Week 2 Reading Summaries: Tensorflow + PyTorch Readings

Amogh Patankar
University of California, San Diego
3869 Miramar Street Box #3037, La Jolla, CA- 92092
apatankar@ucsd.edu

Abstract

With the advent of AI and nowadays, generative AI, there have been continuous innovations and improvements with respect to building systems at scale. Amongst these, TensorFlow and PyTorch, developed by Google and Meta, respectively, operate at large scale(s), focusing on usability, speed, and efficiency. TensorFlow uses dataflow graphs to represent computations, and operations mutating a given state. This allows the system to map nodes across machines, and even within a machine. This had led to numerous optimizations and algorithms, and more importantly, strong support for inference on DNNs. PyTorch provides a similar system allowing for easy coding and debugging, and is an upgrade for many scientists trying to work with deep neural nets, and other AI models.

1 TensorFlow

1.1 Introduction

In the ML domain, many key advances have been driven by creation and adoption of more advanced methods, quality and quantity of data, and software platforms to ease modeling use cases. Google introduced TensorFlow, a system built for experimentation, specifically for training and inference. The idea behind TensorFlow is multi-faceted, but primarily relies on using a multitude of servers, performing local inference, while also being flexible to allow new models and optimizations.

1.2 Dataflow Graph

TensorFlow uses a unified dataflow graph, representing computation and state of algorithm. Primarily, this comes from data flow system models, and parameter server models, but Tensorflow allows vertices to represent computations. These computations own/update the mutable state, and this is quite different from traditional systems, where vertices are computation on immutable data. In this graph model, edges are tensors, which are multi-dimensional arrays, and the distributed subcomputations are contain communication, as inserted by TensorFlow.

The unification of state management and computation allows TensorFlow to experiment with various parallelization methods to execute ideas like offline computation.

1.3 TensorFlow Requirements

TensorFlow requires a few requirements as a machine learning system at scale, namely, distributed execution, accelerator support, training and inference support, and extensibility.

Submitted to 38th Conference on Neural Information Processing Systems (NeurIPS 2024). Do not distribute.

1.3.1 Distributed Execution

TensorFlow uses distributed execution to ensure that a cluster of computational power can be used to solve problems, and especially, problems that scale quickly. TensorFlow uses distributed execution which is incredibly useful in ML, where more data often times improves performance. Distributed execution can shard or mini-batch models, using the network effectively, simultaneously reading and updating a particular model.

1.3.2 Accelerator Support

ML algorithms require matmuls and convolutions, operations that require high computational power. Because of this, hardware companies around the planet try to build specialized chips to execute ML workloads. GPUs from Nvidia are the most popular accelerators, while companies like d-matrix.ai, Cerebras, Groq, and more, build chips with tech like DIMC, large single chip processors, LPUs, etc (respectively). Special-purpose accelerators allow for significant performance improvements as compared to traditional processors; in TensorFlow's case, they use Google TPUs to accelerate and improve performance.

1.3.3 Training and Inference Support

TensorFlow implements highly performant, scalable systems by allowing developers to use the same code for training and inference.

1.3.4 Extensibility

Tensorflow allows users to scale the same code to run in prod as opposed to sandbox environments.

1.4 Similar/Related Work

1.4.1 Single Machine Frameworks

Tensorflow uses a programming model similar to Theano's dataflow representation.

1.4.2 Batch Dataflow Systems

TensorFlow is able to train larger workloads by using a batch dataflow system, allowing TensorFlow to train these workloads on larger clusters, with shorter timesteps.

1.4.3 Parameter Servers

Tensorflow uses a parameter server architecture, used to scale training, GPU acceleration, and a model that interfaces for many programming languages. They use a high level model that allows users to customize their code.

1.5 TensorFlow Execution Model

TensorFlow uses a dataflow graph for ML which allows for parallel and distributed computation. It supports mutable state and concurrent executions, both of which are key to train large workload, allowing experimentation with optimization, consistency, and parallelization strategies.

1.5.1 Dataflow Graph Elements

TensorFlow uses a computation graph consisting of vertices (operations) and edges (tensors). Operations define computations and can have attributes like type and input/output specifications, while tensors are dense arrays. TensorFlow uses dense tensors for simplicity but allows for sparse tensors; stateful operations like variables and queues, enable the "mutable" state, essential for model training. Variables store shared parameters and support operations like Read and AssignAdd, while queues coordinate and synchronize input pipelines. These features let TensorFlow efficiently handle large-scale computation, allow dynamic control flow, and enable flexible experimentation with machine learning algorithms.

1.5.2 Partial + Concurrent Execution

TensorFlow uses a dataflow graph for computations, allowing for specification and concurrent subgraph execution. Typically, multiple subgraphs interact through shared variables and queues, enabling data-parallel training and fault tolerance. TensorFlow's flexibility is due to partial and concurrent execution, and queues, allowing asynchronous execution for weak-consistency algorithms

1.5.3 Distributed Execution

TensorFlow uses dataflow graphs for distributed execution across devices (CPU/GPU/TPU/LPU), where operations are placed on devices, using send/rec ops for cross-device communication. By caching and reusing subgraphs, favoring static graphs but supporting dynamic computations, latency decreases, but automating optimal placement is a challenge.

1.5.4 Differentiation and Optimization

TensorFlow allows for ML algorithms with conditional and iterative control flow and uses dynamic control flow to deal with variable-length sequences and iterative computations. These primitives allow TensorFlow to partition and execute control flow constructs across multiple devices, supporting parallelism and distributed execution. TensorFlow also enables control flow auto-diff, i.e. computing gradients in parallel across devices.

1.6 Evaluation

TensorFlow performs competitively when put up against other ML frameworks, performing the best for certain workloads.

2 PyTorch

2.1 Introduction

Many frameworks use a static dataflow graph, and apply computation to batches of data; this prioritizes visibility over easy of use/debugging, and computational flexibility. PyTorch performs dynamic tensor calculations with autodiff and GPU acceleration.

2.2 Background

Scientific computing as a whole has had four main paradigms. These include using objects as opposed to multidimensional arrays, autodiff, the utilization of open-source pythonic ecosystem packages like numpy/scipy/pandas, and the availability of hardware [accelerators] for ML workloads. PyTorch builds on ALL these aspects of computing.

2.3 Design Principles

PyTorch has four main design principles: being pythonic, researchers, first, provide pragmatic performance, and "worse is better". These principles, allow researchers and scientists to use common interfaces, make working as simple yet productive as possible, ensure great performance, and also allow for efficient, half-cooked solutions as opposed to complete, rigid solutions.

2.4 Usability Centric Design

2.4.1 DL = Python?

PyTorch preserves the abilities and simplicity of Python, and extends it to all aspects of deep learning-layer definition, dataloading, optimization, and parallelization. This ensures new architectures can be implemented consistently easily; models are represented as classes to follow general purpose programming philosophy.

2.4.2 Interoperability and Extensibility

Interoperability is a key priority, as PyTorch hopes to leverage Pythonic libraries, and allows mechanisms to convert data from type to type (numpy array <—-> PyTorch tensor). PyTorch systems are designed to be extensible; a key example is autodifferentiation. PyTorch allows the programmer to specify function, derivative, and much more, allowing interchangeability, flexibility, and easy of use.

2.4.3 AutoDifferentiation

PyTorch uses operator overloading to perform autodiff, by reversemode differentiating, i.e. computing the gradient of a scalar w.r.t. multivariate input, executing more efficiently. In this way, PyTorch allows for easy extension, and ensures safety for tensor mutation as well.

2.5 Performance Focused Implementation

PyTorch optimizes execution in every aspect.

2.5.1 C++ Core

The core of PyTorch is written in C++, to provide primitives, autodiff, and that computation can be done in a multithreaded manner, as opposed to Python's global interpreter lock.

2.5.2 Separate Control + Data Flow

There is a strict separation between control and data flow; the former is done in Python and C++ on CPU, and operators are run on CPU/GPU. PyTorch allows operator execution on GPUs through CUDA, allowing system execution overlap of Python CPU code and tensor ops on GPU.

2.5.3 Custom Caching tensor allocator

PyTorch uses a GPU memory allocator, avoiding bottlenecks from cudaFree. It caches and reuses memory, rounding allocations to reduce fragmentation, and maintains memory pools for each CUDA stream. This allows for minimal use of multiple streams, simplifying reuse and synchronization.

2.5.4 Multiprocessing

Python's Global Interpreter Lock prevents parallel thread execution, so PyTorch extends Python's multiprocessing module, using shared memory for tensor data instead of inefficient serialization, improving performance. PyTorch supports parallel programs on GPUs with gradient synchronization and handles CUDA tensor sharing too.

2.5.5 Reference Counting

PyTorch treats GPU memory as a scarcity, which is key for memory-intensive tasks like batch training. PyTorch uses reference counting to track tensor usage, freeing memory when tensor references drop to zero, and integrate with Python's reference counting. This ensures efficient memory usage; however, it also requires implementations supporting reference counting or custom memory management.

2.6 Evaluation

Across a wide variety of tasks, PyTorch is benchmarked against commonly-used DL and ML libraries, and achieves competitive performance for a variety of tasks.