2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network

Huong Ha
The University of Newcastle
Callaghan, Australia
huong.ha@uon.edu.au

Hongyu Zhang
The University of Newcastle
Callaghan, Australia
hongyu.zhang@newcastle.edu.au

***Abstract*—Many software systems provide users with a set of configuration options and different configurations may lead to different runtime performance of the system. As the combination of configurations could be exponential, it is difficult to exhaustively deploy and measure system performance under all possible configurations. Recently, several learning methods have been proposed to build a performance prediction model based on performance data collected from a small sample of configurations, and then use the model to predict system performance under a new configuration. In this paper, we propose a novel approach to model highly configurable software system using a deep feedforward neural network (FNN) combined with a sparsity regularization technique, e.g. the L_1 regularization. Besides, we also design a practical search strategy for automatically tuning the network hyperparameters efficiently. Our method, called *DeepPerf*, can predict performance values of highly configurable software systems with binary and/or numeric configuration options at much higher prediction accuracy with less training data than the state-of-the art approaches. Experimental results on eleven public real-world datasets confirm the effectiveness of our approach.**

***Index Terms*—software performance prediction, deep sparse feedforward neural network, highly configurable systems, sparsity regularization**

I. Introduction

Many large and complex software systems are highly configurable, i.e., they provide a set of configuration options for users to select. These options allow users to customize the system to meet their specific requirements, hence, improving the usability and reusability of the system. Different configurations may lead to different quality attributes (non-functional properties). Among the quality attributes, performance (such as response time or throughput) is one of the most important quality attributes as it directly affects user experience. It is necessary to understand the performance of a system under a certain configuration, before the system is actually configured and deployed. This helps users make rational decisions in configurations and reduce performance testing cost. However, we cannot exhaustively deploy and measure system performance under all possible configurations as even a small-scale configurable system already results in an exponential number of configurations.

In recent years, researchers have proposed to measure the performance of a system with only a limited set of configurations (*sample*), build a performance prediction model, and then use the model to predict the performance of the system under new configurations (*population*) [19, 21, 37, 40, 42, 43, 48, 55]. In this way, performance can be predicted before a variant of the system is configured and deployed. The difficulty here is to predict system performance with high accuracy while utilizing a small sample. As it takes time and effort to configure the system and collect performance data, it is desirable that the sample size is kept minimum.

The challenge for building a performance prediction model is in the interactions between configuration options (*features*), i.e. a particular combination of features causes an unexpected behaviour on the system performance while their individual presences do not [8, 34, 42]. To address this challenge, an approach, namely *SPLConqueror*, aims to learn the influences of individual configuration options and their interactions from the differences among the measurements of the sample [41–43]. Several sampling heuristics and experimental designs for configuration options are combined with the suggested learning method to achieve good prediction accuracy. A strong point of *SPLConqueror* is that it can derive performance-influence models from binary-numeric configurable software systems, i.e. using the prediction models, users can understand how individual features and their interactions influence the system performance. A disadvantage of this method is related to its flexibility as it might not always be possible to measure the performance values of configurations that meet certain pre-defined coverage criteria. Besides, *SPLConqueror* usually requires more sample than other approaches as their focus is to make the influences of configuration options and their interactions explicit [21, 37, 41].

Another approach is to consider the performance prediction problem as a non-linear regression problem and apply a statistical learning method, e.g. the *Classification And Regression Trees (CART)* technique, to find this non-linear model [19]. Recently, it was further extended by combining with various resampling and machine learning hyperparameter tuning techniques and became a more data-efficient performance learning algorithm (*DECART*) [21]. That is, compared to *CART*, *DECART* uses less measurement effort to learn and validate a performance-prediction model [21]. However, at present, both *CART* and *DECART* can only predict configurable software systems with binary configuration options.

Lately, Zhang et al. [55] addressed this challenge by for-

1558-1225/19/$31.00 ©2019 Crown
DOI 10.1109/ICSE.2019.00113

mulating the software performance function as a Boolean function. Using Fourier transform of the Boolean function, the task of estimating the performance function becomes estimating its associated Fourier coefficients from a small sample. Although the algorithm can derive a sample size that guarantees a theoretical boundary of the prediction accuracy, the size of sample required to achieve a desired accuracy is still very large (sometimes even more than the whole population of the system), especially for a relatively small system [55]. Similar to *CART*/*DECART*, this approach works only on binary configurable software systems.

Inspired by the strengths and weaknesses of all the state-of-the-art methods, in this paper, we aim to derive an approach that can model all types of configurable software systems (i.e. binary and binary-numeric systems) and predict system performance at a high accuracy using a small sample. Similar to *CART*/*DECART*, to address the problem of feature interaction, we also consider the software performance prediction problem as a non-linear regression problem, i.e. the performance is a non-linear function of the configuration options. However, in our work, we suggest to use a deep feedforward neural network (FNN) with non-linear hidden layers to approximate this non-linear performance function. A deep FNN is a multilayer stack of computational units (neurons), with the first layer taking the input, the last layer producing the output and the middle layers (hidden layers) connecting the input and the output layer [18]. The idea of approximating performance function by a deep FNN is possible, as it has been shown that FNNs with hidden layers provide a universal framework, i.e. an FNN with a linear output layer and at least one hidden layer with some specific activation functions and sufficient number of neurons can approximate various function classes, from Boolean functions [4, 46] to continuous real-valued functions [5, 11, 15, 22, 23, 28].

The first challenge when using deep FNNs to model software performance is that a deep neural network usually requires a lot of training data to achieve high prediction accuracy, however, for the software performance prediction problem, we have very limited number of measurements. To address this problem, we need to incorporate some prior knowledge about software performance into the network architecture. A good way to do this is to tell the network how the parameters look like, i.e. whether they follow any particular distribution or have any special properties. For configurable software systems, it has been observed that the software performance functions are usually very sparse (i.e. only a small number of configurations and their interactions have significant impact on system performance) [24, 27, 40, 42]. Based on this observation, to model software performance, we suggest to use a deep sparse FNN. To construct a deep sparse FNN, we will combine a normal deep FNN with a sparsity regularization technique, e.g. the L_1 regularization [49].

The second challenge with using deep sparse FNNs is that they require many hyperparameters (e.g. the number of layers, number of neurons, regularization hyperparameters, etc). These hyperparameters need to be tuned optimally so that the network achieves high prediction accuracy. In practice, these hyperparameters can be optimized either manually by human experts or automatically by some tuning methods. As it is not always possible to find an expert to tune hyperparameters every time we need to predict software performance, automatic tuning is needed. However, even with the use of an efficient automatic tuning method, it usually takes hours or days to find an optimal hyperparameter set since the hyperparameter search space (i.e. the space contains all the possible combinations of the hyperparameters) is very huge. To overcome this challenge, in this work, we also propose a hyperparameter search strategy for our deep sparse FNN such that it takes much less computation time while still obtains a good hyperparameter set that leads to higher prediction accuracy.

We have implemented the proposed performance prediction approach as a tool *DeepPerf* and evaluated it on eleven real-world configurable software systems with up to 10^{31} configurations and from different application domains, e.g. compiler, web server, database library, video encoders, etc. The experimental results show that for most of the systems, *DeepPerf* can achieve much higher prediction accuracy with smaller sample sizes compared to the state-of-the-art methods. For binary software systems (i.e., the systems with binary configuration options), *DeepPerf* statistically outperforms *DECART* (a state-of-the-art method for predicting the performance of binary software systems) on 3 out of 6 systems for all sample sizes and performs similarly on the other 3 systems. For binary-numeric software systems (i.e., the systems with both binary and numeric configuration options), *DeepPerf* outperforms *SPLConqueror* (a state-of-the-art method for predicting the performance of binary-numeric software systems) on 4 out of 5 systems for all sample sizes.

In summary, our contributions are as follows:

1) We are the first to propose to use a deep sparse FNN to model highly configurable software systems with binary and/or numeric configuration options.

2) We suggest a practical hyperparameter search strategy for the deep sparse FNN such that it can automatically find a good set of hyperparameters to achieve high prediction accuracy within a short time.

3) We implement our proposed method, namely *DeepPerf*, and extensively evaluate our method on eleven real-world configurable software systems with various sample sizes. The results show that for most of the systems, *DeepPerf* outperforms other state-of-the-art approaches (i.e. it achieves much higher prediction accuracy with less training data).

4) We empirically compare different types of regularized deep FNNs and other common machine learning methods (e.g. SVM) to show that a deep sparse FNN is a better choice to predict the performance of configurable software systems.

II. Background

A. The formulation of software performance prediction problem

Generally, a performance function of a software system with n configuration options is a function from configuration space

TABLE I
EXAMPLE OF A CONFIGURABLE SOFTWARE SYSTEM AND ITS PERFORMANCE VALUES.

x_1	x_2	x_3	...	x_8	x_9	x_{10}	x_{11}	$f(x)$
1	1	0	...	0	50	5	2	7319.56
1	1	0	...	1	50	1	3	9600.67
1	1	1	...	0	50	5	4	7374.26
1	1	0	...	1	50	1	2	9632.08
.	.	.	...	.	.	.	.	.
1	0	0	...	0	55	0	5	13256.9
1	0	0	...	0	53	3	2	9832.78

to a performance measurement:

$$f(\mathbf{x}) = f(x_1, x_2, \ldots, x_n) \ : \mathbb{X} \rightarrow \mathbb{R}.$$

where

- x_i $(i = \overline{1,n})$ is the variable that stores the value of the configuration option i^{th}. It can either be a Boolean value indicating if the configuration option is (de)selected or a real value in the value range for that configuration option.
- $\mathbb{X}$ is the Cartesian product of the domains of all the configuration options.

The objective is to predict the software performance value $f(\mathbf{x})$ of any new configuration vector $\mathbf{x}$ given a small sample of size m: $\{\mathbf{x}_i, f(\mathbf{x}_i)\}, i = 1, 2, \ldots, m$.

Table I shows an example of a software performance function $f(x_1, \ldots, x_n)$ and its configuration space. The software system has 11 configuration options in which 8 options take binary values and 3 options take numeric values. In total, the system has 2304 valid configurations. Measuring the time performance of all these configurations is difficult since it requires a lot of time and effort. To overcome this challenge, researchers propose to measure only the performance values of a limited number of configurations (*sample*), then build a prediction model from these data to predict the performance values of all configurations (*population*). The challenge here is to use only a small sample while still be able to predict the performance of all other configurations in the population with a high accuracy.

B. The L_1 regularization

The L_1 regularization technique was first introduced by Tibshirani [49] for linear regression with independent Gaussian noise, which is called Least Absolute Shrinkage and Selection Operator (LASSO). The idea is to add to the least squares loss function a penalty term that is constructed from the sum of the absolute values of the parameters, i.e. the L_1 norm of the parameters. By adding this penalty term to the loss function, the sum of the absolute values of the parameters is encouraged to be small. In practice, it has been frequently observed that L_1 regularization can cause many parameters to be equal to zero, which makes the parameter vector sparse [33]. Nowadays, L_1 regularization is not only applied in linear regression, but also in other models such as logistic regression [33] or neural network [17, 51].

C. Hyperparameter Tuning

For any machine learning model, there are variables that determine the model structure or decide how the network is trained, which are called hyperparameters. For example, for neural networks, the hyperparameters are the number of layers, number of neurons/layer, regularization hyperparameter, learning rate, etc. These hyperparameters need to be chosen accurately for the machine learning model to achieve a high prediction accuracy. The problem of identifying a good set of hyperparameters, η, is called *hyperparameter optmization* [7]. The general idea is to choose some trial points $\{\eta^{(1)}, \eta^{(2)}, \ldots\}$ from the hyperparameter space, evaluate the network performance on a validation dataset with these trial points, and pick the $\eta^{(i)}$ that has the smallest prediction error. The critical step in hyperparameter optimization is to choose the set of trials $\{\eta^{(1)}, \eta^{(2)}, \ldots\}$ effectively [7]. There are several widely used approaches to choose this set of trials from the hyperparameter space: grid search, random search [7], bayesian optimization [44], etc. In reality, the hyperparameter optimization (tuning) process is usually very time consuming as the hyperparameter space is huge and many trials are needed to find an optimal hyperparameter set.

III. A DEEP SPARSE FEEDFORWARD NEURAL NETWORK FOR SOFTWARE PERFORMANCE PREDICTION

In this section, we describe in detail the design choices when using neural network to model performance values of configurable software systems, and from there, we propose a neural network architecture that can predict software performance values using a small random sample with high prediction accuracy. Besides, we also suggest a practical hyperparameter search strategy to automatically find a good set of hyperparameters within a short time.

A. The design of the deep neural network

1) Design Rationale: The main consideration when selecting the architecture of an FNN for a specific problem is choosing the depth (i.e. the number of hidden layers) and the width (i.e. the number of neurons per layer). Since our approach is to use an FNN to represent performance function of software system, we are concerned with the question whether we should choose a shallow FNN (network with one hidden layer and a large number of neurons per layer) or a deep FNN (network with a large number of hidden layers and a small number of neurons per layer).

As stated in the universal approximation theorem [11, 23, 28], a shallow FNN having one linear output layer and one hidden layer with enough neurons and a suitable activation function (e.g. sigmoid, ReLU) can approximate any continuous function from one finite dimensional space to the other at any level of accuracy. However, to achieve a high level of accuracy, in the worst case, the number of neurons needed for a shallow FNN with one single hidden layer is an exponential number of the inputs [5]. For example, representing Boolean performance function using Fourier learning [55] can be considered similar to using a shallow FNN with one hidden layer and a large

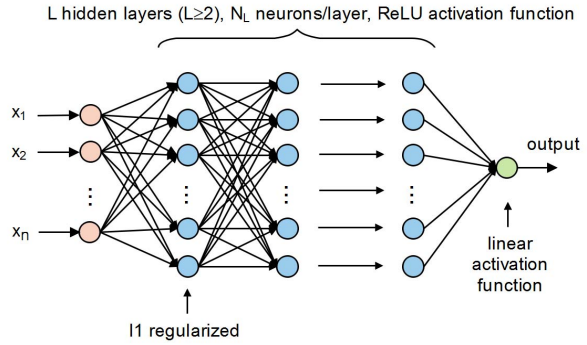

Fig. 1. The proposed L_1 regularized feedforward neural network for configurable software performance prediction. The inputs of the network are the n configuration options of the software system and the output of the network is the performance value.

number of neurons per layer (2^n neurons, where n is the number of configuration options). Unfortunately, in practice, there is no guarantee we can train such a neural network that has a very large number of neurons [18]. The reasons are: (1) the optimization algorithm used might not be able to find the values of the parameters corresponding to the desired function, (2) the training algorithm might choose the wrong function [18]. Besides, training a huge number of parameters costs a lot of computation time and memory. This is known as the curse of dimensionality problem.

Deep FNNs, on the other hand, can avoid the curse of dimensionality that shallow networks encounter [36]. When using FNNs to approximate functions, for a given upper bound of the approximation error, shallow FNNs require exponential more parameters than deep FNNs [13, 29, 53]. Therefore, for our approach, we propose to use a deep FNN to model performance values of configurable software systems. Besides, to make the hyperparameter tuning process easier and faster, we suggest to fix the number of neurons in each hidden layer to be the same. By doing this, the complexity of the FNN will be controlled by only two hyperparamters: the number of hidden layers and the number of neurons in each hidden layer.

2) The Network Architecture: The overall architecture of deep FNN for software performance prediction is as follows:

- The input layer has n neurons, where n is the number of configuration options of the software system needs to be predicted. The output layer has 1 neuron, which outputs the performance value of the software system.
- There are L hidden layers ($L \geq 2$) and each hidden layer has N_L number of neurons.
- All the hidden layers use a ReLU activation function while the output layer uses a linear activation function. Using a linear output layer is required since the performance prediction is a regression problem. Meanwhile, the ReLU is chosen as the activation function of the hidden layers due to its ability to learn much faster in networks with many layers compared to other non-linear activation functions [17, 30].

For this FNN architecture, there are $(N_L+1)\times(n+N_L L+1)$ trainable parameters. This number depends on n, N_L and L, hence, the network is able to represent more complex functions when the software system has more configuration options.

B. Reducing network complexity through regularization

1) Selecting the regularization techniques: A challenge in training the deep FNN architecture described in the previous subsection is that in reality, we often have very limited amount of training data as the main purpose of the software performance prediction problem is to predict the performance values from a small sample. With this condition, the model is ill-posed, meaning that there is an infinite number of parameters that can obtain a perfect fit to the training data. However, these parameters normally do not fit well to the new data. One of the key ideas for solving ill-posed (or overfitting) problems is to introduce additional information to the network by using a suitable regularization technique. At present, there are three most common regularization methods that are used in many deep learning applications:

1) The L_1 regularization. As described in Section II-B, the idea of L_1 regularization is to add to the loss function a penalty term which is constructed by applying an L_1 norm on the parameters [49].
2) The L_2 regularization. Similar to L_1 regularization, L_2 regularization works by adding a penalty term to the loss function, however, in this case, an L_2 norm is being used to generate the penalty term [50]. L_2 regularization is arguably the most popular technique in machine learning to combat overfitting.
3) The dropout technique. Dropout technique was specifically proposed for neural networks [45]. The key insight is to randomly drop neurons (along with their connections) from the network during training to prevent the network to adapt too much to the training data.

To choose a regularization technique that can improve the performance prediction accuracy, we need to select the technique that is able to utilize the prior knowledge about configurable software systems.

2) L_1-Regularization of the network: For configurable software systems, it has been widely observed that even though the possible number of interactions among configuration options is exponential, a very large portion of potential interactions has no influence on the performance of software systems [24, 27, 40, 42]. This means that the parameters of the neural network could be very sparse, i.e. only a small number of parameters have significant impact on the model. Hence, we suggest to use a regularization technique that enables the FNN to satisfy this condition, which means that L_1 regularization technique is the best candidate due to its ability to make the model parameters sparse. Let denote θ as the weights of the FNN, β as the bias, X as the input data, Y as the output data and $J(\theta, \beta, X, Y)$ as the loss function of the network. Applying L_1 regularization to the network means changing the loss function from $J(\theta, \beta, X, Y)$ to:

$$J_{\text{reg}}(\theta, \beta, X, Y) = J(\theta, \beta, X, Y) + \boldsymbol{\lambda}\|\theta\|_1,$$

where $\boldsymbol{\lambda}$ denotes the regularization hyperparameters and $\|.\|_1$ is the L_1 norm.

A consideration when using the L_1 regularization in deep FNNs is whether we need to apply regularization to all the hidden layers in the network. Intuitively, it might increase the prediction accuracy if we apply L_1 regularization to all the layers and each layer has a different regularization hyperparameter. Unfortunately, in practice, this process is infeasible. The reason is that the effectiveness of regularization depends mostly on the choice of the regularization hyperparameter, and finding the right regularization hyperparameter for each layer is difficult, especially when there are many layers in the network. Alternatively, one can use one global regularization hyperparameter for all the layers in the network, however, in this case, the choice of this hyperparameter becomes more sensitive. A slight increase or decrease in this hyperparameter can affect the whole model accuracy greatly because this hyperparameter affects all the layers in the network. When we have a very limited amount of data for validation, such as in the case of software performance prediction, this becomes a weakness since an abnormal segment of validation data can affect the choice of this hyperparameter, which in turn affects adversely the prediction performance of the FNN.

Based on these observations, we suggest to only apply L_1 regularization to the first hidden layer. Note that in FNNs, a layer is a function of the layer that preceded it, so by shrinking some parameters in the first layer to zero, we actually remove a lot of irrelevant connections in the latter layers within the network. Therefore, with only one layer to be regularized, the L_1 regularization still has good effect on the whole network. The proposed L_1 regularized deep FNN for software performance prediction is shown in Figure 1.

C. Efficient Hyperparameter Tuning

For the proposed deep FNN architecture, there are three hyperparameters that control the complexity of the network: number of layers, number of neurons/layer, regularization hyperparameter, and two key hyperparameters that control the model training process: learning rate, number of epochs. The hyperparameter space constructed from these five hyperparemeters is the Cartesian product of the domains of all hyperparameters, which is huge. Searching for an optimal hyperparameter set from this hyperparameter space is computationally expensive as many trials may be required. In this section, we aim to reduce the hyperparameter searching effort by: (1) fixing some dependent hyperparameters, (2) deriving a search strategy to effectively reduce the hyperparameter space.

First, we set some dependent hyperparameters to some fixed values. Specifically,

- Set the number of neurons/layer, N_L, to be a fixed value (e.g. 128). As discussed in Section III-A, when using FNN to approximate functions, a deep FNN is more beneficial. Previous studies also show that the network depth is what matters most [13, 29, 36]. Therefore, we choose to fix the number of neurons per layer (N_L) instead of the number of layers (L). We can control the complexity of the FNN by the number of layers and the regularization hyperparameter.
- Set the number of epochs to be a fixed value (e.g. 2000). With this setting, we can control the training process using the learning rate. This setting is a way to constrain the training time budget, i.e. the tuning method needs to find the optimal learning rate given the training time budget.

Second, we split the sample into two parts: training and validation, and then use the proposed search strategy below to find the optimal hyperparameters. Our search strategy comprises of two steps:

Step 1: Start from a non-regularized FNN with 2 hidden layers, search for the optimal learning rate of this network architecture, train the network with this learning rate and evaluate the prediction error on the validation dataset. Keep adding more hidden layers to the non-regularized FNN until the validation error starts to go up, which is the sign when the FNN starts to overfit. This is the optimal number of hidden layers for a non-regularized FNN to fit well with the performance values of the software system. Note that the optimal learning rate is chosen as the largest value that makes the training error closest to 0.

Step 2: Use the number of hidden layers found in Step 1, add a few more layers and search for the optimal learning rate of this FNN architecture. Finally, add L_1 regularization to this non-regularized FNN and search for the optimal L_1 regularization hyperparameter. The reason we need to add a few layers after finding the optimal non-regularized FNN architecture is that when we apply regularization to a network, the network needs to be deeper compared to a non-regularized one in order to achieve high prediction accuracy [18]. We suggest to add 5 more layers based on our empirical experiments with various configurable software systems.

With this hyperparameter search strategy, the hyperparameter search space is much smaller than the original one. Specifically, if we denote D_n, D_l, D_r as the search space of the number of layers L, the learning rate, and the regularization hyperparameter λ, respectively, the original hyperparameter search space is $D_n D_l D_r$. With our proposed search strategy, the search space will be $(D_n D_l + D_l + D_r)$, which is much smaller than the original search space. Note that in each step of our search strategy, any conventional hyperparameter tuning method (e.g. grid search, random search, bayesian optimization, etc.) can be used to find the optimal hyperparameters.

D. Training of the network

In this section, we describe some technical details we utilized in order to train our proposed model effectively. These details are critical in achieving the high prediction accuracy of our approach.

- Loss function: We choose the loss function to be the mean square errors between the real output and the predicted output as this is the most commonly used regression loss function in machine learning.

- Input/Output/Weights normalization: To make regularization and hyperparameter tuning method work effectively, we normalize the input and the output of the training data. During testing, for predicting performance values of new configurations, we normalize the new data in the same way we did during training. That is, we use exactly the same parameters for normalization during training for any new inputs to the network. In addition, we utilize Xavier initialization [16] to initialize weights of all the hidden layers in the network.
- Optimization algorithm: We use the Adam optimizer to train the neural network as it is a very computational efficient method [26]. Besides, to avoid exploding gradients, gradient clipping technique (i.e. limits the magnitude of the gradients between -1 and 1) is utilized during the training process. Finally, the batch size is set to the whole sample size (i.e. size of the training data) since for performance prediction problem, the size of the training data is small.

E. Tool Implementation

We implement our proposed approach, *DeepPerf*, using Python 3.6 and Tensorflow 1.8.0 [3]. The input is normalized between 0 and 1 since it is a standard way to do normalization. The output is normalized between 0 and 100 (instead of between 0 and 1) as we do not want the network parameters to be too small. All the hyperparameters of the Adam optimizer is set using the default values in Tensorflow.

To search for the hyperparameters, we use 67% of sample for training and 33% for validation. For the learning rate, we use grid search with 4 points logarithmically equally spaced in the range $[0.0001, 0.1]$. We also use a learning rate schedule, i.e. the learning rate is dropped by factor of 0.001 after every epoch. For the L_1 regularization hyperparameter λ, we use grid search with 30 points logarithmically spaced in the range $[0.01, 1000]$. Note that here 0.01 corresponds to a very little regularization and 1000 corresponds to a very large regularization. Since we normalize both input and output of the neural network, all the software systems share the same range for searching for λ. The optimal L_1 regularization hyperparameter is the value that achieves the smallest validation error.

IV. Evaluation

A. Experimental Design

In this section, we aim at answering the following research questions (RQ):

RQ1: How accurate is our proposed approach in predicting performance of configurable software systems with binary options? To answer this RQ, we conduct an experiment to compare *DeepPerf* with other state-of-the-art approaches on binary configurable software systems.

RQ2: How accurate is our proposed approach in predicting performance of configurable software systems with binary and numeric options? To answer this RQ, we conduct an experiment to compare *DeepPerf* with other state-of-the-art approaches on binary-numeric configurable software systems.

RQ3: Is a complex model like the deep sparse FNN actually needed or can we utilize SVM or other NN-based regression methods to achieve the same level of prediction accuracy? To answer this RQ, we conduct an experiment to compare *DeepPerf* with the support vector machine (SVM) regression method and other alternative designs described in Section III-B: the L_1 regularized FNN where the L_1 regularization is applied to all layers, the non-regularized FNN, the L_2 regularized FNN, and the dropout FNN.

RQ4: What is the time cost of *DeepPerf* to predict performance of a configurable software system? This RQ is to evaluate the practicality and feasibility of our proposed approach. To answer this RQ, we show the time consumed by the hyperparameter searching and training process of our approach on various highly configurable software systems, including binary and binary-numeric systems.

The detailed setup for each experiment will be described in the subsequent sections. In general, to compare the prediction accuracy between learning methods, we use the training dataset (sample) to generate a performance model for each method, and then use this model to predict performance values of configurations on the testing dataset. To evaluate the prediction accuracy, we use the mean relative error (MRE), which is computed as,

$$MRE = \frac{1}{|C|} \sum_{c \in V} \frac{|predicted_c - actual_c|}{actual_c} \times 100, \quad (1)$$

where V is the testing dataset, $predicted_c$ is the predicted performance value of configuration c, $actual_c$ is the actual performance value of configuration c. We choose this metric as it is widely used to measure the accuracy of prediction models [14, 21, 25, 41].

B. Subject Systems

For the experiments, we use eleven real-world configurable software systems: six of these systems have only binary configuration options and were used in [21, 31, 37, 42, 55], the other five systems have both binary and numeric configuration options and were used in [41]. These systems have different characteristics and are from different application domains, e.g. multi-grid solver, web server, video encoder, database library, database management system, compiler, etc. They are also of different sizes (45 thousands to more than 300 thousand lines of code) and written in different languages (Java, C, and C++). The number of configuration options ranges from 8 to 60 while the number of valid configurations ranges from 180 to 10^{31}. These software systems were measured and published online [2]. More information about these systems and how they were measured can be found in [41, 42]. The overview of these eleven subject systems is given in Table II.

C. RQ1: Comparison on software systems with binary options

As mentioned in Introduction, at present, there are many learning methods for predicting performance values of software systems with binary options, including *SPLConqueror*

TABLE II
THE SUBJECT SOFTWARE SYSTEMS

System	Domain	#Binary	#Numeric	#Configs
Apache	Web Server	9	0	192
x264	Video Encoder	16	0	1152
LLVM	Compiler	11	0	1024
BDB-C	Database System	18	0	2560
BDB-J	Database System	26	0	180
SQLite	Database System	39	0	4653
DUNE MGS	Multi-Grid Solver	8	3	2304
HIPA^{cc}	Image Processing	31	2	13485
HSMGP	Stencil-Grid Solver	11	3	3456
JavaGC	Runtime Env.	12	23	10^{31}
SaC	Compiler	53	7	10^{23}

#Binary is the number of binary configuration options.
#Numeric is the number of numeric configuration options.
#Configs is the number of valid configurations

[41, 42], *FourierLearning* [55], and *DECART* [21] (the improved version of *CART* [19]). Among these approaches, *DECART* is recently proposed and can achieve higher prediction accuracy than others [21]. Hence, in this experiment, we will only compare our proposed method with *DECART*. We will evaluate the two approaches on six binary subject systems in Table II: Apache, x264, LLVM, BDB-C, BDB-J and SQLite. These six subject systems were also used in [21].

1) Setup: Here, we adopt the experiment setup in [21]. Specifically, for each subject system, we *randomly* select a certain number of configurations and their corresponding performance values to construct the training dataset (sample); all the remaining measurements are then used as the testing dataset. We use five different sizes for the training dataset of each subject system: n, $2n$, $3n$, $4n$, $5n$, where n is the number of options of each system (which is shown in the column #Binary of Table II). To evaluate the consistency and stability of the approaches, for each sample size of each subject system, we repeat this random sampling, training and testing process 30 times. We then report the mean and the 95% confidence interval[1] of the *MREs* obtained after 30 experiments with *DeepPerf* and *DECART*. In addition, we also use *t-test* with the significant level 0.05 to statistically compare the performance of the two methods for each sample size.

To replicate the *DECART* results, we utilize the code published on their project page [1]. We ran *DECART* with the best hyperparameter tuning technique suggested in their paper [21]: grid search combined with 10-fold cross-validation. Other hyperparameter settings are the same as described in Section 4 of [21].

2) Results: Table III shows the prediction MREs of *DeepPerf* and *DECART* on six binary subject systems with multiple sample sizes. It can be seen that *DeepPerf* statistically outperforms *DECART* for Apache, x264 and LLVM with all sample sizes. For these software systems, using *DeepPerf*, both the MRE means and their 95% confidence intervals are much lower than those getting from *DECART*. Specifically,

[1]The 95% confidence interval of a random variable x is computed as $[\bar{x} - 1.95\sigma/n, \bar{x} + 1.95\sigma/n]$, where $\bar{x}$ and σ are the mean and standard deviation of that random variable and n is the number of tests. In our case, the random variable is the MRE and $n = 30$.

TABLE III
COMPARISON BETWEEN *DeepPerf* AND *DECART*

Subject System	Sample Size	*DECART*		*DeepPerf*		Better Algorithm
		Mean	Margin	Mean	Margin	
Apache	n	NA	NA	17.87	1.85	NA
	2n	15.83	2.89	**10.24**	1.15	*DeepPerf*
	3n	11.03	1.46	**8.25**	0.75	*DeepPerf*
	4n	9.49	1.00	**6.97**	0.39	*DeepPerf*
	5n	7.84	0.28	**6.29**	0.44	*DeepPerf*
x264	n	17.71	3.87	**10.43**	2.28	*DeepPerf*
	2n	9.31	1.30	**3.61**	0.54	*DeepPerf*
	3n	6.37	0.83	**2.13**	0.31	*DeepPerf*
	4n	4.26	0.47	**1.49**	0.38	*DeepPerf*
	5n	2.94	0.52	**0.87**	0.11	*DeepPerf*
BDB-J	n	10.04	4.67	**7.25**	4.21	Same
	2n	2.23	0.16	**2.07**	0.32	Same
	3n	2.03	0.16	**1.73**	0.12	*DeepPerf*
	4n	1.72	0.09	**1.67**	0.12	Same
	5n	1.67	0.09	**1.61**	0.09	Same
LLVM	n	6.00	0.34	**5.09**	0.80	Same
	2n	4.66	0.47	**3.87**	0.48	*DeepPerf*
	3n	3.96	0.39	**2.54**	0.15	*DeepPerf*
	4n	3.54	0.42	**2.27**	0.16	*DeepPerf*
	5n	2.84	0.33	**1.99**	0.15	*DeepPerf*
BDB-C	n	151.0	90.70	**133.6**	54.33	Same
	2n	43.8	26.72	**16.77**	2.25	Same
	3n	31.9	22.73	**13.1**	3.39	Same
	4n	**6.93**	1.39	6.95	1.11	Same
	5n	**5.02**	1.69	5.82	1.33	Same
SQLite	n	**4.87**	0.22	5.04	0.32	Same
	2n	4.67	0.17	**4.63**	0.13	Same
	3n	**4.36**	0.09	4.48	0.08	Same
	4n	**4.21**	0.1	4.40	0.14	Same
	5n	**4.11**	0.08	4.27	0.13	Same

Mean: mean of the MREs seen in 30 experiments. Margin: margin of the 95% confidence interval of the MREs in 30 experiments. Better algorithm is chosen using t-test on 30 MRE data points with significant level 0.05.

to get the same level of accuracy, *DeepPerf* needs far less training data than *DECART*. For example, with the system x264, *DeepPerf* only needs $2n$ sample to achieve a prediction MRE of 3.61% (i.e. accuracy of 96.39%) whilst *DECART* needs $4n$ - $5n$ sample to have the similar prediction MRE. For BDB-C, even though there is no statistically significant difference between *DeepPerf* and *DECART*, most of the time *DeepPerf* has much smaller MRE means and margins, which indicates that *DeepPerf* is more consistent than *DECART*. For BDB-J and SQLite, *DeepPerf* and *DECART* perform quite similarly.

D. RQ2: Comparison on software systems with binary-numeric options

Regarding the problem of predicting performance values of software systems with both binary and numeric options, at present, *SPLConqueror* is the only method that can do this task. So in this section, we will compare the effectiveness of *DeepPerf* with that of *SPLConqueror*. We will use the five binary-numeric subject systems in Table II: DUNE MGS,

HIPA^{cc}, HSMGP, JavaGC, and SaC. These subject systems are also used in [41] for evaluating *SPLConqueror*.

1) Setup: *SPLConqueror* combines different sampling heuristics for binary options (i.e. option-wise (OW), negative option-wise (nOW), pair-wise (PW), etc), and several experimental design methodologies (i.e. Plackett-Burman (PBD), Random Design (RD), etc) for numeric options to achieve good prediction accuracy. So in this experiment, we cannot choose sample size as any random value. Thus, to compare the two approaches, for each subject system, we evaluate the two methods using the same sample sizes that *SPLConqueror* suggested, and,

- For *SPLConqueror*, the sample is chosen based on the sampling heuristics and experimental designs that are proposed by the authors of *SPLConqueror*.
- For *DeepPerf*, the sample is chosen using the random sampling heuristic.

Since there are many combinations of sampling heuristics and experimental designs, for each subject system, we will only pick the best four combinations that enable *SPLConqueror* to achieve the highest prediction accuracy using the smallest sample. Besides, since each combination yields a unique sample (except combinations having random experimental design), thus, for *SPLConqueror*, we only report the mean MRE on the testing dataset. For *DeepPerf*, to reduce fluctuations caused by randomness, with each sample size, we repeat the random sampling, training, and testing process 30 times. We then report both the MRE mean and the 95% confidence interval margin on the testing dataset. The testing dataset consists of all the remaining configurations after selecting the training sample. For JavaGC and SaC, because they have too many configurations, therefore we only select 15,000+ randomly selected configurations as their testing datasets.

2) Results: Table IV shows the prediction MREs of *DeepPerf* and *SPLConqueror* on five binary-numeric subject systems with multiple sample sizes. We can observe that *DeepPerf* outperforms *SPLConqueror* on 4/5 subject systems (DUNE MGS, HIPA^{cc}, JavaGC and SaC) under all sample sizes. To get the same level of accuracy, *SPLConqueror* needs much more training data compared to *DeepPerf*. For these software systems, both the means and the margins of *DeepPerf*'s prediction errors are small, which indicates that *DeepPerf* can consistently predict performance values with high accuracy. For HSMGP, *SPLConqueror* performs better than *DeepPerf* but the difference is not too large, only around 1%-2%.

TABLE IV
COMPARISON BETWEEN *DeepPerf* AND *SPLConqueror*

Subject System	Sample Size	*SPLConqueror*		*DeepPerf*		
		Sampling Heuristic	Mean	Sampling Heuristic	Mean	Margin
DUNE MGS	49	OW RD	20.1	RD	**15.73**	0.90
	78	PW RD	22.1	RD	**13.67**	0.82
	240	OW PBD(49,7)	10.6	RD	**8.19**	0.34
	375	OW PBD(125,5)	8.8	RD	**7.20**	0.17
HIPA^{cc}	261	OW RD	14.2	RD	**9.39**	0.37
	528	OW PBD(125,5)	13.8	RD	**6.38**	0.44
	736	OW PBD(49,7)	13.9	RD	**5.06**	0.35
	1281	PW RD	13.9	RD	**3.75**	0.26
HSMGP	77	OW RD	**4.5**	RD	6.76	0.87
	173	PW RD	**2.8**	RD	3.60	0.2
	384	OW PBD(49,7)	**2.2**	RD	2.53	0.13
	480	OW PBD(125,5)	**1.7**	RD	2.24	0.11
JavaGC	423	OW PBD(49,7)	37.4	RD	**24.76**	2.42
	534	OW RD	31.3	RD	**22.98**	2.77
	855	OW PBD(125,5)	21.9	RD	**21.83**	7.07
	2571	PW PBD(49,7)	28.2	RD	**16.48**	6.59
SaC	2060	OW RD	21.1	RD	**15.83**	1.25
	2295	OW PBD(125,5)	20.3	RD	**19.25**	6.03
	2499	OW PBD(49,7)	**16**	RD	16.73	1.13
	3261	PW RD	30.7	RD	**15.64**	1.18

Mean: mean of the MREs seen in 30 experiments. Margin: margin of the 95% confidence interval of the MREs seen in 30 experiments.

E. RQ3: Comparison with SVM and other NN-based regression methods

In this experiment, we aim to evaluate whether we need to use a complex model like deep FNN or a much simpler regression method in order to achieve the same level of accuracy. We also evaluate whether sparsity regularization is actually needed or we can use other regularization methods. We compare *DeepPerf* with the SVM regression method [10] and other design alternatives that were discussed in Section III-B:

1) Deep FNN with the L_1 regularization applied to all layers (*L1-all-FNN*)
2) Deep FNN without regularization (*Plain-FNN*)
3) Deep FNN with the L_2 regularization [50] (*L2-FNN*)
4) Deep FNN with the dropout technique [45] (*Dropout-FNN*)

Five binary subject systems in Table II with three sample sizes per system will be used to evaluate the approaches.

1) Setup: For the SVM regression method, we use the function *SVR* in scikit-learn package [35] to perform model training and prediction. Scikit-learn *SVR* function is implemented using *libsvm*, an SVM library proposed in [9]. To select the best hyperparameters for *SVR*, we utilize grid-search and 10-fold cross validation. We construct the grid by varying four hyperparameters in *SVR*: 10 values of C ranging from 0.01 to 1000, 10 values of $gamma$ ranging from 0.001 to 1, 5 values of $epsilon$ ranging from 0.001 to 1 and the kernel functions are *linear*, *poly* or *rbf*. With this setting, the hyperparameter space contains 1500 different combinations of hyperparameter values, hence, we believe it is sufficient enough to find the optimal setting for *SVR*.

For the *Plain-FNN* approach, we use the Step 1 of our proposed hyperparameter search strategy (described in Section III-C) to find the optimal network architecture and the learning rate. For *L1-all-FNN*, *L2-FNN*, and *Dropout-FNN*, we also use our proposed hyperparameter search strategy to tune the hyperparameters. All the settings of the hyperparameter searching process are the same as those for *DeepPerf*. Except that, for *Dropout-FNN*, the search range for the dropout hyper-

TABLE V
COMPARISON BETWEEN *DeepPerf*, *SVM* AND OTHER NN-BASED REGRESSION METHODS

Subject System	Sample Size	*DeepPerf*		*L1-all-FNN*		*Plain-FNN*		*L2-FNN*		*Dropout-FNN*		*SVM*	
		Mean	Margin	Mean	Margin	Mean	Margin	Mean	Margin	Mean	Margin	Mean	Margin
Apache	n	**17.87**	1.85	19.39	1.60	21.83	1.34	18.18	1.39	23.0	1.93	22.97	1.56
	3n	**8.25**	0.75	8.97	1.52	10.52	0.65	8.71	0.66	9.99	0.60	13.74	0.90
	5n	**6.29**	0.44	7.15	1.41	8.40	0.65	6.94	0.61	7.42	0.49	8.77	0.74
x264	n	**10.43**	2.28	10.70	1.92	13.56	1.07	12.41	1.32	14.1	1.90	13.62	1.64
	3n	**2.13**	0.31	2.77	0.22	4.99	0.43	3.10	0.31	4.58	0.26	5.26	0.41
	5n	**0.87**	0.11	1.72	0.10	3.12	0.18	1.31	0.09	1.52	0.14	2.62	0.18
BDB-J	n	7.25	4.21	12.12	4.97	12.47	1.76	**6.76**	2.00	12.1	3.10	17.1	2.11
	3n	**1.73**	0.12	3.38	2.88	3.11	0.17	1.76	0.12	2.76	0.27	5.49	0.30
	5n	1.61	0.09	1.62	0.09	2.37	0.15	**1.59**	0.09	2.24	0.16	2.70	0.17
LLVM	n	5.09	0.80	**4.91**	0.71	8.03	1.53	4.50	0.34	6.16	0.45	4.64	0.37
	3n	**2.54**	0.15	2.80	0.17	5.07	0.55	2.68	0.18	3.96	0.25	2.56	0.15
	5n	**1.99**	0.15	2.39	0.18	3.87	0.27	2.32	0.15	2.98	0.16	2.22	0.13
BDB-C	n	**133.6**	54.3	221.4	51.4	216.6	47.1	187.8	50.0	273.3	68.2	263.8	39.66
	3n	**13.10**	3.39	118.5	91.4	67.34	9.13	26.61	4.92	47.7	9.73	264.9	36.66
	5n	**5.82**	1.33	50.55	49.3	24.43	3.39	10.70	1.61	16.8	1.95	212.5	19.9
SQLite	n	5.04	0.32	4.72	0.27	6.38	0.40	5.53	0.31	4.94	0.17	**4.51**	0.14
	3n	4.48	0.08	4.49	0.07	5.02	0.08	4.70	0.13	4.51	0.07	**4.08**	0.05
	5n	4.27	0.13	4.07	0.12	4.82	0.13	4.18	0.11	4.29	0.06	**3.80**	0.04

Mean: mean of the MREs seen in 30 experiments. Margin: margin of the 95% confidence interval of the MREs seen in 30 experiments.

parameter is $[0.001, 1]$ since the dropout hyperparameter needs to be smaller than 1. And for *L1-all-FNN*, the search range for the L_1 regularization hyperparameter is $[0.001, 100]$ (whilst *DeepPerf*'s search range is $[0.01, 1000]$), since in this case we apply regularization to all the layers so the regularization hyperparameter needs to be smaller.

2) Results: Table V shows the prediction MREs of the six approaches. As expected, for all subject systems, *Plain-FNN* performs the worst compared to other NN-based approaches. The reason is that a deep neural network is prone to overfit on training data. Without a regularization technique it cannot produce good predictions for new data. The *L2-FNN* and *Dropout-FNN* approaches can overcome the overfitting problem of the *Plain-FNN*. However, for software performance, the L_2 regularization or dropout technique is not as effective as the L_1 regularization. These approaches perform similarly to *DeepPerf* on systems Apache, BDB-J, LLVM, SQLite. However, they perform much worse than *DeepPerf* on systems BDB-C and x264. For *L1-all-FNN*, it performs reasonably well for most of the systems but performs badly on BDB-C. Lastly, for *SVM*, even though it has quite good performance on system SQLite, it does not work well on other systems.

F. RQ4: Time cost of DeepPerf

The time cost of *DeepPerf* on searching for optimal hyperparameters and training a model is reasonable. Specifically,

- For binary subject systems with the number of configuration options n less than 20 (Apache, x264, LLVM, BDB-C), when the sample size increases from n to $5n$, the time taken by hyperparameter searching and model training increases from 1 to 4 minutes. For binary subject systems with more than 20 configuration options (BDB-J, SQLite), it takes *DeepPerf* 2 - 6 minutes to do hyperparameters searching and network training.
- For binary-numeric subject systems with the number of configurations n less than 20 (DUNE MGS, HSMGP), and with the sample sizes as shown in Table IV, the time cost for the hyperparameter searching and model training process ranges from 2 to 5 minutes. For binary-numeric subject systems with more than 20 configuration options (HIPA^{cc}, JavaGC, SaC), and with the sample sizes as shown in Table IV, the hyperparameter tuning and model training process takes 4 to 30 minutes.

Meanwhile, the time cost of *DECART* and *SPLConqueror* is from a few seconds to 2 minutes for the systems and sample sizes in Tables III and IV. Even though *DeepPerf* takes longer time to build the prediction model than *DECART* and *SPLConqueror*, the time cost of *DeepPerf* is still acceptable. Besides, it is worth re-emphasizing that *DeepPerf* can predict performance of binary and/or numeric systems with very high accuracy while *DECART* can only predict performance of binary systems and *SPLConqueror* needs to use some special sampling heuristics to achieve good prediction accuracy. Note that all the time cost here is measured when running all the methods on a Windows 7 computer with Intel Xeon CPU E5-1650 3.2GHz 16GB RAM,

G. Discussions

1) Strengths and limitations of our proposed approach: The first strength of *DeepPerf* is that it can predict performance of highly configurable software systems with both binary and numeric options at higher prediction accuracy than other state-of-the-art approaches. As shown in our experiments, for software systems with binary options, compared to *DECART*, most of the time *DeepPerf* can achieve much better prediction

accuracy while using less sample. For software systems with binary and numeric options, *DeepPerf* outperforms *SPLConqueror* in most of the subject systems for all sample sizes.

DeepPerf's second strength is that it uses random sampling heuristic to select sample for model training, hence, it is flexible when constructing the sample. Furthermore, it can be incorporated with other sampling heuristics and experimental designs to further improve prediction accuracy and reduce measurement effort. *FourierLearning* and *CART/DECART* also use random sampling while *SPLConqueror* needs to use some specific sampling heuristics and experimental designs to achieve high prediction accuracy.

Finally, the third strength of *DeepPerf* is that it is both an automated algorithm (i.e. it does not require human effort to tune the hyperparameters) and a progressive algorithm (i.e. users can always achieve a much higher model accuracy when having more training data). This can be seen in Tables III and IV, when the sample size increases, the prediction accuracy of *DeepPerf* also increases.

A limitation of *DeepPerf* is that it takes longer time to train than *DECART* and *SPLConqueror*. For most of the subject systems, using a Windows 7 computer with Intel Xeon CPU E5-1650 3.2GHz 16GB RAM, *DECART* and *SPLConqueror* normally take a few seconds to a few minutes to train a model. Meanwhile, *DeepPerf* takes a few minutes for systems with less than 20 configuration options and can take up to 30 minutes for systems with more than 50 options.

2) Threats to Validity: To increase the internal validity of our experiment results, for each sample size, we repeat the prediction process 30 times with random training dataset. For each process, the prediction is evaluated on a test dataset which does not include any part of the training dataset. We use mean relative error as a metric as it is a widely-used metric in the literature for evaluating the effectiveness of performance prediction algorithm and it is also used to evaluate other approaches we compared. Note that, in the paper, to evaluate the algorithm stability, we not only evaluate using the mean relative error, but also using the 95% confidence interval.

For external validity, we evaluate the algorithms using eleven public datasets with different characteristics, domains, languages, etc. These subject systems have a large range of configuration options and has been used extensively in the literature to evaluate the effectiveness of performance prediction algorithm.

V. Related Work

Many large and complex software systems are highly configurable. A configuration option can be treated as a feature and a configurable system can be treated as a software product line (a family of similar software systems). A large body of work has been devoted to modeling features and checking consistency of feature configurations (e.g., [6, 12, 47, 52]). Despite their importance, quality attributes (or non-functional requirements) such as performance have not been sufficiently addressed in product line practice.

To predict the quality attributes of a product line member, Zhang et al. [54] proposed a Bayesian Belief Network (BBN) based approach. By performing qualitative analysis over the BBN, the quality of a product line member can be estimated. Their method is good for quality attributes (such as security, reusability, etc) that are "hard to define, impossible to measure, easy to recognize [14]". However, some quality attributes, such as performance, can be relatively easy to measure. A quantitative analysis of these quality attributes can be complementary to the qualitative analysis [20]. Recently, researchers have measured performance of several large-scale configurable systems and proposed various learning methods to build performance prediction models from these measurements. In Introduction, we have described pros and cons of some state-of-the-art methods including *SPLConqueror* [40, 42], *CART/DECART* [19, 21] and *FourierLearning* [55]. Our experimental results show that for most of the evaluated subject systems, the proposed *DeepPerf* approach outperforms all the related methods, i.e. achieve higher prediction accuracy and use less sample data.

There are also some work on selecting an optimal sample of configurations. For example, Sayyad et al. [38, 39] utilized evolutionary algorithms to select optimal features regarding multiple objectives. Sarkar et al. [37] used projective sampling and feature-frequency heuristic to determine sample that is small enough to decrease the measurement effort and large enough to increase the prediction accuracy. Nair et al. [32] proposed to use the *WHAT* spectral learner to select a small number of configurations. *WHAT* computes distance matrix between the configurations and performs dimensionality reduction. Later, they also proposed a rank-based approach [31], which can reduce the cost (in terms of the number of configurations to be measured) as well as the time required to build performance models. Our work aims to suggest a new learning method to construct software performance model from a sample, which is a different goal compared to the above work. In fact, we can combine our learning approach with these work to further improve the accuracy of the performance prediction model and use less sample data.

VI. Conclusion

In this paper, we have proposed *DeepPerf*, a performance prediction model for highly configurable systems based on deep sparse FNN. We also design a practical hyperparameter search strategy, which can automatically find a good set of hyperparameters that can lead to high prediction accuracy within a short time. The experimental results on public datasets show that *DeepPerf* can achieve better performance prediction accuracy with less data, when compared to other state-of-the-art approaches. Furthermore, *DeepPerf* can work with both binary and numeric configuration options.

In the future, we will explore the universal property of neural networks in approximations of different function classes [29] to further improve the design of our model.

Our experimental data and source code are publicly available at: *https://github.com/DeepPerf/DeepPerf*.

Acknowledgment. This work is supported by NSFC grant 61828201.

References

[1] DECART project page. https://github.com/jmguo/DECART, accessed 2019-02-01.

[2] SPLConqueror project page. http://www.fosd.de/SPLConqueror, accessed 2019-02-01.

[3] M. Abadi and P. Barham et all. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016.

[4] M. Anthony. Boolean functions and artificial neural networks. Technical Report CMU/SEI-90-TR-021, Department of Mathematics, London School of Economics, London, UK, 2003.

[5] A. R. Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information Theory*, 39(3):930–945, 1993.

[6] D. Batory, D. Benavides, and A. Ruiz-Cortes. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, 49(12):45–47, December 2006.

[7] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13:281–305, 2012.

[8] M. Calder, M. Kolberg, E. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

[9] C. Chang and C.J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/∼cjlin/libsvm.

[10] C. Cortes and V. Vladimir. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[11] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[12] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *Proceedings of the 11th International Software Product Line Conference (SPLC)*, pages 23–34, 2007.

[13] R. Eldan and O. Shamir. The power of depth for feedforward neural networks. In *Proceedings of the JMLR: Workshop and Conference*, volume 49, pages 1–34, 2016.

[14] T. Foss, E. Stensrud, B. Kitchenham, and I. Myrtveit. A simulation study of the model evaluation criterion mmre. *IEEE Transactions on Software Engineering*, 29(11):985–995, 2003.

[15] K. Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural Networks*, 2(3):183–192, 1989.

[16] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 249–256, 2010.

[17] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 315–323, 2011.

[18] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[19] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wsowski. Variability-aware performance prediction: A statistical learning approach. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 301–311, 2013.

[20] J. Guo, J. White, G. Wang, J. Li, and Y. Wang. A genetic algorithm for optimized feature selection with resource constraints in software product lines. *Journal of Systems and Software*, 84(12):2208 – 2221, 2011.

[21] J. Guo, D. Yang, N. Siegmund, S. Apel, A. Sarkar, P. Valov, K. Czarnecki, A. Wasowski, and H. Yu. Data-efficient performance learning for configurable systems. *Empirical Software Engineering*, 23(3):1826–1867, 2018.

[22] K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

[23] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[24] P. Jamshidi, N. Siegmund, M. Velez, C. Kästner, A. Patel, and Y. Agarwal. Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 497–508, 2017.

[25] M. Jorgensen and M. Shepperd. A systematic review of software development cost estimation studies. *IEEE Transactions on Software Engineering*, 33(1):33–53, 2007.

[26] D. Kingma and J. Ba. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference for Learning Representations (ICLR)*, 2015.

[27] D.R. Kuhn, R.N. Kacker, and Y. Lei. *Introduction to combinatorial testing*. CRC Press, 2013.

[28] M. Leshno, V.Y. Lin, A. Pinkus, and S. Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867, 1993.

[29] S. Liang and R. Srikant. Why deep neural network for function approximation? *Proceedings of the 5th International Conference on Learning Representation (ICLR)*, 2017.

[30] V. Nair and G.E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pages 807–814, 2010.

[31] V. Nair, T. Menzies, N. Siegmund, and S. Apel. Using bad learners to find good configurations. In *Proceedings of the 11th Joint Meeting on Foundations of Software*

Engineering (ESEC/FSE), pages 257–267, 2017.

[32] V. Nair, T. Menzies, N. Siegmund, and S. Apel. Faster discovery of faster system configurations with spectral learning. *Automated Software Engineering*, 25(2):247–277, Jun 2018.

[33] A.Y. Ng. Feature selection, l1 vs. l2 regularization, and rotational invariance. In *Proceedings of the 21st International Conference on Machine Learning (ICML)*, 2004.

[34] A. Nhlabatsi, R. Laney, and B. Nuseibeh. Feature interaction: The security threat from within software systems. *Progress in Informatics*, 5:75–89, 2008.

[35] F. Pedregosa and G. Varoquaux et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

[36] T. Poggio, H. Mhaskar, L. Rosasco, B. Miranda, and Q. Liao. Why and when can deep-but not shallow-networks avoid the curse of dimensionality: A review. *International Journal of Automation and Computing*, 14(5):503–519, 2017.

[37] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki. Cost-efficient sampling for performance prediction of configurable systems. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 342–352, 2015.

[38] A.S. Sayyad, J. Ingram, T. Menzies, and H. Ammar. Scalable product line configuration: A straw to break the camel's back. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 465–474, 2013.

[39] A.S. Sayyad, T. Menzies, and H. Ammar. On the value of user preferences in search-based software engineering: A case study in software product lines. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 492–501, 2013.

[40] N. Siegmund, A. Grebhahn, A. Apel, and C. Kästner. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 284–294, 2015.

[41] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner. Performance-influence models for highly configurable systems. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 284–294, 2015.

[42] N. Siegmund, S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 167–177, 2012.

[43] N. Siegmund, M. Rosenmüller, C. Kästner, P.G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable prediction of non-functional properties in software product lines. In *International Software Product Line Conference*, pages 160–169, 2011.

[44] J. Snoek, H. Larochelle, and R.P. Adams. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS)*, volume 2, pages 2951–2959, 2012.

[45] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[46] B. Steinbach and R. Kohut. Neural Networks: A Model of Boolean Functions. In *Proceedings of the 5th International Workshop on Boolean Problems*, pages 223–240, 2002.

[47] J. Sun, H. Zhang, Y. Fang, and L.H. Wang. Formal semantics and verification for feature modeling. In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 303–312, 2005.

[48] E. Thereska, B. Doebel, A.X. Zheng, and P. Nobel. Practical performance models for complex, popular applications. *ACM SIGMETRICS Performance Evaluation Review*, 38(1):1–12, 2010.

[49] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1996.

[50] A.N. Tikhonov and V.Y. Arsenin. *Solutions of Ill-Posed Problems*. Winston & Sons, Washington, D.C., 1977.

[51] L. Tóth. Phone recognition with deep sparse rectifier neural networks. In *Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6985–6989, 2013.

[52] H.H. Wang., Y.F. Li, J. Sun, H. Zhang, and J. Pan. Verifying feature models using OWL. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):117–129, 2007.

[53] D. Yarotsky. Error bounds for approximations with deep relu networks. *Neural Networks*, 94:103–114, 2017.

[54] H. Zhang, S. Jarzabek, and B. Yang. Quality prediction and assessment for product lines. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 681–695, 2003.

[55] Y. Zhang, J. Guo, E. Blais, and K. Czarnecki. Performance prediction of configurable software systems by fourier learning. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 365–373, 2015.