DiFacto — Distributed Factorization Machines

Mu Li
Joint work with Ziqi Liu, Alex Smola, and Yu-Xiang Wang

Carnegie Mellon University
Linear model is widely used for large-scale datasets

Training data size (TB) of linear model for Ads CTR estimation in an Internet company
Linear model is widely used for large-scale datasets

Training data size (TB) of linear model for Ads CTR estimation in an Internet company

Reach model capacity limit when the data go very large
Factorization Machine

- Linear model predicts by $f(x) = \sum_{i} x_i w_i$
Factorization Machine

- Linear model predicts by \( f(x) = \sum_i x_i w_i \)

- Factorization machine (Rendle et al, ’10) adds a \( k \)-dimensional embedding
Factorization Machine

- Linear model predicts by \( f(x) = \sum_i x_i w_i \)

- Factorization machine (Rendle et al, ’10) adds a \( k \)-dimensional embedding

\[
f(x) = \sum_i x_i w_i + \sum_{i<j} x_i x_j \langle V_i, V_j \rangle
\]
Factorization Machine

Linear model predicts by

\[ f(x) = \sum_i x_i w_i + \sum_{i < j} x_i x_j h_{\langle V_i, V_j \rangle} \]

(can go beyond second-order)

Factorization machine (Rendle et al., '10) adds a k-dimensional embedding

\[ f(x) = \sum_i x_i w_i + \sum_{i < j} x_i x_j V_i, V_j \]

Linear model predicts by

\[ f(x) = \sum_i x_i w_i \]
The Challenge

Both computation and storage costs of k-dimension FM are k times larger than linear model

- On Criteo CTR dataset with 1.5B examples and 0.36B features

<table>
<thead>
<tr>
<th>$k$</th>
<th>model size</th>
<th>FLOP per data pass</th>
<th>time for a single CPU (in theory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>288GB</td>
<td>10 P</td>
<td>5 hours</td>
</tr>
<tr>
<td>1,000</td>
<td>3TB</td>
<td>100 P</td>
<td>50 hours</td>
</tr>
</tbody>
</table>
The Challenge

Both computation and storage costs of k-dimension FM are k times larger than linear model

✧ On Criteo CTR dataset with 1.5B examples and 0.36B features

<table>
<thead>
<tr>
<th>$k$</th>
<th>model size</th>
<th>FLOP per data pass</th>
<th>time for a single CPU (in theory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>288GB</td>
<td>10 P</td>
<td>5 hours</td>
</tr>
<tr>
<td>1,000</td>
<td>3TB</td>
<td>100 P</td>
<td>50 hours</td>
</tr>
</tbody>
</table>

Key Contributions

1. Reduce model capacity by exploring data sparsity
2. Highly efficient distributed training
The Challenge
Both computation and storage costs of k-dimension FM are k times larger than linear model

- On Criteo CTR dataset with 1.5B examples and 0.36B features

<table>
<thead>
<tr>
<th>$k$</th>
<th>model size</th>
<th>FLOP per data pass</th>
<th>time for a single CPU (in theory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>288GB</td>
<td>10 P</td>
<td>5 hours</td>
</tr>
<tr>
<td>1,000</td>
<td>3TB</td>
<td>100 P</td>
<td>50 hours</td>
</tr>
</tbody>
</table>

Key Contributions
1. Reduce model capacity by exploring data sparsity
2. Highly efficient distributed training

Key Take Away
Make large-scale FM as cheap as linear model
Statistic Model
Distributed Optimization
Evaluation
Key Observation

- High-dimensional datasets are often extremely sparse
- The count of feature occurrence often obeys a power law distribution. e.g. the Criteo dataset

<table>
<thead>
<tr>
<th>occurrence &gt;</th>
<th>1</th>
<th>10</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td># features (%)</td>
<td>53%</td>
<td>5.8%</td>
<td>0.49%</td>
</tr>
</tbody>
</table>
Key Observation

✧ High-dimensional datasets are often extremely sparse
✧ The count of feature occurrence often obeys a power law distribution. e.g., the Criteo dataset

<table>
<thead>
<tr>
<th>occurrence &gt;</th>
<th>1</th>
<th>10x</th>
<th>10</th>
<th>10x</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td># features (%)</td>
<td>53%</td>
<td>5.8%</td>
<td>0.49%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Key Observation

✦ High-dimensional datasets are often extremely sparse
✦ The count of feature occurrence often obeys a power law distribution. E.g. the Criteo dataset

<table>
<thead>
<tr>
<th>occurrence &gt;</th>
<th>1</th>
<th>10x</th>
<th>10</th>
<th>10x</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td># features (%)</td>
<td>53%</td>
<td>5.8%</td>
<td>0.49%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

✦ If feature \( i \) appears less than \( k+1 \) times in the data, then the problem of estimating \( (w_i, V_i) \) is underdetermined
Key Observation

- High-dimensional datasets are often extremely sparse
- The count of feature occurrence often obeys a power law distribution. E.g., the Criteo dataset

<table>
<thead>
<tr>
<th>occurrence</th>
<th># features (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 1</td>
<td>53%</td>
</tr>
<tr>
<td>10x</td>
<td>5.8%</td>
</tr>
<tr>
<td>100x</td>
<td>0.49%</td>
</tr>
</tbody>
</table>

- If feature $i$ appears less than $k+1$ times in the data, then the problem of estimating $(w_i, V_i)$ is underdetermined

Our solution

Data and model adaptive regularizations to reduce model capacity on unimportant features
Memory Adaptive Constraints

- Limit the effective embedding dimension $k_i$ for “tail” feature $i$

$$V_{ij} = 0 \text{ for all } j > k_i$$
Memory Adaptive Constraints

- Limit the effective embedding dimension $k_i$ for “tail” feature $i$

  $$V_{ij} = 0 \text{ for all } j > k_i$$

- Choose $k_i$ according to the number of occurrence of feature $i$, $n_i$
  - simple choice:

    $$k_i = \begin{cases} k & \text{if } n_i \geq k \\ 0 & \text{otherwise} \end{cases}$$

  - three levels:

    $$k_i = \begin{cases} 10k & \text{if } n_i \geq 10k \\ k & \text{if } 10k > n_i \geq k \\ \min(n_i, k) & \text{otherwise} \end{cases}$$
Sparse Regularization

- Model adaptive capacity control by sparse regularization
- Encourage a sparse linear term

\[ \lambda_1 \|w\|_1 \]
Sparse Regularization

- Model adaptive capacity control by sparse regularization
- Encourage a sparse linear term

\[ \lambda_1 \| w \|_1 \]

- Need sparse V too:
  - Structured sparsity on V
  - A simpler solution

\[ \sum_i \left[ w_i^2 + \| V_i \|_2^{\frac{1}{2}} \right] + \| V_i \|_2 \]

\[ V_i = 0 \text{ if } w_i = 0 \]
Statistic Model
Distributed Optimization
Evaluation
Distributed by Parameter Server (Li et al, ’14)

Training data
Distributed by Parameter Server (Li et al, ’14)
Distributed by Parameter Server (Li et al, ’14)
Distributed by Parameter Server (Li et al, ’14)

Model

Server machines

Training data

Worker machines
Distributed by Parameter Server (Li et al, ’14)

Model

Server machines

push gradients

Training data

Worker machines
Distributed by Parameter Server (Li et al, ’14)

Model

push gradients

Server machines

pull weight

Training data

Worker machines
Distributed Asynchronous SGD

Server machines

Worker machines
Distributed Asynchronous SGD

Workers run independently

- For each iteration
  - read a new minibatch
  - pull weights from the servers
  - compute gradients
  - push gradients into the servers
Distributed Asynchronous SGD

Workers run independently

- For each iteration
  - read a new minibatch
  - pull weights from the servers
  - compute gradients
  - push gradients into the servers

Servers update weights:

- Update $V$ by adagrad
- Update $w$ by FTRL
Distributed Asynchronous SGD

Workers run independently
- For each iteration
  - read a new minibatch
  - pull weights from the servers
  - compute gradients
  - push gradients into the servers

Servers update weights:
- Update $V$ by adagrad
- Update $w$ by FTRL

Scheduler node:
- manages load balanced
- achieves fault tolerance
Data consistency

- Async SGD trade-off data consistency for system performance
Data consistency

- Async SGD trade-off data consistency for system performance

$t = 0$

pull weight
Data consistency

✦ Async SGD trade-off data consistency for system performance

$t = 0$

pull weight

$t = 0$

pull weight
Data consistency

- Async SGD trade-off data consistency for system performance

$t = 0$
- Pull weight

$t = 0$
- Pull weight

$t = 1$
- Push gradient (delay = 0)
Data consistency

- Async SGD trade-off data consistency for system performance

$t = 0\hspace{1cm} t = 1$

- Pull weight
- Push gradient (delay = 0)
Data consistency

✧ Async SGD trade-off data consistency for system performance

$t = 0$
- pull weight

$t = 1$
- push gradient (delay = 0)

$t = 0$
- pull weight

$t = 2$
- push gradient (delay = 1)
Data consistency

- Async SGD trade-off data consistency for system performance

$t = 0$
- Pull weight

$t = 1$
- Pull weight

$t = 0$
- Pull weight

$t = 2$
- Push gradient (delay = 1)

$t = 1$
- Push gradient (delay = 0)

$t = 3$
- Push gradient (delay = 1)
Theoretical Analysis
Theoretical Analysis

- Key assumptions
  - maximal delay is upper-bonded by $\tau$
  - stochastic gradient is $L$-Lipschitz, and variance is bounded by $\sigma^2$
Theoretical Analysis

✧ Key assumptions

✧ maximal delay is upper-bonded by $\tau$

✧ stochastic gradient is $L$-Lipschitz, and variance is bounded by $\sigma^2$

✧ If choose a constant step-size $\eta = \sqrt{\frac{C}{L \tau \sigma^2}}$
then for every $T \geq 4LC(\tau + 1)^2 / \sigma$

$$\frac{1}{T} \sum_{t=1}^{T} \mathbb{E}\|\nabla f(x_t)\|^2 \leq 4 \sqrt{\frac{CL}{T}} \sigma$$

✧ LHS is an intuitive measure of distance from stationary point

✧ Delay may slow convergence

✧ We can use a large learning rate for large minibatch size or sparse dataset
Statistic Model
Distributed Optimization
Evaluation
Adaptive Memory

- Criteo dataset: 1.5B examples, 360M features
- Run on 10 AWS EC2 machines

![Graph showing model size vs dimension k]

- x-axis: dimension k (log scale)
- y-axis: model size (log scale)
- Log-log plot with a power-law trend
Adaptive Memory

- Criteo dataset: 1.5B examples, 360M features
- Run on 10 AWS EC2 machines

Graph showing model size vs. dimension k for baseline and frequency constraint, with a ~100x improvement.
Adaptive Memory

- Criteo dataset: 1.5B examples, 360M features
- Run on 10 AWS EC2 machines

![Graph showing model size vs. dimension k for baseline, frequency constraint, and frequency constraint + sparse regularization.]

For the baseline, the model size increases significantly with the dimension k. Adding a frequency constraint reduces this increase, and further reducing it with sparse regularization.
Adaptive Memory

- Criteo dataset: 1.5B examples, 360M features
- Run on 10 AWS EC2 machines

![Graphs showing model size and relative logloss.](image)

- Baseline
- Frequency constraint
- Frequency constraint + sparse regularization

- Model size
- Dimension $k$
- Relative logloss (%)

- ~100x improvement in model size
- ~50x improvement in relative logloss

- Comparing to linear model, -3% test LogLoss
Adaptive Memory

- Criteo dataset: 1.5B examples, 360M features
- Run on 10 AWS EC2 machines

![Graphs showing model size versus dimension k and relative logloss versus dimension k.](image)

- Baseline
- Frequency constraint
- + Sparse regularization

- Model size: ~100x
- ~50x

- Relative logloss (%)

- Only 2x more computation cost for FM with k=100 comparing to linear model

- -3% test LogLoss comparing to linear model
Compare to LibFM

- LibFM is a widely used library for FM (Rendle et al)

---

LibFM

Sampled Criteo

![Sampled Criteo Chart](chart.png)
Compare to LibFM

- LibFM is a widely used library for FM (Rendle et al)

![Graph showing test logloss vs time for LibFM and DiFacto, 1 thread on Sampled Criteo dataset. The graph illustrates the performance comparison between the two libraries with test logloss values in the range of $10^{-0.339}$ to $10^{-0.348}$ and time (sec) ranging from $10^1$ to $10^4$. The LibFM line is represented by blue circles, and the DiFacto, 1 thread line is represented by green squares. The graph shows that DiFacto outperforms LibFM in terms of test logloss for the given time range.]
Compare to LibFM

✧ LibFM is a widely used library for FM (Rendle et al)

![Graph showing comparison between LibFM, DiFacto, 1 thread, and DiFacto, 10 threads in a sampled Criteo dataset. The y-axis represents test logloss, and the x-axis represents time in seconds. The graph indicates a clear improvement in performance, with DiFacto, 10 threads showing a significant reduction in test logloss compared to LibFM and DiFacto, 1 thread. The legend indicates that the blue line represents LibFM, the green line represents DiFacto, 1 thread, and the red line represents DiFacto, 10 threads. The graph also includes a '10x' label to highlight the improvement in performance.]
Fixed-point Compression

- Quantize float into n-bytes integer with randomized rounding during communication
Fixed-point Compression

- Quantize float into n-bytes integer with randomized rounding during communication
Fixed-point Compression

- Quantize float into n-bytes integer with randomized rounding during communication

![Graph showing compression ratio]

>4x

- Gigabyte
- #byte per entry
Fixed-point Compression

- Quantize float into n-bytes integer with randomized rounding during communication

![Bar chart showing the number of bytes per entry and relative logloss](chart.png)
Fixed-point Compression

- Quantize float into n-bytes integer with randomized rounding during communication

![Graph showing relative logloss (%) and #byte per entry](image)

- Gigabyte compression improves accuracy!
Scalability

- Scaling from 1 machine to 16 machines
Conclusion

✦ **Goal:** scale factorization machine into large-scale datasets
✦ **Solution 1:** Data and model adaptive *regularizations* to reduce model capacity on unimportant features
✦ **Solution 2:** Efficient *distributed training* by asynchronous SGD using the parameter server framework
✦ **Results:** FM with a 100-dimension embedding provides significant accuracy improvement over linear model, with only ~2x more computation and storage cost
✦ **Codes** are publicly available at the DMLC project

![DMLC](https://github.com/dmlc/difacto)