Learning to Embed Categorical Features without Embedding Tables for Recommendation

Wang-Cheng Kang, Derek Zhiyuan Cheng, Tiansheng Yao, Xinyang Yi, Ting Chen, Lichan Hong, Ed H. Chi
{wckang,zcheng,tyao,xinyang,iamtingchen,lichan,edchi}@google.com
Google Research, Brain Team

ABSTRACT

Embedding learning of categorical features (e.g. user/item IDs) is at the core of various recommendation models. The standard approach creates an embedding table where each row represents a dedicated embedding vector for every unique feature value. However, this method fails to efficiently handle high-cardinality features and unseen feature values (e.g. new video ID) that are prevalent in real-world recommendation systems. In this paper, we propose an alternative embedding framework Deep Hash Embedding (DHE), replacing embedding tables by a deep embedding network to compute embeddings on the fly. DHE first encodes the feature value to a unique identifier vector with multiple hashing functions and transformations, and then applies a DNN to convert the identifier vector to an embedding. The encoding module is deterministic, non-learnable, and free of storage, while the embedding network is updated during the training time to learn embedding generation. Empirical results show that DHE achieves comparable AUC against the standard one-hot full embedding, with smaller model sizes. Our work sheds light on the design of DNN-based alternative embedding schemes for categorical features without using embedding table lookup.

CCS CONCEPTS

• **Information systems** → **Recommender systems**.

KEYWORDS

Embeddings; Categorical Features; Hashing

ACM Reference Format:
Wang-Cheng Kang, Derek Zhiyuan Cheng, Tiansheng Yao, Xinyang Yi, Ting Chen, Lichan Hong, Ed H. Chi. 2021. Learning to Embed Categorical Features without Embedding Tables for Recommendation. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '21), August 14–18, 2021, Virtual Event, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3447548.3467304

1 INTRODUCTION

Machine learning is highly versatile to model various data types, including continuous features, sparse features, and sequential features. Among these, we focus on improving embedding learning for large-vocabulary categorical features. Specifically, we assume a categorical feature is defined by a vocabulary V, with the feature value is (exactly) one of the elements in V. For example, ID features are typically categorical features where each *feature value* is a unique ID (e.g. video ID). Another example is the "device" feature, and "iPhone 12" is a possible *feature value.*

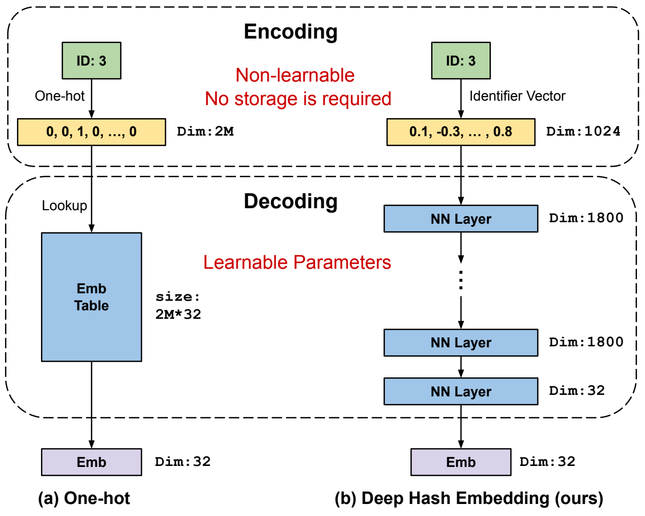


Figure 1: An illustration of one-hot based full embedding and Deep Hash Embedding (DHE) for generating 32-dim embeddings for 2M IDs. The dimension numbers are from our experiments for providing a concrete example. The two models achieve similar AUC while DHE costs 1/4 of the full model size. DHE uses a dense hash encoding to obtain a unique identifier for each feature value, and applies a deep embedding network to generate the feature embedding. DHE doesn't perform any embedding lookup.

Embedding learning has become the core technique for modeling categorical features, and have been adopted in various models, such as Matrix Factorization (MF) [28] and word2vec [24]. The embedding learning technique greatly helps us understand the semantic meaning of feature values (e.g. words). Embeddings have also become the cornerstone of deep models for capturing more complex interactions among feature values (e.g. BERT [7], DeepFM [10]).

Despite the success of embedding learning in various domains like natural language processing (NLP) [24], there are several challenges when applying embedding learning in recommendation:

- **Huge vocabulary size**: Recommender systems usually need to handle high-cardinality categorical features (e.g. billions of video IDs for online video sharing platforms). Moreover, in NLP

KDD '21, August 14–18, 2021, Virtual Event, Singapore
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8332-5/21/08.
https://doi.org/10.1145/3447548.3467304

tasks, the vocabulary size of words is typically small (e.g. the advanced model BERT [7] has a vocabulary of only 30K tokens) due to the common use of sub-word segmentation [29] for reducing the vocabulary size. But it's generally infeasible to apply this approach to the categorical features in recommendation.

- **Dynamic nature of input**: Unlike vocabularies of words that are relatively static, the vocabulary in recommender systems could be highly dynamic. New users and new items enter the system on a daily basis, and stale items are gradually vanishing.
- **Highly-skewed data distribution**: The categorical features in recommendation data usually follow highly skewed power-law distributions. The small number of training examples on infrequent feature values hurts the embedding quality for the tail items significantly.

The one-hot encoding is widely adopted for embedding learning, that maps a feature value to a one-hot vector, and then looks up the embedding vector in an embedding table. However, the one-hot representation often results in a huge embedding table especially for a large-vocab feature, and it also fails to adapt to out-of-vocab feature values. In web-scale neural recommenders, it is not surprising to have most of the parameters spent on the embedding table, while the neural network itself only accounts for a very small portion of parameters [5]. In practice, to better handle new (i.e., out-of-vocab / unseen) feature values and reduce the storage cost, the hashing trick [36] is often adopted, that randomly maps feature values to a smaller number of hashing buckets, though the inevitable embedding collisions generally hurt performance. Essentially, these embedding approaches can be viewed as a 1-layer *wide* neural network (i.e., the embedding table) with *one-hot* encoding.

In this paper, we seek to explore a *deep*, *narrow*, and *collision-free* embedding scheme without using embedding tables. We propose the Deep Hash Embedding (DHE) approach, that uses *dense* encodings and a *deep* embedding network to compute embeddings on the fly. This completely replaces the traditional embedding tables. Specifically, we use multiple hashing and appropriate transformations to generate a unique, deterministic, dense, and real-valued vector as the identifier encoding of the given feature value, and then the deep embedding network transforms the encoding to the final feature embeddings. The feature embeddings are then fed into recommendation models (e.g. MF or deep recommendation models) for end-to-end training. Figure 1 depicts the comparison between the standard one-hot based embedding and DHE. Our main contributions are listed as follows:

- We analyze various embedding methods, including hashing-based approaches for categorical features. Unlike existing methods that heavily rely on one-hot encodings, we encode each feature value to a unique dense encoding vector with multiple hashing, which takes the first step to completely remove the huge embedding tables for large-vocab features.
- With the dense encoding, we replace the commonly used embedding lookup (essentially a shallow and wide network) with *deep* embedding networks, which is more parameter-efficient. We also address the trainability and expressiveness issues to improve the ability of embedding generation.
- We propose Deep Hash Embedding (DHE) based on aforementioned encodings and deep embedding networks. We further improve DHE to better generalize among feature values and to new values, by integrating side features in the encodings.
- We conduct extensive experiments on two benchmark datasets for recommendation tasks with large-vocab categorical features. We compare with state-of-the-art models and analyze the effect of various key components in DHE. The results suggest that DHE is a promising alternative to one-hot full embeddings.

We first discuss various existing one-hot based embedding methods from the perspective of neural networks. Then we introduce DHE's dense hash encodings, deep embedding network, and an extension of using side features for better encodings, before we present our experimental results. Finally, we discuss related work, conclude our paper, and point out promising directions for future work.

2 PRELIMINARY: ONE-HOT BASED EMBEDDING LEARNING

The core idea of embedding learning is to map feature values into a d-dimensional continuous space. These learnable embeddings could be utilized by shallow models like word2vec [24] or MF [28], to directly measure the similarity between two feature values (e.g. large inner products between similar words' embeddings). Moreover, deep models like DeepFM [10] or BERT [7], can model more complex structures via considering interactions among the embeddings.

We define a general framework for describing various existing embedding methods as well as our proposed approach. The embedding function $\mathcal{T} : V \longrightarrow R^d$ maps a feature value (from vocabulary V with size $|V| = n$) to an embedding vector $\mathbf{e} \in \mathbb{R}^d$. Generally, the embedding function can be decomposed into two components: $\mathcal{T} = F \circ E$, where E is an encoding function to represent feature values in some spaces, and F is a decoding function to generate the embedding $\mathbf{v}$. In this section, we introduce full and hashing-based embedding schemes with one-hot encodings. The notation is summarized in Table 9.

2.1 One-hot Full Embedding

This is the most straightforward and commonly used approach to embed categorical features, which assigns each feature value a unique d-dimensional embedding in an embedding table. Specifically, the encoding function E maps a feature value into a unique one-hot vector. In offline settings, this is easy to achieve even if the feature values are non-numeric types like string (e.g. feature values 'Japan' or 'India' for the categorical feature 'Country'), as we can scan and obtain a one-to-one mapping from feature values to $\{1, 2, \ldots, n\}$.

So we assume the feature values are already mapped to $\{1, 2, \ldots, n\}$, then the embedding approach creates an embedding table $\mathbf{W} \in \mathbb{R}^{n\times d}$, and looks up its s-th row $\mathbf{W}_s$ for the feature value s. This is equivalent to the following: (i) we apply the encoding function E to encode feature value s with a one-hot encoding vector: $E(s)=\mathbf{b} \in \{0, 1\}^n$ where $b_s=1$ and $b_j=0$ $(j \neq s)$; (ii) we then apply the decoding function F, a learnable linear transformation $\mathbf{W} \in \mathbb{R}^{n\times d}$ to generate the embedding $\mathbf{e}$, that is, $\mathbf{e} = F(\mathbf{b}) = \mathbf{W}^T\mathbf{b}$. In short, the embedding lookup process can be viewed as a 1-layer neural network (without bias terms) based on the one-hot encoding.

Table 1: Comparison of embedding schemes. The model size of DHE is independent of n or m. DHE is based on dense hash encodings and deep neural networks. DHE can handle out-of-vocab values for online learning, and incorporate side features.

	Full Emb	The Hashing Trick [36]	Bloom Emb [30]	Compositional Emb [32]	Hash Emb [34]	Deep Hash Emb (DHE)
Model Size	$O(nd)$	$O(md)$	$O(md)$	$O(md + \frac{n}{m} d^2)$	$O(nk{+}md)$	$O(kd_{\text{NN}}{+}(h\text{-}1)d^2_{\text{NN}}{+}dd_{\text{NN}})$
#Hash Functions	-	1	2∼4	2	2	∼1000
Encoding Vector	one-hot	one-hot	(multi) one-hot	(multi) one-hot	(multi) one-hot	dense & real-valued
Decoding Function	1-layer NN	1-layer NN	1-layer NN	3-layer NN	1-layer NN	Deep NN
Emb Table Lookup?	✔	✔	✔	✔	✔	✘
Handling OOV Values?	✘	✔	✔	✘	✔	✔
Side Features for Encoding?	✘	✘	✘	✘	✘	✔

2.2 One-hot Hash Embedding

Despite the simplicity and effectiveness of full embeddings, such a scheme has two major issues in large-scale or dynamic settings: (i) the size of the embedding table grows linearly with the vocabulary size, which could cause a huge memory consumption. For example, 100-dimensional embeddings for 1 billion video IDs alone costs near 400 GB of memory; (ii) in online learning settings where new values constantly arise, the full embedding scheme fails to handle unseen (out-of-vocab) feature values. To address the above issues, various hashing-based methods have been proposed (e.g. [30, 34, 36]), and widely used in production-scale systems for handling large-vocab and out-of-vocab categorical features (e.g. *Youtube* [37] and *Twitter* [38]).

The hashing trick [36] is a representative hashing method for reducing the dimension of the one-hot encoding for large vocabularies. The encoding function E still maps a feature value into a one-hot vector, but with a different (typically smaller) cardinality of m: $E(s) = \mathbf{b} \in \{0,1\}^m$ where $s \in V$, $b_{H(s)}$=1 and b_j=0 ($j \neq H(s)$). The hash function H maps feature values (including unseen values) to $\{1, 2, \ldots, m\}$ where m is the hashed vocabulary size. The hash function H seeks to distribute hashing values as uniformly as possible to reduce collision, though it's inevitable when $m < n$. Similarly, the decoding function returns the $H(s)$-th row of the embedding table. In summary, the hashing trick uses hashing to map feature values into m-dim one-hot vectors, and then applies a 1-layer network to generate the embeddings.

Although the hashing trick is able to arbitrarily reduce the cardinality of the original vocabulary V, it suffers from the embedding collision problem. Even in the ideal case (uniformly distributed), each embedding (in the embedding table) is shared by $\lceil n/m \rceil$ feature values on average. This inevitably hurts the model performance, as the model cannot distinguish different feature values due to the same embedding representations. To alleviate this issue, multiple hash functions have been used to generate multiple one-hot encodings: $E(s) = \mathbf{b} = [\mathbf{b}^{(1)}; \mathbf{b}^{(2)}; \ldots; \mathbf{b}^{(k)}] \in \{0,1\}^{m*k}$ where $b^{(i)}_{H^{(i)}(s)}$=1 and $b^{(i)}_j$=0 ($j \neq H^{(i)}(s)$). Here, k hash functions $\{H^{(1)}, H^{(2)}, \ldots, H^{(k)}\}$ are adopted to generate k one-hot encodings $\{\mathbf{b}^{(1)}, \mathbf{b}^{(2)}, \ldots, \mathbf{b}^{(k)}\}$, and the concatenation is used as the encoding.

The core idea is that the concatenated encodings are less likely to be collided. We can lookup k embeddings in k embedding tables (respectively) and aggregate them into the final embedding. A common aggregation approach is 'add' [30, 34, 38], which can be simply expressed as $\mathbf{e} = F(\mathbf{b}) = \mathbf{W}^T\mathbf{b} = \mathbf{W}^T[\mathbf{b}^{(1)}; \mathbf{b}^{(2)}; \ldots; \mathbf{b}^{(k)}]$. That is to say, multiple one-hot vectors are generated with different hash functions, and then the concatenation is fed into a 1-layer neural network without bias terms. It's also common to just create and share a single embedding table [30, 34, 38]. Mathematically, it's equivalent to $\mathbf{v} = \mathbf{W}^T\mathbf{b} = \mathbf{W}^T(\mathbf{b}^{(1)} + \mathbf{b}^{(2)} + \cdots + \mathbf{b}^{(k)})$. Note that existing methods didn't scale to large k and the most commonly used variant is double hashing (k=2) [30, 34, 38].

3 DEEP HASH EMBEDDINGS (DHE)

As introduced in the previous section, both full embeddings and hashing-based embeddings methods are essentially based on one-hot encodings and shallow networks. In this section, we propose Deep Hash Embeddings (DHE), an alternative scheme for embedding learning in large-vocab or dynamic settings. DHE uses real-valued dense encodings and deep neural networks for generating embeddings without any embedding lookup.

Following the embedding framework of encoding and decoding ($\mathcal{T}$=$F \circ E$), we propose several properties for designing good encodings, and then introduce our encoding function E and the decoding function F in DHE, followed by side-feature-enhanced encoding design for enabling generalization.

3.1 Encoding Design

What is a good encoding if we have no prior knowledge about feature values? This is the core question we seek to investigate in this section, and it also leads to our design of the encoding for DHE. We conclude the following properties for designing good encodings:

- **Uniqueness**: The encoding should be unique for each feature value. This is also the target of full embedding and multiple hashing methods. Otherwise, there are feature values that have to share the same encoding. The collided encodings make the subsequent decoding function impossible to distinguish different feature values, which typically hurts model performance.
- **Equal Similarity**: We think only having the uniqueness is not enough. An example is **binary encoding**, which uses the binary representation as the encoding of integers (e.g. IDs): e.g. $H(9) = [1, 0, 0, 1]$. We can see that $H(8) = [1, 0, 0, 0]$ is more similar to $H(9)$, compared with $H(7) = [0, 1, 1, 1]$. We believe this introduces a wrong inductive bias (ID 8 and ID 9 are more similar), which may mislead the subsequent decoding function. The double hashing has a similar issue: the encodings of two feature values that collide in one hash function, are more similar than those of two values that have no collision in both hash functions. As we don't know the semantic similarity among categorical features, we should make any two encodings be equally similar, and not introduce any inductive bias.
- **High dimensionality**: We hope the encodings are easy for the subsequent decoding function to distinguish different feature

Table 2: Encoding comparison regarding the four properties: U: uniqueness; E-S: equal similarity; H-D: high-dimensionality; H-E: high entropy.

Encoding	Length	U	E-S	H-D	H-E
One-hot	n	✔	✔	✔	✘
One-hot Hash	m	✘	✔	✔	✘
Double One-hot Hash	$2m$	✘	✔	✔	✘
Binary	$\lceil \log n \rceil$	✔	✘	✘	✔
Identity	1	✔	✘	✘	✔
DHE (Dense Hash)	k	✔	✔	✔	✔

values. As high-dimensional spaces are often considered to be more separable (e.g. kernel methods), we believe the encoding dimension should be relatively high as well. For example, one-hot encoding has an extremely large dimensionality (n for full embedding and m for hash embedding). Another example is **identity encoding** which directly returns the ID number: e.g. $E(7) = [7]$. Although this gives a unique encoding for each ID, it'd be extremely difficult for the following decoding function to generate embeddings based on the 1-dim encoding.

- **High Shannon Entropy**: The Shannon entropy [31] measures (in the unit of 'bits') the information carried in a dimension. The high entropy requirement is to prevent redundant dimensions from the information theory perspective. For example, an encoding scheme may satisfy the above three properties, but, on some dimensions, the encoding values are the same for all the feature values. So we hope all dimensions are effectively used via maximizing the entropy on each dimension. For example, one-hot encodings have a very low entropy on every dimension, as the encoding on any dimension is 0 for most feature values. Therefore, one-hot encodings need extremely high dimensions (i.e., n) and are highly inefficient.

The formal definitions and analysis of the encoding properties can be found in Appendix, and we summarize the results in Table 2.

3.2 Dense Hash Encoding

After analyzing the properties of various encoding schemes, we found no existing scheme satisfies all the desired properties. Especially we found non-one-hot based encodings like binary and identity encodings are free of embedding tables, but fail to satisfy the *Equal Similarity* and *High dimensionality* properties. Inspired by this, we propose **Dense Hash Encoding**, which seeks to combine the advantages of the above encodings and satisfy all the properties.

Without loss of generality, we assume feature values are integers as we can map string values to integers with string hashing[1]. The proposed encoding function $E : \mathbb{N} \rightarrow \mathbb{R}^k$ uses k universal hash functions to map a feature value to a k-dimensional dense and real-valued encodings. Specifically, we have $E'(s) = [H^{(1)}(s), H^{(2)}(s), \dots, H^{(k)}(s)]$ where $H^{(i)} : \mathbb{N} \rightarrow \{1, 2, \dots, m\}$. Note that m in this case is not related to the embedding table, and we just need to set it to a relatively large number (10^6 in our experiments). A nice property of universal hashing [4] is that the hashed values are evenly distributed (on average) over {1,2,…,m}.

[1]There is basically no collision due to the large output space ($2^{64} \approx 10^{19}$ values for 64-bit integers). An example is *CityHash64*: https://github.com/google/cityhash

However, the integer-based $E'(s)$ encoding is not suitable to be used as the input to neural networks, as the input is typically real-valued and normalized for numeric stability. So we obtain real-valued encodings via appropriate transformations: $E(s) = \text{transform}(E'(s))$. We consider to approximate one of the following commonly used distributions:

- **Uniform Distribution.** We simply normalize the encoding E' to the range of $[-1, 1]$. As the hashing values are evenly distributed (on average) among $\{1, 2, \dots, m\}$, this approximates the uniform distribution $U(-1, 1)$ reasonably well with a large m.
- **Gaussian Distribution.** We first use the above transformation to obtain uniform distributed samples, and then apply the Box–Muller transform [3] which converts the uniformly distributed samples (i.e., $U(-1, 1)$) to the standard normal distribution $\mathcal{N}(0, 1)$. Please refer to Appendix for the implementation.

The choice of the two distributions is partially inspired by Generative Adversarial Networks (GANs) [9] that typically draw random noise from a uniform or Gaussian distribution, and then feed it into neural networks for image generation. Note that the transformation (and the hashing) is deterministic, meaning the encoding (for any feature value) doesn't change over time. Empirically we found the two distributions work similarly well, and thus we choose the uniform distribution by default for simplicity.

Unlike existing hashing methods limited to a few hash functions, we choose a relatively large k for satisfying the *high-dimensionality* property (k=1024 in our experiments, though it's significantly smaller than n). We empirically found our method significantly benefits from larger k while existing hashing methods do not. Moreover, the proposed dense hash encodings also satisfy the other three properties. More analysis can be found in Appendix.

Note that the whole encoding process does not require any storage, as all the computation can be done on the fly. This is also a nice property of using multiple hashing, as we obtain a more distinguishable higher-dimensional encoding without storage overhead. Computation-wise, the time complexity is $O(k)$ and calculation of each hashing is independent and thus amenable for parallelization and hardwares like GPUs and TPUs. As an example, we use the universal hashing for integers [4] as the underlying hashing, and depict the encoding process in Algorithm 2 in Appendix. Other universal hashing (e.g. for strings) could also be adopted for handling different feature types.

3.3 Deep Embedding Network

In DHE, the decoding function $F : \mathbb{R}^k \rightarrow \mathbb{R}^d$ needs to transform a k-dim encoding vector to a d-dim embedding. Obviously, the real-valued encoding is not applicable for embedding lookup. However, the mapping process is very similar to a highly non-linear feature transformation, where the input feature is fixed and non-learnable. Therefore, we use powerful deep neural networks (DNN) to model such a complex transformation, as DNNs are expressive universal function approximators [23]. Moreover, a recent study shows that deeper networks are more parameter-efficient than shallow networks [21], and thus DHE may reduce the model size compared against one-hot full embedding (a 1-layer shallow network).

However, the transformation task is highly challenging, even with DNNs. Essentially, the DNN needs to memorize the information (previously stored in the huge embedding table) in its weights. We hope the hierarchical structures and non-linear activations enable DNNs to express the embedding function more efficiently than the one-hot encodings (i.e., 1-layer wide NN). This is motivated by recent research that shows that deep networks can approximate functions with much fewer parameters compared with wide and shallow networks [21].

Specifically, we use a feedforward network as the decoding function for DHE. We transform the k-dim encoding via h hidden layers with d_{NN} nodes. Then, the outputs layer (with d nodes) transforms the last hidden layer to the d-dim feature value embedding. In practice, d_{NN} is determined by the budget of memory consumption. We can see that the number of parameters in the DNN is $O(k * d_{\text{NN}} + (h-1) * d_{\text{NN}}^2 + d_{\text{NN}} * d)$, which is independent of n or m. This is also the time complexity of DHE. A unique feature of DHE is that it does not use any embedding table lookup, while purely relies on hidden layers to memorize and compute embeddings on the fly.

However, we found that training the deep embedding network is quite challenging (in contrast, one-hot based shallow networks are much easier to train). We observed inferior training and testing performance, presumably due to trainability and expressiveness issues. The expressiveness issue is relatively unique to our task, as NNs are often considered to be highly expressive and easy to overfit. However, we found the embedding network is underfitting instead of overfitting in our case, as the embedding generation task requires highly non-linear transformations from hash encodings to embeddings. We suspect the default ReLU activation ($f(x)$=max$(0, x)$) is not expressive enough, as ReLU networks are piece-wise linear functions [1]. We tried various activation functions[2] and found the recently proposed Mish activation [25] ($f(x)$=$x \cdot \tanh(\ln(1+e^x))$) consistently performs better than ReLU and others. We also found batch normalization (BN) [16] can stabilize the training and achieve better performance. However, regularization methods like dropout are not beneficial, which again verifies the bottleneck of our embedding network is underfitting.

3.4 Side Features Enhanced Encodings for Generalization

An interesting extension for DHE utilizes side features for learning better encodings. This helps to inject structure into our encodings, and enable better generalization among feature values, and to new values. One significant challenge of embedding learning for categorical features is that we can only *memorize* the information for each feature value while we cannot *generalize* among feature values (e.g. generalize from ID 7 to ID 8) or to new values. This is due to the fact that the underlying feature representation does not imply any inherent similarity between different IDs. A typical method for achieving generalization is using side features which provide inherit similarities for generalization (e.g. dense features or bag-of-words features). However, these features are usually used as additional features for the recommendation model, and not used for improving embedding learning of the categorical feature.

[2]we also tried tanh and SIREN [33], and found them on par or inferior to ReLU.

One-hot based full embeddings inherit the property of categorical features, and generate the embeddings independently (i.e., the embeddings for any two IDs are independent). Thus, one-hot based schemes can be viewed as **decentralized** architectures that are good at *memorization* but fail to achieve *generalization*. In contrast, the DHE scheme is a **centralized** solution: any weight change in the embedding network will affect the embeddings for all feature values. We believe the centralized structure provides a potential opportunity for generalization.

As the decoding function of DHE is a neural network, we have a great flexibility to modify the input, like incorporating side features. We propose side feature enhanced encodings for DHE, and hope this will improve the generalization among feature values, and to new values. One straightforward way to enhance the encoding is directly concatenating the generalizable features and the hash encodings. If the dimensionality of the feature vector is too high, we could use locality-sensitive hashing [6] to significantly reduce the cardinality while preserving the tag similarity. The enhanced encoding is then fed into the deep embedding network for embedding generation. We think that the hash encoding provides a unique identifier for *memorization* while the other features enable the *generalization* ability.

3.5 Summary

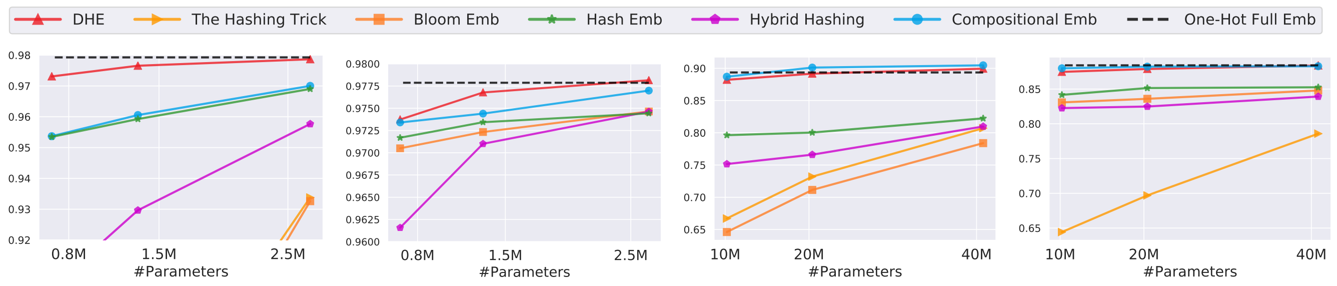
To wrap up, DHE has two major components: 1. dense hash encoding and 2. deep embedding network. The encoding module is fully deterministic and non-learnable, which doesn't require any storage for parameters. The embedding network transforms the identifier vector to the desired embedding. A significant difference of DHE is the absence of the embedding table, as we store all embedding information in the weights of DNN. Hence, all the feature values share the whole embedding network for embedding generation, unlike the hashing trick that shares the same embedding vector for different feature values. This makes DHE free of the embedding collision issue. The bottleneck of DHE is the computation of the embedding network, though it could be accelerated by more powerful hardware and NN acceleration approaches like pruning [8].

Unlike existing embedding methods that explicitly assign each ID the same embedding length, an interesting fact of DHE is that the embedding capacity for each feature value is implicit and could be variable. This could be a desired property for online learning settings (the vocab size constantly grows) and power-law distributions (the embedding network may spend more parameters to memorize popular IDs).

4 EXPERIMENTS

We seek to investigate the following research questions:

- **RQ1:** How does DHE compare against full embedding and hash embedding methods that are based on embedding tables?
- **RQ2:** What's the effect of various encoding schemes for DHE?
- **RQ3:** How does the number of hash functions k affect DHE and other hashing methods?
- **RQ4:** What's the influence of different embedding network architectures, depths, normalization and activation functions?
- **RQ5:** What's the effect of the side feature enhanced encoding?
- **RQ6:** What's the efficiency and GPU acceleration effect of DHE?



(a) ML-20M, GMF (b) ML-20M, MLP (c) Amazon, GMF (d) Amazon, MLP

Figure 2: AUC with different model sizes. The full embedding costs about 5M parameters for *ML-20M*, and 80M for *Amazon*.

4.1 Experimental Setup

4.1.1 Dataset. We use two commonly used public benchmark datasets for evaluating recommendation performance: **Movielens-20M** [11] and **Amazon Books** [26]. The total vocab size (number of users and items) is 165K for *Movielens-20M*, and 2.6M for *Amazon Books*. To reduce the variance, all the results are the average of the outcomes from 5 experiments. The results are on Movielens with 1/4 of the full model size, unless otherwise stated. More details are in Appendix.

4.1.2 Evaluation Metric. We use AUC to evaluate recommendation performance. AUC is a widely used metric in recommendation [13, 28] and CTR prediction [10]. The AUC measures the probability of ranking pairs of a positive item and negative items in the right order, and thus random guesses achieve an AUC of 0.5.

4.1.3 Backbone Recommendation Models. We adopt the **Generalized Matrix Factorization (GMF)** and **Multi-layer Perceptron (MLP)** from [14] as the backbone recommendation models to evaluate the performance of different embedding approaches. We use the two methods to represent both shallow and deep recommendation models. GMF is a shallow model that calculates a weighted sum on the element-wise product of user and item embeddings. With equal weights, GMF is reduced to the classic MF method. Conversely, MLP is a deep model that applies several fully-connected layers on the concatenation of user and item embeddings. Similar deep models have been adopted for recommendation and CTR prediction [10]. The MLP we used has three hidden layers (with [256, 128, 64] nodes), and an output layer to generate the d-dim embedding.

4.2 Baselines

The one-hot **Full Embedding** is a standard way to handle categorical features, which uses a dictionary to map each feature value to a unique one-hot vector. However, to adapt to online learning settings where new items constantly appear and stale items gradually vanish, or to reduce storage cost, hashing-based methods are often adopted. We use the follow hashing-based baselines:

- **The Hashing Trick [36]** A classical approach for handling large-vocab categorical features, which uses a single hash function to map feature value into a smaller vocab. The method often suffers from collision problems.
- **Bloom Embedding [30]** Inspired by bloom filter [2], Bloom Embedding generates a binary encoding with multiple hash functions. Then a linear layer is applied to the encoding to recover the embedding for the given feature value.
- **Hash Embedding (HashEmb) [34]** HashEmb uses multiple (typically two) hash functions and lookups the corresponding embeddings. Then a weighted sum of the embeddings is adopted, where the weights are learned and dedicated for each feature value.
- **Hybrid Hashing [38]** A recently proposed method uses one-hot full embedding for frequent feature values, and uses double hashing for others.
- **Compositional Embedding [32]** A recently proposed method adopts two complementary hashing for avoiding hashing collision. We use the path-based version where the second hashing uses multiple MLPs with one hidden layer of 64 nodes.

We compare our **Deep Hashing Embedding (DHE)** against the above baselines. DHE uses a large number of hash functions (k=1024 in the experiments) to generate a unique identifier for each feature value, followed by a deep embedding network to generate the final embedding. DHE also differs in that it doesn't use any one-hot encoding and embedding table lookup.

4.3 Performance Comparison (RQ1)

We plot the performance with 1/2, 1/4, and 1/8 of the full model size in Figure 2 for the two datasets. We interpret the results via the following comparisons:

- *DHE vs. one-hot full emb:* We observed that DHE effectively approximates Full Embedding's performance. In most cases, DHE achieves similar AUC with only 1/4 of the full model size. This verifies the effectiveness and efficiency (in model sizes) of DHE's hash encoding and deep embedding network, and shows that it's possible to remove one-hot encodings and embedding tables without AUC loss.
- *DHE vs. hashing methods:* We can see that DHE significantly outperforms hashing-based baselines in most cases. This is attributed to its unique hash encoding, which is free of collision and easy for the embedding network to distinguish, and the expressive deep structures for embedding generation. The only exception is Compositional Embedding, which performs slightly better than DHE on the highly sparse *Amazon* dataset. The Hash Trick [36] performs inferior to other methods, especially when the model size is small. This shows that the hash collisions severely hurt the performance.

4.4 Comparison of Encoding Schemes (RQ2)

We investigate the effect of various encodings (not based on one-hot) that are suitable for DHE. We evaluate DHE with encodings mentioned in Section 3.1: identity encoding (1-dim, normalized into [0,1]), binary encoding, random Fourier feature encoding [35], and our proposed hashing encodings with the uniform or Gaussian distribution, and the results are shown in Table 3. We can see that our proposed dense hash encoding with the uniform distribution is the best performer, while the Gaussian distribution variant is the runner-up. The binary encoding performs slightly inferior, and we think it is due to its wrong inductive bias (some IDs have more similar encodings) and the relatively low dimensionality (i.e., $\lceil \log(n) \rceil$). The results also suggest that the random Fourier features [35] are not suitable for our case due to the difference between our problem and signal processing problems where the latter has a meaningful underlying temporal signal. This verifies the effectiveness of the dense hash encoding which satisfies the four properties we proposed.

Table 3: AUC of DHE with different dense encoding schemes.

Encoding	GMF	MLP
Identity Encoding	94.99	95.01
Binary Encoding	97.38	97.36
Random Fourier Encoding [35]	94.98	94.99
Dense Hash Encoding-Gaussian (ours)	97.62	97.66
Dense Hash Encoding-Uniform (ours)	**97.66**	**97.68**

4.5 Scalability Regarding the Number of Hash Functions (RQ3)

Both our method DHE and multiple hashing based methods utilize multiple hash functions to reduce collision. However, as existing hashing methods are limited to a few hash functions (typically 2) [34, 38], we investigate the scalability of DHE and the hashing baselines, in terms of the number of hash functions. Table 4 shows the performance with different k, the number of hash functions. Note that the encoding length of DHE is k, the same as the number of hash functions, while the encoding length for one-hot hashing based methods is $m * k$.

With a small k (e.g. $k \leq 8$), the performance of DHE is inferior to the baselines, mainly because of the shorter encoding length of DHE (i.e., k versus $m * k$ for others). However, when $k \geq 32$, DHE is able to match or beat the performance of alternative methods. When k further increases to more than 100, we can still observe performance gains of DHE, while the one-hot hashing baselines don't benefit from more hash functions. We suspect the reason for the poor utilization of multiple hashing is that each embedding will be shared k times more than single hashing (if sharing embedding tables), and this leads to more collisions. If creating k embedding tables (i.e., not sharing), given the same memory budget, the size for each table will be k times smaller, which again causes the collision issue. However, DHE is free of the collision and embedding sharing problems, and thus can scale to a large k.

Table 4: The effect of the number of hash functions. The results are the AUC of the MLP recommendation model . "-" means the setting is infeasible for the memory budget.

#hash functions (k)	2	4	8	32	128	1024	2048
Bloom Emb [30]	97.21	97.34	97.35	97.43	97.43	97.39	97.28
Hybrid Emb [38]	97.20	97.31	**97.36**	97.42	97.42	97.41	97.30
Hash Emb [38]	**97.29**	**97.40**	-	-	-	-	-
DHE	92.74	95.27	96.77	**97.44**	**97.58**	**97.67**	**97.65**

4.6 Normalization and Activation (RQ4)

Training the deep embedding network is much harder than training embedding lookup based shallow methods. We found there is a trainability issue as well as a unique expressiveness issue. We found that Batch Normalization (BN) [16] greatly stabilizes and accelerates the training, and improves the performance. For the expressiveness issue, we tried various activation functions for replacing ReLU, as ReLU networks are piece-wise linear functions [1] which may not be suitable for the complex transformation in our task. We found the recently proposed Mish [25] activation is superior.

Table 5 shows the results of with and without BN and Mish. We omit results of other activation functions, as we didn't observe performance improvement. We can see that both BN and Mish are critical for enabling deep networks for embedding generation, and improving DHE's performance. Note that for fair comparison, we only use BN and Mish for the embedding network in DHE, while use the same recommendation model (e.g. the MLP model) for all embedding methods.

Table 5: The effect of activations functions and normalization.

Activation functions	GMF	MLP
Without Batch Normalization		
ReLU	97.33	97.47
Mish [25]	97.43	97.50
With Batch Normalization		
ReLU	97.54	97.59
Mish [25] (default)	**97.66**	**97.67**

4.7 The Effect of Depth (RQ4)

The embedding network in DHE takes a hash encoding vector and applies a deep neural network to generate the d-dim output embedding. Specifically, the embedding network consists of several hidden layers with d_{NN} nodes, followed by an output layer with d nodes. We investigate whether deeper embedding networks are more effective against wide & shallow networks, via varying the number of hidden layers while keeping the same number of parameters. Figure 3 shows the results on Movielens. We observed that embedding networks with around five hidden layers are significantly better than wider and shallower networks. This is consistent with our motivation and theoretical results in [21], that deep networks are more parameter-efficient than shallow networks. However, we didn't see further improvement with more hidden layers, presumably because each layer's width is too narrow or due to trainability issues on deep networks.

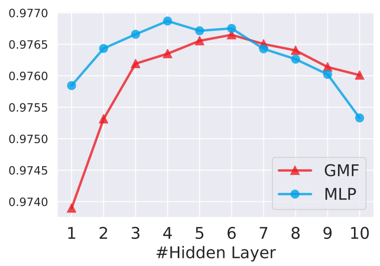


Figure 3: AUC with different depths on Movielens. All the data points are with the same #params. The network depth is #Hidden Layer plus one, where the last layer is for generating the embedding.

4.8 Neural Architectures (RQ4)

The default neural architecture for DHE is equal-width MLP, where each hidden layer has d_{NN} nodes. We also explore various architectures including *Pyramid MLP* (the width of a hidden layer is twice that of the previous layer), *Inverted Pyramid MLP* (opposite to Pyramid MLP), *DenseNet [15]-like MLP* (concatenate all previous layers' output as the input at each layer), and equal-width MLP with residual connections [12]. We adjust the width to make sure all the variants have the same number of parameters. The performance results are shown in Table 6. We can see that the simple equal-width MLP performs the best, and adding residual connections also slightly hurts the performance. We suspect that the low-level representations are not useful in our case, so that the attempts (as in computer vision) utilizing low-level features (like DenseNet [15] or ResNet [12]) didn't achieve better performance. The (inverted) Pyramid MLPs also perform worse than the equal-width MLP, perhaps more tuning on the width multiplier (we used 2 and 0.5) is needed. The results also show it's challenging to design architectures for the embedding generalization tasks, as we didn't find useful prior to guide our designs.

Table 6: AUC with different neural architectures for the embedding network in DHE. All variants use 5 hidden layers with BN and Mish, and have the same number of parameter.

Emb Network	GMF	MLP
Pyramid MLP	97.20	97.49
Inverted Pyramid MLP	97.50	97.58
DenseNet-like MLP	97.53	97.50
Residual Equal-Width MLP	97.61	97.60
Equal-Width MLP (default)	**97.66**	**97.68**

4.9 Side Feature Enhanced Encodings (RQ5)

In previous experiments, we don't use side features in DHE for fair comparison. To investigate the effect of the side feature enhanced encoding, we use the 20 movie *Genres* (e.g. 'Comedy', 'Romance', etc.) in the Movielens dataset, as the side feature. Each movie has zero, one, or multiple genres, and we represent the feature with a 20-dim binary vector. The side features can be used in the encoding function of DHE, and/or directly plugged into the MLP recommendation model (i.e., the MLP takes user, item, and genres vectors as the input).

The results are shown in Table 7. We can see that using side features only in the encoding and only in the MLP have similar performance. This shows DHE's item embeddings effectively capture the *Genres* information, and verifies the generalization ability of item embeddings generated by DHE with enhanced encodings. However, we didn't see further improvement of using the feature in both encoding and MLP. For other embedding methods, adding the feature to the MLP is helpful. However, unlike DHE, they fully rely on IDs and are unable to generate generalizable item embeddings.

Table 7: The effect of side feature enhanced encoding.

Item Embedding	MLP (ID only)	MLP (with genres)
One-hot Full Emb	97.64	97.67
DHE Encoding		
hash encoding (ID) only	97.67	97.72
Genres only	79.17	79.16
hash encoding (ID) +Genres	**97.71**	**97.73**
Hash Emb	97.34	97.42
Hybrid Hashing	97.22	97.31

4.10 Efficiency (RQ6)

One potential drawback of DHE is computation efficiency, as the neural network module in DHE requires a lot of computing resources. However, this is a common problem in all deep learning models, and we hope the efficiency issue could be alleviated by powerful computation hardware (like GPUs and TPUs, optimized for neural networks) that are improving very fast recently. We show the efficiency results in Table 8. With GPUs, DHE is about 9x slower than full embedding, and 4.5x slower than hash embeddings. However, we can see that DHE significantly benefits from GPU acceleration, while full embeddings don't. This is because the embedding lookup process in full embeddings is hard to accelerate by GPUs. The result conveys a promising message that more powerful computation hardware in the future could further accelerate DHE, and gradually close the efficiency gap. Moreover, DHE could also potentially benefit from NN acceleration methods, like pruning [8].

Table 8: Time (in seconds) of embedding generation for 1M queries with a batch size of 100.

	CPU	GPU	GPU Acceleration
Full Emb	3.4	3.4	-1%
Hash Emb [34]	8.4	6.1	-26%
DHE	76.1	27.2	-64%

5 RELATED WORK

Embedding learning has been widely adopted, and representative examples include word2vec [24] and Matrix Factorization (MF) [28]. Other than 'shallow models,' embeddings are also the key component of deep models, like word embeddings for BERT [7]. There are various work on improving the performance or efficiency of embedding learning, via dimensionality search [17], factorization [20], pruning [22], etc. However, these methods are orthogonal to DHE as they are built on top of the standard one-hot encoding and embedding tables.

The hashing trick [36] is a classic method enabling handling large-vocab features and out-of-vocab feature values with one-hot encodings. As only a single hash function is adopted, the collision issue becomes severe when the number of hashing buckets is small. To alleviate this, various improved methods [30, 34, 38] are proposed based on the idea of using multiple hash functions to generate multiple one-hot encodings. Our method also adopts hash functions for generating the encoding. However, our method doesn't rely on one-hot encodings. Also, our approach is able to scale to use a large number of hash functions, while existing methods are limited to use a few (typically two) hash functions.

There is an orthogonal line of work using similarity-preserving hashing for embedding learning. For example, HashRec [18] learns preference-preserving binary representation for efficient retrieval, where a low hamming distance between the embeddings of a user and an item indicates the user may prefer the item. Some other methods utilize locality-sensitive hashing [6] to reduce feature dimensions while maintaining their similarities in the original feature spaces [19, 27]. The main difference is that the hashing we used are designed for reducing collision, while the hashing used in these methods seeks to preserve some kind of similarity.

6 CONCLUSIONS AND FUTURE WORK

In this work, we revisited the widely adopted one-hot based embedding methods, and proposed an alternative embedding framework (DHE), based on dense hash encodings and deep neural networks. DHE does not lookup embeddings, and instead computes embeddings on the fly through its hashing functions and an embedding network. This avoids creating and maintaining huge embedding tables for training and serving. With multiple hash functions, the dense hash encoding generates a unique high-dimensional vector for each feature value without keeping any parameters. Empirical results show that achieves comparable AUC performance against full embeddings, with much smaller model sizes. As a DNN-based embedding framework, DHE could benefit significantly from future deep learning advancement in modeling and hardware, which will further improve DHE's performance and efficiency.

In the future, we plan to investigate several directions for extending and improving DHE: (i) handling multivalent features like bag-of-words; (ii) jointly modeling multiple features with DHE; (iii) hybrid approaches using both embedding tables and neural networks, for balancing efficiency and performance.

REFERENCES

[1] Raman Arora, Amitabh Basu, Poorya Mianjy, and Anirbit Mukherjee. 2018. Understanding Deep Neural Networks with Rectified Linear Units. In *ICLR*.

[2] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.

[3] George EP Box. 1958. A note on the generation of random normal deviates. *Ann. Math. Stat.* 29 (1958), 610–611.

[4] Larry Carter and Mark N. Wegman. 1977. Universal Classes of Hash Functions (Extended Abstract). In *STOC*.

[5] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *RecSys*. ACM, 191–198.

[6] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG*. ACM, 253–262.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*. Association for Computational Linguistics.

[8] Jonathan Frankle and Michael Carbin. 2019. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *ICLR*. OpenReview.net.

[9] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *NIPS*.

[10] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction. In *IJCAI*. ijcai.org.

[11] F. Maxwell Harper and Joseph A. Konstan. 2016. The MovieLens Datasets: History and Context. *ACM Trans. Interact. Intell. Syst.* 5, 4 (2016), 19:1–19:19.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. IEEE Computer Society, 770–778.

[13] Ruining He, Wang-Cheng Kang, and Julian J. McAuley. 2017. Translation-based Recommendation. In *RecSys*. ACM.

[14] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural Collaborative Filtering. In *WWW*. ACM.

[15] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. 2016. Densely Connected Convolutional Networks. *CoRR* abs/1608.06993 (2016). arXiv:1608.06993

[16] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *ICML (JMLR Workshop and Conference Proceedings, Vol. 37)*. JMLR.org.

[17] Manas R. Joglekar, Cong Li, Mei Chen, Taibai Xu, Xiaoming Wang, Jay K. Adams, Pranav Khaitan, Jiahui Liu, and Quoc V. Le. 2020. Neural Input Search for Large Scale Recommendation Models. In *SIGKDD*. ACM.

[18] Wang-Cheng Kang and Julian John McAuley. 2019. Candidate Generation with Binary Codes for Large-Scale Top-N Recommendation. In *CIKM*. ACM.

[19] Karthik Krishnamoorthi, Sujith Ravi, and Zornitsa Kozareva. 2019. PRADO: Projection Attention Networks for Document Classification On-Device. In *EMNLP-IJCNLP*. Association for Computational Linguistics, 5011–5020.

[20] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. In *ICLR*. OpenReview.net.

[21] Shiyu Liang and R. Srikant. 2017. Why Deep Neural Networks for Function Approximation?. In *ICLR*. OpenReview.net.

[22] Siyi Liu, Chen Gao, Yihong Chen, Depeng Jin, and Yong Li. 2020. Learnable Embedding Sizes for Recommender Systems. In *ICLR*.

[23] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. 2017. The Expressive Power of Neural Networks: A View from the Width. In *NIPS*.

[24] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *NIPS*.

[25] Diganta Misra. 2010. Mish: A Self Regularized Non-Monotonic Neural Activation Function. In *BMVC*.

[26] Jianmo Ni, Jiacheng Li, and Julian J. McAuley. 2019. Justifying Recommendations using Distantly-Labeled Reviews and Fine-Grained Aspects. In *EMNLP-IJCNLP*. Association for Computational Linguistics.

[27] Sujith Ravi. 2019. Efficient On-Device Models using Neural Projections. In *ICML*.

[28] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. 2009. BPR: Bayesian Personalized Ranking from Implicit Feedback. In *UAI*.

[29] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *ACL*.

[30] Joan Serrà and Alexandros Karatzoglou. 2017. Getting Deep Recommenders Fit: Bloom Embeddings for Sparse Binary Input/Output Networks. In *RecSys*. ACM.

[31] Claude E. Shannon. 1948. A mathematical theory of communication. *Bell Syst. Tech. J.* 27, 3 (1948), 379–423.

[32] Hao-Jun Michael Shi, Dheevatsa Mudigere, Maxim Naumov, and Jiyan Yang. 2020. Compositional Embeddings Using Complementary Partitions for Memory-Efficient Recommendation Systems. In *SIGKDD*.

[33] Vincent Sitzmann, Julien N. P. Martel, Alexander W. Bergman, David B. Lindell, and Gordon Wetzstein. 2020. Implicit Neural Representations with Periodic Activation Functions. *CoRR* abs/2006.09661 (2020). arXiv:2006.09661

[34] Dan Svenstrup, Jonas Meinertz Hansen, and Ole Winther. 2017. Hash Embeddings for Efficient Word Representations. In *NIPS*.

[35] Matthew Tancik, Pratul P. Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T. Barron, and Ren Ng. 2020. Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains. In *NeurIPS*.

[36] Kilian Q. Weinberger, Anirban Dasgupta, John Langford, Alexander J. Smola, and Josh Attenberg. 2009. Feature hashing for large scale multitask learning. In *ICML*.

[37] Xinyang Yi, Ji Yang, Lichan Hong, Derek Zhiyuan Cheng, Lukasz Heldt, Aditee Kumthekar, Zhe Zhao, Li Wei, and Ed H. Chi. 2019. Sampling-bias-corrected neural modeling for large corpus item recommendations. In *RecSys*. ACM.

[38] Caojin Zhang, Yicun Liu, Yuanpu Xie, Sofia Ira Ktena, Alykhan Tejani, Akshay Gupta, Pranay Kumar Myana, Deepak Dilipkumar, Suvadip Paul, Ikuhiro Ihara, Prasang Upadhyaya, Ferenc Huszar, and Wenzhe Shi. 2020. Model Size Reduction Using Frequency Based Double Hashing for Recommender Systems. In *RecSys*.

Appendix

A. Analysis on Encoding Properties

We formally define and analyze the encoding properties. For demonstration, we use a setting similar to what we used in the experiments: n=10^6, m=10^6, and $k = 1024$.

A.1 Uniqueness

Definition .1 (Uniqueness in encoding). An encoding function E is a unique encoding if $P(E(x)) = P(E(y)) < \epsilon, \forall x, y \in V$, where ϵ is a near-zero constant.

Obviously the *identity encoding*, *one-hot encoding*, and *binary encoding* satisfy the uniqueness property.

For hashing methods, the probability of having collision is $1 - e^{-\frac{n(n-1)}{2m}}$ where m is the total number of hashing buckets (m^2 buckets for double hashing), according to [4]. The probability is 1.0, and 0.39 for *one-hot hashing* and *double one-hot hashing*, respectively. For DHE, the number of possible hashing buckets is $m^k = 10^{6144}$, and the collision rate is extremely small. Thus we can safely assume there is no collision.

A.2 Equal Similarity

Definition .2 (Equal similarity in encoding). An encoding functions E is a equally similar encoding if $\mathbb{E}[\text{Euclidean\_distance}(E(x)-E(y))] = c, \forall x, y \in V$, where c is a non-zero constant.

Obviously the *identity encoding*, and *binary encoding* don't satisfy the property.

For one-hot based hashing methods, the expected Euclidean distance is $k * \frac{m-1}{m}$. For DHE, the expectation is:

$$\begin{aligned}\mathbb{E}[(E(x) - E(y))^2] =& \mathbb{E}[E^2(x) - 2E(x)E(y) + E^2(y)]\\ =& \mathbb{E}[E^2(x)] - 2\mathbb{E}[E(x)]\mathbb{E}[E(y)] + \mathbb{E}[E^2(y)]\\ =& \frac{m(2m+1)(m+1)}{3} - \frac{(m+1)^2}{2}\end{aligned}$$

A.3 High Dimensionality

This is a subjective property, and we generally think larger than 100-dim can be considered as high-dimensional spaces. Following this, the 1-dim *identity encoding* and the $\lceil \log n \rceil$-dim *binary encoding* doesn't satisfy the property.

A.4 High Shannon Entropy

Definition .3 (High Shannon Entropy). An encoding functions E has the high entropy property if for any dimension i, the entropy $H(E(x)_i) = H^*, (x \in V)$, where $H^* = \log o$ is the max entropy for o outcomes (e.g. $H^* = 1$ for binary outcome).

As the zeros and ones are uniformly distributed at each dimension in *binary encoding*, the entropy equals to $H^* = 1$. Similarly, the entropy of *identity encoding* also reaches the maximal entropy $H = -\sum_{i=1}^{n} \frac{1}{n} \log \frac{1}{n} = \log n = H^*$.

For one-hot full embedding, at each dimension, the probability is $\frac{1}{n}$ for having 1, and $\frac{n-1}{n}$ for having 0. So the entropy $H = -\frac{1}{n} \log \frac{1}{n} - \frac{n-1}{n} \log \frac{n-1}{n}$, which quickly converges to zero with a large n. The entropy is significantly less than $H^* = 1$

For one-hot hashing, the probability of having ones is $\frac{n}{m} \cdot \frac{1}{n} = \frac{1}{m}$, and for zeros it's $\frac{m-1}{m}$. Therefore the entropy $H = -\frac{1}{m} \log \frac{1}{m} - \frac{m-1}{m} \log \frac{m-1}{m}$, which is near zero due to the large m. Double one-hot hashing has a similar conclusion.

For DHE, at each dimension, the encodings are uniformly distributed among $[m] = \{1, 2, \ldots, m\}$. Therefor the entropy $H = -\sum_{i=1}^{m} \frac{1}{m} \log \frac{1}{m} = \log m = H^*$, which reaches the maximal entropy.

Table 9: Notation.

Notation	Description
V	set of feature values
$n \in \mathbb{N}$	vocabulary size
$d \in \mathbb{N}$	embedding dimension
$m \in \mathbb{N}$	hashed vocabulary size (usually $m < n$)
$H : V \rightarrow [m]$	hash function mapping feature values to $\{1, 2, \ldots, m\}$
$k \in \mathbb{N}$	number of hash functions, also the encoding length in DHE
$d_{\text{NN}} \in \mathbb{N}$	the width of hidden layers in the embedding network
$h \in \mathbb{N}$	the number of hidden layers in the embedding network

Algorithm 1: Deep Hash Embedding (DHE).

```
Input: a feature value $x \in \mathbb{N}$, encoding length $k$, embedding dim $d$,
       memory budget $B$, network depth $h$
Output: $emb \in \mathbb{R}^d$, a $d$-dim embedding for $x$
/* Calculate the dense hash encoding (parameter-free)      */
$encod \leftarrow$ DenseHashEncoding(x)
/* Define the learnable variables in DNN                   */
$F \leftarrow$ BuildingDNN($k, d, h, B$)
/* Feed the encoding vector into DNN for generating the
   embedding                                               */
$emb \leftarrow F(encod)$
```

Table 10: Dataset statistics

Dataset	#users	#items	total vocab size	#actions	sparsity
MovieLens	138K	27K	165K	20M	99.47%
Amazon	1.9M	0.7M	2.6M	27M	99.99%

B. Experimental Setup

B.1 Dataset Processing

We use two commonly used public benchmark datasets for evaluating recommendation performance:

- **Movielens-20M** is a widely used benchmark for evaluating collaborative filtering algorithms [11]. The dataset includes 20M user ratings on movies[3].
- **Amazon Books** is the largest category in a series of datasets introduced in [26], comprising large corpora of product reviews crawled from *Amazon.com*. We used the latest 5-core version crawled in 2018[4]. The dataset is known for its high sparsity.

The dataset statistics are shown in Table 10. As in [13], we treat all ratings as observed feedback, and sort the feedback according to timestamps. For each user, we withhold their last two actions,

[3]https://grouplens.org/datasets/movielens/20m/
[4]https://nijianmo.github.io/amazon/index.html

Algorithm 2: Dense Hash Encoding in DHE (on-the-fly).

```
Input: a feature value $x \in \mathbb{N}$, encoding length $k$, hash buckets $m$
Output: $encod \in \mathbb{R}^k$, a $k$-dim dense hash encoding for $x$
/* Using a fixed seed to generate the same hash functions at
   each encoding process. The generation can be skipped via
   storing $O(k)$ parameters for hashing.                    */
Set the Random Seed to 0.
for $i \leftarrow 1$ to $k$ do
    /* $a$ and $b$ are randomly chosen integer with $b \neq 0$, $p$ is a
       prime larger than $m$                                  */
    $a \leftarrow$ RandomInteger()
    $b \leftarrow$ RandomNonZeroInteger()
    $p \leftarrow$ RandomPrimerLargerThan$(m)$
    /* Applies universal hashing for integers                 */
    $h[i] \leftarrow ((ax + b) \bmod p) \bmod m$
end
/* Apply a transformation to get the real-valued encoding    */
$encod \leftarrow$ Transform$(h)$
```

Algorithm 3: Encoding Transform.

```
Input: $h \in \{1, 2, \ldots, m\}^k$, $k$ indices of hashing buckets.
Output: $encod \in \mathbb{R}^k$, a $k$-dim dense hash encoding for $x$
for $i \leftarrow 1$ to $k$ do
    $encod'[i] \leftarrow (h[i] - 1)/(m - 1)$          // $encod'[i] \in [0, 1]$
    $encod[i] \leftarrow encod'[i] * 2 - 1$            // $encod[i] \in [-1, 1]$
end
if the distribution is uniform then
    return $encod$                   // uniform distribution $U(-1, 1)$
else
    /* Box-Muller Transform for Gaussian distribution        */
    $i \leftarrow 0$
    while $i < m$ do
        $j \leftarrow i + 1$
        $encod[i] \leftarrow \sqrt{-2 \ln encod'[i]} \cos(2\pi encod'[j])$
        $encod[j] \leftarrow \sqrt{-2 \ln encod'[i]} \sin(2\pi encod'[j])$
        $i \leftarrow i + 2$
    end
    return $encod$                   // Gaussian distribution $\mathcal{N}(0, 1)$
end
```

and put them into the validation set and test set respectively. All the rest are used for model training.

B.2 Implementation Details & Hyper-parameters

We implement all the methods using *TensorFlow*. The embedding dimension d for user and item embeddings is set to 32 for the best Full Emb performance, searched among $\{8, 16, 32, 64\}$, for both datasets. For the recommendation model training, we use the Adam optimizer with a learning rate of 0.001. We apply the embedding schemes on both user and item embeddings. The initialization strategy follows [14]. The model training is accelerated with a single NVIDIA V-100 GPU. To reduce the variance, all the results are the average of the outcomes from 5 experiments.

For HashEmb [34], we use dedicated weights (without collision) for each feature value for better performance. For Hybrid Hashing [38], we use dedicated embeddings for the top 10% of the most frequent feature values, and apply double hashing for the others. By default, we use k=2 hash functions for hashing-based baselines (except for the hashing trick [36] which uses a single hash function), which is suggested by the authors [30, 34, 38]. The given model size budget decides the hashed vocabulary size m for hashing-based methods (e.g. a half of the full model size means m=$n/2$). For compositional embedding [32], we use the quotient-remainder trick to generate two complementary hashing, and adopt the path-based variant with a MLP with one hidden layer of 64 nodes as used in the paper.

For DHE we use the same hyper-parameters for both datasets: k=1024 hash functions to generate the hash encoding vector, followed by a 5-layer feedforward neural network with Batch Normalization [16] and Mish activation function [25]. The width d_{NN} of the network is determined by the given model size. The m in DHE is set to 10^6.

C. Pseudo-code

Algorithm 1 presents the overall process of the proposed Deep Hash Embedding (DHE). Algorithm 2 presents the encoding process of the dense hash encoding. Algorithm 3 presents the transformations for converting the integer vector (after hashing) into real-valued vectors and approximating a uniform or Gaussian distribution. We utilize the evenly distributed property of universal hashing [4] to build the uniform distribution, and adopts the Box-Muller transform[5] to construct a Gaussian distribution from pairs of uniformly distributed samples ($U(0, 1)$) [3].

[5]https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform