AutoFeature: Searching for Feature Interactions and Their Architectures for Click-through Rate Prediction

Conference Paper · October 2020

DOI: 10.1145/3340531.3411912

CITATIONS
53

READS
792

6 authors, including:

Farhan Khawar
Hong Kong University of Science and Technology
16 PUBLICATIONS **110** CITATIONS

SEE PROFILE

Hang Xu
The University of Hong Kong
227 PUBLICATIONS **5,144** CITATIONS

SEE PROFILE

Ruiming Tang
Huawei Technologies
311 PUBLICATIONS **8,673** CITATIONS

SEE PROFILE

Xiuqiang He
Huawei Technologies
192 PUBLICATIONS **9,019** CITATIONS

SEE PROFILE

AutoFeature: Searching for Feature Interactions and Their Architectures for Click-through Rate Prediction*

Farhan Khawar[2], Xu Hang[1], Ruiming Tang[1],
Bin Liu, Zhenguo Li[1], Xiuqiang He[1]
[1]Huawei Noah's Ark Lab [2]The Hong Kong University of Science and Technology
fkhawar@connect.ust.hk; {xu.hang, tangruiming, li.zhenguo, hexiuqiang1}@huawei.com; liubinbh@126.com

ABSTRACT

Click-through rate prediction is an important task in commercial recommender systems and it aims to predict the probability of a user clicking on an item. The event of a user clicking on an item is accompanied by several user and item features. As modelling the feature interactions effectively can lead to better predictions, it has been the focus of many recent approaches including deep learning-based models. However, the existing approaches either (i) model all possible feature interactions for a given order, or (ii) manually select which feature interactions to model. Besides, they use the same network structure or function to model all the feature interactions while ignoring the difference of complexity among them. To address these issues, we propose a neural architecture search based approach called AutoFeature that automatically finds essential feature interactions and selects an appropriate structure to model each of these interactions. Specifically, we first define a flexible architecture search space for the CTR prediction task which covers many popular designs such as PIN, PNN and DeepFM, etc., and enables higher-order interactions. Then we propose an efficient neural architecture search algorithm that recursively refines the search space by partitioning it into several subspaces and samples from higher quality ones. Extensive experiments on multiple CTR prediction benchmarks show the superiority of our AutoFeature over the state-of-the-art baselines. In addition, our experiments show that the learned architectures use fewer flops/parameters and hence can efficiently incorporate higher-order feature interactions. This further boosts the performance. Finally, we show that AutoFeature can find meaningful feature interactions.

KEYWORDS

CTR Prediction, Neural Architecture Search, Feature Generation, Feature modelling.

*This work was done when Farhan Khawar was a research intern at Huawei Noah's Ark Lab and a PhD student at HKUST. He is now affiliated with Kira Systems. Xu Hang and Ruiming Tang are the corresponding authors. Bin Liu was at Huawei Noah's Ark Lab for this work and he is now at ByteDance.

CIKM '20, October 19–23, 2020, Virtual Event, Ireland

ACM ISBN 978-1-4503-6859-9/20/10…$15.00
https://doi.org/10.1145/3340531.3411912

ACM Reference Format:
Farhan Khawar, Xu Hang, Ruiming Tang, Bin Liu, Zhenguo Li, and Xiuqiang He. 2020. AutoFeature: Searching for Feature Interactions and Their Architectures for Click-through Rate Prediction. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM'20), October 19–23, 2020, Virtual Event, Ireland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3340531.3411912

1 INTRODUCTION

Recommender systems are pervasive in online item consumption these days. One common way to provide recommendations is to perform the click-through rate (CTR) prediction. CTR estimates the probability that a user will click on an item. CTR is especially significant in applications involving online advertisement/purchase where the recommendations are made based on CTR × utility, and utility is the revenue the click generates. Better CTR prediction is a crucial aspect in determining the revenue of the system [9, 11].

When a user clicks on an item, this interaction is accompanied by the user and item features. A natural approach would be to model the *feature interactions* and use these interactions for CTR prediction. Since the dependencies among these features are unknown, recent works like [15, 21, 22] have used neural networks to learn them. These approaches start by embedding the categorical features to get continuous low dimensional representations. This is followed by a product layer that explicitly models the feature interactions by performing some operations over the product of feature vectors. The utility of explicitly performing feature interaction in the product layer has been shown before [12, 15, 21, 22]. Finally, these feature interaction vectors are passed through fully connected neural network layers (DNN) with a softmax unit at the end to give the prediction.

The core of these approaches is the product layer, which performs the feature interaction step. However, existing approaches assume that *all* features interact with each other and that each of these interactions should be modelled to the *same* degree. Considering second-order features interactions of the form FEATURE1×FEATURE2, we may observe that (i) not all pairs of feature interactions are relevant, and (ii) not all pairs of features need to be modelled to the same degree. For example, modelling the feature interaction DEVICE_TYPE× WEEKDAY may be unnecessary since these features are likely to be independent, whereas the feature interaction USER_ID×LOCATION might be more interesting. In addition, the interaction of USER_ID× LOCATION appears to be more complex than USER_ID×DEVICE_ID, therefore, we might need a more complex function to model it.

Moving beyond second-order feature interactions, higher-order feature interactions (p^{th} order where $p \geq 2$) are desired since they would let us model the richer interactions of three or more features.

However, since existing approaches model all possible feature interactions for a given order, it either (i) becomes computationally prohibitive to go beyond second-order such as for PIN [22] and IPNN [21] or (ii) includes useless/noisy feature interactions as in xDeepFM [15]. Thus, we need a method that can model higher-order feature interactions, which does not model unnecessary feature interactions and models different feature interactions differently depending on the type of their interaction.

In this paper we propose a novel neural architecture search (NAS) approach to find (i) which feature interactions to model (ii) what network structure/operations to use for modelling these interactions. We call this method AutoFeature. The network used for modelling a feature interaction is called a sub-network. The structure of a sub-network can range from null, in which case the interaction is not modelled, to complex. To perform this search we first encode each sub-network to a string of operations and then concatenate the strings for all sub-networks to form the product layer string called an architecture. We then propose a search algorithm that takes these product layer strings and searches for the best architecture by recursively partitioning the search space to reveal promising subspaces. It then samples new architectures from the subspaces, based on the quality of architectures produced by each subspace, such that it samples more from the subspace that gives better architectures.

To accomplish this, we propose an architecture search space that is general enough to encompass existing feature interaction methods. However, since the architecture space is inherently large, we propose a novel search algorithm to reduce it into subspaces of different qualities. Existing NAS algorithms either don't work well or can't be applied to our scenario. The unique challenge of applying NAS to this domain is that we have many structures (sub-networks) to search as a part of one architecture unlike searching for a single structure (cell) in computer vision tasks.

We use a tree of Naive Bayes classifiers to recursively classify the architectures into promising and not so promising regions until the leaf nodes are reached. Each leaf node represents a subspace of the possible architectures that can be sampled from. Thus, we can find promising subspaces and sample from them to get good architectures. However, to maintain an explore-exploit trade-off, we use the idea of the Chinese Restaurant Process to select the leaf node to sample from. The Chinese Restaurant Process ensures that the leaf node subspace that provides good architectures is sampled more, while at the same time the non-promising subspaces are also explored with some probability.

Our AutoFeature CTR architecture uses a much fewer number of flops and parameters than the corresponding state-of-the-art PIN models and gives better performance. In addition, the learned feature interactions are meaningful. The main contributions of this paper are:

- We introduce NAS for learning the structure of feature interactions in neural network based CTR prediction models.
- As a part of AutoFeature, we present a NAS search space for the CTR prediction task and a NAS algorithm (NBTree) that recursively reduces the search space. We use this approach to find which feature interactions to model and the structure of these interactions.
- We show that this allows efficient modelling of higher-order features as AutoFeature uses much fewer parameters and flops compared to the baseline PIN model.
- We perform experiments on three large scale publicly available datasets and show that we outperform the state-of-the-art feature interaction models on the CTR prediction task.

2 RELATED WORK

2.1 Feature Interactions in CTR Prediction

The key challenge in the CTR Prediction task is how to effectively model feature interactions [12, 15, 20, 22]. Generalized linear models (LR), such as FTRL [20], perform well in practice. However, they are not able to learn feature interactions. A common trick to utilize them in the practice is manually feature interactions. However, they are hard to generalize to higher order feature interactions or those never or rarely appear in the training data. Factorization Machines (FM) [25] and its variant [13] model second-order feature interactions as the inner product of latent vectors representing the corresponding features.

Deep learning has also been used for the CTR prediction task. Modelling feature interactions takes an important role in these models and, therefore, different methods to model feature interactions are proposed. Wide & Deep [9] relies on its wide component to learn second-order and its deep component (an MLP) to learn high order feature interactions. Similarly, DeepFM [12] and IPNN [21] use an FM layer to learn second-order and MLP to learn high order feature interactions. DIN [33] and AFM [29] utilize attention network to differentiate contribution of each second-order feature interactions. A micro-network is used for each second feature interactions in PIN [22], to learn different interaction models for each pair of fields. CCPM [19] extends CNN for CTR prediction, but it is biased to the interactions between neighboring features. The above mentioned models either modelling second-order feature interactions by enumerating and performing operations as (weighted) inner product or micro-networks, or modelling high order feature interactions implicitly by MLP. Higher-order feature interactions cannot be enumerated as second-order ones due to the curse of dimensionality. CrossNet [28] and xDeepFM [15] apply a cross operation and a compressed interaction respectively, to explicitly learn p^{th} order feature interactions from $(p\text{-}1)^{th}$ order ones. However, these two works still try to enumerate all the feature interactions.

The existing deep learning models for CTR prediction task suffer from two limitations. *First*, they enumerate all the feature interactions or require human efforts to identify important feature interactions. Useless interactions may bring unnecessary noise and complicate the training process. *Second*, they use the same operation or network structure to learn feature interactions. For instance, inner product is performed in [12, 13, 21, 25, 29], and the same network architecture is used in [22].

2.2 Neural Architecture Search

NAS aims at automatically finding a neural network architecture for a certain task and dataset without the labor of designing networks, and has drawn an increasing interest. Most works are based on searching CNN architectures for image classification [6, 8, 16] while only a few of them focus on other tasks such as NLP. To

the best of our knowledge, no work has explored the use of neural architecture search for the recommendation problem. There are mainly three approaches in NAS area: 1) Reinforcement learning based methods like [3, 4, 32, 34] train an RNN policy controller to generate a sequence of actions to design cell structure for a specific CNN architecture; 2) Sample-based methods like [17, 23, 24] try to "evolve" architectures or employ Network Morphism by mutating the current best architectures and explore new potential models; 3) Gradient based methods like [5, 18, 30] relax the search space to be continuous instead of searching on a discrete set of architectures. This allows the searched architecture to be optimised using gradient descent based on its validation performance. They also often use weight sharing among searched architectures. Among these approaches, gradient based methods are fast but not so reliable since weight-sharing causes a big gap between the searching and the final training. RL based methods usually require careful design and adjustment of the RL controllers and they generally require a large number of samples to converge. Sample-based methods focus on finding techniques to sample good candidates architectures for evaluation. In [27] a Monte Carlo Tree Search (MCTS) with an inefficient and fixed separation of nodes was used. Each node represented a particular operation and the decision at each node was whether to include this operation or not. The MCTS based approach was designed to search for a single network structure, however, we need to search for many sub-networks as a part of one model. Therefore, using the MCTS based tree for the CTR problem with many sub-networks will result in a huge tree which is infeasible to search over. In this paper, we propose a new efficient search algorithm which leads to a more flexible tree search and is capable of searching for multiple sub-network structures simultaneously.

3 THE CTR MODEL STRUCTURE

Deep learning models for CTR prediction [12, 19, 22] have three general components: (i) the embedding layer, (ii) product layer, and (iii) the DNN classifier. The product layer is a central part of these models and performs explicit feature interactions. While we also follow this general structure and use the standard embedding and DNN layers, we propose a novel NAS based approach called AutoFeature to decide which interactions to model in the product layer and what network structure or operations to use for each of these interactions. Also, we enrich our product layer with higher-order feature interactions. The overall structure of AutoFeature CTR model is shown in Figure 1. We now briefly introduce the three components of the CTR prediction model.

3.1 Embedding Layer

The raw input features are categorical or numeric and multi-field. We follow a similar procedure to previous works [21, 22] to first represent the raw features as binary vectors and then embed them to continuous vectors using a field-wise embedding. The categorical data is transformed via one-hot encoding to a binary vector e.g., [0,1] for females and [1,0] for males. The numeric data is first discretized into buckets and then we have a one-hot encoding for each bucket. For example, we can bucket and encode age into child (0-14 years) [0,0,0,1], youth (14-24 years) [0,0,1,0], adult (25-64 years) [0,1,0,0] and elderly (≥ 65 years) [1,0,0,0].

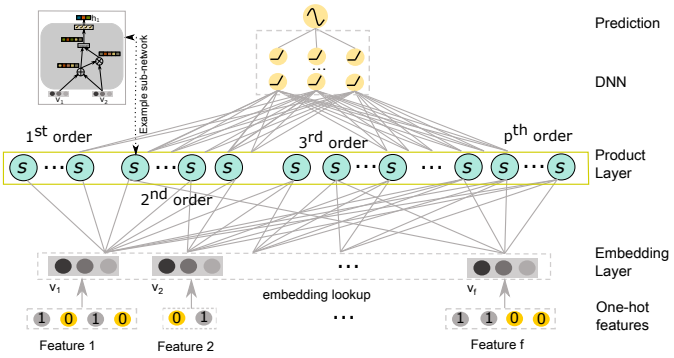

Figure 1: The AutoFeature CTR prediction model.

The one-hot encodings of all features are then concatenated to form a high-dimensional sparse feature vector. Due to the sparsity of this vector and various lengths of different feature vectors, we embed the one-hot encoding of each feature separately to a continuous vector of size n using a linear embedding i.e., the embedding $\mathbf{v}_i$ of feature i is $\mathbf{v}_i = \mathbf{E}_i\mathbf{x}_i$, where $\mathbf{E}_i$ is the embedding matrix and $\mathbf{x}_i$ is the one-hot feature vector respectively for feature i. For multi-valued features, we use a procedure similar to [22] where we average the embeddings.

3.2 Product Layer

The nucleus of the deep learning models for CTR prediction is the product layer. Our product layer models both the lower and higher-order explicit feature interactions. If we consider a maximum of fourth-order interactions then we will model feature interactions with order from 1 to 4. For each order, there will be several sub-networks that will model the feature interactions for that order. Each sub-network is a neural network and is responsible for modelling the interactions of the input features. Each sub-network will output a vector of dimension m and its input will be p n-dimensional feature embedding vectors, where p is the order of feature interaction modelled by this sub-network. The outputs of the sub-networks for all orders are concatenated to form a representation vector $\mathbf{h}$ of feature interactions.

In existing works, all p^{th} order feature interactions were modelled for order p. In contrast, we will model selected feature interactions only. The identity of the features input to these sub-networks and the structure of each sub-network will be determined by our NAS algorithm. Example sub-networks are shown in Figure 8.

3.3 DNN Classifier

The utility of combining the explicit feature interaction vector $\mathbf{h}$ with a DNN classifier to learn more non-linear patterns has been validated previously [15, 21, 22]. The output $\mathbf{h}$ of the product layer is fed into a multi-layer perceptron (MLP) with fully connected layers. The output of the final layer is fed to a sigmoid unit which predicts the probability $\hat{y}$ of the user clicking on the item.

To train the whole CTR model we use the cross-entropy loss (log-loss) function:

$$L(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}), \tag{1}$$

where, y is the binary ground truth label.

4 NAS PROCEDURE

To search for the architecture of the sub-networks, we need a search space to search over, and a search algorithm to carry out this search. In this section, we first define our search space and then introduce our search algorithm that searches for the best architecture by recursively partitioning the search space into subspaces and then samples from these subspaces based on their quality.

4.1 Search Space

The search space for each sub-network is defined in terms of a series of operations. Each operation results in an output vector and it involves the application of an operator on the output of the previous p operations. The value of p is set to be equal to the order of the feature interactions. The first operation always operates on the inputs of the sub-network which are p in number. The output of the final operation is passed through a feed-forward layer to reduce the dimension of the sub-network output to m. The sequence of operations of each sub-network form a directed acyclic graph (DAG). The following operations are available:

- *Pointwise* Addition ($\oplus$): takes input vectors of dimension n and outputs a vector of dimension n that contains their element-wise sum. It has no parameters.
- *Hadamard Product* ($\otimes$): takes input vectors of dimension n and outputs a vector of dimension n that contains their element-wise product. It also has no parameters.
- *Concatenation Layer* $\boxed{concat}$: takes input vectors of dimension n, concatenates them and passes them through a feed-forward layer with a Relu activation function to reduce the dimension of the output vector to n. This dimensionality reduction is necessary so that the output of this operation can be used as the input for other operation.
- *Generalized Product* ($\boxtimes$): takes input vectors of dimension n, passes their element-wise product through a feed-forward layer to output a vector of dimension n.
- *Null* ($\emptyset$): denotes that an operation does not exist. If there is no operation after it then the output is taken from the previous operand if it exists. Otherwise, if there is an operation after $\emptyset$, then the sequence of operations is invalid.

A sample sub-network for second-order interaction (p=2) and three operations is shown in Figure 8 (c). The inputs of the sub-network are the feature vectors BANNER_POS and APP_ID.

Each sub-network can be represented by a series of operations and encoded as a string. For example, a sub-network that performs three operations: pointwise addition, concatenation layer and hadamard product in this order can be represented by the string '1, 3, 2' if we use the following encoding $\oplus = 1, \otimes = 2, \boxed{concat} = 3, \boxtimes = 4, \emptyset = 0$. The set of the encoding symbols is $F = \{0, 1, 2, 3, 4\}$. Using this encoding we can have a string denoting that the sub-network performs two (e.g., '2, 4, 0'), or one (e.g., '3, 0, 0') or no (e.g., '0, 0, 0') operations. In the case the sub-network performs no operations, by definition the corresponding sub-network does not exist and the corresponding feature interaction is not modelled. Let '$\cdot$'$\in F$, then we note that by definition of $\emptyset$, the sequence of operations '$0, *, *$' and '$\cdot, 0, *$' is not valid when '$*$'$\in F \setminus 0$. However, it is valid if '$* = 0$'. By using this scheme, we can represent sub-networks performing a different number of operations, ranging from 0 to l, where l is the maximum number of operations allowed for each sub-network. This will allow us to search for which feature interactions to model and to what degree to model them. We also note that even if two sub-networks perform the same number of operations, the type of operations can also determine a simple v.s. a complex interaction. Finally, we can concatenate the encoding strings of all sub-networks to form an encoding of the product layer and call this encoding an *architecture*.

4.2 Search Algorithm

We first present the search algorithm in the context of second-order interactions and later show how to extend it for p > 2. If there are f features then there are $O(f^2)$ sub-networks for order-2 (p=2) feature interactions. The length of the encoding for the product layer then becomes $O(lf^2)$ (recall that l is the maximum number of operations allowed for each sub-network). This represents a large search space of around $O(5^l lf^p)$ architectures, therefore, we propose to recursively partition the search space into subspaces and then search over these subspaces. We explore promising subspaces more by sampling more architectures from them, while also sampling from less promising subspaces so that samples from the whole search space are potentially explored.

4.2.1 Partitioning the search space. Each architecture can be viewed as a point in the search space. We first randomly sample some architectures from the search space, train them and get their accuracies[1]. We then label this original population according to its quality by labelling all architectures with accuracy more than a predefined threshold with a 1 and 0 otherwise. That is, the label z_i of architecture i is:

$$z_i = \begin{cases} 1 & \text{if } acc_i \geq T, \\ 0 & \text{if } acc_i < T, \end{cases}$$

where, acc_i is the accuracy of the i^{th} architecture. We set T to be equal to the accuracy at the 90^{th} percentile[2]. Together the architecture and accuracy pairs form a dataset that will be used for searching for the best architecture.

To partition the search space we use a tree of Naive Bayes classifiers (NBTree) as shown in Figure 2. Each node in this tree corresponds to a Naive Bayes classifier with a multi-nominal likelihood and it possesses a dataset of architectures and accuracies which constitute its population. Each node also has its own threshold T based on its population. Each node then partitions the architectures into two parts. The architectures which are classified as 1 are sent to the left branch of the tree and others go to the right branch where they are partitioned again until a predefined depth d is reached.

We can see from Figure 2 that each internal node of the NBTree learns a decision boundary that partitions its search space into two parts. The leaf nodes represent the subspaces from which we will sample new architectures. By construction, the leftmost leaf subspace represents the most promising subspace while the rightmost subspace represents the least promising subspace.

The choice of NB as a classification node is motivated by its speed/simplicity, and because the elements of the architecture string

[1] In this paper the accuracy is measured by AUC.
[2] Section 5.8 shows the effect of different values of T.

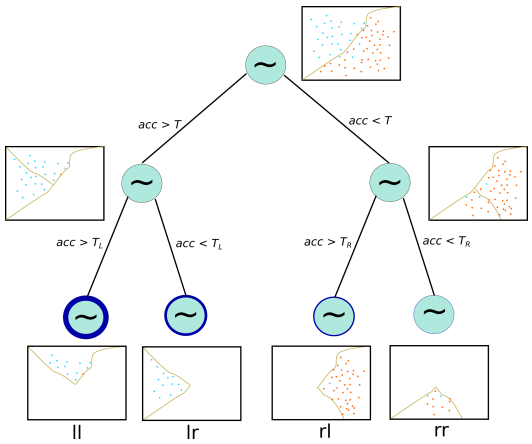

Figure 2: The Naive Bayes tree recursively partitions the search space into disjoint regions. Sample architectures are obtained from the leaf node subspaces. The width of the blue outline of a leaf node denotes the probability of sampling from this node. Each node represents a Naive Bayes classifier and possesses a dataset of architectures and their accuracies. Each architecture is a point in the search space.

are nominal variables. However, the general structure is open to other choices of classifiers. We leave this exploration as future work.

4.2.2 Sampling from leaf nodes. Once the subspaces of the leaf nodes are available we can sample from them. We would like to sample more from the leaf nodes that give us architectures with higher accuracies while also sampling from other less promising leaf nodes. To achieve this, we draw samples from the leaves based on the Chinese Restaurant Process (CRP) [1]. The idea behind CRP is that the probability of sampling from a node is proportional to the number of samples drawn from the node previously. Specifically, the probability that the next sample (architecture) comes from leaf k, i.e., $x_{N+1} = k$ is:

$$P(x_{N+1} = k|\mathbf{c}) = \frac{n_k}{N+1}, \tag{2}$$

where, N is the total number of samples drawn from the CRP so far, n_k is the number of samples from leaf node k and $\mathbf{c}$ is the vector of the leaf node counts i.e., $\mathbf{c} = \left[n_1, n_2, \ldots, n_{(2^d)}\right]$. We then update the count for node k as:

$$n_k = n_k + e^{C(acc_{x_{N+1}} - \tau)}, \tag{3}$$

where, τ is our target accuracy above which we will reward a sample, $acc_{x_{N+1}}$ denotes the accuracy of the last sample drawn from leaf k and C is a constant. This definition of count ensures that the leaf nodes that give higher accuracies are sampled more.

The counts n_k for $k \in \{1 \ldots 2^d\}$ can be initialized by giving more count to the leftmost leaf ($k = 0$) and the least to the rightmost leaf ($k = 2^d$). We used the initialization $n_k = 2^d e^{-0.75k}$.

This Chinese Restaurant Process ensures that the promising leaf node subspaces are sampled more, while at the same time the non-promising subspaces are also explored.

4.2.3 Generate a sample. To generate a sample from a leaf node we can randomly sample architectures and check if they belong to the leaf node's subspace. However, we may have to generate a lot of samples before we get a sample from the desired subspace. A better way would be to pick existing samples from the leaf node and randomly modify them to generate new samples. More concretely, we sort the samples of the leaf node in descending order with respect to their accuracies and then pick the top two samples to perform a crossover operation at the midpoint of the architecture string. This is followed by q mutations. We then check if the resulting architecture belongs to the subspace represented by the leaf node. This is done by recursively classifying the sampled architecture starting from the root until a leaf node is reached, and then checking if the sample is classified to belong to the desired leaf node. If this is not the case then the procedure is repeated by choosing a different crossover point and/or different parents for mutation. The complete procedure is outlined in Algorithm 1.

Algorithm 1 Sample(leaf, D)

```
archs ← sorted_archs = sort D w.r.t accuracy
c_p = midpoint                          ▷ The crossover position
while offspring ∉ leaf_subspace do
    parents = pick top two architectures from archs
    offspring = perform crossover of parents at position c_p
    offspring = perform q mutations on the offspring
    remove the top architecture from archs
    if #iterations > max_iter then
        c_p = c_p − 1; archs ← sorted_archs
return offspring
```

Algorithm 2 NBTreeSearch(D, d, #*samples*)

Inputs: D — a dataset of architecture strings and their accuracies, d — the height of the tree, #*samples* — number of samples to draw each time from CRP.

```
1:  tree = NB.learn(D, d)                    ▷ train the NBTree of depth d
2:  Initialize the CRP counts, c, for the leaves
3:  while accuracy < desired do
4:      leaf_ids = CRP(c, #samples )                    ▷ get leaf ids
5:      leaf_nodes = tree.get(leaf_ids)                 ▷ get leaf nodes
6:      for leaf in leaf_nodes do
7:          archs ← Sample(leaf, leaf.D)                ▷ sample architectures
8:      accs=train(archs)  ▷ train recommenders to get accuracies
9:      c =updateCRP(acc, leaf_nodes)   ▷ update the CRP counts
10:     tree = updateNodes(archs,accs)              ▷ update tree nodes
11:     if  #iterations > max_samples then
12:         tree = NB.learn(tree.root.D, d)         ▷ re-train the NBTree
13:     if max(accs) > accuracy then accuracy = max(accs)
```

4.2.4 Putting it together. The overall search procedure is outlined in Algorithm 2. The search starts with a bootstrapping dataset D of random architectures and their accuracies. This is used to train the initial NBTree. The CRP counts $\mathbf{c}$ are then initialized for the leaf nodes in line 2. These counts are used to draw the IDs of the leaf_nodes from which the architectures will be sampled (line 4-7). Each of these sampled architectures defines a product layer and they are then trained and evaluated to get their accuracies (line 8).

Once the sampled architectures are trained, we update the CRP counts of the leaf nodes based on Equation 3 (line 9). For each sampled architecture, we then update the datasets of all its path nodes,

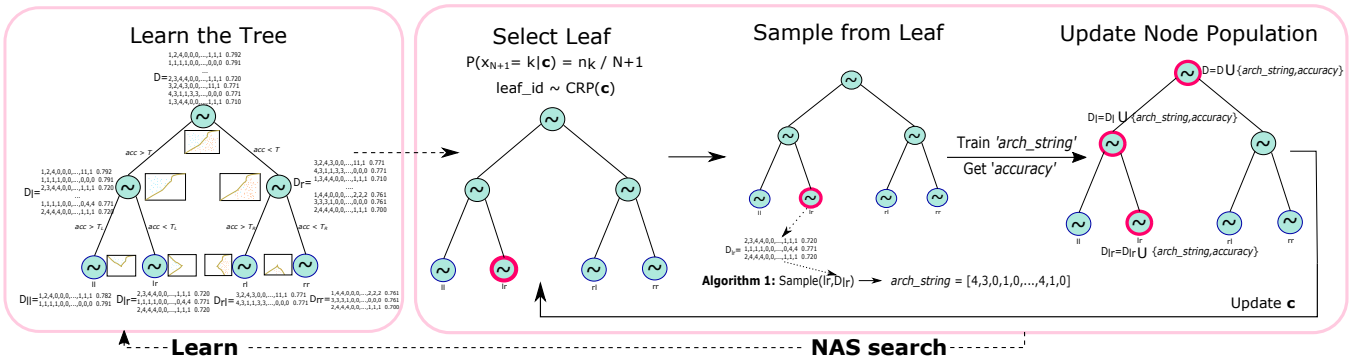

Figure 3: The overview of our proposed neural architecture search algorithm. AutoFeature includes two iterative search phases: 1) Learning the partitions of the tree; 2) Sampling and training. During phase one, we learn the subspaces based on the current architectures at each node. During phase two, we select a leaf according to CRP and randomly sample an architecture from this node. This architecture is fully trained and the result is used to update the population of each path node.

Table 1: The percentage of parameters and flops in the product layer of AutoFeature compared to the PIN model of the corresponding order. The numbers are for the best performing AutoFeature models.

	Criteo		Avazu		iPinYou	
	% Param.	% flops	% Param.	% flops	% Param.	% flops
AutoFeature(2)	28.85	29.23	52.93	53.21	35.67	36.05
AutoFeature(3)	1.44	1.48	5.42	5.44	6.95	7.39
AutoFeature(4)	0.10	0.11	0.85	0.84	1.94	1.95

that lie on the branch the sample came from, with the sample's architecture string and accuracy (line 10). The dataset of a non-leaf node is the union of the datasets of its children. After a few such iterations, the NBTree is rebuilt (line 12). This rebuilding process refines the NBTree, and hence the leaf subspaces, by moving the best architectures to the leftmost leaf and least favourable architectures to the rightmost leaf. The whole procedure continues until the desired accuracy is achieved or the maximum number of steps is reached.

This whole process is shown in Figure 3. For a clearer depiction we show one architecture being sampled at a time during the search phase, however, in practice we can draw multiple leaf IDs from the CRP stochastic process at one time. A benefit of having a probability distribution on the leaves is that we can take samples from multiple leaf nodes during one iteration and train these samples in parallel. We simply draw multiple leaf IDs from the CRP using the current count vector $\mathbf{c}$ and then pick a sample randomly from each of these leaf subspaces. The value of *#samples* is bounded by the parallelism of the machine.

4.2.5 Higher-order Features. Since the number of feature interactions grows exponentially with the order of feature interactions, enumerating them all in the encoding string is not possible for higher-order interactions. For example, for third-order, the number of feature interactions are $O(f^3)$. This limits the modelling capacity of PIN and IPNN [21, 22] models. Even if compression techniques are employed as in xDeepFM, this still results in unnecessary/noisy interactions being modelled. However, using our NAS approach we can search which feature interactions should be modelled. Taking the example of a third-order sub-network string, we achieve this by modifying our sub-network string from *'operation1, operation2, operation3'* to *'feature_id1, feature_id2, feature_id3, operation1, operation2, operation3'* and have a maximum of K such sub-strings. We then let AutoFeature decide the values of the feature IDs and operations for each sub-network. Then we follow the same search procedure as before by using a separate NBTree for each order of feature interactions. The product layer string is formed by concatenating the sampled strings from the p trees. This string is used to define the product layer of the CTR model in Figure 1.

5 EXPERIMENTS

We conduct extensive offline experiments on three publicly available benchmarked datasets to investigate:

- The recommendation performance of AutoFeature compared with the state-of-the-art baselines.
- The comparison in terms of the number of parameters and flops between AutoFeature and the best baseline (PIN).
- The effect of removing different orders of feature interactions and removing the NBTree.
- The internal working of the AutoFeature search and the effect of different search parameters.

5.1 Experimental Setup

5.1.1 Datasets. To make a fair comparison with the state-of-the-art baselines, the three datasets were pre-processed exactly as [22]. We describe the datasets and the pre-processing steps below.

Criteo[3]: contains click logs from the Criteo display advertisement challenge 2013. The data from day "6-12" as used as the training set and day "6-13" is used for evaluation. Following [22], the label imbalance is countered by negative sampling to keep the positive ratio at roughly 50%. The thirteen numerical fields are converted into one-hot features through bucketing, where the features in a certain field appearing less than 20 times are set as a dummy feature "other".

Avazu[4]: was released as a part of a CTR prediction challenge on Kaggle. 80% of randomly sampled data is used for training and validation and the remaining 20% is used for testing. Categories appearing less than 20 times were removed.

[3]https://labs.criteo.com/2013/12/download-terabyte-click-logs/
[4]http://www.kaggle.com/c/avazu-ctr-prediction

Table 2: Parameter Settings for different datasets.

Params	Criteo	Avazu	iPinYou
General	bs=2000 opt=Adam lr=1e-3	bs=2000 opt=Adam lr=1e-3	bs=2000 opt=Adam lr=1e-3 l2_e=1e-6
LR	–	–	–
GBDT	depth=25 #tree=1300	depth=18 #tree=1000	depth=6 #tree=600
FM,AFM	n=20 t=0.01 h=32 l2_a=0.1 sub-net=[40,1]	n=40 t=1 h=256 l2_a=0 sub-net=[80,1]	n=20 t=1 h=256 l2_a=0.1 sub-net=[40,1]
FFM	n=4	n=4	n=4
CCPM	n=20 kernel=[7 × 256] net=[256 × 3,1]	n=40 kernel=[7 × 128] net= [128 × 3,1]	n=20 kernel=[7 × 128] net=[128 × 3,1]
xDeepFM	n=20 net=[400 × 3,1] CIN=[100 × 4]	n=40 net=[700 × 5,1] CIN=[100 × 2]	n=20 net=[300 × 3,1] CIN=[100 × 4] LN=true
FNN,DeepFM, IPNN	n=20 net=[700 × 5,1] LN=true	n=40 net=[500 × 5,1] LN=true	n=20 net=[300 × 3,1] LN=true
PIN	n=20 net=[700 × 5,1] sub-net=[40,5] LN=true	n=40 net=[500 × 5,1] sub-net=[40,5] LN=true	n=20 net=[300 × 3,1] sub-net=[40,5] LN=true
AutoFeature CTR Model	n=20 net=[2048,1024, 512,256,1] LN=true	n=40 net=[500 × 5,1] LN=true	n=20 net=[300 × 3,1] LN=true
AutoFeature Search	q=80, T=90,m=5 max_samples=24, #samples=8, τ=0.8015, d=2	q=80, T=90,m=5 max_samples=24, #samples=8, τ=0.7887, d=2	q=80, T=90,m=5 max_samples=24, #samples=8, τ=0.7843, d=2

* Note: bs=batch size, opt=optimiser, lr=learning rate, l2_e=l_2 regularisation on embedding layer, t=softmax temperature, l2_a=l_2 regularisation on attention network, h=attention network hidden size, n=embedding size, net=MLP structure, sub-net=sub-network, LN=layer normalisation, BN=batch normalisation.

Table 3: Dataset statistics

Dataset	instances	dimensions	features	positive ratio
Criteo	1×10^8	1×10^6	39	50%
Avazu	4×10^7	6×10^5	24	17%
iPinYou	2×10^7	9×10^5	16	0.07%

Table 4: Overall Performance
$\star : p < 10^{-12}$ $\Diamond : p < 10^{-3}$ (two-tailed t-test)

Model	Criteo		Avazu		iPinYou	
	AUC (%)	logLoss	AUC (%)	logLoss	AUC (%)	logLoss
LR [14]	78.00	0.5631	76.76	0.3868	76.38	0.005691
GBDT [7]	78.62	0.5560	77.53	0.3824	76.90	0.005578
FM [25]	79.09	0.5500	77.93	0.3805	77.17	0.005595
FFM [13]	79.80	0.5438	78.31	0.3781	76.18	0.005695
CCPM [19]	79.55	0.5469	78.12	0.3800	77.53	0.005640
FNN [31]	79.87	0.5428	78.30	0.3778	77.82	0.005573
AFM [29]	79.13	0.5517	78.06	0.3794	77.71	0.005562
DeepFM [12]	79.91	0.5423	78.36	0.3777	77.92	0.005588
xDeepFM [15]	80.06	0.5408	78.55	0.3766	78.04	0.005555
IPNN [21]	80.13	0.5399	78.68	0.3757	78.17	0.005549
PIN [22]	80.18	0.5393	78.72	0.3755	78.22	0.005547
AutoFeature(2)	80.20	0.5395	78.74	0.3753	78.27	0.005543
AutoFeature(3)	80.22	0.5392	78.91	0.3744	78.37	0.005540
AutoFeature(4)	**80.23***	**0.5390***	**79.04***	**0.3737***	**78.53**$^\Diamond$	**0.005524**$^\Diamond$

iPinYou[5]: was released as a part of the RTB Bidding Algorithm Competition, 2013. We use data from seasons 2 and 3 and to remove data leakage we follow [22] and remove "user tags". The statistics of the datasets are given it Table 3, where the dimensions refers to the dimension of the one-hot input vector.

5.1.2 Metrics. Following previous works [12, 22] we used the standard offline evaluation metrics of AUC (Area Under ROC) and log loss (cross-entropy) which are known to be good surrogates for measuring the actual CTR. For measuring the computation, we use flops (floating points operations) since they count the number of operations performed and are independent of the device used.

5.1.3 Baselines. The standard and state-of-the-art methods that we used as our baselines are: LR [14], FM [25], FFM [13], FNN [31], AFM [29], IPNN [21, 22], KPNN [22], PIN [22], GBDT [7], CCPM [19], DeepFM [12] and xDeepFM [15].

5.1.4 Parameter settings. Table 2 lists the parameters used in all models. The parameters are set to be the same as in [22] where possible. We tune the hyper-parameters of xDeepFM and AutoFeature on the validation set carved out from the training set. For AutoFeature we set $l = 3$ for second-order and 1 for higher-orders and $K = min(\binom{f}{2}, 275)$ for all our experiments. The Adam optimizer is adopted as in [22]. We have used the Xavier initialisation [10] which initialises the weights in DNN such that their values are subjected to a uniform distribution between $[-\sqrt{6/(n_{in} + n_{out})}, \sqrt{6/(n_{in} + n_{out})}]$ with n_{in} and n_{out} being the input and output sizes of a hidden layer. Such initialization can stabilize activations and gradients of a hidden layer at the early stage of training [22]. To solve the internal covariate shift issue in DNN, Layer normalization [2] has been applied to normalize activations in fully connected layers.

5.1.5 Significance Test. We calculate the p-values for our most promising model (AutoFeature(4)[6]) and the best baseline (PIN) by repeating the experiments 10 times and performing a two-tailed pairwise t-test.

5.2 Overall Performance

The overall performance of AutoFeature and the baselines on the three datasets is reported in Table 4. We first note that amongst the baselines PIN performs the best across all three datasets. PIN is an improved variant of the general IPNN framework. It replaces the product operation of IPNN with a sub-network, however, it models all pairs of features and uses the same sub-network for these pairs. We see that AutoFeature(2), which also models the second-order feature interactions outperforms IPNN significantly on all three datasets. It also performs better than PIN on all three datasets. This is despite the fact that AutoFeature(2) models a subset of all second-order feature interactions and hence has much fewer parameters and flops (further experiments are provided in Section 5.3 to illustrate this point). This gain in performance may be attributed to better modelling of the feature interactions by not using the same structure for each feature pair and avoiding noisy feature interactions.

Nevertheless, another important benefit of using AutoFeature is that we can search for which feature interactions to keep. Therefore, unlike PIN we are not restricted to the second-order. The benefit of modelling these higher-order interactions is evident from the performance of AutoFeature(3) and AutoFeature(4) which further improves over the state-of-the-art performance. This illustrates the benefit of modelling higher-order interactions selectively.

Like AutoFeature, xDeepFM also models higher-order explicit feature interactions. We can see from Table 4 that AutoFeature provides far superior performance. Despite modelling higher-order features the lower performance of xDeepFM may be attributed to

[5] http://data.computational-advertising.org

[6] AutoFeature(p) denotes that feature interactions up to p^{th} order are modeled.

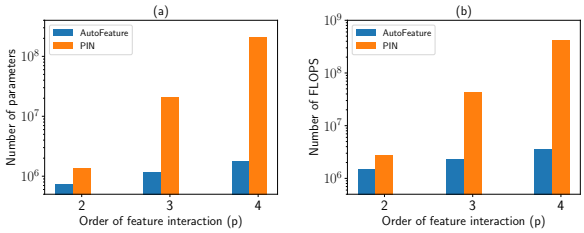

Figure 4: The number of parameters and flops for PIN and AutoFeature in the product layer for different orders of feature interactions on Avazu. The values for AutoFeature correspond to the best model found. The y-axis is on a log scale.

modelling all feature interactions including the noisy ones. Finally, we can see that the relative improvement of AutoFeature(4) over PIN is comparable or more than the improvement of PIN over the second-best baseline PNN. The same is true for the relative improvement of xDeepFM over DeepFM.

5.3 Performance of AutoFeature in terms of the number of parameters and flops

While comparing the overall performance we commented that AutoFeature(2) was able to outperform PIN on all datasets despite modelling only a subset of the feature interactions. Here we show the comparison between AutoFeature and the best baseline PIN in terms of the number of parameters and flops. Since AutoFeature shares the embedding layer and DNN classifier with PIN model, we measure the complexity of AutoFeature and PIN by the number of parameters and flops in the product layer.

Figures 4 (a) and (b) show the comparison (in log scale) between AutoFeature and PIN in terms of the number of parameters and flops respectively. This comparison is performed on the Avazu dataset and a similar trend is observed on the other two datasets. The values reported for AutoFeature are collected from the best performing architectures in Table 4. We see that the number of parameters and flops used by AutoFeature(3) are even less than those of PIN (p = 2), while AutoFeature(3) considerably outperforms PIN in terms of the overall performance as observed in Table 4.

We also see that there is a gradual increase in the number of parameters and flops for AutoFeature as we increase the orders of feature interactions. However, for PIN the growth of parameters and flops is exponential in terms of the order p as it would model all feature pairs[7]. This is the reason PIN cannot scale up to higher-orders and is restricted to the second-order interactions [22].

Table 1 shows the comparison for all three datasets in terms of the percentage of parameters and flops of AutoFeature compared to the corresponding PIN model e.g., AutoFeature(4) will be compared with PIN(4). We see that to achieve similar or better performance AutoFeature(2) requires only around 29%, 53% and 36% of parameters and flops that PIN requires on the Criteo, Avazu and iPinYou datasets respectively. In addition, the number of parameters/flops of AutoFeature are only a small fraction of what PIN would have required for higher-order interactions. This makes the modelling of higher-order practical in AutoFeature.

[7]The parameter and flops for PIN with p>2 are reported to illustrate the comparison with AutoFeature. The actual PIN model for these orders in impractical to run.

Table 5: The effect of removing different orders of feature interaction on the performance of AutoFeature.

Omitted Orders	Criteo		Avazu		iPinYou	
	AUC	log loss	AUC	log loss	AUC	log loss
None	80.23	0.5390	79.04	0.3737	78.53	0.005524
1	80.21	0.5393	78.98	0.3741	78.24	0.005542
2	80.17	0.5398	78.89	0.3745	78.31	0.005543
3	80.21	0.5394	78.91	0.3744	78.28	0.005543
4	80.22	0.5392	78.91	0.3744	78.37	0.005540

5.4 Breakdown of operations

To see where the parameters of AutoFeature come from, we explore the structure of AutoFeature(2) for the Avazu dataset. Figure 5 (a) shows the percentage of sub-networks with different number of operations. We see that around 20% of sub-networks have no operations meaning that these feature interactions are not modelled as they are deemed to be unnecessary by AutoFeature. Another 20% sub-networks have one operation and only around 35% of sub-networks have three operations.

However, not all operations have parameters and some operations model simple interactions like $\oplus$ and $\otimes$. Figure 5 (b) shows the occurrence percentage of each operation in the overall product layer architecture string for AutoFeature(2). We see that almost 40% of the operations are $\emptyset$ meaning that the operation doesn't exist. These operations are used to decide the number of operations in a sub-network as explained in Section 4.1 and don't contribute to the flops and parameters. The remaining 60% are the actual operations. From these, around 45% operations are $\otimes$ and $\oplus$. This means that of the sub-networks that have at least one operation, almost 45% are simple interactions and contribute no parameters and minimal flops (O(pn) flops, where n is the embedding size). This results in a decrease in the overall complexity. The remaining 55% of the operations are concat and $\boxtimes$ which represent more complex interactions and contribute to the majority of the flops/parameters.

5.5 Ablation Studies

Table 5 shows the effect of removing the individual order of feature interactions on the three datasets. We see that no order of interactions is redundant and the performance drops by removing any order. However, the impact of different orders is data-dependent and no clear pattern is discernible. For Criteo and Avazu, removing second-order interactions is the most detrimental and for iPinYou removing first order leads to the most decrease in performance.

5.6 Comparison with NAS algorithms

Table 6 shows the performance of existing NAS algorithms. We can see that NBTree takes much lower number of samples to achieve the best accuracy.

Recent variants of Bayesian optimization (BO) [26] have a predictive network but they work well for small structures with around 10 nodes. Our product layer will have around a thousand nodes leading to a large sparse graph. We found that BO with predictive networks (BO-GCN)[8] was not able to predict the sample performance. Reinforcement based searching is known to be slow even for CV tasks and more so for CTR-prediction given the large number of possible architectures (due to many sub-networks per architecture).

[8]Vanilla BO and BO with other predictive networks cannot scale to our search space.

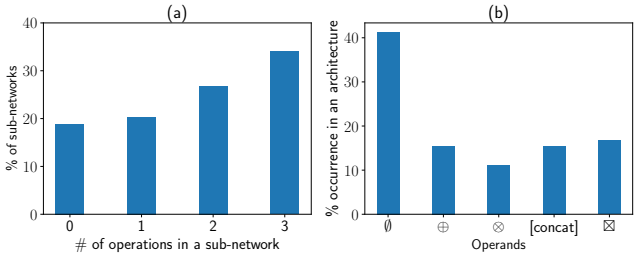

Figure 5: For the best 2nd order architecture (a) shows the percentage occurrence of each operation, (b) shows the % of sub-networks with the different number of operations.

Table 6: Average number of samples to get the best model

	NBTree	Random	Evolutionary	BO-GCN
#Samples	336	1441	1300	1400

5.7 Analysis of the search algorithm

We now demonstrate the internal working of our AutoFeature algorithm. Figure 6 (a) shows the plot of the performance density of the searched architectures as the search progresses on the Avazu dataset. We see that during the first 50 samples there is a large variance in the AUC of the searched architectures. This shows that during the initial phase the AutoFeature algorithm is searching the different subspaces to find better architectures. However, as the search progresses we see the density shifts to the right and around 300-350 samples the search has almost converged to finding the best sample. We note that due to the probabilistic nature of sampling from the CRP, even when the search is almost converged our search algorithm will sample a few samples from other lower-quality leaves to effectively explore the search space. This explore-exploit trade-off is the reason for the variance in the density around 300-350 samples. Moreover, this variance is considerably less than it was during the first 50 samples.

We also plot the cumulative number of samples obtained from each leave as the search progresses in Figure 6 (b). At each step of the search, we take eight samples according to the CRP process. We see that during the initial phases of the search we take comparable number of samples from each leaf node causing high variance in the AUC seen before. As we sample more, the NBTree and hence the search spaces represented by the leaves get more refined and the best architectures are gradually moved to the left of the tree. As a result, the architectures in the left leaves ('ll' and 'lr') are of better quality and are sampled more according to Equation 3. By the end of the 50^{th} step (i.e., 400^{th} sample), we have sampled the most from the leftmost node and least from the rightmost node.

Finally, the quality of architectures produced by the different leaf nodes is shown in Figure 6(c). We can see that the leftmost leaf ('ll') produces the best architectures and has the minimum variance. The accuracy decreases as we move to leaves that are to the right.

5.8 Parameter Study

We investigate the effect of T on the number of steps taken by AutoFeature to converge. Figure 7(a) shows the boxplots for the number of steps required to reach the best architecture when the

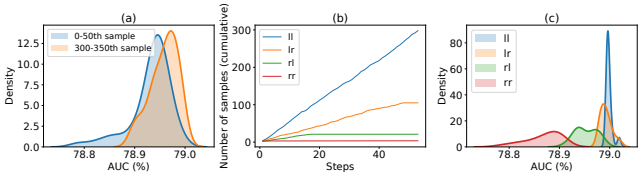

Figure 6: (a) The density of the sample accuracy (AUC) shifts as search proceeds. Each curve shows the density for 50 samples taken from different stages during searching on the Avazu dataset. (b) The cumulative number of samples drawn from each leaf. More samples were drawn from the leftmost leaf ('ll') since it leads to the most accurate architectures. (c) The accuracy of samples from different leaf nodes.

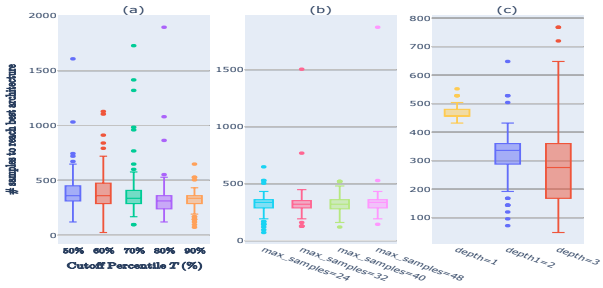

Figure 7: Evaluation of the sample efficiency of AutoFeature on Avazu for different values of T, num_samples and depth d. The experiments were repeated one hundred times.

search algorithm was run 100 times on the Avazu dataset. We can see that when T is set to the 90^{th} percentile we get the best result with the smallest interquartile range. Having a higher value of T encourages the better leaves to have fewer but higher quality architectures.

Figure 7(b) shows the effect of max_samples on the number of steps taken by our AutoFeature algorithm to converge when the search algorithm was run 100 times on the Avazu dataset. A smaller value of max_steps appears to be slightly better as the search converges quicker in the best case. However, there was no marked difference in the interquartile range. Figure 7(c) shows the effect of depth. The more the depth the finer the subspace partitions and faster the convergence. However, greater depth requires more samples to learn the NBTree so that the classifiers at lower levels can be trained properly. For our experiments, d=2 provided the right balance between search complexity and convergence.

5.9 Qualitative Results

In Figure 8 we show some feature interactions learned from the Avazu dataset. In (a) we see that the features BANNER_POS and DEVICE_CONN_TYPE have no feature interactions. We would expect this intuitively as the position of the Ad (advertisement) on the mobile screen is independent of the device connection type. In (b) we see that a simple interaction is learned for APP_ID and APP_CATEGORY, since their relationship is quite straightforward. Finally, in (c) we see that the BANNER_POS and APP_ID have quite a complex relationship and this may be because the layout of different apps dictates

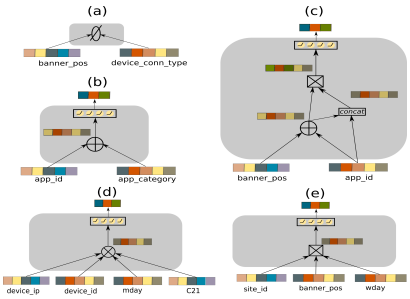

Figure 8: Sub-networks learned from Avazu. AutoFeature finds meaningful and discards useless feature interactions.

the suitable position for the Ad and since there are many Ads and apps this relationship needs more parameters to model. With AutoFeature we can determine which feature interactions to model and the operations to use to model these interactions.

6 CONCLUSION

In this paper, we presented AutoFeature which is a neural architecture search-based method for CTR prediction. AutoFeature can search and identify which feature interactions to model and how to model these interactions. This ability of AutoFeature to identify useful feature interactions makes it capable of modelling higher-order feature interactions efficiently which increases the performance of the recommender. As a part of AutoFeature, we proposed a novel search space suitable for CTR prediction models and a search algorithm capable of effectively searching over this space by recursively partitioning it into promising and less promising regions. Through extensive experiments, we showed that AutoFeature outperforms the state-of-the-art baselines, and achieves this using fewer flops and parameters. In addition, it learns meaningful feature interactions.

REFERENCES

[1] David J Aldous. 1985. Exchangeability and related topics. In *École d'Été de Probabilités de Saint-Flour XIII—1983*. Springer, 1–198.

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

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

[4] Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. 2018. Efficient architecture search by network transformation. In *AAAI*.

[5] Han Cai, Ligeng Zhu, and Song Han. 2019. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware, In ICLR. *arXiv preprint arXiv:1812.00332*.

[6] Liang-Chieh Chen, Maxwell Collins, Yukun Zhu, George Papandreou, Barret Zoph, Florian Schroff, Hartwig Adam, and Jon Shlens. 2018. Searching for efficient multi-scale architectures for dense image prediction. In *NIPS*.

[7] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM, 785–794.

[8] Yukang Chen, Tong Yang, Xiangyu Zhang, Gaofeng Meng, Chunhong Pan, and Jian Sun. 2019. Detnas: Neural architecture search on object detection. *arXiv preprint arXiv:1903.10979* (2019).

[9] Hengtze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Deepak Chandra, Hrishi Aradhye, Glen Anderson, Greg S Corrado, Wei Chai, Mustafa Ispir, et al. 2016. Wide & Deep Learning for Recommender Systems. *conference on recommender systems* (2016), 7–10.

[10] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 249–256.

[11] Alois Gruson, Praveen Chandar, Christophe Charbuillet, James McInerney, Samantha Hansen, Damien Tardieu, and Ben Carterette. 2019. Offline Evaluation to Make Decisions About PlaylistRecommendation Algorithms. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*. ACM, 420–428.

[12] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*. 1725–1731. https://doi.org/10.24963/ijcai.2017/239

[13] Yuchin Juan, Yong Zhuang, Wei Sheng Chin, and Chih Jen Lin. 2016. Field-aware Factorization Machines for CTR Prediction. In *ACM Conference on Recommender Systems*. 43–50.

[14] Kuang-chih Lee, Burkay Orten, Ali Dasdan, and Wentong Li. 2012. Estimating conversion rate in display advertising from past erformance data. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 768–776.

[15] Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. 2018. xDeepFM: Combining Explicit and Implicit Feature Interactions for Recommender Systems. *arXiv preprint arXiv:1803.05170* (2018).

[16] Chenxi Liu, Liang-Chieh Chen, Florian Schroff, Hartwig Adam, Wei Hua, Alan Yuille, and Li Fei-Fei. 2019. Auto-DeepLab: Hierarchical Neural Architecture Search for Semantic Image Segmentation. *arXiv preprint arXiv:1901.02985* (2019).

[17] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. 2017. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436* (2017).

[18] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. Darts: Differentiable architecture search. In *ICLR*.

[19] Qiang Liu, Feng Yu, Shu Wu, and Liang Wang. 2015. A convolutional click prediction model. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. ACM, 1743–1746.

[20] H. Brendan McMahan, Gary Holt, David Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg, Arnar Mar Hrafnkelsson, Tom Boulos, and Jeremy Kubica. 2013. Ad click prediction: a view from the trenches. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*. 1222–1230. https://doi.org/10.1145/2487575.2488200

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

[22] Yanru Qu, Bohui Fang, Weinan Zhang, Ruiming Tang, Minzhe Niu, Huifeng Guo, Yong Yu, and Xiuqiang He. 2018. Product-based neural networks for user response prediction over multi-field categorical data. *ACM Transactions on Information Systems (TOIS)* 37, 1 (2018), 5.

[23] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2018. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548* (2018).

[24] 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*.

[25] Steffen Rendle. 2010. Factorization machines. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*. IEEE, 995–1000.

[26] Han Shi, Renjie Pi, Hang Xu, Zhenguo Li, James T. Kwok, and Tong Zhang. 2020. Multi-objective Neural Architecture Search via Predictive Network Performance Optimization. https://openreview.net/forum?id=rJgffkSFPS

[27] Linnan Wang, Yiyang Zhao, Yuu Jinnai, and Rodrigo Fonseca. 2018. Alphax: exploring neural architectures with deep neural networks and monte carlo tree search. *arXiv preprint arXiv:1805.07440* (2018).

[28] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & cross network for ad click predictions. In *Proceedings of the ADKDD'17*. ACM, 12.

[29] Jun Xiao, Hao Ye, Xiangnan He, Hanwang Zhang, Fei Wu, and Tat-Seng Chua. 2017. Attentional factorization machines: Learning the weight of feature interactions via attention networks. *arXiv preprint arXiv:1708.04617* (2017).

[30] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. 2019. SNAS: stochastic neural architecture search. In *ICLR*.

[31] Weinan Zhang, Tianming Du, and Jun Wang. 2016. Deep learning over multi-field categorical data. In *European conference on information retrieval*. Springer, 45–57.

[32] Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. 2018. Practical block-wise neural network architecture generation. In *Proceedings of the CVPR*.

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

[34] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *CVPR*.