Indian Institute of Technology Bombay

B.Tech Project

Distributed Learning & Inference

Author:
Nilesh Kulkarni

Supervisor:
Prof. Ganesh Ramakrishnan

A thesis submitted in fulfillment of the requirements
for Honors in Computer Science

Department of Computer Science and Engineering
Indian Institute of Technology Bombay

November 2014

Acknowledgements

I would like to take this opportunity to express my deepest gratitude towards my guide Prof. Ganesh Ramakrishnan for his motivating support, guidance and valuable suggestions. I would also like to thank Ajay Nagesh for helping me throughout the project and for interactive discussions

Abstract

Classification has demonstrated success in many areas of applications. Modern algorithms for linear classification can train reasonably good models while going through the data in only tens of rounds. However, large data often does not fit in the memory of a single machine, which makes the bottleneck in large-scale learning the disk I/O, not the CPU. Distributed optimization algorithms are highly attractive for solving big data problems. In particular, many machine learning problems can be formulated as the global consensus optimization problem, which can then be solved in a distributed manner by the alternating direction method of multipliers (ADMM) algorithm. Distributed optimization on structured Data has either been applied to Learning or Inference, this thesis tries to explore the possibilities of learning and inferring for structured data sets in a distributed fashion.

Contents

Chapter 1

Distributed Algorithms

Parallel and Distributed computing systems offer the promise of a quantum leap in computing power that can be bought to bear on many important problems. Work of parallel and distributed computation spans several broad areas, such as the design of parallel machines, parallel programming languages, parallel algorithm development and several application related issues. We consider the area of distributed algorithms which is of utmost important towards design better and faster algorithms for Machine Learning. The choice of algorithms is highly motivated by the large scale of problems for which it is essential to harness power of computation while keeping the communication overhead and delays within tolerable limits. There is a very interesting book by Bertesekas [1] which does a detailed analysis of parallel systems

1.1 Parallel vs Distributed Systems

Parallel Computing Systems are the systems consists of various processors located within a small distance. Their main purpose is to jointly solve a heavy computational problem, the communication between them are reliable and predictable. Whereas when we consider Distributed Systems the computing units might be far from each other and could be connected to each other over the Internet, the communication overhead between them is much more significant than that in parallel systems. Still there is no clear line separating distributed systems from parallel system, several algorithmic issues are similar and we will use the terms interchangeably

1.2 Distinguishing between Different Parallel Systems

There are several parameters that can be used to distinguish between two parallel systems

- **Type and Number of Processors**: Some systems might have thousands of processors within a small distance from each other and these systems are massively parallel. These systems are promising in terms of computing power and might solve huge computation needs. Contrary to this we have a parallel systems with small number of processors (order of 10's) where in processors are loosely coupled and each processor might be performing a different task in parallel
- **Presence and absence of global control mechanism**: One extreme is a global control mechanism manages all the processors loads the instructions and the data is allocated to a processor. An important classification is Single Instruction Multiple Data(SIMD) and Multiple Instruction Multiple Data(MIMD)
- **Asynchronous and Synchronous operations**: This refers to presence and absence of a global clock. SIMD system are synchronous.
- **Processor Interconnection**: Network Topology is an important aspect in parallel computation. This dictates the ways through which different processors can communicate. Generally there are two extreme paradigms of sharing data a) *shared memory* b) *message passing*

While designing parallel systems a designer may use a varied combination of the above classification to suit his problem. The network topology is most important in both parallel and distributed. It is usually pre-determined in distributed systems but is at designers disposal when using parallel systems and more uniform.

1.3 Models & Complexity Measures

Models for parallel and distributed computation are based on computing the power of the processors and interprocessor information transfer mechanisms. We have to consider both aspects while looking at a solution to a problem.

1.3.1 Parallel Algorithms using (DAG)

DAG's are directed graphs with no positive cycles,i.e. no cycles consisting of exclusively forward edges. Let G=(N,A) be a DAG where N = {1,2,. . . ,| N |} represents the set of

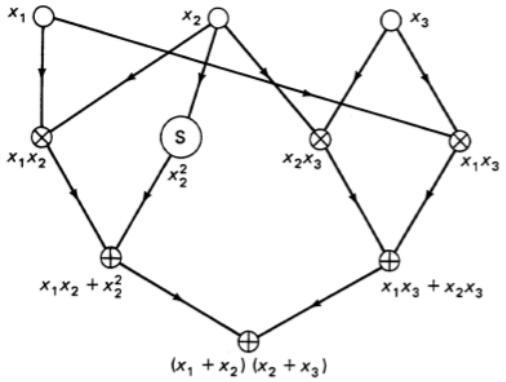

FIGURE 1.1: DAG representing a computation of $(x_1 + x_2)(x_2 + x_3)$ [1]

Nodes, and A is the set of directed edges. In particular a directed edge (i,j) ∈ A indicates that result obtained by node i are required for computation at node j.In such a case we say that node i is called j. Each node in the DAG represent an elementary operation (unit time to execute). DAG is a partial representation of a Parallel Algorithm, it specifies the kind of operations needed and imposes constraints on order of precedence of operations. To determine the completely parallel algorithm we have to specify which processor computes performs what computation on the DAG. For now, we assume we have pool p of processor for any node i ∈ N, not being an input node we assign a Processor P_i to compute it. We also assume that t_i denotes the time instance when computation for node i started We impose some constraints on assigning P_is

- $P_i == P_j \implies t_i \neq t_j$
- $(i, j) \in A \implies t_i < t_j$

Once we compute P_i and t_i for all nodes in the DAG then we have a tuple $(i, t_i, P_i | i \in N)$ scheduled graph. Complexity of a parallel algorithm is measured in terms of

- The number of processors
- The time until the algorithm terminates
- The amount of messages passed between nodes

Consider a schedule S given bt $\{$(i,$P_i, t_i | i \in N\}$ for a DAG that uses p processors. The time spent by this schedule to execute the algorithm is $T_p = max_{i \in N} t_i$. We define $T_\infty = \min_{p \geq 1} T_p$, we note that T_p is a non decreasing function of p and $\exists p \; s.t \; T_{p*} = T_\infty$. We call p* the optimal number processors p* of for the schedule.

1.3.2 Some Important Results: Properties of T_p

Suppose that for some Output node i, there exists a positive path from an input node to i, we also assume that the in-degree of every node is atmost 2 then

$$T_\infty \geq log(n) \tag{1.1}$$

Intuition, is as follows we can now look at the DAG as a binary tree up side down, and if every input node has a in-degree 2 then we can construct a DAG with minimum execution time which is $log(n)$.

1.3.3 Finding Optimal DAG

There is no unique DAG for a given algorithm, there can be multiple ways to compute the same expression within similar complexity. But to every designer the most interesting DAG would be the one that minimized the T_p where is p is the number of available processors. The value of T_{p*} is the measure of complexity for the problem where as T_p is measure of complexity of an algorithm. An explicit evaluation of T_{p*} might not be easy. However there are problem where designing DAGs come into a constant factor of optimal

1.3.4 Speed up & Efficiency

When we choose a particular DAG we assume we have chosen a strategy to compute the solution to our problem. To compute the efficiency we need to consider the throughput of every processor. So we define speed up for p processors as

$$S_p(n) = \frac{T*(n)}{T_p(n)} \tag{1.2}$$

where T*(n) is the time for the optimal serial algorithm with n nodes, maximum speed up for a problem cannot exceed p. Now we can define efficiency of the algorithm, which captures the fact how busy the processors.

$$E_p(n) = \frac{S_p(n)}{p} \tag{1.3}$$

Ideally we want $S_p = p$ and $E_p = 1$, which represent the best case scenario which not possible practically. A more realistic objective is to stay away from converging to zero as n & p increases

1.4 Parallelization of Iterative Methods

Iterative methods remain one of the most important areas of application that needs to be discussed as most of the non-closed form solution are based on trying to converge to the best solution via repetitively taking similar step by looking locally or globally. A general form of iterative methods is represented at

$$x(t+1) = f(x(t)) \tag{1.4}$$

Learning with Structured Outputs where x is a vector having n components. Now updating the update rule can be written as

$$x_i(t+1) = f_i(x_1(t), \ldots, x_n(t)) \tag{1.5}$$

Now the most straight forward parallelization of the above problem is to give every processor the task to update one component of the vector. So, all processors need to communicate the values of their components after computation to all other processor and the communication overhead is defined the dependency graph. Sometime we may employ coarse-parallelization schemes for iterations of $f(x)$ According, to which a processor could update more than one dimension for x if the communication costs out-way the computation. Such kind of parallelization is called block parallelization.

1.4.1 Gauss-Seidel Iterations

The iterations in which we consider to update components x while considering latest value of the earlier computations; they try to incorporate the newest available information while iterating.

$$x_i(t+1) = f_i(x_1(t+1), \ldots, x_{i-1}(t+1), x_i(t), \ldots, x_n(t)) \tag{1.6}$$

A gauss-seidel iteration hence has a strong dependency over the variable updates and may not be always parallelizable for instance every update f_i is dependent on all the variables(components of x). Contrary to which the dependency graph might be very sparse and local wherein we could employ a lot of parallelization while performing the updates. Employing parallelization comes from fact that the order of update might be changed and the algorithm still achieves the same.

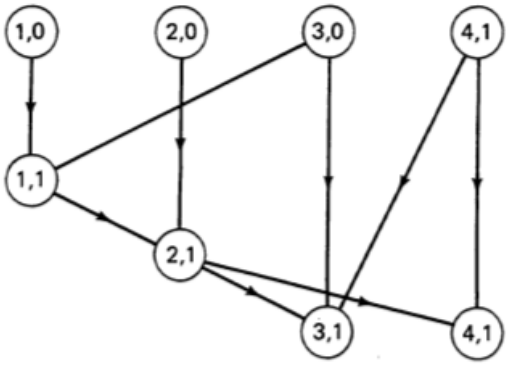

FIGURE 1.2: DAG representing Gauss Seidel Iteration for equations 1.7

$$\begin{aligned} x_1(t+1) &= f_1(x_1(t), x_3(t)), \\ x_2(t+1) &= f_2(x_1(t+1), x_2(t)), \\ x_3(t+1) &= f_3(x_2(t+1), x_3(t), x_4(t)), \\ x_4(t+1) &= f_4(x_2(t+1), x_4(t)), \end{aligned} \tag{1.7}$$

There are four updates preformed but the depth of the DAG is 3, which implies that updates of x_4 and x_3 can be performed together. In [1], a graph theoretical formulation this problem is proposed as follows:
Given a dependency G=(A,N), a coloring of G, using K colors, defined as a mapping $h : N \to \{1, \ldots, K\}$ *that assigns a color* $k = h(i)$ *for each node* $i \in N$. The following are equivalent:

- There exists and ordering of variables such that a sweep of corresponding Gauss-Seidel iterations can be preformed in K steps
- There exists a coloring of the dependency graph that use s K colors and with the property that there exists no positive cycle with all nodes on the cycle having the same color

1.5 Organization of Report

Chapter 2 contains an Introduction to Structured Output Prediction which are supervised machine learning techniques that involve predicting structured objects. Chapter 3 contains a detailed discussion about our of problem of Distant Supervision for Relation Entity Pair Extraction. Chapter 4 contains discussion on using ADMM's as tool for distributed Optimization. In Chapter 5 using all the literature survey done in previous chapters we present a solution to our problem.

Chapter 2

Learning Structured Data

Wikipedia says, **Machine learning** is a scientific discipline that deals with the construction and study of algorithms that can learn from data. Such algorithms operate by building a model based on inputs and using that to make predictions or decisions, rather than following only explicitly rule based instructions.
Data is structured if it consists of several parts, and not only the parts themselves contain information, but also the way in which the parts belong together. Following are the examples of Structured Data

- Automatic Translation and Sentence Parsing
- Speech Processing Automatic Transcription
- Text-to-Speech
- Planning (output: sequence of actions)

2.1 Different Kinds of Learning

Any general Learning Problem is defined as trying to estimate the function f

$$f : X \to Y \tag{2.1}$$

where X is the input space and Y is the output space.

- Scalar: X is R^d, Y is R
- Boolean: X is R^d, Y is $\{0, 1\}$

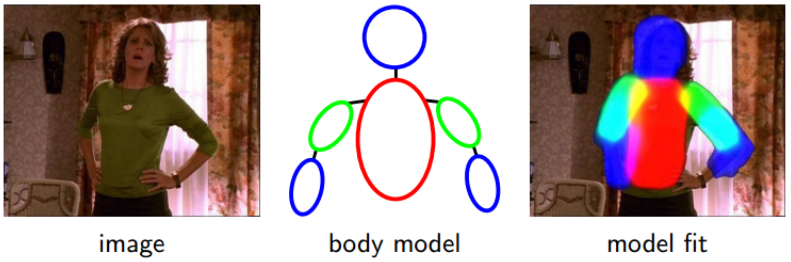

FIGURE 2.1: This is an example of structured data, task of Human Pose Prediction [2]

- Structured Scalar: Domain of X is complex objects, trick is to extract features and convert X to R^d. Y is R
- Structured Output: Domain of X is complex objects. Y's are also complex objects.

In the first three cases case of General Machine Learning setup is as follows which usually involves normal machine learning tasks like classification or regression. Now for any f 2.1, where input X can be any be any kind of objects likes images, sentences, audio data etc.. Output y is a real number involving tasks like classification and regression. There are many ways to construct a f that satisfies above properties like using svms , decision trees, neural networks etc

Now in the last case of **Structured Output** Learning you are task is to predict an object instead of a real number.

$$f : X \to Y \tag{2.2}$$

In this case inputs can be $x \in X$ any objects similar to normal learning, the output $y \in Y$ is a space of object that capture structure in the data which might be very complex. The task of a structured machine learning algorithm or a method is to construct such a f.

Defining auxiliary functions, g: X x $Y \to R$, using joint features of $\phi(x, y)$

$$e.g.\ g(x, y) = \langle w, \phi(x, y) \rangle + b$$

or, using a joint kernel density function $k((x, y), (x', y'))$:

$$e.g.\ g(x, y) = \sum_{j=1}^{N} \alpha_j k((x, y), (x', y'))$$

Now obtain a function $f : X \to Y$ by maximization:

$$f(x) = \arg\max_{y \in Y} g(x, y)$$

2.2 Structured SVMS

SVM^{struct} [3] is a Support Vector Machine (SVM) algorithm for predicting multivariate or structured outputs. It performs supervised learning by approximating a mapping $h : X \to Y$ using labeled training examples $(x_1, y_1), \ldots, (x_n, y_n)$. Unlike regular SVMs, however, which consider only univariate predictions like in classification and regression, SVM_{struct} can predict complex objects y like trees, sequences, or sets. Examples of problems with complex outputs are natural language parsing, sequence alignment in protein homology detection, and markov models for part-of-speech tagging. The SVMstruct algorithm can also be used for linear-time training of binary and multi-class SVMs under the linear kernel. Joachims [3] published a classic paper of Structured SVMs talking in detail about applying svms to structured multivariate data. Important conclusions and results relevant to us are described

The important conclusion from the paper are formulation of a Support Vector Machine for supervised Learning with structured and interdependent outputs. The approach is based on formulation of a joint feature map, which covers a large class of interesting problem described earlier like NLP, Speech Processing etc. The paper also proposes an Cutting Plane algorithm [4], which helps is solving a QP with a large(typically exponential or infinite) number of constraints. This cutting plane approach reduces the problem to small QP's which can be easily optimized by existing optimizers.

Finally the svm equations that are to be optimized are:

$$\min_{w} \frac{1}{2}||w||_2^2 + C\Big[\sum_{i=1} N \max_{\tilde{y_i}} \big\{\delta(y_i, \tilde{y_i}) + w \cdot \phi(x_i, \tilde{y_i})\big\} - \sum_{i=1}^{N} w \cdot \phi(x_i, y_i)\Big] \tag{2.3}$$

where δ is a decomposable loss function over training instances

There is a version of Structural SVM's which incorporates latent variables **h** [5] into the problem and one suggested training objective is as follows

$$\min_{w} \frac{1}{2}||w||_2^2 + C\Big[\sum_{i=1} N \max_{\tilde{y_i}, h} \big\{\delta(y_i, \tilde{y_i}) + w \cdot \phi(x_i, h, \tilde{y_i})\big\} - \sum_{i=1}^{N} \max_{h} w \cdot \phi(x_i, h, y_i)\Big] \tag{2.4}$$

Later in the next Chapter 3 we will use these training objectives to realize our learning.

Chapter 3

Distant Supervision for Relation Entity Extraction

Consider the distant supervision problem setup for information extraction. A training data is given $\{(x_i, y_i)\}_{i=1}^{N}$, where for the i^{th} entity-pair we are given a collection of mentions x_i whose relation label is hidden h_i We are aiming to learn a parameter vector w by which we can predict relation labels for a new entity-pair based on its collection of mentions x

$$y = \arg\max_{\tilde{y}} \max_{h} w \cdot \phi(x, h, \tilde{y}) \tag{3.1}$$

where ϕ is the feature vector defined according to a Markov Random Field detailing interdependencies between between **x** and **y** through **h**.In our problem we are using a non-linear loss function called the "F-Score" which is $\frac{2 * FN * FP}{FN * FP}$, where FN is count for False Negatives and FP is count for False Positives.
We start by reviewing the Structural SVM problem we have mentioned earlier, but the loss function is decomposible over the training set. We formulate equation 2.3 to get a training objective for our problem.

$$\begin{aligned} \min_{w} \frac{1}{2}||w||_2^2 + C \max_{\tilde{y_1},\ldots,\tilde{y_N}} \{\Delta(y_1, \ldots, y_N, \tilde{y_1}, \ldots, \tilde{y_N}) + w \cdot \sum_{i=1}^{N} \phi(x_i, \tilde{y_i})\} \\ - Cw \cdot \sum_{i=1}^{N} \phi(x_i, y_i) \end{aligned} \tag{3.2}$$

Using the equation 2.4 we can handle latent variables and non-decomposible loss then the svm objective looks as follows:

$$\min_{w} \frac{1}{2}||w||_2^2 + C \max_{\tilde{y_1},\ldots,\tilde{y_N}} \left\{ \Delta(y_1,\ldots,y_N,\tilde{y_1},\ldots,\tilde{y_N}) + \sum_{i=1}^{N} \max_{h} w \cdot \phi(x_i,h,\tilde{y_i}) \right\} - C \sum_{i=1}^{N} w \cdot \max_{h} \phi(x_i,h,\tilde{y_i}) \tag{3.3}$$

Firstly, to be able to optimize the above training objective we need to solve the following so-called loss-augmented optimization problem

$$\max_{\tilde{y_1},\ldots,\tilde{y_N}} \left\{ \Delta(y_1,\ldots,y_N,\tilde{y_1},\ldots,\tilde{y_N}) + \sum_{i=1}^{N} \max_{h} w \cdot \phi(x_i,h,\tilde{y_i}) \right\} \tag{3.4}$$

Optimizing the loss-augmented function takes places through maximization of the individual functions with some constraints. The problem is reformulated as follows

$$\max'_{\tilde{y_1},\ldots,\tilde{y_N},\tilde{y_1}',\ldots,\tilde{y_N}} \left\{ \Delta(y_1,\ldots,y_N,\tilde{y_1},\ldots,\tilde{y_N}) + \sum_{i=1}^{N} \max_{h} w \cdot \phi(x_i,h,\tilde{y_i'}) \right\} \quad \text{subject to } \tilde{y_i} = \tilde{y_i}' \; \forall i \in \{1,\ldots,N\} \tag{3.5}$$

This problem can be solved using method of Lagrange multipliers [6] as described in Chapter 4. We use the Lagrange multipliers to put the constraints in the objective itself and then we try to iteratively update the primal and dual variable for the problem. Here comes in the idea to use ADMM so as to form the solution faster and use the ideas in distributed computing. We explored the ideas and applied ADMM to the given problem. We did not achieve a very great speed up in using this method partly because the complete data was used every time and we could only distribute our computation over 2 nodes. Later in Chapter 5 we propose a possible solution the problem which might work.

Chapter 4

Alternating Direction Method of Multipliers

The alternating direction method of multipliers (ADMM) is an algorithm that solves convex optimization problems by breaking them into smaller pieces, each of which are then easier to handle. It has recently found wide application in a number of areas. ADMM is an variant to Augmented Lagrangian methods which helps easy formulation of distributed algorithms based on consensus, which is achieved by partial updates. [7]

4.1 Augmented Lagrangian

Augmented Lagrangian methods are a certain class of algorithms for solving constrained optimization problems. They have similarities to penalty methods in that they replace a constrained optimization problem by a series of unconstrained problems; the difference is that the augmented Lagrangian method adds an additional term to the unconstrained objective. This additional term is designed to mimic a Lagrange multiplier. The augmented Lagrangian is not the same as the method of Lagrange multipliers.
Let us now consider a constrained optimization problem:

$$\min_{x} f(x) \tag{4.1}$$

subject to $c_i(x) = 0 \ \forall$ i $\in$ I According to method of multipliers consider,

$$\phi_k(x) = f(x) + \frac{\rho}{2} \sum_{i \in I}^{N} c_i(x) - \sum_{i \in I} \lambda_i c_i(x) \tag{4.2}$$

So, at each iteration k we update $\lambda's$ and x's following the following the update rule

$$x = \arg\min_x \phi_k(x), \tag{4.3}$$
$$\lambda_i = \lambda_i - \mu * c_i(x_k) \tag{4.4}$$

The Variable λ is and estimate of the Lagrangian Multiplier, and accuracy of the estimate improves every step of optimization. We iterate until x converges to a local minimum.

4.2 Alternating Direction of Multipliers

ADMM is an algorithm that is intended to blend the decomposability of dual ascent with the superior convergence properties of the method of multipliers. The algorithm solves problems in the form

$$\min_{x,z} f(x) + g(z), \tag{4.5}$$

subject to $Ax + Bz = c$ with variables x $\in R^n$ and z $\in R^m$, where A is p x n matrix and B is p x m and c $\in R^p$. We will assume f and g are convex functions. The optimum values p* = inf{f(x)+ g(z) | Ax+Bz=c}. Now as in the method of multipliers we form the augmented Lagrangian

$$L_\rho(x, y, z) = f(x) + g(z) + y^T(Ax + Bz - c) + \frac{\rho}{2}||(Ax + Bz - c)||_2^2 \tag{4.6}$$

The updates for the primal [x,z] and lagrange(dual)[y] variables are

$$x^{k+1} = \arg\min_x L_\rho(x, y^k, z^k), \tag{4.7}$$
$$z^{k+1} = \arg\min_z L_\rho(x^{k+1}, y^k, z), \tag{4.8}$$
$$y^{k+1} = y^k - \rho(Ax^{k+1} + Bz^{k+1} - c) \tag{4.9}$$

where ρ ¿ 0. The algorithm is very similar to dual ascent and the method of multipliers: it consists of an x-minimization followed by a z-minimization and then an update to the dual variable y. The form of updates for this optimization with method of multipliers are

$$(x^{k+1}, z^{k+1}) = \arg\min_{x,z} L_\rho(x, y^k, z), \tag{4.10}$$
$$y^{k+1} = y^k - \rho(Ax^{k+1} + Bz^{k+1} - c) \tag{4.11}$$

So, the only difference lies in the form of updates for x& z. Method of Multipliers does a joint minimization over the variable x & z while the ADMM updates them separately and that is why the name *Alternating Direction*.

4.2.1 Scaled Dual representation of ADMM

ADMM can be written in a slightly different form, which is often more convenient. We define r = Ax+By-c

$$y^T r + \frac{\rho}{2}||r||_2^2 = (\frac{\rho}{2})||r + \frac{1}{\rho}y||_2^2 - \frac{1}{2*\rho}||y||_2^2 = (\frac{\rho}{2})||r + u||_2^2 - \frac{1}{2*\rho}||u||_2^2 \tag{4.12}$$

where $u = \frac{1}{\rho}y$ is the scaled dual variable. Using the scaled dual variable, now we can express ADMM as

$$x^{k+1} = \arg\min_x f(x) + \frac{\rho}{2}||Ax + Bz^k - c + u^k||_2^2, \tag{4.13}$$

$$z^{k+1} = \arg\min_z g(z) + \frac{\rho}{2}||Ax^k + Bz - c + u^k||_2^2, \tag{4.14}$$

$$u^{k+1} = u^k + (Ax^{k+1} + Bz^{k+1} - c) \tag{4.15}$$

We see that $u^{k+1} = u^0 + \sum_{i=1}^{k+1} r_i$, it is the running sum of residuals at each step of iteration

4.2.2 Convergence

The ADMM iterates satisfy the following [8]:

- Residual Convergence: $r^k \to 0$ as k $\to \infty$
- Objective Convergence: $f(x^k) + g(z^k) \to$ p* as k $\to \infty$
- Dual variable convergence. $y^k \to$ y* as k $\to \infty$

Chapter 5

Solution: Distributed Learning & Inference

Many networks are large-scale and comprise of agents with local information and heterogeneous preferences. We surely believe that with one processor or one system we cannot speed up our process of finding the solution to our problem. This motivated much interest in developing distributed schemes for control and optimization on multi-node networked systems. Many of these problems can be represented within the general formulation

$$\min_{x} \sum_{i=1}^{n} f_i(x) \tag{5.1}$$

s.t x $\in$ R^n & f_i(x): $R^n \rightarrow$ R is a convex (possibly non smooth) function know to agent/node i. These kind of problem usually arises

- data that is distributed into several machines,
- data cannot fit into the memory of a single machine,
- the task is computationally too intensive

We would want to employ the methods we learned in Chapters refChapter1 4 to give a distributed solution to our problem.

5.1 Existing Methods

Here we describe various types of Algorithms present in the literature. We talk about synchronous & Asynchronous Algorithms.

5.1.1 Synchronous ADMM

Ermin Wei and Asuman Ozdaglar in [9] in formulated a distributed optimization where nodes on a Network Topology try to learn a function. The network considered was depicted as Graph G = (V,E). Each vertex in the graph represents a set of nodes for computation, that may have data locally on them. Let e_{ij} denote the edge between node i and j. Each node has a cost function f_i and our objective is

$$\min_x \sum_{i=1}^{N} f_i(x) \tag{5.2}$$

Now since we want to reformulate the above problem which will be used to solve the distributed setting. For every edge node in the graph we had the variables which were locally computed the node $x_i's$ for node i. Now we put additional condition for each edge $e_{ij} \in E$ we have constraint $x_i = x_j$. This is similar to the constraint $x_i = \tilde{x}$ for all nodes i for some $\tilde{x}$. We represent all the conditions as a *edge-node* matrix incidence matrix which represents the network topology, A. Each row of matrix A corresponds to the an edge in the graph. The size of this matrix is $|E|$ x $|V|$.

$$[A]_k^{e_ij} = \begin{cases} 1 & \text{if } i = k \\ -1 & \text{if } j = k \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

Now with the new formation our problem looks as follows, F : $R^n \to R$

$$F(x) = \sum_{i=1}^{N} f_i(x) \tag{5.4}$$

s.t $Ax = 0$
where x is the vector $[x_1, \ldots, x_n]'$
Now we have the Lagrangian function L: R^n x $R^m \to R$

$$L(x, \lambda) = F(x) - \lambda' Ax, \tag{5.5}$$

where λ in R^m

Now, coming to the distributed Algorithms, we have a Lagrange dual variables λ_{ij} corresponding to each edge e_{ij}. Each node has local variables as x_i and λ_{ki} which are owned and updated by node i. We define two sets for every node in the graph called the *Predecessor Set* P(i) & *Successor Set* S(i). P(i) = {j $|e_{ji} \in$ E, $j < i$}, S(i) = {i $|e_{ij} \in$ E, $i < j$}. We don't have any self edges in the graph since it is simple. Each row of the

constrain Ax=0 corresponds to an equation $x_i - x_j = 0$ for some $i < j$
Steps in the algorithm

- A Initialization: Choose for every node x_i randomly, Need not be equal for all the nodes
- B For $k \geq 0$,
  - Each agent i updates its estimates x_i^k in a sequential order with

$$x_i^{k+1} = \arg\min f_i(x) + \frac{\beta}{2} \sum_{j \in P(i)} ||x_j^{k+1} - x - \frac{1}{\beta}\lambda_{ji}^k||^2 + \frac{\beta}{2} \sum_{j \in S(i)} ||x - x_{j+1}^k - \frac{1}{\beta}\lambda_{ij}^k||^2 \tag{5.6}$$

  - Each node i updates the $\lambda' s$ it owns

$$\lambda_{ij}^k = \lambda_{ij}^k - \beta(x_j^{k+1} - x_i^{k+1}) \tag{5.7}$$

We assume that the updates take place in sequential order, i.e node 1 to node N. The communication assumption is also there which states that the as soon as x_{ij} and λ_{ij} are computed they are available to neighbors of i immediately through local information exchange.

5.1.2 Asynchronous ADMM

As we have explored in the previous method the major problem we have to wait for all nodes to complete the updates, so if one of the nodes is slow as compared to others than the overall system becomes slow. So we would like to explore the idea of not considering regular updates from all nodes. Ruiliang Zhang and James T. Kwok from HKUST talk about such methods [10]. Our Objective function still remain the same as in case of synchronous ADMM. In this case we need to reformulate the problem to support centralized Asynchronous Computation.

$$\begin{aligned} &\min_{x_1,\ldots,x_n} \sum_{i=1}^{N} f_i(x_i) \\ &\text{subject to } x_i = z \forall i \in \{1, 2, \ldots, N\} \end{aligned} \tag{5.8}$$

Here $x_i' s$ are the individual computations by the nodes and z is the consensus variable. Nodes are penalized for their difference from the consensus variable. The algorithms primary objective is to consider nodes which are relatively slow and try to accommodate

them without compromising on speed and convergence. Just to repeat the ADMM update equations

$$x_i^{k+1} = \arg\min_x f_i(x) + \langle \lambda_i^k, x \rangle + \frac{\beta}{2}||x - z^k||^2, \tag{5.9}$$

$$z^{k+1} = \arg\min_z \sum_{i=1}^{N} -\langle \lambda_i^k, x \rangle + \frac{\beta}{2}||x_i^k - z||^2, \tag{5.10}$$

$$\lambda_i^{k+1} = \lambda_i^k + \beta(x_i^{k+1} - z^{k+1}) \tag{5.11}$$

There are three parts to the algorithm

- **Master and Worker Clocks**

  The master keeps a clock k, which starts from zero and is incremented by 1 after each z update. Every worker also has its own clock k_i , which starts from zero and is incremented by 1 after each λ_i update. All the clocks k and $\{k_i\}_1^N$ are run independently. Let x_i^k , λ_i^k be the values of x_i and λ_i when worker i clock is at k_i and z_k be the value of z when the master's clock is at k

- **Updating x by a worker** Update x_i with at an instant k_i with the most recent value of $\tilde{z}_i$ received by node i from master

$$x_i^{k_i+1} = arg\min_x f_i(x) + \langle \lambda_i^{k_i}, x \rangle + \frac{\beta}{2}||x - \tilde{z}_i||_2^2 \tag{5.12}$$

  Moreover, as the workers have different speeds, the $\tilde{z}_i$'s are in general different for different nodes. After processing the node asks for an new $\tilde{z}_i$ from the master.

- **Updating z by the master** The master waits for workers (λ_i, x_i) before it can update z^k. In synchronous case master waited for all N nodes to update. We allow a *partial barrier* update which ensures that if there are S nodes that have updated x_i's then update z^k, else wait. Consequently these nodes also get to update their (x_i, λ_i). If S is much smaller than N, updates from slow workers will be less frequently incorporated in computing zs. To ensure sufficient freshness of all the updates, we enforce a *bounded delay condition*. Update from every worker has to be serviced by the master at least once every τ iterations, where $\tau \geq 1$ is a user-defined parameter.

  The update equations for z

$$\begin{aligned} z^{k+1} &= \arg\min_z \sum_{i=1}^{N} -\langle \lambda_i^k, x \rangle + \frac{\beta}{2}||x_i^k - z||^2, \\ &= \frac{1}{N}\sum_{i=1}^{N}(\hat{x_i} + \frac{1}{\beta}\hat{\lambda_i}) \end{aligned} \tag{5.13}$$

where $\hat{x_i}$ and $\hat{\lambda_i}$ are most recent updates of variable from the worker nodes. One more important point to be noted here is that though S updates have arrived but $\{\hat{x_i}, \hat{\lambda_i}\}_{i=1}^{N}$ from all the nodes are considered while computing the consensus. Also we must note that many of $\{\hat{x_i}, \hat{\lambda_i}\}$ might be out-dated.
Finally the master's clock is incremented by 1 and then updated z, z^{k+1}, is sent to those nodes which had the most recent values in the iteration, or which updated had $\{\hat{x_i}, \hat{\lambda_i}\}$ in the current iteration.

- **Updating** λ After receiving the updated z^k values from the master (only some nodes receive this updated value), the worker nodes go back to work and compute the dual variables as:

$$\lambda_i^{k+1} = \lambda_i^k + \beta(x_i^{k+1} - \tilde{z_i}) \tag{5.14}$$

$\tilde{z_i}$ is the most updated value of z from the master the node received after sending $x_i's$

5.2 Applying to our Problem

In our problem we could directly employ the above algorithms but there most capacity will not be used because of the inherent nature in which the ADMM feature comes into application. In solving the 3.5 we found out that we needed to rethink some steps to try and apply distributed computation over the data. Our existing formulation did not consider that possibility. There was no consensus sharing between more than two nodes, actually there were just two nodes at max which could be incorporated to parallelize the work. So, here is when we come up with the idea of distributed Learning & and distributed Inference.
At the stage of trying to minimize the svm objective 3.3. One approach could be to try to solve the objective function over different nodes and then get consensus over the weights w.The second place would be to continue to solve optimization problem in 3.5 in a distributed fashion. Solving this would be much faster as the amount of data would be reduced. So, we call this solution as distributed learning and distributed Inference

Chapter 6

Conclusion & Future Work

We have discussed ADMM and illustrated its applicability to distributed convex optimization in general and many problems in statistical machine learning in particular. We argue that ADMM can serve as a good general-purpose tool for optimization problems arising in the analysis and processing of modern massive datasets. Much like gradient descent and the conjugate gradient method are standard tools of great use when optimizing smooth functions on a single machine, ADMM should be viewed as an analogous tool in the distributed regime. ADMM offers a lot scope in distributed optimization by providing a neat trick to solve problems. It provides you a clean set of updates which are independent and respects the philosophy of distributed systems. As a part of future work I would be implementing the Distributed Learning and Inference on structured data as a proof of concept of the raw idea presented as part of BTP

Bibliography

[1] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. ISBN 0-13-648700-9.

[2] Christoph H. Lampert. Learning with structured outputs. 2011.

[3] I. Tsochantaridis, T. Hofmann, T. Joachims, and Y. Altun. Support vector machine learning for interdependent and structured output spaces. In *International Conference on Machine Learning (ICML)*, pages 104–112, 2004.

[4] T. Joachims, T. Finley, and Chun-Nam Yu. Cutting-plane training of structural svms. *Machine Learning*, 77(1):27–59, 2009.

[5] Chun-Nam Yu and T. Joachims. Learning structural svms with latent variables. In *International Conference on Machine Learning (ICML)*, 2009.

[6] Ajay Nagesh. Learning latent svms to optimise multivariate performance measures. pages 1–3, 2014.

[7] Augmented lagrangian method. `http://en.wikipedia.org/wiki/Augmented_Lagrangian_method`.

[8] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Found. Trends Mach. Learn.*, 3(1):1–122, January 2011. ISSN 1935-8237. doi: 10.1561/2200000016. URL `http://dx.doi.org/10.1561/2200000016`.

[9] Ermin Wei and Asuman E. Ozdaglar. Distributed alternating direction method of multipliers. In *CDC*, pages 5445–5450. IEEE, 2012. ISBN 978-1-4673-2065-8. URL `http://dblp.uni-trier.de/db/conf/cdc/cdc2012.html#WeiO12`.

[10] Ruiliang Zhang and James Kwok. Asynchronous distributed admm for consensus optimization. In Tony Jebara and Eric P. Xing, editors, *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1701–1709.

JMLR Workshop and Conference Proceedings, 2014. URL http://jmlr.org/proceedings/papers/v32/zhange14.pdf.