EasyChair Preprint

№ 2746

Exploratory Performance Testing Using Reinforcement Learning

Tanwir Ahmad, Adnan Ashraf, Dragos Truscan and Ivan Porres

February 21, 2020

Exploratory Performance Testing Using Reinforcement Learning

Tanwir Ahmad, Adnan Ashraf, Dragos Truscan, and Ivan Porres
Faculty of Science and Engineering
Åbo Akademi University, Finland
{tanwir.ahmad, adnan.ashraf, dragos.truscan, ivan.porres}@abo.fi

***Abstract*—Performance bottlenecks resulting in high response times and low throughput of software systems can ruin the reputation of the companies that rely on them. Almost two-thirds of performance bottlenecks are triggered on specific input values. However, finding the input values for performance test cases that can identify performance bottlenecks in a large-scale complex system within a reasonable amount of time is a cumbersome, cost-intensive, and time-consuming task. The reason is that there can be numerous combinations of test input values to explore in a limited amount of time. This paper presents PerfXRL, a novel approach for finding those combinations of input values that can reveal performance bottlenecks in the system under test. Our approach uses reinforcement learning to explore a large input space comprising combinations of input values and to learn to focus on those areas of the input space which trigger performance bottlenecks. The experimental results show that PerfxRL can detect 72% more performance bottlenecks than random testing by only exploring the 25% of the input space.**

***Keywords*-Performance testing, reinforcement learning, deep neural network, test data generation**

I. Introduction

Due to the widespread availability of high-speed Internet, the level of expectation of the users with respect to the performance of web-based software systems has changed dramatically [1]. For instance, about 40% of the customers will abandon a web application if the response time is greater than 3 seconds [2]. Furthermore, Amazon, a leading online retailer, reports that 100 milliseconds (ms) additional delay in response time could cost them 1% drop in sales [3]. Therefore, ensuring the reliability and efficiency of a software system is imperative for such companies and the overall success of the software projects.

During the software development and maintenance phase, identifying and fixing the performance bottlenecks of a system under test (SUT) is one of the most critical and challenging tasks for developers [4]. There are high chances of the system crashing due to performance bottlenecks than due to system failures [5]. *Performance bottlenecks* are software defects, which degrade the performance of the SUT unexpectedly [4]. The primary purpose of *performance testing* is to find performance defects [5]. For example, in stress testing, which is a type of performance testing, test engineers create test cases in order to identify the scenarios which will potentially degrade the performance of the SUT or cause a failure [6]. These test cases contain a sequence of actions (e.g., sending an HTTP request to the SUT) and test inputs for those actions (e.g., input parameters for HTTP requests).

In our previous work [7], we proposed an approach to find the worst sequence of actions that maximizes the resource utilization on the SUT using a genetic algorithm. We investigated how the sequence and frequency of user interactions impact the performance of the SUT. In this paper, we aim at finding test input values which worsen the performance of the SUT using reinforcement learning.

In general, two-thirds of the performance bottlenecks are triggered only on specific input values [4]. However, finding those input values for performance test cases that can identify performance bottlenecks in a large-scale complex system within a reasonable amount of time is a notoriously difficult and expensive task [4]. The reason is that there can be numerous combinations of test input values and executing one such combination against the SUT can take a considerable amount of time. Thus, it is almost impossible for test engineers to exhaustively test every possible combination. The problem becomes even more challenging when the SUT is a *black-box*. This means that we cannot inspect the internal system dynamics. The only way to interact with the SUT and monitor different key performance indicators (KPIs) of the SUT is through public interfaces and external observations during the execution of the SUT, respectively.

In most cases, test input values for performance test cases are either (1) created manually by the test engineers based on their experiences and intuitions, (2) calculated by performance profilers, (3) collected during the normal usage of the SUT, or (4) extracted from crash reports sent by the customers [8]. However, there are drawbacks to all of these approaches:

1) Test engineers need to have rigorous domain knowledge about the SUT in order to create test input values manually.
2) Majority of the performance profilers require access to the source code of the SUT which is often infeasible. In addition, they require test inputs to instrument the program under test for performance profiling.
3) Test input values collected during the normal usage represent the most common combinations of the input values executed by the users. However, they miss the rare and infrequent yet problematic combinations.
4) Acquiring the test input values after the system has already crashed in the production environment is not a good strategy because it damages the reputation of a company.

In order to address the above issues, we formulate the test data generation problem for performance test cases as a reinforcement learning problem. *Reinforcement learning* (RL) is a class of machine learning techniques where an *agent* learns to interact with an unknown environment in order to accomplish a given goal through a trial and error process [9]. In recent years, RL techniques have been used to solve a wide variety of problems [10]. In our case, the unknown environment is the SUT and the agent is a tester which interacts with the SUT by executing performance test cases (e.g., sequence of test inputs) with different combinations of the input parameter values. The purpose of the agent is to find the combinations of input parameter values which degrade the performance of the SUT. The contributions of this paper are as follows:

1) We present an approach for **Perf**ormance E**X**ploration using **RL** (*PerfXRL*) to explore a large space of combinations of the input parameter values in order to find performance bottlenecks in a black-box system without any prior domain knowledge. Our approach uses Dueling Deep Q-Network (DDQN) [11], which is an RL technique. To the best of our knowledge, we are not aware of any existing performance testing approaches that employ reinforcement learning for finding performance bottlenecks.
2) We develop tool support for our approach in Python using Keras-rl [12] library to automate the exploration process.
3) We empirically evaluate our approach and show that PerfXRL is able to detect 72% more performance bottlenecks than random testing.

The rest of the paper is structured as follows: Section II provides an introduction to reinforcement learning. In Section III, we describe our approach. We empirically evaluate our approach in Section IV. Section V presents an overview of the related work. Section VI specifies some threats to the validity of this work. Finally, Section VII discusses conclusions and future work.

II. Reinforcement Learning

Reinforcement learning (RL) is a reward-driven machine learning technique in which an agent learns by interacting with an unknown environment in order to accomplish a goal. The agent collects feedback (or a reward) from the environment by performing an action according to the current state of the environment. The goal of the agent is to maximize the expected cumulative rewards over time by finding the optimal (or a near-optimal) sequence of actions.

An RL process can be characterized as a Markov decision process (MDP). The process is represented as a tuple $\langle S, A, P, R, \gamma \rangle$, where

- S is a finite set of states of the environment;
- A represents a finite set of permissible actions;
- $P : S \times A \to S$ is a transition function;
- $R : S \times A \to \mathbb{R}$ is a reward function;
- $\gamma \in [0, 1]$ is a discount factor.

At every time step t, the agent observes the current *state* $s_t \in S$ and then it performs an *action* $a_t \in A$. After each action, the agent receives a scalar *reward* $r_{t+1} \sim R(s_t, a_t)$[1] from the environment, as well as the next state $s_{t+1} \sim P(s_t, a_t)$.

The agent chooses an action according to a *policy* π which maps the states of the environment to the actions which can be performed in those states. The objective of RL is to find an *optimal policy* π^* which maximizes the total excepted discounted return. The *total expected discounted return* G at a time step t is specified as $G_t \doteq \Sigma_{k=t+1}^{T} \gamma^{k-t-1} R(s_k, a_k)$, where T is the maximum number of time steps in a finite MDP. This duration is known as an *episode*.

In order to find an optimal policy, we use an action-value function. The *action-value function* $Q^\pi(s, a) \doteq \mathbb{E}_\pi[G_t | s_t = s, a_t = a]$ defines the best possible reward obtained by performing an action a_t at a state s_t while following policy π. Using the action-value function Q, the agent can select the optimal or a near-optimal action at every state in order to maximize the G_t. Hence, we can define the optimal policy π^* in terms of action-value function: $\pi^* = argmax_\pi Q^\pi(s, a) \forall s \in S, \forall a \in A$. This implies that we can compute the optimal policy π^* by calculating the optimal action-value function. However, in practice, computing the exact optimal action-value function within a reasonable amount of time is often infeasible. Instead, we use an iterative algorithm, called *Q-Learning* [13], to approximate the action-value function. The Q-Learning algorithm estimates the optimal action-value function with the following update rule:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(r_{t+1} + \gamma max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)) \quad (1)$$

where $\alpha \in (0, 1]$ is the learning rate.

In a traditional Q-Learning algorithm, a lookup table is used to store and update Q values for each pair of states and actions. However, this method is not practical for a large number of states and actions. In order to alleviate this problem, *Deep Q Networks* (DQN) [14] algorithm is used where a *neural network* [15] is employed as a function approximator to estimate the action-value function such as $Q(s, a; \theta) \approx Q(s, a)$, where θ represents a set of adjustable parameters of the function approximator. Another benefit of using DQN is that the agent can generalize the knowledge from the observed states to the unseen states.

Wang et al. [11] proposed a new approach called Dueling DQN (DDQN) where they used a new dueling neural network architecture for Q values approximation. In this architecture, they separately estimate two functions: state value function and state-dependent action advantage function. The estimated values from these functions are used to calculate the Q values. The dueling architecture allows the agent to learn the significance of states without learning the value of each action for each state. In this paper, we use DDQN algorithm to train our agent.

[1]We follow the conventions from the book of Sutton et al. [9] where they use r_{t+1} to denote the reward in response to action a_t in state s_t.

III. PERFXRL

The proposed PerfXRL approach uses RL to search for performance bottlenecks in a black-box SUT without any prior domain knowledge about the SUT (see Figure 1). It explores the performance of the SUT by executing different combinations of the input values against it. The objective of PerfXRL is to concentrate the search to those areas of the input space which can reveal performance bottlenecks and to find the maximum number of combinations of the input values which can trigger the performance bottlenecks in the SUT. We refer to those combinations as *good combinations*.

In the following section, we describe an example, formulate the input data generation problem as an RL problem, and present our approach and its components in detail.

A. Example

Consider a hypothetical **C**losed-source **B**lack-box **S**UT (*CBS*), which accepts three input parameters. Each input parameter ranges from 1 to 100, which in total represent an input space of 10^6 combinations. The main challenge is to find a subset of combinations of the input values which trigger performance bottlenecks (e.g., resource-intensive computations on the SUT), without executing every combination. In the following sections, we use this example to explain different components of our approach and show how our approach can be applied to find performance bottlenecks.

B. RL formulation

The test data generation problem for performance testing can be represented as a search problem where we have a large input space S containing all possible combinations of input values. The aim is to find the complete or near-complete set of combinations of input values $\mathcal{B} \subset S$ that trigger resource intensive computations that are reflected (observed) as a degradation of the performance, for instance, lower throughput and higher response time. We refer to these observations as *performance bottlenecks*, while the set of inputs that trigger them are the values of interest which we denote by the *set of good combinations* $\mathcal{B}$ with respect to our approach. In this paper, we aim to solve this problem using RL, where an agent effectively explores different combinations of the test input values and tries to find the performance bottlenecks by following the feedback (or *reward*) from the environment (SUT). This is a continuous process where the agent creates a new combination of test input values by suggesting an action to the *environment*. The environment executes the newly created combination against the SUT and returns the reward to the agent. The reward is used by the agent to improve the selection of its future actions. Furthermore, we maintain a list of all the good combinations of input values which have been executed so far by the agent. The list can later be used by developers for performance debugging.

In the following, we define the fundamental components of our RL process including state representation S, reward function R, action space A, transition function P and agent's policy π.

a) State space: A state $s_t \in S$ is a vector of input values $\{v_t^1, v_t^2, \ldots, v_t^N\}$ at time step t where N is a number of inputs to the SUT. In other words, a state s represents a single combination of input values.

For example, S_{CBS} denotes the input space of the CBS. We have identified three input parameters for the CBS where each parameter ranges from 1 to 100. These input parameters define the input space S_{CBS} for our RL agent where each input parameter represents a dimension in the input space S_{CBS}. The size of the input space S_{CBS} is 10^6 (i.e., the total number of combinations of the input values).

b) Action space: The agent observes the current state s_t (i.e., the current combination of input values) and provides an action a_t to the environment. Based on the action a_t, the environment modifies the s_t in order to produce the next state s_{t+1} (i.e., a new combination of input values). Thus, we can specify the action space A as a set of potential modifications that can be made to the current state in order to produce the next state. At every time step t, based on the selected action the agent either increases or decreases a single input value in the current state in order to get the next state:

$$A = \{a_i | i \in [1, 2, \ldots, N * 2]\}$$

$$a_i = \begin{cases} v_{t+1}^{(i+1)\div 2} \leftarrow v_t^{(i+1)\div 2} + 1 & \text{if } i \text{ is odd} \\ v_{t+1}^{i\div 2} \leftarrow v_t^{i\div 2} - 1 & \text{if } i \text{ is even} \end{cases}$$

We have two possible operations (i.e., increment and decrement) for every input parameter; thus, the size of the action space A for our approach is $N * 2$. For example, the size of the action space A_{CBS} of the CBS is six because we have three input parameters. Table I lists all the actions and how they modify the current state in order to get the next state. In this paper, we have discussed only integer input parameters; however, our approach can easily be used with other types of inputs (e.g., string, float) by modifying the action and input space accordingly.

TABLE I
ACTION SPACE OF THE CBS

Action	**State modification**	**Example**: if $s_t = \{84, 65, 86\}$ then s_{t+1}
a_1	$v_{t+1}^1 \leftarrow v_t^1 + 1$	{**85**, 65, 86}
a_2	$v_{t+1}^1 \leftarrow v_t^1 - 1$	{**83**, 65, 86}
a_3	$v_{t+1}^2 \leftarrow v_t^2 + 1$	{84, **66**, 86}
a_4	$v_{t+1}^2 \leftarrow v_t^2 - 1$	{84, **64**, 86}
a_5	$v_{t+1}^3 \leftarrow v_t^3 + 1$	{84, 65, **87**}
a_6	$v_{t+1}^3 \leftarrow v_t^3 - 1$	{84, 65, **85**}

c) Transition function: A transition function P accepts a state and an action as inputs and returns a new state. In Markov chains, the function is stochastic for a non-deterministic environment. However, in our case, the environment is *deterministic*. It implies that given the current state and the selected action, we can calculate the exact outcome of the transition function at any time step.

d) Policy: Our agent uses *ϵ-greedy* policy to select actions. The ϵ-greedy policy is designed to select either a

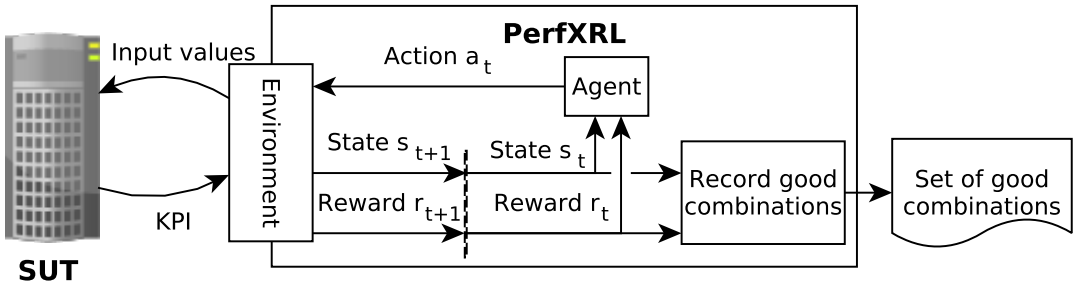

Fig. 1. PerfXRL (Adapted from [9])

random action among from available actions with probability $\epsilon \in [0, 1]$ or the best action based on the Q values, with probability $1-\epsilon$. Introducing bounded randomness in selecting an action inspires the agent to explore the input space as well as guarantees that the agent does not get stuck in a poor strategy. ϵ is an algorithm parameter. In our case, we use an adaptive value of ϵ which decreases linearly during training, causing the agent to favor exploration at first and exploitation at the later stages of training.

e) Reward: The reward value is the primary basis for updating the policy; if the chosen action by the policy is followed by a low reward, then the policy may be updated to choose some other action on that state in the future. The goal of the agent is to maximize the cumulative discounted reward G. Hence, the reward function R should be defined to guide the agent towards good solutions for the given objective.

In our case, the objective is to find the combinations of input values which trigger the performance bottlenecks. We use the elapsed execution time of the SUT for a given combination of the input values as a performance bottleneck indicator. The rationale here is that the elapsed execution time of the SUT for those combinations of input values would be higher than a given *acceptable performance threshold* $\mathcal{L}$ which will cause resource-intensive computations on SUT, and those combinations are most likely to cause performance bottlenecks. We can define our reward $r^{\mathcal{L}}_{t+1}$:

$$r^{\mathcal{L}}_{t+1} = \begin{cases} x \in \mathbb{Z}_{>0} & \text{if } \mathrm{E}(\mathrm{s_{t+1}}) > \mathcal{L} \\ x \in \mathbb{Z}_{<0} & \text{otherwise.} \end{cases} \tag{2}$$

where E is an executor function which runs the performance test cases using the provided combination of the input values s_{t+1} against the SUT and returns the elapsed execution time of the test cases in seconds. In summary, Equation (2) specifies that if the elapsed execution time of the combination s_{t+1} is more than the given acceptable performance threshold $\mathcal{L}$, the combination s_{t+1} has triggered a performance bottleneck on the SUT and, therefore, the agent receives a positive reward (i.e., a positive integer).

We use the elapsed execution time as the KPI because it is widely used for performance evaluation. However, our approach can utilize other KPIs, such as CPU load, memory or disk usage for reward calculation by updating the reward function.

The executor function E_{CBS} runs the CBS using the given input parameter values and returns the elapsed execution time of the SUT. Let us assume $s_0 = \{5, 6, 9\}$ and $\mathcal{L} = 2$, where s_0 is the current state and $\mathcal{L}$ is the performance threshold. The agent observes the current state s_0 and selects the action a_3 among all the actions listed in Table I. Then, based on the selected action, the environment computes the next state $s_1 = \{5, 7, 9\}$ and the executor function for s_1: $E_{CBS}(s_1) = 3$. Since $E_{CBS}(s_1) > \mathcal{L}$, the agent gets the positive reward for performing the action a_3 on the state s_0: $r^{\mathcal{L}}_1 = 10$. A positive reward signals the agent that it took a good action on the given state. Therefore, the agent would increase the probability of selecting that action in the future for that state.

C. Finding bottlenecks

We train our agent in an episodic framework. We simulate a specific number of episodes. In every episode, the agent starts from a random state and performs a fixed number of training steps. In each step, the agent suggests an action to the environment which computes a new combination of input values based on the proposed action and the current combination of input values. Next, the environment executes the newly created combination of input values against the SUT and calculates the reward. Once the agent has performed the fixed amount of steps, the episode is ended. Thereupon, the environment begins the new episode by generating a random combination of input values. Starting every episode from a random combination of input values allows the agent to explore different regions of the input space in each episode. Both parameters, number of episodes (*Ep*) and number of steps in an episode (*EpSteps*), mainly depend on the complexity of the SUT and the duration of test runs against the SUT and vary from case to case.

We use DDQN algorithm to train our agent. In DDQN, the agent uses a *deep neural network* (DNN) [15] to implement a policy called *policy network*. We provide the last five states as input to the policy network instead of the only current state. Preliminary experiments showed that this modification increases the learning capability of our policy network significantly allowing it to learn about dependencies among input values. The policy network returns Q values for possible actions. Then, the agent suggests either the action with the highest Q value or a random action for exploration with

probability $1-\epsilon$ or ϵ, respectively, to the environment in order to produce the next state.

Furthermore, the agent stores the experience (i.e., current state s_t, action a_t, next state s_{t+1}, and reward r_{t+1}), at each time step t, in a cyclic buffer, called *replay memory*. These stored experiences are uniformly sampled from the replay memory to train the neural network. This method is known as *experience replay* [16]. There are two main benefits of using this method: it reduces the variance of learning updates (see Equation (1)), because the consecutive updates are not temporally correlated, and allows reusing stored experiences for multiple learning updates to increase the learning efficiency of the agent.

IV. Evaluation

In this section, we experimentally evaluate PerfXRL by comparing it against random testing. In random testing, we uniformly sample an input value combination without replacement from the input space of the SUT for a given number of times. We choose *random testing* because it is a robust approach [17], [18] as compared to many other systematic testing approaches. Hamlet [19] recommended using random testing for a large and irregular input space.

In the rest of this section, we first describe the subject application used for evaluation, then we discuss the results.

A. Subject application

We use the reference web application RUBiS [20] as a subject application in our experiments. RUBiS is a web-based application that implements the core functionality of an auction site. It has been widely used in academia for performance evaluation, with over 300 citations on Google Scholar[2]. We use an Apache[3] 2.4.29 web server with PHP[4] 7.2.10 to host the front-end of the application. The backend database is MySQL[5] 14.14.

The inputs to RUBiS can only be provided via HTTP requests. We define a performance test case for RUBiS as a sequence of three URLs listed in the last column of Table II, that are used in HTTP requests to RUBiS. These URLs require four input parameters values. Thus, a state $s_t \in S_{RUBiS}$ can be represented as a combination of input values $s_t = \{CID_t, RID_t, IID_t, UID_t\}$ at time step t. Even though the input parameters are in different URLs and are accessed sequentially, we consider them as an atomic definition of the state. The size of the input space S_{RUBiS} is 3 100 000 (i.e., the total number of combinations of the input values). The size of the action space A_{RUBiS} for RUBiS is eight because we have four input parameters.

In order to calculate the reward r_{t+1} for the agent, we need to define the executor function E_{RUBiS} for RUBiS. The executor function $E_{RUBiS}(s_{t+1})$ runs the RUBiS performance test case using the input values s_{t+1} and returns the total elapsed execution time of the URL requests.

TABLE II
RUBiS input parameters[6]

Input parameter	Range	URL
Category ID (CID)	[1, 20]	/SearchItemsByRegion.php?category=
Region ID (RID)	[1, 62]	CID&categoryName=CN&region=RID
Item ID (IID)	[1, 50]	/ViewItem.php?itemId=IID
User ID (UID)	[1, 50]	/ViewUserInfo.php?userId=UID

In order to measure the efficiency of PerfXRL, we need to know all the existing performance bottlenecks in RUBiS which we can compare against the number of bottlenecks identified by PerfXRL. For that purpose, we performed exhaustive performance testing of RUBiS by measuring its performance (i.e., the elapsed execution time of the RUBiS performance test case) against all possible combinations of the input values using the MBPeT [21] tool. Durin this experiemnt, we observed that the input values have no considerable impact on the performance of RUBiS. Thus, we have uniformly injected 20 clusters of artificial performance bottlenecks. Figure 2 shows the distribution of bottleneck clusters where each diamond shape represents one cluster, and the values on the diamonds indicate the identifier of the clusters. A cluster of bottlenecks would invoke computationally expensive operations and increase the elapsed execution time of the RUBiS performance test case by approximately 5 seconds if the given input values are within certain ranges. For instance, a bottleneck cluster with an identifier 19 would be triggered if the given combination of input values $\{CID, RID, IID, UID\}$ satisfies the following condition:

$$2 \leq CID \leq 4 \wedge 5 \leq RID \leq 21 \wedge 6 \leq IID \leq 22 \\ \wedge\, 13 \leq UID \leq 29$$

The reason for injecting clusters of bottlenecks instead of individual bottlenecks is that the performance problems usually tend to affect a group of combinations contrary to a single combination. Since each injected performance bottleneck adds a delay of 5 seconds to the elapsed execution time of the performance test, we set the value $\mathcal{L}$ to 5 and use the following equation for reward calculation:

$$r^{\mathcal{L}}_{t+1} = \begin{cases} 10 & \text{if E}(s_{t+1}) > 5 \\ -1 & \text{otherwise.} \end{cases}$$

In our preliminary tests, we got good results by setting the positive and negative reward to 10 and -1, respectively.

We repeated each experiment 30 times to establish the statistical significance of the results. For every experiment, the approach under evaluation and the SUT ran on different machines. Each machine featured an Intel Core i7-3770K CPU, 16 GB of memory, 7200 rpm hard drive, and Ubuntu 18.04 Operating System. To reduce the network latency, the machines were connected via a 1Gb Ethernet connection in an isolated environment.

[2]http://scholar.google.com/scholar?q=Specification+and+Implementation+of+Dynamic+Web+Site+Benchmarks

[3]https://httpd.apache.org/

[4]https://secure.php.net/

[5]https://www.mysql.com/

[6]The input parameter *category name* (CN) is not listed as an input parameter in the table because its value depends on the value of input parameter CID.

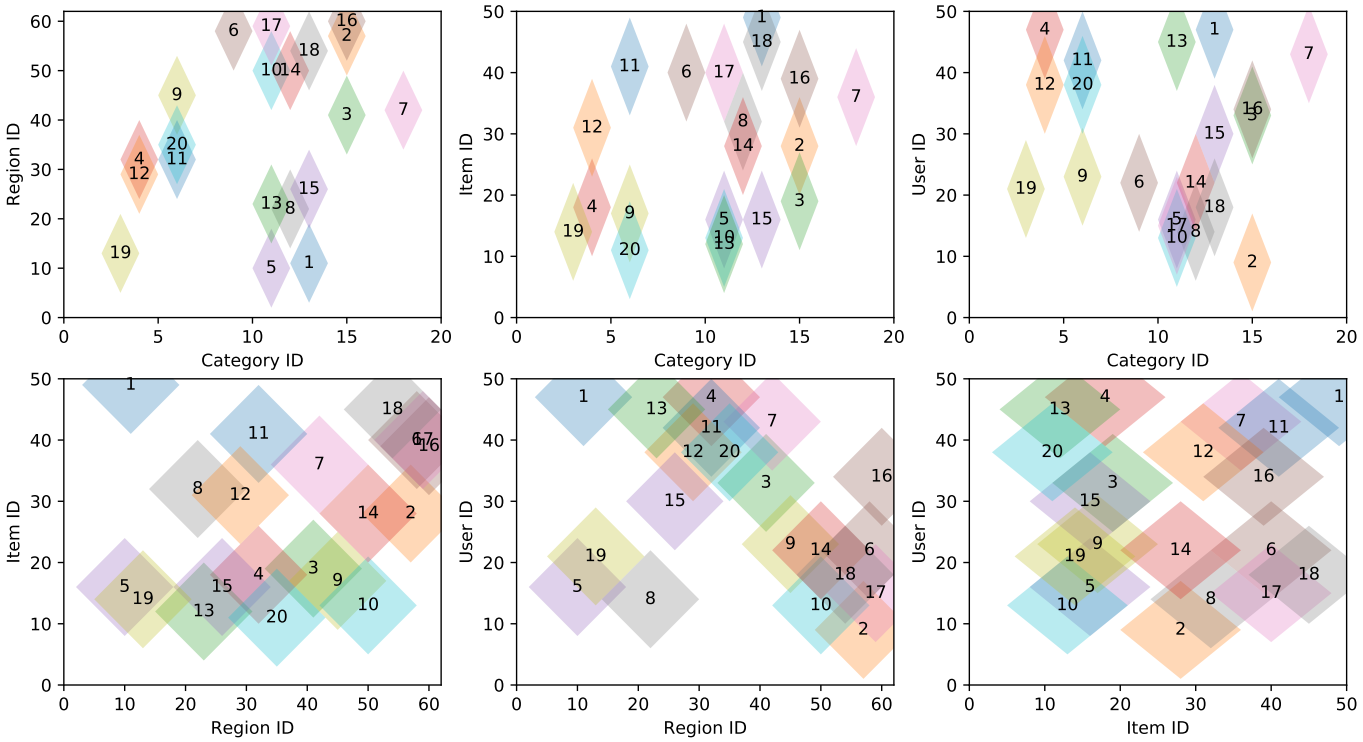

Fig. 2. Uniformly distributed clusters of bottlenecks in RUBiS

B. Results

The purpose of our evaluation is to measure the effectiveness of PerfXRL by comparing it against against random testing. We run PerfXRL and random testing on RUBiS. We used the following parameters for PerfXRL:

- *Number of training steps per episode:* 100
- *Number of episodes:* 7750, in total PerfXRL performs 775 000 training steps. This means that PerfXRL can execute 775 000 combinations of input values against the SUT which is 25% of the total input space.
- *Epsilon* ϵ: the value linearly decays over the total number of training steps from 1 to 0.1
- *Discount factor* γ: 0.99
- *Size of the replay memory*: 500 000
- *Policy network*: we use three fully-connected layers with 32 nodes. The last layer of the network uses the *linear* [22] activation function, but the rest of the layers use the *Rectified Linear Unit* (ReLU) [23] activation function. We use the *Adam* [24] optimization algorithm for our network with the *learning rate* (lr) is set to 0.001.

Figure 3 shows the cumulative number of injected performance bottleneck identified by PerfXRL and random testing after executing a certain number of input value combinations. The solid lines in the figure show the average values, while the shaded region around the lines represents the standard deviation. Both approaches performed similarly for the first 150 000 input value combinations, but after that point, we can observe that PerfXRL was able to identify performance bottlenecks at a much higher rate than random testing. At the end of the experiment, the average number of performance bottlenecks identified by PerfXRL and random testing are around 100 800 and 58 405, respectively. Both PerfXRL and random testing executed the same amount of input value combinations, but PerfXRL found 72% more bottlenecks than random testing. We would like to point out that the standard deviation of the results using random testing was very low; thus, it is not visible in the figure. In summary, the overall results show that PerfXRL is better and faster in finding performance bottlenecks compared to random testing.

V. Related Work

In this section, we discuss the most important related works on performance testing using machine learning and other approaches.

Luo et al. [25] present a technique called FOREPOST, which uses a machine learning algorithm to extract rules that map performance behaviors of the application under test to input combinations. The main difference between FOREPOST and our proposed approach is that FOREPOST uses a rule learning algorithm to extract some rules to guide the selection of test inputs, whereas our proposed approach uses reinforcement learning to rigorously explore the input space and learns to concentrate the search on the most important subsets of the input space. The rule learning approach of FOREPOST does not provide a good coverage of the input space and as a result misses some bottlenecks.

Shen et. al [26] propose an approach called GA-Prof, which performs search-based profiling of the application under test

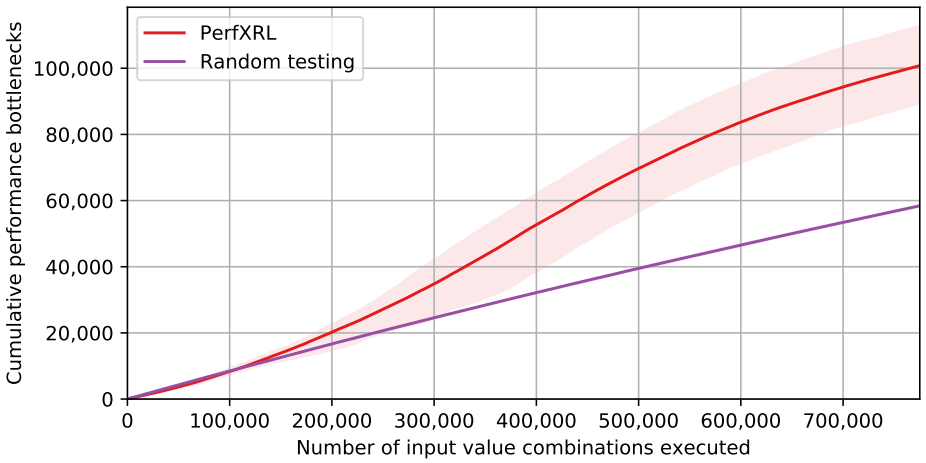

Fig. 3. Cumulative number of performance bottlenecks found during exploration

(AUT). The search is guided by a genetic algorithm. The source code of the AUT is analyzed to identify mappings of the test inputs to the methods in the source code and then to identify the methods involved in performance bottlenecks. There are two main differences between GA-Prof and our proposed approach: (1) our proposed approach uses machine learning instead of a genetic algorithm and most importantly (2) our proposed approach does not rely on the source code of the AUT.

Lemieux et al. [8] presented a performance test generation approach called PerfFuzz. It uses feedback-directed mutational fuzzing to find inputs that reveal worst-case algorithmic complexity in different components of the program under test. The process starts with a set of randomly generated inputs. New inputs are generated iteratively by saving and mutating the previous inputs that increase code coverage. When compared to PerfFuzz, our approach does not rely on the source code of the AUT and uses reinforcement learning to explore the input space of the AUT.

Chen et al. [27] presented a performance analysis framework called PerfPlotter and a symbolic execution approach for generating performance distributions for a program under test. The performance distributions are generated from the source code and usage profiles of the program. The approach uses probabilistic symbolic execution to selectively explore different high-probability and low-probability execution paths in the program and generates performance distributions. The generated distributions can be used to understand and analyze the general trend of program execution times along with the best-case and worst-case execution times. Although performance distributions of a program under test can be useful in performance testing, PerfPlotter does not provide a performance testing approach.

In summary, although a large amount of research work has been done in investigating the methods for finding the performance bottlenecks, we could not find any approach similar to the one presented in this paper, that is identifying a near-complete set of performance bottlenecks in a black-box system by exploring only a subset of the input space of the SUT.

VI. Threats to validity

The first threat to the internal validity of the experiment is that we randomly injected artificial bottlenecks into the subject application. Thus, there is a threat that we may achieve different results by running our approach against a system with real bottlenecks. However, by following this experimental design, we managed to evaluate PerfXRL in a controlled setting effectively. Thus, we believe that we sufficiently minimized this threat, and our results are reliable.

PerfXRL, like many other machine learning approaches, is susceptible to their parameters, for instance, a suitable set of parameter values for one problem environment might not work well for others. For DDQN, we selected the parameter values according to the practical experiences reported in previous researches using deep Q-learning algorithm [11], [14]. Since our approach does not require access to the source code of the SUT nor domain knowledge, in a real-world setting, one only needs to adjust few parameters (e.g., performance threshold $\mathcal{L}$) for a new environment or SUT.

The main threat to external validity is that we used only one SUT in our evaluation. Therefore, the results of our experiment may differ for systems that have different architectures or different input space. However, to the best of our knowledge, there are no publicly available performance benchmark applications which closely represents the real-world applications. This threat needs to be addressed by performing additional experiments using different applications. Another threat is the selection of URLs listed in Table II. The only reason we have chosen those URLs is that we are focusing on integer input parameters in this work. Nevertheless, our approach is agnostic to input parameter types and can be applied to different types of inputs (e.g., string, float) by updating the action and input space accordingly.

A final threat to external validity is that we only compared PerfXRL against random testing in order to evaluate the

efficiency of PerfXRL, which might be seen as an unfair comparison. As we have discussed in Section V, we could not find any approach similar to PerfXRL that is to find performance bottleneck in a black-box system by exploring only a subset of the input space of the SUT. Moreover, it has been reported that random testing performs better than many other systematic approaches, especially when we are testing a black-box SUT [17]–[19].

VII. Conclusion

We presented PerfXRL, a novel approach to find performance bottlenecks in a black-box system without any prior domain knowledge using Reinforcement Learning. In our approach, a learning agent explores a large space of the input value combinations and focuses only on those areas of the input space which are most likely to trigger performance bottlenecks in the system under test. The results presented in this paper show that, after executing only 25% of the total combinations of input values, PerfXRL managed to find approximately 72% more performance bottlenecks than a random testing technique. For our future work, we aim to integrate different key performance indicators (KPIs) such as CPU and memory usage into our reward function. We are also planning to try different types of neural network architectures and to investigate how do they affect the efficiency and effectiveness of PerfXRL.

Acknowledgment

This work has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement number 737494. This Joint Undertaking receives support from the European Unions Horizon 2020 research and innovation programme and Sweden, France, Spain, Italy, Finland, the Czech Republic.

References

[1] C. Rheem, "Consumer response to travel site performance," *A PhoCusWright and Akamai WHITEPAPER*, 2010. [Online]. Available: https://www.phocuswright.com/Free-Travel-Research/Consumer-Response-to-Travel-Site-Performance

[2] N. Patel, "Great expectations: 47% of consumers want a web page to load in two seconds or less," http://insights.wired.com/profiles/blogs/47-of-consumers-expect-a-web-page-to-load-in-2-seconds-or-less, 2009, retrieved: September, 2015.

[3] M. Mazzucco, "Towards autonomic service provisioning systems," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, May 2010, pp. 273–282.

[4] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 77–88. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254075

[5] E. Weyuker and F. Vokolos, "Experience with performance testing of software systems: issues, an approach, and case study," *Software Engineering, IEEE Transactions on*, vol. 26, no. 12, pp. 1147–1156, 2000.

[6] L. C. Briand, Y. Labiche, and M. Shousha, "Stress testing real-time systems with genetic algorithms," in *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '05. New York, NY, USA: ACM, 2005, pp. 1021–1028. [Online]. Available: http://doi.acm.org/10.1145/1068009.1068183

[7] T. Ahmad, D. Truscan, and I. Porres, "Identifying worst-case user scenarios for performance testing of web applications using markov-chain workload models," *Future Generation Computer Systems*, vol. 87, pp. 910 – 920, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X18301341

[8] C. Lemieux, R. Padhye, K. Sen, and D. Song, "Perffuzz: Automatically generating pathological inputs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: ACM, 2018, pp. 254–265. [Online]. Available: http://doi.acm.org/10.1145/3213846.3213874

[9] R. S. Sutton, A. G. Barto, F. Bach *et al.*, *Reinforcement learning: An introduction*. MIT press, 1998.

[10] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, Nov. 2017.

[11] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling Network Architectures for Deep Reinforcement Learning," *arXiv e-prints*, p. arXiv:1511.06581, Nov 2015.

[12] M. Plappert, "keras-rl," https://github.com/keras-rl/keras-rl, 2016.

[13] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992698

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[15] M. T. Hagan, H. B. Demuth, M. H. Beale, and O. De Jesús, *Neural network design*. Pws Pub. Boston, 1996, vol. 20.

[16] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine Learning*, vol. 8, no. 3, pp. 293–321, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992699

[17] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Experimental assessment of random testing for object-oriented software," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 84–94. [Online]. Available: http://doi.acm.org/10.1145/1273463.1273476

[18] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 438–444, July 1984.

[19] D. Hamlet, "When only random testing will do," in *Proceedings of the 1st International Workshop on Random Testing*, ser. RT '06. New York, NY, USA: ACM, 2006, pp. 1–9. [Online]. Available: http://doi.acm.org/10.1145/1145735.1145737

[20] C. Amza, E. Cecchet, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel, "Specification and implementation of dynamic web site benchmarks," 2002. [Online]. Available: http://infoscience.epfl.ch/record/55719

[21] F. Abbors, T. Ahmad, D. Truscan, and I. Porres, "MBPeT - A Model-Based Performance Testing Tool," in *4th International Conference on Advances in System Testing and Validation Lifecycle*. IARIA, 2012, pp. 1–8.

[22] L. V. Fausett *et al.*, *Fundamentals of neural networks: architectures, algorithms, and applications*. prentice-Hall Englewood Cliffs, 1994, vol. 3.

[23] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[24] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv e-prints*, p. arXiv:1412.6980, Dec 2014.

[25] Q. Luo, A. Nair, M. Grechanik, and D. Poshyvanyk, "FOREPOST: finding performance problems automatically with feedback-directed learning software testing," *Empirical Software Engineering*, vol. 22, no. 1, pp. 6–56, Feb 2017. [Online]. Available: https://doi.org/10.1007/s10664-015-9413-5

[26] D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik, "Automating performance bottleneck detection using search-based application profiling," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 270–281.

[27] B. Chen, Y. Liu, and W. Le, "Generating performance distributions via probabilistic symbolic execution," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 49–60.