D

Distributed Machine Learning

Alex Galakatos[1,2], Andrew Crotty[2], and Tim Kraska[2]
[1]Database Group, Brown University, Providence, RI, USA
[2]Department of Computer Science, Brown University, Providence, RI, USA

Synonyms

Data mining; Large-scale learning; Machine learning

Definition

Distributed machine learning refers to multi-node machine learning algorithms and systems that are designed to improve performance, increase accuracy, and scale to larger input data sizes. Increasing the input data size for many algorithms can significantly reduce the learning error and can often be more effective than using more complex methods [8]. Distributed machine learning allows companies, researchers, and individuals to make informed decisions and draw meaningful conclusions from large amounts of data.

Many systems exist for performing machine learning tasks in a distributed environment. These systems fall into three primary categories: *database*, *general*, and *purpose-built* systems. Each type of system has distinct advantages and disadvantages, but all are used in practice depending upon individual use cases, performance requirements, input data sizes, and the amount of implementation effort.

Historical Background

Before the advent of distributed frameworks, users were required to create handwritten solutions in which they were solely responsible for explicitly controlling all aspects of execution. This error-prone and laborious process included managing data distribution, parallelization, synchronization, and fault tolerance, which led to a lengthy development cycle where users had difficulty debugging existing algorithms and implementing new ones.

However, new programming frameworks such as MapReduce [4] greatly simplified the process of distributed computing and allowed users to effortlessly scale algorithms to larger datasets. These frameworks provide primitives with well-defined parallelization semantics along with a distributed runtime environment and file system, thereby allowing users to focus on implementing algorithms rather than managing low-level details.

© Springer Science+Business Media LLC 2017
L. Liu, M.T. Özsu (eds.), *Encyclopedia of Database Systems*,
DOI 10.1007/978-1-4899-7993-3_80647-1

Scientific Fundamentals

Performing machine learning algorithms in a distributed environment first involves conceptually converting single-threaded algorithms to parallel algorithms. This step can often be the most difficult because it is algorithm-specific and requires that the user has a strong understanding of the underlying algorithm. The second step involves actually implementing the parallel algorithms, requiring the user to understand the semantics and runtime of the system in order to achieve correct and efficient parallel execution.

Parallelizing Algorithms

Machine learning algorithms can be divided into two categories: *supervised* and *unsupervised*. Supervised learning takes labeled inputs (e.g., a set of emails labeled spam/not spam) and builds a model that can be used to predict future unlabeled inputs. Unsupervised learning aims to discover properties about the data without relying on labeled instances (e.g., clustering customers into categories for market analysis).

The next section provides an overview of and parallelization details for gradient descent, regression, and k-means clustering, followed by alternative parallelization strategies, including ensemble learning techniques and parallel model training. Note that the following algorithms and strategies are included as examples and that many other methods are used in practice.

Gradient Descent

Gradient descent is a general optimization algorithm used extensively in machine learning that aims to minimize a loss function f. This loss function can be modified to perform a variety of machine learning tasks such as linear regression, recommendation, and support vector machines (SVMs). An example of a loss function for linear regression is the *mean squared error*, which intuitively measures the average distance between the actual and predicted values across all training instances.

The result of the gradient descent algorithm is a vector θ, often referred to as a *model* or *weight vector*, which contains the loss function coefficients that best fit the training data. Once constructed, the model θ can be used to predict unlabeled input data items.

Gradient descent begins with a randomly initialized weight vector that is iteratively updated by stepping in the direction of the largest negative gradient of the loss function. As shown in Algorithm 1, the new weight vector θ_{j+1} is updated by taking the current weight vector θ_j and subtracting the gradient of the loss function evaluated with the current model, where α is the learning rate. In this variant of gradient descent, referred to as *batch gradient descent*, the weights are only updated after processing all n training instances.

Stochastic gradient descent updates model weights for each randomly sampled training instance and is therefore significantly more scalable than batch gradient descent. Stochastic gradient descent has been shown to converge faster than batch gradient descent due to the fact that updates to the model are applied immediately for each instance; hence, each successive instance interacts with a more accurate model. In many cases, stochastic gradient descent can converge on optimal model parameters after only a single pass over the training data.

Algorithm 1 Batch gradient descent

Randomly initialize θ
while !converged **do**
    $\theta_{j+1} = \theta_j - \alpha \nabla f(\theta_j)$
end while

Regression

Linear and *logistic regression* are discriminative classification algorithms that use labeled training instances to find a hyperplane w that optimally separates two classes of data. More specifically, given a set of n training instances, each of the form $\{x_1, x_2, ..., x_m, y\}$, where x_i is a feature value and y is the binary label, the algorithm determines the m coefficients θ of the linear or logistic function that best fit the training data.

Once constructed, this model can be used to classify unlabeled input data.

Using a *least squares* method such as ordinary least squares, a closed-form solution for solving linear regression exists and can be calculated directly. Given the set of feature vectors X and the set of corresponding labels y, θ can be computed using the following equation:

$$\theta = (X^T X)^{-1} X^T y \quad (1)$$

Although the closed-form solution given in Eq. 1 is straightforward to compute using standard linear algebra techniques, the matrix inversion is computationally expensive and does not scale to a large number of training instances. For this reason, the previously described methods based on gradient descent are often used for performing regression tasks in a distributed setting. In this case, each parallel worker locally computes and stores updates to the model using a disjoint subset of the input data. After each worker has finished computing local updates to the model, these updates are then combined globally and redistributed on the subsequent iteration.

K-Means Clustering

K-means clustering is an unsupervised iterative machine learning algorithm that partitions input data into k clusters. K-means selects random initial cluster centroids and then assigns each input data item to the closest centroid. After all data points have been assigned, the algorithm determines the new centroid values by averaging the feature values of all input data items per centroid. The algorithm repeats these steps until completing a set number of iterations or meeting some convergence criteria.

K-means clustering can be executed in a distributed setting using data-level parallelism, where each compute node operates on disjoint data subsets. In this scenario, workers compute the distance to all current centroids and determine the closest centroid to each local data item. Next, for a given data item d assigned to centroid c, the algorithm sets $sum_c = sum_c + d$ and $count_c = count_c + 1$. After all compute nodes have processed their local input data items, the sum and count values across all nodes are globally aggregated to compute the new centroid values.

Ensemble Learning

Ensemble learning involves building a set of diverse classifiers in order to improve the overall accuracy for classification tasks. By building and combining multiple weak learners, users can create a single strong learner with a higher accuracy than any one individual weak learner.

To implement ensemble methods in a distributed environment, users can train each weak learner in parallel using a local subset of the input data. After training n of these classifiers, where n is the number of distributed workers, users can choose between two primary strategies for creating a single strong learner that they can then use to classify unlabeled input data. The first strategy involves actually combining the models from each of the n classifiers to create a single classifier. This method is easy to implement for ensembles that use the same learning algorithm but is generally not possible for ensembles with classifiers that represent their models differently (e.g., decision trees vs. regression models).

For this reason, the second and more popular strategy for merging n distributed classifiers is to combine the output value from each classifier. *Bootstrap aggregation*, often referred to as *bagging*, is a simple yet popular way to combine the outputs from multiple classifiers. Bootstrap aggregation determines the final output value by selecting the mode of the n output values.

Frameworks

A number of systems have been developed to support machine learning applications in a distributed setting. Systems for distributed machine learning can be grouped broadly into three primary categories: database, general, and purpose-built systems. This section summarizes a variety of systems that fall into each category, but note that it is not intended to be a complete survey of all existing systems for machine learning.

Database Systems

A number of solutions have been proposed for implementing and executing machine learning tasks using traditional database management systems (DBMSs). Extensions/modifications are generally required to perform most of these tasks inside a DBMS, since the SQL standard cannot easily express many important aspects of machine learning algorithms (e.g., iteration). The following systems add additional functionality that allows users to execute machine learning tasks directly inside a DBMS.

Bismarck [5] implements an abstraction layer that provides a "unified architecture" and focuses specifically on gradient descent. Users can implement new ML algorithms by specifying an objective function as a user-defined aggregate for the new gradient descent operator.

Other systems such as MADlib [9] provide extensions to SQL that permit users to execute built-in machine learning tasks. For example, a user can perform logistic regression in an existing database by writing a query of the form:

```
SELECT madlib.logregr_train(
            source_table,
            out_table,
            dependent_varname,
            independent_varname,
            grouping_cols,
            max_iter,
            optimizer,
            tolerance,
            verbose
            )
```

Generally, systems in this category enable users to run machine learning tasks on data that is already stored in a DMBS, thus eliminating the need to transfer data into an alternate system. However, users must often transform their data to fit the specified format and are limited to a predefined set of algorithms.

General Frameworks

General frameworks allow users to write custom data processing workflows using a set of API operators directly inside a host language. Although many of these frameworks provide built-in implementations for common machine learning algorithms, they are fundamentally designed for extensibility and arbitrary workloads. Systems in this category range from low-level frameworks that provide only basic functionality to high-level frameworks that provide advanced features, including automatic fault tolerance and a flexible API.

Message Passing Interface (MPI) [6] is a low-level framework designed for high-performance distributed computation. MPI offers many primitives (e.g., send, receive, broadcast, scatter, gather) that allow users to implement a wide range of applications, including machine learning algorithms. However, due to its low-level nature, implementing many machine learning tasks using MPI is often quite labor-intensive and error-prone, since developers must explicitly manage aspects such as data distribution and fault tolerance.

Hadoop [1] is a popular, open-source MapReduce implementation designed for executing workflows on large clusters of commodity machines. Hadoop provides automatic fault tolerance, a distributed file system, and a simple programming abstraction that allows users to analyze petabyte-scale data across thousands of machines. However, Hadoop cannot natively or efficiently support iterative workflows and requires that the user submit a single job for every iteration. Furthermore, intermediate results must be materialized to disk, causing the performance of iterative queries to suffer. Mahout [2] is a library that provides implementations of several machine learning tasks using the Hadoop distributed runtime without requiring users to translate algorithms to the MapReduce paradigm.

Spark [15] is a distributed framework that targets in-memory computations and allows users to compose workflows using a set of predefined API operators in Scala, Java, or Python. A driver program coordinates the parallel execution of tasks and handles synchronization for iterative queries. Spark extends the MapReduce paradigm by providing a more descriptive set of operators, including $filter$, $join$, and $union$, that make it easier to express many machine learning algorithms. Additionally, by keeping the working

dataset in memory, Spark can efficiently support iterative algorithms.

DryadLINQ [14] is a system developed by Microsoft that compiles programs written in the LINQ programming language into jobs that are executed using the Dryad distributed runtime. LINQ is a high-level data manipulation language with a C# programming interface that provides many specialized objects that are useful for machine learning, and Dryad is a parallel execution engine that models tasks as dataflow graphs.

Tupleware [3] is an in-memory analytics system that focuses specifically on complex workloads such as machine learning. Unlike other systems, Tupleware compiles workflows written in any LLVM-based programming language into highly optimized distributed executables for deployment in a cluster. By directly compiling workflows, Tupleware eliminates the substantial overhead associated with interpreted execution models and can also apply many optimizations to the code generation stage that consider properties about the input data, user-defined computations, and underlying hardware.

Purpose-Built Frameworks

Many purpose-built systems have been designed specifically for machine learning. These systems provide either domain-specific languages for machine learning or algorithm-specific optimizations that are not generally applicable.

SystemML [7] offers a declarative, high-level language that can be used to implement machine learning tasks. This language has an R-like syntax and provides many built-in operators for performing matrix operations. Workflows submitted to the system are translated into MapReduce jobs and optimized to avoid multiple passes over the input data.

OptiML [13] provides a Scala-embedded, domain-specific language that is based on linear algebra operations. The language includes Vector, Matrix, and Graph data types, as well as subtypes that permit additional optimizations. The system then generates execution code that targets specialized hardware (e.g., GPUs) and single machines with multiple cores.

Hogwild! [12] is a lock-free implementation of stochastic gradient descent. The system eliminates all locking by allowing processors to update any portion of the model, which is stored in shared memory. This technique eliminates the substantial overhead associated with locking and allows other processors to immediately view the most recent version of the model. The authors show that the elimination of locking yields a sufficiently low error rate while allowing the algorithm to converge significantly faster.

Columbus [10] is a framework specifically designed for performing feature selection in a variety of analytics systems. Columbus provides a set of operations and corresponding optimizations that make it easy and efficient to determine the optimal features for a machine learning task.

MLbase [11] is a system that enables users to specify machine learning tasks using a high-level declarative language. The system's optimizer takes a learning task (e.g., classify a given dataset) and chooses the best algorithm, parameters, and validation strategies. The system then automatically performs the task in a cluster and returns the resulting model and summary to the user.

Key Applications

Machine learning has many important applications, including image recognition, spam filtering, recommendation systems, natural language processing, and bioinformatics. In virtually every application, increasing amounts of data are becoming available, and users need a simple and efficient means of analyzing these disparate data sources. Distributed machine learning allows users to draw conclusions from massive datasets in their entirety within a reasonable amount of time.

Cross-References

- ▸ Data Mining
- ▸ Distributed Systems
- ▸ Machine Learning

Recommended Reading

1. Apache hadoop. http://hadoop.apache.org.
2. Apache mahout. http://mahout.apache.org.
3. Crotty A, Galakatos A, Dursun K, Kraska T, Binnig C, Çetintemel U, Zdonik S. An Architecture for Compiling UDF-centric Workflows. PVLDB, 8(12):1466–1477, 2015.
4. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. In: Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI'04). USENIX Association; 2004.
5. Feng X, Kumar A, Recht B, Ré C. Towards a unified architecture for in-rdbms analytics. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD'12). New York: ACM; 2012. p. 325–36.
6. Forum MP. Mpi: a message-passing interface standard. Technical report, Knoxville; 1994.
7. Ghoting A, Krishnamurthy R, Pednault E, Reinwald B, Sindhwani V, Tatikonda S, Tian Y, Vaithyanathan S. Systemml: declarative machine learning on mapreduce. In: Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE'11). Washington, DC: IEEE Computer Society; 2011. p. 231–42.
8. Halevy A, Norvig P, Pereira F. The unreasonable effectiveness of data. IEEE Intell Syst. 2009;24(2):8–12.
9. Hellerstein JM, Ré C, Schoppmann F, Wang DZ, Fratkin E, Gorajek A, Ng KS, Welton C, Feng X, Li K, Kumar A. The MADlib analytics library: or MAD skills, the SQL. Proc VLDB Endow. 2012;5(12):1700–11.
10. Konda P, Kumar A, Ré C, Sashikanth V. Feature selection in enterprise analytics: a demonstration using an r-based data analytics system. Proc VLDB Endow. 2013;6(12):1306–9.
11. Kraska T, Talwalkar A, Duchi JC, Griffith R, Franklin MJ, Jordan MI. Mlbase: a distributed machine-learning system. In: CIDR; 2013. www.cidrdb.org.
12. Niu F, Recht B, Ré C, Wright SJ. Hogwild: a lock-free approach to parallelizing stochastic gradient descent. In: NIPS; 2011.
13. Sujeeth AK, Lee H, Brown KJ, Chafi H, Wu M, Atreya AR, Olukotun K, Rompf T, Odersky M. Optiml: an implicitly parallel domainspecific language for machine learning. In: Proceedings of the 28th International Conference on Machine Learning (ICML); 2011.
14. Yu Y, Isard M, Fetterly D, Budiu M, Erlingsson U, Gunda PK, Currey J. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08). Berkeley: USENIX Association; 2008. p. 1–14.
15. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12). Berkeley: USENIX Association; 2012. p. 2.