Automated Embedding Size Search in Deep Recommender Systems

Haochen Liu*
Michigan State University
liuhaoc1@msu.edu

Xiangyu Zhao*
Michigan State University
zhaoxi35@msu.edu

Chong Wang
Bytedance
chong.wang@bytedance.com

Xiaobing Liu
Bytedance
will.liu@bytedance.com

Jiliang Tang
Michigan State University
tangjili@msu.edu

ABSTRACT

Deep recommender systems have achieved promising performance on real-world recommendation tasks. They typically represent users and items in a low-dimensional embedding space and then feed the embeddings into the following deep network structures for prediction. Traditional deep recommender models often adopt uniform and fixed embedding sizes for all the users and items. However, such design is not optimal in terms of not only the recommendation performance and but also the space complexity. In this paper, we propose to dynamically search the embedding sizes for different users and items and introduce a novel embedding size adjustment policy network (ESAPN). ESAPN serves as an automated reinforcement learning agent to adaptively search appropriate embedding sizes for users and items. Different from existing works, our model performs hard selection on different embedding sizes, which leads to a more accurate selection and decreases the storage space. We evaluate our model under the streaming setting on two real-world benchmark datasets. The results show that our proposed framework outperforms representative baselines. Moreover, our framework is demonstrated to be robust to the cold-start problem and reduce memory consumption by around 40%-90%. The implementation of the model is released[1].

CCS CONCEPTS

• **Information systems** → **Recommender systems**; • **Computing methodologies** → *Reinforcement learning*; Neural networks.

KEYWORDS

Recommender System; AutoML; Embedding

ACM Reference Format:
Haochen Liu, Xiangyu Zhao, Chong Wang, Xiaobing Liu, and Jiliang Tang. 2020. Automated Embedding Size Search in Deep Recommender Systems. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '20), July 25–30, 2020, Virtual Event, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3397271.3401436

*Both authors contributed equally to this research.
[1]https://github.com/zgahhblhc/ESAPN

SIGIR '20, July 25–30, 2020, Virtual Event, China

ACM ISBN 978-1-4503-8016-4/20/07…$15.00
https://doi.org/10.1145/3397271.3401436

1 INTRODUCTION

Recommender systems have been widely deployed in various commercial scenarios from e-commerce websites, music and video platforms to social media [11]. They model the user-item relationships and seek to predict users' preference for items [18]. Recently, deep learning based recommender models [14, 24] have attracted considerable interests from the research community thanks to their excellent ability to learn the feature representations of users and items as well as modeling the non-linear relationships between users and items [26]. Deep recommender systems consist of two key components: a representation layer and an inference layer. The former learns to map discrete user and item identifiers into real-value embedding representations and the latter takes the embedding representations as input to calculate the prediction results for specific recommendation tasks. A lot of endeavors have been conducted on designing various network architectures for the inference layer to improve the performances of recommender systems while the representation layer remains not well-studied. However, representation learning plays an important role in the success of recommender systems [4, 15, 17]. Learned embedding representations can effectively represent categorical features like user and item identifiers in a low-dimensional embedding space. They preserve the valuable features of the users and items. With them, recommender systems can make a more accurate inference and then boost their performance. Meanwhile, as a part of the whole recommendation framework, the embedding representations are the key model parameters. They are learned jointly with other parts to better adapt to specific recommendation tasks and even show more important impacts on model performance than other parts [7]. Hence, designing a suitable representation layer is important for deep recommender systems.

Traditional deep recommender systems adopt uniform and fixed embedding sizes for all users and items in the representation layer. However, this design could be not optimal for real-world recommendation problems. We propose to dynamically search the embedding sizes for different users and items. The reasons are as follows. First, in a recommendation problem, different users and items have highly varied frequencies. For users/items with high frequency, high-dimensional embeddings can achieve better performances

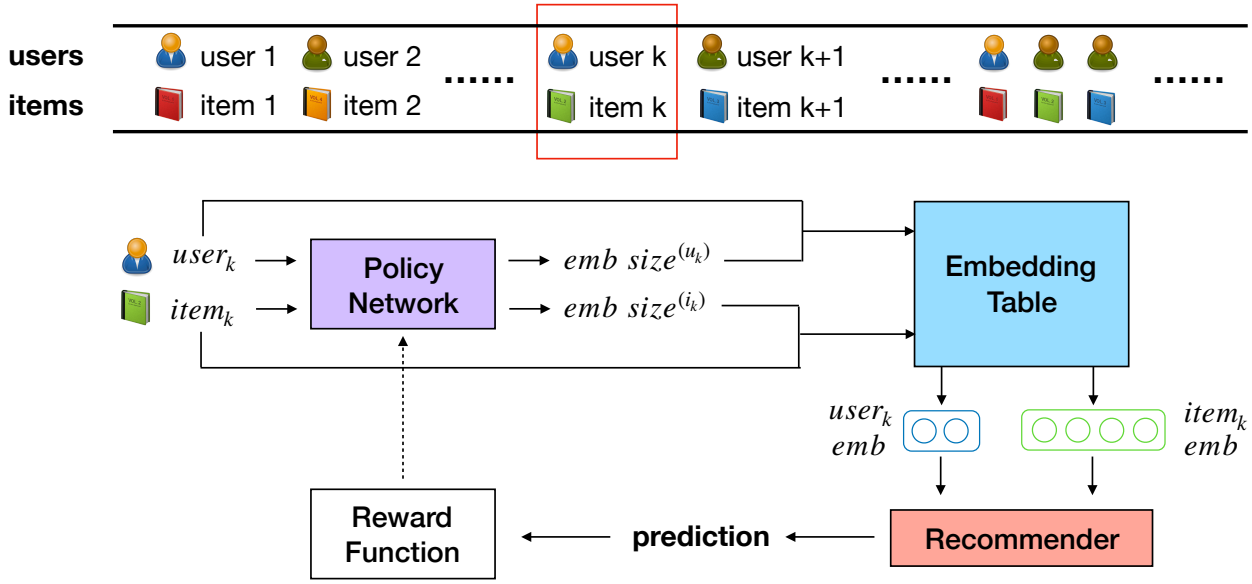

Figure 1: An illustration of the overview of the proposed framework.

since they provide sufficient parameters and capacities to encode the features of the users/items [28]. Moreover, low-dimensional embeddings are more suitable for users/items with low frequency. With little data, low-dimensional embeddings can be well-trained but high-dimensional embeddings may lead to overfitting due to over-parameterization [9]. Thus, uniform embedding sizes for users and items limit the performance of the model. Second, in real-world recommendations, the frequency of a user/item changes dynamically. The embedding size of a user/item should be adjusted along with the change of its frequency so that its features can be encoded more effectively. Fixed-size embeddings are incompetent to deal with users and items whose frequencies change dynamically. Third, given that there are typically a huge number of users and items in a recommendation task, using embeddings with uniform and fixed dimensions for all users/items needs much memory [9]. By selecting appropriate embedding sizes for different users and items, we can save the storage space while maintaining the performance of the model.

To dynamically search the embedding sizes for different users and items, we face the following challenges. First, a recommender system often has a massive amount of users and items, thus it's difficult if possible to search the embedding size for each of them manually. Second, although AutoML methods such as differentiable architecture search (DARTS) [12] have been proposed to automatically search neural network architectures, they cannot be applied directly since dynamically searching the embedding sizes is a non-differentiable operation, which cannot be directly learned by gradient backpropagation. Moreover, although some previous works [28] tried to make this operation differentiable by using a soft selection of embeddings of different sizes, they will increase the storage space and fail to select one embedding size while eliminating the influence of other embedding sizes. To address the above challenges, in this work, we propose a novel **E**mbedding **S**ize **A**djustment **P**olicy **N**etwork (**ESAPN**), which serves as an automated reinforcement learning (RL) agent to dynamically search the embedding sizes for users and items in deep recommender systems under the streaming scenario. In the policy network, we apply hard selection on different embedding sizes so that we can accurately select one embedding size at each time. Accordingly, we also propose an optimization method to update the whole framework, i.e., the policy network and the recommender system. We conduct extensive experiments on two public recommendation datasets and the results show that our proposed framework outperforms several competitive baselines. Besides, our framework is shown to perform better in the cold start problem and reduce storage memory significantly.

The remainder of this paper is organized as follows. First, we will detail the proposed framework, including the basic recommendation model and the embedding size adjustment policy network in Section 2. Then, we carry out our experimental setups and results with discussions in Section 3. Next, in Section 4, we review related work. Finally, Section 5 concludes the work with possible future research directions.

2 THE PROPOSED FRAMEWORK

In this section, we will detail our proposed framework that dynamically adjusts the embedding size of users and items in the streaming recommendation. We will first illustrate the overview of the whole framework, then introduce the deep recommendation model, and next present the policy network for adjusting embedding size. Afterward, we detail the method to optimize our proposed framework.

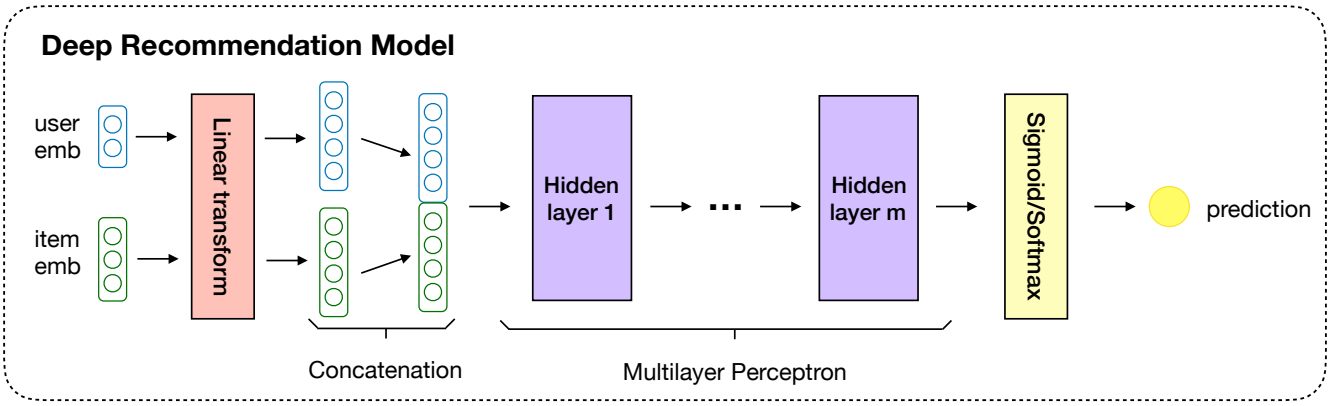

Figure 2: An illustration of the deep recommendation model.

2.1 Overview

In this work, we investigate the task of streaming recommendation. In the streaming setting, user-item transactions are collected continuously from a steady stream [2]. Given each mini-batch of user-item transactions, the recommendation model first makes predictions of the users' preferences for items. The prediction results are recorded for evaluation. Then, based on the ground-truth labels, the recommendation model is optimized by minimizing the prediction errors. Thus, a streaming recommendation model conducts inference and training alternatively on each mini-batch of data that appear in the stream, which is different from the traditional recommendation settings where data is explicitly split to training and test sets. In the streaming setting, the frequencies of users and items change dynamically over time, so as discussed in Section 1, it is desired to find a way to automatically adjust the embedding sizes of the users and items accordingly.

Our framework consists of two core components: (i) the deep recommendation model and (ii) the embedding size adjustment policy network. The deep recommendation model takes user-item pairs as input to predict the user's preference for the item. The policy network serves as an RL agent to adaptively adjust the embedding sizes of the users and items throughout the streaming recommendation process so that the recommendation model can be better trained and its performance will be boosted. An overview of the proposed framework is illustrated in Figure 1. As shown in the figure, given the k-th user-item pair in the stream, the policy network first selects the appropriate embedding sizes of both of them. Next, we look up the embedding table to obtain the embeddings of the user and the item with proper sizes, respectively. Afterward, the embeddings are fed into the deep recommendation model to calculate the prediction results. Combined with the ground-truth label, the recommendation model will be updated. What's more, the predictions will pass through a reward function to compute a reward, which will serve as a signal to update the policy network. Next, we will detail the deep recommendation model and the policy network.

2.2 Deep Recommendation Model

The deep recommendation model is illustrated in Figure 2. Given a pair of user u and item i, the model predicts the preference of the user for the item, which can be a binary 0-1 label indicating whether the user will purchase it or not or a plural categorical rating measuring the degree of preference. We investigate both two cases in this work. We refer to the former as the binary classification task and the latter as the multiclass classification task.

Suppose we have n candidate embedding sizes for both of users and items $D = \{d_1, d_2, \ldots, d_n\}$ where $d_1 < d_2 < \cdots < d_n$ [2]. Given the user-item pair (u, i), the policy network adaptively chooses the $j^{(u)}$-th and the $j^{(i)}$-th embedding sizes for the user and the item, respectively. From the embedding table, we get the embeddings of the user $\mathbf{e}^{(\mathbf{u})}$ with dimension $d_{j^{(u)}}$ and the embedding of the item $\mathbf{e}^{(\mathbf{i})}$ with dimension $d_{j^{(i)}}$. Since the input dimension of the following inference layer is fixed, we need to unify the sizes of the input embeddings. Thus, we first define a series of linear transformations. The transformations map embeddings into the adjacent larger dimension (e.g. $d_1 \rightarrow d_2$, $d_2 \rightarrow d_3$, etc.):

$$\begin{aligned} \mathbf{e_2} &= W_{1\rightarrow 2}\mathbf{e_1} + b_{1\rightarrow 2} \\ \mathbf{e_3} &= W_{2\rightarrow 3}\mathbf{e_2} + b_{2\rightarrow 3} \\ &\cdots \\ \mathbf{e_n} &= W_{n-1\rightarrow n}\mathbf{e_{n-1}} + b_{n-1\rightarrow n} \end{aligned} \tag{1}$$

where $\mathbf{e_k}$ is any embedding with dimension d_k ($k = 1, 2, \ldots, n$). $W_{k-1\rightarrow k}$ and $b_{k-1\rightarrow k}$ are learnable weight and bias parameters. Note that we have two different sets of weight and bias parameters for users and items. For simplicity, we don't explicitly distinguish them. In the implementation, the inference layer requires user/item embeddings of size d_n; thus given user embedding $\mathbf{e}^{(\mathbf{u})} \in \mathbb{R}^{d_{j^{(u)}}}$ and item embedding $\mathbf{e}^{(\mathbf{i})} \in \mathbb{R}^{d_{j^{(i)}}}$, we map them into the largest dimension d_n through consecutive linear transformations to get

[2] Users and items can have two different sets of embedding sizes. For simplicity, we assume that they share the same set in this work.

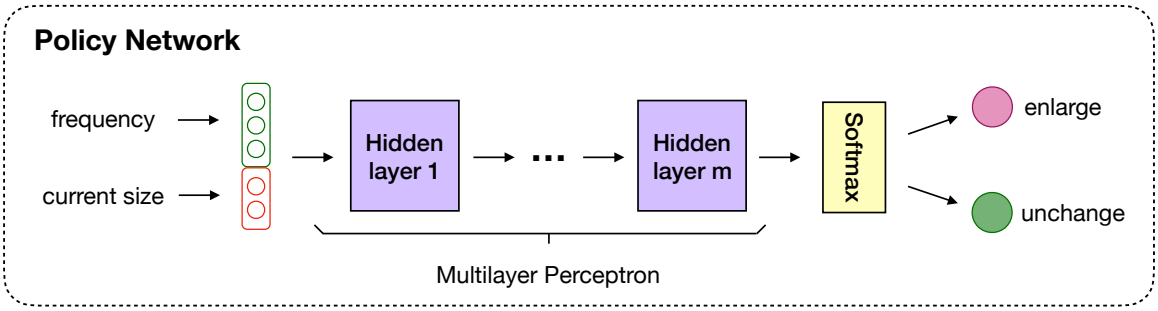

Figure 3: An illustration of the embedding size adjustment policy network.

the transformed embeddings $\hat{\mathbf{e}}^{(\mathbf{u})} \in \mathbb{R}^{d_n}$ and $\hat{\mathbf{e}}^{(\mathbf{i})} \in \mathbb{R}^{d_n}$:

$$\mathbf{e}^{(\mathbf{u/i})} \xrightarrow[+b_{j(u/i)\to j(u/i)+1}]{\times W_{j(u/i)\to j(u/i)+1}} \cdots \xrightarrow[+b_{n-1\to n}]{\times W_{n-1\to n}} \hat{\mathbf{e}}^{(\mathbf{u/i})} \in \mathbb{R}^{d_n}$$

Following [28], we perform batch normalization with Tanh activation function on the transformed embeddings. Next, the concatenation of them $[\hat{\mathbf{e}}^{(\mathbf{u})} : \hat{\mathbf{e}}^{(\mathbf{i})}]$ are fed into the inference layer, which comprises multilayer perceptrons (MLP) with m hidden layers:

$$\begin{aligned} h_1 &= \tanh(W_1[\hat{\mathbf{e}}^{(\mathbf{u})} : \hat{\mathbf{e}}^{(\mathbf{i})}]) + b_1 \\ h_2 &= \tanh(W_2 h_1) + b_2 \\ &\cdots \\ h_m &= \tanh(W_m h_{m-1}) + b_m \end{aligned}$$

The hidden state of the last layer h_m is then used to calculate the prediction results through an activation function, which depends on the specific task. For the binary classification task, it will be the Sigmoid function while it is chosen to be the Softmax function for the multiclass classification task.

2.3 The Policy Network

The policy network serves as an RL agent that dynamically adjusts the embedding sizes of users and items. It maintains the current embedding sizes of all the users and items. Given the state of a user or an item, it takes an action to adjust the embedding size following a policy. It interacts with the environment by feeding the adjusted embeddings into the recommendation model and receives the corresponding reward based on the prediction. Afterward, the policy is updated to maximize the reward. Next, we will introduce the environment, state, policy, action, and reward in detail.

2.3.1 Environment. The environment is the deep recommendation model. Given a pair of user and item along with their adjusted embedding sizes, it returns the prediction result.

2.3.2 State. The state $s = (f, e)$ is the combination of the frequency f and the current embedding size e of a given user or item. Based on the frequency and the current embedding size of the user/item, the policy network decides how to adjust the embedding size.

2.3.3 Policy and Action. We have two policy networks of the same structure for users and items, respectively. Next, we take the user policy network as an example to illustrate the policy network in Figure 3. The user policy network takes the state s (the frequency and the current embedding size of a user) as input to predict an action a indicating how we should adjust its embedding size. Given the frequency and the current embedding size of a user, we first map the frequency (an integer value) into an embedding representation and convert the current embedding size into a one-hot vector with dimension n, where n is the number of candidate embedding sizes. Afterward, these two vectors are concatenated and fed into multilayer perceptrons (MLP) with m hidden layers. Finally, the last hidden state h_m passes through a Softmax layer to predict the probabilities of choosing two actions: (i) enlarge and (ii) unchange. "Enlarge" means to enlarge the embedding size of the given user to the next larger candidate embedding size, such as from d_k to d_{k+1}, while "unchange" indicates to keep the current embedding size. The reason why we only choose to increase and keep the embedding dimensions is that the frequencies of users are always monotonically increasing in the data stream. We only allow the embedding size to change to the adjacent candidate size rather than even larger ones at each time since the frequencies of users change gradually and the change of embedding sizes should be consistent with this fashion. If we choose to enlarge the embedding size of a given user, we initialize its embedding with the next larger size by performing the learned linear transformation in Eq 1. For example, when we choose to enlarge the embedding size of a user from d_k to d_{k+1}, the new embedding $\mathbf{e_{k+1}}$ is initialized from the old embedding $\mathbf{e_k}$ through $\mathbf{e_{k+1}} = W_{k\to k+1}\mathbf{e_k} + b_{k\to k+1}$. In this way, the information in the learned old embedding can be inherited and the new embedding is not initialized from scratch.

Advantages of Hard Selection. The work [28] changes the embedding sizes of users/items by adjusting the weights of various candidate embedding sizes and calculate the embedding to use as a weighted sum of all the candidate embeddings at each time, which is known as soft selection. In contrast, our method selects the embedding of one size at each time, which is known as hard selection. Hard selection has two benefits. First, the hard selection focuses on the embedding of one size at each time and can eliminate the interference of the redundant information from the embeddings of other sizes. Adopting embeddings of one size at each time also

makes the training more efficient and leads to better performance, which will be later verified in Section 3.6. Second, hard selection requires less memory. Soft selection needs to maintain the embeddings of all the candidate sizes while hard selection only needs to dynamically maintain the embeddings of the current size and once we decide to enlarge the size of an embedding, the old embedding is abandoned. It will be later demonstrated in Section 3.7 that our method is significantly more efficient in terms of memory.

2.3.4 Reward. We choose suitable embedding sizes for users and items to boost the performance of the recommendation model. So the policy network should be optimized by maximizing a reward that reflects the prediction ability of the recommendation model. We define the rewards based on the errors of prediction results made by the recommendation model. Given a transaction between user u and item i and their selected embedding sizes, the recommendation model makes predictions and returns a loss L based on the ground-truth label. The loss functions are defined later in Section 2.4.1 for the binary and multiclass classification tasks, respectively. We use two first-in-first-out (FIFO) queues to maintain the latest T prediction losses $L^{(u)} = (L_1^{(u)}, \ldots, L_T^{(u)})$ for the user u and $L^{(i)} = (L_1^{(i)}, \ldots, L_T^{(i)})$ for the item i respectively, where $L_t^{(u/i)}$ indicates the t-th to the last prediction loss on the user u / the item i. Given the current prediction loss L of the user u and the item i, the rewards for the user policy network and the item policy network are defined as the difference between the average of the last T prediction losses and the current loss, respectively:

$$R^{(u)} = \frac{1}{T}\sum_{t=1}^{T} L_t^{(u)} - L \tag{2}$$

$$R^{(i)} = \frac{1}{T}\sum_{t=1}^{T} L_t^{(i)} - L$$

The reward can be viewed as the reduction of the current prediction loss compared to previous losses. Since the prediction losses of one user can be highly varied when it interacts with different items and vice versa, we use the average of the last T losses as the base instead of the last one loss.

2.4 An Optimization Method

In this subsection, we will introduce how to optimize the deep recommendation model and the policy network, respectively. Next, we will detail how to train the whole framework under the streaming setting.

2.4.1 Deep Recommendation Model. Suppose that the deep recommendation model is parameterized by Θ. It is optimized in a supervised manner by minimizing the difference between prediction results and ground truths. In the binary classification task, the model is trained by minimizing the mean squared error (MSE) loss. Given a mini-batch of user-item transactions $\{u_k, i_k\}_{k=1}^N$ along with the corresponding ground truth labels $\{y_k\}_{k=1}^N$, where y_k is 0 or 1, the recommendation model predicts the probabilities $\{\hat{y}_k\}_{k=1}^N$ of the labels to be 1. The MSE loss is defined as:

$$MSE(\Theta) = \frac{1}{N}\sum_{k=1}^{N}(y_k - \hat{y}_k)^2 \tag{3}$$

where N is the size of a mini-batch.

In the multiclass classification task, the recommendation is optimized on the cross-entropy (CE) loss. Suppose $\{c_k\}_{k=1}^N$ are the ground-truth labels of the mini-batch $\{u_k, i_k\}_{k=1}^N$. The CE loss is defined as

$$CE(\Theta) = -\frac{1}{N}\sum_{k=1}^{N}\sum_{m=0}^{M-1} y_{km}\log p_{km} \tag{4}$$

where N is the size of a mini-batch, M is the number of classes and y_{km} is 1 when (u_k, i_k) belongs to class m and 0 otherwise. p_{km} indicates the probability predicted by the recommendation model that (u_k, i_k) belongs to class m.

2.4.2 Policy Network. Suppose the user policy network and the item policy network are parameterized by $\Phi^{(u)}$ and $\Phi^{(i)}$, respectively. As discussed in Section 2.3, we formulate the optimization of the policy network as a reinforcement learning problem. We take the user case as an example and omit the superscript (u) for simplicity. With the reward function defined in Eq 2, the objective function that the policy network aims to maximize can be formulated as:

$$J(\Phi) = \mathbb{E}_{a\sim\Phi(a|s)}R(a|s) \tag{5}$$

where s is the state and a is the action. In practice, it's not easy to obtain the accurate value of $J(\Phi)$ in Eq 5. Various methods have been proposed to estimate its value and gradient. We adopt Monte-Carlo sampling to estimate $\nabla_\Phi J(\Phi)$ and apply the REINFORCE algorithm [23] to optimize the objective in Eq 5. Specifically,

$$\begin{aligned}\nabla_\Phi J(\Phi) &= \sum_a R(a|s)\nabla\Phi(a|s) \\ &= \sum_a R(a|s)\Phi(a|s)\nabla\log\Phi(a|s) \\ &= \mathbb{E}_{a\sim\Phi(a|s)}[R(a|s)\nabla\log\Phi(a|s)] \\ &\approx \frac{1}{N}\sum_{i=1}^{N} R(a|s)\nabla\log\Phi(a|s)\end{aligned} \tag{6}$$

where N is the number of samples. In our implementation, we set sample number $N = 1$ to improve the computational efficiency. With the obtained gradient $\nabla_\Phi J(\Phi)$, the parameters of the policy network can be updated as follows:

$$\Phi \leftarrow \Phi + \alpha_P \nabla_\Phi J(\Phi) \tag{7}$$

where α_P is the learning rate.

2.4.3 The Overall Optimization Algorithm. In this subsection, we detail our AutoML-based algorithm for optimizing the whole framework. The algorithm is presented in Algorithm 1. In the streaming recommendation, user-item transaction data are collected from a steady stream S. We iterate batches of steady data from the stream to optimize the framework. Inspired by ENAS [16], in each iteration, there are two stages: (i) the policy network optimization stage (lines 2-6), where we update the policy network Φ using sampled

Algorithm 1: An optimization method for the whole framework.

Input: User-item transaction data stream $\mathbf{S} = \{D_1, \ldots, D_N\}$ which consists of N batch of data $D_i = \{(u_k, i_k, y_k)\}_{k=1}^{n}$, initial recommendation model Θ and policy network Φ, hyper-parameters α_P and α_R.

Output: a well-trained recommendation model Θ^* and a well-trained policy network Φ^*

1 **repeat**
2 Sample a validation batch $V = \{(u_k, i_k, y_k)\}_{k=1}^{n}$ from the history transaction data
3 Sample action $\hat{a}_V \sim \Phi(\cdot|s_V)$
4 Temporarily adjust the embedding sizes of $(u_k, i_k)_{k=1}^{n}$ according to $\hat{a}_V$
5 Input $(u_k, i_k)_{k=1}^{n}$ into the recommendation model Θ and calculate the rewards by Eq. 2
6 Update the policy network Φ by Eq. 7
7 Collect the current batch of the transaction data $D = \{(u_k, i_k, y_k)\}_{k=1}^{n}$ in the stream $\mathbf{S}$
8 Select action $\hat{a}_D$ from $\Phi(\cdot|s_D)$
9 Permanently adjust the embedding sizes of $(u_k, i_k)_{k=1}^{n}$ according to $\hat{a}_D$
10 Input $(u_k, i_k)_{k=1}^{n}$ into the recommendation model Θ and update Θ by Eq. 3 or Eq. 4
11 **until** *The recommendation model* Θ *and the policy network* Φ *converge.*;

validation data and (ii) the model optimization stage (lines 7-10) where the recommendation model Θ is updated using the training data from the stream.

Specifically, in the policy network optimization stage, we first sample a batch of validation data V from the previous stream data, i.e., the history data (line 2). Based on the states of the users and items $(u_k, i_k)_{k=1}^{n}$ in the validation data V, we sample the actions from the policy network Φ (line 3) and adjust the embedding sizes of them accordingly (line 4). Note that we just change the embedding sizes temporarily. We then input the user-item pairs into the recommendation model Θ to compute the rewards (line 5). Next, the original embedding sizes are recovered. Based on the rewards, the policy network Φ is updated by Eq 7 (line 6). In the model optimization stage, we use the current batch of stream data D to update the recommendation model Θ (line 7). Given the state s_D, we select the action $\hat{a}_D$ with the highest probabilities predicted by the policy network Φ (line 8) and permanently adjust the embedding sizes accordingly (line 9). Next, we optimize the recommendation model Θ by minimizing the MSE loss in Eq 3 or the CE loss in Eq 4 (line 10). The above procedure is repeated until the recommendation model Θ and the policy network Φ converge.

3 EXPERIMENT

To validate the performance of our proposed framework, we conduct extensive experiments on two large-scale real-world recommendation datasets. Through the experiments, we seek to answer the following three questions:

- How does our proposed framework perform compared with the traditional deep recommendation models with fixed embedding sizes? In other words, does dynamically adjust embedding sizes boost the performance on recommendation tasks?
- How does our framework perform compared with other representative AutoML-based recommendation models?
- Can our method alleviate the cold-start problem and reduce memory consumption?

In this section, we will first introduce the experimental settings including the datasets, the baselines and the implementation details. Then we will present the overall performance comparison, the performance comparison with frequency and memory consumption comparison with discussions.

3.1 Datasets

Our proposed method is evaluated on the following two popular public recommendation benchmark datasets:

- MovieLens 20M Dataset[3] (ml-20m): A movie rating dataset collected from the personalized movie recommendation website MovieLens[4]. The dataset contains 20 million 5-star ratings and 465,000 free-text tags applied to 27,278 movies provided by 138,493 users. The users are randomly selected and every selected is guaranteed to have rated at least 20 movies.
- MovieLens Latest Datasets[5] (ml-latest): A newly released movie rating dataset collected from the same website. The dataset contains 27 million 5-star ratings and 1.1 million tags applied to 58,098 movies from 283,228 users. In this dataset, the users are also randomly selected and all the selected users had rated at least 1 movie.

3.2 Implementation Details

In this subsection, we present the implementation details of our recommendation model and the embedding size adjustment policy network such as the selection of model structures and the setting of the hyper-parameters.

First, we set six candidate embedding sizes $D = \{2, 4, 8, 16, 64, 128\}$ for both the users and items. Second, for the recommendation model, we adopt one hidden layer with a size of 512. It is trained via the Adam [10] optimizer with an initial learning rate of 0.001. Third, for the policy network, we also adopt one hidden layer with a size of 512. We use the last $T = 5$ prediction losses to compute the reward. The policy network is trained by Adam optimizer with an initial learning rate of 0.0001. The whole framework is trained on mini-batches with a size of 500.

In the streaming recommendation setting, there is no explicitly split for training and test sets. Given a mini-batch from the data stream, the recommendation model is first tested on it and then trained on it. The model is evaluated and trained alternatively from the beginning to the end. Hence, for both of the two datasets, we artificially choose the last 5 million transactions as the "test set" and report the performance on it. Note that when evaluating the

[3] https://grouplens.org/datasets/movielens/20m/
[4] https://movielens.org/
[5] https://grouplens.org/datasets/movielens/latest/

model on the "test set", the model keeps being updated. All baseline models follow the same setting as our model.

3.3 Baselines

We compare our proposed framework with the following three baseline methods:

- FIXED: The original recommendation model with embeddings of uniform and fixed sizes. We adopt the embedding size of 128 for all the users and items.
- DARTS [12]: A traditional DARTS method that adjusts embedding sizes by learning weights for the six candidate embedding sizes $\{2, 4, 8, 16, 64, 128\}$ for each user or item. Then the embedding of a user or an item is represented as the weighted sum of the embeddings of the six candidate sizes.
- AutoEmb [28]: A DARTS-based method that trains a controller to dynamically assign weights for the six candidate embedding sizes $\{2, 4, 8, 16, 64, 128\}$ based on the frequency of a given user or item. The embeddings are also represented by soft selection.

3.4 Tasks and Evaluation Metrics

We evaluate the baselines and our model on a binary classification task and a multiclass classification task.

- Binary Classification Task: We transform the 5-star ratings to binary labels where ratings of 4 and 5 are viewed as positive (i.e. 1) and the rest as negative (i.e. 0), and ask the recommendation model to predict the correct label. We use predictive probability 0.5 as the threshold to assign the label. We measure the performance of the models by the classification accuracy and the mean-squared-error loss.
- Multiclass Classification Task: We view the 5-star ratings as 5 classes and ask the model to predict the correct rating. In this task, we evaluate the models according to the classification accuracy and the cross-entropy loss.

3.5 Overall Performance Comparison

In Table 1, we show the overall performances of the baselines as well as our model on the two datasets. Based on Table 1, we make the following observations:

- First, our model outperforms the baseline FIXED with a significant margin in terms of all metrics on all the datasets and tasks. It verifies that by adopting dynamically changing embedding sizes for users and items, we can train a better recommendation model with boosted performance.
- Second, the baselines DARTS and AutoEmb achieve better results than FIXED. DARTS and AutoEmb perform a soft selection on a set of embeddings of various sizes with dynamic weights, they realize the dynamic adjustment of embedding sizes to some extent and get the performances improved. What's more, AutoEmb outperforms DARTS. Different from DARTS, AutoEmb introduces a controller to adjust the weights of different embedding sizes based on the frequencies of the users and items. The improvement demonstrates that the selection of a reasonable embedding size is related to the frequency and it can serve as evidence that helps us to adjust the embedding sizes.
- Third, our model outperforms all the baseline models. The embeddings used in DARTS and AutoEmb are soft combinations of embeddings of different sizes. Although the models can learn to assign a high weight on the appropriate size, the other embedding sizes still get non-zero weights and contribute to the final embeddings, which results in inaccurate predictions. On the contrary, our model performs hard selection, which accurately chooses one embedding size at each time and avoids the influence of other embedding sizes. This design makes the model easier to train and enables the model to make more accurate inferences as well.

Table 1: Overall performance comparison.

ml-20m				
Models	**Binary**		**Multiclass**	
	Accuracy (%)	**MSE Loss**	**Accuracy (%)**	**CE Loss**
FIXED	72.13	0.1845	49.45	1.1517
DARTS	72.18	0.1836	49.85	1.1423
AutoEmb	72.27	0.1828	49.94	1.1399
ESAPN	**72.98**	**0.1785**	**51.10**	**1.1126**
ml-latest				
Models	**Binary**		**Multiclass**	
	Accuracy (%)	**MSE Loss**	**Accuracy (%)**	**CE Loss**
FIXED	72.13	0.1845	50.01	1.1414
DARTS	72.22	0.1834	50.46	1.1304
AutoEmb	72.36	0.1823	50.47	1.1311
ESAPN	**72.88**	**0.1790**	**51.11**	**1.1147**

3.6 Performance Comparison with Frequency

In this section, we compare the performances of our model with baselines on users/items with varied frequencies to further understand the superiority of our model. In Figure 4, we show the average performances of the models on users/items with frequencies from 1 to 100. We show the accuracy and MSE loss for the binary classification task and the accuracy and CE loss for the multiclass classification task on the ml-20m dataset. From the figures, we can see that: (i) All the models perform better on users/items with higher frequencies than those with lower frequencies. It is easy to explain since there are more training examples of frequent users/items and we are aware of more information about them. (ii) Our model achieves better performances than the baselines. (iii) If the frequencies of the users/items are lower, the improvements in our model compared with the baselines are larger. The baseline methods with a soft selection of embedding sizes suffer from the cold-start problem. They adopt the combination of embeddings of various sizes as an embedding, which is hard to train when the user or the item appears few times. Moreover, they could hardly assign all the weights to small embedding sizes for infrequent users and items like hard selection, which inevitably brings the noise from other embedding sizes. On the contrary, our model learns to

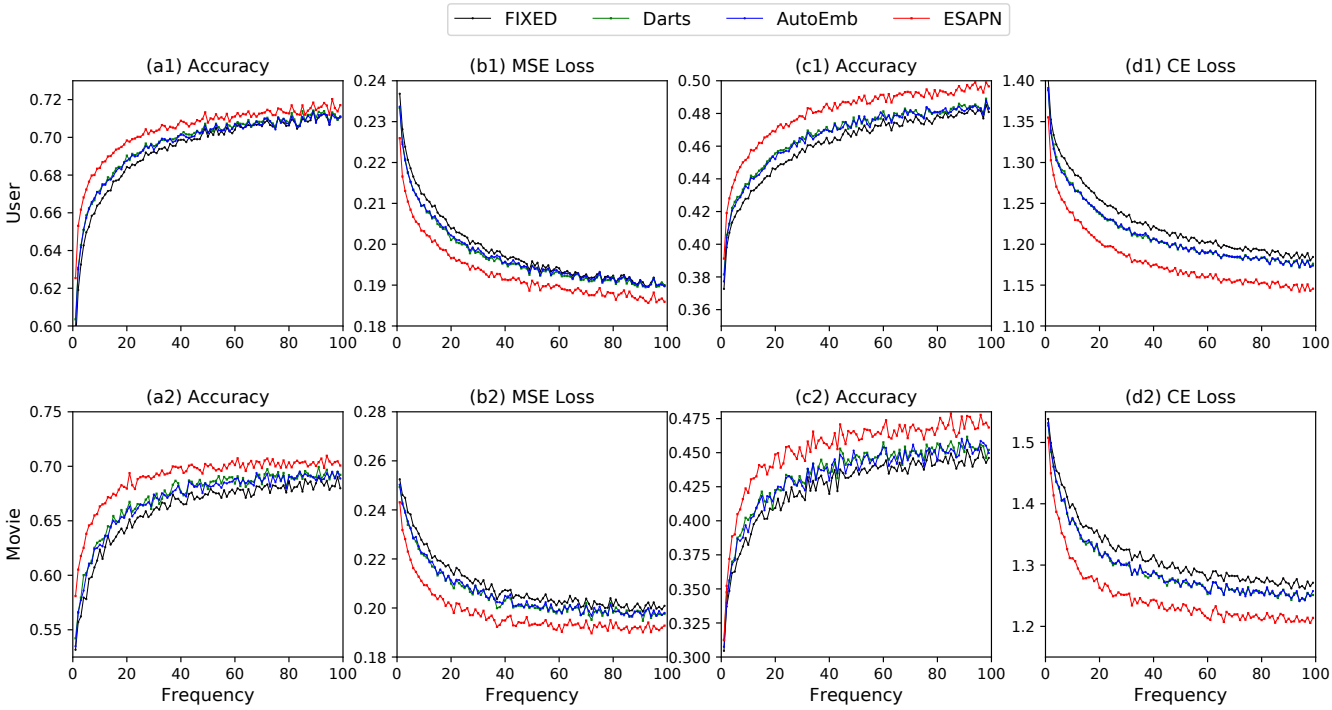

Figure 4: Performance Comparison on users and items with varied frequencies.

adaptively assign small embedding sizes for infrequent users and items so that the cold-start problem is alleviated.

To understand how our model chooses the embedding sizes in terms of the frequencies, we draw Figure 5. In this figure, we show the average embedding sizes chosen for users/items with frequencies ranging from 1 to 50. We show the results on the ml-20m dataset in terms of the binary and multiclass classification tasks. We can see that the model adaptively selects small embedding sizes for infrequent users/items and large ones for frequent users/items. This supports the above observation – our model is more robust to the frequencies of users and items.

3.7 Memory Consumption Comparison

In the traditional recommendation models, the embeddings of users and items are uniform and fixed. Thus the storage space needed to store the embedding table grows linearly with the vocabulary size [9]. However, real-world recommender systems typically involve a huge number of users and items, which need massive storage resources. By adaptively assigning embedding sizes for different users and items, our model can reduce the unnecessary usage of high dimensional embeddings so that the storage space can be saved significantly. Note that the embedding layer often takes up most of the parameters of a deep recommender system [7]; thus reducing the embedding sizes can evidently save the memory. We show the numbers of different embedding sizes the model has assigned to different users and items at the end of the data stream in the middle section of Table 2. We illustrate the comparison of memory consumption between our model and the baseline FIXED in the right

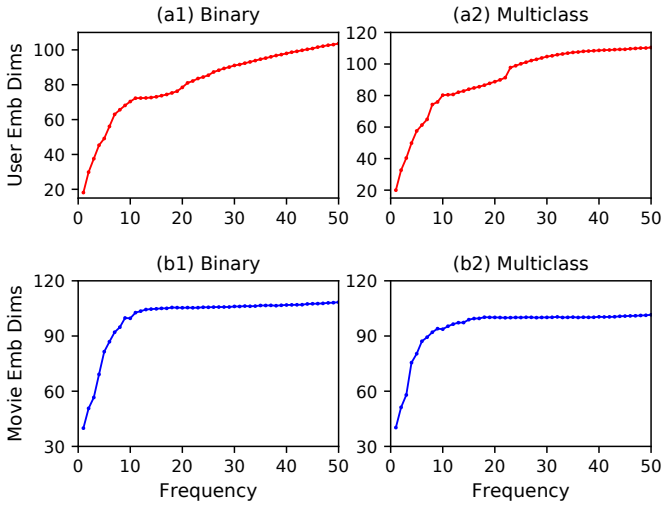

Figure 5: The average embedding sizes selected for users and items with various frequencies.

section of Table 2. The baseline FIXED uses the maximum candidate embedding size 128 for all the users and items. The column "Total Dim" indicates the total number of embedding dimensions assigned to all the users/items of our model. As a comparison, the column "FIXED" shows the total number of embedding dimensions needed in the FIXED model, that is, $\#(user) \times 128$ or $\#(item) \times 128$. The last

Table 2: Memory consumption comparison.

	2	4	8	16	64	128	Total Dim	FIXED	Ratio
user (ml-20m)	1,041	10,854	18,146	30,836	22,830	54,786	9,157,770	17,727,104	51.66%
movie (ml-20m)	17,032	216	380	623	2,237	6,790	1,060,224	3,491,584	30.37%
user (ml-latest)	62,808	83,190	31,450	22,778	31,613	51,389	9,675,448	36,253,184	26.69%
movie (ml-latest)	45,430	2,089	2,006	1,678	1,545	5,350	925,792	7,436,544	12.45%

column "Ratio" shows the ratio of total embedding dimensions of our model to these of the FIXED model. All the results are collected in the binary classification task. We can see that our model assigns low-dimensional embeddings to a lot of infrequent users and items and decreases the memory usage by around 40%-90% compared with the traditional recommendation model with uniform and fixed embedding sizes. Moreover, the AutoML-based models, DARTS and AutoEmb, assign the total embedding size of 222 to all the users and items. They occupy 1.73 times memory as the FIXED model does. Compared with these AutoML models, our model's advantage of reducing memory consumption is even more significant.

4 RELATED WORK

In this section, we go over the related works. We first review the latest studies in deep recommender systems and then discuss related works about AutoML for neural architecture search.

4.1 Deep Recommender System

Deep recommender systems enhance recommendation performance and overcome the limitations of traditional approaches [27]. We classify the existing works according to the types of deep learning techniques. Wide&Deep [5] is an MLP-based model that can handle both regression and classification tasks. The Wide part is a single perceptron layer that can be viewed as a linear model, and the Deep part is MLP that tries to capture more general and abstract representations. AutoRec [19] is an AutoEncoder based model which has two variants, item-based and user-based AutoRec, which aim to learn the low-dimension feature embeddings of users and items. Wang et al. [22] studied the influences of visual features extracted by CNNs for POI (Point-of-Interest) recommendations. The model is based on PMF to learn the interactions between visual content and latent user/location factors. GRU4Rec [6] is an RNN based method to model the sequential influence of items' transition for the session-based recommendation, where a session-parallel mini-batches algorithm and a sampling method for output are proposed to increase the training efficiency. Chen et al. [3] proposed an Attention-based collaborative filtering model with two levels of attention mechanisms, where the item-level one selects the most informative items to represent users, and the component-level one selects the most informative features from auxiliary sources for each user. Zhao et al. [29–32] proposed a series of deep reinforcement learning based recommender systems, which typically model the user-system interaction as Markov decision process, and aim to maximize the long-term reward from the environment (users). IRGAN [21] is the first GAN-based model for information retrieval, which validated the capability of GAN in web search, item recommendation and question answering tasks.

4.2 AutoML Methods for NAS

Another category related to this work is AutoML for neural architecture search. The first paper in this area is NAS [33] which leverages reinforcement learning with RNN to train lots of candidate components to convergence. Because of the high training cost, lots of follow-up works aim to develop NAS with lower resource requirements. One category is to build a large model that connects smaller model components, and select a subset of model components for each candidate architectures, so that the optimal architecture is trained in a single run. For instance, ENAS [16] sample a subset of models by a controller, while SMASH [1] generates weights for sampled networks by a hyper-network. DARTS[12] and SNAS[25] view connections as weights and optimize them via back-propagation. Luo et al. [13] project the neural architectures into embeddings, then the optimal embedding can be trained and decoded back to the final architecture. Another category is to reduce the search space. One way is to propose searching convolution cells which could be stacked repeatedly into a deep network. NASNet [34] uses transfer learning to train cells on smaller datasets, and then apply them on larger datasets. MNAS [20] proposed a search space involving hierarchical convolution cell blocks which could be searched independently to form different structures. Joglekar [8] firstly utilized NAS to optimize embedding components of recommender systems. Nevertheless, it cannot be directly deployed in the streaming recommender systems, where the frequencies of users/items are unknown in advance and highly dynamic. The work [28] investigates dynamically searching embedding sizes for users and items based on their popularity in the streaming setting. Based on DARTS, they train a controller to assign weights on various candidate embedding sizes and calculate a combined embedding for each user and item. Their method suffers from the drawbacks of soft selection such as the cold-start problem and excessive memory consumption.

5 CONCLUSION

Deep recommendation models typically use uniform and fixed embedding sizes for all users and items. This design is sub-optimal for real-world recommendation problems where the frequencies of users and items are highly varied and change dynamically. Also, it could require much more storage resources. In this paper, we propose to dynamically search embedding sizes for different users and items. We develop a novel embedding size adjustment policy network to automatically search embedding sizes for users and items in the recommendation process. It applies hard selection on embedding sizes and overcomes the drawbacks of soft selection adopted by previous works. We formulate the optimization of the policy network as a reinforcement learning problem and propose the corresponding optimization method. We evaluate the performance of

our framework on two real-world recommendation datasets and the results demonstrate the superiority of our model over competitive baselines. Finally, we show that our model effectively alleviates the cold-start problem and reduces storage space significantly.

In this paper, we investigate how to dynamically adjust the embedding sizes for users/items based on their frequencies. Other information about a user/item may also be helpful to determine an appropriate embedding size. We will attempt to incorporate other information. Moreover, the representation layer is only a part of the recommender system. As one future research direction, we are going to investigate how to design the network structure in the inference layer automatically to further boost the performance of the model.

ACKNOWLEDGMENTS

Haochen Liu, Xiangyu Zhao and Jiliang Tang are supported by the National Science Foundation (NSF) under grant number IIS-1907704, IIS-1928278, IIS-1714741, IIS-1715940, IIS-1845081 and CNS-1815636.

REFERENCES

[1] Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. 2017. Smash: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344* (2017).

[2] Shiyu Chang, Yang Zhang, Jiliang Tang, Dawei Yin, Yi Chang, Mark A. Hasegawa-Johnson, and Thomas S. Huang. 2017. Streaming Recommender Systems. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*. 381–389. https://doi.org/10.1145/3038912.3052627

[3] Jingyuan Chen, Hanwang Zhang, Xiangnan He, Liqiang Nie, Wei Liu, and Tat-Seng Chua. 2017. Attentive collaborative filtering: Multimedia recommendation with item-and component-level attention. In *Proceedings of the 40th International ACM SIGIR conference on Research and Development in Information Retrieval*. 335–344.

[4] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. 2016. Wide & Deep Learning for Recommender Systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems, DLRS@RecSys 2016, Boston, MA, USA, September 15, 2016*. 7–10. https://doi.org/10.1145/2988450.2988454

[5] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. 2016. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*. ACM, 7–10.

[6] Balázs Hidasi, Alexandros Karatzoglou, Linas Baltrunas, and Domonkos Tikk. 2015. Session-based recommendations with recurrent neural networks. *arXiv preprint arXiv:1511.06939* (2015).

[7] Manas R. Joglekar, Cong Li, Jay K. Adams, Pranav Khaitan, and Quoc V. Le. 2019. Neural Input Search for Large Scale Recommendation Models. *CoRR* abs/1907.04471 (2019). arXiv:1907.04471 http://arxiv.org/abs/1907.04471

[8] Manas R Joglekar, Cong Li, Jay K Adams, Pranav Khaitan, and Quoc V Le. 2019. Neural input search for large scale recommendation models. *arXiv preprint arXiv:1907.04471* (2019).

[9] 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. *CoRR* abs/2002.08530 (2020). arXiv:2002.08530 https://arxiv.org/abs/2002.08530

[10] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. http://arxiv.org/abs/1412.6980

[11] PN Vijaya Kumar and V Raghunatha Reddy. 2014. A survey on recommender systems (RSS) and its applications. *International Journal of Innovative Research in Computer and Communication Engineering* 2, 8 (2014), 5254–5260.

[12] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2019. DARTS: Differentiable Architecture Search. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. https://openreview.net/forum?id=S1eYHoC5FX

[13] Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. 2018. Neural architecture optimization. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 7827–7838.

[14] Hanh T. H. Nguyen, Martin Wistuba, Josif Grabocka, Lucas Rêgo Drumond, and Lars Schmidt-Thieme. 2017. Personalized Deep Learning for Tag Recommendation. In *Advances in Knowledge Discovery and Data Mining - 21st Pacific-Asia Conference, PAKDD 2017, Jeju, South Korea, May 23-26, 2017, Proceedings, Part I*. 186–197. https://doi.org/10.1007/978-3-319-57454-7_15

[15] Junwei Pan, Jian Xu, Alfonso Lobos Ruiz, Wenliang Zhao, Shengjun Pan, Yu Sun, and Quan Lu. 2018. Field-weighted Factorization Machines for Click-Through Rate Prediction in Display Advertising. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*. 1349–1357. https://doi.org/10.1145/3178876.3186040

[16] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. 2018. Efficient Neural Architecture Search via Parameters Sharing. In *International Conference on Machine Learning*. 4095–4104.

[17] Yanru Qu, Bohui Fang, Weinan Zhang, Ruiming Tang, Minzhe Niu, Huifeng Guo, Yong Yu, and Xiuqiang He. 2019. Product-Based Neural Networks for User Response Prediction over Multi-Field Categorical Data. *ACM Trans. Inf. Syst.* 37, 1 (2019), 5:1–5:35. https://doi.org/10.1145/3233770

[18] Francesco Ricci, Lior Rokach, and Bracha Shapira. 2011. Introduction to recommender systems handbook. In *Recommender systems handbook*. Springer, 1–35.

[19] Suvash Sedhain, Aditya Krishna Menon, Scott Sanner, and Lexing Xie. 2015. Autorec: Autoencoders meet collaborative filtering. In *Proceedings of the 24th international conference on World Wide Web*. 111–112.

[20] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2820–2828.

[21] Jun Wang, Lantao Yu, Weinan Zhang, Yu Gong, Yinghui Xu, Benyou Wang, Peng Zhang, and Dell Zhang. 2017. Irgan: A minimax game for unifying generative and discriminative information retrieval models. In *Proceedings of the 40th International ACM SIGIR conference on Research and Development in Information Retrieval*. 515–524.

[22] Suhang Wang, Yilin Wang, Jiliang Tang, Kai Shu, Suhas Ranganath, and Huan Liu. 2017. What your images reveal: Exploiting visual contents for point-of-interest recommendation. In *Proceedings of the 26th International Conference on World Wide Web*. 391–400.

[23] Ronald J. Williams. 1992. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Mach. Learn.* 8 (1992), 229–256. https://doi.org/10.1007/BF00992696

[24] Sai Wu, Weichao Ren, Chengchao Yu, Gang Chen, Dongxiang Zhang, and Jingbo Zhu. 2016. Personal recommendation using deep recurrent neural networks in NetEase. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. 1218–1229. https://doi.org/10.1109/ICDE.2016.7498326

[25] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. 2018. SNAS: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926* (2018).

[26] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. 2019. Deep Learning Based Recommender System: A Survey and New Perspectives. *ACM Comput. Surv.* 52, 1 (2019), 5:1–5:38. https://doi.org/10.1145/3285029

[27] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. 2019. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys (CSUR)* 52, 1 (2019), 1–38.

[28] Xiangyu Zhao, Chong Wang, Ming Chen, Xudong Zheng, Xiaobing Liu, and Jiliang Tang. 2020. AutoEmb: Automated Embedding Dimensionality Search in Streaming Recommendations. *CoRR* abs/2002.11252 (2020). arXiv:2002.11252 https://arxiv.org/abs/2002.11252

[29] Xiangyu Zhao, Long Xia, Liang Zhang, Zhuoye Ding, Dawei Yin, and Jiliang Tang. 2018. Deep Reinforcement Learning for Page-wise Recommendations. In *Proceedings of the 12th ACM Recommender Systems Conference*. ACM, 95–103.

[30] Xiangyu Zhao, Long Xia, Yihong Zhao, Dawei Yin, and Jiliang Tang. 2019. Model-Based Reinforcement Learning for Whole-Chain Recommendations. *arXiv preprint arXiv:1902.03987* (2019).

[31] Xiangyu Zhao, Liang Zhang, Zhuoye Ding, Long Xia, Jiliang Tang, and Dawei Yin. 2018. Recommendations with Negative Feedback via Pairwise Deep Reinforcement Learning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 1040–1048.

[32] Xiangyu Zhao, Xudong Zheng, Xiwang Yang, Xiaobing Liu, and Jiliang Tang. 2020. Jointly Learning to Recommend and Advertise. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM.

[33] Barret Zoph and Quoc V Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).

[34] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 8697–8710.