Learn Basic Skills and Reuse: Modularized Adaptive Neural Architecture Search (MANAS)

Hanxiong Chen
Rutgers University
New Brunswick, NJ, US
hanxiong.chen@rutgers.edu

Yunqi Li
Rutgers University
New Brunswick, NJ, US
yunqi.li@rutgers.edu

He Zhu
Rutgers University
New Brunswick, NJ, US
hz375@cs.rutgers.edu

Yongfeng Zhang
Rutgers University
New Brunswick, NJ, US
yongfeng.zhang@rutgers.edu

ABSTRACT

Human intelligence is able to first learn some basic skills for solving basic problems and then assemble such basic skills into complex skills for solving complex or new problems. For example, the basic skills "dig hole," "put tree," "backfill" and "watering" compose a complex skill "plant a tree". Besides, some basic skills can be reused for solving other problems. For example, the basic skill "dig hole" not only can be used for planting a tree, but also can be used for mining treasures, building a drain, or landfilling. The ability to learn basic skills and reuse them for various tasks is very important for humans because it helps to avoid learning too many skills for solving each individual task, and makes it possible to solve a compositional number of tasks by learning just a few number of basic skills, which saves a considerable amount of memory and computation in the human brain. We believe machine intelligence should also capture the ability of learning basic skills and reusing them by composing into complex skills. In computer science language, each basic skill is a "module", which is a reusable network of a concrete meaning and performs a specific basic operation. The modules are assembled into a bigger "model" for doing a more complex task. The assembling procedure is adaptive to the input or task, i.e., for a given task, the modules should be assembled into the most suitable model for solving the task. As a result, different inputs or tasks could have different assembled models, which enables Auto-Assembling AI.

In this work, we take recommender system as an example and propose Modularized Adaptive Neural Architecture Search (MANAS) to demonstrate the above idea. Neural Architecture Search (NAS) has shown its power in discovering superior neural architectures. However, existing NAS mostly focus on searching for a global architecture regardless of the specific input, i.e., the architecture is not adaptive to the input. In this work, we borrow the idea from modularized neural logic reasoning and consider three basic logical operation modules: AND, OR, NOT. Meanwhile, making recommendations for each user is considered as a task. MANAS automatically assembles the logical operation modules into a network architecture tailored for the given user. As a result, a personalized neural architecture is assembled for each user to make recommendations for the user, which means that the resulting neural architecture is adaptive to the model's input (i.e., the user's past behaviors). Experiments on different datasets show that the adaptive architecture assembled by MANAS outperforms static global architectures. Further experiments and empirical analysis provide insights to the effectiveness of MANAS. The code is open-source at https://github.com/TalonCB/MANAS.

CIKM '22, October 17–21, 2022, Atlanta, GA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9236-5/22/10…$15.00
https://doi.org/10.1145/3511808.3557385

CCS CONCEPTS

• **Computing methodologies → Machine learning**.

KEYWORDS

Neural Architecture Search; Modularized Architecture Search; Adaptive Architecture Search; Personalized Architecture Search; Recommender Systems; Neural-Symbolic; Auto-Assembling AI

ACM Reference Format:
Hanxiong Chen, Yunqi Li, He Zhu, and Yongfeng Zhang. 2022. Learn Basic Skills and Reuse: Modularized Adaptive Neural Architecture Search (MANAS). In *Proceedings of the 31st ACM International Conference on Information and Knowledge Management (CIKM '22), October 17–21, 2022, Atlanta, GA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3511808.3557385

1 INTRODUCTION

Neural Architecture Search (NAS) has emerged as a popular solution for alleviating the pain of designing neural network architectures. Its goal is to automatically discover the optimal deep neural networks based on data-driven learning so that practitioners are provided with ready-made deep neural architectures without expert knowledge, which reduces the burden on manual network design. NAS has become a prevailing research field in various applications such as computer vision [40, 44, 74], natural language processing [14, 40, 44] and recommender systems [38, 52, 67], which has been shown to have the ability of generating deep neural architectures that outperform some human-designed architectures and achieve state-of-the-art performance.

Existing NAS methods mainly focus on searching for a global neural architecture to fit all the data of a problem, which means that all the training data share one global neural network structure. However, the optimal network structure could be different for different data samples. As a result, we hope the network generation process can be adaptive to the input of the data sample. Under the context of recommender systems, each user is considered as a task (i.e., making recommendations for the user), and the input for this task is the user's historical behaviors (i.e., predicting the user's future behavior based on the histories). As a result, the generated network structure should be "personalized" so that the generated network is different for different users.

In this work, we propose a **M**odularized **A**daptive **N**eural **A**rchitecture **S**earch (MANAS) framework to demonstrate the idea of learning basic skills as neural modules and then automatically assemble the modules into a model for solving different tasks. Most importantly, the module assembling process is adaptive to the input of the data sample and the task at hand, since the modules should be assembled in different ways to solve different problems. Following the idea of neural logic reasoning [8, 9, 51], we consider three basic logical operation skills AND, OR and NOT as the basic modules, which are used to assemble complex neural architectures. The simple and meaningful design of the modules makes them a good fit for architecture search and realizing architecture-level personalization.

In Neural Collaborative Reasoning (NCR) [8], these logical modules are organized into a modularized neural architecture according to the input based on a manually defined rule. Instead, our architecture search algorithm aims to automatically search for the optimal combination of the input variables together with the basic logical modules to generate input-adaptive architectures. Technically, we improve from the Efficient Neural Architecture Search (ENAS) [44] model by allowing the input of the data sample as the input for the neural architecture generation process, so that the input of the data sample can steer the generation process (i.e., input-adaptive). One can treat this as a sentence generation model by taking input from the data sample to generate an output sequence. The inputs are from the data sample while the output is the generated neural architecture. In this case, our MANAS model can generate data-specific architectures to provide an architecture-level personalization for users. We apply one-shot training as well as batch training strategies to speed up the training process. Such auto-architecture search mechanism eliminates the limitation of NCR that uses a manually designed global architecture. As a result, our framework can learn to generate more flexible and personalized neural architectures that are adaptive to different user inputs.

Our contributions can be summarized as follows.

- We demonstrate the idea of learning some basic skills as neural modules and then reusing the modules by assembling them into different models for solving different tasks.
- We propose a modularized adaptive neural architecture search framework MANAS, which allows NAS model to generate adaptive input-dependent architectures for each input.
- We design a search space based on logical modules, which allows the neural architecture to be learned from data instead of using a manually crafted global architecture.
- We demonstrate the effectiveness of MANAS on different datasets compared with human-defined models. Analysis of the models generated by MANAS provides insightful guidance for model design.

In the following, Section 2 presents related work, Section 3 reviews neural collaborative reasoning as preliminaries, Section 4 presents our model, Section 5 provides experiments, and Section 6 concludes the work with outlooks for future work.

2 RELATED WORK

2.1 Personalized Recommendation

The idea of personalized recommendation is to tailor the recommended results to a specific user or a set of users [20, 70]. To achieve this goal, researchers designed multiple algorithms to do representation learning on user preferences. Conventional matrix factorization based approaches [31, 47, 49] decompose the user-item rating matrix into two low-rank matrices for identifying latent features. Users' preferences are represented as low-dimensional vectors for calculating the similarity scores with candidate items. As neural network becomes a powerful approach in recommender systems, many research works explore the ways to incorporate deep neural network to capture personalized information. Early pioneer work designed a two-layer Restricted Boltzmann Machine (RBM) to model users' explicit ratings on items [50]. Later, deep learning based recommendation algorithms [8, 15, 66, 69] were proposed to capture non-linear information in user-item interactions. More works have been proposed to apply deep neural networks to learn personalized user representations by capturing various information, such as sequential information [11, 26, 29, 56, 58], image information [10, 12, 23, 41, 69], textual information [25, 33, 34, 62, 69, 71, 72], attribute information [61], knowledge information [18, 19, 24, 59, 60], causal information [35, 36, 55, 63–65] and relational information [9, 17, 37, 54]. Researchers also consider reinforcement learning in recommendation tasks [21, 22, 43]. Some works also explored neural architecture search for recommendation and click-through-rate prediction [38, 52, 67]. Though the aforementioned models, including the architecture search-based models and plenty of works that cannot be enumerated, have different designs, their personalization is mainly reflected on representation-level, i.e., they employ a global model architecture for all users, and personalization is only captured by the user's personalized vector representation.

Recently, neural collaborative reasoning [8, 9, 51, 73] has been proposed to model recommendation tasks as logical reasoning problems. They take logically constrained neural modules to mimic the logical operations in a fuzzy logical space, and then these modules can be organized together with the input variables based on manually crafted rules to formalize the recommendation problem as a logical satisfiability task. Though these works have shown state-of-the-art performance, the architecture design still depends on human domain knowledge, which may require extra human efforts when transferring to different scenarios. Besides, these works still perform personalization on representation-level. Nevertheless, the design of modularized logical network inspires us to explore the possibility of implementing architecture-level personalization, so that the model can be more adaptive and generalizable to new areas without manually designed logical rules.

2.2 Neural Architecture Search

Neural architecture search (NAS) refers to a type of method that can automatically search for superior neural network architectures for different tasks. The main components for designing NAS algorithms include the *search space*, the *search techniques* and the *performance estimation strategy* [16].

Search Space. An ideal search space should contain sufficient distinct architectures while potentially encompassing superior human-crafted structures [52, 74]. Existing work can be roughly classified into the macro architecture search space [1, 6, 7, 28, 46] and the micro cell search space [40, 42, 44, 45, 75]. Macro search space design is to directly discover the entire neural network while micro search space focuses on learning cell structures and assembling a neural architecture by stacking many copies of the discovered cells. The former considers more diversified structures but is quite time-consuming for finding good architectures due to the huge search space. The latter reduces the computation cost but may not target good architectures due to the inflexible network structure.

Search Techniques. The dominant architecture search techniques include random search [5], Bayesian optimization [4], evolutionary algorithms [46], reinforcement learning [44, 74, 75], gradient-based optimization [40]. In recent work [13], the authors shows the effectiveness of combining different types of search techniques to better balance exploitation and exploration.

Performance estimation strategy. When a model is searched, we need a strategy to estimate the quality of this model. Due to the expensive cost of fully training each sampled model, many performance estimation strategies are proposed, such as low-fidelity estimation [45, 75], one-shot algorithm by weight sharing [40, 44], network morphism [46] and learning curve extrapolation [2].

3 PRELIMINARIES

Before formally introducing our MANAS framework, we first briefly review the concepts of neural logical modules in neural collaborative reasoning (NCR) [8, 51], which is a neural-symbolic framework that integrates learning and reasoning. Neural logical modules use three independent multi-layer perceptron (MLP) to represent propositional logic operations AND, OR and NOT, respectively. To grant logic meanings to these modules, they conduct self-supervised learning by adding logical regularization over the corresponding neural logical modules to enable these modules to behave as logic operators. For example, the following logic regularizer is added over the AND module to make it satisfy the idempotence law in propositional logic, i.e., $x \wedge x = x$:

$$r = \frac{1}{|\mathcal{X}|} \sum_{\mathbf{x} \in \mathcal{X}} 1 - Sim(\text{AND}(\mathbf{x}, \mathbf{x}), \mathbf{x}) \tag{1}$$

where x is the input variable; $\mathcal{X}$ represents the input space; $Sim(\cdot, \cdot)$ is the similarity function to measure the distance of two variables, which is the cosine similarity function in [8]. The idea of other logical regularizers corresponding to other logical laws are similar and can be seen in [8, 51].

With these logical modules, the next step is to formalize the user-item interactions as logic expressions, so that the recommendation problem can be transformed into the problem of predicting the probability that a logical expression is true. NCR uses Horn clause to depict the recommendation scenario. Specifically, it predicts the next user-item interaction by taking the clues from the conjunction of a user's historical interactions. For example, if a user u interacted with a set of items $\{v_1, v_2, v_3\}$ and we want to predict if this user would interact with item $\{v_4\}$ in the future, then the question can be translated into the following expression:

$$\mathbf{e}_{u,v_1} \wedge \mathbf{e}_{u,v_2} \wedge \mathbf{e}_{u,v_3} \longrightarrow \mathbf{e}_{u,v_4} \tag{2}$$

where $\mathbf{e}_{u,v_i}$ is the encoded predicate embedding which represents the event of user u interacted with item v_i; "$\longrightarrow$" is the material implication operator[1] in propositional logic. The expression Eq. (2) can be equivalently converted to $\neg\mathbf{e}_{u,v_1} \vee \neg\mathbf{e}_{u,v_2} \vee \neg\mathbf{e}_{u,v_3} \vee \mathbf{e}_{u,v_4}$ by

[1]Material implication ($\rightarrow$) can be represented by basic operations: $x \rightarrow y \Leftrightarrow \neg x \vee y$

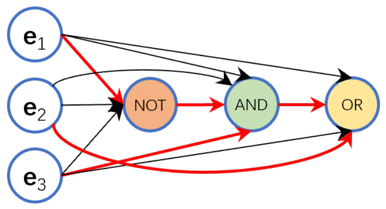


Figure 1: An illustration of an architecture in the designed search space. Red, green and yellow blocks are logical modules; white blocks are raw input variables. The arrows show some potential connections between blocks while the red lines show one possible valid architecture $\neg\mathbf{e}_1 \wedge \mathbf{e}_3 \vee \mathbf{e}_2$.

applying De Morgan's Law[2]. Then the Horn clause can be identically transformed into a neural architecture using the logical neural modules. The output is also a vector which represents the true/false of the expression. The final scoring function is a similarity function such as cosine that can measure the similarity between this output and the true vector $\mathbf{T}$. This true vector is an anchor vector that is fixed after initialization and will not be updated during the training process. It is used to define the logical constant True in the high dimensional vector space. We can then rank items based on these scores to generate ranking lists for users. In this work, we use similar notation $\mathbf{e}_i$ to represent the predicate "item v_i is interacted".

4 MANAS FRAMEWORK

We have briefly introduced the background of neural architecture search, neural logical modules and the challenges of implementing adaptive NAS. In this section, we will give details of our MANAS framework in terms of search space design, search algorithm and prediction/recommendation workflow.

4.1 Search Space Design

In the NAS literature [40, 75], it has been shown that determining the entire architecture of MLP can be extremely time-consuming. Thus, it is preferable to use a tailored but expressive search space that exploits domain-specific knowledge. As we mentioned before, this work can be treated as an extension of NCR, which is to apply NAS techniques and learn to generate modularized logical architectures for each input adaptively, thus realizing architecture-level personalization for the recommendation. In NCR, user historical events are always combined with a conjunction form. However, this form is not flexible and may not always fit the behavior patterns of each user. In our work, we still use the Horn clause to predict a user's future behavior but replace the premise part (i.e. the left part of material implication) from a fixed conjunction architecture to a searched architecture that is adaptive to the user's interaction history. Different from other NAS works which search for hyper-parameters and basic operators such as activation functions in neural architecture, we focus on searching for a valid combination of logical modules and input variables so that a personalized architecture can be searched and dynamically assembled according to the user's history to make recommendations for this user.

As shown in Fig.(1), for a given set of raw input variables $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$ and three logical modules AND, OR and NOT, the model should find

[2]De Morgan's Law: $\neg(x \vee y) \Leftrightarrow \neg x \wedge \neg y; \neg(x \wedge y) \Leftrightarrow \neg x \vee \neg y$

a set of valid connections to combine all the raw inputs together with these logical modules to formulate a valid logical expression. The AND and OR modules are binary operators which take two inputs while NOT is a unary operator with only one input. To cover the cases where there are more than two variables in the expression, we allow the reuse of logical module blocks. We treat the connected modules as a set of directed acyclic graphs (DAGs) by following [44] and each logic module can take as input both raw variables and the intermediate hidden variables, i.e., the variables that are output from other blocks of higher topological order in the DAG.

One challenge is how to make sure the final searched architecture represents a valid logical expression. The most straightforward approach to define the search space is to allow the reuse of both raw input variables and the intermediate hidden variables. In this way, the model can potentially search for any possible combinations of the given input variables and logical modules. However, this would result in an infinite search space which is challenging for search algorithms. On the other hand, if we cannot guarantee to generate a valid logical architecture, the downstream child network cannot be built and trained. To solve the problem, we forbid the reuse of input variables (be it raw or hidden variables). This setting allows our design to be implemented with both correctness and efficiency.

In summary, our search space includes two types of blocks: neural logical module blocks and input variable blocks. Input variable blocks include the raw input variables and the intermediate hidden output variables. These variables are represented as fixed-length vectors and the logical modules as MLPs. The hyper-parameters of these blocks, such as the dimension of vectors, number of layers of MLPs, the activation function to be used, etc., are not considered to be part of the search space. We only search for the superior combination of these blocks to build valid logical architectures for recommendation. We allow the reuse of neural logical module blocks but do not allow the reuse of input variable blocks. For each searched AND or OR logical module block, the model needs to search two-variable blocks as their inputs. Then, for each searched input variable block, we need to determine if this variable needs to couple with a NOT logical module. Once all the variables, except for the final output, are consumed, the search process is done. Since we do not allow the reuse of input variable blocks, the total length of the searched logical expression should be n, where n is the total number of raw input variables. The total number of layers for the entire architecture is $n-1$. To sample architecture for n raw input variables, the total number of distinct architectures in the search space would be:

$$\prod_{i=2}^{n} 4\binom{i}{2}\binom{2}{1} = 4^{n-1}n!(n-1)! \tag{3}$$

which is $O(n!)$ level. The designed search space contains plentiful distinct architectures and can cover most of the valid logic architectures, which allows the model to be adaptive to more recommendation patterns than NCR.

4.2 Search Strategy

NAS usually requires full model training for performance evaluation which is extremely time-consuming. Motivated by the one-shot search algorithms in NAS [6, 44], we allow parameter sharing among architectures so that we can apply one-shot algorithm on MANAS. Specifically, we follow ENAS [44] by using reinforcement learning as the search technique. There are two groups of networks—controller network and child network.

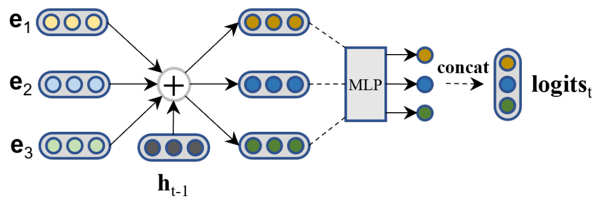


Figure 2: An illustration of the generation process of logits for input variable prediction at time t; $\mathbf{e}_i$ represents predicate vector; $\mathbf{h}_{t-1}$ is the hidden state from the t-1 step; "+" means element-wise addition.

4.2.1 ***Controller Network.*** Controller network is an LSTM [27] which samples architecture modules via softmax classifiers. In ENAS, at each step, the controller takes the hidden state for the previous sampled module as input and makes the next prediction. The entire search process does not involve raw input variable vectors, which makes the architecture generation process independent from the input data. Different from ENAS, as illustrated in Fig.(2), our controller network considers both the hidden states from the previous step and the input variable representations when making decisions. We assign a unique vector to each raw input variable and this is the reason why we treat each raw input variable as a block. For example, at time t, we have predicate vectors $\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$ as the input candidates. We first merge the information from the hidden state and the input vectors by doing element-wise addition. Then these enriched vectors are sent to a mapping function whose output is a scalar. Each scalar can be treated as the logit of its corresponding input variable. We concatenate these logits to form a logits vector $\mathbf{logits}_t$. This process can be represented as

$$\begin{aligned} l_i &= \mathbf{W}_1(\mathbf{e}_i + \mathbf{h}_{t-1}) + b_1 \\ \mathbf{logits}_t &= \text{concat}(l_1, l_2, \ldots, l_i) \quad \forall i \in C_t \end{aligned} \tag{4}$$

where l_i is the logit for candidate $\mathbf{e}_i$; $\mathbf{W}_1 \in \mathbb{R}^{n\times 1}$ is a matrix to map the input vector into a scalar; b_1 is the bias term; C_t represents the collection of candidates at current step. Then this $\mathbf{logits}_t$ is sent to the Softmax function. Each dimension of the normalized logits vector represents the probability of a specific variable to be chosen as the input at the current time t. The prediction process of logical module is similar to the variable prediction process. The only difference is that we only use the hidden state from LSTM to create the logits vector for predicting the next module.

Once we obtain the predicted input variables, the controller needs to decide if any of these inputs should be coupled with a NOT module to be represented as the negation form in logic expression. We take the logits of the selected input variables from the previous step to create a vector for NOT module prediction. For example, if we select $\mathbf{e}_i$ as one of the input at step t, then we can calculate the logit l_i by equation Eq. (4). Then the new logit vector of $\mathbf{e}_i$ for NOT module prediction is $[-l_i, l_i]$, where the two dimensions represent do or do not couple with a NOT module, respectively. This is similar to what ENAS does for predicting skip connections. An example of the aforementioned sampling process is illustrated in Fig.(3).

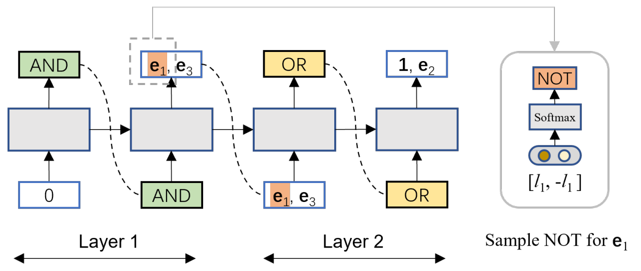


Figure 3: An illustration of the sampling process of logical architecture $(\neg\mathbf{e}_1 \wedge \mathbf{e}_3) \vee \mathbf{e}_2$. $\mathbf{e}_i$ represents predicate embedding; the first "0" means we use zero vector as the initial hidden state for LSTM; "1" in the last generation step represents the output hidden state from layer 1. The red background represents that the corresponding input vector is sent to the NOT module to get the negation form.

4.2.2 ***Child Network****.* Child network is the searched network for the recommendation task. Since our approach is adaptive, for each input sequence of user interaction history, the controller network will first generate a data-specific architecture to organize the history sequence into a logic expression. Then we apply the Horn clause of the logical modules and input vectors to calculate the score for each candidate item. All the parameters of the child network are shared among all the sampled architectures.

4.3 MANAS Training and Deriving

We have introduced the adaptive architecture search strategy and gave a brief illustration of the child network formalization process. To make it more clear and easy to understand, in this subsection, we will go through the training process of MANAS.

In MANAS, the parameters of the controller network, denoted by θ, and the shared parameters of the child network, denoted by ω, are two groups of learnable parameters. These two networks are trained interleaved. First, we fix the controller's policy π_θ and train the child network on a whole pass of the training data, i.e. train one epoch. After that, we fix the parameters of the child network and train the controller's policy network parameters. To make the training process efficient, we apply one-shot searching algorithm since our child network parameters ω are shared. Additionally, we do not train child networks from scratch at each iteration. Instead, we update ω on top of the results of the last iteration. The training algorithm is given in Algorithm 1, and more details about MANAS training are as follows.

4.3.1 ***Child Network Training****.* In the recommendation task, we use a sequence of user interacted items as the input to predict future interactions. For each item v_i in the item set $\mathcal{V}$, we assign a unique vector $\mathbf{e}_i^\omega \in \mathbf{E}_\omega$ to represent the predicate "item i is interacted," where $\mathbf{E}_\omega$ represents the set of all the predicate vectors in the child network parameter space.

Under the settings of neural collaborative reasoning [8], the recommendation task is to answer the question "(?) $\rightarrow v_k$," where (?) is the premise logic expression that contains the input sequence and v_k is a candidate item. Given a sequence $s = \{v_1, v_2, v_3\}$, the controller network will sample an architecture, e.g., $\neg\mathbf{e}_1^\omega \wedge \mathbf{e}_3^\omega \vee \mathbf{e}_2^\omega$, from π_θ to convert the input into a logic expression with an empty embedding as the initial input. Then we replace "(?)" with this

Algorithm 1: MANAS Training Algorithm

```
Input :Controller parameters $\theta$; child logical network
       shared parameters $\omega$; performance evaluator $\mathcal{E}$;
       training steps of controller network $K$; training
       epochs $t$; training data $\mathcal{D}_{train}$; validation data
       $\mathcal{D}_{valid}$; optimizer OPTIM
1  Initialize controller parameters $\theta$ and child logical network
     parameters $\omega$;
2  for $epoch \leftarrow 1$ to $t$ do
3      sample architectures $\mathcal{M}_{train}$ from
         $\pi(s;\theta), \quad \forall s \in \mathcal{D}_{train}$;
4      train child logical network with $\mathcal{M}_{train}$ on $\mathcal{D}_{train}$;
5      for $k \leftarrow 1$ to $K$ do
6          prepare $Batch$ from $\mathcal{D}_{valid}$;
7          for $b \in Batch$ do
8              sample architectures $\mathcal{M}_b$ from
                 $\pi(s';\theta), \quad \forall s' \in Batch$;
9              evaluate model with $\mathcal{E}$ to obtain reward $\mathcal{R}$;
10             update controller parameters $\theta$ with OPTIM;
11         end
12     end
13 end
```

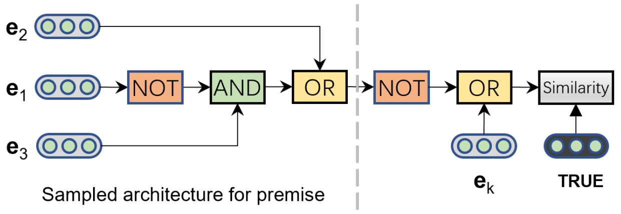


Figure 4: An illustration of a logical network for predicting the probability of item k being interacted (represented by predicate $\mathbf{e}_k$) with a sequence of input predicates $\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$

searched architecture and replace v_k with its predicate vector $\mathbf{e}_k^\omega$ to get the expression $\neg\mathbf{e}_1^\omega \wedge \mathbf{e}_3^\omega \vee \mathbf{e}_2^\omega \rightarrow \mathbf{e}_k^\omega$. Then we follow NCR to build and train the network by converting the expression into $\neg(\neg\mathbf{e}_1^\omega \wedge \mathbf{e}_3^\omega \vee \mathbf{e}_2^\omega) \vee \mathbf{e}_k^\omega$. The entire logical network of the given example is presented in Fig.(4). We use the BPR-loss [47] by sampling one user non-interacted item as the negative sample for each ground-truth item to train the child network. We also use recommendation metrics such as hit ratio and normalized discounted cumulative gain (NDCG) as the performance evaluation measure.

4.3.2 ***Controller Network Training****.* In the controller network parameter space, we also assign a unique vector $\mathbf{e}_i^\theta \in \mathbf{E}_\theta$ for each item $v_i \in \mathcal{V}$. Though $\mathbf{e}_i^\theta$ and $\mathbf{e}_i^\omega$ are independent, they both represent the predicate "item v_i is interacted". This can be a bridge to connect the controller network and the child network so that the generated architecture can be adaptive to the input sequences. When training the controller network, we fix ω and update the policy network parameters θ. Our goal is to maximize the expectation of reward $\mathbb{E}_{m\sim\pi(s;\theta)}[\mathcal{R}(m,\omega,s)]$, where m represents a sampled child network; s is the input sequence. The reward $\mathcal{R}(m,\omega,s)$ is computed on the validation set. In our experiments, the reward function is the summation of hit ratio and NDCG on a minibatch of the validation set. We treat one whole pass of the validation set as one training step for the controller network and we train 50 steps per iteration. We employ the Adam optimizer [30], for which the gradient is computed using REINFORCE, with a moving average baseline to reduce variance.

4.3.3 *Deriving Architecture.*

In ENAS, it only maintains the learned controller network parameters and discard the child network parameters when deriving new architectures. They sample a set of new architectures for validation and select the one with the best performance to train a new child network from scratch. Different from ENAS, we treat both the controller network and the child network as the components of MANAS. In testing time, when a new sequence s' is given, we directly sample a model $m_{s'}$ from the trained policy $\pi(s';\theta)$. Then the child network $m_{s'}$ is assembled based on the sampled architecture by using the trained shared logical modules and predicate vectors.

5 EXPERIMENTS

In this section, we evaluate MANAS as well as several baseline models for the top-K ranking problem on a set of publicly available real-world recommendation datasets. We aim to answer the following research questions:

- **RQ1**: Can MANAS search for architectures that outperform state-of-the-art human-crafted models for recommendation?
- **RQ2**: In MANAS, the searched architecture is adaptive to the input sequence. Is this adaptive nature important for improving recommendation quality?
- **RQ3**: The architecture sampling process in MANAS involves exploration since the modules are sampled according to logits probability rather than directly selecting the module of the largest probability (i.e., exploitation). Is exploration necessary for training MANAS?
- **RQ4**: What about the efficiency of MANAS?
- **RQ5**: How does the recommendation performance and the training time cost change w.r.t the input sequence length?

5.1 Experimental Settings

5.1.1 *Dataset.*

In the experiments, we use the publicly available Amazon e-commerce dataset[3], which includes the user, item and rating information. We take **Beauty**, **Cell Phones and Accessories** and **Grocery and Gourmet Food** sub-categories for our experiments to evaluate the performance of our neural architecture search algorithm. Statistics of the datasets are shown in Table 1.

5.1.2 *Evaluation Protocol.*

We use leave-one-out strategy to split the dataset into train, validation and test, which is a widely used approach in the literature [8, 51]. For each input sequence s_u from a user u, we use the most recent interaction of each user for testing, the second recent item for validation, and the remaining items for training. As the searched neural architectures are used for ranking tasks, for efficiency consideration, we use *real-plus-N* [3, 48] to calculate the measures in the validation and testing stage. More specifically, for each user-item pair in the validation and test set, we randomly sample 99 user non-interacted items, and we rank these 100 items for the ranking evaluation.

[3]https://jmcauley.ucsd.edu/data/amazon/

Table 1: Statistics of the datasets in our experiments.

Dataset	#Users	#Items	#Interaction	Density
Beauty	22,363	12,101	198,502	0.073%
Cellphones	27,879	10,429	194,439	0.067%
Grocery	14,681	8,713	151,254	0.118%

As for the evaluation metrics, we select a recall-based metric Hit Ratio (Hit@K) and a rank-position based metric normalized discounted cumulative gain (NDCG@K) for evaluating the performance of recommendation. The reported results of all metrics are averaged over all users.

5.1.3 *Hyper-parameters Settings.*

For our MANAS model, the length of the input sequence is 4, i.e., each user-item pair comes with 4 histories. In practice, one can set this number to any value but our experiments show that 4 history is good enough for our recommendation tasks. We will also explore the effect of different input lengths in Section 5.7. For the baseline models, we allow sequence length up to 20. The embedding size is set to 64 for both the child network and the controller LSTM hidden vectors. We use Adam [30] to optimize model parameters. ℓ_2 regularization is adopted to prevent child network from overfitting and the weight of ℓ_2 is set to 10^{-5}. The logic regularization weight is set to 10^{-5}. The learning rate for the child network is 0.001 while the controller learning rate is 0.005.

5.2 Baselines

Since our model requires interaction sequence as input, we select the following representative sequential recommendation baselines to verify the effectiveness of our method.

- **GRU4Rec** [26]: This is a sequential/session-based recommendation model, which uses Recurrent Neural Network (RNN) to capture the sequential dependencies in users' historical interactions for recommendations.
- **NARM** [32]: This model utilizes GRU and the attention mechanism to consider the importance of interactions .
- **Caser** [56]: This is a convolutional neural network based sequential model, which learns sequential patterns using vertical and horizontal convolutional filters.
- **SASRec** [29]: This model uses transformer to capture the left-to-right context information from historical interactions.
- **BERT4Rec** [53]: This model utilizes a bi-directional self-attention module to capture context information in user behavior sequences from both left-to-right and right-to-left.
- **NCR** [8]: This is a state-of-the-art neural logic reasoning based recommendation framework. It utilizes logic reasoning to model recommendation tasks.
- **NANAS**: This is the non-adaptive version of our NAS model. It is used to replicate regular NAS which learns a global architecture for all inputs on a specific task.

The baseline implementations are from an open-source recommendation toolkit [57]. We select the best model based on its best performance on the validation set. We implement the models with PyTorch v1.8 and the models are trained on a single 2080Ti GPU.

Table 2: Results of recommendation performance on three datasets with metrics NDCG (N) and Hit Ratio (HR). We use underline (number) to show the best result among the baselines. We use bold font to mark the best result of the whole column. We use one star (*) to indicate that the performance is significantly better than the best non-NAS based baselines, and use two stars (**) to indicate that the performance is significantly better than all baselines including NANAS. The significance is at 0.01 level based on paired t-test. Improvement[1] shows our model improvement over the best result (i.e., over number), while improvement[2] shows our model improvement over NCR.

	Beauty				Cellphones				Grocery			
	N@5	N@10	HR@5	HR@10	N@5	N@10	HR@5	HR@10	N@5	N@10	HR@5	HR@10
GRU4Rec	0.1656	0.2012	0.2534	0.3637	0.1766	0.2101	0.2575	0.3616	0.2000	0.2292	0.2914	0.3816
NARM	0.1900	0.2200	0.2656	0.3584	0.2008	0.2387	0.2923	0.4101	0.2260	0.2541	0.3129	0.3999
Caser	0.2107	0.2416	0.2912	0.3874	0.2054	0.2412	0.2937	0.4052	0.2313	0.2603	0.3191	0.4088
SASRec	0.2599	0.2861	0.3275	0.4091	0.2458	0.2767	0.3275	0.4230	0.2342	0.2658	0.3379	0.4361
BERT4Rec	0.2283	0.2584	0.3102	0.4034	0.2536	0.2908	0.3503	0.4655	0.2420	0.2727	0.3356	0.4310
NCR	0.1883	0.2163	0.2551	0.3420	0.1819	0.2140	0.2604	0.3602	0.2210	0.2490	0.3140	0.4010
NANAS	0.2519	0.2820	0.3383	0.4316	0.2534	0.2915	0.3500	0.4682	0.2462	0.2731	0.3574	0.4402
MANAS	**0.2618**	**0.2933****	**0.3532****	**0.4507****	**0.2785****	**0.3149****	**0.3846****	**0.4971****	**0.2609****	**0.2882****	**0.3598***	**0.4444***
Improvment[1]	0.73%	2.52%	4.40%	4.43%	9.82%	8.03%	9.79%	6.17%	5.97%	5.53%	0.67%	0.95%
Improvment[2]	39.03%	35.60%	38.46%	31.78%	53.11%	47.15%	47.70%	38.01%	18.05%	15.74%	14.59%	10.82%

5.3 Recommendation Performance (RQ1)

We report the overall recommendation ranking performance of all the models in Table 2. The results show that our MANAS consistently outperforms all the baseline models. From these results, we have the following observations.

(1) NCR outperforms GRU4Rec in most cases on all the datasets and achieve a comparable performance with NARM and Caser on *Grocery* dataset. However, on *Beauty* and *Cellphones* datasets, Caser has a much better performance than NCR. This observation indicates that the manually designed architectures are lack of adaptation ability especially for neural logic reasoning. According to the findings in NCR, a correct form of the logic expression could affect the recommendation quality. However, designing a data-specific architecture is non-trivial and requires excessive human efforts, which corroborates the necessity of designing an data-adaptive neural architecture search algorithm.

(2) MANAS can consistently outperform all the baselines. By comparing MANAS and NCR, the significant improvement on all the metrics show that learning to generate architectures does help neural logic reasoning to be more adaptive to various inputs.

5.4 Global vs. Adaptive (RQ2)

We claimed that it is important to design an adaptive NAS algorithm to realize architecture-level personalization for recommendation tasks. To verify the importance of this adaptive feature, we use a non-adaptive NAS, called NANAS. This is to replicate a regular NAS algorithm that learns a global architecture for all inputs on a specific task. The difference between MANAS and NANAS lies in the controller part. In NANAS, the predicate embeddings of the raw input variables do not involve in the search process. Instead, all the input variables $\mathbf{e}_i$ are replaced with position embedding $\mathbf{p}_i$, where $\mathbf{p}_i$ is the representation of the i-th position in the logic expression. For example, suppose we are given two input sequences $s_1 = \{v_1, v_2, v_3\}$ and $s_2 = \{v_4, v_5, v_6\}$, MANAS search space includes the predicate embedding of the input variables $\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \mathbf{e}_4, \mathbf{e}_5, \mathbf{e}_6\}$ as well as logical modules AND, OR, NOT. However, in NANAS, the search space only contains the position embedding $\{\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3\}$ and three logical modules. Since the search space does not relate to the input variables, the searched architecture is a global architecture that fit all the data of the given task. Here we have two observations from the NANAS results in Table 2:

(1) By comparing NANAS with NCR, we see that NANAS can consistently outperform NCR on all the tasks. Such results tell us that a searched neural logic architecture is better than a human-designed architecture.

(2) By comparing NANAS with MANAS, we observe that MANAS compete NANAS on all the datasets over all the metrics. It empirically shows that an adaptive neural architecture is important for improving personalized recommendation quality.

We illustrate the model generated architectures in Fig.(6). It shows that our model can generate diverse architectures to adapt to different inputs. It helps neural logic reasoning to generalize to different inputs and logical expressions without the pain of manually designing suitable logical architectures.

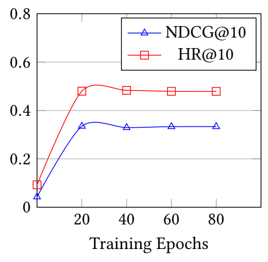
5.5 Importance of Exploration (RQ3)

Exploration is one of the most important concepts in reinforcement learning. Without sufficient exploration, the learning process may not able to converge to a good solution. To verify the importance of exploration, we test a non-sampling version of MANAS.

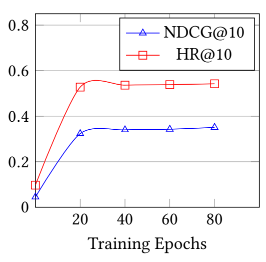
As mentioned in the previous section, the prediction of the next module is based on the probabilities in the logits vector, i.e., we treat all the probabilities as a distribution π_θ and the next module is sampled according to this distribution. The probability of each module being sampled at the current step is its corresponding probability value in the logits vector. This allows the searcher to explore potential cases that are not the best in the current step but may bring large rewards in the future, so as to avoid being trapped into a local optimal. To test the importance of exploration based on such sampling strategy, we test a non-sampling version of MANAS by doing the greedy selection, which always chooses the module with the max current probability as the next predicted module.

Table 3: Results of ranking performance in terms of average, min and max over 20 sampled architectures for each input sequence with MANAS. Non-sample model represents the performance of MANAS without exploration in the searching process. We mark MANAS_{AVG} in bold to show our model consistently outperforms non-sample model. STD is the standard deviation of the reported results on the 20 sampled architectures.

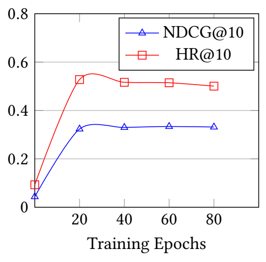
	Beauty				Cellphones				Grocery			
	N@5	N@10	HR@5	HR@10	N@5	N@10	HR@5	HR@10	N@5	N@10	HR@5	HR@10
NCR	0.1883	0.2163	0.2551	0.3420	0.1819	0.2140	0.2604	0.3602	0.2210	0.2490	0.3140	0.4010
Non-Sample	0.2528	0.2819	0.3366	0.4269	0.2627	0.2987	0.3615	0.4734	0.2373	0.2674	0.3456	0.4378
MANAS_{AVG}	**0.2618**	**0.2933**	**0.3532**	**0.4507**	**0.2785**	**0.3149**	**0.3846**	**0.4971**	**0.2609**	**0.2882**	**0.3598**	**0.4444**
MANAS_{MIN}	0.2612	0.2923	0.3523	0.4485	0.2775	0.3138	0.3830	0.4951	0.2596	0.2869	0.3584	0.4432
MANAS_{MAX}	0.2638	0.2954	0.3545	0.4526	0.2791	0.3155	0.3859	0.4980	0.2621	0.2893	0.3619	0.4462
STD	0.0007	0.0008	0.0010	0.0013	0.0006	0.0005	0.0010	0.0008	0.0007	0.0008	0.0011	0.0013



(a) Beauty



(b) Cellphones



(c) Grocery

Figure 5: An illustration of the ranking performance changing with the training epochs.

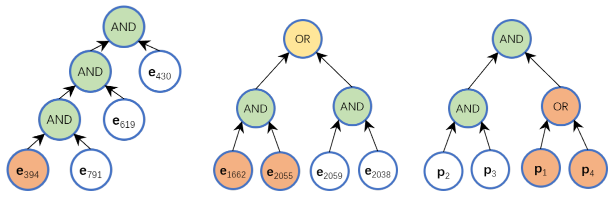


Figure 6: Examples of generated architectures on Beauty dataset. MANAS searched architectures (left) for the input sequence $\{e_{384}, e_{791}, e_{619}, e_{430}\}$ and (middle) for the input sequence $\{e_{1662}, e_{2055}, e_{2059}, e_{2038}\}$; NANAS generated the global architecture (right). The red-colored module means that its output should be negated as the input to its successor module.

From the results in Table 3, we find that MANAS has a significant improvement against the non-sampling model. This result shows the importance of doing exploration in the search process.

Since the exploration brings uncertainty in the architecture generation process, we sample 20 different architectures from the trained model for each input sequence and evaluate their performance. The results in Table 3 are the averaged results over the 20 samples. The small standard deviation reported in the table shows that our model can provide relatively consistent performance.

5.6 Efficiency Analysis (RQ4)

NAS training is time consuming especially when it comes to the adaptive neural architecture search. It is challenging to do batch training when the network architectures are different. To tackle this problem, we engineer the training of MANAS into two steps. Take the child network training process as an example. We first send the training data $\mathcal{D}_{train}$ into the controller to generate architectures for all the samples $s \in \mathcal{D}_{train}$. Since we limit the length of history to be exactly n for each user-item interaction, we can guarantee that each sequence contains n raw input variables. We assign a unique position index to each of the variables for each sequence. During the architecture generation stage, the sampler samples the position index for each input sequence and we can then use these position indices to assemble logical networks. For example, if we have two sequences $s_1 = \{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \mathbf{e}_4\}$ and $s_2 = \{\mathbf{e}_5, \mathbf{e}_6, \mathbf{e}_7, \mathbf{e}_8\}$, we assign position index 0 to $\mathbf{e}_1$ and $\mathbf{e}_5$, 1 to $\mathbf{e}_2$ and $\mathbf{e}_6$, etc. When the controller generates the same sequence for both s_1, s_2 with $\{\text{AND}, 0, 1; \text{OR}, 2, 4; \text{AND}, 3, 5\}$, we can use the position index to locate the variable to create each layer of the logical network. Take the first sequence s_1 as an example, the corresponding logical network is $((\mathbf{e}_1 \wedge \mathbf{e}_2) \vee \mathbf{e}_3) \wedge \mathbf{e}_4$.

After obtaining all the architectures, we can group the sequences based on their sampled architectures. We can do batch training on those data that have the same architecture. We set the batch size to 256 for child network training. This number is an upper bound for the batch size. It is possible to have some architecture pattern groups whose number of sequences is less than the batch size. In that case, the minimum size of batches depends on the smallest group size of architecture patterns. For the controller training process, we apply the same approach to enable batch training. By setting the architecture generation batch to 1024 and the child network training batch to 256, we can achieve at least 5 times speed boost than non-batch training. To show the effectiveness of our batch training design, we report the change of recommendation performance in terms of training epochs in Fig.(5). The results are reported based on the validation set and the sequence length is set to 4. We see that our model converges to a relatively optimal solution in around

30 epochs on all three datasets. To speed up the training process, we allow batch training for both controller and the searched child network. For each training epoch, we train the child network using only one whole pass of the training data and then train the controller network for 50 steps. Under these settings, one training epoch costs about 1.2 hours.

During the inference stage, our model can generate the recommendation results efficiently. Specifically, the average inference time is 0.995 ± 0.054 millisecond(ms), with 0.43 ± 0.14ms for sampling the architecture and 0.57 ± 0.06ms for generating the prediction. For reference, the inference time of SASRec is 0.67 ± 0.04ms and BERT4Rec is 0.70 ± 0.02ms. All the reported inference running time are based on a single record prediction.

5.7 Influence of Sequence Length (RQ5)

The length of input sequence could affect not only the recommendation performance but also the training time. In this subsection, we discuss how the recommendation performance and training time change with respect to the sequence length. We conduct 5 experiments by choosing the input sequence length from $\{2, 4, 6, 8, 10\}$. For example, we limit each training sample to have exactly 4 historical interactions when we refer to the sequence length as 4. One problem is that we cannot guarantee the five length settings will give us the same set of users. For example, a user with exactly 4 histories will not appear in the 6-history setting. In that case, we may have different training data for different sequence length experiments. To guarantee that all the comparisons are reasonable by training and evaluating models on the same data, we first filter the original dataset by only keeping those interactions that have at least 10 histories. Then we cut off the history sequence of each interaction for different sequence length experiments. In this case, we can guarantee that the performance as well as the training time differences only in the sequence length. Since the dataset used in this experiment is a filtered one, the training time and the performance reported in this subsection may be slightly different from previous results. The results are reported in Fig.(7).

From Fig.(7a), we have two observations. First, we find that the model can get recommendation performance gain by reasonably increasing the sequence length. This is because a longer sequence can potentially bring more information to help model capture user preferences. However, our second observation is that we cannot continue to get benefits from increasing the history length. Contrarily, the performance could be harmed when the sequence is too long. One reason is that a longer sequence could introduce noisy information to the model because some very early user histories may have very limited contribution or even be irrelevant to the current recommendation. In this case, a longer sequence does not help to improve the recommendation quality. On the other hand, the noise in the sequence could even result in the performance loss.

In Fig.(7b), we plot the training time per epoch with respect to different sequence lengths. It shows that the training time is closely related with the sequence length. This is because a longer sequence represents a larger search space. By observing Fig.(7a) and (7b) together, we can see that we need to carefully set the history length for MANAS. The limited information from the short sequence may prevent the model from capturing user preferences, thus resulting

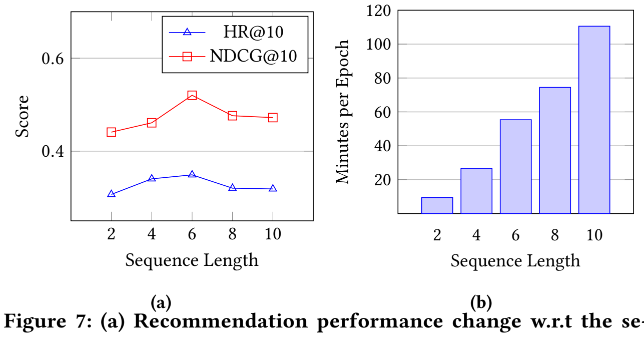


Figure 7: (a) Recommendation performance change w.r.t the sequence length; (b) The histograms represent the architecture search time costs w.r.t different input sequence lengths. The time is given in minutes per epoch.

in low quality recommendations. However, longer sequence does not always help to improve the model performance. After reaching a certain threshold, longer sequences may have very limited or even no contribution to improve the performance but can greatly increase the training time.

6 CONCLUSIONS AND FUTURE WORK

In this work, we propose to learn basic skills as neural modules and automatically assemble them into different models for solving a compositional number of different problems. Technically, we use intelligent recommender system as an example to demonstrate the idea and propose a Modularized Adaptive Neural Architecture Search (MANAS) framework, which automatically assembles logical operation modules into a reasoning network that is adaptive to the user's input sequence, and thus advances personalized recommendation from the learning of personalized representations to the learning of personalized architectures for users. We enable neural-symbolic reasoning to generate flexible logical architectures, which make logical models adaptive to the diverse inputs without using human-crafted logical structure. The experimental results show that our design can provide significantly better prediction accuracy. Besides, we also conduct experiments to show the importance of exploration in the architecture search process and the importance of learning adaptive architectures for the prediction and recommendation tasks.

In this work, we considered three neural modules to demonstrate the idea of modularized adaptive neural architecture search. However, the proposed framework is general and can incorporate more neural modules such as neural predicates or other basic vision, language or recommendation modules, which we will explore in the future. Furthermore, in addition to the performance improvement, the problem-adaptive architecture generated by our framework may improve the model transparency, provide model explainability [39, 70, 72], and enable automatic learning to define problems [68], which are important and promising to explore in the future.

ACKNOWLEDGMENTS

This work was supported in part by NSF IIS-1910154, IIS-2007907, IIS-2046457 and CCF-2124155. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsors.

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] Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. 2017. Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823* (2017).

[3] Alejandro Bellogin, Pablo Castells, and Ivan Cantador. 2011. Precision-oriented evaluation of recommender systems: an algorithmic comparison. In *Proceedings of the fifth ACM conference on Recommender systems*. 333–336.

[4] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems* 24 (2011).

[5] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of machine learning research* 13, 2 (2012).

[6] Andrew Brock, Theo Lim, JM Ritchie, and Nick Weston. 2018. SMASH: One-Shot Model Architecture Search through HyperNetworks. In *International Conference on Learning Representations*.

[7] Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. 2018. Efficient architecture search by network transformation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.

[8] Hanxiong Chen, Shaoyun Shi, Yunqi Li, and Yongfeng Zhang. 2021. Neural Collaborative Reasoning. In *Proceedings of the Web Conference 2021*. 1516–1527.

[9] Hanxiong Chen, Li Yunqi, Shi Shaoyun, Shuchang Liu, He Zhu, and Yongfeng Zhang. 2022. Graph Collaborative Reasoning. In *Proceedings of the 15th ACM International Conference on Web Search and Data Mining*.

[10] Xu Chen, Hanxiong Chen, Hongteng Xu, Yongfeng Zhang, Yixin Cao, Zheng Qin, and Hongyuan Zha. 2019. Personalized fashion recommendation with visual explanations based on multimodal attention network: Towards visually explainable recommendation. In *SIGIR*. 765–774.

[11] Xu Chen, Hongteng Xu, Yongfeng Zhang, Jiaxi Tang, Yixin Cao, Zheng Qin, and Hongyuan Zha. 2018. Sequential recommendation with user memory networks. In *Proceedings of the eleventh ACM international conference on web search and data mining*. 108–116.

[12] Xu Chen, Yongfeng Zhang, Qingyao Ai, Hongteng Xu, Junchi Yan, and Zheng Qin. 2017. Personalized key frame recommendation. In *Proceedings of the 40th international ACM SIGIR conference on research and development in information retrieval*. 315–324.

[13] Yukang Chen, Gaofeng Meng, Qian Zhang, Shiming Xiang, Chang Huang, Lisen Mu, and Xinggang Wang. 2019. Renas: Reinforced evolutionary neural architecture search. In *CVPR*. 4787–4796.

[14] Yi-Wei Chen, Qingquan Song, and Xia Hu. 2021. Techniques for automated machine learning. *ACM SIGKDD Explorations Newsletter* 22, 2 (2021), 35–50.

[15] 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.

[16] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural architecture search: A survey. *The Journal of Machine Learning Research* 20, 1 (2019), 1997–2017.

[17] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph neural networks for social recommendation. In *WWW*. 417–426.

[18] Zuohui Fu, Yikun Xian, Ruoyuan Gao, Jieyu Zhao, Qiaoying Huang, Yingqiang Ge, Shuyuan Xu, Shijie Geng, Chirag Shah, Yongfeng Zhang, and Gerard de Melo. 2020. Fairness-aware explainable recommendation over knowledge graphs. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 69–78.

[19] Zuohui Fu, Yikun Xian, Yaxin Zhu, Shuyuan Xu, Zelong Li, Gerard De Melo, and Yongfeng Zhang. 2021. HOOPS: Human-in-the-Loop Graph Reasoning for Conversational Recommendation. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2415–2421.

[20] Yingqiang Ge, Shuchang Liu, Zuohui Fu, Juntao Tan, Zelong Li, Shuyuan Xu, Yunqi Li, Yikun Xian, and Yongfeng Zhang. 2022. A Survey on Trustworthy Recommender Systems. *arXiv preprint arXiv:2207.12515* (2022).

[21] Yingqiang Ge, Xiaoting Zhao, Lucia Yu, Saurabh Paul, Diane Hu, Chu-Cheng Hsieh, and Yongfeng Zhang. 2022. Toward Pareto Efficient Fairness-Utility Trade-off in Recommendation through Reinforcement Learning. In *WSDM*.

[22] Yingqiang Ge, Xiaoting Zhao, Lucia Yu, Saurabh Paul, Diane Hu, Chu-Cheng Hsieh, and Yongfeng Zhang. 2022. Toward Pareto Efficient Fairness-Utility Trade-off in Recommendation through Reinforcement Learning. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*. 316–324.

[23] Shijie Geng, Zuohui Fu, Yingqiang Ge, Lei Li, Gerard de Melo, and Yongfeng Zhang. 2022. Improving Personalized Explanation Generation through Visualization. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 244–255.

[24] Shijie Geng, Zuohui Fu, Juntao Tan, Yingqiang Ge, Gerard De Melo, and Yongfeng Zhang. 2022. Path Language Modeling over Knowledge Graphsfor Explainable Recommendation. In *Proceedings of the ACM Web Conference 2022*. 946–955.

[25] Shijie Geng, Shuchang Liu, Zuohui Fu, Yingqiang Ge, and Yongfeng Zhang. 2022. Recommendation as Language Processing (RLP): A Unified Pretrain, Personalized Prompt & Predict Paradigm (P5). *RecSys* (2022).

[26] Balázs Hidasi, Alexandros Karatzoglou, Linas Baltrunas, and D Tikk. 2016. Session-based recommendations with recurrent neural networks. In *ICLR*.

[27] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[28] Haifeng Jin, Qingquan Song, and Xia Hu. 2019. Auto-keras: An efficient neural architecture search system. In *SIGKDD*. 1946–1956.

[29] Wang-Cheng Kang and Julian McAuley. 2018. Self-attentive sequential recommendation. In *ICDM*. IEEE, 197–206.

[30] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[31] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* 42, 8 (2009), 30–37.

[32] Jing Li, Pengjie Ren, Zhumin Chen, Zhaochun Ren, Tao Lian, and Jun Ma. 2017. Neural attentive session-based recommendation. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. ACM, 1419–1428.

[33] Lei Li, Yongfeng Zhang, and Li Chen. 2020. Generate neural template explanations for recommendation. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 755–764.

[34] Lei Li, Yongfeng Zhang, and Li Chen. 2021. Personalized Transformer for Explainable Recommendation. In *ACL*. 4947–4957.

[35] Yunqi Li, Hanxiong Chen, Juntao Tan, and Yongfeng Zhang. 2022. Causal factorization machine for robust recommendation. In *Proceedings of the 22nd ACM/IEEE Joint Conference on Digital Libraries*. 1–9.

[36] Yunqi Li, Hanxiong Chen, Shuyuan Xu, Yingqiang Ge, and Yongfeng Zhang. 2021. Towards personalized fairness based on causal notion. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 1054–1063.

[37] Zelong Li, Jianchao Ji, Zuohui Fu, Yingqiang Ge, Shuyuan Xu, Chong Chen, and Yongfeng Zhang. 2021. Efficient non-sampling knowledge graph embedding. In *Proceedings of the Web Conference 2021*. 1727–1736.

[38] Zelong Li, Jianchao Ji, Yingqiang Ge, and Yongfeng Zhang. 2022. AutoLossGen: Automatic Loss Function Generation for Recommender Systems. *SIGIR* (2022).

[39] Zelong Li, Jianchao Ji, and Yongfeng Zhang. 2022. From Kepler to Newton: Explainable AI for Science Discovery. In *ICML 2022 2nd AI for Science Workshop*. https://openreview.net/forum?id=vA9hti-Fi7H

[40] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. DARTS: Differentiable Architecture Search. In *International Conference on Learning Representations*.

[41] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton Van Den Hengel. 2015. Image-based recommendations on styles and substitutes. In *SIGIR*. 43–52.

[42] Niv Nayman, Asaf Noy, Tal Ridnik, Itamar Friedman, Rong Jin, and Lihi Zelnik. 2019. XNAS: Neural Architecture Search with Expert Advice. *NIPS* 32 (2019), 1977–1987.

[43] Changhua Pei, Xinru Yang, Qing Cui, Xiao Lin, Fei Sun, Peng Jiang, Wenwu Ou, and Yongfeng Zhang. 2019. Value-aware recommendation based on reinforcement profit maximization. In *The World Wide Web Conference*. 3123–3129.

[44] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. 2018. Efficient neural architecture search via parameters sharing. In *ICML*. PMLR, 4095–4104.

[45] 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.

[46] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. 2017. Large-scale evolution of image classifiers. In *International Conference on Machine Learning*. PMLR, 2902–2911.

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

[48] Alan Said and Alejandro Bellogín. 2014. Comparative recommender system evaluation: benchmarking recommendation frameworks. In *RecSys*.

[49] Ruslan Salakhutdinov and Andriy Mnih. 2008. Bayesian probabilistic matrix factorization using Markov chain Monte Carlo. In *ICML*. 880–887.

[50] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. 2007. Restricted Boltzmann machines for collaborative filtering. In *Proceedings of the 24th international conference on Machine learning*. 791–798.

[51] Shaoyun Shi, Hanxiong Chen, Weizhi Ma, Jiaxin Mao, Min Zhang, and Yongfeng Zhang. 2020. Neural Logic Reasoning. In *CIKM*. 1365–1374.

[52] 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.

[53] Fei Sun, Jun Liu, Jian Wu, Changhua Pei, Xiao Lin, Wenwu Ou, and Peng Jiang. 2019. BERT4Rec: Sequential recommendation with bidirectional encoder representations from transformer. In *Proceedings of the 28th ACM international conference on information and knowledge management*. 1441–1450.

[54] Juntao Tan, Shijie Geng, Zuohui Fu, Yingqiang Ge, Shuyuan Xu, Yunqi Li, and Yongfeng Zhang. 2022. Learning and evaluating graph neural network explanations based on counterfactual and factual reasoning. In *Proceedings of the ACM Web Conference 2022*. 1018–1027.

[55] Juntao Tan, Shuyuan Xu, Yingqiang Ge, Yunqi Li, Xu Chen, and Yongfeng Zhang. 2021. Counterfactual explainable recommendation. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 1784–1793.

[56] Jiaxi Tang and Ke Wang. 2018. Personalized top-n sequential recommendation via convolutional sequence embedding. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. 565–573.

[57] Chenyang Wang, Min Zhang, Weizhi Ma, Yiqun Liu, and Shaoping Ma. 2020. Make it a chorus: knowledge-and time-aware item modeling for sequential recommendation. In *SIGIR*. 109–118.

[58] Zhenlei Wang, Jingsen Zhang, Hongteng Xu, Xu Chen, Yongfeng Zhang, Wayne Xin Zhao, and Ji-Rong Wen. 2021. Counterfactual data-augmented sequential recommendation. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 347–356.

[59] Yikun Xian, Zuohui Fu, Shan Muthukrishnan, Gerard De Melo, and Yongfeng Zhang. 2019. Reinforcement knowledge graph reasoning for explainable recommendation. In *Proceedings of the 42nd international ACM SIGIR conference on research and development in information retrieval*. 285–294.

[60] Yikun Xian, Zuohui Fu, Handong Zhao, Yingqiang Ge, Xu Chen, Qiaoying Huang, Shijie Geng, Zhou Qin, Gerard De Melo, Shan Muthukrishnan, and Yongfeng Zhang. 2020. CAFE: Coarse-to-fine neural symbolic reasoning for explainable recommendation. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 1645–1654.

[61] Yikun Xian, Tong Zhao, Jin Li, Jim Chan, Andrey Kan, Jun Ma, Xin Luna Dong, Christos Faloutsos, George Karypis, Shan Muthukrishnan, and Yongfeng Zhang. 2021. Ex3: Explainable attribute-aware item-set recommendations. In *Fifteenth ACM Conference on Recommender Systems*. 484–494.

[62] Kun Xiong, Wenwen Ye, Xu Chen, Yongfeng Zhang, Wayne Xin Zhao, Binbin Hu, Zhiqiang Zhang, and Jun Zhou. 2021. Counterfactual Review-based Recommendation. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 2231–2240.

[63] Shuyuan Xu, Yingqiang Ge, Yunqi Li, Zuohui Fu, Xu Chen, and Yongfeng Zhang. 2021. Causal Collaborative Filtering. *arXiv:2102.01868* (2021).

[64] Shuyuan Xu, Juntao Tan, Zuohui Fu, Jianchao Ji, Shelby Heinecke, and Yongfeng Zhang. 2022. Dynamic Causal Collaborative Filtering. *CIKM* (2022).

[65] Shuyuan Xu, Juntao Tan, Shelby Heinecke, Jia Li, and Yongfeng Zhang. 2021. Deconfounded Causal Collaborative Filtering. *arXiv:2110.07122* (2021).

[66] Hong-Jian Xue, Xinyu Dai, Jianbing Zhang, Shujian Huang, and Jiajun Chen. 2017. Deep Matrix Factorization Models for Recommender Systems.. In *IJCAI*, Vol. 17. Melbourne, Australia, 3203–3209.

[67] Quanming Yao, Xiangning Chen, James T Kwok, Yong Li, and Cho-Jui Hsieh. 2020. Efficient neural interaction function search for collaborative filtering. In *Proceedings of The Web Conference 2020*. 1660–1670.

[68] Yongfeng Zhang. 2021. Problem Learning: Towards the Free Will of Machines. *arXiv:2109.00177* (2021).

[69] Yongfeng Zhang, Qingyao Ai, Xu Chen, and W Bruce Croft. 2017. Joint representation learning for top-n recommendation with heterogeneous information sources. In *CIKM*. 1449–1458.

[70] Yongfeng Zhang and Xu Chen. 2020. Explainable recommendation: A survey and new perspectives. *Foundations and Trends® in Information Retrieval* 14, 1 (2020), 1–101.

[71] Yongfeng Zhang, Xu Chen, Qingyao Ai, Liu Yang, and W Bruce Croft. 2018. Towards conversational search and recommendation: System ask, user respond. In *Proceedings of the 27th acm international conference on information and knowledge management*. 177–186.

[72] Yongfeng Zhang, Guokun Lai, Min Zhang, Yi Zhang, Yiqun Liu, and Shaoping Ma. 2014. Explicit factor models for explainable recommendation based on phrase-level sentiment analysis. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*. 83–92.

[73] Yaxin Zhu, Yikun Xian, Zuohui Fu, Gerard de Melo, and Yongfeng Zhang. 2021. Faithfully Explainable Recommendation via Neural Logic Reasoning. In *NAACL*. 3083–3090.

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

[75] 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.