ACTGAN: Automatic Configuration Tuning for Software Systems with Generative Adversarial Networks

Liang Bao*[1], Xin Liu†[2], Fangzheng Wang* and Baoyin Fang*
*School of Computer Science and Technology, XiDian University, Xi'an, ShaanXi, China
†Department of Computer Science, University of California, Davis, Davis, California, USA
[1]baoliang@mail.xidian.edu.cn, [2]xinliu@ucdavis.edu

***Abstract*—Complex software systems often provide a large number of parameters so that users can configure them for their specific application scenarios. However, configuration tuning requires a deep understanding of the software system, far beyond the abilities of typical system users. To address this issue, many existing approaches focus on exploring and learning good performance estimation models. The accuracy of such models often suffers when the number of available samples is small, a thorny challenge under a given tuning-time constraint. By contrast, we hypothesize that good configurations often share certain hidden structures. Therefore, instead of trying to improve the performance estimation of a given configuration, we focus on capturing the hidden structures of good configurations and utilizing such learned structure to generate potentially better configurations. We propose ACTGAN to achieve this goal. We have implemented and evaluated ACTGAN using 17 workloads with eight different software systems. Experimental results show that ACTGAN outperforms default configurations by 76.22% on average, and six state-of-the-art configuration tuning algorithms by 6.58%-64.56%. Furthermore, the ACTGAN-generated configurations are often better than those used in training and show certain features consisting with domain knowledge, both of which supports our hypothesis.**

***Index Terms*—software system, automatic configuration tuning, generative adversarial networks**

I. Introduction

Many software systems, such as middleware, databases, and Web servers, provide a large number of parameters for users to configure in order to achieve desirable functional behaviors and non-functional properties (e.g., performance and cost). For example, both Spark [1] and HBase [2] have 200+ parameters that a user can configure, among which 10-15 are considered critical [3], [4], [5].

Performance (e.g., execution time, throughput, requests per second) is one of the most important non-functional properties [6]. Previous studies have shown that configurations have a significant impact on performance [3], [7], [4], [8], [5], [6], because software systems are complex with many configurable options that control nearly all aspects of their runtime behaviors. With this complexity, tuning a large number of configuration parameters requires a deep understanding of the target system and has far surpassed the abilities of typical system users [9]. As a result, users often (have to) accept the default settings. Alternatively, organizations may choose to hire human experts to configure their software systems. Unfortunately, manual configuration is labor-intensive, time-consuming, and often suboptimal. Therefore, there is a dire need for automatic configuration tuning on software systems.

One intuitive approach of automatic configuration tuning is to measure the performance of all configurations of a system to identify the configuration with the best performance. Unfortunately, this is usually infeasible because of the combinatorial nature of configuration parameters and the high-dimensional configuration space. Specifically, there are three main challenges in configuration tuning of complex software systems:

Heterogeneity. Software systems that require configuration tuning are highly heterogeneous, including message system like Kafka [10], data analytic systems like Spark [1] and Hive [11], database systems like Redis [12], MySQL [13], Cassandra [14] and HBase [2], or Web servers like Tomcat [15]. Performance metrics also differ, including throughput, execution time, transactions per minute, etc. Furthermore, application workload can vary significantly. For example, Figure 1 illustrates the heterogeneity of performance surface under only two parameters. We can see that Redis and Hive have bumpy performance surfaces, while Kafka has a relatively smooth performance surface.

Complexity. Given different performance metrics and different workloads, a deployed software system can have highly complex performance surfaces for a given set of configuration parameters, as shown in Figure 1. It often takes an experienced system developer months of efforts to reason about the underlying interactions between parameters [4], let alone novice users.

Costing evaluation. It is often costly to evaluate the performance of an individual configuration in a large software system. Because no performance simulator exists for general systems, performance evaluation often requires real experiments on production systems. Such an evaluation is not only time-consuming (e.g., executing a complex benchmark takes minutes or hours) but also expensive (e.g., running a real system for one minute may cost hundreds of dollars). Hence, obtaining performance samples, i.e. the performance results with different configurations, is costly.

Existing automatic configuration tuning methods include model-based, simulation-based, search-based, and learning-

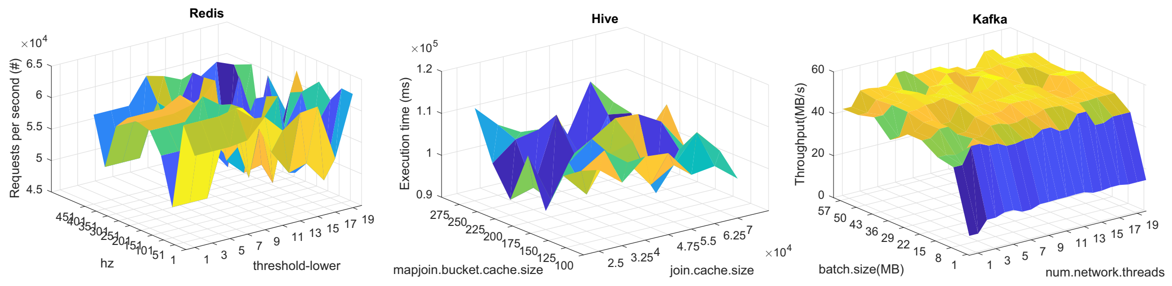


Figure 1. Performance surfaces of three different software systems

based, as discussed in Section II. Among them, learning-based tuning has received much recent attention. The key idea of learning-based tuning is to construct a performance prediction model using training samples of different configurations, and then explores better configurations using some searching algorithms. Although previous studies on learning-based tuning show promising results, one critical drawback is that it requires a lot of samples to train an accurate prediction model. For example, previous study [16] has shown that a typical learning-based approach needs about 2000 samples to obtain a good prediction model with 10 configuration parameters. Apparently, the assumption of the abundance of samples used in many learning-based methods is invalid considering the constrained time for tuning. Given the limited tuning time in practice, only a small number of samples can be obtained. It typically results in a less accurate prediction model that jeopardizes the exploration for better configurations.

To overcome these challenges, we propose ACTGAN (Automatic Configuration Tuning using Generative Adversarial Network) that aims to automatically generate good configurations for different software systems under a given time constraint. We approach the configuration tuning for software system (CTSS) problem in an unconventional manner – our hypothesis is that **good configurations share certain hidden structures**. Therefore, instead of trying to improve the performance estimation of a given configuration, we focus on **capturing the hidden structures of good configurations and using such hidden structures to generate potentially better configurations**.

We choose GAN (Generative Adversarial Network) for this purpose because: 1) GAN's inherent game-like structure forces the neural networks to seek important features hidden in good configuration samples; 2) it circumvents the need to obtain a highly accurate prediction model that is typically required a significant number of samples, and thus is more practical under the limited tuning time; and 3) GAN allows us to generate (better) configurations leveraging the learned hidden structures.

In summary, our work makes the following contributions:

- We introduce and formulate the CTSS problem, and model it as a *combinatorial optimization* problem.
- We propose ACTGAN, a novel approach to address CTSS, from an unconventional direction. ACTGAN can recommend promising configurations by directly learning and utilizing the hidden structures of existing good configurations, without the need of learning a performance prediction model.
- We evaluate the performance of ACTGAN through extensive experiments using 17 workloads under eight different software systems. We show that ACTGAN outperforms default configurations by 76.22% on average, and six state-of-the-art tuning algorithms by 6.58%-64.56%.

II. Related Work

Automatic configuration tuning for software systems has received much attention from both industry and academia. We classify the previous studies on this problem into four categories: model-based, measurement-based, search-based, and learning-based configuration tuning approaches, where the latter two are most recent and relevant to our work.

Model-based configuration tuning. The key to model-based configuration tuning is to construct an analytical performance model for a software system on the early stage of system development [17], see [18] and [19] for reviews. Model-based approaches rely on domain specific knowledge, mathematical theories or software architecture abstraction. They can be labor-intensive, imprecise, and difficult to evolve.

Measurement-based configuration tuning. Measurement-based configuration tuning aims to derive performance models for software systems by statistical inferencing based on benchmarked measurement data [20]. They collect program profiles to identify performance bottlenecks, which often fail to capture the overall program performance [21]. Also, most of these approaches lack generality [22].

Search-based configuration tuning. Search-based approaches regard configuration problem as a black-box optimization problem and use search algorithms to solve it, such as in [23], [24], [25]. The search strategies include recursive random search (RRS) heuristics [26], [16], evolutionary algorithms [27], [28], hill-climbing algorithms [29], and recursive bound & search [4].

Search-based configuration is simpler and more general compared to other approaches, because it takes the target software system as a black-box function and does not need detailed information about the internals. However, they fail to exploit the knowledge of already-known good configurations and need to search for the parameter space more times to obtain good configurations [30].

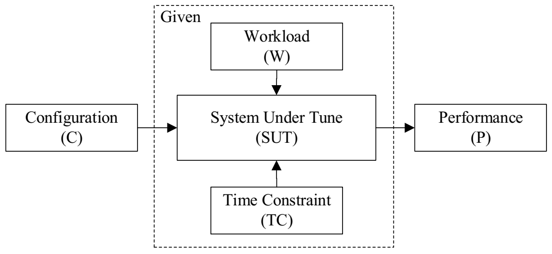


Figure 2. Overview of *CTSS* problem

Learning-based configuration tuning. The most relevant to our work is learning-based configuration tuning. These approaches try to construct performance prediction models first by observing a collection of running results under different configurations, and then apply some search algorithms to find the optimal configuration based on these models. The methods and models often used include regression [31], [32], [33], [34], SVM/SVR [35], [36], CART [6], Fourier learning (FL) [37], random-forest [38], [5], artificial intelligence (AI) planning [39], spectral learning [40], Classification and Regression Training (Caret) [41], Bayesian optimization [42], [43], [44], [30], reinforcement learning [45], comparison-based model [46], and transfer learning [47], [48].

Learning-based approaches require considerable numbers of samples to construct a good performance prediction model for a software system [25]. The high demand of samples is challenging here because only a limited set of samples can be acquired under a given time constraint.

III. Problem Statement

In this paper, we study *C*onfiguration *T*uning for *S*oftware *S*ystems (*CTSS*). As illustrated in Figure 2, for a given system and a specific workload, the goal of *CTSS* is to search for an optimal configuration to achieve the best performance by exploring the high-dimensional parameter space within a limited time period. We call this process of selecting configuration settings as *configuration tuning* (or just *tuning*).

More specifically, a *CTSS* problem can be defined with the following components.

System Under Tune (*S*). Many software systems, such as data analytic framework, databases, and Web servers, provide a large number of parameters for users to configure. The subject system that we want to tune is called the system under tune (SUT) [4], and is represented as S.

Configuration (*C*). We represent a configuration of an SUT as a set C=$(c_1, c_2, \cdots, c_n)$ of n parameters, and the value of each parameter c_i must be within *CB*, the *configuration bound* predefined by the SUT. Each configuration C_i of an SUT is represented as an n-tuple, assigning a valid value to each parameter in C.

Workload (*W*). A workload W represents a specific task running on an SUT. Example workloads include an application on Spark, a set of transactions on MySQL, or a group of requests on Tomcat. In the *CTSS* problem, we assume the workload W is given. Furthermore, we can measure the performance of the SUT under a given configuration C by simply executing the W on SUT under C, and record its performance.

Performance (*P*). Performance is an essential non-functional property of an SUT that can directly affect user perception and running cost. It is the optimization goal of our *CTSS* problem. We treat the performance $P(\cdot)$ as a blackbox function, and use the $P(S, W, C)$ to denote the performance value, given an SUT S, a workload W, and a configuration C. Different SUTs have different performance goals for configuration tuning: for a data-access workload on MySQL, the performance goal is to increase the *throughput*, while for a data analytical job on Spark, it would be to reduce the *execution time*. Without loss of generality, we assume that we want to maximize the performance value over all configurations.

Time Constraint (*TC*). In a practice, the time for configuration tuning is often restricted. We define this restricted tuning time as *time constraint*, denoted as TC. Any solution to the *CTSS* problem must complete within TC.

In summary, the *CTSS* problem can be defined as follows:

$$\max_{C \in CB} P(S, W, C) \tag{1}$$

$$\textit{s.t.}\ \text{tuning time} \leq TC \tag{2}$$

where (1) states that given an SUT S and a workload W on S, the goal of *CTSS* is to find a configuration C among all valid configurations of S that maximize the performance. The constraint (2) is that any solution to the problem must terminate after a TC amount of tuning time.

This definition shows that the goal of *CTSS* is to search for an optimal configuration of a set of parameters to maximize the performance. According to a previous study [49], *CTSS* is essentially an instance of classic combinatorial optimization (CO) problems [50], which is known to be NP-complete. The NP-completeness proof by restriction is established in [51]. Therefore, a naive exhaustive search solutions would not be practical due to the high dimensionality of parameter space and the combinatorial nature of brutal force search.

IV. ACTGAN for CTSS Problem

In this section, we introduce ACTGAN – a generative adversarial network (GAN) model to solve the *CTSS* problem. Its key idea is to capture the hidden structures of good configurations by learning how already-known good configurations are distributed in a high-dimensional configuration space. In the following, we first analyze existing tuning approaches and their limitations. We then present ACTGAN, a novel approach, which solves the CTSS problem in an unconventional manner. Finally, we discuss the details of the ACTGAN algorithm.

A. Motivation

For a given SUT, finding a good configuration is challenging because of the high-dimensionality of the configuration space, the limited amount of tuning time, and the lack of a priori knowledge about its performance surface. To address these challenges, the following approaches have been considered.

Recently, learning-based approaches are the most popular ones for addressing the *CTSS* problem. The key step is to construct a performance prediction model for an SUT using collected performance samples under different configurations. However, given the time constraint for tuning, one can obtain only a limited number of samples because it often takes minutes or hours to evaluate the performance of an individual configuration on an SUT. Having insufficient samples leads to an inaccurate prediction model [25] and directly affects the tuning performance [4].

Bayesian optimization (BO) based approaches are also popular ways to optimize objective functions that are costly to evaluate [30]. They build a surrogate for the objective, quantify the uncertainty in that surrogate using Gaussian process regression, and then use an acquisition function defined from this surrogate to decide where to sample [52]. Essentially, BO uses all the information available from previous evaluations of the objectives instead of relying simply on local gradient and Hessian approximations. While BO methods often have good convergence guarantees, their short-term performance under a limited number of samples is not always desirable in practice because they may spend too much time exploring uncertain spaces.

Search-based approaches, such as random search, on the other hand, take the SUT as a black-box function and do not need the detailed information about its internals. However, because they fail to exploit the knowledge of already-known good configurations, they have to explore the high-dimensional configuration space more times in order to obtain the optimized configuration [30]. Other heuristic algorithms, such as genetic algorithm (GA), particle swarm optimization (PSO), etc., are able to handle high-dimensional combinatorial optimization problems [53]. Unfortunately, they often need at least thousands of evaluations to find good configurations with 10–20 dimensions according to our experiments, which is infeasible in our time-constrained *CTSS* scenario.

Most of the above approaches focus on improving the performance estimation/modeling of a configuration under a limited number of samples. By contrast, our hypothesis is that **good configurations often share certain hidden structures**. Therefore, instead of trying to improve the performance estimation of a given configuration, we focus on **capturing the hidden structures of good configurations**. In the literature, GANs have been shown to be a powerful tool for finding hidden structures, which is why we adopt them here to address the CTSS problem.

Figure 3 illustrates the automatic configuration tuning process of ACTGAN. ACTGAN simultaneously trains two models: a generative model G that captures the hidden structures of the already-known good configurations from the training examples, and a discriminative model D that estimates the probability that a sample comes from the training data rather than from G. More specifically, we use the random sampling method to generate a set of configurations, test them on

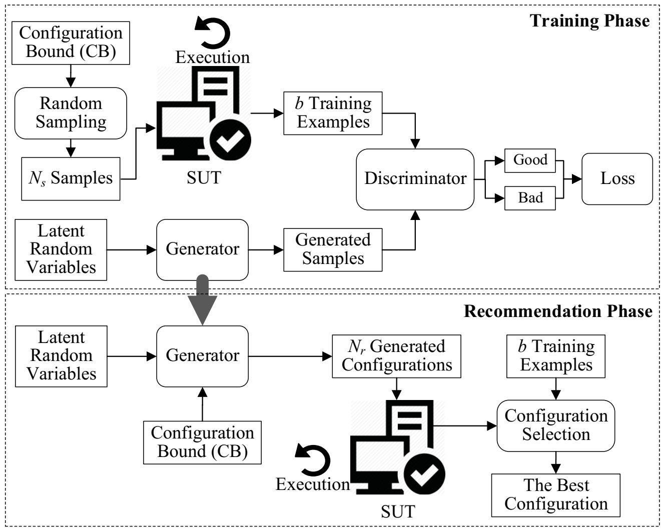


Figure 3. Overview of ACTGAN Approach

the SUT, and select a subset of configurations, i.e. *training examples*, with the best performance. Using this training set, we train a GAN with a discriminator and a generator network. The GAN structure forces the networks to learn the hidden structures of good configurations. To utilize the hidden structures, we use the trained generator to generate a set of potentially good configurations. Finally, we test these configurations on the SUT and select the best one.

In summary, we propose an innovative way of utilizing the GAN structure for *CTSS* that **uncovers and utilizes the hidden structures of good configurations**.

B. ACTGAN Algorithm

Using the tuning process shown in Figure 3, we now discuss the ACTGAN algorithm in Algorithm 1.

Estimating the sample size. Given a tuning time constraint TC, an SUT S consisting of C parameters, and a specific workload W running on S, let $t(C)$ be the execution time under the configuration C. The execution time of two different configurations may differ significantly. For example, the running time of a *pagerank* workload on Spark using the default configuration is 440s, this time falls to 242s under an optimized configuration. Suppose the tuning time constraint is TC and the number of recommended configurations from G is N_r (we need to run S with these N_r configurations to find the best one, see line 18); then we can use the running time under default configuration to make a conservative estimate of the sample size N_s (line 1–2).

Random sampling. Given TC, S, C, W, and a set *BC* representing the configuration bound for every parameter, ACTGAN first generates a set of configurations T from C within *BC*. This collection of configurations serves as a training set for our GAN model. We use the *random sampling* (RS) strategy, as it can effectively search for a larger configuration space, especially when configuration parameters are not equally important [23] (line 3).

Algorithm 1 ***ACTGAN(S, W, C, TC, CB)***

```
Input: S: the target SUT; W: workload; C: configuration; TC: time
    constraint; CB: configuration bounds.
Variables: $T$: the set of initial samples, $|T| = N_s$; $R$: the set of
    potentially good configurations recommended by $G$, $|R| = N_r$.
 1: $t(C_0) \leftarrow$ the time of running $S$ using the default configuration $C_0$;
 2: $N_s = \lfloor \frac{TC}{t(C_0)} \rfloor - N_r$;
 3: $T \leftarrow RS(N_s, S, W, C, CB)$;
 4: $B \leftarrow$ a set of $b$ configurations with the best performance from $T$;
 5: for number of training iterations do
 6:     for any $k$ steps do
 7:         Sample minibatch of $m$ noise samples $\{z^{(1)}, z^{(2)}, \cdots,$
 8: $z^{(m)}\}$ from noise prior $p_g(z)$;
 9:         Sample minibatch of $m$ configurations $\{x^{(1)}, x^{(2)}, \cdots,$
10: $x^{(m)}\}$ from $B$;
11:         Update the discriminator by ascending its stochastic gradient:
                $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))]$
12:     end for
13:     Sample minibatch of $m$ noise samples $\{z^{(1)}, z^{(2)}, \cdots,$
14: $z^{(m)}\}$ from noise prior $p_g(z)$;
15:     Update the generator by descending its stochastic gradient:
                $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} [\log(1 - D(G(z^{(i)})))]$
16: end for
17: $R \leftarrow N_r$ configurations recommended by $G$;
18: Run workload $W$ on $S$ with each configuration in $R$;
19: Select the configuration $C^*$ with the best performance in $R \bigcup B$;
20: return $C^*$;
```

GAN architecture design. After obtaining T, a subset B of T consisting of the best b configurations are selected to serve as the training set for the discriminator D of the ACTGAN (line 4). Specifically, we model both the generator G and the discriminator D using three-layer feedforward neural networks (NNs) with one hidden layer. Simple neural networks work here because the performance surfaces of SUTS are often continuous (although not necessarily smooth) [4], so NNs can capture the hidden structures of the performance surfaces. We note that the original convolution neural network (CNN) structured GAN works well for image processing, because such convolution operators generally capture intensity, gradient, and edge characteristics of images. However, in our case, CNNs fail to capture the hidden structures among the parameters for a given SUT. The detailed architecture of ACTGAN is discussed in the Section V-B.

Model training. At any step of the training process, the generator G takes as input a Gaussian noisy source $z \sim p_g(z)$ and produces a set of synthetic configurations $\{z^{(1)}, z^{(2)}, \cdots, z^{(m)}\}$ that aims to follow a unknown target data distribution for good configurations (line 7–8, 13–14). Meanwhile, the discriminator D takes a subset of already-known good configurations $\{x^{(1)}, x^{(2)}, \cdots, x^{(m)}\}$ as input and learns to maximize the probabilities that z is fake and x is real.

The min-max adversarial loss for training the generator network G and discriminator network D is formulated as:

$$\min_G \max_D \mathbb{E}_{x\sim p(x)}[\log D(x^{(i)})] + \mathbb{E}_{z\sim p_g(z)}[\log(1 - D(G(z^{(i)})))]$$

where we train D to maximize the probability of identifying the good configurations to both training examples and samples from G, and simultaneously train G to minimize $log(1 - D(G(z)))$. The goal of this process is to learn the hidden distribution of already-known good configurations $p(x)$ via G, and generate realistic fake good configurations (line 11).

When we train G given D, we minimize the following loss of G (line 15):

$$\mathbb{E}_{z\sim p_g(z)} \min_G [\log(1 - D(G(z^{(i)})))]$$

In our ACTGAN algorithm, we adopt ADAM [54], an optimized mini-batch stochastic gradient descent (SGD) method, to update the parameters in the networks in an efficient and stable way.

Configuration recommendation. After finishing the training, we use the generator G to construct a set of N_r recommended configurations R and run them on the SUT. Finally, we select the best configuration in $R \bigcup B$, where B is the best configurations generated using random sampling method. (line 17–20).

V. ACTGAN Implementation

A. Data preparation

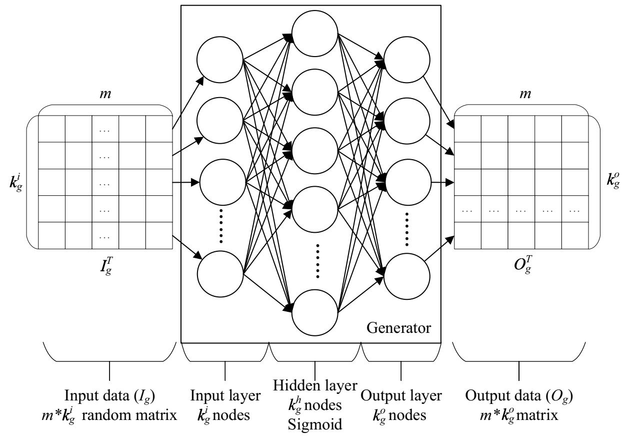
Categorical data processing. Categorical variables are parameters that contain label values rather than numeric values, and the number of possible values is often limited to a fixed set. Categorical variables are common in software configurations. For example, there are two categorical variables, i.e. *spark.io.compression.codec*= $\{$*snappy, lz4, lzf*$\}$ and *spark.serializer*=$\{$*JavaSerializer, KryoSerializer*$\}$, of Spark framework in our experiments. Because our neural networks cannot operate on categorical data directly, they require all input variables and output variables to be numeric.

There are two encoding methods, i.e integer encoding and one hot encoding, that can convert categorical data to numerical data. In CTSS problem, categorical variables often have no ordinal relationship, using integer encoding and allowing the model to assume a natural ordering between categories may result in poor performance or unexpected results. In this case, we use one hot encoding to add binary variable for each unique categorical value. In the "*spark.io.compression.codec*" variable example, there are three categories and therefore three binary variables are needed. A "1" value is placed in the binary variable for the used compression options and "0" values for the other options, as shown in Table I.

Data normalization. Software configuration parameters are often in highly different ranges. Take the Web server Tomcat for example, the *# of minimal processors* parameter ranges from 1–100 in our experiments, while the *connection timeout*

Table I
ONE HOT ENCODING FOR *spark.io.compression.codec*

snappy	lz4	lzf
1	0	0
0	1	0
0	0	1



m
k_g^i
I_g^T
Generator
m
k_g^o
O_g^T
Input data (I_g) $m*k_g^i$ random matrix
Input layer k_g^i nodes
Hidden layer k_g^h nodes Sigmoid
Output layer k_g^o nodes
Output data (O_g) $m*k_g^o$ matrix

Figure 4. The generator architecture of ACTGAN

ranges from 0–30,000 and higher. Further analysis shows that the parameter *connection timeout* intrinsically influences the result more due to its larger value. But this doesn't necessarily mean it is more important as a predictor.

To address problem, we use *standard score* to normalize the dataset. More specifically, let x be the original value of a parameter, z be the normalized value, we have:

$$z = \frac{x - \mu}{\sigma}$$

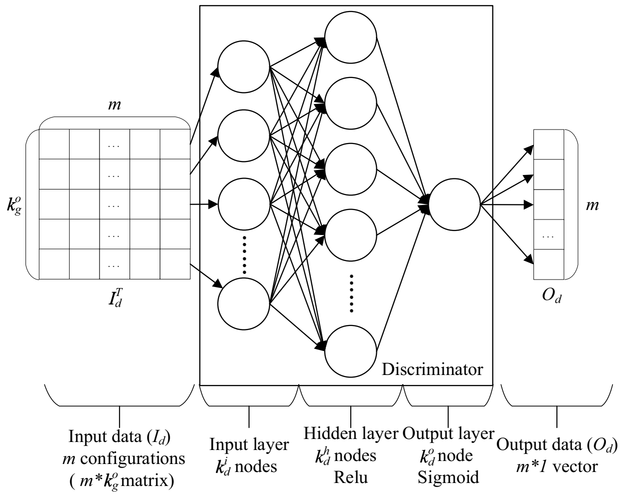
where μ is the mean value of the parameter, and σ is the standard deviation of the parameter values.

B. GAN Architecture

Like any typical GAN architecture, ACTGAN consists of two main components – a generator G and a discriminator D. The goal of the generator is to generate synthetic configurations that are plausible in the input (good) configurations. At the same time, the discriminator learns to distinguish the suboptimal configurations from the good ones that come from the training set. Both G and D are trained end-to-end using *backpropagation*.

At each step of the training process, ACTGAN uses G to generate a set of synthetic configurations, takes one configuration at a time to feed as D's input. D outputs a single score that represents the probability of the configuration being real. An overview of ACTGAN's detailed architecture of the generator and the discriminator can be seen in Figure 4 and 5, respectively.

Generator. The generator G defines an implicit probabilistic model for generating good configurations: $C = \{c_1, c_2, \cdots, c_n\} \sim G$. We model G as a three-layer feedforward neural network. More specifically, G consists of an



m
k_g^o
I_d^T
Discriminator
m
O_d
Input data (I_d) m configurations ($m*k_g^o$ matrix)
Input layer k_d^i nodes
Hidden layer k_d^h nodes Relu
Output layer k_d^o node Sigmoid
Output data (O_d) $m*1$ vector

Figure 5. The discriminator architecture of ACTGAN

input layer of non-computational units (one for each input), one hidden layer of computational units, and an output layer of computational units. The key procedure of G is to pass the input signal forward and the error signal backward through the network.

In the forward pass, a latent $m \times k_g^i$ matrix I_g (represents m noise samples) drawn from a multivariate Gaussian distribution is presented to the input layer, and the input signal is filtered forward through the subsequent hidden layer (with k_g^h nodes) of the network to generate a set of outputs. The network outputs, i.e. a set of m synthetic configurations (a $m \times k_g^o$ matrix O_g), are fed into the discriminator D and the possibility value for each configuration can lead to good performance is generated an error signal.

In the backward pass, the synaptic weights are updated using the learning constant according to the effect they have in generating the incorrect outputs. Figure 4 demonstrates the passing of noise prior I_g through a neural network and the structure of our three-layer network. The presentation of noise samples to the network is repeated in each minibatch until the network is stabilized and no more adjustments are required for the synaptic weights, or the maximum number of epochs has been reached.

As described in Table III, the number of input neurons for G is set to 5 (k_g^i=5); the number of hidden neurons for G is set to 128 (k_g^h=128); and the number of output neurons for G is set to n ($k_g^o = n$), i.e. the number of parameters for a given SUT.

Discriminator. The discriminator D is also a three-layer feedforward neural network with a hidden layer consisting of k_d^h nodes. At every time step, a $m \times k_g^o$ matrix I_d, denoting m configurations generated by G (i.e. $k_d^i = k_g^o = n$), is fed as input. After processing these configurations, the discriminator outputs a vector of m scalar values that represents the probability of each synthetic configuration being good.

As shown in Table III, the number of initial training samples fed into D is set to 32 (b=32); the number of input neurons for D is set to n ($k_d^i = n$), i.e. the number of parameters for a given SUT; the number of hidden neurons for D is set to 128 (k_d^h=128); and the number of output neurons for D is set to 1 (k_d^o=1).

C. Model Training

To stabilize our model training procedure, we modify the negative log likelihood objective by clipping the values between 0 and 1. This loss performs more stably during training and generates higher quality results. We use the ADAM optimizer [54] with a momentum value of 0.5. All networks were trained from scratch with a learning rate of 0.0001 and the total number of epochs are set to 150,000 for all 17 workloads. Finally, we apply instance normalization with batch size of 16 (m=16). We tried the conventional value 64 of batch size at the first place but observed the model oscillation of G and D during the training process, and finally found out that the 16 is an appropriate value for the *CTSS* problem after many attempts.

VI. Experiments

We have implemented the ACTGAN and other algorithms, and conducted extensive experiments under diverse workloads. The source code and the data can be found in the online anonymous repository: https://github.com/anon4review/ACTGAN. In this section, we first describe our experiment setup, and then present the experimental results to prove the efficiency and effectiveness of the proposed approach.

A. Experimental Settings

Running environment. We conduct our experiments on a local cluster of 5 physical servers. Each server is equipped with two 8-core Intel Intel XeonE5-2650v2 2.6GHz processors, 256GiB RAM, 1.5TB disk, and running CentOS 6.0 and Java 1.7.0_55. All of servers are connected via a high-speed 1.5Gbps LAN. To avoid interference and comply with the actual deployment, we run the SUTs, the workload generators and the ACTGAN algorithm on different physical servers at each experiment.

SUTs, benchmarks, parameters, and performance metrics. We choose eight widely used software systems to evaluate ACTGAN, namely Kafka [10], Spark [1], Hive [11], Redis [12], MySQL [13], Cassandra [14], HBase [2] and Tomcat [15]: Kafka is a distributed message system (DMS) for publishing and subscribing to streams of records; Spark is a cluster computing engine for big data analytics; Hive is a data warehouse software that manages large datasets residing in distributed storage using SQL; Redis is an open-source in-memory data structure store; MySQL is an open-source relational database management system (RDBMS); Cassandra is an open-source column-oriented NoSQL database management system; HBase is a distributed and scalable big data store; and Tomcat is an open source implementation of the Java Web technologies.

We use six customized workloads for Kafka; HiBench [55] for Spark; YCSB [56] for Hive, Cassandra and HBase; Redis-Bench[1] for Redis; TPCC[2] for MySQL; and JMeter [57] for Tomcat. For each SUT, we use domain expertise to identify 10–14 parameters that are considered critical to the performance, as in [4], [5], [31]. Note that even with only these parameters, the search space is still enormous, and exhaustive search is infeasible. The reason we choose a small subset of parameters that have great impact on performance instead of all configurable parameters is because reducing the number of tuning parameters can reduce the search space exponentially, and most existing approaches [4], [5], [31] also adopt this manual feature selection strategy. Table II summarizes the SUTs along with the benchmarks, the numbers of tuned parameters, and performance metrics, respectively.

Baseline Algorithms and Hyperparameters. To evaluate the performance of ACTGAN, we compare it with six state-of-the-art algorithms, namely random search [23], BestConfig [4], RFHOC [5], SMAC [43], Hyperopt [44], and AutoConfig [46]. We provide a brief description for each algorithm as follows and report its hyperparameters (including ACTGAN) in Table III.

Random search (Random) is a search-based tuning approach that explores the configuration space uniformly at random [23].

BestConfig[3] is a search-based tuning approach that uses divide-and-diverge sampling and recursive bound-and-search algorithm to find the best configuration.

RFHOC is a learning-based tuning approach that constructs a prediction model using random forests, and a genetic algorithm to automatically explore the configuration space.

SMAC[4] is a learning-based tuning method using random forests and an aggressive racing strategy.

Hyperopt[5] is a learning-based tuning approach based on Bayesian optimization. It is widely used for hyperparameter optimization.

AutoConfig[6] is a learning-based tuning approach using a comparison-based prediction model and a weighted Latin hypercube sampling method.

For each run in our experiments, every algorithm is executed under the same time constraint and stops once the constraint is met. To ensure consistency, we run each workload five times and calculate the average of these five runs.

B. Experiment Results

Performance results. Given a fixed time constraint (about 316 running times using default configuration) for each SUT and workload, we run seven different tuning algorithms plus default configuration independently. Table IV lists the experiment results. As expected, the default configuration does not perform well. ACTGAN achieves an average of 76.22% improvement over the default configurations. Furthermore, ACTGAN outperforms all other six algorithms:

[1]https://redis.io/topics/benchmarks
[2]https://github.com/Percona-Lab/tpcc-mysql
[3]https://github.com/zhuyuqing/bestconf
[4]http://www.cs.ubc.ca/labs/beta/Projects/SMAC
[5]http://jaberg.github.io/hyperopt/
[6]https://github.com/sselab/autoconfig

Table II
SUTS, BENCHMARKS, PARAMETERS, AND PERFORMANCE METRICS

SUTs	Categories	Benchmarks	# of tuned parameters/ # of all parameters	Performance	Tuning Goal
Kafka	DMS	Customized	11/169	Throughput (TP) (MB/s)	*max* ↑
Spark	Data analytics	HiBench	13/227	Execution time (ET) (ms)	*min* ↓
Hive	Data analytics	YCSB	11/432	Execution time (ET) (ms)	*min* ↓
Redis	In-memory DB	Redis-Bench	10/95	Requests per second (RPS)	*max* ↑
MySQL	RDBMS	TPCC	13/46	Transactions per minute (TPM)	*max* ↑
Cassandra	NoSQL DB	YCSB	11/64	Operations per second (OPS)	*max* ↑
HBase	NoSQL DB	YCSB	10/202	Execution time (ET) (ms)	*min* ↓
Tomcat	Web server	JMeter	14/229	Requests per second (RPS)	*max* ↑

Table III
HYPERPARAMETERS FOR EACH ALGORITHM

Algorithm	Hyperparameters
Random	# of configurations: 316
BestConfig	InitialSampleSetSize: 50; RRSMaxRounds: 5; COMT2Iteration: 20; COMT2MultiIteration: 10; TimeOutInSeconds: 50
RFHOC	population size: 100; epoch: 100000; crossover parameters: 0.5; mutation probability: 0.015
SMAC	run_obj: quality; runcount_limit: 316; deterministic: true
Hyperopt	search algorithms: hyperopt.rand.suggest, hyperopt.anneal.suggest; max_evals: 316
AutoConfig	# of iterations: 100; α: 0.4; β: 0.6; h: 10; b: 5
ACTGAN[1]	N_s: 300; N_r: 16; epoch: 150000; learning rate: 0.0001; b: 32; m: 16; k_g^i: 5; k_g^h: 128; k_g^o: n; k_d^i: n; k_d^h: 128; k_d^o: 1

9.28%–53.99% improvement over Random, 7.21%–27.45% improvement over BestConfig, 9.52%–38.44% improvement over RFHOC, 10.41%–49.58% improvement over SMAC, 9.66%–64.56% improvement over Hyperopt, and 6.58%–34.59% over AutoConfig.

Finally, we plot the overall performance improvement percentage of BestConfig, RFHOC, SMAC, Hyperopt, AutoConfig, and ACTGAN on Kafka, Spark, and other six software systems in Figure 6, 7 and 8 respectively, using the random algorithm as the baseline. In each figure, x-axis lists the workloads and y-axis represents the performance improvement over the random algorithm. We observe that compared with the random algorithm, our approach achieves 9.28%–53.99% improvement among all workloads. ACTGAN achieves an average of 20.04% improvement over Random, 17.50% improvement over BestConfig, 31.48% improvement over RFHOC, 18.09% improvement over SMAC, 20.59% improvement over Hyperopt, and 14.32% improvement over AutoConfig. We can conclude from Figure 6, 7 and 8 that ACTGAN achieves stable and significant improvements compared with the other six algorithms. Another interesting observation is that the random search achieves surprisingly good results in our experiments. This is consistent with the findings of Bergstra and Bengio in [23].

In the above experiments, the number of the random sam-

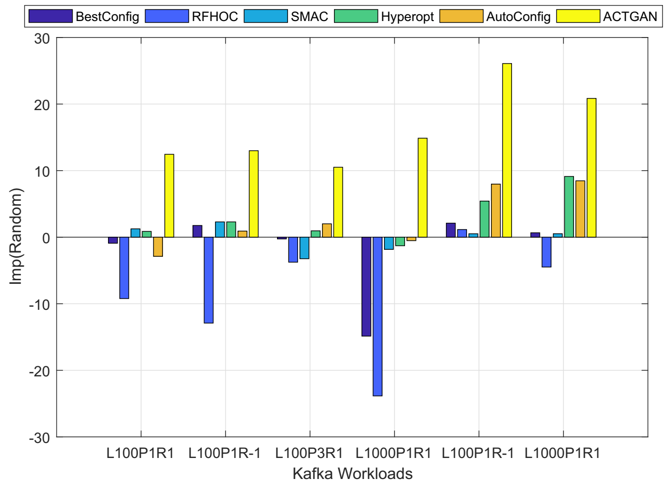


Figure 6. Performance comparison on six Kafka workloads with different algorithms

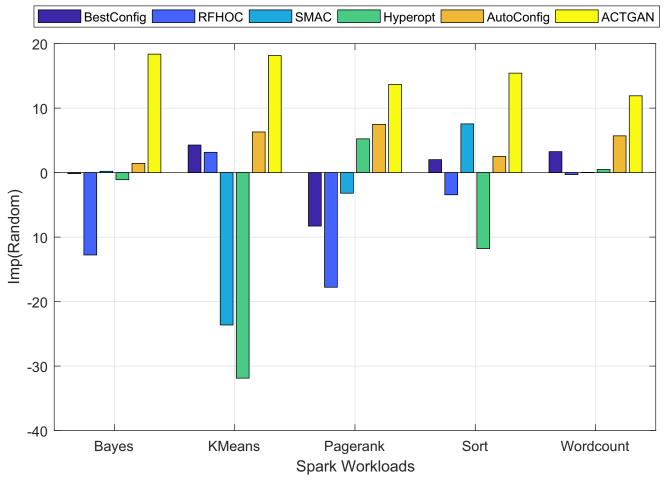


Figure 7. Performance comparison on five Spark workloads with different algorithms

Table IV
PERFORMANCE RESULTS FROM DIFFERENT ALGORITHMS UNDER FIXED TIME CONSTRAINTS

SUT/Perf./↑↓	Workload	TC (h)	Default	Random	BestConfig	RFHOC	SMAC	Hyperopt	AutoConfig	ACTGAN
Kafka/TP (MB/s)/↑	L100P1R1	15	16.41	20.81	20.62	18.89	21.07	20.99	20.21	23.40
	L100P1R-1	17.5	5.50	13.09	13.32	11.40	13.39	13.39	13.21	14.79
	L100P3R1	15	38.27	55.89	55.74	53.79	54.08	56.42	57.01	61.76
	L1000P1R1	15	27.15	32.62	27.77	24.84	32.02	32.20	32.45	37.47
	L1000P1R-1	17.5	8.37	15.68	16.01	15.86	15.76	16.53	16.93	19.77
	L10000P1R1	15	23.18	34.97	35.20	33.40	35.15	38.16	37.93	42.26
Spark/ET (ms)/↓	Bayes	15	150719	104626	104787	117983	104408	105776	103121	85402
	KMeans	17.5	1124410	278854	266915	270065	644736	367715	261267	228236
	Pagerank	36.7	440129	280736	303995	330637	289695	266026	259718	242334
	Sort	20	249135	192932	189047	199555	178346	215654	188092	163168
	Wordcount	12.5	479749	150827	145917	151290	150768	150084	142218	132858
Hive/ET (ms)/↓	YCSB	9.2	111256	94322	90900	111078	92375	92550	92910	83694
Redis/RPS/↑	Redis-Bench	1.5	82576	83633	97494	84947	100268	98032.2	100121	113145.6
MySQL/TPM/↑	TPCC	20	2735	2992	2950	2747	3518	2885	3106.5	4181
Cassandra/OPS/↑	YCSB	1.5	4088.31	12562.81	13550.14	8756.57	11750.88	11560.69	13148.52	14548.52
HBase/ET (ms)/↓	YCSB	1.5	14302	12233	15223	13282	12138	13508	12098	11098
Tomcat/RPS/↑	JMeter	4.6	6390	8243.57	11791.53	7381.80	8640.30	8111.18	9694.64	12694.44

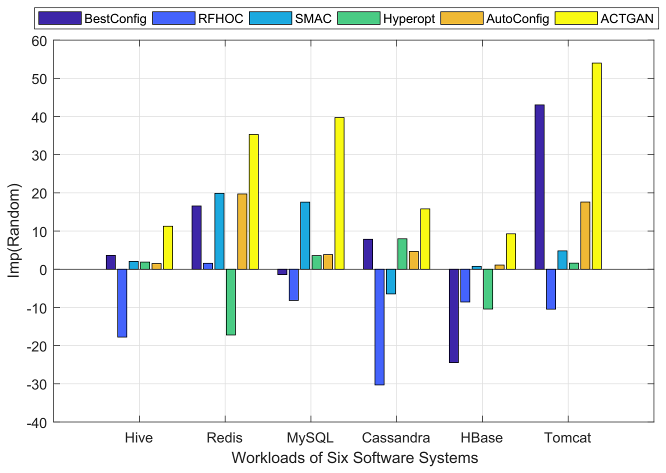


Figure 8. Performance comparison on six SUTs with different algorithms

ples N_s and the number of recommended configurations N_r are set to 300 and 16, respectively. We believe that the optimal values of these two hyperparameters are CTSS-specific, and can be trained. In our experiment, we have evaluated different values of N_s and N_r and observed that the results are not sensitive to these hyperparameters. For the ease of reporting, we choose to report the same set of values.

The quality of configurations. To evaluate the quality of configurations recommended by ACTGAN, we plot the normalized performance (the higher the better: i.e. "0" means the worst performance and "1" means the best) histogram over the initial training samples (blue bars) and the configurations generated by ACTGAN (red bars) for eight different software systems in Figure 9. We can see from Figure 9 that the generated configurations can achieve the best performance in all cases. The generated configurations in general outperform the training samples, a clear demonstration of the effectiveness of the ACTGAN approach.

Another interesting observation is that these ACTGAN-generated configurations incline to choose a fixed value on certain categorical parameters for a specific workload. For example, the Spark parameter *spark.serializer*={*JavaSerializer, KryoSerializer*} is all set to *JavaSerializer* for *Bayes*, *Sort* and *Wordcount* workloads, and for *KMeans* and *Pagerank* workloads it is all set to *KryoSerializer*. Such choices are in accord with domain knowledge and field experience obtained from long term software system operation and maintenance.

C. Threats to Validity

Internal validity: To increase the internal validity, we performed controlled experiments by executing each test case five times and calculate the average of these five runs. Such method can avoid misleading effects of specifically selected test cases and ensures the stability of the result. Besides, we set the same time constraint value suggested by users for all algorithms (including ACTGAN) in each test case and each algorithm is forced to stop when the time constraint is met. Such results are considered to be fair and reliable. In addition, we set the values of hyperparameters for each compared algorithm as suggested by their authors (see Table III). Finally, we have tried multiple values of hyperparameters for ACTGAN in our experiments and observed that the good values of these hyperparameters that can lead to better results are almost the same from test case to test case.

External validity: We increase the external validity by choosing eight different SUTs including one message system (Kafka), two big data processing systems (Hive and Spark), four database systems (Redis, MySQL, Cassandra, and HBase), and one Web application server (Tomcat). Furthermore, we choose 17 different workloads on these SUTs and among them six testing scenarios with various application-level parameters are for Kafka, and five HiBench workloads are for Spark. Finally, we are aware that because our ACTGAN is a general black-box approach and is independent to the SUTs, the results of our evaluations are transferable to other software systems.

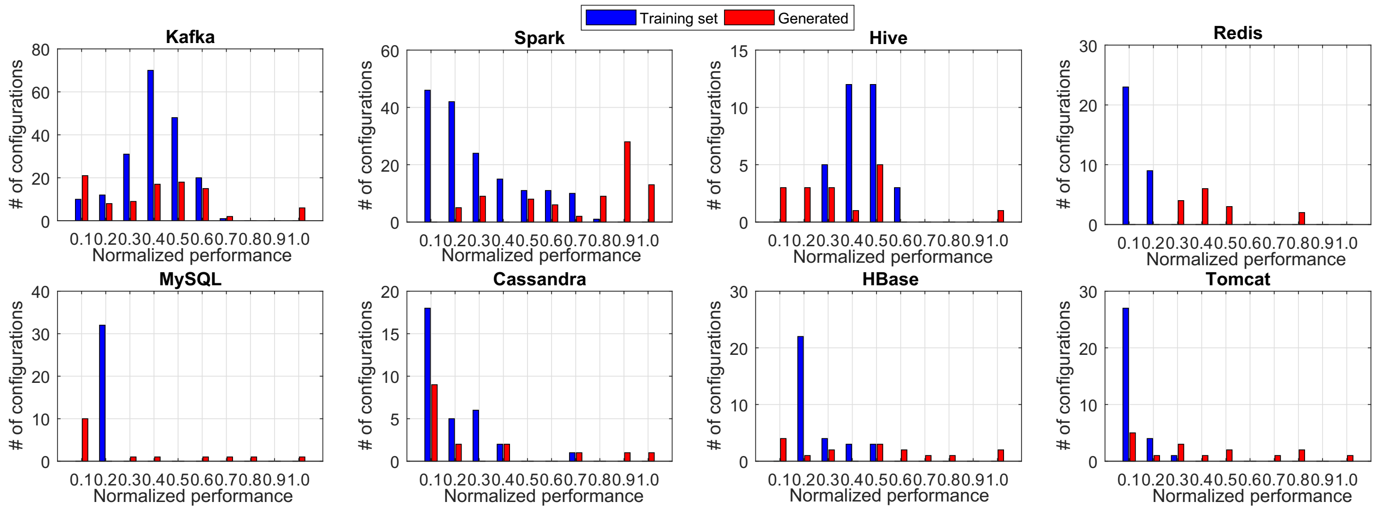


Figure 9. Normalized performance distributions between the initial training set and the ACTGAN generated configurations for eight different software systems

D. Discussion

Our work builds upon the hypothesis that good configurations share hidden structures. Although we could not prove this hypothesis mathematically, our results clearly support this hypothesis. First, our extensive experimental evaluations show that the ACTGAN generates configurations are better than the ones used for training, which suggests that ACTGAN successfully uncovers hidden structures of good configurations. Furthermore, the above observation of ACTGAN "choosing a fixed value on certain categorical parameters for a specific workload" also supports our hypothesis because it is consistent with domain knowledge.

The more interesting question is **why ACTGAN could improve upon training samples.** An analogy is as follows: suppose we feed real pictures of "good-looking" people into a GAN and train it to generate artificial "good-looking" faces. Psychological studies suggest that a face being symmetry is important to "good-looking", a hidden structure that GAN needs to learn. After learning this structure, GAN can generate perfectly symmetric faces. Given no real human faces are perfectly symmetric, the generated pictures are "better", in terms of the hidden structure of symmetry. In the *CTSS* scenario, our intuition is that **ACTGAN can identify such hidden structures in good configurations and then generate configurations that inherit such hidden structures while avoid (some of) the "imperfectness" in existing training samples, which are generated and selected from random sampling**. Therefore, in the end, ACTGAN improves upon the training samples.

We note that given the limited tuning time, **no** algorithm can explore the entire the high-dimensional parameter space. Random sampling is a simple yet robust mechanism to find good configurations without prior knowledge, and thus chosen here. Other effective learning schemes can be easily adopted to replace random sampling to generate the initial set of good training samples. Therefore, our ACTGAN can complement and augment existing learning algorithms.

We also note that because of the high dimensional configuration space and limited tuning time, it is possible that obtained training samples do not contain all hidden structures of an optimal configuration, and thus ACTGAN could not learn all them. This is an inherent limitation that all algorithms are likely to suffer in a high-dimensional non-convex general optimization problem.

VII. Conclusion and Future Work

In this paper, we propose ACTGAN, a novel approach to address CTSS. ACTGAN can recommend promising configurations by directly learning and utilizing the hidden structures of existing good configurations. The performance superiority of ACTGAN has been illustrated by extensive real-system evaluations in comparison to six state-of-the-art algorithms.

ACTGAN is a first attempt to leverage GAN structure to solve CTSS and future work is multifold. First, it is well known that GAN, although a powerful tool to identify hidden structures, has its limitations, such as sensitivity to mode dropping and mode collapsing. It is ongoing research to identify and address these limitations. Our future work is to study how these limitations affect ACTGAN and its performance on CTSS. Furthermore, most existing work on addressing the limitations of GAN focuses on image processing. How can we adapt these studies to CTSS, e.g., how do we quantify and identify mode dropping in CTSS? CTSS has unique features such that existing solutions may or may not work effective, and thus require further investigation.

The holy grail is to leverage ACTGAN to identify and interpret the hidden structures of good configurations. Our observation of fixed values on certain categorical parameters seems to be promising. However, this is a challenging task that requires much domain knowledge and significant future work.

It is also our future work to refine our ACTGAN approach to adaptive adjust the hyperparameters (N_r and N_s) based on time constraint, SUT, and workload. We will also investigate the performance dynamics of more software systems and improve our GAN structure to obtain better configurations.

References

[1] spark, "http://spark.apache.org," Accessed on Decemeber 20th 2018.

[2] HBase, "https://hbase.apache.org/," Accessed on July 31th 2018.

[3] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A self-tuning system for big data analytics." in *CIDR*, vol. 11, 2011, pp. 261–272.

[4] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "Bestconfig: tapping the performance potential of systems via automatic configuration tuning," in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 338–350.

[5] Z. Bei, Z. Yu, H. Zhang, W. Xiong, C. Xu, L. Eeckhout, and S. Feng, "Rfhoc: A random-forest approach to auto-tuning hadoop's configuration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1470–1483, 2016.

[6] 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*, vol. 23, no. 3, pp. 1826–1867, 2018.

[7] C. Liu, D. Zeng, H. Yao, C. Hu, X. Yan, and Y. Fan, "Mr-cof: a genetic mapreduce configuration optimization framework," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2015, pp. 344–357.

[8] K. Wang and M. M. H. Khan, "Performance prediction for apache spark platform," in *High Performance Computing and Communications (HPCC), 2015 IEEE 17th International Conference on*. IEEE, 2015, pp. 166–173.

[9] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 307–319.

[10] Kafka, "https://kafka.apache.org/," Accessed on July 31th 2018.

[11] Hive, "https://hive.apache.org/," Accessed on July 31th 2018.

[12] Redis, "https://redis.io/," Accessed on July 31th 2018.

[13] MySQL, "https://www.mysql.com/," Accessed on July 31th 2018.

[14] Cassandra, "http://cassandra.apache.org/," Accessed on July 31th 2018.

[15] Tomcat, "http://tomcat.apache.org/," Accessed on July 31th 2018.

[16] T. Ye and S. Kalyanaraman, "A recursive random search algorithm for large-scale network parameter configuration," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, pp. 196–205, 2003.

[17] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 171–187.

[18] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: A survey," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 295–310, 2004.

[19] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Performance Evaluation*, vol. 67, no. 8, pp. 634–658, 2010.

[20] D. Paluch, H. Kienegger, and H. Krcmar, "A workload-dependent performance analysis of an in-memory database in a multi-tenant configuration," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 2018, pp. 131–134.

[21] B. Chen, Y. Liu, and W. Le, "Generating performance distributions via probabilistic symbolic execution," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 49–60.

[22] A. Abdelaziz, W. Kadir, and A. Osman, "Comparative analysis of software performance prediction approaches in context of component-based system," *International Journal of Computer Applications*, vol. 23, no. 3, pp. 15–22, 2011.

[23] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012.

[24] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for better configurations: a rigorous approach to clone evaluation," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 455–465.

[25] 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*. ACM, 2017, pp. 257–267.

[26] J. Oh, D. Batory, M. Myers, and N. Siegmund, "Finding near-optimal configurations in product lines by random sampling," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 61–71.

[27] R. Olaechea, D. Rayside, J. Guo, and K. Czarnecki, "Comparison of exact and approximate multi-objective optimization for software product lines," in *Proceedings of the 18th International Software Product Line Conference-Volume 1*. ACM, 2014, pp. 92–101.

[28] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke, "Deep parameter optimisation," in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, 2015, pp. 1375–1382.

[29] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang, "A smart hill-climbing algorithm for application server configuration," in *Proceedings of the 13th international conference on World Wide Web*. ACM, 2004, pp. 287–296.

[30] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, 2012, pp. 2951–2959.

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

[32] N. B. Rizvandi, J. Taheri, R. Moraveji, and A. Y. Zomaya, "On modelling and prediction of total cpu usage for applications in mapreduce environments," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2012, pp. 414–427.

[33] M. Niu, S. Rogers, M. Filippone, and D. Husmeier, "Fast inference in nonlinear dynamical systems using gradient matching," in *Journal of Machine Learning Research: Workshop and Conference Proceedings*, vol. 48. Journal of Machine Learning Research, 2016, pp. 1699–1707.

[34] S. Wang, C. Li, H. Hoffmann, S. Lu, W. Sentosa, and A. I. Kistijantoro, "Understanding and auto-adjusting performance-sensitive configurations," in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 2018, pp. 154–168.

[35] P. Lama and X. Zhou, "Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud," in *Proceedings of the 9th international conference on Autonomic computing*. ACM, 2012, pp. 63–72.

[36] N. Luo, Z. Yu, Z. Bei, C. Xu, C. Jiang, and L. Lin, "Performance modeling for spark using svm," in *Cloud Computing and Big Data (CCBD), 2016 7th International Conference on*. IEEE, 2016, pp. 127–131.

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

[38] C. Tang, "System performance optimization via design and configuration space exploration," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 1046–1049.

[39] S. Soltani, M. Asadi, M. Hatala, D. Gašević, and E. Bagheri, "Automated planning for feature model configuration based on stakeholders' business concerns," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. IEEE, 2011, pp. 536–539.

[40] V. Nair, T. Menzies, N. Siegmund, and S. Apel, "Faster discovery of faster system configurations with spectral learning," *Automated Software Engineering*, pp. 1–31, 2017.

[41] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction models," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 321–332.

[42] S. Falkner, A. Klein, and F. Hutter, "BOHB: Robust and efficient hyperparameter optimization at scale," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. Stockholmsmssan, Stockholm Sweden: PMLR, 10–15 Jul 2018, pp. 1436–1445. [Online]. Available: http://proceedings.mlr.press/v80/falkner18a.html

[43] F. Hutter, H. Hoos, and K. L-Brown, "Sequential model based optimization for general algorithm configuration." *LION*, vol. 5, pp. 507–523, 2011.

[44] J. Bergstra, D. Yamins, and D. D. Cox, “Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms,” in *Proceedings of the 12th Python in Science Conference*. Citeseer, 2013, pp. 13–20.

[45] C. Peng, C. Zhang, C. Peng, and J. Man, “A reinforcement learning approach to map reduce auto-configuration under networked environment,” *International Journal of Security and Networks*, vol. 12, no. 3, pp. 135–140, 2017.

[46] L. Bao, X. Liu, Z. Xu, and B. Fang, “Autoconfig: Automatic configuration tuning for distributed message systems.” in *Automated Software Engineering (ASE), 2018 33rd IEEE/ACM International Conference on*. IEEE, 2018, pp. 29–40.

[47] H. Chen, W. Zhang, and G. Jiang, “Experience transfer for the configuration tuning in large-scale computing systems,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 3, pp. 388–401, 2011.

[48] P. Jamshidi, N. Siegmund, M. Velez, and C. Kästner, “Transfer learning for performance modeling of configurable systems: An exploratory analysis,” in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 497–508.

[49] C. Blum and A. Roli, “Metaheuristics in combinatorial optimization: Overview and conceptual comparison,” *ACM computing surveys (CSUR)*, vol. 35, no. 3, pp. 268–308, 2003.

[50] C. H. Papadimitriou and K. Steiglitz, “Combinatorial optimization: algorithms and complexity,” 1982.

[51] M. R. Garey and D. S. Johnson, “Computers and intractability: a guide to np-completeness,” 1979.

[52] P. I. Frazier, “A tutorial on bayesian optimization,” *arXiv preprint arXiv:1807.02811*, 2018.

[53] X. Li and X. Yao, “Cooperatively coevolving particle swarms for large scale optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 16, no. 2, pp. 210–224, 2012.

[54] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[55] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, “The hibench benchmark suite: Characterization of the mapreduce-based data analysis,” in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*. IEEE, 2010, pp. 41–51.

[56] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with ycsb,” in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

[57] JMeter, “https://jmeter.apache.org/,” Accessed on July 31th 2018.