Tony Nguyen

Dr. Shawn Bowers

CPSC 324 01

26 April 2024

Reading 5

Part 1: Abadi, M. et al. (2016) TensorFlow: A System for Large-Scale Machine Learning. Proc. Of OSDI, pp. 265–283.

Reading Notes

TensorFlow is an ML system that operates on a large scale and in heterogeneous environments.

1. Introduction
    - TF developed for experimenting with new models and training them on large datasets and moved them into production.
    - TF supports both large-scale training and interference.
        - Uses hundreds of powerful servers for fast training
        - Runs them on various platforms, ranging from large distributed clusters to locally
        - At the same time, flexible enough.
    - TF uses a unified dataflow graph to represent both an algorithm's computation and its operating state.
    - TF allows vertices to represent computations that own or update mutable states.
    - Edgess carries tensors (multi-dimensional arrays) between nodes, and TF inserts appropriate communication between distributed sub-computations.

- Unifying computation and state management in a single programming model. TF allows programmers to experiment with different parallelization schemes that offload computations onto the servers that hold the shared state to reduce the amount of work.
- Good results were achieved with synchronous replication, which contradicts the belief that asynchronous replication is required for scalable learning.

2. Background & motivation
    - DistBelief
        - The predecessor of TF – distributed system for training neural network
        - Uses parameter server architecture
            - A job comprises two disjoint sets of processes:
                - Stateless worker processes that perform the bulk of the computation when training a model
                - Stateful parameter server processes that maintain the current version of the model parameters
        - Neural network:
            - Directed acyclic graph of layers that terminates with a loss function.
            - A layer is a composition of math operators.
            - Loss function: A scalar function that quantifies the difference between the predicted value and the ground truth.
        - Parameters: the weight matrix and bias vector, which a learning algorithm will update to minimize the value of the loss function
        - It uses the DAG structure and knowledge of the layer's semantics to compute gradients (to reduce loss) for each model parameter via backpropagation.

    - Cons of DistBelief
        - Uses C++ to create layers
        - Refining the training algorithm requires modifying the input parameters
    - Added support for GPU acceleration
    - After training a model on a cluster, then push the model into production
    - TF provides a single programming model and runtime system for all of the environments.
- Design principles
    - TF is more flexible than DistBelief while retaining its ability to satisfy the demand of ML workload.
    - TF provides a simple dataflow-based programming abstraction that allows users to deploy applications on distributed clusters.
    - High-level scripting interface wraps the construction of dataflow graphs and enables users to experiment with different model architectures and optimization algorithms.
    - Core principles:
        - Dataflow graphs of primitive operators
            - DistBelief uses a dataflow representation for their models but comprises relatively few complex “layers.”
            - TF represents individual math operators as nodes in the dataflow graph

        - Makes it easier for users to compose novel layers using a high-level scripting interface.
        - Many optimization algorithms require each layer to have defined gradients and build layers out of simple operators.

- Deferred execution
    - A typical TF has two distinct phases
        - Defines the program as a dataflow graph
        - Executes an optimized version of the program
    - Deferring the execution until the entire program is available
        - More efficient
        - Pushes more complex features into the dataflow graph to ensure optimization
- Common abstraction for heterogenous accelerators
    - Uses specialized accelerators on top of commodity machines
        - Minimum requirements
            - Issuing kernel for execution
            - Allocating memory for inputs and outputs
            - Transferring buffers to and from host memory
        - Each operator can have specialized implementations for different devices
    - TF uses tensors of primitive values to exchange. All tensors in TF are dense.

        - Ensure the lowest levels of the system have simple implementations for memory allocation, reducing overhead.
    - Consequences (pos)
        - No parameter server
        - This means we deploy TF as a set of tasks on a cluster
        - A subset of tasks assumes that a parameter server plays in other systems (PS Tasks); the others are worker tasks.
        - PS tasks are great as it is more flexible than conventional ones, which is TF's greatest advantage.

- Related work
    - Single-machine frameworks
        - Discussed Caffe, Theano, and Torch
        - Torch offers a powerful imperative programming model for scientific calculation and ML. Also allows fine-grained control.
            - However, it lacks the ability to use a dataflow graph as a portable representation across small-scale experimentation, etc.
    - Batch dataflow systems
        - Limitation: it requires the input data to be immutable and all of the subcomputation to be deterministic to be able to rerun in case of machine failure
        - Great for fault tolerance but really expensive.

- Parameter servers
  - A parameter server architecture uses a set of servers to manage a shared state that is updated by a set of parallel workers.
    - Consistent model, fault tolerance, and elastic rescaling
    - A parameter server specialized for use with GPUs can achieve speedups on small clusters
  - For TF, we are able to achieve a high-level programming model that allows users to customize the code that runs in all parts of the system
    - Lower cost of experimentation

3. TensorFlow execution model

- General
  - TF uses a single dataflow graph to represent all computation and state
    - The dataflow graph expresses the communication between subcomputations explicitly, thus easier execution of independent computations
  - TF differs from batch dataflow systems in two ways
    - The model supports multiple concurrent executions on overlapping subgraphs of the overall graph
    - Individual vertices may have a mutable state that can be shared between different executions of the graph
  - General themes: A mutable state is crucial
    - Enables TF to mimic the functionality of a parameter server, pluses additional flexibility.

- Dataflow graph elements
    - Each vertex represents a unit of local computation
    - Operations: computation at vertices
    - Tensors: the values that flow along the edges
    - Tensors:
        - Model all data as tensors (n-dimensional arrays) with the elements having one of a small number of primitive types
    - Operations
        - Takes m>=0 tensors as input and produces n>=0 tensors as output
        - Has a named type and may have zero or more compile-time attributes
        - Can be polymorphic and variadic at compile-time
    - Stateful operations: variable
        - An operation can contain a mutable state that is read and/or written each time it executes
        - A Variable operation owns a mutable buffer that may be used to store the shared params of a model it trained.
            - No input
            - Produce a reference handler
    - Stateful operations: queues
        - It has many queue implementations which support more advanced forms of computation.
- Partial and concurrent execution
    - The API allows the client to specify the subgraph that it wants to execute first.

    - Choose 0 or more edges to feed the input tensor into the dataflow and one or more edges to fetch output tensors from the DF.
    - Each step is the invocation of the API. TF supports many concurrent steps.
    - Stateful operations allow steps to share data and synchronize the necessary.
    - Partial and concurrent execution is what makes TF flexible. Adding a mutable state and coordination via queue further expands the flexibility.
    - Concurrent execution is async with respect to one another
- Distributed execution
    - DF simplifies distributed execution because it makes the communication between the subcomponents explicit.
    - Each op is on a particular device (CPU/GPU) in a particular task.
        - The device is responsible for executing the kernel.
        - The placement algo takes care of the device selection.
    - TF allows great flexibility in how ops in the DF graph are mapped to devices.
    - It also has low latency. Graphs are cached on their devices, which makes accessing them faster.
- Dynamic control flow
    - TF supports ML algos that contain conditional and iterative control flow, such as RNN.
    - Borrows a “Switch” and “Merge” primitives from class DF architecture
    - Supports automatic differentiation of control flow constructs.
    - Adds subgraphs for computing gradients to the DF graph

4. Extensibility case studies

- Differentiation and optimization
    - TF includes a library that differentiates a symbolic expression for a loss function and produces a new symbolic expression representing the gradients.
    - See paper for the differentiation algo
    - Optimization algo computes new values for the params in each training step
- Training huge models
    - To train on high-dimensional data, it is common to use a distributed representation, which embeds a training example as a pattern of activity across several neurons, which can be learned by backpropagation.
    - See paper for algo
- Fault tolerance
    - Training takes time, so failure may happen
    - However, it is unlikely that it often happens, so it doesn't need an RDD like Spark.
    - Implements checkpointing to address this
- Synchronous replica coordination
    - Many systems train deep neural networks using asynchronous parameter updates.
    - Synchronous training may be faster than asynchronous; however, with the current model version, it is slower.
    - Implements backup workers (similar to MR) to address stragglers.

Answers to Questions

1. Question 1

- DistBelief scripting interface is inadequate for advanced users who want more control over the optimization and new layer architecture. At the same time, implementing new layers and refining algorithms required modifying the parameters, which involved using C++. This makes it less attractive for machine learning engineers as C++ is not a language they usually work on.
- DistBelief also comes with a fixed execution pattern, which is not suitable for more advanced models like RNN, reinforcement learning, and more.

2. Question 2
    - TensorFlow (TF) and DistBelief (DB) differ most in the unified dataflow graph, which handles both computation and state management. By having this integration, TF supports both flexible and state management.
    - The integration allows TF to support flexible and varied parallelization schemes. It also allows computation to be offloaded onto servers to minimize network traffic. This flexibility allows users to play around with different optimization and training algorithms directly within the framework without the need to modify the system.
3. Question 3
    - MapReduce and Spark require the input data to be immutable and all subcomputations to be deterministic. This allows fault tolerance but makes it expensive and inefficient to modify the model parameters since there are many overheads.
    - TF, on the other hand, addresses the shortcomings of MapReduce and Spark by allowing updates to shared model parameters to be efficient and quick. TF architecture supports the execution of complex, mutable dataflow graphs and

leverages stateful computations, which is critical for training neural networks effectively.

4. Question 4
    - TF has a distributed execution model that simplifies deploying the same program across various configurations, from clustering to individual mobile devices. Each operation is assigned to a specific device, and TF optimizes the execution by considering device capabilities and operation requirements.
    - TF also supports dynamic computational structures like loops and conditional branches within its dataflow graphs, enabling it to handle sophisticated ML models. The dynamic control is integrated into the dataflow graph, allowing TF to maintain efficiency and leverage distributed resources with complex, dynamic changing computation.
5. Question 5
    - The distributed master is responsible for coordinating the distributed execution of TF programs. It translates user requests into execution plans, optimizes the dataflow graph for execution, and manages the distribution of computation across available resources.
    - The dataflow executor is in each participating node. It manages the local execution of each task, schedules operations, handles data transfer between operations and executes kernels on computational devices.
    - Fault tolerance approach: implemented primarily through user-level checkpointing. It includes operations in the graph for saving and restoring states. This helps TF to handle partial failures by periodically saving the state of the model to checkpoint

files. This also allows asynchronous recovery without having each operation be recoverable.

Additional Questions and Observations

1. TensorFlow is definitely an interesting system. I have heard about it many times but have never had a chance to actually implement it. I am interested in the flexibility of model experimentation without the need to change the underlying code. I also find it interesting in how it is the tool to allow deep neural network to actually be implemented.

Part 2: Li, S., et al. (2020). PyTorch Distributed: Experiences on Accelerating Data Parallel Training, Proc. of VLDB, vol. 13, no. 12, pp. 3005–3018.

Reading Notes

1. Introduction
    - PT is a scientific computing package used in deep learning research and application.
    - It natively provides several techniques to accelerate distributed data parallel, including bucketing gradients, overlapping computation with communication, and skipping gradient synchronization.
    - Training a DNN model usually needs three steps
        - The forward pass to compute loss
        - The backward pass to compute gradients
        - The optimizer step to update params
    - Data parallelism

    - Applications can create multiple replicas of a model, with each model working on a portion of training data and performing the forward and backward passes independently.
    - After that, the model replicas synchronize either the gradients or updated params.
    - Native distributed data parallel helps application developers focus on optimizing their model.
- Three challenges
    - Math equivalence: speed up training on a large dataset
        - All replicas start from the same initial values
        - Synchronize gradients to keep params consistent across training iterations
    - Non-intrusive and interceptive API: to reduce hurdles in code transition
    - High performance: to efficiently convert more resources into higher training throughput
-

2. Background
    - PyTorch intro
        - Organizes values into tensors (generic n-d arrays) with lots of data manipulating ops
        - Modules: a transform from input to output. Behaviors specified in the forward pass
        - Module Composition: contains tensors as params

    - Training Iteration Steps: Outlines typical training iteration containing a forward pass (generating losses), a backward pass (computing gradients), and an optimizer step (updating parameters using gradients).
    - Autograd Graph: PyTorch constructs an autograd graph to record actions, which is then used for backpropagation in the backward pass to generate gradients during the forward pass

- Data Parallelism
    - PT offers tools to offer distributed training
    - Enables distributed training by communicating gradients before the optimizer step to ensure consistency
    - Param averaging: scale out model training
        - Some caveats:
            - Can produce vastly different results compared to local training
            - The resource to orchestrate computation and communication will always have something idling, meaning a loss of performance.
- AllReduce
    - A primitive communication API used by DistributedDataParallel to compute summation across all processes
    - Expect each participating process to provide an equally sized tensor and collectively apply an arithmetic op to input tensors from all processes.
    - All AllReduce ops significantly impact distributed training speed; communication libraries have implemented more sophisticated algos

- AllReduce is considered a synchronized ops

3. System Design

- API
  - Two design goals
    - Non-intrusive: reuse the local training script with minimal modification when scaling out
    - Interceptive: expose as many optimization opportunities as possible to the internal implementation
    - By not requiring major alterations to existing training scripts, the API promotes broader adoption and flexibility in integrating with various application setups.
- Gradient Reduction
  - Native solution
    - DDP guarantees correctness by letting all training processes
      - Start from the same model state
      - Consume the same gradients in every iteration
        - Broadcasting model states from one process to all others at the construction time of DDP
        - PT autograd engine accepts custom backward hook
    - Two performance concerns
      - Collective communication performs poor on small tensors
      - Separating gradient computation and synchronization forfeits the opportunity to overlap computation with communication.

- Gradient bucketing
  - Address poor performance from collective communication on small tensors
  - Optimizes network communication by bucketing gradients to reduce the number of communications required
- Overlap computation with communication
  - The AllReduce op on gradients can start before the local backward pass
  - AllReduce During Backward Pass:
    - Start before the local backward pass finishes by using gradient bucketing.
    - Autograd hooks are registered for each gradient accumulator to trigger AllReduce when gradients in a bucket are ready.
  - Consistency Challenges:
    - Consistent reduction order across all processes is crucial to avoid mismatches and potential crashes.
    - PyTorch addresses this by using the reverse order of model.parameters() to approximate the gradient computation order in the backward pass.
  - Handling Variability in Training:
    - Training iterations might involve different sub-graphs of the model, leading to skipped gradients in some iterations.

- DDP avoids hangs by marking participating tensors as ready at the end of the forward pass, ensuring only active gradients are synchronized.

- Collective Communication
    - Distributed data-parallel training uses a special communication pattern where every participant provides an equally sized tensor and collects the global sum across all participants.
    - ProcessGroup expects multiple processes to work collectively as a group
        - Using it can achieve higher bandwidth utilization.

4. Implementation

- Python front end
    - DDP is implemented in distributed.py in Python, handling user-facing components like the constructor, forward function, and the no-sync context manager.
    - The constructor provides options to specify the process group, control the AllReduce bucket size, and detect unused parameters.
    - DDP handles models spread across multiple devices, ensuring each device's computations are optimized for performance.
    - DDP synchronizes necessary model buffers (e.g., BatchNorm buffers) from the process with rank 0 to others before computations.
- Core Gradient Reduction

- Core functionalities include parameter-to-bucket mapping, autograd hook installation, bucket AllReduce initiation, and detection of globally unused parameters.
- Parameters are mapped to buckets for efficient AllReduce operations, ensuring parameters in the same bucket are on the same device.
- Hooks are installed on gradient accumulators to trigger AllReduce when gradients are ready, managing the readiness of buckets for reduction.
- Bucket AllReduce Strategy: Balances the trade-off between communication overhead
- Uses a bitmap to track and manage unused parameters across different DDP processes, ensuring gradients are appropriately managed during the forward and backward passes.

Answers to Questions

1. Question 1

- Data parallelism
  - Involves splitting the training data across multiple processors
  - Each processor works on its data subset but shares the same model. It also performs forward and backward passes, in addition to the gradients or model parameters being synchronized across all processors
  - This helps to ensure consistency and improve learning efficiency
- The two approaches

    - Gradient synchronization involves synchronizing the gradients calculated by each processor before updating the model. This guarantees consistency by having all processors update their model in the same way. This is the chosen step due to the mathematical guarantees it offers, ensuring the distributed training is equivalent to training on a single processor.
    - Parameter averaging averages the model parameter after each step instead of synchronizing. Although it is simpler, it may lead to divergences in model training due to different local optimizations.

2. Question 2
    - AllReuce is a collective communication operation where all processors contribute data that are then aggregated and returned to the processor.
    - AllReduced, in the context of DDP, is used to sum up gradients from different processors so that each processor eventually has the correct gradient to update the model. This is important as it is necessary to make sure all replicas are consistent between processors.
3. Question 3
    - Gradient bucketing: this involves grouping gradients into buckets to optimize the AllReduce operations. It reduces the overhead and frequency of communications by sending a larger data chunk at once.
    - Overlapping computation and communication: Allows the backward pass to overlap with the gradient synchronization, essentially the computation of gradients and communication, respectively. We achieve this by starting the AllReduce as soon as possible, even before other parts are still running.

- Gradient accumulating: In case of a communication bottleneck, DDP accumulates gradients over several iterations before performing synchronization. This reduces the frequency of synchronization, which then reduces communication overhead. However, this affects the model's convergence behavior.

Additional Questions and Observations

1. One thing that comes from a previous experience with PyTorch, which I find interesting, is the ability to support different device configurations. However, I don’t exactly know how it would play out in reality. I used to run PyTorch on my older machine. However, I could not get it up and running since the processor was dated. I guess it was not really PyTorch’s problem, but it was probably my machine's problem.