Learning Elastic Embeddings for Customizing On-Device Recommenders

Tong Chen
The University of Queensland
tong.chen@uq.edu.au

Hongzhi Yin*
The University of Queensland
h.yin1@uq.edu.au

Yujia Zheng
Carnegie Mellon University
yjzheng19@gmail.com

Zi Huang
The University of Queensland
huang@itee.uq.edu.au

Yang Wang
Hefei University of Technology
yangwang@hfut.edu.cn

Meng Wang
Hefei University of Technology
eric.mengwang@gmail.com

ABSTRACT

In today's context, deploying data-driven services like recommendation on edge devices instead of cloud servers becomes increasingly attractive due to privacy and network latency concerns. A common practice in building compact on-device recommender systems is to compress their embeddings which are normally the cause of excessive parameterization. However, despite the vast variety of devices and their associated memory constraints, existing memory-efficient recommender systems are only specialized for a fixed memory budget in every design and training life cycle, where a new model has to be retrained to obtain the optimal performance while adapting to a smaller/larger memory budget. In this paper, we present a novel lightweight recommendation paradigm that allows a well-trained recommender to be customized for arbitrary device-specific memory constraints without retraining. The core idea is to compose elastic embeddings for each item, where an elastic embedding is the concatenation of a set of embedding blocks that are carefully chosen by an automated search function. Correspondingly, we propose an innovative approach, namely recommendation with universally learned elastic embeddings (RULE). To ensure the expressiveness of all candidate embedding blocks, RULE enforces a diversity-driven regularization when learning different embedding blocks. Then, a performance estimator-based evolutionary search function is designed, allowing for efficient specialization of elastic embeddings under any memory constraint for on-device recommendation. Extensive experiments on real-world datasets reveal the superior performance of RULE under tight memory budgets.

CCS CONCEPTS

• **Information systems → Recommender systems**.

KEYWORDS

Lightweight Recommendation; Elastic Embeddings

*Corresponding author; contributing equally with the first author.

ACM Reference Format:
Tong Chen, Hongzhi Yin, Yujia Zheng, Zi Huang, Yang Wang, and Meng Wang. 2021. Learning Elastic Embeddings for Customizing On-Device Recommenders. 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, 10 pages. https://doi.org/10.1145/3447548.3467220

1 INTRODUCTION

Recommender systems are now an indispensable component in online services, providing users with tailored experience based on their preferences. With the fast pace of digitalization and hardware revolution, there has been a recent surge of moving data analytics from cloud servers to edge devices [28] to ensure timeliness and privacy. As recommendation services commonly require access to users' personal data, on-device recommendation appears to be an ongoing trend, which facilitates new applications like mobile recommendation [34] and federated recommendation [22].

In this regard, the study on memory-efficient recommender systems [11, 16, 27, 34] emerges, where the resulted models are lightly parameterized and can be deployed on resource-constrained edge devices. Notably, the majority of those work is around latent factor models (e.g., matrix factorization), as they exhibit dominant recommendation accuracy and are less dependent on auxiliary side information. In a general sense, latent factor-based recommenders map each user and item into vector representations (i.e., embeddings), then estimate the pairwise user-item affinity via similarity metrics (e.g., vector dot product) or carefully designed deep neural networks [4, 7, 38]. Due to the sheer volume of categorical features[1] such as user/item IDs and product types, the embeddings in latent factor models are the main source of memory consumption [34] rather than other parameters (i.e., weights and biases). Hence, most efforts are made to compress the embeddings [16, 20, 27] to reduce the size of a recommendation model.

Early memory-efficient recommenders employ discrete hashing to convert real-valued embeddings into compact binary codes [37, 44], with which user-item similarities are preserved in the Hamming space. However, a widely acknowledged fact [18, 37] is that, the expressiveness of the resulted binary codes is greatly impaired due to inevitable information loss in this quantization process, leading to constantly inferior recommendation performance even compared with the plain matrix factorization. In light of this, alternatives that generate informative real-valued embeddings with a minimal amount of parameters are recently developed, where two

[1]We will mainly refer to users and items in our paper for simplicity.

major branches are recommenders based on compositional embeddings [16, 27] and multidimensional embeddings [20, 42]. Compositional embedding-based recommenders consist of a small number (substantially smaller than the number of all users and items) of meta-embeddings, such that a user/item can be represented as a distinct combination of selected meta-embeddings. Meanwhile, multidimensional embeddings allow for each embedding to have multiple candidate dimensions during training. As each user/item does not necessarily need the largest embedding dimension to encode all its information (e.g., long-tail items with limited predictive signals), the optimal embedding dimension for each user/item can be automatically learned to reduce excessive parameters while retaining the recommendation performance.

With the increasing diversity of IoT facilities, on-device recommendation is no longer exclusive to cellphones, and rather, becomes feasible on smaller devices such as smart watches, Internet TV boxes, and routers. Heterogeneity exists not only across different device categories, but also across different device ages, e.g., iPhone 11 has doubled the RAM size of iPhone 8 in just two years. Ideally, for the same recommendation task, a model should be customized for each specific device, such that the recommendation performance and utilization of space are both optimized. Unfortunately, in existing paradigms, a memory-efficient recommender specialized for each on-device memory budget has to be designed and trained from scratch, incurring high engineering costs. Though recent advances in automated machine learning (AutoML) can ease the difficulties in the design phase [20] with neural architecture search (NAS), retraining is still mandatory every time a new memory constraint needs to be met. The model training and architecture search are normally entangled in NAS-based models, making the learned recommender parameters (i.e., embeddings in our case) and architecture only optimized towards one specific memory budget. As a result, such "train once, get one" scheme is too inefficient and unsustainable to keep up with the prosperity of on-device computing, e.g., real-time download requests on a recommender system in app stores from different devices.

To this end, we suggest a new lightweight, once-for-all recommendation paradigm that is able to automatically adapt to heterogeneous devices with minimal hassles after being fully trained. This is achieved by decoupling the embedding training and search phases via our notion of ***elastic embeddings*** for on-device recommendation. It is worth noting that in practice, a user's device can only access and store her/his own user embedding, so we will specifically target on compressing all item embeddings in our work. Essentially, instead of learning a single embedding with relatively high dimensionality (e.g., 128-dimensional) for every item, we view a full item embedding as the concatenation of smaller embeddings (e.g., 16 8-dimensional vectors) which we term ***embedding blocks***. This is distinct from compositional embeddings [27] as each item's embedding blocks are exclusive and not shared, ensuring stronger expressiveness. After all embedding blocks are trained, we aim to build an automated search function to identify an appropriate subset of embedding blocks to be concatenated into an elastic embedding for each item. All searched elastic embeddings are then used for on-device recommendation, providing optimal performance under any given memory constraint. Elastic embeddings offer abundant flexibility on which and how many embedding blocks can be chosen, controlling the information carried and memory consumed by each item embedding. With an efficient search function, the search cost is negligible compared with retraining a new model, so we can effortlessly configure the most suitable elastic embeddings for heterogeneous devices without retraining.

However, non-trivial obstacles need to be tackled before we can benefit from this paradigm. Given the large search space spanning across all items' embedding blocks, it is impractical for the search function to enumerate all embedding block combinations and evaluate their performance. Also, as all embedding blocks are trained before on-device deployment, we need to ensure that every embedding block is sufficiently informative to guarantee high-quality recommendations whenever it is selected in the search phase. To address those issues, we present a novel solution for recommendation with universally learned elastic embeddings, or **RULE** for short. Specifically, to facilitate efficient yet accurate search for elastic embeddings, we propose to identify suitable embedding blocks at the item group level rather than the individual item level. Intuitively, this allows intra-group items to share similar elastic embedding structures and significantly shrinks the search space. Meanwhile, when training all embedding blocks, we propose a diversity-driven regularizer to encourage variety in the information encoded in different embedding blocks, thus ensuring the expressiveness of arbitrary elastic embeddings. With the trained embedding blocks, we design a performance estimator-based method for evolutionary search [24] to bypass the need for evaluating every candidate elastic embedding structure. To further speed up the search process, we introduce a Gaussian prior when determining the number of embedding blocks assigned to each group, so that the search function tend to favor a large/small embedding dimension for only a few really important/unimportant item groups while keeping other groups' embedding dimension balanced. As such, RULE is a low-cost solution to customizing on-device recommenders.

Our contributions in this work are three-fold:

- We point out the inflexibility of the current lightweight recommendation paradigm when facing diverse on-device memory budgets, and propose a new, once-for-all recommendation paradigm using elastic embeddings. Our paradigm allows a recommender to be quickly specialized for arbitrary on-device memory constraints without retraining.
- We propose RULE, a novel solution that addresses the key challenges in the execution of this new paradigm. RULE can efficiently search adequate elastic embedding structures to achieve optimal on-device recommendation performance.
- We conduct extensive experiments on two real-world datasets under multiple memory constraints. Experimental results show that RULE yields superior recommendation performance compared with state-of-the-art baselines.

2 PRELIMINARIES

We hereby define some key concepts and data structures in our method, namely the ***embedding block*** and ***elastic embedding***.

Definition 1 (Embedding Block): For an arbitrary item $v_j \in \mathcal{V}$, its full embedding $\mathbf{v}_j \in \mathbb{R}^D$ is the concatenation of N separate d-dimensional vectors, i.e., $D = Nd$ and $\mathbf{v}_j = [\mathbf{e}_{j(1)}, \mathbf{e}_{j(2)}, ..., \mathbf{e}_{j(N)}]$ where $\mathbf{e}_{j(n)} \in \mathbb{R}^d$ $(1 \leq n \leq N)$ is referred to as an ***embedding block***.

Definition 2 (Elastic Embedding): An elastic embedding $\mathbf{v}_j^*$ for item $v_j \in \mathcal{V}$ is the concatenation of a non-overlapping subset of its embedding blocks. Mathematically, $\mathbf{v}_j^* = \|_{n\in\mathcal{B}}\, \mathbf{e}_{j(n)} \in \mathbb{R}^{|\mathcal{B}|d}$, where we use $\|$ to denote the iterative concatenation of vectors, and $\mathcal{B}$ stores the selected block indexes ($|\mathcal{B}| \leq N$). Hence, the dimensionality and expressiveness of $\mathbf{v}_j^*$ are both dependant on the composition of $\mathcal{B}$.

3 DEFINING THE SEARCH SPACE OF RULE

Following Section 2, at this stage we have $|\mathcal{V}|\times N$ different embedding blocks for all items in total. Assuming each $\mathbf{v}_j^*$ receives at least one embedding block, and all embedding blocks in $\mathbf{v}_j^*$ are indexed in an ascending order, then the total number of possible combinations is $2^N - 1$ for each item, and is $(2^N-1)^{|\mathcal{V}|}$ for all. As the number of items easily reaches millions in modern e-commerce applications, it is impractical to define the search space over $|\mathcal{V}|\times N$ embedding blocks due to the potentially high search cost. Hence, it makes sense for us to narrow down the search space by grouping multiple items together. That is, we segment the item set into G subsets (i.e., groups) via $\mathcal{V} = \{\mathcal{V}_1, \mathcal{V}_2, ..., \mathcal{V}_G\}$ that are equally sized, i.e., $|\mathcal{V}_g| = \frac{|\mathcal{V}|}{G}$ for $1 \leq g \leq G$. For all items in each group $\mathcal{V}_g$, we concatenate their n-th embedding block into a larger, $|\mathcal{V}_g|\times d$-dimensional embedding block[2] $\mathbf{E}_{g(n)}$. Then the selection of embedding blocks becomes a group-level process, making all items within group $\mathcal{V}_g$ have the same embedding block indexes, denoted by $\mathcal{B}_g$.

Figure 1 provides a graphic view of the key notations w.r.t. the search space. The resulted search space has some desirable characteristics. Firstly, with $G \ll |\mathcal{V}|$, we can effectively reduce the number of embedding blocks to $G\times N$. Secondly, assuming items in each group share similar properties (e.g., items in the same category or cluster), then the selection of the best embedding blocks becomes a collective learning goal [30], where $\mathcal{B}_g$ is searched for optimizing the recommendation within $\mathcal{V}_g$ as a whole. As the strategies for segmenting items into groups will shape the search space of RULE, we will discuss and examine different strategies via experiments in Section 6. Recall that each device only needs to store one user embedding which has negligible size compared with all item embeddings, we formulate our problem below.

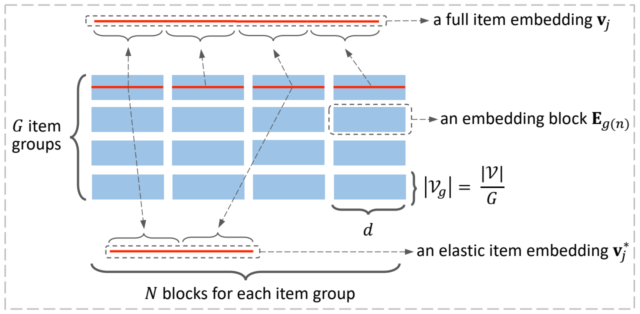
Problem 1 (On-Device Recommendation with Customized Elastic Embeddings): For each on-device memory budget M (measured by MB), RULE searches $\{\mathcal{B}_g\}_{g=1}^G$ for all item groups, providing accurate recommendations with $size(\{\mathbf{v}_j^*\}_{j=1}^{|\mathcal{V}|} \cup \{\mathbf{u}_i\}) \leq M$ where $\mathbf{u}_i \in \mathbb{R}^D$ is an arbitrary user's full embedding.

4 LEARNING THE FULL EMBEDDINGS

4.1 Base Recommender

In RULE, before optimizing its memory footprint, we need to obtain expressive embeddings to guarantee high-quality recommendations. To achieve this, any latent factor model can be a good fit, ranging from matrix factorization [14] to deep models [10]. In this paper, we utilize a graph neural network, namely the LightGCN [9] that recently advances the performance over various recommenders. Note that our main contribution lies in the elastic, memory-efficient recommendation framework rather than the base recommender, hence we describe it in a brief streak.

[2]Directly concatenating $|\mathcal{V}_g|$ column vectors will lead to a $d \times |\mathcal{V}_g|$ matrix but we follow the commonly used look-up table style to keep the notations intuitive.



a full item embedding $\mathbf{v}_j$
G item groups
an embedding block $\mathbf{E}_{g(n)}$
$|\mathcal{V}_g| = \frac{|\mathcal{V}|}{G}$
d
an elastic item embedding $\mathbf{v}_j^*$
N blocks for each item group

Figure 1: A graphic explanation on key notations used.

Recommender Structure Summary. Treating each user/item as a distinct node, the user-item interactions can be formulated as a bipartite graph, where we use $\mathcal{G}(u_i)$ and $\mathcal{G}(v_j)$ to respectively denote the one-hop neighbor sets of user u_i and item v_j. Given an arbitrary user/item, its embedding is updated by propagating its neighbors' embeddings into u_i/v_j:

$$\mathbf{v}_j^{(l)} = \sum_{i\in\mathcal{G}(v_j)} \eta_{ij}\mathbf{u}_i^{(l-1)}, \quad \mathbf{u}_i^{(l)} = \sum_{j\in\mathcal{G}(u_i)} \eta_{ij}\mathbf{v}_j^{(l-1)}, \tag{1}$$

where the user/item embeddings at the l-th layer ($l \geq 1$) is formed by aggregating all its neighbors' D-dimensional embeddings at the previous layer. $\eta_{ij} = (\sqrt{|\mathcal{G}(u_i)|\cdot|\mathcal{G}(v_j)|})^{-1}$ is a graph Laplacian normalization term that is proven effective in a variety of literatures [9, 12]. $\mathbf{u}_i^{(0)}$ and $\mathbf{v}_j^{(0)}$ are randomly initialized and will be trained via back-propagation. For notation simplicity, we denote the output embeddings from the ***final layer*** L as $\mathbf{u}_i = \mathbf{u}_i^{(L)} \in \mathbb{R}^D$ and $\mathbf{v}_j = [\mathbf{e}_{j(1)}, \mathbf{e}_{j(2)}, ..., \mathbf{e}_{j(N)}] = \mathbf{v}_j^{(L)} \in \mathbb{R}^D$, which are used in the subsequent recommendation and elastic embedding search phases.

4.2 Handling Item Embeddings with Elastic Dimensions

This is where our own design emerges, starting with a flexible pairwise scoring function that can adapt to item embeddings with elastic dimensions during inference. Generally, to rank all items in $\mathcal{V}$ for each u_i, a pairwise scoring function that estimates the similarity between a user-item embedding pair $(\mathbf{u}_i, \mathbf{v}_j)$ is indispensable. Specifically, distance metrics like dot product and cosine similarity are the most commonly used ones owing to their simplicity and efficacy. Those metrics work well when user and item vectors are of the same dimension, which is true during training (i.e., $\mathbf{u}_i$, $\mathbf{v}_j \in \mathbb{R}^D$). However, when elastic embedding $\mathbf{v}_j^* \in \mathbb{R}^{|\mathcal{B}_g|d}$ ($|\mathcal{B}_g|d \leq D$) is in use for item v_j, methods like dot product become ill-posed once there is a mismatch between two embedding dimensions. Another common pairwise scoring scheme is to pass the concatenation of user and item embeddings into a multi-layer perceptron (MLP) with a linear regression layer [10]. Though this makes two embeddings with unequal dimensions comparable, it is impractical by design as the MLP must be trained with a fixed input dimension (i.e., $2D$ in our case). So, for item $v_j \in \mathcal{V}_g$, we propose a dimension-independent scoring function below:

$$r_{ij} = \frac{\max(\{|\mathcal{B}_{g'}|\}_{g'=1}^G)}{|\mathcal{B}_g|} \cdot \sum_{\forall n\in\mathcal{B}_g} \mathbf{u}_i^\top \bar{\mathbf{e}}_{j(n)}, \tag{2}$$

where $\bar{\mathbf{e}}_{j(n)} = \|_{\beta=1,\dots,N} \mathbf{e}_{j(n)} \in \mathbb{R}^D$ is constructed by repeating the same embedding block $\mathbf{e}_{j(n)}$ for N times. Noticeably, for all $\mathbf{e}_{j(n)} \in \mathbf{v}_j^*$ ($v_i \in \mathcal{V}_g$), we can result in $|\mathcal{B}_g|$ similarity scores. As different elastic item embeddings have varied numbers of embedding blocks, directly adding up all these scores might lead to significant variance in scale. Hence, we use the proportion between the maximum number of embedding blocks $\max(\{|\mathcal{B}_{g'}|\}_{g'=1}^{G})$ and the number of $\mathbf{v}_j^*$'s embedding blocks $|\mathcal{B}_g|$ as a normalization term, making all user-item affinity scores r_{ij} mutually comparable. Meanwhile, in the training process where the full item embedding $\mathbf{v}_j$ is used, the normalization term is constantly 1 and can be omitted. Also, this parameter-free design prevents the scoring function from stealing the memory budget from elastic embeddings.

4.3 Learning Diversified Embedding Blocks

To train RULE towards the recommendation task, we optimize all embeddings with Bayesian personalized ranking (BPR) [26] loss. At the same time, as each item's elastic embedding is composed by several different embedding blocks, the embedding blocks should be distinct from each other regarding the information they encode, thus maximizing the embeddings' expressiveness. Hence, on top of the recommendation loss, we further impose regularization effect on the diversity of all embedding blocks to be learned. The loss function of RULE is formulated as the following:

$$L_{rec} = -\sum_{(u_i, v_{j^+}, v_{j^-}) \in \mathcal{D}_{rec}} \varrho(r_{ij^+} - r_{ij^-}) - \lambda \sum_{n=1}^{N} \sum_{n'=n+1}^{N} \|\mathbf{E}_n - \mathbf{E}_{n'}\|_F^2, \quad (3)$$

where the first and second terms are respectively the BPR loss and the diversity regularizer. The rationale of the BPR loss is that, for user u_i, the ranking score for a visited item v_{j^+} should always be higher than the ranking score for an unvisited one v_{j^-}. As such, a training sample is defined as a triple $(u_i, v_{j^+}, v_{j^-}) \in \mathcal{D}_{rec}$, and $\mathcal{D}_{rec}$ denotes the set of all training samples. In the regularization term, $\|\cdot\|_F^2$ denotes the squared Frobenius norm, and $\mathbf{E}_n = \|_{g=1}^{G} \mathbf{E}_{g(n)} \in \mathbb{R}^{|\mathcal{V}| \times d}$ is the collection of the n-th embedding block for all G groups. It generally encourages two different embedding blocks to encode different information element-wise, thus posing a regularization effect on the diversity of learned embedding blocks.

5 ELASTIC EMBEDDING SEARCH

Our RULE model can be viewed as an instantiated variant of automated neural architecture search (NAS). A commonly adopted workflow when deploying NAS for recommendation can be summarized as the iteration of two steps: (1) randomly generate candidate models (i.e., elastic embeddings in our case); and (2) train and evaluate the candidate models for the given task to find the currently best one. In RULE, training is exempted in step (2) as all elastic embeddings directly use the embedding blocks learned in Section 4. But unlike most NAS-driven recommenders [20, 31, 35, 43] whose objective only concerns the recommendation accuracy, in RULE, the searched elastic embeddings must also be memory-bounded so as to comply with the memory budget M.

A straightforward solution is to formulate the problem as a reinforcement learning (RL) task [11], where a trainable controller takes over step (1), and a performance reward computed in step (2) binds a memory cost term to demote the controller's selection of over-budget embedding block combinations. However, as the performance evaluation in step (2) is considerably time-consuming, it is prohibitively uneconomical to prepare a recommender for every possible on-device memory budget M. Also, most memory costs used in the reward are merely a "soft" limiter on the memory consumption [11], which can be problematic for on-device applications if the size of elastic embeddings are not strictly bounded by M. Apart from RL, gradient-based NAS methods like DARTS [19] also suffer from both defects in our recommendation paradigm.

In this section, we present our own solution to automated elastic embedding search. To bypass the underlying inflexibility while imposing strict memory constraints, we define a memory-bounded randomizer for generating candidate models, and perform evolutionary search with a pretrained performance estimator to construct the best-performing elastic embeddings. In what follows, we will expand the corresponding technical details.

Algorithm 1 Memory-Bounded Randomizer, i.e., *random*(·)

```
1:  Input: $N$, $G$, memory budget $M$ (converted to the total number of embedding blocks)
2:  Output: Randomly selected embedding blocks $\{\mathcal{B}_g\}_{g=1}^{G}$
3:  $model.EE \leftarrow \varnothing$, $\mathcal{S} \leftarrow \varnothing$, $\mu \leftarrow \frac{M}{G|\mathcal{V}_g|}$, $\sigma^2 \leftarrow 1$;
4:  for $g \leftarrow 1, 2, \cdots, G$ do
5:      draw $s_g \sim \mathcal{N}(s|\mu, \sigma^2)$ where $s \in \{1, 2, \dots, \min(N, M-G+1)\}$;
6:      $\mathcal{S} \leftarrow \mathcal{S} \cup \{s_g\}$;
7:  end for
8:  while $\sum_{s \in \mathcal{S}} s > M$ do
9:      uniformly draw $g$ from $\{1, 2, \dots, G\}$;
10:     if $s_g > 1$ then
11:         $s_g \leftarrow s_g - 1$
12:     end if
13: end while
14: for $g \leftarrow 1, 2, \cdots, G$ do
15:     $\mathcal{B}_g \leftarrow$ draw $s_g$ unique embedding blocks from $\{\mathbf{E}_{g(n)}\}_{n=1}^{N}$;
16:     $model.EE \leftarrow model.EE \cup \{\mathcal{B}_g\}$;
17: end for
18: return $model.EE$
```

5.1 Memory-Bounded Embedding Block Randomizer

In RULE, we propose to enforce the memory constraint at an earlier stage, i.e., when randomly selecting embedding blocks for every *model.EE*. Here *model.EE* denotes a candidate model's elastic embeddings. This is more computationally efficient as the selected embedding blocks fully comply with the memory budget, and the search for optimal elastic embeddings is now purely conditioned on the model performance. This is also achievable as the total number of embedding blocks that *model.EE* will consume can be easily computed with any given M. For example, suppose M = 10MB, $N = 8$, $d = 16$, $|\mathcal{V}| = 100,000$, and $G = 10$. Then, in a 32-bit floating point system (i.e., 4×10^{-6}MB per parameter), each embedding block occupies 0.64MB memory from $\frac{d|\mathcal{V}|}{G} = 1.6 \times 10^5$ parameters. Considering the size of one user embedding $size(\mathbf{u}_i) = 5.12 \times 10^{-4}$MB, the maximum number of total embedding blocks that can fit in M is $\lfloor \frac{M - size(\mathbf{u}_i)}{0.64\text{MB}} \rfloor = 15$.

Then, the problem comes down to deciding how many embedding blocks each item group will have, and which embedding blocks will be chosen. To further reduce the search complexity, we introduce a Gaussian prior, specifically standard normal distribution when determining the number of embedding blocks assigned to each group. The rationale is that, items' influence tend to follow normal distribution in recommendation tasks [17, 41], which can be also reflected via the distribution of their embedding dimensions. In other words, only the minority among items will be excessively impactful or irrelevant to the recommendation results and need substantially more/less embedding blocks, while the influences of most items are similar, so do their embedding dimensions.

We provide a mathematical view of the memory-bounded randomizer *random*$(\cdot)$ via Algorithm 1, where we keep using M to denote the maximum number of embedding blocks to be succinct. Specifically, we firstly assign $s_g \in \{1, 2, ..., \min(N, M-G+1)\}$ to each group $\mathcal{V}_g$ based on standard normal distribution $\mathcal{N}(\mu, \sigma^2)$, where the mean value μ is inferred from M and variance σ^2 is set to 1 (lines 4-7). Then, we iteratively calibrate the total number of embedding blocks to ensure it complies with the memory budget M (lines 8-13). Lastly, once the numbers of embedding blocks is settled for all groups, we uniformly sample s_g embedding blocks from $\{\mathbf{E}_{g(n)}\}_{n=1}^{N}$ for the g-th item group (lines 15-17).

5.2 Performance Estimator

Evolutionary search [24] has been widely used for progressively searching for optimal neural architectures on a given task. However, in our case, evaluating every randomly sampled *model.EE* on a validation dataset will impede its efficiency. In real-life scenarios, customized models need to be retrieved for arbitrary devices within a short time (e.g., a download request). Hence, we advocate training an accurate performance estimator to provide quick feedback on recommendation quality. Given $\{\mathcal{B}_g\}_{g=1}^{G}$, we first construct a vectorized input for the estimation function as follows:

$$\mathbf{x}_g = \mathbf{W} \cdot \underbrace{[\text{one-hot}(g), \text{multi-hot}(\mathcal{B}_g)]}_{\text{multi-hot encoding for the } g\text{-th item group}}, \tag{4}$$

where one-hot(g) is the G-dimensional one-hot encoding of the g-th item group, and multi-hot$(\mathcal{B}_g)$ returns the N-dimensional multi-hot encoding representing the indexes of the selected embedding blocks. $\mathbf{W}$ is a weight matrix projecting the multi-hot input for the g-th group into a d_0-dimensional vector. Then, to thoroughly capture the combinatorial effect among inter-group elastic embeddings, we build our performance estimator by modelling factorized pairwise interactions [5, 25] between item groups:

$$\begin{aligned}\widehat{y} &= estimate(\{\mathbf{x}_g\}_{g=1}^{G}) \\ &= \mathbf{w}^{\top} \cdot FFN(\underbrace{\sum_{g=1}^{G}\sum_{g'=g+1}^{G} \mathbf{x}_g \odot \mathbf{x}_{g'}}_{\text{interactions between groups}}) + b,\end{aligned} \tag{5}$$

where $\odot$ denotes the element-wise multiplication, scalar $\widehat{y}$ is the predicted recommendation performance (e.g., recall value), $FFN(\cdot)$ is a single-layer feed-forward network that outputs a d_0-dimensional vector summarizing the pairwise interactions among all item groups' input features $\mathbf{x}_g$. Then, the encoded combinatorial effect of different groups' elastic embedding compositions is mapped to $\widehat{y}$ with projection weight $\mathbf{w} \in \mathbb{R}^{d_0}$ and scalar bias b.

Training The Performance Estimator. We quantify the difference between the predicted performance and the actual one via:

Algorithm 2 Performance Estimator-based Evolutionary Search

```
1: Input: $M$, $N$, $G$, three search coefficients $P$, $S$, $C$
2: Output: RULE model with searched elastic item embeddings
3: $seed \leftarrow \emptyset$, $cache \leftarrow \emptyset$;
4: while $|seed| < P$ do
5:     $model.EE \leftarrow random(M, N, G)$;
6:     $model.ACC \leftarrow estimate(model.EE)$;
7:     $seed \leftarrow seed \cup \{child\}$;
8:     $cache \leftarrow cache \cup \{child\}$;
9: end while
10: for $c \leftarrow 1, 2, \cdots, C$ do
11:     $sample \leftarrow \emptyset$;
12:     $sample \leftarrow$ draw $S$ models with replacement from $seed$;
13:     $parent \leftarrow model$ s.t. $model.ACC$ is the highest in $sample$;
14:     $child.EE \leftarrow mutate(parent.EE)$;
15:     $child.ACC \leftarrow estimate(model.EE)$;
16:     $seed \leftarrow seed \cup \{child\}$;
17:     $cache \leftarrow cache \cup \{child\}$;
18:     remove $model$ with the lowest $model.ACC$ from $seed$;
19: end for
20: return $model$ with the highest $model.ACC$ in $cache$
```

$$L_{est} = \sum_{(model.EE, y_{model.EE}) \in \mathcal{D}_{est}} (y_{model.EE} - \widehat{y}_{model.EE})^2, \tag{6}$$

where $\mathcal{D}_{est}$ contains all $(model.EE, y_{model.EE})$ ground truth tuples for training. The ground truth is obtained by randomly acquiring a certain amount of different elastic embedding compositions *model.EE* (with no memory restrictions) and evaluate them via any commonly used evaluation metrics like recall and precision (implementation details are provided in the Appendix). By controlling the size of $\mathcal{D}_{est}$, we can achieve a trade-off between the prediction confidence and training time of our performance estimator.

5.3 Evolutionary Search with *estimate*$(\cdot)$

To bypass the time-consuming model evaluation process, we perform evolutionary search with well-trained performance estimator. Algorithm 2 depicts the search process, where *model.ACC* denotes each candidate model's accuracy. Specifically, our performance estimator-based evolutionary search maintains a set of P models (denoted as *seed*) with random elastic embedding combinations, which are initialized with our memory-bounded randomizer *random*$(\cdot)$ based on the given memory budget M. The evolution lasts for a total of C rounds (lines 10-19). In every iteration, we draw S models from *seed* and select the one with the highest estimated performance as *parent*. Then, the *parent* model is modified into a *child* version with our *mutate*$(\cdot)$ function. We design two mutation operations on *parent*: (1) swapping the embedding blocks between two random item groups; (2) reselecting a same amount of embedding blocks for one item group at random. Only one mutation operation will be executed with uniform probability at each iteration, and the resulted model *child* will be evaluated by our performance estimator. The rationale is that, if we keep revising the currently best model *parent*, it is likely to obtain more accurate results. After C evolutionary rounds, the best-performing model is retrieved from *cache* that stores all *seed* and *child* models.

In this way, once the full embeddings and performance estimator are well trained in RULE, whenever it is required to perform recommendation on any device, RULE can quickly adapt to the device's memory constraint by selecting the most suitable embedding blocks so as to generate accurate recommendations.

Table 1: Statistics of experimental datasets.

Dataset	#User	#Item	#Interaction	Sparsity
Amazon-Book	52,643	91,599	2,984,108	99.94%
Yelp2020	138,322	105,843	3,865,586	99.97%

6 EXPERIMENTS

We evaluate the performance of RULE under different on-device memory constraints[3]. In particular, we aim to answer the following research questions (RQs) via experiments:

RQ1: Under certain memory constraints, how is the recommendation accuracy of RULE compared with other baselines?
RQ2: What is the contribution of each key component of RULE?
RQ3: How is the on-device usability, specifically latency of RULE when being deployed on heterogeneous devices?
RQ4: How the hyperparameters affect the performance of RULE?

6.1 Datasets and Baseline Methods

We use two benchmark datasets for evaluation, namely **Amazon-Book** and **Yelp2020**. The Amazon review data is originally crawled and made public by [8]. We select Amazon-Book for our experiments as it is the largest dataset in the collection. Yelp is an open review platform for various businesses (e.g., restaurants and hotels). We use its public dataset released in 2020[4], where businesses are viewed as items. Both datasets are publicly available and consist of million-scale user-item interactions. Following [26], we filter out inactive users with less than 10 interacted items and unpopular items visited by less than 10 users. Their primary statistics are shown in Table 1. We split both datasets with the ratio of 70%, 10% and 20% for training, validation and test, respectively.

We compare with six baselines from three categories. The first ones are mainstream recommenders with fix-sized embeddings, including probabilistic matrix factorization (**PMF**) [21] and **LightGCN** [9]. The second type is AutoML-based lightweight recommenders using multidimensional embeddings, which are **ESAPN** [20] and **AutoEmb** [43]. The key difference between them is that ESAPN is optimized via RL while AutoEmb has a differentiable search process. Lastly, we have two compositional embedding methods. One is **DLRM-CE** [27] that generates compositional embeddings with a quotient-remainder trick. The other is **LightRec** [16] which composes item embeddings with parallel codebooks.

6.2 Evaluation Protocols

We test all methods with three on-device memory budgets, i.e., 5MB, 10MB, and 25MB. To ensure all tested methods fit in the given memory budget, we adjust the corresponding embedding dimensions according to M so that the total size of: (1) the embedding parameters needed for representing all items and one user; and (2) all weights and biases used for ranking, is right below M. We leverage two ranking metrics, namely recall at rank K (Recall@K) and normalized discounted cumulative gain at rank K (NDCG@K) that are widely adopted in recommendation research [2, 6, 32, 36]. We adopt $K = 50, 100$ where all items unvisited by each user are taken as negative samples for evaluation. In short, Recall@K measures the ratio of the ground truth items that are present on the top-K list, and NDCG@K evaluates whether the model can rank the ground truth items as highly as possible.

[3]Public access to codes: http://bit.ly/bitbucket-RULE
[4]https://www.yelp.com/dataset

6.3 Recommendation Performance (RQ1)

Implementation Notes. We report the hyperparameter settings of RULE in the Appendix for reproducibility, which are tuned via grid search. The Appendix also covers details on how the performance estimator's training set $\mathcal{D}_{est}$ is constructed and how it is trained. Regarding the elastic embeddings, the optimal hyperparameters are $\{d = 8, N = 16, D = Nd = 128, G = 20\}$, whose impact will be further discussed in subsequent sections. By default, RULE randomly segments all items into G groups, and two other segmentation strategies will be investigated in Section 6.4.

We list the overall recommendation performance of all tested methods in Table 2. The first observation we can draw from the results is that, RULE consistently outperforms all baselines with a significant margin ($p < 0.05$ in paired t-test w.r.t. with the best baseline in each column). Specifically, compared with methods that use compositional embeddings (i.e., DLRM-CE and LightRec), RULE shows advantageous performance with all memory budgets, especially when the memory budgets are relatively small (i.e., 5MB and 10MB). This showcases the privilege of our proposed elastic embedding paradigm, which learns a set of unique embedding blocks for each item rather than sharing meta-embedding vectors among items. It is also worth mentioning that, with the automated search function, RULE is efficiently customized for three different memory budgets after one training cycle, while all other baseline methods are individually trained towards each memory constraint.

At the same time, as fix-sized embedding methods, PMF and LightGCN still demonstrate adequate performance under the on-device memory constraints. It is worth noting that, LightGCN, on which our method is based, yields competitive results on Amazon-Book. On the sparser Yelp2020 dataset, when the memory size is too small to allow for sufficient embedding dimensions (e.g., 5MB), its performance deteriorates heavily. In contrast, by actively selecting the most useful embedding block for the current memory setting, RULE is able to maintain its performance at a higher level. Surprisingly, in spite of the sophisticated design of the NAS-based methods ESAPN and AutoEmb, they cannot achieve satisfying recommendation results when the search space for embedding dimensions is constrained by a tight memory budget. Another possible reason is that both methods are originally specialized for rating prediction, hence inevitably facing difficulties generalizing to the ranking task.

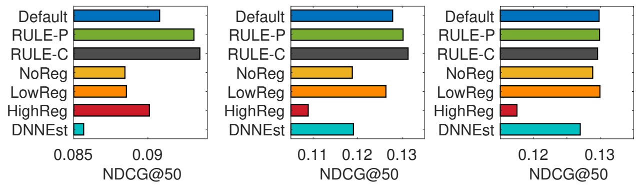
6.4 Effectiveness of Model Components (RQ2)

To better understand the performance gain from different key components, we implement several variants of RULE by making modifications to one component each time, and record the new performance achieved. The results with all three memory budgets are reported via Figure 2. We choose NDCG@50 as a performance indicator in this study. In what follows, we introduce all variants and analyze the effect of corresponding model components.

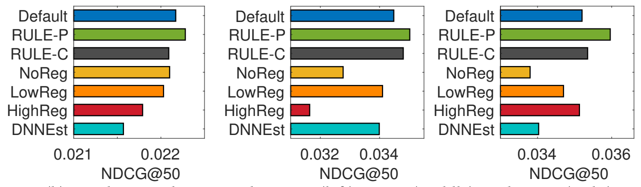
Different Item Segmentation Strategies. We further testify two empirical strategies for grouping items, namely popularity-based (RULE-P) and clustering-based item groups (RULE-C) with $G = 20$ for both. More implementation details are provided in the Appendix. As the results suggest, by grouping items based on their common properties, the recommendation accuracy can be slightly improved in some cases. To further interpret the rationale of these

Table 2: Recommendation results. Numbers in bold face are the best results for corresponding metrics.

Dataset	Method	5MB				10MB				25MB			
		Recall@K		NDCG@K		Recall@K		NDCG@K		Recall@K		NDCG@K	
		$K=50$	$K=100$	$K=50$	$K=100$	$K=50$	$K=100$	$K=50$	$K=100$	$K=50$	$K=100$	$K=50$	$K=100$
Amazon-Book	PMF	0.02457	0.04301	0.02552	0.03169	0.02717	0.04921	0.02873	0.03525	0.02878	0.05195	0.03050	0.03848
	LightGCN	0.05070	0.08578	0.08864	0.10193	0.06460	0.09300	0.10738	0.13292	0.06517	0.08931	0.10937	0.14894
	ESAPN	0.02148	0.04454	0.03943	0.06377	0.02336	0.04969	0.05269	0.08795	0.02561	0.06193	0.05611	0.08979
	AutoEmb	0.02024	0.04215	0.03890	0.05949	0.02327	0.05146	0.05033	0.08581	0.02496	0.06010	0.05412	0.08792
	DLRM-CE	0.02331	0.04722	0.04218	0.06948	0.02711	0.06247	0.05817	0.09726	0.03280	0.08363	0.06755	0.10010
	LightRec	0.04562	0.07040	0.07419	0.08982	0.05185	0.08289	0.08277	0.09825	0.05956	0.08093	0.09890	0.13765
	RULE	**0.05335**	**0.08785**	**0.09079**	**0.12261**	**0.06537**	**0.09416**	**0.12787**	**0.15785**	**0.06622**	**0.10161**	**0.12983**	**0.18165**
Yelp2020	PMF	0.00798	0.01726	0.01048	0.01537	0.08160	0.01900	0.01273	0.01861	0.09126	0.02013	0.01542	0.02394
	LightGCN	0.01327	0.02435	0.01971	0.02670	0.01784	0.02823	0.02772	0.04219	0.01992	0.03441	0.03188	0.05398
	ESAPN	0.00713	0.01959	0.01372	0.01980	0.00806	0.02186	0.01452	0.02119	0.00903	0.02325	0.01543	0.02179
	AutoEmb	0.00765	0.01912	0.01321	0.01920	0.07838	0.02087	0.01417	0.02236	0.00875	0.02214	0.01503	0.02093
	DLRM-CE	0.01214	0.02753	0.01239	0.01756	0.01381	0.02992	0.01596	0.02232	0.01484	0.03098	0.01945	0.02712
	LightRec	0.00805	0.02314	0.01536	0.02372	0.01259	0.02146	0.01487	0.02233	0.01334	0.02524	0.02098	0.03475
	RULE	**0.02007**	**0.03602**	**0.02217**	**0.03282**	**0.02181**	**0.03875**	**0.03448**	**0.05447**	**0.02288**	**0.03947**	**0.03520**	**0.05529**



(a) Results on Amazon-Book within 5MB (left), 10MB (middle), and 25MB (right).



(b) Results on Yelp2020 within 5MB (left), 10MB (middle), and 25MB (right).

Figure 2: Performance of different variants of RULE.

two item segmentation strategies, we visualize the searched elastic embeddings in RULE-P and RULE-C in Figure 3. As can be told from Figure 3(a), popular item groups are more likely to receive more embedding blocks (i.e., larger embedding dimensions), thus fully encoding the rich information from interactions with users. In Figure 3(b), with the item clusters visualized, we empirically find that similar elastic embedding compositions can be found from two item groups if their clusters are close to each other, and vice versa. Hence, it might be of interest to practitioners to pursue better recommendation results with finer item segmentation strategies.

Diversity-driven Regularizer. To verify the contribution of the diversity-driven regularizer in Eq.(3), we derive three variants of RULE by setting λ to 0, 1×10^{-6} and 1×10^{-2} (denoted by NoReg, LowReg, and HighReg in Figure 2, respectively, and $\lambda = 1 \times 10^{-4}$ by default). Compared with LowReg, a slightly higher regularization weight as used in the default setting is more beneficial. When the regularizer is removed, the embedding blocks become less diversified, hence the composed elastic embeddings are less informative. Meanwhile, an overwhelmingly strong regularization will let the optimization process overlook the accuracy objective, impeding the final performance.

FM-based Performance Estimator. By replacing our factorization machine-based (FM-based) performance estimator with a single-layer deep neural network (denoted by DNNEst in Figure 2) fed with the concatenation of all item groups’ input vectors from Eq.(4), a performance drop is observed. This verifies that modelling the combinatorial effect between item groups leads to more accurate results on both estimated and actual recommendation performance. Furthermore, we quantitatively evaluate the association between the FM-based estimator and the final recommendation performance. For the FM-based performance estimator, we plot its training root mean square errors (RMSEs) at four different training stages and the corresponding recommendation results achieved on both datasets in Figure 4. Apparently, the actual recommendation performance of RULE is positively associated with the precision of function $estimate(\cdot)$ that directs the search process.

6.5 Latency on Heterogeneous Devices (RQ3)

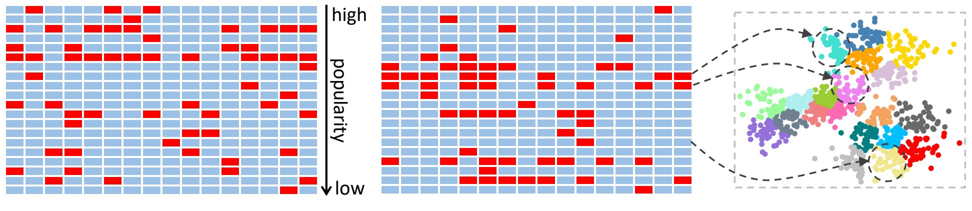
Since the main use case of our proposed recommendation paradigm is on-device services, we test the real-life practicality, specifically the latency of RULE in two hardware configurations. With the help of a Linux virtual machine, we deploy RULE in simulated IoT and cellphone environments, and report the average inference time for generating a full ranked item list for each user. We describe the on-device settings and the latency of RULE below.

IoT: 1× **vCPU (Intel i7-7900K), 64MB RAM.** Note that for both IoT and cellphone configurations, we choose a more realistic number for the RAM size rather than simply our largest memory budget (i.e., 25MB) due to real-world multitasking demands. As we can see from Figure 5(a), with a compact embedding size under 5MB, it takes less than 200MS for the item ranking list to be prepared. Even when M stretches to 25MB on Yelp2020, RULE can still keep the inference time under the 1000MS mark.

Cellphone: 4× **vCPU (Intel i7-7900K), 512MB RAM.** With three more computing cores, RULE has demonstrated minor latency when performing recommendation in a cellphone environment. When the memory budget increases from 5MB to 25MB, the execution time grows from under 50MS on both datasets to 148.5MS and 203.19MS on Amazon-Book and Yelp2020, respectively. To summarize, RULE is capable of learning lightweight elastic embeddings for a wide range of on-device recommendation tasks.

6.6 Hyperparameter Analysis (RQ4)

We further study the impact of two key hyperparameters, namely the number of item groups G and the number of embedding blocks



(a) Popularity-based Item Groups (b) Clustering-based Item Groups

Figure 3: Visualization of searched elastic embeddings with both popularity-based item groups and clustering-based item groups. The visualization corresponds to the Yelp2020 dataset under 10MB budget (59 embedding blocks searched). For better clarity, each cluster is downsampled to 50 items in (b).

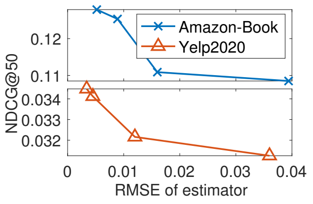
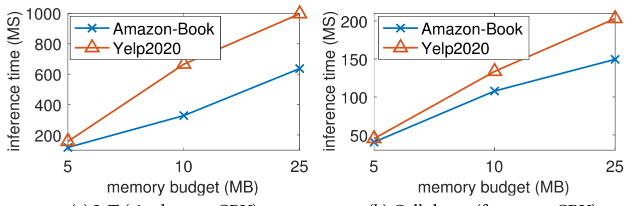
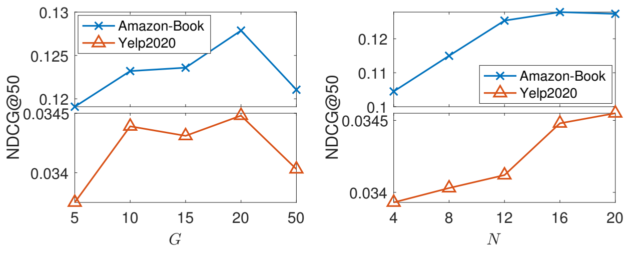


Figure 4: How the performance estimator affects the final recommendation accuracy on Yelp2020. 10MB budget is used for illustration.



(a) IoT (single-core CPU) (b) Cellphone (four-core CPU)

Figure 5: Average inference time of RULE for each user, tested with two on-device environments.



(a) Performance of RULE w.r.t. G. (b) Performance of RULE w.r.t. N.

Figure 6: Impact of G and N to the recommendation performance. Note that we set M =10MB for demonstration while similar trends are observed with other memory budgets.

N, to the performance of RULE. Specifically, we vary the value of G or N at a time while keeping other hyperparameters unchanged, and report the recommendation results achieved via Figure 6. Similar to Section 6.4, we set $M = 10$MB and use NDCG@50 for demonstration, and similar performance fluctuations are observed with other memory budgets.

Impact of G. We study the impact of group number with $G \in \{5, 10, 15, 20, 50\}$. The size of G affects the granularity of the segmented item groups. As can be seen from Figure 6(a), better results are obtained when G ranges from 10 to 20. When the item groups are too coarse (i.e., $G = 5$), it limits the potential of identifying the most suitable embedding blocks for all items. Meanwhile, with $G = 50$, it creates a large search space, making it challenging for RULE to identify the optimal elastic embedding compositions.

Impact of N. We vary N in $\{4, 8, 12, 16, 20\}$. Note that $d = 8$ remains fixed, resulting in a full embedding dimension $D = Nd \in \{32, 64, 96, 128, 160\}$. As Figure 6(b) suggests, more embedding block candidates generally contribute to higher recommendation accuracy. However, as N reaches 16 and 20, the performance increase becomes marginal because the number of embedding blocks that can be chosen is strictly controlled by the memory budget.

7 RELATED WORK

Recommendation models that are memory-efficient have gained immense attention over the years, owing to the increasing need for decentralization [22] and IoT-compatibility [34]. Different from models designed for tasks like neural translation and image processing [1, 13] where the excessive model parameters (i.e., weights and bias) are the main source of memory consumption, most recommender systems' parameterization concentrates on the embedding layers for discrete features, especially user/item IDs. In this regard, a straightforward solution is to convert all real-valued embedding vectors into fixed-length binary codes (e.g., discrete collaborative filtering [15, 39, 40, 44]) so as to shrink the space needed for storing them. However, it is well-known that the binarization of real-valued parameters will lead to significant performance drop due to the quantization loss [18, 37], making discrete methods less ideal for recommendation. Though deep neural networks are later applied as an enhancement upon the binary user/item codes [40], discrete methods inherently falls short in generating semantically rich user/item representations, and hence face an inevitable compromise in recommendation accuracy.

Recently, another variant of lightweight recommendation models are devised based on compositional embeddings [3, 16, 27], some of which are inspired by word embedding compression in language modelling [23, 29, 33]. The key idea of compositional embeddings is to quantize a full user/item embedding matrix into a smaller one with substantially less embedding vectors (a.k.a. meta-embeddings). In such embedding systems, each user/item is represented as a learned composition of several meta-embeddings from the small embedding matrix, such that the embeddings used for recommendation are still highly distinctive and significant parameter reduction can be achieved. In [34], tensor-train decomposition is adopted to compress embeddings, which can also be interpreted as a specific quantization operation applied to each element of the full embedding table [27]. However, as pointed out in Section 1, those methods are all designed towards a predefined memory constraint. When being applied to heterogeneous devices, a model have to be rebuilt and retrained in order to comply with a different memory budget. In summary, the pursuit of lightweight recommender systems that are adaptive to heterogeneous devices is still underway.

As recommender systems start to benefit from memory efficiency in contemporary real-life applications, there have also been emerging research efforts [11, 20, 43] on developing lightweight recommendation models with neural architecture search (NAS). The core idea is to reduce the memory footprint of embeddings by searching for optimal embedding dimensions of each object (e.g., item, user, feature, etc.) instead of using fixed embedding dimensions for all. In [20, 42, 43], the search space is defined by assigning each object a set of embeddings with different dimensions, e.g., learning 3 embeddings for each item with dimensionality of 32, 64, and 128. To avoid over-parameterization of the original embedding space, an alternative definition of the search space is utilized in [11] where a full embedding matrix can be sliced into multiple embedding blocks. Then, automated optimization strategies based on either reinforcement learning or DARTS are respectively adopted by [11, 20] and [42, 43] in order to determine the most suitable embedding size for each object. Unfortunately, after a long training and search process, existing NAS-based recommenders can only produce one final model, making them unsustainable for real-time deployment with heterogeneous memory budgets on different devices during. In contrast, our proposed RULE enables an "once-for-all" recommendation paradigm that can efficiently scale up to heterogeneous on-device memory constraints.

8 CONCLUSION

In this paper, we propose a new paradigm for on-device recommendation, which automatically customizes elastic item embeddings for different memory constraints without retraining. With our novel solution RULE, we firstly learn diversified yet informative embedding blocks, then design a performance estimator-based evolutionary search method to efficiently identify the optimal elastic embedding composition for each item group. The resulted elastic item embeddings fully comply with the strict on-device memory budget, and experiments verify that RULE offers strong guarantees on the recommendation performance and efficiency.

ACKNOWLEDGEMENT

This work is supported by the Australian Research Council under the streams of Discovery Project (No. DP190101985), Training Centre (No. IC200100022), and Centre of Excellence (No. CE200100025).

REFERENCES

[1] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2020. Once-for-all: Train one network and specialize it for efficient deployment. *ICLR* (2020).

[2] Hongxu Chen, Hongzhi Yin, Tong Chen, Weiqing Wang, Xue Li, and Xia Hu. 2020. Social Boosted Recommendation with Folded Bipartite Network Embedding. *TKDE* (2020).

[3] Ting Chen, Lala Li, and Yizhou Sun. 2020. Differentiable product quantization for end-to-end embedding compression. In *ICML*. 1617–1626.

[4] Tong Chen, Hongzhi Yin, Hongxu Chen, Rui Yan, Quoc Viet Hung Nguyen, and Xue Li. 2019. AIR: Attentional intention-aware recommender systems. In *ICDE*. 304–315.

[5] Tong Chen, Hongzhi Yin, Quoc Viet Hung Nguyen, Wen-Chih Peng, Xue Li, and Xiaofang Zhou. 2020. Sequence-Aware Factorization Machines for Temporal Predictive Analytics. *ICDE* (2020).

[6] Tong Chen, Hongzhi Yin, Guanhua Ye, Zi Huang, Yang Wang, and Meng Wang. 2020. Try this instead: Personalized and interpretable substitute recommendation. In *SIGIR*. 891–900.

[7] Lei Guo, Hongzhi Yin, Tong Chen, Xiangliang Zhang, and Kai Zheng. 2021. Hierarchical Hyperedge Embedding-based Representation Learning for Group Recommendation. *TOIS* (2021).

[8] Ruining He and Julian McAuley. 2016. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *WWW*.

[9] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. 2020. LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation. *SIGIR* (2020).

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

[11] 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*. 2387–2397.

[12] Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. *ICLR* (2017).

[13] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The efficient transformer. *ICLR* (2020).

[14] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* (2009).

[15] Defu Lian, Rui Liu, Yong Ge, Kai Zheng, Xing Xie, and Longbing Cao. 2017. Discrete content-aware matrix factorization. In *SIGKDD*. 325–334.

[16] Defu Lian, Haoyu Wang, Zheng Liu, Jianxun Lian, Enhong Chen, and Xing Xie. 2020. LightRec: A Memory and Search-Efficient Recommender System. In *The Web Conference*. 695–705.

[17] Bin Liu, Yanjie Fu, Zijun Yao, and Hui Xiong. 2013. Learning geographical preferences for point-of-interest recommendation. In *SIGKDD*. 1043–1051.

[18] Han Liu, Xiangnan He, Fuli Feng, Liqiang Nie, Rui Liu, and Hanwang Zhang. 2018. Discrete factorization machines for fast feature-based recommendation. In *IJCAI*. 3449–3455.

[19] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2019. Darts: Differentiable architecture search. *ICLR* (2019).

[20] Haochen Liu, Xiangyu Zhao, Chong Wang, Xiaobing Liu, and Jiliang Tang. 2020. Automated Embedding Size Search in Deep Recommender Systems. In *SIGIR*.

[21] Andriy Mnih and Russ R Salakhutdinov. 2007. Probabilistic matrix factorization. *NIPS* 20 (2007), 1257–1264.

[22] Khalil Muhammad, Qinqin Wang, Diarmuid O'Reilly, Elias Tragos, Barry Smyth, Neil Hurley, James Geraci, and Aonghus Lawlor. 2020. Fedfast: Going beyond average for faster training of federated recommender systems. In *SIGKDD*.

[23] Aliakbar Panahi, Seyran Saeedi, and Tom Arodz. 2019. word2ket: Space-efficient Word Embeddings inspired by Quantum Entanglement. In *ICLR*.

[24] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2019. Regularized evolution for image classifier architecture search. In *AAAI*, Vol. 33. 4780–4789.

[25] Steffen Rendle. 2010. Factorization machines. In *ICDM*. 995–1000.

[26] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. 2009. BPR: Bayesian personalized ranking from implicit feedback. In *UAI*. 452–461.

[27] Hao-Jun Michael Shi, Dheevatsa Mudigere, Maxim Naumov, and Jiyan Yang. 2020. Compositional embeddings using complementary partitions for memory-efficient recommendation systems. In *SIGKDD*. 165–175.

[28] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE internet of things journal* 3, 5 (2016), 637–646.

[29] Raphael Shu and Hideki Nakayama. 2018. Compressing Word Embeddings via Deep Compositional Code Learning. In *ICLR*.

[30] Amila Silva, Shanika Karunasekera, Christopher Leckie, and Ling Luo. 2020. METEOR: Learning Memory and Time Efficient Representations from Multi-modal Data Streams. In *CIKM*. 1375–1384.

[31] Qingquan Song, Dehua Cheng, Hanning Zhou, Jiyan Yang, Yuandong Tian, and Xia Hu. 2020. Towards automated neural interaction discovery for click-through rate prediction. In *SIGKDD*. 945–955.

[32] Ke Sun, Tieyun Qian, Tong Chen, Yile Liang, Quoc Viet Hung Nguyen, and Hongzhi Yin. 2020. Where to go next: Modeling long-and short-term user preferences for point-of-interest recommendation. In *AAAI*, Vol. 34. 214–221.

[33] Jun Suzuki and Masaaki Nagata. 2016. Learning Compact Neural Word Embeddings by Parameter Space Sharing. In *IJCAI*. 2046–2052.

[34] Qinyong Wang, Hongzhi Yin, Tong Chen, Zi Huang, Hao Wang, Yanchang Zhao, and Nguyen Quoc Viet Hung. 2020. Next Point-of-Interest Recommendation on Resource-Constrained Mobile Devices. In *The Web Conference*. 906–916.

[35] Ting-Hsiang Wang, Xia Hu, Haifeng Jin, Qingquan Song, Xiaotian Han, and Zirui Liu. 2020. AutoRec: An Automated Recommender System. In *RecSys*.

[36] Hongzhi Yin and Bin Cui. 2016. *Spatio-temporal recommendation in social media*. Springer.

[37] Hanwang Zhang, Fumin Shen, Wei Liu, Xiangnan He, Huanbo Luan, and Tat-Seng Chua. 2016. Discrete collaborative filtering. In *SIGIR*. 325–334.

[38] Shijie Zhang, Hongzhi Yin, Qinyong Wang, Tong Chen, Hongxu Chen, and Quoc Viet Hung Nguyen. 2019. Inferring substitutable products with deep network embedding. *IJCAI* (2019), 4306–4312.

[39] Yan Zhang, Ivor Tsang, Hongzhi Yin, Guowu Yang, Defu Lian, and Jingjing Li. 2020. Deep Pairwise Hashing for Cold-start Recommendation. *TKDE* (2020).

[40] Yan Zhang, Hongzhi Yin, Zi Huang, Xingzhong Du, Guowu Yang, and Defu Lian. 2018. Discrete deep learning for fast content-aware recommendation. In *WSDM*.

[41] Pengpeng Zhao, Xiefeng Xu, Yanchi Liu, Ziting Zhou, Kai Zheng, Victor S Sheng, and Hui Xiong. 2017. Exploiting hierarchical structures for POI recommendation. In *ICDM*. 655–664.

[42] Xiangyu Zhao, Haochen Liu, Hui Liu, Jiliang Tang, Weiwei Guo, Jun Shi, Sida Wang, Huiji Gao, and Bo Long. 2020. Memory-efficient Embedding for Recommendations. *arXiv preprint arXiv:2006.14827* (2020).

[43] Xiangyu Zhao, Chong Wang, Ming Chen, Xudong Zheng, Xiaobing Liu, and Jiliang Tang. 2020. AutoEmb: Automated Embedding Dimensionality Search in Streaming Recommendations. *arXiv preprint arXiv:2002.11252* (2020).

[44] Ke Zhou and Hongyuan Zha. 2012. Learning binary codes for collaborative filtering. In *SIGKDD*. 498–506.

APPENDIX ON REPRODUCIBILITY

A HYPERPARAMETERS OF RULE

As discussed in Section 6.3, the hyperparameters in RULE are optimized via grid search. Table 3 provides details on the optimal parameters used in RULE, as well as their corresponding search intervals. Unless specified, we use the following hyperparameters as the default settings of RULE in our experiments.

Table 3: Hyperparameter settings.

Dataset	Hyper-parameter	Value	Search Interval
Amazon-Book	L	3	{1,2,3,4,5}
	$(N, d)^*$	(16,8)	{(4,8), (8,8), (12,8), (16,8), (20,8)}
	G	20	{5,10,15,20,50}
	λ	1×10^{-4}	{0, 1×10^{-2}, 1×10^{-4}, 1×10^{-6}}
	d_0	64	{8,16,32,64,128}
	P	20	{10,20,30,40,50}
	C	50	{10,20,30,40,50}
	S	5	{2,5,10,15,20}
Yelp2020	L	2	{1,2,3,4,5}
	$(N, d)^*$	(16,8)	{(4,8), (8,8), (12,8), (16,8), (20,8)}
	G	20	{5,10,15,20,50}
	λ	1×10^{-4}	{0, 1×10^{-2}, 1×10^{-4}, 1×10^{-6}}
	d_0	128	{8,16,32,64,128}
	P	50	{10,20,30,40,50}
	C	50	{10,20,30,40,50}
	S	5	{2,5,10,15,20}

$^*D$ is determined via $D = Nd$

B TRAINING THE PERFORMANCE ESTIMATOR

An accurate performance estimator is crucial for searching high-quality elastic embeddings. After finish learning the full user and item embeddings as described in Section 4, we build the training dataset $\mathcal{D}_{est}$ for *estimate*$(\cdot)$. To obtain $(model.EE, y_{model.EE})$ tuples, we randomly sample $\beta = 15,000$ elastic embedding compositions, and evaluate their performance on the validation set. To let RULE easily generalize to different memory budgets without retraining, the β different elastic embedding compositions are sampled without any predefined memory budget. Then, each sampled *model.EE* will be evaluated on the same set of $\mu = 1,000$ users from the validation set to obtain $y_{model.EE}$. To measure the performance of each *model.EE*, the performance metric we choose is Recall@100, however this is fully customizable for other purposes (e.g., using AUC for click-through rate prediction). Note that we evaluate sampled elastic embeddings on μ users instead of all users because this will substantially reduce the time consumption, while still ensuring each resulted $y_{model.EE}$ is a high-quality indicator on the recommendation performance. With the randomizer (i.e., *random*$(\cdot)$) defined in Algorithm 1, the construction process of $\mathcal{D}_{est}$ is described in Algorithm 3.

With the training set $\mathcal{D}_{est}$, we train our performance estimator with the loss defined in Eq.(6). At the initial training stage, we perform two-fold cross-validation on $\mathcal{D}_{est}$ when determining its hyperparameters. After the hyperparameters are set, we retrain the estimator with the full set $\mathcal{D}_{est}$ until the loss converges.

Algorithm 3 Construction Process of $\mathcal{D}_{est}$

```
1: Input: $N$, $G$, $\mu$, $\beta$
2: Output: $\mathcal{D}_{est}$
3: $\mathcal{D}_{est} \leftarrow \varnothing$;
4: for $b \leftarrow 1, 2, \cdots, \beta$ do
5:     draw $\widetilde{M}$ uniformly at random from $\{G, G+1, ..., NG\}$;
6:     $model.EE \leftarrow random(\widetilde{M}, N, G)$;
7:     $y_{model.EE} \leftarrow$ evaluate model.EE with Recall@100 on the same $\mu$ validation users;
8:     $\mathcal{D}_{est} \leftarrow (model.EE, y_{model.EE})$;
9: end for
10: return $\mathcal{D}_{est}$
```

C ITEM SEGMENTATION STRATEGIES

By default, RULE segments all items randomly into G groups. In Section 6.4, we further testify two more item segmentation strategies, i.e., popularity-based and clustering-based segmentation. We present more implementation details as follows.

Popularity-based Segmentation. It is widely acknowledged that item popularity plays an important role in recommendation [20], which motivates our practice of treating items with varied popularity discriminatively. Intuitively, the embedding size determines the capacity to encode information. As popular items tend to have abundant interaction records for training, a larger embedding dimension (i.e., more embedding blocks) is helpful for comprehensively capturing contexts associated with them. On the contrary, less embedding blocks are needed when representing long-tail items that offer limited predictive signals. So, we sort all items $v_j \in \mathcal{V}$ in descending order according to their popularity (i.e., times interacted by different users), such that v_1 and $v_{|\mathcal{V}|}$ are respectively the most and least popular items. Then, G item groups can be formed by evenly segmenting the sorted item set.

Clustering-based Segmentation. As the search space needs to be constructed only after all items' full embeddings $\mathbf{v}_j = [\mathbf{e}_{j(1)}, \mathbf{e}_{j(2)}, ..., \mathbf{e}_{j(N)}]$ are learned, another natural strategy for segmenting item groups is to cluster items based on their finely trained full embeddings. Specifically, we leverage k-Means that is a scalable and well-established algorithm to map all items into G clusters. With principal component analysis (PCA), we first convert all items' full embeddings into 2-dimensional vectors. Then, by setting $k = G = 20$ and applying k-Means on all 2-dimensional item embeddings, we can obtain 20 clusters, which are treated as item groups in RULE. Note that we apply PCA prior to k-Means to facilitate our 2-dimensional visualization in Figure 3, and this step is optional for normal use. Essentially, items falling in the same cluster are more likely to exhibit close properties, making it ideal for them to share the same elastic embedding compositions.