High-dimensional Sparse Embeddings for Collaborative Filtering

Jan Van Balen
Bart Goethals
janvanbalen@gmail.com
bart.goethals@uantwerpen.be
University of Antwerp
Antwerpen, Belgium

ABSTRACT

A widely adopted paradigm in the design of recommender systems is to represent users and items as vectors, often referred to as latent factors or embeddings. Embeddings can be obtained using a variety of recommendation models and served in production using a variety of data engineering solutions. Embeddings also facilitate transfer learning, where trained embeddings from one model are reused in another. In contrast, some of the best-performing collaborative filtering models today are high-dimensional linear models that do not rely on factorization, and so they do not produce embeddings [27, 28]. They also require pruning, amounting to a trade-off between the model size and the density of the predicted affinities. This paper argues for the use of high-dimensional, sparse latent factor models, instead. We propose a new recommendation model based on a full-rank factorization of the inverse Gram matrix. The resulting high-dimensional embeddings can be made sparse while still factorizing a dense affinity matrix. We show how the embeddings combine the advantages of latent representations with the performance of high-dimensional linear models.

CCS CONCEPTS

• **Information systems → Retrieval models and ranking**; **Personalization**.

KEYWORDS

collaborative filtering, high-dimensional embeddings, sparse embeddings, cholesky decomposition

ACM Reference Format:
Jan Van Balen and Bart Goethals. 2021. High-dimensional Sparse Embeddings for Collaborative Filtering. In *Proceedings of the Web Conference 2021 (WWW '21), April 19–23, 2021, Ljubljana, Slovenia.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3442381.3450054

1 INTRODUCTION

Wherever humans are faced with a large number of possible actions, subjective preferences, and limited time, recommender systems may help them make decisions. At the heart of many of these systems sit collaborative filtering algorithms. They model preferences as a function of past interactions, and use it to recommend future actions. Many collaborative filtering algorithms rely on latent factors or *embeddings*. These are vectors of numbers representing typically either a *user* of the system or an *item* from its catalog.

Latent factor models were first developed by participants of the Netflix Prize, a machine learning competition on film ratings that ran from 2006 to 2009 [6, 9, 14, 22, 23], and have become extremely popular in the years since the Netflix Prize, becoming the *de facto* industry standard for recommendation. Item embedding models have been developed or deployed by a range of prominent industry players, including Microsoft, Facebook, Spotify, Pinterest and Deezer [1–3, 19, 32]. They have several benefits. Most importantly, they generally work well. They are conceptually simple, and offer a geometric interpretation of users and items as points in high-dimensional space. Compared to neighborhood models, they also have the benefit that two items may be related even if no user ever interacted with both. And lastly, embeddings allow for transfer learning: embeddings learned on one model may be reused in another.

For as long as there have been latent factors, there have also been linear collaborative filtering algorithms *not* relying on embeddings. These models extract an item-item similarity matrix and recommend the items that are most similar to those in the user histories. The item-item matrix may be based on co-counts (how often do two items occur together across all users' history) or learned through optimization, e.g. the SLIM model [21].

Many models of either kind are *linear* and *shallow*. Recent work has shown that, despite the success of deep learning elsewhere, linear and shallow recommenders continue to produce state-of-the-art results, while results for neural recommenders are brittle and difficult to reproduce [5, 23, 28]. In particular, in 2019, Steck proposed EASE^R, a simple linear model that is easy to train and achieves state-of-the-art performance on ranking metrics [28]. Crucially, Steck's is not a factor model, but a variant of SLIM; it is 'full-rank'. Might it be more useful for a model to be 'wide' than 'deep'?

1.1 Prediction Coverage

Many real-world recommenders consist of a two-stage architecture. For a given user or query, a big number of candidate items is first retrieved by a *candidate generation* system. The candidate items are then ranked by a *ranking* system that is trained to assign scores to arbitrary query-candidate pairs, perhaps given some additional context [4, 31]. With some abuse of terminology, we will refer

WWW '21, April 19–23, 2021, Ljubljana, Slovenia

ACM ISBN 978-1-4503-8312-7/21/04.
https://doi.org/10.1145/3442381.3450054

here to the number of items receiving a non-zero score for a given user or query as the *prediction coverage*. Coverage is important for two-stage recommender systems in that candidate generation and ranking systems need to be able to score, respectively, a large or arbitrary number of query-item pairs.

Two-stage architectures expose a downside of current high-dimensional models like SLIM and EASE^R. Having $O(N^2)$ parameters, these models typically need to be pruned to maintain a practical memory footprint. As we will see, this comes at the expensive of prediction coverage. Meanwhile, models based on factorization are able to assign scores to any user-item pair: the user and item embeddings factorize the dense matrix of predicted affinities.[1]

In this paper, we ask whether we can make high-dimensional embedding representations practical. In particular, we propose a simple linear model based on *Cholesky embeddings*: sparse, high-dimensional item embeddings that combine the advantages of embeddings with the beneficial performance of high-rank models like Steck's. It offers great ranking performance, fast training and better prediction coverage—assigning non-zeros scores to over 99% of items—at practical sparsity levels. An is available at https://github.com/jvbalen/cholesky.

2 RELATED WORK

We distinguish three types of latent factor models: user-item models, item-item models and non-linear models. In *user-item models* each user and each item are assigned an embedding, and the recommendation score of an item i for a user j is modeled as the dot product of their embeddings. In *item-item models*, only items have embeddings. To get recommendation scores for given a user, the embeddings of all items in a user's history have to be summed, first, to obtain an "on-the-fly", user embedding. *Non-linear models* tend to follow the same approach, but apply some non-linear transformation to the summed embeddings before scoring items, typically using a neural network.

2.1 Factor Models

User-item factor models algorithms are trained on a dataset of observed user-item interactions, typically represented as a sparse $M \times N$ matrix X, with M and N the number of users and items. The non-zero elements of X may encode explicit feedback, such as ratings, or implicit feedback: whether a user i engaged with an item j. The central idea in these algorithms is to approximate X with matrices $U \in \mathbb{R}^{M \times K}$ and $V \in \mathbb{R}^{N \times K}$.

$$X \approx UV^\intercal \tag{1}$$

U and V are learned through an optimization procedure. Many variants include item and user bias vectors and ℓ_2 regularization [14]. In weighted matrix factorization (WMF), non-uniform weights are applied to the reconstruction error during optimization [9]. To recommend new items to a user, user-item scores can be read directly from the dense reconstruction $Y = UV^\intercal$.

Item-item models, such as NSVD [22], asymmetric SVD [14] and FISM [10], are inspired by neighborhood models, in which recommendations $Y = XS$ are a function of X and a given item-item similarity matrix S. In item-item factor models, however, S is replaced with two $N \times K$ matrices P and Q that are learned during training:

$$X \approx XPQ^\intercal \tag{2}$$

plus again an optional vector b of item biases. In this setup, the model learns two sets of item embeddings, P and Q. The purpose of the latter is the same as V above, while the former is used to compute on-the-fly user factors XP. This generally reduces the number of parameters and generalizes more easily to new users.

2.2 Neural Networks

The most commonly used neural architectures for collaborative filtering are auto-encoders [16, 18, 24, 26, 30]. Auto-encoders take rows of X as inputs, transform these using one or more "hidden" layers and try to reconstruct them on the output side. The limited width of the hidden layers prevents the model from learning the identity function, along with other regularization techniques such as dropout, Gaussian noise or weight decay [18]. At the time of writing, the best results achieved with neural recommenders have come from variational auto-encoders [12, 13, 18]. Auto-encoders as described here are similar to item-item models: most of the model capacity is in the first and last layer, which perform the same role as the factors P and Q in an item-item factor model. The *I-AutoRec* model, for example, has only two layers, of which the first one is linear, and so the only difference with an item-item factor model is in the training objective [24].

A similar strand of neural recommenders is the *item2vec* family. It is inspired by language models, and in particular, *word2vec*. Word2vec learns word embeddings by trying to predict a masked word from its immediate context [20]. Its architecture is that of a simple item-item model (two layers, linear bottleneck). Its objective however is different, and involves *negative sampling*: instead of computing a loss over all possible N predictions, output values are computed only for the true positive items and a random sample of "negatives". The resulting loss is an approximation, but makes the final layer of the model many times more efficient and drastically reduces training time. Item2vec and *prod2vec*, among other models, applied these ideas to recommendation [1, 8].

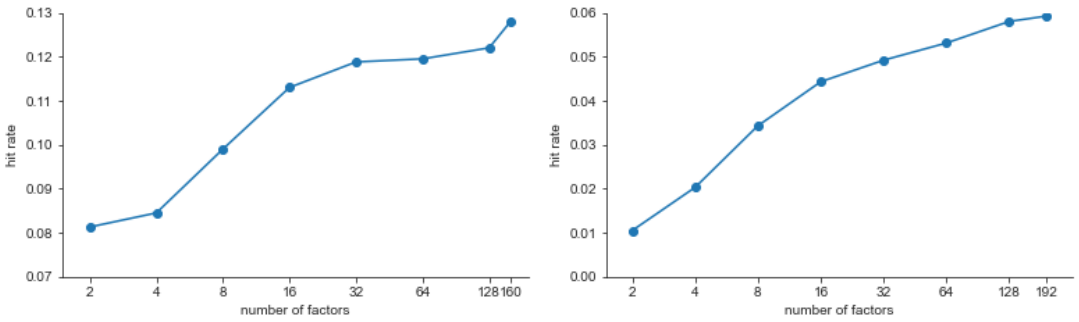
2.3 High-dimensional Linear Models

If item-item models take the S out of neighborhood models and replace it with the product of two learned embedding matrices P and Q, high-dimensional linear models instead replace S with a more general $N \times N$ weight matrix B that doesn't necessarily decompose into K-dimensional factors.

$$X \approx XB \tag{3}$$

The training objective is to minimize the squared error $\|X - XB\|_2^2$. The diagonal of B can be constrained to zero to avoid recovering the trivial solution $B = I$. Nonetheless, this essentially performs a simple but very high-dimensional linear regression. Like the factor models above, the idea first appeared in solutions to the Netflix Prize, e.g. [14, 22]. Since B has many more parameters than P and Q, the *SLIM* model adds sparsity-inducing ℓ_1 and ℓ_1 penalties to the training objective, as well as a hard positivity constraint. Estimation

[1] Note also that, while EASE^R and SLIM are *trained* to predict zeros for unseen items, this is not what we use them for in practice. Even in papers, they are evaluated based on how they rank unseen items in the hold-out set, which requires non-zero scores.



(a) ML100K Dataset **(b) Netflix Dataset**

Figure 1: Adapted from [10]. Hit rate as a function of the embedding dimension K of the FISM model, on two datasets. Similar figures can be found in other publications proposing embedding models [14, 24].

of B is parallelized across the column of B, each sub-problem solving a linear regression with one dependent and $N-1$ independent variables [21].

Further research has shown that the positivity constraint and the ℓ_1 penalty can be dropped, enabling faster training [17, 28]. One model, called EASE^R, has the following closed-form solution:

$$B = I - PD_p^{-1} \tag{4}$$

where P (for *precision* matrix) is the regularized inverse of the Gram or co-count matrix $X^\intercal X$.

$$P = (X^\intercal X + \lambda I)^{-1} \tag{5}$$

D_p is a diagonal matrix containing the diagonal of P and λ a regularization parameter. Computing B is still a memory-intensive task, due to the N^2 parameters, but it is much faster than the N separate regressions required for SLIM.

Experiments with EASE^R have shown it to match or outperform state-of-the-art matrix factorization and auto-encoder models, despite its simple formulation. They also show that the weights B may be pruned down to a much sparser matrix, by setting all elements except those with the highest absolute value to zero. A sparsity of 99%-99.9% could be achieved with less than 1% relative loss of recall [27].

3 FULL-RANK EMBEDDINGS

From Steck's experiments comparing EASE^R to existing factor and auto-encoder models, we hypothesize that the low-rank bottlenecks inherent to existing embedding models are also recommendation quality bottlenecks. This is consistent with results from several works on low-rank factor models: performance goes up as the embedding dimension increases, see e.g. Figure 1, adapted from [10], as well as tables and figures in [10, 24].

Of course, researchers and practitioners have a good reason for choosing a modest number of dimensions: efficiency. Beyond some number of dimensions, additional performance gains may not be worth the linear increase in memory required to store high-dimensional embeddings, or the cost of making predictions. However, this is only true for *dense* vectors.

We propose a combination of high-dimensional factorization and pruning as a way to increase performance without increasing memory footprint. Or alternatively: to reduce the memory footprint of high-dimensional embedding models without having to reduce the embedding dimension K. This ties our work to *embedding compression*. However, while compressed embeddings have made an appearance in recommender systems before, work so far has, to our knowledge, focused only on bloom filters [25] and quantization [11, 30] rather than pruning.

3.1 Cholesky Decomposition

The main idea of this paper is to turn the the embedding dimension K all the way up to $K = N$ and compute *full-rank* embeddings. That is, for a catalog with N items, we use sparse N-dimensional embeddings. We can obtain such embedding efficiently by building on the high-dimensional linear recommender proposed by Steck [28]. We propose to decompose its $N \times N$ weight matrix B into the following form:

$$B = LL^\intercal D_\pi \tag{6}$$

with D_π a diagonal matrix. Using $Y = XB$, we arrive at a formula for the recommendation scores that admits a familiar interpretation:

$$Y = XLL^\intercal D_\pi \tag{7}$$

In other words, our model is an item-item factor model with $P = L$ and $Q = D_\pi L$. Or, alternatively, a model with $P = Q = L$ of which the item scores are multiplied by a vector π of multiplicative item priors.

To find L, we perform *Cholesky decomposition*. The Cholesky decomposition $LL^\intercal$ of a real, symmetric matrix A is an exact factorization of A into a *lower-triangular* matrix L and its transpose. In particular, we decompose $\beta D_p - P$, where β is a scalar hyperparameter and D_p the diagonal matrix from section 2.3. We have:

$$\begin{aligned} LL^\intercal &= \beta D_p - P \\ Y &= XLL^\intercal D_\pi \end{aligned} \tag{8}$$

For $\beta = 1$ and $D_\pi = D_p^{-1}$, this reduces exactly to EASE^R: $Y = XB$. The Cholesky factor L exists when A is a positive-definite matrix. While $D_p - P$ is generally *not* positive-definite, we find that small values of $\beta \geq 2$ are typically enough to make the rows of $\beta D_p - P$ diagonal dominant, allowing it to be decomposed. Note that choosing $\beta > 1$ will cause $LL^\intercal D_\pi$ to deviate from B in the diagonal,

Algorithm 1: Cholesky embeddings in Python

```
# let X be the sparse user-item matrix
# let l2_reg a regularization parameter
# let beta >= 2
G = (X.T @ X).toarray()                    # compute Gram matrix and make dense
diag_indices = numpy.diag_indices_from(G)
G[diag_indices] += l2_reg                  # G + l2_reg * I
P = numpy.linalg.inv(G)                    # invert
A = -P
A[diag_indices] += beta * numpy.diag(P)    # beta * D_p - P
L = scipy.linalg.cholesky(A, lower=True)   # decompose
```

however we note here that in SLIM-like models, the diagonal of B does not affect the recommendation scores of *new* items (those not already part of a user's history), which is what we're interested in.

3.2 Singular Value Decomposition

As an alternative to Cholesky factorization, we may also consider the more commonly-used singular value decomposition (SVD), where a matrix is decomposed into left-singular vectors U, right-singular vectors V, and a diagonal matrix Σ of singular values. Since the matrix we decompose can be made positive-definite and symmetric, the singular values are positive and $U = V$. We can therefore choose $H = U\Sigma^{\frac{1}{2}} = V\Sigma^{\frac{1}{2}}$ and rewrite as:

$$\begin{aligned} U\Sigma V^\intercal &= \beta D_p - P \\ HH^\intercal &= \beta D_p - P \\ Y &= XHH^\intercal D_\pi \end{aligned} \tag{9}$$

In practice, we find full-rank singular value decomposition to be unstable when using commonly available python implementations; for typical $\beta D_p - P$, none converged. However, we can exploit some additional knowledge of about the factors and use another algorithm that, for symmetric and positive-definite matrices, also yields the SVD: eigen-decomposition.

$$Q\Lambda Q^\intercal = \beta D_p - P \tag{10}$$

Using $H = Q\Lambda^{\frac{1}{2}}$ we then obtain the embeddings H.

3.3 Complexity

The time complexity of both eigen- and Cholesky decomposition is $O(N^3)$, the same as that of computing the inverse Gram matrix. This inversion is also required to compute B. Hence, our method for computing L and H has the same complexity as training EASE^R. The train-time memory footprint is also the same: $O(N^2)$. In practice, training is efficient in that involves only a single pass over the data, but, like EASE^R, its complexity can be prohibitive for large-catalog recommendation. Preliminary experiments indicate that it is possible to compute sparse approximate Cholesky embeddings in $O(kN)$ time (k the number of non-zeros per item), using an procedure similar to the one in [29]. At prediction time, the complexity is greatly reduced by pruning.

Optimized matrix compositions are available in most standard linear algebra packages, including Numpy and Scipy.[2] CHOLMOD, part of SuiteSparse[3] also provides a fast Cholesky implementation but requires pruning P down to a sparse matrix. Following the Python code example in [28], Algorithm 1 shows an example implementation of a function that computes L from X using Scipy, without pruning.

3.4 Adaptive Sparsity

The Cholesky factor L is lower-triangular: all elements above the diagonal are zero. This can be exploited to induce a kind of adaptive sparsity. Note first that we are free to choose the order of the rows of L. It depends on the order of the columns of X which is essentially an arbitrary choice made ahead of training.

For example, we may choose to rank items by their popularity rank r, popular items first. The resulting embeddings (that is, the rows of L) will have r non-zeros in their embeddings. We then independently prune each embedding down to at most k non-zeros. This procedure will ensure that the first k items are not affected by pruning (they can keep all $r < k$ of their parameters). All other items get to keep k of their r parameters, constituting an "effective" density of k/r.

A similar adaptive compression strategy was previously shown to be useful in another, quantization-based (as opposed to pruning-based) approach to embedding compression [11]. Preliminary experiments confirm that here, too, ordering items by number of interactions or number of non-zero co-counts gives better results than ordering randomly. In the next section, we will look at the performance of the proposed embeddings under varying levels of sparsity.

3.5 Relationship with Low-Rank Embeddings

Having defined our embeddings as a factorization of the EASE^R weights matrix, we can interpret existing linear, low-rank embedding models as similar factorizations. The most salient difference is that existing factorization models effectively drop dimensions from the latent representations, instead of making them sparse. Seeing that sparse models are currently gaining in popularity, we hope to see our pruning approach become a starting point for further research, similar to the way the Netflix Prize-era models have inspired several deep and non-linear alternatives since they were first proposed.

[2] See http://www.numpy.org/ and http://www.scipy.org/

[3] See http:www.suitesparse.com/

Table 1: Dimensions and sparsity of the used datasets, after filtering.

dataset	#users	#items	#interactions	density
MovieLens-20M	136677	20108	9990030	0.0036
Netflix dataset	463435	17769	56880037	0.0069
Million Song Dataset	571355	41140	33633450	0.0014

Table 2: Performance on our test split of the MovieLens-20M data for SSVD and CHOL, after pruning embeddings to k non-zeros per item.

		NDCG@100	
k	density	SSVD	CHOL
N	1.0	0.416	0.416
6000	0.3	0.415	0.416
2000	0.1	0.408	0.417
600	0.03	0.360	0.417
200	0.01	0.224	0.416

4 EXPERIMENTS

We now evaluate the recommendation performance of the proposed sparse SVD (*SSVD*) and Cholesky (*CHOL*) embedding models. We also show results for an implementation of EASE^R, and four low-rank models: a user-item weighted matrix factorization model [9], and three auto-encoders [13, 18]. Except for one, all models were trained and evaluated on three commonly-used recommendation datasets: MovieLens-20M, the Netflix prize data and the Million Song Dataset. Table 1 gives an overview of the dimensions and sparsity of each dataset's user-item matrix. We measure so-called 'strong generalization', using the same evaluation set-up as [18, 28]: we rank held-out items for unseen users and compute the *NDCG@100* ranking metric, which assigns a higher weight to positives appearing near the top of the results list, as well as *recall@50*, which reports the total fraction of true positives returned within the top 50—see the aforementioned publications for a more extensive motivation.

4.1 Singular Value Decomposition

A first set of experiments, on the smallest of the three datasets, already shows that the embeddings based on Cholesky are much more amenable to pruning those based on SVD. Table 2 shows the test set performance on MovieLens-20M of SSVD and CHOL, each of them pruned down to $k \in \{N, 6000, 2000, 600, 200\}$ non-zeros per item. The ℓ_2 regularization parameter was held constant at $\lambda = 100$. Performance of the SSVD embeddings drops from NDCG@100 = 0.416 to 0.224 while the CHOL embeddings' performance is not meaningfully affected. In the next sections, we will no longer consider the SSVD embedding model and focus on CHOL.

4.2 Cholesky Decomposition

Experiments with regularized item-item models consistently show how performance increases with embedding size, see e.g. figures and tables in [10, 14, 24]. In the limit $K = N$, the high-dimensional EASE^R model outperforms, on big datasets, weighted matrix factorization and several non-linear auto-encoders. On the challenging Million Song Dataset, EASE^R is, at this time and to our knowledge, the best-performing "pure" collaborative filtering model.[4] For this reason, and because Cholesky decomposition is exact, we know that a *dense* Cholesky embeddings model would reach the same state-of-the-art performance. Our second experiment confirms this. The top half of Table 3 shows the performance of a dense EASE model, and a fully dense CHOL embedding model. As expected, their performance is exactly the same.

Dense high-dimensional embeddings however are impractical. Therefore, we focus our experiments on measuring the effect of *sparsity* on the quality and prediction coverage of the embeddings. In particular, we compare CHOL to EASE^R and hypothesize that CHOL (i) achieves similar or better performance (NDCG) compared to EASE^R, (ii) yields more recommendations with non-zero scores.

To address the first hypothesis, Figure 2 illustrates the recommendation performance of both EASE^R and CHOL on a validation split (10K users) of the Million Song Dataset. It shows NDCG@100 for various densities and regularization strengths. The EASE^R models were pruned based on a global magnitude threshold as in [27]. Cholesky embeddings were pruned down to a fixed number of non-zeros k per embedding (k = 100–200).

The figure makes clear that using pruned Cholesky embeddings rather than pruned EASE^R weights does not impact performance negatively—it even comes with a small but consistent improvement.

Figures 3a and 3b show, for three datasets, the total train time and prediction coverage (average number of non-zero item scores per user). The two models were pruned as above, to an equivalent density of $200/N$ or $k = 200$. While training CHOL takes a little longer than training EASE^R, the number of non-zero scores is a lot higher: 99–99.9% vs 6-12%, confirming our second hypothesis, and illustrating an important benefit of using embeddings.[5]

An overview of all test set metrics is shown in Table 3. A line separates models with $O(N^2)$ parameters from those with $O(N)$. The lower section of the table contains results for pruned versions of EASE^R and CHOL, again to an equivalent density of $200/N$ or $k = 200$. The last three models are'low-rank' models, copied from [18] (same datasets, preprocessing and split size).[6] All three have a 200-dimensional bottleneck. The numbers are consistent with Steck's: while the low-rank autoencoders perform best on MovieLens-20M, EASE and CHOL do much better on the larger Million Song Dataset [28]. And they are consistent with Figure 2: CHOL tends to perform slightly better than EASE^R.

4.3 Benefits and Limitations

To highlight the benefits of our approach, we return to our discussion of the conditions that determine when embeddings are useful and when they are not. Embeddings will generally be useful if we would like to give scores to arbitrary user-item pairs without

[4] I.e., not content-based or aided by side information.

[5] We note here that for completeness, experiments should be done to determine the quality of the scores beyond the top-50 and top-100 cut-offs used in our evaluation set-up, and perhaps after which point beyond they become too noisy.

[6] We have reproduced a subset of these numbers for our own test splits and find the difference to be small (≈ 0.002).

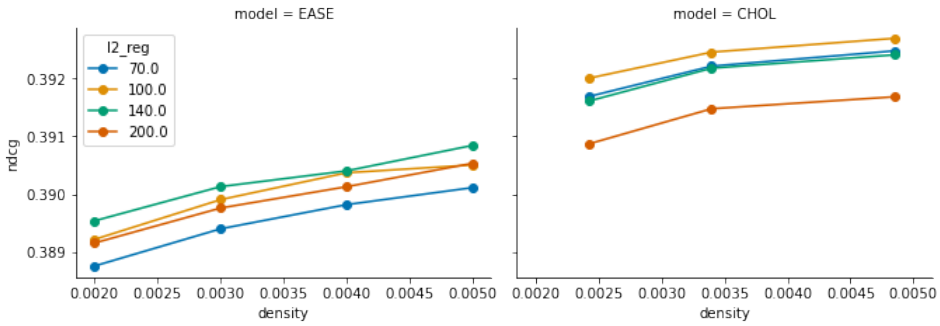


Figure 2: Recommendation performance of the EASE^R and CHOL models on the Million Song Dataset as a function of density and the ℓ_2 regularization strength, shown here as l2_reg.

Table 3: Test set metrics for dense and pruned versions of the CHOL and EASE models, and a set of low-rank models. A line separates models with $O(N^2)$ parameters from those with $O(N)$. Results in bold indicate best performance among this second group.

			MovieLens-20M		Netflix		Million Song Dataset	
model	k	K	recall@50	NDCG@100	recall@50	NDCG@100	recall@50	NDCG@100
EASE	N	N	0.521	0.420	0.445	0.395	0.430	0.390
CHOL	N	N	0.521	0.420	0.445	0.395	0.430	0.390
EASE	200	N	0.516	0.417	0.440	**0.390**	0.427	0.389
CHOL	200	N	0.516	0.417	0.440	**0.390**	**0.429**	**0.390**
H+Vamp**		1000	0.551	0.445	0.463	0.409	-	-
Mult-VAE*		200	**0.537**	**0.426**	**0.444**	0.386	0.364	0.316
Mult-DAE*		200	0.524	0.419	0.438	0.380	0.363	0.313
WMF*		200	0.498	0.386	0.404	0.351	0.312	0.257

Note. k is the number of non-zeros per item, K is the rank (smallest latent dimension). WMF is a weighted matrix factorization model, VAE and DAE are a variational en denoising autoencoder and H+Vamp is a variational auto-encoder with a more flexible prior [9, 13, 18]. Results for $*$ and $**$ are taken from [18] and [13], respectively.

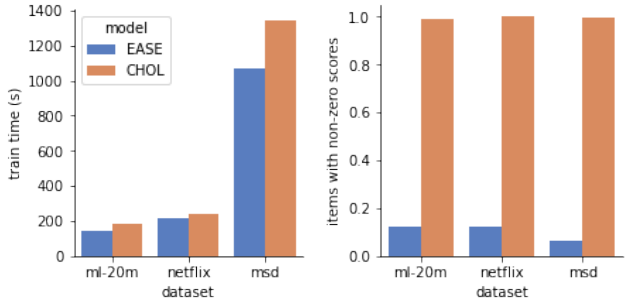


Figure 3: (a) total train time for the two models (both pruned to 0.05% sparsity). (b) average number of items with a non-zero score.

having to store a dense item-item matrix, as is the case in a typical ranking system. In contrast, for applications that only ever require the top k items per user, a pruned EASE^R or SLIM may work just fine. However, if k is large, as is the case for a typical real-world candidate generation system, embeddings become increasingly useful again. The embeddings we propose allow us to recover prediction coverage while outperforming existing embeddings systems.

We also believe that, for existing production frameworks that are already optimized for serving embeddings and re-using them in various contexts, the embeddings introduced in this paper may be relatively easy to adopt—especially as machine learning frameworks and hardware continue to extend support for sparse matrix and tensor computations [7, 15].

The main limitations of our approach are the memory requirement we inherit from EASE^R, and prediction time. Even though the model trains fast, predicting item recommendation scores with Cholesky embeddings was found to take longer than with other models: up to 500ms for a batch of 100 Million Song Dataset users.[7] This is 3–10 times longer than low-rank models. In an offline setting, this may not be a problem, but online applications may suffer. The problem is that XL can be >15% dense even if X and L are sparse. This makes the downstream matrix multiplications more expensive. Pruning these user embeddings XL may help, but more

[7] wall time on a 2019 6-core Macbook Pro

experiments need to be done to analyze its effect on prediction time and ranking performance. Alternatively, indexing and approximate nearest neighbors methods may be used to speed up prediction.

Finally, we want to underline the generality of the proposed factorization: embeddings are not just used in recommender systems, but also in NLP and in machine learning models operating on tabular data. Perhaps some of those application could also benefit from a shift to sparse rather than low-dimensional embeddings.

Of course, while embeddings generally facilitate transfer learning, sparse and high-dimensional embeddings are not as easy to integrate as their low-rank counterparts. Here, we note that our Cholesky model can be viewed as a sparse, two-layer neural network of which not just the weights, but also the sparsity structure can be re-used. The sparsity structure of the Cholesky embeddings also defines a directed acyclic graph, and we expect this kind of transfer learning to receive further attention as research interest in sparse neural networks and graph neural networks increases. We eventually hope to see the ideas in this paper contribute to the advancement of sparse and graph neural networks in recommendation research.

5 CONCLUSION

We have proposed a simple and efficient way to learn two types of sparse high-dimensional embeddings. The proposed embeddings based on Cholesky factorization show great ranking performance while retaining the benefits of embeddings. In particular, they enable scoring of arbitrary items at practical sparsity levels. We also showed that SVD-based embeddings did not reach the same performance, and discussed the CHOL model's benefits and limitations. In future work, we hope to investigate more efficient training procedures, and hope to use Cholesky embeddings in a transfer learning application that exploits item interactions and side-information.

6 ACKNOWLEDGEMENTS

This research received funding from the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen" program.

REFERENCES

[1] Oren Barkan and Noam Koenigstein. 2016. Item2Vec: Neural item embedding for collaborative filtering. In *2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP)*.

[2] Eric Bernhardsson. 2017. Quora Answer. https://www.quora.com/How-did-Spotify-get-so-good-at-machine-learning-Was-machine-learning-important-from-the-start-or-did-they-catch-up-over-time/answer/Erik-Bernhardsson. Accessed: 2020-05-26.

[3] Hugo Caselles-Dupré, Florian Lesaint, and Jimena Royo-Letelier. 2018. Word2vec Applied to Recommendation: Hyperparameters Matter. In *Proc. of the 12th ACM Conference on Recommender Systems* (Vancouver, British Columbia, Canada) *(RecSys '18)*. Association for Computing Machinery, New York, NY, USA, 352–356. https://doi.org/10.1145/3240323.3240377

[4] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*. 191–198.

[5] Maurizio Ferrari Dacrema, Paolo Cremonesi, and Dietmar Jannach. 2019. Are We Really Making Much Progress? A Worrying Analysis of Recent Neural Recommendation Approaches. In *Proc. of the 13th ACM Conference on Recommender Systems* (Copenhagen, Denmark) *(RecSys '19)*. ACM, 101–109.

[6] Simon Funk. 2006. Netflix Update: Try This at Home. https://sifter.org/~simon/journal/20061211.html. Accessed: 2020-05-26.

[7] Scott Gray, Alec Radford, and Durk Kingma. 2017. Block-Sparse GPU Kernels. https://openai.com/blog/block-sparse-gpu-kernels/. Accessed: 2020-10-19.

[8] Mihajlo Grbovic, Vladan Radosavljevic, Nemanja Djuric, Narayan Bhamidipati, Jaikit Savla, Varun Bhagwan, and Doug Sharp. 2015. E-commerce in your inbox: Product recommendations at scale. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. 1809–1818.

[9] Yifan Hu, Yehuda Koren, and Chris Volinsky. 2008. Collaborative Filtering for Implicit Feedback Datasets. In *Proceedings of the 8th IEEE International Conference on Data Mining (ICDM '08)*. 263–272.

[10] Santosh Kabbur, Xia Ning, and George Karypis. 2013. Fism: factored item similarity models for top-n recommender systems. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 659–667.

[11] Wang-Cheng Kang, Derek Zhiyuan Cheng, Ting Chen, Xinyang Yi, Dong Lin, Lichan Hong, and Ed H. Chi. 2020. Learning Multi-Granular Quantized Embeddings for Large-Vocab Categorical Features in Recommender Systems. In *Companion Proceedings of the Web Conference 2020* (Taipei, Taiwan) *(WWW '20)*. Association for Computing Machinery, New York, NY, USA, 562–566. https://doi.org/10.1145/3366424.3383416

[12] Farhan Khawar, Leonard Poon, and Nevin L. Zhang. 2020. Learning the Structure of Auto-Encoding Recommenders. *Proceedings of The Web Conference 2020* (Apr 2020). https://doi.org/10.1145/3366423.3380135

[13] Daeryong Kim and Bongwon Suh. 2019. Enhancing VAEs for collaborative filtering. *Proceedings of the 13th ACM Conference on Recommender Systems* (Sep 2019). https://doi.org/10.1145/3298689.3347015

[14] Yehuda Koren. 2008. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 426–434.

[15] Ronny Krashinsky, Giroux. Olivier, Stephen Jones, Nick Stam, and Sridhar Ramaswamy. 2020. NVIDIA Ampere Architecture in Depth. https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/. Accessed: 2020-10-19.

[16] Oleksii Kuchaiev and Boris Ginsburg. 2017. Training Deep Autoencoders for Collaborative Filtering. *arXiv preprint arXiv:1708.01715* (2017).

[17] Mark Levy and Kris Jack. 2013. Efficient top-N recommendation by linear regression. In *RecSys Large Scale Recommender Systems Workshop*.

[18] Dawen Liang, Rahul G. Krishnan, Matthew D. Hoffman, and Tony Jebara. 2018. Variational autoencoders for collaborative filtering. In *Proc. of the 2018 World Wide Web Conference (WWW '18)*. International World Wide Web Conferences Steering Committee, ACM, 689–698.

[19] Kevin Ma. 2017. Applying deep learning to Related Pins. https://medium.com/the-graph/applying-deep-learning-to-related-pins-a6fee3c92f5e. Accessed: 2020-05-29.

[20] 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*. 3111–3119.

[21] X. Ning and G. Karypis. 2011. SLIM: Sparse Linear Methods for Top-N Recommender Systems. In *Proc. of the 2011 IEEE 11th International Conference on Data Mining (ICDM '11)*. IEEE Computer Society, 497–506.

[22] Arkadiusz Paterek. 2007. Improving regularized singular value decomposition for collaborative filtering. In *Proceedings of KDD Cup and Workshop*, Vol. 2007. 5–8.

[23] Steffen Rendle, Li Zhang, and Yehuda Koren. 2019. On the Difficulty of Evaluating Baselines: A Study on Recommender Systems. arXiv:1905.01395 [cs.IR]

[24] Suvash Sedhain, Aditya Krishna Menon, Scott Sanner, and Lexing Xie. 2015. AutoRec: Autoencoders Meet Collaborative Filtering. In *Proc. of the 24th International Conference on World Wide Web* (Florence, Italy). 111–112.

[25] Joan Serrà and Alexandros Karatzoglou. 2017. Getting deep recommenders fit: Bloom embeddings for sparse binary input/output networks. In *Proceedings of the Eleventh ACM Conference on Recommender Systems*. 279–287.

[26] I. Shenbin, A. Alekseev, E. Tutubalina, V. Malykh, and S. I. Nikolenko. 2020. RecVAE: A New Variational Autoencoder for Top-N Recommendations with Implicit Feedback. In *Proc. of the 13th International Conference on Web Search and Data Mining* (Houston, TX, USA) *(WSDM '20)*. 528–536.

[27] Harald Steck. 2019. Collaborative Filtering via High-Dimensional Regression. *arXiv preprint* abs/1904.13033 (2019). arXiv:1904.13033

[28] Harald Steck. 2019. Embarrassingly Shallow Autoencoders for Sparse Data. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. 3251–3257.

[29] Harald Steck. 2019. Markov Random Fields for Collaborative Filtering. In *Advances in Neural Information Processing Systems 32*. 5473–5484.

[30] Jan Van Balen and Mark Levy. 2019. PQ-VAE: Efficient Recommendation Using Quantized Embeddings. In *Proc. of the 13th ACM Conference on Recommender Systems, Late Breaking Results* (Copenhagen, Denmark) *(RecSys '19)*.

[31] Maksims Volkovs, Himanshu Rai, Zhaoyue Cheng, Ga Wu, Yichao Lu, and Scott Sanner. 2018. Two-Stage Model for Automatic Playlist Continuation at Scale. In *Proceedings of the ACM Recommender Systems Challenge 2018* (Vancouver, BC, Canada) *(RecSys Challenge '18)*. Association for Computing Machinery, New York, NY, USA, Article 9, 6 pages. https://doi.org/10.1145/3267471.3267480

[32] Ledell Yu Wu, Adam Fisch, Sumit Chopra, Keith Adams, Antoine Bordes, and Jason Weston. 2018. Starspace: Embed all the things!. In *Thirty-Second AAAI Conference on Artificial Intelligence*.