10-708: Probabilistic Graphical Models 10-708, Spring 2016

28 : Distributed Algorithms for ML

Lecturer: Eric P. Xing *Scribes: Joe Runde, Michael Muehl*

1 Introduction

Big data has been all the rage for the last few years- there are even markets where it is sold or used as a form of currency. It has been around for many years, for example biologists sequenced the human genome more than a decade ago. The hope is that big data can generate a lot of value, function, or capabilities for us. But the question is, how do we take advantage of this big data? We need systems that can search, digest, index, and understand contents of big data, but there are a few key challenges with in doing so. This lecture will outline those challenges, and describe big picture solutions with distributed algorithms.

2 Challenges with understanding Big Data

Looking into the data to find the information we need is not an easy task as data grows larger. Fundamental issues arise with figuring out how to make the computations, storage, transmission, and imputation of the data happen in a rapid way. Then there are technical challenges beyond that, of ensuring the models continue to work better as the size of the system increases. Often naiive implementations will end up with very poor performance, such as the red line in Figure 1. Below we outline 3 key challenges of scaling up ML methods.

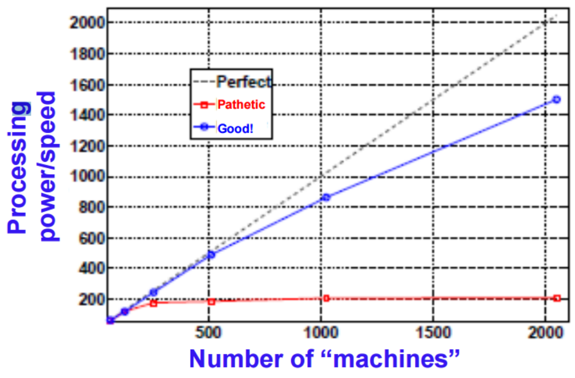


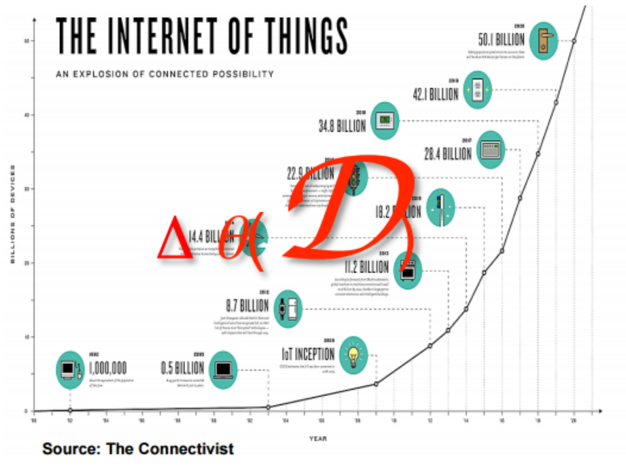
Figure 1: Common scalability issue with ML methods

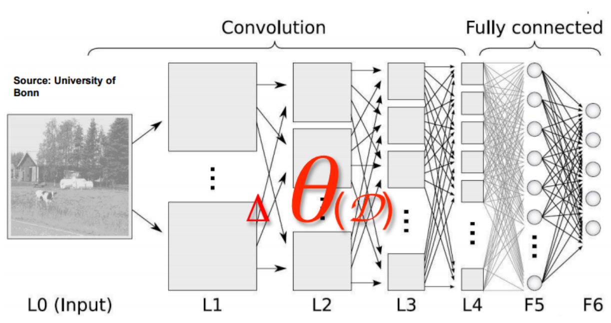
2.1 Challenge 1: Massive Data Scale

The first problem is very obvious: the data is very big. For example, an Internet of Things dataset may contain data from over 50B devices, or in the realm of social data, Facebook now has over 1.6 billion active users. Even neglecting most of the information from these devices or people, we need to deal with a graph with billions of nodes, and edges among them. Already we can see there will be a big problem just storing and transmitting this data, as it cannot fit into the memory of a single machine.

2.2 Challenge 2: Massive Model Scale

Once the problem of transmitting and storing all of the data is dealt with, there is the even bigger challenge of doing computations with it. Sophisticated models like very large deep CNNs, or models with parameters that grow with the amount of data may have billions of parameters. How can we ensure that computations of these parameters are done correctly and efficiently across many machines, and then attempt to do inference?





2.3 Challenge 3: Supporting New Methods

Finally, older methods like KNN, K means, or Naive Bayes are often insufficient for picking out the concepts that we want to learn from big data. In fact, even for smaller data like image sets that could be stored on a single computer, larger models such as deep neural nets with billions of nodes are used. While some newer software packages like Torch, YahooLDA, or Vowpal Wabbit provide newer models, many packages still only implement the classic methods that have been in place for decades. Implementing newer models to gain insights from big data is often a challenge left up to the system engineers.

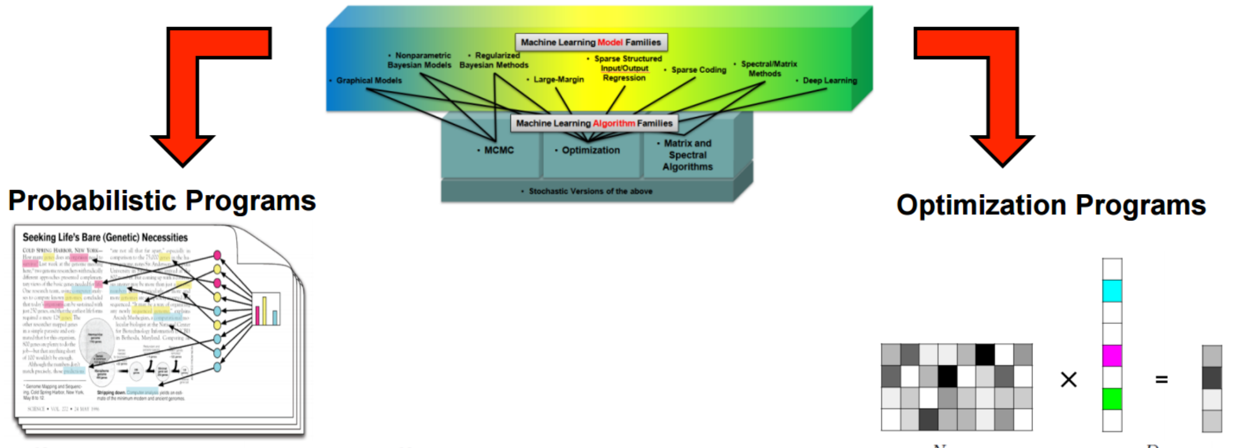
2.4 Solution

How can we start to solve these challenges? They are still open questions- it is unclear even whether the best approach is to try to make a serial algorithm faster, or try to use a simpler approach in parallel and try to guarantee correctness across many machines. We will need to present a formal view of machine learning methods, and explain how to design scalable systems based on a few mathematical and functional properties of the method.

3 ML Methods

We'll start by defining some broad properties of ML programs. An ML program consists of a mathematical model from some family, which is solved by an ML algorithm from one of a few types. Further, we can view each ML program as either a probabilistic program, or an optimization program. A visualization is provided in figure 3

We can then break down an ML program into its constituent parts. Usually we have:


Machine Learning Model Families
Nonparametric Bayesian Models
Regularized Bayesian Methods
Sparse Structured Input/Output Regression
Sparse Coding
Spectral/Matrix Methods
Graphical Models
Large-Margin
Deep Learning
Machine Learning Algorithm Families
MCMC
Optimization
Matrix and Spectral Algorithms
Stochastic Versions of the above
Probabilistic Programs
Optimization Programs
Seeking Life's Bare (Genetic) Necessities
×
=

$$\sum_{i=1}^{N}\sum_{j=1}^{N_i}\ln \mathbb{P}_{Categorical}(x_{ij} \mid z_{ij}, B) + \sum_{i=1}^{N}\sum_{j=1}^{N_i}\ln \mathbb{P}_{Categorical}(z_{ij} \mid \delta_i)$$

$$\sum_{i=1}^{N}\|y_i - X_i\beta\|_2^2 + \lambda\sum_{j=1}^{D}|\beta_j|$$

Figure 2: Classification of ML programs

- An Objective/Loss function: $L(\theta, D)$

  Some examples are data log likelihood, or MSE between data and predictions.

- Regularization / Prior or Structural Knowledge: $r(\theta)$

  Such as: L2 regularization to prevent overfitting, L1 regularization for sparseness, or Dirichlet priors for smoothing

- An algorithm to solve for the model given the data

Usually, ML models are solved with an iterative-convergent ML algorithm. These algorithms repeatedly update θ until it is stationary. For example, MCMC methods or Variation Inference are used to solve probabilistic problems, where Gradient Descent or Proximal methods are used to solve optimization problems. Here are a couple examples of optimization programs and probabilistic programs:

3.1 Example 1: Lasso Regression

Lasso Regression is an example of an optimization problem. Here the data D is a feature matrix X with response vector y, and the model θ is a parameter vector. The objective, $L(\theta, D)$, used is the least squares difference between y and $x\theta$. The regularization, $r(\theta)$, is the L1 norm of θ, times a tunable scalar, λ. The possible algorithms used to solve the problem are Cooredinate Descent or Stochastic Proximal Gradient Descent.

3.2 Example 2: Topic Models

Topic models are an example of a probabilistic problem. Here the objective, $L(\theta, D)$ is the log likelihood of documents given topic indicators, document-topic distributions, and topic-word distributions. The prior, $r(\theta)$, is a Dirichlet over the document-topic and topic-word distributions, with tunable hyperparameters controlling their strength. The algorithm used to learn the model is usually Collapsed Gibbs sampling.

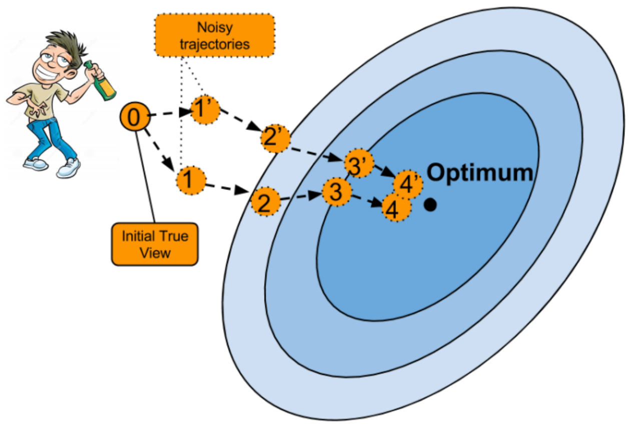


Figure 3: Error tolerance of ML programs

4 ML vs Classical Computing

In classical computing, programs are deterministic and operation-centric: a strict set of instructions with predetermined length is followed until completion. A problem with this approach is that strict operational correctness is required; one error in the middle of a sorting algorithm and the end result is useless. Atmoic correctness is required, and there is an entire field of research on fault-tolerance to address this issue.

On the other hand, ML algorithms are probabilistic, optimization-centric, and iterative-convergent. Update rules which may or may bot be deterministic are followed for an indeterminate amount of time until some solution is reached. In this case, a few errors here and there don't really matter, as the program will continue seeking an optimum regardless of a few missteps.

ML Programs do have pitfalls of their own as well though. First, there can be structural dependency which isn't known a-prior, and can change as the program runs. As correlations between parameters change, the parallel structure of the program must change as well to stay efficient. Convergence of the parameters themselves is also non uniform: Often most of the parameters converge very quickly, while the rest take thousands more iterations to fit. Structuring a system around these properties can be challenging.

Having this formal classification and breakdown of ML programs can be immensely helpful to build scalable systems. We can implement a smaller number of "workhorse" algorithms which solve the basic challenges outlined above, and then apply these algorithms to newer, more sophisticated models as they are created.

5 Distributed ML Programs

Generally, there are two types of programs we may wish to parallelize: optimization programs such as Stochastic Gradient Descent or LASSO or probabilistic programs implementing various Markov Chain Monte Carlo methods such as Gibbs Sampling. Additionally, we can take two approaches to the parallelization procedure: we can parallelize the model or parallelize the data. For this lecture, we are going to focus on optimization programs, specifically Stochastic Gradient Descent.

5.1 Parallel Methods for Stochastic Gradient Descent

Stochastic Gradient Descent is an algorithm that at first glance seems poorly suited to parallelization. Recall the basic Stochastic Gradient Descent Template:

$$x \leftarrow x - \eta_t \lambda \Omega'[x] - \eta_t \partial x f_t(x)$$

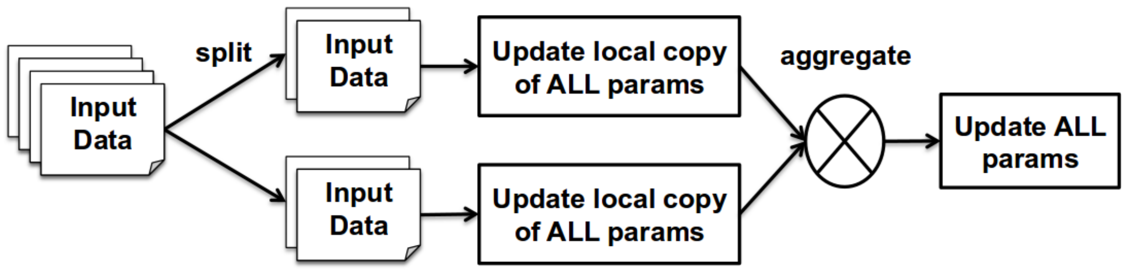
where $f(x)$ is the loss function and $\Omega[x]$ is the regularizer.

5.2 Standard Parallel Methods

The most immediately obvious problem is that Stochastic Gradient Descent is inherently serial: the parameters θ must be updated after seeing every example. The next example needs to wait until the update is done before it can calculate the gradient, so any sort of parallelization scheme needs to address this issue. However, the method provided by Zinkevich et al. provides theoretical guarantees of convergence in a multicore setting.

Here we execute the above loop in each core independently while working with a common shared weight vector x. This means that if we update x in a round-robin fashion we will have a delay of $k-1$ when using k cores. The delay is due to the time it takes between seeing an instance of f and when we can actually apply the update to x. The intuition behind this is that as we converge optimization becomes more and more an averaging procedure and in this case a small delay does not hurt at all.

The Achilles heel of the above algorithm is that it requires extremely low latency between the individual processing nodes. While this is OK on a computer or on a GPU, it will fail miserably on a network of computers since we may have many miliseconds latency between the nodes. This suggests an even simpler algorithm for further parallelization:


Input Data
split
Input Data
Input Data
Update local copy of ALL params
Update local copy of ALL params
aggregate
Update ALL params

1. Overpartition the data into k blocks for k clusters (i.e. for each cluster simply randomly draw a fraction $c = O(m/k)$ of the entire dataset).

2. Now perform stochastic gradient descent on each machine separately with constant learning rate.

3. Average the solutions between different machines.

This can also be shown to converge.

5.3 Hogwild!

Interestingly, in many machine learning problems the cost can be represented as just a function of x $f(x)$, and additionally this function is sparse. Consider Sparse SVM:

$$\min_x \sum_{\alpha \in E} \max(1 - y_\alpha x^T z_\alpha, 0) + \lambda \|x\|_2^2$$

where z_α is sparse, or Matrix Completion:

$$\min_{W,H} \sum_{(u,v)\in E} (A_{uv} - W_u H_v^T)^2 + \lambda_1 \|W\|_F^2 + \lambda_2 \|H\|_F^2$$

where the input matrix A is sparse, or Graph Cuts:

$$\min_x \sum_{(u,v)\in E} w_{uv} \|x_u - x_v\|_1 \text{subject to } x_v \in S_D, v = 1, \ldots, n$$

where W is a sparse similarity matrix, encoding a graph.

In the Hogwild! algorithm, we take advantage of this sparsity as follows, in parallel for each core:

1. Sample e uniformly at random from E
2. Read current parameter x_e, evaluate gradient of function f_e
3. Sample uniformly from e a coordinate v
4. Perform SGD on only coordinate v with a small constant step size.

Since we are only updating a single coordinate at a time, there is no need for memory locking of any kind. This gives a near-linear speedup if done on a single machine. However, there are problems if we try to use this algorithm in a distributed environment. In particular, since the variance of the parameter estimate is:

$$\text{Var}_{t+1} = \text{Var}_t - 2\eta_t cov(\mathbf{x}_t, \text{E}^{\Delta_t}[\mathbf{g}_t]) + \mathcal{O}(\eta_t \xi_t) + \mathcal{O}(\eta_t^2 \rho_t^2) + \mathcal{O}_{\epsilon_t}^*$$

where $\mathcal{O}_{\epsilon_t}^*$ represents 5th and higher order terms, as a function of the delay among machines ϵ_t. So the higher the delay between machines, the higher the variance in the parameter estimate, and hence the more instability in convergence. Since distributed systems generally have much higher delay than single machines, this algorithm is better suited for use on a single machine.