# Neural Feature Search: A Neural Architecture for Automated Feature Engineering

Xiangning Chen[1,*], Qingwei Lin[2,*,**], Chuan Luo[2,*], Xudong Li[3], Hongyu Zhang[4],
Yong Xu[2], Yingnong Dang[5], Kaixin Sui[2], Xu Zhang[6], Bo Qiao[2],
Weiyi Zhang[2], Wei Wu[7], Murali Chintalapati[5] and Dongmei Zhang[2]
[1]Tsinghua University, China
[2]Microsoft Research, China
[3]University of California, Los Angeles, United States
[4]The University of Newcastle, Australia
[5]Microsoft Azure, United States
[6]Nanjing University, China
[7]University of Technology Sydney, Australia
cxn15@mails.tsinghua.edu.cn, {qlin, chuan.luo, yox, yidang, kasui, boqiao, v-weiyzh, muralic, dongmeiz}@microsoft.com,
lixudong@ucla.edu, hongyu.zhang@newcastle.edu.au, zhangxu037@smail.nju.edu.cn, william.third.wu@gmail.com

***Abstract*—Feature engineering is a crucial step for developing effective machine learning models. Traditionally, feature engineering is performed manually, which requires much domain knowledge and is time-consuming. In recent years, many automated feature engineering methods have been proposed. These methods improve the accuracy of a machine learning model by automatically transforming the original features into a set of new features. However, existing methods either lack ability to perform high-order transformations or suffer from the feature space explosion problem. In this paper, we present Neural Feature Search (NFS), a novel neural architecture for automated feature engineering. We utilize a recurrent neural network based controller to transform each raw feature through a series of transformation functions. The controller is trained through reinforcement learning to maximize the expected performance of the machine learning algorithm. Extensive experiments on public datasets illustrate that our neural architecture is effective and outperforms the existing state-of-the-art automated feature engineering methods. Our architecture can efficiently capture potentially valuable high-order transformations and mitigate the feature explosion problem.**

***Keywords*-Feature Engineering, Automated Feature Engineering, Neural Architecture**

## I. Introduction

Feature Engineering is a crucial step for building an effective machine learning model. However, traditional feature engineering is largely a manual process, which is time-consuming and requires much domain knowledge. The traditional practice is also prone to bias and error as there is a lack of standard procedures to perform feature engineering.

In recent years, many automated feature engineering methods have been proposed. These methods apply transformation functions such as arithmetic operators to the raw features and generate the new, engineered features. Generally, automated feature engineering can be realized by *expansion-reduction*, which first expands the feature set and then performs feature selection. In the phase of expansion, all possible or randomly sampled transformation functions are applied to raw features [1]–[3]. In the phase of reduction, features are selected based on the improvement in model performance. However, the possible number of feature transformations is unbounded since transformation functions can be composed and applied recursively to features generated by previous transformation functions. The space of constructed features could grow exponentially (e.g., the possible number of features constructed by $division$ based on $n$ features is $O(n^2)$. It will become $O(n^4)$ by applying another transformation function on newly generated features). As a result, *expansion-reduction* suffers from the feature explosion problem. Recently, the paper [4] builds a transformation graph and employs Q-learning [5] to search for feature transformations. The resulting machine learning model built using the transformed features can achieve higher performance. However, it also has an exponential number of features in the bottom layer despite the introduction of the feature selection operator. LFE [6] manages to eliminate the problem by recommending a few promising transformation functions. However, it has a performance bottleneck since they do not consider a composition of transformations, i.e. high-order transformations, which have been demonstrated valuable by our experiments.

This paper presents Neural Feature Search (NFS), a neural architecture for automated feature engineering. In our architecture, we address the feature explosion problem by employing an effective searching algorithm to explore the most promising transformation rule (a series of transformation functions) for each raw feature. That is, we only perform the most valuable transformation rather than expansion and reduction. We support a number of transformation functions such as *logarithm*, *square root*, and *multiply*. We also support two special functions $delete$ and $terminate$, which perform feature selection and terminate the transformation process, re-

*Xiangning Chen, Qingwei Lin and Chuan Luo contributed equally to this work and share the first authorship of this work.
**Qingwei Lin is the corresponding author of this work.

spectively. In this way, the process of finding optimal features for the whole dataset can be divided into finding a series of transformation functions for each raw feature. We design a Recurrent Neural Network (RNN) based controller to generate such a series. The controller is trained through reinforcement learning to maximize the expected performance of the machine learning algorithm.

As an example, consider the problem of predicting heart disease based on the features $weight$ and $height$. Although the raw features could be useful indicators for this problem, a more effective feature is $Body\ Mass\ Index$ $(BMI)$ [7], which is defined as $BMI = \frac{weight}{height^2}$. Such a new feature is actually a series of transformation functions $[square, reciprocal, \times weight]$ based on the raw feature $height$: $reciprocal(square(height)) \times weight$. NFS is able to generate this third-order new feature by composing the raw features through multiple functions.

To evaluate the proposed approach, we have conducted extensive experiments on 41 public datasets, which have different numbers of features and instances and cover both classification and regression problems. The results show that NFS is effective and outperforms the existing state-of-the-art automated feature engineering methods. NFS has achieved the best performance on 37 out of 41 datasets. The experimental results also demonstrate that the performance of our approach continues to increase when the order of transformations increases.

Our main contributions can be summarized as follows:

- We propose a novel neural architecture for automated feature engineering. To our best knowledge, it is the first work that addresses the feature explosion problem and supports high-order transformations through deep learning.
- We have conducted extensive experiments on public datasets. The experiments show that our approach significantly outperforms the existing state-of-the-art methods for automated feature engineering.

The rest of this paper is organized as follows. In Section II, we review the related work on automated feature engineering and neural network controller. In Section III, we propose NFS and use it to find the best feature transformations. In Section IV, we perform extensive experiments to show the effectiveness of NFS. Section V briefly introduces a success story of our work, and Section VI concludes this paper.

## II. Related Work

In this section, we briefly review the related work on automated feature engineering and neural network controller.

### A. Automated Feature Engineering

Many automated feature engineering methods are based on domain knowledge. For example, the work presented in [8] automates feature engineering by constructing features from semantic knowledge bases. Similarly, [9] utilizes the explicitly captured relations and dependencies modeled in a knowledge graph to perform feature selection and construction. The knowledge-based methods imitate traditional manual feature engineering and relies heavily on domain knowledge.

Recently, the general framework of *expansion-reduction* is proposed for automated feature engineering. It first expands the feature space and then performs feature selection to reduce redundancy. FEADIS [1] randomly selects raw features and mathematical functions to generate new features. The Data Science Machine (DSM) [10] includes a Deep Feature Synthesis component, which synthesizes new features by recursively enumerating all possible combinations. DSM can generate a large number of features per entity. To reduce the size of the feature space, it performs feature selection via truncated SVD transformation. [3] adopts a similar approach and develops a system called One Button Machine for automating feature engineering in relational databases. Instead of utilizing transformation functions, AutoLearn [11] generates and selects new features by applying regularized regression models to feature pairs. Being data-driven, it requires no domain knowledge. However, learning a regression model on every feature pair is time-consuming. Also, as mentioned above, *expansion-reduction* based approaches cannot handle the feature space explosion problem.

Like the *expansion-reduction* approach, ExploreKit [2] performs all transformation functions on the whole dataset. It then chooses the most promising features by ranking the features through supervised meta-learning. This method also suffers from the feature space explosion problem and is limited by the effectiveness of meta-learning, which demands massive amount of training data. Due to the complex nature of this method, it does not allow for functional compositions. Another plausible work is *Learning Feature Engineering* (LFE) [6], which recommends the most promising transformation function for features via offline meta-learning. This method is fast in computation but is constrained to classification problems. Most importantly, LFE does not take into account high-order transformations and thus is not able to generate complex features.

There are also search-based feature engineering methods. For example, [12] employs genetic algorithm to determine a proper single transformation function for a given dataset. Cognito [13] introduces the notion of a tree-like exploration of transform space, and presents some heuristic search strategies such as depth-first traversal. [4] is a related work of Cognito, which is based on performance driven exploration of a transformation graph. It utilizes Q-learning to systematically and compactly enumerates the space of given options. It is the current state of the art and achieves better performance than other related approaches. However, it still suffers from the feature explosion problem since the number of features grows exponentially in the hierarchically structured transformation graph, especially at the bottom of the graph.

Deep Neural Network has the capability of performing complex feature transformation as well. However, deep learning methods require massive amount of training data and face overfitting problem when it comes to real-world small and unbalanced datasets. Most importantly, the learned implicit feature transformations lack interpretability. In contrast, NFS outputs human readable features as insights into the data via explicit transformation functions.

Our approach explores the most promising transformation rule for each raw feature, which is time-efficient and eliminates feature space explosion problem. The discovered transformation rule can be very complex since new features can be generated through a series of functional composition.

## B. Neural Network Controller

NAS [14] proposes a general framework to search for the architecture of a deep neural network, which employs a recurrent neural network controller to generate a string. Training the network specified by the string will result in an accuracy on a validation set. Using this accuracy as reward signal, NAS utilizes policy gradient to update the controller. Apart from searching for the neural network architectures, Searching for Activation Functions [15] employs a similar controller to search for activation function named Swish. A Hierarchical Model for Device Placement [16] uses hierarchical controller to search for optimal placement of computational graphs onto hardware devices. In this paper, we apply a neural controller to generate transformation rule (i.e. a series of transformation functions) for each raw feature.

## III. Proposed Approach

In this section, we propose and introduce Neural Feature Search (NFS), a novel neural architecture for automated feature engineering.

## A. Problem Formulation

Given a dataset $D = \langle F, y \rangle$ that consists of raw features $F = \{f_1, f_2, ..., f_n\}$ with a target vector $y$, we use $V_L^E(F, y)$ to denote the performance of a machine learning algorithm $L$ (e.g. Random Forest or Logistic Regression) constructed on $F$ and measured by an evaluation metric $E$ (e.g. F1-score or mean squared error).

In addition, we transform a raw feature set $F$ into $\hat{F}$ through a set of transformation functions $\{A_1, A_2, ..., A_n\}$. As described in [4], [6], there are two categories of mathematical transformation functions: unary functions that are performed on a single feature (e.g. $logarithm$ and $square$) and binary functions that are performed on a pair of features (e.g. $plus$ and $minus$). Note that transformations can be performed on three or more features, since it is equivalent to applying binary transformations recursively.

Generally, feature transformation should be able to perform the following two activities: (1) *generating* new features via transforming the raw features. Note that transformation functions can be composed to form a higher-order transformation (e.g. $BMI = \frac{weight}{height^2}$, a third-order transformation, is formed by transforming the feature $height$ through a $square$ function, then a $reciprocal$ function, and subsequently a $multiply$ function of feature $weight$). (2) *deleting* existing features through feature selection.

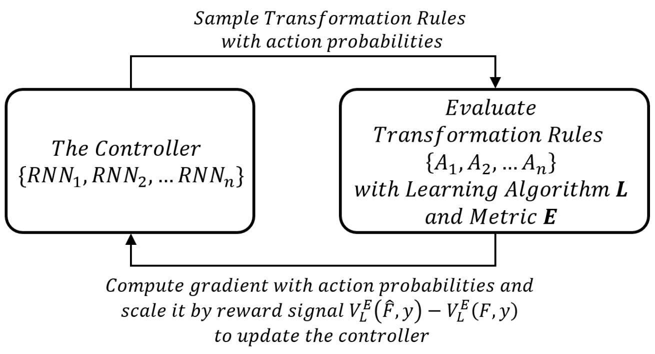

Sample Transformation Rules with action probabilities
The Controller $\{RNN_1, RNN_2, ... RNN_n\}$
Evaluate Transformation Rules $\{A_1, A_2, ... A_n\}$ with Learning Algorithm $\mathbf{L}$ and Metric $\mathbf{E}$
Compute gradient with action probabilities and scale it by reward signal $V_L^E(\hat{F}, y) - V_L^E(F, y)$ to update the controller

Fig. 1. An Overview of Neural Feature Search

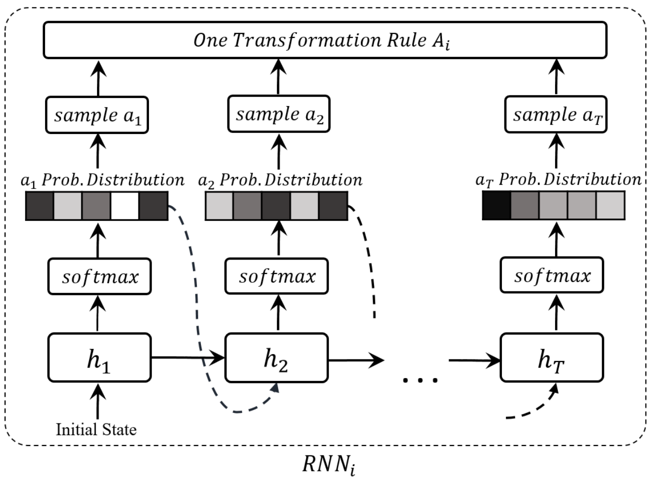
Formally, the goal of automated feature engineering is to search the optimal transformed feature set $F^*$ where $V_L^E(F^*, y)$ is maximized, $F^* = \arg\max_F V_L^E(F, y)$.

## B. An Overview of the Proposed Approach

In this paper, we propose Neural Feature Search (NFS), a neural architecture for automated feature engineering. We employ a Recurrent Neural Network (RNN) to generate a transformation rule $A$ (i.e., a series of transformation functions with the maximum order of $T$, such as $reciprocal(square(height)) \times weight$) for each raw feature within $T$ time steps. That is, rather than performing transformation on the whole dataset or on the sampled features, we aim to find the most promising transformation rule for every feature. One big advantage of our approach is that rather than adopting a hierarchical structure which is likely to introduce feature space explosion problem, our architecture performs the most promising transformation for each feature in parallel through an efficient search algorithm.

Figure 1 gives an overview of the NFS architecture. For datasets with $n$ raw features, we build $n$ RNNs as our controller to generate $n$ transformation rules $\{A_1, A_2, ..., A_n\}$ in parallel, where $A_i$ is related to raw feature $f_i$. Guided by the transformation rules, we transform the raw feature set $F$ into $\hat{F}$ and compute its performance $V_L^E(\hat{F}, y)$ with respect to a learning algorithm $L$ and an evaluation metric $E$. Next, we utilize policy gradient [17] to train our controller via reward signal $V_L^E(\hat{F}, y) - V_L^E(F, y)$, which represents the performance improvement of those transformation rules. Figure 2 illustrates how each RNN predicts a transformation rule. A transformation function is predicted at each time step and they together form a transformation rule. Every prediction is carried out by a softmax classifier and then fed into the next time step as input.

In the following subsection, we will first show how our transformation rules transform raw features. We then describe our RNN-based controller, which generates transformation rules. Finally, we show how the proposed NFS architecture



**Note:** Each RNN predicts transformation function $a_t$ at time step $t$. A series of transformations $[a_1, a_2, .., a_T]$ form a transformation rule. Every prediction is carried out by a softmax classifier and then fed into the next time step.

Fig. 2. The Generation of Transformation Rule by a RNN

**Algorithm 1** Dataset Transformation

**Input:** Dataset $D = \{f_1, f_2, ..., f_n\}$ with $n$ raw features; Transformation Rules $\mathbb{A} = \{A_1, A_2, ..., A_n\}$, where each $A_i = [a_1, a_2, ..., a_T]$;

**Output:** New Dataset $\hat{D}$ after the transformation;

```
1:  for each $f_i \in D$ do
2:      tailor $A_i$ to be $\hat{A}_i = [a_1, a_2, .., a_t]$, where $a_{t+1}$ is the first terminate transformation in $A_i$;
3:      if $t == 0$ then
4:          continue;
5:      else
6:          if $\exists a_i \in \hat{A}_i = delete$ then
7:              delete $f_i$ from $D$;
8:          else
9:              $\hat{f}_i = a_t\,(a_{t-1}...\,(a_2\,(a_1\,(f_i))))$;
10:             add $\hat{f}_i$ to $D$;
11:         end if
12:     end if
13: end for
```

can be effectively trained with policy gradient to maximize the expected performance of the learning algorithm.

### *C. Feature Reconstruction through High-order Transformations*

As mentioned, there are two types of transformations: unary and binary. Our approach focuses on finding the most promising transformation rule $A$ for each feature. However, binary transformations take two features as input and thus cannot be performed on a single feature. Therefore, we unify a binary transformation by converting it into a set of unary transformations for each individual feature $f_i$. For example, for the binary transformation $plus$, we convert it into a set of unary transformation $addition(f_i)$.

We have two special unary transformations: *delete* and *terminate*. *delete* removes the corresponding feature and *terminate* indicates a stop command for the current transformation. Such two transformations enable our architecture to eliminate features and stop at the most suitable order of transformation (i.e., the number of feature composition), respectively.

For every raw feature, we utilize a series of transformation functions $A_{1:T} = [a_1, a_2, ..., a_T]$ to represent a transformation rule, where the length of the series $T$ is the max order of transformations it can take. Each element in the series is a unary transformation or a unified binary transformation. For dataset with $n$ raw features, we have $n$ transformation rules, and the transformation process for each raw feature is independent of each other since we have already captured the correlation between features by unifying every binary transformation. As a result, we can perform transformation of each feature in parallel.

Guided by all transformation rules, NFS transforms the raw dataset as shown in Algorithm 1. In order to reduce the feature space, NFS first conducts feature selection on the raw dataset by selecting $\beta$ features according to the feature importance via random forest [18] if the number of features of the raw dataset is greater than $\beta$. Given a raw feature $f_i$ and its corresponding transformation rule $A_i$, NFS first tailors $A_i$ to be $\hat{A}_i$ by removing all actions after the first $terminate$ transformation ($\hat{A}_i = A_i$ if $terminate$ does not exist) (line 2). This enables NFS to automatically find the proper length of transformation rule $A$, which is the order of transformation. If the length of $\hat{A}_i$ is 0, which means that the first element of $A_i$ is $terminate$, then no feature is generated or eliminated (line 4). If $\hat{A}_i$ has length $T > 0$ and there exists an *delete* element in $\hat{A}_i$, we then delete the raw feature $f_i$ (line 7). Otherwise, we generate a new feature $\hat{f}_i = a_t\,(a_{t-1}...\,(a_2\,(a_1\,(f_i))))$ (line 9), where $t$ is the length of $\hat{A}_i$. Finally, we add $\hat{f}_i$ to the dataset (line 10).

NFS traverses and transforms every feature in parallel, which facilitates high-order transformations. We make each binary transformation to $n$ unary transformations and reduce the searching space from $(\binom{n}{2})^T = O(n^{2T})$ to $n \times n^T = O(n^{T+1})$, where $n$ is the number of raw features and $T$ is the max transformation order. Within the reduced searching space, our architecture explores the most promising transformation for each raw feature. Therefore, NFS does not suffer from the feature explosion problem and can capture high-order transformation meanwhile. Our architecture also supports feature selection because of the special action *delete*. Furthermore, the *terminate* function enables NFS to automatically stop at a certain time step, so transformation rules of variable lengths can be generated.

## D. Generate Transformations with a Neural Network Controller

In the architecture of NFS, we utilize a controller to generate $n$ transformation rules $\mathbb{A} = \{A_1, A_2, ..., A_n\}$. The controller consists of $n$ RNNs $\{RNN_1, RNN_2, ..., RNN_n\}$, where $n$ is the number of raw features. In other words, we build a separate recurrent neural network for each feature. Each RNN predicts a transformation rule $A_{1:T}$ for the corresponding raw feature within $T$ time steps, where $T$ is the max order of transformation. Every prediction is carried out by a softmax classifier and then fed into the next time step as input, as shown in Figure 2.

We utilize RNN in our architecture to generate transformation rules $\mathbb{A}$ because of its ability to support sequences of variable lengths. The controller has a fixed number of parameters regardless of the max transformation order $T$. As a result, NFS is efficient because an increase of $T$ only results in linearly increasing time steps, in contrast to the exponentially increasing feature space in *expansion-reduction*.

## E. Training with Policy Gradient

Policy gradient [17] has achieved tremendous success in many application areas [19], [20]. It aims to learn an explicit policy for an agent such that it behaves optimally according to a reward function. In our work, at the time step $t$ and the state $s$, the network controller chooses a transformation function according to its current policy $\pi(a_t|s_t)$, and then the state changes to $s_{t+1}$ according to dynamics $p(s_{t+1}|s_t, a_t)$ and the controller receives a reward $r(s_t, a_t)$. Let $R_t = \sum_{t'=t}^{T} \gamma^{\,t'-t} r(s_{t'}, a_{t'})$ denote a $\gamma$-discounted cumulative return from $t$. The goal of policy gradient is to maximize the expected return $J(\theta) = \mathbb{E}[R_0]$. We replace the $\gamma$-discounted cumulative return by $\lambda$-$return$ [21] to mix long-term and short-term returns when training our controller.

More specifically, for $n$ transformation rules, there are $n \times T$ transformations to predict. We define the state $s_t$ in reinforcement learning as $\mathbb{A}_{1:t}, t = 1, 2, ..., n \times T$, where $s_0$ is the initial state and $\mathbb{A}$ is the universal set of $n$ transformation vectors. On each state $s_t$, we can achieve a cross-validation score $V_t$ given a predefined learning algorithm $L$ and an evaluation metric $E$. Then we compute the reward of each transformation as follows: for taking transformation $a_t$, the improvement $R_t$ is the score $V_t$ of state $s_t$ minus $V_{t-1}$ of state $s_{t-1}$, which means the gain of performing $a_t$ intuitively. Then, we utilize the $\lambda$-$return$ $G_t^{\lambda}$ that combines all $k$-$step$ returns $G_t^{(k)}$ as the final reward signal for transformation $a_t$:

$$R_t = V_t - V_{t-1} \tag{1}$$

$$G_t^{(k)} = R_t + \gamma R_{t+1} + ... + \gamma^k R_{t+k} \tag{2}$$

$$G_t^{\lambda} = (1-\lambda) \sum_{k=1}^{n \times T} \lambda^{k-1} G_t^{(k)} \tag{3}$$

The $k$-$step$ return $G_t^{(k)}$ looks $k$ steps forward into the future from current time step $t$, $G_t^{(0)}$ equals to $R_t$ at time step $t$ and $G_t^{(\infty)} = \sum_{t'=t}^{\infty} \gamma^{\,t'-t} R_{t'}$ traverses the future. When $n$ is small, short-term return could lead to low variance but high bias as the function is shortsighted. When $n$ is large, the long-term return could lead to low bias but high variance. We employ $\lambda$-$return$ $G_t^{\lambda}$ rather than the original $R_t$ because of its superiority in balancing between bias and variance.

To find the optimal transformation vectors, we ask our controller to maximize its expected return, represented by $J(\theta)$, where $\theta$ is parameters of the controller:

$$J(\theta) = \mathbb{E}_{P(a_{1:n \times T};\theta)} \left[ \sum_{t=1}^{n \times T} G_t^{\lambda} \right] \tag{4}$$

We utilize policy gradient to iteratively update $\theta$. Without loss of generality, we utilize the REINFORCE rule [22] and Monte Carlo simulation [23] to obtain an empirical approximation:

$$\nabla_\theta J(\theta) \approx \frac{1}{m} \sum_{k=1}^{m} \sum_{t=1}^{n \times T} \nabla_\theta \log P(a_t|a_{1:t-1};\theta) \, G^{\lambda}_{t\,[k]}) \tag{5}$$

where $m$ is the number of different feature engineering paradigms that the controller samples per epoch and $G^{\lambda}_{t\,[k]}$ is the cross-validation score that the $k$-$th$ sample achieves.

## IV. Experiments

In this section, we conduct extensive experiments in order to answer the following research questions:

- **RQ1:** How effective is the proposed NFS approach?
- **RQ2:** How effective are the high-order transformations?
- **RQ3:** Can NFS work with different learning algorithms?
- **RQ4:** Is NFS robust to hyperparameters?

## A. Experimental Settings

In our experiments, we utilize the following 12 transformation functions: 10 mathematical functions (including *logarithm*, *square*, *square-root*, *min-max-normalization*, *reciprocal*, *addition*, *subtraction*, *multiplication*, *division*, *modulo operation*) and 2 special functions $delete$ and $terminate$.

In our experiments, we set the parameter $\beta$ to 50. Each RNN is implemented as a one-layer LSTM. We utilize Monte-Carlo simulation to estimate the gradient, and the sample size of every epoch $m$ is 32. We set the discount factor $\gamma = 0.99$ and choose the Adam optimizer [24] to train the controller.

For RQ1, we set the maximum transformation order to 5, which is the maximum time steps of RNN. Without loss of generality, we set $\lambda = 0.4$, which is the parameter of $\lambda$-$return$. In RQ2, we evaluate NFS with different numbers of the maximum transformation order ($T$). In RQ3, we evaluate NFS with different machine learners. In RQ4, we evaluate the impact of different hyperparameters on the accuracy of NFS.

TABLE I
COMPARISON BETWEEN NFS AND THE EXISTING AUTOMATED FEATURE ENGINEERING METHODS

| Dataset | Source | C\R | Instances\ Features | Random | Random$_{NFS}$ | Exp-Red | Trans-Graph | NFS |
|---|---|---|---|---|---|---|---|---|
| Higgs Boson | UCIrvine | C | 50000\28 | 0.699 | 0.723 | 0.682 | 0.729 | **0.731** |
| Amazon Employee | Kaggle | C | 32769\9 | 0.740 | **0.945** | 0.744 | 0.806 | **0.945** |
| PimaIndian | UCIrvine | C | 768\8 | 0.709 | 0.772 | 0.712 | 0.756 | **0.805** |
| SpectF | UCIrvine | C | 267\44 | 0.748 | 0.801 | 0.790 | 0.788 | **0.876** |
| SVMGuide3 | LibSVM | C | 1243\21 | 0.753 | 0.834 | 0.711 | 0.776 | **0.865** |
| German Credit | UCIrvine | C | 1001\24 | 0.655 | 0.745 | 0.680 | 0.724 | **0.805** |
| Bikeshare DC | Kaggle | R | 10886\11 | 0.381 | 0.980 | 0.693 | 0.798 | **0.990** |
| Housing Boston | UCIrvine | R | 506\13 | 0.637 | 0.658 | 0.621 | 0.680 | **0.686** |
| Airfoil | UCIrvine | R | 1503\5 | 0.753 | 0.771 | 0.771 | **0.801** | 0.796 |
| AP-omentum-ovary | OpenML | C | 275\10936 | 0.710 | 0.831 | 0.725 | 0.820 | **0.864** |
| Lymphography | UCIrvine | C | 148\18 | 0.680 | 0.887 | 0.727 | 0.895 | **0.929** |
| Ionoshpere | UCIrvine | C | 351\34 | 0.934 | 0.923 | 0.939 | 0.941 | **0.969** |
| Openml_618 | OpenML | R | 1000\50 | 0.428 | 0.403 | 0.411 | 0.587 | **0.640** |
| Openml_589 | OpenML | R | 1000\25 | 0.571 | 0.511 | 0.650 | 0.689 | **0.754** |
| Openml_616 | OpenML | R | 500\50 | 0.343 | 0.284 | 0.450 | 0.559 | **0.673** |
| Openml_607 | OpenML | R | 1000\50 | 0.411 | 0.366 | 0.590 | 0.647 | **0.688** |
| Openml_620 | OpenML | R | 1000\25 | 0.524 | 0.471 | 0.533 | 0.683 | **0.732** |
| Openml_637 | OpenML | R | 500\50 | 0.313 | 0.248 | 0.581 | 0.585 | **0.634** |
| Openml_586 | OpenML | R | 1000\25 | 0.549 | 0.495 | 0.598 | 0.704 | **0.780** |
| Credit Default | UCIrvine | C | 30000\25 | 0.766 | 0.798 | 0.802 | **0.831** | 0.799 |
| Messidor_features | UCIrvine | C | 1150\19 | 0.655 | 0.743 | 0.703 | 0.752 | **0.762** |
| Wine Quality Red | UCIrvine | C | 999\12 | 0.380 | 0.692 | 0.344 | 0.387 | **0.708** |
| Wine Quality White | UCIrvine | C | 4900\12 | 0.678 | 0.689 | 0.654 | **0.722** | 0.707 |
| SpamBase | UCIrvine | C | 4601\57 | 0.937 | 0.954 | 0.951 | **0.961** | 0.955 |

### B. *Effectiveness of NFS (RQ1)*

*1) Design:* To evaluate the effectiveness of NFS, we compare it with the state-of-the-art and bseline methods, which are described as follows:

- **Random** is a baseline method, which randomly applies a transform function to a random feature and adds the result to the original dataset. This is the same Random experiment described in [4].
- **Random$_{\text{NFS}}$** is a baseline method, which uses random selection under NFS architecture to generate features instead of policy gradient. Although this baseline seems naive, it is actually very hard to surpass [25]. We sample the same number of actions as in the Monte Carlo estimator by using uniformly distributed action probabilities in each training epoch and report the best result achieved.
- **Expansion-Reduction** is a state-of-the-art approach. All transformations are applied separately and added to the raw features, followed by a feature selection routine. In this experiment, we compare with Data Science Machine (DSM) [10], which is a well-known *expansion-reduction* method.
- **Transformation Graph** [4] builds a transformation graph and employs Q-learning to search. The transformation graph is a directed acyclic graph in which each node corresponds to the raw dataset or a derived dataset through the transformation path. To generate a new dataset node, they either perform transformation on the whole dataset or merge two transformed nodes together.
- **LFE** [6] recommends the most promising transformation for each feature. At the core of LFE, there is a set of Multi-Layer Perceptron (MLP) classifiers, each corresponding to a transformation. Note that LFE is limited to the classification problems only.

In our experiments, we utilize the same datasets used for evaluating the related methods, which are publicly available at OpenML[1], UCI repository[2], Kaggle[3], and LibSVM[4]. In particular, Table I shows the 24 datasets used for evaluating Expansion-Reduction (Exp-Red) and Transformation Graph (Trans-Graph). There are 14 classification datasets and 10 regression datasets that are collected from various sources and have various numbers of features (5 to 10936) and instances (148 to 50000). For all datasets in Table I, we use Random Forest as the learning algorithm ($L$). For evaluation metrics ($E$), we use the metric *(1 - (relative absolute error))* [26] for regression (R) problems:

$$1 - rae = 1 - \frac{\sum |\hat{y} - y|}{\sum |\bar{y} - y|} \tag{6}$$

where $1 - rae$ means the metric *(1 - (relative absolute error))*, $\hat{y}$ is the model prediction, $y$ is the actual target and $\bar{y}$ is the mean of $y$. We use F1-score (the harmonic mean of precision and recall) for classification (C) problems. Both metrics are also used for evaluating the related work [4]. In addition, following the related work, stratified 5-fold cross-validation is adopted to obtain the results. For comparison, we use the results of Expansion-Reduction (Exp-Red) and Transformation Graph (Trans-Graph) on the same datasets as reported in [4].

Table II presents another group of datasets for comparing with related methods including LFE[5]. All datasets are classification problems since LFE is restricted to classification. We utilize the Random Forest as $L$ and the F1-score of 10-fold cross-validation as $E$. These settings are also used for evaluating LFE. For comparison, we use the results reported in the LFE paper [6] on the same datasets.

*2) Results:* Table I shows the comparison results between NFS and Transformation Graph, Expansion-Reduction, and two baseline methods (Random and Random$_{\text{NFS}}$). The results show that our neural architecture achieves the best performance in all but four cases. On these datasets, NFS exceeds the performance of Random, Random$_{\text{NFS}}$, Expansion-Reduction and Transformation Graph. It is also noteworthy that applying random search in our NFS architecture outperforms purely random generation of new features since it has the advantage of reducing searching space.

Table II shows the comparison between NFS and LFE on the datasets used in [6]. The experimental results show that NFS performs better than the related methods on these datasets too. More specifically, NFS outperforms LFE on all datasets.

The experimental results also demonstrate that NFS can handle different numbers of instances and features. For example, on a small dataset such as *Wine Quality Red* (with size $999 \times 12$), NFS achieves an average improvement of $82.95\%$ compared with Transformation Graph. On big datasets such as *Amazon Employee* (32769 instances) and *lymph* (10936 features), NFS also achieves an improvement with an average of $17.25\%$ and $30.38\%$ over Transformation Graph and LFE, respectively.

[1]https://www.openml.org/

[2]https://archive.ics.uci.edu/ml/index.php

[3]https://www.kaggle.com/

[4]https://www.csie.ntu.edu.tw/~cjlin/libsvm/

[5]We omit a few datasets that could no longer be found online.

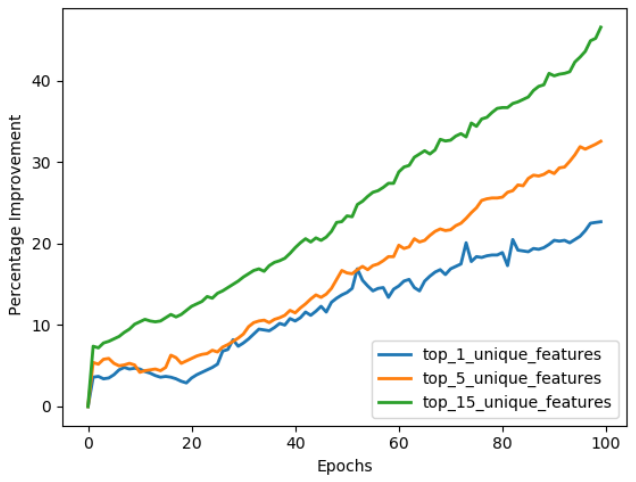


Fig. 3. Performance improvement of NFS over Random$_{\text{NFS}}$ on different numbers of epoch. We plot the average across all datasets in Table 1

*3) Efficiency of Policy Gradient:* Under NFS architecture, instead of policy gradient, one can utilize random search to find the best feature transformations. Although this baseline seems simple, it is often difficult to surpass [25]. In Figure 3, we show the performance improvement of policy gradient over random search under the NFS architecture (i.e., the Random$_{\text{NFS}}$ method) during the training process. The performance improvement is defined as the improvement of F1-score or *(1 - (relative absolute error))* between the model trained with the top $k$ features generated by policy gradient and the model trained with the top $k$ features generated by random search. To evaluate the effectiveness of the proposed model, we select different $k$ settings ($k = 1, 5, 15$). Figure 3 shows that the resulting three lines all exhibit a tendency of steady increase, which illustrates the effectiveness of the training. The results also confirm that our controller always finds better features than random search.

In summary, the experimental results in this subsection show that the proposed automated feature engineering architecture NFS is effective and outperforms the existing state-of-the-art and the baseline methods.

### C. Effectiveness of High-Order Transformation (RQ2)

*1) Design:* This RQ evaluates the ability of NFS to perform complex high-order transformation. We choose the max transformation order $T$ from 1 to 10. Note that $T = 0$ means the base dataset without feature transformation and $T = 5$ is the default setting in RQ1. For classification (C) datasets, we report the relative improvement in F1-score. For regression (R) datasets, we report the relative improvement using the metric *(1 - (relative absolute error))*.

TABLE II
COMPARISON BETWEEN NFS, LFE, AND OTHER RELATED METHODS

| Dataset | Source | Instances\Features | Random | Exp-Red | LFE | Random$_{NFS}$ | NFS |
|---|---|---|---|---|---|---|---|
| AP-omentum-lung | OpenML | 203\10936 | 0.903 | 0.925 | 0.929 | 0.947 | **0.981** |
| AP-omentum-ovary | OpenML | 275\10936 | 0.714 | 0.801 | 0.811 | 0.833 | **0.873** |
| credit-a | UCIrvine | 690\6 | 0.657 | 0.521 | 0.771 | 0.770 | **0.803** |
| diabetes | UCIrvine | 768\8 | 0.729 | 0.737 | 0.762 | 0.760 | **0.786** |
| fertility | UCIrvine | 100\9 | 0.846 | 0.861 | 0.873 | 0.903 | **0.913** |
| gisette | UCIrvine | 2100\5000 | 0.871 | 0.741 | 0.942 | 0.948 | **0.959** |
| hepatitis | UCIrvine | 155\6 | 0.751 | 0.753 | 0.831 | 0.821 | **0.905** |
| higgs-boson-subset | UCIrvine | 50000\28 | 0.660 | 0.661 | 0.68 | 0.824 | **0.827** |
| ionosphere | UCIrvine | 351\34 | 0.899 | 0.912 | 0.932 | 0.941 | **0.972** |
| labor | UCIrvine | 57\8 | 0.877 | 0.855 | 0.896 | 0.853 | **0.960** |
| lymph | OpenML | 138\10936 | 0.594 | 0.534 | 0.757 | 0.943 | **0.987** |
| madelon | UCIrvine | 780\500 | 0.602 | 0.585 | 0.617 | 0.697 | **0.836** |
| megawatt1 | UCIrvine | 253\37 | 0.865 | 0.882 | 0.894 | 0.897 | **0.933** |
| pima-indians-subset | UCIrvine | 768\8 | 0.729 | 0.751 | 0.745 | 0.760 | **0.790** |
| secom | UCIrvine | 470\590 | 0.911 | 0.913 | 0.918 | 0.931 | **0.934** |
| sonar | UCIrvine | 208\60 | 0.723 | 0.468 | 0.801 | 0.709 | **0.839** |
| spambase | UCIrvine | 4601\57 | 0.911 | 0.39 | 0.947 | 0.938 | **0.948** |

*2) Results:* Figure 4 shows experimental results on four randomly sampled datasets, which have different instances-features combinations and cover both classification and regression problems. When $T$ is small, the performance of our NFS is basically the same as that of the baselines. However, the performance increases stably as we increase the max transformation order $T$. As illustrated in Figure 4, when the transformation order is as high as 10, NFS can still achieve some improvements. The results confirm the effectiveness of the high-order transformation.

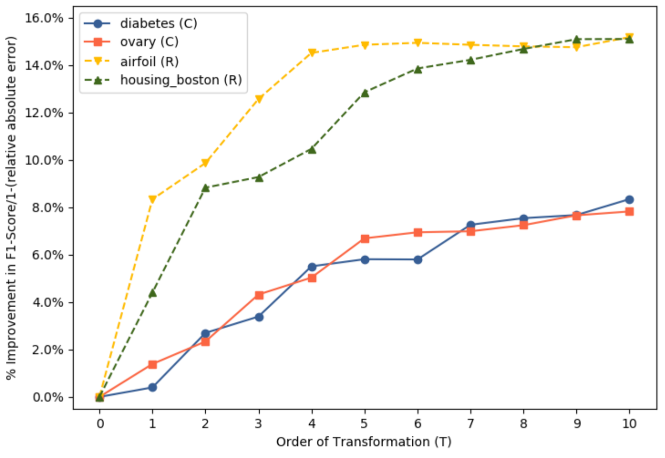


Fig. 4. Effect of High-order Transformations

## D. Different Learning Algorithms (RQ3)

*1) Design:* NFS is designed to be independent of machine learning algorithm $L$. In RQ1 and RQ2, we use Random Forest as the learning algorithm $L$. In this RQ, we evaluate whether NFS is still effective when a different learner is used. We utilize lasso regression [27] (for regression) and logistic regression [28] (for classification) as $L$ in this experiment. All other experimental settings are the same as those in RQ1.

*2) Results:* We have conducted the experiments of evaluating NFS equipped with other learners on the datasets shown in Table I and Table II, and the related experimental results demonstrate that, for classification problems, NFS with logistic regression outperforms the pure logistic regression, which is evaluated on the raw datasets without any feature transformation. For regression problems, NFS with lasso regression outperforms the pure lasso regression, which is evaluated on the raw datasets without any feature transformation. These results confirm that the effectiveness of NFS is independent of machine learners.

## E. Robustness (RQ4)

*1) Design:* In this subsection, we evaluate whether NFS is sensitive to different hyperparameters, i.e. the sample size of every training epoch and the $\lambda$ value in $\lambda$-$return$. We perform the experiments on the same datasets used in RQ2.

*2) Results:* Table III shows the experimental results on four randomly selected datasets that represent both classification

TABLE III
COMPARATIVE RESULTS OF NFS WITH DIFFERENT PARAMETER SETTINGS OF $\lambda$

| Dataset | $\lambda=0.2$ | $\lambda=0.25$ | $\lambda=0.3$ | $\lambda=0.35$ | $\lambda=0.4$ | $\lambda=0.45$ | $\lambda=0.5$ | $\lambda=0.55$ | $\lambda=0.6$ |
|---|---|---|---|---|---|---|---|---|---|
| Airfoil | 0.792 | 0.798 | 0.796 | 0.801 | 0.796 | 0.797 | 0.800 | 0.795 | 0.799 |
| Housing Boston | 0.690 | 0.690 | 0.694 | 0.691 | 0.686 | 0.686 | 0.685 | 0.686 | 0.687 |
| AP-omentum-ovary | 0.873 | 0.880 | 0.873 | 0.873 | 0.873 | 0.874 | 0.873 | 0.873 | 0.873 |
| diabetes | 0.785 | 0.784 | 0.785 | 0.786 | 0.786 | 0.785 | 0.784 | 0.789 | 0.786 |

TABLE IV
COMPARATIVE RESULTS OF NFS WITH DIFFERENT PARAMETER SETTINGS OF $m$

| Dataset | $m=1$ | $m=2$ | $m=4$ | $m=8$ | $m=16$ | $m=32$ | $m=64$ |
|---|---|---|---|---|---|---|---|
| Airfoil | 0.792 | 0.796 | 0.792 | 0.794 | 0.797 | 0.796 | 0.801 |
| Housing Boston | 0.678 | 0.688 | 0.688 | 0.687 | 0.692 | 0.686 | 0.693 |
| AP-omentum-ovary | 0.866 | 0.862 | 0.863 | 0.877 | 0.866 | 0.873 | 0.873 |
| diabetes | 0.776 | 0.775 | 0.785 | 0.777 | 0.784 | 0.786 | 0.784 |

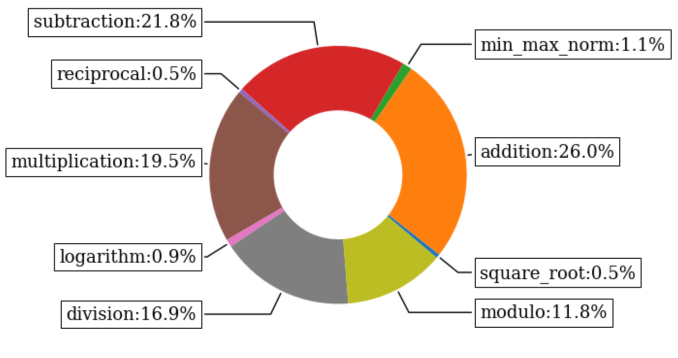


Fig. 5. Statistics of Effective Transformations

and regression problems. The results show that our NFS method is robust to different settings of $\lambda$, but relatively-small $\lambda$ ($\lambda < 0.5$) works slightly better on most cases of the studied four datasets. Relatively-small $\lambda$ implies preference for shorter-term reward with low variance but higher bias. Since our policy gradient is trained within 100 epochs for time performance, variance plays a more important role than bias from our perspective: relatively-large variance will make the training loss hard to converge during the first few epochs and the advantage of low bias will only reveal after enough training epochs. This helps explain why a relatively-small $\lambda$ leads to better performance in our experiments. With respect to the sample size of each training epoch, the performance of our NFS remains stable across different settings, as shown in Table IV.

## F. Discussion of the Results

We perform statistical analysis by calculating the frequency of every effective transformation in NFS across all datasets. Figure 5 shows an interesting observation that most of the transformations are binary operations, i.e., *addition*, *subtraction*, *multiplication*, *division*, and *modulo operation* between two features. Unary operations such as *logarithm*, *min-max-normalization*, *square*, *square root*, *reciprocal* are less common. Many of the transformations are not very intuitive. However, we still find some explainable/meaningful feature transformations that could provide some insights into the datasets. For example, in dataset $Airfoil$, the *Scaled Sound Pressure level* is related to the ratio between *Chord length* and *Frequency* (i.e., *Chord length*/*Frequency*). In dataset *Wine Quality Red*, the quality of wine is related to the following transformed feature:

$$\sqrt{(\mathit{Chlorides} \times \mathit{Density}) - \mathit{pH\ Value}} \tag{7}$$

To our best knowledge, NFS is the first automated feature engineering framework that can solve the feature explosion problem and support high-order transformations, in the meantime being time-efficient and parameter-insensitive. In addition, NFS is extendable as users can specify and add more transformations.

There are some limitations for NFS as well. Currently, it only applies transformations to numerical features. It could also be affected by randomness due to sampling, so it does not guarantee to always find the best transformation. We will address these limitations in our future work.

## V. SUCCESS STORY

It is also worth mentioning that NFS demonstrated its efficiency in the NeurIPS AutoML Challenge: AutoML for

Lifelong Machine Learning [6]. The challenge was to develop an AutoML system (including feature engineering, model selection and hyperparameter tuning) for large-scale lifelong learning such as customer relationship management, online recommendation, sentiment analysis, fraud detection, spam filtering, transportation monitoring, econometrics, patient monitoring, climate monitoring, manufacturing and so on. There are many difficulties in tackling this challenge: very large dataset ($\sim$10 Million), various feature types (continuous, binary, ordinal, categorical, multi-value categorical, temporal), and restricted time and computing resources (within half an hour on a 4 Cores / 16 GB Memory computer), NFS was able to discover valuable new features and largely improved the base performance achieved by XGBoost [29] and LightGBM [30]. More specifically, we applied NFS to those public datasets used in the challenge[7] and achieved performance improvement in accuracy over the base models. Finally, we obtained an outstanding score among thousands of participants of the challenge and is ranked among the very top $k$.

## VI. Conclusion and Future Work

In this paper, we propose NFS, a novel neural architecture for automated feature engineering. NFS is able to explore a feature space and automates the process of feature construction and selection through a RNN-based controller. The controller is trained with reinforcement learning to maximize the expected performance of the machine learning algorithm. Extensive experiments on public datasets demonstrate that our neural architecture is effective and outperforms the existing state-of-the-art automated feature engineering methods.

In the future, we will consider more complex transformation functions. We would also like to explore model selection and hyperparameter tuning methods to further improve the effectiveness of the proposed approach. We also plan to further enhance our proposed approach using the effective idea of programming by optimization [31], which has shown success in solving computationally hard problems including Boolean satisfiability [32] and minimum vertex cover [33].

## References

[1] O. Dor and Y. Reich, "Strengthening learning algorithms by feature discovery," *Information Sciences*, vol. 189, pp. 176–190, 2012.

[2] G. Katz, E. C. R. Shin, and D. Song, "ExploreKit: Automatic feature generation and selection," in *Proceedings of ICDM 2016*, 2016, pp. 979–984.

[3] H. T. Lam, J. Thiebaut, M. Sinn, B. Chen, T. Mai, and O. Alkan, "One button machine for automating feature engineering in relational databases," *CoRR*, vol. abs/1706.00327, 2017.

[4] U. Khurana, H. Samulowitz, and D. S. Turaga, "Feature engineering for predictive modeling using reinforcement learning," in *Proceedings of AAAI 2018*, 2018, pp. 3407–3414.

[5] C. J. C. H. Watkins and P. Dayan, "Technical note Q-Learning," *Machine Learning*, vol. 8, pp. 279–292, 1992.

[6] F. Nargesian, H. Samulowitz, U. Khurana, E. B. Khalil, and D. S. Turaga, "Learning feature engineering for classification," in *Proceedings of IJCAI 2017*, 2017, pp. 2529–2535.

[7] G. Dong and H. Liu, Eds., *Feature Engineering for Machine Learning and Data Analytics*. CRC Press, 2018.

[6] https://www.4paradigm.com/competition/nips2018

[7] https://competitions.codalab.org/competitions/20203#participate-get_data

[8] W. Cheng, G. Kasneci, T. Graepel, D. H. Stern, and R. Herbrich, "Automated feature generation from structured knowledge," in *Proceedings of CIKM 2011*, 2011, pp. 1395–1404.

[9] M. Atzmueller and E. Sternberg, "Mixed-initiative feature engineering using knowledge graphs," in *Proceedings of K-CAP 2017*, 2017, pp. 45:1–45:4.

[10] J. M. Kanter and K. Veeramachaneni, "Deep feature synthesis: Towards automating data science endeavors," in *Proceedings of DSAA 2015*, 2015, pp. 1–10.

[11] A. Kaul, S. Maheshwary, and V. Pudi, "AutoLearn - automated feature generation and selection," in *Proceedings of ICDM 2017*, 2017, pp. 217–226.

[12] M. G. Smith and L. Bull, "Feature construction and selection using genetic programming and a genetic algorithm," in *Proceedings of EuroGP 2003*, 2003, pp. 229–237.

[13] U. Khurana, D. S. Turaga, H. Samulowitz, and S. Parthasrathy, "Cognito: Automated feature engineering for supervised learning," in *Proceedings of ICDM Workshops 2016*, 2016, pp. 1304–1307.

[14] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *Proceedings of ICLR 2017*, 2017.

[15] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," in *Proceedings of ICLR 2018 Workshop Track*, 2018.

[16] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, "A hierarchical model for device placement," in *Proceedings of ICLR 2018*, 2018.

[17] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proceedings of NIPS 1999*, 1999, pp. 1057–1063.

[18] T. K. Ho, "The random subspace method for constructing decision forests," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 8, pp. 832–844, 1998.

[19] J. Peters and S. Schaal, "Reinforcement learning of motor skills with policy gradients," *Neural Networks*, vol. 21, no. 4, pp. 682–697, 2008.

[20] 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. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–503, 2016.

[21] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[22] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, pp. 229–256, 1992.

[23] C. P. Robert and G. Casella, *Monte Carlo Statistical Methods*, ser. Springer Texts in Statistics. Springer, 2004.

[24] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of ICLR 2015*, 2015.

[25] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, pp. 281–305, 2012.

[26] M. V. Shcherbakov, A. Brebels, N. L. Shcherbakova, A. P. Tyukov, T. A. Janovsky, and V. A. Kamaev, "A survey of forecast error measures," *World Applied Sciences Journal*, vol. 24, no. 2013, pp. 171–176, 2013.

[27] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996.

[28] D. W. Hosmer and S. Lemeshow, *Applied Logistic Regression, Second Edition*. Wiley, 2000.

[29] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of KDD 2016*, 2016, pp. 785–794.

[30] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T. Liu, "LightGBM: A highly efficient gradient boosting decision tree," in *Proceedings of NIPS 2017*, 2017, pp. 3146–3154.

[31] H. H. Hoos, "Programming by optimization," *Communications of the ACM*, vol. 55, no. 2, pp. 70–80, 2012.

[32] A. R. KhudaBukhsh, L. Xu, H. H. Hoos, and K. Leyton-Brown, "SATenstein: Automatically building local search SAT solvers from components," *Artificial Intelligence*, vol. 232, pp. 20–42, 2016.

[33] C. Luo, H. H. Hoos, S. Cai, Q. Lin, H. Zhang, and D. Zhang, "Local search with efficient automatic configuration for minimum vertex cover," in *Proceedings of IJCAI 2019*, 2019, pp. 1297–1304.