Large-Scale Matrix Factorization

Rainer Gemulla

November 23, 2012

P. J. Haas    Y. Sismanis    E. Nijkamp    C. Teflioudi    F. Makari
Outline

Matrix Factorization

Stochastic Gradient Descent

Distributed SGD with MapReduce

Experiments

Summary
Outline

Matrix Factorization

Stochastic Gradient Descent

Distributed SGD with MapReduce

Experiments

Summary
Collaborative Filtering

- Problem
  - Set of users
  - Set of items (movies, books, jokes, products, stories, ...)
  - Feedback (ratings, purchase, click-through, tags, ...)

Example
- Netflix competition: 500k users, 20k movies, 100M movie ratings, 3M question marks
Collaborative Filtering

- Problem
  - Set of users
  - Set of items (movies, books, jokes, products, stories, ...)
  - Feedback (ratings, purchase, click-through, tags, ...)

- Predict additional items a user may like
  - Assumption: Similar feedback $\implies$ Similar taste
Collaborative Filtering

▶ Problem
  ▶ Set of users
  ▶ Set of items (movies, books, jokes, products, stories, ...)
  ▶ Feedback (ratings, purchase, click-through, tags, ...)

▶ Predict additional items a user may like
  ▶ Assumption: Similar feedback $\implies$ Similar taste

▶ Example

<table>
<thead>
<tr>
<th></th>
<th>Avatar</th>
<th>The Matrix</th>
<th>Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Bob</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Charlie</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Collaborative Filtering

- Problem
  - Set of users
  - Set of items (movies, books, jokes, products, stories, ...)
  - Feedback (ratings, purchase, click-through, tags, ...)
- Predict additional items a user may like
  - Assumption: Similar feedback $\implies$ Similar taste
- Example

<table>
<thead>
<tr>
<th></th>
<th>Avatar</th>
<th>The Matrix</th>
<th>Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>?</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Bob</td>
<td>3</td>
<td>2</td>
<td>?</td>
</tr>
<tr>
<td>Charlie</td>
<td>5</td>
<td>?</td>
<td>3</td>
</tr>
</tbody>
</table>
Collaborative Filtering

- **Problem**
  - Set of users
  - Set of items (movies, books, jokes, products, stories, ...)
  - Feedback (ratings, purchase, click-through, tags, ...)

- **Predict additional items a user may like**
  - Assumption: Similar feedback $\implies$ Similar taste

- **Example**

<table>
<thead>
<tr>
<th></th>
<th>Avatar</th>
<th>The Matrix</th>
<th>Up</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Alice</strong></td>
<td>?</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td><strong>Bob</strong></td>
<td>3</td>
<td>2</td>
<td>?</td>
</tr>
<tr>
<td><strong>Charlie</strong></td>
<td>5</td>
<td>?</td>
<td>3</td>
</tr>
</tbody>
</table>

- Netflix competition: 500k users, 20k movies, 100M movie ratings, 3M question marks
Semantic Factors (Koren et al., 2009)

A simplified illustration of the latent factor approach, which characterizes both users and movies using two axes—male versus female and serious versus escapist.

The major challenge is computing the mapping of each item and user to factor vectors whose magnitudes are penalized. The constant $\kappa$ controls $\sqrt{\frac{\kappa}{2}}$ pairs for which $\kappa$ is known, such that $\kappa$ is the set of the $(r_{ui})$ pairs for which $\kappa$ is known. Thus, the system should avoid overfitting the previous ratings in a way that predicts future, unknown observed ratings. However, the goal is to generalize those previous ratings to complete this mapping, it can easily estimate $r_{ui}$, leading to the estimate:

$$
\hat{r}_{ui} = \mathbf{q}_i^T \mathbf{p}_u + \nu + \epsilon
$$

where $\mathbf{q}_i$ and $\mathbf{p}_u$ are the factor vectors associated with item $i$ and user $u$, respectively. The constants $\nu$ and $\epsilon$ denote, respectively, the user's overall interest in the item's characteristics. This approximates the rating a user will give to any item which the item possesses those factors, measure the extent of interest the user has in items that are high positive or negative. The resulting dot product, generally, can be very expensive as it significantly increases computation can be very expensive as it significantly increases inaccuracy imputation.

For a given item $f$, and each user $u$ is associated with a vector $p_u \in \mathcal{F}$.

Recommender systems rely on different types of feedback. Usually, explicit feedback comprises user preferences using thumbs-up and thumbs-down buttons. We refer to explicit user feedback as ***basic matrix factorization model***. To learn the factor vectors $(\mathbf{q}_i, \mathbf{p}_u)$, the system learns the model by fitting the previously observed data by regularizing the learned parameters, which might distort the data considerably. Hence, more recent works suggested modeling directly the observed ratings. However, the goal is to generalize those observed ratings. Therefore, the amount of data. In addition, inaccurate imputation is high-quality complete. Moreover, carelessly addressing only the relatively few known entries is highly prone to overfitting. Thus, the system should avoid overfitting the model by using Equation 1.

$$
\min_{\mathbf{Q}, \mathbf{P}} \sum_{(i,j) \in \mathcal{D}} \left( r_{ij} - \mathbf{q}_i^T \mathbf{p}_j \right)^2 + \lambda \left( \| \mathbf{Q} \|_F^2 + \| \mathbf{P} \|_F^2 \right)
$$

Here, $\mathbf{Q}$ and $\mathbf{P}$ are the factor matrices, $\mathbf{Q} \in \mathcal{R}^{N \times d}$ represents items of interest, and $\mathbf{P} \in \mathcal{R}^{M \times d}$ represents users and the other dimension characterizes both users and movies using two axes—male versus female and serious versus escapist.
### Latent Factor Models

- Discover latent factors \((r = 1)\)

<table>
<thead>
<tr>
<th></th>
<th>Avatar</th>
<th>The Matrix</th>
<th>Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Bob</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Charlie</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Latent Factor Models

- Discover latent factors \( r = 1 \)

<table>
<thead>
<tr>
<th></th>
<th>Avatar (2.24)</th>
<th>The Matrix (1.92)</th>
<th>Up (1.18)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>4</td>
<td>1.98</td>
<td>2</td>
</tr>
<tr>
<td>Bob</td>
<td>3</td>
<td>1.21</td>
<td>2</td>
</tr>
<tr>
<td>Charlie</td>
<td>5</td>
<td>2.30</td>
<td>3</td>
</tr>
</tbody>
</table>
Latent Factor Models

- Discover latent factors ($r = 1$)

<table>
<thead>
<tr>
<th></th>
<th>Avatar (2.24)</th>
<th>The Matrix (1.92)</th>
<th>Up (1.18)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td></td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>(1.98)</td>
<td></td>
<td>(3.8)</td>
<td>(2.3)</td>
</tr>
<tr>
<td>Bob</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>(1.21)</td>
<td>(2.7)</td>
<td>(2.3)</td>
<td></td>
</tr>
<tr>
<td>Charlie</td>
<td>5</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>(2.30)</td>
<td>(5.2)</td>
<td></td>
<td>(2.7)</td>
</tr>
</tbody>
</table>

- Minimum loss

$$\min_{W, H} \sum_{(i,j) \in Z} (V_{ij} - [WH]_{ij})^2$$
Latent Factor Models

- Discover latent factors ($r = 1$)

<table>
<thead>
<tr>
<th></th>
<th>Avatar (2.24)</th>
<th>The Matrix (1.92)</th>
<th>Up (1.18)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice (1.98)</td>
<td>?</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>(4.4)</td>
<td>(3.8)</td>
<td>(2.3)</td>
</tr>
<tr>
<td>Bob (1.21)</td>
<td>3</td>
<td>2</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>(2.7)</td>
<td>(2.3)</td>
<td>(1.4)</td>
</tr>
<tr>
<td>Charlie (2.30)</td>
<td>5</td>
<td>?</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>(5.2)</td>
<td>(4.4)</td>
<td>(2.7)</td>
</tr>
</tbody>
</table>

- Minimum loss

$$\min_{W,H} \sum_{(i,j) \in Z} (V_{ij} - [WH]_{ij})^2$$
Latent Factor Models

- Discover latent factors \((r = 1)\)

<table>
<thead>
<tr>
<th></th>
<th>Avatar (2.24)</th>
<th>The Matrix (1.92)</th>
<th>Up (1.18)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>?</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>(1.98)</td>
<td>(4.4)</td>
<td>(3.8)</td>
</tr>
<tr>
<td></td>
<td>(1.18)</td>
<td>(2.3)</td>
<td></td>
</tr>
<tr>
<td>Bob</td>
<td>3</td>
<td>2</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>(1.21)</td>
<td>(2.7)</td>
<td>(1.4)</td>
</tr>
<tr>
<td>Charlie</td>
<td>5</td>
<td>?</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>(2.30)</td>
<td>(5.2)</td>
<td>(4.4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(2.7)</td>
<td></td>
</tr>
</tbody>
</table>

- Minimum loss

\[
\min_{W,H,u,m} \sum_{(i,j) \in Z} (V_{ij} - \mu - u_i - m_j - [WH]_{ij})^2
\]

- Bias
Latent Factor Models

- Discover latent factors ($r = 1$)

<table>
<thead>
<tr>
<th></th>
<th>Avatar (2.24)</th>
<th>The Matrix (1.92)</th>
<th>Up (1.18)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>1.98</td>
<td>(4.4)</td>
<td>(3.8)</td>
</tr>
<tr>
<td>Bob</td>
<td>1.21</td>
<td>(2.7)</td>
<td>(2.3)</td>
</tr>
<tr>
<td>Charlie</td>
<td>2.30</td>
<td>(5.2)</td>
<td>(4.4)</td>
</tr>
</tbody>
</table>

- Minimum loss

$$\min_{W,H,u,m} \sum_{(i,j) \in Z} (V_{ij} - \mu - u_i - m_j - [WH]_{ij})^2$$

$$+ \lambda (\|W\| + \|H\| + \|u\| + \|m\|)$$

- Bias, regularization
Latent Factor Models

- Discover latent factors \( (r = 1) \)

<table>
<thead>
<tr>
<th></th>
<th>Avatar</th>
<th>The Matrix</th>
<th>Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>?</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>(1.98)</td>
<td>(3.8)</td>
<td>(2.3)</td>
</tr>
<tr>
<td>Bob</td>
<td>3</td>
<td>2</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>(1.21)</td>
<td>(2.3)</td>
<td>(1.4)</td>
</tr>
<tr>
<td>Charlie</td>
<td>5</td>
<td>?</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>(2.30)</td>
<td>(4.4)</td>
<td>(2.7)</td>
</tr>
</tbody>
</table>

- Minimum loss

\[
\min_{W,H,u,m} \sum_{(i,j,t) \in Z_t} (V_{ij} - \mu - u_i(t) - m_j(t) - [W(t)H]_{ij})^2 \\
+ \lambda (\|W(t)\| + \|H\| + \|u(t)\| + \|m(t)\|)
\]

- Bias, regularization, time
Another Matrix
Matrix Reconstruction (unregularized)
Matrix Reconstruction (unregularized)
Matrix Reconstruction (unregularized)
Matrix Reconstruction (unregularized)
Latent Factor Models (unregularized)

Data

LFM
Latent Factor Models (unregularized)

Data

1% 10% 100%

LFM

SVD
Generalized Matrix Factorization

- A general machine learning problem
  - Recommender systems, text indexing, face recognition, ...
Generalized Matrix Factorization

- A general machine learning problem
  - Recommender systems, text indexing, face recognition, . . .
- Training data
  - $V$: $m \times n$ input matrix (e.g., rating matrix)
  - $Z$: training set of indexes in $V$ (e.g., subset of known ratings)
Generalized Matrix Factorization

- A general machine learning problem
  - Recommender systems, text indexing, face recognition, ... 
- Training data
  - $V$: $m \times n$ input matrix (e.g., rating matrix)
  - $Z$: training set of indexes in $V$ (e.g., subset of known ratings)
- Parameter space
  - $W$: row factors (e.g., $m \times r$ latent customer factors)
  - $H$: column factors (e.g., $r \times n$ latent movie factors)

$$L_{ij}(W_i^*, H_j^*): \text{loss at element } (i, j)$$

- Includes prediction error, regularization, auxiliary information, ...
- Constraints (e.g., non-negativity)

Find best model

$$\arg\min_{W, H} \sum_{(i, j) \in Z} L_{ij}(W_i^*, H_j^*)$$
Generalized Matrix Factorization

- A general machine learning problem
  - Recommender systems, text indexing, face recognition, ...
- Training data
  - $\mathbf{V}$: $m \times n$ input matrix (e.g., rating matrix)
  - $\mathbf{Z}$: training set of indexes in $\mathbf{V}$ (e.g., subset of known ratings)
- Parameter space
  - $\mathbf{W}$: row factors (e.g., $m \times r$ latent customer factors)
  - $\mathbf{H}$: column factors (e.g., $r \times n$ latent movie factors)
- Model
  - $L_{ij}(\mathbf{W}_{i*}, \mathbf{H}_{*j})$: loss at element $(i, j)$
  - Includes prediction error, regularization, auxiliary information, ...
  - Constraints (e.g., non-negativity)
Generalized Matrix Factorization

- A general machine learning problem
  - Recommender systems, text indexing, face recognition, ...
- Training data
  - $\mathbf{V}$: $m \times n$ input matrix (e.g., rating matrix)
  - $\mathbf{Z}$: training set of indexes in $\mathbf{V}$ (e.g., subset of known ratings)
- Parameter space
  - $\mathbf{W}$: row factors (e.g., $m \times r$ latent customer factors)
  - $\mathbf{H}$: column factors (e.g., $r \times n$ latent movie factors)
- Model
  - $L_{ij}(\mathbf{W}_{i*}, \mathbf{H}_{*j})$: loss at element $(i, j)$
  - Includes prediction error, regularization, auxiliary information, ...
  - Constraints (e.g., non-negativity)
- Find best model

$$\operatorname{argmin}_{\mathbf{W}, \mathbf{H}} \sum_{(i,j) \in \mathbf{Z}} L_{ij}(\mathbf{W}_{i*}, \mathbf{H}_{*j})$$
Successful Applications

- Movie recommendation (Netflix)
  - >20M users, >20k movies, 4B ratings (projected)
  - 60GB data, 15GB model (projected)
  - Collaborative filtering

- Website recommendation (Microsoft, WWW10)
  - 51M users, 15M URLs, 1.2B clicks
  - 17.8GB data, 161GB metadata, 49GB model
  - Gaussian non-negative matrix factorization

- News personalization (Google, WWW07)
  - Millions of users, millions of stories, ? clicks
  - Probabilistic latent semantic indexing
Successful Applications

- Movie recommendation (Netflix)
  - >20M users, >20k movies, 4B ratings (projected)
  - 60GB data, 15GB model (projected)
  - Collaborative filtering

- Website recommendation (Microsoft, WWW10)
  - 51M users, 15M URLs, 1.2B clicks
  - 17.8GB data, 161GB metadata, 49GB model
  - Gaussian non-negative matrix factorization

- News personalization (Google, WWW07)
  - Millions of users, millions of stories, ? clicks
  - Probabilistic latent semantic indexing

How to handle such massive scale?

- Big data
- Large models
- Expensive, iterative computations
Outline

Matrix Factorization

Stochastic Gradient Descent

Distributed SGD with MapReduce

Experiments

Summary
Stochastic Gradient Descent

- Find minimum $\theta^*$ of function $L$

![Diagram](attachment:image.png)

Stochastic difference equation

$$
\theta_{n+1} = \theta_n - \epsilon_n \hat{L}'(\theta_n)
$$

Under certain conditions, asymptotically approximates (continuous) gradient descent
Stochastic Gradient Descent

- Find minimum $\theta^*$ of function $L$
- Pick a starting point $\theta_0$
Stochastic Gradient Descent

- Find minimum $\theta^*$ of function $L$
- Pick a starting point $\theta_0$
Stochastic Gradient Descent

- Find minimum $\theta^*$ of function $L$
- Pick a starting point $\theta_0$
Stochastic Gradient Descent

- Find minimum $\theta^*$ of function $L$
- Pick a starting point $\theta_0$
- Approximate gradient $\hat{L}'(\theta_0)$
Stochastic Gradient Descent

- Find minimum $\theta^*$ of function $L$
- Pick a starting point $\theta_0$
- Approximate gradient $\hat{L}'(\theta_0)$
- Jump “approximately” downhill
Stochastic Gradient Descent

- Find minimum $\theta^*$ of function $L$
- Pick a starting point $\theta_0$
- Approximate gradient $\hat{L}'(\theta_0)$
- Jump “approximately” downhill
- Stochastic difference equation

$$\theta_{n+1} = \theta_n - \epsilon_n \hat{L}'(\theta_n)$$
Stochastic Gradient Descent

- Find minimum \( \theta^* \) of function \( L \)
- Pick a starting point \( \theta_0 \)
- Approximate gradient \( \hat{L}'(\theta_0) \)
- Jump “approximately” downhill
- Stochastic difference equation

\[
\theta_{n+1} = \theta_n - \epsilon_n \hat{L}'(\theta_n)
\]

- Under certain conditions, asymptotically approximates (continuous) gradient descent
Stochastic Gradient Descent for Matrix Factorization

▶ Set $\theta = (W, H)$ and use

$$L(\theta) = \sum_{(i,j) \in Z} L_{ij}(W_{i*}, H_{*j})$$

SGD epoch

1. Pick a random entry $z \in Z$
2. Compute approximate gradient $\hat{L}'(\theta, z)$
3. Update parameters $\theta_{n+1} = \theta_n - \epsilon_n \hat{L}'(\theta_n, z)$
4. Repeat $N$ times

Random data access patterns.
Stochastic Gradient Descent for Matrix Factorization

- Set $\theta = (W, H)$ and use

$$L(\theta) = \sum_{(i,j) \in Z} L_{ij}(W_{i*}, H_{*j})$$

$$L'(\theta) = \sum_{(i,j) \in Z} L'_{ij}(W_{i*}, H_{*j})$$
Stochastic Gradient Descent for Matrix Factorization

- Set $\theta = (W, H)$ and use

$$L(\theta) = \sum_{(i, j) \in Z} L_{ij}(W_{i*}, H_{*j})$$

$$L'(\theta) = \sum_{(i, j) \in Z} L'_{ij}(W_{i*}, H_{*j})$$

$$\hat{L}'(\theta, z) = NL'_{ij}(W_{i*}, H_{*j})$$

where $N = |Z|$
Stochastic Gradient Descent for Matrix Factorization

- Set $\theta = (W, H)$ and use

$$L(\theta) = \sum_{(i,j) \in Z} L_{ij}(W_{i*}, H_{*j})$$

$$L'(\theta) = \sum_{(i,j) \in Z} L'_{ij}(W_{i*}, H_{*j})$$

$$\hat{L}'(\theta, z) = NL'_{i_zj_z}(W_{i_z*}, H_{*j_z})$$,

where $N = |Z|$

- SGD epoch
  1. Pick a random entry $z \in Z$
  2. Compute approximate gradient $\hat{L}'(\theta, z)$
  3. Update parameters
     $$\theta_{n+1} = \theta_n - \epsilon_n \hat{L}'(\theta_n, z)$$
  4. Repeat $N$ times
Set $\theta = (W, H)$ and use

$$L(\theta) = \sum_{(i,j) \in Z} L_{ij}(W_{i*}, H_{*j})$$

$$L'(\theta) = \sum_{(i,j) \in Z} L'_{ij}(W_{i*}, H_{*j})$$

$$\hat{L}'(\theta, z) = NL'_{izjz}(W_{iz*}, H_{*jz})$$

where $N = |Z|$

**SGD epoch**

1. Pick a random entry $z \in Z$
2. Compute approximate gradient $\hat{L}'(\theta, z)$
3. Update parameters
   $$\theta_{n+1} = \theta_n - \epsilon_n \hat{L}'(\theta_n, z)$$
4. Repeat $N$ times

Random data access patterns.
Stochastic Gradient Descent on Netflix Data

![Graph showing Mean Loss vs. Epoch for different optimization methods: LBFGS, SGD, and ALS. The graph illustrates the convergence of the loss function over epochs, demonstrating the effectiveness of each method in minimizing the loss.]
Outline

Matrix Factorization

Stochastic Gradient Descent

Distributed SGD with MapReduce

Experiments

Summary
Problem Structure

- SGD steps depend on each other
  \[ \theta_{n+1} = \theta_n - \epsilon_n \hat{L}'(\theta_n) \]
- An SGD step on example \( z \in Z \) ...
  1. Reads \( W_{iz^*} \) and \( H_{*jz} \)
  2. Performs gradient computation \( L'_{ij}(W_{iz^*}, H_{*jz}) \)
  3. Updates \( W_{iz^*} \) and \( H_{*jz} \)
Problem Structure

- SGD steps depend on each other

\[ \theta_{n+1} = \theta_n - \epsilon_n \hat{L}'(\theta_n) \]

- An SGD step on example \( z \in Z \) …
  1. Reads \( W_{iz^*} \) and \( H_{jz^*} \)
  2. Performs gradient computation \( L'_{ij}(W_{iz^*}, H_{jz^*}) \)
  3. Updates \( W_{iz^*} \) and \( H_{jz^*} \)
Problem Structure

- SGD steps depend on each other
  \[
  \theta_{n+1} = \theta_n - \epsilon_n \hat{L}'(\theta_n)
  \]
- An SGD step on example \( z \in Z \) . . .
  1. Reads \( W_{iz*} \) and \( H_{*jz} \)
  2. Performs gradient computation \( L_{ij}'(W_{iz*}, H_{*jz}) \)
  3. Updates \( W_{iz*} \) and \( H_{*jz} \)

Synchronization provides an efficient shared-memory parallel SGD algorithm.
Problem Structure

- SGD steps depend on each other

\[ \theta_{n+1} = \theta_n - \epsilon_n \hat{L}'(\theta_n) \]

- An SGD step on example \( z \in Z \) …
  1. Reads \( W_{i_z^*} \) and \( H_{*j_z} \)
  2. Performs gradient computation \( L'_{ij}(W_{i_z^*}, H_{*j_z}) \)
  3. Updates \( W_{i_z^*} \) and \( H_{*j_z} \)

- Not all steps are dependent
Problem Structure

- SGD steps depend on each other

\[ \theta_{n+1} = \theta_n - \epsilon_n \hat{L}'(\theta_n) \]

- An SGD step on example \( z \in Z \ldots \)
  1. Reads \( W_{i_z^*} \) and \( H_{j_z^*} \)
  2. Performs gradient computation \( L'_{ij}(W_{i_z^*}, H_{j_z^*}) \)
  3. Updates \( W_{i_z^*} \) and \( H_{j_z^*} \)

- Not all steps are dependent

Synchronization provides an efficient shared-memory parallel SGD algorithm.
Exploitation in MapReduce (DSGD: WWW11, Biglearn11)

- Block and distribute the input matrix $V$

```
Node 1
  W1  V11  V12  V13
  H1

Node 2
  W2  V21  V22  V23
  H2

Node 3
  W3  V31  V32  V33
  H3
```
Exploitation in MapReduce (DSGD: WWW11, Biglearn11)

- Block and distribute the input matrix $\mathbf{V}$
- High-level approach (Map only)
  1. Pick a “diagonal”
  2. Run SGD on the diagonal (in parallel)
  3. Merge the results
  4. Move on to next “diagonal”

- Steps 1–3 form a cycle

Node 1

$W_1$
$V_{11}$ $V_{12}$ $V_{13}$

Node 2

$W_2$
$V_{21}$ $V_{22}$ $V_{23}$

Node 3

$W_3$
$V_{31}$ $V_{32}$ $V_{33}$
Exploitation in MapReduce (DSGD: WWW11, Biglearn11)

- Block and distribute the input matrix $V$
- High-level approach (Map only)
  1. Pick a “diagonal”
  2. Run SGD on the diagonal (in parallel)
  3. Merge the results
  4. Move on to next “diagonal”

Steps 1–3 form a cycle

1. Node 1
2. Node 2
3. Node 3
Block and distribute the input matrix $V$.

High-level approach (Map only):
1. Pick a “diagonal”
2. Run SGD on the diagonal (in parallel)
3. Merge the results
4. Move on to next “diagonal”

Steps 1–3 form a cycle.

Step 2: Simulate sequential SGD
- Interchangeable blocks
- Throw dice of how many iterations per block
- Throw dice of which step sizes per block
Exploitation in MapReduce (DSGD: WWW11, Biglearn11)

- Block and distribute the input matrix $V$

- High-level approach (Map only)
  1. Pick a “diagonal”
  2. Run SGD on the diagonal (in parallel)
  3. Merge the results
  4. Move on to next “diagonal”

  - Steps 1–3 form a cycle

- Step 2:
  Simulate sequential SGD

  - Interchangeable blocks
  - Throw dice of how many iterations per block
  - Throw dice of which step sizes per block
Exploitation in MapReduce (DSGD: WWW11, Biglearn11)

- Block and distribute the input matrix $V$
- High-level approach (Map only)
  1. Pick a “diagonal”
  2. Run SGD on the diagonal (in parallel)
  3. Merge the results
  4. Move on to next “diagonal”

- Steps 1–3 form a cycle

- Step 2:
  Simulate sequential SGD
  - Interchangeable blocks
  - Throw dice of how many iterations per block
  - Throw dice of which step sizes per block

![Diagram of nodes and matrices](image)
Exploitation in MapReduce (DSGD: WWW11, Biglearn11)

- Block and distribute the input matrix \( \mathbf{V} \)
- High-level approach (Map only)
  1. Pick a “diagonal”
  2. Run SGD on the diagonal (in parallel)
  3. Merge the results
  4. Move on to next “diagonal”

  - Steps 1–3 form a cycle

- Step 2:
  Simulate sequential SGD
  - Interchangeable blocks
  - Throw dice of how many iterations per block
  - Throw dice of which step sizes per block

- Instance of “stratified SGD”
- Provably correct
How does it work?

$L = 0.3L_1 + 0.7L_2$

Cycle 0
How does it work?

$L = 0.3L_1 + 0.7L_2$

Cycle 1
How does it work?

$L = 0.3L_1 + 0.7L_2$

Cycle 2
How does it work?

$L = 0.3L_1 + 0.7L_2$

Cycle 3
How does it work?

$L_1$ and $L_2$ are represented in the plots.

Cycle 4

$L = 0.3L_1 + 0.7L_2$
How does it work?

$L = 0.3L_1 + 0.7L_2$

Cycle 5
How does it work?

\[ L = 0.3L_1 + 0.7L_2 \]
How does it work?

$L_1 = 0.3L_1 + 0.7L_2$

Cycle 100
How does it work?

$L = 0.3L_1 + 0.7L_2$

Cycle 100
How does it work?

$L = 0.3L_1 + 0.7L_2$

Cycle 100
Beyond MapReduce (DSGD++: ICDM12)

Can we do better in an MPI environment (i.e., shared nothing)?
Can we do better in an MPI environment (i.e., shared nothing)?

Yes, with careful engineering.
Beyond MapReduce (DSGD++: ICDM12)

Can we do better in an MPI environment (i.e., shared nothing)?

Yes, with careful engineering.

- Prefetch data/parameters for next SGD step(s)
Can we do better in an MPI environment (i.e., shared nothing)?

Yes, with careful engineering.

- Prefetch data/parameters for next SGD step(s)
- Exploit multi-core
Can we do better in an MPI environment (i.e., shared nothing)?

Yes, with careful engineering.

- Prefetch data/parameters for next SGD step(s)
- Exploit multi-core

---

Node 1 (yellow)

Node 2 (green)

Node 3 (blue)
Can we do better in an MPI environment (i.e., shared nothing)?

Yes, with careful engineering.

- Prefetch data/parameters for next SGD step(s)
- Exploit multi-core
Beyond MapReduce (DSGD++: ICDM12)

Can we do better in an MPI environment (i.e., shared nothing)?

Yes, with careful engineering.

- Prefetch data/parameters for next SGD step(s)
- Exploit multi-core
Beyond MapReduce (DSGD++: ICDM12)

Can we do better in an MPI environment (i.e., shared nothing)?

Yes, with careful engineering.

▶ Prefetch data/parameters for next SGD step(s)
▶ Exploit multi-core
▶ Directly communicate parameters between nodes
Beyond MapReduce (DSGD++: ICDM12)

Can we do better in an MPI environment (i.e., shared nothing)?

Yes, with careful engineering.

- Prefetch data/parameters for next SGD step(s)
- Exploit multi-core
- Directly communicate parameters between nodes
- Overlay subepochs
Can we do better in an MPI environment (i.e., shared nothing)?

Yes, with careful engineering.

- Prefetch data/parameters for next SGD step(s)
- Exploit multi-core
- Directly communicate parameters between nodes
- Overlay subepochs
- Overlay computation and communication
Outline

Matrix Factorization

Stochastic Gradient Descent

Distributed SGD with MapReduce

Experiments

Summary
Setup

- Small blade cluster
  - 16 compute nodes
  - Intel Xeon E5530, 8 cores, 2.4GHz
  - 48GB memory

- All algorithms implemented in C++ and MPI
  - Alternating least squares (ALS)
  - Stochastic gradient descent (SGD)
  - Parallel ALS (PALS)
  - Parallel SGD (PSGD)
  - Distributed ALS (DALS)
  - Asynchronous SGD (ASGD)
  - Distributed SGD (DSGD-MR)
  - Distributed SGD++ (DSGD++)

- Datasets
  - Netflix (480k × 18k, 99M entries)
  - KDD (1M × 625k, 253M entries)
  - Synthetic (varying size, 1B–10B entries)
Example: Netflix data, 4x8 (relatively small, few items)
Example: Netflix data, 4x8 (relatively small, few items)

MapReduce algorithms slow; ASGD best, DSGD++ close.
Example: KDD data, 4x8 (moderately large, many items)
Example: KDD data, 4x8 (moderately large, many items)

DSGD++ best, ALS competitive.
Strong scalability: Large syn. data (10M × 1M, 1B entries)

- DSGD++ fastest, best scalability.
- (DALS converged to bad solution.)
Strong scalability: Large syn. data (10M × 1M, 1B entries)

DSGD++ fastest, best scalability.

(DALS converged to bad solution.)
Strong scalability: Huge syn. data (10M × 1M, 10B)

DSGD++ faster on 4 nodes than any other technique on 8 nodes.

ASGD converged to bad solution.

Insufficient memory
Strong scalability: Huge syn. data (10M \times 1M, 10B)

DSGD++ faster on 4 nodes than any other technique on 8 nodes.

(ASGD converged to bad solution.)
Outline

Matrix Factorization

Stochastic Gradient Descent

Distributed SGD with MapReduce

Experiments

Summary
Summary

- Matrix factorization
  - Currently best single approach for collaborative filtering
  - Widely applicable via customized loss functions
  - Large instances (millions × millions, billions of entries)

- Distributed Stochastic Gradient Descent
  - Simple and versatile
  - Fully distributed data/model
  - Fully distributed processing
  - Fast, good scalability

- DSGD++ variant for shared-nothing (and shared-memory) environments
Summary

- Matrix factorization
  - Currently best single approach for collaborative filtering
  - Widely applicable via customized loss functions
  - Large instances (millions × millions, billions of entries)

- Distributed Stochastic Gradient Descent
  - Simple and versatile
  - Fully distributed data/model
  - Fully distributed processing
  - Fast, good scalability

- DSGD++ variant for shared-nothing (and shared-memory) environments

Thank you!