Scaling Distributed Machine Learning with a Raft-based parameter server framework

CPSC 490 Proposal - Michelle Lim

Introduction

As datasets increase in size, there is no longer enough space on the RAM of a computer for data and hence machine learning cannot be conducted on just one computer. Distributed machine learning—running machine learning tasks on separate devices—is now the norm. One of the requirements in scaling distributed learning is fault tolerance. Machines can be unreliable at scale, and we often have to rerun the whole computation on the machine during crashes. However, this reduces throughput and decreases scalability. For instance, in Distributed TensorFlow, when a chief worker crashes, computation on all workers is paused until the chief worker comes back on. This could be inefficient if the chief worker is prone to failure. In Apache Spark, when an RDD crashes, we have to recompute the RDD from the start. We need to strive for reliability and scalability. In this project, I plan on building a Raft-based configuration framework manager that will both pick a chief worker and automatically deal with hot failover. Raft is well-suited for fault tolerance because it is a consensus algorithm, which allows a group of machines to work as a coherent group despite multiple machine failures. The goal is to continue computation even when the chief worker fails, so as to make training run faster and also to remove reliance on the engineer to restart the chief worker.

Fault tolerance of Existing Distributed Learning Systems

Distributed TensorFlow

TensorFlow allows users to compute part of their graph on different servers. Its architecture takes on the Parameter Server paradigm (picture from Li et al, 2014), whereby parameter servers contain partitions...
of globally shared parameters. Each worker node is stateless and only contains some training data temporarily only to compute a statistic, such as a gradient. Workers communicate updated parameters to the server nodes.

Fault tolerance in TensorFlow is achieved through checkpoint tracking and the use of a chief worker. One of the worker nodes is assigned as the chief worker. The chief worker performs two main functions. 1) It saves checkpoints: it periodically saves the entire state of `tf.Session` to a database, such as HDFS or Google Cloud. 2) The chief worker is also in charge of coordinating the work of all worker nodes to ensure consistency. Upon a PS failure, the chief worker is in charge of interrupting all workers’ training until the PS comes back up.

However, upon the chief worker’s own failure, the system will not be able to detect PS failures. As a result worker nodes go on with their jobs even though the PS nodes they have to communicate with is down. One crude solution to this failure of the chief worker is to interrupt all other worker nodes until the chief worker comes back on. This way, no worker node will run when a PS fails. This solution is inefficient because while waiting for the chief worker to come back on, the computation is stalled and no work is done on the worker nodes.

Distributed TensorFlow also runs on the ring-all-reduce architecture (on the right from O’Reilly). Unlike in the parameter server framework where parameter servers and worker nodes are separate, in the ring all-reduce architecture, each node has the parameters and the data on the worker nodes. In each round, each worker calculates its gradients and sends its gradients to its successor neighbor on the ring, and receives gradients from its predecessor neighbor on the ring. Once a worker receives N-1 messages, it has all the gradiences needed for the updated model. This is bandwidth optimal. The trouble comes when any of the GPUs were to crash.
Apache Spark

Spark’s central data structure is the RDD—its immutable fault tolerant data structure. When Spark runs code, it first records the dependency of each operation among the RDDs in what is called the RDD lineage graph. Based on the graph, Spark’s scheduler then constructs a dependency DAG and broadcasts the DAG and task to workers who compute their assigned portion of the RDD.

In case of a failure, Spark has to recompute the RDD based on the lineage graph. This might seem trivial but the lineage graph for Spark is unusually long for machine learning. This is because in iterative machine learning training, repeatedly updated model parameters are stored as an RDD. Hence, the iterations are repeated thousands of times. We need to find a way that will checkpoint the intermediate RDDs in the process to avoid having to recompute it.

Deliverables

I aim to build a Raft-based configuration framework manager that will both pick a chief worker and automatically deal with hot failover.

First, I will design the distributed algorithm and architecture that will elect the chief worker and automatically detect changes in cluster membership. The latter will be achieved by keeping track of membership changes in the replicated logs.

Second, I will write the library that will integrate with either TensorFlow or Spark, that will allow for the new framework to be implemented.

Finally, I will evaluate the library and algorithm by testing it using 2 machine learning examples, namely large-scale image classification, and perhaps topic modelling across a large-scale network. The evaluation metrics will be throughput while maintaining fault tolerance.
References

Distributed TensorFlow talk: https://www.youtube.com/watch?v=la_M6bCV91M&t=1355s
Distributed TensorFlow OReilly link: https://www.oreilly.com/ideas/distributed-tensorflow
Lineage-based issues in Apache Spark: