Efficient Optimization Methods for Extreme Similarity Learning with Nonlinear Embeddings

Bowen Yuan
National Taiwan University
f03944049@csie.ntu.edu.tw

Yu-Sheng Li
National Taiwan University
r07922087@csie.ntu.edu.tw

Pengrui Quan
University of California, Los Angeles
prquan@g.ucla.edu

Chih-Jen Lin
National Taiwan University
cjlin@csie.ntu.edu.tw

ABSTRACT

We study the problem of learning similarity by using nonlinear embedding models (e.g., neural networks) from all possible pairs. This problem is well-known for its difficulty of training with the extreme number of pairs. For the special case of using linear embeddings, many studies have addressed this issue of handling all pairs by considering certain loss functions and developing efficient optimization algorithms. This paper aims to extend results for general nonlinear embeddings. First, we finish detailed derivations and provide clean formulations for efficiently calculating some building blocks of optimization algorithms such as function, gradient evaluation, and Hessian-vector product. The result enables the use of many optimization methods for extreme similarity learning with nonlinear embeddings. Second, we study some optimization methods in detail. Due to the use of nonlinear embeddings, implementation issues different from linear cases are addressed. In the end, some methods are shown to be highly efficient for extreme similarity learning with nonlinear embeddings.

CCS CONCEPTS

• **Computing methodologies** → **Supervised learning**.

KEYWORDS

Similarity learning, Representation learning, Non-convex optimization, Newton methods, Neural networks

ACM Reference Format:
Bowen Yuan, Yu-Sheng Li, Pengrui Quan, and Chih-Jen Lin. 2021. Efficient Optimization Methods for Extreme Similarity Learning with Nonlinear Embeddings. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '21), August 14–18, 2021, Virtual Event, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3447548.3467363

KDD '21, August 14–18, 2021, Virtual Event, Singapore

ACM ISBN 978-1-4503-8332-5/21/08…$15.00
https://doi.org/10.1145/3447548.3467363

1 INTRODUCTION

Many applications can be cast in the problem of learning similarity between a pair of two entities referred to as the left and the right entities respectively. For example, in recommender systems, the similarity of a user-item pair indicates the preference of the user on the item. In search engines, the similarity of a query-document pair can be used as the relevance between the query and the document. In multi-label classifications, for any instance-label pair with high similarity, the instance can be categorized to the label.

A popular approach for similarity learning is to train an embedding model as the representation of each entity in the pair, such that any pair with high similarity are mapped to two close vectors in the embedding space, and vice versa. For the choice of the embedding model, some recent works [30] report the superiority of nonlinear over conventional linear ones. A typical example of applying nonlinear embedding models is the two-tower structure illustrated in Figure 1, where a multi-layer neural network serves as the embedding model of each entity. Some successful uses in real-world applications have been reported [7, 11, 12, 23, 24].

While many works only consider observed pairs for similarity learning, more and more works argue that better performance can be achieved by considering all possible pairs. For example, recommender systems with implicit feedback face a one-class scenario, where all observed pairs are labeled as similar while the dissimilar ones are missing. To achieve better similarity learning, a widely-used setting [16, 26, 27] is to include all unobserved pairs as dissimilar ones. Another example is counterfactual learning [22, 28], where because the observed pairs carry the selection bias caused by confounders, an effective way to eliminate the bias is to additionally consider the unobserved pairs by imputing their labels.

However, for many real-world scenarios, both the number m of left entities and the number n of right entities can be tremendous. The learning procedure is challenging as a prohibitive $O(mn)$ cost occurs by directly applying any optimization algorithm. To avoid the $O(mn)$ complexity, one can subsample unobserved pairs, but it has been shown that such subsampled settings are inferior to the non-subsampled setting [17, 25]. We refer to the problem of the similarity learning from extremely large mn pairs as *extreme similarity learning*.

To tackle the high $O(mn)$ complexity, the aforementioned works that can handle all pairs consider a loss function with a certain structure for the unobserved pairs. Then they are able to replace the $O(mn)$ complexity with a much smaller $O(m+n)$ one. In particular, the function value, gradient, or other information can be calculated

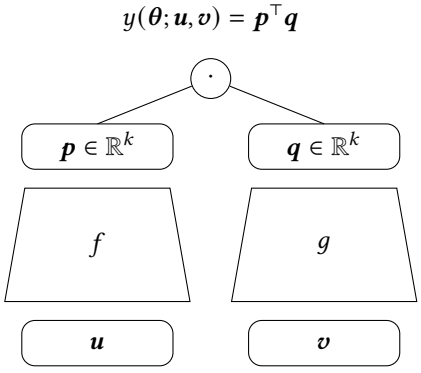


Figure 1: An illustration of the two-tower structure, adopted from [18].

in $O(m+n)$ cost, so various types of optimization methods can be considered. Most exiting works consider linear embeddings (e.g., matrix factorization in [9, 16, 25]), where the optimization problem is often in a multi-block convex form. Thus many consider a block-wise setting to sequentially minimize convex sub-problems. The needed gradient or other information on each block of variables can also be calculated in $O(m+n)$.

If general nonlinear embeddings are considered, so far few works have studied the optimization algorithm. This work aims to fill the gap with the following main contributions.

- To calculate function value, gradient, or other information in $O(m+n)$ cost, the extension from cases in linear embeddings seems to be possible, but detailed derivations have not been available. We finish tedious calculations and provide clean formulations. This result enables the use of many optimization methods for extreme similarity learning with nonlinear embeddings.
- We then study some optimization methods in detail. Due to the use of nonlinear embeddings, some implementation issues different from linear cases are addressed. In the end, some methods are shown to be highly efficient for extreme similarity learning with nonlinear embeddings.

The paper is organized as follows. A review on extreme similarity learning is in Section 2. We derive efficient computation for some important components in Section 3. In Section 4, we demonstrate that many optimization algorithms can be used for extreme similarity learning with nonlinear embeddings. Section 5 describes some implementation issues. Finally, we present experiments on large-scale data sets in Section 6, and conclude in Section 7. Table 1 gives main notations in this paper. The supplementary materials and data/code for experiments are available at https://www.csie.ntu.edu.tw/~cjlin/papers/similarity_learning/.

2 EXTREME SIMILARITY LEARNING

We review extreme similarity learning problems.

2.1 Problem Formulation

Many real-world applications can be modeled as a learning problem with an incomplete similarity matrix $\boldsymbol{R}$ of m left entities and n right entities. The set of observed pairs is denoted as $\mathbb{O} \subseteq \{1,\dots,m\} \times \{1,\dots,n\}$ where typically $|\mathbb{O}| \ll mn$. Only $R_{ij}, \forall (i,j) \in \mathbb{O}$ are revealed. Besides $\boldsymbol{R}$, we assume that side features $\mathbb{U} = \{\dots, \boldsymbol{u}_i, \dots\}$ of left entities and $\mathbb{V} = \{\dots, \boldsymbol{v}_j, \dots\}$ of right entities are available. Our goal is to find a similarity function $y : \mathbb{R}^D \to \mathbb{R}$,

$$\hat{Y}_{ij} = y(\boldsymbol{\theta}; \boldsymbol{u}_i, \boldsymbol{v}_j), \tag{1}$$

where $\boldsymbol{\theta} = \begin{bmatrix} \boldsymbol{\theta}^u \\ \boldsymbol{\theta}^v \end{bmatrix}$ is the vectorized variables of y with $\boldsymbol{\theta}^u \in \mathbb{R}^{D_u}$, $\boldsymbol{\theta}^v \in \mathbb{R}^{D_v}$, and $D = D_u + D_v$. For the similarity function, we consider the following dot product similarity that has been justified in a recent comparative study of similarity functions in recommender systems [18]

$$y(\boldsymbol{\theta}; \boldsymbol{u}_i, \boldsymbol{v}_j) = f(\boldsymbol{\theta}^u; \boldsymbol{u}_i)^\top g(\boldsymbol{\theta}^v; \boldsymbol{v}_j),$$

where $f : \mathbb{R}^{D_u} \to \mathbb{R}^k$ and $g : \mathbb{R}^{D_v} \to \mathbb{R}^k$ are two embedding models that learn representations of left and right entities, respectively; see Figure 1. In this work, we focus on nonlinear embedding models. For convenience, we abbreviate $f(\boldsymbol{\theta}^u; \boldsymbol{u}_i)$ and $g(\boldsymbol{\theta}^v; \boldsymbol{v}_j)$ to $f^i(\boldsymbol{\theta}^u)$ and $g^j(\boldsymbol{\theta}^v)$, respectively. We also write the embedding vectors as

$$\boldsymbol{p}_i = f^i(\boldsymbol{\theta}^u), \boldsymbol{q}_j = g^j(\boldsymbol{\theta}^v) \tag{2}$$

so that

$$\hat{Y}_{ij} = \boldsymbol{p}_i^\top \boldsymbol{q}_j = f^i(\boldsymbol{\theta}^u)^\top g^j(\boldsymbol{\theta}^v). \tag{3}$$

By considering all mn pairs, we learn $\boldsymbol{\theta}$ through solving

$$\min_{\boldsymbol{\theta}} \quad L(\boldsymbol{\theta}), \tag{4}$$

where the objective function is

$$L(\boldsymbol{\theta}) = \sum_{i=1}^m \sum_{j=1}^n \ell_{ij}(Y_{ij}, \hat{Y}_{ij}) + \lambda \Psi(\boldsymbol{\theta}), \tag{5}$$

ℓ_{ij} is an entry-wise twice-differentiable loss function convex in $\hat{Y}_{ij}$, $\Psi(\boldsymbol{\theta})$ is a twice-differentiable and strongly convex regularizer to avoid overfitting, and $\lambda > 0$ is the corresponding parameter. We assume Ψ is simple so that in all complexity analyses we ignore the costs related to $\Psi(\boldsymbol{\theta})$, $\nabla\Psi(\boldsymbol{\theta})$, and $\nabla^2\Psi(\boldsymbol{\theta})$. In (5), we further assume

$$Y_{ij} = \begin{cases} R_{ij} & (i,j) \in \mathbb{O}, \\ \tilde{Y}_{ij} & (i,j) \notin \mathbb{O}, \end{cases} \tag{6}$$

where for any unobserved pair $(i,j) \notin \mathbb{O}$ we impute $\tilde{Y}_{ij}$ as an artificial label of the similarity. For instance, in recommender systems with implicit feedback, by treating unobserved pairs as negative, usually $\tilde{Y}_{ij} = 0$ or -1 is considered.

Table 1: Main notation

Notation	Description
(i,j)	(left entity i, right entity j) pair
m, n	numbers of left entities and right entities
$\boldsymbol{u}, \boldsymbol{v}$	feature vectors of a left entity and a right entity
$\mathbb{O}$	set of observed pairs
$y(\cdot), \boldsymbol{\theta} \in \mathbb{R}^{D_u+D_v}$	a similarity function and its variables
$L(\boldsymbol{\theta})$	objective function
$\boldsymbol{R}$	observed incomplete similarity matrix
$\hat{\boldsymbol{Y}}$	similarity matrix predicted by $y(\cdot)$
$\tilde{\boldsymbol{Y}}$	imputed similarity matrix (for unobserved pairs)
$f(\cdot), g(\cdot)$	embedding models of left and right entities
$\boldsymbol{p} \in \mathbb{R}^k, \boldsymbol{q} \in \mathbb{R}^k$	embedding vectors of left and right entities
$F(f), F(g)$	cost of operations related to $f(\cdot)$, and $g(\cdot)$

Clearly (5) involves a summation of mn terms, so the cost of directly applying an optimization algorithm is prohibitively proportional to $O(mn)$.

2.2 Linear Embedding Models

As mentioned in Section 1, most existing works of extreme similarity learning focus on linear embedding models. To avoid any $O(mn)$ cost occurs in solving the optimization problem, they consider a special kind of loss functions

$$\ell_{ij}(Y_{ij}, \hat{Y}_{ij}) = \begin{cases} \ell(Y_{ij}, \hat{Y}_{ij}) & (i,j) \in \mathbb{O}, \\ \frac{1}{2}\omega a_i b_j (\tilde{Y}_{ij} - \hat{Y}_{ij})^2 & (i,j) \notin \mathbb{O}, \end{cases} \tag{7}$$

where $\ell(a, b)$ is any non-negative and twice-differentiable function convex in b, and ω is a parameter for balancing the two kinds of losses. Two vectors $\boldsymbol{a} \in \mathbb{R}^m$, and $\boldsymbol{b} \in \mathbb{R}^n$ are chosen so that $a_i b_j$ is a cost associated with an unobserved pair. Besides, for the imputed label $\tilde{Y}_{ij}$, it is required [13, 28] that

$$\tilde{Y}_{ij} = \tilde{\boldsymbol{p}}_i^\top \tilde{\boldsymbol{q}}_j, \tag{8}$$

where $\tilde{\boldsymbol{p}}_i, \tilde{\boldsymbol{q}}_j$ are vectors output by the prior imputation model, and are fixed throughout training. For easy analysis, we assume that $\tilde{\boldsymbol{p}}_i, \tilde{\boldsymbol{q}}_j \in \mathbb{R}^k$ have the same length as $\boldsymbol{p}_i, \boldsymbol{q}_j$. With the loss in (7), past works were able to reduce the $O(mn)$ cost to $O(m+n)$. The main reason is because derived computations on all i and all j can be obtained by using values solely related to i and j, respectively. A conceptual illustration is in the following equation.

$$\sum\nolimits_{i=1}^{m} \sum\nolimits_{j=1}^{n} (\cdots) = \sum\nolimits_{i=1}^{m} (\cdots) \quad \times \quad \sum\nolimits_{j=1}^{n} (\cdots). \tag{9}$$

The mainline of most past works is to incorporate more linear models into extreme similarity learning, which can range from simple matrix factorization [9, 16, 25] to complex models, e.g., matrix factorization with side information and (field-aware) factorization machines [3, 26, 29].

3 INFORMATION NEEDED IN OPTIMIZATION ALGORITHMS

Most optimization methods involve a procedure that repeatedly calculates the function value, the gradient, and/or other information. For example, a gradient descent method must calculate the negative gradient as the direction for an update. Crucially, from past developments for linear embeddings, the $O(mn)$ cost mainly occurs in these building blocks of optimization algorithms. The aim of this section is to derive $O(m+n)$ computation for these important components when nonlinear embeddings are applied.

3.1 From Linear to Nonlinear Embedding Models

We begin by discussing the differences between using linear and nonlinear embeddings. If linear embeddings are used, a key feature is that (4) becomes a multi-block convex problem. That is, (4) is reduced to a convex sub-problem when embedding variables of one side are fixed. Then existing works apply alternating least squares [16], (block) coordinate descent [3, 9, 25, 29], and alternating Newton method [26, 28] to sequentially solve each sub-problem by convex optimization techniques, which often iteratively calculate the following information.

- Function value,
- Gradient, and/or
- Hessian-vector product

Thus a focus was on deriving $O(m+n)$ operations for them.

If nonlinear embeddings are considered, in general (4) is not a block-convex problem. Thus a block-wise setting may not possess advantages so that optimization algorithms updating all variables together is a more natural choice. In any case, if we can obtain formulations for calculating the function value or the gradient over all variables, then the information over a block of variables is often in a simplified form. For example, the gradient over the block $\boldsymbol{\theta}^u$ is a sub-vector of the whole gradient over $\boldsymbol{\theta} = \begin{bmatrix} \boldsymbol{\theta}^u \\ \boldsymbol{\theta}^v \end{bmatrix}$. Therefore, in the rest of this section, we check the general scenario of considering all variables.

3.2 Evaluation of the Objective Value

The infeasible $O(mn)$ cost occurs if we directly compute $L(\boldsymbol{\theta})$ in (5). To handle this, our idea is to follow past works of linear embeddings to consider (7) as the loss function.

By applying (7), (5) is equivalent to

$$\begin{aligned} L(\boldsymbol{\theta}) &= \sum_{(i,j)\in\mathbb{O}} \ell(Y_{ij}, \hat{Y}_{ij}) + \sum_{(i,j)\notin\mathbb{O}} \frac{1}{2}\omega a_i b_j (\tilde{Y}_{ij} - \hat{Y}_{ij})^2 + \lambda\Psi(\boldsymbol{\theta}) \\ &= \underbrace{\sum_{(i,j)\in\mathbb{O}} \ell^+_{ij}(Y_{ij}, \hat{Y}_{ij})}_{L^+(\boldsymbol{\theta})} + \omega \underbrace{\sum_{i=1}^{m}\sum_{j=1}^{n} \ell^-_{ij}(Y_{ij}, \hat{Y}_{ij})}_{L^-(\boldsymbol{\theta})} + \lambda\Psi(\boldsymbol{\theta}), \end{aligned} \tag{10}$$

where

$$\begin{aligned} \ell^+_{ij}(Y_{ij}, \hat{Y}_{ij}) &= \ell(Y_{ij}, \hat{Y}_{ij}) - \frac{1}{2}\omega a_i b_j (\tilde{Y}_{ij} - \hat{Y}_{ij})^2, \\ \ell^-_{ij}(Y_{ij}, \hat{Y}_{ij}) &= \frac{1}{2} a_i b_j (\tilde{Y}_{ij} - \hat{Y}_{ij})^2. \end{aligned} \tag{11}$$

The first term $L^+(\boldsymbol{\theta})$ involves the summation of $|\mathbb{O}|$ values, so with $|\mathbb{O}| \ll mn$ the bottleneck is on the second term $L^-(\boldsymbol{\theta})$.

Let $F(f)$ and $F(g)$ respectively represent the cost of operations related to the two nonlinear embedding models,[1] and

$$\boldsymbol{P} = \begin{bmatrix} \boldsymbol{p}_1^\top \\ \vdots \\ \boldsymbol{p}_m^\top \end{bmatrix}, \boldsymbol{Q} = \begin{bmatrix} \boldsymbol{q}_1^\top \\ \vdots \\ \boldsymbol{q}_n^\top \end{bmatrix}, \tilde{\boldsymbol{P}} = \begin{bmatrix} \tilde{\boldsymbol{p}}_1^\top \\ \vdots \\ \tilde{\boldsymbol{p}}_m^\top \end{bmatrix}, \tilde{\boldsymbol{Q}} = \begin{bmatrix} \tilde{\boldsymbol{q}}_1^\top \\ \vdots \\ \tilde{\boldsymbol{q}}_n^\top \end{bmatrix}, \tag{12}$$

where $\tilde{\boldsymbol{P}}$ and $\tilde{\boldsymbol{Q}}$ are constant matrices, but $\boldsymbol{P}$ and $\boldsymbol{Q}$ depend on $\boldsymbol{\theta}$. To compute $L^+(\boldsymbol{\theta})$ in (10), we first compute $\boldsymbol{P}$ and $\boldsymbol{Q}$ in $O(mF(f) + nF(g))$ time and store them in $O((m+n)k)$ space. Then for each $(i,j) \in \mathbb{O}$, from (3) and (8), we compute $\hat{Y}_{ij} = \boldsymbol{p}_i^\top \boldsymbol{q}_j$ and $\tilde{Y}_{ij} = \tilde{\boldsymbol{p}}_i^\top \tilde{\boldsymbol{q}}_j$ in $O(k)$ time. As a result, we can compute the entire $L^+(\boldsymbol{\theta})$ in $O(mF(f) + nF(g) + |\mathbb{O}|k)$ time.

For $L^-(\boldsymbol{\theta})$, we follow the idea in (9) to derive

$$L^-(\boldsymbol{\theta}) = \frac{1}{2}\langle \tilde{\boldsymbol{P}}_c, \tilde{\boldsymbol{Q}}_c \rangle_{\mathrm{F}} - \langle \hat{\boldsymbol{P}}_c, \hat{\boldsymbol{Q}}_c \rangle_{\mathrm{F}} + \frac{1}{2}\langle \boldsymbol{P}_c, \boldsymbol{Q}_c \rangle_{\mathrm{F}}, \tag{13}$$

where details are left in Appendix A.1. Later for gradient calculation we show details as a demonstration of using (9).

[1] If neural networks are considered, operations such as forward or backward computation are considered. See more discussion in Section 5

In (13), $\langle \cdot, \cdot \rangle_{\mathrm{F}}$ is a Frobenius inner product between two matrices,

$$\begin{aligned} &\boldsymbol{P}_c = \boldsymbol{P}^\top \boldsymbol{A}\boldsymbol{P}, \hat{\boldsymbol{P}}_c = \tilde{\boldsymbol{P}}^\top \boldsymbol{A}\boldsymbol{P}, \tilde{\boldsymbol{P}}_c = \tilde{\boldsymbol{P}}^\top \boldsymbol{A}\tilde{\boldsymbol{P}}, \\ &\boldsymbol{Q}_c = \boldsymbol{Q}^\top \boldsymbol{B}\boldsymbol{Q}, \hat{\boldsymbol{Q}}_c = \tilde{\boldsymbol{Q}}^\top \boldsymbol{B}\boldsymbol{Q}, \text{ and } \tilde{\boldsymbol{Q}}_c = \tilde{\boldsymbol{Q}}^\top \boldsymbol{B}\tilde{\boldsymbol{Q}}, \end{aligned} \tag{14}$$

where $\boldsymbol{A} = \mathrm{diag}(\boldsymbol{a}), \boldsymbol{B} = \mathrm{diag}(\boldsymbol{b})$ are two diagonal matrices. As $\boldsymbol{P} \in \mathbb{R}^{m\times k}$ and $\boldsymbol{Q} \in \mathbb{R}^{n\times k}$ have been pre-computed and cached in memory during computing $L^+(\boldsymbol{\theta})$, the matrices in (14) can be computed in $O((m+n)k^2)$ time and cached in $O(k^2)$ space. Then the Frobenius inner products between these matrices cost only $O(k^2)$ time.

From complexities of $L^+(\boldsymbol{\theta})$ and $L^-(\boldsymbol{\theta})$, the overall cost of evaluating $L(\boldsymbol{\theta})$ is

$$O(|\mathbb{O}|k + (m+n)k^2 + mF(f) + nF(g)). \tag{15}$$

3.3 Computation of Gradient Information

For $L(\boldsymbol{\theta})$ defined in (5), the gradient is

$$\nabla L(\boldsymbol{\theta}) = \sum_{i=1}^{m}\sum_{j=1}^{n} \frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}}^\top \frac{\partial \ell_{ij}}{\partial \hat{Y}_{ij}} + \lambda \nabla \Psi(\boldsymbol{\theta}). \tag{16}$$

Analogous to (5), it is impractical to directly compute $\nabla L(\boldsymbol{\theta})$ with infeasible $O(mn)$ costs. For derivatives with respect to a vector, we let $\partial \hat{Y}_{ij}/\partial \boldsymbol{\theta}$ be a row vector, and later for operations such as $\partial f^i/\partial \boldsymbol{\theta}^u$, we let it be a $k \times D_u$ matrix. In all other situations, a vector such as $\nabla L(\boldsymbol{\theta})$ is a column vector.

From (7) and (10), $\nabla L(\boldsymbol{\theta})$ in (16) is equivalent to

$$\nabla L(\boldsymbol{\theta}) = \underbrace{\sum_{(i,j)\in\mathbb{O}} \frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}}^\top \frac{\partial \ell^+_{ij}}{\partial \hat{Y}_{ij}}}_{\nabla L^+(\boldsymbol{\theta})} + \omega \underbrace{\sum_{i=1}^{m}\sum_{j=1}^{n} \frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}}^\top \frac{\partial \ell^-_{ij}}{\partial \hat{Y}_{ij}}}_{\nabla L^-(\boldsymbol{\theta})} + \lambda \nabla \Psi(\boldsymbol{\theta}).$$

To calculate $\nabla L^+(\boldsymbol{\theta})$, let $\boldsymbol{X} \in \mathbb{R}^{m\times n}$ be a sparse matrix with

$$X_{ij} = \begin{cases} \frac{\partial \ell^+_{ij}}{\partial \hat{Y}_{ij}} & (i,j) \in \mathbb{O}, \\ 0 & \text{otherwise.} \end{cases} \tag{17}$$

From (2) and (3),

$$\frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}} = \left[\frac{\partial \hat{Y}_{ij}}{\partial f^i}\frac{\partial f^i}{\partial \boldsymbol{\theta}^u}, \frac{\partial \hat{Y}_{ij}}{\partial g^j}\frac{\partial g^j}{\partial \boldsymbol{\theta}^v}\right] = \left[\boldsymbol{q}_j^\top \frac{\partial f^i}{\partial \boldsymbol{\theta}^u}, \boldsymbol{p}_i^\top \frac{\partial g^j}{\partial \boldsymbol{\theta}^v}\right], \tag{18}$$

so with (17) we have

$$\nabla L^+(\boldsymbol{\theta}) = \sum_{i=1}^{m}\sum_{j=1}^{n} \left(\frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}}^\top X_{ij}\right) = \begin{bmatrix} \sum_{i=1}^{m}\sum_{j=1}^{n} \frac{\partial f^i}{\partial \boldsymbol{\theta}^u}^\top \boldsymbol{q}_j X_{ij} \\ \sum_{i=1}^{m}\sum_{j=1}^{n} \frac{\partial g^j}{\partial \boldsymbol{\theta}^v}^\top \boldsymbol{p}_i X_{ij} \end{bmatrix}. \tag{19}$$

Let $\mathbf{J}^u \in \mathbb{R}^{m\times D_u \times k}, \mathbf{J}^v \in \mathbb{R}^{n\times D_v \times k}$ be tensors with

$$\text{the } i\text{-th slice } \mathbf{J}^u_{i,:,:} = \frac{\partial f^i}{\partial \boldsymbol{\theta}^u}^\top \text{ and the } j\text{-th slice } \mathbf{J}^v_{j,:,:} = \frac{\partial g^j}{\partial \boldsymbol{\theta}^v}^\top, \tag{20}$$

respectively. The top part of (19) can be computed by

$$\begin{aligned} \sum_{i=1}^{m}\sum_{j=1}^{n} \frac{\partial f^i}{\partial \boldsymbol{\theta}^u}^\top \boldsymbol{q}_j X_{ij} &= \sum_{i=1}^{m} \frac{\partial f^i}{\partial \boldsymbol{\theta}^u}^\top \left(\sum_{j=1}^{n} X_{ij}\boldsymbol{q}_j^\top\right)^\top \\ &= \sum_{i=1}^{m} \frac{\partial f^i}{\partial \boldsymbol{\theta}^u}^\top \left(\boldsymbol{X}_{i,:}\boldsymbol{Q}\right)^\top = \langle \mathbf{J}^u, \boldsymbol{X}\boldsymbol{Q}\rangle, \end{aligned} \tag{21}$$

where (21) is from (12) and the following definition of the tensor-matrix inner product

$$\langle \mathbf{J}, \boldsymbol{M}\rangle = \sum_{i=1}^{m} \mathbf{J}_{i,:,:}(\boldsymbol{M}_{i,:})^\top. \tag{22}$$

As the computation of the bottom part of (19) is similar, we omit the details. Then (19) can be written as

$$\nabla L^+(\boldsymbol{\theta}) = \begin{bmatrix} \langle \mathbf{J}^u, \boldsymbol{X}\boldsymbol{Q}\rangle \\ \langle \mathbf{J}^v, \boldsymbol{X}^\top \boldsymbol{P}\rangle \end{bmatrix}, \tag{23}$$

where $\boldsymbol{P}$ and $\boldsymbol{Q}$ are first computed in $O(mF(f) + nF(g))$ time and stored in $O((m+n)k)$ space. Then we compute $\boldsymbol{X}\boldsymbol{Q}$ and $\boldsymbol{X}^\top\boldsymbol{P}$ in $O(|\mathbb{O}|k)$ time and $O((m+n)k)$ space.

To calculate $\nabla L^-(\boldsymbol{\theta})$, from (3) and (11) we have

$$\frac{\partial \ell^-_{ij}}{\partial \hat{Y}_{ij}} = a_i b_j (\boldsymbol{p}_i^\top \boldsymbol{q}_j - \tilde{\boldsymbol{p}}_i^\top \tilde{\boldsymbol{q}}_j).$$

This and (18) imply that

$$\nabla L^-(\boldsymbol{\theta}) = \sum_{i=1}^{m}\sum_{j=1}^{n} \begin{bmatrix} \frac{\partial f^i}{\partial \boldsymbol{\theta}^u}^\top \boldsymbol{q}_j \\ \frac{\partial g^j}{\partial \boldsymbol{\theta}^v}^\top \boldsymbol{p}_i \end{bmatrix} a_i b_j (\boldsymbol{p}_i^\top \boldsymbol{q}_j - \tilde{\boldsymbol{p}}_i^\top \tilde{\boldsymbol{q}}_j) \tag{24}$$

$$\begin{aligned} &= \begin{bmatrix} \sum_{i=1}^{m} \frac{\partial f^i}{\partial \boldsymbol{\theta}^u}^\top \sum_{j=1}^{n} \boldsymbol{q}_j a_i b_j (\boldsymbol{q}_j^\top \boldsymbol{p}_i - \tilde{\boldsymbol{q}}_j^\top \tilde{\boldsymbol{p}}_i) \\ \sum_{j=1}^{n} \frac{\partial g^j}{\partial \boldsymbol{\theta}^v}^\top \sum_{i=1}^{m} \boldsymbol{p}_i a_i b_j (\boldsymbol{p}_i^\top \boldsymbol{q}_j - \tilde{\boldsymbol{p}}_i^\top \tilde{\boldsymbol{q}}_j) \end{bmatrix} \\ &= \begin{bmatrix} \sum_{i=1}^{m} \frac{\partial f^i}{\partial \boldsymbol{\theta}^u}^\top \left((\sum_{j=1}^{n} b_j \boldsymbol{q}_j \boldsymbol{q}_j^\top) a_i \boldsymbol{p}_i - (\sum_{j=1}^{n} b_j \boldsymbol{q}_j \tilde{\boldsymbol{q}}_j^\top) a_i \tilde{\boldsymbol{p}}_i\right) \\ \sum_{j=1}^{n} \frac{\partial g^j}{\partial \boldsymbol{\theta}^v}^\top \left((\sum_{i=1}^{m} a_i \boldsymbol{p}_i \boldsymbol{p}_i^\top) b_j \boldsymbol{q}_j - (\sum_{i=1}^{m} a_i \boldsymbol{p}_i \tilde{\boldsymbol{p}}_i^\top) b_j \tilde{\boldsymbol{q}}_j\right) \end{bmatrix}, \end{aligned}$$

where the last two equalities follow from the idea in (9). From (14),

$$\begin{aligned} &\boldsymbol{P}_c = \textstyle\sum_{i=1}^{m} a_i \boldsymbol{p}_i \boldsymbol{p}_i^\top, \boldsymbol{Q}_c = \sum_{j=1}^{n} b_j \boldsymbol{q}_j \boldsymbol{q}_j^\top, \\ &\hat{\boldsymbol{P}}_c = \textstyle\sum_{i=1}^{m} a_i \tilde{\boldsymbol{p}}_i \boldsymbol{p}_i^\top, \text{ and } \hat{\boldsymbol{Q}}_c = \sum_{j=1}^{n} b_j \tilde{\boldsymbol{q}}_j \boldsymbol{q}_j^\top, \end{aligned} \tag{25}$$

so we can compute $\nabla L^-(\boldsymbol{\theta})$ by

$$\begin{aligned} \nabla L^-(\boldsymbol{\theta}) &= \begin{bmatrix} \sum_{i=1}^{m} \frac{\partial f^i}{\partial \boldsymbol{\theta}^u}^\top (\boldsymbol{Q}_c a_i \boldsymbol{p}_i - \hat{\boldsymbol{Q}}_c^\top a_i \tilde{\boldsymbol{p}}_i) \\ \sum_{j=1}^{n} \frac{\partial g^j}{\partial \boldsymbol{\theta}^v}^\top (\boldsymbol{P}_c b_j \boldsymbol{q}_j - \hat{\boldsymbol{P}}_c^\top b_j \tilde{\boldsymbol{q}}_j) \end{bmatrix} \\ &= \begin{bmatrix} \langle \mathbf{J}^u, \boldsymbol{A}\boldsymbol{P}\boldsymbol{Q}_c - \boldsymbol{A}\tilde{\boldsymbol{P}}\hat{\boldsymbol{Q}}_c\rangle \\ \langle \mathbf{J}^v, \boldsymbol{B}\boldsymbol{Q}\boldsymbol{P}_c - \boldsymbol{B}\tilde{\boldsymbol{Q}}\hat{\boldsymbol{P}}_c\rangle \end{bmatrix}, \end{aligned} \tag{26}$$

where (26) follows from (12), (20), and (22). As $\boldsymbol{P}$ and $\boldsymbol{Q}$ have been pre-computed and cached in memory during computing $\nabla L^+(\boldsymbol{\theta})$, all matrices in (14) can be computed in $O((m+n)k^2)$ time and cached in $O(k^2)$ space. Then the right hand sides of $\langle \cdot, \cdot \rangle$ in (26) can be computed in $O((m+n)k^2)$ time as $\boldsymbol{A}$ and $\boldsymbol{B}$ are diagonal.

By combining (23) and (26), the entire $\nabla L(\boldsymbol{\theta})$ can be efficiently computed by

$$\nabla L(\boldsymbol{\theta}) = \begin{bmatrix} \langle \mathbf{J}^u, \boldsymbol{X}\boldsymbol{Q} + \omega(\boldsymbol{A}\boldsymbol{P}\boldsymbol{Q}_c - \boldsymbol{A}\tilde{\boldsymbol{P}}\hat{\boldsymbol{Q}}_c)\rangle \\ \langle \mathbf{J}^v, \boldsymbol{X}^\top\boldsymbol{P} + \omega(\boldsymbol{B}\boldsymbol{Q}\boldsymbol{P}_c - \boldsymbol{B}\tilde{\boldsymbol{Q}}\hat{\boldsymbol{P}}_c)\rangle \end{bmatrix} + \lambda \nabla \Psi(\boldsymbol{\theta}), \tag{27}$$

which has a time complexity of

$$O(|\mathbb{O}|k + (m+n)k^2 + mF(f) + nF(g)). \tag{28}$$

The $mF(f) + nF(g)$ terms come from calculating $\boldsymbol{P}$ and $\boldsymbol{Q}$ in (23) and from $\partial f^i/\partial \boldsymbol{\theta}^u, \forall i$, $\partial g^j/\partial \boldsymbol{\theta}^v, \forall j$ in (20). The procedure of evaluating $\nabla L(\boldsymbol{\theta})$ is summarized in Algorithm 1.

From (15) and (28), the cost of one function evaluation is similar to that of one gradient evaluation.

Algorithm 1: Gradient evaluation: $\nabla L(\boldsymbol{\theta})$

Input: $\boldsymbol{\theta}, \mathbb{O}, \tilde{\boldsymbol{P}}, \tilde{\boldsymbol{Q}}, \boldsymbol{A}, \boldsymbol{B}$.

Calculate $\boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{P}_c, \boldsymbol{Q}_c, \hat{\boldsymbol{P}}_c, \hat{\boldsymbol{Q}}_c$.[2]

$X_{ij} \leftarrow \frac{\partial \ell_{ij}^+}{\partial \hat{Y}_{ij}}, \forall (i,j) \in \mathbb{O}$

$\boldsymbol{X}_q \leftarrow \boldsymbol{X}\boldsymbol{Q}, \boldsymbol{X}_p \leftarrow \boldsymbol{X}^\top \boldsymbol{P}$

$\nabla L(\boldsymbol{\theta}) \leftarrow \begin{bmatrix} \langle \mathbf{J}^u, \boldsymbol{X}_q + \omega(\boldsymbol{A}\boldsymbol{P}\boldsymbol{Q}_c - \boldsymbol{A}\tilde{\boldsymbol{P}}\hat{\boldsymbol{Q}}_c)\rangle \\ \langle \mathbf{J}^v, \boldsymbol{X}_p + \omega(\boldsymbol{B}\boldsymbol{Q}\boldsymbol{P}_c - \boldsymbol{B}\tilde{\boldsymbol{Q}}\hat{\boldsymbol{P}}_c)\rangle \end{bmatrix} + \lambda \nabla \Psi(\boldsymbol{\theta})$.

Output: $\nabla L(\boldsymbol{\theta})$

3.4 Computation of Gauss-Newton Matrix-vector Products

For optimization methods using second-order information, commonly the product between Hessian and some vector $\boldsymbol{d}$ is needed:

$$\nabla^2 L(\boldsymbol{\theta})\boldsymbol{d}. \tag{29}$$

The Hessian of (5) is

$$\nabla^2 L(\boldsymbol{\theta}) = \sum_{i=1}^{m}\sum_{j=1}^{n} \frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}}^\top \frac{\partial^2 \ell_{ij}}{\partial \hat{Y}_{ij}^2} \frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}} + \sum_{i=1}^{m}\sum_{j=1}^{n} \frac{\partial^2 \hat{Y}_{ij}}{\partial \boldsymbol{\theta}^2} \frac{\partial \ell_{ij}}{\partial \hat{Y}_{ij}} + \lambda \nabla^2 \Psi(\boldsymbol{\theta}). \tag{30}$$

Clearly, a direct calculation costs $O(mn)$. However, before reducing the $O(mn)$ cost to $O(m+n)$, we must address an issue that $\nabla^2 L(\boldsymbol{\theta})$ is not positive definite. In past studies of linear embeddings (e.g., [4, 26, 29]) the block-convex $L(\boldsymbol{\theta})$ implies that for a strictly convex sub-problem, the Hessian is positive definite. Without such properties, here we need a positive definite approximation of $\nabla^2 L(\boldsymbol{\theta})$. Following [15, 20], we remove the second term in (30) to have the following Gauss-Newton matrix [19]

$$\boldsymbol{G} = \sum_{i=1}^{m}\sum_{j=1}^{n} \frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}}^\top \frac{\partial^2 \ell_{ij}}{\partial \hat{Y}_{ij}^2} \frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}} + \lambda \nabla^2 \Psi(\boldsymbol{\theta}), \tag{31}$$

which is positive definite from the convexity of ℓ_{ij} in $\hat{Y}_{ij}$ and the strong convexity of $\Psi(\boldsymbol{\theta})$. The matrix-vector product becomes

$$\boldsymbol{G}\boldsymbol{d} = \sum_{i=1}^{m}\sum_{j=1}^{n} \frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}}^\top \frac{\partial^2 \ell_{ij}}{\partial \hat{Y}_{ij}^2} \frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}}\boldsymbol{d} + \lambda \nabla^2 \Psi(\boldsymbol{\theta})\boldsymbol{d}. \tag{32}$$

From (11) we have

$$\frac{\partial^2 \ell_{ij}^-}{\partial \hat{Y}_{ij}^2} = a_i b_j. \tag{33}$$

Following (10), we use (33) to re-write $\boldsymbol{G}\boldsymbol{d}$ in (32) as

$$\underbrace{\sum_{(i,j)\in\mathbb{O}} \frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}}^\top \frac{\partial^2 \ell_{ij}^+}{\partial \hat{Y}_{ij}^2} \frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}}\boldsymbol{d}}_{\boldsymbol{G}^+\boldsymbol{d}} + \omega \underbrace{\sum_{i=1}^{m}\sum_{j=1}^{n} a_i b_j \frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}}^\top \frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}}\boldsymbol{d}}_{\boldsymbol{G}^-\boldsymbol{d}} + \lambda \nabla^2 \Psi(\boldsymbol{\theta})\boldsymbol{d}. \tag{34}$$

Let $\boldsymbol{d} = \begin{bmatrix} \boldsymbol{d}^u \\ \boldsymbol{d}^v \end{bmatrix}$ be some vector. By defining

$$\boldsymbol{w}_i = \frac{\partial f^i}{\partial \boldsymbol{\theta}^u}\boldsymbol{d}^u \in \mathbb{R}^k, \boldsymbol{h}_j = \frac{\partial g^j}{\partial \boldsymbol{\theta}^v}\boldsymbol{d}^v \in \mathbb{R}^k, \tag{35}$$

[2] If for the same $\boldsymbol{\theta}$, $L(\boldsymbol{\theta})$ is calculated before $\nabla L(\boldsymbol{\theta})$, then these matrices have been stored and are available as input.

from (18), we have

$$\frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}}\boldsymbol{d} = \boldsymbol{q}_j^\top \boldsymbol{w}_i + \boldsymbol{p}_i^\top \boldsymbol{h}_j. \tag{36}$$

To compute $\boldsymbol{G}^+\boldsymbol{d}$ in (34), we first compute

$$\boldsymbol{W} = \begin{bmatrix} \boldsymbol{w}_1^\top \\ \vdots \\ \boldsymbol{w}_m^\top \end{bmatrix} \in \mathbb{R}^{m\times k} \text{ and } \boldsymbol{H} = \begin{bmatrix} \boldsymbol{h}_1^\top \\ \vdots \\ \boldsymbol{h}_n^\top \end{bmatrix} \in \mathbb{R}^{n\times k}$$

in $O(mF(f) + nF(g))$ time and store them in $O((m+n)k)$ space. From (36), let $Z \in \mathbb{R}^{m\times n}$ be a sparse matrix with

$$Z_{ij} = \begin{cases} \frac{\partial^2 \ell_{ij}^+}{\partial \hat{Y}_{ij}^2}\frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}}\boldsymbol{d} = \frac{\partial^2 \ell_{ij}^+}{\partial \hat{Y}_{ij}^2}(\boldsymbol{q}_j^\top \boldsymbol{w}_i + \boldsymbol{p}_i^\top \boldsymbol{h}_j) & (i,j) \in \mathbb{O}, \\ 0 & \text{otherwise}, \end{cases}$$

which can be constructed in $O(|\mathbb{O}|k)$ time. Then from (19) and similar to the situation of computing (21) and (23), we have

$$\boldsymbol{G}^+\boldsymbol{d} = \sum_{(i,j)\in\mathbb{O}} \frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}}^\top \frac{\partial^2 \ell_{ij}^+}{\partial \hat{Y}_{ij}^2} \frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}}\boldsymbol{d} = \sum_{i=1}^{m}\sum_{j=1}^{n} \frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}}^\top Z_{ij} = \begin{bmatrix} \langle \mathbf{J}^u, \boldsymbol{Z}\boldsymbol{Q}\rangle \\ \langle \mathbf{J}^v, \boldsymbol{Z}^\top \boldsymbol{P}\rangle \end{bmatrix}, \tag{37}$$

where $\boldsymbol{Z}\boldsymbol{Q}$ and $\boldsymbol{Z}^\top \boldsymbol{P}$ can be computed in $O(|\mathbb{O}|k)$ time and $O((m+n)k)$ space.

By using (18) and (36), $\boldsymbol{G}^-\boldsymbol{d}$ in (34) can be computed by

$$\begin{aligned} \boldsymbol{G}^-\boldsymbol{d} &= \sum_{i=1}^{m}\sum_{j=1}^{n} a_i b_j \frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}}^\top \frac{\partial \hat{Y}_{ij}}{\partial \boldsymbol{\theta}}\boldsymbol{d} \\ &= \sum_{i=1}^{m}\sum_{j=1}^{n} \begin{bmatrix} \frac{\partial f^i}{\partial \boldsymbol{\theta}^u}^\top \boldsymbol{q}_j \\ \frac{\partial g^j}{\partial \boldsymbol{\theta}^v}^\top \boldsymbol{p}_i \end{bmatrix} a_i b_j (\boldsymbol{w}_i^\top \boldsymbol{q}_j + \boldsymbol{p}_i^\top \boldsymbol{h}_j) \qquad (38) \\ &= \begin{bmatrix} \langle \mathbf{J}^u, \boldsymbol{A}\boldsymbol{W}\boldsymbol{Q}_c + \boldsymbol{A}\boldsymbol{P}\boldsymbol{H}_c\rangle \\ \langle \mathbf{J}^v, \boldsymbol{B}\boldsymbol{H}\boldsymbol{P}_c + \boldsymbol{B}\boldsymbol{Q}\boldsymbol{W}_c\rangle \end{bmatrix}, \qquad (39) \end{aligned}$$

where (39) is by a similar derivation from (24) to (26), and by first computing the following $k \times k$ matrices in $O((m+n)k^2)$ time:

$$\boldsymbol{W}_c = \sum_{i=1}^{m} a_i \boldsymbol{w}_i \boldsymbol{p}_i^\top = \boldsymbol{W}^\top \boldsymbol{A}\boldsymbol{P}, \quad \boldsymbol{H}_c = \sum_{j=1}^{n} b_j \boldsymbol{h}_j \boldsymbol{q}_j^\top = \boldsymbol{H}^\top \boldsymbol{B}\boldsymbol{Q}.$$

Combining (37) and (39), we can compute (34) by

$$\boldsymbol{G}\boldsymbol{d} = \begin{bmatrix} \langle \mathbf{J}^u, \boldsymbol{Z}\boldsymbol{Q} + \omega(\boldsymbol{A}\boldsymbol{W}\boldsymbol{Q}_c + \boldsymbol{A}\boldsymbol{P}\boldsymbol{H}_c)\rangle \\ \langle \mathbf{J}^v, \boldsymbol{Z}^\top \boldsymbol{P} + \omega(\boldsymbol{B}\boldsymbol{H}\boldsymbol{P}_c + \boldsymbol{B}\boldsymbol{Q}\boldsymbol{W}_c)\rangle \end{bmatrix} + \lambda \nabla^2 \Psi(\boldsymbol{\theta})\boldsymbol{d}. \tag{40}$$

Thus the total time complexity of (40) is

$$O(|\mathbb{O}|k + (m+n)k^2 + 2mF(f) + 2nF(g)),$$

where the term '2' comes from the operations in (35) and (38). More details are in Section 5.

4 OPTIMIZATION METHODS FOR EXTREME SIMILARITY LEARNING WITH NONLINEAR EMBEDDINGS

With the efficient computation of key components given in Section 3, we demonstrate that many optimization algorithms can be used. While block-wise minimization is applicable, we mentioned in Section 3.1 that without a block-convex $L(\boldsymbol{\theta})$, such a setting may not be advantageous. Thus in our discussion we focus more on standard optimization techniques that update all variables at a time.

4.1 (Stochastic) Gradient Descent Method

The standard gradient descent method considers the negative gradient direction

$$\boldsymbol{s} = -\nabla L(\boldsymbol{\theta}) \tag{41}$$

and update $\boldsymbol{\theta}$ by

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \delta \boldsymbol{s}, \tag{42}$$

where δ is the step size. To ensure the convergence, usually a line-search procedure is conducted to find δ satisfying the following sufficient decrease condition

$$L(\boldsymbol{\theta} + \delta \boldsymbol{s}) \le L(\boldsymbol{\theta}) + \eta \delta \boldsymbol{s}^\top \nabla L(\boldsymbol{\theta}), \tag{43}$$

where $\eta \in (0, 1)$ is a pre-defined constant. During line search, given any $\boldsymbol{\theta} + \delta \boldsymbol{s}$, the corresponding function value must be calculated.

Combined with the cost of $\nabla L(\boldsymbol{\theta})$ in (28) and the cost of $L(\boldsymbol{\theta})$ in (15), the complexity of each gradient descent iteration is

$$O((\#\text{LS} + 1) \times (|\mathbb{O}|k + (m+n)k^2 + mF(f) + nF(g))), \tag{44}$$

where the term '1' in $(\#\text{LS} + 1)$ comes from the cost of $\nabla L(\boldsymbol{\theta})$, and #LS is the number of line search steps.

In some machine learning applications, the gradient calculation is considered too expensive so an approximation using partial data samples is preferred. This leads to the stochastic gradient (SG) method. Without the full gradient, the update rule (42) may no longer lead to the decrease of the function value. Thus a line search procedure is not useful and the step size δ is often decided by some pre-defined rules or tuned as a hyper-parameter.

Unfortunately, it is known that for extreme similarity learning, a direct implementation of SG methods faces the difficulty of $O(mn)$ cost (e.g., [25]). Now all (i, j) pairs are considered as our data set and the cost of each SG step is proportional to the number of sampled pairs. Thus to go over all or most pairs many SG steps are needed and the cost is $O(mn)$. In contrast, by techniques in Section 3.3 to calculate the gradient in $O(m+n)$ cost, all pairs have been considered. A natural remedy is to incorporate techniques in Section 3.3 to the SG method. Instead of randomly selecting pairs in an SG step, [13] proposed to select pairs in a set $\mathbb{B} = \hat{\mathbb{U}} \times \hat{\mathbb{V}}$ where $\hat{\mathbb{U}} \subseteq \mathbb{U}$ and $\hat{\mathbb{V}} \subseteq \mathbb{V}$. If $\hat{m} = |\hat{\mathbb{U}}|$ and $\hat{n} = |\hat{\mathbb{V}}|$, then by slightly modifying the derivation in (27), calculating the subsampled gradient involves $O(\hat{m}+\hat{n})$ instead of $O(\hat{m}\hat{n})$ cost. Analogous to (28), the subsampled gradient on $\mathbb{B}$ can be calculated in

$$O(|\hat{\mathbb{O}}|k + (\hat{m}+\hat{n})k^2 + \hat{m}F(f) + \hat{n}F(g)),$$

where $\hat{\mathbb{O}}$ is the subset of observed pairs falling in $\hat{\mathbb{U}} \times \hat{\mathbb{V}}$. Then the complexity of each data pass of handling mn pairs via $\frac{mn}{\hat{m}\hat{n}}$ SG steps is

$$O(|\mathbb{O}|k + (\frac{mn}{\hat{n}} + \frac{mn}{\hat{m}})k^2 + \frac{mn}{\hat{n}}F(f) + \frac{mn}{\hat{m}}F(g)). \tag{45}$$

Thus the $O(mn)$ cost is deducted by a factor.

We consider another SG method that has been roughly discussed in [13]. The idea is to note that if $\mathbb{O}_{i,:} = \{j : (i,j) \in \mathbb{O}\}$ and $\mathbb{O}_{:,j} = \{i : (i,j) \in \mathbb{O}\}$, then

$$\begin{aligned}\sum_{i=1}^{m}\sum_{j=1}^{n} a_i b_j (\cdots) &= \sum_{i=1}^{m}\sum_{j' \in \mathbb{O}_{i,:}}\sum_{j=1}^{n}\sum_{i' \in \mathbb{O}_{:,j}} \frac{a_i}{|\mathbb{O}_{i,:}|}\frac{b_j}{|\mathbb{O}_{:,j}|}(\cdots) \\ &= \sum_{(i,j') \in \mathbb{O}}\sum_{(i',j) \in \mathbb{O}} \frac{a_i}{|\mathbb{O}_{i,:}|}\frac{b_j}{|\mathbb{O}_{:,j}|}(\cdots).\end{aligned} \tag{46}$$

Then instead of selecting a subset from mn pairs, the setting becomes to select two independent sets $\hat{\mathbb{O}}_1, \hat{\mathbb{O}}_2 \subset \mathbb{O}$ for constructing the subsampled gradient, where $\hat{\mathbb{O}}_1$ and $\hat{\mathbb{O}}_2$ respectively corresponding to the first and the second summations in (46). Therefore, the task of going over mn pairs is replaced by selecting $\hat{\mathbb{O}}_1 \subset \mathbb{O}$, and $\hat{\mathbb{O}}_2 \subset \mathbb{O}$ at a time to cover the whole $\mathbb{O} \times \mathbb{O}$ after many SG steps. We leave details of deriving the subsampled gradient in Appendix A.2, and the complexity for the task comparable to going over all mn pairs by standard SG methods is

$$O((\frac{|\mathbb{O}|^2}{|\hat{\mathbb{O}}_1|} + \frac{|\mathbb{O}|^2}{|\hat{\mathbb{O}}_2|})(k + k^2 + F(f) + F(g))). \tag{47}$$

A comparison with (45) shows how (47) might alleviate the $O(mn)$ cost. This relies on $|\mathbb{O}| \ll mn$ and some appropriate choices of $|\hat{\mathbb{O}}_1|$ and $|\hat{\mathbb{O}}_2|$. We refer to this method as the SG method based on (46).

The above setting is further extended in [13] to Stochastic Online Gramian (SOGram), which applies a variance reduction scheme to improve the subsampled gradient. Details are in Appendix A.3.

We compare the above methods in Table 2 by checking the cost per data pass, which roughly indicates the process of handling mn pairs. Note that the total cost also depends on the number of data passes. In experiments we will see the trade-off between the cost per data pass and the number of total passes.

4.2 Methods Beyond Gradient Descent

We can further consider methods that incorporate the second-order information. By considering the following second-order Taylor expansion

$$L(\boldsymbol{\theta} + \boldsymbol{s}) \approx L(\boldsymbol{\theta}) + \nabla L(\boldsymbol{\theta})^\top \boldsymbol{s} + \frac{1}{2}\boldsymbol{s}^\top \nabla^2 L(\boldsymbol{\theta}) \boldsymbol{s}, \tag{48}$$

Newton methods find a search direction $\boldsymbol{s}$ by minimizing (48). However, this way of finding $\boldsymbol{s}$ is often complicated and expensive. In the rest of this section, we discuss some practically viable options to use the second-order information.

4.2.1 Gauss-Newton Method. It is mentioned in Section 3.4 that Gauss-Newton matrix $\boldsymbol{G}$ in (31) is a positive-definite approximation of $\nabla^2 L(\boldsymbol{\theta})$. Thus replacing $\nabla^2 L(\boldsymbol{\theta})$ in (48) with $\boldsymbol{G}$ leads to a convex quadratic optimization problem with the solution satisfying

$$\boldsymbol{G}\boldsymbol{s} = -\nabla L(\boldsymbol{\theta}). \tag{49}$$

However, for large-scale applications, not only is $\boldsymbol{G}$ in (31) too large to be stored in memory, but the matrix inversion is also computationally expensive.

Following [14, 15], we may solve (49) by the conjugate gradient (CG) method [10], which involves a sequence of matrix-vector products $\boldsymbol{G}\boldsymbol{d}$ between $\boldsymbol{G}$ and some vector $\boldsymbol{d}$. The derivation in Section 3.4 has shown how to calculate $\boldsymbol{G}\boldsymbol{d}$ without explicitly forming and storing $\boldsymbol{G}$. Therefore, Gauss-Newton methods are a feasible approach for extreme similarity learning without the $O(mn)$ cost.

After a Newton direction $\boldsymbol{s}$ is obtained, the line search procedure discussed in Section 4.1 is conducted to find a suitable δ to ensure the convergence. The complexity of each Newton iteration is

$$\begin{aligned}&O(\#\text{CG} \times (|\mathbb{O}|k + (m+n)k^2 + 2mF(f) + 2nF(g)) \\ &+(\#\text{LS} + 1) \times (|\mathbb{O}|k + (m+n)k^2 + mF(f) + nF(g))),\end{aligned} \tag{50}$$

Table 2: A comparison of gradient-based methods on time complexity of each data pass. For the last setting, $|\hat{\mathbb{O}}| = \min(|\hat{\mathbb{O}}_1|, |\hat{\mathbb{O}}_2|)$.

Method	Time complexity per data pass
Naive SG	$mnF(f) + mnF(g)$
Gradient (Alg. 1)	$\|\mathbb{O}\|k + (m+n)k^2 + mF(f) + nF(g)$
Selecting $\mathbb{B} = \hat{\mathbb{U}} \times \hat{\mathbb{V}}$	$\|\mathbb{O}\|k + (\frac{n}{\hat{n}}m + \frac{m}{\hat{m}}n)k^2 + \frac{n}{\hat{n}}mF(f) + \frac{m}{\hat{m}}nF(g)$
SG based on (46)	$\frac{\|\mathbb{O}\|^2}{\|\hat{\mathbb{O}}\|}k + \frac{\|\mathbb{O}\|^2}{\|\hat{\mathbb{O}}\|}k^2 + \frac{\|\mathbb{O}\|^2}{\|\hat{\mathbb{O}}\|}F(f) + \frac{\|\mathbb{O}\|^2}{\|\hat{\mathbb{O}}\|}F(g)$

where #CG is numbers of CG steps. The overall procedures of the CG method and the Gauss-Newton method are summarized in Appendix A.4.

4.2.2 Diagonal Scaling Methods. From (50), the CG procedure may be expensive so a strategy is to replace $\boldsymbol{G}$ in (49) with a positive-definite diagonal matrix, which becomes a very loose approximation of $\nabla^2 L(\boldsymbol{\theta})$. In this case, $\boldsymbol{s}$ can be efficiently obtained by dividing each value in $-\nabla L(\boldsymbol{\theta})$ by the corresponding diagonal element. Methods following this idea are referred to as diagonal scaling methods.

For example, the AdaGrad algorithm [6] is a popular diagonal scaling method. Given $\nabla L(\boldsymbol{\theta})$ of current $\boldsymbol{\theta}$, AdaGrad maintains a zero-initialized diagonal matrix $M \in \mathbb{R}^{D \times D}$ by

$$\boldsymbol{M} \leftarrow \boldsymbol{M} + \operatorname{diag}(\nabla L(\boldsymbol{\theta}))^2.$$

Then by considering $(\mu \boldsymbol{I} + \boldsymbol{M})^{\frac{1}{2}}$, where $\boldsymbol{I}$ is the identity matrix and $\mu > 0$ is a small constant to ensure the invertibility, the direction is

$$\boldsymbol{s} = -(\mu \boldsymbol{I} + \boldsymbol{M})^{-\frac{1}{2}} \nabla L(\boldsymbol{\theta}). \tag{51}$$

In comparison with the gradient descent method, the extra cost is minor in $O(D)$, where $D = D_u + D_v$ is the total number of variables. Instead of using the full $\nabla L(\boldsymbol{\theta})$, this method can be extended to use subsampled gradient by the SG framework discussed in Section 4.1.

5 IMPLEMENTATION DETAILS

So far we used $F(f)$ and $F(g)$ to represent the cost of operations on nonlinear embeddings. Here we discuss more details if automatic differentiation is employed for easily trying different network architectures. To begin, note that $[\partial f^i/\partial \boldsymbol{\theta}^u, \partial g^j/\partial \boldsymbol{\theta}^v]$ is the Jacobian of $(f^i(\boldsymbol{\theta}^u), g^j(\boldsymbol{\theta}^v))$. Transposed Jacobian-vector products are used in $\nabla L(\boldsymbol{\theta})$ for all methods and $\boldsymbol{Gd}$ for the Gauss-Newton method. In fact, (27) and (40) are almost in the same form. It is known that for such an operation, the reverse-mode automatic differentiation (i.e., back-propagation) should be used so that only one pass of the network is needed [2]. For feedforward networks, including extensions such as convolutional networks, the cost of a backward process is within a constant factor of the forward process for computing the objective value (e.g., Section 5.2 of [21]). The remaining major operation to be discussed is the Jacobian-vector product in (35) in Section 4.2.1. This operation can be implemented with the forward-mode automatic differentiation in just one pass [2], and the cost is also within a constant factor of the cost of forward objective evaluation.[3] We conclude that if neural networks are used, using $F(f)$ and $F(g)$ as the cost of nonlinear embeddings is suitable.

[3]See, for example, the discussion at https://www.csie.ntu.edu.tw/~cjlin/courses/optdl2021/slides/newton_gauss_newton.pdf.

Table 3: Data statistics and the selected hyper-parameters.

Data set	m	n	$\|\mathbb{O}\|$	$\log_2 \omega$	$\log_2 \lambda$
`ml1m`	6,037	3,513	517,770	−4	2
`ml10m`	69,786	10,210	4,505,820	−8	−2
`netflix`	478,251	17,768	51,228,351	−8	0
`wiki-simple`	85,270	55,695	5,478,549	−10	2

Neural networks are routinely trained on GPU instead of CPU for better efficiency, but the smaller memory of GPU causes difficulties for larger problems or network architectures. This situation is even more serious for our case because some other matrices besides the two networks are involved; see (27) and (40). We develop some techniques to make large-scale training feasible, where details are left in supplementary materials.

6 EXPERIMENTS

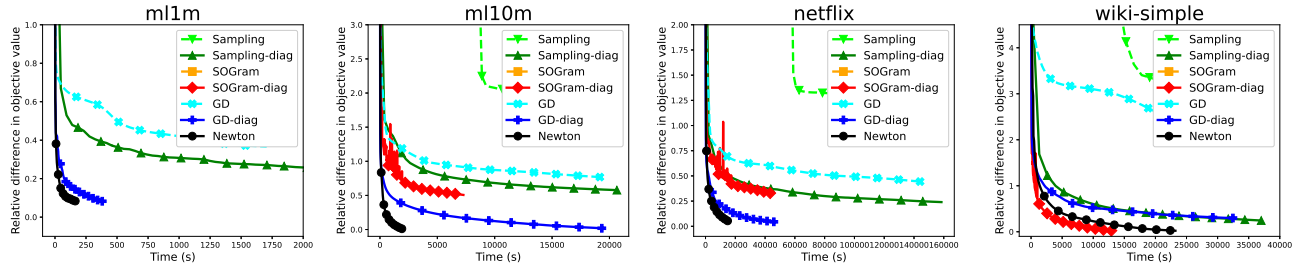
In this section, after presenting our experimental settings, we compare several methods discussed in Section 4 in terms of the convergence speed on objective value decreasing and achieving the final test performance.

6.1 Experimental Settings

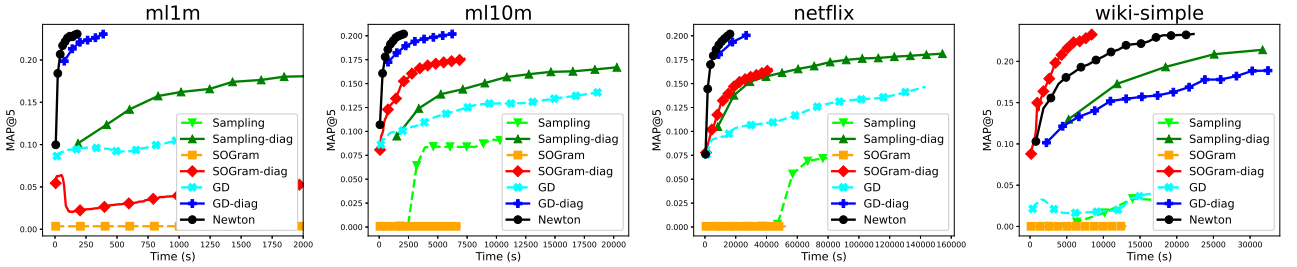
6.1.1 Data sets. We consider three data sets of recommender systems with implicit feedback and one data set of link predictions. For data sets of recommender systems with implicit feedback, the target similarity of each observed user-item pair is 1, while those of the unobserved pairs are unrevealed. We consider `ml1m`, `ml10m`, and `netflix` provided by [25]. For the data set of link predictions, we follow [13] to consider the problem of learning the intra-site links between Wikipedia pages, where the target similarity of each observed $\text{page}_{\text{from}}$-$\text{page}_{\text{to}}$ pair is 1 if there is a link from $\text{page}_{\text{from}}$ to page_{to}, and unrevealed otherwise. However, because the date information of the dump used in [13] is not provided, their generated sets are irreproducible. We use the latest Wikipedia graph[4] on pages in simple English and denote the generated data set as `wiki-simple`. For `ml1m`, `ml10m` and `netflix`, training and test sets are available. For `wiki-simple`, we follow [13] to use a 9-to-1 ratio for the training/test split. The statistics of all data sets are listed in Table 3. Note that the aim here is to check the convergence behavior, so we do not need a validation set for hyper-parameter selection.

6.1.2 Model and Hyper-parameters. We train a two-tower neural network as in Figure 1. Both towers have three fully-connected layers, where the first two layers contain 256 hidden units equipped with ELU [5] activation, and the last layer contains $k = 128$ hidden units without activation. For all experiments, we choose the logistic loss $\ell(y, \hat{y}) = \log(1 + \exp(-y\hat{y}))$, an L2 regularizer $\Psi(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{\theta}\|_2^2$, and uniform weights $a_i = 1, \forall i = 1, \ldots, m,\ b_j = 1, \forall j = 1, \ldots, n$ in (7). For the imputed labels $\tilde{\boldsymbol{Y}}$, we follow past works [25, 26] to apply a constant -1 for any unobserved pair by setting $\tilde{\boldsymbol{p}}_i = -\mathbf{1}/\sqrt{k}$, and $\tilde{\boldsymbol{q}}_j = \mathbf{1}/\sqrt{k}$. For ω and λ, for each data set, we consider a grid of hyper-parameter combinations and select the one achieving the best results on the test set. The selected ω and λ for each data set are listed in Table 3.

[4]https://dumps.wikimedia.org/simplewiki/20200901/



(a) Relative difference in objective value versus training time.



(b) MAP@5 versus training time.

Figure 2: Comparison of different algorithms on four data sets. SOGram **and** Sampling **may be too slow to be shown.**

6.1.3 Compared Optimization Methods. We compare seven optimization methods for extreme similarity learning, which are categorized into two groups. The first group includes the following three gradient-based methods discussed in Section 4.1.

- GD: This is the gradient descent method. For each step, we compute $\boldsymbol{s} = -\nabla L(\boldsymbol{\theta})$ by (27). To ensure the convergence, backtracking line search is conducted to find the step size δ.
- SOGram: This is a stochastic gradient method proposed in [13]; see the discussion in Appendix A.3. We set $|\hat{\mathbb{O}}_1| = |\hat{\mathbb{O}}_2| = \rho|\mathbb{O}|$, where $\rho = 10^{-2}$ is the sampling ratio of observed pairs. We follow [13] to set $\alpha = 0.1$, which is a hyper-parameter of controlling the bias-variance tradeoff.
- Sampling: This is the stochastic gradient method of choosing $\mathbb{B} = \hat{\mathbb{U}} \times \hat{\mathbb{V}}$ [13]. To make the number of steps in each data pass comparable to SOGram, for Sampling we set $\hat{m} = \rho m$ and $\hat{n} = \rho n$, where ρ is the ratio used in SOGram.

The step size δ in SOGram and Sampling is set to 2^{-25}, 2^{-23}, 2^{-25} and 2^{-25} respectively for `ml1m`, `ml10m`, `netflix` and `wiki-simple`. These values are from GD's most used δ.

The second group includes the following four methods incorporating the second-order information discussed in Section 4.2.

- Newton: This is the Gauss-Newton method proposed in Section 4.2.1. For CG, we set the stopping tolerance $\xi = 0.1$ in Appendix Algorithm 3 and 30 as the maximal number of CG steps.
- GD-diag, SOGram-diag, Sampling-diag: These three methods are respectively extended from GD, SOGram, and Sampling by applying the diagonal scaling in (51). For GD-diag, the same as GD, it finds δ by conducting backtracking line search. For SOGram-diag and Sampling-diag, we set $\delta = 0.01$.

For GD, Newton, and GD-diag, which conduct line search, we set $\eta = 10^{-4}$ for the sufficient decrease condition (43) and sequentially check δ in $\{\delta_{\text{init}}, \frac{\delta_{\text{init}}}{2}, \frac{\delta_{\text{init}}}{4}, \dots\}$, where δ_{init} is the step size taken in the last iteration, and is doubled every five iterations to avoid being stuck with a small step size.

6.1.4 Environment and Implementation. We use TensorFlow [1] compiled with Intel® Math Kernel Library (MKL) to implement all algorithms. Because most operations are matrix-based and MKL is optimized for such operations, our implementation should be sufficiently efficient. For sparse operations involving iterating $\mathbb{O}$ (e.g., $L^+(\boldsymbol{\theta})$ and $\boldsymbol{XQ}$), we implement them by NumPy [8] and a C extension, where we parallelize these operations by OpenMP. As our applied neural networks described in Section 6.1.2 are not complex, we empirically do not observe many advantages of GPU over CPU on the running time. Thus all experiments are conducted on a Linux machine with 10 cores of Intel® Core™ i7-6950X CPUs and 128GB memory.

6.1.5 Evaluation Criterion of Test Performance. On test sets, we report mean average precision (MAP) on top-5 ranked entities. For left entity i, let $\tilde{\mathbb{O}}_{i,:}$ be the set of its similar right entities in the test set, and $\tilde{\mathbb{O}}^{\text{sorted}}_{i,:K}$ be the set of top-K right entities with the highest predicted similarity. We define

$$\text{MAP@5} = \frac{1}{5}\sum_{K=1}^{5}\frac{1}{\tilde{m}}\sum_{i=1}^{\tilde{m}}\frac{1}{K}|\tilde{\mathbb{O}}_{i,:} \cap \tilde{\mathbb{O}}^{\text{sorted}}_{i,:K}|, \tag{52}$$

where $\tilde{m}$ is the number of left entities in the test set.

6.2 A Comparison on the Convergence Speed

We first investigate the relationship between the running time and the relative difference in objective value

$$(L(\boldsymbol{\theta}) - L^*)/L^*,$$

where L^* is the lowest objective value reached among all settings. The result is presented in Figure 2a.

For the three gradient-based methods, SOGram, and Sampling are slower than GD. This situation is different from that of training general neural networks, where SG are the dominant methods. The reason is that though Sampling and SOGram take more iterations than GD in each data pass, from Table 2, they have much higher complexity per data pass.

For GD-diag, SOGram-diag, Sampling-diag, and Newton, which incorporate the second-order information, they are consistently faster than gradient-based methods. Specifically, between Newton and GD, the key difference is that Newton additionally considers the second-order information for yielding each direction. The great superiority of Newton confirms the importance of leveraging the second-order information. By comparing GD-diag and GD, we observe that GD-diag is much more efficient. Though both have similar complexities for obtaining search directions, GD-diag leverages partial second-order information from AdaGrad's diagonal scaling. However, the effect of the diagonal scaling is limited, so GD-diag is slower than Newton. Finally, GD-diag is still faster than SOGram-diag, Sampling-diag as in the case of without diagonal scaling.

Next, we present in Figure 2b the relationship between the running time and MAP@5 evaluated on test sets. It is observed that a method with faster convergence in objective values is also faster in MAP@5.

7 CONCLUSIONS

In this work, we study extreme similarity learning with nonlinear embeddings. The goal is to alleviate the $O(mn)$ cost in the training process. While this topic has been well studied for the situation of using linear embeddings, a systematic study for nonlinear embeddings was lacking. We fill the gap in the following aspects. First, for important operations in optimization algorithms such as function and gradient evaluation, clean formulations with $O(m+n)$ cost are derived. Second, these formulations enable the use of many optimization algorithms for extreme similarity learning with nonlinear embedding models. We detailedly discuss some of them and check implementation issues. Experiments show that efficient training by some algorithms can be achieved.

ACKNOWLEDGMENTS

We would like to thank Hong Zhu, Yaxu Liu, and Jui-Nan Yen for helpful discussions. This work was supported by MOST of Taiwan grant 107-2221-E-002-167-MY3.

REFERENCES

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *OSDI*.

[2] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. Automatic differentiation in machine learning: a survey. *JMLR* 18 (2018), 1–43.

[3] Immanuel Bayer, Xiangnan He, Bhargav Kanagal, and Steffen Rendle. 2017. A generic coordinate descent framework for learning from implicit feedback. In *WWW*.

[4] Wei-Sheng Chin, Bo-Wen Yuan, Meng-Yuan Yang, and Chih-Jen Lin. 2018. An Efficient Alternating Newton Method for Learning Factorization Machines. *ACM TIST* 9 (2018), 72:1–72:31.

[5] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. 2016. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). In *ICLR*.

[6] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *JMLR* 12 (2011), 2121–2159.

[7] Ali Mamdouh Elkahky, Yang Song, and Xiaodong He. 2015. A multi-view deep learning approach for cross domain user modeling in recommendation systems. In *WWW*.

[8] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362.

[9] Xiangnan He, Hanwang Zhang, Min-Yen Kan, and Tat-Seng Chua. 2016. Fast matrix factorization for online recommendation with implicit feedback. In *SIGIR*.

[10] Magnus Rudolph Hestenes and Eduard Stiefel. 1952. Methods of Conjugate Gradients for Solving Linear Systems. *JRNBS* 49 (1952), 409–436.

[11] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. 2020. Embedding-based Retrieval in Facebook Search. In *KDD*.

[12] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. 2013. Learning deep structured semantic models for web search using clickthrough data. In *CIKM*.

[13] Walid Krichene, Nicolas Mayoraz, Steffen Rendle, Li Zhang, Xinyang Yi, Lichan Hong, Ed Chi, and John Anderson. 2019. Efficient Training on Very Large Corpora via Gramian Estimation. In *ICLR*.

[14] Chih-Jen Lin, Ruby C. Weng, and S. Sathiya Keerthi. 2008. Trust region Newton method for large-scale logistic regression. *JMLR* 9 (2008), 627–650.

[15] James Martens. 2010. Deep learning via Hessian-free optimization. In *ICML*.

[16] Rong Pan and Martin Scholz. 2009. Mind the Gaps: Weighting the Unknown in Large-scale One-class Collaborative Filtering. In *KDD*.

[17] Rong Pan, Yunhong Zhou, Bin Cao, Nathan N Liu, Rajan Lukose, Martin Scholz, and Qiang Yang. 2008. One-class collaborative filtering. In *ICDM*.

[18] Steffen Rendle, Walid Krichene, Li Zhang, and John Anderson. 2020. Neural Collaborative Filtering vs. Matrix Factorization Revisited. In *RecSys*.

[19] Nicol N. Schraudolph. 2002. Fast curvature matrix-vector products for second-order gradient descent. *Neural Comput.* 14 (2002), 1723–1738.

[20] Chien-Chih Wang, Chun-Heng Huang, and Chih-Jen Lin. 2015. Subsampled Hessian Newton Methods for Supervised Learning. *Neural Comput.* 27 (2015), 1766–1795.

[21] Chien-Chih Wang, Kent Loong Tan, and Chih-Jen Lin. 2020. Newton Methods for Convolutional Neural Networks. *ACM TIST* 11 (2020), 19:1–19:30.

[22] Xiaojie Wang, Rui Zhang, Yu Sun, and Jianzhong Qi. 2019. Doubly robust joint learning for recommendation on data missing not at random. In *ICML*.

[23] Yinfei Yang, Steve Yuan, Daniel Cer, Sheng-Yi Kong, Noah Constant, Petr Pilar, Heming Ge, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. 2018. Learning Semantic Textual Similarity from Conversations. In *RepL4NLP*.

[24] Xinyang Yi, Ji Yang, Lichan Hong, Derek Zhiyuan Cheng, Lukasz Heldt, Aditee Kumthekar, Zhe Zhao, Li Wei, and Ed Chi. 2019. Sampling-Bias-Corrected Neural Modeling for Large Corpus Item Recommendations. In *RecSys*.

[25] Hsiang-Fu Yu, Mikhail Bilenko, and Chih-Jen Lin. 2017. Selection of Negative Samples for One-class Matrix Factorization. In *SDM*.

[26] Hsiang-Fu Yu, Hsin-Yuan Huang, Inderjit S. Dihillon, and Chih-Jen Lin. 2017. A Unified Algorithm for One-class Structured Matrix Factorization with Side Information. In *AAAI*.

[27] Hsiang-Fu Yu, Prateek Jain, Purushottam Kar, and Inderjit S. Dhillon. 2014. Large-scale Multi-label Learning with Missing Labels. In *ICML*.

[28] Bowen Yuan, Jui-Yang Hsia, Meng-Yuan Yang, Hong Zhu, Chihyao Chang, Zhenhua Dong, and Chih-Jen Lin. 2019. Improving Ad Click Prediction by Considering Non-displayed Events. In *CIKM*.

[29] Bowen Yuan, Meng-Yuan Yang, Jui-Yang Hsia, Hong Zhu, Zhirong Liu, Zhenhua Dong, and Chih-Jen Lin. 2019. *One-class Field-aware Factorization Machines for Recommender Systems with Implicit Feedbacks.* Technical Report. National Taiwan Univ.

[30] Kai Zhong, Zhao Song, Prateek Jain, and Inderjit S Dhillon. 2019. Provable Non-linear Inductive Matrix Completion. In *NIPS*.

A APPENDIX

A.1 Detailed Derivation of $L^-(\boldsymbol{\theta})$

To compute $L^-(\boldsymbol{\theta})$, we rewrite it as

$$\begin{aligned}
L^-(\boldsymbol{\theta}) =& \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{n} a_i b_j (\tilde{\boldsymbol{p}}_i^\top \tilde{\boldsymbol{q}}_j - \boldsymbol{p}_i^\top \boldsymbol{q}_j)^2 \\
=& \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{n} a_i b_j \Big(\tilde{\boldsymbol{p}}_i^\top \tilde{\boldsymbol{q}}_j (\tilde{\boldsymbol{q}}_j^\top \tilde{\boldsymbol{p}}_i) - 2\tilde{\boldsymbol{p}}_i^\top \tilde{\boldsymbol{q}}_j (\boldsymbol{q}_j^\top \boldsymbol{p}_i) + \boldsymbol{p}_i^\top \boldsymbol{q}_j (\boldsymbol{q}_j^\top \boldsymbol{p}_i)\Big) \\
=& \frac{1}{2}\sum_{i=1}^{m}\Big(a_i \tilde{\boldsymbol{p}}_i^\top (\sum_{j=1}^{n} b_j \tilde{\boldsymbol{q}}_j \tilde{\boldsymbol{q}}_j^\top)\tilde{\boldsymbol{p}}_i - 2a_i \tilde{\boldsymbol{p}}_i^\top (\sum_{j=1}^{n} b_j \tilde{\boldsymbol{q}}_j \boldsymbol{q}_j^\top)\boldsymbol{p}_i \\
&+ a_i \boldsymbol{p}_i^\top (\sum_{j=1}^{n} b_j \boldsymbol{q}_j \boldsymbol{q}_j^\top)\boldsymbol{p}_i\Big) \\
=& \frac{1}{2}\langle \tilde{\boldsymbol{P}}_c, \tilde{\boldsymbol{Q}}_c\rangle_{\mathrm{F}} - \langle \hat{\boldsymbol{P}}_c, \hat{\boldsymbol{Q}}_c\rangle_{\mathrm{F}} + \frac{1}{2}\langle \boldsymbol{P}_c, \boldsymbol{Q}_c\rangle_{\mathrm{F}}.
\end{aligned} \tag{53}$$

To calculate (53), we need matrices listed in (14). For $\tilde{\boldsymbol{P}}_c$ and $\tilde{\boldsymbol{Q}}_c$, they are constant matrices throughout the training process, so we can calculate them and $\langle \tilde{\boldsymbol{P}}_c, \tilde{\boldsymbol{Q}}_c\rangle_{\mathrm{F}}$ within $O((m+n)k^2)$ at the beginning. Thus this cost can be omitted in the complexity analysis. For other matrices in (14), $\boldsymbol{P}$ and $\boldsymbol{Q}$ are needed, but they have been pre-computed and cached in memory during computing $L^+(\boldsymbol{\theta})$. Thus all matrices in (14) can be obtained in $O((m+n)k^2)$ time and cached in $O(k^2)$ space. Then in (53), the Frobenius inner products between these matrices cost only $O(k^2)$ time. Through combining $L^+(\boldsymbol{\theta})$ calculation discussed in Section 3.2, Algorithm 2 summarizes the procedure of evaluating the objective function.

A.2 Detailed Derivation of a Stochastic Gradient Method Based on (46)

From (46), by defining $\bar{a}_i = \frac{a_i}{|\mathbb{O}_{i,:}|}$ and $\bar{b}_j = \frac{b_j}{|\mathbb{O}_{:,j}|}$, $L(\boldsymbol{\theta})$ in (10) can be written as

$$\lambda\Psi(\boldsymbol{\theta}) + \sum_{(i,j)\in\mathbb{O}} \ell_{ij}^+(Y_{ij}, \hat{Y}_{ij}) + \omega \sum_{(i,j')\in\mathbb{O}}\sum_{(i',j)\in\mathbb{O}} \frac{1}{2}\bar{a}_i\bar{b}_j(\tilde{Y}_{ij} - \hat{Y}_{ij})^2. \tag{54}$$

Then $\nabla L^-(\boldsymbol{\theta})$ in (24) can be rewritten as

$$\nabla L^-(\boldsymbol{\theta}) = \sum_{(i,j')\in\mathbb{O}}\sum_{(i',j)\in\mathbb{O}} \begin{bmatrix} \frac{\partial f^i}{\partial \boldsymbol{\theta}^u}^\top \boldsymbol{q}_j \\ \frac{\partial g^j}{\partial \boldsymbol{\theta}^v}^\top \boldsymbol{p}_i \end{bmatrix} \bar{a}_i\bar{b}_j(\boldsymbol{p}_i^\top \boldsymbol{q}_j - \tilde{\boldsymbol{p}}_i^\top \tilde{\boldsymbol{q}}_j) \tag{55}$$

$$= \begin{bmatrix} \sum_{(i,j')\in\mathbb{O}} \frac{\partial f^i}{\partial \boldsymbol{\theta}^u}^\top (\boldsymbol{Q}_{\mathbb{O}}\bar{a}_i\boldsymbol{p}_i - \hat{\boldsymbol{Q}}_{\mathbb{O}}^\top \bar{a}_i\tilde{\boldsymbol{p}}_i) \\ \sum_{(i',j)\in\mathbb{O}} \frac{\partial g^j}{\partial \boldsymbol{\theta}^v}^\top (\boldsymbol{P}_{\mathbb{O}}\bar{b}_j\boldsymbol{q}_j - \hat{\boldsymbol{P}}_{\mathbb{O}}^\top \bar{b}_j\tilde{\boldsymbol{q}}_j) \end{bmatrix}, \tag{56}$$

where similar to (14), we define

$$\begin{aligned}
\boldsymbol{P}_{\mathbb{O}} &= \sum_{(i,j')\in\mathbb{O}} \bar{a}_i\boldsymbol{p}_i\boldsymbol{p}_i^\top, \quad \boldsymbol{Q}_{\mathbb{O}} = \sum_{(i',j)\in\mathbb{O}} \bar{b}_j\boldsymbol{q}_j\boldsymbol{q}_j^\top, \\
\hat{\boldsymbol{P}}_{\mathbb{O}} &= \sum_{(i,j')\in\mathbb{O}} \bar{a}_i\tilde{\boldsymbol{p}}_i\boldsymbol{p}_i^\top, \quad \hat{\boldsymbol{Q}}_{\mathbb{O}} = \sum_{(i',j)\in\mathbb{O}} \bar{b}_j\tilde{\boldsymbol{q}}_j\boldsymbol{q}_j^\top.
\end{aligned} \tag{57}$$

and we omit details from (55) to (56) as the derivations are similar to those between (24) and (26).

On the other hand, from (17) and (18), we write $\nabla L^+(\boldsymbol{\theta})$ as

$$\nabla L^+(\boldsymbol{\theta}) = \begin{bmatrix} \sum_{(i,j)\in\mathbb{O}} \frac{\partial f^i}{\partial \boldsymbol{\theta}^u}^\top \boldsymbol{q}_j X_{ij} \\ \sum_{(i,j)\in\mathbb{O}} \frac{\partial g^j}{\partial \boldsymbol{\theta}^v}^\top \boldsymbol{p}_i X_{ij} \end{bmatrix}. \tag{58}$$

Because both (56) and (58) are now operations summing over $\mathbb{O}$, $\nabla L(\boldsymbol{\theta})$ can be written as

$$\begin{bmatrix} \sum_{(i,j)\in\mathbb{O}} \frac{\partial f^i}{\partial \boldsymbol{\theta}^u}^\top \big(\boldsymbol{q}_j X_{ij} + \omega(\boldsymbol{Q}_{\mathbb{O}}\bar{a}_i\boldsymbol{p}_i - \hat{\boldsymbol{Q}}_{\mathbb{O}}^\top \bar{a}_i\tilde{\boldsymbol{p}}_i)\big) \\ \sum_{(i,j)\in\mathbb{O}} \frac{\partial g^j}{\partial \boldsymbol{\theta}^v}^\top \big(\boldsymbol{p}_i X_{ij} + \omega(\boldsymbol{P}_{\mathbb{O}}\bar{b}_j\boldsymbol{q}_j - \hat{\boldsymbol{P}}_{\mathbb{O}}^\top \bar{b}_j\tilde{\boldsymbol{q}}_j)\big) \end{bmatrix} + \lambda\nabla\Psi(\boldsymbol{\theta}). \tag{59}$$

To derive a subsampled approach, we must consider two subsets $\hat{\mathbb{O}}_1 \subset \mathbb{O}$ and $\hat{\mathbb{O}}_2 \subset \mathbb{O}$ because from (57), two summations respectively over $\mathbb{O}$ are involved in the above gradient. This results in the following estimate of $\nabla L(\boldsymbol{\theta})$.

$$\frac{|\mathbb{O}|}{|\hat{\mathbb{O}}_1|} \begin{bmatrix} \sum_{(i,j)\in\hat{\mathbb{O}}_1} \frac{\partial f^i}{\partial \boldsymbol{\theta}^u}^\top \big(\boldsymbol{q}_j X_{ij} + \omega(\boldsymbol{Q}_2\bar{a}_i\boldsymbol{p}_i - \hat{\boldsymbol{Q}}_2^\top \bar{a}_i\tilde{\boldsymbol{p}}_i)\big) \\ \sum_{(i,j)\in\hat{\mathbb{O}}_1} \frac{\partial g^j}{\partial \boldsymbol{\theta}^v}^\top \big(\boldsymbol{p}_i X_{ij} + \omega(\boldsymbol{P}_2\bar{b}_j\boldsymbol{q}_j - \hat{\boldsymbol{P}}_2^\top \bar{b}_j\tilde{\boldsymbol{q}}_j)\big) \end{bmatrix} + \lambda\nabla\Psi(\boldsymbol{\theta}), \tag{60}$$

where

$$\begin{aligned}
\boldsymbol{P}_2 &= \frac{|\mathbb{O}|}{|\hat{\mathbb{O}}_2|}\sum_{(i,j)\in\hat{\mathbb{O}}_2} \bar{a}_i\boldsymbol{p}_i\boldsymbol{p}_i^\top, \quad \boldsymbol{Q}_2 = \frac{|\mathbb{O}|}{|\hat{\mathbb{O}}_2|}\sum_{(i,j)\in\hat{\mathbb{O}}_2} \bar{b}_j\boldsymbol{q}_j\boldsymbol{q}_j^\top, \\
\hat{\boldsymbol{P}}_2 &= \frac{|\mathbb{O}|}{|\hat{\mathbb{O}}_2|}\sum_{(i,j)\in\hat{\mathbb{O}}_2} \bar{a}_i\tilde{\boldsymbol{p}}_i\boldsymbol{p}_i^\top, \quad \hat{\boldsymbol{Q}}_2 = \frac{|\mathbb{O}|}{|\hat{\mathbb{O}}_2|}\sum_{(i,j)\in\hat{\mathbb{O}}_2} \bar{b}_j\tilde{\boldsymbol{q}}_j\boldsymbol{q}_j^\top.
\end{aligned} \tag{61}$$

To make (60) an unbiased estimate of $\nabla L(\boldsymbol{\theta})$, we need that $\hat{\mathbb{O}}_1$ and $\hat{\mathbb{O}}_2$ are independent of each other, and have the scaling factors $\frac{|\mathbb{O}|}{|\hat{\mathbb{O}}_1|}$, $\frac{|\mathbb{O}|}{|\hat{\mathbb{O}}_2|}$ respectively in (60) and (61).

Because $\hat{\mathbb{O}}_1$ and $\hat{\mathbb{O}}_2$ are independent subsets, we can swap them in (60) to have another estimate of $\nabla L(\boldsymbol{\theta})$. Then four matrices $\boldsymbol{P}_1, \boldsymbol{Q}_1, \hat{\boldsymbol{P}}_1$ and $\hat{\boldsymbol{Q}}_1$ similar to those in (61) must be calculated, but the summations are now over $\hat{\mathbb{O}}_1$. In the end, the two subsampled gradients can be averaged.

Similar to the discussion on (14) and (27), for each pair of $\hat{\mathbb{O}}_1$ and $\hat{\mathbb{O}}_2$, (59) can be computed in

$$O((|\hat{\mathbb{O}}_1| + |\hat{\mathbb{O}}_2|)(k + k^2 + F(f) + F(g))) \tag{62}$$

time. We take $O(|\hat{\mathbb{O}}_2|k^2)$ term as an example to illustrate the difference from (27). Now for matrices in (61), each involves $O(|\hat{\mathbb{O}}_2|k^2)$ cost, but for those in (14), $O(mk^2)$ or $O(nk^2)$ are needed. Therefore, from (28) to (62), we can see the $m+n$ term should be replaced by $|\hat{\mathbb{O}}_1| + |\hat{\mathbb{O}}_2|$.

Next we check the complexity. We have mentioned in Section 4.1 that the task of going over all mn pairs is now replaced by sampling $\hat{\mathbb{O}}_1 \times \hat{\mathbb{O}}_2$ to cover $\mathbb{O}\times\mathbb{O}$. Thus the time complexity is by multiplying the $O(\frac{|\mathbb{O}|^2}{|\hat{\mathbb{O}}_1||\hat{\mathbb{O}}_2|})$ steps and the cost in (62) for each SG step to have

$$O((\frac{|\mathbb{O}|^2}{|\hat{\mathbb{O}}_1|} + \frac{|\mathbb{O}|^2}{|\hat{\mathbb{O}}_2|})(k + k^2 + F(f) + F(g))). \tag{63}$$

Algorithm 4: An efficient implementation of Gauss-Newton method for solving (4)

Input: $\mathbb{U} = \{\dots, \boldsymbol{u}_i, \dots\}, \mathbb{V} = \{\dots, \boldsymbol{v}_j, \dots\}, \mathbb{O}, \tilde{\boldsymbol{P}}, \tilde{\boldsymbol{Q}}, \boldsymbol{A}, \boldsymbol{B}$.
Init:
Draw $\boldsymbol{\theta}$ randomly
$\langle \tilde{\boldsymbol{P}}_c, \tilde{\boldsymbol{Q}}_c \rangle_{\mathrm{F}} \leftarrow \langle \tilde{\boldsymbol{P}}^\top \boldsymbol{A} \tilde{\boldsymbol{P}}, \tilde{\boldsymbol{Q}}^\top \boldsymbol{B} \tilde{\boldsymbol{Q}} \rangle_{\mathrm{F}}$
$L(\boldsymbol{\theta}), \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{P}_c, \boldsymbol{Q}_c, \hat{\boldsymbol{P}}_c, \hat{\boldsymbol{Q}}_c \leftarrow$
Alg. 2$(\boldsymbol{\theta}, \mathbb{U}, \mathbb{V}, \mathbb{O}, \tilde{\boldsymbol{P}}, \tilde{\boldsymbol{Q}}, \boldsymbol{A}, \boldsymbol{B}, \langle \tilde{\boldsymbol{P}}_c, \tilde{\boldsymbol{Q}}_c \rangle_{\mathrm{F}})$
repeat
  $\nabla L(\boldsymbol{\theta}) \leftarrow$ Alg. 1$(\boldsymbol{\theta}, \mathbb{O}, \tilde{\boldsymbol{P}}, \tilde{\boldsymbol{Q}}, \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{P}_c, \boldsymbol{Q}_c, \hat{\boldsymbol{P}}_c, \hat{\boldsymbol{Q}}_c)$
  $\boldsymbol{s} \leftarrow$ Alg. 3$(\boldsymbol{\theta}, \mathbb{O}, \nabla L(\boldsymbol{\theta}), \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{P}_c, \boldsymbol{Q}_c)$
  $\delta \leftarrow 1$
  while *line-search steps are within a limit* **do**
    $L(\boldsymbol{\theta} + \delta \boldsymbol{s}), \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{P}_c, \boldsymbol{Q}_c, \hat{\boldsymbol{P}}_c, \hat{\boldsymbol{Q}}_c \leftarrow$
    Alg. 2$(\boldsymbol{\theta} + \delta \boldsymbol{s}, \mathbb{U}, \mathbb{V}, \mathbb{O}, \tilde{\boldsymbol{P}}, \tilde{\boldsymbol{Q}}, \boldsymbol{A}, \boldsymbol{B}, \langle \tilde{\boldsymbol{P}}_c, \tilde{\boldsymbol{Q}}_c \rangle_{\mathrm{F}})$
    if $L(\boldsymbol{\theta} + \delta \boldsymbol{s}) \le L(\boldsymbol{\theta}) + \eta \delta \boldsymbol{s}^\top \nabla L(\boldsymbol{\theta})$ **then break**
    $\delta \leftarrow \delta / 2$
  end
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \delta \boldsymbol{s}$
until *stopping condition is satisfied*

Algorithm 2: Objective function evaluation: $L(\boldsymbol{\theta})$

Input: $\boldsymbol{\theta}, \mathbb{U}, \mathbb{V}, \mathbb{O}, \tilde{\boldsymbol{P}}, \tilde{\boldsymbol{Q}}, \boldsymbol{A}, \boldsymbol{B}, \langle \tilde{\boldsymbol{P}}_c, \tilde{\boldsymbol{Q}}_c \rangle_{\mathrm{F}}$.
$\boldsymbol{P} \leftarrow [\cdots f(\boldsymbol{\theta}^u; \boldsymbol{u}_i) \cdots]^\top, \boldsymbol{Q} \leftarrow [\cdots g(\boldsymbol{\theta}^v; \boldsymbol{v}_j) \cdots]^\top$
$\boldsymbol{P}_c \leftarrow \boldsymbol{P}^\top \boldsymbol{A} \boldsymbol{P}, \boldsymbol{Q}_c \leftarrow \boldsymbol{Q}^\top \boldsymbol{B} \boldsymbol{Q}$
$\hat{\boldsymbol{P}}_c \leftarrow \tilde{\boldsymbol{P}}^\top \boldsymbol{A} \boldsymbol{P}, \hat{\boldsymbol{Q}}_c \leftarrow \tilde{\boldsymbol{Q}}^\top \boldsymbol{B} \boldsymbol{Q}$
$L^+(\boldsymbol{\theta}) \leftarrow \sum_{(i,j) \in \mathbb{O}} \ell^+_{ij}(Y_{ij}, \boldsymbol{p}_i^\top \boldsymbol{q}_j)$
$L^-(\boldsymbol{\theta}) \leftarrow \frac{1}{2} \langle \tilde{\boldsymbol{P}}_c, \tilde{\boldsymbol{Q}}_c \rangle_{\mathrm{F}} - \langle \hat{\boldsymbol{P}}_c, \hat{\boldsymbol{Q}}_c \rangle_{\mathrm{F}} + \frac{1}{2} \langle \boldsymbol{P}_c, \boldsymbol{Q}_c \rangle_{\mathrm{F}}$
$L(\boldsymbol{\theta}) \leftarrow L^+(\boldsymbol{\theta}) + \omega L^-(\boldsymbol{\theta}) + \lambda \Psi(\boldsymbol{\theta})$
Output: $L(\boldsymbol{\theta}), \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{P}_c, \boldsymbol{Q}_c, \hat{\boldsymbol{P}}_c, \hat{\boldsymbol{Q}}_c$

Algorithm 3: Conjugate gradient (CG) procedures

Input: $\boldsymbol{\theta}, \mathbb{O}, \nabla L(\boldsymbol{\theta}), \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{P}_c, \boldsymbol{Q}_c$.
$\boldsymbol{r} \leftarrow -\nabla L(\boldsymbol{\theta}), \boldsymbol{d} \leftarrow \boldsymbol{r}, \boldsymbol{s} \leftarrow \boldsymbol{0}, \gamma \leftarrow \boldsymbol{r}^\top \boldsymbol{r}, \gamma_0 \leftarrow \gamma$
repeat
  $\boldsymbol{W} \leftarrow [\cdots \frac{\partial f^i}{\partial \boldsymbol{\theta}^u} \boldsymbol{d}^u \cdots]^\top, \boldsymbol{H} \leftarrow [\cdots \frac{\partial g^j}{\partial \boldsymbol{\theta}^v} \boldsymbol{d}^v \cdots]^\top$
  $\boldsymbol{W}_c \leftarrow \boldsymbol{W}^\top \boldsymbol{A} \boldsymbol{P}, \boldsymbol{H}_c \leftarrow \boldsymbol{H}^\top \boldsymbol{B} \boldsymbol{Q}$
  $Z_{ij} \leftarrow \frac{\partial^2 \ell^+_{ij}}{\partial \hat{Y}^2_{ij}} (\boldsymbol{w}_i^\top \boldsymbol{q}_j + \boldsymbol{p}_i^\top \boldsymbol{h}_j), \forall (i,j) \in \mathbb{O}$
  $\boldsymbol{Z}_q \leftarrow \boldsymbol{Z} \boldsymbol{Q}, \boldsymbol{Z}_p \leftarrow \boldsymbol{Z}^\top \boldsymbol{P}$
  $\boldsymbol{G}\boldsymbol{d} \leftarrow \begin{bmatrix} \langle \mathbf{J}^u, \boldsymbol{Z}_q + \omega(\boldsymbol{A}\boldsymbol{W}\boldsymbol{Q}_c + \boldsymbol{A}\boldsymbol{P}\boldsymbol{H}_c) \rangle \\ \langle \mathbf{J}^v, \boldsymbol{Z}_p + \omega(\boldsymbol{B}\boldsymbol{H}\boldsymbol{P}_c + \boldsymbol{B}\boldsymbol{Q}\boldsymbol{W}_c) \rangle \end{bmatrix} + \lambda \nabla^2 \Psi(\boldsymbol{\theta}) \boldsymbol{d}$
  $t \leftarrow \gamma / \boldsymbol{d}^\top \boldsymbol{G}\boldsymbol{d}$
  $\boldsymbol{s} \leftarrow \boldsymbol{s} + t\boldsymbol{d}$
  $\boldsymbol{r} \leftarrow \boldsymbol{r} - t\boldsymbol{G}\boldsymbol{d}$
  $\gamma_{\mathrm{new}} = \boldsymbol{r}^\top \boldsymbol{r}$
  $\beta \leftarrow \gamma_{\mathrm{new}} / \gamma$
  $\boldsymbol{d} \leftarrow \boldsymbol{r} + \beta \boldsymbol{d}$
  $\gamma \leftarrow \gamma_{\mathrm{new}}$
until $\sqrt{\gamma} \le \xi \sqrt{\gamma_0}$ *or the max number of CG steps is reached*
Output: $\boldsymbol{s}$

A.3 Detailed Derivation of the SOGram Method

We discuss the Stochastic Online Gramian (SOGram) method proposed in [13], which is very related to the method discussed in Section A.2. It considers a special case of the objective function in (54) by excluding $\bar{a}_i$ and $\bar{b}_j$. From this we show that SOGram is an extension.

In [13], they view matrices in (61) as estimates of matrices in (14). The main idea behind SOGram is to apply a variance reduction scheme to improve these estimates. Specifically, they maintain four zero-initialized matrices with the following exponentially averaged updates before each step

$$\begin{aligned} \boldsymbol{P}'_c &\leftarrow (1-\alpha)\boldsymbol{P}'_c + \alpha \boldsymbol{P}_2, \quad \boldsymbol{Q}'_c \leftarrow (1-\alpha)\boldsymbol{Q}'_c + \alpha \boldsymbol{Q}_2, \\ \hat{\boldsymbol{P}}'_c &\leftarrow (1-\alpha)\hat{\boldsymbol{P}}'_c + \alpha \hat{\boldsymbol{P}}_2, \quad \hat{\boldsymbol{Q}}'_c \leftarrow (1-\alpha)\hat{\boldsymbol{Q}}'_c + \alpha \hat{\boldsymbol{Q}}_2, \end{aligned} \tag{64}$$

where α is a hyper-parameter for controlling the bias-variance tradeoff. Then they replace (60) with

$$\frac{|\mathbb{O}|}{|\hat{\mathbb{O}}_1|} \begin{bmatrix} \sum_{(i,j) \in \hat{\mathbb{O}}_1} \frac{\partial f^i}{\partial \boldsymbol{\theta}^u}^\top \left(\boldsymbol{q}_j X_{ij} + \omega(\boldsymbol{Q}'_c \bar{a}_i \boldsymbol{p}_i - \hat{\boldsymbol{Q}}'^\top_c \bar{a}_i \tilde{\boldsymbol{p}}_i) \right) \\ \sum_{(i,j) \in \hat{\mathbb{O}}_1} \frac{\partial g^j}{\partial \boldsymbol{\theta}^v}^\top \left(\boldsymbol{p}_i X_{ij} + \omega(\boldsymbol{P}'_c \bar{b}_j \boldsymbol{q}_j - \hat{\boldsymbol{P}}'^\top_c \bar{b}_j \tilde{\boldsymbol{q}}_j) \right) \end{bmatrix} + \lambda \nabla \Psi(\boldsymbol{\theta}), \tag{65}$$

which is considered as the subsampled gradient vector for each SG step.

A.4 Detailed Procedures of Gauss-Newton Method

Details of the CG method involving a sequence of $\boldsymbol{G}\boldsymbol{d}$ operations are given in Algorithm 3. The overall procedures of the Gauss-Newton method are summarized in Algorithm 4.