CHAPTER 1

Scaling Up Machine Learning: Introduction

Ron Bekkerman, Mikhail Bilenko, and John Langford

Distributed and parallel processing of very large datasets has been employed for decades in specialized, high-budget settings, such as financial and petroleum industry applications. Recent years have brought dramatic progress in usability, cost effectiveness, and diversity of parallel computing platforms, with their popularity growing for a broad set of data analysis and machine learning tasks.

The current rise in interest in scaling up machine learning applications can be partially attributed to the evolution of hardware architectures and programming frameworks that make it easy to exploit the types of parallelism realizable in many learning algorithms. A number of platforms make it convenient to implement concurrent processing of data instances or their features. This allows fairly straightforward parallelization of many learning algorithms that view input as an unordered batch of examples and aggregate isolated computations over each of them.

Increased attention to large-scale machine learning is also due to the spread of very large datasets across many modern applications. Such datasets are often accumulated on distributed storage platforms, motivating the development of learning algorithms that can be distributed appropriately. Finally, the proliferation of sensing devices that perform real-time inference based on high-dimensional, complex feature representations drives additional demand for utilizing parallelism in learning-centric applications. Examples of this trend include speech recognition and visual object detection becoming commonplace in autonomous robots and mobile devices.

The abundance of distributed platform choices provides a number of options for implementing machine learning algorithms to obtain efficiency gains or the capability to process very large datasets. These options include customizable integrated circuits (e.g., Field-Programmable Gate Arrays – FPGAs), custom processing units (e.g., general-purpose Graphics Processing Units – GPUs), multiprocessor and multicore parallelism, High-Performance Computing (HPC) clusters connected by fast local networks, and datacenter-scale virtual clusters that can be rented from commercial cloud computing providers. Aside from the multiple platform options, there exists a variety of programming frameworks in which algorithms can be implemented. Framework choices tend to be particularly diverse for distributed architectures, such as clusters of commodity PCs.

The wide range of platforms and frameworks for parallel and distributed computing presents both opportunities and challenges for machine learning scientists and engineers. Fully exploiting the available hardware resources requires adapting some algorithms and redesigning others to enable their concurrent execution. For any prediction model and learning algorithm, their structure, dataflow, and underlying task decomposition must be taken into account to determine the suitability of a particular infrastructure choice.

Chapters making up this volume form a representative set of state-of-the-art solutions that span the space of modern parallel computing platforms and frameworks for a variety of machine learning algorithms, tasks, and applications. Although it is infeasible to cover every existing approach for every platform, we believe that the presented set of techniques covers most commonly used methods, including the popular "top performers" (e.g., boosted decision trees and support vector machines) and common "baselines" (e.g., k-means clustering).

Because most chapters focus on a single choice of platform and/or framework, the rest of this introduction provides the reader with unifying context: a brief overview of machine learning basics and fundamental concepts in parallel and distributed computing, a summary of typical task and application scenarios that require scaling up learning, and thoughts on evaluating algorithm performance and platform trade-offs. Following these are an overview of the chapters and bibliography notes.

1.1 Machine Learning Basics

Machine learning focuses on constructing algorithms for making predictions from data. A machine learning task aims to identify (to *learn*) a function $f : \mathcal{X} \to \mathcal{Y}$ that maps input domain $\mathcal{X}$ (of data) onto output domain $\mathcal{Y}$ (of possible predictions). The function f is selected from a certain function class, which is different for each family of learning algorithms. Elements of $\mathcal{X}$ and $\mathcal{Y}$ are application-specific representations of data objects and predictions, respectively.

Two canonical machine learning settings are *supervised learning* and *unsupervised learning*. Supervised learning algorithms utilize *training data* to construct a prediction function f, which is subsequently applied to *test* instances. Typically, training data is provided in the form of *labeled examples* $(x, y) \in \mathcal{X} \times \mathcal{Y}$, where x is a data instance and y is the corresponding *ground truth* prediction for x.

The ultimate goal of supervised learning is to identify a function f that produces accurate predictions on test data. More formally, the goal is to minimize the prediction error (*loss*) function $l : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$, which quantifies the difference between any $f(x)$ and y – the predicted output of x and its ground truth label. However, the loss cannot be minimized directly on test instances and their labels because they are typically unavailable at training time. Instead, supervised learning algorithms aim to construct predictive functions that *generalize* well to previously unseen data, as opposed to performing optimally just on the given training set, that is, *overfitting* the training data.

The most common supervised learning setting is *induction*, where it is assumed that each training and test example (x, y) is sampled from some unknown joint probability distribution P over $\mathcal{X} \times \mathcal{Y}$. The objective is to find f that minimizes *expected loss* $\mathbb{E}_{(x,y)\sim P}\ l(f(x), y)$. Because the joint distribution P is unknown, expected loss cannot be minimized in closed form; hence, learning algorithms approximate it based on training examples. Additional supervised learning settings include *semi-supervised learning* (where the input data consists of both labeled and unlabeled instances), *transfer learning*, and *online learning* (see Section 1.6.3).

Two classic supervised learning tasks are *classification* and *regression*. In classification, the output domain is a finite discrete set of categories (*classes*), $\mathcal{Y} = \{c_1, ..., c_k\}$, whereas in regression the output domain is the set of real numbers, $\mathcal{Y} = \mathbb{R}$. More complex output domains are explored within advanced learning frameworks, such as *structured learning* (Bakir et al., 2007).

The simplest classification scenario is *binary*, in which there are two classes. Let us consider a small example. Assume that the task is to learn a function that predicts whether an incoming email message is spam or not. A common way to represent textual messages is as large, sparse vectors, in which every entry corresponds to a vocabulary word, and non-zero entries represent words that are present in the message. The label can be represented as 1 for spam and -1 for nonspam. With this representation, it is common to learn a vector of weights w optimizing $f(x) = \text{sign}\left(\sum_i w_i x_i\right)$ so as to predict the label.

The most prominent example of unsupervised learning is *data clustering*. In clustering, the goal is to construct a function f that partitions an *unlabeled* dataset into $k = |\mathcal{Y}|$ clusters, with $\mathcal{Y}$ being the set of cluster indices. Data instances assigned to the same cluster should presumably be more similar to each other than to data instances assigned to any other cluster. There are many ways to define similarity between data instances; for example, for vector data, (inverted) Euclidean distance and cosine similarity are commonly used. Clustering quality is often measured against a dataset with existing class labels that are withheld during clustering: a quality measure penalizes f if it assigns instances of the same class to different clusters and instances of different classes to the same cluster.

We note that both supervised and unsupervised learning settings distinguish between *learning* and *inference* tasks, where learning refers to the process of identifying the prediction function f, while inference refers to computing $f(x)$ on a data instance x. For many learning algorithms, inference is a component of the learning process, as predictions of some interim candidate f' on the training data are used in the search for the optimal f. Depending on the application domain, scaling up may be required for either the learning or the inference algorithm, and chapters in this book present numerous examples of speeding up both.

1.2 Reasons for Scaling Up Machine Learning

There are a number of settings where a practitioner could find the scale of a machine learning task daunting for single-machine processing and consider employing parallelization. Such settings are characterized by:

1. **Large number of data instances:** In many domains, the number of potential training examples is extremely large, making single-machine processing infeasible.

2. **High input dimensionality:** In some applications, data instances are represented by a very large number of features. Machine learning algorithms may partition computation across the set of features, which allows scaling up to lengthy data representations.
3. **Model and algorithm complexity:** A number of high-accuracy learning algorithms either rely on complex, nonlinear models, or employ computationally expensive subroutines. In both cases, distributing the computation across multiple processing units can be the key enabler for learning on large datasets.
4. **Inference time constraints:** Applications that involve sensing, such as robot navigation or speech recognition, require predictions to be made in real time. Tight constraints on inference speed in such settings invite parallelization of inference algorithms.
5. **Prediction cascades:** Applications that require sequential, interdependent predictions have highly complex joint output spaces, and parallelization can significantly speed up inference in such settings.
6. **Model selection and parameter sweeps:** Tuning hyper-parameters of learning algorithms and statistical significance evaluation require multiple executions of learning and inference. Fortunately, these procedures belong to the category of so-called *embarrassingly parallelizable* applications, naturally suited for concurrent execution.

The following sections discuss each of these scenarios in more detail.

1.2.1 Large Number of Data Instances

Datasets that aggregate billions of events per day have become common in a number of domains, such as internet and finance, with each event being a potential input to a learning algorithm. Also, more and more devices include sensors continuously logging observations that can serve as training data. Each data instance may have, for example, thousands of non-zero features on average, resulting in datasets of 10^{12} instance–feature pairs per day. Even if each feature takes only 1 byte to store, datasets collected over time can easily reach hundreds of terabytes.

The preferred way to effectively process such datasets is to combine the distributed storage and bandwidth of a cluster of machines. Several computational frameworks have recently emerged to ease the use of large quantities of data, such as MapReduce and DryadLINQ, used in several chapters in this book. Such frameworks combine the ability to use high-capacity storage and execution platforms with programming via simple, naturally parallelizable language primitives.

1.2.2 High Input Dimensionality

Machine learning and data mining tasks involving natural language, images, or video can easily have input dimensionality of 10^6 or higher, far exceeding the comfortable scale of $10 - 1{,}000$ features considered common until recently. Although data in some of these domains is sparse, that is not always the case; sparsity is also lost in the parameter space of many algorithms. Parallelizing the computation across features can thus be an attractive pathway for scaling up computation to richer representations, or just for speeding up algorithms that naturally iterate over features, such as decision trees.

1.2.3 Model and Algorithm Complexity

Data in some domains has inherently nonlinear structure with respect to the basic features (e.g., pixels or words). Models that employ highly nonlinear representations, such as decision tree ensembles or multi-layer (deep) networks, can significantly outperform simpler algorithms in such applications. Although feature engineering can yield high accuracies with computationally cheap linear models in these domains, there is a growing interest in learning as automatically as possible from the base representation. A common characteristic of algorithms that attempt this is their substantial computational complexity. Although the training data may easily fit on one machine, the learning process may simply be too slow for a reasonable development cycle. This is also the case for some learning algorithms, the computational complexity of which is superlinear in the number of training examples.

For problems of this nature, parallel multinode or multicore implementations appear viable and have been employed successfully, allowing the use of complex algorithms and models for larger datasets. In addition, coprocessors such as GPUs have also been employed successfully for fast transformation of the original input space.

1.2.4 Inference Time Constraints

The primary means for reducing the testing time is via embarrassingly parallel replication. This approach works well for settings where *throughput* is the primary concern – the number of evaluations to be done is very large. Consider, for example, evaluating 10^{10} emails per day in a spam filter, which is not expected to output results in real time, yet must not become backlogged.

Inference latency is generally a more stringent concern compared to throughput. Latency issues arise in any situation where systems are waiting for a prediction, and the overall application performance degrades rapidly with latency. For instance, this occurs for a car-driving robot making path planning decisions based on several sensors, or an online news provider that aims to improve user experience by selecting suggested stories using on-the-fly personalization.

Constraints on throughput and latency are not entirely compatible – for example, data pipelining trades throughput for latency. However, for both of them, utilizing highly parallelized hardware architectures such as GPUs or FPGAs has been found effective.

1.2.5 Prediction Cascades

Many real-world problems such as object tracking, speech recognition, and machine translation require performing a sequence of interdependent predictions, forming *prediction cascades*. If a cascade is viewed as a single inference task, it has a large joint output space, typically resulting in very high computational costs due to increased computational complexity. Interdependencies between the prediction tasks are typically tackled by stagewise parallelization of individual tasks, along with adaptive task management, as illustrated by the approach of Chapter 21 to speech recognition.

1.2.6 Model Selection and Parameter Sweeps

The practice of developing, tuning, and evaluating learning algorithms relies on workflow that is embarrassingly parallel: it requires no intercommunication between the tasks with independent executions on the same dataset. Two particular processes of this nature are parameter sweeps and statistical significance testing. In parameter sweeps, the learning algorithm is run multiple times on the same dataset with different settings, followed by evaluation on a validation set. During statistical significance testing procedures such as cross-validation or bootstrapping, training and testing is performed repeatedly on different dataset subsets, with results aggregated for subsequent measurement of statistical significance. Usefulness of parallel platforms is obvious for these tasks, as they can be easily performed concurrently without the need to parallelize actual learning and inference algorithms.

1.3 Key Concepts in Parallel and Distributed Computing

Performance gains attainable in machine learning applications by employing parallel and distributed systems are driven by concurrent execution of tasks that are otherwise performed serially. There are two major directions in which this concurrency is realized: *data parallelism* and *task parallelism*. Data parallelism refers to simultaneous processing of multiple inputs, whereas task parallelism is achieved when algorithm execution can be partitioned into segments, some of which are independent and hence can be executed concurrently.

1.3.1 Data Parallelism

Data parallelism refers to executing the same computation on multiple inputs concurrently. It is a natural fit for many machine learning applications and algorithms that accept input data as a batch of independent samples from an underlying distribution. Representation of these samples via an instance-by-feature matrix naturally suggests two orthogonal directions for achieving data parallelism. One is partitioning the matrix rowwise into subsets of instances that are then processed independently (e.g., when computing the update to the weights for logistic regression). The other is splitting it columnwise for algorithms that can decouple the computation across features (e.g., for identifying the split feature in decision tree construction).

The most basic example of data parallelism is encountered in embarrassingly parallel algorithms, where the computation is split into concurrent subtasks requiring no intercommunication, which run independently on separate data subsets. A related simple implementation of data parallelism occurs within the *master–slave communication model*: a master process distributes the data across slave processes that execute the same computation (see, e.g., Chapters 8 and 16).

Less obvious cases of data parallelism arise in algorithms where instances or features are not independent, but there exists a well-defined relational structure between them that can be represented as a graph. Data parallelism can then be achieved if the computation can be partitioned across instances based on this structure. Then, concurrent execution on different partitions is interlaced with exchange of information across them; approaches presented in Chapters 10 and 15 rely on this algorithmic pattern.

The foregoing examples illustrate coarse-grained data parallelism over subsets of instances or features that can be achieved via algorithm design. Fine-grained data parallelism, in contrast, refers to exploiting the capability of modern processor architectures that allow parallelizing vector and matrix computations in hardware. Standard libraries such as BLAS and LAPACK[1] provide routines that abstract out the execution of basic vector and matrix operations. Learning algorithms that can be represented as cascades of such operations can then leverage hardware-supported parallelism by making the corresponding API calls, dramatically simplifying the algorithms' implementation.

1.3.2 Task Parallelism

Unlike data parallelism defined by performing the same computation on multiple inputs simultaneously, task parallelism refers to segmenting the overall algorithm into parts, some of which can be executed concurrently. Fine-grained task parallelism for numerical computations can be performed automatically by many modern architectures (e.g., via pipelining) but can also be implemented semimanually on certain platforms, such as GPUs, potentially resulting in very significant efficiency gains, but requiring in-depth platform expertise. Coarse-grained task parallelism requires explicit encapsulation of each task in the algorithm's implementation as well as a scheduling service, which is typically provided by a programming framework.

The partitioning of an algorithm into tasks can be represented by a directed acyclic graph, with nodes corresponding to individual tasks, and edges representing inter-task dependencies. Dataflow between tasks occurs naturally along the graph edges. A prominent example of such a platform is MapReduce, a programming model for distributed computation introduced by Dean and Ghemawat (2004), on which several chapters in this book rely; see Chapter 2 for more details. Additional cross-task communication can be supported by platforms via point-to-point and broadcast messaging. The Message Passing Interface (MPI) introduced by Gropp et al. (1994) is an example of such messaging protocol that is widely supported across many platforms and programming languages. Several chapters in this book rely on it; see Section 4.4 of Chapter 4 for more details. Besides wide availability, MPI's popularity is due to its flexibility: it supports both point-to-point and collective communication, with synchronous and asynchronous mechanisms.

For many algorithms, scaling up can be most efficiently achieved by a mixture of data and task parallelism. Capability for hybrid parallelism is realized by most modern platforms: for example, it is exhibited both by the highly distributed DryadLINQ framework described in Chapter 3 and by computer vision algorithms implemented on GPUs and customized hardware as described in Chapters 18 and 19.

1.4 Platform Choices and Trade-Offs

Let us briefly summarize the key dimensions along which parallel and distributed platforms can be characterized. The classic taxonomy of parallel architectures proposed

[1] http://www.netlib.org/blas/ and http://www.netlib.org/lapack/.

by Flynn (1972) differentiates them by concurrency of algorithm execution (single vs. multiple instruction) and input processing (single vs. multiple data streams). Further distinctions can be made based on the configuration of shared memory and the organization of processing units. Modern parallel architectures are typically based on hybrid topologies where processing units are organized hierarchically, with multiple layers of shared memory. For example, GPUs typically have dozens of multiprocessors, each of which has multiple stream processors organized in "blocks". Individual blocks have access to relatively small locally shared memory and a much larger globally shared memory (with higher latency).

Unlike parallel architectures, distributed computing platforms typically have larger (physical) distances between processing units, resulting in higher latencies and lower bandwidth. Furthermore, individual processing units may be heterogeneous, and direct communication between them may be limited or nonexistent either via shared memory or via message passing, with the extreme case being one where all dataflow is limited to task boundaries, as is the case for MapReduce.

The overall variety of parallel and distributed platforms and frameworks that are now available for machine learning applications may seem overwhelming. However, the following observations capture the key differentiating aspects between the platforms:

- **Parallelism granularity:** Employing hardware-specific solutions – GPUs and FPGAs – allows very fine-grained data and task parallelism, where elementary numerical tasks (operations on vectors, matrices, and tensors) can be spread across multiple processing units with very high throughput achieved by pipelining. However, using this capability requires redefining the entire algorithm as a dataflow of such elementary tasks and eliminating bottlenecks. Moving up to parallelism across cores and processors in generic CPUs, the constraints on defining the algorithm as a sequence of finely tuned stages are relaxed, and parallelism is no longer limited to elementary numeric operations. With cluster- and datacenter-scale solutions, defining higher-granularity tasks becomes imperative because of increasing communication costs.
- **Degree of algorithm customization:** Depending on platform choice, the complexity of algorithm redesign required for enabling concurrency may vary from simply using a third-party solution for automatic parallelization of an existing imperative or declarative-style implementation, to having to completely re-create the algorithm, or even implement it directly in hardware. Generally, implementing learning algorithms on hardware-specific platforms (e.g., GPUs) requires significant expertise, hardware-aware task configuration, and avoiding certain commonplace software patterns such as branching. In contrast, higher-level parallel and distributed systems allow using multiple, commonplace programming languages extended by APIs that enable parallelism.
- **Ability to mix programming paradigms:** Declarative programming languages are becoming increasingly popular for large-scale data manipulation, borrowing from a variety of predecessors – from functional programming to SQL – to make parallel programming easier by expressing algorithms primarily as a mixture of logic and dataflow. Such languages are often hybridized with the classic imperative programming to provide maximum expressiveness. Examples of this trend include Microsoft's DryadLINQ,

Google's Sawzall and Pregel, and Apache Pig and Hive. Even in applications where such declarative-style languages are insufficient for expressing the learning algorithms, they are often used for computing the basic first- and second-order statistics that produce highly predictive features for many learning tasks.

- **Dataset scale-out:** Applications that process datasets too large to fit in memory commonly rely on distributed filesystems or shared-memory clusters. Parallel computing frameworks that are tightly coupled with distributed dataset storage allow optimizing task allocation during scheduling to maximize local dataflows. In contrast, scheduling in hardware-specific parallelism is decoupled from storage solutions used for very large datasets and hence requires crafting manual solutions to maximize throughput.
- **Offline vs online execution:** Distributed platforms typically assume that their user has higher tolerance for failures and latency compared to hardware-specific solutions. For example, an algorithm implemented via MapReduce and submitted to a virtual cluster typically has no guarantees on completion time. In contrast, GPU-based algorithms can assume dedicated use of the platform, which may be preferable for real-time applications.

Finally, we should note that there is a growing trend for hybridization of the multiple parallelization levels: for example, it is now possible to rent clusters comprising multicore nodes with attached GPUs from commercial cloud computing providers. Given a particular application at hand, the choice of the platform and programming framework should be guided by the criteria just given to identify an appropriate solution.

1.5 Thinking about Performance

The term "performance" is deeply ambiguous for parallel learning algorithms, as it includes both predictive accuracy and computational speed, each of which can be measured by a number of metrics. The variety of learning problems addressed in the chapters of this book makes the presented approaches generally incomparable in terms of predictive performance: the algorithms are designed to optimize different objectives in different settings. Even in those cases where the same problem is addressed, such as binary classification or clustering, differences in application domains and evaluation methodology typically lead to incomparability in accuracy results. As a consequence of this, it is not possible to provide a meaningful quantitative summary of relative accuracy across the chapters in the book, although it should be understood in every case that the authors strove to create effective algorithms.

Classical analysis of algorithms' complexity is based on O-notation (or its brethren) to bound and quantify computational costs. This approach meets difficulties with many machine learning algorithms, as they often include optimization-based termination conditions for which no formal analysis exists. For example, a typical early stopping algorithm may terminate when predictive error measured on a holdout test set begins to rise – something that is difficult to analyze because the core algorithm does not have access to this test set by design.

Nevertheless, individual subroutines within learning algorithms do often have clear computational complexities. When examining algorithms and considering their application to a given domain, we suggest asking the following questions:

1. What is the computational complexity of the algorithm or of its subroutine? Is it linear (i.e., $O(\text{input size})$)? Or superlinear? In general, there is a qualitative difference between algorithms scaling as $O(\text{input size})$ and others scaling as $O(\text{input size}^\alpha)$ for $\alpha \geq 2$. For all practical purposes, algorithms with cubic and higher complexities are not applicable to real-world tasks of the modern scale.
2. What is the bandwidth requirement for the algorithm? This is particularly important for any algorithm distributed over a cluster of computers, but is also relevant for parallel algorithms that use shared memory or disk resources. This question comes in two flavors: What is the aggregate bandwidth used? And what is the maximum bandwidth of any node? Answers of the form $O(\text{input size})$, $O(\text{instances})$, and $O(\text{parameters})$ can all arise naturally depending on how the data is organized and the algorithm proceeds. These answers can have a very substantial impact on running time, as the input dataset may be, say, 10^{14} bytes in size, yet have only 10^{10} examples and 10^8 parameters.

Key metrics used for analyzing computational performance of parallel algorithms are speedup, efficiency, and scalability:

- *Speedup* is the ratio of solution time for the sequential algorithms versus its parallel counterpart.
- *Efficiency* measures the ratio of speedup to the number of processors.
- *Scalability* tracks efficiency as a function of an increasing number of processors.

For reasons explained earlier, these measures can be nontrivial to evaluate analytically for machine learning algorithms, and generally should be considered in conjunction with accuracy comparisons. However, these measures are highly informative in empirical studies. From a practical standpoint, given the differences in hardware employed for parallel and sequential implementations, viewing these metrics as functions of costs (hardware and implementation) is important for fair comparisons.

Empirical evaluation of computational costs for different algorithms should be ideally performed by comparing them on the same datasets. As with predictive performance, this may not be done for the work presented in subsequent chapters, given the dramatic differences in tasks, application domains, underlying frameworks, and implementations for the different methods. However, it is possible to consider the general *feature throughput* of the methods presented in different chapters, defined as $\frac{\text{running time}}{\text{input size}}$. Based on the results reported across chapters, well-designed parallelized methods are capable of obtaining high efficiency across the different platforms and tasks.

1.6 Organization of the Book

Chapters in this book span a range of computing platforms, learning algorithms, prediction problems, and application domains, describing a variety of parallelization techniques to scale up machine learning. The book is organized in four parts. The