Towards Automated Neural Interaction Discovery for Click-Through Rate Prediction*

Qingquan Song[1], Dehua Cheng[2], Hanning Zhou[2], Jiyan Yang[2], Yuandong Tian[2], Xia Hu[1]
[1]Texas A&M University, College Station, TX
[2]Facebook Inc. Menlo Park, CA
{song_3134,xiahu}@tamu.edu,{dehuacheng,hanningz,chocjy,yuandong}@fb.com

ABSTRACT

Click-Through Rate (CTR) prediction is one of the most important machine learning tasks in recommender systems, driving personalized experience for billions of consumers. Neural architecture search (NAS), as an emerging field, has demonstrated its capabilities in discovering powerful neural network architectures, which motivates us to explore its potential for CTR predictions. Due to 1) diverse unstructured feature interactions, 2) heterogeneous feature space, and 3) high data volume and intrinsic data randomness, it is challenging to construct, search, and compare different architectures effectively for recommendation models. To address these challenges, we propose an automated interaction architecture discovering framework for CTR prediction named AutoCTR. Via modularizing simple yet representative interactions as virtual building blocks and wiring them into a space of direct acyclic graphs, AutoCTR performs evolutionary architecture exploration with learning-to-rank guidance at the architecture level and achieves acceleration using low-fidelity model. Empirical analysis demonstrates the effectiveness of AutoCTR on different datasets comparing to human-crafted architectures. The discovered architecture also enjoys generalizability and transferability among different datasets.

CCS CONCEPTS

• **Information systems → Recommender systems**; • **Theory of computation → Evolutionary algorithms**; • **Computing methodologies → Neural networks**.

KEYWORDS

CTR prediction; neural architecture search; evolutionary algorithm

ACM Reference Format:
Qingquan Song[1], Dehua Cheng[2], Hanning Zhou[2], Jiyan Yang[2], Yuandong Tian[2], Xia Hu[1]. 2020. Towards Automated Neural Interaction Discovery for Click-Through Rate Prediction. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '20), August 23–27, 2020, Virtual Event, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3394486.3403137

1 INTRODUCTION

Predicting Click-Through Rate (CTR) is a crucial problem in many web applications such as real-time bidding, display advertising, and search engine optimization [20]. Due to the large-scale dataset and high-cardinality feature property, extensive efforts have been devoted to designing architectures for effectively learning combinatorial feature interactions towards condensed low-dimensional feature representations [7, 9, 16, 30].

Classical approaches usually put the efforts on designing explicit feature interactions and compose it with implicit interactions learning from multi-layer perceptrons (MLP) into a two-tower model [9, 28, 30]. Beyond simple stacking strategies, how to organically bond the explicit and implicit interactions is still underexplored and may further promote effectiveness. Besides, existing work has shown that diamond MLP structure may be more powerful compared to the triangle and rectangular MLP structures [35], which motivates us to explore more powerful implicit interactions via designing delicate MLP structures. In addition, conjoining the advantages of diversified explicit feature interactions such as inner and outer products could potentially boost the performance owning to the ensemble effect.

Neural architecture search (NAS) has emerged as a prevailing research field upon the prevalent adoption of deep learning techniques. It aims to discover optimal deep learning solutions automatically given a data-driven problem, thereby enabling practitioners to access the off-the-shelf deep learning techniques without extensive experience, and alleviating data scientists from the burden of manual network design. The rapid development of NAS research and systems has enabled the automation of state-of-the-art deep learning tools for various learning tasks in computer vision (CV) and natural language processing [6, 8]. This motivates us to explore its potential in the context of tabular data in discovering complex neural interactions, specifically for CTR predictions.

Developing novel NAS approaches for neural interaction discovery and better CTR models is technically challenging. First, different from structure image data in CV tasks, CTR features are often heterogeneous, high-dimensional, and have both sparse and dense components, which are structureless and of diversified meanings in reality. Second, different from the dominant convolutional neural networks in CV tasks that consist of multiple structured convolutional operations, existing models for CTR prediction usually adopt multiple diverse and ad-hoc operations, leading to unstructured search space. Third, a practical model for CTR prediction is often trained on billions of data (e.g., Facebook has millions of daily active users and over 1 million active advertisers [11], yielding billions of instances), requiring the NAS process to be time and space efficient.

*A majority of this work was done while the first author was interning at Facebook.

KDD '20, August 23–27, 2020, Virtual Event, CA, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7998-4/20/08...$15.00
https://doi.org/10.1145/3394486.3403137

Finally, the performance of CTR models with different architectures are often quite close in practice [28, 30], asking for the NAS approach to be sensitive and discriminating.

To cope with these challenges, we propose an automated neural interaction discovering framework for CTR prediction named **AutoCTR**. We abstract and modularize simple yet representative operations in existing CTR prediction approaches to formulate a generalizable search space. A hybrid search algorithm, composed of an evolutionary backbone and a learnable guider, is designed to perform orientated exploration. To enhance the exploitation power and balance the trade-off among different search objectives, we utilize a learning-to-rank strategy among the architecture level to filter out the locally inferior architectures, and conduct the survivor selection based on a mixture of rank-based measurement including aging, accuracy as well as architecture complexity. The search speed is further accelerated through a composited strategy of low-fidelity estimation, including data subsampling and hash size reduction. The main contributions are summarized from the following aspects:

- Design *virtual blocks* and a hierarchical search space for CTR prediction by abstracting and unifying the commonly used operations in the existing literature.
- Provide ranking consistency analysis for three strategies combining low-fidelity estimation and weight inheritance. Empirically prove the availability of employing them for search acceleration.
- Propose a novel multi-objective evolutionary search algorithm with architectural-level learning-to-rank guidance.
- Empirically demonstrate the effectiveness of AutoCTR on different datasets comparing to human-crafted architectures, and validate the generalizability and transferability of the discovered architecture across different datasets.

2 PRELIMINARIES

Click-Through Rate Prediction: The CTR prediction problem could be mathematically defined as: given a dataset $D = \{\mathbf{X}, \mathbf{y}\}$, where $\mathbf{X} \in \mathbb{R}^{N \times d}$ denotes the d-dimensional features matrix of N instances. The features here consist of both sparse and dense features, where we assume the sparse features are ordinally encoded as integer vectors. $\mathbf{y} \in \{0, 1\}^N$ indicates the clicks of users to items. The goal is to predict the probability of a user clicking a target item.

Since its inception, the mainstream models have roughly experienced three-stage evolution starting from linear regression and tree-based models [11, 20, 32], to interaction-based models [14, 26], and then the deep neural networks (DNNs) [7, 9, 10, 16, 19, 21, 23, 30, 35, 36]. Recent DNN-based work usually combines DNN with interaction-based or tree-based models to extracting condensed representations via learning combinatorial feature interactions.

Neural Architecture Search: Neural architecture search (NAS) aims to promote the design of neural network architectures automatically. Designing the NAS algorithm requires the specification of three main components, including *search space*, *search techniques*, and *performance estimation strategy* [8].

From the search space perspective, existing work could be roughly divided into the exhaustive-architecture search space [13, 25] and the constrained cell space [18, 22, 24, 38]. The former one provides a more diversified set of architectures and allows a more comprehensive exploration, while the later one inductively limits the space to accelerate the search speed and reduce the search variance. Some tailored spaces are also designed [17] for specific tasks.

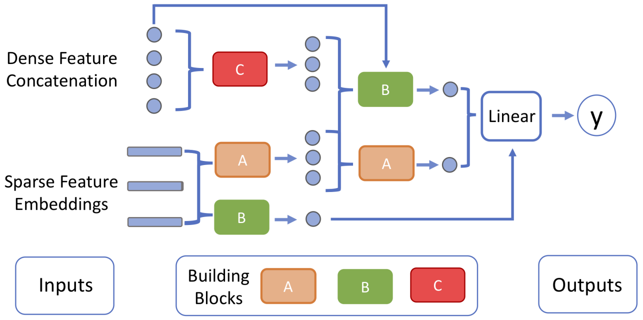


Figure 1: An illustration of an architecture in the designed search space. The virtual building blocks are wired together to form a DAG. Blocks are allowed to be selected repetitively.

From the perspective of search techniques, several dominant ones include Bayesian optimization [13], reinforcement learning [22, 37, 38], evolutionary algorithms [24, 25], gradient-based optimization [18], and tree-based methods [29]. Recent work has shown the effectiveness of combining different types of search techniques to better balance exploitation and exploration [5].

Due to the high complexity of training the DNNs and the particularity of evaluation criteria in different tasks, various performance estimation strategies are proposed, including low-fidelity estimation [24, 38], weight sharing [18, 22], learning curve extrapolation [31], and network morphism [13].

3 HIERARCHICAL SEARCH SPACE DESIGN

An ideal search space should contain sufficient distinct architectures while potentially encompassing superior human-crafted structures [13, 37]. Inspired by the search space tailored to the vision tasks [17], we design a two-level hierarchical search space by extracting and abstracting representative structures in existing human-crafted CTR prediction architectures into virtual blocks, and wire them together as a set of direct acyclic graphs (DAGs) with dimensionality alignment among features. The inner space is composed of the appendant hyperparameters of blocks, such as the number of units and layers in a multi-layer perceptron block, while the connections among blocks form the outer search space.

As shown in Figure 1, for a given instance, we assume its raw input dense features are concatenated into a vector, and its sparse features are embedded into low-dimensional vectors based on the look-up-table operation following similar preprocessing done in various CTR models [21, 30]. Blocks are wired together to form a DAG, and each block could take both raw input features and the outputs from blocks with higher topological order via feature concatenation and dimensionality alignment. The final block of the network is set to be a linear transformation. It collects all the untouched features from either raw inputs or outputs provided by other blocks. It is worth noting that to simplify the setting, the following hyperparameters are not taken into account in the search space: (1) Hash size of sparse features. (2) Embedding dimension of sparse features. (3) Optimizers and other model training hyperparameters such as learning rate and batch size.

3.1 Virtual Block Abstraction

We select and extract building blocks by considering two aspects described as follows:

- **Functionality:** blocks should accommodate and complement each other. Each type of block should accommodate both dense and sparse features as inputs, and can be quantitatively evaluated.
- **Complexity Aware:** The computational and memory cost of a block should be a simple function of its input specification and hyperparameters. The involvement of these primitives could benefit the design of complexity-aware search algorithms to achieve better resource management and low-complex architectures.

Upon these requirements, we could abstract various operations from existing work as blocks with different functionality and levels of complexity such as multi-layer perceptron (MLP) [21], dot product (DP), factorization machine (FM) [26], outer product [30], and self-attention [28], etc. We elaborate on the construction of the three example blocks (i.e., MLP, DP, FM) adopted in the experiments in Appendix A and describe the way of aligning the dimension and accommodating different inputs for each of them. Other blocks could also be easily abstracted and integrated into the framework, which is left for future exploration.

3.2 Summarization of Search Components

We summarize the main components to be searched in the hierarchical search space as follows:

- **Block Type:** MLP, FM, DP.
- **Raw Feature Input Selection:** each block is allowed to take the raw feature with four choices, i.e., dense only, sparse only, both or none. Without particular emphasis, we group the raw input dense features as one single component to be selected rather than considering them independently. A similar procedure is done for the sparse features to reduce the search complexity.
- **Inter-Block Connection:** a block could receive the outputs from any block that appeared before it. The order is defined as the topological order in the DAG.
- **Block Appendant Hyperparameters:** We only consider the number of hidden units of MLP block in this work. The embedding sizes for aligning the input dimensions in different blocks are fixed and set to be the same with the embedding size of the raw input sparse features to reduce the number of parameters.

To enable the feasible adoption of different searching algorithms, we provide a vector representation of each architecture as a concatenation of multiple block vectors following [29]. Each block is vectorized as a concatenation of the four components, i.e., *[Block Type, Raw Feature Input Selection, Inter-Block connection, Block Appendant Hyperparameters]*, where the *Block Type* and *Raw Feature Input Selection* are encoded as one-hot vectors respectively, *Inter-Block connection* is encoded as a multi-hot vector, and *Block Appendant Hyperparameters* is encoded as a vector of ordinals.

The designed search space contains plentiful distinct architectures. Even with three types of blocks to be selected and assume each architecture contains no more than seven blocks, the space would still contain over 10^{11} distinct architectures. Moreover, it could cover multiple representative human-crafted architectures such as deepFM [9], DLRM [21], IPNN [23], and Wide&Deep [7].

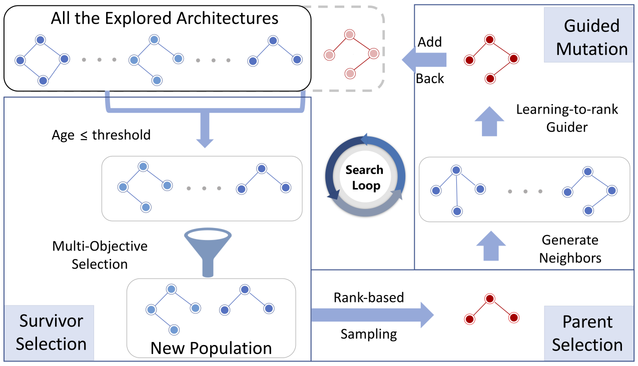


Figure 2: An illustration of the AutoCTR search loop

4 MULTI-OBJECTIVE EVOLUTIONARY SEARCH WITH HYPERRANK GUIDANCE

The proposed searcher is a mixed searcher composed of an evolutionary algorithm and a learning-to-rank guider. We select the evolutionary algorithm in this pilot study due to its simplicity and effectiveness in balancing the exploitation and exploration [24]. We adopt a multi-objective evolutionary searcher as the backbone and employ a tree-based learner to guide the mutation of a selected parent architecture in each iteration to facilitate the exploitation of superior offsprings. The search process could be described as a loop of three stages, i.e., parent selection, guided mutation, and survivor selection. The initial population is constructed via randomly selecting and evaluating a predefined number of architectures. Stratified selections could also be used to potentially enhance the performance, which we leave for future exploration.

The basic idea of the search loop is shown in Figure 2. In each search loop, we leverage a mixture of rank-based meta-features to perform the survivor selection and maintain a new fix-size population. Then we select a parent architecture from the population based on designed discrete probabilistic distribution. After that, a set of neighbors is generated via mutating the selected parent architecture. We adopt a learning-to-rank mechanism upon the architecture level and select the best offspring from the generated neighbors. After evaluating the performance of this offspring, we add it back to the architecture pool explored so far. The three stages are elaborated in turn in the following subsections.

4.1 Multi-Objective Survivor Selection

To maintain a superior population with diversified architectures, we design a survivor selection metric f to measure the survival value of each architecture and select the top-p ones as the population for parent selection. Three types of objectives are considered in the metric, i.e., fitness, age, and model complexity. The “fitness” here represents the performance of an explored architecture to ensure the exploitation ability, while the “age” reflects the reverse order of the architectures explored so far. We use the rank of logloss as the fitness measure to mitigate the difference in scale and define the “age” as the existing time of an architecture explored so far, motivated from the age-based evolutionary methods [24], to enhance the exploration of diversified architectures. At each search loop, we set the current time to be 0 and set the age of each observed architecture as the number of architectures explored after it. In particular, all initial architectures are assigned with the same age. Besides the two objectives, since CTR tasks are usually resource hungry in practice, we also take the "model complexity" into account to explicitly constrain the model complexity during search. We adopt floating point operations per second (FLOPs) as the complexity metric and use the rank of flops to mitigate the scale influence.

The designed selection schema consists of two steps as shown in Figure 2, which could be formulated as follows:

$$f(q, a_A, r_A^q, c_A^q) = \mathbb{1}_{[a_A \le q]} \cdot (\mu_1 a_A + \mu_2 r_A^q + \mu_3 c_A^q), \qquad (1)$$

where a_A denotes the age of an architecture A. The indicator function $\mathbb{1}_{[a_A \le q]}$ is used to filter out the architectures that is older than q, where q is a hyperparameter larger than the population size p, such that the architectures with high-performance or low-complexity in the pool would not be selected consistently. $r_A^q, c_A^q \in \{1, 2, \ldots q\}$ are the performance and complexity ranking of architecture A within the q "youngest" architectures, respectively. $\{\mu_i\}_{i=1,2,3}$ are the trade-off hyperparameters to balance the different objectives.

4.2 Rank-Based Parent Selection

Suppose we maintain a population of size $p \in \mathbb{Z}^+$ after the survivor selection step. The goal of parent selection is to select an architecture from the population for generating a premium offspring to be evaluated. Several popular strategies used in conventional evolutionary algorithms include proportional, ranking, tournament, and genitor selection [34]. In this work, we adopt a ranking selection schema and design a nonlinear ranking distribution borrowing the idea from tournament selection. The intuition comes from three aspects: (1) The performance (logloss or AUC) of different architectures in CTR prediction tasks is often extremely close, and the scale may vary a lot on different datasets. It is hard to design a unified performance-based distribution to achieve adaptive selective pressure on different datasets. (2) Existing work has proved the effectiveness of ranking and tournament selection comparing with proportional and genitor on balancing the selective intensity and selection diversity [2, 34]. (3) Classical tournament methods usually randomly select a fixed number of candidates first and then select the best one of them. This may result in a portion of architectures in the population never being selected as the parent, especially when the ratio between candidate size and population size is large.

The concrete design of the probability is as follows:

$$p(r_A^*) = \frac{\binom{r_A^*+\lambda-1}{\lambda}}{\binom{p+\lambda}{1+\lambda}}, \qquad r_A^* \in \{1, 2, \ldots, p\}, \lambda \in \mathbb{N}^0, \qquad (2)$$

where r_A^* denotes the rank of an architecture A in the population, $\binom{n}{k} = \frac{n!}{(n-k)!k!}, (k = 0, 1, .., n)$, and λ is a hyperparameter to balance the trade-off between selection intensity and selection diversity. Given a fixed size of population, with the increasing of λ, the selection intensity increases while the selection diversity reduces. In particular, $\lambda = 0$ is the same as uniform selection and when $\lambda = 1$ is the same as linear rank selection [1, 2].

4.3 Guided Mutation by Learning to Hyperrank

After the parent architecture is selected, the last step is to generate a worth exploring offspring upon it. A naive way of doing this is to randomly select and modify an operation in the parent architecture [24]. Nevertheless, it could become an inefficient strategy due to the huge search space and the waste of the architecture information explored so far. Existing work has demonstrated the effectiveness of using a learning-based model to guide the mutation process [5]. However, with a limited number of explored architectures and the extremely close performance among them, it is non-trivial to learn an effective fitness-based guider.

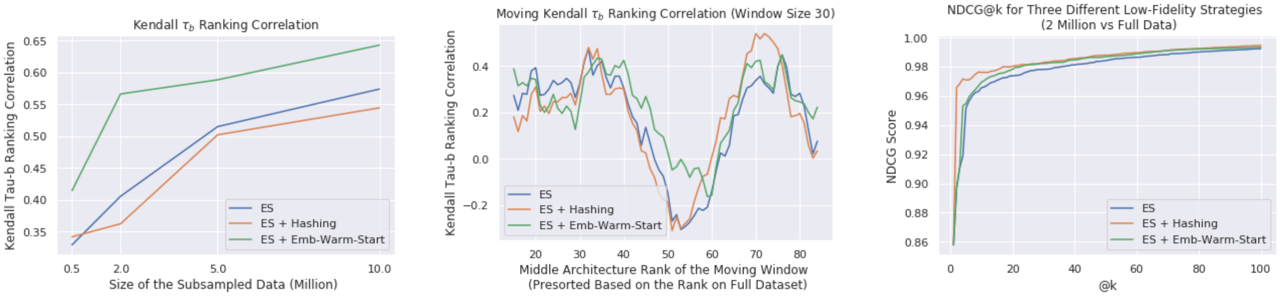
Instead of learning a fitness-based guider, we adopt a learning-to-rank strategy to learn the relative ranking among architectures based on a pairwise ranking loss and the gradient boosted tree learner. The whole offspring generation process is done via three steps: (1) train a guider based on the exploitable dataset (e.g., all the architectures explored so far); (2) randomly generate a set of unique neighbors around the parent architecture; (3) select the best neighbor as the offspring based on the guider. We call it "learning-to-Hyperrank" as it conducts a model-level ranking. The intuition comes from two perspectives. On the one hand, the search problem itself is essentially a ranking problem. Learning the ranking relationship is a commensurable strategy comparing with learning the fitness but is weaker and more flexible. On the other hand, the number of instances are implicitly augmented from the point-wise inputs to the pairwise inputs.

For the pairwise ranking-loss, we use LambdaRank [3] due to its simplicity and efficiency. Though different types of models could serve as the learner, we choose a gradient boosted tree learner as an example here due to its general stable and superior performance on small-scale datasets. To feed architectures as the input for the tree-based learner, we encode each block as a vector and concatenate them based on their topological order. Each block vector follows the vector representation described in section 3.2.

5 PERFORMANCE ESTIMATION ACCELERATION

One of the most crucial challenges in modern NAS research, which could be even more severe in recommender systems, is the high time complexity of network training. Low-fidelity performance estimation and weight inheritance are two of the most widely adopted methods for speeding up performance estimation [24, 38]. However, extra bias could be introduced, leading to the variation of relative ranking among architectures, thereby affecting the searching effectiveness [8]. We consider two strategies of low-fidelity estimation and adopt a warm-start embedding trick leveraging the weight inheritance among architectures to mitigate the time and resource complexity. These strategies are all general and practical ways to speed up the manual tuning of recommender systems. Several rank consistency tests are described afterward to provide the evidence and illustrate the feasibility of adopting these methods.

- **Data Subsampling.** For each dataset, we randomly subsample a predefined portion of data for searching and transfer the searched best architecture on the full dataset for final evaluations. On average, the training time of every single architecture in our



(a) Rank consistency varying with the number of subsamples and different training strategy

(b) Local ranking consistency shifting with window size 30 upon the sorted "ground-truth" rank

(c) NDCG@K score of different training strategies on 2 million subsamples

Figure 3: Rank consistency analysis from both global perspective and local perspective

experiment is roughly linearly correlated with the subsampling ratio under the hardware setting described in Appendix C.3.

- **Reducing Hash Size.** For high-level categorical features ($> 10^4$), we hash the cardinality of them to 10^4 before the embedding step to reduce the embedding size, and set back to the original cardinality during final fit on the full dataset.
- **Warm-Start Embedding.** We pretrain a simple three-layer MLP model (units: 128-1024-128) on the full dataset and use the pretrained embeddings of sparse features as the warm-start for each architecture before the low-fidelity training.

The rest of this section provides an analysis of the effect of adopting low-fidelity training on ranking consistency. We use logloss as the evaluation metric and use the early-stopping (ES) strategy to alleviate overfitting. We focus on the global and local rank consistency respectively, to pursue the analysis. The global rank consistency inspects whether the estimation could reflect the actual ranking of performance among architectures[1], and the local rank consistency testing zooms into the architectures with relatively closer performance and analyze their localized rank consistency.

5.1 Global Rank Consistency

Settings: We use Criteo dataset[2] here for experiments. Without loss of generality, we narrow down the search space by assuming each architecture contains five blocks, and each MLP block has one layer with 128 units. We randomly sample 100 valid architectures and evaluate them on Criteo with different sizes of subsamples, i.e., $\{0.5, 2, 5, 10\}$ million. Each subsampled dataset is split into training (80%), validation (10%), and test (10%) sets. We run the 100 models on the full dataset three times to provide a "ground-truth" rank, and the rank consistency is measured by the Kendall τ_b coefficient ranging from -1 (perfect inversion) to 1 (perfect agreement).
Observations: Figure 3(a) depicts the varying curves of three training strategies measured by the Kendall τ_b coefficient with the increasing of the subsample size. The three strategies are: (1) early stopping; (2) early stopping with sparse feature hashing; (3) early stopping with warm-start embedding. We observe that: firstly, with the increasing of the subsample size, the rank among architectures become more consistent with the âĂIJground-truthâĂİ rank, and the growth speed gradually becomes slow. The non-linear relationship between sample size and τ_b coefficient offers the opportunity of adopting the low-fidelity setting during searching, which will be further evaluated in the experimental section. Secondly, adopting the hashing strategy for sparse features with high cardinality would marginally decrease the rank consistency. Finally, the warm-start embedding strategy could increase ranking consistency.

5.2 Local Rank Consistency

The above analysis provides a macro view of the ranking consistency among the architectures. Beyond this, we are also curious if the τ_b score is harmoniously distributed across different intervals of the actual rank. We first sort the 100 architectures based on their âĂIJground-truthâĂİ performance evaluated on the full dataset. Better architectures are indexed with smaller numbers. Then we depict two plots for analysis. Figure 3(b) displays the variation of τ_b coefficient with a length-30 sliding window among the sorted architectures. For example, the score of the first point denotes the τ_b coefficient of the top-30 architectures, where its x-coordinate is 15, indicating the middle architecture rank within this sliding window. Figure 3(c) shows the variation of the NDCG@k score with the increase of k. Exponential gain is used to calculate the NDCG score.

From the two figures, two main observations could be found: (1) From Figure 3(b), we can see that, best- and worst-performing architectures seem to be more locally rank consistent. This is partially aligned with our expectations since we expect the top architectures and the bottom ones to be more easily discernible than others. The warm-start strategy is more helpful for maintaining the local rank of middle architectures rather than the polar ones. (2) From Figure 3(c), we can observe that the rank of the top architectures could generally maintain high rankings in the low-fidelity setting even if the rank consistency for some mediocre architectures is affected more compared to the polar ones.

6 EXPERIMENTAL ANALYSIS

In this section, we empirically evaluate AutoCTR as well as several baseline searchers and compare the discovered architecture to the human-crafted architectures. We use logloss and AUC score as the core evaluation metrics. Four questions are mainly explored:

[1] We assume the actual rank of architectures is reflected by their high-fidelity performance achieved on the full dataset.
[2] www.kaggle.com/c/criteo-display-ad-challenge

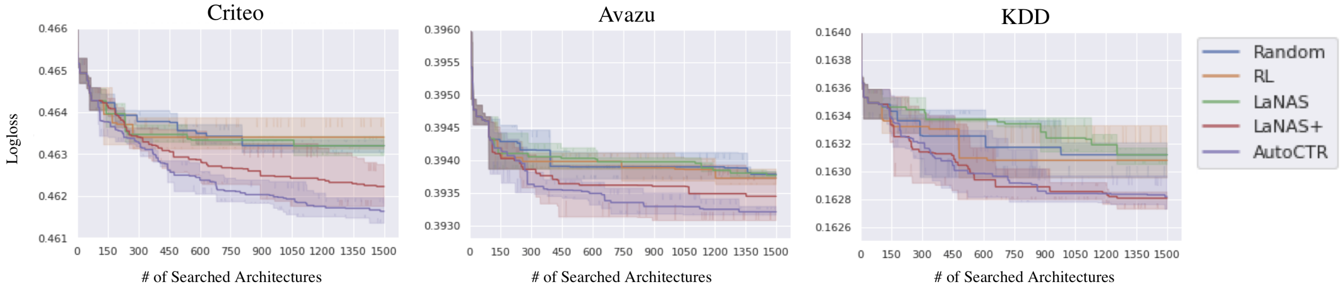


Figure 4: The performance drifting of the best architecture during the search process on different datasets

Q1. How is AutoCTR comparing with other baseline searchers on both the search efficiency and effectiveness?

Q2. How is the performance of the best architecture explored by the AutoCTR comparing with the state-of-the-art (SOTA) human-crafted architectures?

Q3. Are the searched architectures able to be transferred between different datasets?

Q4. How sensitive is the AutoCTR to its key hyperparameters?

6.1 Baselines

We select baselines from both NAS methods and human-crafted CTR architectures for comparison. Since several searchers are not directly applicable in our setting, we modify and improve their flexible components and elaborate the details in the Appendix B. The selected NAS methods include random search, reinforcement learning based search (RL), and a sample-efficient tree-based search: latent action neural architecture search (LaNAS) [29]. We also include a variation of LaNAS named LaNAS+, which is improved from two perspectives. (1) We borrow the idea from AutoCTR to perform evolutionary sampling in LaNAS after a narrowed search space is selected, which is originally done by random. It highly improves the search speed and the exploitation power upon the experimental results. (2) As LaNAS requires each architecture to be finished in order to explore a new one, we include virtual loss to enable its parallelization. See Appendix B for more details. Some other searchers are not directly considered due to the complexity of converting them to fit our search space, such as the kernel design and adjustment in the Bayesian optimization approach (e.g., [13]), and the super graph construction in gradient-based methods (e.g., [18]), which are left for future exploration. We select three representative human-crafted networks DLRM [21], DeepFM [9], and AutoInt+ [28], in which the first two are covered in our search space. More details about the baselines are elaborated in Appendix B.

6.2 Experimental Settings

6.2.1 Data Preprocessing. We adopt three benchmark datasets in this paper, i.e., Criteo[2], Avazu[3], KDD Cup[4]. The basic statistics of them are summarized in Table 5 in Appendix C.1. All three datasets are processed based on the way and codes provided in [28]. During the search phase, we subsample the first 2 million data of each dataset and further divide it into the tiny training (80%), validation (10%), and test (10%) sets for low-fidelity evaluation.

6.2.2 Hyperparameter Settings. We search seven intermediate blocks for each architecture. Blocks are allowed to be empty. Three example blocks are adopted in the final experiments, i.e., MLP, FM, and DP. The detailed construction of them is provided in Appendix A. We randomly sample 100 architectures as initialization for all the searchers. For the RL searcher, we adopt the off-policy training to go through the 100 architectures. For AutoCTR, the 100 architectures will directly form the initial population. For LaNAS and LaNAS+, the initial tree splits are learned with these 100 architectures. We repeat the experiments three times with three different seeds. Detailed hyperparameter settings for each searcher and the network training in both the search phase and final fit phase, are specified in Appendix C due to the page limitation.

6.3 General Comparison Among Searchers

We first compare the general search performance of all five searchers. Figure 4 depicts the logloss drifting of the best architecture during searching on the three datasets. The x-axis indicates the number of architectures searched so far (in total, 1500). The y-axis denotes the validation logloss of the architectures. Several observations can be summarized as follows. Firstly, based on the performance of the best architecture searched in the low-fidelity setting, AutoCTR generally outperforms other baselines, and our modified LaNAS+ outperforms LaNAS. Secondly, by comparing the search efficiency, AutoCTR and LaNAS+ still outperform other searchers consistently.

We then transfer the best architectures searched so far by each searcher onto the full dataset and display the final evaluation results in Table 1. The results shown from row four to eight are the average performance of the best architectures searched in the three rounds with different seeds. We can observe that: (1) After transferring the best architectures found by the searchers on the full datasets, the architectures found by AutoCTR still performs the best. Moreover, the ranking of the results aligns well with the one in low-fidelity setting, which implies the correctness of the rank consistency testing and the feasibility of adopting low-fidelity estimation in the search process. (2) On all three datasets, the final architectures searched by AutoCTR and LaNAS+ could achieve even better performance comparing to the SOTA architectures. The architectures searched by Random search and RL-based search could also achieve comparable or even better performance. This empirically validates the effectiveness of the designed search space and the feasibility of adopting NAS algorithms on the CTR prediction problem. It is worth pointing out that an improvement of around 0.0005-0.001 is already regarded as practically significant on these CTR prediction

[3] https://www.kaggle.com/c/avazu-ctr-prediction/data
[4] https://www.kaggle.com/c/kddcup2012-track2/data

Table 1: General CTR prediction results on the three benchmark datasets

		Criteo		Avazu		KDD		Search cost
		Logloss	AUC	Logloss	AUC	Logloss	AUC	(GPU Days)
SOFA human-crafted Networks	DeepFM	0.4432	0.8086	0.3816	0.7767	0.1529	0.7974	-
	DLRM	0.4436	0.8085	0.3814	0.7766	0.1523	0.8004	-
	AutoInt+	0.4427	0.8090	0.3813	0.7772	0.1523	0.8002	-
Best Networks Found by the NAS Methods	Random	0.4421 ± 0.0003	0.8096 ± 0.0004	0.3824 ± 0.0030	0.7765 ± 0.0029	0.1531 ± 0.0001	0.8001 ± 0.0003	~ 0.75
	RL	0.4422 ± 0.0005	0.8094 ± 0.0005	0.3810 ± 0.0003	0.7778 ± 0.0005	0.1531 ± 0.0001	0.7999 ± 0.0002	~ 0.75
	LaNAS	0.4421 ± 0.0004	0.8096 ± 0.0005	0.3814 ± 0.0006	0.7772 ± 0.0011	0.1533 ± 0.0002	0.8001 ± 0.0009	~ 5
	LaNAS+	0.4417 ± 0.0001	0.8101 ± 0.0000	0.3800 ± 0.0004	0.7790 ± 0.0007	0.1521 ± 0.0001	0.8009 ± 0.0004	~ 0.75
	AutoCTR	**0.4413 ± 0.0002**	**0.8104 ± 0.0003**	**0.3800 ± 0.0001**	**0.7791 ± 0.0001**	**0.1520 ± 0.0000**	**0.8011 ± 0.0001**	~ 0.75
	AutoCTR (warm)	0.4417 ± 0.0005	0.8099 ± 0.0005	0.3804 ± 0.0004	0.7784 ± 0.0006	0.1523 ± 0.0001	0.8004 ± 0.0003	~ 0.75

Table 2: Architecture complexity comparison (parameters in the embedding tables are included)

	# Params (Million)			Flops (Million)		
	Criteo	Avazu	KDD	Criteo	Avazu	KDD
DeepFM	22.51	30.34	101.73	22.74	22.50	21.66
DLRM	23.55	29.29	102.77	26.92	18.29	25.84
AutoInt+	20.44	28.28	99.66	18.33	17.49	14.88
AutoCTR	19.89	26.49	97.06	12.31	7.12	3.02

benchmarks [28, 30]. (3) We further examine the influence of adopting the warm-start embedding in searching. Although it highly improves the performance of most architectures in the low-fidelity setting (not shown in Figure 4 due to the difference in scale), the performance on the full dataset is not improved. We attribute this observation to the overfitting issue of the warm-start embedding dictionary and the evaluation dataset during searching. Since we set a fixed search iteration (i.e., 1500 architectures) rather than adopting early-stop for the search algorithm, the overfitting issue may happen during the search process, which has also been pointed out by several recent works [12, 33].

Beyond accuracy, we also display the time complexity of the search algorithms in Table 1 and compare the model complexity of the best-discovered architectures of AutoCTR with and the SOTA human-crafted architectures in Table 2. The time for training the searcher could generally be ignored due to the limited size of the sampled architectures and the parallel CPU-GPU training schema we adopted. Results show that the explored architecture is smaller than the human-crafted ones on both the number of parameters and FLOPs. This is mainly because: (1) we explicitly constrain search space and adopt the complexity control term in the survivor selection, which restricts the exploration of overcomplicated architectures; (2) the searchers tend to find architectures with diamond or inverted triangle MLP structures, while the human-crafted ones directly adopt rectangular MLP structures as defined in the original work, and are set with 1024 units in each layer in our experiments.

6.4 Architecture Transferability Analysis

As the human-crafted architectures are not designed for a specific dataset, we explore the transferability of the searched architectures across different datasets. We select the best architectures searched by AutoCTR on each dataset, and apply them on the other two. From Table 3, we can observe that the architectures searched on one dataset could still perform well when applying to the others. This is mainly because: (1) the three benchmark datasets share common characteristics of feature relationships; (2) the blocks incorporated in the designed search space and the discovered connectivity among the blocks is general enough to uncover the high-order feature interactions of different CTR prediction datasets. Comparably, the architecture discovered on Avazu performs a bit worse when doing the transfer. One reason is that Avazu only contains sparse features, which results in no exploration of the dense features in searching. For these models, the dense features are only considered in the final embedding layer in our implementation during the transfer.

Table 3: Transferability of architectures found by AutoCTR

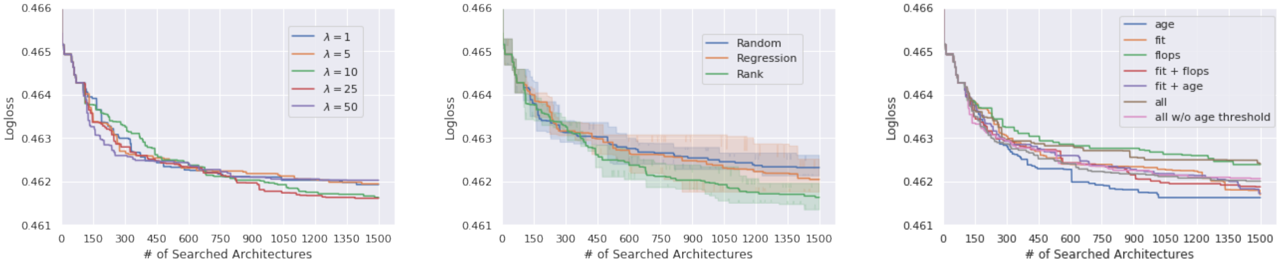
Target Dataset Original Dataset	Criteo	Avazu	KDD
Criteo	0.4413	0.3799	0.1520
Avazu	0.4421	0.3800	0.1535
KDD	0.4418	0.3803	0.1521

6.5 Hyperparameter Sensitivity Analysis

In this section, we study the sensitivity of AutoCTR on the key hyperparameters using the Criteo dataset and analyze the impact of the core components at different stages.

6.5.1 Effects of Selection Intensity (λ) in Parent Selection. We first analyze the influence of the selection intensity hyperparameter λ in the parent selection stage by fixing the population size as 100. As discussed in section 4.2, the higher the λ is, the more intense the selection would become. We choose $\lambda = 1, 5, 10, 25, 50$ and depict the curve of search effectiveness in Figure 5(a). With the increase of λ, the exploitation ability of AutoCTR generally increases while the exploration power decreases. It increases the initial search speed but would degrade the exploration ability in the long term. Under the current experimental setting, $\lambda = 25$ seems to be a more balanced option between exploitation and exploration.

6.5.2 Effects of Different Mutation Guider. Secondly, we focus on the mutation stage and analyze the influence of the different types of guiders to the search efficacy. Three types of guiders are compared here: (1) random guider (random): it generates the candidate offspring by randomly selecting and mutating an operation of the selected parent architecture. (2) fitness-based guider (regression): it uses a gradient-boosted tree to conduct regression on the explored architectures and their performance. It selects the best architecture among the 100 randomly generated neighbors of the parent architecture as the new candidate offspring. (3) rank-based guider (rank): the one we used in AutoCTR. Different from the fitness-based guider, it learns the pairwise ranking relationship among the architectures rather than directly fit their performance. We fine-tune the tree-based learner for both fitness-based guider and rank-based guider, respectively. From Figure 5(b), we can observe



(a) Effects of the selection intensity hyperparameter λ to the search process given population size p=100.

(b) Effects of different type of guider in the mutation stage to the search process.

(c) Effects of different survivor selection objectives.

Figure 5: Analysis of the key hyperparameters in the three stages of AutoCTR

Table 4: Performance and complexity comparison of architectures found with different survivor selection objectives

Objective	Performance		Model Size (Million)	
	Logloss	AUC	# Params	Flops
a_A	0.4418	0.8010	21.97	16.62
r_A	0.4417	0.8100	20.58	15.07
$a_A + r_A$	0.4415	0.8103	19.77	11.85
$a_A + c_A$	0.4418	0.8099	18.08	5.06
$r_A + c_A$	0.4417	0.8101	19.59	11.10
$a_A + r_A + c_A$	0.4415	0.8103	20.50	14.73
$a_A + r_A + c_A$ w/o threshold	0.4416	0.8102	19.35	10.14

that the AutoCTR with rank-based guider outperforms the other two. Although the fitness-based guider could improve search effectiveness comparing to the random strategy, it also suffers more on the overfitting issue, which results in the search variance to be large and makes it difficult to be tuned.

6.5.3 Effects of Different Survivor Selection Objectives. Finally, we explore the effect of adopting different objectives in the survivor selection stage. Figure 5(c) and Table 4 compare the search and final evaluation performance of AutoCTR with different search objectives. Except for the last objective, each of them adopts the age threshold $\mathbb{1}_{[a_A \leq q]}$ described in Equation (1). We set the trade-off weights for each term as 0.5. The results show that the age-based objective benefits more to the search speed comparing to the fitness-based objective. By adding the complexity constraint in the objective, the size of the best model explored could be reduced while the performance remains comparable.

6.6 Discussion

In this section, we visualize the best-explored architectures and analyze the importance of the block components learned from the tree-based guider. Some limitations and conjectures of the current study are discussed afterward to promote future exploration.

6.6.1 Case Study. We first visualize two of the best architectures found by AutoCTR on Criteo and KDD datasets, respectively, in Figure 6. It shows that both of the architectures ensemble multiple FM and DP blocks and tend to adopt MLP blocks in the later stage. Besides, dense features prefer MLP block while sparse features prefer DP and FM in the early stages, which shows the ability of DP and FM in modeling sparse features explicitly. Moreover, the MLP blocks display a diamond structure, i.e., MLP layers in the middle of the graph are wider than the ones in both ends, which aligns with some analysis in existing works [35]: diamond networks are preferable to the increasing/decreasing width networks (triangular networks) and the constant width networks (rectangular networks).

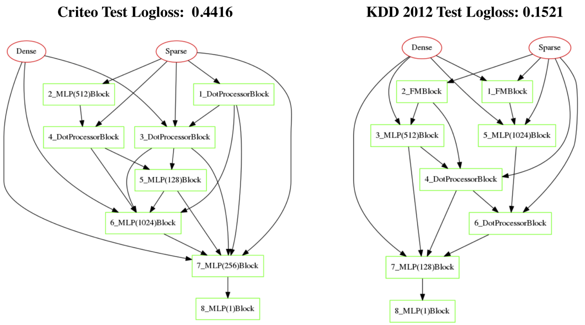


Figure 6: Two architectures found by AutoCTR.

6.6.2 Interpretation of Important Blocks. To provide a better understanding of the block-type influence to the architectures, we display the feature importance of high influential components learned from the <architecture, performance rank> pairs on Criteo and Avazu via the tree-based guiders. We randomly select 10, 000 architectures with seven valid blocks in each of them to train the tree-based guider and display the importance score of the top-20 influential block types in Figure 7. The "id_type" tick below the x-axis indicates the topological order of a block and its type. We observe that: (1) the structure of the polar blocks based on the topological order have larger impacts compared with the middle ones; (2) MLP block dominates the architectures; (3) DP and FM blocks are relatively more impactful on Avazu than Criteo since Avazu only contains sparse features. We need to emphasize that this interpretation may be biased by the search-space design and the way of representing the architectures. Interpreting the NAS process and involving the interpretations into the architecture design could be promising.

6.6.3 Limitations. Despite the analysis discussed above, several limitations are mentioned here for future investigation.

Search Space. Though the number of architectures contained in the search space is quite large (> 10^{11}), the number of block types we have currently explored is still limited. This also explains why Random and RL searchers could achieve acceptable performance. Moreover, the flexibility can be further enlarged via independent feature selection towards more dedicated and delicate interactions.

Overfitting. The overfitting issue is enlarged in the low-fidelity setting due to the limited subsample size and the stop criteria we

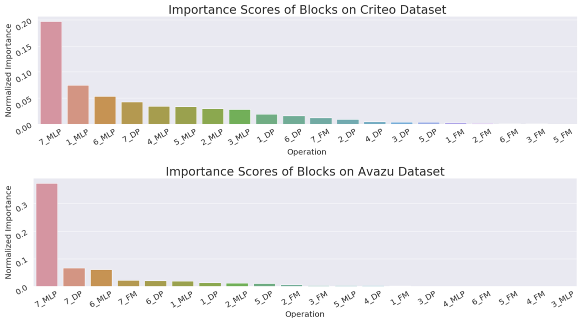


Figure 7: Normalized importance scores of top-20 block type operations learned by AutoCTR guider on Criteo and Avazu

adopted, i.e., search 1500 architectures for every searcher in each experiment. Although AutoCTR and LaNAS+ still show their superiority, the improvements compared with other searchers are weakened. One possible way to migrate this issue is to adopt early-stopping strategies or add regularizations for the search process [12, 33].

7 CONCLUSIONS AND FUTURE WORK

In this paper, we conduct a pilot study of automatically designing architectures for the CTR prediction task. We construct a hierarchical search space via wiring representative blocks extracting from human-crafted networks and explore the rank consistency among the architectures under the low-fidelity setting. A tailored evolutionary search algorithm with a multi-objective survivor selection strategy is proposed guided by an architectural-level learning-to-rank method. Experimental results on three benchmark datasets demonstrate the effectiveness of the proposed search algorithm and the feasibility of adopting low-fidelity estimation during the search phase. Future work includes considering independent feature interaction design, incorporating more diversified blocks to enrich the search space, as well as exploring more efficient search strategies.

8 ACKNOWLEDGMENTS

We would like to sincerely thank everyone who has provided their generous feedback for this work. Thank all the members of the Facebook personalization team for your feedback on the paper content, and thank the anonymous reviewers for their thorough comments and suggestions. This work is, in part, supported by DARPA Awards #W911NF-16-1-0565 and #FA8750-17-2-0116, and NSF Awards #IIS-1657196 and #IIS-1718840, granted to the co-author Xia Hu in his academic role at the Texas A&M University. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing any funding agencies.

REFERENCES

[1] Thomas Back. 1994. Selective pressure in evolutionary algorithms: A characterization of selection mechanisms. In *WCCI*.
[2] Tobias Blickle and Lothar Thiele. 1996. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation* (1996).
[3] Christopher JC Burges. 2010. From ranknet to lambdarank to lambdamart: An overview. *Learning* (2010).
[4] Guillaume MJ-B Chaslot, Mark HM Winands, and H Jaap van Den Herik. [n.d.]. Parallel monte-carlo tree search. In *ICCG*.
[5] Yukang Chen, Gaofeng Meng, Qian Zhang, Shiming Xiang, Chang Huang, Lisen Mu, and Xinggang Wang. 2019. RENAS: Reinforced Evolutionary Neural Architecture Search. In *CVPR*.
[6] Yi-Wei Chen, Qingquan Song, and Xia Hu. 2019. Techniques for Automated Machine Learning. *arXiv preprint arXiv:1907.08908* (2019).
[7] 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 *RecSys Workshop*.
[8] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural Architecture Search: A Survey. *JMLR* (2019).
[9] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: a factorization-machine based neural network for CTR prediction. In *IJCAI*.
[10] Xiangnan He and Tat-Seng Chua. 2017. Neural factorization machines for sparse predictive analytics. In *SIGIR*.
[11] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, et al. 2014. Practical lessons from predicting clicks on ads at facebook. In *ADKDD Workshop*.
[12] Yang Jiang, Cong Zhao, and Lei Pang. 2019. Neural Architecture Refinement: A Practical Way for Avoiding Overfitting in NAS. *arXiv:1905.02341* (2019).
[13] Haifeng Jin, Qingquan Song, and Xia Hu. 2019. Auto-keras: An efficient neural architecture search system. In *SIGKDD*.
[14] Yuchin Juan, Yong Zhuang, Wei-Sheng Chin, and Chih-Jen Lin. 2016. Field-aware factorization machines for CTR prediction. In *RecSys*.
[15] Liam Li and Ameet Talwalkar. 2019. Random search and reproducibility for neural architecture search. *arXiv:1902.07638* (2019).
[16] 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 *SIGKDD*.
[17] Chenxi Liu, Liang-Chieh Chen, Florian Schroff, Hartwig Adam, Wei Hua, Alan Yuille, and Fei-Fei Li. 2019. Auto-deeplab: Hierarchical Neural Architecture Search for Semantic Image Segmentation. In *CVPR*.
[18] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2019. DARTS: Differentiable Architecture Search. In *ICLR*.
[19] Qiang Liu, Feng Yu, Shu Wu, and Liang Wang. 2015. A convolutional click prediction model. In *CIKM*.
[20] H Brendan McMahan, Gary Holt, David Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, et al. 2013. Ad click prediction: a view from the trenches. In *SIGKDD*.
[21] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *arXiv:1906.00091* (2019).
[22] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. 2018. Efficient Neural Architecture Search via Parameters Sharing. In *ICML*.
[23] 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 *ICDM*.
[24] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2019. Regularized evolution for image classifier architecture search. In *AAAI*.
[25] 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 *ICML*.
[26] Steffen Rendle. 2010. Factorization machines. In *ICDM*.
[27] Christian Sciuto, Kaicheng Yu, Martin Jaggi, Claudiu Musat, and Mathieu Salzmann. 2019. Evaluating the search phase of neural architecture search. *arXiv:1902.08142* (2019).
[28] 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 *CIKM*.
[29] Linnan Wang, Saining Xie, Teng Li, Rodrigo Fonseca, and Yuandong Tian. 2019. Sample-Efficient Neural Architecture Search by Learning Action Space. *arXiv:1906.06832* (2019).
[30] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & cross network for ad click predictions. In *ADKDD*.
[31] Martin Wistuba and Tejaswini Pedapati. 2019. Inductive Transfer for Neural Architecture Optimization. *arXiv:1903.03536* (2019).
[32] Ling Yan, Wu-Jun Li, Gui-Rong Xue, and Dingyi Han. 2014. Coupled group lasso for web-scale ctr prediction in display advertising. In *ICML*.
[33] Arber Zela, Thomas Elsken, Tonmoy Saikia, Yassine Marrakchi, Thomas Brox, and Frank Hutter. 2020. Understanding and Robustifying Differentiable Architecture Search. *ICLR* (2020).
[34] Byoung-Tak Zhang and Jung-Jib Kim. 2000. Comparison of selection methods for evolutionary optimization. *Evolutionary Optimization* (2000).
[35] Weinan Zhang, Tianming Du, and Jun Wang. 2016. Deep learning over multi-field categorical data. In *ECIR*.
[36] 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 *SIGKDD*.
[37] Barret Zoph and Quoc V Le. 2017. Neural architecture search with reinforcement learning. In *ICLR*.
[38] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *CVPR*.

REPRODUCIBILITY SUPPLEMENTARY DOC

A DETAILED CONSTRUCTION OF THE THREE EXAMPLE BUILDING BLOCKS

We elaborate on the three building blocks used in our final experiments here, i.e., dense MLP block, Dotprocessor Block, and FM Block. Other blocks, such as outer product block [30] and self-attention block [28]„ etc., could also be easily integrated.

Dense MLP Block (MLP) The MLP block serves as the most commonly used building block in literature. We design the dense MLP block similar to the one applied in the DLRM model [21]. The input of it will be concatenated into a single long vector and transformed via a multi-layer perceptron with the ReLU activation function. We set the layer to be 1 for each MLP block to maximize the flexibility of the final constructed architectures. The width of each MLP block is searched within the unit set: {32, 64, 128, 256, 512, 1024}.

FM Block (FM) The FM block refers to the factorization machine block [26]. A conventional factorization machine model takes real-valued feature vector $\mathbf{x}$ as inputs, and outputs a single value via low-dimensional embedding and summation operations, as shown in Equation (3), where $\mathbf{v}$ denotes the embedding matrix, $\mathbf{w}$ is the transformation parameter. In the designed FM block, we assume all the input features are already transformed into the low-dimensional space and conduct the dot product and summation operations directly. Three specific designs are listed here for dimensionality alignment, and search space robustness: (1) We concatenate all the dense input features, including dense raw features and the dense outputs from other blocks, into a single vector. (2) If the dimensions of sparse inputs and the concatenated dense input are conflicted, we will linearly embed them into the same size in advance. (3) If only dense features are obtained as inputs, the block will degenerate to a linear embedding block with a sum pooling afterward.

$$\hat{y}(\mathbf{x}) := w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} < \mathbf{v}_i, \mathbf{v}_j > x_i x_j. \qquad (3)$$

Dotprocessor Block (DP) The DP block calculates the dot product of every pair of the input embeddings and concatenates the results into a single long vector. The self-interaction is also added, i.e., the dot product of each embedding vector and itself, so that when only dense features are collected as the block input, it is also feasible to proceed. In this case, this block degenerates into an element-wise square operator. Similar concatenation and dimensionality alignment strategy are used as for the FM block.

B BASELINE CONSTRUCTION DETAILS

B.1 Searcher Baselines

To prove the effectiveness of the proposed search algorithm, we select three representative NAS algorithms in existing work. We tailor random search and reinforcement learning based search to our designed search space and include a variation of LaNAS based on the proposed method and the parallel Monte Carlo Tree Search [4].

- **Random Search.** Recent work in NAS has pointed out that random search could be a strong baseline, even comparing to the most advanced search algorithm [15, 27]. We implement a random searcher as follows: given a random seed, in every search epoch, it randomly selects the four components described in section 3.2 for each block and builds an architecture based on the topological order.
- **Reinforcement Learning Based Search (RL)** . We adopt a single-layer RNN controller with length seven to generate architectures with seven intermediate blocks, as shown in Figure 8. The output of each step decides the structure of the next block and serves as the recurrent input for the next step. Initial inputs and hidden state values are set as zero by default. We use the REINFORCE algorithm to update the controller and use the logloss difference between the current architecture and the best architecture so far on the validation set as the reward. The average reward is used as a baseline to reduce variance. An entropy term is added in the loss function to enhance the exploration diversity.

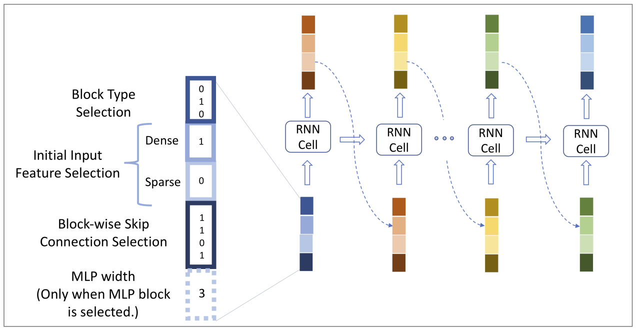

Block Type Selection
0
1
0
Initial Input Feature Selection
Dense
1
Sparse
0
Block-wise Skip Connection Selection
1
1
0
1
MLP width (Only when MLP block is selected.)
3
RNN Cell
RNN Cell
RNN Cell
RNN Cell

Figure 8: The illustration of the RL searcher

- **Latent Action Neural Architecture Search (LaNAS)** [29]. A sample-efficient Monte Carlo Tree Search (MCTS) algorithm, which has been proven to be effective than various advanced search algorithms, including the Regularized evolutionary algorithm [24] and Bayesian optimization algorithm in the image classification setting. The search strategy follows the standard MCTS algorithm with the upper-confidence bound (UCB) policy. The search space is partitioned via customizable regressors contained in the nodes of the tree. The search algorithm could narrow down to a specific subspace when traversing from the root to the leaf. Each regressor is updated based on the architectures in the corresponding node and their validation logloss.
- **LaNAS+**. We improve LaNAS from two perspectives.

(1) Convert its rollout sampling policy from random sampling to rank-based method borrowing the idea from the evolutionary method. Random rollout policy could be slow due to the lack of exploitation. Following the original paper, the random rollout policy here means when we have selected a path in the tree and collected the constraints in each node along the path to narrow down to a specific subspace, the way to sample an architecture from this subspace is naively random sampling. However, this could potentially cause a high rejection rate due to the complicacy of sampling from a non-convex polytope and may lose the usage of existed good architectures. To address the problem, we design an evolutionary style rollout policy: every time after narrowing down to a specific subspace, we retrieve the currently searched architectures within this subspace and

set them as the population. Then we do the same thing as we do in the AutoCTR, i.e., select a subset of candidates, pick the best one, and mutate it to a new architecture.

(2) Introduce virtual loss to enable parallel training of multiple architectures to improve the search speed. Within each update internal of the LaNAS searcher, we are supposed to sample multiple architectures in order to explore different tree branches. After each architecture is sampled, the back-propagate step requires the logloss of it to update the tree statistics so that it could conduct exploration on different branches rather than always exploring one single path. However, this may prevent the searching process from being parallelly proceeded since each new architecture can only be sampled after the evaluation of the former one. To circumvent this problem, a virtual loss [4] could be applied as padding during the back-propagate step to ensure the feasible adoption of parallel training and evaluation of multiple architectures. The virtual losses will be removed after the true loss is obtained.

B.2 SOTA Human-Crafted Networks

We select three representative human-designed networks to examine if the explored network is able to achieve comparable accuracy or even beat the SOTA human-crafted networks.

- **DeepFM** [9]: a two-tower model composed of a six-layer MLP with 1024 hidden units in each layer and factorization machine block, the sparse input embeddings are shared between the two components, and the output of the two components are linearly embedded with sigmoid transformation as the final prediction.
- **DLRM** [21]: a SOTA MLP-based recommendation model. It encodes dense features and sparse look-up embeddings with two MLP modules and embed their outputs with a top-level MLP module jointly. We adopt two single-layer MLP on the dense and sparse features respectively and stack a six-layer MLP on top of them. All MLP layers are with 1024 units except the final one.
- **AutoInt+** [28]: a two-tower model composed of a four-layer MLP with 1024 hidden units in each layer and three-layer self-interaction layer, which adopts the multi-head self-attention schema to learn high-order feature interactions. The outputs of the two components are linearly embedded with sigmoid transformation into the final prediction.

C EXPERIMENTAL SETTINGS

C.1 The Statistics of Three Benchmarks

Table 5: Statistics of Datasets

Datasets	# Samples	# Dense Features	# # Sparse Features	Total Cardinality of Sparse Features
Criteo[2]	45,840,617	13	26	998,960
Avazu[3]	40,428,967	0	23	1,544,488
KDD[4]	149,639,105	3	10	6,019,086

We would like to gratefully acknowledge the organizers of KDD Cup 2012 track 2 as well as the contributors of Criteo and Avazu for making these CTR prediction benchmarks publicly available.

C.2 Hyperparameter Settings

We elaborate on the detailed hyperparameter settings adopted in the experiments in this section. The three random seeds used in the experiments are 42, 2019, and 1234, respectively.

- **Search Phase Training:** *Adam* and *Sparse Adam* optimizers are adopted for dense and sparse features, respectively, in training. The batch size is set as 4096. The learning rate is 0.001. The hash size is 10^4 for all sparse features. The embedding tables for sparse features are randomly initialized based on a normal distribution with 0 mean and 0.01 standard deviation. The embedding size for each sparse feature is 16.
- **Final-Fit Phase Training:** Except for the hash sizes of all sparse features that are set back to their original cardinality, and the dataset is the full data, all the other settings are the same with the ones adopted in the search phase.

Hyperparameter settings for the searchers are elaborated as follows.

- **RL searcher:** the input encoding size and the LSTM hidden embeddings size are both set to 10, the trade-off hyperparameter of the entropy term is set to be 0.1.
- **LaNAS:** the tree-depth is set to 5. In the search phase, the update step is done once after 20 architectures are evaluated. The UCB trade-off parameter is set as 0.5. The space split classifier is defined as ridge regression with 0.1 regularization hyperparameter.
- **LaNAS+:** besides the hyperparameters mentioned above in LaNAS, the candidate size for the modified rollout policy in LaNAS+ is set as to be half of the architectures contained in a selected region. The virtual loss is set to be the mean logloss of the architectures contained in each leaf node.
- **AutoCTR:** the population size is set to be 100, and the survivor selection threshold q is set as 200. The trade-off hyperparameters μ_1, μ_2, μ_3 are set to be 1, 0.1, and 0.1, respectively. The parent selection trade-off hyperparameter λ is set to be 10. The guider is implemented with the lightgbm package with NDCG@3 as the early-stop evaluation metric.
- **AutoCTR (warm):** the warm-start embedding for each dataset is achieved from a four-layer MLP architecture with units: 128-1024-128-1, which is pretrained on each full dataset.

C.3 Software and Hardware Descriptions

All the deep learning related frameworks are implemented with the PyTorch package[5]. Every single search experiment is run on a single GPU (NVIDIA GeForce RTX 2080 Ti) with three architectures parallelly trained on it. Multi-core CPUs are used for data preprocessing and searcher training. Specifically, we use five cores for data preprocessing and the searcher training in the search phase, and 5 CPU cores + 1 GPU for the final fit of each discovered architecture after searching. We adopt parallel CPU-GPU training for searching, and the training speed is accelerated by loading, preprocessing, and saving the data batches in the GPU memory. The evolutionary guider is implemented with the lightgbm package[6], and the FLOPs are calculated based on the thop package[7]. Plots in the case study are drawn with the graphviz package[8].

[5] https://pytorch.org

[6] https://lightgbm.readthedocs.io/

[7] https://github.com/Lyken17/pytorch-OpCounter

[8] https://graphviz.readthedocs.io