DiFacto — Distributed Factorization Machines

Mu Li
Computer Science, Carnegie Mellon University
Marianas Labs Inc., Pittsburgh, PA
muli@cs.cmu.edu

Ziqi Liu
MOEKLINNS Lab, Computer Science
Xi'an Jiaotong University, Xi'an, China
ziqilau@gmail.com

Alexander J. Smola
Machine Learning, Carnegie Mellon University
Marianas Labs Inc., Pittsburgh, PA
alex@smola.org

Yu-Xiang Wang
Machine Learning Department
Carnegie Mellon University, Pittsburgh, PA
yuxiangw@cs.cmu.edu

ABSTRACT

Factorization Machines offer good performance and useful embeddings of data. However, they are costly to scale to large amounts of data and large numbers of features. In this paper we describe DiFacto, which uses a refined Factorization Machine model with *sparse memory adaptive constraints* and *frequency adaptive regularization*. We show how to distribute DiFacto over multiple machines using the Parameter Server framework by computing distributed subgradients on minibatches *asynchronously*. We analyze its convergence and demonstrate its efficiency in computational advertising datasets with billions examples and features.

1. INTRODUCTION

Nonlinear models for recommendation and estimation have experienced significant interest in recent years. The growing body in deep learning [3], kernels [7], and decision trees [22] bears witness of this development. For recommender systems and polynomial generalized linear models Factorization Machines (FM) [15, 16] offer a computationally efficient and powerful alternative. They achieve excellent results by using a low-rank expansion of the higher degree polynomial terms. Moreover, they offer a principled framework for a number of feature space heuristics in recommender systems, such as bias, features, cold-start strategies, and temporal models. This makes them a very attractive target for statistical modeling of high-dimensional sparse data occurring in many settings such as computational advertising, personalization, user profiling, recommendation, and search.

Unfortunately, despite low-rank expansions the memory cost remains tremendous for real-world settings. This occurs since each feature, each user, and each object need to be embedded into a low-dimensional space. A quick calculation shows that even on modest datasets such as Criteo's CTR estimation contest [6] we have up to 10^9 features. Realistic problems can have up to 10^{11} terms [8]. Even a modest 100 dimensional representation would require in the order of 1TB of data (parameters, preconditioners, auxiliary key storage). Even worse, the latter may be too simplistic for frequently occurring attributes. This makes the problem unsolvable on a single machine, when tackling industrial scale problems. Moreover, even multiple machines are heavily taxed by the large memory footprint. In order to address the above problems, we need:

1. A compact model with low computation and communication cost that nonetheless offers high-quality embeddings of the features.
2. An efficient mechanism for distributed optimization. This includes problem partitioning, optimization, and high performance communications protocols.

One key observation we made is that the importance of features in real datasets is not uniform, which deserves adaptive model capacities. In this paper, we propose a refined FM model, called DiFacto, which adaptively chooses the effective embedding dimension and regularization for each feature according to the importance, which is based on the feature frequency count and sparse regularization. The resulted model is up to 100x more compact than conventional FM and even provides better generalization accuracy.

We propose a fast distributed optimization algorithm based on the asynchronous stochastic gradient descent, with explicit modeling of the possibly non-uniform sparsity patterns in the training data and guarantee of satisfied sparse models. We prove that the proposed algorithm converges for this highly challenging nonconvex objective function even under asynchronous updates.

We describe a *Di*stributed *Facto*rization code (DiFacto) based on the Parameter Server framework [8, 17]. Experiments show that DiFacto scales to sparse data with billions of examples and features. To the best of our knowledge, our results describe one of the largest statistical model ever computed (using up to $1.2 \cdot 10^{11}$ features) over [1] by at least one order of magnitude. Our algorithms require only modest resources on a per-machine basis.

Outline. In section 2 we begin with an overview of Factorization Machines and of the Parameter Server framework. This is followed in Section 3 by a description of the statistical model employed in this paper, since it differs in a number of key parts from a simple Factorization Machine. Section 4 has details on the distributed optimization and convergence aspects of the problem. Experimental results are provided in Section 5. We conclude with a summary and discussion.

2. BACKGROUND

2.1 Objectives

In this paper we address two related goals — recommendation and prediction. The distinction between those problems is partly due to different notation conventions, and partly due to slightly different objectives. In recommender systems [2] one is typically given *pairs* of entities, say users u and movies m for which a rating $y(u, m)$ needs to be estimated. The quality of an estimate $f(u, m)$ is evaluated, e.g. by the squared discrepancy between rating and estimate $\frac{1}{2}(y(u, m) - f(u, m))^2$, or by the quality of the relative ordering of ratings, or by per-session ordering of preferences. For an exhaustive summary of different such objectives see e.g. [13]. In short, we are interested in the performance of $f(u, m)$ relative to $y(u, m)$. Whenever we have additional features z_u, z_m regarding u and m respectively and context c, they form part of the estimate via $f(u, z_u, m, z_m, c)$. In the following we use the shorthand

$$x := (u, z_u, m, z_m, c) \text{ and } f(x) := f(u, z_u, m, z_m, c) \tag{1}$$

to indicate that we are interested in obtaining a *function* that consumes x as an argument to generate $f(x)$.

In prediction we are interested in the slightly more mundane goal of obtaining $f(x)$, given pairs of *features* x and *labels* y. Typical goals are to minimize the squared deviation $\frac{1}{2}(y - f(x))^2$ or the log-likelihood of a particular label $\log(1 + e^{-yf(x)})$. Note that y and $f(x)$ need not be of the same datatype. This was exploited substantially in structured estimation [19].

In the following we will not make major distinctions between both problems above. Instead, we will simply assume that x is either a set of covariates, such as the words occurring in a document for which the click through rate (CTR) of an advertisement is sought, or a set of recommendation and rating parameters. The quality will be evaluated via a loss function $l(x, y, f(x))$, yielding the risk functional

$$R[f, X, Y] := \frac{1}{|X|} \sum_{(x_i, y_i) \in (X, Y)} l(x_i, y_i, f(x_i)). \tag{2}$$

It is our goal to find some f such that, given a training set $X_{\text{train}}, Y_{\text{train}}$, the expected risk (or the risk on an unseen test set) is minimized.

2.2 Factorization Machine

The typical setting in our context is that of a very high d-dimensional (and sparse) $x \in \mathbb{R}^d$. For instance, in recommender systems x might only contain two nonzero terms — an indicator for the user, and one for the movie respectively, i.e. $x_u = 1$ and $x_m = 1$. The consequence is that linear models are of limited use. For instance, in a recommender system this would amount to

$$f(x) = \langle w, x \rangle = w_u + w_m. \tag{3}$$

This is the trivial model where recommendations are simply the sum of user and movie biases. Nonetheless, in CTR estimation problems [12], such assumptions are quite popular, due to the very high dimensionality of the model and the associated uncertainty in determining even just linear parameters. In particular, even a quadratic model is too expensive to estimate, since this would require $O(d^2)$ rather than $O(d)$ parameters. This is invariably too large since typically the number of observations $n \ll d^2$.

Rendle et al. [15, 16] introduced a principled strategy for alleviating this problem via a low-rank expansion instead of a general high-dimensional expansion. They propose

$$f(x) = \langle w, x \rangle + \sum_{i<j} x_i x_j \operatorname{tr}\left(V_i^{(2)} \otimes V_j^{(2)}\right) + \sum_{i<j<k} x_i x_j x_k \operatorname{tr}\left(V_i^{(3)} \otimes V_j^{(3)} \otimes V_k\right) + \ldots \tag{4}$$

In other words, FM use a low-dimensional embedding of the (typically very sparse set of) features in x to a much smaller k_i-dimensional space via the embedding matrices $V^{(i)} \in \mathbb{R}^{d \times k_i}$. For the purpose of the current paper we limit ourselves to expansions up to second order (this is not a technical limitation of our algorithm but a mere convenience). This simplifies the exposition and it is sufficient, given the sparsity of the problem as we will discuss subsequently. Hence we may rewrite (4) via

$$f(x) = \langle w, x \rangle + \frac{1}{2} \|Vx\|_2^2 - \sum_{i=1}^{d} x_i^2 \|V_i\|_2^2 \tag{5}$$

Here we used the shorthand $V := V^{(2)}$ and $k_2 = k$, the polarization equality to rewrite the ordered sum, and the fact that $\operatorname{tr}(a \otimes b) = \langle a, b \rangle$. Moreover, V_i is the i-th column in V. Note also that (5) can be computed efficiently in $O(dk)$ time, requiring $O(dk)$ storage. This also illustrates a shortcoming of Factorization Machines.

Lemma 1 *Assume that we estimate (5) with an embedding dimension rank k. Then for features i for which $x_i \neq 0$ for less than $k + 1$ times, the problem of estimating (w_i, V_i) is underdetermined.*

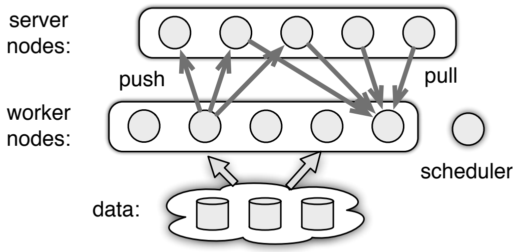
PROOF. This follows directly from linear algebra: whenever $x_i = 0$ the terms v_i, V_i do not occur in (5). To solve a linear system of $k + 1$ variables we need at least the same number of constraints. $\square$

Even worse, we need to store $O(k)$ memory even for variables that hardly ever occur. Given the power-law nature of many natural feature distributions, this is clearly undesirable. Lastly, large numbers of variables are not just hard to estimate and hard to store, they are also cumbersome in terms of optimization purposes, since they slow down convergence and need to be addressed, e.g. via additional stabilizers (regularization).

However, even in those cases the problem size remains formidable and we need tools for distributed inference: quite often the number of parameters significantly exceeds the amount of available memory on a computer. This calls for distributed *representations* and *optimization algorithms.*

2.3 Parameter Server

The underlying computational engine for the algorithm is the Parameter Server [8, 17]. In it, computation is performed by two separate groups of computers — workers and servers. The servers act as a generalized version of a (key,value) store which manages and updates the model parameters, while the workers process the training data that has been sharded (typically at random). The diagram below illustrates the machine layout.



Task schedulers and resource manager allow for a controlled data flow. Moreover, they manage fault tolerance, e.g. to cope with failure of individual nodes. The pertinent aspects of this setup are that each one of the workers processes a distinct (random) block of data. w and V are stored in a distributed fashion on the server. Retrieval and updates are achieved by the following two operations:

push(key, value) sends a vector of (key, value) pairs in combination to the recipient (e.g. a group of servers holding parts of the data). To be more specific — in a distributed gradient update procedure, the workers might send the locally computed gradients to the servers. Due to the data sparsity, only a part of the gradients is nonzero, denoted by $(g_{i_1}, \ldots, g_{i_n})$ and $n \ll d$. Often it is desirable to present the gradients as a list of (key, value) pairs, namely $\left\{(i_j, g_{i_j})\right\}_{j=1}^{n}$, where the feature index is the key and the according gradient item is the value.

pull(key) requests the values associated with a list of keys. This is particularly useful whenever the amount of memory available is insufficient to hold a full model. Instead, workers prefetch the model entries relevant for solving the model only when needed.

The parameter server provides flexible data consistency model via a dependency graph (a DAG). It allows for a many algorithms ranging from bulk-synchronous processing to fully asynchronous algorithms and sketching services. See e.g. [8] for details on a range of related strategies. It also supports various filters to reduce the data communication costs. For the purpose of the present paper we ignore detailed systems considerations and focus mostly on the algorithm, the statistical model, and its application to a number of datasets.

3. STATISTICAL MODEL

3.1 Memory Adaptive Constraints

The first step in making factorization machines tractable is to modify (4) in line with Lemma 1. That is, whenever we have insufficient evidence of a feature occurring, it makes no sense to allocate much capacity to it. More to the point, instead of allocating a *fixed* number of dimensions k to *each* feature i, regardless of how frequently the i-th feature of example x, denoted by x_i, not equal to 0, we make this number dependent on n_i, the number of times $x_i \neq 0$ on the training set, i.e. more formally

$$n_i := \left|\{x : x_i \neq 0\}\right|. \tag{6}$$

Given a set of dimensionalities $\{k_i\}$ we modify the mapping $x_i \to x_i V_i$ such as to require that $V_{ij} = 0$ for all $j > k_i$. This forces infrequently occurring features to assume a rather lower-dimensional embedding using only k_i dimensions rather than the full set $k > k_i$, e.g. via

$$k_i = \begin{cases} k & \text{if } n_i > r \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

where the threshold r could be simply $r = k$. This would effectively allow only for sufficiently frequent terms to be considered in the Factorization Machine embedding. We can also allocate small embeddings for these infrequent features:

$$k_i = \begin{cases} k & \text{if } n_i > r \\ \min(n_i, k) & \text{otherwise} \end{cases} \tag{8}$$

More refined choices use more than one level, e.g.

$$k_i = \begin{cases} k^{(1)} & \text{if } n_i > r^{(1)} \\ k^{(2)} & \text{if } r^{(1)} \geq n_i > r^{(2)} \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

For instance set $k^{(1)} = r^{(1)} = 1000$ and $k^{(2)} = r^{(2)} = 100$, yielding a larger embedding for the most frequent keys.

3.2 Sparse Regularization

While the above strategy of variable elimination is entirely independent of the actual problem we solve, we also need a data adaptive capacity control mechanism that depends on the predictive power of features relative to the labels y_i. A popular choice is ℓ_1 regularization [20], which has been widely used in linear models for high dimensional data such as computational advertising [12]. We pick the penalty

$$\Omega[w] := \lambda_1 \left\|w\right\|_1$$

Here λ_1 controls the degree of sparsity. The sparse model induced by the ℓ_1 regularization not only penalizes complex model, it also reduces the computation cost of the gradient and saves the communication traffic. It results in a smaller final model which further makes deploying this model on an online service easier.

We can also apply a similar structured sparsity [14] on V to obtain sparse solution, such as

$$\Omega[w, V] = \sum_i \left[w_i^2 + \|V_i\|_2^2\right]^{\frac{1}{2}} + \|V_i\|_2 \tag{10}$$

Recall that in this case the derivative of the penalty vanishes at $w_i = 0$, provided that $V_i \neq 0$. Unfortunately, this penalty is harder to handle from the system perspective. Hence we replace it by a rather straightforward approximation that *directly* implements what (10) aims to accomplish — we require that $V_i = 0$ whenever $w_i = 0$. As experiments show, this is by no means detrimental to the overall outcome of the estimation. This also resembles most closely the ANOVA decomposition hierarchy proposed e.g. by [21].

3.3 Frequency Adaptive Regularization

It is well known that penalizing terms occurring at different frequency adaptively can lead to improved generalization performance [18]. For instance, in collaborative filtering the strategy to penalize frequent terms *more aggressively*, e.g. by performing shrinkage only whenever a user (or a movie) is being updated, has proven successful.

In DiFacto we use a slightly more general and flexible method for capacity control — to shrink only whenever an feature occurs in a minibatch. The consequence of this is that parameters associated with frequent features are less overregularized. The penalty can be presented as

$$\Omega[w, V] = \frac{1}{2} \sum_i \left[\lambda_i w_i^2 + \mu_i \|V_i\|_2^2\right] \text{ with } \lambda_i \propto \mu_i. \quad (11)$$

As we shall see, this approach can be implemented extremely conveniently, simply by adjusting the minibatch size.

Lemma 2 (Dynamic Regularization) *Assume that we solve a factorization machines problem with the following updates in a minibatch update setting: for each feature i,*

$$I_i \longleftarrow I\left\{[x_j]_i \neq 0 \text{ for some } (x_j, y_j) \in B\right\} \quad (12)$$

$$w_i \longleftarrow w_i - \frac{\eta_t}{b} \sum_{(x_j,y_j)\in B} \partial_{w_i} l(f(x_j), y_j) - \eta_t \lambda w_i I_i \quad (13)$$

$$V_i \longleftarrow V_i - \frac{\eta_t}{b} \sum_{(x_j,y_j)\in B} \partial_{V_i} l(f(x_j), y_j) - \eta_t \mu V_i I_i \quad (14)$$

Here B denotes the minibatch and b the according size, and I_i denotes whether or not feature i appears in this minibatch. Then effective regularization is given by

$$\lambda_i = \lambda\rho_i \text{ and } \mu_i = \mu\rho_i \text{ where } \rho_i = 1 - (1 - n_i/m)^b \approx \frac{n_i b}{m}$$

Proof. The probability that a particular feature occurs in a random minibatch is $\rho_i = 1 - (1 - n_i/m)^b$. Observe that while the amount of regularization on the minibatches is not independent, it is additive (and exchangeable). Hence the expected amount of regularizations are $\rho_i\lambda$ and $\rho_i\mu$, respectively. Expanding the Taylor series in ρ_i yields the approximation. □

Note that for $b = 1$ we obtain the conventional frequency dependent regularization, whereas in the batch setting $b = m$ we obtain the Frobenius regularization. That is, choosing the minibatch size conveniently allows us to interpolate between both extremes efficiently.

3.4 Putting All Things Together

We obtain the following model and optimization problem:

$$\begin{aligned} \underset{w,V}{\text{minimize}} \; & \frac{1}{|X|} \sum_{(x,y)} l(f(x), y) + \lambda_1 \|w\|_1 \\ & + \frac{1}{2} \sum_i \left[\lambda_i w_i^2 + \mu_i \|V_i\|_2^2\right] \quad (15) \\ \text{subject to } & V_{ij} = 0 \text{ for } j > k_i \end{aligned}$$

where the choice of constants λ_1, λ_i, μ_i and k_i are as discussed above. This model differs in two key parts from standard FM models: We add frequency adaptive regularization to better model the possible nonuniform sparsity pattern in the data. Secondly, we use sparsity regularization and memory adaptive constraints to control the model size, which benefits both the statistical model and system performance.

4. DISTRIBUTED OPTIMIZATION

Minimizing (15) is challenging since the potentially large model brings large communication cost. We focus on asynchronous optimization, which hides synchronization cost by communicating and computing in parallel. It has been shown that asynchronous block subspace descent can effectively solve nonconvex objective functions with a nonsmooth ℓ_1 regularizer [9]. However, it requires expensive data preprocessing. In this paper, we consider asynchronous stochastic gradient descent (SGD) [9] for optimization.

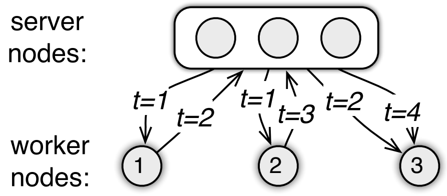


Figure 1: An illustration of asynchronous SGD. Workers W_1 and W_2 first pull weights from the servers at time 1. Next W_1 pushes the gradient back and therefore increase the timestamp at the servers. Worker W_3 pulls the weight immediately after that. Then W_2 and W_3 push the gradient back. The delays for W_1, W_2, and W_3 are 0, 1, and 1.

4.1 Asynchronous Stochastic Gradient Descent

For simplicity we assume that there is a single (virtual) server node, which maintains the model w and V (the parameter server framework provides for a clean abstraction for multiple servers). Multiple workers run SGD independently of each other. Each worker repeatedly reads data, pulls the recent model from the server, computes a gradient, and then pushes the gradient back. Due to the network delay, the received gradient the server used to update the model may be not computed based on the most recent model. An example is illustrated in Figure 1.

A worker can calculate the gradient of the loss in the standard way [15]. We first compute the partial gradient of f

$$\partial_{w_i} f(x, w, V) = x_i \quad (16)$$

$$\partial_{V_{ij}} f(x, w, V) = x_i [Vx]_j - x_i^2 V_{ij}, \quad (17)$$

Note that the term Vx can be pre-computed. Invoking the chain rule yields the gradient of l. Denote by $w(t)$ and $V(t)$ the model stored at the server on time t. Assume that at time t the server received gradient pushed from one worker

$$g^\theta(t) \leftarrow \partial_\theta l(f(x, w(t - \tau), V(t - \tau)), y) \quad (18)$$

where θ can be either w or V. τ is the delay, indicating that the gradient was computed using the model at time $t - \tau$.

As mentioned in Section 3.3, we use frequency adaptive regularization for w and V and the sparse induce ℓ_1 norm for w. In addition, we use AdaGrad [4] for a better model of the possible nonuniform sparsity in the data. In other words, assuming scalars λ in (14) and scalars η_V and β_V, the server updates V_{ij} by

$$n_{ij} \longleftarrow n_{ij} + \left[g_{ij}^V(t)\right]^2$$

$$V_{ij}(t+1) \longleftarrow V_{ij}(t) - \frac{\eta_V}{\beta_V + \sqrt{n_{ij}}} \left(g_{ij}^V(t) + \mu V_{ij}(t)\right) \quad (19)$$

where n_{ij} is initialized to 0. Updating w is slightly different to V due to the nonsmooth ℓ_1 regularizer. We adopted FTRL [11], which solves a "smoothed" proximal operator based on AdaGrad. Similarly denote by η_w and β_w the global learning rate. Updates in w_i are given by

$$\begin{aligned}
\sigma_i &\leftarrow \frac{1}{\eta_w}\left(\sqrt{n_i + [g_i^w(t)]^2} - \sqrt{n_i}\right) \\
z_i &\leftarrow z_i - g_i^w(t) + \sigma_i w_i(t) \\
n_i &\leftarrow n_i + [g_i^w(t)]^2 \qquad (20)
\end{aligned}$$

$$w_i(t+1) \leftarrow \begin{cases} 0 & \text{if } |z_i| \le \lambda_1 \\ \left(\frac{\beta_w + \sqrt{n_i}}{\eta_w} + \lambda_2\right)^{-1}(z_i - \operatorname{sgn}(z_i)\lambda_1) & \text{otherwise} \end{cases}$$

where both n_i and z_i is set to 0 at the beginning.

4.2 Convergence Analysis

The objective (15) is a challenging non-convex and non-smooth optimization problem and its convergence analysis is non-trivial, especially in the face of asynchronous updates. Here we provide a preliminary analysis of Algorithm 1 on a special case with $\lambda_1 = 0$. For notation simplicity we assume a fixed learning rate. It is straightforward to extend the results to handle the adaptive learning rate by assuming n_{ij} in (19) is bounded, see [9] for more details.

We will show that the simplified algorithm has an $O(1/\sqrt{t})$ ergodic convergence rate [5]. Due to the non-convexity, our results do not imply convergence to a KKT point. However, it is stronger than typical non-convex analysis in that there is an explicit convergence rate. Our results are based on a generic result in [10]. By carefully incorporating the specific properties of our problem, we arrive at a convergence bound that explicitly reveals the intuitive dependency on the sparsity of the data and the size of the minibatch in addition to the more typical maximum delay parameter for the asynchronous update. First we show Algorithm 1 reduces to simple asynchronous SGD.

Lemma 3 *Let η be the fixed learning rate for both parameter w and V, and in addition, $\lambda_1 = 0$, then the update equations (19) and (20) for solving the problem (15) becomes plain asynchronous SGD.*

Proof. First of all, we note that despite the memory adaptive constraints $V_{ij} = 0$ for some i, j, we are essentially solving an *unconstrained* smooth optimization problem in a pre-defined coordinate subspace, which admits a simple (non-projected) SGD algorithm.

It remains to show that the stochastic gradient is unbiased. Since the minibatch is picked randomly, the expectation of the loss-function half of the partial stochastic gradient with respect to V is just the gradient. Since λ_i is frequency adaptively chosen, it does not affect the expectation of the gradient at all. Now we turn to the partial gradient with respect to w. Under the assumption that $\lambda_1 = 0$ and without AdaGrad it follows that $n_i = 0$. Hence for every $t \in \{1, \ldots T\}$ the update equation (16) becomes

$$w_i(t+1) \leftarrow \frac{\eta_w}{\beta_w} z_i(t+1) \text{ where } z_i(t+1) \leftarrow z_i(t) - g_i^w(t).$$

Substituting in $z_i(1) = \frac{\beta_w}{\eta_w} w_i(1)$, this recursion is essentially the SGD update equation

$$w_i(t+1) \leftarrow w_i(t) - \frac{\eta_w}{\beta_w} g_i^w(t)$$

with learning rate $\eta = \frac{\eta_w}{\beta_w}$. To show that the approximation is unbiased is again trivial since the data points are picked uniformly at random and the regularization weights μ_i are frequency adaptive. □

Theorem 4 (Lian et al. [10, Corollary 2]) *Assume that the stochastic gradient is unbiased and has variance bounded by σ^2 and that the gradient functional $\nabla f(\cdot)$ is L-Lipschitz. Also, assume that the global optimal solution x^* exists and $f^* > -\infty$. Moreover, assume the delay at time t τ_t is upper bounded by τ, then if we use a constant stepsize*

$$\eta := \sqrt{\frac{f(x_1) - f(x^*)}{L\tau\sigma^2}}.$$

then for every integer $T \ge \frac{4L(f(x_1) - f(x^*))}{\sigma}(\tau+1)^2$, we have the output of the SGD obeying

$$\frac{1}{T}\sum_{t=1}^{T} \mathbb{E}(\|\nabla f(x_t)\|^2) \le 4\sqrt{\frac{(f(\theta_1) - f(\theta^*))L\sigma^2}{T}} \qquad (21)$$

Simplifying it further in our problem results in the following corollary, where the constants explicitly depend on the average sparsity of the data and size of the minibatch we use.

Corollary 1 *Denote by $\theta = \{w, V\}$ the parameters of the model and let $\phi_i(\theta) = l(f(x_i), y_i)$ be the objective function. Moreover, decompose the objective via*

$$\Phi(\theta) = \frac{1}{|X|}\sum_i \phi_i(\theta) + \sum_j [\lambda_j w_j^2 + \mu_j \|V_j\|^2].$$

Let the delay of the asynchronous gradient $\tau_t \le \tau$ for any update t and the minibatch size be b. Assume that the data X is bounded. More specifically, assume that it obeys the following property $s_i := \|x_i\|_0$, $\|X\|_\infty \le 1$ and $s_i \le s$.

In addition, assume that we work within a sublevel set of the problem such that $\|[w_{\text{supp}(x_i)}, V_{\text{supp}(x_i \otimes x_i)}]\| \le B_i$. Furthermore, assume that the regularization parameter is sufficiently small so that the regularization term does not weight more than the data term in the gradient.

Then there is a universal constant C and a data (sparsity) dependent constant

$$K := \frac{1}{|X|b} \max_i (s_i B_i)^2 \sum_{i=1}^{|X|} (s_i B_i)$$

such that setting stepsize $\eta = \sqrt{\frac{C\phi(\theta_1)}{\tau K}}$ results in a stochastic sequence of parameters that satisfies

$$\min_{t \in \{1,\ldots,T\}} \mathbb{E}\|\nabla\phi(\theta_t)\|^2 \le \frac{1}{T}\sum_{t=1}^{T} \mathbb{E}\|\nabla\phi(\theta_t)\|^2 \le 4\sqrt{\frac{\phi(\theta_1)\tau K}{T}}.$$

We first interpret the result before stating the proof.

1. The expected magnitude of the gradient on the LHS is an intuitive measure of the distance from the stationarity condition. While it is weaker than typical measures in convex optimization (e.g., primal suboptimality, mean square distance from the optimal solution), it should be in the same ball park for "nice" loss functions, as can be seen e.g. in discussions in [5, 10].
2. When the data is sparser or the minibatch size is bigger, we can afford to use a larger learning rate and hence get faster convergence. Note that the results

captures the average sparsity so the quantity remains small even when a small number of data points are dense. This is especially important in face of the practical power law distributions.

Proof of Corollary 1. This is a simple specialization of Theorem 4 to our specific problem, which involves calculating the Lipschitz constant of the gradient and the variance of the stochastic gradient and upper bounding them using the intuitive quantities of interests. By the chain rule

$$\partial_\theta \phi_i(\theta) = \frac{\partial}{\partial f(x_i)} l(f(x_i), y_i) \frac{\partial}{\partial \theta} f(x_i).$$

Since l is the logistic loss, its Lipschitz constant is bounded by 1. Hence it follows from (16) and (17) that

$$\begin{aligned}\|\partial_\theta \phi_i(\theta)\| &\leq 1 \cdot \sqrt{\sum_j (\partial_{w_j} f(x_i))^2 + \sum_{j,\ell} (\partial_{V_{j\ell}} f(x_i))^2} \\ &= \sqrt{\sum_j x_i^2 + \sum_{j,\ell} (x_j [Vx]_\ell - x_j^2 V_{j\ell})^2} \\ &\leq \sqrt{s_i} \|x_i\|_\infty + s_i \|x_i [Vx_i]^T\|_{2,\infty} \leq s_i B_i\end{aligned}$$

Since the objective is differentiable, the Lipschitz constant can be obtained by upper bounding the gradient

$$\begin{aligned}\|\partial_\theta \phi(\theta)\| &\leq \left\| \frac{1}{|X|} \sum_{i=1}^{|X|} \partial_\theta \phi_i(\theta) + 2\Lambda\theta \right\| \\ &\leq \frac{1}{|X|} \sum_{i=1}^{|X|} s_i B_i + \|\Lambda\theta\|_2 \leq \frac{2}{|X|} \sum_{i=1}^{|X|} s_i B_i\end{aligned}$$

where Λ is a big diagonal matrix that contains the frequency adaptive regularization weights λ_j and μ_j and the last inequality holds whne these weights are small.

The variance of the stochastic gradient is taken over an iid minibatch of size b, and it can be trivially bounded using the boundedness of the gradient

$$\sigma^2 \leq \frac{\max_i \|\partial_\theta \phi_i(\theta)\|^2}{b} \leq \frac{\max_i (s_i B_i + B)^2}{b} \leq \frac{4 \max_i (s_i B_i)^2}{b}.$$

The last inequality can again be simplified under the assumptions that the regularization terms are small. Finally, we note that $\phi(\theta) \geq 0$ for any θ, so $\phi(\theta^*) \geq 0$. We arrive at the claim by substituting these bounds into (21) in Theorem 4. The proof is complete by noting that the minimum is always smaller than the mean in a sequence of numbers. □

4.3 Implementation

The sketch of DiFacto is shown in Algorithm 1. It is divided into three parts: the scheduler node runs the control logic, server nodes update the model, and the worker nodes compute the gradient.

Scheduler node issues commands, such as *process file i* or *save the model* to worker and server nodes. It also monitors the progress of workers. Once a straggler or a dead node is detected, the scheduler will re-issue the command which was sent to this node to another available node. Furthermore, the scheduler decides whether the stopping criteria have been reached. For example, we may terminate once the objective value on an additional validation set stops decreasing.

Algorithm 1 Implement DiFacto in the parameter server

```
Start: Create one scheduler node, $m$ worker nodes and $n$ server nodes over multiple machines.
Scheduler Node:
 1: Assume the data is partitioned into $s$ parts $p_1, \ldots, p_s$,
 2: for $t = 1$ to $T$ do
 3:     Work packages $\mathcal{P} = \{p_1, \ldots, p_s\}$
 4:     Accomplished packages $\mathcal{A} = \emptyset$
 5:     while $\mathcal{P} \neq \emptyset$ do
 6:         switch detected event from worker $i$ do
 7:             case idle
 8:                 Pick $p \in \mathcal{P} \setminus \mathcal{A}$ and assign $p$ to worker $i$
 9:                 $\mathcal{A} = \mathcal{A} \cup \{p\}$,
10:             case finished $p$
11:                 $\mathcal{P} = \mathcal{P} \setminus \{p\}$
12:             case dead or timeout
13:                 $\mathcal{A} = \mathcal{A} \setminus \{p\}$,
14:     end while
15: end for
Worker $i$:
 1: Receive command "processing $p$" from the scheduler
 2: while read a minibatch from $p$ do
 3:     Pull $w_i$ and $V_i$ from server nodes for all features $i$ that appear in this minibatch
 4:     Compute the gradient based on (16) and (17)
 5:     Push gradient back to servers
 6: end while
Server $i$:
 1: if received gradient from a worker then
 2:     update $w$ and $V$ by using (20) and (19)
 3: end if
```

Server nodes maintain and update the model w and V. Due to the adaptive memory constraints, the elements of V_i could be partly (or entirely) zeros. Therefore a server node only needs to allocate physical memory to a nonzero element V_i to reduce the memory footprint. This also saves bandwidth when handling model pull requests from the worker nodes. Conversely, when receiving gradients from a worker node, the server node updates the model using the steps discussed above.

Worker nodes perform most of the computation. After receiving a *process data p* command from the scheduler, a worker repeatedly reads minibatches from p, which is often a file stored in a distributed filesystem. For each minibatch B, the worker first finds the supporting feature indices $\mathcal{I} = \{i : x_i \neq 0 \text{ for some } x \in B\}$. Then it pulls (prefetches) the working set of the model from server nodes, namely $\{w_i, V_i : i \in \mathcal{I}\}$. Next the worker calculates the gradient of this minibatch and then pushes them to the servers.

Note that a worker node only needs to cache the minibatches being processed and the associated working sets to minimize the memory consumption. It also parallelizes data IO, computation, and communication (in particular prefetching) to hide IO cost. Besides, a worker node uses various filters to reduce the amount of data that need to be communicated. We adopt standard filters provided by the parameter server [8] including key caching (caching the keys in both sender and receiver to avoid duplicated communica-

name	# examples	# features	# entries
Criteo1 [1]	4.6×10^7	3.4×10^7	1.5×10^9
CTR1	2.9×10^5	1.4×10^7	3.5×10^7
Criteo2 [2]	1.5×10^9	3.6×10^8	5.0×10^{10}
CTR2 [3]	1.1×10^8	1.9×10^9	1.3×10^{10}

Table 1: A collection of click-through rate datasets.

tion) and lossless data compression. In addition, we use fixed-point encoding. Namely we convert the floating-point weight and gradient into shorter fixed-point integers during communication.

The codes are publicly available as part of the DMLC project http://dmlc.github.io/.

5. EXPERIMENTS

Quite clearly Factorization Machines are useful for many problems *beyond* computational advertising. That said, we feel that CTR estimation offers the most difficult challenges, both in terms of data set size and dimensionality.

5.1 Setup

Datasets: We collected several click-through rate datasets at various scales, as shown in Table 1. Criteo1 and Criteo2 are display advertising datasets from Criteo. The former is used in a recent Kaggle competition, for which we used the first 80% examples for training and the rest for test. The latter is by far the largest public released click-through rate dataset. We used the first 10 days data for training, and the 11-th day data for test. In both Criteo1 and Criteo2, each example has 13 integer features and 26 category features. We extracted sparse binary features via the widely-used one-hot encoding.

CTR2 is another advertising dataset sampled from a one month period from an anonymous Internet company. We used the on-production feature extraction module to obtain sparse binary anonymized features. Compared to the Criteo datasets, CTR2 has more 3 times more features per example due to the extensive feature engineering. We sampled a smaller dataset CTR1 from CTR2 for comparison with other systems which cannot scale to CTR2.

Machines: All experiments with performance numbers were run on Amazon EC2 c4.8xlarge instances. Each instance is equipped with dual Intel E5-2666 V3 2.8 GHz CPUs, 60 GB memory, and 10 Gigabit Ethernet. Auxiliary experiments, such as tuning the hyperparameters were carried out on a local cluster with 10 machines, with dual Intel E5-2680 v2 2.80GHz CPUs, 128 GB memory, and 40 GbE.

Hyperparameters: Unless stated otherwise, we choose the hyperparameters as follows: We run DiFacto with 100 workers and 100 servers on 10 physical machines. We fixed the dimension $k = 16$ and the minibatch size 10^4 for Criteo2 and 10^3 for CTR2. While the smaller dataset Criteo1 and CTR1, we decrease the minibatch sizes by 10 times respectively. We used a fixed regularizer for w with $\lambda_1 = 4$ and $\lambda = 0$, while choosing the constant μ for V within the range of $\{0, 10^{-5}, 10^{-4}, 10^{-3}\}$ based on an additional validation set.

[1]https://www.kaggle.com/c/criteo-display-ad-challenge
[2]http://labs.criteo.com/downloads/download-terabyte-click-logs
[3]https://github.com/mli/data

The elements in V were initialized uniformly at random in the range $[-0.01, 0.01]$. The fast learning rate η_w is selected between 0.001 and 0.1 with $\eta_V = \eta_w$ and $\beta_w = \beta_V = 1$. Finally, we stop the algorithm when the objective value on the validation set stops decreasing.

5.2 Adaptive memory

We first study the effectiveness of adaptive memory. We compare the following three setups which use different memory adaptive constraints in (15):

No memory adaption No constraint is applied, i.e. we do not force elements in V_i to 0.

Frequency threshold Only the constraint $V_i = 0$ for infrequent keys $n_i < k$ is used, where n_i is the occurrence of feature i in the data.

Frequency threshold + ℓ_1 shrinkage We add the constraint $V_i = 0$ if $w_i = 0$. That is, we mark V_i as inactive if w_i is set to 0 by the sparse induction ℓ_1 regularization.

We use the same hyper-parameters for all three setups and vary the dimension k from 1 to 64. The results are shown in Figure 2.

As can be seen, these memory adaptive constraints effectively reduce the model size. The reduction is about 100x for Criteo2 and 300x for CTR2 when $k = 64$. Therefore these constraints significantly decrease the server node memory consumption and network traffic. It also brings about a 20% reduction in the runtime. Especially when k is large, the resulted amount of communication may overwhelm the system. Being able to reduce the model size benefits the system performance a lot.

A more interesting observation is that these memory adaptive constraints do not affect the test accuracy. To the contrary, we even see a slight improvement when the dimension k is greater than 8 for CTR2. The reason could be that the model capacity control is of great importance when the dimension k is large. And these memory adaptive constraints can provide additional capacity control besides the ℓ_2 and ℓ_1 regularizers.

5.3 Fixed-point Compression

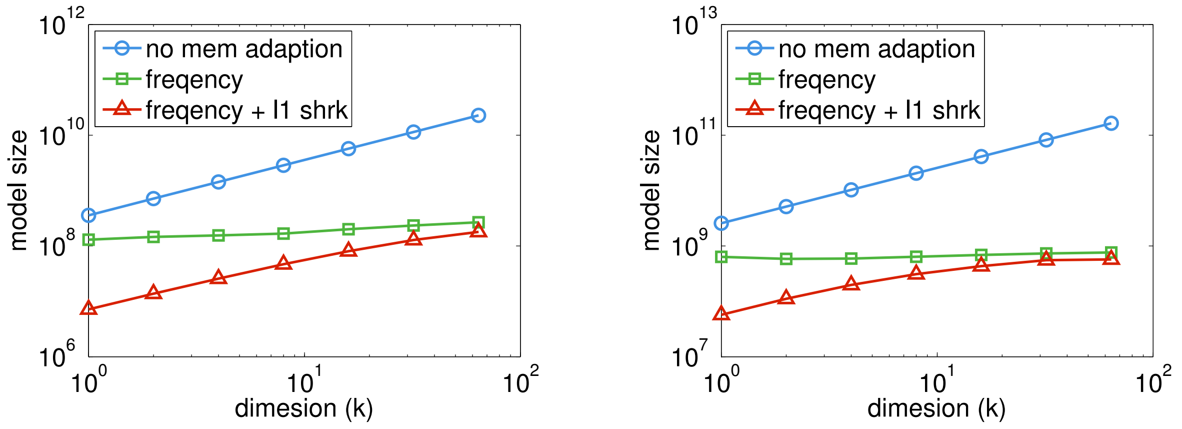
We evaluate lossy fixed-point compression for data communication. By default, both the model and gradient entries are represented as 32 bit floats. In this experiment, we compress these values to lower precision integers. More specifically, given a bin size b and number of bits n, we represent x by the following n-bit integer

$$z := \left\lfloor \frac{x}{b} \times 2^n \right\rfloor + \sigma, \tag{22}$$

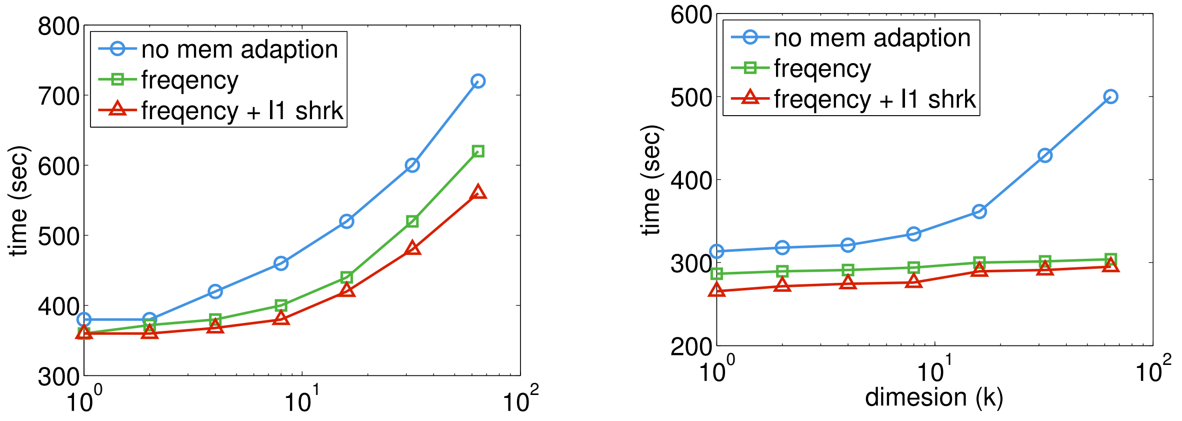
where σ is a Bernoulli random variable chosen such as to ensure that $\mathbf{E}[z] = 2^n \frac{x}{b}$.

We implemented the fixed-point compression as a user-defined filter in the parameter server framework. Since multiple numbers are communicated in each round, we choose b to be the absolute maximum value of these numbers. In addition, we used the key caching and lossless data compression (via LZ4) filters.

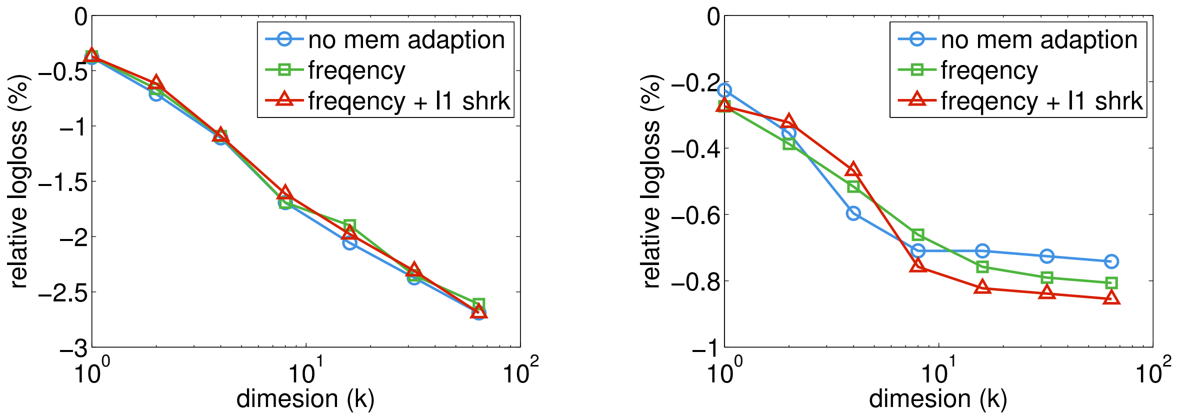
The results for $n = 8, 16, 24$ are shown in Figure 3. As expected, fixed-point compression linearly reduces the network traffic volume, since the traffic is dominated by communicating the model and gradient. A more interesting observation is that we obtained a 4.2x compression rate from 32-bit



(a) Number of non-zero entries in V.



(b) Runtime for one iteration.



(c) Relative test logloss comparing to logistic regression ($k = 0$ and 0 relative loss).

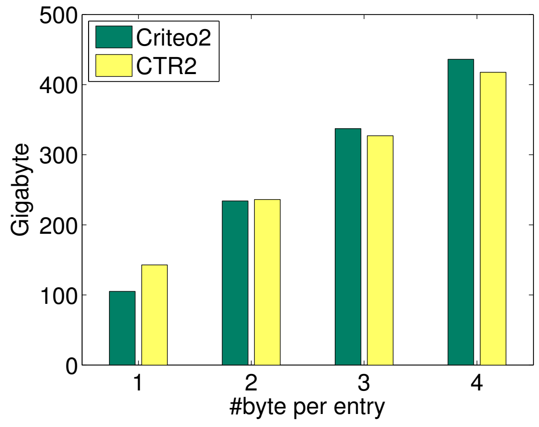
Figure 2: Using different adaptive memory constraints when varying the embedding dimension. Left: Criteo2. Right: CTR2.

floating-point to 8-bit fixed-point on Criteo2. The reason is the latter improves the compression rate for the following lossless LZ4 compression.

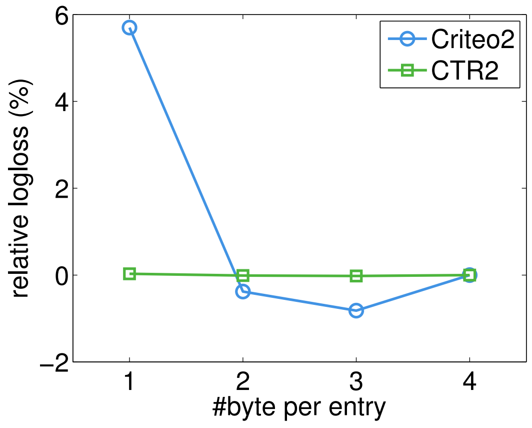
We observed different effects of accuracy on these two datasets: CTR2 is robust to the number precision, while Criteo2 has a 6% increase of logloss if only using 1-byte presentation. However, a medium compression rate even improves the model accuracy. This might be because the lossy compression acts as regularizer to the objective function.

5.4 Comparison with LibFM

To our best knowledge, there is no publicly released distributed FM solver. Hence we only compare DiFacto to the popular single machine package LibFM developed by Rendle [15]. We only report results on Criteo1 and CTR1 on a single machine, since LibFM fails on the other two larger datasets. We perform a similar grid search of the hyperparameters as we did for DiFacto. As LibFM only uses single thread, we run DiFacto with 1 worker and 1 server

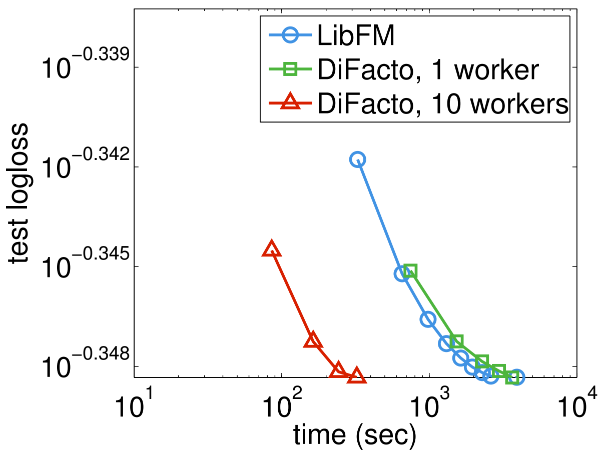


(a) Total data sent by workers in one iteration. The compression rates from 4-byte to 1-byte are 4.2x and 2.9x for Criteo2 and CTR2, respectively.

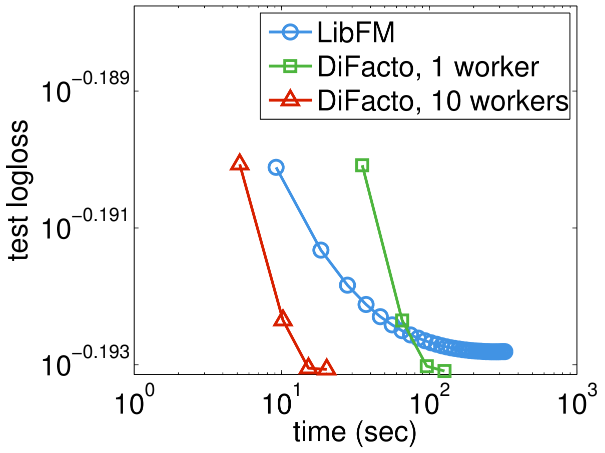


(b) The relative test logloss comparing to no fixed-point compression.

Figure 3: Compressing model and gradient using the fixed-point compression, where 4-byte means using the default 32-bit floating-point format.



(a) Criteo1



(b) CTR1

Figure 4: Comparison with LibFM on a single machine. The data preprocessing time for LibFM is omitted.

in sequential execution order. We also report the performance using 10 workers and 10 servers on a single machine for reference.

The results are shown in Figure 4. As can been seen, DiFacto converges significantly faster than LibFM, it uses 2 times fewer iterations to reach the best model. This is because the adaptive learning rate used in DiFacto better models the data sparsity and the adaptive regularization and constraints can further accelerate the convergence. In particular, the latter results in a lower test logloss on the CTR1 dataset, where the number of features exceeds the number of examples, requiring improved capacity control.

Also note that DiFacto with a single worker is twice slower than LibFM per iteration. This is because the data communication overhead between the worker and the server cannot be ignored in the sequential execution. More importantly, DiFacto does not require any data preprocessing to map arbitrary 64-bit integer and string feature indices, which are used in both Criteo1 and CTR2, to continuous integer indices. The cost of this data preprocessing step, required by LibFM but not shown in Figure 4, even exceeds the cost of training (1,400 seconds for Criteo1). Nevertheless, DiFacto with a single worker still outperforms LibFM thanks to the faster convergence. In addition, it is 10 times faster than LibFM when using 10 workers on the same machine.

5.5 Scalability

Finally, we study the scalability of DiFacto by varying the number of physical machines used in training. We run 10 workers and 10 servers in each machine, and increase the number of machines from 1 to 16. Both the time for each iteration and the logloss on the test dataset are recorded and shown in Figure 5.

We observe an 8x speedup from 1 machine to 16 machines on both Criteo2 and CTR2. The reason for the satisfactory performance is twofold. First, asynchronous SGD eliminates the need for synchronization between workers and it is tolerant to stragglers. Even though we used dedicated machines for the job, they still share network bandwidth with others. In particular, we observed a large variation of read speed when streaming data from Amazon's S3 service, despite using the IO optimized c4.8xlarge series of machines. This ensures that we actually take advantage of parallelization relative to LibFM, which is single core.

Second, DiFacto uses several filters which effectively re-

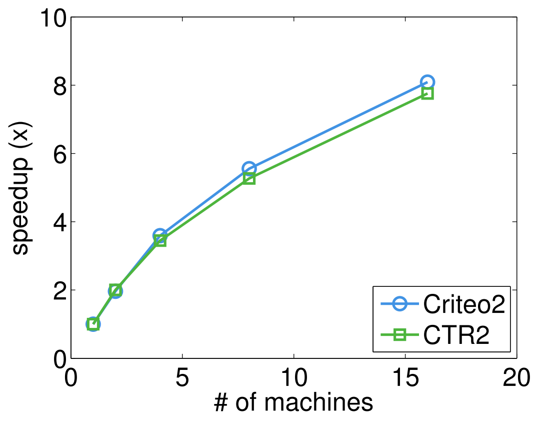


Figure 5: The speedup from 1 machine to 16 machines, where each machine runs 10 workers and 10 servers. The difference between the test logloss is within 0.5%, which is omitted in the figure.

duce the amount of network traffic. Even though CTR2 produces 10 times more network traffic than Criteo2, they have similar speedup performance.

There is a longstanding suspicion that the convergence of asynchronous SGD slows down when increasing the number of workers. Nonetheless, we did not observe a substantial difference in model accuracy. In other words, the relative difference of the objective logloss on test datasets is below 0.5% when increasing the number of workers from 10 to 160. This might be because the datasets we used are highly sparse and the features are not extremely correlated. Hence inconsistency due to concurrently updating by multiple workers may not have a major effect. These observations correspond well with what we predict in the convergence analysis.

6. CONCLUSION

In this paper we presented DiFacto, a high performance distributed factorization machine. DiFacto uses a refined factorization machine model with adaptive memory constraints and frequency adaptive regularization, which perform fine-grained capacity control based on both data and model statistics. DiFacto uses asynchronous stochastic gradient descent. We give theoretical convergence analysis and implement it in the parameter server framework. We evaluated DiFacto thoroughly on two real computational advertising datasets with up to billions of examples and features. We showed that DiFacto produces a very compact model, yet it retains similar generalization accuracy. We also demonstrated that DiFacto converges faster than the state-of-the-art package LibFM. Furthermore, DiFacto is able to solve TB scale problems on modest machines within hours.

References

[1] A. Ahmed, N. Shervashidze, S. Narayanamurthy, V. Josifovski, and A. J. Smola. Distributed large-scale natural graph factorization. In *World Wide Web Conference*, Rio de Janeiro, 2013.

[2] R. M. Bell and Y. Koren. Lessons from the netflix prize challenge. *SIGKDD Explorations*, 9(2):75–79, 2007. URL `http://doi.acm.org/10.1145/1345448.1345465`.

[3] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Ng. Large scale distributed deep networks. In *Neural Information Processing Systems*, 2012.

[4] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2010.

[5] S. Ghadimi and G. Lan. Stochastic first-and zeroth-order methods for nonconvex stochastic programming. *SIAM Journal on Optimization*, 23(4):2341–2368, 2013.

[6] Criteo Labs. Criteo terabyte click logs, 2014. http://labs.criteo.com/downloads/download-terabyte-click-logs.

[7] Q.V. Le, T. Sarlos, and A. J. Smola. Fastfood — computing hilbert space expansions in loglinear time. In *International Conference on Machine Learning*, 2013.

[8] M. Li, D. G. Andersen, J. Park, A. J. Smola, A. Amhed, V. Josifovski, J. Long, E. Shekita, and B. Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[9] M. Li, D. G. Andersen, A. J. Smola, and K. Yu. Communication efficient distributed machine learning with the parameter server. In *Neural Information Processing Systems*, 2014.

[10] X. Lian, Y. Huang, Y. Li, and J. Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. *arXiv preprint arXiv:1506.08272*, 2015.

[11] B. McMahan. Follow-the-regularized-leader and mirror descent: Equivalence theorems and l1 regularization. In *International Conference on Artificial Intelligence and Statistics*, pages 525–533, 2011.

[12] B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, and D. Golovin. Ad click prediction: a view from the trenches. In *KDD*, 2013.

[13] T. Moon, A. J. Smola, Y. Chang, and Z. Zheng. Intervalrank: isotonic regression with listwise and pairwise constraints. In B.D. Davison, T. Suel, N. Craswell, and B. Liu, editors, *Proceedings of the Third International Conference on Web Search and Web Data Mining, WSDM*, pages 151–160. ACM, 2010.

[14] S. Negahban, P. Ravikumar, M.J. Wainwright, and B. Yu. A unified framework for high-dimensional analysis of m-estimators with decomposable regularizers. *arXiv preprint arXiv:1010.2731*, 2010.

[15] S. Rendle and L. Schmidt-Thieme. Pairwise interaction tensor factorization for personalized tag recommendation. In *Web search and data mining*, pages 81–90. ACM, 2010.

[16] S. Rendle. Time-Variant Factorization Models Context-Aware Ranking with Factorization Models. volume 330 of *Studies in Computational Intelligence*, chapter 9, pages 137–153. 2011. ISBN 978-3-642-16897-0.

[17] A. J. Smola and S. Narayanamurthy. An architecture for parallel topic models. In *Very Large Databases (VLDB)*, 2010.

[18] N. Srebro, N. Alon, and T. Jaakkola. Generalization error bounds for collaborative prediction with low-rank matrices. In L. K. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems 17*, Cambridge, MA, 2005. MIT Press.

[19] B. Taskar, C. Guestrin, and D. Koller. Max-margin Markov networks. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*, pages 25–32, Cambridge, MA, 2004. MIT Press.

[20] R. Tibshirani. Regression shrinkage and selection via the lasso. *J. R. Stat. Soc. Ser. B Stat. Methodol.*, 58:267–288, 1996.

[21] G. Wahba. *Spline Models for Observational Data*, volume 59 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM, Philadelphia, 1990.

[22] J. Ye, J.-H. Chow, J. Chen, and Z. Zheng. Stochastic gradient boosted distributed decision trees. In D. W.-L. Cheung, I.-Y. Song, W.W. Chu, X. Hu, and J.J. Lin, editors, *Conference on Information and Knowledge Management, CIKM*, pages 2061–2064. ACM, 2009.