Noname manuscript No.
(will be inserted by the editor)

Efficient, Simple and Automated Negative Sampling for Knowledge Graph Embedding

Yongqi Zhang · **Quanming Yao** · **Lei Chen**

Received: date / Accepted: date

Abstract Negative sampling, which samples negative triplets from non-observed ones in knowledge graph (KG), is an essential step in KG embedding. Recently, generative adversarial network (GAN), has been introduced in negative sampling. By sampling negative triplets with large gradients, these methods avoid the problem of vanishing gradient and thus obtain better performance. However, they make the original model more complex and harder to train. In this paper, motivated by the observation that negative triplets with large gradients are important but rare, we propose to directly keep track of them with the cache. In this way, our method acts as a "distilled" version of previous GAN-based methods, which does not waste training time on additional parameters to fit the full distribution of negative triplets. However, how to sample from and update the cache are two critical questions. We propose to solve these issues by automated machine learning techniques. The automated version also covers GAN-based methods as special cases. Theoretical explanation of NSCaching is also provided, justifying the superior over fixed sampling scheme. Besides, we further extend NSCaching with skip-gram model for graph embedding. Finally, extensive experiments show that our method can gain significant improvements on various KG embedding models and the skip-gram model, and outperforms the state-of-the-art negative sampling methods.

Keywords Knowledge Graph · Graph Embedding · Negative Sampling · Automated Machine Learning

Yongqi Zhang
4Paradigm Inc.
E-mail: zhangyongqi@4paradigm.com

Quanming Yao
Department of Electronic Engineering, Tsinghua University and 4Paradigm Inc.
E-mail: yaoquanming@4paradigm.com

Lei Chen
Hong Kong University of Science and Technology
E-mail: leichen@cse.ust.hk

Code: `https://github.com/AutoML-4Paradigm/NSCaching`
Correspondance is to: Quanming Yao.

1 Introduction

Knowledge graph (KG) is a special kind of graph structure, with entities as nodes and relations as directed edges. Each edge (also called a fact) is represented as a triplet with the form *(head entity, relation, tail entity)*, denoted as (h,r,t), indicating that two entities are connected by a specific relation, e.g. *(Steve Jobs, founded, Apple Inc.)* in the example in Figure 1. These triplets are usually extracted manually or based on automatically constructed knowledge bases [47]. KG is very general and useful. It has been used as fundamental building blocks for many applications like structured search [16, 39], question answering [7], recommendation [42, 64] and medical diagnosis [63]. This importance has also inspired many famous KG projects, such as FreeBase [6], DBpedia [2], and YAGO [47].

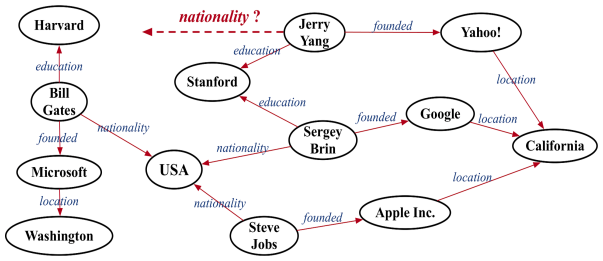


Fig. 1: An example of knowledge graph.

As these triplets are hard to manipulate, how to find a good representation for entities and relations in the KG [41]

is a fundamental problem. Early works towards this goal lie in statistical relational learning by using the symbolic triplet data [29, 30, 32]. However, these methods neither lead to good generalization performance, nor can they be applied for large scale knowledge graphs. In comparison, the embedding based methods have better generalization ability and inference efficiency [8, 44, 71]. Recently, graph embedding techniques [55] have been introduced in KG learning. These methods attempt to encode entities and relations in KG into a low-dimensional vector space while capturing the original connection properties. They are scalable and have also shown promising performance in basic KG tasks, such as link prediction and triplet classification [8, 55].

In recent years, constructing new scoring functions that can better model the complex interactions between entities and relations has been the main focus for improving the performance of KG embedding approaches [25, 51, 56, 60]. However, another very important perspective of KG embedding, i.e., negative sampling, is not sufficiently emphasized. The need for negative sampling comes from the fact that there are usually only positive triplets in KG [30, 55]. First, to avoid trivial solutions of the embedding, a set that contains all the possible negative samples needs to be hand-made. Then, in consideration of both computation cost and memory space, stochastic training is needed in each iteration. Specifically, once we have picked up a positive triplet, we also need to sample some negative triplets from its corresponding negative sample set. Besides, the quality of these negative triplets does matter.

Due to its simplicity and efficiency, uniform sampling is broadly used in KG embedding [55]. However, it is a fixed scheme and ignores changes in the distribution of negative triplets during the training process. As a result, it suffers seriously from the *vanishing gradient* problem [54] and biased estimation problem [45]. As observed in [54], most negative triplets in the sampling set can be easily classified. Since the scoring functions tend to give observed (positive) triplets large values, as training goes, most of the non-observed (probably negative) triplets will have smaller values. Thus, when negative triplets are uniformly sampled, it is very likely that we pick up one with zero gradients. As a result, the training process of KG embedding will be impeded by the vanishing gradients rather than by the optimization algorithm. This problem prevents KG embedding from getting the desired performance. A better sampling scheme, the Bernoulli sampling, is introduced in [56]. It improves uniform sampling by considering one-to-many, many-to-one, and many-to-many mapping in the relations between entities. For example, given a one-to-many relation, replacing the tail entity will have a larger chance of getting a false negative triplet compared with replacing the head. However, it is still a fixed and biased sampling scheme.

Therefore, dynamically sampling from the negative triplets' distribution to help the training process is important and non-trivial. To efficiently capture them during training, we have two main challenges for negative sampling: (i). How to capture and model the negative triplets' dynamic distribution? and (ii). How can we effectively sample the negative triplets? Recently, there are two pioneering works, i.e., IGAN [54] and KBGAN [11], attempting to address these challenges. Their ideas are to replace the fixed sampling scheme with a generative adversarial network (GAN) [20] based sampling scheme. However, GAN-based solutions still have many problems. First, GAN increases the number of training parameters because an extra generator should be learned as sampler. Second, the training of the GAN model usually suffers from instability and degeneracy [1, 22]. The REINFORCE gradient [58], which is known to have high variance, should be used to train the generator. Besides, since only a few negative triplets can lead to large gradient, IGAN and KBGAN take a lot of effort to model the distributions of all the negative ones. These drawbacks lead to instable performance for different scoring functions, and hence pre-training becomes a must for both IGAN and KBGAN. Self adversarial sampling (Self-Adv) [48] uses the self embedding model to replace the generator. It solves the problem of training GAN model, but it cannot guarantee to sample enough negative triplets with large gradient in each iteration.

In this paper, to address the challenges of capturing the dynamic and complex negative sampling distribution while avoid the problems of using GANs, we propose a simple and efficient negative sampling method based on the cache, called NSCaching. By empirically analyzing the gradient norm distribution of negative triplets, we find that the distribution is highly skewed. In other words, there are only a few pairs of training triplets (i.e., a positive triplet and a negative triplet) have large gradient and the others are useless. This observation motivates us to mainly maintain the negative triplets that lead to large gradients during the training, and dynamically update the maintained triplets. First, we use the cache to store large-gradient negative triplets. Then, we carefully design the updating and sampling rules for the cache. In detail, the cache-based sampling problem is formed as a hyper-parameter optimization (HPO) problem, and we use automated machine learning (AutoML) [61] to efficiently solve it. In this way, we automatically take good care of "exploration and exploitation" (E&E) [34], which balances exploring all possible high-quality negative triplets and sampling from a few of them in the cache. Contributions of our work are summarized as follows:

– We propose a simple, efficient and automated negative sampling algorithm NSCaching, which is a general negative sampling scheme and can be easily injected into many

popularly used KG embedding models. NSCaching has fewer parameters than both IGAN [54] and KBGAN [11].
- We provide intuitions about how NSCaching helps the KG training under convex and non-convex cases. in the convex case, we show that the negative sampling scheme in NSCaching can lead to a smaller approximation error. In the general non-convex case, we show that NSCaching can benefit from *self-paced learning* [3, 31] by learning easy samples first and gradually switching to harder ones.
- A critical issue in the NSCaching algorithm is how to balance "exploration and exploitation"(E&E) in updating and sampling from the cache. Motivated by the success of automated machine learning [61], we propose an automated version of NSCaching, i.e., NSCaching (auto). The AutoML-based method has a unified view of the hyper-parameters (related to E&E) of NSCaching, and also covers IGAN/KBGAN as special cases. Thus, it enables us to automatically balance E&E.
- We conduct experiments on five popular data sets, i.e., WN18 and FB15K (and their variants WN18RR and FB15K237), and YAGO3-10. Experimental results demonstrate that the NSCaching algorithm is very efficient and is more effective than the baselines as well. The automated version further improves NSCaching by balancing between exploration and exploitation better.
- We extend the negative sampling algorithm from KG embedding to graph embedding. Random walk based graph embedding [21], which is trained with the widely used skip-gram model [36], is chosen as the testbed. The cache-based negative sampling is used to replace the frequency-based negative sampling in skip-gram models. Experiments on the graph embedding show that NSCaching adapts well to the new task.

The preliminary version of this paper has been published in ICDE 2019 [69]. Comparing with [69], we have made the following important improvements:

1. We systematically extend the algorithm by introducing AutoML to automatically balance E&E by hyper-parameter optimization (Section 3.4). The AutoML approach not only helps to improve the performance of NSCaching, but also offers insight on how GAN-based methods work;
2. We extend NSCaching to a new task, i.e., graph embedding based on the skip-gram model (Section 4). We show that NSCaching algorithm can be easily adopted into such a new application scenario and get good empirical performance (Section 5.7).
3. We add theoretical explanation of how NSCaching leads to a smaller approximation error when the objective is convex (Section 3.3.1). The new result provides intuitions about how NSCaching helps train embeddings.
4. Based on the theoretical analysis, we reform the problem from gradient point of view in Section 3.1, and added positive sampling into the algorithm (Section 3.2.2). To our knowledge, the positive sampling problem has not been explored in KG embedding area.
5. Moreover, we have conducted more experiments with new data sets, scoring functions and tasks to show the effectiveness of our algorithm (Section 5.3), automated machine learning to further boost performance (Section 5.4), ablation study to analyze the design components (Section 5.5), synthetic setting to illustrate the convergence properties (Section 5.6.2), and graph embedding to verify the extension to the skip-gram model (Section 5.7).

Table 1: Symbols and notations.

Symbol	Description
$\mathscr{E}, \mathscr{R}$	the set of entities and set of relations
$h,t \in \mathscr{E}, r \in \mathscr{R}$	head and tail entity, relation
$\mathscr{S} \equiv \{(h,r,t)\}$	the set of triplets
$\bar{\mathscr{S}}_i = \{(\bar{h}_i, r_i, \bar{t}_i)\}$	the set of negative triplets for (h_i, r_i, t_i)
$\mathbf{h}, \mathbf{t} \in \mathbb{R}^{d_1}$	embedding of head entity and tail entity
$\mathbf{r} \in \mathbb{R}^{d_2}$	embedding of relation
$\mathscr{C}$	the cache
$f(h,r,t)$	the scoring function of the triplet (h,r,t)
$\mathscr{N}_i \subset \bar{\mathscr{S}}_i$	candidate subset of negative triplets
N_1, N_2	cache size $\lvert\mathscr{C}_i\rvert$, candidate size $\lvert\mathscr{N}_i\rvert$
$\alpha_1, \alpha_2, \alpha_3 > 0$	temperature values for softmax function

Notations. The mostly used symbols and their descriptions are given in Table 1. Vectors are denoted by lowercase boldface, and matrices by uppercase boldface. $Re(\cdot)$ takes the real part of complex numbers, $conj(\mathbf{t}) = \mathbf{t}_{real} - i\mathbf{t}_{image}$ is the conjugate of complex vectors $\mathbf{t} = \mathbf{t}_{real} + i\mathbf{t}_{image} \in \mathbb{C}^d$. $\langle \mathbf{a}, \mathbf{b}, \mathbf{c} \rangle = \sum_{i=1}^{d} \mathbf{a}_i \cdot \mathbf{b}_i \cdot \mathbf{c}_i$ is the inner product.

2 Preliminaries and Related Works

In this section, we introduce the stochastic training algorithm for KG embedding in Section 2.1, current strategies for negative sampling in Section 2.2, and AutoML techniques in Section 2.3.

2.1 Knowledge Graph (KG) Embedding

To build a KG embedding model, we first need to pick up a scoring function f, which captures the similarities between two entities based on a relation [55]. Different scoring functions have their own weaknesses and strengths in capturing the underneath interactions. Some popularly used scoring functions are presented in Table 2. Besides, the observed facts in KG are supposed to have larger scores than the non-observed ones [55]. With the factual information, the

Table 2: Definitions of some popular scoring functions. All model embeddings are real values, except that ComplEx has complex values. $\mathbf{h}_1, \mathbf{r}_1, \mathbf{t}_1$ and $\mathbf{h}_2, \mathbf{r}_2, \mathbf{t}_2$ are indexed from two different sets of embeddings.

model	scoring function	definition
transna-	TransE [8]	$-\|\mathbf{h}+\mathbf{r}-\mathbf{t}\|_1$
tional	TransH [56]	$-\left\|\mathbf{h}-\mathbf{w}_r^\top \mathbf{h}\mathbf{w}_r+\mathbf{r}-(\mathbf{t}-\mathbf{w}_r^\top \mathbf{t}\mathbf{w}_r)\right\|_1$
distance	TransD [25]	$-\left\|\mathbf{h}+\mathbf{w}_r\mathbf{w}_h^\top \mathbf{h}+\mathbf{r}-(\mathbf{t}+\mathbf{w}_r\mathbf{w}_t^\top \mathbf{t})\right\|_1$
semantic	DistMult [60]	$\langle \mathbf{h}, \mathbf{r}, \mathbf{t}\rangle$
matching	ComplEx [51]	$\mathrm{Re}\left(\langle \mathbf{h}, \mathbf{r}, \mathrm{conj}(\mathbf{t})\rangle\right)$
	SimplE [27]	$\langle \mathbf{h}_1, \mathbf{r}_1, \mathbf{t}_2\rangle + \langle \mathbf{h}_2, \mathbf{r}_2, \mathbf{t}_1\rangle$

embeddings are learned by solving the optimization problem that maximizes the scoring function for observed triplets and minimizes it for non-observed triplets at the same time. Based on the properties of scoring functions, KG embedding models are generally divided into two categories.

- The *translational distance model* exploits the distance-based scoring functions. Inspired by the word analogy results in word embeddings [37], the similarity is measured by the distance between two entities, after a translation carried out by the relation. TransE [8], as a representative translational model, is defined by the (negative) distance between $\mathbf{h}+\mathbf{r}$ and $\mathbf{t}$, i.e., $f(h,r,t) = -||\mathbf{h}+\mathbf{r}-\mathbf{t}||_1$. Other translational distance models like TransH [56], TransD [25] enhance over TransE by introducing extra mapping matrices. The translational distance models are generally optimized by minimizing the ranking based loss function

$$\sum_{(h_i,r_i,t_i)\in\mathscr{S}} \sum_{(\bar{h}_i,r_i,\bar{t}_i)\in\bar{\mathscr{S}}_i} \left[\gamma - f(h_i,r_i,t_i) + f(\bar{h}_i,r_i,\bar{t}_i)\right]_+, \quad (1)$$

  where $\gamma > 0$ is the margin value for the loss function.
- Scoring functions in *semantic matching models* exploit the similarity of a triplet by matching latent semantics of entities and relations embedied in their vector space representations. Bilinear models are the state-of-the-art among the semantic matching models and they share the form as $f(h,r,t) = \mathbf{h}^\top \mathbf{R}\mathbf{t}$, where $\mathbf{R} \in \mathbb{R}^{d\times d}$ is a matrix referring to the embedding of relation r [55]. DistMult [60] measures the similarity by directly computing the element-wise product of the embedding vectors, i.e., $f(h,r,t) = \langle \mathbf{h}, \mathbf{r}, \mathbf{t}\rangle$, which restricts $\mathbf{R}$ to be a diagonal matrix. However, it can not model asymmetric triplets since $f(h,r,t) = f(t,r,h)$ is always satisfied. ComplEx [51] and SimplE [27] improve over DistMult by dealing with asymmetric triplets in different ways. Another type of models conducts semantic matching using neural networks. Multi-Layer Perceptron (MLP) is used in [15] to measure the similarities. ConvE [13] takes advantage of convolutional neural network to increase the interactions among different dimensions. Even though neural network models are more complex than the bilinear models, they empirically perform worse than the bilinear models [27, 51]. The semantic matching models are mainly optimized by minimizing the classification based loss function

$$\sum_{(h_i,r_i,t_i)\in\mathscr{S}} \sum_{(\bar{h}_i,r_i,\bar{t}_i)\in\bar{\mathscr{S}}_i} \ell\left(1, f(h_i,r_i,t_i)\right) + \ell\left(-1, f(\bar{h}_i,r_i,\bar{t}_i)\right), \quad (2)$$

  where $(\bar{h},r,\bar{t}) \notin \mathscr{S}$ is the hand-made negative triplet for (h,r,t) and $\ell(\alpha,\beta) = \log(1+\exp(-\alpha\beta))$.

Algorithm 1 Stochastic gradient descent for knowledge graph embedding [8, 55].

Require: training set $\mathscr{S} = \{(h,r,t)\}$, embedding dimension d and scoring function f;
1: initialize the embeddings for each $e \in \mathscr{E}$ and $r \in \mathscr{R}$.
2: **for** $i = 1, \cdots, T$ **do**
3: sample an observed triplet $(h_i,r_i,t_i) \in \mathscr{S}$;
4: sample the corresponding negative triplet $(\bar{h}_i,r_i,\bar{t}_i) \in \bar{\mathscr{S}}_i$; // *negative sampling*
5: update parameters of embeddings w.r.t. the gradients using (i). translational distance models:

$$\nabla\left[\gamma - f(h_i,r_i,t_i) + f(\bar{h}_i,r_i,\bar{t}_i)\right]_+, \quad (3)$$

or (ii). semantic matching models:

$$\nabla\left[\ell(+1, f(h_i,r_i,t_i)) + \ell\left(-1, f(\bar{h}_i,r_i,\bar{t}_i)\right)\right]; \quad (4)$$

6: **end for**

The above two objectives, i.e., (1) and (2), can be optimized by using stochastic gradient descent in an unified manner (Algorithm 1). In each iteration, an observed (positive) triplet (h_i,r_i,t_i) is firstly sampled from the training set $\mathscr{S}$ at step 3. Since there are no negative triplets in $\mathscr{S}$, in step 4, a negative triplet of (h_i,r_i,t_i) is sampled from the corresponding negative triplets set $\bar{\mathscr{S}}_i$ [8], i.e.,

$$\bar{\mathscr{S}}_i = \left\{(\bar{h},r_i,t_i) \notin \mathscr{S} \,|\, \bar{h} \in \mathscr{E}\right\} \cup \left\{(h_i,r_i,\bar{t}) \notin \mathscr{S} \,|\, \bar{t} \in \mathscr{E}\right\}. \quad (5)$$

Finally, embedding parameters are updated in step 5. Since the quality of negative triplets in $\bar{\mathscr{S}}_i$ is diverse, how to sample a proper $(\bar{h}_i,r_i,\bar{t}_i)$ has been developed as an important perspective affecting the performance of knowledge graph embedding [11, 48, 54, 55].

2.2 Negative Sampling in KG Embedding

Negative sampling is important for improving learning models when there are only positive samples. The typical applications include natural language processing [12,33,38],

computer vision [59], graph embedding [21, 44, 49], recommender system [14, 46, 62, 66], and KG embedding [8, 55] here. Existing works on negative sampling can be divided into two categories, i.e., sampling from fixed distribution and sampling from dynamic distribution.

A uniform distribution over the negative samples in the candidate set is a simple yet efficient choice [8, 46]. Whereas, many of the uniformly generated negative samples are not informative and too trivial to recognize [14, 33, 38, 59]. In order to generate more informative negative samples, important statistics in the data set can be used to define the distribution, such as the frequency of words [37] and personalized PageRank score of items [65]. However, the uniform sampling methods are biased-estimator of the full negative sampling distribution [45].

As the training goes on, the distributions of scores and gradients of negative samples keep changing. In order to capture the dynamic distribution of negative samples, several works are proposed to sample according to the scores [9,12,14,18,33,54,59]. There are two approaches in general. In one direction, the high-quality negative samples are selected based on the scores in a small pool sampled from all the candidates [33, 48]. This approach is efficient since only a small number of the scores need to be computed. In another, a distribution on all the candidates is modeled to generate the high-quality negative samples [18, 54]. Having an overall distribution of the negative samples makes these method more flexible.

In the following content, we discuss the sampling methods specifically used in KG embedding tasks.

2.2.1 Sample from Fixed Distributions

In the early work [8], negative triplets are *uniformly* sampled from the set $\mathscr{T}_i$. This strategy is simple yet very efficient. Later, a better sampling scheme, i.e., Bernoulli sampling, is introduced in [56]. It improves uniform sampling by reducing the appearance of false negative triplets existing in one-to-many, many-to-many, and many-to-one relations between head and tail entities. However, as mentioned in the introduction, the Bernoulli method still samples from fixed distributions, which can neither model the dynamic changes in distributions of negative triplets nor can it sample triplets with large gradient.

As introduced in Section 1, the vanishing gradient (or zero loss) problem [54] means a certain number of negative triplets will lead to zero gradient and thus is not informative for training with gradient-based optimization algorithms. Taking the ranking based loss (3) for example, the score of negative triplets $f(\bar{h}_i, r_i, \bar{t}_i)$ are gradually minimized as training goes on. For most of the negative triplets, the score will be very small and the loss will decrease to zero soon, leading to zero gradients. The gradient on classification loss (4) will go close to zero if $f(\bar{h}_i, r_i, \bar{t}_i)$ is small. As a result, those negative triplets cannot provide enough gradient value to update the embeddings. Besides, the fixed sampling methods cannot capture the dynamic distribution of negative triplets, leading to a biased estimator.

2.2.2 Sample from Generative Adversarial Network (GAN)

GAN [20] is originally introduced as a powerful model for image generation. It contains two modules: a *generator* that serves as the sampler, and a *discriminator* that measures the quality of generated samples. Under elaborate control on the training procedure of generator and discriminator, GAN has achieved significant success in many fields, e.g., computer vision [1, 22], natural language processing [17], information retrieval [53] and graph mining [52]. It has also been shown to generate negative samples with high-quality for knowledge graph embedding [11, 48, 54].

When GAN is applied to negative sampling, the jointly trained generator can dynamically adapt to the new distributions by confusing the discriminator and keeping training. The discriminator, i.e., the KG embedding model, learns to distinguish between the positive triplets and the negative triplets sampled by the generator. Under an alternating training process, the generator dynamically approximates the negative sample distribution and the KG embedding model is improved by the negative triplets with relatively large gradient sampled by the generator.

Given a positive triplet (h,r,t), IGAN [54] models the distribution $\bar{h},\bar{t} \sim p(e|(h,r,t))$ over all the entities to sample a negative triplet $(\bar{h},r,\bar{t})$. The gradient of $(\bar{h},r,\bar{t})$ is approximately measured by the discriminator, i.e., the loss function $\ell = \left[\gamma - f(h,r,t) + f(\bar{h},r,\bar{t})\right]_+$ of the target KG embedding model. By joint training, IGAN can dynamically capture the distribution of all negative triplets. Instead of modeling a distribution over the whole entity set, KBGAN [11] learns to sample from a subset of random entities. A set of entities $\mathscr{N} = \{(\bar{h},r,\bar{t})\}$ is uniformly sampled first and then the negative triplet is picked up from $\mathscr{N}$. KBGAN is more efficient than IGAN, but less effective since it is hard to guarantee the candidate set $\mathscr{N}$ to contain enough large-gradient negative triplets.

Even though GAN provides a solution to model the dynamic negative sample distribution, it is famous for suffering from instability and degeneracy [1, 22]. Besides, REINFORCE gradient [58], which is known to have high variance, has to be used to optimize the generator. Therefore, pretraining is a must for both IGAN and KBGAN. It increases the number of model's parameters and brings extra costs on training. A concurrent method Self-Adv [48] adopts a similar approach as KBGAN. The differences are that (i) Self-Adv uses the self model embedding to measure the quality rather than training an extra generator; (ii) Self-Adv

treats the probability as weights rather than the sampling procedure. However, it still cannot guarantee the candidate set $\mathcal{N}$ to contain large-gradient negative triplets.

2.3 Automated Machine Learning (AutoML)

Automated machine learning (AutoML) [24, 61] has recently shown its power in easing the usage of and in designing better machine learning models. It can be regarded as a black-box optimization problem where we target at efficiently searching for better hyper-parameters or model structures. Regarding the success of AutoML, there are two important perspectives

- Search space: This helps us to figure out important properties of the underlying learning model. The search space cannot be too general, otherwise the searching cost in such a space will be too expensive.
- Search algorithm: Since the computation cost of evaluating the settings in search space is high, efficient algorithms should be designed to search efficiently. Taking hyper-parameter optimization (HPO) as an example, gird search or random search [5] are the mostly used method due to their simplicity. However, the searching is usually inefficient and Bayesian optimization [4, 23] is a well-known method to improve the efficiency in HPO.

Considering that the distribution of negative triplets is highly skewed, as will be discussed in Section 3, we should take the sampling distribution seriously. As will be shown in Section 3.4, the proposed NSCaching method naturally allows a search space to automatically balance the exploration and exploitation (E&E) problem in negative sampling, which can further improve the quality of embeddings.

3 Proposed Model

In this section, we first describe our key observations regarding the negative sampling in Section 3.1, which are ignored by existing works but are the main motivations of our work. The proposed method is described in Section 3.2, where we show how to address the challenges in negative sampling by using cache. Then, we analyze the proposed method from theoretical perspectives in Section 3.3. In Section 3.4, we balance exploration and exploitation through AutoML techniques for the proposed method. Finally, we discuss the problem of false negative triplets in Section 3.5.

3.1 Revisiting Distribution of Training Pairs

Before introducing the proposed method, we analyze the distribution of gradients for training pairs here. This motivates us to use cache to efficiently approximate an unbiased distribution of the training pairs. Recall that the gradient at stochastic training of KG embedding is determined by a pair of triplets, i.e., a positive one (h_i, r_i, t_i) from the training set and a negative one $(\bar{h}_i, r_i, \bar{t}_i)$ from negative set $\bar{\mathcal{S}}_i$ (step 3-4 in Algorithm 1). We show the distribution with the ℓ_2-norm of all the training pairs' gradients for a positive (h_i, r_i, t_i).

- Figure 2(a) shows changes of the training pair distribution for a fixed positive triplet (h_i, r_i, t_i) in different epochs; and
- Figure 2(b) shows the training pair distributions for five different positive triplets $(h_i, r_i, t_i) \in \mathcal{S}$.

First, we can see that the distribution of training pairs' gradient are dynamic and highly skewed. Second, the training pairs with large gradients become rare along the iterating (epoch gets more), which is consistent with the observations in [11, 54]. Besides, the distributions of training pairs' gradient for different positive triplets are various.

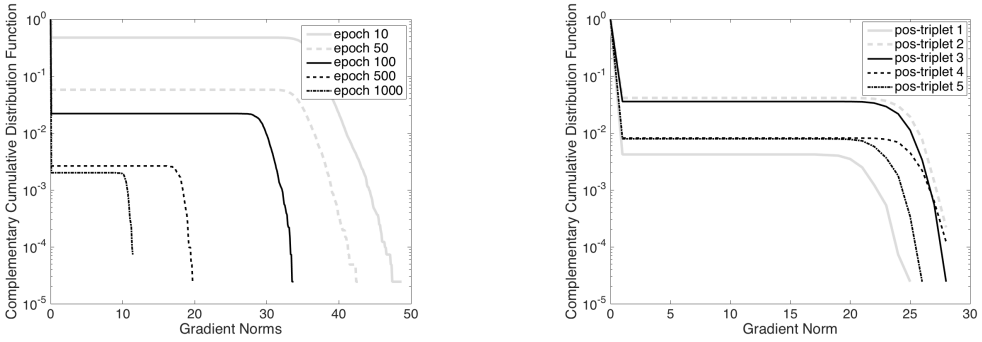
Even though GAN has strong ability in monitoring the full distribution of negative triplets, the GAN-based methods still have a lot of limitations. First, they waste a lot of parameters and computational costs on learning how negative triplets with small gradient norms are distributed. Second, reinforcement learning, which provides gradient to the generator but increases the training difficulties, should be applied in the GAN-based algorithms [11, 54].

Besides, the GAN-based methods ignore the distribution of positive triplets. Since each training pair is composed of a positive triplet and a negative triplet, the differences among positive triplets should also be considered. Some positive triplets can have more large gradient negative samples (see positive triplet 2 v.s. positive triplet 1 in Figure 2(b)). Thus, we want to more frequently pick up the positive triplets with more large gradient negative samples over the others during the stochastic training, i.e., in step 3 of Algorithm 1. However, all existing works including [11,48,54,69] use uniform sampling over positive triplets. For methods sampling from fixed distributions [8, 56], they cannot model the difference of both positive and negative triplets. GAN-based ones are already too complex [11,54] to model the positive sampling. Capturing the difference of positive triplets will further increase the model's parameters and make the training even harder.

3.2 NSCaching: the Proposed Method

From observations in Section 3.1, we have three questions on how to sample the training pairs (i.e., a positive and a negative triplet).

1). Is it possible to directly keep track of negative triplets which can give large gradient for a given positive triplet, rather than the whole negative triplets' distribution?



(a) Grad norm v.s epoch. (b) Grad norm v.s pos-triplet.

Fig. 2: Distribution of training pairs' gradients on WN18 trained by Bernoulli-TransE (see Section 5.3.1). For a given triplet (h_i, r_i, t_i), we fix the head entity h and relation r, and compute the ℓ_2-norm of gradient $\|\mathbf{g}\|_2$ in (3) for all $\bar{t} \in \mathscr{E}$. We measure the *complementary cumulative distribution function* (CCDF) $F_{\|\mathbf{g}\|_2}(x) = P(\|\mathbf{g}\|_2 \geq x)$ to show the proportion of negative triplets that satisfy $\|\mathbf{g}\|_2 \geq x$. (a) is the distribution of training pairs in 5 timestamp of a certain triplet (h_i, r_i, t_i). (b) is the distribution of 5 different triplets (h_i, r_i, t_i) after the pretraining stage.

2). How can we adaptively sample positive triplets having more large-gradient training pairs?

Besides, considering the distribution is dynamic and hard to estimate,

3). how to balance exploring all the training pairs leading to large gradient and exploiting those that have the largest gradient norms?

In this section, we describe the proposed method to address these three questions.

3.2.1 Core Idea: Caching training pairs

As in Section 3.1, the total number of large-gradient negative triplets associated with a positive one is small. Therefore, we are motivated to use a small amount of extra memory, which caches negative samples with large gradient norms for each triplet $(h_i, r_i, t_i) \in \mathscr{S}$. The designed cache acts as a truncated representation of triplets' distribution $(\bar{h}_i, r_i, \bar{t}_i) \in \bar{\mathscr{S}}_i$. Such an idea is previously explored in Word2Vec [37], where the estimated distribution of negative samples is also truncated. This improves both the efficiency and quality of negative sampling.

Note that, as in (5), the negative triplet $(\bar{h}_i, r_i, \bar{t}_i) \notin \mathscr{S}$ is formed by either $(\bar{h}_i, r_i, t_i)$ or $(h_i, r_i, \bar{t}_i)$. Thus, we associate each (h_i, r_i, t_i) with

– a cache $\mathscr{C}_i$, which keeps large-gradient triplets from $\bar{\mathscr{S}}_i$, to store a set of $(\bar{h}_i, r_i, t_i)$ or $(h_i, r_i, \bar{t}_i)$ and the corresponding gradient norms $\|\mathbf{g}_i\|$ (given by (3) or (4)).

However, since the size of $\bar{\mathscr{S}}_i$ is very large, evaluating all of them in $\bar{\mathscr{S}}_i$ to pick up the large-gradient triplets is intractable. The proposed method will adaptively sample a pair of positive and negative triplets directly through the cache. In the sequel, we show how the cache is updated and sampled.

3.2.2 Algorithm Framework

Algorithm 2 shows the KG embedding framework based on our cache-based negative sampling scheme. Note that the proposed algorithm does not depend on the choice of scoring functions, all those in Table 2 can be used here. In Algorithm 2: first, a pair of positive triplet and negative triplet is sampled in step 3; then, the cache is updated in step 5; finally, in step 7, the embeddings are updated based on the choice of scoring functions and loss functions.

Algorithm 2 NSCaching: Cache-based KG embedding.

Require: training set $\mathscr{S} = \{(h, r, t)\}$, embedding dimension d, scoring function f.
1: initialize embeddings for each $e \in \mathscr{E}$ and $r \in \mathscr{R}$, and cache $\mathscr{C}$;
2: **for** $i = 1, \cdots, T$ **do**
3: sample a pair of positive and negate triplet, i.e., (h, r, t) and $(\bar{h}, r, \bar{t})$, using Algorithm 3;
4: **if** $i\%(n+1) == 0$ **then**
5: update the cache $\mathscr{C}_i$ using Algorithm 4;
6: **end if**
7: update embeddings using (3) or (4);
8: **end for**

An overview comparison of the proposed method with state-of-the-art negative sampling method is in Table 3. The main difference with general KG embedding framework in Algorithm 1 is step 3 in Algorithm 2, where the sampling scheme is based on the cache rather than a uniform Bernoulli sampling. Besides, compared with the complex

Table 3: Comparison of the proposed approach with state-of-the-arts, which address the negative sample. Model parameters are based on TransE, m is the size of mini-batch, n is the epoch of lazy-update.

	triplet sampling		training	mini-batch computation		model
	positive	negative		time	space	parameters
baseline	uniform random (UR)		gradient descent (from scratch)	$O(md)$	$O(md)$	$(\|\mathscr{E}\|+\|\mathscr{R}\|)d$
IGAN [54]	UR	GAN	reinforce learning (with pretraining)	$O(m\|\mathscr{E}\|d)$	$O(m\|\mathscr{E}\|d)$	$3(\|\mathscr{E}\|+\|\mathscr{R}\|)d$
KBGAN [11]	UR	GAN	reinforce learning (with pretraining)	$O(mN_1d)$	$O(mN_1d)$	$2(\|\mathscr{E}\|+\|\mathscr{R}\|)d$
NSCaching	using the cache		gradient descent (from scratch)	$O(\frac{m}{n+1}(N_1+N_2)d)$	$O(m(N_1+N_2)d)$	$(\|\mathscr{E}\|+\|\mathscr{R}\|)d$

GAN-based works [11, 54], our method in Algorithm 2 acts like a discriminative and distilled model of GAN, and it only cares about negative triplets leading to large gradient norms during the training. Thus, the proposed NSCaching algorithm not only has fewer parameters, but also can be easily trained from randomly initialized models (from the scratch). Moreover, experimental results in Section 5 show that NSCaching achieves the best performance.

However, in order to achieve a good performance, we need to carefully design how to sample from the cache (step 3) and how to update the cache (step 5). In these steps, "*exploration and exploitation*" (E&E) [34] is the main concern. Specifically, how to keep the balance between *exploration* (explore all the possible large-gradient negative triplets in $\bar{\mathscr{S}}$) and *exploitation* (sample the training pair leading to the largest gradient norm in cache $\mathscr{C}$).

1). Sampling from the cache (step 3). Before describing the sampling scheme, we introduce some notations for subsequent usage. Let $\mathbf{c}^{(i)}$ be a N_1-dimensional vector containing gradient norms of $(\bar{h}_{i_j}, r_i, \bar{t}_{i_j}) \in \mathscr{C}_i, j = 1 \dots N_1$. Thus, if the j-th element $\mathbf{c}_j^{(i)}$ is larger, it means that the j-th negative triplet $(\bar{h}_{i_j}, r_i, \bar{t}_{i_j})$ in cache is of higher quality. Finally, we further define a vector $\mathbf{p}$, of which the length is the number of positive triplets $\mathscr{S}$ and each element $\mathbf{p}_i = \|\mathbf{c}^{(i)}\|_2$. Intuitively, if $\mathbf{p}_i$ is larger, then (h_i, r_i, t_i) is more likely to have more large-gradient negative triplets.

As in Algorithm 2, we need to sample a pair of positive and negative triplets. Based on above notations, we can do it as follows. First, we can pick up a positive triplet $(h_i, r_i, t_i) \in \mathscr{S}$ following a probability distribution given by

$$p((h_i, r_i, t_i)) = \sigma_1(\mathbf{p}_i; \mathbf{p}), \tag{6}$$

where the distribution $\sigma_1(\mathbf{p}_i; \mathbf{p})$ satisfies that

Algorithm 3 Sampling from the cache (step 3).

Require: Training set $\mathscr{S}$ and cache $\mathscr{C}$.
1: sample a positive triplet $(h_i, r_i, t_i) \in \mathscr{S}$ according to $p((h_i, r_i, t_i))$ in (6);
2: index the specific cache $\mathscr{C}_i$ of (h_i, r_i, t_i);
3: sample a negative triplet $(\bar{h}_j, r_j, \bar{t}_j)$ from $\mathscr{C}_i$ according to $p((\bar{h}_{i_j}, r_i, \bar{t}_{i_j}))$ in (7).

– $\sum_a \sigma_1(\mathbf{p}_a; \mathbf{p}) = 1$; and $\sigma_1(\mathbf{p}_a; \mathbf{p}) \geq \sigma_1(\mathbf{p}_b; \mathbf{p})$ if $\mathbf{p}_a \geq \mathbf{p}_b$.

In this way, (h_i, r_i, t_i) will be more frequently sampled if $\mathbf{p}_i$ is larger. Then, after picking up the positive triplet (h_i, r_i, t_i), we sample the negative triplet $(\bar{h}_{i_j}, r_i, \bar{t}_{i_j}) \in \mathscr{C}_i$ following

$$p\left((\bar{h}_{i_j}, r_i, \bar{t}_{i_j})\right) = \sigma_2(\mathbf{c}_j^{(i)}; \mathbf{c}^{(i)}), \tag{7}$$

where $\sigma_2(c)$ is defined in the same way as $\sigma_1(p)$. The full procedures are shown in Algorithm 3.

Remark 1 The choice of σ is important, as it greatly affects E&E and how we can adapt to the sampling distributions. Let us consider two extreme examples. First, if we pick σ_1 as an indicator function (as in Figure 3(c)) where $\sigma_1(\mathbf{p}_i; \mathbf{p}) = 1$ if $\mathbf{p}_i$ is the largest and $\sigma_1(\mathbf{p}_j; \mathbf{p}) = 0$ for $j \neq i$. Then, it is equal to deterministically select the negative triplet (h_i, r_i, t_i) with the highest-quality. However, as the distribution can change during iterations of the algorithm, both of the embedding quality and the negative triplets in the cache may not be accurate enough for the sampling in the latest iteration. Besides, consistently sampling the largest one may make the algorithm only focus on a small amount of triplets, failing to capture the distribution well. Thus, we also need to consider the other candidates except the one with the largest $\mathbf{p}_i$. Second, if we take $\sigma_1(\mathbf{p}_i; \mathbf{p}) = 1/N$ for any $i \in \{1, \dots, N\}$ (as in Figure 3(a)) where N is the total number of training triplets (i.e., uniform sampling is used), then all triplets have equal possibilities to be sampled. However, this ignores the difference of candidates. The two cases also adapt to σ_2 when negative triplets are sampled from $\mathscr{C}_i$. In Section 3.4, we will propose a novel method to balance E&E automatically.

2). Updating the cache (step 5). As mentioned in Section 3.1, the cache needs to be dynamically changed during iterations of the algorithm. Otherwise, the content in cache will not be changed and the sampling will be highly biased since most of the negative triplets will not be visited. Thus, we need to refresh the cache periodically. Moreover, the cache needs to be updated in an efficient way.

As in (5), the number of negative triplets in $\bar{\mathscr{S}}_i$ is quite large for a given positive triplet (h_i, r_i, t_i). However, it is impossible for us to evaluate all the candidates in $\bar{\mathscr{S}}_i$. Since

Algorithm 4 Updating the cache (step 4).

Require: cache $\mathscr{C}_i$ of size N_1.
1: initialize $\hat{\mathscr{C}}_i \leftarrow \emptyset$, $\hat{\mathbf{c}}^{(i)} = \mathbf{0}$.
2: sample a subset $\mathscr{N}_i \subset \bar{\mathscr{T}}_i$ with N_2 triplets;
3: compute $\|\mathbf{g}_k\|_2$ for all $(\bar{h}, r, \bar{t}) \in \mathscr{N}_i \cup \mathscr{C}_i$;
4: **for** $j = 1, \cdots, N_1$ **do**
5: sample $(\bar{h}_{i_k}, r_i, \bar{t}_{i_k})$ with probability in (8);
6: remove $(\bar{h}_{i_k}, r_i, \bar{t}_{i_k})$ from $\mathscr{N}_i \cup \mathscr{C}_i$;
7: $\hat{\mathscr{C}}_i \leftarrow \hat{\mathscr{C}}_i \cup \{(\bar{h}_{i_k}, r_i, \bar{t}_{i_k})\}$;
8: $\hat{\mathbf{c}}_k^{(i)} = \|\mathbf{g}_k\|_2$;
9: **end for**
10: update by $\mathscr{C}_i \leftarrow \hat{\mathscr{C}}_i$.
11: **return** $\mathscr{C}_i \leftarrow \hat{\mathscr{C}}_i$ and $\mathbf{c}^{(i)} \leftarrow \hat{\mathbf{c}}^{(i)}$.

we want to efficiently capture the large-gradient negative triplets in $\mathscr{C}_i$, we sample a small subset $\mathscr{N}_i \subset \bar{\mathscr{T}}_i$ of size N_2, with $N_2 \ll |\bar{\mathscr{T}}_i|$. Then for each $(\bar{h}_{i_k}, r_i, \bar{t}_{i_k}) \in \mathscr{N}_i \cup \mathscr{C}_i$, we evaluate the gradient norm $\|\mathbf{g}_k\|_2$ by (3) or (4). Then we construct a new set $\hat{\mathscr{C}}_i \subset \mathscr{N}_i \cup \mathscr{C}_i$, whose components are sampled from $\mathscr{N}_i \cup \mathscr{C}_i$ without replacement N_1 times following the probability distribution

$$p\left((\bar{h}_{i_k}, r_i, \bar{t}_{i_k})\right) = \sigma_3\left(\mathbf{g}_k; \mathbf{g}\right). \tag{8}$$

Finally, $\hat{\mathscr{C}}_i$, which contains N_1 negative triplets and their corresponding gradient norms $\hat{\mathbf{c}}^{(i)}$, are returned.

Remark 2 Exploration and exploitation also need to be carefully balanced in Algorithm 4. As the cache needs to be updated, we have to sample from $\bar{\mathscr{T}}_i$. The subset $\mathscr{N}_i$ is chosen as a substitute of $\bar{\mathscr{T}}_i$ in consideration of efficiency. Therefore, a bigger N_1 implies more exploitation, while a larger N_2 leads to more exploration. In step 5, indeed, the choice of σ_3 is important under the same consideration as σ_1 and σ_2. The balance of E&E on N_1, N_2 and σ_3 is further discussed in Section 3.4.

3.2.3 Space and Time Complexities

In this part, we analyze the space and time complexities of NSCaching (Algorithm 2). Comparing with basic training framework in Algorithm 1, the main additional cost by introducing cache comes from step 5 in Algorithm 2, i.e. updating the cache using Algorithm 4. In Algorithm 4, the main time cost comes from computing the gradients $\|\mathbf{g}_k\|_2$ for $N_1 + N_2$ training pairs, whose complexity is $O((N_1 + N_2)d)$. The cost of step 3 in Algorithm 2 is rather small, which comes from importance sampling according to the gradient norms. This part takes $O(N_1)$ time. Hence, the total cost by introducing the cache is $O((N_1 + N_2)d)$ for a single training pair. In practice, we can lazily update the cache every n epochs rather than do immediate updating, which can further reduce the updating complexity to $O\left((N_1+N_2)d/(n+1)\right)$.

As for the space complexity, evaluating the gradients for $N_1 + N_2$ training pairs takes $O((N_1 + N_2)d)$ space. Since we only store indexes in the cache, it takes $O(|\mathscr{S}|N_1)$ space to store these indexes for negative triplets. Note that, N_1 is small since large-gradient negative triplets are rare. This is also verified in our experiments in Section 5.4.3.

In comparison, to generate a training pair, the generator in IGAN [54] takes $O(|\mathscr{E}|d)$ time since it needs to compute the distribution over all entities. KBGAN [11] needs $O(N_1 d)$ time to measure a candidate set of N_1 triplets. The additional space cost for IGAN and KBGAN is also $O(|\mathscr{E}|d)$ and $O(N_1 d)$ respectively. Finally, the comparison of space and time complexities is summarized in Table 3 with TransE as the scoring function.

3.3 Theoretical Analysis

In this part, we theoretically analyze the convergence and learning performance of the proposed method.

3.3.1 Convex Case: Faster convergence

Before presenting our analysis, we first simplify and take a uniform treatment over (1) and (2). Let $\mathbf{w} = \{\mathbf{h}, \mathbf{r}, \mathbf{t}\}$, then we can take the loss $\phi_i(\mathbf{w})$ on a training pair for translational distance model as

$$\phi_i(\mathbf{w}) = \left[\gamma - f(h_i, r_i, t_i) + f(\bar{h}_i, r_i, \bar{t}_i)\right]_+, \tag{9}$$

and for semantic matching model as

$$\phi_i(\mathbf{w}) = \ell\left(1, f(h_i, r_i, t_i)\right) + \ell\left(-1, f(\bar{h}_i, r_i, \bar{t}_i)\right). \tag{10}$$

Thus, we can express (1) and (2) as

$$\min_{\mathbf{w}} F(\mathbf{w}) \equiv \frac{1}{n}\sum_{i=1}^{n} \phi_i(\mathbf{w}), \tag{11}$$

where n is the number of all the training pair of positive and negative triplets. Using above notation, we can abstract NSCaching (Algorithm 2) as in Algorithm 5. Basically, the cache scheme is used to generate a probability distribution $\mathbf{p}^t$ over all ϕ_i, which changes over iterations.

Algorithm 5 Abstraction of NSCaching.

1: **for** $t = 1, \cdots, T$ **do**
2: sample ϕ_{i_t} from $\{\phi_i\}_{i=1}^n$ based on $\mathbf{p}^t$;
3: update $\mathbf{w}^{t+1}$ by $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta (n\mathbf{p}_i^t)^{-1} \nabla \phi_{i_t}(\mathbf{w}^t)$;
4: **end for**

The convergence of Algorithm 5 is in Theorem 1, which is inspired by some recent works in stochastic optimization [40, 70].

Theorem 1 *If F is smooth and convex, then*

$$\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}\left[F(\mathbf{w}^t)\right]-\mathbb{E}\left[F(\mathbf{w}^*)\right] \leq \frac{2}{\eta T}\|\mathbf{w}^*-\mathbf{w}^t\|^2+\frac{\eta}{2\sigma T}\sum_{t=1}^{T}\mathbb{E}\left[\|\nabla\phi_{i_t}(\mathbf{w}^t)/np_{i_t}^t\|^2\right], \quad (12)$$

where $\mathbf{w}^*=\arg\min_{\mathbf{w}}F$, η is the step-size for stochastic optimization, and expectation is taken w.r.t. distribution $\mathbf{p}^t$.

The proof is in Appendix A. As we can see, how fast and well $\mathbf{w}^t$ converges to the optimal solution depends on $\mathbf{p}^t$ via the second term in (12). The solution which minimizes this term is offered in Proposition 1.

Proposition 1 ([70]) $\mathbb{E}\left[\|\nabla\phi_{i_t}(\mathbf{w}^t)/np_{i_t}^t\|^2\right]$ *is minimized when the possibility $\mathbf{p}^t$ follows $\mathbf{p}_i^t=\|\nabla\phi_i(\mathbf{w}^t)\|/\sum_{j=1}^{n}\|\nabla\phi_j(\mathbf{w}^t)\|$.*

Since the cache scheme is used to avoid vanishing gradient problem and distill the full sampling distribution, the samples with larger $\|\nabla\phi_i(\mathbf{w}^t)\|$ should have higher possibility to be sampled. If the dynamic distribution $\mathbf{p}^t$ is captured, we can then adaptively sample from it. In other words, the sample i_t with larger $\mathbf{p}_{i_t}^t$ has larger possibility to be sampled. As a result, the last term in (12) in NSCaching can have smaller value compared with the uniform sampling. This indicates that NSCaching has both faster convergence speed and smaller approximation error. In practice, most of the existing embedding models are non-convex and stochastic optimization [28] is used to update the parameters. However, the above bound still offers insights on how the proposed method works.

3.3.2 Nonconvex Case: Self-paced learning

The main idea of self-paced learning (or curriculum learning) [3, 31] is to pick up easy samples first, and then gradually switch to harder ones. In this way, the classifier can firstly identify the rough position where the decision boundary should locate. Then the boundary can be further refined by the hard examples. It is effective for complex and noncovex models. Recently, self-paced learning is also introduced into graph embedding and the improvement on the quality of embeddings has been reported [18]. Besides, GAN is also used to monitor the distribution of edges in the network, and negative edges with scores above a given threshold are sampled from the generator in GAN. Self-paced learning is achieved by increasing the threshold during the training of embedding [18]. As a comparison, the GAN models used in KBGAN and IGAN are not benefited from self-paced learning.

In contrast, our caching scheme can explicitly benefit from it. The reason is that the embedding model only has weak discriminative ability in the beginning of the training. Thus, while there exist a lot of negative triplets leading to large gradient norms, it is more likely that they are easy ones as most of the negative samples are easily classified. As training process continuous, those easy samples will gradually have small gradients and are removed from the cache. These mean NSCaching will learn from easy samples first, but then gradually focus on hard ones, which is exactly the principle of self-paced learning. The above explanations are also verified by experiments, where we can see the negative triplets in the cache change from easy to hard ones (Section 5.6) and NSCaching (training from scratch) can already achieve better performance than IGAN and KBGAN with pretraining (Section 5.3).

3.4 Automatic Balancing Exploration and Exploitation

In previous parts, we have described the proposed framework (Section 3.2) and analyzed why it works (Section 3.3). Based on Proposition 1, we aim to capture the dynamic sampling distribution $\mathbf{p}_i^t=\|\nabla\phi_i(\mathbf{w}^t)\|/\sum_{j=1}^{n}\|\nabla\phi_j(\mathbf{w}^t)\|$. To guarantee efficiency, we design the sampling scheme (Algorithm 3) and updating scheme (Algorithm 4) to distill such a distribution. However, the distributions for different tasks and for different training status are different in practice. Therefore, we should carefully adjust the sampling and updating schemes. As mentioned in Section 3.2, to achieve better performance for different scenarios, the remained question is how can we carefully balance E&E? Here, we show how AutoML techniques can be combined with the proposed framework to automatically balance E&E.

3.4.1 Search Space from NSCaching

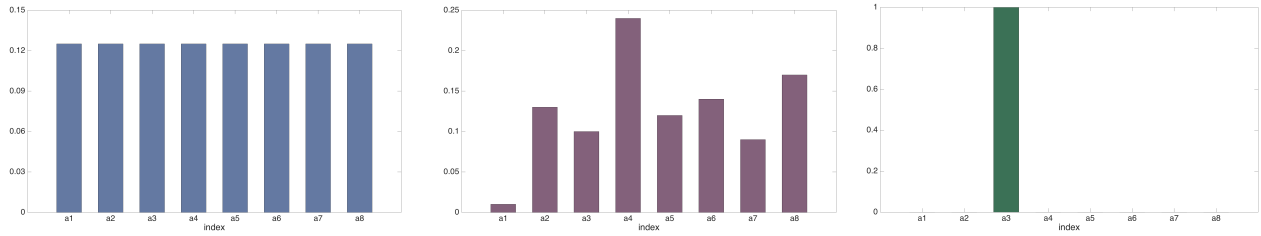
From Remarks 1, 2 and Proposition 1, we can see that $\sigma(\mathbf{a}_i;\mathbf{a})$ needs to cover three special cases, i.e., (i). uniformly sampling on all elements, (ii). deterministically sampling the max, and (iii). importance sampling as Proposition 1. Thus, we are motivated to choose the weighted softmax distribution as the probability function

$$\sigma(\mathbf{a}_i;\mathbf{a})=\exp(\alpha\cdot\mathbf{a}_i)/\sum_j\exp(\alpha\cdot\mathbf{a}_j), \quad (13)$$

where $\alpha\geq 0$ is a hyper-parameter to be tuned. We can see $\alpha=0$ covers (i) as in Figure 3(a), $\alpha=\infty$ covers (ii) as in Figure 3(c), and other values of α cover (iii) as in Figure 3(b). Specifically, we use three different α's for (6), (7) and (8) respectively.

3.4.2 Search by Bayesian Optimization

All hyper-parameters balancing E&E are summarized in Table 4. Manually tuning these hyper-parameters is time consuming. Simple search approaches such as grid search and random search are usually inefficient. Inspired by the



(a) uniform function $\alpha = 0$ (b) general softmax $\alpha = 1$ (c) indicator function $\alpha = \text{inf}$

Fig. 3: Example of the different distributions of σ, where x-axis indicates each dimension of $\mathbf{a}$ in (13) and y-axis is the sampling probability.

Table 4: How exploration and exploitation (E&E) in Algorithm 2 are affected by hyper-parameters.

hyper-parameters	functionality	larger leads to
α_1	positive sample (h_i, r_i, t_i)	exploitation
α_2	negative sample $(\bar{h}_i, r_i, \bar{t}_i)$	exploitation
α_3	update $\mathscr{C}_i$	exploitation
N_1	size of cache $\mathscr{C}_i$	exploitation
N_2	size of subset $\mathscr{N}_i$	exploration

choice of (13), and the recent success of automated machine learning (AutoML) [61], especially hyper-parameter optimization, we use a versatile hyper-parameter optimization method, i.e., Sequential Model-based optimization for general Algorithm Configuration (SMAC) [23]. SMAC allows efficiently and automatically tuning of both discrete (N_1 and N_2) and continuous (α_1, α_2 and α_3) hyper-parameters.

3.4.3 Discussion: connection with existing methods

The hyper-parameters in Table 4 can not only help us balance E&E, but also give a unified view of the baselines, namely covering Bernoulli, KBGAN and IGAN as special cases. In this part, α_1 is always 0 since none of the three methods consider non-uniform positive sampling.

- *Bernoulli* [56]: In Bernoulli sampling, negative samples are generated uniformly from the whole candidate space. In this case, we can set both α_2 and α_3 to be 0. Then, the cache updating is never dependent on scores as well as the sampling schemes. Therefore, Bernoulli sampling is a special case of NSCaching when $\alpha_2 = \alpha_3 = 0$;
- *KBGAN* [11]: The key thought in KBGAN is to use a generator to pick up large-gradient negative samples in a random subset $\mathscr{N}_i \in \bar{\mathscr{T}}_i$. Given a positive triplet (h_i, r_i, t_i), the scores $\mathbf{c}_j^{(i)}$ stored in cache work as an alternative of the generator in KBGAN to measure the quality of $(\bar{h}_j, r_j, \bar{t}_j)$. Different from standard NSCaching, we use $\alpha_3 = 0, \alpha_2 > 0$ and $N_2 = \max_i\left(|\bar{\mathscr{T}}_i|\right)$ so that the content in cache $\mathscr{C}_i$ is similar as that in $\mathscr{N}_i$ of KBGAN. Self-Adv [48] uses in the similar approach as KBGAN. Differently, Self-Adv uses the model's embedding itself to measure the quality of negative samples in $\mathscr{N}_i$. Compared with KBGAN and Self-Adv, NSCaching improves upon them by controlling the quality of negative triplets in $\mathscr{N}_i$.
- *IGAN* [54]: When $N_1 = \max_i\left(|\bar{\mathscr{T}}_i|\right)$ and $N_2 = 0$, NSCaching resembles IGAN. In IGAN, the generator chooses large-gradient negative samples from the entire candidate set $\bar{\mathscr{T}}_i$. Thus, we can set the cache size to be $\max_i\left(|\bar{\mathscr{T}}_i|\right)$, and mask the positive positions. Besides, the cache does not need to be updated by setting $N_2 = 0$. We use $\alpha_2 > 0$ to select large-gradient negative samples from cache to replace the generator. In this way, NSCaching can also approximate the sampling procedure in IGAN.

Table 5: A unified view of the proposed method and the existing methods. "-" means not care.

hyper-parameters	Bernoulli [56]	KBGAN [11] / Self-Adv [48]	IGAN [54]
α_1	0	0	0
α_2	0	> 0	> 0
α_3	0	0	-
N_1	-	-	$\max_i\left(\lvert\bar{\mathscr{T}}_i\rvert\right)$
N_2	-	$\max_i\left(\lvert\bar{\mathscr{T}}_i\rvert\right)$	0

We show the values of α's and N's in Table 5 about how to cover the baselines by NSCaching. This finding also explains why NSCaching (auto) is better than the baselines. Besides, comparing with Bernoulli, KBGAN, IGAN and Self-Adv, NSCaching (auto) adapts the sampling distribution to approximate a relatively unbiased estimator for specific tasks.

3.5 Understanding the false negative triplets.

Since KG is incomplete [55], there exist triplets that do not appear in the training set but are not necessarily false. For

example, the triplets in the valid or test set can be viewed as false negative triplets during training. From the literature [33,55,59], the false negative samples will have larger scores but lower variance than the large-gradient negative samples. We also have such an observation in Section 5.5.3. Thus, the false negative triplets may be detected by tracing the variance during the training. However, since the number of false negative triplets is small and evaluating the variance is expensive, we show in Section 5.5.3 that false negatives are empirically not a concern.

4 Extension to Skip-gram Model

The skip-gram model [36] is a popular method originating from word embedding, which can explore surrounding words given the current embedded one. Due to its superior performance in natural language processing, two representative methods Deepwalk [44] and Node2vec [21] adopt the skip-gram model for graph embedding. And they have achieved significant improvements over previous works on the embedding quality [10]. Moreover, motivated by the success of the skip-gram model on the graph and word embedding [21,36,37,44], we extend the NSCaching algorithm to the skip-gram model here. We first introduce the skip-gram model [36] in Section 4.1. Then, we adapt the cache-based negative sampling algorithm to the skip-gram model for graph embedding in Section 4.2. In this way, we show that the NSCaching algorithm (Algorithm 2) is not limited to KG embeddings.

4.1 Negative Sampling in Skip-gram Model

Skip-gram model is originally used to learn word embeddings [36]. It aims at maximizing the co-occurrence probability among the words that appear within a window $\mathscr{W}$. Given a positive word u, the training objective in skip-gram model is to learn word embeddings that are good at predicting the words $v \in \mathscr{W}_u$, where $\mathscr{W}_u$ is a set of nearby or context words of u. More formally, for a sequence of training words $u_1, u_2, \ldots, u_L$ with length L, the objective is to maximize the average log probability

$$\frac{1}{L}\sum_{i=1}^{L}\sum_{v_j \in \mathscr{W}_{u_i}} \log p(v_j|u_i), \tag{14}$$

where $\mathscr{W}_{u_i} = \{v_j | -c \le j \le c, j \neq 0\}$, c is a pre-defined window size. Basically, the probability $p(v_j|u_i)$ is defined as the softmax function

$$p(v_j|u_i) = \exp(\boldsymbol{v}_j^\top \boldsymbol{u}_i) / \sum_{k=1}^{|V|} \exp(\boldsymbol{v}_k^\top \boldsymbol{u}_i), \tag{15}$$

where the boldface represents the embedding and $|V|$ is the vocabulary size. However, since $|V|$ is usually large, negative sampling is used to avoid computing the dot product similarity among all the words [37]. In this way, the log probability $\log p(v_j|u_i)$ is computed based on

$$\log \sigma(\boldsymbol{v}_j^\top \boldsymbol{u}_i) + \sum_{n=1}^{N} \log \sigma(-\boldsymbol{v}_n^\top \boldsymbol{u}_i), \tag{16}$$

where $\sigma(x) = 1/(1+\exp(-x))$ is the sigmoid function and v_n's are the negative samples drawn from the noise distribution $p(u_i)$. Generally, the noise distribution $p(u_i)$ is a unigram distribution, or a weighted distribution proportional to the word frequency [37]. Similar as the methods in Section 2.2.1, the negative sampling used for (16) is also fixed. Hence, the quality of negative samples cannot be dynamically captured.

4.2 Graph Embedding with Skip-gram Model

In order to preserve the graph structure while make it easy for a machine learning model to process, random walks are widely used to learn graph embeddings [10,21,44]. A graph is firstly represented as a set of random walk paths sampled from it. Then skip-gram model [36] is applied to preserve graph properties carried by the paths [10].

DeepWalk [44], as a representative random walk based graph embedding model, first samples a set of paths from the input graph. Then, the sampled paths are regarded as sentences that describe the graph, and the nodes are regarded as words. Skip-gram model is applied on the paths to maximize the probability of observing a node's neighborhood conditioned on its embedding. In this way, nodes with similar neighborhoods will have larger co-occurrence and more similar embeddings. Node2vec [21] improves upon DeepWalk [44] by using a biased random walk. Two parameters p and q are used to control breadth-first sampling or depth-first sampling, which is shown to better capture the local topologies [21].

Different from KG, skip-gram model does not have exact positive samples since the context in the window $\mathscr{W}_{u_i} = \{v_j | -c \le j \le c, j \neq 0\}$ does not necessarily to have strong connection with u_i. Therefore, we build a cache $\mathscr{C}_i$ for each word rather than each training sample. Then, the negative part in (16) is sampled from the cache. We treat the number of negative samples N as a hyper-parameter and optimize it together with the cache-related hyper-parameters. In addition, we use all the nodes that are not in $\mathscr{W}_{u_i}$ as the negative samples. Therefore, balancing between E&E is also important in this setting.

We use the Node2vec [21] model as the testbed for skip-gram model. In Node2vec, a biased random walk method is used to generate a sequence of walks from the graph. In this way, embeddings are updated through the cache-based skip-gram algorithm. The cache-based Node2vec method is given in Algorithm 6.

Algorithm 6 Node2vec (NSCaching).

Require: Graph $G = (V, E)$, embedding dimension d, walks per node r, walk length l, window size w, p, q.
Initialize embeddings for each node $v \in V$.
computing the neighborhood sampling probability of each node based on p and q;
Initialize *walks* to empty
for $i = 1, \cdots, r$ **do**
  for all nodes $v \in V$ **do**
    sample a *walk* starting from u with length l;
    append *walk* to *walks*.
  end for
end for
repeat
  sample a node u_i and its context v_j in the window $\mathscr{W}_{u_i}$.
  sample a set of negative nodes $\bar{v}_n$'s from the cache $\mathscr{C}_i$.
  update embedding using the gradient of (16).
until converge

5 Experiments

In this section, we conduct empirical study of our method. All algorithms are written in Python with PyTorch framework [43] and run on a TITAN Xp GPU with 12GB memory. Our code is public available in xxx.

5.1 Implementation Details

Since a lot of triplets share the same *(head, relation)* or *(relation, tail)* pairs, we use two caches, namely a head cache $\mathscr{H}_{(r,t)}$ and a tail cache $\mathscr{T}_{(h,r)}$, to separately store negative triplets in $\{(\bar{h}, r, t) \notin \mathscr{S} | \bar{h} \in \mathscr{E}\}$ and $\{(h, r, \bar{t}) \notin \mathscr{S} | \bar{t} \in \mathscr{E}\}$. Using two caches instead of one can help us to reduce the time and space cost. The value we stored in cache is the score of negative triplets according to the predefined scoring function f, instead of gradient norms. The main consideration is that gradient norms for each training pair can not be efficiently obtained through mini-batches, especially for complex scoring functions like TransD [25]. Given a positive triplet (h_i, r_i, t_i), the value $\mathbf{p}_i$ is computed by the sum of scores stored in $\mathscr{H}_{(r,t)}$ and $\mathscr{T}_{(h,r)}$.

In general, the training of KG embedding model is under the open world assumption, which means that KGs contain only positive triplets and non-observed triplets can be either false or just missing [55]. To reduce sampling the negative examples that are just missing, we use the same scheme proposed in Bernoulli sampling [56] to get the subset $\mathscr{N}_i$. Specifically, different probabilities are given when replacing the head or the tail for different relations. For each relation r, we compute and denote the average number of tail entities per head as tph, and the average number of head entities per tail as hpt. Then the probabilities of replacing the head and the tail are $tph/(tph+hpt)$ and $hpt/(tph+hpt)$ respectively.

To constrain values of α's in a certain range, we rescale the value of $\mathbf{p}_i$, $\mathbf{c}_j^{(i)}$ and $\mathbf{g}_i$ to lie in the interval $[0, 1]$ before computing the sampling probability $\sigma(\mathbf{a}_i; \mathbf{a})$. Specifically, given the vector $\mathbf{a}$ and let q^{low} and q^{high} be the quantiles of $\mathbf{a}$, we choose q^{low} to be 20^{th} and q^{high} to be 80^{th} percentiles, respectively. Then the rescaling function $r(\mathbf{a}_i)$ is formed as:

$$r(\mathbf{a}_i) = \begin{cases} 1 & \mathbf{a}_i > q^{\text{high}} \\ 0 & \mathbf{a}_i < q^{\text{low}} \\ \frac{\mathbf{a}_i - q^{\text{low}}}{q^{\text{high}} - q^{\text{low}}} & \text{otherwise} \end{cases}. \tag{17}$$

The rescaling function can also help us to avoid the case that some samples have extremely large score. In this case, these samples will be selected for too many times.

5.2 Experiment Setup

Five datasets are used here, i.e., WN18, FB15K and their variants WN18RR, FB15K237, and YAGO3-10. WN18 and FB15K are firstly introduced in [8]. They are widely tested in the literature [8, 11, 25, 27, 51, 54, 69]. WN18RR [13] and FB15K237 [50] are variants that remove near-duplicate or inverse-duplicate relations from WN18 and FB15K. The two variants are harder and more realistic. YAGO3-10 is much larger than the others and is a subset of YAGO [47]. Their statistics are shown in Table 6.

Table 6: Detailed information of the datasets used in KG embedding experiments.

Dataset	#entity	#relation	#train	#valid	#test
WN18	40,943	18	141,442	5,000	5,000
WN18RR	40,943	11	86,835	3,034	3,134
FB15K	14,951	1,345	484,142	50,000	59,071
FB15K237	14,541	237	272,115	17,535	20,466
YAGO3-10	123,188	37	1,079,040	5,000	5,000

Following previous KG embedding works [8, 25, 51, 56] and the GAN-based works [11, 54], we mainly test the performance on *link prediction* task. This is also the testbed to measure KG embedding models. Link prediction aims to predict the missing entity h or t for a positive triplet (h, r, t). In this task, we measure the rank of head entity h and tail entity t among all the entities in $\mathscr{E}$. Thus, link prediction emphasizes the rank of the correct entities rather than their concrete scores. Besides, to further verify the quality of the learned embedding, we test the learned embeddings on *triplet classification* task. This task is to confirm whether a given triplet (h, r, t) is correct or not, i.e., binary classification of triplet [56]. In practice, it can help us to quickly answer the truth-or-false questions.

As in previous works [8, 11, 27, 51, 54], we evaluate the link prediction performance based on the following two metrics [1]:

- Mean reciprocal ranking (MRR): It is computed by the average of the reciprocal ranks $1/|\mathscr{S}|\sum_{i=1}^{|\mathscr{S}|}\frac{1}{\text{rank}_i}$ where $\text{rank}_i, i \in \{1,\dots,|\mathscr{S}|\}$ is a set of ranking results;
- Hit@10: The percentage of appearance in the top-10 ranking: $1/|\mathscr{S}|\sum_{i=1}^{|\mathscr{S}|}\mathbb{I}(\text{rank}_i \leq 10)$, where $\mathbb{I}(\cdot)$ is the indicator function;

MRR and Hit@10 measure the top rankings of positive entity in different levels. Hit@10 cares about general top rankings while the top 1 samples contribute most to MRR. The larger value of MRR and Hit@10 indicates better performance. To avoid underestimating the performance of different models, we report the performance in a "filtered" setting, i.e., all the corrupted triplets that exist in train, valid and test set are filtered out [11, 54]. A large amount of scoring functions have been proposed in the literature, please see a recent survey [55] for a review. In this part, following [11, 54], TransE [8], TransH [56], TransD [25], DistMult [60] and ComplEx [51] will be used as scoring functions for comparison; besides, the recently developed scoring function SimplE [27] is also included (see Table 2).

5.3 Comparison with State-of-the-arts

In this section, we focus on the comparison between our proposed cache-based negative sampling with the other baseline sampling methods.

5.3.1 Compared Methods

Following methods for negative sampling in KG embedding are compared:

- *Bernoulli* [56]: As an extension of the uniform sampling scheme used in TransE, Bernoulli sampling controls the probability for sampling $(\bar{h},r,t)$ or $(h,r,\bar{t})$ in the one-to-many, many-to-one and many-to-many relations. Specifically, it samples $(\bar{h},r,t)$ or $(h,r,\bar{t})$ under a fixed Bernoulli distribution for each r.
- *KBGAN* [11][2]: This method firstly samples a set $\mathscr{N}$ uniformly from the whole entity set $\mathscr{E}$. Then the head or tail entity is replaced with the entities in $\mathscr{N}$ to form a set of candidate $(\bar{h},r,t)$ and $(h,r,\bar{t})$. The generator in KBGAN tries to pick up one triplet among them. As proposed in [11], we choose the simplest model TransE as the generator. For a fair comparison, the size of set $\mathscr{N}$ is the same as our cache size N_1. We use the published code and change the configure same as ours in the comparison. *Self-Adv* [48] works similarly as KBGAN. The main difference is that Self-Adv uses the target embedding model itself as the generator.
- *NSCaching* (Algorithm 2): As in Section 3 and 5.1, the negative triplets are formed by replacing the head entity h or tail entity t with the one sampled from head-cache $\mathscr{H}$ or tail-cache $\mathscr{T}$. The cache is updated as in Algorithm 4. Note that we can also lazily update the cache several iterations later to save time. We use $n = 0$ without lazy-update unless otherwise specified. Besides, we use *AutoML* to denote the improved version which tunes the hyper-parameters to balance E&E.

As the source code of *IGAN* [54] is not available, we do not compare with it here. Instead, we directly use the reported performance in the sequel. Finally, we also use Bernoulli sampling to choose between $(\bar{h},r,t)$ and $(h,r,\bar{t})$ for *KBGAN* and *NSCaching*. Besides, as in [11, 54], two strategies are used for *KBGAN* and *NSCaching*:

- *Scratch*: The embedding of relations and entities are initialized by the Xavier uniform initializer [19], and the models (denoted as *KBGAN + scratch* and *NSCaching + scratch*) are directly applied to train the given KG;
- *Pretrain*: Same as [11, 54], we firstly pretrain each scoring function with *Bernoulli* sampling, several epochs on the data sets. We denote it as *pretrained*. Then the obtained parameters are used to warm-start the given KG. We keep training the warm-started KG embedding and evaluate the performance under different sampling methods, i.e., *Bernoulli*, *KBGAN + pretrain* and *NSCaching + pretrain*. Besides, the generator in *KBGAN* is warm-started with corresponding TransE model.

Same as the KG embedding works in the literature [8, 26, 27], we use grid search to select the KG embedding related hyper-parameters: hidden dimension $d \in \{50, 100, 200\}$, batch size $m \in \{1024, 2048, 4096\}$, learning rate $\eta \in \{0.0001, 0.0003, 0.001, 0.003, 0.01, 0.03, 0.1\}$. For translational distance models, we tune the margin value $\gamma \in \{1, 2, 3, 4\}$. And for semantic matching models, we tune the penalty value $\lambda \in \{0.001, 0.01, 0.1\}$ [51]. We use Adam [28], which is a popular variant of SGD algorithm for the training, and adopt its default settings except for the learning rate. The best hyper-parameters are tuned under Bernoulli sampling scheme and evaluated by the MRR metric on the valid set. We keep them fixed for the baselines *Bernoulli*, *KBGAN* and *NSCaching* here. Following [11], we save and record the *pretrained* model after initial training epochs. Then, *Bernoulli* method keeps training until 3000 epochs; and the results of *KBGAN* and *NSCaching* algorithm are evaluated within 1000 epochs, either from scratch or with pretraining. All the recorded results are tested based on

[1] The mean rank (MR) metric, which is given in the conference version [69], is removed in the journal version since (i) space is limited. and (ii) the mean rank is easily influenced by low ranking samples.

[2] https://github.com/cai-lw/KBGAN

Table 7: Comparison of various algorithms on the five data sets. Performance of the pretrained model is included as reference. As code of IGAN is not available, its performance is directly copied from [54]. Note that MRR, and those on WN18RR, FB15K237, YAGO3-10 data sets are not reported as they are not shown in IGAN. Besides, KBGAN on YAGO3-10 is not reported since it easily runs out of memory. Bold number means the best performance, and underline means the second best under each setting.

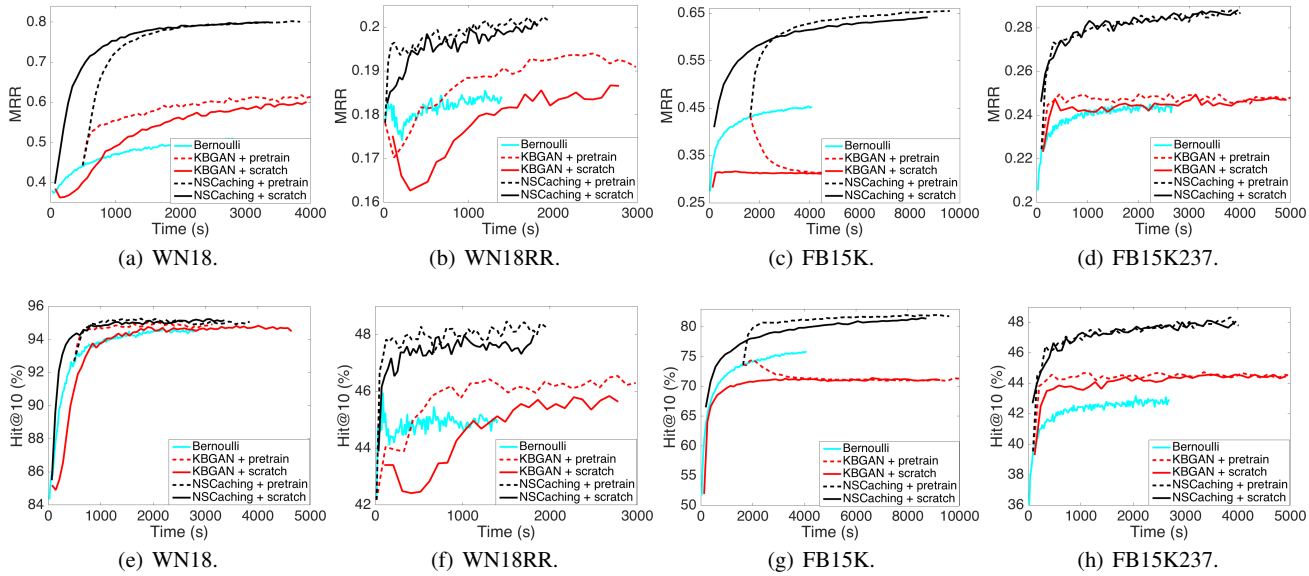
scoring functions	Dataset		WN18		WN18RR		FB15K		FB15K237		YAGO3-10	
	Metrics		MRR	Hit@10	MRR	Hit@10	MRR	Hit@10	MRR	Hit@10	MRR	Hit@10
TransE	pretrained		0.4213	91.50	0.1753	44.48	0.4679	74.70	0.2262	38.64	0.1723	32.24
	Bernoulli		0.5001	94.13	0.1784	45.09	0.4951	77.37	0.2556	41.89	0.2053	37.87
	KBGAN	pretrain	0.6880	**94.92**	0.1864	45.39	0.4858	77.02	0.2938	46.69	——	——
		scratch	0.6606	94.80	0.1808	43.24	0.3771	72.67	0.2926	46.59	——	——
	NSCaching	pretrain	**0.7867**	**94.92**	**0.2048**	47.38	**0.6475**	**81.54**	**0.3004**	47.36	0.3065	**51.36**
		scratch	0.7818	94.63	0.2002	**47.83**	0.6391	80.95	0.2993	**47.64**	**0.3074**	50.65
	IGAN*	pretrain	——	91.3	——	——	——	74.0	——	——	——	——
		scratch	——	92.7	——	——	——	73.1	——	——	——	——
TransH	pretrained		0.4527	92.71	0.1755	43.30	0.4316	73.98	0.2222	38.80	0.1444	29.70
	Bernoulli		0.5206	94.52	0.1862	45.09	0.4518	76.55	0.2329	40.10	0.1783	35.35
	KBGAN	pretrain	0.6168	94.84	0.1923	45.31	0.4262	75.91	0.2807	46.39	——	——
		scratch	0.6018	94.60	0.1869	44.81	0.3364	72.53	0.2779	46.19	——	——
	NSCaching	pretrain	**0.8063**	**95.32**	0.2038	**48.04**	**0.6520**	**81.96**	0.2812	46.48	0.2988	50.82
		scratch	0.8038	95.29	**0.2041**	**48.04**	0.6391	81.05	**0.2832**	**46.59**	**0.3013**	**51.07**
	IGAN*	pretrain	——	94.0	——	——	——	77.0	——	——	——	——
		scratch	——	86.9	——	——	——	73.3	——	——	——	——
TransD	pretrained		0.4426	92.69	0.1782	42.18	0.4320	73.98	0.2244	39.53	0.1571	31.95
	Bernoulli		0.5093	94.61	0.1901	46.41	0.4529	76.55	0.2451	42.89	0.2014	38.61
	KBGAN	pretrain	0.6130	94.92	0.1917	46.49	0.4069	74.27	0.2487	44.33	——	——
		scratch	0.5950	94.68	0.1875	46.41	0.3151	69.77	0.2465	44.40	——	——
	NSCaching	pretrain	**0.8022**	94.99	**0.2013**	48.36	**0.6567**	**82.02**	**0.2883**	**48.33**	**0.3180**	**53.05**
		scratch	0.7994	**95.16**	**0.2013**	**48.39**	0.6415	81.32	0.2863	47.85	0.3146	52.65
	IGAN*	pretrain	——	93.3	——	——	——	77.6	——	——	——	——
		scratch	——	93.0	——	——	——	74.0	——	——	——	——
DistMult	pretrained		0.6340	92.28	0.3765	44.85	0.4985	78.28	0.2247	36.03	0.2805	48.81
	Bernoulli		0.7918	93.38	0.3964	45.25	0.5376	78.69	0.2491	42.03	——	——
	KBGAN	pretrain	0.6955	93.11	0.3849	44.32	0.5568	75.57	0.2670	45.34	0.3262	54.58
		scratch	0.7275	93.08	0.2039	29.52	0.4227	64.35	0.2272	39.91	——	——
	NSCaching	pretrain	0.8297	**93.83**	**0.4148**	**45.80**	0.7447	**84.16**	**0.2882**	**45.79**	**0.4112**	**57.24**
		scratch	**0.8306**	93.74	0.4128	45.45	**0.7448**	83.91	0.2834	45.56	0.4032	56.58
ComplEx	pretrained		0.8046	93.75	0.3934	41.63	0.5191	78.02	0.2201	35.55	0.3008	49.94
	Bernoulli		0.9115	**94.39**	0.4431	**51.77**	0.6253	80.72	0.2596	43.54	0.3568	54.67
	KBGAN	pretrain	0.8976	93.73	0.4287	47.03	0.6254	80.95	0.2818	45.37	——	——
		scratch	0.7233	85.81	0.3180	35.51	0.5002	76.10	0.1910	32.07	——	——
	NSCaching	pretrain	0.9326	94.03	**0.4487**	51.76	0.7994	**86.32**	0.3017	47.75	0.4020	56.50
		scratch	**0.9355**	93.98	0.4463	50.89	**0.7995**	86.28	**0.3021**	**48.05**	**0.4045**	**57.79**
SimplE	pretrained		0.9056	94.34	0.3989	46.09	0.5684	79.62	0.2355	35.71	0.3405	55.46
	Bernoulli		0.9300	94.40	0.4256	46.90	0.6704	81.47	0.2388	36.80	0.3614	56.90
	KBGAN	pretrain	0.9335	94.69	0.4331	47.64	0.7692	85.76	0.2454	40.32	——	——
		scratch	0.9258	94.76	0.4126	46.43	0.4737	66.44	0.2278	35.94	——	——
	NSCaching	pretrain	0.9412	94.71	0.4352	48.01	0.8033	86.90	0.2711	43.88	0.4211	59.77
		scratch	0.9415	94.76	0.4361	47.67	0.8026	86.86	0.2718	43.88	0.4193	56.47
		auto	**0.9446**	**94.98**	**0.4388**	**48.36**	**0.8148**	**88.47**	**0.2966**	**46.22**	**0.4532**	**61.84**

the best hyper-parameters chosen by the MRR value on valid set. For cache related hyper-parameters, we choose $\alpha_1 = \alpha_2 = 0, \alpha_3 = 1$ and $N_1 = N_2 = 50$ for *NSCaching*.

5.3.2 Results on Translational Distance Models

The performance on *link prediction task* is compared in Table 7. First, we can see that, for the translational distance models (TransE, TransH, TransD), *KBGAN*, *NSCaching* and *IGAN* (both *pretrain* and *scratch*) gain significant improvements upon the baseline scheme *Bernoulli*, especially for the performance gaining on the MRR metric, which is mainly influenced by the top rankings. This verifies the advantages of using large-gradient negative triplets during negative sampling and these methods can effectively pick up these negative triplets.

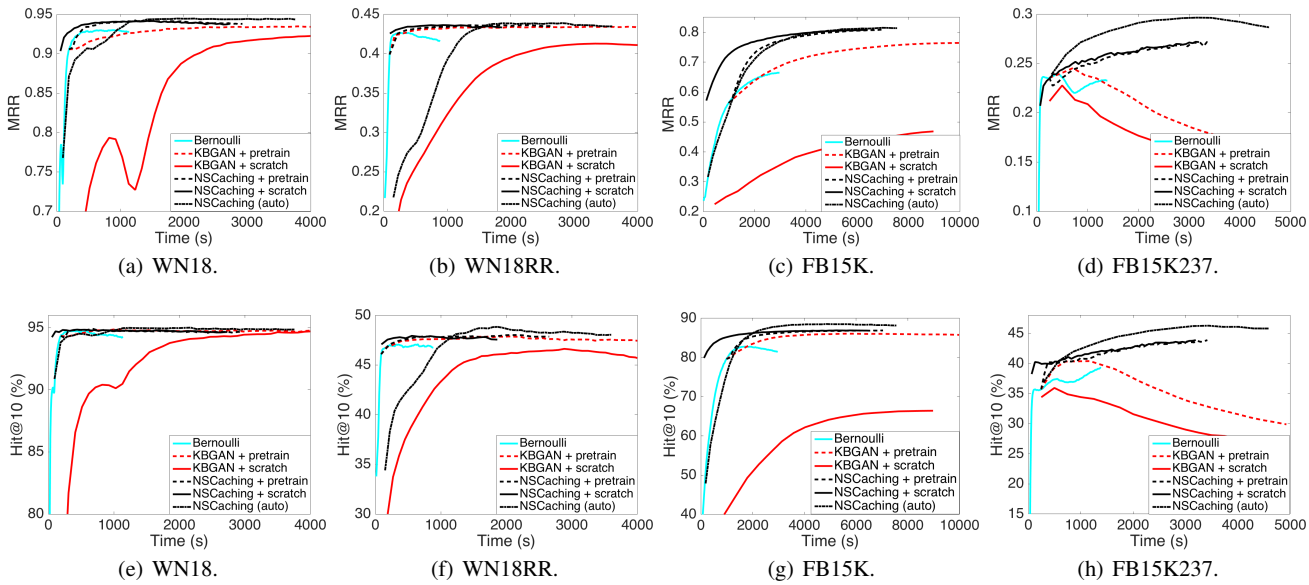
Then, *IGAN* and *KBGAN* with pretraining can perform better, indicated by MRR and Hit@10, than from scratch.



(a) WN18. (b) WN18RR. (c) FB15K. (d) FB15K237.

(e) WN18. (f) WN18RR. (g) FB15K. (h) FB15K237.

Fig. 4: Testing performance v.s. clock time (in seconds) based on TransD (best viewed in color).



(a) WN18. (b) WN18RR. (c) FB15K. (d) FB15K237.

(e) WN18. (f) WN18RR. (g) FB15K. (h) FB15K237.

Fig. 5: Testing performance v.s. clock time (in seconds) based on SimplE (best viewed in color).

This shows that pretraining is helpful for the GAN-based methods. In comparison, *NSCaching* trained from either state (*pretrain* or *scratch*) can outperform *IGAN* and *KBGAN* on all the scoring functions.

The learning curve of the testing performance for various algorithms is shown in Figure 4. We use TransD here as the testbed. As can be seen, all algorithms will converge to some points with stable testing performance, which empirically verifies the convergence of Adam optimizer [28]. Then, pretrain is a must for *KBGAN* to achieve good performance. When the generator is trained from scratch, the whole model will suffer from instability, especially at the initial training stages. As a result, it prevents the GAN-based models converging to some good local points. *NSCaching* can obtain good performance either from *scratch* or with *pretrain*. Note that, even through the updating and sampling scheme introduces extra training cost, we can achieve better performance with fewer iterations. As a result, in all cases, *NSCaching* has better anytime performance than both *Bernoulli* and *KBGAN*.

5.3.3 Results on Semantic Matching Models

The performance on semantic matching models is shown in the bottom rows of Table 7. Same as that on translational distance models, *NSCaching* outperforms baseline scheme *Bernoulli* significantly as indicated by the bold and underline numbers. However, *KBGAN* does not show a consistent performance. In most settings, *KBGAN from scratch* performs even worse than *Bernoulli*. This observation further verifies the fact that GAN-based methods usually suffer from instability and degeneracy. They need careful balance between the generator and the discriminator, i.e., the target KG embedding model. However, *NSCaching* works consistently and performs the best both with pretrain or from scratch.

The learning curve of the testing performance for various algorithms is shown in Figure 5. SimplE is used in this part. As can be seen, both *Bernoulli* and *NSCaching* will converge to some stable points. In the contrast, *KBGAN* will turn down and overfit on FB15K237 data set. However, *NSCaching*, either with pretrain or from scratch, leads the performance and is well adopted on the semantic matching models without further tuning. Besides, as for *NSCaching (auto)*, we find that even though the sampling cost is higher, the performance improvement is obvious and consistent on all these data sets.

5.3.4 Results on Triplets Classification

We do triplets classification in the same way as [56]. This task is to confirm whether a given triplet (h,r,t) is correct or not, i.e., do binary classification on the triplet. Compared with link prediction, triplets classification is more convenient in answering yes-or-no questions. The decision rule of classification is learned as follows: for each (h,r,t), if its score is no less than the relation-specific threshold σ_r, then we predict it to be positive. Otherwise, negative. The threshold σ_r is determined by maximizing the classification accuracy on the valid set. We test this task on WN18RR and FB15K237 data sets based on TransD and SimplE. As shown in Table 8, *NSCaching* still outperforms various baselines. This task further justifies that *NSCaching* can help learn a better embedding of the KG.

Table 8: Comparison of various algorithms on triplet classification task. Bold number indicates the best performance.

model	Dataset		WN18RR	FB15K237
TransD	Bernoulli		86.81	78.24
	KBGAN	pretrained	85.93	79.03
		scratch	86.01	79.05
	NSCaching	pretrained	**87.84**	<u>80.63</u>
		scratch	<u>87.64</u>	**80.69**
SimplE	Bernoulli		84.48	77.64
	KBGAN	pretrained	79.87	74.11
		scratch	71.73	72.61
	NSCaching	pretrained	**84.96**	<u>79.88</u>
		scratch	<u>84.83</u>	**80.21**

5.4 Balancing E&E

In this part, we analyze the designing concerns on the hyper-parameters regarding "exploration and exploitation" in Table 4. SimplE and WN18RR are used as the scoring function and data set respectively.

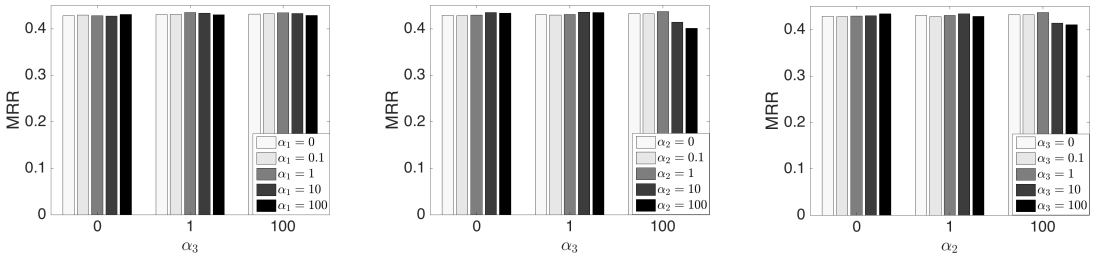
5.4.1 α_1: Sampling positive triplet

Given a set of training triplets and the cache, how to sample the positive triplet is the first question we care about. In Algorithm 3, the most related hyper-parameters are α_1, which controls the distribution of positive samples in cache $\mathscr{C}$, and α_3, which controls the content in the indexed cache $\mathscr{C}_i$. The testing performance with different values of α_1 is compared in Figure 6(a). Since the content in cache is influenced by α_3, we use 0 (low), 1 (middle), and 100 (high) as values of α_3 for the testing. As can be seen, when α_3 is small, different choices of α_1 perform relatively bad and does not have regular influence on embedding performance. As α_3 becomes larger, we see that a larger value of α_1 performs better, which verifies the better convergence property in Theorem 1. However, it will decrease with too large α_1. Take the distribution in Figure 3(c) as an example, some positive triplets will not be selected when α_1 is too large, leading to a problematic training process.

5.4.2 α_2 and α_3: Sampling and updating cache

Once a positive triplet (h_i,r_i,t_i) is sampled, we can get its corresponding cache $\mathscr{C}_i$ which stores the negative triplets. The main parameters influencing the choice of negative triples are α_2 and α_3, namely the temperature for sampling from cache and updating the cache. To show how α_2 and α_3 balance E&E, we fix α_3 in certain ranges (low: 0, middle: 1 and high: 100) and change α_2 in Figure 6(b), and then do it alternatively in Figure 6(c).

From both Figure 6(b) and 6(c), we can see that balancing α_2 and α_3 are of vital importance. When α_2 or α_3 has small value, increasing the other one will improve the performance since exploitation is limited at this stage. However, when α_2 or α_3 becomes larger, the other one should choose an appropriate value in order to avoid exploiting too much. The performance goes up at initial stage and turns down as the break up of balance. The reason is that too large values of α_2 and α_3 will limit exploration such that a few negative triplets will be selected too many times. Besides,



(a) Impact of α_1 (given α_3). (b) Impact of α_2 (given α_3). (c) Impact of α_3 (given α_2).

Fig. 6: Balancing on exploration and exploitation with different values of α_1, α_2 and α_3 with SimplE on WN18RR.

Table 9: Searching range of hyper-parameters and searched best value for different data sets.

hyper-parameters	ranges	best searched				
		WN18	WN18RR	FB15K	FB15K237	YAGO3-10
α_1	[0, 1]	0.0668	0.013	0.0069	0	6.7e-4
α_2	[0, 100]	3.463	0	0	3.122	16.62
α_3	[0, 100]	24.25	2.759	1.848	2.579	0.2228
N_1	$\{10, 30, \ldots, 90\}$	70	90	30	70	70
N_2	$\{10, 30, \ldots, 90\}$	70	50	10	70	70

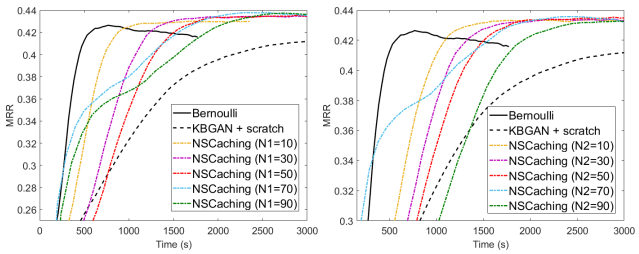
false negative triplets will be more frequently sampled. Fortunately, E&E is well balanced under a wide range of α's values where *NSCaching* performs well without much effort in tuning α's.

Besides, when the cache is updated without referring to the gradient norms, i.e. $\alpha_3 = 0$ and $\alpha_2 > 0$, NSCaching works as an alternative version of KBGAN and outperforms Bernoulli baseline approach. It works stabler than KBGAN which suffers from the instable training of the generator. However, the performance is still not the best since balance of E&E is not in the best state when cache is updated without considering the negative triplets' qualities.

5.4.3 N_1 and N_2: Cache size

Basically, N_1 is the size of cache $\mathscr{C}_i$. Then, N_2 is the size of randomly sampled subset $\mathscr{R}_i$ of negative triplets from $\mathscr{T}_i$, which will later be used to update the cache. In this part, we show their impact on the performance of *NSCaching*. The three temperature values are set as $\alpha_1 = \alpha_2 = 0, \alpha_3 = 1$. Figure 7(a) shows how performance changes by varying the cache size N_1 among $\{10, 30, 50, 70, 90\}$ with fixed $N_2 = 50$. When the cache size is small, average quality of triplets stored in the cache should be larger than those in a cache with larger size. As a result, false negative triplets will be more likely to be sampled, which will influence the embedding quality. With the others values of N_1, *NSCaching* performs quite stable. The convergence speed is similar, as well as the values in converged state. Thus, when we need to set appropriate cache size, the value of N_1 can be searched from smaller values to larger ones until the performance is stable.

Different performance of the random candidate subset size N_2 is shown in Figure 7(b). The entities in cache will be updated more frequently when N_2 gets larger, which lead to better exploration. But the trade-off is that larger value of N_2 is more expensive. As shown by the colored lines in Figure 7(b), NSCaching performs consistently when N_2 is larger than 10. However, if the random subset is small, the content in cache will be harder to be updated, thus leading to poor performance as the yellow dashed line ($N_2 = 10$).



(a) Diff. N_1. (b) Diff. N_2.

Fig. 7: Comparison of different N_1 when random subset size N_2 is fixed to 50, and different N_2 when cache size N_1 is fixed to 50. Evaluated by SimplE model on WN18RR (best viewed in color).

By combining together the influence of cache size N_1 and the random subset size N_2 in Figure 7, we find that (i)

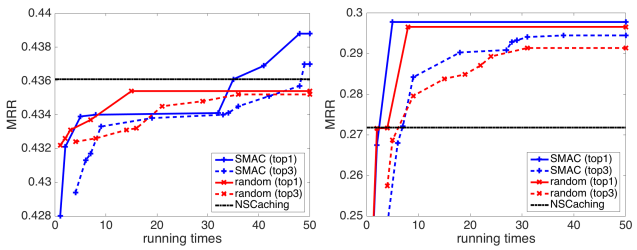
NSCaching is not sensitive to the two sizes; (ii) both sizes can not be too small; (iii) $N_1 = N_2$ is a good balance.

5.4.4 Automatically Balancing E&E

In previous part, we have shown the importance of balancing between E&E. Here, we use AutoML techniques [61] to automatically balance E&E and further improve the performance on five benchmarks.

For each dataset, we use the same value of learning rate, batch size, embedding dimension and regularizer penalty as the Bernoulli baseline to make a fair comparison. The other hyper-parameters related to the negative sampling algorithm are searched within the ranges given in Table 9 by SMAC [23], a well-known AutoML algorithm for hyper-parameter optimization. The initial hyper-parameter setting is $\alpha_1 = \alpha_2 = \alpha_3 = 0$ and $N_1 = N_2 = 50$, namely a setting similar to Bernoulli sampling. Besides, we take random search [5] as a baseline rather than the grid search, as random search is generally more effective [5].

Figure 8 shows the MRR from the best (denoted by "top1") and top three (denoted as "top3") models obtained during the search procedure. Once the searching starts, the top performance will soon be boosted for both SMAC and random by exploring hyper-parameters with better balance of E&E. Both of the two search algorithms find hyper-parameters that outperform the original *NSCaching+scratch*. However, with the help of Bayesian optimization, SMAC is more efficient and effective than random. This verifies the importance of introducing AutoML into the framework of *NSCaching*. After searching and running for 50 hyper-parameter settings, we show the performance of the best hyper-parameter on testing data in Table 7. The hyper-parameter settings with best performance are given in Table 9.



(a) WN18RR. (b) FB15K237.

Fig. 8: Performance comparison of SMAC and random search (top1 and top3 average). x-axis is number of running times for different hyper-parameter. y-axis is the mean MRR on validation set. The MRR peformance of NSCaching is given as the black-dashed line for a reference (best viewed in color).

5.5 Ablation study

5.5.1 Lazy update

In Section 3, we have introduced the lazy update parameter n to reduce the computation cost of *NSCaching*. In this part, we analyze the influence of n on the learning curve in Figure 9. We run each model for 1,000 epochs, and report the best performance and running time. The relative time and MRR are divided by the corresponding values of $n = 0$ respectively. When n increases, the computation cost is reduced since less update operations are conducted. However, the performance is gradually decreasing since the cache will be less frequently updated, reducing the exploration. Fortunately, the decrease of performance is not obvious (less than 3%) when $n \leq 10$. So the value of n can be regarded as a trade-off for time and performance, adapting to different application requirements.

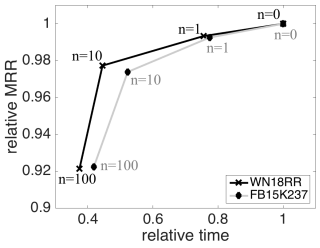
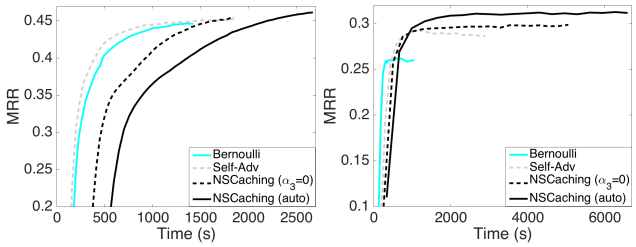


Fig. 9: Influence of different lazy update value n of *NSCaching* (SimplE is used as the testbed).

5.5.2 Comparing with Self-adv

In this part, we compare *NSCaching* with the concurrent work *Self-Adv* by using the RotatE scoring function [48]: $f(h,r,t) = -\|\mathbf{h} \circ \mathbf{r} - \mathbf{t}\|_1$, where $\mathbf{h}, \mathbf{r}, \mathbf{t}$ are complex embeddings and $\circ$ is the Hadmard product in the complex space.

As discussed in Section 5.3.1, *Self-Adv* relies on sampling a small subset $\mathscr{N}$ from $\mathscr{E}_i$ and then sampling from $\mathscr{N}$. A similar setting in *NSCaching* is when $\alpha_3 = 0$, where the cache is updated without depending on the scores so that the cache can have the same distribution with $\mathscr{R}_m$. As shown in Figure 10, both *Self-Adv* and *NSCaching* outperform *Bernoulli* sampling. This again demonstrates the universality of using large-gradient negative samples to improve the performance. Then, *Self-Adv* and *NSCaching* ($\alpha_3 = 0$) have the similar best performance, but *NSCaching* ($\alpha_3 = 0$) is a bit slower due to the updating procedure. Besides, *NSCaching (auto)* achieves the best performance by using the cache $\mathscr{C}_i$, which has more large-gradient samples than $\mathscr{N}$ in *Self-Adv*.



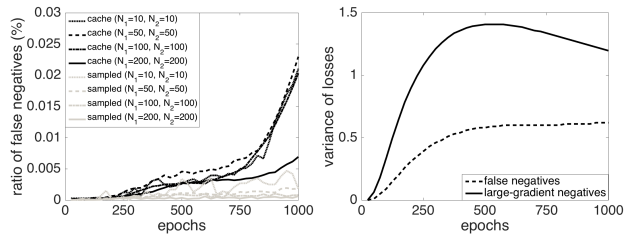
(a) WN18RR. (b) FB15K237.

Fig. 10: Learning curve of different negative sampling method on RotatE.

5.5.3 *Influence of false negatives*

In Section 3.5, we show that variance can be used as a standard to detect the false negatives. In this part, we empirically analyze the potential influence of introducing variance into the sampling method with SimplE model and WN18RR data set. We use the valid and test set here as the set of false negative samples.

First, we analyze the possibility of sampling the false negative triplets from the cache. In Figure 11(a), we show the ratio of false negative triplets in the cache (denoted as *cache*) and the percentage of false negatives in the sampled negative triplets (denoted as *sampled*) during training. The cache does contain a few false negative triplets, but less than 0.03% contents are false negative. The ratio of false negatives among the sampled triplets is even smaller. Besides, we set $N_1 = N_2$ and use different values to show how the cache size influences the ratio of false negative triplets. When the cache size N_1 increases, the possibility of sampling the false negative triplets decreases. Since in each iteration, only one negative triplet is generated and the cache keeps updating, we can avoid frequently picking up the false negative samples.



(a) Ratio of FN in the cache. (b) Average variance.

Fig. 11: Understanding false negative (FN) triplets with SimplE on WN18RR.

Second, we show that variance can be used as a standard to detect false negatives. In Figure 11(b), we plot the average variance of 5,000 false negative triplets and 5,000 large-gradient negative triplets. The false negative triplets are sampled from the valid and test set. For the large-gradient negative triplets, we run *NSCaching* for 1,000 epochs first and then sample 5,000 triplets, which are not false negative, from the final cache. The scores of triplets are estimated per epoch, and the variances are computed based on the recorded scores. As shown in Figure 11(b), the variance of large-gradient negative samples is larger than that of false negative ones. This indicates that the false negative triplets may be detected by tracing the variance during training.

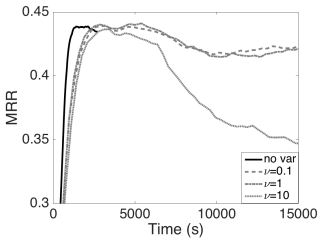


Fig. 12: Adding variance with different γ with SimplE on WN18RR.

Finally, we show the influence of adding variance into the sampling method. We record the *score* measured by SimplE and *std* of the scores for the negative samples. To save the memory cost, we use the Welford's online algorithm [57] to estimate the *std* of negative triplets. Then, we use $score + \nu std$, where $\nu > 0$ is a weighting parameter, as the metric here to sample the large-gradient negative triplets. As shown in Figure 12, when ν is given an appropriate value like $\nu \leq 1$, the best performance is almost the same, but the computation cost increases a lot. The training will be instable after adding the *std* term, especially when the value of ν is large. Therefore, it is not necessary to consider the variance to reduce the problem of false negative triplets in this work. Instead, we control α's to avoid frequently sampling the false negative triplets.

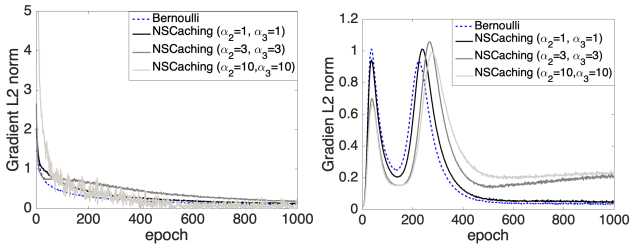
5.6 Theoretical Explanation

Here, we study the theoretical perspective of the proposed approach, which gives more insights and helps us understand NSCaching better.

5.6.1 *Illustration of Vanishing Gradients*

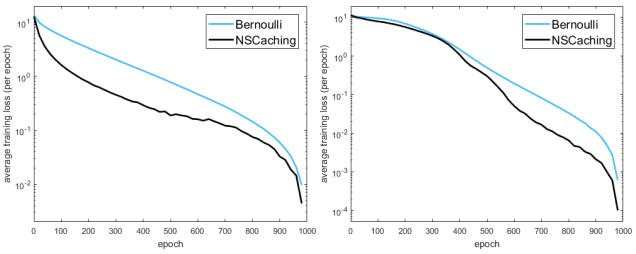
To further clarity the vanishing gradient problem, we plot the average ℓ_2-norm of gradients v.s. number of epochs in Figure 13. Adam [28], a stochastic gradient descent algorithm, is used as the optimizer. First, we can see that while the norms of gradients for both *NSCaching* and *Bernoulli*

become smaller, they will not decrease to zero since the sampling process of the mini-batch will introduce noise into gradients. However, the norm from *NSCaching* is larger than that from *Bernoulli* due to the usage of cache-based negative sampling scheme. Thus, we can see *NSCaching* can successfully avoid the problem of vanishing gradient. We also show the changes with different α's. When the value of α's increases, the gradient norm will become larger, especially after the warm-up procedure, i.e., after 400 epochs. For the TransD model, when $\alpha_2 = \alpha_3 = 10$, the training will become unstable. This also verifies that we should control the value of α through the AutoML technique.



(a) TransD. (b) SimplE.

Fig. 13: Average ℓ_2-norm of gradients within a mini-batch v.s. number of epochs for Bernoulli and NSCaching on WN18RR.



(a) WN18RR. (b) FB15K237.

Fig. 14: Average training loss in one epoch v.s. number of epochs for Bernoulli and NSCaching.

5.6.2 Convex Case: Faster convergence

To demonstrate the faster convergence illustrated in Theorem 1, we use TransE as the scoring function and use classification loss in (2) for KG embedding. As mentioned in Section 3.3.1, this does not fall into any existing KG embedding models, but it satisfies assumptions in Theorem 1. Thus, it is a good synthetic model to be studied. We use the same hyper-parameters identified for *NSCaching* in Section 5.3, and compare it with *Bernoulli* scheme. FB15K237 and WN18RR are considered here. The comparison of average training loss per epoch v.s. epoch between *NSCaching* and *Bernoulli* is shown in Figure 14. As we can see, *NSCaching* indeed leads to faster convergence than *Bernoulli*, which is explained by Theorem 1.

5.6.3 Nonconvex Case: Self-paced learning

Finally, we visualize the changes of content in the cache, which verifies the effects of self-paced learning introduced in Section 3.3.2. Following [54], we also use FB13 here, which has 75,043 entities, 13 relations and 316,232 training triplets, since triplets in this dataset are more interpretable than the five evaluated datasets. We pick up $(manorama, profession, actor)$ as the positive triplet, and the changes in its tail-cache $\mathscr{T}_{r,t}$ are shown in Table 10. As can be seen, negative tails are firstly meaningless, e.g., *ostrava* and *ben_lilly*, then they gradually changes to human jobs, e.g., *artist* and *sex_worker*.

Table 10: Examples of negative entities in cache on FB13. Each line displays 5 random sampled negative entities from tail-cache of a positive fact *(manorama, profession, actor)* in different epochs.

epoch	entities in cache
0	*allen_clarke, jose_gola, ostrava, ben_lilly, hans_zinsser*
20	*accountant, frank_pais, laura_marx, como, domitia_lepida*
100	*artist, , aviator, hans_zinsse, john_h_cough*
200	*physician, artist, raich_carter, coach, mark_shivas*
500	*artist, physician, cavan, sex_worker, attorney_at_law*

5.7 Graph Embedding

In this part, we perform experiments with the skip-gram model on the task of graph embedding.

5.7.1 Experiment Setup

Two famous graph data sets are used here: Cora and Citeseer, both of which are academic citation networks introduced in [35]. Cora contains 2,708 papers with 5,429 connections in machine learning area. These papers belong to 7 different classes. Citeseer is formed by 3,312 papers in 6 areas. The total number of connections is 4,660. In this part we compare NSCaching (in Algorithm 6) with the following methods.

– *LINE* [49]: In this method, the embeddings are trained to preserve the first order, i.e., the direct connections, and second order, i.e., the neighborhood similarities, in graphs.

Table 11: Node classification results.

datasets	methods	30%		50%		70%	
		Micro-F1	Macro-F1	Micro-F1	Macro-F1	Micro-F1	Macro-F1
Cora	LINE [49]	0.8093±0.0027	0.8037±0.0021	0.8120±0.0025	0.8024±0.0029	0.8096±0.0043	0.8037±0.0057
	Node2vec	0.8130±0.0031	0.8061±0.0031	0.8142±0.0024	0.8064±0035	0.8376±0.0038	0.8323±0.0025
	SeedNE [18]	0.8142±0.0033	0.8089±0.0023	0.8195±0.0024	0.8081±0.0030	0.8364±0.0047	0.8315±0.0024
	NSCaching	**0.8185**±0.0031	**0.8108±0.0028**	**0.8273±0.0039**	**0.8184±0.0043**	**0.8413±0.0019**	**0.8334±0.0027**
Citeseer	LINE [49]	0.5465±0.0027	0.4984±0.0044	0.5488±0.0035	0.5074±0.0032	0.5754±0.0027	0.5175±0.0039
	Node2vec	0.5755±0.0019	0.5253±0.0017	0.5793±0.0013	0.5357±0.0016	0.5936±0.0044	0.5410±0.0013
	SeedNE [18]	0.5762±0.0028	0.5279±0.0030	0.5918±0.0025	0.5369±0.0031	0.6059±0.0015	0.5552±0.0038
	NSCaching	**0.5864±0.0018**	**0.5297±0.0047**	**0.5982±0.0020**	**0.5531±0.0029**	**0.6120±0.0015**	**0.5712±0.0018**

– *Node2vec* [21]: Different from LINE, *Node2vec* uses biased random walk to preserve the topology information on graphs. The generated walks are regarded as sentences in the language model. In this way, skip-gram model is used to learn the embeddings. The distribution used to sample negative nodes is proportional to $3/4$ of the nodes' frequency.
– *SeedNE* [18]: To improve the negative sampling in skip-gram model, SeedNE selects the negative nodes whose similarities with the positive node are higher than an increasing threshold. Either the self-embedding or a learned generator can be used to indicate the similarities. To guarantee stability, we use self-embedding in this part. This sampling method also has some connection with the self-paced learning, but the problem of E&E is not well addressed.

In order to make a fair comparison with the baselines, we set the embedding dimension to be 100 for all the datasets as in [18]. Similar as [21], 10 random walks are generated for each node with $p = 0.25, q = 0.25$, both of which are hyper-parameters controlling the biased random walk. The window size $|\mathcal{W}_u|$ is set to be 10 for each node in the walks. The walks are fixed when comparing different models. Finally, we use Adam [28] as the optimizer. The learning rate is set as the default value 10^{-3} and we use $\lambda = 10^{-7}$ as the weight decay value.

To measure the quality of the learned embeddings of different methods, we use node classification task as the testbed. After the embeddings are learned, a logistic regression model is learned as the classifier. Specifically, we use the cache-based skip-gram model to train the node embeddings. Then the embeddings and their corresponding labels are fed into the classifier. Following [18, 21, 49], we use F1-score, which considers both the precision and recall, to measure the test accuracy. It is a widely used metric in binary classification and is computed by $F_1 = 2 \cdot \text{precision} \cdot \text{recall} / \text{precision} + \text{recall}$. Considering that the problem here is a multi-class task, we use two variants

– Micro-F1: the precision and recall are computed by ignoring the type, and then computing the F1-score.
– Macro-F1: the precision and recall are computed separately on each class, and return the average F1-score.

The larger values of the F1-scores indicate better performance.

5.7.2 Empirical Results

We randomly select {30%, 50%, 70%} labeled nodes to train the classifier five times and evaluate the performance on the remaining nodes respectively. We report the average and the standard deviation on the node classification task in Table 11. Comparing with the frequency-based negative sampling method, i.e., *Node2vec* [21], the *NSCaching* based skip-gram method and *SeedNE* gain significant improvement since they are able to capture the dynamic distribution of negative samples. By comparing *Node2vec* (skip-gram) with *LINE*, we can see that the random walk based model is better than *LINE*, which only uses direct connection and neighbors to measure the similarity. Besides, *NSCaching* outperforms another self-paced negative sampling method *SeedNE* with better control of E&E.

6 Conclusion

In this paper, we propose NSCaching, a novel negative sampling method for knowledge graph embedding learning. The negative samples are sampled from a cache that can dynamically hold large-gradient negative samples. We theoretically understand the convergence and effectiveness from both convex and non-convex case. In order to balance exploration and exploitation during the sampling procedure, we use AutoML to automatically balance E&E for NSCaching in regard of sampling and updating the cache. In addition, we extend NSCaching to skip-gram model, which is widely used in word embedding and graph embedding. Experimentally, we test NSCaching on benchmark datasets and various scoring functions. Empirical results show that the method can generalize well under various settings and achieves state-of-the-arts performance. In future work, we will use AutoML technique to search for scoring functions

[68] and explore recurrent neural architectures to learn from relational paths in knowledge graphs [67].

Acknowledgements

A Appendix: Proof of Theorem 1

Proof. Following [70], we consider a more general optimization formulation as $\min_{\mathbf{w}} \bar{F}(\mathbf{w}) \equiv \frac{1}{n}\sum_{i=1}^{n} \phi_i(\mathbf{w}) + \lambda r(\mathbf{w})$, which covers (11) as a special case with $\lambda = 0$. Then, the stochastic training gives $\mathbf{w}^{t+1}$ as

$$\mathbf{w}^{t+1} = \arg\min_{\mathbf{w}} \left[(np^t_{i_t})^{-1} \mathbf{w}^\top \nabla \phi_{i_t}(\mathbf{w}^t) + \lambda r(\mathbf{w}) + \frac{1}{\eta_t} \mathscr{B}_\psi(\mathbf{w}, \mathbf{w}^t) \right], \quad (18)$$

where $\mathscr{B}_\psi$ is a Bregman distance function measuring the difference between $\mathbf{w}$ and $\mathbf{w}^t$. Based on (18), we state Theorem 3 in [70] as

Theorem 2 ([70]) *Let $\mathbf{w}^t$ be generated from* (18)*. Assume $\mathscr{B}_\psi$ is σ-strongly convex w.r.t. a norm $\|\cdot\|$, $\bar{F}$ and r are convex, if $\eta_t = \eta$, the following inequality holds for any $T \geq 1$*

$$\frac{1}{T}\sum_{t=1}^{T} \mathbb{E}[\bar{F}(\mathbf{w}^t)] - \mathbb{E}[\bar{F}(\mathbf{w}^*)]$$
$$\leq \frac{1}{T}\left[\frac{1}{\eta} \mathscr{B}_\psi(\mathbf{w}^*, \mathbf{w}^1) + \frac{\eta}{2\sigma} \sum_{t=1}^{T} \mathbb{E}\| {}^{\nabla \phi_{i_t}(\mathbf{w}^t)}/_{n\mathbf{p}^t_{i_t}} \|^2 \right]$$

where the expectation is take with distribution $\mathbf{p}^t$.

In our Algorithm 5, we do not have r and thus $\lambda = 0$ in (11). Besides, $\mathscr{B}_\psi(\mathbf{w}, \mathbf{w}^t) = \frac{1}{2}\|\mathbf{w} - \mathbf{w}^t\|^2$ in our case, thus $\sigma = 1$. Above all, we have Theorem 1 which is derived from Theorem 2. □

References

1. M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein GAN. In *ICLR*, 2017.
2. S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. DBpedia: A nucleus for a web of open data. In *The Semantic Web*, pages 722–735. Springer, 2007.
3. Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum learning. In *ICML*, pages 41–48, 2009.
4. J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *NIPS*, pages 2546–2554, 2011.
5. J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *JMLR*, 13(Feb):281–305, 2012.
6. K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *ACM SIGMOD*, pages 1247–1250, 2008.
7. A. Bordes, S. Chopra, and J. Weston. Question answering with subgraph embeddings. In *Conference on EMNLP*, pages 615–620, 2014.
8. A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko. Translating embeddings for modeling multi-relational data. In *NIPS*, pages 2787–2795, 2013.
9. A. Bose, H. Ling, and Y. Cao. Adversarial contrastive estimation. In *ACL (Volume 1: Long Papers)*, pages 1021–1032, 2018.
10. H. Cai, V. Zheng, and K. Chang. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE TKDE*, 30(9):1616–1637, 2018.
11. L. Cai and W. Wang. KBGAN: Adversarial learning for knowledge graph embeddings. In *Conference of NAACL*, volume 1, pages 1470–1480, 2018.
12. L. Chen, F. Yuan, J. Jose, and W. Zhang. Improving negative sampling for word representation using self-embedded features. In *WSDM*, pages 99–107, 2018.
13. T. Dettmers, P. Minervini, P. Stenetorp, and S. Riedel. Convolutional 2d knowledge graph embeddings. In *AAAI*, 2018.
14. J. Ding, Y. Quan, X. He, Y. Li, and D. Jin. Reinforced negative sampling for recommendation with exposure data. In *IJCAI*, pages 2230–2236. AAAI Press, 2019.
15. L. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *ACM SIGKDD*, pages 601–610, 2014.
16. D. Dori. Visweb-the visual semantic web: unifying human and machine knowledge representations with object-process methodology. *The VLDB Journal*, 13(2):120–147, 2004.
17. W. Fedus, I. Goodfellow, and A. Dai. Maskgan: better text generation via filling in the _. In *ICLR*, 2018.
18. H. Gao and H. Huang. Self-paced network embedding. In *ACM SIGKDD*, pages 1406–1415, 2018.
19. X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.
20. I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *NIPS*, pages 2672–2680, 2014.
21. A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *ACM SIGKDD*, pages 855–864, 2016.
22. I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville. Improved training of wasserstein gans. In *NIPS*, pages 5767–5777, 2017.
23. F. Hutter, H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *ICLIO*, pages 507–523, 2011.
24. F. Hutter, L. Kotthoff, and J. Vanschoren, editors. *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 2018.
25. G. Ji, S. He, L. Xu, K. Liu, and J. Zhao. Knowledge graph embedding via dynamic mapping matrix. In *ACL*, volume 1, pages 687–696, 2015.
26. R. Kadlec, O. Bajgar, and J. Kleindienst. Knowledge base completion: Baselines strike back. In *the 2nd Workshop on Representation Learning for NLP*, pages 69–74, 2017.
27. S. Kazemi and D. Poole. SimplE embedding for link prediction in knowledge graphs. In *NeurIPS*, pages 4289–4300, 2018.
28. D. Kingma and J. Ba. Adam: A method for stochastic optimization. Technical report, arXiv:1412.6980, 2014.
29. S. Kok and P. Domingos. Statistical predicate invention. In *ICML*, pages 433–440, 2007.
30. D. Koller, N. Friedman, S. Džeroski, C. Sutton, A. McCallum, A. Pfeffer, P. Abbeel, M. Wong, D. Heckerman, C. Meek, et al. *Introduction to statistical relational learning*. The MIT Press, 2007.
31. M. Kumar, B. Packer, and D. Koller. Self-paced learning for latent variable models. In *NIPS*, pages 1189–1197, 2010.
32. N. Lao, T. Mitchell, and W. Cohen. Random walk inference and learning in a large scale knowledge base. In *Conference on EMNLP*, pages 529–539. ACL, 2011.
33. J. Li, C. Tao, Y. Feng, D. Zhao, R. Yan, et al. Sampling matters! an empirical study of negative sampling strategies for learning of matching models in retrieval-based dialogue systems. In *Proceedings of the 2019 EMNLP-IJCNLP*, pages 1291–1296, 2019.
34. J. March. Exploration and exploitation in organizational learning. *Organization science*, 2(1):71–87, 1991.
35. A. McCallum, K. Nigam, J. Rennie, and K. Seymore. Automating the construction of internet portals with machine learning. *Information Retrieval*, 3(2):127–163, 2000.
36. T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. In *ICLR*, 2013.

37. T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111–3119, 2013.
38. T. Mikolov and G. Yih, W.and Zweig. Linguistic regularities in continuous space word representations. In *NAACL*, pages 746–751, 2013.
39. D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas. Exemplar queries: a new way of searching. *The VLDB Journal*, 25(6):741–765, 2016.
40. D. Needell, R. Ward, and N. Srebro. Stochastic gradient descent, weighted sampling, and the randomized kaczmarz algorithm. In *NIPS*, pages 1017–1025, 2014.
41. M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33, 2015.
42. N. Noy, Y. Gao, A. Jain, A. Narayanan, A. Patterson, and J. Taylor. Industry-scale knowledge graphs: lessons and challenges. *Queue*, 17(2):48–75, 2019.
43. A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pages 8024–8035, 2019.
44. B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. In *ACM SIGKDD*, pages 701–710, 2014.
45. A. Rawat, J. Chen, F. Yu, A. Suresh, and S. Kumar. Sampled softmax with random fourier features. In *NeurIPS*, pages 13857–13867, 2019.
46. S. Rendle, C. Freudenthaler, Z. Gantner, and L. Schmidt-Thieme. BPR: Bayesian personalized ranking from implicit feedback. In *Conference on UAI*, pages 452–461, 2009.
47. F. Suchanek, G. Kasneci, and G. Weikum. YAGO: a core of semantic knowledge. In *WWW*, pages 697–706, 2007.
48. Z. Sun, Z. Deng, J. Nie, and J. Tang. Rotate: Knowledge graph embedding by relational rotation in complex space. In *ICLR*, 2018.
49. J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei. Line: Large-scale information network embedding. In *WWW*, pages 1067–1077, 2015.
50. K. Toutanova and D. Chen. Observed versus latent features for knowledge base and text inference. In *Workshop on CVSMC*, pages 57–66, 2015.
51. T. Trouillon, C. Dance, É. Gaussier, J. Welbl, S. Riedel, and G. Bouchard. Knowledge graph completion via complex tensor factorization. *JMLR*, 18(1):4735–4772, 2017.
52. H. Wang, J. Wang, J. Wang, M. Zhao, W. Zhang, F. Zhang, X. Xie, and M. Guo. Graphgan: Graph representation learning with generative adversarial nets. In *AAAI*, 2018.
53. J. Wang, L. Yu, W. Zhang, Y. Gong, Y. Xu, B. Wang, P. Zhang, and D. Zhang. IRGAN: A minimax game for unifying generative and discriminative information retrieval models. In *ACM SIGIR*, pages 515–524, 2017.
54. P. Wang, S. Li, and R. Pan. Incorporating GAN for negative sampling in knowledge representation learning. In *AAAI*, 2018.
55. Q. Wang, Z. Mao, B. Wang, and L. Guo. Knowledge graph embedding: A survey of approaches and applications. *IEEE TKDE*, 29(12):2724–2743, 2017.
56. Z. Wang, J. Zhang, J. Feng, and Z. Chen. Knowledge graph embedding by translating on hyperplanes. In *AAAI*, volume 14, pages 1112–1119, 2014.
57. B. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.
58. R. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.
59. C. Wu, R. Manmatha, A. Smola, and P. Krahenbuhl. Sampling matters in deep embedding learning. In *Proceedings of the ICCV*, pages 2840–2848, 2017.
60. B. Yang, W. Yih, X. He, J. Gao, and L. Deng. Embedding entities and relations for learning and inference in knowledge bases. In *ICLR*, 2017.
61. Q Yao and M. Wang. Taking human out of learning applications: A survey on automated machine learning. Technical report, arXiv preprint, 2018.
62. R. Ying, R. He, K. Chen, P. Eksombatchai, W. Hamilton, and J. Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *ACM SIGKDD*, pages 974–983, 2018.
63. C. Zhang, Y. Li, N. Du, W. Fan, and P. Yu. On the generative discovery of structured medical knowledge. In *SIGKDD*, pages 2720–2728, 2018.
64. F. Zhang, N. Yuan, D. Lian, X. Xie, and W. Ma. Collaborative knowledge base embedding for recommender systems. In *ACM SIGKDD*, pages 353–362, 2016.
65. S. Zhang, L. Yao, A. Sun, and Y. Tay. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys*, 52(1):1–38, 2019.
66. W. Zhang, T. Chen, J. Wang, and T. Yu. Optimizing top-n collaborative filtering via dynamic negative item sampling. In *ACM SIGIR*, pages 785–788, 2013.
67. Y. Zhang, Q. Yao, and L. Chen. Neural recurrent structure search for knowledge graph embedding. Technical report, 2019.
68. Y. Zhang, Q. Yao, W. Dai, and L. Chen. Autosf: Searching scoring functions for knowledge graph embedding. In *ICDE*, pages 433–444. IEEE, 2020.
69. Y. Zhang, Q. Yao, Y. Shao, and L. Chen. NSCaching: Simple and efficient negative sampling for knowledge graph embedding. In *ICDE*, pages 614–625, 2019.
70. P. Zhao and T. Zhang. Stochastic optimization with importance sampling for regularized loss minimization. In *ICML*, pages 1–9, 2015.
71. L. Zou, L. Chen, M. Özsu, and D. Zhao. Answering pattern match queries in large graph databases via graph embedding. *The VLDB Journal*, 21(1):97–120, 2012.