Engineering 2 (2016) 179–195

Contents lists available at ScienceDirect

Engineering

journal homepage: www.elsevier.com/locate/eng

Research
iCity & Big Data—Review

Strategies and Principles of Distributed Machine Learning on Big Data

Eric P. Xing *, Qirong Ho, Pengtao Xie, Dai Wei

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA

ARTICLE INFO

Article history:
Received 29 December 2015
Revised 1 May 2016
Accepted 23 May 2016
Available online 30 June 2016

Keywords:
Machine learning
Artificial intelligence big data
Big model
Distributed systems
Principles
Theory
Data-parallelism
Model-parallelism

ABSTRACT

The rise of big data has led to new demands for machine learning (ML) systems to learn complex models, with millions to billions of parameters, that promise adequate capacity to digest massive datasets and offer powerful predictive analytics (such as high-dimensional latent features, intermediate representations, and decision functions) thereupon. In order to run ML algorithms at such scales, on a distributed cluster with tens to thousands of machines, it is often the case that significant engineering efforts are required—and one might fairly ask whether such engineering truly falls within the domain of ML research. Taking the view that "big" ML systems can benefit greatly from ML-rooted statistical and algorithmic insights—and that ML researchers should therefore not shy away from such systems design—we discuss a series of principles and strategies distilled from our recent efforts on industrial-scale ML solutions. These principles and strategies span a continuum from application, to engineering, and to theoretical research and development of big ML systems and architectures, with the goal of understanding how to make them efficient, generally applicable, and supported with convergence and scaling guarantees. They concern four key questions that traditionally receive little attention in ML research: How can an ML program be distributed over a cluster? How can ML computation be bridged with inter-machine communication? How can such communication be performed? What should be communicated between machines? By exposing underlying statistical and algorithmic characteristics unique to ML programs but not typically seen in traditional computer programs, and by dissecting successful cases to reveal how we have harnessed these principles to design and develop both high-performance distributed ML software as well as general-purpose ML frameworks, we present opportunities for ML researchers and practitioners to further shape and enlarge the area that lies between ML and systems. .

1. Introduction

Machine learning (ML) has become a primary mechanism for distilling structured information and knowledge from raw data, turning them into automatic predictions and actionable hypotheses for diverse applications, such as: analyzing social networks [1]; reasoning about customer behaviors [2]; interpreting texts, images, and videos [3]; identifying disease and treatment paths [4]; driving vehicles without the need for a human [5]; and tracking anomalous activity for cybersecurity [6], among others. The majority of ML applications are supported by a moderate number of families of well-developed ML approaches, each of which embodies a continuum of technical elements from model design, to algorithmic innovation, and even to perfection of the software implementation, and which attracts ever-growing novel contributions from the research and development community. Modern examples of such approaches include graphical models [7–9], regularized Bayesian models [10–12], nonparametric Bayesian models [13,14], sparse structured models [15,16], large-margin methods [17,18], deep learning [19,20], matrix factorization [21,22], sparse coding [23,24], and latent space modeling [1,25]. A common ML practice that ensures mathematical soundness and outcome reproducibility is for practitioners and

* Corresponding author.
E-mail address: epxing@cs.cmu.edu

http://dx.doi.org/10.1016/J.ENG.2016.02.008

researchers to write an ML program (using any generic high-level programming language) for an application-specific instance of a particular ML approach (e.g., semantic interpretation of images via a deep learning model such as a convolution neural network). Ideally, this program is expected to execute quickly and accurately on a variety of hardware and cloud infrastructure: laptops, server machines, graphics processing units (GPUs), cloud computing and virtual machines, distributed network storage, Ethernet and Infiniband networking, to name just a few. Thus, the program is hardware-agnostic but ML-explicit (i.e., following the same mathematical principle when trained on data and attaining the same result regardless of hardware choices).

With the advancements in sensory, digital storage, and Internet communication technologies, conventional ML research and development—which excel in model, algorithm, and theory innovations—are now challenged by the growing prevalence of big data collections, such as hundreds of hours of video uploaded to video-sharing sites every minute[†], or petabytes of social media on billion-plus-user social networks[‡]. The rise of big data is also being accompanied by an increasing appetite for higher-dimensional and more complex ML models with billions to trillions of parameters, in order to support the ever-increasing complexity of data, or to obtain still higher predictive accuracy (e.g., for better customer service and medical diagnosis) and support more intelligent tasks (e.g., driverless vehicles and semantic interpretation of video data) [26,27]. Training such big ML models over such big data is beyond the storage and computation capabilities of a single machine. This gap has inspired a growing body of recent work on distributed ML, where ML programs are executed across research clusters, data centers, and cloud providers with tens to thousands of machines. Given P machines instead of one machine, one would expect a nearly P-fold speedup in the time taken by a distributed ML program to complete, in the sense of attaining a mathematically equivalent or comparable solution to that produced by a single machine; yet, the reported speedup often falls far below this mark. For example, even recent state-of-the-art implementations of topic models [28] (a popular method for text analysis) cannot achieve 2× speedup with 4× machines, because of mathematical incorrectness in the implementation (as shown in Ref. [25]), while deep learning on MapReduce-like systems such as Spark has yet to achieve 5× speedup with 10× machines [29]. Solving this scalability challenge is therefore a major goal of distributed ML research, in order to reduce the capital and operational cost of running big ML applications.

Given the iterative-convergent nature of most—if not all—major ML algorithms powering contemporary large-scale applications, at a first glance one might naturally identify two possible avenues toward scalability: faster convergence as measured by iteration number (also known as convergence rate in the ML community), and faster per-iteration time as measured by the actual speed at which the system executes an iteration (also known as throughput in the system community). Indeed, a major current focus by many distributed ML researchers is on algorithmic correctness as well as faster convergence rates over a wide spectrum of ML approaches [30,31] However, it is difficult for many of the "accelerated" algorithms from this line of research to reach industry-grade implementations because of their idealized assumptions on the system—for example, the assumption that networks are infinitely fast (i.e., zero synchronization cost), or the assumption that all machines make the algorithm progress at the same rate (implying no background tasks and only a single user of the cluster, which are unrealistic expectations for real-world research and production clusters shared by many users). On the other hand, systems researchers focus on high iteration throughput (more iterations per second) and fault-recovery guarantees, but may choose to assume that the ML algorithm will work correctly under non-ideal execution models (such as fully asynchronous execution), or that it can be rewritten easily under a given abstraction (such as MapReduce or Vertex Programming) [32–34]. In both ML and systems research, issues from the other side can become oversimplified, which may in turn obscure new opportunities to reduce the capital cost of distributed ML. In this paper, we propose a strategy that combines ML-centric and system-centric thinking, and in which the nuances of both ML algorithms (mathematical properties) and systems hardware (physical properties) are brought together to allow insights and designs from both ends to work in concert and amplify each other.

Many of the existing general-purpose big data software platforms present a unique tradeoff among correctness, speed of execution, and ease-of-programmability for ML applications. For example, dataflow systems such as Hadoop and Spark [34] are built on a MapReduce-like abstraction [32] and provide an easy-to-use programming interface, but have paid less attention to ML properties such as error tolerance, fine-grained scheduling of computation, and communication to speed up ML programs. As a result, they offer correct ML program execution and easy programming, but are slower than ML-specialized platforms [35,36]. This (relative) lack of speed can be partly attributed to the bulk synchronous parallel (BSP) synchronization model used in Hadoop and Spark, in which machines assigned to a group of tasks must wait at a barrier for the slowest machine to finish, before proceeding with the next group of tasks (e.g., all Mappers must finish before the Reducers can start) [37]. Other examples include graph-centric platforms such as GraphLab and Pregel, which rely on a graph-based "vertex programming" abstraction that opens up new opportunities for ML program partitioning, computation scheduling, and flexible consistency control; hence, they are usually correct and fast for ML. However, ML programs are not usually conceived as vertex programs (instead, they are mathematically formulated as iterative-convergent fixed-point equations), and it requires non-trivial effort to rewrite them as such. In a few cases, the graph abstraction may lead to incorrect execution or suboptimal execution speed [38,39]. Of recent note is the parameter server paradigm [28,36,37,40,41], which provides a "design template" or philosophy for writing distributed ML programs from the ground up, but which is not a programmable platform or work-partitioning system in the same sense as Hadoop, Spark, GraphLab, and Pregel. Taking into account the common ML practice of writing ML programs for application-specific instances, a usable software platform for ML practitioners could instead offer two utilities: ① a ready-to-run set of ML workhorse implementations—such as stochastic proximal descent algorithms [42,43], coordinate descent algorithms [44], or Markov Chain Monte Carlo (MCMC) algorithms [45]—that can be re-used across different ML algorithm families; and ② an ML distributed cluster operating system supporting these workhorse implementations, which partitions and executes these workhorses across a wide variety of hardware. Such a software platform not only realizes the capital cost reductions obtained through distributed ML research, but even complements them by reducing the human cost (scientist- and engineer-hours) of big ML applications, through easier-to-use programming libraries and cluster management interfaces.

With the growing need to enable data-driven knowledge distil-

[†] https://www.youtube.com/yt/press/statistics.html
[‡] https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb/