Parallel and Distributed Stochastic Learning

Towards Scalable Learning for Big Data Intelligence

李武军

LAMDA Group
南京大学计算机科学与技术系
软件新技术国家重点实验室

liwujun@nju.edu.cn

Dec 10, 2016
Outline

1. Introduction
2. AsySVRG
3. SCOPE
4. Conclusion
Machine Learning

- **Supervised Learning:**
  Given a set of training examples \( \{(x_i, y_i)\}_{i=1}^n \), supervised learning tries to solve the following regularized empirical risk minimization problem:

  \[
  \min_{w} f(w) = \frac{1}{n} \sum_{i=1}^{n} f_i(w),
  \]

  where \( f_i(w) \) is the loss function (plus some regularization term) defined on example \( i \), and \( w \) is the parameter to learn.

  **Examples:**
  - Logistic regression: \( f(w) = \frac{1}{n} \sum_{i=1}^{n} [\log(1 + e^{-y_i x_i^T w}) + \frac{\lambda}{2} \|w\|^2] \)
  - SVM: \( f(w) = \frac{1}{n} \sum_{i=1}^{n} [\max\{0, 1 - y_i x_i^T w\} + \frac{\lambda}{2} \|w\|^2] \)
  - Deep learning models

- **Unsupervised Learning:**
  Many unsupervised learning models, such as PCA and matrix factorization, can also be reformulated as similar problems.
Introduction

Machine Learning for Big Data

For big data applications, first-order methods have become much more popular than other higher-order methods for learning (optimization). Gradient descent methods are the most representative first-order methods.

- **(Deterministic) gradient descent (GD):**

  \[
  w_{t+1} \leftarrow w_t - \eta_t \left[ \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(w_t) \right],
  \]

  where \( t \) is the iteration number.

  - *Linear convergence rate:* \( O(\rho^t) \)
  - Iteration cost is \( O(n) \)

- **Stochastic gradient descent (SGD):** In the \( t^{th} \) iteration, randomly choosing an example \( i_t \in \{1, 2, \ldots, n\} \), then update

  \[
  w_{t+1} \leftarrow w_t - \eta_t \nabla f_{i_t}(w_t)
  \]

  - Iteration cost is \( O(1) \)
  - The convergence rate is *sublinear:* \( O(1/t) \)
Stochastic Learning for Big Data

Researchers recently proposed improved versions of SGD: SAG [Roux et al., NIPS 2012], SDCA [Shalev-Shwartz and Zhang, JMLR 2013], SVRG [Johnson and Zhang, NIPS 2013].

Number of gradient ($\nabla f_i$) evaluation to reach $\epsilon$ for smooth and strongly convex problems:

- GD: $O(n\kappa \log(\frac{1}{\epsilon}))$
- SGD: $O(\kappa(\frac{1}{\epsilon}))$
- SAG: $O(n \log(\frac{1}{\epsilon}))$ when $n \geq 8\kappa$
- SDCA: $O((n + \kappa) \log(\frac{1}{\epsilon}))$
- SVRG: $O((n + \kappa) \log(\frac{1}{\epsilon}))$

where $\kappa = \frac{L}{\mu} > 1$ is the condition number of the objective function.

Stochastic Learning:

- Stochastic GD
- Stochastic coordinate descent
- Stochastic dual coordinate ascent
Parallel and Distributed Stochastic Learning

To further improve the learning scalability (speed):

- **Parallel stochastic learning:**
  One machine with multiple cores and a shared memory

- **Distributed stochastic learning:**
  A cluster with multiple machines

**Key issues: cooperation**

- **Parallel stochastic learning:**
  lock vs. lock-free: waiting cost and lock cost

- **Distributed stochastic learning:**
  synchronous vs. asynchronous: waiting cost and communication cost
Our Contributions

- **Parallel stochastic learning**: AsySVRG

- **Distributed stochastic learning**: SCOPE
  Scalable Composite Optimization for Learning
Outline

1. Introduction

2. AsySVRG

3. SCOPE

4. Conclusion
Motivation and Contribution

Motivation:

- Existing asynchronous parallel SGD: Hogwild! [Recht et al. 2011], and PASSCoDe [Hsieh, Yu, and Dhillon 2015]
- No parallel methods for SVRG.
- Lock-free: empirically effective, but no theoretical proof.

Contribution:

- A fast asynchronous method to parallelize SVRG, called AsySVRG.
- A lock-free parallel strategy for both read and write
- Linear convergence rate with theoretical proof
- Outperforms Hogwild! in experiments

AsySVRG is the first lock-free parallel SGD method with theoretical proof of convergence.
AsySVRG: a multi-thread version of SVRG

Initialization: $p$ threads, initialize $w_0, \eta$;

for $t = 0, 1, 2, \ldots$ do

$u_0 = w_t$;

All threads parallelly compute the full gradient

$\nabla f(u_0) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(u_0)$;

$u = w_t$;

For each thread, do:

for $m = 1$ to $M$ do

Read current value of $u$, denoted as $\hat{u}$, from the shared memory.

And randomly pick up an $i$ from $\{1, \ldots, n\}$;

Compute the update vector: $\hat{v} = \nabla f_i(\hat{u}) - \nabla f_i(u_0) + \nabla f(u_0)$;

$u \leftarrow u - \eta \hat{v}$;

end for

Take $w_{t+1}$ to be the current value of $u$ in the shared memory;

end for
Lock-free Analysis

In all the GD or SGD methods to solve the objective function, the key step can be written as

\[ \mathbf{u} \leftarrow \mathbf{u} + \Delta \]

Notation

- \( \Delta_{i,j} \): the \( j^{th} \) update vector computed by the \( i^{th} \) thread;
- \( \mathbf{u} \in \mathbb{R}^d \) and \( \mathbf{u} = (u^{(1)}, u^{(2)}, \ldots, u^{(d)}) \);
- \( t^{(k)}_{i,j} \): the time when the operation \( \mathbf{u}^{(k)} \leftarrow \mathbf{u}^{(k)} + \Delta^{(k)}_{i,j} \) has been completed (Not the time when the operation begins);
- Assuming \( \forall i, j, t^{(1)}_{i,j} < t^{(2)}_{i,j} < \ldots < t^{(d)}_{i,j} \), which can be easily guaranteed by programming
Lock-free Analysis: update sequence

Since $u^{(1)}$ can only be changed by at most one thread at any absolute time, these $t_{i,j}^{(1)}$ are different from each other. So we can:

- Sort these $t_{i,j}^{(1)}$ as $t_0^{(1)} < t_1^{(1)} < \ldots < t_{\tilde{M}-1}^{(1)}$ ($\tilde{M} = p \times M$);
- $\Delta_0, \Delta_1, \ldots, \Delta_{\tilde{M}-1}$ are the corresponding update vectors.

Since it is lock-free, for each update vector $\Delta_m$, the real update vector is $B_m\Delta_m$ because of over-written. The $B_m$ is a diagonal matrix whose diagonal elements are 0 or 1.

After all the inner-loop stop, we can get:

$$w_{t+1} = u_0 + \sum_{m=0}^{\tilde{M}-1} B_m\Delta_m$$  \hspace{1cm} (1)
Lock-free Analysis: update sequence

According to (1), we define a sequence \( \{u_m\} \) as follows:

\[
u_m = u_0 + \sum_{i=0}^{m-1} B_i \Delta_i
\] (2)

which means \( u_{m+1} = u_m + B_m \Delta_m \).

**Note**
The sequence \( \{u_m\} \) \((m = 1, 2, \ldots, \tilde{M} - 1)\) is synthetic, and the whole \( u_m \) may never occur in the shared memory. What we can get is only the final value of \( u_{\tilde{M}} \).
Lock-free Analysis: read sequence

Assume the old update vectors $\Delta_0, \Delta_1, \ldots, \Delta_{a(m)−1}$ have been completely applied to $u$ when one thread is reading the shared variable. At the same time, some new update vectors might be updating $u$. So we can write $\hat{u}_m$ read by the thread to compute $\Delta_m$ as follows:

\[
\hat{u}_m = u_{a(m)} + \sum_{i=a(m)}^{b(m)} P_{m,i-a(m)} \Delta_i
\]

where $P_{m,i-a(m)}$ is a diagonal matrix whose diagonal elements are 0 or 1.

According to the principle of the order, $\Delta_i (i \geq m)$ should not be read by $\hat{u}_m$. So $b(m) < m$, which means:

\[
\hat{u}_m = u_{a(m)} + \sum_{i=a(m)}^{m-1} P_{m,i-a(m)} \Delta_i
\]
Convergence Result for Strongly Convex Problems

With some assumptions, our algorithm gets a linear convergence rate for strongly convex problems:

$$\mathbb{E} f(w_{t+1}) - f(w^*) \leq (c_1 \tilde{M} + \frac{c_2}{1 - c_1})(\mathbb{E} f(w_t) - f(w^*)),$$

where $c_1 = 1 - \alpha\eta\mu + c_2$ and $c_2 = \eta^2(\frac{8\tau L^3 \eta \rho^2 (\rho^\tau - 1)}{\rho - 1} + 2L^2 \rho)$, $\tilde{M} = p \times M$ is the total number of iterations of the inner-loop.

**Note**
Since it is lock-free, we do not know the exact $B_m$ and we cannot take the average sum of $B_m u_m$ to be $w_{t+1}$. 
Convergence Result for Non-Convex Problems

With some assumptions, our algorithm gets a sub-linear convergence rate for non-convex problems:

$$\frac{1}{T\tilde{M}} \sum_{t=0}^{T-1} \sum_{m=0}^{\tilde{M}-1} \mathbb{E}\|\nabla f(u_{t,m})\|^2 \leq \frac{\mathbb{E}f(w_0) - \mathbb{E}f(w_T)}{T\tilde{M}\gamma}.$$  

Similar to the analysis for strongly convex problems, we construct an equivalent write sequence \(\{u_{t,m}\}\) for the \(t^{th}\) outer-loop:

\[ u_{t,0} = w_t, \]
\[ u_{t,m+1} = u_{t,m} - \eta B_{t,m} \hat{v}_{t,m}, \]

where \(\hat{v}_{t,m} = \nabla f_{i_{t,m}}(\hat{u}_{t,m}) - \nabla f_{i_{t,m}}(u_{t,0}) + \nabla f(u_{t,0}). B_{t,m}\) is a diagonal matrix whose diagonal entries are 0 or 1. And \(\hat{u}_{t,m}\) is read by the thread who computes \(\hat{v}_{t,m}\).
Experiments

**Experimental platform:** A server with 12 Intel cores and 64G memory.

**Model:** Logistic regression with $L^2$-norm

\[
f(w) = \frac{1}{n} \sum_{i=1}^{n} \log(1 + e^{-y_i x_i^T w}) + \frac{\lambda}{2} \|w\|^2
\]

**Data set**

<table>
<thead>
<tr>
<th>dataset</th>
<th>instances</th>
<th>features</th>
<th>memory</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>rcv1</td>
<td>20,242</td>
<td>47,236</td>
<td>36M</td>
<td>sparse</td>
</tr>
<tr>
<td>real-sim</td>
<td>72,309</td>
<td>20,958</td>
<td>90M</td>
<td>sparse</td>
</tr>
<tr>
<td>news20</td>
<td>19,996</td>
<td>1,355,191</td>
<td>140M</td>
<td>sparse</td>
</tr>
<tr>
<td>epsilon</td>
<td>400,000</td>
<td>2,000</td>
<td>11G</td>
<td>dense</td>
</tr>
</tbody>
</table>

We set $\lambda = 10^{-4}$. 
Experiments: Computation Cost

Figure: Convergence rate with respect to the number of effective passes (the vertical axis is in a log scale). Please note that in (a) and (b), the curves of AsySVRG-1, AsySVRG-lock-10 and AsySVRG-10 overlap with each other.
Experiments: Total Time Cost

(a) rcv1

(b) realsim

(c) news20

(d) epsilon

Figure: Convergence rate with respect to CPU time (the vertical axis is in a log scale, and the horizontal axis is the ratio to the CPU time of Hogwild with the stopping condition $f(w) - f(w^*) < \epsilon$.)
Experiments: Speed up

(a) rcv1

(b) real–sim

(c) news20

(d) epsilon

Figure: Speedup results.
Outline

1. Introduction
2. AsySVRG
3. SCOPE
4. Conclusion
Motivation and Contribution

Motivation:

- Bulk synchronous parallel (BSP) models, such as MapReduce, are commonly considered to be inefficient for distributed stochastic learning. Is there any technique to solve the issues of BSP models?

Contribution:

- A novel distributed stochastic learning method, called scalable composite optimization for learning (SCOPE), on BSP models
- Both computation-efficient and communication-efficient
- Linear convergence rate with theoretical proof
- Can be easily integrated into the data processing pipeline of Spark
- Outperform other state-of-the-art distributed learning methods on Spark
Framework of SCOPE

Figure: Distributed framework of SCOPE.
Optimization Algorithm: Master

Task of Master in SCOPE:

Initialization: \( p \) Workers, \( w_0 \);

for \( t = 0, 1, 2, \ldots, T \) do

Send \( w_t \) to the Workers;

Wait until it receives \( z_1, z_2, \ldots, z_p \) from the \( p \) Workers;

Compute the full gradient \( z = \frac{1}{n} \sum_{k=1}^{p} z_k \), and then send \( z \) to each Worker;

Wait until it receives \( \tilde{u}_1, \tilde{u}_2, \ldots, \tilde{u}_p \) from the \( p \) Workers;

Compute \( w_{t+1} = \frac{1}{p} \sum_{k=1}^{p} \tilde{u}_k \);

end for
Optimization Algorithm: Workers

Task of Workers in SCOPE:

Initialization: initialize $\eta$ and $c > 0$;

For the Worker $k$:

\begin{itemize}
  \item \textbf{for} $t = 0, 1, 2, \ldots, T$ \textbf{do}
    \begin{itemize}
      \item Wait until it gets the newest parameter $w_t$ from the Master;
      \item Let $u_{k,0} = w_t$, compute the local gradient sum $z_k = \sum_{i \in D_k} \nabla f_i(w_t)$, and then send $z_k$ to the Master;
      \item Wait until it gets the full gradient $z$ from the Master;
      \item \textbf{for} $m = 0$ to $M - 1$ \textbf{do}
        \begin{itemize}
          \item Randomly pick up an instance with index $i_{k,m}$ from $D_k$;
          \item $u_{k,m+1} = u_{k,m} - \eta(\nabla f_{i_{k,m}}(u_{k,m}) - \nabla f_{i_{k,m}}(w_t) + z + c(u_{k,m} - w_t))$;
        \end{itemize}
      \end{itemize}
    \end{itemize}

Send $u_{k,M}$ or $\frac{1}{M} \sum_{m=1}^{M} u_{k,m}$, which is called the \textit{locally updated parameter} and denoted as $\tilde{u}_k$, to the Master;

\end{itemize}
Convergence

Let $\alpha = 1 - \eta(2\mu + c) < 1$, $\beta = c\eta + 3L^2\eta^2$ and $\alpha + \beta < 1$. We have the following theorems:

**Theorem**

*If we take $w_{t+1} = \frac{1}{p} \sum_{k=1}^{p} u_{k,M}$, then we can get the following convergence result:*

$$E\|w_{t+1} - w^*\|^2 \leq \left( \alpha^M + \frac{\beta}{1 - \alpha} \right) E\|w_t - w^*\|^2.$$

**Theorem**

*If we take $w_{t+1} = \frac{1}{p} \sum_{k=1}^{p} \tilde{u}_k$ with $\tilde{u}_k = \frac{1}{M} \sum_{m=1}^{M} u_{k,m}$, then we can get the following convergence result:*

$$E\|w_{t+1} - w^*\|^2 \leq \left( \frac{1}{M(1 - \alpha)} + \frac{\beta}{1 - \alpha} \right) E\|w_t - w^*\|^2.$$
Communication Cost

- Traditional mini-batch based methods: $O(Tn)$
- SCOPE: $O(T)$
Experiment

Logistic regression (LR) with a $L_2$-norm regularization term:

$$f(w) = \frac{1}{n} \sum_{i=1}^{n} \left[ \log(1 + e^{-y_i x_i^T w}) + \frac{\lambda}{2} \|w\|^2 \right].$$

Table: Datasets for evaluation.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#instances</th>
<th>#features</th>
<th>memory</th>
<th>λ</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST-8M</td>
<td>8,100,000</td>
<td>784</td>
<td>39G</td>
<td>1e-4</td>
</tr>
<tr>
<td>epsilon</td>
<td>400,000</td>
<td>2,000</td>
<td>11G</td>
<td>1e-4</td>
</tr>
<tr>
<td>KDD12</td>
<td>73,209,277</td>
<td>1,427,495</td>
<td>21G</td>
<td>1e-4</td>
</tr>
<tr>
<td>Data-A</td>
<td>106,691,093</td>
<td>320</td>
<td>260G</td>
<td>1e-6</td>
</tr>
</tbody>
</table>

Two Spark clusters with Intel CPUs:

- small: 1 Master and 16 Workers
- large: 1 Master and 128 Workers
Experiment

Baselines:

- **MLlib** [Meng et al., 2015]: MLlib is an open source library for distributed machine learning on Spark. We compare our method with distributed lbfgs for MLlib, which is a batch learning method and faster than the SGD version of MLlib.

- **LibLinear** [Lin et al., 2014a]: LibLinear is a distributed Newton method, which is also a batch learning method.

- **Splash** [Zhang and Jordan, 2015]: Splash is a distributed SGD method by using the local learning strategy to reduce communication cost.

- **CoCoA** [Jaggi et al., 2014]: CoCoA is a distributed dual coordinate ascent method.

- **CoCoA+** [Ma et al., 2015]: CoCoA+ is an improved version of CoCoA. CoCoA+ adopts adding rather than average to combine local updates.
Experiment

(a) MNIST-8M

(b) epsilon

(c) KDD12

(d) Data-A
Experiment

(e) Speedup

(f) Synchronization cost
Outline

1. Introduction
2. AsySVRG
3. SCOPE
4. Conclusion
Conclusion

- Stochastic learning is becoming popular for big data machine learning.

- Lock-free strategy is the key to get a good speedup in parallel stochastic learning.

- With properly designed techniques, BSP models are also efficient for distributed stochastic learning.
Future Work

Open source project:

**LIBBLE: A library for big learning**

- **LIBBLE-Spark**: [https://github.com/LIBBLE/LIBBLE-Spark/](https://github.com/LIBBLE/LIBBLE-Spark/)
  - **Classification**: LR, SVM, LR with L1-norm Regularization
  - **Regression**: Linear Regression, Lasso
  - **Generalized Linear Models**: with L2-norm/L1-norm Regularization
  - **Dimensionality Reduction**: PCA, SVD
  - **Matrix Factorization**
  - **Clustering**: K-Means

- **LIBBLE-MPI**

- **LIBBLE-GPU**

- **LIBBLE-MultiThread**


Q & A

Thanks!