Distributed Machine Learning

Ajesh Koyatan Chathoth, Evangelos Karageorgos, Ravneet Singh

December 7, 2018

Abstract

In this paper, we are trying to synthesize various approaches followed and proposed by researches towards Distributed Machine Learning. The motivations behind the idea of DML are requirement of processing huge volume of data which required to improve the accuracy of the machine learning model, of computations required to perform the machine learning algorithms, computation based on data locality and availability to perform data cleaning or local inference if possible, to address the security and privacy concerns of the IoT or sensor networks such that avoiding huge amount of row data being transmitted over the network.

Contents

1 Introduction

Machine Learning is a method of data prediction or analysis by using the learning algorithms that automates analytical model building by continuously learning from the past experience. It is a branch of artificial intelligence based on the idea that systems can learn from data, identify patterns and make decisions with minimal human intervention. A simple machine learning algorithm can be divided into Data and Model units. Data includes the training set of observations and feature selection module, where the features can be fed manually or by algorithmic process.

A typical Machine learning process in a single core machine or on a centralized server is shown in Figure 1.

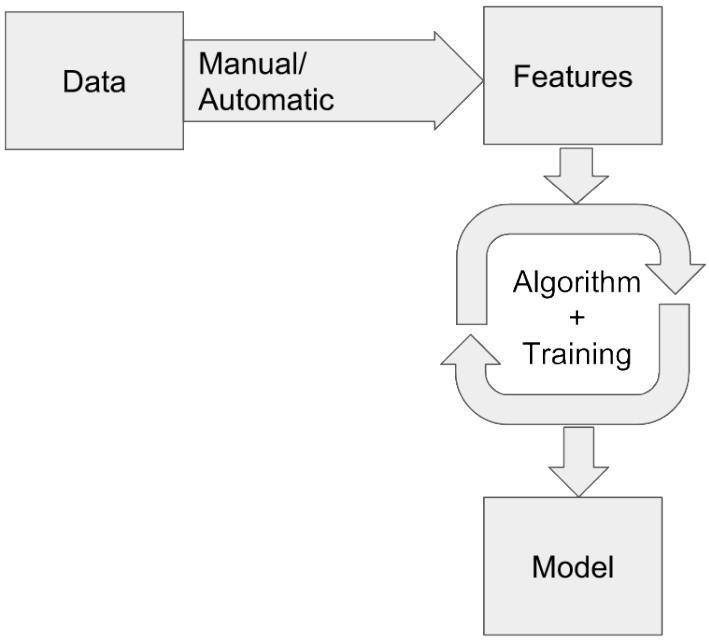

Figure 1: A figure showing the general Machine Learning Process.

To make a better prediction, we need to have better model along with large amount of training data. As we increase the amount of data we use for training the model, we increase the processing time and potentially run out of memory. So if we can run a machine learning algorithm by distributing into various machines which can handle increased memory and computation,

we can, ideally, increase model accuracy.

In this paper we talk about the challenges of distributed machine learning, various strategies of distributed machine learning, some of the early implementation and the state of the art models.

2 Challenges

Any distributed algorithm has inherent limitations and challenges, and if we also add the peculiarities and requirements of machine learning on top, we have a lot of hurdles to overcome. The number-one restriction in all distributed system is communication cost. The applications will be bound by bandwidth and latency. Since the most computation-intensive and data-intensive part of most ML algorithms is their training, computation throughput is far more important than latency, the network bandwidth is the main concern. Having said that, high network latency can also affect the throughput since synchronization costs depend on waiting for messages with the work to arrive. Given that machine learning algorithms are notoriously data-intensive, splitting the algorithm across machines will force a lot of data movement. Anything but excellent network bandwidth can restrict the throughput of the algorithms, while latency can hinder synchronization and slow everything down.

The machines these algorithms will run on can have diverse capabilities and restrictions. If the computation power is asymmetric, slower nodes can slow down the computation. A similar thing can be said about memory capacity of machines. If you want to have any kind of efficiency, you should account for the differences across your devices. As another note, performance characteristics are not our only concern regarding diversity of resources. When we are including mobile devices or IoT devices, in our computations, power management will definitely be a big concern as well. All this can result in poor resource utilization, which is a problem for all distributed systems, but given the requirements of ML algorithms, it can be exacerbated. Also, given that training is the main focus when it comes to computation bottlenecks, we have the problem of disseminating the presumed huge training sets.

Error handling and tolerance is always a concern in distributed systems. But when it comes to ML algorithms, it is a special case. Some error can be tolerated in a lot of cases. Therefore, you can loosen the fault tolerance re-

quirements in order to alleviate communication costs and boost performance. This comes with a cost of accuracy, of course, so you can gauge consistency and error tolerance to tune the balance between accuracy and performance. This makes handling errors non-trivial for distributed ML algorithms. Combined with the fact that most machine learning algorithms were not developed to be distributed, it makes DML applications notoriously difficult to develop.

3 Current State of the Art

The strategies for Distributed Machine Learning can be organized into three categories, each of which distributing one or more parts of the machine learning process. In particular, the parts of the machine learning process that they distribute are the data, model, and the computation, with some of these parts being inspired from typical Distributed Systems. These strategies are a data-parallel approach, model-parallel approach, and a hybrid approach.

The data-parallel approach works by distributing the data to separate workers, and the workers only process on that subset of the data. This strategy takes advantage of how most machine learning algorithms run the same computational steps on each data element, therefore multiplying the processing speed by around the number of workers present, assuming that all the workers are the same in compute speed. This strategy has the disadvantage that the distribution of the data depends heavily on which algorithm is being used and places the burden on the researcher to implement this distributed process manual. An example of the difference in data distribution can be seen when comparing the computation of Neural Networks with that of Decision Trees. In Neural Nets, the weights that we are trying to find are updated through back propagation. In order to get the difference to back propagate we need one data observation to have been sent through the network, where an observation is all of the feature values for one particular instance in the data. Therefore, the data for processing Neural Nets will be distributed at the observation level between workers. Alternatively for Decision Trees, we determine what feature to split on by computing the information gain for each feature [1]. To fit Decision Trees into a data-parallel strategy would mean we should split the data based on the features. For example, $worker_i$ would get all of the data observations with just features a, b, and c, and $worker_j$ could get the rest of the features. This data separation fits better into the calculation for Information Gain instead of separating the data

based on the observations. Also, it can be seen that the researcher would need to carefully consider what algorithms they want to use and focus on before implementing a data-parallel computation well.

The model-parallel approach focuses on placing parts of the model onto different machines. In the case of a Neural Network, $worker_i$ can have some of the layers of a model and any of the computations that are involved with those layers are prepared (the input values are collected) and then executed on those layers. This has the advantage of making it easier to store and manage all of the values of the model and the intermediate computation values in the memory of the machines, letting us increase the size of a Neural Net. One example of this is a Distributed Deep Neural Net made by Teerapittayanon, McDanel, and Kung [2]. In their work, they separate the model and place it on the edge devices and in the cloud to ease the computation burden and to limit the memory used in the cloud. The main disadvantage to this approach is that most Machine Learning algorithms' models are not separable in the way Neural Networks are, for example in a Decision Tree once we determine a split for 2 branches in a tree, we do not need to do any more computation at that part of the model. A variant of this model parallelism is suggested by Lee, Kim, Zheng, *et al.* [3] where we create multiple models at once on separate machines to speed up the process.

The hybrid approach focuses more on distributing the computation in a classical distributed systems sense. We are calling it hybrid because it is doing both data parallelism, in the sense that the data is distributed to particular workers to run that computation, and model parallelism, in the sense of parts of the model are being computed at different workers. One main package that uses this approach is Apache Spark [4]. The approach works by taking all of the work needed in a Machine Learning process and creating a computation DAG that is then allocated to the workers. This method has the advantage in that it is possible to create a DAG for any Machine Learning algorithm that is created, allowing flexibility. The main disadvantage is that Machine Learning models are not typically well suited for distribution in this manner, requiring long chains of dependent executions, and end up being inefficient. We will further discuss the data-parallel and model-parallel strategies since this hybrid approach can easily be described as breaking the work into a computation DAG.

3.1 Data-Parallel Approach

Recall that a data-parallel approach to DML is to distribute the data to different worker machines. One particular implementation of a data-parallel approach is a Parameter Server. This strategy works by breaking the data into Data shards by observation and distributing it to each of the workers. The coordinator of all of this work is the parameter server itself, as shown in Figure 2. The purpose of the parameter server is to hold the the global weights for the model being computed. Each of the workers will have a local set of these weights and one partition of the data. Since the computation of Machine Learning algorithms can be organized into iterations, or sets of the same computational steps, then each worker, P, will use it's subset of the data and compute updates on the local weights. After each iteration, or n iterations, the workers will send their weight updates to the parameter server. This server will aggregate the weight changes using some calculation, such as averaging. Once the weights have been aggregated, the parameter server will send the new values to the workers to continue processing. This process continues either until all of the data has been processed or until the weight updates have converged. Note that one of the main reasons we can separate the data and handle processing in this manner is because there is a level of error tolerance that we can allow in Machine Learning Models [5].

One part that we can control in this parameter server approach is how and when we synchronize the weights. The typical, and most consistent, approach is the Bulk Synchronous Parallel. This strategy initiates a weight update after n iterations by the workers. This is effective for a consistent and accurate model, but is slowed down by the slowest worker. Another strategy that is a slight improvement is Stale Synchronous Parallel. This strategy takes advantage of the difference in speed of machines and the error tolerance of Machine Learning algorithms, by not synchronizing until the fastest and slowest workers are s iterations apart [6]. Once they are this distance, then a synchronization barrier is introduced, we wait for all of the workers to catch up, and then synchronize the values. One issue that Zhang, Tu, Ren, *et al.* [6] note is that if the slowest and longest worker stay within s iterations (called the staleness threshold), then there will never be a synchronization. Zhang, Tu, Ren, *et al.* [6] suggest a strategy to alleviate this called Adaptive Synchronous Parallel (ASP) which works by having 2 thresholds, the normal staleness threshold s and a weak threshold w, the minimum number of iterations of the slowest worker. ASP works by a monitoring the performance of

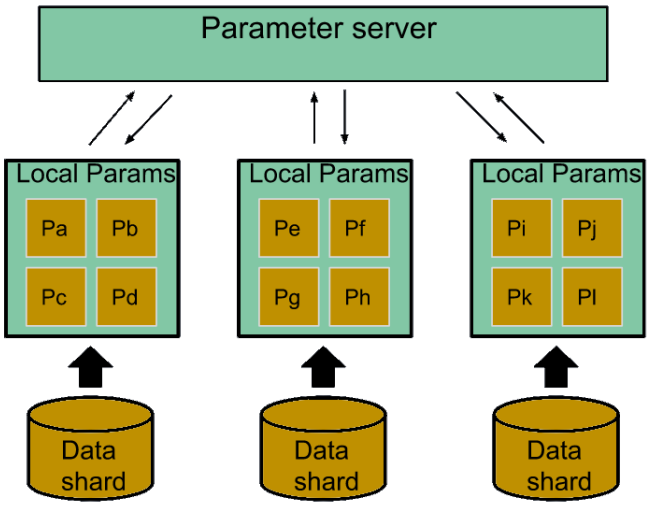

Figure 2: Figure created to highlight how a parameter server works. The workers work with local data on their subset and synchronize their local updates with a parameter server

each of the machines and adjusting the s if there is a large performance gap and uses w to introduce a synchronization barrier when the slowest worker has done w iterations. This helps to ensure that some work is done, and that there is synchronization that occurs despite changing speeds of machines.

3.2 Model-Parallel Approach

Another way to handle distribution is through model-parallelism. One way to parallelize the model is to split parts of the computation over different nodes or machines. In this case, the data is available to all nodes, but different parts of the computation is done on different nodes. In the case of Neural Nets, this can mean that a node can be responsible for a single layer, or part of one layer, or even many parts of different layers. This, of course, means that the dense connectivity of neurons will introduce heavy communication cost between the nodes. Every node will be responsible for its part of the parameter space though, which makes managing such a system simpler. A consequence of such an architecture is that the entire cluster will, in the general case, process one training iteration at a time. This can severely diminish the utilization of the cluster, as while one layer computes, the nodes

that are responsible for other layers are idle. The model parameters won't have to be synchronized, but the intermediate results and potential back propagation will have to traverse the network for every iteration.

A potential solution to the problems mentioned above is to have a pipelined approach to the computations [7]. If the algorithm allows it, for example in case of deep belief networks, there is no back propagation between most of the layers. This means that the data can flow in one direction only, from the training set to every layer in succession. As a consequence, one individual iteration goes from layer to layer until it reaches the end. So, when the iteration is on a particular layer, the machines responsible for other layers can compute part of another iteration. This can ensure the entire cluster is busy computing many successive iterations at once, boosting resource utilization. Despite this strategy's benefits, this can only apply to algorithms that have this linear data flow, such as Neural Nets. Synchronization of weights will be easy to implement, as we only have to include barriers between layers, and possibly with lower cost. However, this approach, as with all pipelined algorithms, suffers from the bottlenecking issue. If one layer is slower to compute than the rest, the entire pipeline will be slower, so care must be taken to make sure that the computation is distributed evenly.

3.3 Current Developments

The initial methods of handling DML involved implementing these strategies from scratch, such as PMLS a parameter server, in C++. The next iteration of implementations is focused on specialized libraries for DML and experimentation. Libraries like Spark/MLlib, TensorFlow and GraphLab employ a complex data flow approach to their computations. They create an execution graph, optimize it with graph-related algorithms [8] and schedule the appropriate computations on the nodes. Their performance is superior to approaches like MapReduce, and are easier to develop ML applications on. Industrial DML solutions with top performance are still hard to develop and deploy, because they need to be hand-crafted, but using these high level libraries for research and prototyping is easy and have decent performance[9]. We start seeing novel approaches to DML, like various hybrid parallelization schemes or variants of parameter server architectures. Researchers experiment with sacrificing consistency and accuracy to gain performance, and develop distributed variants of classic ML algorithms, like distributed gradient descent[10], [11].

It is currently very easy to use Distributed Machine Learning to process data. Packages exist that allow users to use the processing frameworks without worrying too much about the intricacies of distributed systems. One example is Apache Spark and it's library that allow users to create any of the supported machine learning algorithms easily [4]. Unfortunately, we still face the issue of setting up the clusters we need for this ease of use. We also have novel approaches that attempt to improve how we process our data to create new machine learning models, such as Distributed Deep Neural Nets that distribute the model to edge devices [2] and new strategies to improve the speed of convergence for our current strategies such as Adaptive Synchronous Parallel by Zhang, Tu, Ren, *et al.* [6].

4 Further Research

We discuss three strategies to improve the space of DML, one involves ease of use at the machine learning researcher level, another involves improving the processing framework from a distributed systems sense, and finally we discuss a new machine learning algorithm to potentially distribute.

One direction for new research for DML is to increase the ease of use for Machine Learning researchers. Currently with the state of parameter server implementations, such as PMLS in C++, and the hybrid implementations, such as Spark/MLlib or GraphLab [4], [5], [12] it is relatively difficult to implement and use, especially when compared to language like R or Matlab. One direction for research could be to construct an API that hides the distributed implementation details and only reveals the interface in the same primitive that are available in R or MatLab. MLi, an API Machine Learning, suggests this approach but only in the context of Spark integration [9]. We suggest that this be implemented to interface with R at the primitive level and provide little to no difference in the usage. This work would require being able to dynamically select the appropriate version of the algorithm to use and what strategy to select on an unknown, but set, number of workers. This would simplify the distributed Machine Learning Process and and make it easier and faster for researchers to create new and more accurate models.

Current popular implementations of DML have been based on the message-passing model of parallelism. Given that machine learning seem to better suited to a shared memory model of parallelism, we can try exploring those types of solutions. The shared memory model focuses more on mutual exclu-

sion and synchronization over shared resources(memory) and less on message passing. Message-passing architectures have dominated the HPC field in recent times, especially with libraries like MPI, but there are many recent shared-model approaches as well that have great potential (Tasoulas, Anagnostopoulos, Papadopoulos, *et al.* [13] for example). Although it has been established that both memory models are theoretically equivalent [14], since ML algorithms have been developed with shared memory in mind, we should pursue adapting a distributed shared memory framework to work with these algorithms. We want to exploit the potential computation power of clusters with the ML algorithms' affinity to shared state architectures.

Another research avenue is to further study Deep Belief Networks [15] specifically for distributed computation. Their architecture is more suitable to pipelining and has the tendency to require less connectivity(and thus less communication) than other types of networks. Since every layer of Restricted Boltzmann Machines is almost independent for the bulk of its computations, it appears to be a better candidate.

References

[1] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997, ISBN: 978-0-07-042807-2.

[2] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Distributed Deep Neural Networks Over the Cloud, the Edge and End Devices," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, Jun. 2017, pp. 328–339. DOI: 10.1109/ICDCS.2017.226.

[3] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing, "On Model Parallelization and Scheduling Strategies for Distributed Machine Learning," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2014, pp. 2834–2842.

[4] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "MLlib: Machine Learning in Apache Spark," *Journal of Machine Learning Research*, vol. 17, no. 34, pp. 1–7, 2016.

[5] K. Zhang, S. Alqahtani, and M. Demirbas, "A Comparison of Distributed Machine Learning Platforms," in *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, Jul. 2017, pp. 1–9. DOI: 10.1109/ICCCN.2017.8038464.

[6] J. Zhang, H. Tu, Y. Ren, J. Wan, L. Zhou, M. Li, and J. Wang, "An Adaptive Synchronous Parallel Strategy for Distributed Machine Learning," *IEEE Access*, vol. 6, pp. 19 222–19 230, 2018, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2820899.

[7] T. Ben-Nun and T. Hoefler, "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis," *ArXiv:1802.09941 [cs]*, Feb. 2018.

[8] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, Apr. 2012, ISSN: 21508097. DOI: 10.14778/2212351.2212354.

[9] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska, "MLI: An API for Distributed Machine Learning," *ArXiv:1310.5426 [cs, stat]*, Oct. 2013.

[10] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in Neural Information Processing Systems 23*, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, Eds., Curran Associates, Inc., 2010, pp. 2595–2603.

[11] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*, Y. Lechevallier and G. Saporta, Eds., Heidelberg: Physica-Verlag HD, 2010, pp. 177–186, ISBN: 978-3-7908-2604-3.

[12] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012, ISSN: 2150-8097. DOI: 10.14778/2212351.2212354.

[13] Z.-G. Tasoulas, I. Anagnostopoulos, L. Papadopoulos, and D. Soudris, "A message-passing microcoded synchronization for distributed shared memory architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2018, ISSN: 0278-0070. DOI: 10.1109/TCAD.2018.2834423.

[14] H. C. Lauer and R. M. Needham, "On the duality of operating system structures," *ACM SIGOPS Operating Systems Review*, vol. 13, no. 2, pp. 3–19, Apr. 1979. DOI: 10.1145/850657.850658.

[15] G. E. Hinton, "Deep belief networks," *Scholarpedia*, vol. 4, no. 5, p. 5947, 2009, revision #91189. DOI: 10.4249/scholarpedia.5947.