10-708: Probabilistic Graphical Models, Spring 2017

Lecture 22 : Distributed Systems for ML

Lecturer: Qirong Ho *Scribes: Zihang Dai, Fan Yang*

1 Introduction

Big data has been very popular in recent years. It appears in many application domains, such as computational biology, online advertisement, and recommendation systems, etc. Big data generates a lot values and capabilities for machine learning algorithms, but also creates many challenges. This lecture will cover the challenges for machine learning systems in this big data era, and describe the big picture solutions with distributed algorithms.

Figure 1: The amount of data hosted in popular websites.

2 Challenges

As the amount of data grows, efficiently and effectively mining information from data becomes very challenging. Fundamental issues arise when figuring out how to make the computations, storage, transmission, and imputation of data happen in a low-latency way. There are also theoretical challenges, such as how to design algorithms that scale but do not lose the theoretical convergence guarantees.

2.1 Massive Data Scale

The first challenge is straightforward: there are more and more data. For example, Facebook now has over 1.6 billion active users. Simple computations on the Facebook network will need to deal with a graph of

billions of nodes, and the edges among them. Because of the data size, simple manipulation of the data becomes challenging. Storing and transmitting can be the biggest bottlenecks of the whole model pipeline.

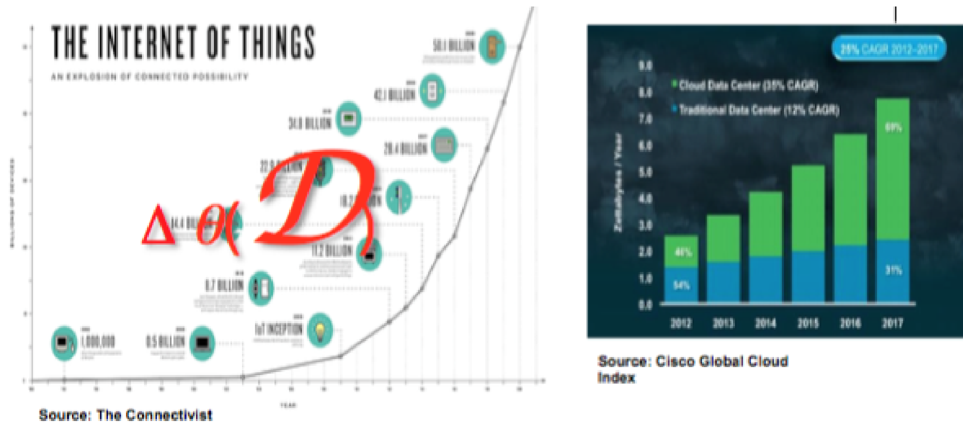

Figure 2: Internet of things dataset contains data from over 50 GB devices

2.2 Gigantic Model Size

The second challenge comes closely with the first one. As the data size grows, machine learning algorithms tend to have larger models to fully capture the information in the data. Therefore, the model size also grows. Examples as convolutional neural networks can easily have millions of parameters. Efficient model training, as well as inference, become very challenging.

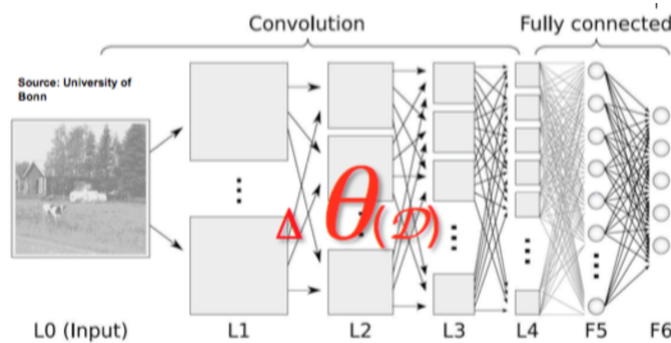

Figure 3: Architecture of a convolutional neural network for image classification

2.3 Inadequate support for newer methods

Third challenge is that currently we only have scalable algorithms for very simple machine learning methods, such as k-means, logistic regression, Naive Bayes, etc. While some recently developed software packages, such as YahooLDA and Vowpal Wabbit, provide newer models, many packages still only implement classic

methods that are not designed for big data. There is inadequate scalability support for newer methods, and it is challenging to provide a general distributed system that supports all machine learning algorithms.

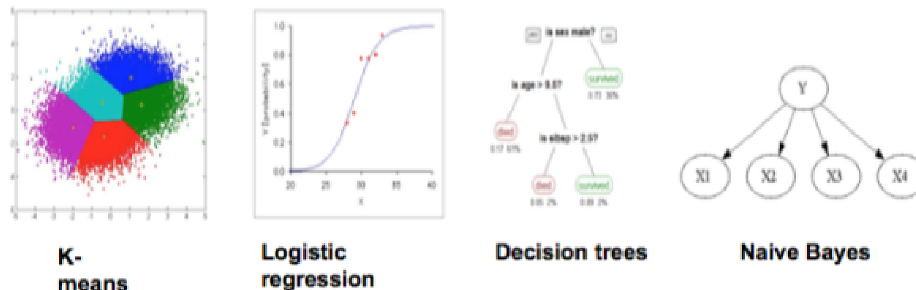

Figure 4: Machine learning algorithms that are easy to scale.

3 ML methods

We will define some general properties of machine learning algorithms. These properties will be useful, since they will serve as the guidelines for designing general distributed systems to scale machine learning algorithms.

An ML program can be written in general as

$$\arg\max_{\theta} = \mathcal{L}(\{x_i, y_i\}_{i=1}^{N}; \theta) + \Omega(\theta)$$

where $\mathcal{L}$ and Ω represent the model, $\{x_i, y_i\}_{i=1}^{N}$ are the data, and θ is the set of all parameters. Usually machine learning algorithms are solved in an iterative fashion. The parameters are updated based on data until convergence.

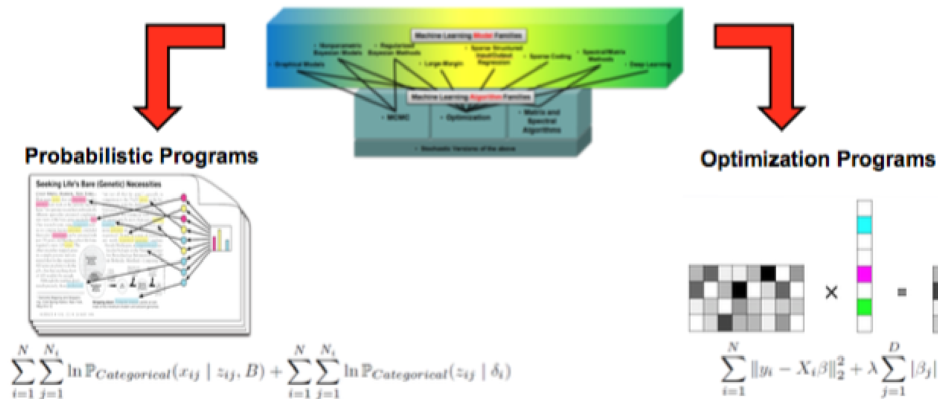

Figure 5: An overview of the components in machine learning programs

There are two basic approaches for parallelizing such iterative-convergent machine learning methods.

- Data-Parallelism: The data is partitioned and distributed onto the different workers. Each worker typically updates all parameters based on its share of the data
- Model-Parallelism: Each worker has access to the entire dataset but only updates a subset of the parameters at a time

Of course, both principle approaches are not exclusive and thus can also be combined to yield better performance in practice.

4 Software to Implement Distributed ML

4.1 Spark

Spark defines a set of general-purpose APIs for Big Data processing, where MLlib is a specific implementation and ready-to-run ML library for Spark APIs. The key feature of Spark is its Resilient Distributed Datasets (RDDs), which can be represented by the nodes in a data processing graph. Then, the edges represent the correspond transformation the RDDs will go through. In spirit, the RDD-based programming model is quite similar to Hadoop Maprecduce, and supports functional-style manipulations on RDDs via "transformations", "actions". The difference between Spark and Hadoop lies in that Spark can cache the intermediate computation results in memory rather than dumping all results to the disk which involves much slower disk I/O. Since Spark can also also dumping result to the dist, it is strictly a superset of Hadoop Mapreduce, leading to much faster speed especially after the first iteration.

4.2 GraphLab

GraphLab relies on a Gather-Apply-Scatter (GAS) programming model, where the ML algorithms are written as vertex programs, and data or model parameters can stored either on nodes or edges. Specifically, the GAS vertex programs can be defined by the three generic APIs

- Gather(): Accumulate data, params from my neighbors + edges
- Apply(): Transform output of Gather(), write to myself
- Scatter(): Transform output of Gather(), Apply(), write to my edges

There are three key benefits of GraphLab

- Supports asynchronous execution, which allows the program to be fast and avoids straggler problems
- Edge-cut partitioning, which ensures the program can run in parallel for large, power-law graphs
- Graph-correctness, which provides the flexibility to specify more fine-grained constraints for ML algorithms

4.3 Google TensorFlow

TensorFlow is a dataflow-style system, which supports auto-differentiation and distributed computation. The dataflow is conceptually similar to Spark RDDs, while more tailored towards tensor/matrix computation

often found in neural networks. Also, asynchronous execution is allowed in TensorFlow as long as it does not violate the data dependency. As the its distributed features, while performance is OK for smaller models, large models do not benefit that much from going distributed.

4.4 Parallel ML System (Formerly Petuum)

There are two key components of PMLS as follows.

1. A key-value store (Parameter Server): the key-value parameter server can be seen as a type of distributed shared memory (DSM), allowing the model parameters to be shared across workers globally. For programing, one only needs to replace the local parameter calls with parameter server queries. Thus, it enables data-parallel ML algorithms in an efficient way.
2. A scheduler: the scheduler is responsible for analyzing the ML model structure for best execution order, and thus supports model-parallel ML algorithms. For programming, it defines some high level APIs such as schedule(), push(), pull(), etc.

5 Systems and Architectures for Distributed Machine Learning

5.1 ML Computation vs. Classical Computing Programs

In classical computing, programs are deterministic and operation-centric: a strict set of instructions with predetermined length is followed until completion. A problem with this approach is that strict operational correctness is required; one error in the middle of a sorting algorithm and the end result is useless. Atmoic correctness is required, and there is an entire field of research on fault-tolerance to address this issue. On the other hand, ML algorithms are probabilistic, optimization-centric, and iterative-convergent. More specifically, ML computation usually have the following critical properties

- Error tolerance: often robust against limited errors in intermediate calculations. In this case, a few errors here and there dont really matter, as the program will continue seeking an optimum regardless of a few missteps.
- Dynamic structural dependency: changing correlations between model parameters critical to efficient parallelization. As a result, as correlations between parameters change, the parallel structure of the program must change as well to stay efficient, which can be highly challenging in practice.
- Non-uniform convergence: parameters can converge in very different number of steps. Often, most of the parameters converge very quickly, while the rest take thousands more iterations to fit. Thus, for the system to be more efficient, it makes sense to allocate the computational resource according to the convergence speed.

5.2 Challenges and Solutions in Data Parallelism

There are two major challenges in data parallelism:

1. Need for partial synchronicity: Synchronization (and therefore communication costs) should only happen when necessary. Workers waiting for synchronization should be avoided when possible.

2. Need for straggler tolerance: Slow worker need to be allowed to catch up.

One attempt at overcoming these challenges is the Stale Synchronous Parallel (SSP) model where each thread runs at its own pace. Meanwhile, fastest/slowest threads are not allowed to drift more than S iterations apart, which leads to sound theoretical guarantee. To reduce communication overhead, each worker has its own cached version of parameters (which need not be up-to date). As a result, it not only achieves asynchronous-like speed, but also retains the correctness guarantees like a bulk synchronous parallel (BSP).

This approach has been further improved in the Eager SSP protocol (ESSP). The basic idea to take advantage of the spare bandwidth to push fresh parameters sooner proactively. Hence, ESSP has fewer stale reads, leading to lower staleness variance.

6 Theory of Real Distributed ML Systems

There are different type of convergence guarantees for analyzing ML algorithms

- Regret/Expectation bounds on parameters, which measures the convergence progress per iteration
- Probabilistic bounds on parameters, which is intuitively similar to the regret bound but usually stronger
- Variance bounds on parameters, which measures the stability near optimum or convergence

Here we are mostly interested in the convergence properties of the bounded asynchronous parallel (BAP) bridging model, including both SSP and ESSP. Define the optimization objective to be

$$\min f(x) = \frac{1}{T}\sum_{i=1}^{T} f_t(x)$$

where f is convex w.r.t. x. Also, technically, we further require that the function is L-Lipschitz, and feasible space is compact, i.e. problem diameter bounded by F^2. Also, we denote the staleness by s, and the number of threads across all machines by P. Then, it can be proved that with step size $\eta_t = \frac{\sigma}{\sqrt{t}}$ with $\sigma = \frac{F}{L\sqrt{2(s+1)P}}$, (E)SSP achieves the following the **regret bound**

$$R(x) = \left[\frac{1}{T} f_t(\tilde{x}_t)\right] - f(x^*) \leq 4FL\sqrt{\frac{2(s+1)P}{T}} \tag{1}$$

Similarly, we can establish the probability bound of (E)SSP. Let the real staleness observed by the system be γ_t, and its mean and variance be $\mu_\gamma, \sigma_\gamma^2$, we probability bound has the form

$$P\left[\underbrace{\frac{R(x)}{T}}_{\text{Gap}} - \frac{1}{\sqrt{T}}\left(\eta L^2 + \frac{F^2}{\eta} + \underbrace{2\eta L^2 \mu_\gamma \geq \epsilon}_{\text{Mean penalty}}\right)\right] \leq \exp\left\{\frac{-T\epsilon^2}{\underbrace{2\bar{\eta}_T \sigma_\gamma^2}_{\text{Variance penalty}} + \frac{2}{3}\eta L^2 (2s+1)P\epsilon}\right\} \tag{2}$$

Intuitively, it says the (E)SSP distance between true optima and current estimate decreases exponentially with more iterations, where lower staleness mean, and staleness variance improve the convergence rate.

Finally, the variance in the (E)SSP estimate is

$$\text{Var}_{t+1} = \text{Var}_t - 2\eta_t \text{Cov}(x_t, \mathbb{E}(g_t)) + O(\eta_t \xi_t) + O(\eta_2^2 \rho_t^2) + O_{\gamma_t}^* \tag{3}$$

where $O_{\gamma_t}^*$ represents 5th order or higher terms in γ_t. Basically, it says that the variance in the (E)SSP parameter estimate monotonically decreases when close to an optimum.