Distributed Large-Scale Learning

Oluwashina Aladejubelo

24/06/2015

Research project performed at AMA-LIG Laboratory
Under the supervision of:
Prof. Massih-Reza Amini, Universite Joseph Fourier
Co-supervised by:
Dr. Marianne Clausel, Universite Joseph Fourier
Defended before a jury composed of:
Prof. Massih-Reza Amini
Prof. Catherine Berrut
Dr. Nabil Layaida
Dr. Jerome Euzenat/Prof. Eric Gaussier
Abstract

An interesting problem in the fields of search engine and ad placement engines, is learning with Big data. Big data is usually too large to fit into the memory of a single machine, in which case mini-batches have to be distributed across several machines. Existing distributed learning methods come with expensive synchronization. This synchronization could be a bad idea in a heterogenous setting where the machines might be of different capacities in which case the slower machine slows down the faster ones and the cost of each iteration equals the computation cost plus the synchronization plus the communication cost. This paper introduces an asynchronous technique for large-scale learning in which the speed of the minimization at the slower machine is at the rate of the faster machine. We experimented this technique with two machines, for which one is fast and the other slower. Experimental result shows that this technique converges.
Contents

Abstract i

1 Introduction 1
  1.1 Machine Learning Systems ........................................... 1
  1.2 Mathematical Problem ................................................ 3
  1.3 Large-scale Learning .................................................. 5
  1.4 Thesis Outline and Contribution ..................................... 5

2 State of the art 7
  2.1 Single Machine Optimization ........................................ 7
  2.2 Multiple Machine Optimization ..................................... 9

3 Asynchronous Large-scale Learning 11
  3.1 Examples of Gossip Schemes ........................................ 11
  3.2 Framework of Gossip Algorithms ................................... 13
  3.3 Our Heterogenous Model .............................................. 14
  3.4 Experiment .............................................................. 18
  3.5 Conclusion ............................................................... 20

4 Conclusion and Future Work 23
  4.1 Conclusion ............................................................... 23
  4.2 Future Work ............................................................. 24

Bibliography 27
  Algorithms ................................................................. 31
1

Introduction

In this chapter we give a background and motivation to the entire work by giving a brief introduction on machine learning and give our motivation for working on a distributed algorithm for large-scale learning.

1.1 Machine Learning Systems

The goal of a learning system is to learn a prediction function that maps a set of input to output space with minimum probability of error, from a set of functions \( \mathcal{F} : \{ f \rightarrow \mathcal{X} : \mathcal{Y} \} \) where \( \mathcal{X} \) is a set of input space and \( \mathcal{Y} \) is a set of output space. The set of inputs for example can be a set of handwritten digits, sensor readings, photographic images, satellite images, a set of click-through data, keywords and so on. The inputs are separated by taking into account their

(a) Learning a function for handwriting recognition [23]

(b) Unsupervised learning can be used for customer segmentation [15]
In some cases, the set of input used to train the system is labeled and this case is termed supervised learning. In a typical supervised learning problem, there is an instance space $\mathcal{X}$ containing (descriptions of) instances or objects for which predictions are to be made, often $\hat{y} = y$, which is not always the case. The training data consists of a finite number of labeled examples $S = ((x_1, y_1), \ldots, (x_n, y_n)) \in (\mathcal{X} \times \mathcal{Y})^n$, and the goal is to learn from these samples a model $f_S : x \rightarrow \hat{y}$ that given a new instance $x \in \mathcal{X}$, predicts $\hat{y} = h_S(x) \in \hat{y}$.

In a supervised learning system each input $x \in \mathcal{X}$ is a pair that comprises of the input object and the desired class or label. In the end it is expected that the learning system generalizes well enough from the training data to be able to make correct predictions when supplied with instances other than the training data. A popular example is the Spam filtering classification problems at popular email services such as Yahoo and Google in which we would like to build an email filter that can predict whether a new message is spam or not, having data containing previous examples of messages labeled as spam or non-spam. Another example is building a model that can automatically recognize handwritten digits from images, having data containing examples of such images labeled with the correct digit.

In another case, a learning system is trained using unlabeled data. This scenario is referred to as unsupervised learning. In an unsupervised learning system, the goal is to find hidden structures in the unlabeled data and cluster them into these groups. An example of unsupervised
learning can found in the e-commerce domain where there is need to identify groups of similar customers based on clickstream patterns and purchase histories. These customer groups with similar behavior and/or preferences could help a company execute a more effective targeted marketing campaign.

In this work we are more concerned about supervised learning, especially in the large-scale context where the training data cannot fit in the memory of a single machine and has to be distributed in minibatches to more than one machine.

1.2 Mathematical Problem

1.2.1 Gradient-Based Learning

Considering a simple supervised learning scenario, let each example \((x, y)\) be represented as \(z\). We consider a loss function \(e(\hat{y}, y)\) which measures the cost of predicting \(\hat{y}\) when the actual answer is \(y\) [11], and we choose a family \(F\) of functions \(f_w(x)\) parameterized by a weight vector \(w\). We seek the function \(f \in F\) that minimizes the loss \(e(f_w(x), y)\) averaged on the examples. Although we would like to average over the unknown distribution \(dP(z)\), we must often settle for computing the average on a sample \(z_1, \ldots, z_n\).

\[
E(f) = \int e(f(x), y) dP(z) \quad (1)
\]
\[
E_n(f) = \frac{1}{n} \sum_{i=1}^{n} e(f(x_i), y_i) \quad (2)
\]

The expected risk \(E(f)\) measures the generalization performance that is expected on future examples, while the empirical risk \(E_n(f)\) measures the training set performance. Based on Vapnik and Chervonenkis(1971), minimizing the empirical risk instead of the expected risk when the chosen family \(F\) is sufficiently restrictive, is justified.

One of the popular methods for minimizing the empirical risk \(E_n(f_w)\) is the gradient descent (GD) technique [10],

\[
w^{(t)} = w^{(t-1)} - \frac{\gamma}{n} \nabla e(f_{w^{(t-1)}}(x_i), y_i) \quad (3)
\]

where \(\gamma\) is the learning rate. Under sufficient regularity assumptions, when the initial estimate \(w_0\) is close enough to the optimum, and when the learning rate \(\gamma\) is sufficiently small, this algorithm converges.
1.2.2 Stochastic Gradient Descent

The stochastic gradient descent (SGD) is a drastic simplification of the gradient descent method by computing the gradient of $E_n(f_w)$ on the basis of a single randomly picked example $z_t$:

$$ w(t) = w(t-1) - \gamma \nabla e(f_w(t-1), x_i, y_i) \quad (4) $$

The stochastic process $w_t, t = 1, ..., \infty$ depends on the examples randomly picked at each iteration. It is hoped that (4) behaves like its expectation (3) despite the noise introduced by this less computationally intensive procedure [9]. Since SGD does not need to remember which examples were visited during the previous iterations, it can process examples on the fly on the machine where it is deployed.

The convergence of SGD has been extensively studied in the stochastic approximation literature [9]. Convergence results usually require decreasing learning rate satisfying the conditions $\sum_t \gamma_t^2 < \infty$ and $\sum_t \gamma_t = \infty$.

Unfortunately, this modification does not reduce the stochastic noise and therefore does not significantly improve the variance of $w_t$. Nevertheless, this method is very much applicable in the large-scale learning scenario. [10] also showed that SGD is amenable to a number of classical machine learning algorithms including Adaline, Perceptron, K-Means, SVM and Lasso.
1.3 Large-scale Learning

Large-scale learning scenarios exist in domains such as ad placement engines where machine learning is used to determine which ad to place where, on a search engine, and search engine domain where machine learning is used for the re-ranking of search engine results. The data to learn from is usually enormous and the number of instances, \( n \) ranging to millions or billions in some cases. In this situation two major problems arise the first being that this data to the tune of terabytes and petabytes can not fit in the memory of a single machine. Secondly, The averaging operation in (3) becomes burdensome as \( n \) number loss function gradient must be computed. SGD techniques become more applicable in this context, since SGD update scheme only depends on one or few samples unbiasedly chosen at random.

It is also important to note that large-scale learning is bounded by time rather than by the number of data available [10]. Hence, we can have as many passes as possible over the training set within the allowable time.

In these era of Big data learning usually come with the additional requirement of distributed learning in which case, the data is distributed to multiple machine in mini-batches on which the learning is done with or without synchronization. With synchronization, each machine has to wait until every other machine has completed its local optimization before proceeding to the next iteration. This means that in a heterogenous setting where one machine may be faster than another the faster machine has to wait for the slower one. While in an asynchronous setting each machine does not have to wait for others but only use the weights learned on the other machines when they are available. Our work focuses on asynchronous technique for learning with Big data, even in an heterogenous setting. The strategy of our framework is such that in the end, even the objective function on the slower machine would converge at the capacity of the faster machine.

1.4 Thesis Outline and Contribution

Our contribution is an asynchronous algorithm for large-scale distributed heterogenous settings. Chapter one gives the background of the study, while chapter two presents the state of the art on single machine optimization and multiple machine optimization. In chapter three we describe
our asynchronous large-scale learning technique. In chapter four we give a summary of the entire study and discuss possible directions for future work on the subject.
Large-scale learning can be carried out on a single or multiple machines, depending on the size of the training dataset. And there are different considerations under the respective schemes. Some interesting works have been done on these with different algorithms being proposed. Some of the recent developments in these areas are presented below under single machine optimization and multiple machine optimization sections.

2.1 Single Machine Optimization

The original motivation for this research was the parallelization of computationally intensive tasks in order to benefit from multiple cores which are now readily available. In this context, a computationally intensive task can be parallelized such that each of the cores on a single machine can solve an objective function, while they all access the same pool of data and use shared memory for storing model vectors [25],[22].

Although SGD’s small memory footprint and rapid learning rates make the algorithm well suitable for data-intensive machine learning tasks [10], its inherent sequential nature makes it difficult to parallelize. Researchers have, however, taken advantage of the availability of web-scale datasets and develop several intuitive parallelization scheme for SGD. Since many large datasets are currently being pre-processed in a MapReduce-like parallel-processing framework, much of the recent work on parallel SGD have focused on MapReduce implementations. MapReduce, which is a powerful tool developed at Google for extracting information from huge logs and designed to ensure fault tolerance and to simplify the maintenance and
programming of large clusters of machine, is not ideal for online, numerically intensive data analysis as iterative computations are difficult to express in MapReduce, and the overhead to ensure fault tolerance can result in dismal throughput [24].

Notably, [25] proposed a lock-free approach, called HOGWILD, to parallelized SGD algorithm such that the shared variable is accessible to all the processors with the same priority, that is, no processor needs to wait while another is updating the shared variable. This approach might appear doomed to fail as processors could overwrite each other’s progress. However, when the data access is sparse, meaning that individual SGD steps only modify a small part of the decision variable, this approach turns out to be surprisingly successful. A processor uses compare and update method, and only update the index which corresponds to the index of the sample it used for its SGD computation. That is, each processor samples an instance indexed by \( e \in E \) uniformly at random, computes the gradient and then writes

\[
    w_v \leftarrow w_v - \gamma \nabla e_v (x_e), \text{ for each } v \in e
\]

This ensures that a processor does not overwrite the other and yields a fast algorithm with a convergence rate of \( \tilde{O}(1/t) \). This technique was applied in the single machine with multiple processors settings unlike our approach which is applicable in the distributed settings.

[22] was an improvement on HOGWILD with the addition of a bound to the age of updates, that is, no more than \( \tau \) updates to \( w \) occur between the time at which a processor reads \( w \) and uses it to evaluate one element of the gradient, and the time at which this processor makes its updates. This approach yields a linear convergence for strongly convex functions and sub-linear convergence \( O(1/t) \) for general convex functions. They also provably showed that the convergence of inconsistent read is slightly weaker.

While most techniques are based on SGD [26] presents a new analysis of Stochastic Dual Coordinate Ascent (SDCA) showing that this class of methods enjoy strong theoretical guarantees that are comparable or better than SGD. This analysis justifies the effectiveness of SDCA for practical applications.
2.2 Multiple Machine Optimization

Multiple machine optimization refers to a situation whereby more than one machine is used for the learning process. In this setting the training data is split into mini-batches which are distributed across the machines [14, 21, 13, 27, 31, 29] as a result of the training dataset being too large and not able to fit in the memory of a single machine.

Multiple machine parallelization becomes imminent when the size of data cannot fit into a single machine, which is typical in large-scale learning scenarios, for instance, in search engines and ad engines.

[32] algorithm provides a one-shot averaging of local sub-problems solved on each machine with mini-batches distributed across the machines. This method is asynchronous and SGD based. It is not clear, however, how the method will converge by the one-shot averaging.

A common practical solution is to employ mini-batch training which aggregates multiple examples at each iteration. However, the synchronization cost of this mini-batch approach is potentially still too large for large-scale experiments.

[13] proposed the use of the alternating direction method of multipliers (ADMM) for distributed learning, which learns the model weight from a maximization of the augmented Lagrangian of objective function. At each iteration local sub-problems are solved on local data on each machine and the results are aggregated to a global solution. Model vectors are moved across machines, the communication cost is kept low. [29] is an extension of ADMM for stochastic or sub-batch update. Convergence is however slowed down with more machines. ADMM also uses batch update over the mini-batches unlike in our algorithm which is both asynchronous and stochastic.

[31] proposed the distributed Newton method for regularized logistic regression. Their method is similar to ADMM but rather uses feature-wise split rather than instance-wise split used in ADMM. They showed that given $n$ number of samples and $d$ dimension of features, it is better to split the data instance-wise when $n \gg d$ and feature-wise when $d \gg n$. [31] showed that the problem addressed in [13] can be solved more efficiently by applying a truncated Newton algorithm for solving the problem in a way that eliminates the need for the computation of the Hessian which is the most expensive part of the procedure. The new approach was benchmarked against the state of the art, and shows a better performance.
[27] proposed a parallel batch update using mini-batches distributed across several machines. At each iteration every machine solves its local-subproblem and shares the learned weight which is averaged and distributed to all the machines for the next iteration. This method enjoys a linear rate of convergence which they provably show that it improves with data size. It is however a synchronous method as each iteration depends on the global average from the previous one.

[21] seek to solve the problem of decreased convergence rate with increase in mini-batch size. This problem is significant because large mini-batches reduce communication cost. For general convex objective functions, the convergence of SGD is $O(1/\sqrt{T})$ while for mini-batch SGD size $b$, the convergence is $O(1/\sqrt{bT} + 1/T)$. They therefore proposed an update strategy which adds a conservative penalty to the objective function and obtained a convergence rate of $O(1/\sqrt{bT})$. There approach however orthogonal to ours and could be used together complementarily.

[14] proposed an accelerated method to speedup the rate of convergence of SGD method when mini-batches are used, as in the case of large-scale settings. This method, though accelerates the convergence, is synchronous and is aimed at a different problem from what we intend to accomplish. [1] presents a very interesting package called Vowpal Wabbit (VW), which was reported being scalable and efficient in distributed environments. VW applies SGD method with adaptive learning rate in the beginning. Then, to get a faster convergence, VW weightedly averages the model as the initial solution for the subsequent quasi Newton method on the whole data. switches to parallel LBFGS for faster convergence. To support both numerical and string feature indices in the input data, VW uses feature hashing to have a fast feature lookup. That is, it applies feature hash function on the feature index to generate a new index for that feature value.

In the next section we shall discuss generally framework of gossip algorithms and give our proposed algorithm for heterogenous asynchronous large-scale learning with experimental results.
Asynchronous Large-scale Learning

Having studied the state of the art in chapter 2, in this chapter we will give a bit of introduction to gossip schemes for communication between participant machines in a distributed large-scale learning experiment and give the results from our experiment.

3.1 Examples of Gossip Schemes

We describe three standard gossip schemes so called pairwise, broadcast and dropout schemes. The reader may refer to [3] for a more complete picture and for more general gossip strategies, while [8] provides further analysis of gossip algorithms.

3.1.1 Pairwise gossip

This example can be found in [12], [5]. At time \( n \), two connected nodes, say \( i \) and \( j \), wake up, independently from the past. Nodes \( i \) and \( j \) compute the weighted average \( \theta_{n,i} = \theta_{n,j} = \frac{1}{2}(\theta_{n,i} + \theta_{n,j}) \) and for \( k \in \{i, j\} \) the nodes do not gossip. In this example, given the edge \( \{i, j\} \) wakes up, \( W_n \) is equal to \( I_N - (e_i - e_j)(e_i - e_j)^T \) where \( e_i \) denotes the \( i \)th vector of the canonical basis in \( \mathbb{R}^N \) and the matrices \( (W_n) \) are i.i.d. and doubly stochastic.

3.1.2 Broadcast gossip

This example is adapted from the broadcast scheme in [2]. At time \( n \), a node \( i \) wakes up at random with uniform probability and broadcasts its temporary update \( \tilde{\theta}_{n,i} \) to all its neighbors \( N_i \). Any neighbor \( j \) computes the weighted average \( \tilde{\theta}_{n,j} = \beta \tilde{\theta}_{n,i} + (1 - \beta) \tilde{\theta}_{n,j} \). On the other
hand, the nodes $k$ which do not belong to the neighborhood of $i$ (including $i$ itself) sets $\theta_{n,k} = \tilde{\theta}_{n,k}$. Note that, as opposed to the pairwise scheme, the transmitter node $i$ does not expect any feedback from its neighbors.

### 3.1.3 Network dropouts

In this simple example, the network is subjected from time to time to a dropout: consider any sequence of gossip matrices $W_n$ and put $W_n' = B_n W_n + (1 - B_n) I_N$ where $B_n$ is a sequence of i.i.d. Bernoulli random variables independent of the $W_n$. The network whose gossip matrices are the $W_n$ incurs a dropout at the moments where $B_n = 0$. At these moments, the nodes locally update their estimates and skip the gossip step.
3.2 Framework of Gossip Algorithms

We are given a network composed by $N$ nodes (sensors, robots, computing units,...). Node $i$ generates a $\mathbb{R}^d$-valued stochastic process $\theta_{n,i}$ through a two-step iterative algorithm: a local and a so-called gossip step. At iteration $n$

- **(Local step)** Node $i$ generates a temporary iterate $\tilde{\theta}_{n,i}$ given by:

$$\tilde{\theta}_{n,i} = \theta_{n-1,i} + \gamma_n Y_{n,i}$$

where $\gamma_n$ is a deterministic positive step size and where the $\mathbb{R}^d$-valued random process $Y_{n,i}$ represents the observations made by agent $i$.

- **(Gossip step)** Node $i$ is able to observe the values $\theta_{n,j}$ of some other $j$’s and computes the weighted average:

$$\theta_{n,i} = \frac{1}{N} \sum_{j=1}^{N} w_n(i,j) \tilde{\theta}_{n,j}$$

where the $w_n(i,j)$ are scalar non-negative random coefficients such that $\sum_j w_n(i,j) = 1$ for any $i$. The sequence of random matrices $W_n := [w_n(i,j)]$ represents the time-varying communication network between the nodes.

In [4], the index $n$ refers to time and performances of the algorithm are described with respect to $n$. Hence in an heterogeneous setting, the framework has to be slightly modified. We follow [4] and make the assumptions that follow.

**Assumption 1**

- $(W_n)$ is a sequence of $N \times N$ random matrices with non-negative elements such that:

  - $W_n$ is row stochastic: $W_n 1 = 1$
  - $E(W_n)$ is column stochastic: $1^T E(W_n) = 1^T$

- The sequence $(W_n)$ is identically distributed and the spectral norm $\rho$ of matrix $E(W^T (I_N - 11^T /N) W_n)$ satisfies $\rho < 1$. 
It is also assumed that the step-size sequence $\gamma_n$ in the stochastic approximation scheme satisfies the following conditions which are rather usual in the framework of stochastic approximation algorithms [20]:

**Assumption 2** The deterministic sequence $\gamma_n$ is positive and such that

$$\sum_n \gamma_n = \infty \text{ and } \sum_n \gamma_n^2 < \infty$$

### 3.3 Our Heterogenous Model

We assume that we are given $L$ machines $M_1, \ldots, M_L$ having different performances. The training sample is assumed to be divided in $L$ mini-batch of equal size. The inputs of an EOM are the weight $w$, the training sample $B_l$ and the machine $M_l$ whereas the outputs are the new weight $w$ and the duration of the EOM $t$. The duration $T_l$ of an EOM on machine $l$ is a random variable following an exponential distribution of parameter $\tau_l > 0$ which may be known or has to be estimated. Without loss of generality, one assumes $\tau_1 < \ldots < \tau_L$. These random variables are assumed to be pairwise independent and for a given machine each EOM is assumed to be independent of the others. The parameters $\tau_l$ may be different or not, reflecting the possible difference of performance of the different machines.

We denote $t_{i,l}$ the time of the $i$-th iteration of EOM on machine $l$ and $u_{i,l} = \sum_{i=0}^t t_{i,l}$. Each machine $l$ is connected with a local disk where some values $w_1^{(l)}, \ldots, w_L^{(l)}$ are stocked, which are referred as the local weights of machine $l$ at step $j$. We denote $s_m$ the boolean variable which equals NEW if the weight $w_m^{(l)}$ has been updated on machine $M_l$, OLD otherwise. When at the $j$-th step, an update occurs on machine $l$, the whole result $w_1^{(l)}, \ldots, w_{j,L}^{(l)}$, or a part of the result $w_{j,m}^{(l)}, \ldots, w_{j,m_p}^{(l)}$ with $p < L$ can be sent to the other disks of a neighborhood $N_l$ of machine $l$, and then erase totally or partially the previous results $w_{l,m}^{(l)}, \ldots, w_{l,m_p}^{(l)}$ for $l' \in N_l$. We then increase $j$. In what follows, we propose a simple asynchronous algorithm to take into account the heterogeneous setting for two machines.
3.3.1 An asynchronous algorithm for heterogeneous network with two machines whose characteristics are unknown

Denote $\sigma$ the permutation $\sigma = (1, 2)$. The algorithm works such that the training data is split into two equal sizes, as we assume we do not know the characteristics of the two machine, that is, we do not know which is faster or slower. The mini-batches are distributed to the two machines. We maintain a status file on each of the machines with the default values of OLD. On each of the machines we open a communication pipe through which new weights can be received from the other machine.

Each of the machines minimizes a logistic loss over the mini-batch in its memory with conjugate descent optimization, over an EOM that comprises of 10 iterations. After every EOM, each of the machines writes the learned weight through the pipe opened for communication on the other machine. The newly received weight is written to a file at the receiving machine and the status file is updated with the value NEW. When the machine checks and see a new status, it loads the newly received weight and swaps the value for its current weight, therefore, continue the next EOM using this weight as input. Anytime the machine reads the status and read the value OLD, it continues the EOM with it’s current weights.

Now, if we have two machines say $A$ and $B$, such that $A$ is 4 times faster than $B$, it means before $B$ finish computing an EOM, $A$ would have done 4. Here we assume that there is no other task running on $A$ that can lead to a memory swap which can make $A$ slower than usual, onces in a while. In this scenario, the next EOM that would be computed by $B$ will start from the point to which the weight had being minimized on $A$. It minimizes from this point for one EOM then send the new weights to $A$. $A$ would have minimized further from the initial point it received from $B$.

Also on machine $A$ every time it reduces the weight by 4 EOM, it is slowed down by a probably higher weights received from $B$, this cycle goes on and on until the maximum mean computation time is reached.

3.3.2 Conjugate Gradient

Conjugate Gradient (CG) is the is the most popular iterative method for solving large systems of linear equations [28, 19, 18]. It is an algorithm for the numerical solution of particular
Algorithm 1 Asynchronous mini-batch algorithm for heterogenous network with two machines whose characteristics are unknown

1: procedure HETEROGENOUSASYNC\text{SGD}(Z,L,T) \triangleright z = (x,y) is an observation

2: \hspace{1cm} L : number of machines, here \( L = 2 \)

3: \hspace{1cm} T : maximum mean computation time

4: \hspace{1cm} x \in X, X = (x_1, \ldots, x_n)

5: \hspace{1cm} y \in Y, Y = (y_1, \ldots, y_n)

6: \hspace{1cm} Initialize; \( \omega(0)_l \leftarrow 0, j_l \leftarrow 0, t(0)_l \leftarrow 0, \sigma(m)_l \leftarrow OLD \)

7: \hspace{1cm} Divide the examples into \( L \) batches \( B_1, \ldots, B_L \) of equal size and give each \( B_l \) to each machine \( M_l, l \in \{1, \ldots, L\} \)

8: \hspace{1cm} Iterate while \( \sum_{l=1}^L \sum_{i=0}^j t(i)_l \leq LT \) do:

9: \hspace{2cm} For each \( l = 1,2 \)

10: \hspace{3cm} if \( \sigma(l) = OLD \)

11: \hspace{4cm} (\omega(l)_l, t(l)_l) \leftarrow EOM(\omega(l)_l, B_l, M_l)

12: \hspace{4cm} (\omega(l)_{\sigma(l)}_l, t(l)_{\sigma(l)}_l) \leftarrow (\omega(l)_l, t(l)_l)

13: \hspace{4cm} \sigma(l)_l \leftarrow NEW

14: \hspace{2cm} else

15: \hspace{3cm} SWITCH (\omega(l)_l, \omega(l)_{\sigma(l)}_l)

16: \hspace{4cm} \sigma(l)_l \leftarrow NEW

17: \hspace{3cm} (\omega(l)_l, t(l)_l) \leftarrow EOM(\omega(l)_l, B_l, M_l)

18: \hspace{3cm} (\omega(l)_{\sigma(l)}_l, t(l)_{\sigma(l)}_l) \leftarrow (\omega(l)_l, t(l)_l)

19: \hspace{4cm} \sigma(l)_l \leftarrow NEW

20: \hspace{2cm} return \( \bar{w} \leftarrow \frac{1}{L} \sum_{i=1}^L w(i)LT \cdot \xi_{LT} \)
systems of linear equations, namely those whose matrix is symmetric and positive-definite. CG is often implemented as an iterative algorithm, applicable to sparse systems that are too large to be handled by a direct implementation or other direct methods such as the Cholesky decomposition. Large sparse systems often arise when numerically solving partial differential equations or optimization problems. CG is effective for systems of the form

$$Ax = b$$

where $x$ is an unknown vector, $b$ is a known vector, $A$ and is a known, square, symmetric, positive-definite (or positive-indefinite) matrix.

CG is an algorithm for finding the nearest local minimum of a function of $n$ variables which presupposes that the gradient of the function can be computed. It uses conjugate directions instead of the local gradient for going downhill. If the vicinity of the minimum has the shape of a long, narrow valley, the minimum is reached in far fewer steps than would be the case using the method of steepest descent [30].

Another reason why GG is the method of choice for large models because, in contrast to Newton-Raphson methods, where storage of a second-derivative matrix is required, is that only the previous gradients and directions have to be stored. However, to ensure that the directions are mutually conjugate, more complete line search optimizations must be performed along each direction. Since these line searches consume several function evaluations per search, the time per iteration may be longer for conjugate gradients than for steepest descents. This is more than compensated for by the more efficient convergence to the minimum achieved by conjugate gradients.

### 3.3.3 Logistic Loss

The binary logistic model is used to predict a binary response based on one or more features. The probabilities describing the possible outcomes of a single trial are modeled, as a function of the predictor variables, using a logistic function [6, 17].

The logistic loss function is given as

$$l(w) = log(1 + exp(-yp))$$
Figure 3.2: The weight reduces along the different axes at each iteration

where \( y \) is the label or class in a supervised learning training data and

\[
p = \langle w, x \rangle
\]

Examples of this kind of loss include; probability click on ad or not, is this email spam or not, is this review written by a customer or a robot

### 3.4 Experiment

The experiment was carried out in a heterogeneous settings with two machines in the LIG laboratory named **Machine A** and **Machine B**. The two machines have 4GB of random access memory (RAM) with similar configuration. However, a heterogeneous setting was created but running a mathematical computation on that takes actively engages **Machine B** and use up to 40 percent of RAM and CPU.

#### 3.4.1 Data

The data used for our experiment is DMOZ-5000 multiclass dataset which has been transformed into a binary class of +1 and -1. The features dimension has also been reduced to 10 and has a total size of 150GB. The original dataset was however split into smaller sizes of 250MB, 500MB and 1GB which were used for the experiment. The following table provides the details of experimental settings
### 3.4.2 Convergence

The goal of our experiment is to show that indeed the asynchronous gossip algorithm converges. We ran the conjugate descent algorithm for solving differentiable logistic loss minimization problem. We indeed observed a convergence of the method. Each epoch of minimization (EOM), is made of 10 iterations and it is after each epoch that the weight is communicated to the other machine. The vertical axis comprises of the loss function computed over the entire data used for the distributed experiment so that we can observe a global convergence. The horizontal axis has the time it takes to run the function plus communication cost, for the distributed experiment.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Hertog</th>
<th>Corona</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experiment 1</td>
<td>250MB</td>
<td>250MB</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>500MB</td>
<td>500MB</td>
</tr>
<tr>
<td>Experiment 4</td>
<td>1024MB</td>
<td>1024MB</td>
</tr>
</tbody>
</table>

### 3.4.3 Comparison to single machine optimization

We know that the problem is convex on single machine and therefore the algorithm converges. For our distributed experiment, we run our proposed asynchronous heterogenous distributed algorithm with a conjugate gradient optimization on each of the machines. Figure 3.3 shows the result and that our algorithm converges. We increased the maximum mean computation time with increase in data from 250MB to 500MB.

For the first experiment Figure 3.3(a), we have 250MB of mini-batch on each of the machines and it converges to a value between 0.45 and 0.46.

For the second experiment in Figure 3.3(b) we have 500MB on each of the two machines, which gives a total of 1GB of data. Once again there is convergence. It convergences at an objective value less than what we have for 250MB, which is probably due to the fact that we have more data. Also it too much more time to converge than (a) due to the difference in size, as it takes more time to load a larger training set to memory and the function computation time further increases with increase in the training set.

Figure 3.3 (c), whereby we have 1024MB on each of the machines also show convergence at a reduced value for the objective function as compared to (a) and (b). The system, however,
seem pertubated as values are swapped between the machines. We should also remember that the points shown in the graph is for every 10 epoch.

It is also important to note that the experimentation condition is not as perfect as it seems in theory as a machine perceived to be much slower could suddenly become much faster depending on whether some other takes taking up system resources have been completed or not while our distributed experiment is still ongoing.

3.5 Conclusion

The distributed asynchronous algorithm works and provide a strategy that can be used for large-scale learning with Big data, where the data cannot fit into the memory of a single machine. This strategy is also better than distributed techniques with synchronization as the waiting time for each iteration has been eliminated. This means if we are going to solve an objective function on a training data of say 1 terabyte, we can split the data into mini-batches and solve local objectives on each machine while we swap the learned weight after each epoch of minimization until the maximum mean computation time is exceeded. Our algorithm eliminates time that would have been wasted due to synchronization during which more iterations can be computed.

Beyond Big data, it provides a heterogeneous strategy by which even a small machine or device such as handheld devices, microcontrollers and so on can be used in large-scale learning and benefit from the speed of a much faster node to learn.
Figure 3.3: Convergence of asynchronous distributed learning with two machines.

(a) 250MB

(b) 500MB

(c) 1024MB
Conclusion and Future Work

Now we have given the motivation for distributed large-scale learning in chapter 1, which is because of the memory constraint when carrying out large-scale learning in which the training data does not fit into the memory of a single machine. In chapter 2 we reviewed the state of the art in large-scale learning and discussed our proposed heterogenous algorithm with experimental results in chapter 3. We are therefore going to give a conclusion on the study and state possible directions for future work on this subject.

4.1 Conclusion

Large-scale learning has become a commonplace in this era of web data in which we have massively large dataset that can not fit the memory of a single machine to take advantage of the traditional optimization techniques. The dataset has to be distributed or spread to the number of machines that have enough memory to store the data for the computations involved in the learning process.

In this thesis we seek to develop a distributed large-scale learning algorithm without the usual synchronization. The convergence of distributed algorithms with synchronization has been well studied as shown in the state of the art section, however, much has not been done in the area of asynchronous algorithm.

Typically in distributed settings, each of the machines have to synchronize the weight which it has learned with its local objective function and wait for every other machine in the distributed experiment to do the same. In the end the average weight is used in the next iteration at each
of the machines. This strategy does not put into consideration that each of the participating machine does not necessarily have the same configuration. The cost each iteration comprises of the computation cost, the synchronization cost and the waiting cost, which make the distributed learning more expensive.

Our idea is that in an heterogeneous setting, where the machines do not have the same capability, the waiting time can be eliminated by applying an asynchronous technique in such a way that each participant machine continues the next iteration even whe the other machine have not finish their computations. When a new weight is available, the other machine swaps its weight with this an continues the next iteration from there. The idea is such that if the faster machine has 4 times the capability of the slower one, for example, the slower machine will minimize at the rate of the faster one. Also, swapping the weight of the slower machine on the faster one would slow down the faster machine a little so that it does not decrease too fast and jump below the global minimum. This continues until the algorithm converges to a global minimum.

Our experiment proves this to be true as can be observed in the graphs shown in chapter three, and also when compared to the graph of the single machine optimization.

We limited the scope of our experiment to two machines and the algorithm converges to a global minimum in all the scenarios investigated. We also believe the same result is applicable when we have more than two machines in the distributed setting.

This result will be very significant in domains such as ad placement and search engines where we usually have Big data.

### 4.2 Future Work

Now that we have seen that the algorithm converges for two machine we would like to carry out further experiments to study the behaviour with many more machines and possibly provide theoretical analysis of the convergence observed. In future works we will also investigate into strategies for separating communication from computation in our distributed algorithm and see how it compares to single machine optimization. In this case separate threads would perform both the computation and communication such that they can be done concurrently without one task having to necessarily wait for the other.
We would also like to investigate the impact of the asynchronous algorithm on the learned classifier in terms of prediction accuracy. In this case we will compare the single machine, distributed techniques with synchronization and our asynchronous technique for heterogeneous settings.

Another possibly interesting scenario is when each of the machine learns with a different objective function. We would like to know if some additional benefits can be derived in this instance.
Bibliography


[16] DigitalMind. List of the key terms and concepts of ml. 2015.


Algorithm 2 Stochastic Gradient Descent (SGD)

1: procedure SGD($X, Y, \gamma, w^{(0)}$) 
2:     $x \in X, X = (x_1, \ldots, x_n)$
3:     $y \in Y, Y = (y_1, \ldots, y_n)$
4:     $z =$observation ($x,y$) for $x \in X$ and $y \in Y$
5:     Initialize; $t = 0, \gamma > 0$ \hspace{1cm} $\triangleright \gamma$ is the learning rate
6:     Randomly shuffle examples in the training set
7:     Repeat until an approximate minimum is obtained
8:     Iterate: for $t := 1$ to $n$ do:
9:         $w^{(t)} := w^{(t-1)} - \gamma^{(t)} \nabla e^{(t)}(f_{w^{(t-1)}}, z^{(t)})$
10:     end
11: return $w$

Algorithm 3 Mini-Batch SGD ([14])

1: procedure MiniSGD($X, Y, \gamma, b$) \hspace{1cm} $\triangleright b$ is mini-batch size
2:     $x \in X, X = (x_1, \ldots, x_n)$
3:     $y \in Y, Y = (y_1, \ldots, y_n)$
4:     $z \in Z, Z = (z_1, \ldots, z_n)$ \hspace{1cm} $\triangleright z$ is observation ($x,y$)
5:     Initialize; $w^{(1)} = 0$
6:     Iterate: for $i = 1$ to $k = n/b$ do:
7:         Compute $\nabla e^{(i)}(f_{w^{(i)}}, z^{(i)}) = \frac{1}{b} \sum_{t=b(i-1)+1}^{bi} \nabla e(f_{w^{(i)}}, z^{(i)})$
8:         $w^{(i+1)} := w^{(i)} - \gamma \nabla e^{(i)}(f_{w^{(i)}}, z^{(i)})$
9:         $w^{(i+1)} = F_w(w^{(i+1)})$
10:     end
11: return $\bar{w} = \frac{1}{k} \sum_{i=1}^{k} w^{(i)}$
Algorithm 4 Distributed Approximate Newton-type method ([27])

1: procedure DANE($X, Y, \gamma, \mu$) \Comment{$\mu$ is regularization parameter}
2: \quad $z \in Z, Z = (z_1, \ldots, z_n)$ \Comment{$z$ is observation (x,y)}
3: \quad Initialize; $w^{(0)} = 0, \gamma > 0, \mu > 0$ \Comment{$m$ is number of machines}
4: \quad Iterate: for $t=1,2,\ldots$
5: \quad \quad Compute $\nabla e(f_{w^{(t-1)}}) = \frac{1}{m} \sum_{i=1}^{m} \nabla e_i(f_{w^{(t-1)}}, z^{(t)})$
6: \quad \quad Distribute to all machines $\nabla e(f_{w^{(t-1)}})$
7: \quad \quad Iterate: for $i = 1$ to $m$ do:
8: \quad \quad \quad $w^{(t)}_i = \arg\min_w \left[ e_i(f_w, z) - (\nabla e_i(f_{w^{(t-1)}}, z^{(t)}) - \gamma \nabla e(f_{w^{(t-1)}}))^T w + \frac{\mu}{2} \| w - w^{(t-1)} \|_2^2 \right]$
9: \quad \quad \quad end
10: \quad \quad Compute $w^{(t)} = \frac{1}{m} \sum_{i=1}^{m} w^{(t)}_i$ and distribute to all machines
11: \quad \quad end
12: \quad return $w$

Algorithm 5 One-shot Averaging Parallel SGD ([32])

1: procedure ASYNC SGD($X, Y, \gamma, m, w^{(0)}$) \Comment{$m$ is the number of machines}
2: \quad $x \in X, X = (x_1, \ldots, x_n)$
3: \quad $y \in Y, Y = (y_1, \ldots, y_n)$
4: \quad Initialize; $t = 0$
5: \quad Iterate for all $i \in \{1, \ldots, m\}$ parallel do:
6: \quad \quad $v^{(i)} = SGD(X, Y, \gamma, w^{(0)})$
7: \quad \quad end
8: \quad Aggregate from all machines $v = \frac{1}{m} \sum_{i=1}^{k} v_i$
9: \quad return $v$
Algorithm 6 Synchronous Parallel SGD ([32])

1: procedure SYNC\_SGD($X, Y, \gamma, m$)
2: \hspace{1cm} $x \in X, X = (x_1, \ldots, x_n)$
3: \hspace{1cm} $y \in Y, Y = (y_1, \ldots, y_n)$
4: \hspace{1cm} $z \in Z, Z = (z_1, \ldots, z_m)$ \hspace{0.5cm} ▷ $z$ is observation $(x, y)$
5: \hspace{1cm} $T = n/m$
6: Randomly partition the examples to each machine
7: Iterate: for $i \in \{1, \ldots, m\}$ parallel do:
8: \hspace{1cm} Randomly shuffle the data on machine $i$
9: \hspace{1cm} Initialize; $t = 0, w(i, 0) = 0$
10: Iterate: for all $t \in \{1, \ldots, T\}$: do \hspace{0.5cm} ▷ $m$ is number of machines
11: \hspace{2cm} Get the $t$th example on the $i$th machine (this machine), $z(i,t)$
12: \hspace{2cm} $w(i,t) = w(i,t-1) - \gamma \nabla e(f_{w(i,t-1), z(i,t)})$
13: \hspace{1cm} end
14: end
15: Aggregate from all machines $v = \frac{1}{m} \sum_{i=1}^{m} w(i,t)$
16: return $v$
Algorithm 7 Stochastic Dual Coordinate Ascent (SDCA): [26]

1: procedure SDCA($X, Y, \alpha^{(0)}$)
2: \hspace{1em} $x \in X, X = (x_1, \ldots, x_n)$
3: \hspace{1em} $y \in Y, Y = (y_1, \ldots, y_n)$
4: \hspace{1em} Let $w^{(0)} = w(\alpha^{(0)})$
5: \hspace{1em} Iterate: for $t=1,2,\ldots,T$ do:
6: \hspace{2em} Randomly pick $i$
7: \hspace{2em} Compute:
8: \hspace{3em} $\max_{\Delta \alpha} - e_i^*(f(\alpha^{(t-1)} - \Delta \alpha^{(i)}), x^{(i)}, y^{(i)}) - \lambda g^*(\frac{1}{\lambda n} x^{(i)} \alpha^{(i)})$
9: \hspace{3em} $\alpha^{(i)} \leftarrow \alpha^{(t-1)} + \Delta \alpha^{(i)}$
10: \hspace{1em} Output (Averaging option)
11: \hspace{2em} Let $\bar{\alpha} = \frac{1}{T - T_0} \sum_{t=T_0+1}^{T} \alpha^{(t-1)}$
12: \hspace{2em} Let $\bar{w} = w(\bar{\alpha}) = \frac{1}{T - T_0} \sum_{t=T_0+1}^{T} w^{(t-1)}$
13: \hspace{2em} return $\bar{w}$
14: 
15: Output (Random option)
16: \hspace{2em} Let $\bar{\alpha} = \alpha^{(t)}$ and $\bar{w} = w^{(t)}$ for some random $t \in T_0 + 1, \ldots, T$
17: \hspace{2em} return $\bar{w}$