AutoTower: Efficient Neural Architecture Search for Two-Tower Recommendation

GUANGHUI ZHU, SHEN JIANG, and FENG CHENG, State Key Laboratory for Novel Software Technology, Nanjing University, China

DEFU LIAN, University of Science and Technology of China, China

CHUNFENG YUAN and YIHUA HUANG, State Key Laboratory for Novel Software Technology, Nanjing University, China

Recommender systems are ubiquitous in people's daily lives. A common practice for large-scale recommendations is the retrieval-and-ranking method. The retrieval stage that retrieves a small fraction of related items from a large corpus plays a critical role in the performance of recommender systems. Recently, two-tower models have gained ever-increasing interest in large-scale retrieval. However, most of the existing architectures of two-tower models are based on vanilla deep neural networks, lacking the exploration of more novel architectures. Moreover, the two-tower architecture that takes the available computation resources and the data characteristics into account should be designed. Inspired by neural architecture search (NAS) in automated machine learning, we propose an effective and efficient NAS-based method called AutoTower for the two-tower architecture search. First, by revisiting existing two-tower architectures, we design a simple yet general and expressive search space based on the single-path paradigm. Furthermore, we propose an efficient one-shot search strategy, which consists of a weight-sharing-based supernet training stage and a complexity-aware evolutionary search stage. Extensive experiments on four real-world datasets demonstrate that AutoTower can consistently outperform human-crafted models with much fewer parameters and FLOPs. Moreover, the architectures searched by AutoTower are transferable. The open-source code of AutoTower is now available at https://github.com/autotower/AutoTower.

CCS Concepts: • **Information systems** → **Recommender systems**; • **Computing methodologies** → **Machine learning**; **Search methodologies**.

Additional Key Words and Phrases: Two-tower model, Recommender systems, Neural architecture search, Large-scale retrieval

ACM Reference Format:
Guanghui Zhu, Shen Jiang, Feng Cheng, Defu Lian, Chunfeng Yuan, and Yihua Huang. 2018. AutoTower: Efficient Neural Architecture Search for Two-Tower Recommendation. In . ACM, New York, NY, USA, 23 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Nowadays, recommender systems, which provide accurate and useful content to address the information overload problem, are ubiquitous in people's daily lives[4, 42]. One of the most critical challenges in building real-world recommender systems is to accurately score millions to billions of items in real-time for ranking and recommending [42]. A common practice for large-scale recommendation is the two-stage method where the retrieval stage first retrieves a small fraction of related items from a large corpus as candidate items, and then the ranking stage re-ranks these candidate items based on click histories or user ratings. The retrieval stage plays a fundamental role in recommender

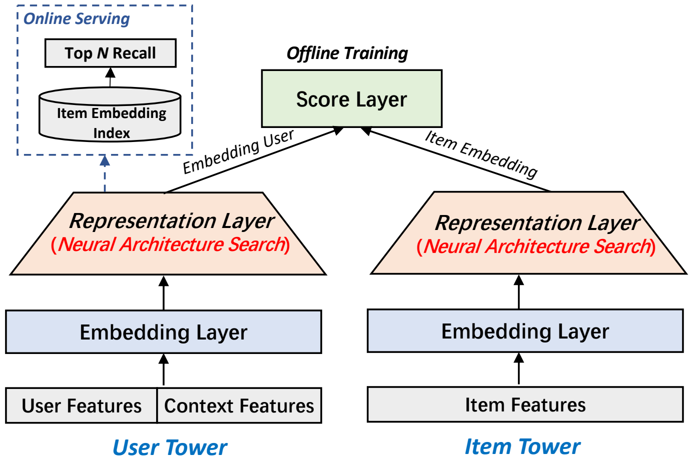


Fig. 1. General framework of two-tower recommendation.

systems [20], not only because it allows recommender systems to scale to billions of users and items, but also because the quality of retrieved candidate items is critical for the performance of the whole recommender system. In this work, we mainly focus on the retrieval stage.

Recently, the two-tower model consisting of a user tower and an item tower attains ever-increasing interest and has become popular in large-scale retrieval due to its efficiency and accuracy. Figure 1 shows the general framework of the two-tower model. The user tower and the item tower are responsible for learning the user and item representations, separately and respectively. Usually, the preference of a user-item pair can be scored as the inner product of user representation and item representation. Other score functions such as Euclidean distance and Cosine similarity can also be used. The embeddings of all the items will be first calculated and stored offline for retrieval. During online serving, the user features and other context features (e.g., historical behavior sequence) are first fed into the user tower to get the user embeddings. Then, the approximate nearest neighbor (ANN) search such as Faiss [18] is employed to retrieve candidate items by calculating the preference scores between the online-calculated user embedding and the offline-stored item embeddings.

As shown in Figure 1, the architecture of the representation layer in the two-tower model plays a critical role in the quality of representation learning. In the large-scale retrieval scenarios, considering the training efficiency and the online serving latency, existing two-tower models such as YoutubeDNN [5] and GoogleDNN [42] apply vanilla deep neural networks (DNNs) like multi-layer perceptrons (MLP) as the architecture of the representation layer. In practice, such relatively general architecture design does not yet fulfill the following requirements:

(1) **Better model performance.** To enhance the representation quality, it is well worth exploring more complex and novel architectures beyond vanilla DNNs. For example, diverse representation operations (e.g., self-attention [37] and element-wise operations) can be flexibly combined as the architecture of the representation layer.
(2) **Resource-specific architecture design.** Besides the model performance, the architecture complexity (e.g., FLOPs or the number of parameters) of the two-tower model should be constrained to match the available computational resources. Especially for the user tower, when the online computation resource is limited, architectures with low complexity (low FLOPs or a small number of parameters) should be designed.

(3) **Data-specific architecture design.** Since the user and item features usually have different characteristics (e.g., the number of instances, the number of features, the degree of sparseness, etc.), there is no single architecture that can perform well on different datasets so the data-specific architectures should be designed.

As the core of automated machine learning (AutoML), neural architecture search (NAS) has achieved great success in computer vision tasks [1, 47]. It aims to search for the optimal deep architecture for the given task in a data-driven manner without human intervention. Inspired by the idea of NAS, we propose to utilize the NAS method to search for resource-specific and data-specific architectures for the two-tower recommendation. However, there exist two challenges in the design of NAS-based tower architecture.

First, designing a general and effective search space that can cover both the existing and unexplored tower architectures is challenging. Also, taking the search efficiency of the large-scale retrieval into consideration, the search space cannot be too large. Moreover, how to utilize user historical behavior features to capture user interests also needs to be considered in the search space. Second, we should design an effective search strategy that can handle large-scale corpus efficiently, and support simultaneously optimizing multiple objectives (i.e., model performance and architecture complexity).

To address the two challenges, we propose an effective and efficient NAS-based method called AutoTower for the two-tower architecture search. First, by revisiting existing two-tower architectures, we design a simple, yet general and expressive search space based on the single-path with shortcut connections. Specifically, the overall search space is composed of a pooling block and a stack of choice blocks. The pooling block is used to model the user behavior features and the choice block is used for representation learning. Multiple choice blocks are stacked to enhance the representation ability. Each block contains multiple candidate operations, and only one operation will be activated. The proposed single-path-based search space can cover existing and novel unexplored architectures, and can be easily expanded by adding new representation operations.

Furthermore, based on the single-path search space, we propose an efficient one-shot search strategy, which consists of a supernet training stage and an architecture search stage. In the supernet training stage, we first leverage uniform sampling to sample child architectures from the weight-sharing supernet. Then, we employ the mini-batch Softmax as the objective to improve the training efficiency of the sampled child architecture. In the architecture search stage, we utilize a model complexity-aware evolutionary algorithm to search for optimal architectures. Owning to the pretrained supernet, the evolutionary search can directly evaluate candidate architectures on the validation dataset without training. In this way, the search efficiency can be greatly improved.

In summary, our contributions can be summarized as follows:

- We approach the problem of two-tower recommendation from the perspective of neural architecture search and propose a NAS-based method called AutoTower to search for novel two-tower architectures under the model complexity constraints. To our best knowledge, we are the first to utilize NAS to search for two-tower models.
- First, we design a simple, yet general and expressive search space based on the single-path with shortcut connections. The overall search space is composed of a pooling block and a stack of choice blocks. Only one operation can be activated on each block.
- Then, we propose an efficient one-shot search strategy based on the single-path search space, which contains two stages, i.e., the weight-sharing-based supernet training stage and the complexity-aware evolutionary search stage.

- Extensive experiments on four real-world datasets demonstrate that AutoTower can consistently outperform human-crafted two-tower models by **28.59%-37.22%** average relative improvement of Recall@50, and **33.01%-43.20%** average relative improvement of NDCG@50 with **50.55%-88.03%** average decrease of parameters and **8.00%-70.06%** average decrease of FLOPs. Moreover, the architectures searched by AutoTower are transferable. The analysis of the searched architectures provides insightful guidance to help human experts to design effective two-tower models, e.g., compact models also can achieve better performance.

2 RELATED WORK

2.1 Deep Recommender Systems

With the success of deep learning in computer vision and natural language processing [19], developing deep recommendation models has also become a research hotspot. NCF [14] and NFM [12] utilize neural networks to extend collaborative filtering (CF) and factorization machine (FM), respectively. Wide & Deep [4] jointly trains a wide component for memorization and a deep component for generalization. DeepFM [9] uses an FM layer to replace the wide component in Wide & Deep. xDeepFM [22], PNN [31], DCN [38], and AutoInt [36] design specific structures to model high-order feature interactions. FiBiNET [17] utilizes Squeeze-Excitation network (SENET) [15] to learn the importance of features. DIN [45], DIEN [44], and DSIN [6] focus on capturing user interests from user historical behaviors. Such methods are mainly used for the ranking stage, especially for the click-through rate (CTR) prediction.

For the retrieval stage, CF is one of widely used methods. Item-based CF [24, 34] precomputes the item-item similarity matrix, items similar to clicked ones are recommended to the user. NAIS [13], a deep learning method for item-based CF, uses an attention mechanism to distinguish different importance of user behaviors. Unlike item-based CF, another line follows the inner product paradigm like matrix factorization, which is also known as embedding-based methods.

One typical embedding-based method is the two-tower model, which has attained ever-increasing interest in large-scale retrieval. DSSM [16] is a pioneering work that applies the two-tower model to the web search. YoutubeDNN [5] and GoogleDNN [42] use two DNNs to model user features and item features, respectively, and formulate the learning problem of retrieval as an extreme classification problem. MIND [20] uses capsule networks to capture multiple user interests from user behaviors for better modeling user representation. The works in [8, 43, 46] focus on the entire corpus retrieval to eliminate the mismatch caused by ANN. In this paper, we follow the two-tower model and propose a novel NAS approach to find optimal architectures for the two-tower model.

2.2 Neural Architecture Search

Neural architecture search aims to automatically design optimal neural networks in a data-driven manner. Recently, many NAS methods such as evolutionary-based [32], reinforcement-learning-based [30], and gradient-based [27] methods have been proposed. As a pioneering NAS work, [47] uses an RNN controller to generate architectures and employs the policy gradient algorithm to train the controller. However, this method suffers from low search efficiency. To improve the search efficiency, the cell-based micro search space has been proposed [48]. Moreover, ENAS [30] employs a weight-sharing mechanism to further improve search efficiency. DARTS [27] and NASP [41] take differentiable approaches by relaxing the discrete search space to be continuous and applying the bi-level optimization to optimize architecture parameters and model weights alternatively. SPOS [10] first trains the supernet through randomly sampled children architectures, then searches for the optimal architecture by the evolutionary algorithm.

2.3 AutoML for Recommender Systems

With the success of NAS, there is an increasing interest in applying NAS to recommender systems. AutoFIS [26] and AutoGroup [25] automatically identify important cross features in a differentiable manner. AutoCTR [35] modularizes representative feature interactions as virtual building blocks and performs evolutionary architecture exploration for the feature interaction architecture. AutoPI [28] designs a comprehensive search space for feature interaction architectures and a gradient-based search strategy. SIF [41] exploits the NASP method to search for interaction functions for CF. AutoCF [7], an extension of SIF, splits CF methods into disjoint stages and utilizes random search to find optimal CF models. To our best knowledge, we are the first to apply NAS to the two-tower model in the retrieval stage.

3 THE PROPOSED METHOD: AUTOTOWER

In this section, we first present the problem definition and define a unified framework of the tow-tower model. Then, we introduce the search space and the search strategy of AutoTower in detail.

3.1 Problem Definition

3.1.1 A Unified Framework of Two-Tower Model. As shown in Figure 1, we abstract existing two-tower models into a unified framework. In general, the two-tower model is composed of a user tower and an item tower. Each tower can be divided into three layers, i.e., the embedding layer, the representation layer, and the score layer. During offline training, the two-tower model is trained by specific loss functions (e.g., cross-entropy loss or log-loss) according to the outputs of the score layer. After the training process, the representations of all items are calculated and stored offline. During online serving, when a user query arrives, the user tower will calculate the user's representation in real-time. Then, the ANN search is used to retrieve the most relevant items from the pre-calculated item corpus for the given query based on the score function. We refer to the output of embedding layer as embedding, and the output of representation layer as representation for clarity. The representation is also known as user (or item) embedding. The details of the three layers are as follows:

- **Embedding Layer.** Following the common embedding paradigm in recommender systems, the embedding lookup tables are used to transform sparse features into dense embedding vectors. The embeddings of all user (or item) features are concatenated as the output of the embedding layer in the user (or item) tower. In this work, the user historical behavior sequence containing features of items that the user has interacted with before also serves as input to the user representation layer.
- **Representation Layer.** As the core of the two-tower model, the representation layer encodes the input feature as a low-dimensional representation based on the deep neural architecture. Let e_u and e_v denote the user representation and the item representation, respectively. They will be used to calculate the preference score of the user-item pair in the score layer. In this paper, we utilize the NAS-based method to search for novel architectures for both user representation and item representation layers.
- **Score Layer.** The user representation e_u and the item representation e_v then go through the score layer to obtain the similarity score of the user-item pair. One commonly used score function in the score layer is the inner product of e_u and e_v, i.e., $\langle e_u, e_v \rangle$. In fact, if e_u and e_v are normalized before calculating the score, the Euclidean distance, the Cosine similarity, and the inner product of them are all equivalent. With the similarity score, losses can be calculated according to the loss function and the two-tower model can be trained through gradient descent.

3.1.2 Formulating as AutoML Problem. Users and items are represented by feature vectors $\{\boldsymbol{u}_i\}_{i=1}^N$ and $\{\boldsymbol{v}_j\}_{j=1}^M$, respectively, where N is the number of users and M is the number of items. For each user $\boldsymbol{u_i}$, we have a sequence of user historical behaviors $\boldsymbol{h}_i = (\boldsymbol{h}_i^{(1)}, \boldsymbol{h}_i^{(2)}, \cdots, \boldsymbol{h}_i^{(l)})$, sorted by occurrence time. Where $\boldsymbol{h}_i^{(t)}, 0 \le t \le l$ records the t^{th} item interacted by user $\boldsymbol{u}_i$, and l denotes the length of user behavior sequence. Let t_u and t_v represent the user tower and the item tower, respectively. The corresponding model parameters of the user tower and the item tower are θ_u and θ_v. Let $t_u(\boldsymbol{u}, \boldsymbol{h}, \theta_u)$ and $t_v(\boldsymbol{v}, \theta_v)$ denote the user and item representations generated by the user tower and item tower, respectively. We omit subscripts of the user and the item for clarity. In this work, the inner product is used as the score function, namely

$$s(\boldsymbol{u}, \boldsymbol{h}, \boldsymbol{v}) = \langle t_u(\boldsymbol{u}, \boldsymbol{h}, \theta_u), t_v(\boldsymbol{v}, \theta_v) \rangle. \tag{1}$$

The goal of NAS is to automatically design optimal user tower and item tower architectures for the given dataset. The training dataset $\mathcal{S}_{tra}$ and the validation dataset $\mathcal{S}_{val}$ are indicated as

$$\mathcal{S}_{train} := \{(\boldsymbol{u}_i, \boldsymbol{h}_i, \boldsymbol{v}_i, r_i)\}_{i=1}^T,$$
$$\mathcal{S}_{val} := \{(\boldsymbol{u}_i, \boldsymbol{h}_i, \boldsymbol{v}_i, r_i)\}_{i=1}^V,$$

where $(\boldsymbol{u}_i, \boldsymbol{h}_i, \boldsymbol{v}_i)$ denotes a tuple of observed user $\boldsymbol{u}_i$, user behavior sequence $\boldsymbol{h}_i$, and item $\boldsymbol{v}_i$. $r_i \in \mathbb{R}$ is the corresponding score of the user-item pair in each tuple. T and V denote the number of instances in $\mathcal{S}_{train}$ and $\mathcal{S}_{val}$, respectively. The architecture search of the two-tower model can be formulated as a bi-level optimization problem under the model complexity constraint.

$$\begin{aligned} (t_u^*, t_v^*) = &\mathop{\arg\min}_{t_u \in \mathcal{T}_u, t_v \in \mathcal{T}_v} \mathcal{L}_{val}(t_u, t_v, \theta_u^*, \theta_v^*, \mathcal{S}_{val}), \\ \text{s.t.} \quad & (\theta_u^*, \theta_v^*) = \mathop{\arg\min}_{(\theta_u, \theta_v)} \mathcal{L}_{train}(t_u, t_v, \theta_u, \theta_v, \mathcal{S}_{train}), \\ & C(t_u) \le c_u \text{ and } C(t_v) \le c_v, \end{aligned} \tag{2}$$

where $\mathcal{T}_u$ and $\mathcal{T}_v$ are architecture search spaces of the user tower and the item tower, respectively. $\mathcal{L}_{train}$ and $\mathcal{L}_{val}$ denote the losses on the training dataset and the validation dataset, respectively. $C(\cdot)$ is the function used to evaluate the architecture complexity. In general, we can use FLOPs or the number of parameters to define the model complexity [40]. c_u and c_v are complexity constraints for the user tower and the item tower, respectively.

3.2 Search Space Design

3.2.1 Design Criterion. In large-scale retrieval scenarios, the design of the architecture search space of the two-tower model should satisfy two criteria as far as possible.

- **Generality and Expressiveness.** The search space can cover not only the existing tower architectures but also the potential architectures not yet explored.
- **Simplicity.** In practice, to improve the search efficiency of the large-scale recommendation, the search space should be simple and compact.

To meet the two criteria, we propose a simple, yet general and expressive search space in this paper.

3.2.2 Single-Path Based Search Space Design. As shown in Figure 2, the overall search space can be represented by the single-path with shortcut connections, which consists of a pooling block and a stack of choice blocks. The pooling block, which is used to capture the user behavior features from the sequence of interacted items, only exists in the user

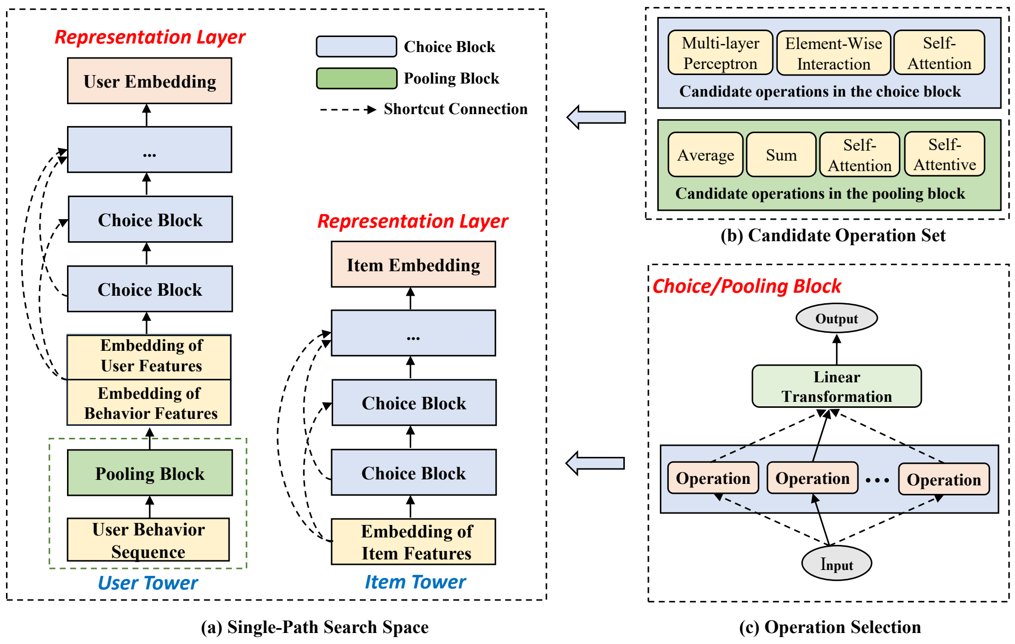


Fig. 2. (a) Overview of the search space. (b) Candidate operation set in the pooling block and the choice block. (c) Discrete operation selection.

tower. The choice block is used to construct the representation layer for learning representations from the user and item feature embeddings. Both the pooling block and the choice block are composed of multiple candidate operations. In each block, only one operation can be selected and activated during the search.

In the pooling block, we choose four commonly used pooling operations to model the user historical behavior sequence. Specifically,

- **Average Pooling** over the sequence length.
- **Sum Pooling** over the sequence length.
- **Self-Attention Pooling**. The user behavior sequence first goes through a standard Multi-Head Self-Attention layer [37], then the average pooling is adopted over the sequence length. In this work, we only use one head for simplicity, i.e.,

$$H_{att} = \text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^\top}{\sqrt{d_k}})V,$$

$$\boldsymbol{u}_h = \text{Average}(H_{att}),$$

where Q, K, and V are the query, key, and value matrices used to transform the user behavior sequence $\boldsymbol{h}$ linearly. d_k denotes the dimension of the query and key. $\boldsymbol{u}_h$ denotes the user behavior feature after pooling.
- **Self-Attentive Pooling**. Following the Self-Attentive in [23], we use the attention mechanism to obtain a weighted sum over the user behavior sequence, i.e.,

$$\boldsymbol{a} = \text{softmax}(\boldsymbol{w}_2^\top \tanh(W_1 H))^\top,$$

$$\boldsymbol{u}_h = H\boldsymbol{a},$$

Table 1. Details of choice blocks.

Operation in the Choice Block	Number of Block Types	Hyperparameter of the Choice Block	Operating on Inputs
Multi-layer Perceptron (MLP)	7	*The number of units in the hidden layer:* **16, 32, 64, 128, 256, 512, 1024**	Concatenation
Element-Wise Interaction (EW)	5	*Element-wise operations:* **sum, average, inner product, max, min**	Element-wise
Self-Attention (SA)	4	*The number of heads:* **1, 2, 3, 4**	Stack

where H is the matrix transformed from the user behavior sequence $\boldsymbol{h}$, with each row representing an item, and $\boldsymbol{w}_2$ and W_1 are trainable parameters. The vector $\boldsymbol{a}$ represents the attention weights of items in the behavior sequence.

There is another line of work that focuses on user behavior sequence modeling for the recommendation, but it is beyond the scope of this paper. Note that the Self-Attention and Self-Attentive are different in the mechanism. The Self-Attention mainly captures relationships of tokens in the sequence, but the Self-Attentive directly learns weights to aggregate representations of tokens.

In the choice block, the commonly used representation learning operations such as Multi-Layer Perceptron (MLP), Element-Wise Interaction (EW), and Self-Attention (SA) are employed. Table 1 shows the details of the candidate operations in the choice block. One type of operation may correspond to multiple block types with different hyperparameters, e.g., different numbers of hidden units in MLP represent different block types. Also, other representation learning operations such as SENET Layer [15] and Product Layer [31] can be easily integrated into the search space.

3.2.3 Shortcut Connection. Moreover, we further introduce a shortcut connection technique [11, 33], which enables the low-level feature representation to flow to high-level choice blocks directly. The shortcut connection can help low-level block to obtain more information from the final user-item interaction during back-propagation. Each choice block takes the outputs of previous blocks and original input feature embeddings as inputs. We further perform dimension alignment for each choice block (except the last one) to assist the aggregation of multiple inputs. As shown in Figure 2(c), the dimension of the output of each choice block is transformed to the same size D_{blk} by an additional linear transformation. The output of the last choice block is viewed as the final user representation or item representation.

As shown in Figure 2, the user and item towers share the same architecture search space (except the pooling block). However, considering the differences in application scenarios and input features, the complexities of architectures and the numbers of choice blocks of the user tower and the item tower can be different.

3.2.4 Search Space Size Analysis. Given the above single-path search space, the search objective is to find the best operation choice in each block, i.e., the operation activated in each choice block or pooling block. Correspondingly, the size of search space is determined by the number of pooling types $|\mathcal{P}|$, the number of choice block types $|\mathcal{B}|$, the number of choice blocks in the user tower N_u, and the number of choice blocks in the item tower N_v. Thus, the size of search space can be calculated as $|\mathcal{P}| \cdot |\mathcal{B}|^{N_u+N_v}$. In this work, we use 4 types of pooling blocks, 16 types of choice blocks, and N_u and N_v are from 3 to 7. The size of the search space is at least 6.72×10^7. Although the design of search space is simple and compact, its size is still substantial. To improve search efficiency within such a search space, we further propose an efficient one-shot search strategy.

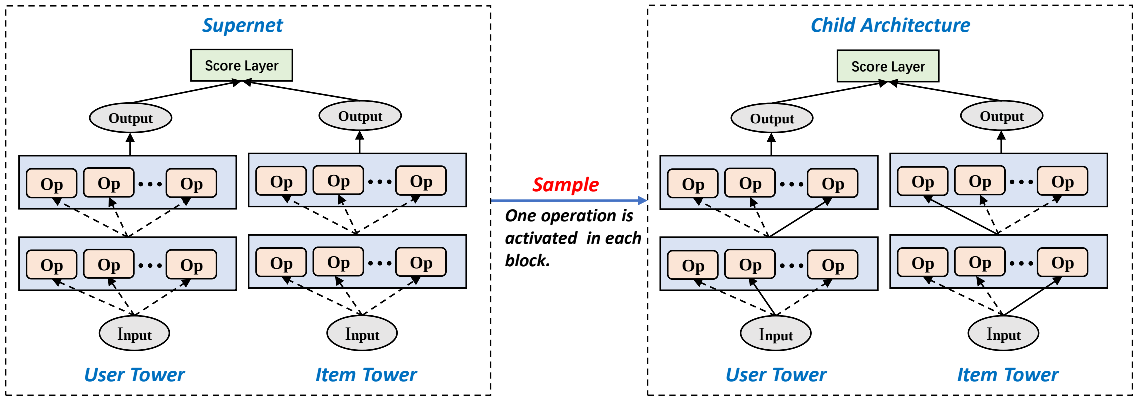


Fig. 3. Supernet and child architecture.

3.3 One-Shot Search Strategy

3.3.1 Overview of Search Strategy. The proposed search space follows the single-path paradigm. As shown in Figure 3, the overall search space can be represented as a supernet, which contains all possible architectures. The basic unit of the supernet is the block (i.e., choice block and pooling block). Each block contains multiple operations, and the child architecture is obtained by sampling and activating one operation in each block. The topology of the supernet is relatively simple for search efficiency, and the search space can be general and expressive by choosing appropriate operations in each block.

The supernet subsuming all architectures can be denoted by $\mathcal{N}(\mathcal{T}, \Theta)$, where $\mathcal{T}$ is the search space and Θ is the weights in the supernet. For brevity, the user and item towers are not distinguished here.

Training the two-tower model from scratch incurs huge computation overhead, especially for large-scale retrieval. To address the issue, we propose a one-shot search strategy. As shown in Figure 4, the search strategy contains two decoupled stages, i.e., the supernet training stage and the architecture search stage. The supernet training stage trains the supernet through a weight-sharing mechanism. Since the supernet is trained once and all architectures inherit their weights directly from Θ, we can only perform fine-tuning for each sampled child architecture and avoid training from scratch. After the supernet is well-trained, we utilize a model complexity-aware evolutionary algorithm to search for optimal architectures by directly computing the validation loss of sampled architectures from the supernet without any training overhead. Formally, the one-shot NAS can be expressed as

$$\begin{aligned} (t_u^*, t_v^*) &= \underset{t_u \in \mathcal{T}_u, t_v \in \mathcal{T}_v}{\arg\min} \ \mathcal{L}_{val}(t_u, t_v, \theta_u^*, \theta_v^*, \mathcal{S}_{val}), \\ \text{s.t.} \quad \Theta &= \underset{(\theta_u, \theta_v) \in \Theta}{\arg\min} \ \mathbb{E}_{(t_u, t_v) \sim \Gamma(\mathcal{T}_u, \mathcal{T}_v)} \mathcal{L}_{train}(t_u, t_v, \theta_u, \theta_v, \mathcal{S}_{train}), \\ & C(t_u) \leq c_u \text{ and } C(t_v) \leq c_v, \end{aligned} \tag{3}$$

where $\Gamma(\mathcal{T}_u, \mathcal{T}_v)$ is the prior distribution of the search space, (θ_u^*, θ_v^*) is the best supernet parameter. Finally, the searched candidate architectures will be re-trained on the full training dataset to obtain the best architecture.

3.3.2 Supernet Training Based on Weight-Sharing. The supernet training algorithm is described in Algorithm 1. In the beginning, the weights of the supernet are randomly initialized. The supernet training process is composed of many

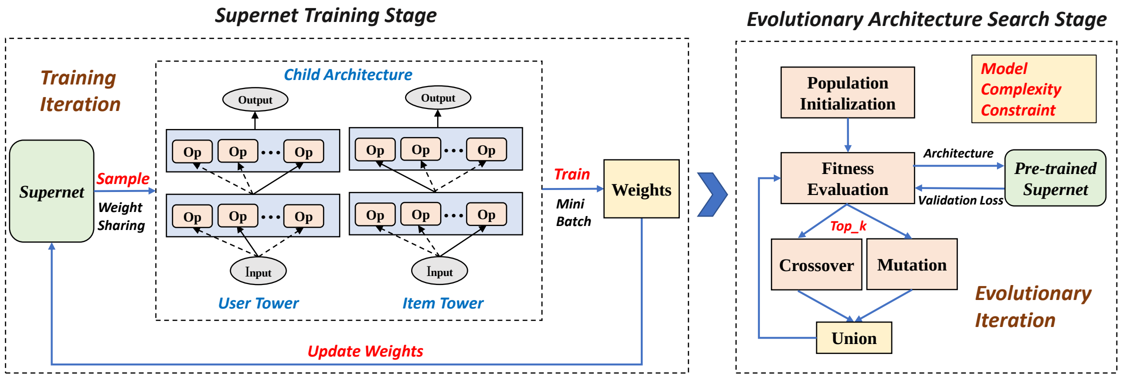


Fig. 4. Overview of the one-shot search strategy.

Algorithm 1 Supernet training in AutoTower

```
Require: Search space of the user tower $\mathcal{T}_u$, search space of the item tower $\mathcal{T}_v$, the number of training iterations $M$,
    the training dataset $\mathcal{S}_{train}$;
Ensure: Weights of supernet $\Theta$;
1: $\Theta := initialize\_weights()$;
2: for $i = 1 : M$ do
3:     $t_u := uniform\_sampling(\mathcal{T}_u)$;        ▷ sample item tower
4:     $t_v := uniform\_sampling(\mathcal{T}_v)$;        ▷ sample item tower
5:     $\mathcal{B} := batch\_generating(\mathcal{S}_{train})$;
6:     $\Theta := train(t_u, t_v, \Theta, \mathcal{B})$;   ▷ Train child architecture and optimize corresponding weights of the supernet
7: end for
8: return Weights of the trained supernet;
```

training iterations. In each iteration, we randomly sample an architecture containing a user tower and an item tower from the supernet, and then we sample a batch of data from the training dataset to train the sampled child architecture. Inspired by [10], with no prior knowledge of Γ, we can leverage uniform sampling which proves to be empirically good enough and efficient to sample as many child architectures as possible. Then, the sampled child architectures can be trained in batches.

To improve the training efficiency, we adopt two optimization techniques: weight-sharing [2, 27, 30] and mini-batch Softmax. First, all the child architectures will inherit weights from the supernet. In other words, they share the weights in the common block of the single-path. When training the child architecture (t_u, t_v), the corresponding weights $\Theta(t_u, t_v)$ of the supernet will be activated, fine-tuned, and updated. Thus, we can eschew training each child architecture from scratch to convergence, and the computation cost can be greatly reduced.

Second, following previous works [5, 42], we treat the large-scale retrieval as an extreme classification problem and exploit the mini-batch Softmax as the loss function. Given a mini-batch of user-item pairs $\mathcal{B} = \{(\boldsymbol{u}_i, \boldsymbol{h}_i, \boldsymbol{v}_i, r_i)\}_{i=1}^{B}$, where B denotes the batch size. For each $i \in [1, B]$,

$$P_B^c(\boldsymbol{v}_i | \boldsymbol{u}_i, \boldsymbol{h}_i; \theta_u, \theta_v) = \frac{e^{s^c(\boldsymbol{u}_i, \boldsymbol{h}_i, \boldsymbol{v}_i)}}{e^{s^c(\boldsymbol{u}_i, \boldsymbol{h}_i, \boldsymbol{v}_i)} + \sum_{j=1, j\neq i}^{B} e^{s^c(\boldsymbol{u}_i, \boldsymbol{h}_i, \boldsymbol{v}_j)}}, \tag{4}$$

$$s^c(\boldsymbol{u}_i, \boldsymbol{h}_i, \boldsymbol{v}_j) = s(\boldsymbol{u}_i, \boldsymbol{h}_i, \boldsymbol{v}_j)/\tau - \log(p_j), \tag{5}$$

where $s(\cdot,\cdot,\cdot)$ is the score function that calculates the preference score of the given user-item pair. The temperature τ is a hyperparameter added to sharpen the predicted logits. $\log(p_j)$ is a correction to the sampling bias, and p_j denotes the sampling probability of the item v_j in a random batch. As in GoogleDNN [42], p_j is estimated by the average number of steps between two consecutive hits of the item. As a result, the batch loss function can be expressed as:

$$\mathcal{L}_B = -\frac{1}{B}\sum_{i=1}^{B} r_i \cdot \log(P_B^c(\boldsymbol{v}_i|\boldsymbol{u}_i, \boldsymbol{h}_i; \theta_u, \theta_v)). \tag{6}$$

In practice, we ignore the scores in the datasets and set all scores to one. Since the supernet training process is performed iteratively and a batch of data is sampled in each iteration to train the sampled child architecture, if we fix the batch size and the number of iterations, the supernet training time can be irrelevant to the dataset scale. This property enables the search strategy to adapt to industry large-scale scenarios.

3.3.3 Complexity-Aware Evolutionary Search. After the supernet is well-trained, we begin to search for superior architectures guided by the supernet. In the search process, since the supernet has been fully trained, we no longer need to train from scratch for evaluation when we sample a child architecture. In other words, we can approximate its performance by directly evaluating it with inherited weights on the validation dataset. Evolutionary algorithm is employed for the architecture search due to its effectiveness and efficiency. Each child architecture containing a user tower and an item tower is viewed as an individual for evolution. Meanwhile, we explicitly impose model complexity constraints during the evolutionary search.

As shown in Figure 2, for the user (or item) tower, the architecture can be expressed as a stack of pooling or choice blocks. In each block, only one operation will be activated. The complexity of the whole architecture can be decoupled into the complexity of each pooling or choice block. It is easy to calculate the complexity of architecture by accumulating the complexity of each block. Thus, in the evolutionary search process, we can effectively restrict the model complexity to ensure that it does not exceed the specified upper bound. Therefore, this is a multi-objective evolutionary algorithm that simultaneously takes the model performance and complexity into account. Usually, the model complexity can be represented by the number of parameters, FLOPs, and the model inference time [40]. Since the inference time cannot be precisely broken down into blocks of the architecture, coupled with the fact that it depends on specific hardware and runtime environment, we adopt the number of parameters and FLOPs as complexity measures.

The overall evolutionary search process is described in Algorithm 2. In the evolutionary search process, whenever a new architecture is generated through sampling, crossover, or mutation, we verify whether it satisfies the predefined complexity constraints, and the architectures that do not meet the constraints will not be retained. The fitness of each individual is determined by the validation loss of the corresponding architecture. Due to the weight-sharing mechanism, we do not need to train the architecture, but directly evaluate it on the validation dataset with the inherited weights from the supernet. We initialize the population by random sampling architectures from the search space (i.e., supernet). Then, the evolution is performed iteratively. In addition, we maintain a global set of top performance architectures, which contains top_k architectures with the highest fitness.

In each iteration, we first evaluate each architecture in the population to get its fitness. Then, the top_k set is updated according to the fitness of the current population. The crossover and mutation are all performed on the top_k set. Specifically,

- **Crossover:** We randomly sample two architectures as parents from the top_k set, and each block in the offspring architecture is randomly selected from the corresponding two blocks in parents.

Algorithm 2 Evolutionary search in AutoTower

```
Require: Search space of the user tower $\mathcal{T}_u$, search space of the item tower $\mathcal{T}_v$, weights of the supernet $\Theta$, population size $P$, complexity constraint of the user tower $C_u$, complexity constraint of the item tower $C_v$, the number of search iterations $M_{search}$, the validation dataset $\mathcal{S}_{val}$;
Ensure: Top architectures under the complexity constraints;
 1: $n := P/2$;                                   ▷ the number of crossovers in each iteration
 2: $m := P/2$;                                   ▷ the number of mutations in each iteration
 3: $k := P/2$;                                   ▷ the number of top architectures
 4: $p := 0.1$;                                   ▷ the probability of mutation
 5: $top\_k = \emptyset$;                         ▷ maintain top-k architectures
 6: $\mathcal{P}_0 := population\_initialize(P, \mathcal{T}_u, \mathcal{T}_v, C_u, C_v)$;   ▷ initialize population with randomly sampled architectures
 7: for $i = 1 : M_{search}$ do
 8:     $fitness_{i-1} := evaluate(\Theta, \mathcal{S}_{val}, \mathcal{P}_{i-1})$;
 9:     $top\_k := update\_topk(top\_k, \mathcal{P}_{i-1}, fitness_{i-1})$;
10:     $\mathcal{P}_{crossover} := crossover(top\_k, n, C_u, C_v)$;
11:     $\mathcal{P}_{mutation} := mutation(top\_k, m, p, C_u, C_v)$;
12:     $\mathcal{P}_i := \mathcal{P}_{crossover} \cup \mathcal{P}_{mutation}$;
13: end for
14: return Top architectures in the $top\_k$ set;
```

- **Mutation:** We randomly sample one architecture from the top_k set, and each block in the architecture will be replaced by another random block with the mutation probability.

The crossover and mutation are repeated until a specific number of offspring are produced. Then, a new population is generated by combining crossover and mutation offspring. The evolutionary search process stops when the number of iterations M_{search} is reached. Then, we choose several best architectures from the last top_k set. These architectures will be re-trained on the whole training dataset, and the best one with the highest validation performance is selected as the final result of the architecture search.

4 EXPERIMENTS

In this section, we conduct extensive experiments to answer the following research questions.

- **RQ1**: How does the performance of the architectures searched by AutoTower under different complexity constraints compare with the state-of-the-art human-crafted architectures?
- **RQ2**: How do different design strategies (i.e., shortcut connection, search strategy, and complexity constraint) in AutoTower influence the performance?
- **RQ3**: How do the hyperparameters (e.g., the number of blocks and embedding dimension) influence the performance of AutoTower?
- **RQ4**: Are the searched architectures able to be transferred across different datasets?

4.1 Experimental Setup

4.1.1 Datasets. Experiments are conducted on the following four public datasets, whose statistics are summarized in Table 2. For all datasets, we treat all observed instances as positive samples and all items that the user has not interacted with as negative items for the retrieval task. Moreover, we aggregate user behavior sequences by the timestamps.

Table 2. Statistics of the datasets.

Dataset	#Users	#Categories	#Items	#Categories	#Records
ML-100K	943	4	1, 682	3	100, 000
ML-1M	6, 040	4	3, 706	3	1, 000, 209
Yelp	60, 543	6	74, 249	6	2, 880, 522
Amazon	165, 036	1	159, 551	2	5, 073, 469

- **MovieLens-100K**[1] **(ML-100K) & MovieLens-1M**[2] **(ML-1M)**. Two widely used movie-rating datasets contain different numbers of ratings on movies. We set the maximum length of the behavior sequence for each user in both datasets to 20.
- **Yelp**[3]. It is published officially by Yelp, a crowd-sourced review forum website where users can write their comments and reviews for various POIs, such as hotels, restaurants, etc. The maximum length of the user behavior sequence for each user is set to 40.
- **Amazon**[4]. It consists of product reviews and metadata from Amazon. We use the *Books* category of the Amazon dataset in our experiments. The maximum length of the behavior sequence for each user is set to 50.

In a commonly used manner, we filter out users and items with less than 20 records on *Amazon* and *Yelp*. For all preprocessed datasets, we split each dataset with 2 : 1 : 1 to generate the training set, validation set, and test set.

4.1.2 Evaluation Metrics. We adopt two widely used evaluation metrics for top-K recommendation to evaluate the recommendation performance, i.e., Hit Ratio (HR@K) and Normalized Discounted Cumulative Gain (NDCG@K).

- The hit ratio is the fraction of items for which the correct answer is included in the top-K recommendation list. Formally,
$$HR@K = \frac{NumberOfHits@K}{|\mathcal{I} - \mathcal{I}_{inter}|},$$
where the denominator represents the total number of items that the user has not interacted with in the training set, and the numerator represents the number of items in the top-K list that belong to the test set.
- To define the NDCG, we first define the cumulative gain (CG), which is the sum of scores to a position k in the top-K list, i.e.,
$$CG@k = \sum_{i=1}^{k} Score_i.$$
One obvious drawback of CG is that it does not take into account of ordering. By swapping the relative order of any two items, the CG would be unaffected. To tackle this issue, the discounted CG (DCG) is further introduced,
$$DCG@k = \sum_{i=1}^{k} \frac{Score_i}{\log_2(i+1)}.$$
However, DCG scores of different recommendation lists cannot be consistently compared. So the DCG scores need to be normalized. The ideal DCG (IDCG) is introduced for normalizing, which is the DCG score for the

[1]https://grouplens.org/datasets/movielens/100k
[2]https://grouplens.org/datasets/movielens/1m
[3]https://www.yelp.com/dataset/download
[4]https://nijianmo.github.io/amazon/

most ideal ranking (i.e., ranking the items top to down according their relevance scores).

$$IDCG@k = \sum_{i=1}^{|I(k)|} \frac{Score_i}{\log_2(i+1)},$$

where $I(k)$ represents the ideal list of items up to position k, $|I(k)| = k$. As a result, NDCG is to normalize the DCG score by the IDCG, i.e.,

$$NDCG@k = \frac{DCG@k}{IDCG@k}$$

The full-ranking protocol as in [7] is used to calculate HR@K and NDCG@K. Specifically, for each test instance, we calculate scores between the user and all unobserved items (both in the test set and training set), and then take top-K items to calculate the metrics. For each metric, the performance is computed based on the top 50 results, and the reported results are the average values across all the testing users.

Besides the two performance metrics, we also report FLOPs and the number of parameters of the architecture as complexity metrics. We extend *torchprofile*[5] to accurately compute the FLOPs of the two-tower model, and the modified codes are also provided in our open-source codes.

4.1.3 Baselines. We compare four widely-used human-crafted two-tower models to verify the effectiveness of Auto-Tower.

- **GoogleDNN** [42], a state-of-the-art two-tower model, which is a variation of YoutubeDNN [5]. GoogleDNN uses efficient in-batch negative sampling and introduces a correction into the mini-batch Softmax loss to correct the sampling bias.
- **SENET**. Borrowing the idea of applying SENET [15] to assign importance to input features in FiBiNet [17], we also add a SENET layer after the embedding layer of each tower based on GoogleDNN.
- **BST** [3] uses the powerful *Transformer* structure to capture sequential signals underlying users' behavior sequences. Then, it combines these signals with other user features and feeds them into a DNN structure to form the user tower. The item tower remains the same as GoogleDNN.
- **MIND** [20] is a recent state-of-the-art model for the retrieval stage of recommendation. It designs a multi-interest extractor layer based on the capsule routing mechanism, which is applicable for clustering past behaviors and extracting diverse user interests.

To ensure fair comparisons, we use exactly the same loss function for all the baselines and AutoTower, i.e., mini-batch Softmax with correlation as described in Section 3.3.2. Great efforts are put to tune the hyperparameters of these baselines, and the best performance is reported.

4.1.4 Complexity Constraint Scenarios. Since a main advantage of AutoTower is that it can explicitly constrain the complexity (i.e. the number of parameters or FLOPs) of the searched architecture. As Table 3 shows, we design three complexity constraint scenarios of different sizes (i.e., low, medium, and high) to verify the effectiveness of AutoTower under different constraints. The higher complexity constraint means that AutoTower can find more complex architectures. For the baselines, the complexities are distinguished by the number of DNN layers and the number of neurons of each DNN layer. We first define three complexity scenarios by different DNN layer settings of the baselines. Then, the complexity constraint of AutoTower is set according to the complexity of GoogleDNN under the same scenario.

[5]https://github.com/zhijian-liu/torchprofile

Table 3. Three different complexity constraint scenarios. "L", "M", and "H" denote low, medium, and high, respectively. "Parms" denotes the number of parameters.

	AutoTower		Baseline
	Parms Constraint	FLOPs Constraint	DNN Layers
L	0.24×10^6	0.15×10^6	[256, 192, 128]
M	0.48×10^6	0.28×10^6	[512, 256, 128]
H	1.20×10^6	0.80×10^6	[1024, 512, 256, 128]

Table 4. Overall Performance Comparison. "H@50" and "N@50" denote HR@50 and NDCG@50, respectively. The values of "Parms" and "FLOPs" omit unit 10^6, and other values are percentage numbers with "%" omitted. The best results are marked in bold, and the best results of baselines are marked by underline. The symbol * denotes statistically significant improvement (measured by t-test with p-value < 0.005) over the best baseline under the corresponding complexity constraint scenario.

	Model	MovieLens-100K				MovieLens-1M				Yelp				Amazon				$\Delta_{H@50}\uparrow$	$\Delta_{N@50}\uparrow$	$\Delta_{Parms}\downarrow$	$\Delta_{Flops}\downarrow$
		H@50	N@50	Parms	FLOPs	H@50	N@50	Parms	FLOPs	H@50	N@50	Parms	FLOPs	H@50	N@50	Parms	FLOPs				
L	GoogleDNN	36.49	40.86	0.21	**0.14**	32.12	43.64	0.21	**0.14**	8.11	13.95	0.24	0.15	41.68	32.81	0.19	0.13	-	-	-	-
	SENET	34.65	39.53	0.21	**0.14**	33.01	44.89	0.21	**0.14**	7.73	13.52	0.24	0.15	41.68	32.81	0.19	0.13	-1.74	-0.87	+0.00	+0.00
	BST	35.71	39.75	0.43	**0.14**	32.14	44.07	0.43	**0.14**	8.15	13.91	0.72	0.16	49.18	41.68	0.33	0.13	+4.10	+6.25	+120.80	+1.67
	MIND	35.53	37.32	0.10	0.16	22.72	30.87	0.10	0.16	9.10	14.95	0.11	0.17	51.78	44.51	**0.08**	0.14	+1.14	+1.23	**−54.21**	+12.40
	AutoTower	**39.22***	**43.39***	**0.05**	**0.14**	**39.44***	**51.56***	**0.09**	**0.14**	**10.81***	**17.08***	**0.10**	**0.05**	**62.85***	**64.92***	0.17	**0.11**	**+28.59**	**+36.16**	-50.55	**−20.51**
M	GoogleDNN	36.48	40.36	0.43	0.25	31.73	43.61	0.43	0.25	8.17	13.91	0.50	**0.28**	46.43	37.98	0.39	**0.23**	-	-	-	-
	SENET	34.95	38.74	0.43	0.25	33.56	45.69	0.43	0.25	7.99	13.71	0.50	**0.28**	46.43	37.98	0.39	**0.23**	-0.16	-0.17	+0.00	+0.00
	BST	36.03	40.20	0.65	0.25	32.18	44.06	0.65	0.25	8.29	14.04	0.98	0.29	51.41	45.24	0.53	**0.23**	+3.09	+5.17	+58.56	+0.89
	MIND	35.68	37.50	**0.21**	0.32	19.08	26.44	0.21	0.32	8.52	14.26	0.22	0.35	55.00	46.94	0.18	0.29	-4.83	-5.09	-53.04	+26.77
	AutoTower	**40.28***	**43.21***	0.22	**0.23**	**39.77***	**51.84***	**0.15**	**0.19**	**12.65***	**19.34***	**0.20**	**0.28**	**62.16***	**63.45***	**0.12**	**0.23**	**+31.12**	**+33.01**	**−60.80**	**−8.00**
H	GoogleDNN	34.40	39.35	1.57	0.81	31.51	43.36	1.57	0.81	8.07	13.82	1.70	0.88	41.45	31.39	1.49	0.77	-	-	-	-
	SENET	33.09	38.18	1.57	0.81	30.84	42.35	1.57	0.81	7.96	13.78	1.70	0.88	41.45	31.39	1.49	0.77	-1.82	-1.40	+0.00	+0.00
	BST	35.33	39.46	1.79	0.82	31.89	43.77	1.79	0.82	8.01	13.73	2.18	0.89	46.95	38.64	1.63	0.77	+4.11	+5.92	+16.41	+0.90
	MIND	36.58	38.87	0.77	1.17	22.64	30.62	0.77	1.17	8.67	14.46	0.81	1.22	53.30	46.25	0.73	1.10	+3.55	+5.34	-51.32	+42.60
	AutoTower	**39.60***	**43.72***	**0.19**	**0.22**	**39.58***	**51.62***	**0.17**	**0.19**	**12.72***	**19.43***	**0.23**	**0.30**	**62.39***	**63.42***	**0.17**	**0.27**	**+37.22**	**+43.20**	**−88.03**	**−70.06**

4.1.5 Implementation Details. All baselines and AutoTower are implemented with PyTorch[29]. We use the Adam optimizer and the batch size is 1024. For the baselines, the learning rates are tuned among $\{0.1, 0.01, 0.001\}$, and the embedding dimensions are tuned among $\{8, 16, 24, 32, 64\}$. The best results are reported. The number of supernet training iterations is set to 50,000, and the number of evolutionary search iterations is set to 50. The population size and the mutation probability are set to 50 and 0.1, respectively. The learning rates for the supernet training and re-training are both set to 0.001. The embedding dimension is set to 16, the dimension of the output of each tower is set to 64 (i.e., the dimension of the final user or item representation), and the number of choice blocks for AutoTower is set to 5, except for experiments in the hyperparameter study. We perform early-stopping with the patience of 5 based on the validation loss. All experiments are run five times with different random seeds, and the average performance is reported.

4.2 Overall Performance Comparison (RQ1)

Table 4 shows the overall performance comparison between AutoTower and baselines under three different complexity constraints. In addition to HR@50 and NDCG@50, we also report the number of parameters and FLOPs. Moreover, we calculate the relative performance improvement and complexity decrease over GoogleDNN under different complexity constraints. The reported results are the average values across different datasets. From Table 4, we have the following conclusions:

Table 5. Ablation study of the shortcut connection.

	MovieLens-100K		MovieLens-1M		Yelp		Amazon	
Short Connection	H@50	FLOPs	H@50	FLOPs	H@50	FLOPs	H@50	FLOPs
Without	37.49	0.27	37.46	0.28	10.50	0.40	59.45	0.26
With	**40.28**	**0.23**	**39.77**	**0.19**	**12.65**	**0.28**	**62.16**	**0.23**

- For the human-designed baselines, a more complex model with much more parameters or sophisticated structures may perform worse than a more compact one, which is mainly due to the overfitting problem. Moreover, there is no single human-designed model that can outperform other baselines on all datasets. MIND performs better on *Yelp* and *Amazon*, indicating that it can better capture users' interests with longer behavior sequences. But MIND performs much worse on *MovieLens*. By observing the training process, we find that the capsule network in MIND is more prone to overfitting on *MovieLens*. **It illustrates the necessity of AutoTower, which can design suitable two-tower models automatically in a data-specific manner.**
- AutoTower can achieve the best performance in all datasets and complexity scenarios with much fewer parameters or FLOPs, especially on *Amazon* which is the most widely-used industry dataset. Compared with GoogleDNN, AutoTower achieves **28.59%-37.22%** average relative improvement of HR@50, and **33.01%-43.20%** average relative improvement of NDCG@50 with **50.55%-88.03%** average decrease of parameters and **8.00%-70.06%** average decrease of FLOPs. **The significant improvements demonstrate that AutoTower can effectively find better-performing architectures with much lower complexity.**
- The architectures searched by AutoTower under different complexity constraints can achieve comparable performance. **It indicates that complex and redundant models may not be necessary for many retrieval applications.** A more compact model can also achieve good performance while having a shorter inference time during online serving.

4.3 Ablation Study (RQ2)

4.3.1 Shortcut Connection. To verify the effectiveness of the shortcut connection in AutoTower, we conduct experiments to remove the shortcut connection from the search space and keep other components unchanged. The results are shown in Table 5. After removing the shortcut connections, the performance of AutoTower drops significantly, but it still outperforms most baselines. It not only demonstrates the importance of shortcut connection but also indicates the advantage of operations in the single-path search space. In addition, the complexities of searched architectures also increase obviously after the shortcut connections are removed. This illustrates that the forward and backward flows generated by shortcut connections can help AutoTower to learn more compact models.

4.3.2 Search Strategy. To verify the effectiveness of the proposed one-shot search strategy, we also compare AutoTower with other widely used search strategies in our search space. We choose three representative search strategies for comparison: *differentiable method NASP* [41], *reinforcement learning (RL)* [30], and *random search* [21, 39]. Note that our complexity constraints cannot be applied to these methods. The best results of these search strategies are reported. From Table 6, we can observe that AutoTower can consistently outperform other search strategies. Moreover, all search strategies can achieve better performance than the human-crafted baselines, which can also verify the effectiveness of the proposed search space. Another advantage of the proposed one-shot strategy is that it can constrain the model

Table 6. Comparison with different search strategies. The best is marked in bold. The complexity constraint scenario of AutoTower is "medium".

	MovieLens-100K				MovieLens-1M				Yelp				Amazon			
Strategy	H@50	N@50	Parms	FLOPs	H@50	N@50	Parms	FLOPs	H@50	N@50	Parms	FLOPs	H@50	N@50	Parms	FLOPs
Random	36.72	40.79	0.28	**0.18**	38.50	50.15	0.41	0.24	11.49	18.07	0.54	1.19	58.98	58.03	0.38	**0.22**
RL	34.21	36.86	0.54	0.28	38.58	50.43	0.39	0.22	11.64	18.26	0.74	1.26	61.60	62.53	0.32	0.25
NASP	39.48	42.45	0.35	0.26	38.94	50.74	0.29	0.20	11.91	18.65	0.75	1.32	59.49	58.90	0.40	0.26
AutoTower	**40.28**	**43.21**	**0.22**	0.23	**39.77**	**51.84**	**0.15**	**0.19**	**12.65**	**19.34**	**0.20**	**0.28**	**62.16**	**63.45**	**0.12**	0.23

Table 7. Ablation study of the complexity constraint.

	MovieLens-100K		MovieLens-1M		Yelp		Amazon	
Complexity Constraint	H@50	FLOPs	H@50	FLOPs	H@50	FLOPs	H@50	FLOPs
Without	38.89	0.22	39.42	0.39	**12.42**	0.94	60.98	0.32
With	**39.22**	**0.14**	**39.44**	**0.14**	10.81	**0.05**	**62.85**	**0.11**

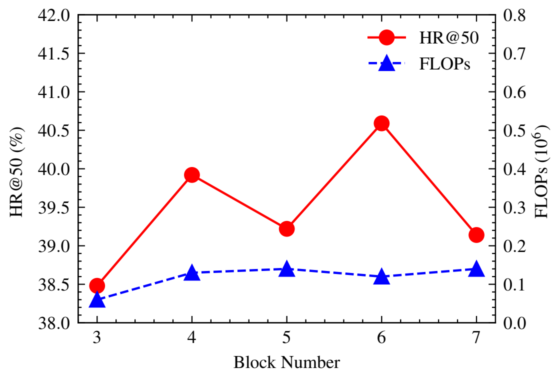
complexity explicitly and flexibly. As a result, AutoTower can find superior and compact architectures with fewer parameters and FLOPs.

4.3.3 Complexity Constraint. To investigate the influence of the complexity constraint in AutoTower, we further evaluate AutoTower with the low complexity constraint and without constraint. The results are shown in Table 7. After removing the constraints, the FLOPs of searched architectures significantly increase. The performance increases with a much more complex architecture on *Yelp*, but it still cannot meet the best performance. The low complexity constraint cannot satisfy *Yelp* due to its sophisticated features. However, the searched architectures do not perform as well as those searched with complexity constraints on all datasets except for *Yelp*, which shows that a complex and redundant model may not provide additional advantages in the retrieval task and imposing complexity constraints is effective for architecture search.

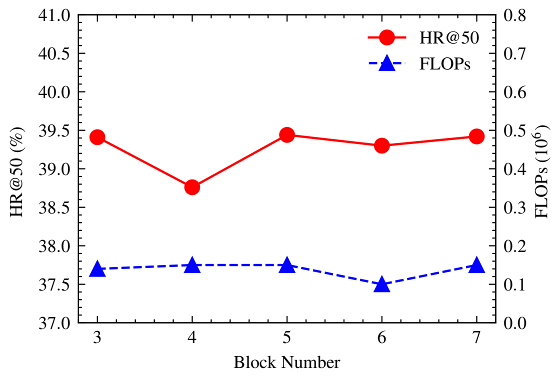
4.4 Hyperparameter Study (RQ3)

4.4.1 The number of Choice Blocks. Figure 5 shows HR@50 and FLOPs of searched architectures with different numbers of choice blocks (i.e., from 3 to 7) under the low complexity constraint. The number of choice blocks means the number of layers in searched architectures. The FLOPs of the searched architectures keep stable as the number of choice blocks increases, due to the explicit complexity constraint. Meanwhile, the performance of AutoTower is also not sensitive to the number of choice blocks.

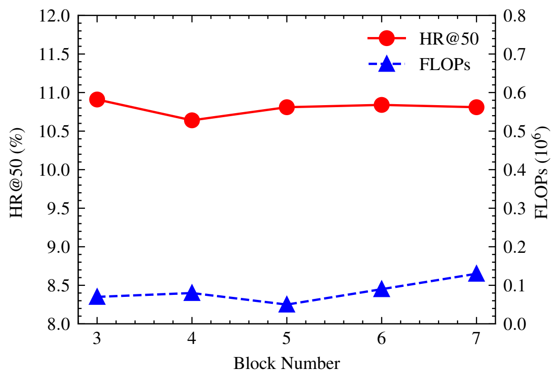
4.4.2 Output Dimension of the Tower. For two-tower models, the output of each tower serves as the user or item representation and will be used to predict the similarity score of the given user-item pair. Therefore, the output dimension of the tower is critical for two-tower models. A higher output dimension is more expressive. But, more space is needed to store item representations. Moreover, the computation of high-dimensional embeddings also slows down the retrieval process during the online service. We conduct experiments to verify the influence of the output dimension on AutoTower. The results are shown in Figure 6. In general, the performance of AutoTower increases with the output dimension, albeit with slight fluctuations, except for the result on the *MovieLens-100K* dataset with small size. Also, due to the massive size, the performance on the *Amazon* dataset is more affected by the output dimension. It demonstrates



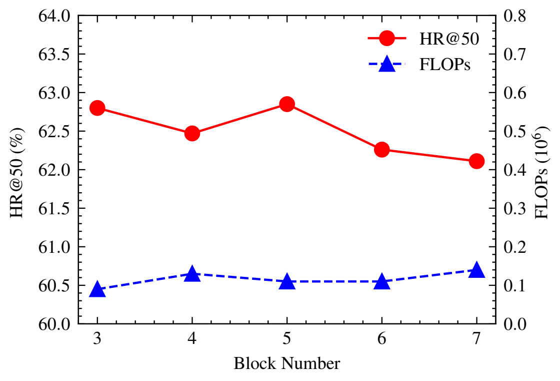
(a) MovieLens-100K



(b) MovieLens-1M



(c) Yelp



(d) Amazon

Fig. 5. HR@50 and FLOPs with different numbers of choice blocks.

that the output of the tower in AutoTower can actually capture the implicit representation of user or item. Considering both performance and storage footprint, we set the output dimension to 64 in other experiments.

4.4.3 Embedding Dimension. The embedding dimension is important to the performance of the recommendation model. As shown in Figure 7, we further verify the influence of embedding dimension on AutoTower. As the embedding dimension grows larger, the performance on *MovieLens-1M* and *Yelp* gradually drop, and the performance on *Amazon* first increases and then significantly drops, and the trend of performance on *MovieLens-100K* is also down, albeit with some volatility. In general, deep recommendation models are prone to suffer from overfitting when the embedding dimension is too large, especially for the dataset with a small size (e.g., *MovieLens-1M*).

4.4.4 Different Constraints for the User Tower and the Item Tower. In real-world application scenarios, we may expect different complexities for the user tower and the item tower, e.g. a more complex user tower for richer user features. AutoTower supports searching for architectures with different complexity constraints for the two towers. We conduct experiments with different constraints for the user tower and the item tower. The constraints are also set according to

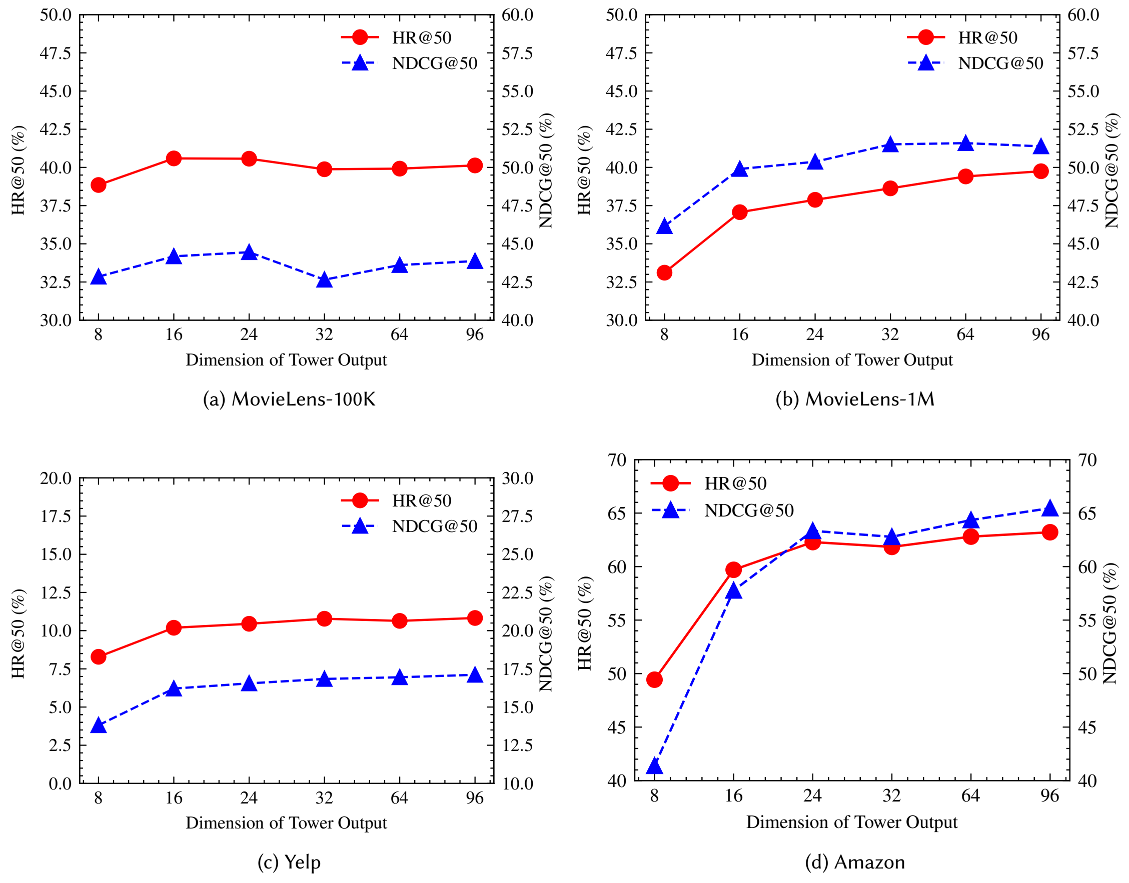
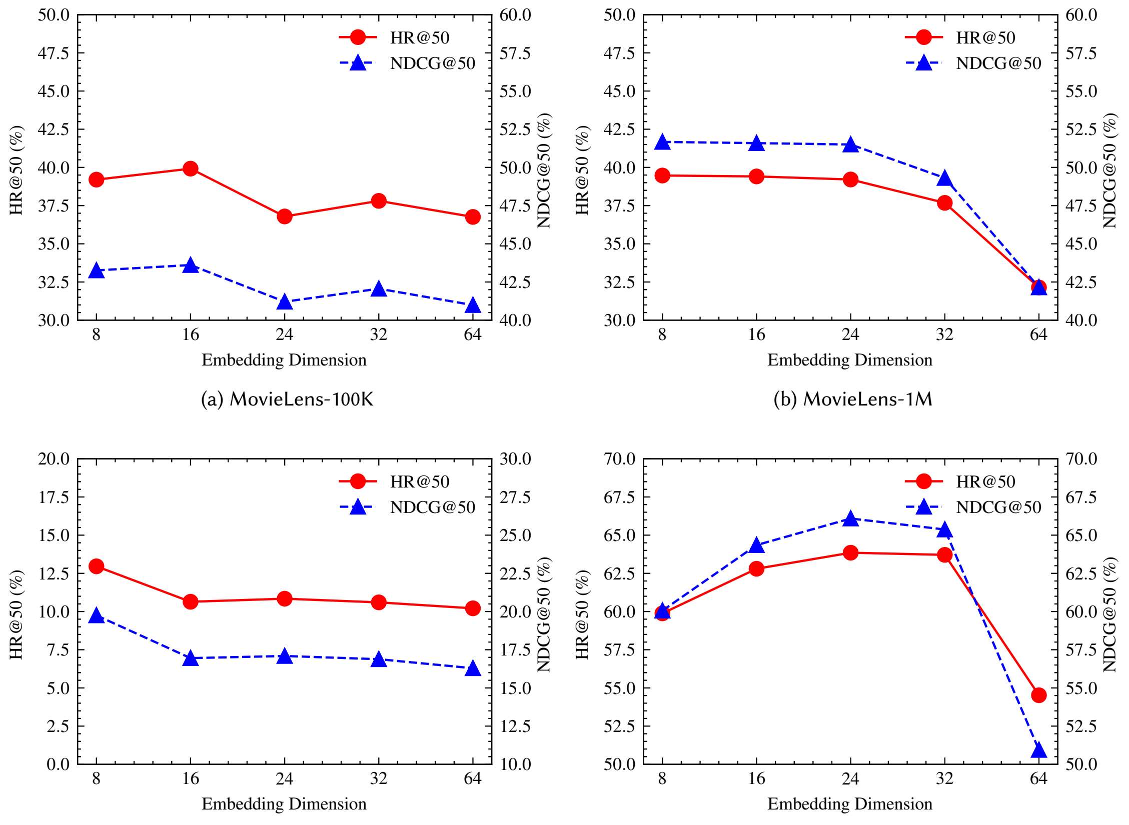


Fig. 6. HR@50 and NDCG@50 with different output dimensions of the tower.

Table 8. Different complexity constraints for the user and item towers. "UP" and "UF" denote the number of parameters and FLOPs of the user tower. "IP" and "IF" correspond to the item tower. The first two lines use "high" constraint for the item tower, and use "low" and "medium" constraints for the user tower. The last two lines use "high" constraint for the user tower, and use "low" and "medium" constraints for the item tower.

Complexity Constraint	MovieLens-1M					Amazon				
	UP	UF	IP	IF	H@50	UP	UF	IP	IF	H@50
User Tower: L	0.04	0.13	0.20	0.11	39.56	0.04	0.05	0.19	0.09	62.71
User Tower: M	0.06	0.15	0.25	0.16	39.54	0.03	0.20	0.03	0.02	59.67
Item Tower: L	0.10	0.17	0.02	0.01	38.76	0.07	0.22	0.02	0.01	61.54
Item Tower: M	0.09	0.18	0.03	0.02	38.97	0.05	0.22	0.04	0.03	60.77

Table 3, but we set constraints for the user tower and the item tower separately, rather than an overall constraint. We only report results on *MovieLens-1M* and *Amazon* here, the results on other datasets are similar. From Table 8, we can



(a) MovieLens-100K (b) MovieLens-1M

(c) Yelp (d) Amazon

Fig. 7. HR@50 and NDCG@50 with different embedding dimensions.

see that all the searched architectures can achieve comparable performance, indicating the robustness of AutoTower. Also, different datasets may prefer different settings. *Amazon* prefers a simpler user tower since it contains limited user features. But *MovieLens-1M* does not show obvious preference.

4.5 Architecture Transferability (RQ4)

To verify the transferability of the searched architectures, we choose the best architecture searched on each dataset to transfer. We re-train the architecture searched on each dataset on other datasets and report the results in Table 9. The searched architectures on all datasets exhibit good transferability. They can also achieve comparable performance on other datasets. The transferability demonstrates that the proposed search space in AutoTower is general enough for the two-tower recommendation tasks.

4.6 Case study

Table 10 shows the best architectures searched by AutoTower, and their performance and complexities. The architecture of each tower is represented as a stack of selected operations. We only present the best architecture searched on

Table 9. Transferability of the searched architectures. HR@50 is reported.

Source \ Target	ML-100K	ML-1M	Yelp	Amazon
Origin	40.28	39.77	12.88	63.33
ML-100K	-	38.61	12.68	60.97
ML-1M	39.63	-	12.36	60.96
Yelp	38.04	38.84	-	63.14
Amazon	38.29	39.13	12.41	-

Table 10. Case study. "IP" is short for Inner Product, and "avg" is short for average.

DataSet	Pooling	User Tower	Parms	FLOPs	Item Tower	Parms	FLOPs	H@50	N@50
ML-100K	SelfAttention	<MLP-512, SA-1, SA-4>	0.10	0.17	<MLP-128, MLP-512, SA-1>	0.12	0.06	40.28	43.21
ML-1M	SelfAttention	<SA-3, EW-IP, MLP-128>	0.06	0.14	<MLP-512, MLP-64, SA-2>	0.09	0.05	39.77	51.84
Yelp	SelfAttentive	<MLP-32, EW-max, EW-avg, EW-IP, EW-min, EW-max, SA-3>	0.07	0.24	<SA-2, EW-IP, EW-max, MLP-1024, EW-avg, SA-2, MLP-256>	0.50	0.26	12.88	19.66
Amazon	SelfAttentive	<SA-1, EW-avg, SA-2, EW-IP, EW-min, MLP-32, SA-1>	0.05	0.06	<SA-2, EW-IP, EW-avg, EW-avg, SA-1, EW-avg, MLP-256>	0.16	0.08	63.33	65.60

each dataset here. AutoTower is a data-specific method, so the resulting architectures on different datasets are very different. Moreover, we can observe that: *SelfAttention* and *SelfAttentive* are common choices of the pooling block, which demonstrates their advantages in processing the user behavior sequence. *Element-Wise* operations that contain few additional parameters and computations are common choices for deeper architectures. These observations, combined with the designed search space in AutoTower, can inspire human experts to design more powerful two-tower models with the considerations of element-wise operations and shortcut connections etc.

5 CONCLUSION AND FUTURE WORK

In this paper, we proposed a NAS-based method called AutoTower to design superior architectures for the two-tower recommendation in a resource-specific and data-specific manner. To our best knowledge, we are the first to utilize NAS to search for two-tower models. Specifically, we first designed a single-path-based search space, which is simple yet general and expressive. The overall search space is mainly composed of a stack of choice blocks. A pooling block is introduced to model user history behaviors. Moreover, we introduced a novel shortcut connection technique to the single-path search space. Then, we proposed an efficient one-shot search strategy based on the single-path search space, which contains a weight-sharing based supernet training stage and a complexity-aware evolutionary search stage. Extensive experiments on four real-world datasets demonstrate that AutoTower can consistently outperform human-crafted models with much fewer parameters and FLOPs. Moreover, the architectures searched by AutoTower are transferable.

In the future, we plan to introduce more effective operations in the search space. We also plan to perform NAS for modeling the user behavior sequence.

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China (#62102177 and #U1811461), the Natural Science Foundation of Jiangsu Province (#BK20210181), the Key R&D Program of Jiangsu Province (#BE2021729), and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

REFERENCES

[1] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. 2016. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167* (2016).

[2] 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).

[3] Qiwei Chen, Huan Zhao, Wei Li, Pipei Huang, and Wenwu Ou. 2019. Behavior sequence transformer for e-commerce recommendation in alibaba. In *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*. 1–4.

[4] 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*. 7–10.

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

[6] Yufei Feng, Fuyu Lv, Weichen Shen, Menghan Wang, Fei Sun, Yu Zhu, and Keping Yang. 2019. Deep session interest network for click-through rate prediction. *arXiv preprint arXiv:1905.06482* (2019).

[7] Chen Gao, Quanming Yao, Depeng Jin, and Yong Li. 2021. Efficient Data-specific Model Search for Collaborative Filtering. *arXiv preprint arXiv:2106.07453* (2021).

[8] Weihao Gao, Xiangjun Fan, Chong Wang, Jiankai Sun, Kai Jia, Wenzi Xiao, Ruofan Ding, Xingyan Bin, Hui Yang, and Xiaobing Liu. 2021. Learning An End-to-End Structure for Retrieval in Large-Scale Recommendations. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 524–533.

[9] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: a factorization-machine based neural network for CTR prediction. *arXiv preprint arXiv:1703.04247* (2017).

[10] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. 2020. Single path one-shot neural architecture search with uniform sampling. In *European Conference on Computer Vision*. Springer, 544–560.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[12] Xiangnan He and Tat-Seng Chua. 2017. Neural factorization machines for sparse predictive analytics. In *Proceedings of the 40th International ACM SIGIR conference on Research and Development in Information Retrieval*. 355–364.

[13] Xiangnan He, Zhankui He, Jingkuan Song, Zhenguang Liu, Yu-Gang Jiang, and Tat-Seng Chua. 2018. Nais: Neural attentive item similarity model for recommendation. *IEEE Transactions on Knowledge and Data Engineering* 30, 12 (2018), 2354–2366.

[14] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*. 173–182.

[15] Jie Hu, Li Shen, and Gang Sun. 2018. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 7132–7141.

[16] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. 2013. Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. 2333–2338.

[17] Tongwen Huang, Zhiqi Zhang, and Junlin Zhang. 2019. FiBiNET: combining feature importance and bilinear feature interaction for click-through rate prediction. In *Proceedings of the 13th ACM Conference on Recommender Systems*. 169–177.

[18] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale similarity search with GPUs. *arXiv preprint arXiv:1702.08734* (2017).

[19] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[20] Chao Li, Zhiyuan Liu, Mengmeng Wu, Yuchi Xu, Huan Zhao, Pipei Huang, Guoliang Kang, Qiwei Chen, Wei Li, and Dik Lun Lee. 2019. Multi-interest network with dynamic routing for recommendation at Tmall. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 2615–2623.

[21] Liam Li and Ameet Talwalkar. 2020. Random search and reproducibility for neural architecture search. In *Uncertainty in artificial intelligence*. PMLR, 367–377.

[22] Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. 2018. xdeepfm: Combining explicit and implicit feature interactions for recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1754–1763.

[23] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. 2017. A structured self-attentive sentence embedding. *arXiv preprint arXiv:1703.03130* (2017).

[24] Greg Linden, Brent Smith, and Jeremy York. 2003. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing* 7, 1 (2003), 76–80.

[25] Bin Liu, Niannan Xue, Huifeng Guo, Ruiming Tang, Stefanos Zafeiriou, Xiuqiang He, and Zhenguo Li. 2020. AutoGroup: Automatic feature grouping for modelling explicit high-order feature interactions in CTR prediction. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 199–208.

[26] Bin Liu, Chenxu Zhu, Guilin Li, Weinan Zhang, Jincai Lai, Ruiming Tang, Xiuqiang He, Zhenguo Li, and Yong Yu. 2020. Autofis: Automatic feature interaction selection in factorization models for click-through rate prediction. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2636–2645.

[27] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055* (2018).

[28] Ze Meng, Jinnian Zhang, Yumeng Li, Jiancheng Li, Tanchao Zhu, and Lifeng Sun. 2021. A General Method For Automatic Discovery of Powerful Interactions In Click-Through Rate Prediction. *arXiv preprint arXiv:2105.10484* (2021).

[29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[30] 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*. PMLR, 4095–4104.

[31] Yanru Qu, Han Cai, Kan Ren, Weinan Zhang, Yong Yu, Ying Wen, and Jun Wang. 2016. Product-based neural networks for user response prediction. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 1149–1154.

[32] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2019. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, Vol. 33. 4780–4789.

[33] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*. Springer, 234–241.

[34] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. 2001. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*. 285–295.

[35] 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 *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 945–955.

[36] Weiping Song, Chence Shi, Zhiping Xiao, Zhijian Duan, Yewen Xu, Ming Zhang, and Jian Tang. 2019. Autoint: Automatic feature interaction learning via self-attentive neural networks. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 1161–1170.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[38] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & cross network for ad click predictions. In *Proceedings of the ADKDD'17*. 1–7.

[39] Saining Xie, Alexander Kirillov, Ross Girshick, and Kaiming He. 2019. Exploring randomly wired neural networks for image recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 1284–1293.

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

[41] Quanming Yao, Ju Xu, Wei-Wei Tu, and Zhanxing Zhu. 2020. Efficient neural architecture search via proximal iterations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 6664–6671.

[42] Xinyang Yi, Ji Yang, Lichan Hong, Derek Zhiyuan Cheng, Lukasz Heldt, Aditee Kumthekar, Zhe Zhao, Li Wei, and Ed Chi. 2019. Sampling-bias-corrected neural modeling for large corpus item recommendations. In *Proceedings of the 13th ACM Conference on Recommender Systems*. 269–277.

[43] Han Zhang, Hongwei Shen, Yiming Qiu, Yunjiang Jiang, Songlin Wang, Sulong Xu, Yun Xiao, Bo Long, and Wen-Yun Yang. 2021. Joint Learning of Deep Retrieval Model and Product Quantization based Embedding Index. *arXiv preprint arXiv:2105.03933* (2021).

[44] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. 2019. Deep interest evolution network for click-through rate prediction. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 5941–5948.

[45] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1059–1068.

[46] Jingwei Zhuo, Ziru Xu, Wei Dai, Han Zhu, Han Li, Jian Xu, and Kun Gai. 2020. Learning optimal tree models under beam search. In *International Conference on Machine Learning*. PMLR, 11650–11659.

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

[48] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 8697–8710.