Parameter Servers

(slides courtesy of Aurick Qiao, Joseph Gonzalez, Wei Dai, and Jinliang Wei)
Regret analysis for on-line optimization
Slow Learners are Fast

John Langford
Alexander J. Smola
Martin Zinkevich
Yahoo! Labs, Great America Parkway, Santa Clara, CA 95051 USA

JL@HUNCH.NET
ALEX@SMOLA.ORG
MAZ@YAHOO-IN.COM

2009

Algorithm 1 Delayed Stochastic Gradient Descent

**Input:** Feasible space $X \subseteq \mathbb{R}^n$, annealing schedule $\eta_t$ and delay $\tau \in \mathbb{N}$

**Initialization:** set $x_1 \ldots, x_{\tau} = 0$ and compute corresponding $g_t = \nabla f_t(x_t)$.

**for** $t = \tau + 1$ **to** $T + \tau$ **do**

- Obtain $f_t$ and incur loss $f_t(x_t)$
- Compute $g_t := \nabla f_t(x_t)$
- Update $x_{t+1} = \text{argmin}_{x \in X} \|x - (x_t - \eta_t g_{t-\tau})\|$ (Gradient Step and Projection)

**end for**
1. Take a gradient step: $x' = x_t - \eta_t g_t$

2. If you’ve restricted the parameters to a subspace $X$ (e.g., must be positive, ...) find the closest thing in $X$ to $x'$: $x_{t+1} = \text{argmin}_X \text{dist}(x - x')$

3. But.... you might be using a “stale” $g$ (from $\tau$ steps ago)

---

**Algorithm 1 Delayed Stochastic Gradient Descent**

**Input:** Feasible space $X \subseteq \mathbb{R}^n$, annealing schedule $x_1 \ldots, x_{\tau} = 0$ and compute cost function $g_t = \nabla f_t(x_t)$.

**for** $t = \tau + 1$ **to** $T + \tau$ **do**

- Obtain $f_t$ and incur loss $f_t(x_t)$
- Compute $g_t := \nabla f_t(x_t)$
- Update $x_{t+1} = (x_t - \eta_{t}g_{t-\tau})$ (Gradient Step)

**end for**
Regret: how much extra loss was incurred **during learning**, relative the loss incurred with an **optimal** choice of $x$

Regret = \# mistakes made in learning

\[
R[X] := \sum_{t=1}^{T} f_t(x_t) - f_t(x^*).
\]

Special case:
- $f_t$ is 1 if a mistake was made, 0 otherwise
- $f_t(x^*) = 0$ for optimal $x^*$
**Theorem:** you can find a learning rate so that the regret of delayed SGD is bounded by

\[ R[X] \leq 4FL\sqrt{\tau T} \]

\[ \max_{x, x' \in X} D(x \| x') \leq F^2 \]

\[ D(x \| x') := \frac{1}{2} \|x - x'\|^2 \]

\[ \|\nabla f_t(x)\| \leq L \]

\[ T = \# \text{ timesteps} \]

\[ \tau = \text{ staleness} > 0 \]
Summary of “Slow Learners are Fast”

• Generalization of iterative parameter mixing
  – run multiple learners in parallel
  – conceptually they share the same weight/parameter vector BUT ...

• Learners share weights *imperfectly*
  – learners are *almost* synchronized
  – there’s a bound \( \tau \) on **how stale** the shared weights get
Background: Distributed Coordination Services

- Example: Apache ZooKeeper
- Distributed processes coordinate through shared “data registers” (aka *znodes*) which look a bit like a shared in-memory filesystem
Background: Distributed Coordination Services

- Client:
  - create /w_foo
  - set /w_foo “bar”
  - get /w_foo → “bar”

- Better with more reads than writes
One Machine Learning Platform to Serve Many Industries

Petuum is a software infrastructure and ecosystem provider that enables AI for enterprise. Petuum’s operating system gives users a single platform to build any Machine Learning or Deep Learning application using large amounts of data, and deploy it at scale on any hardware – such as workstations, datacenters, the internet of things, and edge computing.

The Petuum development platform and gallery of AI building blocks work with any programming language and any type of data, allowing managers and analysts to quickly build AI applications without any coding, while engineers and coders can further re-program applications as needed. With Petuum, many AI applications and hardware can be created and managed from a single laptop or terminal, driving higher productivity, better service, lower costs, and faster delivery. By standardizing AI solutions, Petuum lowers the barrier to AI adoption and allows for the integration of AI into every industry.

Petuum closes $93 Million Series B round led by SoftBank with participation from previous investor Advantech Capital, becoming one of the highest funded early-stage Artificial Intelligence and Machine Learning startups.
Parameter Servers

(slides courtesy of Aurick Qiao, Joseph Gonzalez, Wei Dai, and Jinliang Wei)
ML Systems

Scalable Machine Learning Algorithms

Abstractions

Scalable Systems
ML Systems Landscape

Dataflow Systems

- Hadoop
- Spark

Graph Systems

- GraphLab
- Tensorflow

Shared Memory Systems

- Bosen
- DMTK
- ParameterServer.org
ML Systems Landscape

Dataflow Systems

Graph Systems

Shared Memory Systems

Algorithms

Hadoop, Spark

GraphLab, Tensorflow

Bosen, DMTK, ParameterServer.org
ML Systems Landscape

Dataflow Systems

Graph Systems

Shared Memory Systems

- Naïve Bayes, Rocchio
- Graph Algorithms, Graphical Models
- SGD, Sampling

- Hadoop, Spark
- GraphLab, Tensorflow
- Bosen, DMTK, ParameterServer.org

[NIPS’09, NIPS’13]
ML Systems Landscape

Dataflow Systems

Naïve Bayes, Rocchio

Graph Systems

Graph Algorithms, Graphical Models

SGD, Sampling [NIPS’09, NIPS’13]

Shared Memory Systems

Model

Abstractions

Hadoop & Spark

GraphLab, Tensorflow

Bosen, DMTK, ParameterServer.org
ML Systems Landscape

Dataflow Systems
- Hadoop & Spark
- PIG, GuineaPig, ...

Graph Systems
- Graph Algorithms, Graphical Models
  - Vertex-Programs [UAI’10]

Graph Systems
- SGD, Sampling
  - [NIPS’09, NIPS’13]

Shared Memory Systems
- Parameter Server [VLDB’10]
- Bosen, DMTK, ParameterServer.org

- Naïve Bayes, Rocchio
- GraphLab, Tensorflow
Simple case: Parameters of the ML system are stored in a distributed hash table that is accessible thru the network.

Param Servers used in Google, Yahoo, .... Academic work by Smola, Xing, ...

[VLDB’10]

[NIPS’09, NIPS’13]
Model parameters are stored on PS machines and accessed via key-value interface (distributed shared memory)

- Extensions: multiple keys (for a matrix); multiple “channels” (for multiple sparse vectors, multiple clients for same servers, ...)

[Smola et al 2010, Ho et al 2013, Li et al 2014]
Parameter Server (PS)

➢ Extensions: push/pull interface to send/receive most recent copy of (subset of) parameters, blocking is optional

➢ Extension: can block until push/pulls with clock < \( t - \tau \) complete

[Smola et al 2010, Ho et al 2013, Li et al 2014]
Data parallel learning with PS

Parameter Server

$W_1$ $W_2$ $W_3$

Parameter Server

$W_4$ $W_5$ $W_6$

Parameter Server

$W_7$ $W_8$ $W_9$

Split Data Across Machines
Data parallel learning with PS

1. Different parts of the model on different servers.
2. Workers retrieve the part needed as needed

Split Data Across Machines
Abstraction used for Data parallel learning with PS

Key-Value API for workers:

1. `get(key) \rightarrow value`

\[ \delta_i \leftarrow f(x_i, \text{Model}) \]

2. `add(key, delta)`

\[ \text{Model} \leftarrow \text{Model} \oplus \delta_i \]
Parameter Servers

Stale Synchronous Parallel Model

(slides courtesy of Aurick Qiao, Joseph Gonzalez, Wei Dai, and Jinliang Wei)
Parameter Server (PS)

- Model parameters are stored on PS machines and accessed via key-value interface (distributed shared memory)

[Smola et al 2010, Ho et al 2013, Li et al 2014]
<table>
<thead>
<tr>
<th>Data Model</th>
<th>Independent Records</th>
<th>Parameter Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming Abstraction</td>
<td>Map &amp; Reduce</td>
<td>Independent Data</td>
</tr>
<tr>
<td>Execution Semantics</td>
<td>Bulk Synchronous Parallel (BSP)</td>
<td>Key-Value Store (Distributed Shared Memory)</td>
</tr>
</tbody>
</table>
The Problem: Networks Are Slow!

➢ Network is slow compared to local memory access
➢ We want to explore options for handling this....

[Smola et al 2010, Ho et al 2013, Li et al 2014]
Solution 1: Cache Synchronization
Parameter Cache Synchronization

Sparse Changes to Model
Parameter Cache Synchronization (aka IPM)
Solution 2: Asynchronous Execution

Enable more frequent coordination on parameter values
Asynchronous Execution

Parameter Server (Logical)

Machine 1

Machine 2

Machine 3

[Smola et al 2010]
Asynchronous Execution

Problem:
Async lacks theoretical guarantee as distributed environment can have arbitrary delays from network & stragglers

But....
1. Take a gradient step: \( x' = x_t - \eta_t g_t \)
2. If you’ve restricted the parameters to a subspace \( X \) (e.g., must be positive, ...) find the closest thing in \( X \) to \( x' \): \( x_{t+1} = \text{argmin}_X \text{dist}(x - x') \)
3. But.... you might be using a “stale” \( g \) (from \( \tau \) steps ago)

**Algorithm 1 Delayed Stochastic Gradient Descent**

**Input:** Feasible space \( X \subseteq \mathbb{R}^n \), annealing schedule \( \eta_t \), delay \( \tau \in \mathbb{N} \)

**Initialization:** set \( x_1 \ldots, x_{\tau} = 0 \) and compute corresponding gradients \( g_t = \nabla f_t(x_t) \).

**for** \( t = \tau + 1 \) **to** \( T + \tau \) **do**
- Obtain \( f_t \) and incur loss \( f_t(x_t) \)
- Compute \( g_t := \nabla f_t(x_t) \)
- Update \( x_{t+1} = \text{argmin}_{x \in X} \|x - (x_t - \eta_t g_{t-\tau})\| \) (Gradient Step and Projection)

**end for**
Stale synchronous parallel (SSP):
• Global clock time $t$
• Parameters workers “get” can be out of date
  but can’t be older than $t-\tau$
• $\tau$ controls “staleness”
• aka stale synchronous parallel (SSP)
Stale Synchronous Parallel (SSP)

➢ Interpolate between BSP and Async and subsumes both
➢ Allow workers to usually run at own pace
➢ Fastest/slowest threads not allowed to drift >s clocks apart
➢ Efficiently implemented: Cache parameters

[Ho et al 2013]
Suitable delay (SSP) gives big speed-up
Stale Synchronous Parallel (SSP)

LDA on NYtimes Dataset
LDA 32 machines (256 cores), 10% docs per iter

Log-Likelihood vs. Seconds

[BSP (stale 0)]
[stale 32]
[async]

[Ho et al 2013]
Big LDA on Parameter Server

- Collapsed Gibbs sampler
- Size: 50B tokens, 2000 topics, 5M words in vocab
- 1k~6k nodes

[Li et al 2014]
Case Study: Topic Modeling with LDA
Example: Topic Modeling with LDA

- Word-Topic Dist. \( \beta_t \)
  - \( t \in \{1, \ldots, T\} \)
  - Maintained by the Parameter Server

- Local Variables: Documents
  - Tokens \( x_i \)
  - \( i \in \{1, \ldots, \text{Len}(d)\} \)
  - \( d \in \{1, \ldots, D\} \)
  - Maintained by the Workers Nodes

- Tokens \( z_i \)
  - \( i \in \{1, \ldots, \text{Len}(d)\} \)
  - \( d \in \{1, \ldots, D\} \)
**Gibbs Sampling for LDA**

**Title:** Oh, The Places You’ll Go!

**Doc-Topic Distribution** $\theta_d$

- $z_1$
- $z_2$
- $z$
- $z_4$
- $z_5$
- $z_6$
- $z_7$

**Word-Topic Counts**

- **Brains:**
- **Choose:**
- **Direction:**
- **Feet:**
- **Head:**
- **Shoes:**
- **Steer:**

*You have brains in your head.*
*You have feet in your shoes.*
*You can steer yourself any direction you choose.*
Ex: Collapsed Gibbs Sampler for LDA

Partitioning the model and data

Parameter Server: $W_{1:10K}$
Parameter Server: $W_{10k:20K}$
Parameter Server: $W_{20k:30K}$
Ex: Collapsed Gibbs Sampler for LDA

Get model parameters and compute update

Parameter Server

Car

1:1

Cat

Parameter Server

W

Tire

20K

Parameter Server

W

Mouse

30K

get('car')

get('cat')

get('tire')

get('mouse')
Ex: Collapsed Gibbs Sampler for LDA

Send changes back to the parameter server

Parameter Server

\[ W_{1:10K} \]

\[ \text{add("car", 5)} \]

Parameter Server

\[ W_{10k:20K} \]

\[ \text{add("hire", 5)} \]

Parameter Server

\[ W_{20k:30K} \]

\[ \text{add("mouse", 6)} \]
Beyond the PS/SSP Abstraction...
Network loads for PS/SSP

Existing:

• How can we use network capacity better?
  • Maybe tell the system a little more about what the problem we’re solving is so it can manage communication better

[Wei et al 2015]
Ways To Manage Communication

• Model parameters are not equally important
  • E.g. Majority of the parameters may converge in a few iteration.

• Communicate the more important parameter values or updates
  • Magnitude of the changes indicates importance

• Magnitude-based prioritization strategies
  • Example: Relative-magnitude prioritization

[Wei et al 2015]

We saw many of these ideas in the signal/collect paper
Iterative ML Algorithms

Many ML algorithms are iterative-convergent

A: params at time $t$

F: update

$L$: loss

$\Delta$: grad

D: data

Examples: Optimization, sampling methods

Topic Model, matrix factorization, SVM, Deep Neural Network...
Iterative ML with a Parameter Server: (1) Data Parallel

\[ A(t) = F(A(t-1), \sum_{p=1}^{P} \Delta(A(t-1), D_p) \]
(2) Model parallel

$A^{(t)} = F\left( A^{(t-1)}, \{ \Delta(A^{(t-1)}, S_p^{(t-1)}(A^{(t-1)})) \}^{P}_{p=1} \right)$

$S^{(t-1)}_p()$ outputs a set of indices $\{j_1, j_2, \ldots, \}$

ignore D as well as L

$S_p$ is a scheduler for processor $p$ that selects params for $p$
Optional scheduling interface

1. schedule(key) → param keys svars

2. push(p=workerId, svars) → changed key

3. pull(svars, updates=(push₁,...,pushₙ))

- signal: broadcast changes to PS
- collect: aggregate changes from PS
Support for model-parallel programs
similar to signal-collect: schedule() defines graph, workers push params to scheduler, scheduler pulls to aggregate, and makes params available via get() and inc()

```c
// Petuum Program Structure

schedule() {
    // This is the (optional) scheduling function
    // It is executed on the scheduler machines
    A_local = PS.get(A) // Parameter server read
    PS.inc(A,change) // Can write to PS here if needed
    // Choose variables for push() and return
    svars = my_scheduling(DATA,A_local)
    return svars
}

push(p = worker_id(), svars = schedule()) {
    // This is the parallel update function
    // It is executed on each of P worker machines
    A_local = PS.get(A) // Parameter server read
    // Perform computation and send return values to pull()
    // Or just write directly to PS
    change1 = my_update1(DATA,p,A_local)
    change2 = my_update2(DATA,p,A_local)
    PS.inc(A,change1) // Parameter server increment
    return change2
}

pull(svars = schedule(), updates = (push(1), ..., push(P))) {
    // This is the (optional) aggregation function
    // It is executed on the scheduler machines
    A_local = PS.get(A) // Parameter server read
    // Aggregate updates from push(1..P) and write to PS
    my_aggregate(A_local,updates)
    PS.put(A,change) // Parameter server overwrite
}```
A Model Parallel Example:
Lasso
Regularized logistic regression

Replace log conditional likelihood $LCL$ with $LCL + \text{penalty}$ for large weights, eg

$$ \log P(Y = y|X = x, w) = \begin{cases} \log p & \text{if } y = 1 \\ \log(1 - p) & \text{if } y = 0 \end{cases} $$

$$ LCL - \mu \sum_{j=1}^{\mu} (w^j)^2 = LCL - \mu \|w\|_2 $$

Alternative penalty:

$$ LCL - \mu \sum_{j=1} |w^j| = LCL - \mu \|w\|_1 $$
Regularized logistic regression

\[ LCL - \mu \sum_{i=1}^{i-1} (w^i)^2 = LCL - \mu \|w\|_2 \]

\[ LCL - \mu \sum_{j=1}^{j} |w^j| = LCL - \mu \|w\|_1 \]

L1-regularization pushes parameters to zero: \textbf{sparse}

shallow grad near 0

steep grad near 0
SGD

Repeat for $t=1,\ldots,T$

» For each example

• Compute gradient of regularized loss (for that example)
  – Move all parameters in that direction (a little)
Parallel stochastic coordinate descent (shotgun)

Repeat for $t=1,\ldots,T$

» Pick several coordinates $j_1,\ldots,j_p$ \textbf{in parallel}

• Compute gradient of regularized loss (for each parameter $j_k$)
  
  – Move each parameter $j_k$
Parallel coordinate descent (shotgun)

Algorithm 2 Shotgun: Parallel SCD

Choose number of parallel updates $P \geq 1$.
Set $x = 0 \in \mathbb{R}^{2d}$
while not converged do
    Choose random subset of $P$ weights in $\{1, \ldots, 2d\}$.
    In parallel on $P$ processors
        Get assigned weight $j$.
        Set $\delta x_j \leftarrow \max\{-x_j, -(\nabla F(x))_j / \beta\}$.
        Update $x_j \leftarrow x_j + \delta x_j$.
end while
shotgun works best when you select uncorrelated parameters to process in parallel
Example: Model parallel SGD

Basic ideas:

- Pick parameters stochastically
- Prefer large parameter values (i.e., ones that haven’t converged)
- Prefer nearly-independent parameters
// Model-Parallel Lasso

schedule() {
    for j=1..J // Update priorities for all coeffs beta_j
        c_j = square(beta_j) + eta // Magnitude prioritization
        (s_1, ..., s_L’) = random_draw(distribution(c_1, ..., c_J))
    // Choose L< L’ pairwise-independent beta_j
    (j_1, ..., j_L) = correlation_check(s_1, ..., s_L’)
    return (j_1, ..., j_L)
}

push(p = worker_id(), (j_1, ..., j_L) = schedule()) {
    // Partial computation for L chosen beta_j; calls PS.get(beta)
    (z_p[j_1], ..., z_p[j_L]) = partial(DATA[p], j_1, ..., j_L)
    return z_p
}

pull((j_1, ..., j_L) = schedule(),
     (z_1, ..., z_P) = (push(1), ..., push(P))) {
    for a=1..L // Aggregate partial computation from P workers
        newval = sum_threshold(z_1[j_a], ..., z_P[j_a])
        PS.put(beta[j_a], newval) // Overwrite to parameter server
    }
