# Automatic exploratory performance testing using a discriminator neural network

Porres, Ivan; Ahmad, Tanwir; Rexha, Hergys; Lafond, Sébastien; Truscan, Dragos

*Published in:*
2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)

*DOI:*
10.1109/ICSTW50294.2020.00030

Published: 01/01/2020

*Document Version*
Accepted author manuscript

*Document License*
Publisher rights policy

# Automatic exploratory performance testing using a discriminator neural network

Ivan Porres, Tanwir Ahmad, Hergys Rexha, Sébastien Lafond, Dragos Truscan
Faculty of Science and Engineering,
Åbo Akademi University
Turku, Finland
name.surname@abo.fi

***Abstract*—We present a novel exploratory performance testing algorithm that uses supervised learning to optimize the test suite generation process. The goal of the proposed approach is to generate test suites that contain a large number of positive tests, revealing performance defects or other issues of interest in the system under test. The key idea is to use a deep neural network to predict which test could be positive and to train this network online during the test generation process, designing and executing the test suite simultaneously. The proposed algorithm assumes that the system under test is stateless and the outcome of the tests is deterministic. Also, only integer and floating point inputs are supported. Otherwise, the approach is completely automatic and it does not require any prior knowledge about the internals of the system under test. It can also be used effectively in a continuous integration setting where small variations of a system are tested successively.**

**We evaluate our algorithm using two example problems: searching for bottlenecks in a web service and searching for efficient hardware configurations in a single-board computer. In both examples, the presented algorithm performed several times better than a random test generator and significantly better compared to our previously published algorithm, producing test suites with a large proportion of positive tests.**

## I. Introduction

Performance is a crucial quality of today's IT systems and can be regarded from different perspectives, such as throughput, response time, resource utilization, etc. Performance problems, ranging from degradation in performance indicators to system failures, can result in unhappy customers and end users, and consequently may produce financial losses for stakeholders. According to statistics by Dun&Bradstree, 59% of the Fortune 500 companies experience a downtime of 1.7 hours a week amounting to a projected $46 million impact annually for companies of over 10,000 employees [18]. In fact, according to different studies [20], there are higher chances of a system to fail due to performance issues rather than functional ones.

Performance testing is the process of identifying how a system performs in terms of stability and responsiveness under increased usage. The end goal of performance testing is to find and eliminate *performance defects* [19], [20] by creating and executing tests against the system and observing the performance of the system via *key performance indicators* (KPIs). Compared to functional testing, where the tests are meant to reveal errors that manifest in the wrong output, in performance testing we are interested in finding tests which will make the KPIs exceed target values.

Performing performance testing in practice is a challenging task not only because it is typically performed in the last phases of the development process, where the system is typically regarded as a black-box, but also because in complex systems the number of possible test inputs can be very large which will make it impractical to test the system exhaustively. The latter is specifically observed in systems which have many configuration parameters, which will result in a combinatorial explosion problem.

In this paper, we propose an algorithm for performance testing of systems with large input spaces, that regards the system under test as a black box and that learns about the system performance during the test generation process. The main idea is to train a deep neural network [11] to select the tests that may reveal performance defects. The neural network starts without previous knowledge of the system, but learns about its performance while the tests are being generated. This allows the algorithm to produce test suites that reveal many performance defects, while executing a limited number of tests against the target system.

The algorithm is fully automatic and only requires a description of the inputs of the system under test and a test driver with a performance testing oracle. We consider that the use of a fully automated performance test generation having no prior system knowledge can be beneficial in many settings but especially in a continuous integration (CI) pipeline. In this setting, performance tests could be executed automatically and continuously after each build, without human intervention.

We proceed as follows. In the next section, we present previous work related to our proposed new algorithm. Section III formalizes our test generation problem and describes our assumptions about the systems to be tested using our approach. We proceed to present our main algorithm in Section IV and its implementation in the Python language in Section V. We evaluate the implementation with two example systems in Section VI. Section VII discusses how to use the algorithm in a continuous integration pipeline efficiently. Finally, we conclude in Section VIII with some closing remarks.

## II. Related work

Many existing approaches focus on the generation of test data for functional testing [16], [8]. Most of these approaches aim to find combinations of input values according to certain test adequacy criteria for code coverage [10], including branch coverage [4] and path coverage [2]. These approaches often rely on the source code or the internal structure of the SUT in order to generate test data.

GA-Prof [17] performs search-based application profiling to detect performance bottlenecks using a genetic algorithm to guide the search process. It relies on the source code of the system under test to map test inputs to different methods and then to relate the methods to different performance bottlenecks. PerfFuzz [12] is also a performance testing approach that uses mutational fuzzing to find program inputs that can reveal worst-case algorithmic complexity in different parts of the program under test. PerfFuzz begins the test generation process with a set of randomly generated inputs. Then in each iteration, it generates new inputs by mutating the previous inputs and saving the ones that increase code coverage.

When compared to GA-Prof and PerfFuzz, our approach uses machine learning over the observed behavior of the system and therefore it is a black-box approach that does not require the source code of the system under test.

FOREPOST [13] is a performance testing approach that uses a feedback-oriented machine learning approach to find performance problems. It extracts a set of rules that map the application performance to certain combinations of input variables. The main idea is to use a rule learning algorithm, which provides a set of rules to guide the selection of test inputs. A drawback of the rule learning approach of FOREPOST is that it does not provide rigorous exploration and extensive coverage of the input space.

The work by Aichernig et al. [3] on performance testing uses a neutral network to learn the response time of IoT systems and predict their performance under different usage scenarios.

We have published a performance testing algorithm named PerfXRL [1] that uses reinforcement learning to guide the test generation process and overcome the main drawback of FOREPOST. The algorithm presented in this article is built using the experiences learned from PerfXRL and uses supervised learning to train a discriminator network. We show in Section VI how our new algorithm improves the defect finding rate compared to our previous work.

## III. Problem description

We aim to perform automatic exploratory performance testing of a digital system equipped with a dichotomic oracle. We assume that there is a deterministic test driver that can execute individual tests: given a system input, the driver will invoke the system with the concrete input values, monitor it and, with the help of the oracle, determine if that particular input leads to a positive test. We consider a positive test a system input that leads to excessive response time, excessive resource usage or unwanted behavior, as determined by the oracle. Similarly, a negative test does not lead to an observable performance issue.

The output of the exploratory testing process is a set of tests called a test suite. We assume that executing each test in the target system is an expensive operation and we are restricted by a test budget that limits the number of tests that can be executed. We thus restrict the test suite to the size allowed by the test budget.

We define the positive predictive value ($ppv$) of a test suite as the ratio between the positive tests and the total number of tests in the suite. Our goal then is to generate a test suite with a given budget that maximizes the expected $ppv$.

We do not have any knowledge about the system under test, except for its input and output space. We assume that the system is stateless from a performance point of view, that is, the outcome of a performance test does not depends on what tests have been executed previously. Finally, we require that the system under test is online, that is, we can use the test driver to learn the outcome of a given test during the test suite generation. This way, the test generation algorithm can learn about the system under test and aim to produce test suites with a high $ppv$.

### A. Input space

We assume that the system under test has an input space represented by finite, nonempty set $I$. The input space can model the API of the system, describing all possible methods and their input parameters, a set of system configuration parameters or a combination of both. Each input in the space $I$ is represented as a finite tuple, and each element of this tuple is an integer or floating-point number.

We partition the input space into two sets, $I = I_p \cup I_n$, that describe the system inputs representing positive and negative tests. We define the positive input density $pid$ of the input space as the ratio between the number of positive inputs and total inputs $pid(I) = \frac{|I_p|}{|I|}$. This is the probability for any given input to be a positive test.

### B. Test execution and test oracle

We assume that there is a test driver that can execute a test $t \in I$, and determine its outcome. We require that test execution is deterministic and stateless. That is, the outcome of a test does not vary if we repeat the test and it does not depend on the previously executed tests. In the rest of the article, the function $execute\_test(t)$ returns the outcome of a test while the boolean function $is\_positive$ tells us if an outcome is positive or not.

### C. Positive predictive value of a test suite and its bounds

As discussed previously, we use the positive predictive value ($ppv$) as the main metric for the effectiveness of a test suite. Given a test suite $T$, containing $T_p \subseteq I_p$ positive tests, $T_n \subseteq I_n$ negative tests, and $T = T_p \cup T_n$, we define the positive predictive value of $T$ as:

$$ppv(T) = \begin{cases} 0 & \text{if T is empty} \\ \frac{|T_p|}{|T|} & \text{otherwise} \end{cases}$$

The number of positive tests in a test suite cannot be in any case larger than the number of positive inputs of a system nor the size of the test suite. Similarly, the number of positive tests cannot be negative nor smaller than the difference between the test suite size and the number of negative inputs. Formally, the number of positive tests is constrained by $|T_p| \leq |T|, |T_p| \leq |I_p|, |T_p| \geq 0, |T_p| \geq |T| - |I_n|$. Thanks to these constraints, we can define the following bounds for the positive prediction value:

- the lower bound for $ppv(T)$ is $\frac{\max(|T|-|I_n|,0)}{|T|}$
- the upper bound for $ppv(T)$ is $\frac{\min(|T|,|I_p|)}{|T|}$

However, in most cases, the size of the input space is much larger than the test suite. In the rest of the text, we assume that $|T| < |I_p|$ and $|T| < |I_n|$ . In this case, the lower and upper bounds of $ppv$ are $\frac{0}{|T|}$ and $\frac{|T|}{|T|}$, i.e. $[0, 1]$.

### D. A random test generator

A simple strategy for test suite generation in explorative testing is random testing, where each possible input has the same probability to be selected for the test suite. If the input space contains $|I|$ unique values, and $|I_p|$ of these correspond to positive tests, a random test generator $RTG$ will produce tests suites with an expected ppv corresponding to the positive input density of the system under test: $E[ppv(T \sim RTG)] = \frac{|I_p|}{|I|}$.

Since we expect that few inputs lead to positive tests, $|I_p| << |I|$, random testing is not an effective approach. In the next section, we propose an alternative algorithm based on deep learning that can produce suites with a better $ppv$.

## IV. A DEEP LEARNING TEST GENERATOR USING A DISCRIMINATOR NETWORK

In this section, we describe a novel algorithm, named Online DN Testing algorithm or DN. The algorithm is based on the idea of using a deep neural network working as a test discriminator. The algorithm constructs the test suite iteratively while learning about the system under test.

The overall approach is shown in Figure 1. The intuition behind the algorithm is as follows: During the first iterations, the algorithm behaves like a random tester sampling the input space. We expect that it will find some positive tests by chance, and those can be used to train a neural network. The neural network can then be used to filter out tests that may have a negative outcome, thus becoming a negative test discriminator. Eventually, the algorithm will become more and more selective about what tests should be included in the suite and it will generate a test suite with a high positive predictive value of $ppv$.

In order to avoid overfitting the discriminator, we include a two-arm bandit [9] that chooses between exploiting the prediction from the discriminator versus continue exploring the input space. The bandit favors exploitation once the test suite has reached a high $ppv$.

### A. The online DN testing algorithm

We formalize this schema in Algorithm 1. Here, the function *uniform* selects an element randomly from a given input set. The function *train* returns a neural network fitted with the data passed as a parameter, while the function *predict* returns the prediction for a given input and network.

**Algorithm 1:** Online DN testing algorithm

```
1  input input space I, budget
2  requires budget ≤ |I|
3  T := ∅ ; D := ∅;
4  DN := new model;
5  ϵ := k1 + k2;
6  while |T|<budget do
7      t := uniform(I-[T]1-D);
8      if predict(DN,t) ≥ threshold ∨ uniform([0,1]) ≤ ϵ
         then
9          outcome := execute_test(t);
10         T := T ∪ {(t,outcome)};
11         ppv := {(t,o)∈T:is_positive(o)} / |T| ;
12         ϵ := k1 * (1 − ppv) + k2 ;
13         DN := train(T);
14     else
15         D := D ∪ {t};
16     end
17 end
18 result test suite T, ppv
```

The algorithm works as follows. Lines 3–5 initialize the test suite (T) and the set of discarded tests (D) to an empty set, the discriminator network, and initializes the $\epsilon$ value for the bandit.

Lines 6–17 describe the main loop that is executed while the size of the test suite is less than the testing budget:

1) The random tester generates an input test that is not in the test suite nor the set of discard tests (L7).
2) The randomly generated test is fed into the discriminating network, which classifies the test as positive or negative (L8).
   a) If the discriminator prediction is positive, then the test will be executed against the SUT (L8, the first clause of the disjunction).
   b) If the discriminator prediction is negative, a two-arm bandit chooses between exploitation by following the advice of the discriminator and discarding the test or exploration by acknowledging the random tester and executing the test anyway (L8, second clause).
3) If the test should be executed:
   a) The test is executed against the SUT and the test (L9) and its outcome is added to the test suite (L10). The $ppv$ of the suite is updated (L11).
   b) The strategy of the exploitation-exploration bandit is updated using the current test suite $ppv$ (L12).

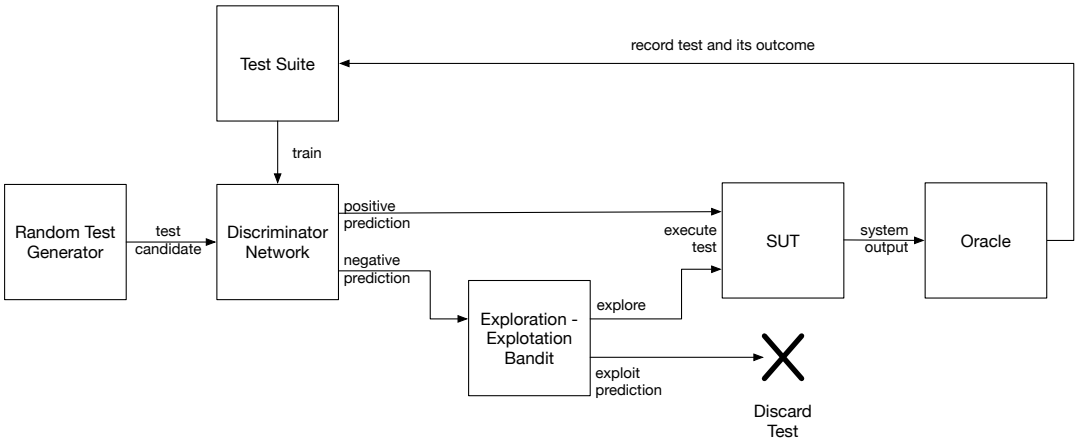


Fig. 1. Overview of the Online DN testing algorithm

c) The discriminator is trained using the test suite as training data (L13).

4) If the test should not be executed then it is added to the set of discarded tests (L15).

In the rest of this section, we discuss different details about the inner working of the algorithm.

### B. Feature modeling and neural network architecture

As we have discussed, the proposed algorithm uses an artificial neural network to predict tests that may be positive. The architecture of the neural network used in our implementation consists of three sequential dense layers with 64 neurons each and one output layer.

The input layer has the same dimensions as the SUT input space. All inputs, including those representing categorical parameters, are normalized in the range $[-1, 1]$. The sequential layers use a $tanh$ activation function.

The output layer has only one output, representing the prediction for a test to be positive and uses a $Sigmoid$ activation function.

The optimizer used for network training is RMSprop. It is an improvement over Rprop [15] proposed by Geoffrey Hinton and it is expected to perform better when the network is trained in multiple small batches, as it is done by our algorithm.

### C. Exploitation-exploration bandit

The algorithm includes a two-arm bandit to choose between the exploitation of the neural network predictions and the exploration of the input space. In our design, the bandit follows a rather simple Epsilon-Greedy strategy [9]. Initially, the $\epsilon$ value should be high enough to allow for enough exploration of the input space. Once the test suite $ppv$ becomes high enough, the exploration should be reduced.

In our algorithm, the $\epsilon$ value is adjusted at each iteration according to the formula $k_1 * (1 - ppv) + k_2$ and we require that $k_1 > k_2, k_2 > 0$. Initially, $ppv$ equals 0 and we expect it to grow towards 1 during the test generation process. In the experiments described below, we use $k_1 = 0.01, k_2 = 0.001$ with good results.

Termination of the main loop is guaranteed if $k_2 > 0$ since the two-arm bandit always has a chance to include a newly generated test in the suite.

## V. Implementation

We have implemented the proposed Online DN testing algorithm using the Python programming language. The implementation uses the Keras [6] deep learning library to train the discriminator network and predict the test outcome.

While the described algorithm works with one test per iteration, the actual implementation works in batches for performance reasons. In our experiments, we use a batch size of 50 tests. That is, the neural network is (re)trained only after we have executed 50 new tests. The batch size can be adjusted based on the relative effort to train the neural network with respect to executing the tests.

The implementation uses an interface to represent the system under test called SUT. The SUT interface has only four methods, one to return a sample of a given size of the input space of the system under test, another method to execute a test and return its outcome and two methods to normalize and denormalize the values in the input space into the range expected by the discriminator network $[-1, 1]$.

We consider that thanks to this simple SUT interface the algorithm can be used in many different systems as long as they match the requirements described in Section III.

## VI. Experimental results

We have evaluated the proposed test generation algorithm using two systems: RUBiS, a web application, and Odroid, a single board computer. In RUBiS, the objective is to discover web service requests that may exceed an expected response time. In the case of Odroid, the objective is to study how different hardware configuration parameters affect the performance, power efficiency and total power consumption of the

TABLE I
TASK PARAMETERS

| Task | Size Input Space | Positive Inputs | $pid$ | Test Suite Size | bounds $ppv$ |
|---|---|---|---|---|---|
| 1 RUBiS$_{POI}$ | 3 100k | 250 184 | 0.08 | 40 000 | [0,1] |
| 2 RUBiS$_{UNI}$ | 3 100k | 283 391 | 0.09 | 40 000 | [0,1] |
| 3 Performance | 479 600 | 15 476 | 0.03 | 12 000 | [0,1] |
| 4 Power | 479 600 | 4 736 | 0.01 | 4 000 | [0,1] |
| 5 Efficiency | 479 600 | 47 900 | 0.10 | 12 000 | [0,1] |

TABLE II
MEAN TEST SUITE $ppv$

| Task | Test Suite Size | $ppv$ Random | $ppv$ PerfXRL | $ppv$ DN | DN / PerfXRL |
|---|---|---|---|---|---|
| 1 RUBiS$_{POI}$ | 40k | 0.08 | 0.36 | 0.87 | x 2.4 |
| 2 RUBiS$_{UNI}$ | 40k | 0.09 | 0.25 | 0.84 | x 3.4 |
| 3 Performance | 12k | 0.03 | 0.21 | 0.83 | x 3.9 |
| 4 Power | 4k | 0.01 | 0.02 | 0.58 | x 29 |
| 5 Efficiency | 12k | 0.10 | 0.36 | 0.93 | x 2.6 |

system running a standard benchmark. In total, we evaluate the algorithm in 5 different test generation tasks.

The main parameters for the tasks are described in Table I. This table shows for each task the size of the input space, the total number of inputs that lead to a positive test, the positive input density, the size of the test suite to generate and the lower or upper bounds for the $ppv$ of the generated test suites. We have performed an exhaustive search in our example systems in order to generate this table. However, we should note that usually the number of positive inputs is unknown. As a consequence, it is not either possible to know the $pid$ value and the bounds for the test suite $ppv$.

### A. RUBiS

RUBiS [5] is a web-based application that implements the core functionality of an auction site. It has been widely used in academia for performance evaluation in many publications. RUBiS is accessed via an HTTP interface. Each HTTP request contains 4 integer parameters, and the total size of the input space is 3 100 000 unique requests.

In order to evaluate our algorithm, we deliberately inject performance bottlenecks in RUBiS. In a bottleneck, the time used to process a request is increased to exceed the required maximum response time threshold. We set up two performance testing tasks using a different bottleneck injection method in each task. In the RUBiS$_{UNI}$ task, the clusters of bottlenecks are distributed uniformly [7] which is a challenging approach since the bottlenecks do not follow any heuristic or pattern. In the second task, RUBiS$_{POI}$, we used a Poisson distribution [7] to distribute the clusters of bottlenecks. In the case of RUBiS$_{POI}$, as opposed to RUBiS$_{UNI}$, the bottlenecks are packed together in the input space. In total, we have injected bottlenecks on 250 184 and 283 391 unique combinations of the input values in RUBiS$_{POI}$ and RUBiS$_{UNI}$, respectively. This gives a positive input density $pid_{POI} = 250184/3100000 \approx 0.08$ and $pid_{UNI} = 283391/3100000 \approx 0.09$

To demonstrate the performance of the algorithm, we collect at each iteration the total number of positive tests generated, the $ppv$ for the test suite and the $ppv$ for the tests generated at each batch. We run the algorithm 30 times for each task and show the mean, maximum and minimum statistics for the collected variables. We also compare our new algorithm (DN) with a random testing algorithm (Random), and our previous performance exploration approach (PerfXRL), presented in [1].

We represent experimental results for RUBiS$_{POI}$ and RUBiS$_{UNI}$ in Figure 2. In the left plot, we can observe that our newly proposed algorithm clearly outperforms the existing approaches by generating test suites with many more positive tests. The center and right plot show how the discriminator network learns about the system at each iteration. As expected, initially it does not perform much better than a random tester, but it soon learns the underlying distribution of the performance issues in the tested systems and starts generating tests that have a high chance of being positive. In the case of RUBiS$_{POI}$, the algorithm achieves a batch $ppv$ of 0.9 after 300 tests. This means that after executing 300 tests in the system, the algorithm is generating tests that are positive in 90% of the cases. Please note that the theoretical maximum for the $ppv$ is 1, and achieving this result requires complete prior knowledge of the system under test. In comparison, the random tester generates tests that are positive in 9% of the cases, as expected by the positive input density of the system.

We can observe similar results during the performance test exploration of the RUBiS$_{UNI}$ task. In this case, the discriminator requires 500 tests to achieve a $ppv$ of 0.9. This is expected because the bottlenecks were injected following a more challenging uniform distribution. Still, the new algorithm clearly outperforms our previous work and the random tester.

Table II shows the mean test suite $ppv$ achieved by each algorithm. We consider this as the main performance indicator for this problem. We can observe that for the RUBiS$_{POI}$ task, the new DN algorithm performs more than two times better than our previous PerfXRL algorithm and more than 10 times better than the random tester. In the case of RUBiS$_{UNI}$, the new algorithm performs 3.4 times better than the previous one.

We should note that we generated test suites with 40 000 tests. We consider that this number is rather large for any practical purposes, but it is used in our experiments to demonstrate how the algorithm performs in large searches.

### B. Odroid

In our second experiment, we used an ODROID XU3 development board from HARDKERNEL. Its Exynos 5422 processor implements the ARM big.LITTLE architecture with two clusters composed of 4 cores each. The big cluster consists of a high-performance Cortex-A15 quad-core CPUs, and the little cluster is a low power Cortex-A7 quad-core CPUs. The board also contains a Mali-T628 GPU and 2GB LPDDR3

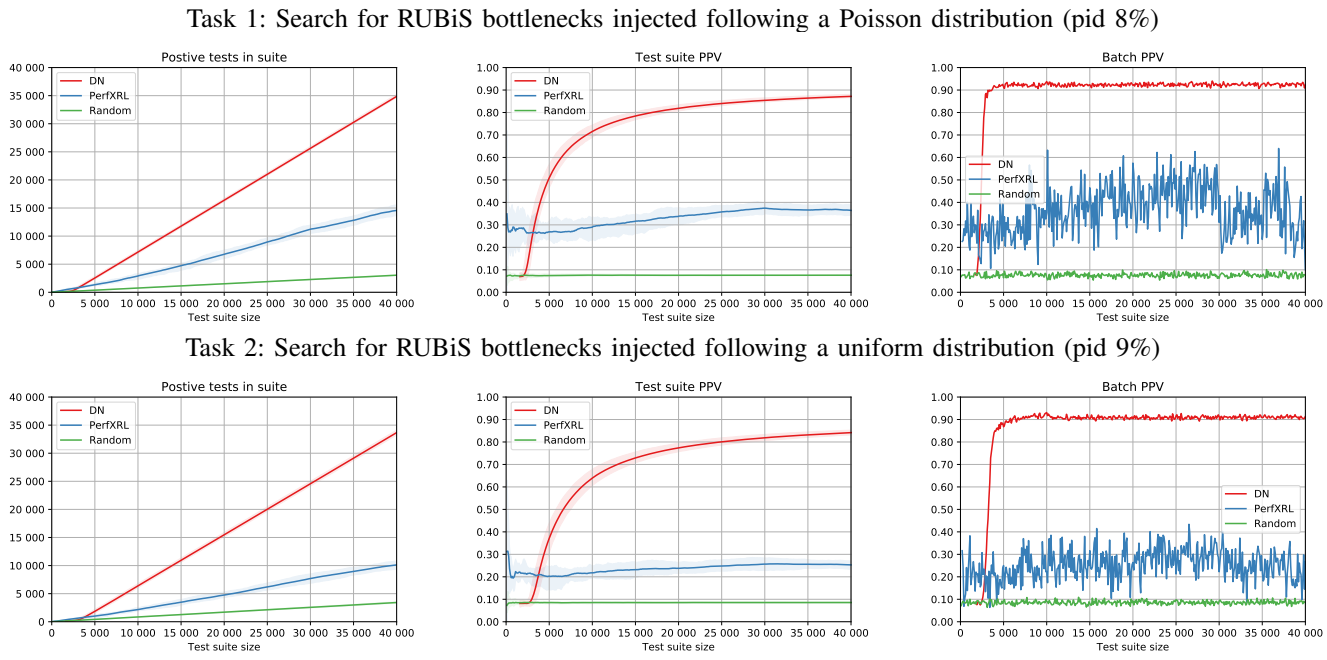


Fig. 2. Experimental results for the RUBiS tasks

of memory. The board has 4 current sensors that offer the possibility to measure power dissipation in four different domains: big cluster (A15), LITTLE cluster (A7), GPU and memory.

As input space, we consider the different board configurations in terms of the number of CPUs, type of CPU, core performance level (i.e. Dynamic voltage and frequency scaling, DVFS, level) and core utilization level. We select the type and number of CPUs via the Linux sysfs virtual filesystem. We use the frequency governor cpufreq in Linux to define 14 frequency levels on the A7 cores, from 200MHz to 1.4GHz and 18 levels on the A15 cores from 200MHz to 2GHz, having a linear impact of the power dissipation of the CPUs. These frequency intervals correspond to 4 discrete voltage levels for driving the cores, having a quadratic impact on the power dissipation of the CPUs.

The core utilization level is set using a scheduler framework, which is based on the concept of setting the CPU bandwidth of a certain process. The scheduler is called sched_deadline and belongs to the Earliest Deadline First class of schedulers. By setting the runtime of a process inside a period we can control the level of CPU it uses. In this experiment, we considered 10 possible core utilization levels from 10% to 100% with 10% steps.

The used input space has therefore 6 dimensions and is defined for each cluster as the number of CPUs, clock frequency and utilization level.

The following three outcomes are observed: a) the board performance measured in the number of executed instructions per second and b) the power dissipation in Watt of the Exynos 5422 processor and c) the board efficiency measured in the number of executed instructions per joule.

The goal of the performance exploration for an embedded platform is usually to find in the input space, i.e. from all possible board configurations, the configurations leading to a defined minimum performance level while avoiding, if possible, the ones leading to a high power dissipation level.

To evaluate the proposed approach we used the workload Core from the EEMBC CoreMark-Pro Suite [1]. There are 479 600 different configurations and executing the benchmark through the entire input space requires around 80 hours. The used experimental settings to evaluate the entire input space through an exhaustive search are equivalent to the one presented in [14]. This makes an exhaustive search approach unpractical in a continuous integration pipeline.

We set up three tasks, one for each outcome. For the first outcome, we search for system configurations where the performance is greater than 8254M instructions/s. After an exhaustive search, we know that 3% of the configurations match this requirement. For the second outcome, we search for configurations where the power is greater than 6 W (1% of the configurations). Finally, for the third outcome, we require the efficiency of more than 2870M instructions/J (10% of the configurations).

We run our test generation algorithm 30 times for each of the three tasks. The mean test suite $ppv$ is represented in Table II while the plots showing the number of positive tests, test suite $ppv$ and batch $ppv$ per iteration are shown in Figure 3. The results show that the new DN algorithm clearly outperforms the existing ones. For the efficiency task, the new

[1]https://github.com/eembc/coremark-pro

algorithm performs 2.6 times better than the previous one, while for the power task it performs 29 times better.

We can also observe how the performance of the algorithm depends on the positive input density of the system. The random test generator needs to feed the neural network with some positive tests before the network starts being accurate enough to discriminate tests. If the positive input density is low the neural network may not get enough tests for training and the algorithm may require more iterations to perform well.

### C. Execution time

We measured the execution time of the test generator algorithm without accounting for the time used executing the tests in the SUT. The results show that our algorithm implementation produced from 120 to 180 tests per second in our experimental setup[2].

The actual execution time depends on how much time is required to run each test in the SUT. We consider that in most cases this will be a slow operation and the time spent in test generation time will be negligible compared with the time spent in test execution.

Currently, most of the time is used in training the discriminator network. The temporal performance of the generator could be improved by using a GPU accelerator for the network training and by optimizing the batch sizes.

## VII. Exploratory Testing in Continuous Integration

The previous algorithm always starts tabula rasa. In a continuous integration setting, where small variations of a system are tested again and again it can be more effective to reuse a discriminator model produced during the build of a previous iteration of the system.

We can update the proposed algorithm to support continuous integration by introducing just some few changes. The key idea is that the discriminator network is initialized by a model created while testing a previous version of the system under test.

The performance of this new algorithm, called CI DN, depends on how similar is the new version of the system under test. We expect that small development increments in the system will lead to small variations in the set of positive inputs and that reusing a previous neural network model can help to produce better test suites.

Still, in some cases, the new version of the system under test can be so different that this approach becomes a hindrance. This can be detected by a sharp reduction in the batch iteration $ppv$ of the algorithm with respect to the previous test suite. In this situation, the previous model can be discarded and the discriminator be retrained using only results from the current system.

The new algorithm is shown as Algorithm 2. It uses a new constant $k_3$ to decide after each iteration (L13) if the previous discriminator model should be reused and updated with the results from the latest test (L14) or if it should be discarded and a new model trained using only tests from the current system version (L15).

**Algorithm 2:** CI DN testing algorithm

```
1  input input space I, budget, model DN', expected_ppv
2  requires budget ≤ |I|
3  T := ∅ ; D := ∅;
4  DN := DN';
5  ϵ := k1 + k2;
6  while |T|<budget do
7      t := uniform(I-[T]1-D);
8      if predict(DN,t) ≥ threshold ∨ uniform([0,1]) ≤ ϵ
         then
9          outcome := execute_test(t);
10         T := T ∪ {(t,outcome)};
11         ppv := |{(t,o)∈T:is_positive(o)}| / |T| ;
12         ϵ := k1 * (1 − ppv) + k2 ;
13         if ppv ≥ expected_ppv * k3 then
14             DN:= update(DN,{(t,outcome)};
15         else
16             DN:= train(T);
17         end
18     else
19         D := D ∪ {t};
20     end
21 end
22 result T, ppv, DN
```

### A. Experimental results

To demonstrate the CI DN algorithm in a continuous integration setting, we repeat the search for Odroid configurations where power > 6W in five successive versions of the SUT. In each version, we have mutated 5% of the positive configurations to simulate changes in the system due to its continuous development.

We present the results in Figure 4. We can observe that it is feasible to reuse the discriminator network between CI builds. The first version of the system is explored without any previous knowledge, and it requires more than 1200 tests until achieving a $ppv$ of 0.5. However, in the next iterations the algorithm can successfully reuse the neural network model and start generating tests with a satisfactory $ppv$ right from the start.

## VIII. Conclusions

We have presented a novel exploratory performance testing algorithm that uses machine learning to optimize the test suite generation process. The goal is to generate test suites that contain a large number of positive tests, revealing performance defects or other issues of interest in the system under test. The key idea is to use a deep neural network to predict what test could be positive and to train this network online during the test generation process, designing and executing the test

[2] 2.3Ghz 8-core Intel Core i9 computer

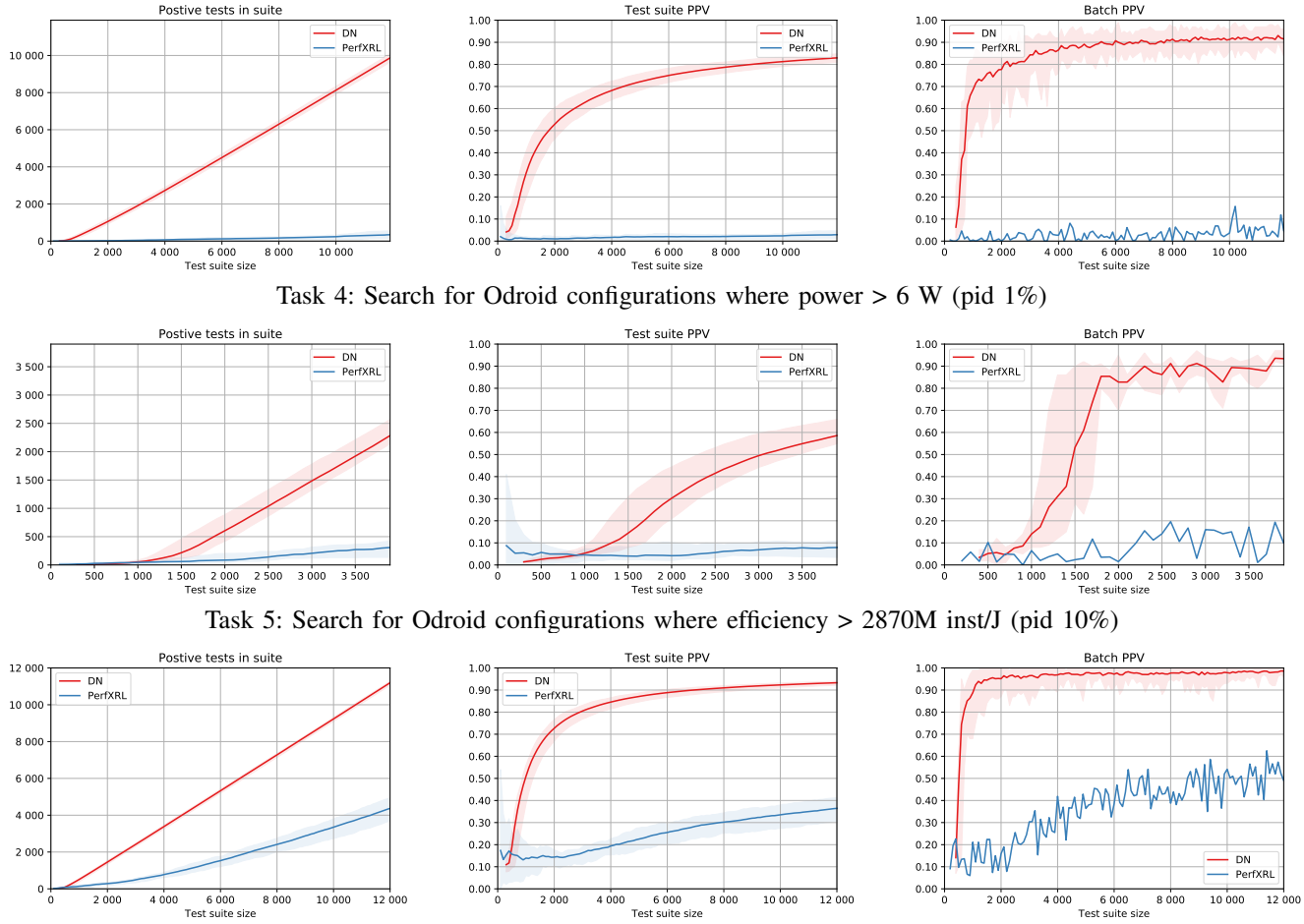


Fig. 3. Experimental results for the Odroid tasks

suite simultaneously. The approach is completely automatic and it does not require any prior knowledge about the internals of the system under test. It can also be used effectively in a continuous integration setting where small variations of a system are tested successively.

The current algorithm has several important limitations. It assumes that the SUT is stateless and the outcome of the tests is deterministic. Also, only integer and floating point inputs are supported.

We have evaluated our algorithm in six different performance exploration tasks for two different systems: searching for bottlenecks in a web service and searching for efficient hardware configurations in a single board computer. In both examples, the presented algorithm performed several times better than a random tester and our previously published algorithm, producing test suites with a large proportion of positive tests. We acknowledge that further experimentation with other systems and performance properties is needed.

As future work, we would like to investigate other approaches to increase the performance of the algorithm during the first iterations of the test generation process and when the positive input density of the system is really low.

## Acknowledgments

This work has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737494. This Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation programme and Sweden, France, Spain, Italy, Finland, and Czech Republic.

## References

[1] Tanwir Ahmad, Adnan Ashraf, Dragos Truscan, and Ivan Porres. Exploratory performance testing using reinforcement learning. In Miroslaw Staron, Rafael Capilla, and Amund Skavhaug, editors, *45th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2019, Kallithea-Chalkidiki, Greece, August 28-30, 2019*, pages 156–163. IEEE, 2019.

[2] Moataz A. Ahmed and Fakhreldin Ali. Multiple-path testing for cross site scripting using genetic algorithms. *Journal of Systems Architecture - Embedded Systems Design*, 64:50–62, 2016.

[3] Bernhard K. Aichernig, Franz Pernkopf, Richard Schumi, and Andreas Wurm. Predicting and testing latencies with deep learning: An iot case study. In Dirk Beyer and Chantal Keller, editors, *Tests and Proofs - 13th International Conference, TAP 2019, Held as Part of the Third*

Fig. 4. Experimental results for the Odroid tasks in a Continuous Integration setting

*World Congress on Formal Methods 2019, Porto, Portugal, October 9-11, 2019, Proceedings*, volume 11823 of *Lecture Notes in Computer Science*, pages 93–111. Springer, 2019.

[4] Nadia Alshahwan and Mark Harman. Automated web application testing using search based software engineering. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 3–12. IEEE Computer Society, 2011.

[5] Amza, Chanda, Cox, Elnikety, Gil, Rajamani, Zwaenepoel, Cecchet, and Marguerite. Specification and implementation of dynamic web site benchmarks. In *2002 IEEE International Workshop on Workload Characterization*, pages 3–13, Nov 2002.

[6] François Chollet et al. Keras. `https://keras.io`, 2015.

[7] Merran Evans, Nicholas Hastings, and Brian Peacock. *Statistical Distributions*. Wiley-Interscience, 3 edition, June 2000.

[8] Robert Feldt and Simon Poulding. Finding test data with specific properties via metaheuristic search. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 350–359. IEEE, 2013.

[9] Jeffrey D. Johnson, Jinghong Li, and Zengshi Chen. Reinforcement learning: An introduction: R.S. sutton, A.G. barto, MIT press, cambridge, MA 1998, 322 pp. ISBN 0-262-19398-1. *Neurocomputing*, 35(1-4):205–206, 2000.

[10] Chahine Koleejan, Bing Xue, and Mengjie Zhang. Code coverage optimisation in genetic algorithms and particle swarm optimisation for automatic software test data generation. In *2015 IEEE Congress on Evolutionary Computation (CEC)*, pages 1204–1211. IEEE, 2015.

[11] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[12] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 254–265, New York, NY, USA, 2018. ACM.

[13] Qi Luo, Aswathy Nair, Mark Grechanik, and Denys Poshyvanyk. FOREPOST: finding performance problems automatically with feedback-directed learning software testing. *Empirical Software Engineering*, 22(1):6–56, Feb 2017.

[14] H. Rexha, S. Holmbacka, and S. Lafond. Core level utilization for achieving energy efficiency in heterogeneous systems. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 401–407, March 2017.

[15] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: the RPROP algorithm. In *Proceedings of International Conference on Neural Networks (ICNN'88), San Francisco, CA, USA, March 28 - April 1, 1993*, pages 586–591. IEEE, 1993.

[16] Davi Silva Rodrigues, Márcio Eduardo Delamaro, Cléber Gimenez Corrêa, and Fátima L. S. Nunes. Using genetic algorithms in test data generation: A critical systematic mapping. *ACM Comput. Surv.*, 51(2):41:1–41:23, 2018.

[17] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Automating performance bottleneck detection using search-based application profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 270–281, New York, NY, USA, 2015. ACM.

[18] Vision Solutions. Assessing the Financial Impact of Downtime. White paper, 2014. Available at https://luminet.co.uk/wp-content/uploads/2014/10/Assessing-the-Financial-Impact-of-Downtime-UK.pdf. Last accessed 11-01-2020.

[19] BM Subraya and SV Subrahmanya. Object driven performance testing of web applications. In *Quality Software, 2000. Proceedings. First Asia-Pacific Conference on*, pages 17–26. IEEE, 2000.

[20] E.J. Weyuker and F.I. Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *Software Engineering, IEEE Transactions on*, 26(12):1147–1156, 2000.