TensorFlow: A System for Large-Scale Machine Learning

Abadi et al. OSDI 2016

Course updates

- *Proposal*: change piazza response due time from 24 hours before class to 18 hours before class
    - e.g., 2PM instead of 8AM the day before class (Vancouver time)
    - *Proposal passed unanimously in class with 10 people present.*
- Update + email to schedule chat with me due next week on Friday (feel free to do this earlier :-)

Machine Learning

- *Application-focused paper*
- Systems pov: what are ML requirements?
  - Flexible to *customize* for an ML engineer (plug in different strategies for optimization, (a)synchrony, model types, data types, scale, device types, parallelism)
  - Huge data — high throughput is critical
  - Huge number of parameters that have to be updated frequently (amount of state to maintain)
  - *ML* training is less efficient on a CPU (GPUs and TPUs)
  - ML doesn't require strong consistency (substantial flexibility)

TF versus Spark

- Breakout chat:
  - How+why is TF similar to Spark, and how is it different?

TF versus Spark

- Different designs, but with similarities!
  - Failures aren't more/less likely in a TF cluster versus a Spark cluster.
  - Same data flow abstraction — ML as a graph versus analytics as a process
- Fault tolerance (sec 4.3): both have a "checkpoint" mechanism. Spark achieves this primarily with RDDs and lineage.
- State mutability: TF chooses mutability, Spark uses immutable RDDs.
- Can you use TF for Spark? Yes.. if you frame everything as a tensor :-) TF *dynamic* control flow in a data graph: can reproduce anything that Spark supports
  - Dynamic control flow => materialization necessary, immutability isn't as helpful
- Granularity of operations: TF fine-grained, Spark coarse-grained. RDD high overhead when fine-grained.
- ML dataset might be large (input, and parameters) — wouldn't fit in memory! Need to use sharding (TF automates this), Spark uses partitions to shard RDDs.
- Different attitudes towards failure: Spark as general-purpose compute cannot lose results or be inconsistent. TF by contrast can shard/lose compute as long as it works for ML.
  - Slight randomization is good for ML (e.g., compute on batches of random data)

TF design

- *Dataflow*: nodes are operations, data flows on vertices from node to node, which transform it.
- *Device specialization:* an implementation of an operator per device. e.g., matrix multiply for CPU (x86/ARM..)/GPU (Nvidia/…)/TPU (v1/v2)
  - *Device abstraction:* allocating memory for input/output, issuing kernel for exec, transfer data to/from memory.
  - Compiler selecting the implementation to use (without developer needing to make a choice)
  - Matching problem: mapping operators to devices — what's a good heuristic?
    - Efficiency of operators on a specific device, data transfer to/from device
- Concurrent executions on overlapping subgraphs (ML specific) — to support looping over a graph (classical data-flow operators); good for RNNs
  - Resolve writes shared state (consistency issues)
  - Resolve reads from shared state (sharding)
  - Dynamic runtime scheduling of *operators* on *tasks*

TF eval

- Eval criteria:
  - Throughput (data/time): training time
  - Training step time: latency per iteration
  - Efficiency (single machine); Table 1
  - Straggler mitigation (use backup workers to make up for slow nodes)
  - Sparcity : sparse versus dense vectors
- Baselines:
  - ML frameworks: MXNet (centralized parameter server), Caffee, Neon, Torch
- What's missing?
  - Fault tolerance not covered
  - No comparison to Spark?! But Spark wasn't designed for neural nets
  - Only Fig 8(a) for distributed comparison against another framework
  - Missing design evaluation — matching eval results to specific design choices

Discussion points

- Data-flow to the rescue? Especially good match for big data and commodity resources (requiring a smart compiler)?

- App-specific *compute* specialization. ML clearly important. Other app optimizations? BitCoin.. HFT (networking).. Industrial applications.. Scientific computing (supercomputers!)

- Consistency has a cost; is there a more rigorous way to relax consistency? (TF is not very rigorous about relaxing consistency).

Next: CAP theorem

- Done with distribute compute (Spark + TF)
- Back to data consistency, this time at scale
- Start with *CAP theorem*
- Then onward to weak consistency (CRDT, OR)