
- **Caffe** was a very early framework before TensorFlow.
- **Caffe2** was an evolved version and later merged into PyTorch.

In 2024, only **JAX, PyTorch, and TensorFlow** survive (Figure 4).

After-Class Question: Why PyTorch took more market share than TensorFlow despite being a later framework.

The lecturer gave two comments:

1. **Imperative programming is more convenient**, and **academia prefers it**.
2. It is related to **compilation** and will be discussed in the next lecture.

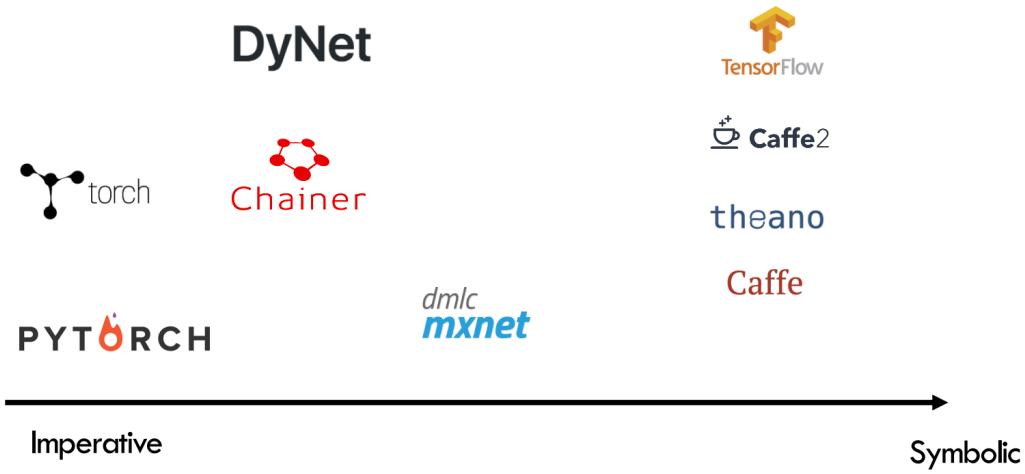

Figure 3: ML Frameworks in 2016

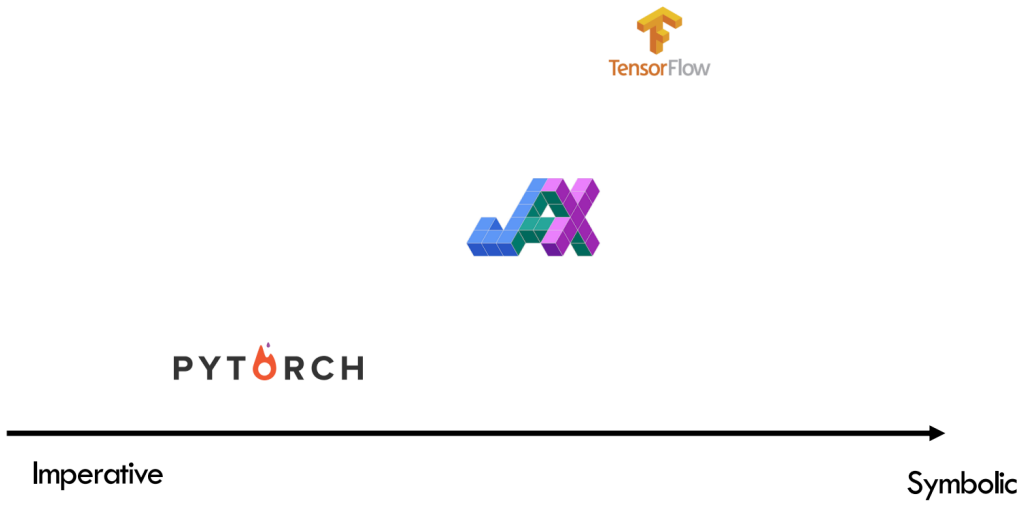

Figure 4: ML Frameworks in 2024