High Performance Computing for Machine Learning (Distributed Stochastic Gradient Descent Implementation using MPI)

Cem Orhan

Abstract

In this report we will summarize our toy project, which is on distributed implementation of stochastic gradient descent (SGD) algorithm using message-passing interface (MPI). We assume that every machine has its own subset of the data, and the software is capable of learning with quadratic loss, logistic loss and hinge loss with ℓ_1 and ℓ_2 regularization using different parameters.

1 Introduction

The field of machine learning proves itself useful on various different tasks such as image recognition [6], speech recognition [5], computer game playing [10] etc. During the last two decades, there is an enormous boost in size of available data and computer power. Besides, along with these advancements more complex machine learning models are proposed, such as deep convolutional neural networks, deep reinforcement learning etc. These complex methods have a successful empirical record in terms of accuracy, but there is a need for vast computing power, because training time of these models are long.

Along with the discussion of vast computing power requirement, most of the machine learning algorithms does not scale well with the increasing amount of data. For example, one of the most popular machine learning algorithm, kernelized SVM (Support Vector Machine) has a memory requirement of n^2, where n stands for the size of the data [9]. This fact opens up scalability problems when the data size is in the order of millions. For addressing such problems, using stochastic optimization based algorithms yields a better training time and scalability performance [3].

In addition to previous point, training a model with millions of data on a single machine is not a good idea, as the training time increases with increasing data, and some of the machine learning models have a structure that makes the training procedure possible in a parallel/distributed manner. As a result, distributed machine learning algorithms get significant attraction from the field, where the idea is dividing the data into equivalent sized partitions and train models on these seperate partitions with a synchronization mechanism. There are various frameworks and platforms that makes distributed training possible, such as Spark [1] and Tensorflow [2] [1], where the distributed setting is handled by the platform. While giving ease in development phase, there are drawbacks of such platforms, such as communication time. On the other hand writing distributed machine learning models that uses message-passing interface (MPI) is significantly harder than

[1]spark.apache.org/

[2]https://www.tensorflow.org/

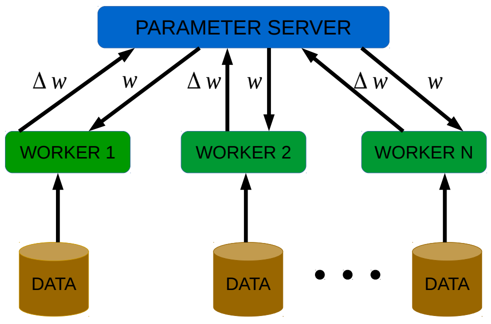

Figure 1: Parameter server approach.

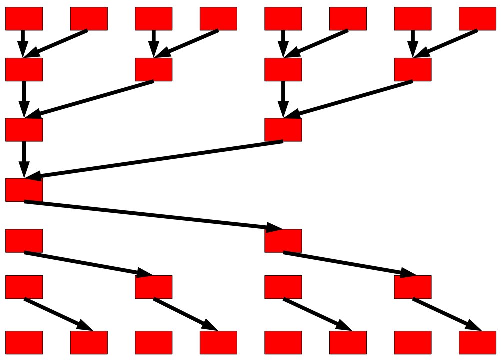

Figure 2: All-reduce approach.

using the aforementioned platforms, since the developer is responsible for handling the distributed aspects such as communication and synchronization. But as a plus, using MPI yields a much better performance in communication time (and training time as a result). Other than the platforms, communication strategy is also a crucial part of the system, and we will discuss about these strategies later in report.

With its parallelizable structure, stochastic optimization based models are quite popular in distributed machine learning setting. We will consider the Stochastic Gradient Descent (SGD) algorithm among these methods to train distributed machine learning models, and the models we will train are linear models.

2 Background

Distributed machine learning aims to scatter the computing in training phase to different machines, every one of which is assumed to have their own subset of data. One can easily observe that, the strategy for synchronization among machines is the most important aspect of this type of algorithms. To this aim various distributed machine learning models are based on two basic communication strategies, parameter server approach and all-reduce approach.

Parameter server approach [4, 7, 8] assumes a central server which has the parameters of the machine learning model, and every machine has its own connection with this server. Assuming the data is scattered among worker machines, every worker can process their own portion of data. While processing the data, intermediate results that are produced by these workers are shared through parameter server. That is, when a worker has an update on parameters, it simply connects to parameter server, sends the update and receives the new parameter set to continue working on. The main issue with this scheme is, if the updates are sychronous, in other words when a worker tries to send an update, it needs to lock the parameter variable for updating so that there will be no other worker with the possibility of changing the parameter. But with this synchronization, the overhead for waiting for an update for a worker will be high. One immediate solution is, if we can assume that the updates are sparse, then we can work with asynchronous updates, where each worker can immediately send the parameter update and receive new parameter. Because with the sparse updates, we know that every worker will update a sparse portion of the parameter, and d epending on the sparsity, probability of having a collision is low. A simple illustration of this scheme can be seen in Figure 1.

Table 1: Table of gradients.

Models	f_i	∇f_i
Linear Regression	$\frac{1}{2}\|\|y_i - \mathbf{w}^T\mathbf{x}_i\|\|_2^2$	$(\mathbf{w}^T\mathbf{x}_i - y_i)\mathbf{x}_i$
Ridge Regression	$\frac{1}{2}\|\|y_i - \mathbf{w}^T\mathbf{x}_i\|\|_2^2 + \frac{\lambda}{2}\|\|\mathbf{w}\|\|_2^2$	$(\mathbf{w}^T\mathbf{x}_i - y_i)\mathbf{x}_i + \lambda\mathbf{w}$
Logistic Regression	$-\log(1+\exp(-(\mathbf{w}^T\mathbf{x}_i)y_i)) + \frac{\lambda}{2}\|\|\mathbf{w}\|\|_2^2$	$\frac{y_i}{1+\exp(-(\mathbf{w}^T\mathbf{x}_i)y_i)}\mathbf{x}_i$
Linear SVM	$\max\{0, 1-(\mathbf{w}^T, \mathbf{x}_i)y_i\} + \frac{\lambda}{2}\|\|\mathbf{w}\|\|_2^2$	$\begin{cases} -y_i\mathbf{x}_i + \lambda\mathbf{w}, & \text{if } (\mathbf{w}^T\mathbf{x}_i)y_i < 1 \\ \lambda\mathbf{w}, & \text{if } (\mathbf{w}^T\mathbf{x}_i)y_i \geq 1 \end{cases}$

All-reduce approach [2] assumes no central server for parameters, instead every machine has updated parameters with a continuous synchronization mechanism based on all-reduce scheme. After computing a parameter update on each machine, every worker connects in such a way that at the end of the protocol new parameter set is distributed across them. In Figure 2, this scheme is illustrated. This approach has the communication after each round of computation, and the communication across the nodes is blocking. In aforementioned figure, a tree-based all-reduce scheme is illustrated, where the communication steps required to complete the task is in the order of logarithm of the number of nodes, which is scalable yet the main bottleneck of such systems.

In our implementation, we focused on all-reduce based scheme.

3 Method

Assume we have a machine learning model that can be written as minimization of finite sums, i.e.

$$\min_{\mathbf{w}\in\mathcal{R}^d} f(\mathbf{w}) = \min_{\mathbf{w}\in\mathcal{R}^d} \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{w}) = \min_{\mathbf{w}\in\mathcal{R}^d} \frac{1}{n}\sum_{i=1}^{n} \ell(\mathbf{w}^T\mathbf{x}_i, y_i) + \lambda\Omega(\mathbf{w})$$

where ℓ is the loss function (data-fitting term) and Ω is regularizer.

As $f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{w})$, we assume that $\nabla f(\mathbf{w}) \approx \frac{1}{n}\sum_{i=1}^{n}\nabla f_i(\mathbf{w})$.

Define $s = \mathbf{w}^T\mathbf{x}$, so $\frac{\partial s}{\partial \mathbf{w}} = \mathbf{x}$.

$$\nabla f_i(\mathbf{w}) = \frac{\partial}{\partial \mathbf{w}}\ell(\mathbf{w}^T\mathbf{x}_i, y_i) + \frac{\partial}{\partial \mathbf{w}}\lambda\Omega(\mathbf{w}) = \frac{\partial}{\partial s}\ell(s, y_i)\mathbf{x_i} + \frac{\partial}{\partial \mathbf{w}}\lambda\Omega(\mathbf{w})$$

In Table 1 we summarized the models in implementation along with their regularized losses and gradients (subgradients where required).

For distributed computing strategy, we used MPI with all-reduce scheme. Assuming each machine has its own data, initializes the weight vector as zero vector. Then they compute the updates using a sampled mini-batch from their data, using the gradients given in Table 1. After computing an update vector on given mini-batch, they start an all-reduce protocol to compute the total weight update. All-reduce is nothing but a reduce protocol followed by a broadcast, hence after each iteration every machine has updated weight vector to continue training. Weight update is summarized in Algorithm 1, and overall procedure is summarized in Algorithm 2.

Algorithm 1 `weightUpdate`

```
Input: $\mathbf{x}$: sampled mini-batch; $y$: labels; $\mathbf{w}_c$: current weights; $k$: mini-batch size; $\gamma$: learning rate
Output: $\Delta\mathbf{w}$: weight update.
$\Delta\mathbf{w} \leftarrow \mathbf{0}$
for $i = 0$ to $k$ do
    Compute $\nabla f_i$ on $\mathbf{x}_i$ using Table 1.
    $\Delta\mathbf{w} \leftarrow \Delta\mathbf{w} + \nabla f_i$
end for
Return $-\gamma\Delta\mathbf{w}$
```

Algorithm 2 `SGD-MPI`

```
Input: $\mathbf{X}_m$: data on machine $m$; $y_m$: labels on machine $m$; $k$: mini-batch size; $\gamma_0$: learning rate; learning method; $\lambda$: regularization parameter; $T$: number of iterations, $M$: number of machines.
Output: $\mathbf{w}$: model weight.
On each machine $m$ do:
$\mathbf{w}_0 \leftarrow \mathbf{0}$
for $t = 0$ to $T$ do
    if learning method is constant then $\gamma = \gamma_0$
    else $\gamma = \gamma_0/(\lambda t)$
    Sample $k$ points from $\mathbf{X}_m$ as $\mathbf{x}$ with corresponding labels $y$
    $\Delta\mathbf{w}_m \leftarrow$weightUpdate$(\mathbf{x}, y, \mathbf{w}^{(t)}, k, \gamma)$
    $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} + \sum_{m=1}^{M} \Delta\mathbf{w}_m$ // Using all-reduce
end for
```

3.1 Implementation Details

Implementation is done using C++ programming language. For implementation of MPI, Open-MPI[3] is used. For main data structures and dot product calculations, Eigen[4] library is chosen because of speed and documentation.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *Journal of Machine Learning Research*, 15(1):1111–1133, 2014.

[3] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[3]https://www.open-mpi.org/
[4]eigen.tuxfamily.org/

[4] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[5] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[7] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, volume 14, pages 583–598, 2014.

[8] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems*, pages 19–27, 2014.

[9] Bernhard Schölkopf and Alexander J Smola. *Learning with kernels: support vector machines, regularization, optimization, and beyond.* MIT press, 2002.

[10] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.