Large-scale Matrix Factorization

Kijung Shin
Ph.D. Student, CSD

Carnegie Mellon University
Roadmap

• Matrix Factorization (review)

• Algorithms
  ◦ Distributed SGD: DSGD
  ◦ Alternating Least Square: ALS
  ◦ Cyclic Coordinate Descent: CCD++

• Experiments

• Extension to Tensor Factorization

• Conclusions
Roadmap

• **Matrix Factorization (review) <=**

• Algorithms
  ◦ Distributed SGD: DSGD
  ◦ Alternating Least Square: ALS
  ◦ Cyclic Coordinate Descent: CCD++

• Experiments

• Extension to Tensor Factorization

• Conclusions
Matrix Factorization: Problem

- **Given:**
  - $V$: $n$ by $m$ matrix
    - possibly with missing values
  - $r$: target rank (a scalar)
    - usually $r \ll n$ and $r \ll m$
Matrix Factorization: Problem

• Given:
  ◦ $V$: $n$ by $m$ matrix
    ▪ possibly with missing values
  ◦ $r$: target rank (a scalar)
    ▪ usually $r << n$ and $r << m$

• Find:
  ◦ $W$: $n$ by $r$ matrix
  ◦ $H$: $r$ by $m$ matrix
    ▪ without missing values
Matrix Factorization: Problem

• Goal: $WH \approx V$
Loss Function

\[ L(V, W, H) = \sum_{(i,j) \in Z} (V_{ij} - [WH]_{ij})^2 + \ldots \]

Indices of non-missing entries i.e., \((i, j) \in Z \leftrightarrow V_{ij} \text{ is not missing} \)

\((i, j)\)-th entry of \(V\)

\((i, j)\)-th entry of \(WH\)

Goal: to make \(WH\) similar to \(V\)
Loss Function (cont.)

\[ L(V, W, H) \]

\[ = \sum_{(i, j) \in Z} (V_{ij} - [WH]_{ij})^2 \]

\[ + \lambda \left( \|W\|_F^2 + \|H\|_F^2 \right) \]

**Regularization parameter**

Frobenius Norm:

\[ \|W\|_F^2 = \sum_{i=1}^{n} \sum_{k=1}^{r} (W_{ik})^2 \]

Goal: to prevent overfitting

(by making the entries of \(W\) and \(H\) close to zero)
Algorithms

• How can we minimize this loss function $L$?
  ◦ Stochastic Gradient Descent: SGD (covered in the last lecture)
  ◦ Alternating Least Square: ALS (covered today)
  ◦ Cyclic Coordinate Descent: CCD++ (covered today)

• Are these algorithms parallelizable?

• Yes, all of them!
Roadmap

• Matrix Factorization (review)

• Algorithms
  ◦ Distributed SGD: DSGD <<
  ◦ Alternating Least Square: ALS
  ◦ Cyclic Coordinate Descent: CCD++

• Experiments

• Extension to Tensor Factorization

• Conclusions
Distributed SGD (DSGD)

Large-scale matrix factorization with distributed stochastic gradient descent (KDD 2011)

Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis
Stochastic GD for MF (review)

• Let \( W = \begin{bmatrix} - & W_1 & - \\ - & \vdots & - \\ - & W_n & - \end{bmatrix}, \quad H = \begin{bmatrix} H_1 \\ \vdots \\ H_m \end{bmatrix} \)

• \( L(V, W, H) \): sum of loss for each non-missing entry

\[
L(V, W, H) = \sum_{(i,j) \in Z} L'(V_{ij}, W_{i:}, H_{j:})
\]

where loss at each non-missing entry \( V_{ij} \) is

\[
L'(V_{ij}, W_{i:}, H_{j:}) := (V_{ij} - W_{i:}H_{j:})^2 + \lambda \left( \frac{\|W_{i:}\|^2}{N_i} + \frac{\|H_{j:}\|^2}{N_{j:}} \right)
\]

= \([WH]_{ij} \)
Stochastic GD for MF (cont.)

• Stochastic Gradient Descent (SGD) for MF

• **repeat** until convergence
  ◦ randomly shuffle the non-missing entries of $V$
  ◦ **for each** non-missing entry:
    ▪ perform an SGD step on it
Stochastic GD for MF (cont.)

- **An SGD** step on each non-missing entry $V_{ij}$:
  - Read $W_i$ and $H_j$
  - Compute gradient of $L'(V_{ij}, W_i, H_j)$
  - Update $W_i$ and $H_j$
    - Detailed update rules were covered in the last lecture
Simple Parallel SGD for MF

- Parameter Mixing: **MSGD**
  - entries of $V$ are distributed across multiple machines

![Diagram of parameter mixing across machines](image)

**Machine 1**

**Machine 2**

**Machine 3**
Simple Parallel SGD for MF (cont.)

- Parameter Mixing: **MSGD**
  - entries of $V$ are distributed across multiple machines
  - **Map step**: each machine runs SGD independently on the assigned entries until convergence
Simple Parallel SGD for MF (cont.)

- Parameter Mixing: **MSGD**
  - **Map step**: each machine runs
    - an independent instance of SGD on subsets of the data
    - until convergence
  - **Reduce step**: average results (i.e., **$W$ and $H$**)

- Problem: does not converge to correct solutions
  - no guarantee that “the average” reduces the loss function $L$
Simple Parallel SGD for MF (cont.)

- **Iterative Parameter Mixing: ISGD**
  - entries of $V$ are distributed across multiple machines
  - **Repeat** until convergence
    - **Map step:** each machine runs SGD independently on the assigned entries for some time
    - **Reduce step:**
      - average results (i.e., $W$ and $H$)
      - broadcast averaged results

- **Problem:** slow convergence
  - still has the averaging step
Interchangeability

• How can we avoid the averaging step?
  ◦ let different machines update different entries of $W$ and $H$

• Two entries $V_{ij}$ and $V_{kl}$ of $V$ are *interchangeable* if they share neither row nor column

![Interchangeability Diagram]

Not interchangeable!

Interchangeable!
Interchangeability (cont.)

- If $V_{ij}$ and $V_{kl}$ are interchangeable,
- SGD steps on $V_{ij}$ and $V_{kl}$ can be parallelized safely

Read $W_i: \text{ and } H_{.j}$
Compute gradient of $L'(W_i:, H_{.j})$
Update $W_i: \text{ and } H_{.j}$

Read $W_k: \text{ and } H_{.l}$
Compute gradient of $L'(W_k:, H_{.l})$
Update $W_k: \text{ and } H_{.l}$

no conflicts!
Distributed SGD

• Block $W$, $H$, and $V$
Distributed SGD (cont.)

- Block $W$, $H$, and $V$
- **repeat** until convergence
  - **for** a set of interchangeable blocks of $V$
    - 1. run SGD on the blocks in parallel
    - no conflict between machines
    - 2. merge results (i.e., $W$ and $H$)
      - averaging is not needed

*Blocks on a diagonal are interchangeable!*
Interchangeable Blocks

- Example of interchangeable blocks

Blocks are interchangeable!
Interchangeable Blocks (cont.)

- Example of interchangeable blocks

Blocks are interchangeable!
Interchangeable Blocks (cont.)

• Example of interchangeable blocks

Blocks are interchangeable!
Interchangeable Blocks (cont.)

- **Not** interchangeable blocks
  - Not used in DSGD

Blocks are not interchangeable!
More details of DSGD

• How should we choose sets of interchangeable blocks?
  ◦ In each iteration, we choose sets of interchangeable blocks so that every block of $V$ is used exactly once
  ◦ E.g.,
    
    ![Diagram showing the use of blocks]
    
    ◦ But the order and grouping are random

• SGD within each block
  ◦ every non-missing entry in the block is used exactly once
  ◦ but the order is random
More details of DSGD (cont.)

- Use “bold driver” to set step size (or learning rate)
  - After each iteration,
    - increase the step size if the loss decreases
    - decrease the step size if the loss increases

- Implemented Hadoop and R/Snowfall
  - Snowfall: package for parallel R programs
  - [https://cran.r-project.org/web/packages/snowfall/index.html](https://cran.r-project.org/web/packages/snowfall/index.html)
Pros & Cons of DSGD

• Pros:
  ◦ **Faster convergence** than MSGD and ISGD
    ▪ no averaging step
  ◦ **Memory efficiency**: each machine needs to maintain a single block of $W$ and a single block of $H$ in memory

![Diagram of $W$ and $H$ blocks for Machine 1, Machine 2, and Machine 3]

• Cons:
  ◦ **Many hyperparameters**: step size ($\epsilon$) in addition to regularization parameter ($\lambda$) and rank ($r$)
Roadmap

• Matrix Factorization (review)

• Algorithms
  ◦ Distributed SGD: DSGD
  ◦ *Alternating Least Square: ALS* <<
  ◦ Cyclic Coordinate Descent: CCD++

• Experiments

• Extension to Tensor Factorization

• Conclusions
Alternating Least Square

Large-scale Parallel Collaborative Filtering for the Netflix Prize

(AAIM 2008)

Yunhong Zhou, Dennis Wilkinson, Robert Schreiber and Rong Pan

Cited by 520
Main Idea behind ALS

• How hard is matrix factorization?
  ◦ i.e., finding

\[
\arg\min_{W,H} L(V,W,H)
\]

• Solving the entire problem is difficult
  ◦ \(L(V,W,H)\) is a \textbf{non-convex} function of \(W\) and \(H\)
    ▪ many local optima
  ◦ finding a global optimum is NP-hard
Main Idea behind ALS (cont.)

• How hard is solving a smaller problem?
  ◦ Specifically, finding
    \[
    \arg\min_{H} L(V, W, H)
    \]
    while fixing \(W\) to its current value

• Much easier!
  ◦ \(L(V, W, H)\) is a \textit{convex} function of \(H\) (once \(W\) is fixed)
    ▪ one local optimum, which is also globally optimal
  ◦ Moreover, there exists the \textit{closed-form solution}
    ▪ we can directly compute the global optimum
Updating H

- Finding
  \[
  \argmin_H L(V, W, H)
  \]
  while fixing \( W \) to its current value

- This problem is solved by the following update rule

  \[
  \text{For } j = 1 \ldots m \\
  H_{.j} \leftarrow \left( \sum_{i \in Z_{.j}} W_{i.} (W_{i.})^T + \lambda I_k \right)^{-1} \left( \sum_{i \in Z_{.j}} V_{ij} W_{i.} \right)
  \]

  \( k \) by \( k \) identity matrix

  row indices of non-missing entries in the \( j \)-th column of \( V \)

  - derived from \( \frac{\partial L}{\partial H_{kj}} = 0, \forall k, \forall j \) (first-order condition)
Updating $W$

- Likewise, finding  
  \[
  \arg\min_{W} L(V, W, H)
  \]
  while fixing $H$ to its current value.

- This problem is solved by the following update rule

  \[
  W_i \leftarrow \left( \sum_{i \in Z_i} (H\cdot j)^T H\cdot j + \lambda I_k \right)^{-1} \left( \sum_{j \in Z_i} V_{ij} H\cdot j \right)
  \]

  column indices of non-missing entries in the $i$-th row of $V$

  - derived from $\frac{\partial L}{\partial W_{ik}} = 0, \forall i, \forall k$ (first-order condition)
Alternating Least Square (ALS)

randomly Initialize $W$ and $H$

repeat until convergence

update $W$ while fixing $H$ to its current value

update $H$ while fixing $W$ to its current value

• Each step never increases $L(V, W, H)$
  ◦ $W$ is updated to the “best” $W$ that minimizes $L(V, W, H)$ for current $H$
  ◦ $H$ is updated to the “best” $H$ that minimizes $L(V, W, H)$ for current $W$

• $L(V, W, H)$ monotonically decreases until convergence
Parallel ALS: Idea

• Recall the update rule for each $j$-th column of $H$

$$H_{:j} \leftarrow \left( \sum_{i \in Z_{:j}} W_{i:} (W_{i:})^T + \lambda I_k \right)^{-1} \left( \sum_{i \in Z_{:j}} V_{ij} W_{i:} \right)$$
Parallel ALS: Idea (cont.)

- Potentially every entry of $W$ needs to be read.
- Only the $j$-th column of $V$ needs to be read.
- $H$ does not need to be read.

⇒ can update the columns of $H$ independently in parallel.
Updating H in Parallel

• A toy example with 3 machines

• 1. Divide $V$ column-wise into 3 pieces

\[ W \begin{align*}
V^{(1)} & \quad V^{(2)} & \quad V^{(3)}
\end{align*} \]
Updating H in Parallel (cont.)

• 2. Distribute the pieces across the machines
• 3. $W$ is broadcast to every machine
Updating H in Parallel (cont.)

- 4. Compute the corresponding columns of $H$ in parallel
5. Broadcast the part of $H$ that each machine computes.
Parallel ALS: Idea

- Recall the update rule for each row of $W$

$$W_{i:} \leftarrow \left( \sum_{i \in Z_{i:}} (H_{:j})^T H_{:j} + \lambda I_k \right)^{-1} \left( \sum_{j \in Z_{i:}} V_{ij} H_{:j} \right)$$
Parallel ALS: Idea (cont.)

- Potentially every entry of $H$ needs to be read
- Only the $i$-th row of $V$ needs to be read
- $W$ does not need to be read
  $\Rightarrow$ can update the rows of $W$ independently in parallel
Updating W in Parallel

- A toy example with 3 machines
- 1. Divide $V$ row-wise into 3 pieces

\[
H = V^{(1)} + V^{(2)} + V^{(3)}
\]
Updating W in Parallel (cont.)

- 2. Distribute the pieces across the machines
- 3. $H$ is broadcast to every machine
4. Compute the corresponding rows of $W$ in parallel

- Machine 1: $W^{(1)}$, $V^{(1)}$
- Machine 2: $W^{(2)}$, $V^{(2)}$
- Machine 3: $V^{(3)}$, $V^{(3)}$
5. Broadcast the part of $W$ that each machine computes.

Machine 1

$W^{(1)}$

$W^{(2)}$

$W^{(3)}$

$V^{(1)}$

$H$

Machine 2

$W^{(1)}$

$W^{(2)}$

$W^{(3)}$

$V^{(2)}$

$H$

Machine 3

$W^{(1)}$

$W^{(2)}$

$W^{(3)}$

$V^{(3)}$

$H$
Pros & Cons of ALS

• Pros:
  ◦ **Less hyper-parameters**: not requiring step size

• Cons:
  ◦ **High computational cost**: e.g., matrix inversion takes $O(r^3)$

\[
H_{:j} \leftarrow \left( \sum_{i \in Z:j} W_{i:} (W_{i:})^T + \lambda I_k \right)^{-1} \left( \sum_{i \in Z:j} V_{ij} W_{i:} \right)
\]
Pros & Cons of ALS (cont.)

• Cons (cont.)
  ◦ High memory requirement:
    ▪ while updating $W$ (or $H$),
    ▪ each machine maintains the entire $H$ (or $W$) in memory
Roadmap

• Matrix Factorization (review)

• Algorithms
  ◦ Distributed SGD: DSGD
  ◦ Alternating Least Square: ALS
  ◦ **Cyclic Coordinate Descent: CCD++** <<

• Experiments

• Extension to Tensor Factorization

• Conclusions
Cyclic Coordinate Descent

Scalable Coordinate Descent Approaches
to Parallel Matrix Factorization
for Recommender Systems

(ICDM 2012)

Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit Dhillon

Best Paper Award

Cited by 175
Matrix Factorization: Revisit

- Goal: $V$ is approximated by the product of $W$ and $H$

\[ \begin{align*}
    V & \approx W \times H \\
    n \times m & \approx r \times m \\
    V & = \begin{bmatrix}
        \vdots & \vdots & \vdots \\
        \vdots & \vdots & \vdots \\
        \vdots & \vdots & \vdots \\
        \vdots & \vdots & \vdots \\
    \end{bmatrix} \\
    W & = \begin{bmatrix}
        \vdots & \vdots & \vdots \\
        \vdots & \vdots & \vdots \\
        \vdots & \vdots & \vdots \\
        \vdots & \vdots & \vdots \\
    \end{bmatrix} \\
    H & = \begin{bmatrix}
        \vdots & \vdots & \vdots \\
        \vdots & \vdots & \vdots \\
        \vdots & \vdots & \vdots \\
        \vdots & \vdots & \vdots \\
    \end{bmatrix}
\end{align*} \]
Matrix Factorization: Revisit

• Goal: $V$ is approximated by the product of $W$ and $H$

$$V_{ij} \approx [WH]_{ij} = \sum_{k=1}^{r} W_{ik} H_{kj} = W_{i1} H_{1j} + \cdots + W_{ir} H_{rj}$$
Matrix Factorization: Revisit

- Equivalently, $V$ is approximated by the sum of $r$ matrices

\[
V_{ij} \approx [W:1 H:1]_{ij} + \cdots + [W:r H:r]_{ij} = W_{i1} H_{1j} + \cdots + W_{ir} H_{rj}
\]
Cyclic Coordinate Descent: CCD++

• **repeat** until convergence
  ◦ **for** \( k = 1 \ldots r \)
    
    update the \( k \)-th column of \( W \) and the \( k \)-th row of \( H \)
    while fixing the other entries of \( W \) and \( H \)

• Consider a toy example with rank 3

**Step 1:**

\[
V ≈ W_1 \times H_1 + W_2 \times H_2 + W_3 \times H_3
\]
Order of Updates in CCD++

Step 2:

\[
V \approx W_1 \times H_1 + W_2 \times H_2 + W_3 \times H_3
\]

Step 3:

\[
V \approx W_1 \times H_1 + W_2 \times H_2 + W_3 \times H_3
\]
Performing one update

• Let $W_1$ and $H_1$ be the updated row and column

\[ V \approx W_1 \times H_1 + W_2 \times H_2 + W_3 \times H_3 \]

residual matrix $\hat{R}$
Performing one update (cont.)

- Updating $W_1$ and $H_1$: equals to factorizing $\hat{R}$ with rank 1

- $\hat{R} = V - W_2 H_2 - W_3 H_3$

  $= V - W_1 H_1 - W_2 H_2 - W_3 H_3 + W_1 H_1$

  $= V - WH + W_1 H_1$
Performing one update (cont.)

- Factorizing $\hat{R}$ with rank 1 can be performed using ALS
- with the following rules:
  - The updates rules can be derived from those in ALS

\[
\text{For } j = 1 \ldots m \quad H_{kj} \leftarrow \frac{\sum_{i \in Z_j} \hat{R}_{ij} W_{ik}}{\left(\sum_{i \in Z_j} (W_{ik})^2 + \lambda\right)}
\]

\[
\text{For } i = 1 \ldots n \quad W_{ik} \leftarrow \frac{\sum_{j \in Z_i} \hat{R}_{ij} H_{kj}}{\left(\sum_{j \in Z_i} (H_{kj})^2 + \lambda\right)}
\]
Pseudocode of CCD++

**Pseudocode (Simple)**

randomly initialize $W$ and $H$

repeat until convergence
  for $k = 1 \ldots r$:
    $\hat{R} \leftarrow V - WH + W_k H_k$
    update $W_k$ and $H_k$
  (rank-1 factorization of $\hat{R}$)

**Pseudocode (Detailed)**

randomly initialize $W$ and $H$

$R \leftarrow V - WH$  

repeat until convergence
  for $k = 1 \ldots r$:
    $\hat{R} \leftarrow R + W_k H_k$
    update $W_k$ and $H_k$
  (rank-1 factorization of $\hat{R}$)

$R \leftarrow \hat{R} + W_k H_k$
Parallel CCD++

- Updating $H_k$: in parallel
  - 1. divide $R$ column-wise
  - 2. distribute the pieces across the machines
  - 3. $W:k$ is broadcast to every machine
Parallel CCD++ (cont.)

• Updating $H_k$: in parallel (cont.)
  ◦ 4. compute the corresponding entries of $H_k$: in parallel
Parallel CCD++ (cont.)

- Updating $H_k$: in parallel (cont.)
  - 5. send the computed entries to the other machines.
Pros & Cons of CCD++

• Pros:
  ◦ Low computational cost
    ▪ update rules do not need matrix inversion

\[
H_{kj} \leftarrow \frac{\sum_{i \in \mathbb{Z}_j} \hat{R}_{ij} W_{ik}}{\left( \sum_{i \in \mathbb{Z}_j} (W_{ik})^2 + \lambda \right)} \quad W_{ik} \leftarrow \frac{\sum_{j \in \mathbb{Z}_i} \hat{R}_{ij} H_{kj}}{\left( \sum_{j \in \mathbb{Z}_i} (H_{kj})^2 + \lambda \right)}
\]
Pros & Cons of CCD++

- Pros (cont.):
  - **Low memory requirements**
    - each machine needs to maintain
    - one row of $H$ (or one column of $W$) in memory at a time
    - instead of entire $H$ (or $W$)
Pros & Cons of CCD++

• Cons:
  ◦ **Slow for dense matrices** with many non-missing entries
    ▪ To update \( \hat{R} \), every non-missing entry
    ▪ needs to be read and rewritten.
    ▪ This is especially slow if \( \hat{R} \) is stored on disk.
Roadmap

• Problem (review)

• Algorithms
  ◦ Distributed SGD: DSGD
  ◦ Alternating Least Square: ALS
  ◦ Cyclic Coordinate Descent: CCD++

• Experiments <<

• Extension to Tensor Factorization

• Conclusions
Experimental Settings

- Datasets:

  - Large-scale Matrix Factorization (by Kijung Shin)
  - Netflix
  - Yahoo! Music
  - movielens

  Non-commercial, personalized movie recommendations.

Movies (or songs)

<table>
<thead>
<tr>
<th>Users</th>
<th>COCO</th>
<th>BRIDGET JONES'S DIARY</th>
<th>Ratings</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>?</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>?</td>
<td>2</td>
</tr>
</tbody>
</table>
Experimental Settings (cont.)

• Datasets:
  - Netflix: Large-scale Matrix Factorization (by Kijung Shin)
    (2.7M users, 18K movies, 100M ratings)
  - Yahoo Music: (1M users, 625K songs, 256M ratings)
  - MovieLens: (71K users, 65K movies, 10M ratings)
Experimental Settings (cont.)

• Split of Datasets:
  ◦ Training set: about 80% of non-missing entries
  ◦ Test set: about 20% of non-missing entries

• Evaluation Metric:
  ◦ Test RMSE (Root-mean square error)

\[
RMSE(V, W, H) = \sqrt{\frac{1}{|Z_{test}|} \sum_{(i,j) \in Z_{test}} (V_{ij} - [WH]_{ij})^2}
\]

index of test entries

True rating

Estimated rating
Experimental Settings (cont.)

• Machines and implementations
  ◦ Shared-memory setting:
    ▪ 8 cores
    ▪ OpenMP in C++
  ◦ Distributed setting:
    ▪ up to 20 machines
    ▪ MPI in C++
Convergence Speed

• Shared-memory setting with 8 cores
• CCD++ decreases Test RMSE fastest

Figure 3(b) of [YHS+12]
Convergence Speed (cont.)

- Shared-memory setting with 8 cores
- CCD++ converges Test RMSE fastest

Figure 3(c) of [YHS+12]
Convergence Speed (cont.)

- Shared-memory setting with 8 cores
- CCD++ converges fastest

Figure 3(a) of [YHS+12]
Speedup in Shared Memory

- **CCD++** and **ALS** show near-linear machine scalability
- **DSGD** suffers from high cache-miss rates due to its randomness
  - Cache-miss rate increases as more cores are used

Figure 4 of [YHS+12]
Speedup in Distributed Settings

- All algorithms show similar speedups

Figure 6(b) of [YHS+12]
Roadmap

• Matrix Factorization (review)

• Algorithms
  ◦ Distributed SGD: DSGD
  ◦ Alternating Least Square: ALS
  ◦ Cyclic Coordinate Descent: CCD++

• Experiments

• Extension to Tensor Factorization <<

• Conclusions
Tensors

- An $N$-way tensor is an $N$-dimensional array

\[
\begin{bmatrix}
t_1 \\
\vdots \\
t_{I_1}
\end{bmatrix}
\quad \begin{bmatrix}
t_{11} & \cdots & t_{1I_2} \\
\vdots & \ddots & \vdots \\
t_{I_11} & \cdots & t_{I_1I_2}
\end{bmatrix}
\]

1-way tensor (= vector)

2-way tensor (= matrix)

3-way tensor
Tensor Factorization: Problem

• Given:
  ◦ $X$: $I$ by $J$ by $K$ tensor
    ▪ possibly with missing values
  ◦ $R$: target rank (a scalar)
    ▪ usually $R \ll I, J, K$
Tensor Factorization: Problem

• Given:
  ◦ $X$: $I$ by $J$ by $K$ tensor (i.e., 3-dim array)
    ▪ possibly with missing values
  ◦ $R$: target rank (a scalar)
    ▪ usually $R \ll I, J, K$

• Find:
  ◦ $U$: $I$ by $R$ matrix
  ◦ $H$: $J$ by $R$ matrix
  ◦ $W$: $K$ by $R$ matrix
    ▪ without missing values
Tensor Factorization: Problem

- **Goal:** $X \approx [UHW]$
  - where $[UHW]_{ijk} = \sum_{r=1}^{R} U_{ir} H_{jr} W_{kr}$
Loss Function

\[ L(X, U, H, W) = \sum_{(i,j,k) \in Z} (X_{ijk} - \text{[UHW]_{ijk}})^2 + ... \]

Indices of non-missing entries
i.e., \((i, j, k) \in Z\)
\(\leftrightarrow X_{ijk}\) is not missing

Goal: to make \(X\) and \(UHW\) similar
Loss Function (cont.)

\[
L(X, U, H, W) = \sum_{(i,j,k)\in Z} (X_{ijk} - [UHW]_{ijk})^2 \\
+ \lambda (\|U\|_F^2 + \|H\|_F^2 + \|W\|_F^2)
\]

Goal: to prevent overfitting
(by making the entries of \(U, H\) and \(V\) close to zero)

Regularization parameter

Frobenius Norm:
\[
\|U\|_F^2 = \sum_{i=1}^{I} \sum_{r=1}^{R} (U_{ir})^2
\]
SGD for TF

- \( L(X, U, H, W) \): sum of loss for each non-missing entry
  \[
  L(X, U, H, W) = \sum_{(i,j,k) \in Z} L'(X_{ijk}, U_i: H_j: W_k:)
  \]

- An SGD step on each non-missing entry \( X_{ijk} \):
  - Read \( U_i: \), \( H_j: \), and \( W_k: \)
  - Compute gradient of \( L'(V_{ij}, W_i:, H_j:, W_k:) \)
  - Update \( U_i: \), \( H_j: \), \( W_k: \)
Parallel Algorithms for TF

- Parallel algorithms for MF are extended to TF
  - DSGD $\rightarrow$ FlexiFaCT [BKP+14]
  - ALS [SPK16]
  - CCD++ $\rightarrow$ CDTF [SK17]
Applications

- Context-aware recommendation [KABO10]

\[
\hat{X}_{ijk} = [UHW]_{ijk}
\]

Users

Seasons

Winter

Fall

Summer

Spring

Movies

Ratings

Missing rating
Applications (cont.)

- Context-aware recommendation [KABO10]

Levels of hunger

Users

Restaurants

\[
\hat{X}_{ijk} = \left[ UHW \right]_{ijk}
\]
Applications (cont.)

- Video Restoration [LMWY13]

$$\hat{X}_{ijk} = [UHW]_{ijk}$$

Pixels

Missing or corrupted pixel

$\hat{X}_{ijk}$

X-axis

Y-Axis

Large-scale Matrix Factorization (by Kijung Shin)
Applications (cont.)

- Video Restoration [LMWY13]

Corrupted frame

Restored frame
Applications (cont.)

- Personalized Web Search [SZL+05]

![Diagram showing matrix factorization for personalized web search.](image)

- Users
- Query keywords
- Web pages

$\hat{X}_{ijk} = [UHW]_{ijk}$

Preference

Missing preference
Roadmap

• Matrix Factorization (review)

• Algorithms
  ◦ Distributed SGD: DSGD
  ◦ Alternating Least Square: ALS
  ◦ Cyclic Coordinate Descent: CCD++

• Experiments

• Extension to Tensor Factorization

• Conclusions <<
Conclusions

• Matrix Factorization (MF)

• Parallel algorithms for MF
  ◦ Distributed SGD: DSGD
  ◦ Alternating Least Square: ALS
  ◦ Cyclic Coordinate Descent: CCD++

• Tensor Factorization (TF)
  ◦ Extension of MF to tensors
  ◦ Applications: context-aware recommendation, video restoration, personalized web search, etc.
Questions?

• These slides are available at:
  
  www.cs.cmu.edu/~kijungs/ETC/10-405.pdf

• Email: kijungs@cs.cmu.edu
References


References (cont.)


Closed-Form Solution of ALS

Let $W = \begin{bmatrix}
- & W_1: & - \\
- & : & - \\
- & W_n: & - \\
\end{bmatrix}$, $H = \begin{bmatrix}
H_{1:} & \cdots & H_{m:} \\
\vdots & \ddots & \vdots \\
\vdots & \cdots & \vdots \\
\end{bmatrix}$

Then,

$L(V, W, H)$

$$= \sum_{(i,j)\in Z} (V_{ij} - W_{i:H:j})^2 + \lambda \left( \sum_{i=1}^n \|W_{i:}\|_2 + \sum_{j=1}^m \|H_{j:}\|_2 \right)$$

$$= \sum_{(i,j)\in Z} (V_{ij} - \sum_{k=1}^r W_{ik}H_{kj})^2$$

$$+ \lambda \left( \sum_{i=1}^n \sum_{k=1}^r W_{ik}^2 + \sum_{j=1}^m \sum_{k=1}^r H_{kj}^2 \right)$$
Closed-Form Solution of ALS (cont.)

\[ \frac{1}{2} \frac{\partial L}{\partial H_{kj}} = 0, \quad \forall k, \forall j \]  
(first-order conditions)

\[ \Rightarrow \sum_{i \in Z,j} (V_{ij} - W_{i,H,j})W_{ik} + \lambda H_{kj} = 0, \quad \forall k, \forall j \]

rows of non-missing entries in the \( j \)-th column of \( V \)

\[ \Rightarrow \sum_{i \in Z,j} W_{i,H,j}W_{ik} + \lambda H_{kj} = \sum_{i \in Z,j} V_{ij}W_{ik}, \quad \forall k, \forall j \]

\[ \Rightarrow \sum_{i \in Z,j} W_{i,H,j}W_{ik} + \lambda H_{kj} = \sum_{i \in Z,j} V_{ij}W_{ik}, \quad \forall k, \forall j \]
Closed-Form Solution of ALS (cont.)

(If we stack $r$ conditions on $H_{1j}, \ldots, H_{rj}$)

$\Rightarrow (\sum_{i \in Z \cdot j} W_i (W_i)^T + \lambda I_k) H_{:j} = \sum_{i \in Z \cdot j} V_{ij} W_{i:}, \forall j$

$k$ by $k$ identity matrix

$\Rightarrow H_{:j} = \left(\sum_{i \in Z \cdot j} W_i (W_i)^T + \lambda I_k\right)^{-1} \left(\sum_{i \in Z \cdot j} V_{ij} W_{i:}\right), \forall j$