

# On Sampling Strategies for Neural Network-based Collaborative Filtering

Ting Chen

University of California, Los Angeles  
Los Angeles, CA 90095  
tingchen@cs.ucla.edu

Yue Shi\*

Yahoo! Research  
Sunnyvale, CA 94089  
yueshi@acm.org

Yizhou Sun

University of California, Los Angeles  
Los Angeles, CA 90095  
yzsun@cs.ucla.edu

Liangjie Hong

Etsy Inc.  
Brooklyn, NY 11201  
lhong@etsy.com

## ABSTRACT

Recent advances in neural networks have inspired people to design hybrid recommendation algorithms that can incorporate both (1) user-item interaction information and (2) content information including image, audio, and text. Despite their promising results, neural network-based recommendation algorithms pose extensive computational costs, making it challenging to scale and improve upon. In this paper, we propose a general neural network-based recommendation framework, which subsumes several existing state-of-the-art recommendation algorithms, and address the efficiency issue by investigating sampling strategies in the stochastic gradient descent training for the framework. **We tackle this issue by first establishing a connection between the loss functions and the user-item interaction bipartite graph, where the loss function terms are defined on links while major computation burdens are located at nodes.** We call this type of loss functions “graph-based” loss functions, for which varied mini-batch sampling strategies can have different computational costs. **Based on the insight, three novel sampling strategies are proposed, which can significantly improve the training efficiency of the proposed framework (up to  $\times 30$  times speedup in our experiments), as well as improving the recommendation performance.** Theoretical analysis is also provided for both the computational cost and the convergence. We believe the study of sampling strategies have further implications on general graph-based loss functions, and would also enable more research under the neural network-based recommendation framework.

## ACM Reference format:

Ting Chen, Yizhou Sun, Yue Shi, and Liangjie Hong. 2017. On Sampling Strategies for Neural Network-based Collaborative Filtering. In *Proceedings of KDD '17, Halifax, NS, Canada, August 13-17, 2017*, 10 pages. <https://doi.org/10.1145/3097983.3098202>

\*Now at Facebook.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

KDD '17, August 13-17, 2017, Halifax, NS, Canada

© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.

ACM ISBN 978-1-4503-4887-4/17/08...\$15.00

<https://doi.org/10.1145/3097983.3098202>

## 1 INTRODUCTION

Collaborative Filtering (CF) has been one of the most effective methods in recommender systems, and methods like matrix factorization [17, 18, 27] are widely adopted. However, one of its limitation is the dealing of “cold-start” problem, where there are few or no observed interactions for new users or items, such as in news recommendation. To overcome this problem, hybrid methods are proposed to incorporate side information [7, 25, 28], or item content information [11, 31] into the recommendation algorithm. Although these methods can deal with side information to some extent, they are not effective for extracting features in complicated data, such as image, audio and text. On the contrary, deep neural networks have been shown very powerful at extracting complicated features from those data automatically [15, 19]. Hence, it is natural to combine deep learning with traditional collaborative filtering for recommendation tasks, as seen in recent studies [1, 4, 32, 37].

In this work, we generalize several state-of-the-art neural network-based recommendation algorithms [1, 4, 30], and propose a more general framework that combines both collaborative filtering and deep neural networks in a unified fashion. The framework inherits the best of two worlds: **(1) the power of collaborative filtering at capturing user preference via their interaction with items, and (2) that of deep neural networks at automatically extracting high-level features from content data.** However, it also comes with a price. Traditional CF methods, such as sparse matrix factorization [17, 27], are usually fast to train, while the deep neural networks in general are much more computationally expensive [19]. Combining these two models in a new recommendation framework can easily increase computational cost by hundreds of times, thus require a new design of the training algorithm to make it more efficient.

We tackle the computational challenges by first establishing a connection between the loss functions and the user-item interaction bipartite graph. We realize the key issue when combining the CF and deep neural networks are in: the loss function terms are defined over the links, and thus sampling is on links for the stochastic gradient training, while the main computational burdens are located at nodes (e.g., Convolutional Neural Network computation for image of an item). For this type of loss functions, varied mini-batch sampling strategies can lead to different computational costs, depending on how many node computations are required in a mini-batch. The existing stochastic sampling techniques, such as IID sampling, are

inefficient, as they do not take into account the node computations that can be potentially shared across links/data points.

Inspired by the connection established, we propose three novel sampling strategies for the general framework that can take coupled computation costs across user-item interactions into consideration. The first strategy is **Stratified Sampling**, which try to amortize costly node computation by partitioning the links into different groups based on nodes (called stratum), and sample links based on these groups. The second strategy is **Negative Sharing**, which is based on the observation that interaction/link computation is fast, so once a mini-batch of user-item tuples are sampled, we share the nodes for more links by creating additional negative links between nodes in the same batch. Both strategies have their pros and cons, and to keep their advantages while avoid their weakness, we form the third strategy by combining the above two strategies. Theoretical analysis of computational cost and convergence is also provided. Our contributions can be summarized as follows.

- We propose a general hybrid recommendation framework (Neural Network-based Collaborative Filtering) combining CF and content-based methods with deep neural networks, which generalize several state-of-the-art approaches.
- We establish a connection between the loss functions and the user-item interaction graph, based on which, we propose sampling strategies that can significantly improve training efficiency (up to  $\times 30$  times faster in our experiments) as well as the recommendation performance of the proposed framework.
- We provide both theoretical analysis and empirical experiments to demonstrate the superiority of the proposed methods.

## 2 A GENERAL FRAMEWORK FOR NEURAL NETWORK-BASED COLLABORATIVE FILTERING

In this section, we propose a general framework for neural network-based Collaborative Filtering that incorporates both interaction and content information.

### 2.1 Text Recommendation Problem

In this work, we use the text recommendation task [1, 4, 31, 32] as an illustrative application for the proposed framework. However, the proposed framework can be applied to more scenarios such as music and video recommendations.

We use  $\mathbf{x}_u$  and  $\mathbf{x}_v$  to denote features of user  $u$  and item  $v$ , respectively. In text recommendation setting, we set  $\mathbf{x}_u$  to one-hot vector indicating  $u$ 's user id (i.e. a binary vector with only one at the  $u$ -th position)<sup>1</sup>, and  $\mathbf{x}_v$  as the text sequence, i.e.  $\mathbf{x}_v = (w_1, w_2, \dots, w_t)$ . A response matrix  $\tilde{\mathbf{R}}$  is used to denote the historical interactions between users and articles, where  $\tilde{r}_{uv}$  indicates interaction between a user  $u$  and an article  $v$ , such as "click-or-not" and "like-or-not". Furthermore, we consider  $\tilde{\mathbf{R}}$  as implicit feedback in this work, which means only positive interactions are provided, and non-interactions are treated as negative feedback implicitly.

<sup>1</sup>Other user profile features can be included, if available.

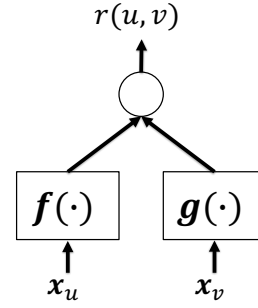


Figure 1: The functional embedding framework.

Given user/item features  $\{\mathbf{x}_u\}, \{\mathbf{x}_v\}$  and their historical interaction  $\tilde{\mathbf{R}}$ , the goal is to learn a model which can rank new articles for an existing user  $u$  based on this user's interests and an article's text content.

### 2.2 Functional Embedding

In most of existing matrix factorization techniques [17, 18, 27], each user/item ID is associated with a latent vector  $\mathbf{u}$  or  $\mathbf{v}$  (i.e., embedding), which can be considered as a simple linear transformation from the one-hot vector represented by their IDs, i.e.  $\mathbf{u}_u = \mathbf{f}(\mathbf{x}_u) = \mathbf{W}^T \mathbf{x}_u$  ( $\mathbf{W}$  is the embedding/weight matrix). Although simple, this direct association of user/item ID with representation make it less flexible and unable to incorporate features such as text and image.

In order to effectively incorporate user and item features such as content information, it has been proposed to replace embedding vectors  $\mathbf{u}$  or  $\mathbf{v}$  with functions such as decision trees [38] and some specific neural networks [1, 4]. Generalizing the existing work, we propose to replace the original embedding vectors  $\mathbf{u}$  and  $\mathbf{v}$  with general differentiable functions  $\mathbf{f}(\cdot) \in \mathbb{R}^d$  and  $\mathbf{g}(\cdot) \in \mathbb{R}^d$  that take user/item features  $\mathbf{x}_u, \mathbf{x}_v$  as their inputs. Since the user/item embeddings are the output vectors of functions, we call this approach *Functional Embedding*. After embeddings are computed, a score function  $r(u, v)$  can be defined based on these embeddings for a user/item pair  $(u, v)$ , such as vector dot product  $r(u, v) = \mathbf{f}(\mathbf{x}_u)^T \mathbf{g}(\mathbf{x}_v)$  (used in this work), or a general neural network. The model framework is shown in Figure 1. It is easy to see that our framework is very general, as it does not explicitly specify the feature extraction functions, as long as the functions are differentiable. In practice, these function can be specified with neural networks such as CNN or RNN, for extracting high-level information from image, audio, or text sequence. When there are no features associated, it degenerates to conventional matrix factorization where user/item IDs are used as their features.

For simplicity, we will denote the output of  $\mathbf{f}(\mathbf{x}_u)$  and  $\mathbf{g}(\mathbf{x}_v)$  by  $\mathbf{f}_u$  and  $\mathbf{g}_v$ , which are the embedding vectors for user  $u$  and item  $v$ .

### 2.3 Loss Functions for Implicit Feedback

In many real-world applications, users only provide positive signals according to their preferences, while negative signals are usually implicit. This is usually referred as "implicit feedback" [13, 23, 26].

**Table 1: Examples of loss functions for recommendation.**

Pointwise loss
SG-loss [22]: $-\sum_{(u,v) \in \mathcal{D}} \left( \log \sigma(\mathbf{f}_u^T \mathbf{g}_v) + \lambda \mathbb{E}_{v' \sim P_n} \log \sigma(-\mathbf{f}_u^T \mathbf{g}_{v'}) \right)$
MSE-loss [30]: $\sum_{(u,v) \in \mathcal{D}} \left( (\tilde{r}_{uv}^+ - \mathbf{f}_u^T \mathbf{g}_v)^2 + \lambda \mathbb{E}_{v' \sim P_n} (\tilde{r}_{uv'}^- - \mathbf{f}_u^T \mathbf{g}_{v'})^2 \right)$
Pairwise loss
Log-loss [26]: $-\sum_{(u,v) \in \mathcal{D}} \mathbb{E}_{v' \sim P_n} \log \sigma \left( \gamma(\mathbf{f}_u^T \mathbf{g}_v - \mathbf{f}_u^T \mathbf{g}_{v'}) \right)$
Hinge-loss [33]: $\sum_{(u,v) \in \mathcal{D}} \mathbb{E}_{v' \sim P_n} \max \left( \mathbf{f}_u^T \mathbf{g}_{v'} - \mathbf{f}_u^T \mathbf{g}_v + \gamma, 0 \right)$

In this work, we consider two types of loss functions that can handle recommendation tasks with implicit feedback, namely, pointwise loss functions and pairwise loss functions. Pointwise loss functions have been applied to such problems in many existing work. In [1, 30, 32], mean square loss (MSE) has been applied where “negative terms” are weighted less. And skip-gram (SG) loss has been successfully utilized to learn robust word embedding [22].

These two loss functions are summarized in Table 1. Note that we use a weighted expectation term over all negative samples, which can be approximated with small number of samples. We can also abstract the pointwise loss functions into the following form:

$$\mathcal{L}_{\text{pointwise}} = \mathbb{E}_{u \sim P_d(u)} \left[ \mathbb{E}_{v \sim P_d(v|u)} c_{uv}^+ \mathcal{L}^+(u, v|\theta) + \mathbb{E}_{v' \sim P_n(v')} c_{uv'}^- \mathcal{L}^-(u, v'|\theta) \right] \quad (1)$$

where  $P_d$  is (empirical) data distribution,  $P_n$  is user-defined negative data distribution,  $c$  is user defined weights for the different user-item pairs,  $\theta$  denotes the set of all parameters,  $\mathcal{L}^+(u, v|\theta)$  denotes the loss function on a single positive pair  $(u, v)$ , and  $\mathcal{L}^-(u, v|\theta)$  denotes the loss on a single negative pair. Generally speaking, given a user  $u$ , pointwise loss function encourages her score with positive items  $\{v\}$ , and discourage her score with negative items  $\{v'\}$ .

When it comes to ranking problem as commonly seen in implicit feedback setting, some have argued that the pairwise loss would be advantageous [26, 33], as pairwise loss encourages ranking of positive items above negative items for the given user. Different from pointwise counterparts, pairwise loss functions are defined on a triplet of  $(u, v, v')$ , where  $v$  is a positive item and  $v'$  is a negative item to the user  $u$ . Table 1 also gives two instances of such loss functions used in existing papers [26, 33] (with  $\gamma$  being the pre-defined “margin” parameter). We can also abstract pairwise loss functions by the following form:

$$\mathcal{L}_{\text{pairwise}} = \mathbb{E}_{u \sim P_d(u)} \mathbb{E}_{v \sim P_d(v|u)} \mathbb{E}_{v' \sim P_n(v')} c_{uvv'} \mathcal{L}(u, v, v'|\theta) \quad (2)$$

where the notations are similarly defined as in Eq. 1 and  $\mathcal{L}(u, v, v'|\theta)$  denotes the loss function on the triplet  $(u, v, v')$ .

## 2.4 Stochastic Gradient Descent Training and Computational Challenges

To train the model, we use stochastic gradient descent based algorithms [3, 16], which are widely used for training matrix factorization and neural networks. The main flow of the training algorithm

**Algorithm 1** Standard model training procedure

---

```

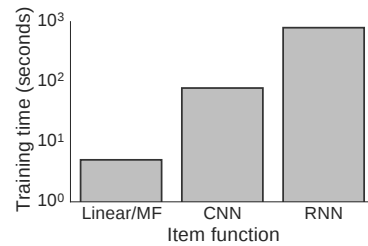
while not converged do
  // mini-batch sampling
  draw a mini-batch of user-item tuples  $(u, v)^2$ 
  // forward pass
  compute  $\mathbf{f}(\mathbf{x}_u)$ ,  $\mathbf{g}(\mathbf{x}_v)$  and their interaction  $\mathbf{f}_u^T \mathbf{g}_v$ 
  compute the loss function  $\mathcal{L}$ 
  // backward pass
  compute gradients and apply SGD updates
end while

```

---

is summarized in Algorithm 1. By adopting the functional embedding with (deep) neural networks, we can increase the power of the model, but it also comes with a cost. Figure 2 shows the training time (for CiteULike data) with different item functions  $\mathbf{g}(\cdot)$ , namely linear embedding taking item id as feature (equivalent to conventional MF), CNN-based content embedding, and RNN/LSTM-based content embedding. We see orders of magnitude increase of training time for the latter two embedding functions, which may create barriers to adopt models under this framework.

Breaking down the computation cost of the framework, there are three major parts of computational cost. The first part is the user based computation (denoted by  $t_f$  time units per user), which includes forward computation of user function  $\mathbf{f}(\mathbf{x}_u)$ , and backward computation of the function output w.r.t. its parameters. The second part is the item based computation (denoted by  $t_g$  time units per item), which similarly includes forward computation of item function  $\mathbf{g}(\mathbf{x}_v)$ , as well as the back computation. The third part is the computation for interaction function (denoted by  $t_i$  time units per interaction). The total computational cost for a mini-batch is then  $t_f \times \#$  of users +  $t_g \times \#$  of items +  $t_i \times \#$  of interactions, with some other minor operations which we assume ignorable. In the text recommendation application, user IDs are used as user features (which can be seen as linear layer on top of the one-hot inputs), (deep) neural networks are used for text sequences, vector dot product is used as interaction function, thus the dominant computational cost is  $t_g$  (orders of magnitude larger than  $t_f$  and  $t_i$ ). In other words, we assume  $t_g \gg t_f, t_i$  in this work.

**Figure 2: Model training time per epoch with different types of item functions (in log-scale).**

<sup>2</sup>Draw a mini-batch of user-item triplets  $(u, v, v')$  if a pairwise loss function is adopted.

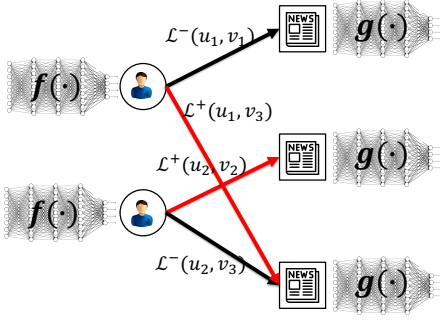


Figure 3: The bipartite interaction graph for pointwise loss functions, where loss functions are defined over links. The pairwise loss functions are defined over pairs of links.

### 3 MINI-BATCH SAMPLING STRATEGIES FOR EFFICIENT MODEL TRAINING

In this section, we propose and discuss different sampling strategies that can improve the efficiency of the model training.

#### 3.1 Computational Cost in a Graph View

Before the discussion of different sampling strategies, we motivate our readers by first making a connection between the loss functions and the bipartite graph of user-item interactions. In the loss functions laid out before, we observed that each loss function term in Eq. 1, namely,  $\mathcal{L}(u, v)$ , involves a pair of user and item, which corresponds to a link in their interaction graph. And two types of links corresponding to two types of loss terms in the loss functions, i.e., positive links/terms and negative links/terms. Similar analysis holds for pairwise loss in Eq. 2, though there are slight differences as each single loss function corresponds to a pair of links with opposite signs on the graph. We can also establish a correspondence between user/item functions and nodes in the graph, i.e.,  $f(u)$  to user node  $u$  and  $g(v)$  to item node  $v$ . The connection is illustrated in Figure 3. Since the loss functions are defined over the links, we name them “graph-based” loss functions to emphasize the connection.

The key observation for graph-based loss functions is that: the loss functions are defined over links, but the major computational burden are located at nodes (due to the use of costly  $g(\cdot)$  function). Since each node is associated with multiple links, which are corresponding to multiple loss function terms, the computational costs of loss functions over links are coupled (as they may share the same nodes) when using mini-batch based SGD. Hence, varied sampling strategies yield different computational costs. For example, when we put links connected to the same node together in a mini-batch, the computational cost can be lowered as there are fewer  $g(\cdot)$  to compute<sup>3</sup>. This is in great contrast to conventional optimization problems, where each loss function term dose not couple with others in terms of computation cost.

<sup>3</sup>This holds for both forward and backward computation. For the latter, the gradient from different links can be aggregated before back-propagating to  $g(\cdot)$ .

#### 3.2 Existing Mini-Batch Sampling Strategies

In standard SGD sampler, (positive) data samples are drawn uniformly at random for gradient computation. Due to the appearance of negative samples, we draw negative samples from some predefined probability distribution, i.e.  $(u', v') \sim P_n(u', v')$ . We call this approach “IID Sampling”, since each positive link is dependently and identical distributed, and the same holds for negative links (with a different distribution).

Many existing algorithms with graph-based loss functions [1, 22, 29] adopt the “Negative Sampling” strategy, in which  $k$  negative samples are drawn whenever a positive example is drawn. The negative samples are sampled based on the positive ones by replacing the items in the positive samples. This is illustrated in Algorithm 2 and Figure 4(a).

---

##### Algorithm 2 Negative Sampling [1, 21, 29]

---

**Require:** number of positive links in a mini-batch  $b$ , number of negative links per positive one:  $k$   
draw  $b$  positive links uniformly at random  
**for** each of  $b$  positive links **do**  
draw  $k$  negative links by replacing true item  $v$  with  $v' \propto P_n(v')$   
**end for**

---

The IID Sampling strategy dose not take into account the property of graph-based loss functions, since samples are completely independent of each other. Hence, the computational cost in a single mini-batch cannot be amortized across different samples, leading to very extensive computations with (deep) neural networks. The Negative Sampling does not really help, since the item function computation cost  $t_g$  is the dominant one. To be more specific, consider a mini-batch with  $b(1 + k)$  links sampled by IID Sampling or Negative Sampling, we have to conduct item based  $g(\cdot)$  computation  $b(1 + k)$  times, since items in a mini-batch are likely to be non-overlapping with sufficient large item sets.

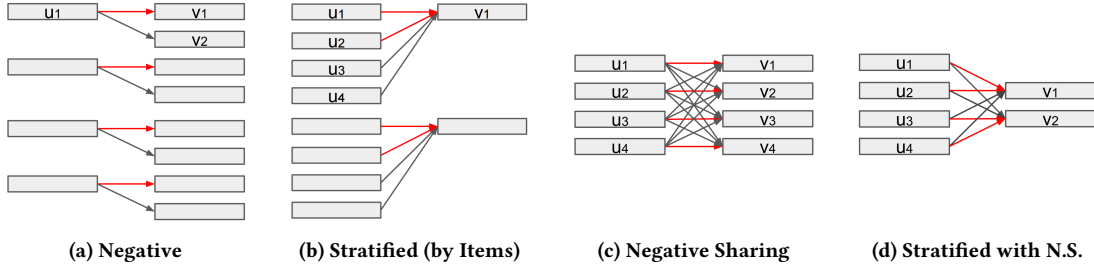
#### 3.3 The Proposed Sampling Strategies

**3.3.1 Stratified Sampling (by Items).** Motivated by the connection between the loss functions and the bipartite interaction graph as shown in Figure 3, we propose to sample links that share nodes, in particular those with high computational cost (i.e.  $t_g$  for item function  $g(\cdot)$  in our case). By doing so, the computational cost within a mini-batch can be amortized, since fewer costly functions are computed (in both forward and backward propagations).

In order to achieve this, we (conceptually) partition the links, which correspond to loss function terms, into *strata*. A *stratum* in the strata is a set of links on the bipartite graph sharing the same source or destination node. Instead of drawing links directly for training, we will first draw stratum and then draw both positive and negative links. Since we want each stratum to share the same item, we can directly draw an item and then sample its links. The details are given in Algorithm 3 and illustrated in Figure 4(b).

Compared to Negative Sampling in Algorithm 2, there are several differences: (1) Stratified Sampling can be based on either item or user, but in the negative sampling only negative items are drawn; and (2) each node in stratified sampling can be associated with





**Figure 4: Illustration of four different sampling strategies. 4(b)-4(d) are the proposed sampling strategies. Red lines denote positive links/interactions, and black lines denote negative links/interactions.**

---

**Algorithm 3** Stratified Sampling (by Items)

---

**Require:** number of positive links in a mini-batch:  $b$ , number of positive links per stratum:  $s$ , number of negative links per positive one:  $k$

**repeat**

draw an item  $v \propto P_d(v)$

draw  $s$  positive users  $\{u\}$  of  $v$  uniformly at random

draw  $k \times s$  negative users  $\{u'\} \propto P_d(u')$

**until** a mini-batch of  $b$  positive links are sampled

---

more than 1 positive link (i.e.,  $s > 1$ , which can help improve the speedup as shown below), while in negative sampling each node is only associated with one positive link.

Now we consider its speedup for a mini-batch including  $b$  positive links/interactions and  $bk$  negative ones, which contains  $b(1+k)$  users and  $b/s$  items. The Stratified Sampling (by Items) only requires  $b/s$  computations of  $g(\cdot)$  functions, while the Negative Sampling requires  $b(1+k)$  computations. Assuming  $t_g \gg t_f, t_i$ , i.e. the computation cost is dominated by the item function  $g(\cdot)$ , the Stratified Sampling (by Items) can provide  $s(1+k)$  times speedup in a mini-batch. With  $s = 4, k = 10$  as used in some of our experiments, it yields to  $\times 40$  speedup optimally. However, it is worth pointing out that item-based Stratified Sampling cannot be applied to pairwise loss functions, which compare preferences over items based on a given user.

**3.3.2 Negative Sharing.** The idea of Negative Sharing is inspired from a different aspect of the connection between the loss functions and the bipartite interaction graph. Since  $t_i \ll t_g$ , i.e. the computational cost of interaction function (dot product) is ignorable compared to that of item function, when a mini-batch of users and items are sampled, increasing the number of interactions among them may not result in a significant increase of computational cost. This can be achieved by creating a complete bipartite graph for a mini-batch by adding negative links between all non-interaction pairs between users and items. Using this strategy, we can draw NO negative links at all!

More specifically, consider the IID Sampling, when  $b$  positive links are sampled, there will be  $b$  users and  $b$  items involved (assuming the sizes of user set and item set are much larger than  $b$ ). Note that, there are  $b(b-1)$  non-interactions in the mini-batch, which are not considered in IID Sampling or Negative Sampling, instead they

---

**Algorithm 4** Negative Sharing

---

**Require:** number of positive links in a mini-batch:  $b$

draw  $b$  positive user-item pairs  $\{(u, v)\}$  uniformly at random  
construct negative pairs by connecting non-linked users and items in the batch

---

draw additional negative samples. Since the main computational cost of training is on the node computation and the node set is fixed given the batch of  $b$  positive links, we can share the nodes for negative links without increasing much of computational burdens. Based on this idea, Algorithm 4 summarizes an extremely simple sampling procedure, and it is illustrated in Figure 4(c).

Since Negative Sharing avoids sampling  $k$  negative links, it only contains  $b$  items while in Negative Sampling contains  $b(1+k)$  items. So it can provide  $(1+k)$  times speedup compared to Negative Sampling (assuming  $t_g \gg t_f, t_i$ , and total interaction cost is still insignificant). Given the batch size  $b$  is usually larger than  $k$  (e.g.,  $b = 512, k = 20$  in our experiments), much more negative links (e.g.  $512 \times 511$ ) will also be considered, this is helpful for both faster convergence and better performance, which is shown in our experiments. However, as the number of negative samples increases, the performance and the convergence will not be improved linearly. diminishing return is expected.

**3.3.3 Stratified Sampling with Negative Sharing.** The two strategies above can both reduce the computational cost by smarter sampling of the mini-batch. However, they both have weakness: Stratified Sampling cannot deal with pairwise loss and it is still dependent on the number of negative examples  $k$ , and Negative Sharing introduces a lot of negative samples which may be unnecessary due to diminishing return.

The good news is, the two sampling strategies are proposed from different perspectives, and combining them together can preserve their advantages while avoid their weakness. This leads to the Stratified Sampling with Negative Sharing, which can be applied to both pointwise and pairwise loss functions, and it can have flexible ratio between positive and negative samples (i.e. more positive links given the same negative links compared to Negative Sharing). To do so, basically we sample positive links according to Stratified Sampling, and then sample/create negative links by treating non-interactions as negative links. The details are given in Algorithm 5 and illustrated in Figure 4(d).

**Algorithm 5** Stratified Sampling with Negative Sharing

---

**Require:** number of positive links in a mini-batch:  $b$ , number of positive links per stratum:  $s$   
**repeat**  
    draw an item  $v \propto P_d(v)$   
    draw  $s$  positive users of item  $v$  uniformly at random  
**until** a mini-batch of  $b/s$  items are sampled  
construct negative pairs by connecting non-linked users and items in the batch

---

Computationally, Stratified Sampling with Negative Sharing only involve  $b/s$  item nodes in a mini-batch, so it can provide the same  $s(1+k)$  times speedup over Negative Sampling as Stratified Sampling (by Items) does, but it will utilize much more negative links compared to Negative Sampling. For example, in our experiments with  $b = 512, s = 4$ , we have 127 negative links per positive one, much larger than  $k = 10$  in Negative Sampling, and only requires  $1/4$  times of  $g(\cdot)$  computations compared to Negative Sharing.

**3.3.4 Implementation Details.** When the negative/noise distribution  $P_n$  is not unigram<sup>4</sup>, we need to adjust the loss function in order to make sure the stochastic gradient is unbiased. For pointwise loss, each of the negative term is adjusted by multiplying a weight of  $\frac{P_n(v')}{P_d(v')}$ ; for pairwise loss, each term based on a triplet of  $(u, v, v')$  is adjusted by multiplying a weight of  $\frac{P_n(v')}{P_d(v')}$  where  $v'$  is the sampled negative item.

Instead of sampling, we prefer to use shuffling as much as we can, which produces unbiased samples while yielding zero variance. This can be a useful trick for achieving better performance when the number of drawn samples are not large enough for each loss terms. For IID and Negative Sampling, this can be easily done for positive links by simply shuffling them. As for the Stratified Sampling (w./wo. Negative Sharing), instead of shuffling the positive links directly, we shuffle the randomly formed strata (where each stratum contains roughly a single item)<sup>5</sup>. All other necessary sampling operations required are sampling from discrete distributions, which can be done in  $O(1)$  with Alias method.

In Negative Sharing (w./wo. Stratified Sampling), We can compute the user-item interactions with more efficient operator, i.e. replacing the vector dot product between each pair of  $(f, g)$  with matrix multiplication between  $(F, G)$ , where  $F = [f_{u_1}, \dots, f_{u_n}]$ ,  $G = [g_{v_1}, \dots, g_{v_m}]$ . Since matrix multiplication is higher in BLAS level than vector multiplication [14], even we increase the number of interactions, with medium matrix size (e.g.  $1000 \times 1000$ ) it does not affect the computational cost much in practice.

### 3.4 Computational Cost and Convergence Analysis

Here we provide a summary for the computational cost for different sampling strategies discussed above, and also analyze their convergences. Two aspects that can lead to speedup are analyzed: (1) the

<sup>4</sup>Unigram means proportional to item frequency, such as node degree in user-item interaction graph.

<sup>5</sup>This can be done by first shuffling users associated with each item, and then concatenating all links according to items in random order, random strata is then formed by segmenting the list.

computational cost for a mini-batch, i.e. per iteration, and (2) the number of iterations required to reach some referenced loss.

**3.4.1 Computational Cost.** To fairly compare different sampling strategies, we fix the same number of positive links in each of the mini-batch, which correspond to the positive terms in the loss function. Table 2 shows the computational cost of different sampling strategies for a given mini-batch. Since  $t_g \gg t_f, t_i$  in practice, we approximate the theoretical speedup per iteration by comparing the number of  $t_g$  computation. We can see that the proposed sampling strategies can provide  $(1+k)$ , by Negative Sharing, or  $s(1+k)$ , by Stratified Sampling (w./w.o. Negative Sharing), times speedup for each iteration compared to IID Sampling or Negative Sampling. As for the number of iterations to reach a reference loss, it is related to number of negative samples utilized, which is analyzed below.

**3.4.2 Convergence Analysis.** We want to make sure the SGD training under the proposed sampling strategies can converge correctly. The necessary condition for this to hold is the stochastic gradient estimator has to be unbiased, which leads us to the following lemma.

**LEMMA 1.** (*unbiased stochastic gradient*) Under sampling Algorithm 2, 3, 4, and 5, we have  $\mathbb{E}_B[\nabla \mathcal{L}_B(\theta^t)] = \nabla \mathcal{L}(\theta^t)$ . In other words, the stochastic mini-batch gradient equals to true gradient in expectation.

This holds for both pointwise loss and pairwise loss. It is guaranteed since we draw samples stochastically and re-weight certain samples accordingly. The detailed proof can be found in the supplementary material.

Given this lemma, we can further analyze the convergence behavior of the proposed sampling behaviors. Due to the highly non-linear and non-convex functions composed by (deep) neural networks, the convergence rate is usually difficult to analyze. So we show the SGD with the proposed sampling strategies follow a local convergence bound (similar to [10, 24]).

**PROPOSITION 1.** (*local convergence*) Suppose  $\mathcal{L}$  has  $\sigma$ -bounded gradient; let  $\eta_t = \eta = c/\sqrt{T}$  where  $c = \sqrt{\frac{2(\mathcal{L}(\theta^0) - \mathcal{L}(\theta^*))}{L\sigma^2}}$ , and  $\theta^*$  is the minimizer to  $\mathcal{L}$ . Then, the following holds for the proposed sampling strategies given in Algorithm 2, 3, 4, 5

$$\min_{0 \leq t \leq T-1} \mathbb{E}[\|\nabla \mathcal{L}(\theta^t)\|^2] \leq \sqrt{\frac{2(\mathcal{L}(\theta^0) - \mathcal{L}(\theta^*))}{T}} \sigma$$

The detailed proof is also given in the supplementary material.

Furthermore, utilizing more negative links in each mini-batch can lower the expected stochastic gradient variance. As shown in [35, 36], the reduction of variance can lead to faster convergence. This suggests that Negative Sharing (w./wo. Stratified Sampling) has better convergence than the Stratified Sampling (by Items).

## 4 EXPERIMENTS

### 4.1 Data Sets

Two real-world text recommendation data sets are used for the experiments. The first data set CiteULike, collected from CiteULike.org, is provided in [31]. The CiteULike data set contains users bookmarking papers, where each paper is associated with a title

**Table 2: Computational cost analysis for a batch of  $b$  positive links. We use  $\text{vec}$  to denote vector multiplication, and  $\text{mat}$  to denote matrix multiplication. Since  $t_g \gg t_f, t_i$  in practice, the theoretical speedup per iteration can be approximated by comparing the number of  $t_g$  computation, which is colored red below. The number of iterations to reach a referenced loss is related to the number of negative links in each mini-batch.**

Sampling	# pos. links	# neg. links	# $t_f$	# $t_g$	# $t_i$	pointwise	pairwise
IID [3]	$b$	$bk$	$b(1+k)$	$b(1+k)$	$b(1+k)$ vec	✓	×
Negative [1, 21, 29]	$b$	$bk$	$b$	$b(1+k)$	$b(1+k)$ vec	✓	✓
Stratified (by Items)	$b$	$bk$	$b(1+k)$	$\frac{b}{s}$	$b(1+k)$ vec	✓	×
Negative Sharing	$b$	$b(b-1)$	$b$	$\frac{b}{s}$	$b \times b$ mat	✓	✓
Stratified with N.S.	$b$	$\frac{b(b-1)}{s}$	$b$	$\frac{b}{s}$	$b \times \frac{b}{s}$ mat	✓	✓

and an abstract. The second data set is a random subset of Yahoo! News data set<sup>6</sup>, which contains users clicking on news presented at Yahoo!. There are 5,551 users and 16,980 items, and total of 204,986 positive interactions in CiteULike data. As for Yahoo! News data, there are 10,000 users, 58,579 items and 515,503 interactions.

Following [4], we select a portion (20%) of items to form the pool of test items. All user interactions with those test items are held-out during training, only the remaining user-item interactions are used as training data, which simulates the scenarios for recommending newly-emerged text articles.

## 4.2 Experimental Settings

The main purpose of experiments is to compare the efficiency and effectiveness of our proposed sampling strategies against existing ones. So we mainly compare Stratified Sampling, Negative Sharing, and Stratified Sampling with Negative Sharing, against IID sampling and Negative Sampling. It is worth noting that several existing state-of-the-art models [1, 4, 30] are special cases of our framework (e.g. using MSE-loss/Log-loss with CNN or RNN), so they are compared to other loss functions under our framework.

*Evaluation Metrics.* For recommendation performance, we follow [1, 32] and use recall@M. As pointed out in [32], the precision is not a suitable performance measure since non interactions may be due to (1) the user is not interested in the item, or (2) the user does not pay attention to its existence. More specifically, for each user, we rank candidate test items based on the predicted scores, and then compute recall@M based on the list. Finally the recall@M is averaged over all users.

As for the computational cost, we mainly measure it in three dimensions: the training time for each iteration (or epoch equivalently, since batch size is fixed for all methods), the number of iterations needed to reach a referenced loss, and the total amount of computation time needed to reach the same loss. In our experiments, we use the smallest loss obtained by IID sampling in the maximum 30 epochs as referenced loss. Noted that all time measure mentioned here is in Wall Time.

*Parameter Settings.* The key parameters are tuned with validation set, while others are simply set to reasonable values. We adopt Adam [16] as the stochastic optimizer. We use the same batch size  $b = 512$  for all sampling strategies, we use the number of positive link per sampled stratum  $s = 4$ , learning rate is set to 0.001 for MSE-loss, and

**Table 3: Comparisons of speedup for different sampling strategies against IID Sampling: per iteration, # of iteration, and total speedup.**

Model	Sampling	CiteULike			News		
		Per it.	# of it.	Total	Per it.	# of it.	Total
CNN	Negative	1.02	1.00	1.02	1.03	1.03	1.06
	Stratified	8.83	0.97	8.56	6.40	0.97	6.20
	N.S.	8.42	<b>2.31</b>	19.50	6.54	<b>2.21</b>	14.45
	Strat. w. N.S.	<b>15.53</b>	1.87	<b>29.12</b>	<b>11.49</b>	2.17	<b>24.98</b>
LSTM	Negative	0.99	0.96	0.95	1.0	1.25	1.25
	Stratified	3.1	0.77	2.38	3.12	1.03	3.22
	N.S.	2.87	<b>2.45</b>	7.03	2.78	<b>4.14</b>	<b>11.5</b>
	Strat. w. N.S.	<b>3.4</b>	2.22	<b>7.57</b>	<b>3.13</b>	3.32	10.41

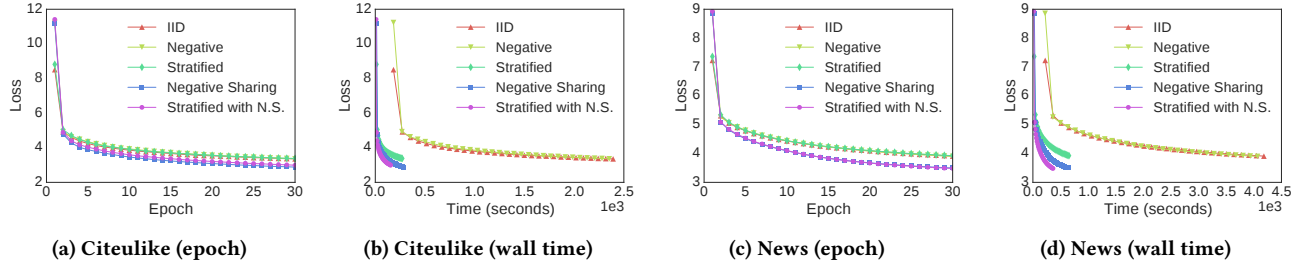
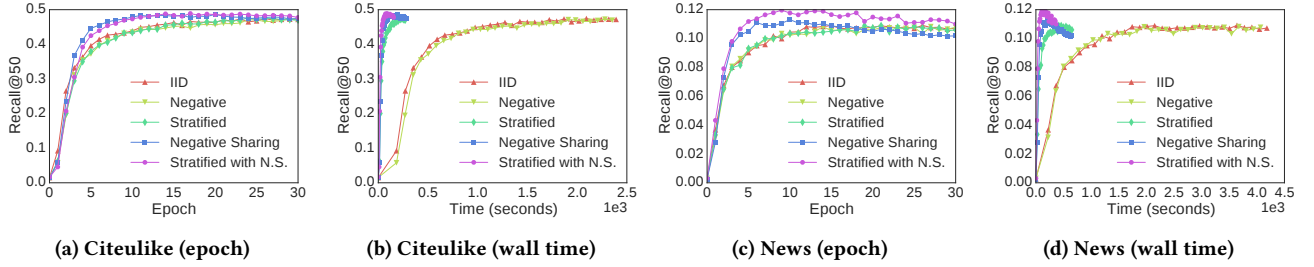
0.01 for others.  $\gamma$  is set to 0.1 for Hinge-loss, and 10 for others.  $\lambda$  is set to 8 for MSE-loss, and 128 for others. We set number of negative examples  $k = 10$  for convolutional neural networks, and  $k = 5$  for RNN/LSTM due to the GPU memory limit. All experiments are run with Titan X GPUs. We use unigram noise/negative distribution.

For CNN, we adopt the structure similar in [15], and use 50 filters with filter size of 3. Regularization is added using both weight decay on user embedding and dropout on item embedding. For RNN, we use LSTM [12] with 50 hidden units. For both models, the dimensions of user and word embedding are set to 50. Early stop is utilized, and the experiments are run to maximum 30 epochs.

## 4.3 Speedup Under Different Sampling Strategies

Table 3 breaks down the speedup into (1) speedup for training on a given mini-batch, (2) number of iterations (to reach referenced cost) speedup, and (3) the total speedup, which is product of the first two. Different strategies are compared against IID Sampling. It is shown that Negative Sampling has similar computational cost as IID Sampling, which fits our projection. All three proposed sampling strategies can significantly reduce the computation cost within a mini-batch. Moreover, the Negative Sharing and Stratified Sampling with Negative Sharing can further improve the convergence w.r.t.

<sup>6</sup><https://webscope.sandbox.yahoo.com/catalog.php?datatype=r&did=75>

Figure 5: Training loss curves (all methods have the same number of  $b$  positive samples in a mini-batch)Figure 6: Test performance/recall curves (all methods have the same number of  $b$  positive samples in a mini-batch).

the number of iterations, which demonstrates the benefit of using larger number of negative examples.

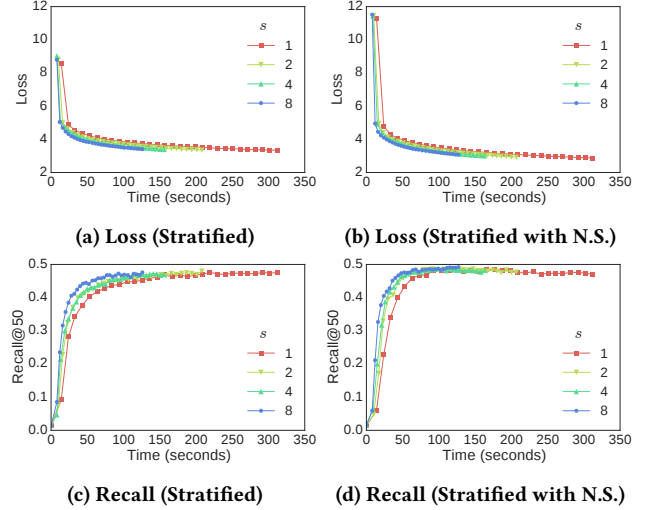
Figure 5 and 6 shows the convergence curves of both loss and test performance for different sampling strategies (with CNN + SG-loss). In both figures, we measure progress every epoch, which is equivalent to a fixed number of iterations since all methods have the same batch size  $b$ . In both figures, we can observe mainly two types of convergences behavior. Firstly, in terms of number of iterations, Negative Sharing (w./wo. Stratified Sampling) converge fastest, which attributes to the number of negative samples used. Secondly, in terms of wall time, Negative Sharing (w./wo. Stratified Sampling) and Stratified Sampling (by Items) are all significantly faster than baseline sampling strategies, i.e. IID Sampling and Negative Sampling. It is also interesting to see that that overfitting occurs earlier as convergence speeds up, which does no harm as early stopping can be used.

For Stratified Sampling (w./wo. negative sharing), the number of positive links per stratum  $s$  can also play a role to improve speedup as we analyzed before. As shown in Figure 7, the convergence time as well as recommendation performance can both be improved with a reasonable  $s$ , such as 4 or 8 in our case.

#### 4.4 Recommendation Performance Under Different Sampling Strategies

It is shown in above experiments that the proposed sampling strategies are significantly faster than the baselines. But we would also like to further access the recommendation performance by adopting the proposed strategies.

Table 4 compares the proposed sampling strategies with CNN/RNN models and four loss functions (both pointwise and pairwise). We

Figure 7: The number of positive links per stratum  $s$  VS loss and performance.

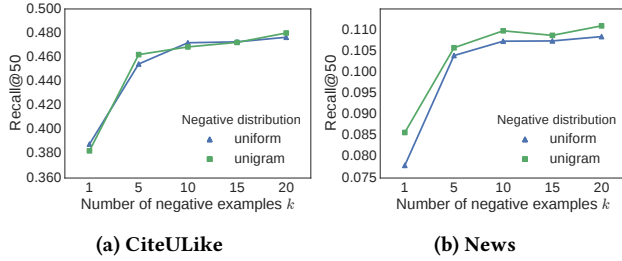
can see that IID Sampling, Negative Sampling and Stratified Sampling (by Items) have similar recommendation performances, which is expected since they all utilize same amount of negative links. For Negative Sharing and Stratified Sampling with Negative Sharing, since there are much more negative samples utilized, their performances are significantly better. We also observe that the current recommendation models based on MSE-loss [1, 30] can be improved by others such as SG-loss and pairwise loss functions [4].

To further investigate the superior performance brought by Negative Sharing. We study the number of negative examples  $k$  and



**Table 4: Recall@50 for different sampling strategies under different models and losses.**

Model	Sampling	CiteULike				News			
		SG-loss	MSE-loss	Hinge-loss	Log-loss	SG-loss	MSE-loss	Hinge-loss	Log-loss
CNN	IID	0.4746	0.4437	-	-	0.1091	0.0929	-	-
	Negative	0.4725	0.4408	0.4729	0.4796	0.1083	0.0956	0.1013	0.1009
	Stratified	0.4761	0.4394	-	-	0.1090	0.0913	-	-
	Negative Sharing	0.4866	0.4423	<b>0.4794</b>	0.4769	0.1131	0.0968	0.0909	0.0932
	Stratified with N.S.	<b>0.4890</b>	<b>0.4535</b>	0.4790	<b>0.4884</b>	<b>0.1196</b>	<b>0.1043</b>	<b>0.1059</b>	<b>0.1100</b>
LSTM	IID	0.4479	0.4718	-	-	0.0971	0.0998	-	-
	Negative	0.4371	0.4668	0.4321	0.4540	0.0977	0.0977	0.0718	0.0711
	Stratified	0.4344	0.4685	-	-	0.0966	0.0996	-	-
	Negative Sharing	0.4629	0.4839	0.4605	0.4674	<b>0.1121</b>	0.0982	0.0806	0.0862
	Stratified with N.S.	<b>0.4742</b>	<b>0.4877</b>	<b>0.4703</b>	<b>0.4730</b>	0.1051	<b>0.1098</b>	<b>0.1017</b>	<b>0.1002</b>

**Figure 8: The number of negatives VS performances.**

the convergence performance. Figure 8 shows the test performance against various  $k$ . As shown in the figure, we observe a clear diminishing return in the improvement of performance. However, the performance seems still increasing even we use 20 negative examples, which explains why our proposed method with negative sharing can result in better performance.

## 5 RELATED WORK

Collaborative filtering [18] has been one of the most effective methods in recommender systems, and methods like matrix factorization [17, 27] are widely adopted. While many papers focus on the explicit feedback setting such as rating prediction, implicit feedback is found in many real-world scenarios and studied by many papers as well [13, 23, 26]. Although collaborative filtering techniques are powerful, they suffer from the so-called “cold-start” problem since side/content information is not well leveraged. To address the issue and improve performance, hybrid methods are proposed to incorporate side information [5, 7, 25, 28, 38], as well as content information [4, 11, 31, 32].

Deep Neural Networks (DNNs) have been showing extraordinary abilities to extract high-level features from raw data, such as video, audio, and text [8, 15, 34]. Compared to traditional feature detectors, such as SIFT and n-grams, DNNs and other embedding methods [5, 6, 29] can automatically extract better features that produce

higher performance in various tasks. To leverage the extraordinary feature extraction or content understanding abilities of DNNs for recommender systems, recent efforts are made in combining collaborative filtering and neural networks [1, 4, 30, 32]. [32] adopts autoencoder for extracting item-side text information for article recommendation, [1] adopts RNN/GRU to better understand the text content. [4] proposes to use CNN and pairwise loss functions, and also incorporate unsupervised text embedding. The general functional embedding framework in this work subsumes existing models [1, 4, 30].

Stochastic Gradient Descent [3] and its variants [16] have been widely adopted in training machine learning models, including neural networks. Samples are drawn uniformly at random (IID) so that the stochastic gradient vector equals to the true gradient in expectation. In the setting where negative examples are overwhelming, such as in word embedding (e.g., Word2Vec [22]) and network embedding (e.g., LINE [29]) tasks, negative sampling is utilized. Recent efforts have been made to improve SGD convergence by (1) reducing the variance of stochastic gradient estimator, or (2) distributing the training over multiple workers. Several sampling techniques, such as stratified sampling [35] and importance sampling [36] are proposed to achieve the variance reduction. Different from their work, we improve sampling strategies in SGD by reducing the computational cost of a mini-batch while preserving, or even increasing, the number of data points in the mini-batch. Sampling techniques are also studied in [9, 39] to distribute the computation of matrix factorization, their objectives in sampling strategy design are reducing the parameter overlapping and cache miss. We also find that the idea of sharing negative examples is exploited to speed up word embedding training in [14].

## 6 DISCUSSIONS

While it is discussed under content-based collaborative filtering problem in this work, the study of sampling strategies for “graph-based” loss functions have further implications. The IID sampling strategy is simple and popular for SGD-based training, since the

loss function terms usually do not share the common computations. So no matter how a mini-batch is formed, it almost bears the same amount of computation. This assumption is shattered by models that are defined under graph structure, with applications in social and knowledge graph mining [2], image caption ranking [20], and so on. For those scenarios, we believe better sampling strategies can result in much faster training than that with IID sampling.

We would also like to point out limitations of our work. The first one is the setting of implicit feedback. When the problem is posed under explicit feedback, Negative Sharing can be less effective since the constructed negative samples may not overlap with the explicit negative ones. The second one is the assumption of efficient computation for interaction functions. When we use neural networks as interaction functions, we may need to consider constructing negative samples more wisely for Negative Sharing as it will also come with a noticeable cost.

## 7 CONCLUSIONS AND FUTURE WORK

In this work, we propose a hybrid recommendation framework, combining conventional collaborative filtering with (deep) neural networks. The framework generalizes several existing state-of-the-art recommendation models, and embody potentially more powerful ones. To overcome the high computational cost brought by combining “cheap” CF with “expensive” NN, we first establish the connection between the loss functions and the user-item interaction bipartite graph, and then point out the computational costs can vary with different sampling strategies. Based on this insight, we propose three novel sampling strategies that can significantly improve the training efficiency of the proposed framework, as well as the recommendation performance.

In the future, there are some promising directions. Firstly, based on the efficient sampling techniques of this paper, we can more efficiently study different neural networks and auxiliary information for building hybrid recommendation models. Secondly, we can also study the effects of negative sampling distributions and its affect on the design of more efficient sampling strategies. Lastly but not least, it would also be interesting to apply our sampling strategies in a distributed training environments where multi-GPUs and multi-machines are considered.

## ACKNOWLEDGEMENTS

The authors would like to thank anonymous reviewers for helpful suggestions. The authors would also like to thank NVIDIA for the donation of one Titan X GPU. This work is partially supported by NSF CAREER #1741634.

## REFERENCES

- [1] Trapit Bansal, David Belanger, and Andrew McCallum. 2016. Ask the GRU: Multi-task Learning for Deep Text Recommendations. In *RecSys'16*. 107–114.
- [2] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. In *NIPS'13*. 2787–2795.
- [3] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *COMPSTAT'2010*. Springer, 177–186.
- [4] Ting Chen, Liangjie Hong, Yue Shi, and Yizhou Sun. 2017. Joint Text Embedding for Personalized Content-based Recommendation. In *arXiv preprint arXiv:1706.01084*.
- [5] Ting Chen and Yizhou Sun. 2017. Task-Guided and Path-Augmented Heterogeneous Network Embedding for Author Identification. In *WSDM'17*. 295–304.
- [6] Ting Chen, Lu-An Tang, Yizhou Sun, Zhengzhang Chen, and Kai Zhang. 2016. Entity Embedding-based Anomaly Detection for Heterogeneous Categorical Events. In *IJCAI'16*. Miami.
- [7] Tianqi Chen, Weinan Zhang, Qiuxia Lu, Kailong Chen, Zhao Zheng, and Yong Yu. 2012. SVDFeature: a toolkit for feature-based collaborative filtering. *Journal of Machine Learning Research* 13, Dec (2012), 3619–3622.
- [8] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural language processing (almost) from scratch. *Journal of Machine Learning Research* 12, Aug (2011), 2493–2537.
- [9] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. 2011. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD'11*. 69–77.
- [10] Saeed Ghadimi and Guanghui Lan. 2013. Stochastic first-and zeroth-order methods for nonconvex stochastic programming. *SIAM Journal on Optimization* 23, 4 (2013), 2341–2368.
- [11] Prem K Gopalan, Laurent Charlin, and David Blei. 2014. Content-based recommendations with poisson factorization. In *NIPS'14*. 3176–3184.
- [12] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
- [13] Yifan Hu, Yehuda Koren, and Chris Volinsky. 2008. Collaborative filtering for implicit feedback datasets. In *ICDM'08*. 263–272.
- [14] Shihao Ji, Nadathur Satish, Sheng Li, and Pradeep Dubey. 2016. Parallelizing word2vec in shared and distributed memory. *arXiv preprint arXiv:1604.04661* (2016).
- [15] Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).
- [16] Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
- [17] Yehuda Koren. 2008. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *KDD'08*. 426–434.
- [18] Yehuda Koren, Robert Bell, Chris Volinsky, et al. 2009. Matrix factorization techniques for recommender systems. *Computer* 42, 8 (2009), 30–37.
- [19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *NIPS'12*. 1097–1105.
- [20] Xiao Lin and Devi Parikh. 2016. Leveraging visual question answering for image-caption ranking. In *ECCV'16*. Springer, 261–277.
- [21] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
- [22] T Mikolov and J Dean. 2013. Distributed representations of words and phrases and their compositionality. *NIPS'13* (2013).
- [23] Rong Pan, Yunhong Zhou, Bin Cao, Nathan N Liu, Rajan Lukose, Martin Scholz, and Qiang Yang. 2008. One-class collaborative filtering. In *ICDM'08*. 502–511.
- [24] Sashank J Reddi, Ahmed Hefny, Suvrit Sra, Barnabas Poczos, and Alex Smola. 2016. Stochastic Variance Reduction for Nonconvex Optimization. In *ICML'16*. 314–323.
- [25] Steffen Rendle. 2010. Factorization machines. In *ICDM'10*. 995–1000.
- [26] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. 2009. BPR: Bayesian personalized ranking from implicit feedback. In *UAI'09*. AUAI Press, 452–461.
- [27] Ruslan Salakhutdinov and Andriy Mnih. 2011. Probabilistic matrix factorization. In *NIPS'11*, Vol. 20. 1–8.
- [28] Ajit P Singh and Geoffrey J Gordon. 2008. Relational learning via collective matrix factorization. In *KDD'08*. 650–658.
- [29] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. Line: Large-scale information network embedding. In *WWW'15*. 1067–1077.
- [30] Aaron Van den Oord, Sander Dieleman, and Benjamin Schrauwen. 2013. Deep content-based music recommendation. In *NIPS'13*. 2643–2651.
- [31] Chong Wang and David M Blei. 2011. Collaborative topic modeling for recommending scientific articles. In *KDD'11*. 448–456.
- [32] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. 2015. Collaborative deep learning for recommender systems. In *KDD'15*. 1235–1244.
- [33] Markus Weimer, Alexandros Karatzoglou, and Alex Smola. 2008. Improving maximum margin matrix factorization. *Machine Learning* 72, 3 (2008), 263–276.
- [34] Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In *NIPS'15*. 649–657.
- [35] Peilin Zhao and Tong Zhang. 2014. Accelerating minibatch stochastic gradient descent using stratified sampling. *arXiv preprint arXiv:1405.3080* (2014).
- [36] Peilin Zhao and Tong Zhang. 2015. Stochastic Optimization with Importance Sampling for Regularized Loss Minimization. In *ICML'15*. 1–9.
- [37] Yin Zheng, Bangsheng Tang, Wenkui Ding, and Hanning Zhou. 2016. A Neural Autoregressive Approach to Collaborative Filtering. In *ICML'16*. 764–773.
- [38] Ke Zhou, Shuang-Hong Yang, and Hongyuan Zha. 2011. Functional matrix factorizations for cold-start recommendation. In *SIGIR'11*. 315–324.
- [39] Yong Zhuang, Wei-Sheng Chin, Yu-Chin Juan, and Chih-Jen Lin. 2013. A fast parallel SGD for matrix factorization in shared memory systems. In *Recsys*. 249–256.