Reading Summary Week 2

Aakash Agrawal
HDSI
University of California San Diego
San Diego, CA, 92092
aaa015@ucsd.edu

1 TensorFlow: A system for large-scale machine-learning

TensorFlow is a machine learning framework that uses a **static dataflow graph**, where computations are represented as nodes (operations) and edges that carry tensors between nodes. This graph is defined before execution, allowing for optimizations and efficient execution across different hardware platforms. The static nature of the graph means it can be abstracted out, leading to a clean and powerful representation of the computation.

1.1 Desiderata of large-scale ML Systems

The paper outlines several key requirements for a large-scale ML framework, setting the stage for the discussion of TensorFlow.

- **Distributed Execution**: A distributed system splits the model across multiple processes to increase network bandwidth, allowing many workers to read and update the model at the same time. From a data perspective, a distributed file system helps eliminate I/O bottlenecks by enabling parallel access to input data. Additionally, preprocessing operations can be applied independently to the input data, further optimizing performance and scalability.
- **Accelerator Support**: ML models perform extensive computations, such as matrix multiplication and multi-dimensional convolutions, that can greatly benefit from hardware accelerators like GPUs and TPUs. ML systems should be designed to support a wide range of hardware devices, ensuring compatibility and efficient utilization of these accelerators to maximize performance and speed up model training and inference.
- **Training & Inference support**: Scalable and high-performance training and inference are essential for deploying deep learning models in production. A unified, well-optimized system that supports both training and inference ensures consistency and efficiency, allowing developers to use the same code for both stages.
- **Extensibility**: The system should enable experimentation, support stateful constructs, and allow expressive control flow for implementing complex models and algorithms.

1.2 TensorFlow execution model

The dataflow approach in TensorFlow enables parallel execution of independent computations and efficient partitioning across multiple distributed devices.

- **Partial and concurrent execution**: TensorFlow supports partial execution, allowing users to run a part of the computation graph rather than the entire graph. Concurrent execution refers to the ability to execute multiple operations or tasks simultaneously, which helps improve efficiency. TensorFlow enables this by leveraging stateful operations such as variables and queues, allowing different parts of the graph to run in parallel while managing dependencies effectively.

- **Distributed execution**: Dataflow simplifies distributed execution by making communications between sub-computations explicit. TensorFlow partitions the operations in a graph into per-device subgraphs across a cluster of devices. These subgraphs are cached on their respective devices and can be reused, improving efficiency and resource utilization in a distributed environment.
- **Dynamic control flow**: Some machine learning algorithms require dynamic control flow based on input conditions. TensorFlow supports this by providing primitives like Switch and Merge, which enable the implementation of loops and conditional control flows within the computation graph, allowing for more flexible and adaptable models.

1.3 Cases for Extensibility

The paper highlights four extensions of TensorFlow built using simple dataflow primitives and "user-level" code.

- **Differentiation and Optimization**: Computing gradients is fundamental to training deep learning models. TensorFlow allows users to define neural networks as a composition of layers and a loss function, automatically computing gradients and managing backpropagation. TensorFlow users have implemented optimizations like batch normalization and gradient clipping to accelerate training and make it more robust. Users have also implemented advanced optimization algorithms like Momentum, Adagrad, Adam, etc, simply using primitive mathematical operations provided by TensorFlow.
- **Handling very large models**: TensorFlow implements sparse embedding layers using primitive operations like `Gather`, `Partition`, and `Stitch` to efficiently handle distributed and sparse data. These operations support automatic differentiation, enabling sparse updates and efficient gradient computation, which optimizes memory and computation in large-scale training.
- **Fault tolerance**: refers to the ability of a system to continue functioning correctly even when some of its components fail. TensorFlow jobs may encounter failures during the training process, so TensorFlow provides user-level checkpointing for fault tolerance using primitives like `Save` and `Restore`.
- **Synchronous replica coordination**: TensorFlow allows users to change how parameters are read and written when training a model. Though initially designed for async training, TensorFlow allows for experimenting with synchronous methods as well. The issue with the async approach is that some workers might use **stale information** to update the model, which can potentially lead to less accurate updates and slower convergence. In contrast, the synchronous approach uses queues to coordinate training. However, a common issue with this method is the bottleneck caused by **slow workers**, which can limit overall throughput.

1.4 Benchmarking TensorFlow

The paper also presents TensorFlow's performance metrics across various workloads:

- **Single machine benchmarks**: On a single GPU, TensorFlow achieves shorter step times compared to Caffe and performs within 6% of Torch.
- **Synchronous replica microbenchmark**: While increasing the number of workers boosts throughput (batches per second), it also introduces significant communication overhead and synchronization delays during fetching and updating.
- **Image Classification**: The authors investigate the scalability of training the Inception-v3 model using multiple replicas. Training throughput (images processed per second) improves upon increasing the number of workers; however, the step time increases. The synchronous training step takes longer as compared to asynchronous replica. To reduce median step time in a synchronous setup, users can add backup workers, a strategy that has proven to be highly effective.
- **Language Modeling**: Here, the training throughput, i.e., words processed per second, increases as the number of PS tasks increases, as it allows TensorFlow to exploit distributed model parallelism. They also experiment with sampled softmax, which boosts throughput by reducing the amount of computation required.

2 PyTorch: An Imperative Style, High-Performance Library

This paper primarily discusses the key design decisions made by the PyTorch team that enabled the library to stand out on a number of metrics as compared to other competitive frameworks. PyTorch offers an imperative and Pythonic programming interface, making debugging easier, supporting the "**code-as-a-model**" paradigm, and ensuring compatibility with hardware accelerators.

Unlike popular frameworks like TensorFlow and Caffe, which rely on **static dataflow graphs**, PyTorch employs a **dynamic dataflow graph**. While the static approach provides visibility into the entire model upfront and supports clean, powerful optimization, it sacrifices flexibility and user-friendliness. Once a static graph is defined, it cannot be modified without redefinition and recompilation, making such libraries less adaptable for certain workflows.

2.1 Trends in Scientific Computing

PyTorch builds upon several key trends in computing that are well-reflected in the library:

- **Array-based programming**: Libraries like NumPy, Torch, and Lush have made array-based programming highly productive in general-purpose languages like Python, offering a wide range of operations on multi-dimensional arrays.
- **Automatic Differentiation**: Earlier computing derivatives was considered to be a very time-consuming and laborious prospect; the development of packages like autograd made it easy for practitioners to experiment with different ML models.
- **Open-Source philosophy**: The open-source Python ecosystem has provided researchers with access to a vast collection of libraries for dataset manipulation, statistical analysis, visualization, and more. This expansive ecosystem, coupled with a vibrant developer community, has made Pythonic interfaces standard across most deep learning frameworks.
- **Harware Accelerators**: The availability of general-purpose GPUs has provided the necessary computational power to run deep learning models efficiently.

2.2 Design Principles

The success of PyTorch, which is evident from its widespread adoption, stems from several design choices that balance performance and ease of use.

- **Being Pythonic**: PyTorch integrates seamlessly with other Python libraries by maintaining consistency with the language's interfaces.
- **User first approach**: PyTorch prioritizes ease of experimentation with models, optimizers, data loaders, and more by separating the backend to handle the internal complexities of machine learning.
- **Provide practical performance**: Beyond simplicity and usability, PyTorch is committed to delivering high-performance capabilities.
- **Worse is better philosophy**: PyTorch embraces the principle that a simple, slightly incomplete solution is often better than a comprehensive but complex and hard-to-maintain design.

2.3 Usability Centric Design

PyTorch's usability-centric design aligns with the philosophy that "**everything is just a program**", allowing users to build models, optimizers, data loaders, and more with just a few lines of code. Its clear and flexible API fosters seamless experimentation with new training techniques. Moreover, because PyTorch uses eager execution, debugging is simplified—users can add print statements, generate plots with matplotlib, and check intermediate computations without waiting for lengthy compilation, as is required in static dataflow models like TensorFlow.

Additionally, PyTorch ensures easy and efficient **interoperability** by integrating seamlessly with the broader Python ecosystem, including popular libraries like NumPy. It allows for bidirectional data exchange with external libraries; for example, users can convert between NumPy arrays and PyTorch tensors using the `torch.from_numpy()` function and the `.numpy()` tensor method. These operations are efficient and take constant time due to **zero-copy**, meaning no data is duplicated during the conversion process.

Since gradients are central to training deep learning models, PyTorch leverages the Autograd system to automatically compute gradients for different user-defined models. During runtime, PyTorch dynamically constructs the computation graph and performs **reverse mode automatic differentiation**. In reverse mode, the graph is built during the forward pass, and gradients are computed during the backward pass.

2.4 Performance focused implementation

2.4.1 Efficient C++ Core

The Global Interpreter Lock (GIL) in Python prevents parallel execution of threads, but PyTorch overcomes this limitation with an efficient C++ backend. This backend optimizes performance by enabling operations to run on multiple threads, bypassing the constraints of the GIL. The **C++ core** includes libraries for implementing the tensor data structure, CPU and GPU operators, and essential parallel primitives, ensuring high-performance execution.

2.4.2 Separate control and async data flow

PyTorch maintains a clear separation between control and data flow. Control flow is handled on the CPU, while data operations are executed **asynchronously** on the GPU using CUDA streams. These streams queue operations for efficient execution. While the GPU processes tensor operations, Python code continues to run on the CPU, allowing both resources to be utilized concurrently. This approach maximizes performance and parallelism, ensuring optimal use of both the CPU and GPU.

2.4.3 Memory management and Custom caching tensor allocator

PyTorch employs an efficient memory management strategy to handle GPU memory, which is crucial for high-performance deep learning. When PyTorch begins execution, it needs to allocate memory on the GPU for tensors and operations. This is done using functions like `cudaMalloc` (to allocate memory) and `cudaFree` (to release memory), which are quite expensive and block the CPU thread. Because there is a delay in the despatch of new GPU operations from the CPU, the GPU utilization decreases. This effect disappears in the subsequent iterations because PyTorch uses a **caching memory allocator** to reuse previously allocated regions. This caching reduces the latency of future operations and allows faster execution.

2.4.4 Multiprocessing

To leverage the benefits of parallel and distributed computing, tensors need to be shared across processes. However, due to the GIL, Python does not allow threads to execute in parallel. To address this, the Python community developed the multiprocessing library, which enables spawning multiple processes and sharing data between them through serialization and deserialization. Serialization converts Python objects into byte streams for inter-process communication. However, this approach can be inefficient for large arrays, as the serialization and deserialization processes are slow and memory-intensive.

Hence, PyTorch uses its own `torch.multiprocessing` module, which employs a shared memory accessible by multiple processes. This design eliminates the need for serialization and deserialization, allowing tensors to be moved efficiently between processes and significantly improving performance.

2.4.5 Reference Counting

Traditional garbage collection can delay memory deallocation, leading to short-term spikes in memory usage—an issue that is critical in resource-constrained computing environments. To address this, PyTorch employs reference counting, a method that tracks the number of references to each tensor. Once the reference count drops to zero, the underlying memory is freed immediately, ensuring efficient memory management.

Finally, the authors provide a comprehensive comparative analysis of PyTorch's performance against other popular frameworks like TensorFlow, MXNet, and CNTK. The performance benchmarking is based on training speed, measured in terms of **throughput** (i.e., the number of images processed by the model per second) for various architectures, including VGG-19, AlexNet, and ResNet-50. Across all benchmarks, PyTorch's performance is consistently within 17% of the fastest framework.