Learning Backward Compatible Embeddings

Weihua Hu
Stanford University
weihuahu@cs.stanford.edu

Rajas Bansal
Stanford University
rajasb@cs.stanford.edu

Kaidi Cao
Stanford University
kaidicao@cs.stanford.edu

Nikhil Rao
Amazon
nikhilsr@amazon.com

Karthik Subbian
Amazon
ksubbian@amazon.com

Jure Leskovec
Stanford University
jure@cs.stanford.edu

ABSTRACT

Embeddings, low-dimensional vector representation of objects, are fundamental in building modern machine learning systems. In industrial settings, there is usually an embedding team that trains an embedding model to solve *intended tasks* (*e.g.*, product recommendation). The produced embeddings are then widely consumed by consumer teams to solve their *unintended tasks* (*e.g.*, fraud detection). However, as the embedding model gets updated and retrained to improve performance on the intended task, the newly-generated embeddings are no longer compatible with the existing consumer models. This means that historical versions of the embeddings can never be retired or all consumer teams have to retrain their models to make them compatible with the latest version of the embeddings, both of which are extremely costly in practice.

Here we study the problem of embedding version updates and their backward compatibility. We formalize the problem where the goal is for the embedding team to keep updating the embedding version, while the consumer teams do not have to retrain their models. We develop a solution based on learning backward compatible embeddings, which allows the embedding model version to be updated frequently, while also allowing the latest version of the embedding to be quickly transformed into any backward compatible historical version of it, so that consumer teams do not have to retrain their models. Our key idea is that whenever a new embedding model is trained, we learn it together with a light-weight backward compatibility transformation that aligns the new embedding to the previous version of it. Our learned backward transformations can then be composed to produce any historical version of embedding. Under our framework, we explore six methods and systematically evaluate them on a real-world recommender system application. We show that the best method, which we call BC-Aligner, maintains backward compatibility with existing unintended tasks even after multiple model version updates. Simultaneously, BC-Aligner achieves the intended task performance similar to the embedding model that is solely optimized for the intended task.[1]

[1]Code is publicly available at https://github.com/snap-stanford/bc-emb

CCS CONCEPTS

• **Information systems → Recommender systems**.

KEYWORDS

embeddings, backward compatibility, graph neural networks, recommender systems

ACM Reference Format:
Weihua Hu, Rajas Bansal, Kaidi Cao, Nikhil Rao, Karthik Subbian, and Jure Leskovec. 2022. Learning Backward Compatible Embeddings. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '22), August 14–18, 2022, Washington, DC, USA.* ACM, Washington, DC, USA, 11 pages. https://doi.org/10.1145/3534678.3539194

1 INTRODUCTION

Embeddings are widely used to build modern machine learning systems. In the context of recommender systems, embeddings are used to represent items and to capture similarity between them. Such embeddings can be then used for many tasks like item recommendations, item relevance prediction, item property prediction, item sales volume prediction as well as fraud detection [18, 22].

The universality of embeddings and the proliferation of state of the art methods to generate these embeddings [7] pose an interesting challenge. Many machine learning practitioners develop embeddings for a specific purpose (*e.g.*, for item recommendation) but then the embeddings get utilized by many other downstream consumer teams for their own purposes and tasks. Oftentimes, the number of such consumer teams is very large and hard to track. At the same time, the original embedding team aims to further evolve their embedding model architecture, training data, and training protocols, with the goal to improve performance on their specific task. In this process, the embedding team generates new and improved versions of the embeddings but these are incompatible with existing downstream consumer models. This means the downstream consumers of these embeddings must retrain their models to use the improved embeddings, or choose to stick with the older, potentially poorer embeddings as inputs to their models. To maintain compatibility with all the existing consumer tasks, the embedding team needs to maintain all historical versions of their embedding model, generate all historical versions of the embeddings and make them available to the consumer teams. This means that the historical embedding models can never be retired and significant human and computational resources are wasted. An alternative approach would be to retire old versions of the embeddings and encourage the consumer teams to retrain their models and migrate to the new version of the embeddings. In practice, this is extremely hard. It can take years before all the consumer teams move to a new version

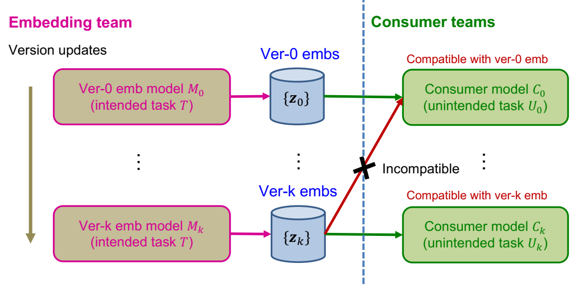
Embedding team
Version updates
Ver-0 emb model M_0 (intended task T)
Ver-0 embs
$\{z_0\}$
Ver-k emb model M_k (intended task T)
Ver-k embs
$\{z_k\}$
Consumer teams
Compatible with ver-0 emb
Consumer model C_0 (unintended task U_0)
Incompatible
Compatible with ver-k emb
Consumer model C_k (unintended task U_k)

Figure 1: Problem formulation. The embedding team trains *embedding model* M_0 to solve their *intended task* T. The consumer teams may then utilize the produced embeddings $\{z_0\}$ to solve some *unintended task* U_0 using consumer model C_0. The issue arises when the embedding team releases new improved versions of the embedding model $M_1, M_2, \ldots$ over time. At version k, the latest-version embedding model M_k produces ver-k embeddings $\{z_k\}$ that are incompatible with consumer model C_0 that is trained on the ver-0 embeddings $\{z_0\}$. Our goal is to quickly transform the latest-version embedding z_k into a backward compatible historical version of it so that existing consumer models can readily use the transformed embedding without being retrained.

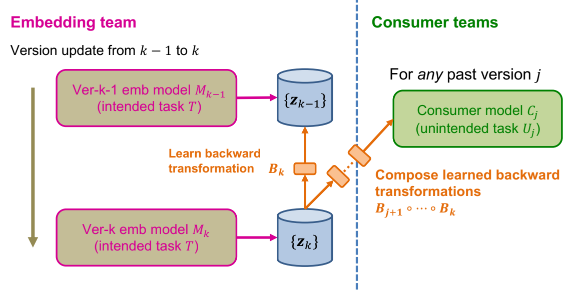
Embedding team
Version update from $k-1$ to k
Ver-k-1 emb model M_{k-1} (intended task T)
$\{z_{k-1}\}$
Learn backward transformation B_k
Ver-k emb model M_k (intended task T)
$\{z_k\}$
Consumer teams
For *any* past version j
Consumer model C_j (unintended task U_j)
Compose learned backward transformations
$B_{j+1} \circ \cdots \circ B_k$

Figure 2: An overview of our framework. We train a new embedding model M_k and a light-weight backward transformation function B_k by optimizing the two training objectives simultaneously: (1) to solve the intended task T, and (2) to align ver-k embeddings $\{z_k\}$ to ver-$k-1$ embeddings $\{z_{k-1}\}$ using B_k. We use the latest-version embedding model M_k to produce embeddings $\{z_k\}$ and store them. For any existing consumer model C_j requesting a ver-j compatible embedding $\widetilde{z}_j$, we compose the learned backward transformations as $B_{j+1} \circ \cdots \circ B_k$ *on-the-fly, i.e.,* $\widetilde{z}_j = B_{j+1} \circ \cdots \circ B_k(z_k)$.

of the embeddings, and the old versions can be retired. In general, the problem of backward incompatible embeddings slows down iteration cycles and leads to significant human and computing cost.

Present work: Backward compatible embeddings. Here we study the problem of evolving embedding models and their backward compatibility (Figure 1). We formalize a realistic setting where the embedding team works on developing an *embedding model* M that is trained to predict a given *intended task* T (intended for the embedding team), *e.g.*, item recommendation. Over time, the embedding team adds new training data, keeps experimenting with different model architectures, embedding dimensions, and hyperparameters, which results in evolving *versions* of the embedding model, $M_0, M_1, M_2, \ldots$, where we use the subscript to indicate the version. Given an input data point x, each such embedding model M_i produces its own D_i-dimensional embedding $z_i \equiv M_i(x) \in \mathbb{R}^{D_i}$. Notice that at version k, we have $k+1$ versions of embeddings $z_0, z_1, \ldots, z_k$ for the same input data point x. We call z_i *ver-i embedding* of x. In practice, we have a collection of input data points, for which we use the embedding model to generate the embeddings. Such embeddings are then stored and shared with other teams. We use $\{\}$ to denote the collection of the generated embeddings, *e.g.*, $\{z_k\}$ denotes the collection of ver-k embeddings. Note that $\{z_k\}$ can be refreshed frequently as new data points arrive, *e.g.*, item embeddings may be refreshed every day as new items/iteractions arrive.

At the same time, we have many other *consumer teams* that utilize the produced embeddings to solve their *unintended tasks* (unintended for the embedding team), *e.g.*, fraud detection. Consider a consumer team that started to use the embeddings at some version j. They would use ver-j embeddings to train their consumer model C_j for their own unintended task U_j. However, later ver-k embeddings ($k > j$) are generally incompatible with consumer model C_j; hence, C_j cannot simply use ver-k embeddings as input. So, either the consumer model C_j need to be retrained and recalibrated on the later ver-k embeddings, or both the old ver-j embeddings have to be also maintained. Both solutions lead to significant cost and overhead.

We first formalize the problem where the goal is for the embedding team to keep improving and updating the embedding model, while the existing consumer teams do not have to retrain their models when a new version of the embedding model is released. We then develop a solution based on learning backward compatible embeddings (Figure 2), which allows the embedding model version to be updated frequently, while also allowing the latest-version embedding to be quickly transformed into any backward compatible historical version of it, so that the existing consumer teams do not have to retrain their models. Our key idea is that whenever a new embedding model M_k is trained, we learn it together with a light-weight *backward (compatibility) transformation* $B_k : \mathbb{R}^{D_k} \longrightarrow \mathbb{R}^{D_{k-1}}$ that aligns ver-k embeddings $z_k \in \mathbb{R}^k$ to its previous version $z_{k-1} \in \mathbb{R}^{k-1}$, *i.e.*, $B_i(z_k) \approx z_{k-1}$. At version k, we maintain the learned B_k as well as all past backward transformation functions learned so far: $B_1, B_2, \ldots, B_{k-1}$.

Importantly, our learned backward transformations can be composed to approximate any historical version of the embedding. Specifically, at version k, the latest-version embedding z_k can be transformed to approximate any historical ver-j embedding z_j ($j < k$) by $B_{j+1} \circ \cdots \circ B_{k-1} \circ B_k(z_k) \approx z_j$, where $\circ$ denotes the function composition. We call the transformed embedding *ver-j compatible embedding*, and denote it as $\widetilde{z}_j$. Because $\widetilde{z}_j \approx z_j$, the embedding $\widetilde{z}_j$ can be used by consumer model C_j in a compatible manner. The backward transformations are fast and lightweight so they can be applied on-the-fly whenever a consumer team requests a historical version of the latest-version embedding. Furthermore our solution is fully inductive: given an unseen input data point x, the learned backward transformations can be used to quickly transform the newly-generated ver-k embedding $z_k \equiv M_k(x)$ into its historically compatible version $\widetilde{z}_j$ for any $j < k$.

At version k, we use the linear model for B_k and jointly train it with M_k to solve the intended task T as well as to encourage B_k to align the embedding output of M_k to that of M_{k-1}, *i.e.*, $B_k \circ M_k \approx M_{k-1}$. Additionally, we develop a novel loss that suppresses the amplification of the alignment error caused by the composition of backward transformations. Altogether, we arrive at our proposed method, which we call BC-Aligner. In addition, we consider five other method variants under our framework, with different design choices of B_k, loss function, and training strategy of B_k and M_k, some of which includes prior methods [12, 16].

To systematically evaluate different methods, we introduce a realistic experimental setting in the real-world recommender system application [22]. We consider link prediction as the intended task T, graph neural networks as the embedding model M, and propose five different unintended tasks U that are of practical interest. We empirically show that BC-Aligner provides the best performance compared to the other method variants including the prior works. BC-Aligner maintains nearly-perfect backward compatibility with existing unintended tasks even after multiple rounds of embedding model updates over time. Simultaneously, BC-Aligner achieves the intended task performance comparable to the embedding model that is solely optimized for the intended task.

Overall, our work presents the first step towards solving the critical problem of incompatibility between the embedding model and unintended downstream consumer models. We hope our work spurs an interest in community to solve the new problem setting.

2 PROBLEM SETTING

Here we formalize our problem setting. We ground our application to recommender systems, though our formulation is general and applicable to broader application domains that use embeddings to perform a wide range of tasks, such as computer vision, natural language processing, search and information retrieval. The new concepts and terminology introduced in this paper are bolded.

2.1 Terminology and Setting

We consider two types of machine learning tasks: The ***intended task*** T and ***unintended tasks*** U. The intended task is the task that the embedding team originally intended to solve and is solved by using embeddings produced by a deep learning model, which we call the ***embedding model*** M. This embedding model is trained to solve the intended task T. At the same time, these embeddings can be used by many consumer teams to solve their tasks that may not be originally intended by the embedding team; we call such a task the unintended task U. Precisely, to solve the unintended task U, a consumer team trains their ***consumer model*** C on top of the produced embeddings.

The above setting is prevalent in industry, where the produced embeddings are widely shared within the organization to power a wide variety of unintended tasks [22]. As a concrete example, let us consider a recommender system application. The intended task T could be user-item interaction prediction. We can use a Graph Neural Network (GNN) [5, 22] as the embedding model M to generate user and item embeddings, which are then used to produce the likelihood score that the user will interact with a given item. At the same time these embeddings can be used by many consumer teams to perform their unintended tasks. For instance, a consumer team can use the item embeddings to build model C to solve the task U of detecting fraudulent items.

Embedding model version updates. The embedding team updates the embedding model M every once in a while to improve the performance on the intended task T. We use $M_0, M_1, M_2, \ldots$ to denote the evolving versions of the embedding model, where M_k is the k-th model version. At version k, we learn M_k to solve T by minimizing the objective:

$$L_k(M_k). \tag{1}$$

Given single input data x, each ***ver-k embedding model*** M_k produces ***ver-k embedding*** $\boldsymbol{z}_k \equiv M_k(x)$. The collection of the ver-k embeddings is denoted as $\{\boldsymbol{z}_k\}$, which is computed over current sets of input data points and may be refreshed as new data arrives. Moreover, for each version k, we consider a consumer team that uses ver-k embeddings and consumer model C_k to solve their unintended task U_k.

Compatibility of embeddings. Different versions $j < k$ of the embeddings $\boldsymbol{z}_k, \boldsymbol{z}_j$ may be *incompatible* because of the difference between M_k and M_j that generate them. This presents an issue that consumer model C_j trained on ver-j embeddings will not be compatible with the later ver-k embeddings. Feeding $\boldsymbol{z}_k$ into C_j will give random/arbitrary predictions.

To resolve this issue, in Section 3, we develop a cost-efficient framework to generate ***ver-j compatible embedding*** $\widetilde{\boldsymbol{z}}_j$ from later-version embedding $\boldsymbol{z}_k$ such that feeding $\widetilde{\boldsymbol{z}}_j$ into C_j gives robust predictive performance. We consider the problem of ***backward compatible embedding learning***, *i.e.*, learn to produce $\widetilde{\boldsymbol{z}}_j$ from $\boldsymbol{z}_k$ for any historical version $j < k$.

2.2 Generality of our Setting

We show that the above simple problem setting is general enough to capture complex real-world scenarios.

Complex embedding model evolution. Our assumption on the evolving embedding model M is minimal: Each model version M_k just needs to output an embedding given an input data point. Our setting, therefore, allows different M_k's to have different internal model architectures and output dimensionality. For instance, our setting allows a later-version embedding model to use a more advanced model architecture.

Evolving training data and loss functions. Not only can different M_k's have different architectures, they can also be trained on different data and with different loss functions. All such differences are absorbed into ver-k-specific objective L_k in Eq. (1). For instance, our setting allows later-version embedding model to be trained on more data with an improved loss function.

Multiple different intended tasks. We consider a single shared intended task T for simplicity, but our setting naturally handles multiple intended tasks that are different for different embedding model versions. This is because we only assume each M_k to be trained on objective L_k, which can be designed to solve different intended tasks for different version k.

Multiple consumer teams. For each version k, we consider a single consumer team solving U_k using model C_k for simplicity. However, our setting naturally handles *multiple* consumer teams using the same embedding version k. We can simply index the consumer teams as solving $U_{k,0}, U_{k,1}, U_{k,2}, \ldots$ using models $C_{k,0}, C_{k,1}, C_{k,2}, \ldots$, respectively.

3 METHODOLOGICAL FRAMEWORK

Here we present our framework to learn backward compatible embeddings. At version k, our framework mainly keeps the latest ver-k embedding model M_k and its generated embeddings $\{\boldsymbol{z}_k\}$; hence, we say our framework follows the ***keep-latest*** (embedding model) approach. We start by contrasting it with what we call ***keep-all*** (embedding models).

3.1 Ideal but Costly Baseline: Keep-All

Given unlimited human and computational budget, we can simply *keep all versions of the embedding model* $M_0, M_1, \ldots, M_k$ and let them produce embeddings $\{\boldsymbol{z}_0\}, \{\boldsymbol{z}_1\}, \ldots, \{\boldsymbol{z}_k\}$. Then, for any $j \leq k$, $\{\boldsymbol{z}_j\}$ can be directly used by consumer model C_j in a compatible manner. We refer to this approach as *keep-all* (embedding models), which we formalize below.

Training setting. At version k, we learn each M_k by minimizing objective L_k of Eq. (1) to solve the intended task T. Once M_k is trained to produce ver-k embeddings, a consumer team trains their model C_k on them to solve their unintended task U_k.

Inference setting. At version k, we keep all versions of the embedding model learned so far: $M_0, M_1, \ldots, M_k$. To solve the intended task T, we use the latest-version embedding model M_k to produce embeddings $\{\boldsymbol{z}_k\}$ and store them. In addition, we use all the historical versions of the embedding model to produce the embeddings $\{\boldsymbol{z}_0\}, \{\boldsymbol{z}_1\}, \ldots, \{\boldsymbol{z}_{k-1}\}$ and store all of them. For any $j \leq k$, we can perform unintended task U_j by simply feeding compatible embeddings $\{\boldsymbol{z}_j\}$ to consumer model C_j.

Issues with Keep-All. The issue with the keep-all approach is that it is too costly in large-scale applications, *e.g.*, web-scale recommender systems that utilize billions of embeddings [22, 24]. This is because embeddings need to be produced and stored for *every* version.[2] The cost of maintaining all versions of the embeddings and the embedding model quickly grows especially when the embedding model version is updated frequently. Despite the impracticality, the keep-all approach sets the high standard in terms of the intended and unintended task performance, which we try to approximate with our cost-efficient framework.

3.2 Our Framework: Keep-Latest

Our framework follows the keep-latest approach, where only the latest-version embedding model and embeddings are kept at any given timestamp.

Backward transformation. The key to our approach is to learn a ***backward (compatibility) transformation*** $B_k : \mathbb{R}^{D_k} \rightarrow \mathbb{R}^{D_{k-1}}$ that aligns ver-k embedding $\boldsymbol{z}_k \in \mathbb{R}^{D_k}$ to its previous version $\boldsymbol{z}_{k-1} \in \mathbb{R}^{D_{k-1}}$. B_k has to be light-weight so that the alignment can be performed cheaply on-the-fly. Whenever we update M_{k-1} to M_k, we learn the backward transformation B_k to align the output of M_k back to that of M_{k-1}. At version k, we maintain all the backward transformations learned so far: $B_1, \ldots, B_k$. Maintaining $B_1, \ldots, B_k$ is much cheaper than maintaining and storing all historical versions of the embeddings $\{\boldsymbol{z}_0\}, \{\boldsymbol{z}_1\}, \ldots, \{\boldsymbol{z}_{k-1}\}$.

The key insight is that we can compose the backward transformations to align $\boldsymbol{z}_k$ into any of its historical version $j < k$. Let us introduce the *composed backward function* $B_k^j \equiv B_{j+1} \circ \cdots \circ B_k$. We see that $B_k^{k-1} \equiv B_k$ and B_k^j aligns ver-k embedding $\boldsymbol{z}_k$ to ver-j embedding $\boldsymbol{z}_j$. As the alignment may not be perfect in practice, we say B_k^j transforms $\boldsymbol{z}_k$ into ver-j *compatible* embedding $\widetilde{\boldsymbol{z}}_j$:

$$\widetilde{\boldsymbol{z}}_j = B_k^j(\boldsymbol{z}_k). \tag{2}$$

Our aim is to have $\widetilde{\boldsymbol{z}}_j \approx \boldsymbol{z}_j$ so that $\widetilde{\boldsymbol{z}}_j$ can be fed into C_j to give robust predictive performance on unintended task U_j.

Function alignment. We wish to use B_k to align $\boldsymbol{z}_k \equiv M_k(x)$ to $\boldsymbol{z}_{k-1} \equiv M_{k-1}(x)$ for every x, which reduces to aligning two *functions*: $B_k \circ M_k$ and M_{k-1}. We introduce function alignment objective $L_{\text{align}}(B_k \circ M_k, M_{k-1})$, which encourages the two functions to be similar, *i.e.*, given same input, produce similar output. We discuss the specific realization of L_{align} in Section 3.2.2.

Training setting. We propose to add the function alignment objective L_{align} to the original objective L_k for solving T:

$$L_k(M_k) + \lambda \cdot L_{\text{align}}(B_k \circ M_k, M_{k-1}), \tag{3}$$

where $\lambda > 0$ is a trade-off hyper-parameter. At version k, parameters of M_k and B_k are learned, and the parameters of the previous version M_{k-1} are fixed. Once M_k and B_k are learned, we can safely discard M_{k-1} because M_{k-1} can be approximately reproduced by $B_k \circ M_k$.

Inference setting. At version k, we only need to maintain the latest-version embedding model M_k, and a series of transformation functions learned so far: $B_1, \ldots, B_k$. Consider an ideal situation after minimizing Eq. (3), where we have the perfect function alignment *for every version* until version k. Then, we have the following single-step equations:

$$B_k \circ M_k = M_{k-1},\ B_{k-1} \circ M_{k-1} = M_{k-2}, \ldots,\ B_1 \circ M_1 = M_0. \tag{4}$$

In this ideal situation, we see that the composed backward function in Eq. (2) can exactly reproduce *any* historical version of the embedding model $M_j (j < k)$ as

$$B_k^j \circ M_k = M_j. \tag{5}$$

In practice, each equation in Eq. (4) only holds approximately, and the approximation error of Eq. (5) increases for smaller j or larger k due to more accumulation of the single-step approximation errors in Eq. (4). In our experiments, however, we find that the approximation error stays relatively stable over time and only increases sub-linearly with larger k (*i.e.*, using later-version embedding model to approximate M_0). In Section 3.2.2, we mathematically analyze the approximation error and provide a possible explanation for the robust approximation performance even after multiple rounds of model updates.

Inductive capability. Eq. (5), or more realistically, $B_k^j \circ M_k \approx M_j$, implies that our framework is fully inductive. Given unseen data x, we can first obtain its latest ver-k embedding $\boldsymbol{z}_k$ and store it. Then, ver-j compatible embedding $\widetilde{\boldsymbol{z}}_j$ for any j can be quickly obtained by

[2]We assume the inference cost of M_k is high, which makes it costly to infer ver-k embedding every time it is requested.

$B_k^j(\boldsymbol{z}_k)$ and is expected to be similar to the actual ver-j embedding $\boldsymbol{z}_j \equiv M_j(x)$. Importantly, $\widetilde{\boldsymbol{z}}_j$ is obtained on-the-fly without being stored nor requiring the past model M_j. In our experiments, we utilize this inductive capability of B_k^j to transform embeddings unseen during training.

3.2.1 *Choices of* B_k.

We want backward transformation B_k to be light-weight. Here we consider two natural choices.

Linear. We use $B_k(\boldsymbol{z}_k) = \boldsymbol{W}_k \boldsymbol{z}_k$, where $\boldsymbol{W}_k \in \mathbb{R}^{D_{k-1} \times D_k}$ is a learnable weight matrix. In this case, Eq. (2) is written as

$$\widetilde{\boldsymbol{z}}_j = \boldsymbol{W}_k^j \boldsymbol{z}_k, \tag{6}$$

where $\boldsymbol{W}_k^j \equiv \boldsymbol{W}_{j+1} \cdots \boldsymbol{W}_k \in \mathbb{R}^{D_j \times D_k}$ is pre-computed for every j.

NoTrans. As a baseline, we also consider not applying any transformation to $\boldsymbol{z}_k$. In other words, our backward transformation functions are all identity functions. This means that when training M_k, the produced ver-k embedding $\boldsymbol{z}_k$ is learned to be directly similar to its previous version $\boldsymbol{z}_{k-1}$. Therefore, $\boldsymbol{z}_k$ is directly backward compatible with $\boldsymbol{z}_{k-1}$ and any of its historical version $\boldsymbol{z}_j$. In the case of $D_k \geq D_{k-1}$, we simply take the first D_{k-1} elements of $\boldsymbol{z}_k$.

Remark on limited expressiveness of NoTrans. NoTrans seems like a very natural solution to our problem as it enforces $\boldsymbol{z}_k$ to be directly similar to $\boldsymbol{z}_{k-1}$ (*e.g.*, by minimizing Euclidean distance between embeddings $\boldsymbol{z}_k$ and $\boldsymbol{z}_{k-1}$). However, NoTrans is not desirable when the embeddings suitable for performing the intended task T change over time as a result of distribution shift. For instance, in recommender systems, users' interests and items' popularity change over time, so we want their embeddings to also change over time, which is discouraged in the NoTrans case. In contrast, the Linear case allows $\boldsymbol{z}_k$ to be different from $\boldsymbol{z}_{k-1}$. The additional expressiveness is crucial for $\boldsymbol{z}_k$ to perform well on the intended task T, as we will show in our experiments.

3.2.2 *Choices of* L_{align} *and Preventing Error Amplification.*

Single-step alignment loss. The role of the alignment objective L_{align} is to make $B_k \circ M_k$ similar to M_{k-1}. We enforce the alignment on a set of data points $\mathcal{X} = \{x\}$, which we assume to be given. For instance, in recommender systems, $\mathcal{X}$ can simply be all the users and items. Then, our alignment objective becomes:

$$\begin{aligned} L_{\text{align}}\left(B_k \circ M_k, M_{k-1}\right) &= \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \|B_k \circ M_k(x) - M_{k-1}(x)\|^2 \\ &= \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \|B_k\left(\boldsymbol{z}_k\right) - \boldsymbol{z}_{k-1}\|^2 \\ &= \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \|\boldsymbol{\delta}_k(x)\|^2, \end{aligned} \tag{7}$$

where $\boldsymbol{\delta}_k(x) \equiv B_k\left(\boldsymbol{z}_k\right) - \boldsymbol{z}_{k-1}$ is the *single-step* alignment error between $B_k\left(\boldsymbol{z}_k\right)$ and $\boldsymbol{z}_{k-1}$ on a data point x.

While natural, it is unclear how well the single-step alignment loss of Eq. (7) enforces the small *multi-step* alignment error $\boldsymbol{\delta}_k^j(x) \equiv B_k^j\left(\boldsymbol{z}_k\right) - \boldsymbol{z}_j$, where $j < k-1$. Below we mathematically characterize their relation.

Error amplification. Let us focus on the linear case of Eq. (6). Then, the multi-step alignment error on a single data point x becomes:

$$\boldsymbol{\delta}_k^j(x) = \boldsymbol{W}_k^j \boldsymbol{z}_k - \boldsymbol{z}_j. \tag{8}$$

We note that Eq. (8) *cannot* be optimized directly because the keep-latest approach assumes M_j and $\boldsymbol{z}_j$ are no longer kept when M_k is being developed. However, we learn from Eq. (7) that all the historical backward transformation weights $\boldsymbol{W}_1, \boldsymbol{W}_2, \ldots, \boldsymbol{W}_{k-1}$, have been learned to minimize the L2 norm of the single-step alignment errors, $\boldsymbol{\delta}_1(x), \boldsymbol{\delta}_2(x), \ldots, \boldsymbol{\delta}_{k-1}(x)$, respectively.

We can rewrite the multi-step alignment error $\boldsymbol{\delta}_k^j(x)$ in Eq. (8) using the single-step alignment errors, $\boldsymbol{\delta}_{j+1}(x), \boldsymbol{\delta}_{j+2}(x), \ldots, \boldsymbol{\delta}_k(x)$:

$$\begin{aligned} \boldsymbol{\delta}_k^j(x) &= \boldsymbol{W}_{k-1}^j \left(\boldsymbol{W}_k \boldsymbol{z}_k\right) - \boldsymbol{z}_j \\ &= \boldsymbol{W}_{k-1}^j \left(\boldsymbol{z}_{k-1} + \boldsymbol{\delta}_k(x)\right) - \boldsymbol{z}_j \\ &= \left\{\boldsymbol{W}_{k-2}^j \left(\boldsymbol{W}_{k-1} \boldsymbol{z}_{k-1}\right) - \boldsymbol{z}_j\right\} + \boldsymbol{W}_{k-1}^j \boldsymbol{\delta}_k(x) \\ &\quad \vdots \\ &= \left(\boldsymbol{W}_{j+1}^j \boldsymbol{z}_{j+1} - \boldsymbol{z}_j\right) + \boldsymbol{W}_{j+1}^j \boldsymbol{\delta}_{j+2}(x) + \cdots + \boldsymbol{W}_{k-1}^j \boldsymbol{\delta}_k(x) \\ &= \boldsymbol{\delta}_{j+1}(x) + \boldsymbol{W}_{j+1}^j \boldsymbol{\delta}_{j+2}(x) + \cdots + \boldsymbol{W}_{k-1}^j \boldsymbol{\delta}_k(x). \end{aligned} \tag{9}$$

From Eq. (9), we see that the single-step errors are not simply added up but are potentially *amplified* by the historical backward transformation weights. We call this ***error amplification***. Minimizing the single-step error does not necessarily lead to the smaller amplified error, causing the large multi-step alignment error of Eq. (8) in practice. This in turn deteriorates the unintended task performance.

Multi-step alignment loss. To suppress the error amplification, here we develop the *multi-step alignment loss*. We see from the last term of Eq. (9) that the error $\boldsymbol{\delta}_k(x)$ made at version k (learning M_k and $\boldsymbol{W}_k$) gets amplified by $\boldsymbol{W}_{k-1}^j$ in the multi-step error $\boldsymbol{\delta}_k^j(x)$. Our multi-step alignment loss explicitly suppresses the amplified error for every $j = 0, \ldots, k-2, k-1$:

$$\frac{1}{k} \cdot \left(\left\|\boldsymbol{W}_{k-1}^0 \boldsymbol{\delta}_k(x)\right\|^2 + \cdots + \left\|\boldsymbol{W}_{k-1}^{k-2} \boldsymbol{\delta}_k(x)\right\|^2 + \left\|\boldsymbol{W}_{k-1}^{k-1} \boldsymbol{\delta}_k(x)\right\|^2\right), \tag{10}$$

where we note that $\boldsymbol{W}_{k-1}^{k-1} = \boldsymbol{I}$. Thus, our multi-step alignment loss ensures the error $\boldsymbol{\delta}_k(x)$ made at version k would not get amplified when we compute its historical version $\widetilde{\boldsymbol{z}}_j$ for any $j < k$.

The final multi-step alignment loss is the average of Eq. (10) over $x \in \mathcal{X}$, analogous to Eq. (7). Although Eq. (10) contains k terms, computing the loss itself is often much cheaper than computing embedding $\boldsymbol{z}_k$, so Eq. (10) adds negligible computational cost compared to the single-step loss of Eq. (7).

Remark on the NoTrans case. We note that NoTrans method does not suffer from the error amplification. To see this, we can replace all the weight matrices in Eq. (9) with identity matrices, resulting in the simple additive accumulation of the non-amplified errors. However, NoTrans suffers from the limited expressiveness, as discussed in Section 3.2.1.

Remark on Error Accumulation. As shown in Eq. (9), both the Linear and NoTrans would suffer from the additive accumulation of the single-step errors. However, in practice, the L2 norm of the

Table 1: A set of all 6 method variants we consider under our framework. We vary transformation function (linear vs. no transformation), alignment loss function (single- vs. multi-step) and alignment (joint vs. posthoc)[3].

Trans function / L_{align}	Joint-Align	Posthoc-Align
Linear / Single-Step-Loss	Joint-Lin-SLoss	Post-Lin-SLoss [11]
Linear / Multi-Step-Loss	Joint-Lin-MLoss **(BC-Aligner)**	Post-Lin-MLoss
NoTrans / Single-Step-Loss	Joint-NoTrans [16]	Non-BC

multi-step error grows gradually (Figure 3). As a result, the unintended task performance stays robust even after multiple rounds of embedding version updates (Figure 4)

3.2.3 Choices of Training Strategies of B_k. We consider two strategies to train B_k via Eq. (3).

Joint-Align. We jointly train B_k with M_k to minimize Eq. (3).

Posthoc-Align. We first train M_k to minimize L_k. We then fix M_k and train B_k to minimize L_{align} in a post-hoc manner.

3.2.4 Method Variants. In all, we explore six different method variants under our framework, as shown in Table 1. We note that the techniques used in some method variants were already presented by prior works in different contexts. Specifically, the Joint-NoTrans was originally presented by [16] in the context of backward compatible representation learning in open-set image recognition. The Posthoc-Lin-SLoss is broadly adopted in cross-lingual word embedding alignment [11]. Details discussion are in Section 6.

Empirically, we will show that these two variants are outperformed by the best method under our framework, namely Joint-Lin-MLoss. We give a special name to this method, BC-Aligner.

4 EVALUATION FRAMEWORK

Here we present an evaluation framework to measure the success of the keep-latest approach presented in Section 3. We consider a series of embedding model updates, $M_0, \cdots, M_K$ to improve the performance of the intended task T. At the same time, consumer teams train their models to solve their unintended tasks. Without loss of generally, we consider multiple consumer teams that use ver-0 embeddings (generated by M_0) to solve their unintended tasks.

We provide three summary metrics calculated at every version $k = 0, 1, \ldots, K$. To make the comparisons meaningful, we assume the keep-all and keep-latest approaches share the same base objective L_k and model architecture M_k for every k.

(1) Degradation of intended task performance compared to keep-all. The keep-all approach provides an upper bound in terms of the intended task T performance, as M_k is solely optimized for L_k. Note, however, that this upper bound is impractical to achieve in most settings, as the keep-all approach is prohibitively costly. The keep-latest approach (and potentially other approaches) needs to maintain backward compatibility in addition to optimizing for L_k, which could deteriorate its intended task performance. Therefore we measure the intended task performance *degradation* compared to M_k trained solely with L_k.

(2) Degradation of unintended task performance compared to keep-all. The keep-all approach also provides an upper bound in terms of the unintended task U_0 performance. This is because consumer model C_0 is optimized to perform well on ver-0 embeddings, which can be directly produced by the keep-all approach via kept M_0. On the other hand, at version k, the keep-latest approach does not have access to M_0 and can only approximate $\boldsymbol{z}_0$ by backward compatible embedding $\widetilde{\boldsymbol{z}}_0$. Therefore, we measure the unintended task U_0 performance *degradation* when using ver-0 compatible embedding $\widetilde{\boldsymbol{z}}_0$ instead of the actual ver-0 embedding $\boldsymbol{z}_0$.

(3) Embedding alignment error. Metric (2) above is dependent on the choice of the unintended task U_0, which may not cover the entire spectrum of unintended tasks for which the embeddings can be used. It is therefore useful to have task-agnostic metric that generally correlates well with a wide range of unintended tasks. To this end, we propose to measure the embedding alignment error between $\widetilde{\boldsymbol{z}}_0$ (*e.g.*, we use $B_1 \circ \cdots \circ B_k\,(\boldsymbol{z}_k)$ in our methods) and the actual $\boldsymbol{z}_0$. We calculate it as the L2 distance between $\widetilde{\boldsymbol{z}}_0$ and $\boldsymbol{z}_0$, which we then average over a set of data points.

Table 2: Statistics of 3 different dynamic datasets we use.

Dataset	#Users	#Items	#Interact.	Feature stats #Brands	#Sub cat.
Musical Instruments	27,530	10,611	231,312	391	349
Video Games	55,223	17,389	496,315	330	149
Grocery	127,496	41,280	1,143,063	1,806	774

5 EXPERIMENTS

We evaluate our methods following the evaluation protocol presented in Section 4. We start with introducing a new benchmark in Section 5.1. Then, we present experimental results in Section 5.2.

5.1 Recommender System Benchmark

5.1.1 Overview. We consider evolving user-item bipartite graph datasets in recommender systems, where new users/items and their interactions appear over time. As the intended task, we consider the standard user-item link prediction task, and use GNNs trained on this task as the embedding model that generate user/item embeddings. To simulate the phenomenon of increasing dataset sizes and model capabilities over time, we consider larger GNN models trained on more edges over time. We consider five different unintended tasks that are of interest to practitioners. Below we explain the benchmark in more details.

5.1.2 Datasets. We use public Amazon Product Reviews dataset[4] that contains timestamped Amazon product reviews, spanning May 1996 - July 2014 [13]. The entire dataset is partitioned according to the categories of products/items. In our experiments, we use the datasets from three categories: Musical Instruments, Video Games, and Grocery. Each item has brand and subcategory features, that can be expressed as multi-hot vectors. User features are not available in this dataset. Dataset statistics is summarized in Table 2.

[3]The NoTrans model does not require the multi-step alignment loss. The combination of the NoTrans model and the Posthoc-Align does *not* guarantee backward compatibility, as the identity has no parameter to learn in the post-hoc alignment stage; hence, we call it "Non-BC".

[4]Available at https://jmcauley.ucsd.edu/data/amazon/

Table 3: Results over 3 dataset. Absolute performance of the Keep-All is included in brackets. For all the metrics, the relative degradation is computed after the average is taken across different timestamps and the five unintended tasks. We see from the 3rd column that the BC-Aligner provides the best trade-off between the intended and unintended task performance, yielding the closest performance to the costly Keep-All approach.

Dataset	**Approach**	**Method**	**(1) Intented task Degradation** from Keep-All (%) ↑	**(2) Unintended task Degradation** from Keep-All (%) ↑	**(1)+(2) Degradation** from Keep-All (%) ↑	**(3) Emb Align Error** ↓
Musical Instruments	Keep-All	Keep-All (Abs. perf.)	-0.00 (12.13)	-0.00 (68.66)	-0.00	0.00
	Keep-M_0	Fix-M_0	-26.81	-0.00	-26.81	0.00
		Finetune-M_0 [3]	-7.69	-7.46	-15.15	1.01
	Keep-Latest	Non-BC	-0.00	-26.45	-26.45	2.61
		Post-Lin-SLoss [11]	-0.00	-10.25	-10.25	1.27
		Post-Lin-MLoss	-0.00	-14.43	-14.43	1.34
		Joint-NoTrans [16]	-9.00	-0.80	-9.80	0.41
		Joint-Lin-SLoss	-3.67	-1.07	-4.74	0.48
		BC-Aligner	-2.96	-0.65	**-3.62**	**0.38**
Video Games	Keep-All	Keep-All (Abs. perf.)	-0.00 (12.69)	-0.00 (72.76)	-0.00	-0.00
	Keep-M_0	Fix-M_0	-25.55	-0.00	-25.55	0.00
		Finetune-M_0 [3]	-10.61	-6.72	-17.32	0.87
	Keep-Latest	Non-BC	-0.00	-30.81	-30.81	2.65
		Post-Lin-SLoss [11]	-0.00	-8.25	-8.25	1.10
		Post-Lin-MLoss	-0.00	-14.56	-14.56	2.51
		Joint-NoTrans [16]	-9.12	-0.62	-9.74	0.32
		Joint-Lin-SLoss	-3.98	-1.03	-5.01	0.40
		BC-Aligner	-3.35	-0.59	**-3.94**	**0.28**
Grocery	Keep-All	Keep-All (Abs. perf.)	-0.00 (7.78)	-0.00 (65.82)	-0.00	-0.00
	Keep-M_0	Fix-M_0	-27.79	-0.00	-27.79	0.00
		Finetune-M_0 [3]	-24.74	-5.52	-30.26	1.12
	Keep-Latest	Non-BC	-0.00	-22.07	-22.07	2.69
		Post-Lin-SLoss [11]	-0.00	-6.36	-6.36	1.30
		Post-Lin-MLoss	-0.00	-15.69	-15.69	4.84
		Joint-NoTrans [16]	-11.31	-0.33	-11.64	0.18
		Joint-Lin-SLoss	-2.90	-2.17	-5.07	0.52
		BC-Aligner	-3.21	-0.07	**-3.28**	**0.13**

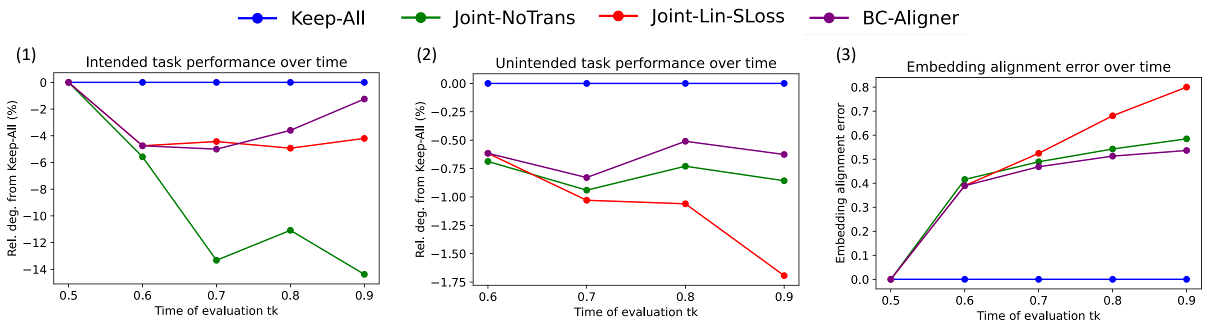

Figure 3: Performance over time. For the sub-figures (1) and (2), we plot the relative performance degradation from Keep-All in the y-axis (closer to zero, the better).

The datasets can be naturally modeled as evolving graphs, where nodes represent users/items, and each edge with a timestamp represents a user-item interaction, *i.e.*, a user review an item at the particular timestamp. We scale the timestamp between 0 and 1, representing the ratio of edges observed so far, *e.g.*, a timestamp of 0.6 means 60% of edges have been observed until that timestamp. This allows us to compare methods across the datasets.

We consider embedding models to be updated at $t_0 = 0.5, t_1 = 0.6, \ldots t_K = 0.9, K = 4$. In total, five versions of the embedding model are developed. We use E_k to denote the set of all edges up to t_k.

5.1.3 *Intended Task.*

As the intended task, we consider the standard link prediction in recommender systems. [19]. At every timestamp t_k for $k = 0, \ldots, K$, we train M_k on E_k and use it to predict on the edges between time t_k and t_{k+1}, *i.e.*, $E_{k+1} \setminus E_k$, where $t_{K+1} = 1$ by construction. We follow the same strategy as [6, 19] to train and evaluate M_k. Specifically, given user/item embeddings generated by M_k, we use the dot product to score the user-item interaction and use the BPR loss [14] to train M_k. We then evaluate M_k on $E_{k+1} \setminus E_k$ using Recall@50.

5.1.4 *Embedding Models.*

We use the GraphSAGE models [5] as our core embedding models. To simulate the updates in model

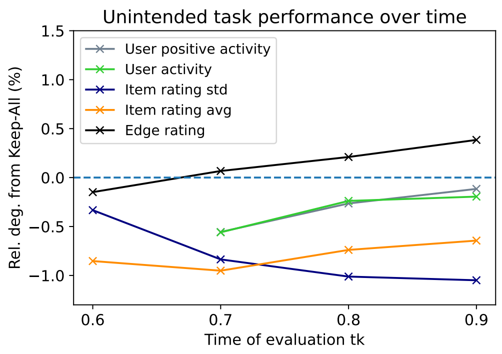

Figure 4: Unintended task performance degradation of BC-Aligner compared to Keep-All (dotted line) over time. The performance degradation of each unintended task stays relatively stable over time.

architecture over time, we start with a small GraphSAGE mode at t_0 and make it larger (both deeper and wider) over time. Please refer to Appendix A.1 for more details.

5.1.5 Methods and Baselines. We consider the six methods under our framework, as depicted in Table 1. They all follow the cost-efficient Keep-Latest approach. For the joint-training methods, we set $\lambda = 16$ in Eq. (3) unless otherwise specified.

We also consider the following two cost-efficient baseline methods that only keep the ver-0 embedding model M_0 (as opposed to the latest embedding model); we group the methods under **Keep-**M_0.

Fix-M_0. We train M_0 at timestamp t_0 and fix its parameters throughout the subsequent timestamps. For all the timestamps, the same M_0 is used to perform the link prediction task as well as to generate the embeddings for the unintended tasks. As ver-0 embeddings are always produced, there is no backward compatibility issue for Fix-M_0. However, the method always uses M_0 and cannot benefit from additional data and better model architectures available in the future. This will impact performance on the intended task T.

Finetune-M_0. We also consider a method more advanced than the Fix-M_0, originally introduced by [3]. Specifically, we train M_0 at timestamp t_0. Then, in the subsequent timestamp t_k with $1 \leq k$, we finetune M_{k-1} to obtain M_k. Note that fine-tuning allows M_0 to learn from more data over time. However, the approach cannot benefit from improved model architecture over time, as fine-tuning is not possible for different model architectures.

We evaluate the methods against the costly Keep-All approach by measuring the performance degradation and the embedding alignment error explained in Section 4. We specifically consider the relative performance degradation in %, which is calculated as $\frac{100\cdot(\text{Perf}-\text{Perf}_{\text{Keep-All}})}{\text{Perf}_{\text{Keep-All}}}$, where Perf is the performance of a method of interest. The value should be negative in most cases (as $\text{Perf}_{\text{Keep-All}}$ is the upper bound); the closer to zero, the better. The trained M_0 is exactly the same across all the methods, as all the methods train M_0 using only L_0 and the same random seed.

Note that at every timestamp, we encounter new data (*e.g.*, existing users/items with more interactions, new users/items), which are never seen at previous timestamps. As our embedding model and transformations are inductive, we can generate backward compatible embeddings $\widetilde{\boldsymbol{z}}_0$ on the new data and make its prediction.

5.1.6 Unintended Tasks. We prepare five unintended binary classification tasks that include prediction on users, items, and user-item interactions. We utilize the review rating information in designing the unintended tasks, which is not used by the intended link prediction task. For all the tasks, we use 1-hidden-layer MLPs and evaluate the performance using ROC-AUC. Refer to Appendix A.2 for details.

- **User activity prediction:** Predict whether a given user will interact with at least a single item in the near future.
- **User positive activity prediction:** Predict if a given user will have at least one positive interaction in the near future.
- **Item rating avg prediction:** Predict whether the average rating of a given item until the near future will be above a threshold.
- **Item rating std prediction:** Predict whether the standard deviation of the ratings of a given item until the near future will be above a threshold.
- **Edge rating prediction:** Predict whether a given user will give a positive rating to a given item.

5.2 Results

In Table 3, we summarize the averaged results of different methods on the three metrics presented in Section 4, namely, (1) intended task degradation, (2) unintended task degradation, and (3) alignment error. We use the simple addition of intended task degradation and unintended task degradation as the unified metric to capture the trade-off between the intended and unintended task performance.

First, we observe high correlation between the unintended task degradation and alignment error; the smaller the unintended task performance degradation is, the smaller the embedding alignment error is. This validates our claim that alignment error can be used as a general proxy for unintended task performance degradation.

Second, Non-BC suffers from large unintended task degradation and alignment error, as the new versions of the embedding model are not trained to be backward compatible.

Third, we see that Fix-M_0 suffers from large intended task degradation, indicating that it is highly sub-optimal to not update the embedding model over time to improve the intended task performance. The more advanced Finetune-M_0 [3] still suffers from large intended task degradation due to its inability of adopting the new model architectures over time. Moreover, the unintended task performance of Finetune-M_0 degrades by 5–7%, implying that fine-tuned embedding models are generally *no longer* compatible with the original model.

Fourth, Post-Lin-SLoss [11] performs poorly on unintended task, degrading the performance by 6–10%. This is likely due to the large embedding alignment error.

Fifth, Joint-NoTrans [16] provides the small unintended task degradation and alignment error, but falls short on the intended task, degrading its performance by 9–11%. Compared to Joint-NoTrans, Joint-Lin-SLoss produces gives smaller intended task degradation. However, Joint-Lin-SLoss performs relatively poorly on unintended

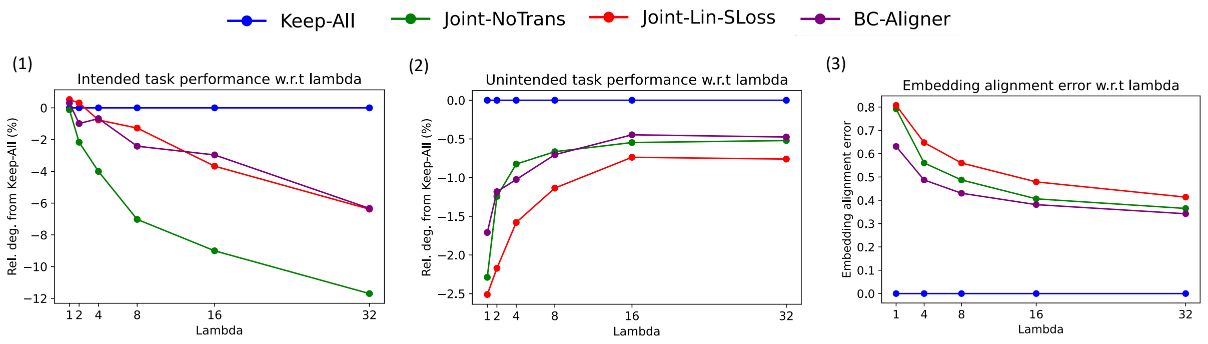

Figure 5: Performance as a function of the trade-off hyper-parameter λ. For sub-figures (1) and (2), we plot the relative degradation from Keep-All (closer to zero, the better).

tasks, degrading the performance by 1–2%. As we will see in Figure 3, this is possibly due to the amplification of the single-step embedding alignment error.

Overall, BC-Aligner provides the best trade-off between the intended and unintended task performance, only suffering from around 3% (resp. 0.5%) degradation in the intended (resp. unintended) task performance. It also achieves the smallest embedding approximation error among all the methods.

From now on, we consider the Musical Instruments dataset for all results. We focus on the three methods that give the most promising averaged results: Joint-NoTrans, Joint-Lin-SLoss, and BC-Aligner.

Results over time. Figure 3 shows the intended task degradation, unintended task degradation, and alignment error over time. For all the metrics, we observe that BC-Aligner provides the best performance *across almost all the timestamps*. In Figure 3 (3), we see the sharp increase in the embedding alignment error over time for Joint-Lin-Sloss. This is likely due to the error amplification, as the single-step error (the error at $t = 0.6$) is comparable across the methods. Indeed, once the multi-step alignment loss is used in BC-Aligner, the embedding alignment error increases less sharply.

Results on each unintended task performance over time. Figure 4 shows the performance degradation of each unintended task over time, when BC-Aligner is used. We see that the degradation from Keep-All is relatively stable over time.

Averaged results with varying λ. Figure 5 shows how the trade-off parameter λ in Eq. (3) affects the three metrics. We consider $\lambda \in \{1, 2, 4, 8, 16, 32\}$, and the results are averaged over timestamps and unintended tasks. As we increase λ, all the methods have larger degradation in the intended task performance, smaller degradation in the unintended task performance, and smaller embedding alignment error, as expected. For fixed λ, BC-Aligner often gives the best or comparable performance compared to the other methods. In practice, λ should be chosen based on the trade-off one wants to achieve between the intended and unintended task performance.

6 RELATED WORK

Backward compatible representation learning. Our problem formulation shares a similar motivation as [16], which considers backward compatible representation learning for open-set image recognition. Meng et al. [10], Shen et al. [16] update the embedding model so that embeddings computed by the updated model are directly comparable to those generated by the previous model. Our work differs from this work in two aspects. First, Shen et al. [16] only considers a single task of interest (face recognition), while our work considers a more practical scenario of having both intended and unintended tasks and evaluates the trade-offs between the two. Second, Shen et al. [16] mainly consider single-step backward compatibility, while we focus on multi-step backward compatibility. We show that our novel multi-step alignment loss achieves very small degradation of unintended task performance even after multiple version updates.

Embedding alignment. Our work builds on cross-lingual embedding alignment methods, where source word embeddings are aligned to target embeddings by learning a linear transformation function [1, 2, 9, 12, 15, 20, 21]. Tagowski et al. [17] applies the embedding alignment technique to the graph domain, where they align a set of node2vec embeddings [4] learned over different snapshots of an evolving graph. However, all these methods assume the embeddings are fixed, which could result in a large alignment error if two sets of pretrained embeddings are very distinct [23]. Unlike these methods, we jointly learn the embeddings along with the backward transformation function, achieving much better alignment performance and better unintended task performance.

7 CONCLUSION

In this paper, we formulated the practical problem of learning backward compatible embeddings. We presented a cost-efficient framework to achieve the embedding backward compatibility even after multiple rounds of updates of the embedding model version. Under the framework, we proposed a promising method, BC-Aligner, that achieves a better trade-off between the intended and unintended task performance compared to prior approaches. There are numerous future directions to investigate. For instance, the trade-off could be further improved by using more expressive backward transformation functions with non-linearity. It is also of interest to assume some partial knowledge about the unintended tasks (*e.g.*, the pre-trained consumer models are accessible to the embedding team) to actually *improve* the unintended task performance without re-training the consumer models. Finally, it is useful to apply our framework to other applications domains, such as those involving sentence and image embeddings.

ACKNOWLEDGEMENTS

We thank Andrew Wang and Rex Ying for their early discussion on the work. Weihua Hu is supported by Funai Overseas Scholarship and Masason Foundation Fellowship. We also gratefully acknowledge the support of DARPA under Nos. HR00112190039 (TAMI), N660011924033 (MCS); ARO under Nos. W911NF-16-1-0342 (MURI), W911NF-16-1-0171 (DURIP); NSF under Nos. OAC-1835598 (CINES), OAC-1934578 (HDR), CCF-1918940 (Expeditions), NIH under No. 3U54HG010426-04S1 (HuBMAP), Stanford Data Science Initiative, Wu Tsai Neurosciences Institute, Amazon, Docomo, Hitachi, Intel, JPMorgan Chase, Juniper Networks, KDDI, NEC, Toshiba, and UnitedHealth Group.

The content is solely the responsibility of the authors and does not necessarily represent the official views of the funding entities.

REFERENCES

[1] Mikel Artetxe, Gorka Labaka, and Eneko Agirre. 2016. Learning principled bilingual mappings of word embeddings while preserving monolingual invariance. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2289–2294.

[2] Goran Glavas, Robert Litschko, Sebastian Ruder, and Ivan Vulic. 2019. How to (properly) evaluate cross-lingual word embeddings: On strong baselines, comparative analyses, and some misconceptions. *arXiv preprint arXiv:1902.00508* (2019).

[3] Palash Goyal, Nitin Kamra, Xinran He, and Yan Liu. 2018. Dyngem: Deep embedding method for dynamic graphs. *arXiv preprint arXiv:1805.11273* (2018).

[4] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 855–864.

[5] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*. 1025–1035.

[6] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *www*. 173–182.

[7] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[8] Diederik P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*.

[9] Alexandre Klementiev, Ivan Titov, and Binod Bhattarai. 2012. Inducing crosslingual distributed representations of words. In *International Conference ON Computational Linguistics (COLING)*. 1459–1474.

[10] Qiang Meng, Chixiang Zhang, Xiaoqiang Xu, and Feng Zhou. 2021. Learning compatible embeddings. In *International Conference on Computer Vision (ICCV)*. 9939–9948.

[11] Tomas Mikolov, Quoc V Le, and Ilya Sutskever. 2013. Exploiting similarities among languages for machine translation. *arXiv preprint arXiv:1309.4168* (2013).

[12] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems (NeurIPS)*. 3111–3119.

[13] Jianmo Ni, Jiacheng Li, and Julian McAuley. 2019. Justifying Recommendations using Distantly-Labeled Reviews and Fine-Grained Aspects. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 188–197.

[14] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. 2012. BPR: Bayesian personalized ranking from implicit feedback. *arXiv preprint arXiv:1205.2618* (2012).

[15] Sebastian Ruder, Ivan Vulić, and Anders Søgaard. 2019. A survey of cross-lingual word embedding models. *Journal of Artificial Intelligence Research* 65 (2019), 569–631.

[16] Yantao Shen, Yuanjun Xiong, Wei Xia, and Stefano Soatto. 2020. Towards backward-compatible representation learning. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 6368–6377.

[17] Kamil Tagowski, Piotr Bielak, and Tomasz Kajdanowicz. 2021. Embedding Alignment Methods in Dynamic Networks. In *International Conference on Computational Science*. Springer, 599–613.

[18] Andrew Z Wang, Rex Ying, Pan Li, Nikhil Rao, Karthik Subbian, and Jure Leskovec. 2021. Bipartite Dynamic Representations for Abuse Detection. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 3638–3648.

[19] Xiang Wang, Xiangnan He, Meng Wang, Fuli Feng, and Tat-Seng Chua. 2019. Neural graph collaborative filtering. In *ACM SIGIR conference on Research and development in Information Retrieval (SIGIR)*. 165–174.

[20] Chao Xing, Dong Wang, Chao Liu, and Yiye Lin. 2015. Normalized word embedding and orthogonal transform for bilingual word translation. In *North American Chapter of the Association for Computational Linguistics (NAACL)*. 1006–1011.

[21] Zijun Yao, Yifan Sun, Weicong Ding, Nikhil Rao, and Hui Xiong. 2018. Dynamic word embeddings for evolving semantic discovery. In *Proceedings of the eleventh acm international conference on web search and data mining*. 673–681.

[22] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*. 974–983.

[23] Mozhi Zhang, Keyulu Xu, Ken-ichi Kawarabayashi, Stefanie Jegelka, and Jordan Boyd-Graber. 2019. Are Girls Neko or Sh\= ojo? Cross-Lingual Alignment of Non-Isomorphic Embeddings with Iterative Normalization. *arXiv preprint arXiv:1906.01622* (2019).

[24] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. Aligraph: A comprehensive graph neural network platform. *arXiv preprint arXiv:1902.08730* (2019).

A ADDITIONAL IMPLEMENTATION DETAILS

A.1 Embedding Models

We evolve the GraphSAGE embedding models as follows. For the Musical Instruments and Video Games, we start with a 2-layer GraphSAGE model with the hidden dimensionality of 256 at t_0. Then, we increase the layer size to 3 at t_2 and increase the hidden dimensionality by 64 at every timestamp. For the Grocery dataset, we use smaller models due to the limited GPU memory; we start with a 1-layer GraphSAGE model with hidden diemensionality of 256 at t_0. Then, we increase the layer size to 2 at t_2 and increase the hidden dimensionality by 64 until t_2. All the models are trained for 500 epochs with Adam [8], with a learning rate of 0.001, and the weight decay of 0.01.

A.2 Unintended Tasks

In designing unintended tasks, we utilize the review rating information associated with each edge, which takes an integer value of between 1 and 5. Below we explain each unintended task, how training is performed on ver-0 embeddings and how predictions are made at each timestamp. We let $V_k^{(\text{user})}$ and $V_k^{(\text{item})}$ denote the set of users and items appearing at least once in E_k.

- **User activity prediction:** Given a user embedding obtained at t_k, we predict whether the user will interact with at least a single item between t_k and t_{k+1}. At training time, we train on users in $V_0^{(\text{user})}$ at t_0, and validate on users in $V_1^{(\text{user})}$ at t_1. At test time $t_k, 2 \leq k$, we make predictions on users in $V_k^{(\text{user})}$.
- **User positive activity prediction:** The task is the same as the above, except that we predict whether a user will have at least one positive interaction with an item (*i.e.*, rating ≥ 4) or not.
- **Item rating avg prediction:** Given an item embedding obtained at t_k, predict the average rating of the item until t_{k+1}, where we only consider items that receive more than 10 reviews until t_k. We binarize the average rating by thresholding it at the median value at t_0. At training time t_0, we train on items in $V_0^{(\text{item})}$ for the average item rating until t_0 and validate on the average item rating until t_1. At test time $t_k, 1 \leq k$, we make predictions on items in $V_k^{(\text{item})}$.
- **Item rating std prediction:** The task is the same as the above, except that we predict the standard deviation of the item ratings. The standard deviation is binarized by thresholding at 1.
- **Edge rating prediction:** Given a pair of user and item embeddings at t_k, predict whether the user gives a positive rating (*i.e.*, ≥ 4) to the item between t_k and t_{k+1}. At training time t_0, we train on edges in E_0 and validate on edges in $E_1 \setminus E_0$. At test time $t_k, 1 \leq k \leq K$, we make predictions on edges in $E_{k+1} \setminus E_k$.

Note that the test prediction for the first two tasks is performed for $t_k, 2 \leq k$, while the test prediction for the last three tasks is performed for $t_k, 1 \leq k$. This is because the first two tasks are predicting future activity of existing users.

All the tasks are binary classification, and we use ROC-AUC on the test set series as the performance metric. All consumer models are 1-hidden-layer MLP models and are trained on ver-0 embeddings generated by M_0. For each task, we tune the hidden embedding dimensionality from $\{128, 256, 512, 1024\}$, the dropout ratio from $\{0, 0.25, 0.5\}$, and performed early-stopping based on performance on the validation set. We report the unintended task performance averaged over 10 random seeds.