Feature Augmentation with Reinforcement Learning

Jiabin Liu[1], Chengliang Chai[1*], Yuyu Luo[1], Yin Lou[2], Jianhua Feng[1], Nan Tang[3]
[1]*Department of Computer Science, Tsinghua University,* [2]*Ant Group,* [3]*QCRI*
{liujb19@mails.,ccl@,luoyy18@mails., fengjh@}tsinghua.edu.cn, yin.lou@antgroup.com, ntang@hbku.edu.qa

***Abstract*—Sufficient good features are indispensable to train well-performed machine learning models. However, it is common that good features are not always enough, where feature augmentation is necessary to enrich high-quality features by joining with other tables. There are two main challenges for the problem. Given a set of tables where we can augment features from, the first challenge is that there are a lot of ways of joining multiple tables and deciding which features (or attributes) to use – selecting the best set of features to augment is hard. Moreover, we may need to materialize the join results for different join options, doing full materialization might be time consuming – efficient but approximate methods are needed.**

In this paper, we first introduce the design space of the feature augmentation problem. Then, to address the above challenges, we propose a reinforcement learning based framework, namely `AutoFeature`, to augment the features following an *exploration-exploitation* strategy. `AutoFeature` keeps *exploring* the features in tables that have led to performance improvement. At the same time, `AutoFeature` also *exploits* the tables (features) that are rarely selected. In this way, the search space of tables (features) to be augmented can be well explored and a subset of good features can be selected. `AutoFeature` utilizes sampling techniques to achieve high efficiency. We implement two algorithms, one with multi-arm bandit and the other with branch Deep Q Networks (branch DQN), to realize the framework of `AutoFeature`. We conducted experiments on three real-world datasets `School`/`XuetangE`/`Air` using 16/23/34 candidate tables with 695/204/338 candidate features. Extensive results show that `AutoFeature` outperforms other methods by 12.4% and 9.8% on `AUC` values on two classification datasets (`School` and `XuetangE`) and by 0.113 on the `MSE` value on `Air` in terms of the model performance.

***Index Terms*—Feature Augmentation, Machine Learning**

I. Introduction

A machine learning (ML) model is only as good as the data (and features) that it is trained on. However, it is not uncommon that the training data (even from experienced data scientists) may not always contain sufficient good features. For example, assume that a data scientist from the payment team in Ant Group wants to predict whether a user is a scammer or not, although she might be able to quickly collect useful information (features) from her domain (such as basic user profile, payment transaction history, etc.), there is plenty valuable information outside of her domain (possibly even out of her awareness) that is also useful (such as credit data or LBS data).

In this case, the initial training data that she prepared is *sub-optimal*, and it is desirable to augment the initial training dataset with useful features from other domain. We call such a process *feature augmentation*.

Feature augmentation includes two steps: deciding which tables can be used to provide more features through "Join" operations, and selecting which features should be used.

The first step can be done by either domain experts who specify which tables to join and how to join them, or piggy-backing existing data discovery tools [1], [17], [18] to find joinable tables. Note that, in either case, human-in-the-loop is typically inevitable.

Therefore, the main focus of this paper is on step 2, *i.e.,* given an initial training set and a set of tables that may contain useful features, decide which features (from those tables) could be selected to augment the initial training data.

Challenges. There are two main challenges. (1) *Effectiveness.* How to select an optimal subset of features for augmentation? (2) *Efficiency.* Join operations could be time-consuming, especially when we might need to join many tables in different orders. Hence, how to make the overall feature augmentation process efficient is important.

Existing work and their limitations Existing research has attempted to approach this problem from two perspectives. (I) *Join-or-not [27].* The basic idea is that they avoid joining tables that do not significantly affect the prediction accuracy, based on the Vapnik-Chervonenkis (VC) dimension of downstream ML algorithms, when the foreign key has already included all the information of the tables to be joined. (II) *Finding top-k relevant tables [14].* ARDA computes a score for each candidate table, selects top-k tables, and joins them with the base table for feature augmentation. However, the score of a candidate table is simply based on its relevance to the base table [1], [17], rather than the model performance.

Their limitations. (I) solved the problem of given features that can be used for augmentation, whether these features should be used or not, which does not solve how to discovery these features in the first place. (II) selected tables and features based on relevance, without measuring their practical impact on the particular ML task. Consequently, it is hard for this heuristic approach to obtain the optimal feature augmentation in terms of the model performance.

Design space. In order to find the optimal solution for feature augmentation, we have to carefully explore the large search space that consists of the candidate tables (as well as the contained features), with the consideration of the model performance in mind. There are several straightforward solutions to explore the space.

* Chengliang Chai is the corresponding author.

(i) *Forward selection* is an iterative method which uses a greedy strategy to select new features. In each iteration, it evaluates each feature by joining the corresponding table and selects the feature which best improves the model performance.

(ii) *Backward selection* is also a greedy method. It joins all candidate tables and starts with all features in the dataset. It then iteratively removes the feature with least impact on predictive performance at each iteration.

Although the above methods (i and ii) can explore the search space to some extent, it is difficult for them to discover good tables (features) combination due to their simple and heuristic search strategy. The reasons are two folds. On the one hand, they just add (or remove) one feature at a time, and thus it is unable to sufficiently explore the search space of features. Therefore, they are not accurate enough to evaluate the importance of each feature. Thus, some potentially beneficial features cannot be augmented. On the other hand, they use greedy augmentation strategies and do not consider the relationship between features. Features from different tables will influence each other. Thus, they fail to discovery useful feature combinations. In this work, we employ a trial-and-error strategy that keeps refining the search criteria, in the pursuit of the optimal solution.

Our Methodology. To judiciously explore the table/feature space, we propose a reinforcement learning based feature augmentation framework `AutoFeature`, which adds features to augment features using an *exploration-exploitation* strategy. At a high level, `AutoFeature` iteratively augments tables as well as features in a model-aware, trial-and-error approach, to progressively understand which table (feature) or table (feature) combination is good. To be specific, based on the model feedback during iterations, we should *exploit* the features in tables that have brought much performance improvement, so as to further maximize the model performance. However, we also have to *explore* the rarely selected tables (features) to avoid local optimum.

Contributions. We make the following contributions.

(1) We explore the design choices of feature augmentation, including the Forward, Backward and RL-based approaches.

(2) We propose a multi-armed bandit (MAB) based RL solution, which is a simple yet effective approach that uses the standard MAB method to tackle the exploration-exploitation dilemma, where each table can be regarded as an arm and pulling the arm (*i.e.,* an action) means to join the table for feature augmentation.

(3) We propose a branch deep Q network (DQN) based solution following the similar idea as MAB, which is more effective and leverages the powerful neural network to encode more features to learn the feature augmentation policy.

(4) The experimental results show that RL-based method outperforms existing solutions by 12.4% (`AUC`), 9.8% (`AUC`) and 0.113 (`MSE`) on three datasets in terms of the model performance.

II. Problem Definition and Solution Overview

A. Problem Definition

Machine learning task. Given a set of data $\{(x_i, y_i)\}_{i=1}^{N}$, a machine learning task M is to learn a function $f : \mathbb{X}^d \to \mathbb{Y}$, where each $x_i \in \mathbb{X}$ is a d-dimension feature vector and $y_i \in \mathbb{Y}$ is the label of the data instance x_i. For example, $\mathbb{Y} = \{0, 1\}$ indicates that M is a binary classification task, and $\mathbb{Y} \in \mathcal{R}$ suggests that M is a regression task.

In our problem setup, we assume an initial dataset (stored in a table) is provided by the user. We are also given a repository of tables that may contain useful information and can be joined (either directly or indirectly) with the initial training data. We want to augment the initial training data with more useful features from other tables (through join operations) to enhance the model performance.

The initial dataset can be further divided into training, validation (T_{val}) and test dataset (T_{test}). The T_{val} set is used to develop the model to tune the hyperparameters, and the T_{test} set is to provide an unbiased evaluation of the final ML model on the training dataset.

Base tables. The tables that store the initial training, validation, and test data is called the base training, validation, and test table, respectively. In the rest of this paper, we use base table to indicate base training table, $T_b = \{b_1, b_2, ..., b_d, L\}$, where each $b_i (i \in [1, d])$ represents an attribute and L denotes the label column.

Candidate tables. Candidate tables, $\mathcal{T} = \{T_1, T_2, ..., T_m\}$, are m joinable tables that can be used to augment base table. Each table is denoted by $T_k = \{f_{k1}, f_{k2}, ...\}, k \in [1, m]$, where f_{kj} denotes the j-th attribute in table T_k.

At a high level, these tables can be represented as a join graph, where each node corresponds to a table, and each edge denotes that two tables can be joined through two attributes respectively in the tables.

Example 1: As shown in Fig. 1(a), we have the base table $T_b =$ `new_enroll`, which contains four features (*i.e.,* $d = 4$) and the label column $L =$ `enroll_or_not`. We also have four candidate tables $\mathcal{T} =$ {`Stu_info`, `Course_info`, `Enroll_info`, `Score`}. T_b can join with `stu_info`, `enroll_info`, `score` via the key `student_id`, which is the typically PK-FK join.

Note that in addition to the edges between T_b and tables in $\mathcal{T}$, there can exist edges in every two candidate tables, which indicates that our framework supports the transitive join.

Join base table and candidate tables. For flexibility, we allow the user to specify the join operators between the base table T_b and candidate tables T_i from $\mathcal{T}$, *e.g.,* which tables can be joined, along with the operators (*e.g.,* natural join, fuzzy join, left join, and so on) required for each join operation.

Next we formally define our problem.

The feature augmentation problem. Given T_b, a set of candidate tables $\mathcal{T}$ and a machine learning task M, the feature augmentation problem is to select a subset of tables $\mathcal{T}^* \subset \mathcal{T}$ to

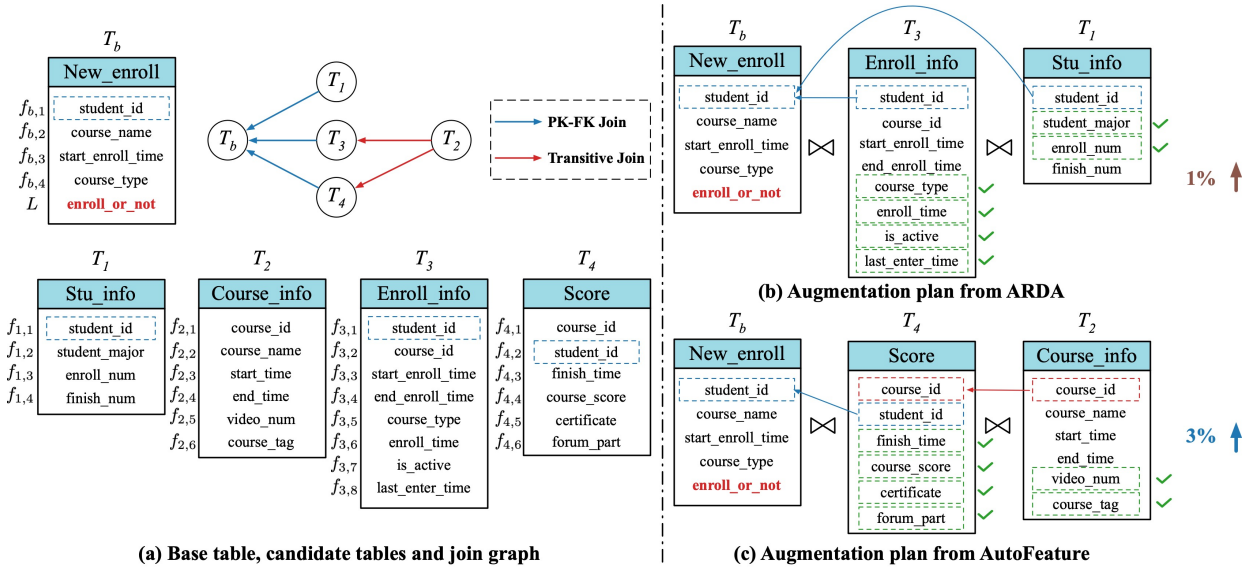

Fig. 1. An example of Feature Augmentation

join with T_b, and $\forall T \in \mathcal{T}^*$, select a set of features (attributes) $T[F]$ to augment T_b (T_{val} and T_{test} are augmented in the same way), so as to improve the predictive performance of model under M, if trained on the augmented T_b.

More concretely, the selected $\mathcal{T}^*$ and the corresponding features in tables of $\mathcal{T}^*$ can be viewed as an optimal **feature augmentation plan**. This indicates that given a table with the same schema as T_b, in order to achieve good performance, we should augment features following the plan, *i.e.,* joining with tables in $\mathcal{T}^*$ and augment the features in $T[F], T \in \mathcal{T}$.

Example 2: As shown in Fig.1, given T_b, $\mathcal{T}$ and a machine learning task M, the problem is to select the optimal subset $\mathcal{T}^* = \{\texttt{Score}, \texttt{Course\_info}\}$, as well as the features $\texttt{Score}[F] = \{\texttt{finish\_time}, \texttt{course\_score}, \texttt{certificate}, \texttt{forum\_part}\}$ and $\texttt{video\_num}, \texttt{Course\_info}[F] = \{\texttt{course\_tag}\}$, which serves as the feature augmentation plan. In this way, we can obtain the best model performance improvement if we augment these features.

As we can see, the search space of the problem is the exponential w.r.t. the number of tables as well as features. Even worse, join operation between tables is not efficient, so it is non-trivial to judiciously explore the search space.

Our goals. We have two goals. (1) Effectiveness. We want to improve the performance of the predictive model after feature augmentation. (2) Efficiency. The feature augmentation plan should be executed within reasonable time (*e.g.,* hours instead of weeks).

Also, we treat the machine learning model (linear model, deep neural network, etc.) and corresponding the task M as a black box such that the user can leverage our framework to augment features regardless of any particular model, and the model parameters can be updated iteratively.

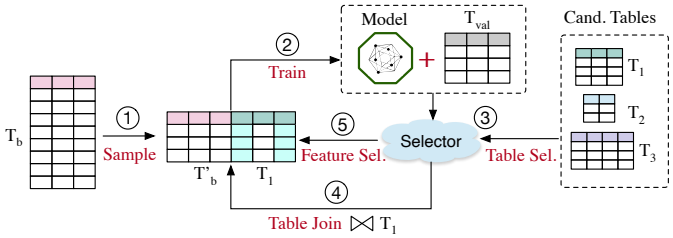

Fig. 2. Overall Framework of `AutoFeature`

B. Solution Overview

Next, we will present the framework of `AutoFeature` for approaching the problem of feature augmentation.

An iterative framework. As shown in Fig. 2, in Step ①, we first sample some data instances to improve the efficiency. Then, we conduct an iterative framework on the sampled result to select the feature augmentation plan (*i.e.,* selecting tables and features) following an *exploration-exploitation* strategy.

Initially, we feed the sampled data into the model for training, and use the validation set T_{val} to test the performance (Step ②). Then, based on the feedback, *i.e.,* the change of model performance, we select a table among candidate tables to join (Step ③ and ④), so as to choose features for augmentation (Step ⑤). Next, the base table together with the augmented features will be fed into the model, and another feature augmentation iteration begins.

The above process will terminate when reaching the maximum iteration times (*e.g.,* 20 iterations).

Overall, the core part of `AutoFeature` is the *Selector*, which adopts the exploration-exploitation strategy to select tables and features. Basically, based on existing knowledge, if some features of a table can significantly improve the model performance, we are likely to keep selecting features in the table, *i.e.,* exploitation. However, we will also explore more rarely selected tables (features) to acquire new knowledge, in case of leading to a local optimum.

Obviously, using the above iterative process, we can achieve the goal of effectiveness. However, when using full base table T_b in the feature augmentation process, it is hard to achieve the goal of efficiency, since the join and training operations are both time-consuming. Thus, we use the sampling strategy to reduce the size of the base table T_b.

The sampling strategy. As we know, the join operation is always time-consuming, especially in our case that there exist a number of candidate tables. Hence, we propose to sample from the base table T_b, producing T_b', and use T_b' to join candidate tables, which greatly improves the efficiency. Specifically, we use the stratified sampling method [33] to get T_b' that can well represent T_b, such that the the optimal augmented features for T_b' are equivalent to those for T_b. Since the base table is only sampled once, and afterwards, we just use T_b' to discover the features, we still use T_b to denote T_b' for ease of representation.

In the subsequent sections, we will introduce two approaches to implement the *Selector*, *i.e.,* Multi-armed Bandit based (Section III) and Branch DQN based (Section IV) solution. The former one is a subtle yet effective method that follows the exploration-exploitation insight. The latter one is more powerful because deep neural network is utilized to consider more features of data, and has strong learning ability.

III. Multi-armed Bandit for AutoFeature

In this section, we first introduce the motivation of using MAB to solve the feature augmentation problem and then the detailed implementation.

A. Motivation of MAB

Multi-armed bandit (MAB) [40] aims to model an *agent* that simultaneously attempts to acquire new knowledge (*i.e., exploration*) and optimizes their decisions based on existing knowledge (*i.e., exploitation*) from the *environment*. Given multiple arms, MAB allows the agent to pull an arm in each iteration, *i.e., an action*, using an exploration-exploitation strategy [42]. After the action is executed, the environment produces feedback, which is utilized to guide the action selection of subsequent iterations, so as to maximize the long-term benefits.

Relation between MAB and `AutoFeature`. At a high level, our selective feature augmentation problem naturally fits the MAB framework. Specifically, we have m candidate tables, corresponding to m arms. In each iteration, pulling an arm means that a table is selected and some features of it will be used for augmentation. Afterwards, a reward/penalty has to be computed based on the performance improvement or degradation after the features are augmented. Then the result is utilized to guide the augmentation strategy in next iterations, where the essential idea is to handle the exploration-exploitation trade-off between selecting features in the table that currently leads to high rewards and the tables that are rarely selected.

B. `AutoFeature` with MAB

In this subsection, we first introduce what is the action like in the MAB solution in detail, followed by how to compute the reward after the action is executed, and how to select the action based on the reward. The above steps are repeated iteratively, which is the overall MAB algorithm introduced at the end of this subsection.

The action: pulling an arm. At the k-th iteration, an action a_k denotes that a candidate table as well as a feature subset are selected for feature augmentation, *i.e.,* $a_k = T_i[F]$, where $T_i \in \mathcal{T}$ and F is the selected feature subset of T_i in this iteration. $T_i[*]$ denotes that we augment all features of T_i. Apparently, if T_i is the first time to be selected, in order to evaluate the features in it, we have to conduct the join operation. Otherwise, the join results should be previously stored, and thus we can just select more features from T_i.

We first introduce some notations. We use $\mathbf{T}_k$ to denote the table after the action a_k is executed, and $\mathbf{T}_k$ will be served as the training data fed into the model. More concretely, we represent $\mathbf{T}_k = \bowtie_{i=0}^{k} a_k$, where $\mathbf{T}_0 = a_0 = T_b[*]$.

Example 3: Initially, $\mathbf{T}_0$ = `New_enroll`[*]. Suppose that $a_1 = T_4$[`course_score, finish_time`], *i.e.,* selecting the table T_4 = `Score` to join and augment features $T_4[F] = \{$`course_score, finish_time`$\}$. Then $\mathbf{T}_1 = T_b[*] \bowtie T_4$[`course_score, finish_time`] is the training data at the first iteration. Then, suppose that a_2 = T_4[`certificate, finish_time`]. Since T_4 has been joined before, we just need to augment more features in T_4 without joining again, so $\mathbf{T}_2 = T_b[*] \bowtie T_4$[`course_score, finish_time`] $\bowtie$ T_4[`certificate, finish_time`] which can be directly rewritten as $\mathbf{T}_2 = T_b[*] \bowtie T_4$[`course_score, finish_time, certificate, finish_time`] for ease of representation. Next, suppose that $a_3 = T_2$[`course_tag, video_num`]. T_2 = `Course_info` can be selected because we support the transitive join, *i.e.,* `Course_info` can join with `Score`, under the situation where `Score` has been joined with T_b. Thus $\mathbf{T}_3 = T_b[*] \bowtie T_4$[`course_score, finish_time, certificate, finish_time`] $\bowtie T_2$[`course_tag, video_num`].

As discussed above, each action can be divided into two steps, *i.e.,* select a table and a feature subset of the table. For the first step, we consider the accumulated rewards of the candidate tables, which will be discussed next (the UCB-based solution [3]). Here we mainly discuss the second step, *i.e.,* how to select the features given a table. To be specific, once a table T is joined at the iteration k, we have $\mathbf{T}_{k-1} \bowtie T[F]$. Then we run XGBoost [13] to compute the significance of features in T, rank these features and select top-ℓ ones for augmentation. If at a subsequent iteration, table T is selected again, we select the next ℓ ranked features to augment without joining. Then $\mathbf{T}_{k-1}$ together with the augmented features are fed into the model for training and testing, and the returned performance will be utilized to update the reward of table T.

Example 4: Suppose that at the first iteration, we

select the table T_4. Then we join T_b with T_4, producing the table $\mathbf{T}_1$, which is fed into XGBoost [13] for feature significance computation. The ranked result is [course_score, finish_time, certificate, forum_part], and thus in this iteration, $a_1 = T_4$[course_score, finish_time] when $\ell = 2$. Following Example 3, T_4 is selected again at the second iteration, two subsequent features [certificate, forum_part] are selected based on the previous ranking.

Note that many ML algorithms can be used for feature significance ranking. XGBoost is a default choice because it is a widely-used algorithm in the industry, and it can well handle the missing values produced by the join operation.

Table reward (penalty) per iteration. Intuitively, we should give each table a score, once features of the table are augmented and change the model performance. To be specific, if the performance is improved, the score is regarded as a *reward*. If the performance is degraded, the score is regarded as a *penalty*.

To this end, we should first measure the performance difference between the two models if a feature subset of a table is augmented. More concretely, we denote the performance at the k-th iteration as $M(\mathbf{T}_k, T_{val})$, which is trained on $\mathbf{T}_k$ and evaluated on T_{val}. The performance difference when $T_i[F]$ is selected at the k-th iteration can be represented as $r_k^i = M(\mathbf{T}_k, T_{val}) - M(\mathbf{T}_{k-1}, T_{val})$, where $\mathbf{T}_k = \mathbf{T}_{k-1} \bowtie T_i(F)$. When $r_k^i > 0$, the table T_i will be assigned a reward. Otherwise, it will be assigned a penalty.

Example 5: Following Example 3, consider $\mathbf{T}_1 = T_b[*] \bowtie T_4$[course_score, finish_time], which means that a model will be trained on this table (*i.e.,* $\mathbf{T}_1$), and $M(\mathbf{T}_1, T_{val}) = 0.72$ denotes the performance of model testing on the validation dataset, *i.e.,* $T_{val} \bowtie T_4$[course_score, finish_time]. Suppose $M(\mathbf{T}_0, T_{val}) = 0.71$, and then $r_1^4 = M(\mathbf{T}_1, T_{val}) - M(\mathbf{T}_0, T_{val}) = 0.01$, which is a reward.

Algorithm 1: A UCB-based MAB Algorithm

Input: T_b, T_{val}, candidate tables $\mathcal{T}$, M, # iterations $\mathbf{k}$
Output: A feature augmentation plan $\mathbf{T_k}$.

1 $\mathbf{T}_0 = T_b$
2 **for each** $T_i \in \mathcal{T}$ **do**
3 Initialize $R_0^i = 0$, $n_0^i = 0$, $U_0^i = 0$;
4 **for** k *from* 1 *to* $\mathbf{k}$ **do**
5 Select the table with the largest U_k^i(*i.e.,* T_i);
6 **if** T_i *has not been selected* **then**
7 Join T_i with $\mathbf{T}_{k-1}$;
8 Rank features in T_i ;
9 $\mathbf{T}_k$ = Augment top-ℓ unselected features F_i^k in T_i;
10 $r_k^i = M(\mathbf{T}_k, T_{val}) - M(\mathbf{T}_{k-1}, T_{val})$;
11 Update R_k^i and U_k^i ;
12 **if** $r_k^i < 0$ **then**
13 Discard F_i^k from $\mathbf{T}_k$;
14 **return** $\mathbf{T_k}$;

Accumulated scores of tables. As discussed above, in each iteration, one of the candidate tables will be assigned with a reward/penalty. However, feature selection is an iterative process, thus a table is likely to have multiple score assignments. Therefore, an accumulated score has to be computed for each table. To be specific, we use $R_k^i = \frac{1}{n_k^i} \sum_{j=1}^{k} r_j^i$ to denote the accumulated score from the 1st to k-th iteration, where r_j^i is the score of table T_i at the i-th iteration, and n_k^i is the number of times that T_i is selected from iteration 1 to k.

Example 6: Suppose at the first iteration, we select $T_4 =$ Score, $a_1 = T_4$[course_score, finish_time] and $r_1^4 = 0.01$. Then, $a_2 = T_4$[certificate, finish_time] and $r_2^4 = 0.02$ tested on T_{val}, followed by $a_3 = T_2$[course_tag, video_num] and $r_3^2 = 0.01$. We have $R_3^4 = \frac{1}{2}(r_1^4 + r_2^4) = 0.15$ and $R_3^2 = r_3^2 = 0.01$.

Upper Confidence Bound (UCB) based solution. As discussed above, a table with a high accumulated score indicates that features in the table always improve the model performance. Thus a straightforward approach is to keep *exploiting* the table with the highest R_k^i at any iteration. However, this method will easily lead to the local optimum because it does not have the chance to *explore* the tables that are rarely visited but potentially helpful.

To address the above issue, we can leverage the popular UCB-based solution [3] to balance the *exploitation-exploration* trade-off. Considering the two factors, UCB computes another score for each table at the k-th iteration, as follows.

$$U_k^i = R_k^i + \gamma \sqrt{\frac{2k}{n_k^i + 1}} \tag{1}$$

where γ is a pre-defined parameter that controls the balance between exploitation and exploration. Intuitively, a table with a high accumulated score, *i.e.,* R_k^i, will have a high UCB score, leading to a higher probability to be selected, but meanwhile, a rarely selected table with a small n_k^i will also be likely to be chosen. At each iteration, we select the table with the maximum UCB score.

Algorithm. Algorithm 1 illustrates the overall process of the UCB-based solution. It first initializes the parameters (line 3). Then the algorithm iterates $\mathbf{k}$ times, which is specified by the user (line 4-11). In each iteration, it selects a table with the largest UCB score (line 5). If the table has not been joined, we join it with the existing table, *i.e.,* $\mathbf{T}_{k-1}$, and then rank features (by default using XGBoost) in T_i (line 7-8). Next, we augment top-ℓ unselected features to $\mathbf{T}_{k-1}$, producing $\mathbf{T}_k$. Then we test the performance difference after augmentation, and use the result to update the accumulated score and UCB score for further augmentation in subsequent iterations (line 10-11). After evaluating the performance on T_{val}, if the r_k^i is a penalty (*i.e.,* $r_k^i < 0$), we discard the selected feature set $\mathbf{T}_k$ from $\mathbf{T}_k$, thus $\mathbf{T}_k = \mathbf{T}_{k-1}$(line 12-13).

Example 7: We show a running example of Algorithm 1 in Fig. 3. We suppose that $\mathbf{k} = 4$, $l = 2$ and $\gamma = 0.02$.

(1) We first initialize $U_0^i = 0, R_0^i = 0, i \in [1, 4]$.

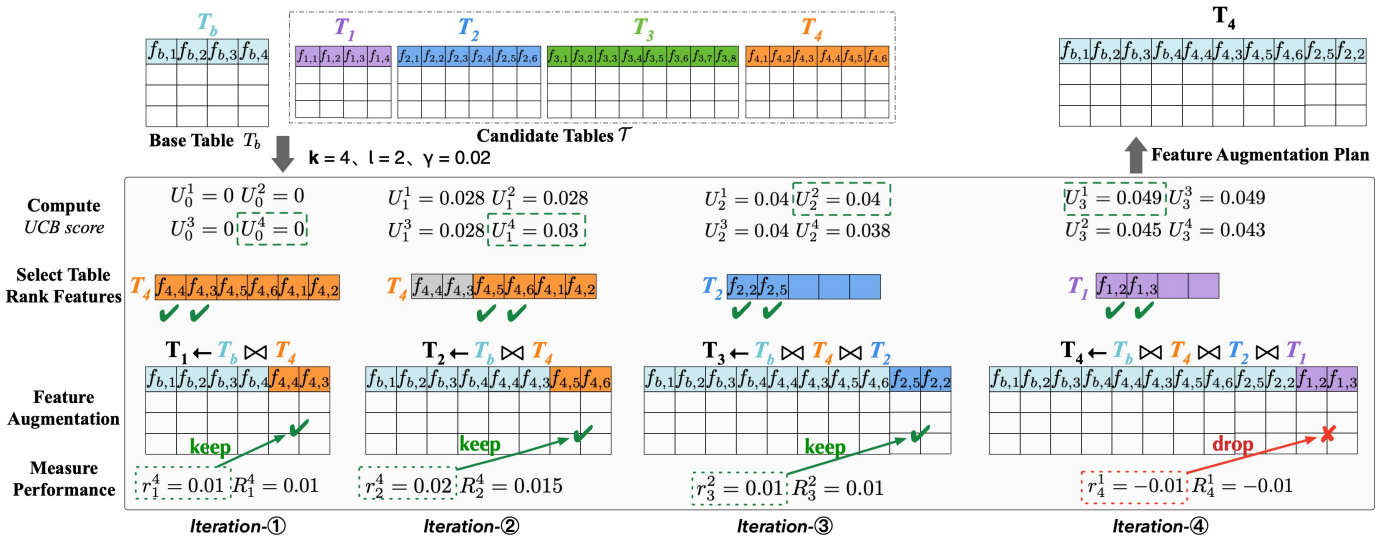

Fig. 3. A running example of AutoFeature

(2) In *Iteration*-①, since $U_0^i = 0, i \in [1, 4]$, we randomly select a table from $\mathcal{T}$. Suppose that T_4 is selected, it can directly join with T_b on key `student_id` and we rank the features. We augment the first two features from T_4 to get the new training data $\mathbf{T}_1$, which is used to retrain M. After testing on T_{val}, we get $r_1^4 = 0.01$. Then, $R_1^4 = 0.01$ and $U_1^4 = 0.01 + 0.02(\sqrt{(2*1)/2}) = 0.03$. Thus, $R_1^1 = R_1^2 = R_1^3 = 0$ and $U_1^1 = U_1^2 = U_1^3 = 0.028$. Since $r_1^4 > 0$ (*i.e.,* r_1^4 is reward), we keep these two new features in $\mathbf{T}_1$.

(3) In *Iteration*-②, since U_1^4 is the largest, we select T_4 again. As we ranked the features in the first iteration, we can directly select the first two available features and get $\mathbf{T}_2$. Next, we can obtain $r_2^4 = 0.02$ and $R_2^4 = \frac{1}{2}(0.01 + 0.02) = 0.015$. Using R_2^4, we compute $U_2^4 = 0.015 + 0.02(\sqrt{(2*2)/3}) = 0.038$. Meanwhile, $U_2^1 = U_2^2 = U_2^3 = 0.04$. We also keep the new features since r_2^4 is positive.

(4) In *Iteration*-③, since $U_2^1 = U_2^2 = U_2^3 = 0.04 > U_2^4 = 0.038$, we randomly select a new table from $\{T_1, T_2, T_3\}$, and suppose it is T_2. After ranking the features in T_2, we select the top-2 features, *i.e.,* $f_{2,2}$ and $f_{2,5}$, to augment and obtain $\mathbf{T}_3$. We find $r_3^2 = 0.01$ and $R_3^2 = 0.01$. Then, we can compute $U_3^2 = 0.01 + 0.02(\sqrt{(2*3)/2}) = 0.045$ and $U_3^4 = 0.015 + 0.02(\sqrt{(2*3)/3}) = 0.043$. Also, $U_3^1 = U_3^3 = 0.049$.

(5) In *Iteration*-④, since $U_3^1 = U_3^3 > U_3^2 > U_3^4$, we randomly pick a table from $\{T_1, T_3\}$. Suppose the candidate table T_1 is selected. Next, we can augment the first two as introduced. However, we find $r_4^1 = -0.01$ after testing on T_{val}. Since r_4^1 is negative (*i.e.,* r_4^1 is a penalty), we drop these two features. Then, we obtain the feature augmentation plan $\mathbf{T}_4 = T_b[*] \bowtie T_4[f_{4,4}, f_{4,3}, f_{4,5}, f_{4,6}] \bowtie T_2[f_{2,5}, f_{2,2}]$.

IV. Branch DQN for AutoFeature

The aforementioned MAB-based method leverages the UCB function with fixed parameters to augment the features, which cannot incorporate more data characteristics and has poor flexibility. Therefore, in this section, we propose a branch-DQN based RL to build a more powerful and flexible framework for feature augmentation.

A. Motivation of Using Branch DQN

RL and selective feature augmentation. RL is a typical ML paradigm where an agent conducts trial-and-error interactions with the environment, and in each interaction, the feedback of the environment is learned to find the optimal solution. For instance, in the famous AlphaGo [37] project, considering the current state (*i.e.,* the board situation), the agent predicts the next optimal action (*i.e.,* where is the next piece on the board) which can lead to the largest probability to win, *i.e.,* the highest reward.

As discussed above, it is necessary for the selective feature augmentation to (1) require feedback (*i.e.,* performance) from the model training and testing, and (2) update the feature augmentation policy based on the feedback. Therefore, RL naturally has a close relation with the problem, where the training data $\mathbf{T}_k$ at each iteration can be regarded as the *state*. Similar to the MAB solution, the *action* is to select table as well as features to augment given each state. Then the *environment* retrains the model, computes the *reward* and updates the augmentation criteria.

Deep reinforcement learning (DRL). Although the MAB-based solution can also leverage the feedback, it has several limitations. First, the augmentation criteria is implemented with a fixed function with poor flexibility. Second, some significant feature characteristics for feature selection, such as the mutual information between features and label column L [6], are not considered. To address these issues, we propose to use DRL that encodes the state in a unified representation (including the data characteristic), supports the "deletion" action for features tracing back and utilizes more powerful neural network to build sophisticated augmentation criteria.

Branch DQN. Recall that in selective feature augmentation, the action potentially consists of two types, *i.e.,* table selection and feature selection. However, in MAB based method, only

the table is selected by Equation (1), and the feature selection is conducted separately. Apparently, we hope that they are solved in a consolidated solution such that the correlation between tables and features can be well captured, leading to a more robust model. Intuitively, if a table can significantly improve the performance of the downstream model, it will likely contain useful features. Therefore, table selection can achieve feature filtering and deletion of batch features. Then, the action of selecting a feature is a more fine-grained way to generate the augmentation plan. Therefore, we propose to use the branch DQN [38], [41] that simultaneously embeds both the table and feature selection into the powerful deep neural network for more holistic and accurate augmentation. Next, we illustrate the details of the branch DQN.

B. Feature Augmentation Using Branch DQN

In this section, we first introduce the key factors in the branch DQN, and then the overall training process.

State. Each state $\mathbf{s}_k$ represents the situation of the table $\mathbf{T}_k$ in each iteration. Recall that $\mathbf{T}_k$ consists of the joined tables and the augmented features, which should be considered in the state representation. Moreover, we also compute some feature characteristics from $\mathbf{T}_k$ as features for neural network training.

In short, the representation of $\mathbf{s_k}$ considers 3 aspects.

(1) *The joined tables.* Given $\mathcal{T}$ with m tables, we use an m-dimension vector Ψ_k to denote which tables have been joined. To be specific, each element ψ_k^i of Ψ_k is either 0 or 1, where $\psi_k^i = 1$ denotes that T_i is joined, and 0 otherwise.

(2) *The augmented features.* Suppose that each table T_i has d_i attributes (features), and thus Φ_k^i is a d_i-dimension vector that indicates which features in T_i have been selected. Each dimension of Φ_k^i corresponds to a feature, either 1 (the feature is augmented) or 0 (not augmented). Then, we concatenate the m vectors as $\mathcal{F}_k$, with a dimension $D_F = \sum_{i=1}^{m} d_i$, to represent the augmented features.

(3) *Feature characteristics.* For each attribute (feature) f_{ij}, it is necessary to provide some criteria indicating the importance of each attribute. We consider three different criteria–Variance, Pearson correlation coefficient and Mutual information–to evaluate each attribute.

(a) Variance (VAR). Variance $var(f_{ij})$ [35] can reflect the volatility of feature values. If the variance is small, such a feature has little effect on the model. The greater the variance, the better the feature distinguishes itself in the model. For feature f_{ij} from T_i, we can compute the variance $var(f_{ij})$ as

$$var(f_{ij}) = \frac{1}{|T_i|}\sum_{j=1}^{N}(x_j - \mu)^2, \tag{2}$$

where x_j is the value of feature f_{ij} on data instance x from the table T_i and $\mu = \frac{1}{|T_i|}\sum_{j=1}^{N} x_j$.

(b) Pearson correlation coefficient (PCC). Pearson correlation coefficient $\mathcal{P}(f_{ij}, L)$ [5] is a normalized measurement of the covariance, measuring the strength of linearity between the attribute f_{ij} and label column L. Pearson correlation coefficient is between -1 and 1. The larger the absolute value, the stronger the correlation. $\mathcal{P}(f_{ij}, L)$ can be computed as

$$\mathcal{P}(f_{ij}, L) = \frac{cov(f_{ij}, L)}{\sqrt{var(f_{ij})var(L)}}, \tag{3}$$

where $cov(f_{ij}, L)$ is the covariance of f_{ij} and L, and $var()$ is the variance mentioned above.

(c) Mutual information (MI). Mutual information $\mathcal{M}(f_{ij}, L)$ [19] between f_{ij} and L quantifies the amount of information obtained about L, through taking f_{ij} into consideration. Mutual information can evaluate both linear and non-linear relationships between f_{ij} and L. The greater the mutual information, the more important the feature. We can compute $\mathcal{M}(f_{ij}, L)$ as

$$\mathcal{M}(f_{ij}, L) = \int_{f_{ij}}\int_{L} P(f_{ij}, L) log\frac{P(f_{ij}, L)}{P(f_{ij}), P(L)}, \tag{4}$$

where $P(f_{ij}, L)$ is the joint probability distribution of f_{ij} and L; $P(f_{ij})$ and $P(L)$ are the marginal probability distribution.

Overall, we use the vector Ω_F to denote the data characteristics of the current tables. Specifically, if table T_i has been selected (*i.e.,* T_i has been joined with T_b), for each feature in T_i, the data characteristics of the feature constitute the triple (VAR, PCC, MI), otherwise the triple is (0, 0, 0). The reason is that for those tables, which are not joined with T_b, we cannot obtain the data distribution of the join results before executing the join operation. Thus, we cannot compute the feature characteristics. Therefore, Ω_F is a $3D_F$-dimension vector that concatenates the data characteristics of all features.

Finally, we represent $\mathbf{s}_k$ as concatenating the vectors w.r.t. the above three factors, *i.e.,* $\mathbf{s}_k = (\Psi_k, \Phi_k, \Omega_F)$.

Example 8: For each iteration, we first construct the state representation vector $\mathbf{s}_k$. As shown in Fig. 4-(a), we can see that T_3 is selected, the corresponding position of Ψ_k is set to 1. We can also see that the feature $f_{3,3}$ is selected, so the corresponding position in Φ_k is 1 now. Additionally, we compute the characteristic of the features in T_3. Then, after concatenating Ψ_k, Φ_k and Ω_F, we can obtain the representation $\mathbf{s}_k$ of current state, which is then fed into the branch DQN model (Fig. 4-(d)).

Remark. Since the base table T_b is in $\mathbf{T}_i$, we also take the base table T_b into consideration when representing the state. Note that the label column L is in the base table, we exclude L in $\mathbf{s}_k$ because it cannot provide any more information for the learning process. Thus, we have the state representation vector $\mathbf{s}_k = (\Psi_k, \Phi_k, \Omega_F)$, where Ψ_k, Φ_k and Ω_F are $m+1$, $D_F + d_b$ and $3(D_F + d_b)$ dimension vector, respectively.

Action. At a high level, there are two types of actions, *i.e.,* joinable table selection and augmentation feature selection, which belong to different action spaces. For each type, we also have two choices, *i.e.,* add or delete a table (feature). To be specific, the action space w.r.t. joinable table selection is represented by $\mathcal{A}_T = \{a_T^1, a_T^2, ..., a_T^m, a_T^{m+1}, ..., a_T^{2m}\}$, where $a_k = a_T^i$ (or a_T^{m+i}) denotes that at the k-th iteration, we join (or remove) the table T_i from $\mathbf{T}_k$. Similarly, $\mathcal{A}_F$ =

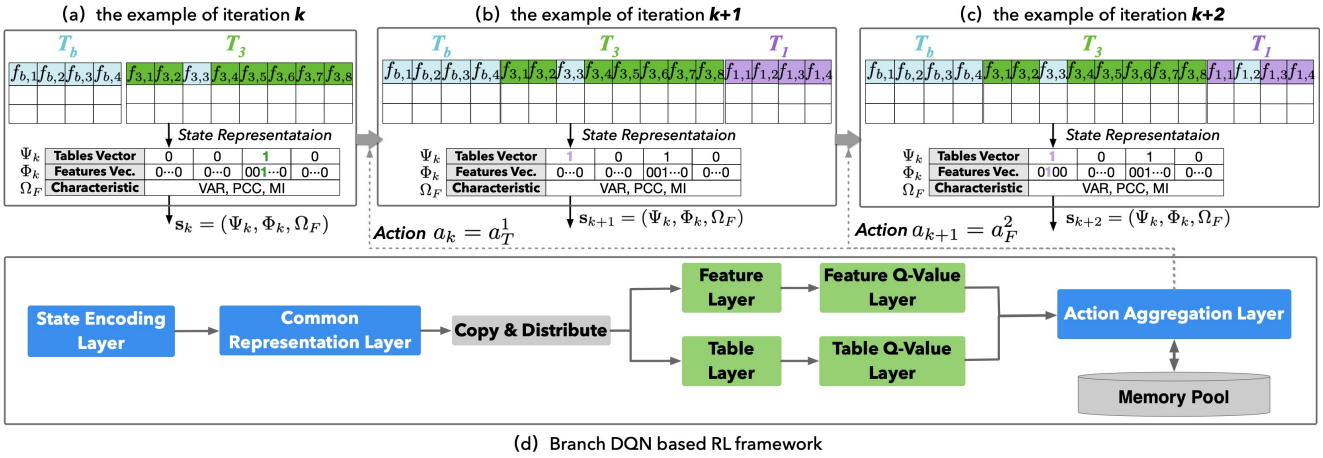

Fig. 4. Branch DQN based RL framework and a running example

$\{a_F^1, a_F^2, ..., a_F^{M_F}, ..., a_F^{2M_F}\}$ denotes the action space w.r.t. feature augmentation.

Remark. Note that at each state, we will mask some invalid actions. More concretely, if a table has not been joined, features in the table should be masked. If there is a transitive join pattern like $T_1 \bowtie T_2 \bowtie T_3$, T_3 should be masked if T_2 is not joined with T_1.

Example 9: As shown in Fig. 4 with dot line, the agent selects a new action a_k according to the state vector. We can see that $a_k = a_T^1$, *i.e.,* a new table T_1 is selected (Fig. 4-(b)). Then, we join T_1 with $\mathbf{T}_k$ and obtain $\mathbf{T}_{k+1} = \mathbf{T}_k \bowtie T_1$. After constructing the new state vector $\mathbf{s}_{k+1}$ and feeding it into the network (Fig. 4-(d)), we get a new action $a_{k+1} = a_F^2$, which means that the feature $f_{1,2}$ from T_1 is selected and added to train the downstream model.

Reward. The reward r_k indicates the performance change of the downstream model before and after the action a_k is executed, *i.e.,* the model difference of the downstream model between the $(k-1)$-th iteration and the k-th iteration. r_k can be calculated by $M(\mathbf{T}_k, T_{val}) - M(\mathbf{T}_{k-1}, T_{val})$.

Remark. Note that if $a_k = T_i$ from $\mathcal{A}_T$, we just use the joined table $\mathbf{T}_{k-1} \bowtie T_i$ to calculate the model performance improvement as the reward of a_k. We do not augment any features from T_i. In other words, after executing $a_k \in \mathcal{A}_T$, $\mathbf{T}_k$ equals to $\mathbf{T}_{k-1}$.

Environment. The environment encodes the joinable tables, augmentation features and corresponding data characteristics as the state to support the decision making. Given an action, it either adds a table (feature) or deletes a table (feature), and retrains the downstream model. In addition, the environment executes the action, calculates the reward and sends the transition to the agent.

Agent. The agent takes current selected tables and features into account and uses the RL model to choose a new action according to the estimated long-term reward. In addition, the agent receives the transition from the environment of each iteration and conducts the learning process by using the experiences from the memory pool.

Model Design. Now, considering the typical DQN framework [39], the agent interacts with the environment iteratively. The agent perceives the state s at each iteration. Then, it chooses an action from $\mathcal{A}$ for the corresponding state and observes a reward signal r from the environment. The agent seeks to maximize the long-term reward $R_t = \sum_{k=t}^{\infty} r_k$, through learning an augmentation policy π. Actually, the learning the policy π is to learn a state-action value function (Q function for short), which is defined as $Q^\pi(s,a) = \mathbb{E}[r + \gamma \max'_a Q^\pi(s', a')]$ and can be computed recursively with dynamic programming.

However, in our feature augmentation scenario, our actions are from two different action spaces, *i.e.,* joinable table selection $\mathcal{A}_T$ and augmentation feature selection $\mathcal{A}_F$.

Apparently, it is difficult for the DQN framework to learn a policy π coping with the action from different action spaces. Since $\mathcal{A}_T$ and $\mathcal{A}_F$ are from different action spaces, it is better to optimize for each action space relatively independently. However, there still exists connection between $\mathcal{A}_T$ and $\mathcal{A}_F$, and thereby we also need to take them into consideration together when making decisions. Thus, we use the branch DQN [38] to design the architecture of our neural network. The key idea is that the network distributes the representation of the Q function across multiple network branches while keeping a shared decision layer to make a final decision (either an action in $\mathcal{A}_T$ or $\mathcal{A}_F$ in this paper).

As shown in Fig. 4, once the state vector is fed into the branch network (Fig. 4-(d)), a common representation layer is used to learn the general information across the two types of actions (*i.e.,* $\mathcal{A}_T$ and $\mathcal{A}_F$). Then, we distribute the representation of two different actions on two action branches, *i.e.,* table dimension layer and feature dimension layer. The two branches (*i.e.,* Table Q-Value layer and Feature Q-Value layer) estimate the Q-value of each action relatively independently and send them to the aggregation layer (*i.e.,* Action layer). Finally, the two types of actions are combined to produce estimates of the distributed action values and the action with the largest Q-value is chosen. In the back propagation stage, the two branches update their strategies relatively independently (The green layer in Fig. 4). The representation layer can use the information from the two branches to update the common strategy. Then, the two branches can use each other's informa-

TABLE 1
STATISTICS OF DATASETS.

Datasets	Task types	# Cand. tables	# Cand. features
School	Classification	16	695
XuetangE	Classification	23	204
Air	Regression	34	338

tion to make decisions based on the common representation layer.

Model Training. Similar to traditional DQN, branch DQN is also trained as the Q-value estimator, *i.e.,* finding the optimal parameters θ to estimate the Q-value($Q(s, a; \theta) \approx Q^*(s, a)$). In our AutoFeature, we use the prioritized memory replay [34] to enable online and off-policy learning via reusing past experiences. The transitions together with their priorities are stored in a prioritized memory pool in order to replay important experience transitions more frequently, which have a high expected learning progress.

Model Inference. After the model is trained, it can be used to infer the feature augmentation plan. Concretely, the model repeatedly chooses the action based on the state of $\mathbf{T}_k$. During the inference process, we do not update the model.

V. EXPERIMENT

A. Experimental Setup

Datasets. We use three real-world datasets (School, XuetangE and Air) to conduct our experimental study. We have two datasets (School and XuetangE) for classification tasks and one dataset (Air) for the regression task. As shown in Table 1, we can obtain the basic information about the three datasets. The number of candidate joinable tables and the number of the candidate features are shown in the column "# Cand. tables" and "# Cand. features." For each dataset, we use 60% of the data as the training set, 20% as the validation set and 20% as the test set. For efficiency, we sample each dataset. The sampling ratios are 30% for School, 10% for XuetangE and 15% for Air.

Next, we provide detailed information about these datasets.

(1) School is a binary classification dataset that contains 16 candidate joinable tables, which are collected from NYU Auctus [1]. The task of School is to predict the performance of each school based on student attributes, course attributes and some historical surveys.

(2) XuetangE is another classification dataset in an online education scenario, predicating "whether a student enrolls a certain course or not." XuetangE contains 23 candidate tables including student information, course selection information and other tables. All of them are collected in real scenario.

(3) Air is a regression dataset aiming to predict the air quality of a city on a given date. Air contains 34 candidate tables of different air metrics from NYU Auctus [1] and google [2].

Evaluation Metrics. Since we have two different types of tasks, *i.e.,* classification and regression, we use *Area Under Curve*(AUC) and *Mean Squared Error*(MSE) for these two tasks, respectively.

(1) AUC [16] is a commonly used metric for classification tasks to assess the discriminative power of the predictive classification model. AUC is the area under the ROC curve. Thus, the value of AUC is between [0,1]. ROC curve is the plot of the rate of true positives (TPR, computed by TPR = TP/(TP+FN)) versus the rate of false positives (FPR, computed by FPR = FP/(TN+FP)) at different probability cutoffs. The higher the AUC score, the better the model.

(2) MSE [28] is often used to evaluate the performance of the regression model. MSE takes the distances from the normalized points of ground truth(y) to the regression line ($y_{predict}$) and squaring them, *i.e.,* $\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y - y_{predict})^2$. The lower the MSE, the better the prediction.

Baselines. We compared AutoFeature with a variety of typical solutions, including basic solutions like Non and All, filter-based solutions, wrapper-based solutions like Backward and Forward, as well as ARDA. All these methods are iterative augmentation, except Non and All.

(1) Non is the original baseline that no new feature is added to train the model.

(2) All is a simple method that all the candidate tables from $\mathcal{T}$ are joined with the base table and all features are added to train the downstream model.

(3) Random is a straightforward baseline that treats the features from different tables as independent individuals. Given the number of required features (*i.e.,* α), it randomly selects α features to form the feature subset.

(4) Filter-based methods are simple but straightforward baselines. Filter-based methods first join all candidate tables from $\mathcal{T}$ with the base table T_b. Then, they evaluate the correlation between the features and the target variable. We use Chi-square, Gini index, mutual information and XGBoost score for classification tasks and use mutual information and XGBoost score for regression tasks, because Chi-square and Gini index are only suitable for classification. Afterwards, these methods select the most top-k important features based on the metrics.

(5) Backward is a wrapper-based feature augmentation method [21], which first joins all candidate tables together. Afterwards, in each iteration, it evaluates which feature degrades the performance most and removes it. Since Backward joins all features together first, and then eliminates one-by-one, we just report the final result of this method in Fig. 5.

(6) Forward is another wrapper-based feature augmentation method [21]. Initially, Forward only has the features from the base table. Then, in each iteration, Forward keeps adding the feature which best improves the performance of the downstream model.

(7) ARDA is a feature augmentation system, which heuristically selects top-k tables to join and uses a random injection-based feature augmentation to search candidate feature subsets.

Solutions. We compare our MAB-based and Branch-DQN-based solutions with the baselines.

B. The Effectiveness of AutoFeature

For different datasets, we are going to answer the question that *Does AutoFeature solve the feature augmentation problem better than other baselines?* We apply all the methods mentioned in section V-A to the three different datasets.

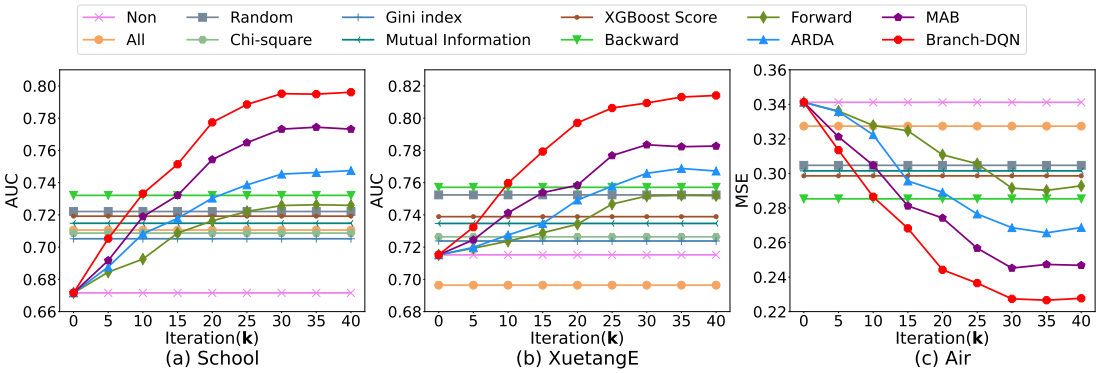

Fig. 5. The performance of different methods

As shown in Fig. 5, we show the effectiveness of different methods on the three datasets. The three figures correspond to the results on the three datasets (*i.e.,* `School`, `XuetangE` and `Air`), where the x-axis represents the number of iterations while applying different methods, and the y-axis represents the performance of the downstream model. For `School` and `XuetangE`, we use `AUC` to evaluate the performance of the model. For `Air`, we use `MSE` to evaluate the performance.

Remark. k represents the number of iterations for different methods. For `Branch-DQN`, k refers to the number of iterations of action selection in the inference stage rather than the number of training iterations. That is, at each $k-$th iteration, we aim to either join a table or augment a feature.

Result on `School`. The effectiveness of different methods on `School` is shown in Fig. 5 (a). On this dataset, we can see that all methods can improve the performance of the downstream model. We can find that as more features are added, the `AUC` increases and gradually converges after about 30 iterations. The reason is that as the useful features are selected, the remaining features can bring little information and some of them may be noise.

For `All` and `Random`, they can both improve the performance of the model (*i.e.,* 71.1% for `All` and 72.2% for `Random` after 40 iterations), but they do not outperform `Backward`, `Forward`, `ARDA` and `AutoFeature`. For `All`, the reason is that although some new features can help, adding all features without filtering and selection is likely to introduce some irrelevant features, which are likely to be the noise for the model. This tells us that not all features in joinable tables are useful, so it is necessary to select relevant features. For `Random`, compared with `All`, there is no much improvement, and the effect is not stable enough, *i.e.,* sometimes adding more features might decrease the performance because it does not consider the quality of the features.

The two wrapper-based `Backward` and `Forward` perform better than `Random`, and they achieve the `AUC` of 73.2% and 72.6% (after 40 iterations), respectively. This is because both of them use the feedback from the downstream model. Furthermore, they both use the prediction result to rank the features, then select new features that best improve the model performance (`Forward`), or exclude the features that hurt the model performance most (`Backward`). However, `Backward` and `Forward` do not significantly improve the performance of the model. The main reason is that they are unable to well explore the search space of features. In addition, their greedy augmentation strategies can easily lead to local optimal, and thus fail to discovery more useful feature combinations.

For filter-based methods, we can see that they are worse than wrapper-based methods (*i.e.,* `Backward` and `Forward`). `Backward` (73.2%) and `Forward` (72.6%) outperform `Chi-square` (70.9%), `Gini index` (70.5%), `Mutual information` (71.5%) and `XGBoost score` (71.9%). The reason is that filter-based methods ignore the relationship between different features and do not consider the feedback from the downstream model.

Among the baselines, `ARDA` performs better (74.8%) than `Random`, `Backward` and `Forward`. The reason is that the random injection based feature selection method of `ARDA` can select useful features. However, it still has some limitations. First, `ARDA` selects candidate tables based on the relevance to the base table, lacking consideration of the impact on the model. Second, `ARDA` separates the steps candidate table selection and feature augmentation, making it difficult to dynamically update the strategy.

Our solution `MAB` and `Branch-DQN` of `AutoFeature` significantly outperforms all baselines, obtaining the `AUC` of 77.3% and 79.6%, respectively. The reasons are two-fold. On the one hand, we use the trial-and-error approach to discover the useful feature combinations, which can capture the relationship between tables and features. On the other hand, we directly use the feedback from the model to guide the augmentation process and use the exploration-exploitation strategy to discover potentially usefully features. We can see that `Branch-DQN` performs better than `MAB`. That is because (1) `Branch-DQN` takes more characteristics of the features in the augmentation process, (2) `Branch-DQN` can update the augmentation policy based on the feedback from the model.

Result on `XuetangE`. The effectiveness on `XuetangE` is shown in Fig. 5 (b). We can see that `MAB` and `Branch-DQN` achieve the `AUC` of 78.3% and 81.4%, respectively, which outperform `All` (69.6%), `Random` (75.2%), `Backward` (75.7%), `Forward` (75.1%) and `ARDA` (76.8%). We can see that `All` does not improve the performance of the downstream model since the proportion of noise features is greater than that of `School`, and thus too many noisy features are introduced. In addition, the performance of `Forward` is similar

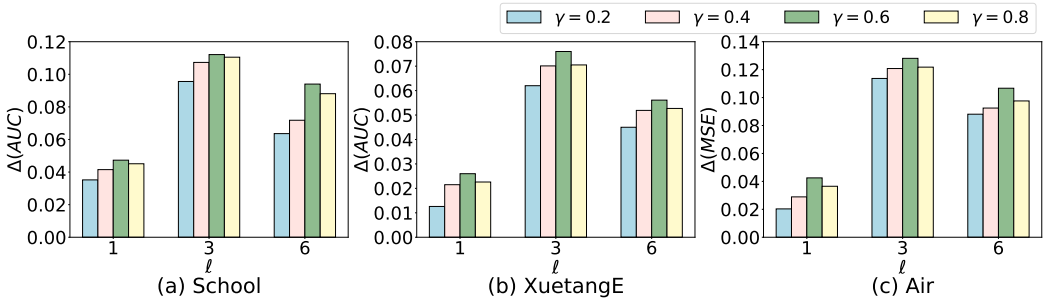

Fig. 6. Sensitivity analysis of MAB

to Random. The reason is that Forward can hardly discover feature combinations, but in XuetangE, feature combinations are more useful than evaluating a single feature each time.

Result on Air. The effectiveness on Air is shown in Fig. 5 (c). Different from School and XuetangE, Air is a regression dataset. Thus, we use MSE to evaluate the performance of the downstream model. The lower the MSE, the better the model performance. We can see that all methods can improve the performance of the model, where MAB (0.247) and Branch−DQN (0.228) significantly outperform other methods, *e.g.,* Backward (0.285), Forward (0.293) and ARDA (0.269). We can see that at 40 iterations, for ARDA, the performance of the downstream model decreases. The reason is that ARDA just selects candidate tables that are semantically relevant to the base table, but these features may not directly improve the model performance.

Efficiency of different methods. In terms of inference efficiency of different methods, MAB is rather efficient. For example, on XuetangE, it takes 473.2s for MAB to iterate 30 times, while other methods take 14.58s (Random), 7476s (Backward), 1754 (Forward), 832s (ARDA) and 359s (Branch−DQN). Since Random only randomly augments features and only trains once, it is very fast. However, the effect of Random is not as good as other methods, and it is not stable enough. Although Branch−DQN needs several hours to train the RL model, it is acceptable since the most performance improvement of the downstream model is the main goal of our solution.

C. Sensitivity Analysis of MAB

For MAB, recall that there are two parameters in the solution, namely, γ and ℓ. We are going to answer the question that *how γ and ℓ affect the performance of MAB?* Therefore, we try different γ and ℓ on three datasets.

As shown in Fig. 6, we show the effect of different γ and ℓ on the model performance on the three datasets, where the x-axis represents different values of ℓ and y-axis represents the performance improvement of the downstream model. The effect of different γ is represented by different colored bars.

Varying ℓ. As shown in Fig. 6, we can see that the different values of ℓ will affect the performance. When $\ell = 3$, the performance of the downstream model can be improved most. $\ell = 3$ is better than $\ell = 1$. For example, on School, the improvement of AUC (Δ(AUC)) is nearly three times higher. The main reason is that when ℓ is too small, it is hard for MAB to capture the relationship between features, *i.e.,* MAB cannot discovery enough useful feature combinations. Similarly, $\ell = 3$ is better than $\ell = 6$. That is because when ℓ is set too large, the size of the feature subset in each iteration increases. Thus, noisy features can be easily introduced, which may hurt the performance of the downstream model.

Varying γ. As shown in Fig. 6, we can see that the difference of γ also affects the effect of MAB. We can see that $\gamma = 6$ is the reasonable choice. For example, on XuetangE, when $\gamma = 0.6$, MAB performs better than $\gamma = 0.2/0.4/0.8$. The reason is that when γ is close to 0, based on Equation (1), MAB actually tends to use "exploitation" strategy to augment features, *i.e.,* MAB will augment features from the explored candidate tables more. Thus, MAB is more likely to lose potentially useful features from the unexplored tables. Instead, when γ is close to 1, MAB tends to use the "exploration" strategy during the feature augmentation process, *i.e.,* MAB will pay more attention to those tables that are rarely explored, aiming to discover more potentially useful features. However, too much exploration may easily introduce noise features.

D. DQN and Branch−DQN

For Branch−DQN, since the architecture is different from traditional DQN, we carefully compare DQN and Branch−DQN in terms of performance and efficiency. The results are shown in Fig. 7 and Fig. 8, respectively.

Apply DQN to feature augmentation. Recall that in Section IV, we use Branch−DQN to solve the feature augmentation problem. Thus, we can also use DQN to solve this problem. Compared with Branch−DQN, there are two differences when using DQN. First, we only consider the actions in $\mathcal{A}_F$, *i.e.,* the action is either "augment" or "drop" a feature. Second, for the neural network, we use the typical fully connected neural network architecture instead of the branch architecture.

As shown in Fig. 7, we show the effect of DQN and Branch−DQN on different datasets, where x-axis represents the iteration times while inference, and y-axis represents the improvement of the downstream model (Δ(AUC) for School and XuetangE, Δ(MSE) for Air) after feature augmentation by DQN and Branch−DQN. As shown in Fig. 8, we show the training time of DQN and Branch−DQN on the three different datasets, where x-axis represents the different dataset and y-axis represents the training time in minutes.

Difference in performance. From Fig. 7, we can see that Branch−DQN obviously outperforms DQN on the three

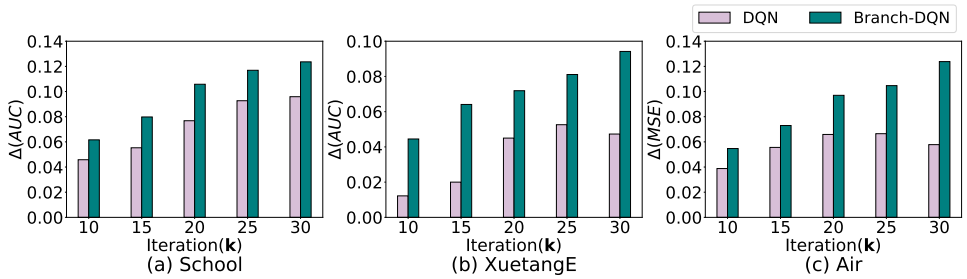

Fig. 7. Comparison of DQN and Branch-DQN

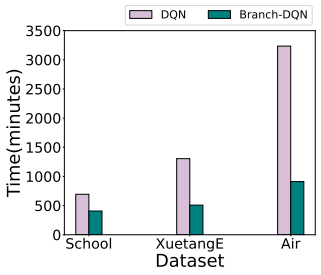

Fig. 8. Comparison of training time

datasets, *i.e.,* Branch-DQN can achieve larger performance improvement of the downstream model. For example, for XuetangE, after 30 iterations, the performance improvement of using Branch-DQN is nearly two times than that of using DQN. In addition, in Fig. 7 (b) and (c), the performance improvement for DQN decreases after 25 (for XuetangE) and 20 (for Air) iterations. The reason is that Branch-DQN can capture the relationship between candidate tables and the relationship between tables and features.

Difference in efficiency. From Fig. 8, we can see that DQN takes much more time than Branch-DQN to train the RL model. For example, on Air, compared with Branch-DQN, DQN takes three times more time to train the RL model, which is unacceptable in practice. In addition, we can see that the training time spent by DQN varies greatly on different datasets, however, the training time of Branch-DQN has not changed that much on different datasets. The reason is that DQN needs more exploration to capture the relationship of the features, which are from different candidate tables.

VI. Related Work

Feature augmentation for relational data. There exists some work on feature augmentation. (1) Kumar et al. [27] and Shah et al. [36] mainly solve the problem that when performing key-foreign key join, where join operations can be avoided without sacrificing the performance of the model. This is different from the problem we are trying to solve. On the one hand, their goal is to avoid unnecessary joins and to keep the performance of the model. However, our goal is to augment useful features and to improve the performance of the model. On the other hand, [27] and [36] are limited to PK-FK join, but we do not limit the join type. We allow the users to specify the join operators. (2) Chepurko et al. [14] proposed ARDA, a feature augmentation framework that can handle different types of join. ARDA leverages existing data discovery tools (*e.g.,* [17]) to score the candidate tables and uses a heuristic algorithm to select features. Hence, the performance of ARDA heavily relies on the scores given by Aurum. Unfortunately, the scores are not always accurate since they are not model-aware.

Feature selection. Given a dataset, feature selection selects an optimal subset of features, aiming to maximize the performance of the model. The feature selection methods can be categorized into filtering methods, wrapper methods and embedded methods [12], [21]. (1) Filtering methods [7], [24], [26] rank features by some evaluation metrics (*e.g.,* mutual information) and select the top-ranked features. Although filtering methods are efficient, they do not take the dependencies of features and the variation of the downstream model into consideration. (2) Wrapper methods [20], [25], [26], [32] mainly consist of feature subset generation and feature subset evaluation. Wrapper methods iteratively repeat the two steps until the model performance meets the requirements. Wrapper methods use the downstream model for subset evaluation. Thus, the feature subset provided by wrapper methods can improve the model performance well. However, training a new model for each subset makes wrapper methods computationally expensive. (3) Embedded methods [4], [4], [7], [22] integrate the feature selection into the model training, using their built-in feature selection mechanism to select feature subsets. Embedded methods are less computationally expensive than wrapper methods and can provide better feature subsets than filtering methods.

Database management techniques for ML. In addition to features, acquiring more data instances can also benefit the ML model, but the data may be insufficient, dirty or unlabled. Hence, data preparation [11] can be leveraged to improve the ML model performance, including data discovery [10], [31], data cleansing [8], [23], data labeling [9], [15], [29], [30], [43].

VII. Conclusion

In this paper, we study the problem of automatic feature augmentation for supervised learning. To be specific, we conduct the design space exploration (*e.g.,* forward selection, backward selection, RL-based method, etc.) of this problem. In particular, we design two algorithms leveraging the idea of handling the exploration-exploitation trade-off. One is the MAB-based solution that regards each table as an arm and conducts feature selection after the arm (table) is pulled (joined). The other one uses the branch DQN to leverage the neural network to judiciously choose tables or features.

Acknowledgment

This work is supported by NSF of China (61925205, 62102215, 62072261), China National Postdoctoral Program for Innovative Talents (BX2021155), China Postdoctoral Science Foundation (2021M691784), Shuimu Tsinghua Scholar and Zhejiang Lab's International Talent Fund for Young Professionals. This work was also supported by Ant Group through Ant Research Program.

REFERENCES

[1] https://auctus.vida-nyu.org/.

[2] https://www.google.com/.

[3] P. Auer. Using upper confidence bounds for online learning. In *FOCS 2000*, pages 270–279. IEEE Computer Society, 2000.

[4] R. Battiti. Using mutual information for selecting features in supervised neural net learning. *IEEE Trans. Neural Networks*, 5(4):537–550, 1994.

[5] J. Benesty, J. Chen, Y. Huang, and I. Cohen. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009.

[6] M. Beraha, A. M. Metelli, M. Papini, A. Tirinzoni, and M. Restelli. Feature selection via mutual information: New theoretical insights. In *IJCNN 2019*, pages 1–9. IEEE, 2019.

[7] A. Blum and P. Langley. Selection of relevant features and examples in machine learning. *Artif. Intell.*, 97(1-2):245–271, 1997.

[8] C. Chai, L. Cao, G. Li, J. Li, Y. Luo, and S. Madden. Human-in-the-loop outlier detection. In *SIGMOD Conference 2020*, pages 19–33, 2020.

[9] C. Chai, G. Li, J. Li, D. Deng, and J. Feng. Cost-effective crowdsourced entity resolution: A partial-order approach. In *SIGMOD Conference 2016*, pages 969–984. ACM, 2016.

[10] C. Chai, J. Liu, N. Tang, G. Li, and Y. Luo. Selective data acquisition in the wild for model charging. *PVLDB*, 15(7):1466–1478, 2022.

[11] C. Chai, J. Wang, Y. Luo, Z. Niu, and G. Li. Data management for machine learning: A survey. *TKDE*, pages 1–1, 2022.

[12] G. Chandrashekar and F. Sahin. A survey on feature selection methods. *Comput. Electr. Eng.*, 40(1):16–28, 2014.

[13] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In B. Krishnapuram, M. Shah, A. J. Smola, C. C. Aggarwal, D. Shen, and R. Rastogi, editors, *SIGKDD 2016*, pages 785–794. ACM, 2016.

[14] N. Chepurko, R. Marcus, E. Zgraggen, R. C. Fernandez, T. Kraska, and D. R. Karger. ARDA: automatic relational data augmentation for machine learning. *VLDB*, 13(9):1373–1387, 2020.

[15] L. Cui, J. Chen, W. He, H. Li, W. Guo, and Z. Su. Achieving approximate global optimization of truth inference for crowdsourcing microtasks. *Data Science and Engineering*, 6(3):294–309, 2021.

[16] T. Fawcett. An introduction to ROC analysis. *Pattern Recognit. Lett.*, 27(8):861–874, 2006.

[17] R. C. Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, and M. Stonebraker. Aurum: A data discovery system. In *ICDE*, pages 1001–1012, 2018.

[18] R. C. Fernandez, E. Mansour, A. A. Qahtan, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, M. Stonebraker, and N. Tang. Seeping semantics: Linking datasets using word embeddings for data discovery. In *ICDE 2018*, pages 989–1000. IEEE Computer Society, 2018.

[19] B. Gierlichs, L. Batina, P. Tuyls, and B. Preneel. Mutual information analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 426–442. Springer, 2008.

[20] D. E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley, 1989.

[21] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, 2003.

[22] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik. Gene selection for cancer classification using support vector machines. *Mach. Learn.*, 46(1-3):389–422, 2002.

[23] S. Hao, C. Chai, G. Li, N. Tang, N. Wang, and X. Yu. Outdated fact detection in knowledge bases. In *ICDE 2020*, pages 1890–1893. IEEE, 2020.

[24] G. H. John, R. Kohavi, and K. Pfleger. Irrelevant features and the subset selection problem. In W. W. Cohen and H. Hirsh, editors, *Machine Learning, Proceedings of the Eleventh International Conference, Rutgers University, New Brunswick, NJ, USA, July 10-13, 1994*, pages 121–129. Morgan Kaufmann, 1994.

[25] J. Kennedy and R. Eberhart. Particle swarm optimization. In *ICNN'95*, pages 1942–1948. IEEE, 1995.

[26] R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artif. Intell.*, 97(1-2):273–324, 1997.

[27] A. Kumar, J. F. Naughton, J. M. Patel, and X. Zhu. To join or not to join?: Thinking twice about joins before feature selection. In *SIGMOD 2016*, pages 19–34. ACM, 2016.

[28] E. L. Lehmann and G. Casella. *Theory of point estimation*. Springer Science & Business Media, 2006.

[29] G. Li and C. C. et al. CDB: optimizing queries with crowd-based selections and joins. In *SIGMOD Conference 2017*, pages 1463–1478. ACM, 2017.

[30] G. Li and C. C. et al. CDB: A crowd-powered database system. *PVLDB*, 11(12):1926–1929, 2018.

[31] J. Liu, F. Zhu, C. Chai, Y. Luo, and N. Tang. Automatic data acquisition for deep learning. *PVLDB*, 14(12):2739–2742, 2021.

[32] P. M. Narendra and K. Fukunaga. A branch and bound algorithm for feature subset selection. *IEEE Trans. Computers*, 26(9):917–922, 1977.

[33] D. Pfeffermann and C. R. Rao. *Sample surveys: design, methods and applications*. Elsevier, 2009.

[34] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. In Y. Bengio and Y. LeCun, editors, *ICLR 2016*, 2016.

[35] H. Scheffe. *The analysis of variance*, volume 72. John Wiley & Sons, 1999.

[36] V. Shah, A. Kumar, and X. Zhu. Are key-foreign key joins safe to avoid when learning high-capacity classifiers? *Proc. VLDB Endow.*, 11(3):366–379, 2017.

[37] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nat.*, 529(7587):484–489, 2016.

[38] A. Tavakoli, F. Pardo, and P. Kormushev. Action branching architectures for deep reinforcement learning. In S. A. McIlraith and K. Q. Weinberger, editors, *AAAI 2018*, pages 4131–4138. AAAI Press, 2018.

[39] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In D. Schuurmans and M. P. Wellman, editors, *AAAI 2016*, pages 2094–2100. AAAI Press, 2016.

[40] J. Vermorel and M. Mohri. Multi-armed bandit algorithms and empirical evaluation. In *ECML 2005*, volume 3720, pages 437–448. Springer, 2005.

[41] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas. Dueling network architectures for deep reinforcement learning. In M. Balcan and K. Q. Weinberger, editors, *ICML 2016*, volume 48, pages 1995–2003. JMLR.org, 2016.

[42] R. C. Wilson, E. Bonawitz, V. D. Costa, and R. B. Ebitz. Balancing exploration and exploitation with information and randomization. *Current Opinion in Behavioral Sciences*, 38:49–56, 2021.

[43] X. Zhou, C. Chai, G. Li, and J. Sun. Database meets artificial intelligence: A survey. *IEEE TKDE*, 34(3):1096–1116, 2022.