A Learning-rate Schedule for Stochastic Gradient Methods to Matrix Factorization

Wei-Sheng Chin, Yong Zhuang, Yu-Chin Juan, and Chih-Jen Lin

Department of Computer Science
National Taiwan University, Taipei, Taiwan
{d01944006,r01922139,r01922136,cjlin}@csie.ntu.edu.tw

Abstract. Stochastic gradient methods are effective to solve matrix factorization problems. However, it is well known that the performance of stochastic gradient method highly depends on the learning rate schedule used; a good schedule can significantly boost the training process. In this paper, motivated from past works on convex optimization which assign a learning rate for each variable, we propose a new schedule for matrix factorization. The experiments demonstrate that the proposed schedule leads to faster convergence than existing ones. Our schedule uses the same parameter on all data sets included in our experiments; that is, the time spent on learning rate selection can be significantly reduced. By applying this schedule to a state-of-the-art matrix factorization package, the resulting implementation outperforms available parallel matrix factorization packages.

Keywords: Matrix factorization, stochastic gradient method, learning rate schedule

1 Introduction

Given an incomplete matrix $R \in \mathbb{R}^{m\times n}$, matrix factorization (MF) finds two matrices $P \in \mathbb{R}^{k\times m}$ and $Q \in \mathbb{R}^{k\times n}$ such that $r_{u,v} \simeq \boldsymbol{p}_u^T \boldsymbol{q}_v, \forall u, v \in \Omega$, where Ω denotes the indices of the existing elements in R, $r_{u,v}$ is the element at the uth row and the vth column in R, $\boldsymbol{p}_u \in \mathbb{R}^k$ is the uth column of P, $\boldsymbol{q}_v \in \mathbb{R}^k$ is the vth column of Q, and k is the pre-specified number of latent features. This task is achieved by solving the following non-convex problem

$$\min_{P,Q} \quad \sum_{(u,v)\in\Omega}(r_{u,v} - \boldsymbol{p}_u^T\boldsymbol{q}_v)^2 + \lambda(\|\boldsymbol{p}_u\|^2 + \|\boldsymbol{q}_v\|^2), \tag{1}$$

where λ is a regularization parameter. Note that the process to solve P and Q is referred to as the training process. To evaluate the quality of the used solver, we can treat some known elements as missing in the training process and collect them as the test set. Once P and Q are found, root-mean-square error (RMSE) on the test set is often used as an evaluation criterion. It is defined as

$$\sqrt{\frac{1}{|\Omega_{\text{test}}|}\sum_{(u,v)\in\Omega_{\text{test}}} e_{u,v}^2}, \quad e_{u,v} = r_{u,v} - \boldsymbol{p}_u^T\boldsymbol{q}_v, \tag{2}$$

where Ω_{test} represents the indices of the elements belonging to test set.

Matrix factorization is widely used in recommender systems [11], natural language processing [16], and computer vision [9]. Stochastic gradient method[1](SG) is an iterative procedure widely used to solve (1), e.g., [7,14,2]. At each step, a single element $r_{u,v}$ is sampled to obtain the following sub-problem.

$$(r_{u,v} - \boldsymbol{p}_u^T \boldsymbol{q}_v)^2 + \lambda(\|\boldsymbol{p}_u\|^2 + \|\boldsymbol{q}_v\|^2). \tag{3}$$

The gradient of (3) is

$$\boldsymbol{g}_u = \frac{1}{2}(-e_{u,v}\boldsymbol{q}_v + \lambda \boldsymbol{p}_u), \quad \boldsymbol{h}_v = \frac{1}{2}(-e_{u,v}\boldsymbol{p}_u + \lambda \boldsymbol{q}_v). \tag{4}$$

Note that we drop the coefficient $1/2$ to simplify our equations. Then, the model is updated along the negative direction of the sampled gradient,

$$\boldsymbol{p}_u \leftarrow \boldsymbol{p}_u - \eta \boldsymbol{g}_u, \quad \boldsymbol{q}_v \leftarrow \boldsymbol{q}_v - \eta \boldsymbol{h}_v, \tag{5}$$

where η is the learning rate. In this paper, an update of (5) is referred to as an iteration, while $|\Omega|$ iterations are called an outer iteration to roughly indicate that all $r_{u,v}$ have been handled once. Algorithm 1 summarizes the SG method for matrix factorization. In SG, the learning rate can be fixed as a constant while some schedules dynamically adjust η in the training process for faster convergence [4]. The paper aims to design an efficient schedule to accelerate the training process for MF.

Algorithm 1 Stochastic gradient methods for matrix factorization.

Require: Z: user-specified outer iterations
1: **for** $z \leftarrow 1$ to Z **do**
2: **for** $i \leftarrow 1$ to $|\Omega|$ **do**
3: sample $r_{u,v}$ from R
4: calculate sub-gradient by (4)
5: update $\boldsymbol{p}_u$ and $\boldsymbol{q}_v$ by (5)
6: **end for**
7: **end for**

The rest sections are organized as follows. Section 2 investigates the existing schedules for matrix factorization and a per-coordinate schedule for online convex problems. Note that a per-coordinate schedule assigns each variable a distinct learning rate. We improve upon the per-coordinate schedule and propose a new schedule in Section 3. In Section 4, experimental comparisons among schedules and state-of-the-art packages are exhibited. Finally, Section 5 summarizes this paper and discusses potential future works. In summary, our contributions include:

1. We propose a new schedule that outperforms existing schedules.
2. We apply the proposed schedule to an existing package. The resulting implementation, which will be publicly available, outperforms state-of-the-art parallel matrix factorization packages.

2 Existing Schedules

In Section 2.1, we investigate three schedules that are commonly used in matrix factorization. The per-coordinate schedule that inspired the proposed method is introduced in Section 2.2.

[1] It is often called stochastic gradient descent method. However, it is actually not a "descent" method, so we use the term stochastic gradient method in this paper.

2.1 Existing Schedules for Matrix Factorization

Fixed Schedule (FS) The learning rate is fixed throughout the training process. That is, η equals to η_0, a pre-specified constant. This schedule is used in, for example, [8].

Monotonically Decreasing Schedule (MDS) This schedule decreases the learning rate over time. At the zth outer iteration, the learning rate is

$$\eta^z = \frac{\alpha}{1 + \beta \cdot z^{1.5}},$$

where α and β are pre-specified parameters. In [19], this schedule is used. For general optimization problems, two related schedules [12,6,10] are

$$\eta^z = \frac{\alpha}{z} \text{ and } \eta^z = \frac{\alpha}{z^{0.5}}, \tag{6}$$

but they are not included in some recent developments for matrix factorization such as [4,19]. Note that [4] discusses the convergence property for the use of (6), but finally chooses another schedule, which is introduced in the next paragraph, for faster convergence.

Bold-driver Schedule (BDS) Some early studies on neural networks found that the convergence can be dramatically accelerated if we adjust the learning rate according to the change of objective function values through iterations [15,1]. For matrix factorization, [4] adapts this concept and considers the rule,

$$\eta^{z+1} = \begin{cases} \alpha\eta^z & \text{if } \Delta_z < 0 \\ \beta\eta^z & \text{otherwise,} \end{cases} \tag{7}$$

where $\alpha \in (1, \infty)$, $\beta \in (0, 1)$, and $\eta^0 \in (0, \infty)$ are pre-specified parameters, and Δ_z is the difference on the objective function in (1) between the beginning and the end of the zth outer iteration. Clearly, this schedule enlarges the rate when the objective value is successfully decreased, but reduces the rate otherwise.

2.2 Per-coordinate Schedule (PCS)

Some recent developments discuss the possibility to assign the learning rate coordinate-wisely. For example, ADAGRAD [3] is proposed to coordinate-wisely control the learning rate in stochastic gradient methods for convex online optimization. For matrix factorization, if $r_{u,v}$ is sampled, ADAGRAD adjusts two matrices G_u and H_v using

$$G_u \leftarrow G_u + \boldsymbol{g}_u \boldsymbol{g}_u^T, \quad H_v \leftarrow H_v + \boldsymbol{h}_v \boldsymbol{h}_v^T,$$

and then updates the current model via

$$\boldsymbol{p}_u \leftarrow \boldsymbol{p}_u - \eta_0 G_u^{-1/2} \boldsymbol{g}_u, \quad \boldsymbol{q}_v \leftarrow \boldsymbol{q}_v - \eta_0 H_v^{-1/2} \boldsymbol{h}_v. \tag{8}$$

ADAGRAD also considers using only the diagonal elements because matrix inversion in (8) is expensive. That is, G_u and H_v are maintained by

$$G_u \leftarrow G_u + \begin{bmatrix} (\boldsymbol{g}_u)_1^2 & & \\ & \ddots & \\ & & (\boldsymbol{g}_u)_k^2 \end{bmatrix}, \quad H_v \leftarrow H_v + \begin{bmatrix} (\boldsymbol{h}_v)_1^2 & & \\ & \ddots & \\ & & (\boldsymbol{h}_v)_k^2 \end{bmatrix}. \tag{9}$$

We consider the setting of using diagonal matrices in this work, so the learning rate is related to the squared sum of past gradient elements.

While ADAGRAD has been shown to be effective for online convex classification, it has not been investigated for matrix factorization yet. Similar to ADAGRAD, other per-coordinate learning schedules such as [20,13] have been proposed. However, we focus on ADAGRAD in this study because the computational complexity per iteration is the lowest among them.

3 Our Approach

Inspired by PCS, a new schedule, *reduced per-coordinate schedule* (RPCS), is proposed in Section 3.1. RPCS can reduce the memory usage and computational complexity in comparison with PCS. Then, in Section 3.2 we introduce a technique called *twin learners* that can further boost the convergence speed of RPCS. Note that we provide some experimental results in this section to justify our argument. See Section 4 for the experimental settings such as parameter selection and the data sets used.

3.1 Reduced Per-coordinate Schedule (RPCS)

The cost of implementing FS, MDS, or BDS schedules is almost zero. However, the overheads incurred by PCS can not be overlooked. First, each coordinate of $\boldsymbol{p}_u$ and $\boldsymbol{q}_v$ has its own learning rate. Maintaining G_u and H_v may need $\mathcal{O}((m+n)k)$ extra space. Second, at each iteration, $\mathcal{O}(k)$ additional operations are needed for calculating and using diagonal elements of G_u and H_v.

Algorithm 2 One iteration of SG algorithm when RPCS is applied.

$e_{u,v} \leftarrow r_{u,v} - \boldsymbol{p}_u^T \boldsymbol{q}_v$
$\bar{G} \leftarrow 0, \quad \bar{H} \leftarrow 0$
$\eta_u \leftarrow \eta_0 (G_u)^{-\frac{1}{2}}, \quad \eta_v \leftarrow \eta_0 (H_v)^{-\frac{1}{2}}$
for $d \leftarrow 1$ to k **do**
  $(\boldsymbol{g}_u)_d \leftarrow -e_{u,v}(\boldsymbol{q}_v)_d + \lambda(\boldsymbol{p}_u)_d$
  $(\boldsymbol{h}_v)_d \leftarrow -e_{u,v}(\boldsymbol{p}_u)_d + \lambda(\boldsymbol{q}_v)_d$
  $\bar{G} \leftarrow \bar{G} + (\boldsymbol{g}_u)_d^2, \quad \bar{H} \leftarrow \bar{H} + (\boldsymbol{h}_v)_d^2$
  $(\boldsymbol{p}_u)_d \leftarrow (\boldsymbol{p}_u)_d - \eta_u (\boldsymbol{g}_u)_d$
  $(\boldsymbol{q}_v)_d \leftarrow (\boldsymbol{q}_v)_d - \eta_v (\boldsymbol{h}_v)_d$
end for
$G_u \leftarrow G_u + \bar{G}/k, \quad H_v \leftarrow H_v + \bar{H}/k$

These overheads can be dramatically reduced if we apply the same learning rate for all elements in $\boldsymbol{p}_u$ (or $\boldsymbol{q}_v$). Specifically, at each iteration, G_u and H_v are reduced from matrices to scalars. Instead of (9), G_u and H_v are now updated by

$$G_u \leftarrow G_u + \frac{\boldsymbol{g}_u^T \boldsymbol{g}_u}{k}, \quad H_v \leftarrow H_v + \frac{\boldsymbol{h}_v^T \boldsymbol{h}_v}{k}. \tag{10}$$

In other words, the learning rate of $\boldsymbol{p}_u$ or $\boldsymbol{q}_v$ is the average over its k coordinates. Because each $\boldsymbol{p}_u$ or $\boldsymbol{q}_v$ has one learning rate, only $(m+n)$ additional values must be maintained. This storage requirement is much smaller than $(m+n)k$ of PCS. Furthermore, the learning rates,

$$\eta_0 (G_u)^{-\frac{1}{2}} \text{ and } \eta_0 (H_v)^{-\frac{1}{2}},$$

become scalars rather than diagonal matrices. Then the update rule (8) is reduced to that in (5). However, the cost of each iteration is still higher than that of the standard stochastic gradient method because of the need to maintain G_u

and H_v by (10). Note that the $\mathcal{O}(k)$ cost of (10) is comparable to that of (5). Further, because $\boldsymbol{g}_u$ and $\boldsymbol{h}_v$ are used in both (10) and (8), they may need to be stored. In contrast, a single **for** loop for (5) does not require the storage of them. We detailedly discuss the higher cost than (5) by considering two possible implementations.

1. Store $\boldsymbol{g}_u$ and $\boldsymbol{h}_v$.
   - A **for** loop to calculate $\boldsymbol{g}_u, \boldsymbol{h}_v$ and G_u, H_v. Then $\boldsymbol{g}_u$ and $\boldsymbol{h}_v$ vectors are stored.
   - A **for** loop to update $\boldsymbol{p}_u, \boldsymbol{q}_v$ by (8).
2. Calculate $\boldsymbol{g}_u$ and $\boldsymbol{h}_v$ twice.
   - A **for** loop to calculate $\boldsymbol{g}_u, \boldsymbol{h}_v$ and then G_u, H_v.
   - A **for** loop to calculate $\boldsymbol{g}_u, \boldsymbol{h}_v$ and update $\boldsymbol{p}_u, \boldsymbol{q}_v$ by (8).

Clearly, the first approach requires extra storage and memory access. For the second approach, its second loop is the same as (5), but the first loop causes that each SG iteration is twice expensive. To reduce the cost, we decide to use G_u and H_v of the previous iteration. Specifically, at each iteration, we can use a single **for** loop to calculate $\boldsymbol{g}_u$ and $\boldsymbol{h}_v$, update $\boldsymbol{p}_u$ and $\boldsymbol{q}_v$ using past G_u and H_v, and calculate $\boldsymbol{g}_u^T \boldsymbol{g}_u$ and $\boldsymbol{h}_v^T \boldsymbol{h}_v$ to obtain new G_u and H_v for the next iteration. Details are presented in Algorithm 2. In particular, we can see that in the **for** loop, we can finish the above tasks in an element-wise setting. In compared with the implementation for (5), Line 7 in Algorithm 2 is the only extra operation. Thus, the cost of Algorithm 2 is comparable to that of a standard stochastic gradient iteration.

In Figure 1, we check the convergence speed of PCS and RPCS by showing the relationship between RMSE and the number of outer iterations. The convergence speeds of PCS and RPCS are almost identical. Therefore, using the same rate for all elements in $\boldsymbol{p}_u$ (or $\boldsymbol{q}_v$) does not cause more iterations. However, because each iteration becomes cheaper, a comparison on the running time in Figure 2 shows that RPCS is faster than PCS.

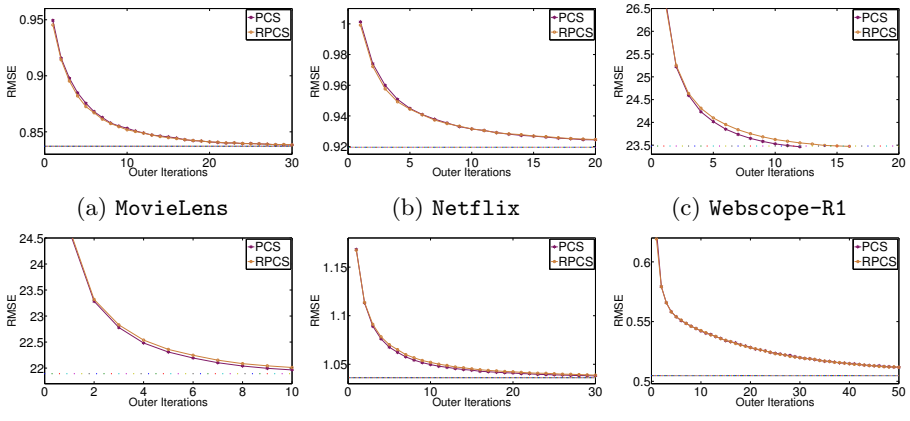
We explain why using the same learning rate for all elements in $\boldsymbol{p}_u$ (or $\boldsymbol{q}_v$) is reasonable for RPCS. Assume $\boldsymbol{p}_u$'s elements are the same,

$$(\boldsymbol{p}_u)_1 = \cdots = (\boldsymbol{p}_u)_k,$$

and so are $(\boldsymbol{q}_v)$'s elements. Then (4) implies that all elements in each of $\boldsymbol{g}_v$ and $\boldsymbol{h}_v$ has the same value. From the calculation of G_u, H_v in (9) and the update rule (8), elements of the new $\boldsymbol{p}_u$ (or $\boldsymbol{q}_v$) are still the same. This result implies that learning rates of all coordinates are the same throughout all iterations. In our implementation of PCS, elements of $\boldsymbol{p}_u$ and $\boldsymbol{q}_v$ are initialized by the same random number generator. Thus, if each element is treated as a random variable, their expected values are the same. Consequently, $\boldsymbol{p}_u$'s (or $\boldsymbol{q}_v$'s) initial elements are identical in statistics and hence our explanation can be applied.

3.2 Twin Learners (TL)

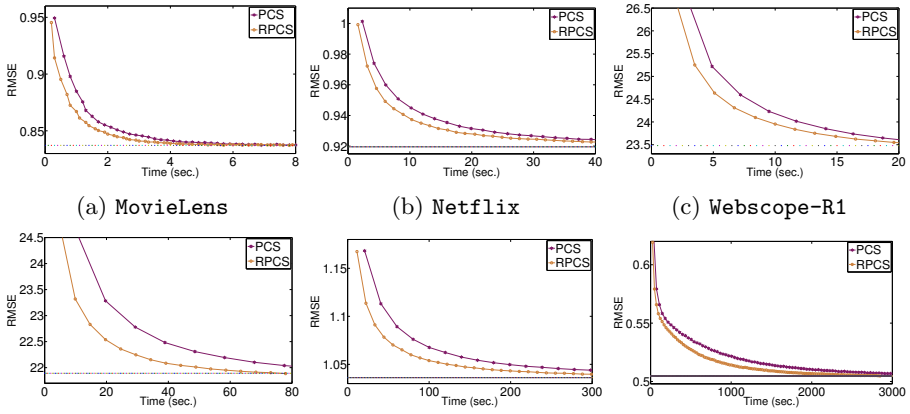
Conceptually, in PCS and RPCS, the decrease of a learning rate should be conservative because it never increases. We observe that the learning rate may be



(a) `MovieLens` (b) `Netflix` (c) `Webscope-R1`

(d) `Yahoo!Music` (e) `Webscope-R2` (f) `Hugewiki`

Fig. 1: A comparison between PCS and RPCS: convergence speed.



(a) `MovieLens` (b) `Netflix` (c) `Webscope-R1`

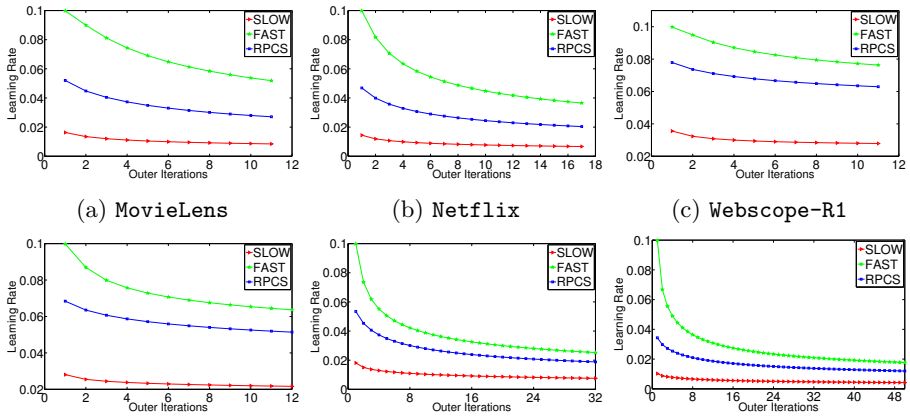
(d) `Yahoo!Music` (e) `Webscope-R2` (f) `Hugewiki`

Fig. 2: A comparison between PCS and RPCS: running time.

too rapidly decreased at the first few updates. The reason may be that the random initialization of P and Q causes comparatively large errors at the beginning. From (4), the gradient is likely to be large if $e_{u,v}$ is large. The large gradient further results in a large sum of squared gradients, and a small learning rate $\eta_0(G_u)^{-\frac{1}{2}}$ or $\eta_0(H_v)^{-\frac{1}{2}}$.

To alleviate this problem, we introduce a strategy called *twin learners* which deliberately allows some elements to have a larger learning rate. To this end, we split the elements of $\boldsymbol{p}_u$ (or $\boldsymbol{q}_v$) to two groups $\{1,\ldots,k_s\}$ and $\{k_s+1,\ldots,k\}$, where the learning rate is smaller for the first group, while larger for the second. The two groups respectively maintain their own factors, G_u^{slow} and G_u^{fast}, via

$$G_u^{\text{slow}} \leftarrow G_u^{\text{slow}} + \frac{(\boldsymbol{g}_u)_{1:k_s}^T(\boldsymbol{g}_u)_{1:k_s}}{k_s}, \quad G_u^{\text{fast}} \leftarrow G_u^{\text{fast}} + \frac{(\boldsymbol{g}_u)_{k_s+1:k}^T(\boldsymbol{g}_u)_{k_s+1:k}}{k-k_s}. \tag{11}$$



(a) `MovieLens` (b) `Netflix` (c) `Webscope-R1`

(d) `Yahoo!Music` (e) `Webscope-R2` (f) `Hugewiki`

Fig. 3: A comparison among the average learning rates of the slow learner (SLOW), the fast learner (FAST), and RPCS. Note that we use $\eta_0 = 0.1$ and initial $G_u = H_v = 1$ following the same settings in our experimental section. Hence the initial learning rate is 0.1.

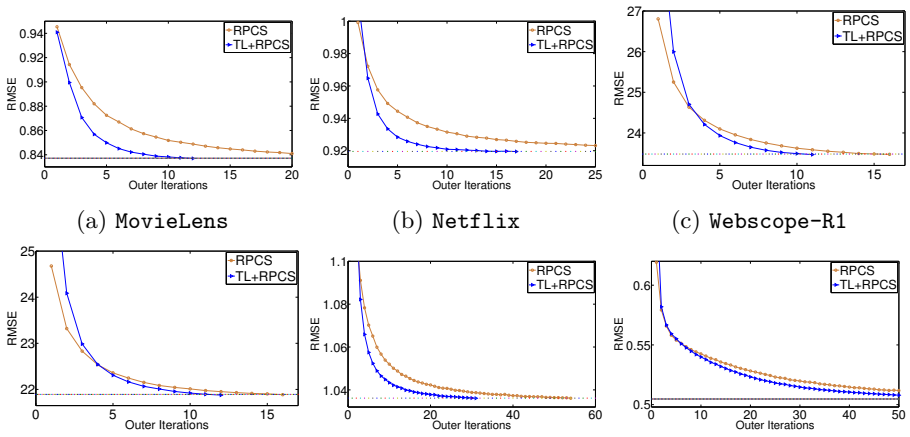
We refer to the first group as the "slow learner," while the second group as the "fast learner." To make G_u^{fast} smaller than G_u^{slow}, we do not apply the second rule in (11) to update G_u^{fast} at the first outer iteration. The purpose is to let the slow learner "absorb" the sharp decline of the learning rate brought by the large initial errors. Then the fast learner can maintain a larger learning rate for faster convergence. We follow the setting in Section 3.1 to use G_u^{slow}, H_v^{slow}, G_u^{fast}, and H_v^{fast} of the previous iteration. Therefore, at each iteration, we have

1. One **for** loop going through the first k_s elements to calculate $(\boldsymbol{g}_u)_{1:k_s}$, $(\boldsymbol{h}_v)_{1:k_s}$, update $(\boldsymbol{p}_u)_{1:k_s}$, $(\boldsymbol{q}_v)_{1:k_s}$, and obtain the next G_u^{slow}, H_v^{slow}.
2. One **for** loop going through the remaining $k - k_s$ elements to calculate $(\boldsymbol{g}_u)_{k_s+1:k}$, $(\boldsymbol{h}_v)_{k_s+1:k}$, update $(\boldsymbol{p}_u)_{k_s+1:k}$, $(\boldsymbol{q}_v)_{k_s+1:k}$, and obtain the next G_u^{fast}, H_v^{fast}.

Figure 3 shows the average learning rates of RPCS (TL is not applied), and slow and fast learners (TL is applied) at each outer iteration. For RPCS, the average learning rate is reduced by around half after the first outer iteration. When TL is applied, though the average learning rate of the slow learner drops even faster, the average learning rate of the fast learner can be kept high to ensure fast learning. A comparison between RPCS with and without TL is in Figure 4. Clearly, TL is very effective. In this paper, we fix k_s as 8% of k. We also tried {2, 4, 8, 16}%, but found that the performance is not sensitive to the choice of k_s.

4 Experiments

We conduct experiments to exhibit the effectiveness of our proposed schedule. Implementation details and experimental settings are respectively shown in Sections 4.1 and 4.2. A comparison among RPCS and existing schedules is in Section 4.3. Then, we compare RPCS with three state-of-the-art packages on both ma-



(a) `MovieLens` (b) `Netflix` (c) `Webscope-R1`

(d) `Yahoo!Music` (e) `Webscope-R2` (f) `Hugewiki`

Fig. 4: A comparison between RPCS with/without TL.

trix factorization and non-negative matrix factorization (NMF) in Sections 4.4 and 4.5, respectively.

4.1 Implementation

For the comparison of various schedules, we implement them by modifying `LIBMF`,[2] which is a parallel SG-based matrix factorization package [21]. We choose it because of its efficiency and the ease of modification. Note that TL is applied to RPCS in all experiments. In `LIBMF`, single-precision floating points are used for data storage, and Streaming SIMD Extensions (SSE) are applied to accelerate the computation.

The inverse square root operation required in (8) is very expensive if it is implemented in a naive way by writing `1/sqrt(·)` in C++. Fortunately, SSE provides an instruction `_mm_rsqrt_ps(·)` to efficiently calculate the approximate inverse square roots for single-precision floating-point numbers.

4.2 Settings

Data Sets Six data sets listed in Table 1 are used. We use the same training/test sets for `MovieLens`, `Netflix`, and `Yahoo!Music` following [21], and the official training/test sets for `Webscope-R1` and `Webscope-R2`.[3] For `Hugewiki`,[4] the original data set is too large for our machine, so we sample first half of the original data. Within this sub-sampled data set, we randomly sample 1% as the test set, and using the remaining for training.

Platform and Parameters We run the experiment on a machine with 12 cores on two Intel Xeon E5-2620 2.0GHz processors and 64 GB memory. We ensure that no other heavy tasks are running on the same computer.

A higher number of latent features often leads to a lower RMSE, but needs a longer training time. From our experience, 100 latent features is an accept-

[2] `http://www.csie.ntu.edu.tw/~cjlin/libmf`
[3] `http://webscope.sandbox.yahoo.com/catalog.php?datatype=r`
[4] `http://graphlab.org/downloads/datasets/`

Data Set	m	n	k	λ	#training	#test	RMSE MF	RMSE NMF
MovieLens	71,567	65,133	100	0.05	9,301,274	698,780	0.831	0.835
Netflix	2,649,429	17,770	100	0.05	99,072,112	1,408,395	0.914	0.916
Webscope-R1	1,948,883	1,101,750	100	1	104,215,016	11,364,422	23.36	23.75
Yahoo!Music	1,000,990	624,961	100	1	252,800,275	4,003,960	21.78	22.10
Webscope-R2	1,823,180	136,737	100	0.05	699,640,226	18,231,790	1.031	1.042
Hugewiki	39,706	25,000,000	100	0.05	1,703,429,136	17,202,478	0.502	0.504

Table 1: Data statistics, parameters used in experiments, and the near-best RMSE's (see Section 4.2 for explanation) on all data sets.

able balance between speed and RMSE, so we use it for all data sets. For the regularization parameter, we select the one that leads to the best test RMSE among {2, 1, 0.5, 0.1, 0.05, 0.01} and present it in Table 1. In addition, P and Q are initialized so that every element is randomly chosen between 0 and 0.1. We normalize the data set by its standard deviation to avoid numerical difficulties. The regularization parameter and the initial values are scaled by the same factor as well. A similar normalization procedure has been used in [18].

The best parameters of each schedule are listed in Table 2. They are the fastest setting to reach 1.005 times the best RMSE obtained by all methods under all parameters. We consider such a "near-best" RMSE to avoid selecting a parameter that needs unnecessarily long running time. Without this mechanism, our comparison on running time can become misleading. Note that PCS and RPCS shares the same η_0. For BDS, we follow [4] to fix $\alpha = 1.05$ and $\beta = 0.5$, and tune only the parameter η_0. The reason is that it is hard to tune three parameters η_0, α, and β together.

Data Set	FS	MDS		BDS	PCS
	η_0	α	β	η_0	η_0
MovieLens	0.005	0.05	0.1	0.05	0.1
Netflix	0.005	0.05	0.1	0.05	0.1
Webscope-R1	0.005	0.05	0.1	0.01	0.1
Yahoo!Music	0.01	0.05	0.05	0.01	0.1
Webscope-R2	0.005	0.05	0.1	0.05	0.1
Hugewiki	0.01	0.05	0.01	0.01	0.1

Table 2: The best parameters for each schedule used.

4.3 Comparison among Schedules

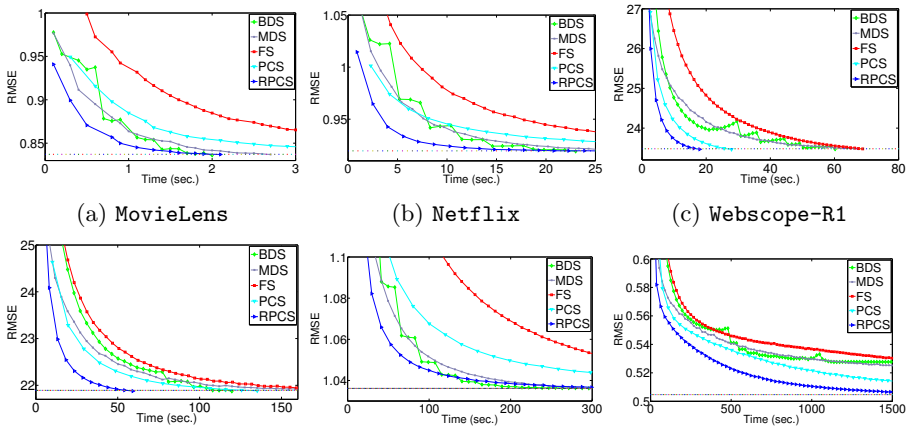
In Figure 5, we present results of comparing five schedules including FS, MDS, BDS, PCS, and RPCS. RPCS outperforms other schedules including the PCS schedule that it is based upon.

4.4 Comparison with State-of-the-art Packages on Matrix Factorization

We compare the proposed schedule (implemented based on LIBMF, and denoted as LIBMF++) with the following packages.

- The standard LIBMF that implements the FS strategy.
- An SG-based package NOMAD [19] that has claimed to outperform LIBMF.
- LIBPMF:[5] it implements a coordinate descent method CCD++ [17].

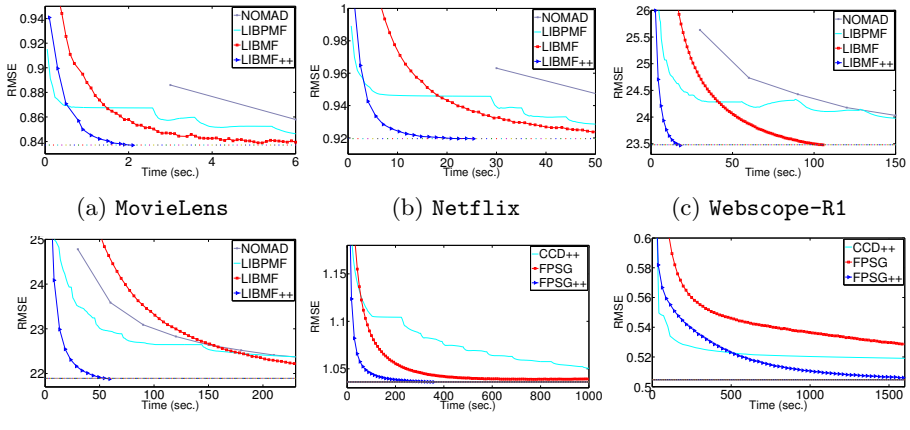
[5] http://www.cs.utexas.edu/~rofuyu/libpmf



(a) `MovieLens` (b) `Netflix` (c) `Webscope-R1`

(d) `Yahoo!Music` (e) `Webscope-R2` (f) `Hugewiki`

Fig. 5: A comparison among different schedules.



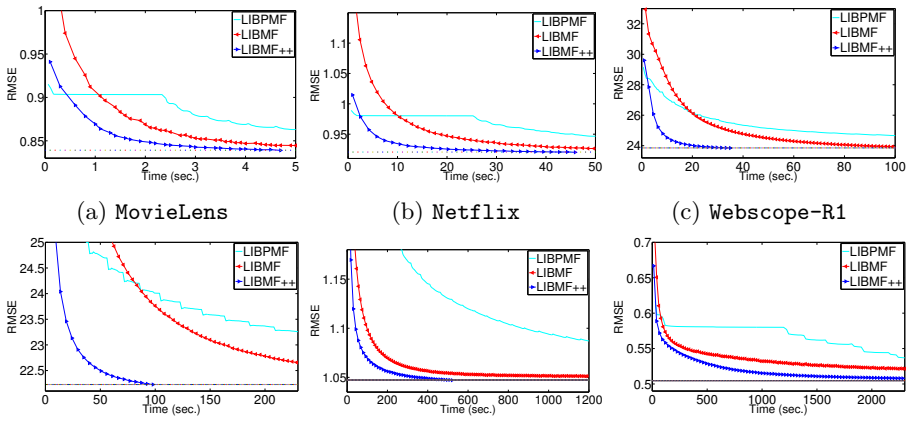
(a) `MovieLens` (b) `Netflix` (c) `Webscope-R1`

(d) `Yahoo!Music` (e) `Webscope-R2` (f) `Hugewiki`

Fig. 6: A comparison among packages for MF.

For all packages, we use single-precision storage[6] and 12 threads. The comparison results are presented in Figure 6. For `NOMAD`, we use the same α and β parameters in [19] for `Netflix` and `Yahoo!Music`, and use parameters identical to MDS for `MovieLens` and `Webscope-R1`. We do not run `NOMAD` on `Webscope-R2` and `Hugewiki` because of the memory limitation. Taking the advantage of the proposed schedule RPCS, `LIBMF++` is significantly faster than `LIBMF` and `LIBPMF`. Our experimental results for `NOMAD` are worse than what [19] reports. In [19], `NOMAD` outperforms `LIBMF` and `CCD++`, but our experiments show an opposite result. We think the reason may be that in [19], 30 cores are used and `NOMAD` may have comparatively better performance if using more cores.

[6] `LIBPMF` is implemented using double precision, but we obtained a single-precision version from its authors.



(a) `MovieLens` (b) `Netflix` (c) `Webscope-R1`

(d) `Yahoo!Music` (e) `Webscope-R2` (f) `Hugewiki`

Fig. 7: A comparison among packages for NMF.

4.5 Comparison with State-of-the-art Methods for Non-negative Matrix Factorization (NMF)

Non-negative matrix factorization [9] requires that all elements in P and Q are non-negative. The optimization problem is

$$\begin{aligned} \min_{P,Q} \quad & \textstyle\sum_{(u,v)\in\Omega}(r_{u,v} - \boldsymbol{p}_u^T\boldsymbol{q}_v)^2 + \lambda(\|\boldsymbol{p}_u\|^2 + \|\boldsymbol{q}_v\|^2) \\ \text{subject to} \quad & P_{du} \geq 0,\ Q_{dv} \geq 0,\ \forall d \in \{1,\ldots,k\},\ u \in \{1,\ldots,m\},\ v \in \{1,\ldots,n\}. \end{aligned}$$

SG can perform NMF by a simple projection [4], and the update rules used are

$$\boldsymbol{p}_u \leftarrow \max\left(\boldsymbol{0}, \boldsymbol{p}_u - \eta\boldsymbol{g}_u\right), \quad \boldsymbol{q}_v \leftarrow \max\left(\boldsymbol{0}, \boldsymbol{q}_v - \eta\boldsymbol{h}_v\right),$$

where the max operator is element-wise. Similarly, the coordinate descent method in `LIBPMF` [5] solves NMF by projecting the negative value back to zero at each update. Therefore, except `NOMAD`, all packages used in the previous experiment can be applied to NMF. We compare them in Figure 7.

A comparison between Figure 6 and Figure 7 shows that all methods converge slower for NMF. This result seems to be reasonable because NMF is a more complicated optimization problem. Interestingly, we see the convergence degradation is more severe for CCD++ (`LIBPMF`) than SG (`LIBMF` and `LIBMF++`). Here we provide a possible explanation. To update $\boldsymbol{p}_u$ once, CCD++ goes through elements in the uth row of the rating matrix k times (for details, see [17]), while SG needs only an arbitrary element in the same row. Therefore, CCD++ performs a more expensive but potentially better update. However, such an update may become less effective because of projecting negative values back to zero. That is, even though the update accurately minimizes the objective value, the result after projection is not as good as in standard matrix factorization.

5 Conclusions

In this paper, we propose a new and effective learning-rate schedule for SG methods applied to matrix factorization. It outperforms existing schedules according

to the rich experiments conducted. By using the proposed method, an extension of the package `LIBMF` is shown to be significantly faster than existing packages on both standard matrix factorization and its non-negative variant. The experiment codes are publicly available at

`http://www.csie.ntu.edu.tw/~cjlin/libmf/exps`

Finally, we plan to extend our schedule to other loss functions such as logistic loss and squared hinge loss.

References

1. Battiti, R.: Accelerated backpropagation learning: Two optimization methods. Complex systems 3(4), 331–342 (1989)
2. Chen, P.L., Tsai, C.T., Chen, Y.N., Chou, K.C., Li, C.L., et al.: A linear ensemble of individual and blended models for music rating prediction. In: ACM SIGKDD KDD-Cup WorkShop (2011)
3. Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. JMLR 12, 2121–2159 (2011)
4. Gemulla, R., Nijkamp, E., Haas, P.J., Sismanis, Y.: Large-scale matrix factorization with distributed stochastic gradient descent. In: KDD. pp. 69–77 (2011)
5. Hsieh, C.J., Dhillon, I.S.: Fast coordinate descent methods with variable selection for non-negative matrix factorization. In: KDD (2011)
6. Kiefer, J., Wolfowitz, J.: Stochastic estimation of the maximum of a regression function. The Annals of Mathematical Statistics 23(3), 462–466 (1952)
7. Koren, Y., Bell, R.: Advances in collaborative filtering. In: Ricci, F., Rokach, L., Shapira, B., Kantor, P.B. (eds.) Recommender Systems Handbook, pp. 145–186. Springer US (2011)
8. Koren, Y., Bell, R.M., Volinsky, C.: Matrix factorization techniques for recommender systems. Computer 42(8), 30–37 (2009)
9. Lee, D.D., Seung, H.S.: Learning the parts of objects by non-negative matrix factorization. Nature 401, 788–791 (1999)
10. Polyak, B.T.: A new method of stochastic approximation type. Avtomat. i Telemekh. 7, 98–107 (1990)
11. Ricci, F., Rokach, L., Shapira, B.: Introduction to recommender systems handbook. In: Ricci, F., Rokach, L., Shapira, B., Kantor, P.B. (eds.) Recommender Systems Handbook, pp. 1–35. Springer (2011)
12. Robbins, H., Monro, S.: A stochastic approximation method. The Annals of Mathematical Statistics 22(3), 400–407 (1951)
13. Schaul, T., Zhang, S., LeCun, Y.: No more pesky learning rates. In: ICML. pp. 343–351 (2013)
14. Takács, G., Pilászy, I., Németh, B., Tikk, D.: Scalable collaborative filtering approaches for large recommender systems. JMLR 10, 623–656 (2009)
15. Vogl, T., Mangis, J., Rigler, A., Zink, W., Alkon, D.: Accelerating the convergence of the back-propagation method. Biological Cybernetics 59(4-5), 257–263 (1988)
16. Xu, W., Liu, X., Gong, Y.: Document clustering based on non-negative matrix factorization. In: SIGIR (2003)
17. Yu, H.F., Hsieh, C.J., Si, S., Dhillon, I.S.: Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In: ICDM (2012)
18. Yu, Z.Q., Shi, X.J., Yan, L., Li, W.J.: Distributed stochastic ADMM for matrix factorization. In: CIKM (2014)

19. Yun, H., Yu, H.F., Hsieh, C.J., Vishwanathan, S., Dhillon, I.S.: Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. In: VLDB (2014)
20. Zeiler, M.D.: ADADELTA: An adaptive learning rate method. CoRR (2012)
21. Zhuang, Y., Chin, W.S., Juan, Y.C., Lin, C.J.: A fast parallel SGD for matrix factorization in shared memory systems. In: RecSys (2013)