Web Application Testing Using Deep Reinforcement Learning

Mohammadreza Abbasnezhad[1], Amir Jahangard Rafsanjani[1*], Amin Milani Fard[2]

[1]Department of Computer Engineering, Yazd University, Yazd, Iran.
[2]Department of Computer Science, New York Institute of Technology, Vancouver, BC, Canada.

abbasnezhad.m.r@stu.yazd.ac.ir
jahangard@yazd.ac.ir
amilanif@nyit.edu

[*]Corresponding author

Received: 02/05/2023, Revised;19/08/2023, Accepted: 28/10/2023.

Abstract
Web applications (apps) are integral to our daily lives. Before users can use web apps, testing must be conducted to ensure their reliability. There are various approaches for testing web apps. However, they still require improvement. In fact, they struggle to achieve high coverage of web app functionalities. On the one hand, web apps typically have an extensive state space, which makes testing all states inefficient and time-consuming. On the other hand, specific sequences of actions are required to access certain functionalities. Therefore, the optimal testing strategy extremely depends on the app's features. Reinforcement Learning (RL) is a machine learning technique that learns the optimal strategy to solve a task through trial-and-error rather than explicit supervision, guided by positive or negative reward. Deep RL extends RL, and exploits the learning capabilities of neural networks. These features make Deep RL suitable for testing complex state spaces, such as those found in web apps. However, modern approaches support fundamental RL. We have proposed WeDeep, a Deep RL testing approach for web apps. We evaluated our method using seven open-source web apps. Results from experiments prove it has higher code coverage and fault detection than other existing methods.

Keywords
Deep reinforcement learning; Automated testing; Test generation; Web application.

1. Introduction

Web technologies have advanced and innovated at a remarkable rate over the past decades. Web applications (apps) are currently as effective as desktop apps. Besides being competitively convenient, they do not require complicated installation. According to the most recent survey by Internet World Stats [1], nearly 68 percent of the world's population uses web apps. With the widespread adoption of web apps, there is a greater need to improve their reliability. This is especially true as more users rely on web apps to perform daily tasks and as more companies rely on these apps to drive their enterprises. There are also many complexities associated with web apps, such as sophisticated business logic implemented in multiple languages on the client and server sides. Since the majority of their success depends on user feedback, it is essential for them to be reliable while they are being used. Consequently, an effective testing phase is essential for reducing the likelihood of web app failures.

A web app is composed of one or more HTML pages, which serve as the user interface (UI). A Document Object Model (DOM) [2] represents each HTML page during runtime. DOM specifies the logical structure of web pages and the means by which they are accessed and altered. DOM can be changed dynamically by its API which generates various states for web apps. Each DOM or state contains various UI elements, such as buttons, links, and so on. A web app is event-based, which implies that its behavior is determined by user actions such as button clicks. Because of the event-based nature of web apps, numerous approaches [3]–[7] have focused on auto-generating events to test web apps.

Regardless of the testing strategy, the objective is to provide events that disclose states associated with web app functionalities. However, significant obstacles exist in this area. The state explosion challenge [8] refers to the fact that web apps typically comprise a huge state space because of a large number of elements and potential events. For web apps, testing the vast space of all possible events, states, and their transitions takes a lot of effort and is challenging to scale. As a result, automated testing methodologies must only test the sequences of events and states that are important to the web app's functionalities. Notably, certain web app functionalities can only be accessed through a specific sequence of events.

There are various techniques for web app automated testing that attempt to maximize code coverage and fault detection during testing. Using random events, random testing strategies [3] stimulate the Application Under Test (AUT). However, when dealing with difficult transitions,

random testing without direction may get stuck. model-based solutions [4]–[6] create a behavioral model of the AUT and then utilize it to test the web app and produce events. In this instance, high-quality models are crucial for achieving a successful testing outcome. Nevertheless, the constructed behavioral model may only encompass a portion of the web app's functionalities, thereby limiting the effectiveness of the generated test cases.

Recent research on Reinforcement Learning (RL) [9] has shown that it can learn a policy to test web apps [7]. RL is an approach in machine learning that does not require a labeled training set as input because the learning process is guided by the positive or negative reward encountered during task execution. As a result, it represents a method for dynamically developing an appropriate testing approach based on previous successful or unsuccessful experiences. Even though RL has been used to address the issue of web testing [7], only the most fundamental form of RL—namely, tabular RL—has been used to test web apps so far. In tabular RL, the state-action values are saved in a table. Deep neural networks replaced tabular techniques with Deep learning approaches, in which the action-value function is learned from a neural network's previous positive and negative experiences. Deep RL has proven significantly superior to tabular RL [10], [11] when the state space is large (e.g., many events and states within a web app). In this regard, we argue that the state space of web apps is an excellent candidate for the effective use of Deep RL rather than tabular RL for testing reasons.

This article introduces the first Deep RL approach for automated web app black-box testing, named WeDeep (**We**b Application Testing Using **Deep** Reinforcement Learning). WeDeep employs a Deep neural network to discover the optimal testing strategy by analyzing past attempts. It achieves high scalability and the ability to manage complex web app functionalities because of this Deep neural network. WeDeep applied to a benchmark of seven web apps. The benchmark compared WeDeep's performance to that of state-of-the-art testing approaches for web apps, such as WebExplor [7] and DIG [6]. The experimental results confirmed the hypothesis that Deep RL outperforms tabular RL in testing the state space of web apps, with WeDeep detecting the most of the faults and achieving the highest code coverage.

The contributions of this paper are summarized as follows:

- We propose WeDeep, the first testing approach based on Deep RL.
- We present an empirical evaluation of the proposed approach. Results show that our approach outperforms existing ones in terms of both code coverage and fault detection.

The remainder of this paper is organized as follows: Section 2 gives a motivation example and an introduction to Deep RL. Section 3 surveys related work. Section 4 describes our Deep RL based testing approach. Section 5 presents an empirical evaluation of our proposed approach on seven web apps. Section 6 concludes the paper and outlines future work.

2. Motivation and Background

In this section, we describe our motivation for our research, followed by some background on Deep RL.

2.1. Motivating

Testing web apps is one of the primary activities that contributes significantly to web app quality [12], [13]. We view the automatic testing of web app as a problem involving the generation of action sequences to attain various web app states. To put it another way, carrying out tasks in the appropriate order can assist in detecting possible faults. To that end, it will be necessary to devise an efficient testing method to enable us to visit a wider variety of states within the available time constraint. Some states can only be reached via specific action sequences; thus, the approach should be able to effectively create those as well as cover a wide variety of states.

For example, Claroline [14] is one of our subject web apps. It is a collaborative learning environment that enables academic institutions or teachers to design and manage online courses. To create an announcement for their students in the Claroline web app, a teacher should perform the following actions: typing username, typing password, clicking on the login button, clicking on the target course, clicking on the announcement link, clicking on the add announcement button, typing the title, typing the description, and clicking on the OK button. If there is a fault in the creation of an announcement, this path will lead to its discovery. Nevertheless, web apps typically have a large state space due to the large number of elements and possible actions [8], making it difficult to generate valid action sequences efficiently. Any interruption during the action sequence would prevent the objective state from being reached. In the described sequence, if in the final stage another element, such as the course homepage, is clicked instead of the OK button, neither an announcement nor the potential fault will be generated.

We observe that if an agent can imitate human behavior with the app, it can generate valid action sequences efficiently. For instance, if the agent clicks on the OK button after filling out the title and description fields, an announcement will be generated rather than the course homepage. Motivated by this observation, we design a Deep RL agent, which has been shown to be vastly superior to tabular RL [10], [11], when the state space is large (e.g., many actions and states within a web app). Our Deep RL agent generates actions for which the action sequence generated is consistent with human behavior. In fact, action sequences are generated by the Deep RL agent to effectively attain beneficial states.

2.2. Deep RL in brief

A model-free RL technique called Q-learning [15] aims to learn a policy for any Markov decision process by identifying the best possible policy, π, to maximize the expected cumulative reward for a series of actions. Q-learning is based on trial-and-error learning, in which an agent interacts with the environment and assigns Q values, which are approximated values, to each state-action pair.

As depicted in Fig. 1, the agent interacts iteratively with the environment. Assuming S and A are the sets of all states and actions, at each iteration t, the agent selects and executes an action $a_t \in A$ based on the current state $s_t \in S$. s_t and a_t represents the state and action at time t, respectively. After performing the action, the agent can

observe a new state $s_{t+1} \in S$. In the meantime, an instant reward $r_t = R(s_t, a_t)$ is received. This is the immediate reward for doing action a_t in state s_t. The agent will then use the Bellman equation [16] to update the Q value, as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha * (r_t + \gamma * \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \quad (1)$$

α is a learning rate between 0 and 1 and γ is a discount factor between 0 and 1 in this equation. After being learned, these Q values can determine the optimal behavior in each state by selecting the actions $a_t = \arg\max_{a_{t+1}} Q(s_t, a_{t+1})$.

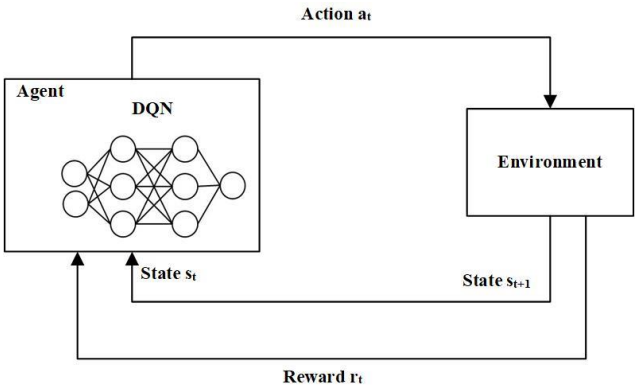

Fig. 1: Deep RL overview

Deep Q-Networks (DQN) are used to scale traditional Q-learning to larger state and action spaces [10], [11]. $Q(s_t, a_t)$ are stored and visited in a Q-table for traditional Q-learning. It can only manage state and action spaces with low dimensions. As shown in Fig. 1, DQN is a multi-layered neural network that outputs Q values for each action a_t in a given state s_t, i.e., $Q(s_t, a_t)$. DQN can scale more complicated state and action spaces because a neural network can input and output high-dimensional state and action spaces. In contrast to a Q-table, a neural network can generalize Q values to previously unobserved states. It employs the following loss function [10], [11] to modify the neural network in order to reduce the error:

$$loss = \left(r_t + \gamma * \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right)^2 \quad (2)$$

In other words, the neural network is trained to predict the value of Q as follows, given the input (s_t, a_t):

$$Q(s_t, a_t) = r_t + \gamma * \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \quad (3)$$

In a training sample, therefore, the input is (s_t, a_t) and the output is the corresponding Q value, which can be calculated as $r_t + \gamma * \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$.

3. Related Work

Various techniques have been suggested for web testing. In the following, we will provide a concise discussion of the most important solutions and their limitations, which will highlight the need for an innovative and effective approach to web testing.

Random testing techniques, such as Crawljax [3] by Mesbah et al., generate and send random events to a web app. Although this technique is straightforward, there is no guidance to ensure that the testing is efficient. Testing therefore comprises numerous inefficient or repetitive actions and has a low likelihood of exposing difficult-to-reach functionalities. Some states are more accessible than others, so they will be executed more frequently under random strategy, whereas some states that are difficult to reach may not be executed at all.

The model-based method is a major paradigm that may be used to achieve automatic web testing [4]–[6]. In order to detect faults in web apps, this method creates models to characterize their behaviors in advance, and then produces test cases from these models. The model is a directed graph in its most basic form, with nodes representing various DOM states and edges representing the event-driven transitions connecting those states.

ATUSA [4] by Mesbah et al. finds the shortest path from the main page to each of the nodes that has no outgoing edges. A test case is created in this manner. Biagiola et al. created SubWeb [5], a search-based approach to web testing. SubWeb samples the input space iteratively, selecting the most suitable candidates for test cases and evolving them using genetic search operators to generate new test cases. However, defining an effective fitness function requires a manual analysis of behavioral model data. This is a tedious and lengthy process for testers. In fact, this kind of information relies on the business logic of the web app, so it cannot be made fully automatic. In addition, the evaluation of the fitness function is expensive because a large number of candidates must be generated and executed in the browser prior to convergent on an adequate set of tests.

Biagiola et al.'s DIG [6] is a diversity-based method for generating web tests that draws inspiration from adaptive random testing [17]. DIG prioritizes candidate tests by selecting those that are most different from those that have already been generated. DIG can evaluate a large number of candidate test cases without executing them in the browser, resulting in quicker test generation than SubWeb. DIG, unlike SubWeb, is automated and does not require any manual effort to develop test cases.

The completeness of the derived behavioral model has a significant impact on the effectiveness of the generated test cases in model-based testing. It's likely that the model will only include some web app functionalities while excluding others from testing. In addition, web apps frequently have dynamic content updates (through languages like JavaScript and the DOM API), which are difficult for behavioral models to capture. It is also necessary to have expert knowledge of the subject in order to construct high-quality models [6]. In order to create a behavioral model, it is common practice to randomly traverse the UI of a web app. However, this technique is redundant and limited when considering web apps. Model-based methods tend to make test cases that cannot be run on web apps and are not useful [6].

Similarly, research has investigated the use of RL for web app testing. WebExplor [7] by Zheng et al. utilizes RL, allowing it to anticipate and generate test cases

incrementally while interacting with a web app. WebExplor employs Q-learning [15], a model-free RL method, based on curiosity reward to accomplish testing. Similarly, QExplore [18] is another existing work that is most relevant to both the RL and web domain, since it uses RL to develop behavioral models for web apps while interacting with them. Both WebExplor and QExplore, in contrast to our study, are based on the most basic type of RL, tabular RL. In contrast, WeDeep learns the action-value function based on Deep RL during its interaction with the AUT. To the best of our knowledge, WeDeep is the first Deep RL-based approach that conducts testing for web apps and outperforms state-of-the-art methods in terms of effectiveness.

4. Proposed Approach

This section covers WeDeep (**We**b Application Testing Using **Deep** Reinforcement Learning), our proposed Deep RL-based approach to testing web apps. In Fig. 2, we can see the main building blocks of the proposed approach, which are Browser, DOM Analyzer, DQN, Action Selector, Observer, Calculator, and Memory.

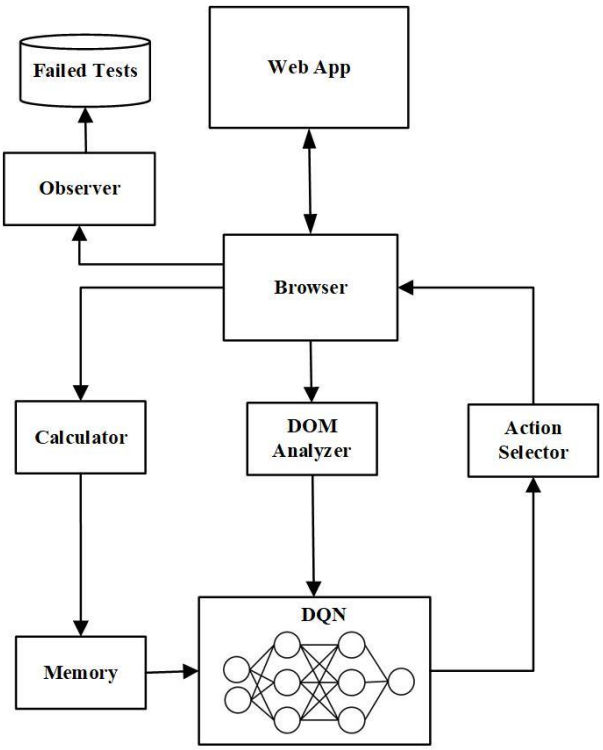

Fig. 2: Proposed approach Overview

It is the responsibility of Browser to provide a common interface for communicating with the web app. It has access to runtime DOMs and the JavaScript engine. Additionally, Browser executes the actions in web app states. DOM Analyzer parses the DOM tree and extracts the state and actions associated with it. The current state and actions are converted into an input by DOM Analyzer, which is subsequently sent into the DQN. DQN receives the web app's state and its actions. DQN uses a model of a neural network to figure out Q values for actions which it then sends to Action Selector. The next action to execute is selected by Action Selector based on an Epsilon-Greedy policy [9]. The browser executes the selected action. The web app enters a new state. Observer is responsible for monitoring the status of Browser given that failures can be automatically caught. If a failure occurred, Observer would append the sequence of actions to the failed test set for the tester to examine. Using equation (3), Calculator computes the transition reward and obtains the Q value. The transition is stored in Memory along with the state, action, and Q value. DQN learns from a sampling batch of transitions in Memory to update its weights. DQN would learn poorly if it merely used sequential samples of experience from the environment because of their correlation [10], [11]. This correlation is broken by sampling Memory at random.

4.1. Problem Formulation

To use Deep RL, we must first convert the web app testing problem to the conventional mathematical formalization of RL. The web app testing problem can be formalized formally as a Markov decision process, which can be demonstrated by a 4-tuple, $\langle S, A, P, R \rangle$. These are described below.

S: *States*

Our approach is black-box because it does not access the AUT source code. It only uses the UI of AUT. WeDeep extracts the DOM from the web app's current UI. WeDeep analyzes the DOM to find clickable elements in the current state. A DOM element is clickable if it has a click event listener attached to it, or if it is clickable in general, such as element $\langle a \rangle$. State s_t is represented by $(c_1, c_2, \dots, c_n)$, where c_i are the clickable elements in s_t. Each c_i is an index that indicates the element's position in the DOM tree's pre-order traversal.

Fig. 3 illustrates the partial DOM trees of two pages, DOM 1 and DOM 2, as an example. They are both made up of elements $\langle body \rangle, \langle div \rangle, \langle p \rangle$ and $\langle a \rangle$. The elements that can be clicked (only elements $\langle a \rangle$) are highlighted.

DOM 1's pre-order traversal is $(\langle body \rangle, \langle div \rangle, \langle a \rangle, \langle a \rangle, \langle div \rangle, \langle a \rangle, \langle p \rangle)$, and DOM 2's is $(\langle body \rangle, \langle div \rangle, \langle p \rangle, \langle p \rangle, \langle div \rangle, \langle a \rangle, \langle a \rangle)$. For simplicity, we replace elements that can be clicked with 1 and those that cannot with 0. They are transformed into $(0, 0, 1, 1, 0, 1, 0)$ and $(0, 0, 0, 0, 0, 1, 1)$. To acquire s_1 and s_2, the respective states of DOM 1 and DOM 2, we must take into account the positions of clickable elements, i.e., the positions of number 1. As a result, $s_1 = (2,3,5)$ and $s_2 = (5,6)$.

A: *Actions*

Clickable elements indicate actions. In other words, clickables and click events in web apps are formulated as actions in the Markov decision process. Actions are represented by the index of the clickables in the relevant state, which is similar to states. In s_1, for example, there are three actions marked by $A_1 = (2, 3, 5)$. In the same way, $A_2 = (5, 6)$ shows that s_2 has two actions. In this paper, we don't make a difference between actions and events, because they are the same. In web apps, clickables suffice to complete the majority of tasks. We focus on producing action sequences rather than input values, as in previous work [6], [7]. When acting on elements that require input data, random values will be generated in

accordance with W3C guidelines [19]. For instance, when interacting with HTML input elements such as email or text that require user input data, a random email or text is generated for them. WeDeep allows you to manually enter data for certain inputs (e.g., username and password).

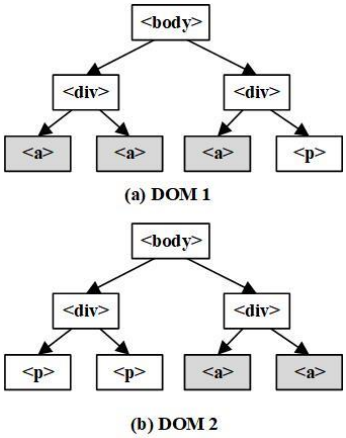

Fig. 3: Partial DOM trees

P: *Transition Function*
The transition function indicates the state the web app will enter once an event occurs. We have no control over it; the AUT decides what it is.

R: *Reward*
When WeDeep executes an event, it receives a reward. We present a mechanism for determining the reward that complements our testing approach. The reward function gives a bigger reward to actions that change the state a lot. This is a heuristic way to understand which actions lead to new functionalities. The intuition is to provide greater rewards for actions that can result in multiple new actions. In fact, a state with more new clickables is more likely to result in additional states and functionalities. It is important to note that our thinking about reward function is similar to that used in [20], a method for testing Java desktop apps that is based on RL. The reward function is defined by the following equation:

$$r_t = R(s_t, a_t, s_{t+1}) = \frac{|s_{t+1} - s_t|}{|s_{t+1}|} \quad (4)$$

The reward function, given two states, s_t and s_{t+1}, estimates the degree of change from s_t to s_{t+1} by comparing and detecting the number of clickables in s_{t+1} that were not present in s_t, which is described as $|s_{t+1} - s_t|$. The ratio $\frac{|s_{t+1} - s_t|}{|s_{t+1}|}$, where $|s_{t+1}|$ is the number of clickables in s_{t+1}, defines the relative change. This reward function takes into account the actions that are introduced in s_{t+1} but are absent in s_t.

As an illustration, given $s_1 = (2,3,5)$ and $s_2 = (5,6)$, as previously defined, $\frac{|(6)|}{|(5,6)|} = \frac{1}{2} = 0.5$ is the reward of the transition from s_1 to s_2. In fact, the clickable (6) is not in s_1, but in s_2, and there are only two clickables in s_2: $(5, 6)$.

4.2. Algorithm

Algorithm 1 details the WeDeep approach for Deep RL-based testing. It takes the AUT, the testing time budget, and the maximum number of actions per episode as input. In fact, we need to turn the testing problem into an RL task that is broken up into several episodes. A series of actions is referred to as an episode. In other words, each episode consists of multiple steps or iterations in which an action is conducted. WeDeep outputs a list of failed test cases F. A memory is used to store samples from previous iterations, each of which comprises the state, action, and related Q value. WeDeep begins by initializing memory M (line 1) and the failed test set F (line 2). Now, testing starts and goes on until the time limit is met (lines 3–23). WeDeep restarts the web app and navigates to the homepage (line 4). Line 5 returns the AUT's initial state. Line 6 initializes test case TC. In each episode, we limit the number of steps that a test case can take (lines 7–23). The default setting for the episode length in WeDeep is 25, but it can be changed. Each test case starts with an action in the initial state.

Algorithm 1 WeDeep for Deep RL based testing
Require: App under test AUT, the time budget for testing, the length of each episode
Ensure: The set of failed test cases F
1: $M \leftarrow \emptyset$
2: $F \leftarrow \emptyset$
3: **while** time budget not completed **do**
4: reset(AUT)
5: $s_t \leftarrow$ getState(AUT)
6: $TC \leftarrow \emptyset$
7: **while** the episode not completed **do**
8: **if** A random number $\in [0, 1] < \epsilon$ **then**
9: $a_t \leftarrow$ getRandomAction(s_t)
10: **else**
11: $a_t \leftarrow$ getBestAction(s_t, DQN)
12: execute(a_t, AUT)
13: **if** isFailed(AUT) **then**
14: $F \leftarrow F \cup \{TC\}$
15: **break**
16: $s_{t+1} \leftarrow$ getState(AUT)
17: $r_t \leftarrow$ getReward(s_t, s_{t+1})
18: $Q(s_t, a_t) = r_t + \gamma \times \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$ where $\gamma = 0.9$
19: $batch \leftarrow$ getMiniBatch(M) $\cup \{(s_t, a_t, Q(s_t, a_t))\}$
20: $DQN \leftarrow$ updateModel($batch, DQN$)
21: M $\leftarrow$ M $\cup \{(s_t, a_t, Q(s_t, a_t))\}$
22: $TC \leftarrow TC \cup \{a_t\}$
23: $s_t \leftarrow s_{t+1}$
return F

Epsilon-Greedy is the policy that WeDeep employs (lines 8–11). It decides based on a predefined threshold ε in the interval $[0, 1]$ to determine whether it will explore new actions or exploit its existing knowledge. In fact, it chooses the action with the highest Q value based on the DQN (exploitation, Line 11) with a chance of $1 - \varepsilon$ and a random action (exploration, Line 9) with a chance of ε. Randomness is required for an agent to discover the optimal strategy [21]. We want the WeDeep to explore as various states as possible at the start of the testing in order to explore new actions more; thus, a high value of ε should be used. WeDeep is therefore expected to follow the Q values in order to exploit its knowledge, so a smaller value of ε is expected. By default, WeDeep starts with 1 to enable maximum exploration, then decreases its value uniformly during the first 30 episodes until a final minimum value of 0.2 transforming its behavior towards exploitation.

WeDeep executes the selected action (line 12). In fact, WeDeep does on-the-fly testing by executing the appropriate action in the current state of AUT. Following the execution of the action, WeDeep monitors the browser's status so that failures can be automatically captured (lines 13-15). If a failure is found (line 13), the test case is added to the set of failed tests (line 14). WeDeep puts an end to the episode and begins a new test case (line 15). It is noteworthy that future iterations of WeDeep will avoid performing the actions that resulted in failures, as this will allow WeDeep to avoid finding failures discovered previously.

WeDeep retrieves the new state (line 16) and computes the reward (line 17) using equation (4) based on s_t, s_{t+1} when the action has been performed without failure. WeDeep uses equation (3) to calculate the Q value of action a_t in s_t with parameters s_t, s_{t+1}, a_t, and r_t (line 18). The discount factor, γ, balances how important the immediate reward is compared to future actions, and a number of 0.9 maximizes the reward earned over the whole episode, not just the immediate reward.

WeDeep employs a set of random training samples, including both the current sample and historical samples (line 19), to train the neural network (line 20). Each sample takes (s_t, a_t) as input and has $Q(s_t, a_t)$ as output. The current transition is saved in memory M, which stores historical samples from previous iterations (line 21). The action is added to the current test case (line 22). The prior state is then updated to continue testing (line 23).

5. Evaluation

In this section, we describe empirical evaluation to assess the effectiveness of our Deep RL approach for web app testing. We specifically aim to find answers to the following research questions:

- RQ1: How does WeDeep compare with state-of-the-art web testing approaches in terms of code coverage? (Code Coverage)
- RQ2: How does WeDeep compare with state-of-the-art web testing approaches in terms of fault detection? (Fault Detection)

5.1. Setup

WeDeep was implemented in Python using WebExplor [7] to assess its effectiveness. WebExplor supports the RL approach. We changed the RL strategy by replacing it with our Deep RL algorithm. To interact with the web app, Selenium [22] was utilized. The DQN was built and executed using Keras [23]. DQN uses a 3-layer fully connected neural network, and Adam to optimize the model with a learning rate of 0.001. Two state-of-the-art strategies were chosen for a comparison study. These include WebExplor [7] and DIG [6]. It is worth noting that QExplore [18] focuses on constructing behavioral models and uses tabular RL as a foundation. Our research focuses on testing using Deep RL, which has never been done previously. Despite having some similarities with WebExplor, QExplor is not a test case generator. So, it is not possible to compare WeDeep directly to QExplore. Extending our Deep RL technique to develop behavioral models for web apps and comparing its effectiveness with that of QExplore is an interesting piece of future work that can be done.

Seven open-source web apps were chosen for our evaluation. These web apps belong to multiple categories and serve various functions. Attendance [24] is a web app for tracking student attendance. Patient Record [25] is a web app that allows you to manage patient records. Bus Booking [26] is a web app for making bus reservations. Addressbook [27] is a web app for managing addresses and phone numbers. Timeclock [28] is a web app for managing timeclocks. Claroline [14] is a web app for managing collaborative e-learning. DimeShift [29] is an expense tracking web app.

Code coverage was employed to assess testing quality because it has been found to be a good predictor of test suite quality [30]. Code coverage is a testing metric that calculates the percentage of code lines that are successfully executed during testing. Similar to [6], [7], each web app was instrumented to acquire code coverage. We insert mutations into web apps to simulate faults and test each approach's ability to recognize the mutation in order to determine how well it can detect faults. Mutation analysis is a technique in which faults (mutants) are introduced into an app to evaluate the effectiveness of its test suite [31]. It has proven to be an excellent way for evaluating the test suites of the apps [31]. The usefulness of mutation analysis for empirical evaluations in software testing has been demonstrated in numerous studies [32], [33].

Using mutant generators [34], [35], we generated 10 faulty versions of each web app at random. We also manually checked if faulty versions of the apps would throw an exception because of the seeded faults. Similar to [6], [7], we studied the ability to detect faults by keeping track of the number of exceptions and errors that the browser reported during tests. Similar to previous research [6], [7], we state that a fault is detected if an exception is generated, and our manual examination verifies that the exception is detecting the seeded faults.

Each approach was tested on each subject web app. We gave each strategy the same 90-minute time limit. In addition, we repeated each experiment three times and calculated the average of all the results to confirm the general trend. The experiments were conducted on a PC running Windows 10, with a processor of an Intel Core i7-13700K 3.40 GHz and memory RAM 31,7 GB.

5.2. Results

In this section, we present the outcomes of our assessment and the answers to our research questions.

RQ1: Code Coverage

Table I displays the average code coverage of three test runs on seven open source web apps using three approaches: WeDeep, WebExplor, and DIG. For each web app, the highest average code coverage over three runs is shaded in gray. WeDeep obtains 68.57% code coverage on average, which is greater than WebExplorer (60.44%) and DIG (51.56%). In other words, when compared to WebExplor and DIG, WeDeep resulted in an average improvement of 13% and 33%, respectively. Furthermore, WeDeep has the highest code coverage in all the web apps compared with the other approaches. This experiment demonstrates that a Deep RL strategy can direct testing toward more effective actions, resulting in

higher code coverage. This suggests that the Deep RL approach is more valuable than the RL approach for testing web apps.

Table I. Code coverage results.

App	WeDeep	WebExplor	DIG
Attendance	68.97	59.15	51.68
Patient Record	78.85	71.34	60.86
Bus Booking	64.16	54.79	44.96
Addressbook	70.19	64.35	56.48
Timeclock	66.71	60.49	52.81
Claroline	71.10	61.57	51.74
DimeShift	60.01	51.41	42.43
Average	68.57	60.44	51.56

Figure 4 depicts the evolution of code coverage for each technique over a 90-minute execution time across all subject web apps. All approaches increase their code coverage almost to the end, but WeDeep comes out on top with the highest code coverage across execution duration. This demonstrates that our Deep RL method is effective in gradually learning a testing strategy that can be utilized at a later stage of the testing process.

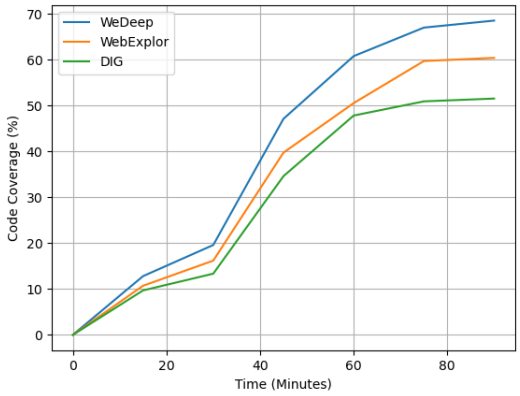

Fig. 4: The comparison of the evolution of code coverage

RQ2: Fault Detection

The average number of faults detected by the three testing approaches in three test runs is shown in Table II. For each app, the highest average number of detected faults over three runs is shaded in gray. In three test runs, WeDeep detected 9.29 faults, WebExplor detected 8.1 faults, and DIG detected 7.29 faults on average. Furthermore, WeDeep reveals the greatest number of faults in all the web apps compared with the other approaches. The ability to detect faults is a crucial part of testing. Deep RL surpassed RL, suggesting that Deep RL is capable of developing and executing test cases to exercise the app's functionalities and assist in fault detection.

We looked over the testing done using WeDeep and the other two approaches, including WebExplor. It is worth noting that there are patterns, i.e., a series of actions that must be performed in a precise order to enable the transition from one state to the next. Because potential faults might be identified by tracing these sequences of actions, they make testing more difficult. Take the Claroline subject as an example. A fault found by WeDeep has an execution trace that shows the following steps: typing the username, typing the password, clicking the login button, clicking on the target course, clicking on the announcement link, clicking the add announcement button, typing the title, typing the description, and clicking the OK button. In another case, WeDeep found a fault in DimeShift after the following steps were taken: typing the username, typing the password, clicking "login," clicking "goals," clicking "create new," typing the name, clicking "next," typing the amount, and clicking "save." WeDeep found faults with such action sequences in other subject web apps as well. Notably, any interruption in the process will result in a redirect to another page, reducing the efficacy of the testing. Deep RL guidance enables WeDeep to efficiently execute these sequences of actions in subject web apps. Indeed, we discovered that the Deep RL algorithm outperforms the RL algorithm when it comes to replicating human behaviors in order to generate these sequences of actions without being distracted by previously seen states or ineffective actions in high dimensional action or state space, and in order to learn an effective testing strategy. Producing such behaviors is feasible due to the learning capabilities of the DQN utilized by the Deep RL algorithm, whereas it is more difficult for the RL algorithm, which has limited adaptation capabilities, to reproduce the correct action sequences.

Table II. Fault detection results.

App	WeDeep	WebExplor	DIG
Attendance	9.33	8.67	7.67
Patient Record	9.67	8.33	7.67
Bus Booking	9.67	8.67	8.00
Addressbook	8.67	8.00	7.33
Timeclock	9.33	7.67	6.67
Claroline	9.33	8.00	6.33
DimeShift	9.00	7.33	7.33
Average	9.29	8.10	7.29

Summary

WeDeep's good results in code coverage and fault detection are due to the Deep RL approach and its reward function, which encourage the execution of actions that lead to new states and allow access to the majority of an AUT's functionalities. Therefore, in the presence of complex state spaces in web apps (with a large number of actions and states) that require the ability to utilize knowledge acquired through previous trial and error, the learning capabilities of Deep RL are more advantageous to web app testing than RL.

6. Conclusions

Automated web app testing involves simulating user actions. Testing web apps is hard because there are so many actions and states that could be taken. In this paper, we present WeDeep, an approach based on Deep RL for web app testing. A Deep Q-network agent is used in this strategy to test the web app systematically through trial and error, optimizing action selection and obtaining the best action to achieve a greater reward for discovering the functionalities. The empirical evaluation yielded encouraging results, as WeDeep achieved greater code coverage than state-of-the-art techniques across seven

subject web apps and also performed well in detecting faults.

This is the first study to successfully integrate web app testing and Deep RL. However, it does not encompass the entire field. There is a significant amount of space for additional work in the future. We intend to test our approach with different parameters (for example, different discount factors and Epsilons) across a larger set of web apps, extend the state space (for example, by adding more information to Deep Q-network), and conduct a detailed investigation into the reward function (for example, by taking action frequency into account).

7. References

[1] "World Internet Users Statistics and 2023 World Population Stats." https://www.internetworldstats.com/stats.htm (accessed Feb. 03, 2023).

[2] "What is the Document Object Model?" https://www.w3.org/TR/WD-DOM/introduction.html (accessed Jan. 05, 2023).

[3] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes," *ACM Trans. Web*, vol. 6, no. 1, Mar. 2012, doi: 10.1145/2109205.2109208.

[4] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-Based Automatic Testing of Modern Web Applications," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 35–53, Jan. 2012, doi: 10.1109/TSE.2011.28.

[5] M. Biagiola, F. Ricca, and P. Tonella, "Search Based Path and Input Data Generation for Web Application Testing," in *Search Based Software Engineering*, T. Menzies and J. Petke, Eds., Cham: Springer International Publishing, 2017, pp. 18–32.

[6] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "Diversity-based Web Test Generation," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, in ESEC/FSE 2019. New York, NY, USA: ACM, 2019, pp. 142–153. doi: 10.1145/3338906.3338970.

[7] Y. Zheng *et al.*, "Automatic Web Testing Using Curiosity-Driven Reinforcement Learning," in *Proceedings of the 43rd International Conference on Software Engineering*, in ICSE '21. IEEE Press, 2021, pp. 423–435. doi: 10.1109/ICSE43902.2021.00048.

[8] A. van Deursen, A. Mesbah, and A. Nederlof, "Crawl-based analysis of web applications: Prospects and challenges," *Sci. Comput. Program.*, vol. 97, pp. 173–180, 2015, doi: https://doi.org/10.1016/j.scico.2014.09.005.

[9] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[10] V. Mnih *et al.*, "Playing Atari with Deep Reinforcement Learning," *CoRR*, vol. abs/1312.5, 2013, [Online]. Available: http://arxiv.org/abs/1312.5602

[11] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep Reinforcement Learning: A Brief Survey," *IEEE Signal Process. Mag.*, vol. 34, no. 6, pp. 26–38, Nov. 2017, doi: 10.1109/MSP.2017.2743240.

[12] مریم عسگری عراقی، وحید رافع و اکرم کلائی، «تولید مورد آزمون مبتنی بر مدل از توصیفات تبدیل گراف با استفاده از الگوریتم جستجوی پرتو»، *مجله مهندسی برق دانشگاه تبریز، جلد ۴۹، شماره ۱، صفحات ۳۵۶-۳۴۳، ۱۳۹۸.*

[13] مجتبی وحیدی اصل، محمدرضا دهقانی تفتی و علیرضا خلیلیان، «رویکردی جدید مبتنی‌بر سنجه‌های نرم‌افزاری جهت افزایش سودمندی آزمون بازگشت»، *مجله مهندسی برق دانشگاه تبریز، جلد ۵۰، شماره ۱، صفحات ۴۷۶-۴۶۳، ۱۳۹۹.*

[14] "Claroline." https://sourceforge.net/projects/claroline/ (accessed Sep. 15, 2022).

[15] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, no. 3, pp. 279–292, 1992, doi: 10.1007/BF00992698.

[16] R. Bellman, "On the Theory of Dynamic Programming," *Proc. Natl. Acad. Sci.*, vol. 38, no. 8, pp. 716–719, 1952, doi: 10.1073/pnas.38.8.716.

[17] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive Random Testing," in *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, M. J. Maher, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 320–329.

[18] S. Sherin, A. Muqeet, M. U. Khan, and M. Z. Iqbal, "QExplore: An exploration strategy for dynamic web applications using guided search," *J. Syst. Softw.*, p. 111512, 2022, doi: https://doi.org/10.1016/j.jss.2022.111512.

[19] "The Input element - HTML." https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input (accessed Feb. 12, 2023).

[20] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "Automatic Testing of GUI-Based Applications," *Softw. Test. Verif. Reliab.*, vol. 24, no. 5, pp. 341–366, Aug. 2014, doi: 10.1002/stvr.1538.

[21] A. D. Tijsma, M. M. Drugan, and M. A. Wiering, "Comparing exploration strategies for Q-learning in random stochastic mazes," in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, Dec. 2016, pp. 1–8. doi: 10.1109/SSCI.2016.7849366.

[22] "Selenium." https://www.selenium.dev/ (accessed Jan. 11, 2022).

[23] "Keras: Deep Learning for humans." https://keras.io/ (accessed Feb. 28, 2022).

[24] "Attendance Management System." https://code-projects.org/attendance-management-system-using-php-source-code/ (accessed Sep. 22, 2022).

[25] "Patient Record Management System." https://code-projects.org/patient-record-management-system-in-php-with-source-code/ (accessed Sep. 22, 2022).

[26] "Bus Booking System." https://code-projects.org/bus-booking-system-in-php-with-source-code/ (accessed Sep. 23, 2022).

[27] "Addressbook." https://sourceforge.net/projects/php-addressbook/ (accessed Sep. 15, 2022).
[28] "Timeclock." https://sourceforge.net/projects/timeclock/ (accessed Sep. 16, 2022).
[29] "dimeshift." https://github.com/jeka-kiselyov/dimeshift (accessed Sep. 04, 2022).
[30] R. Gopinath, C. Jensen, and A. Groce, "Code Coverage for Suite Evaluation by Developers," in *Proceedings of the 36th International Conference on Software Engineering*, in ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 72–82. doi: 10.1145/2568225.2568278.
[31] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, 2011, doi: 10.1109/TSE.2010.62.
[32] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?," in *Proceedings of the 27th International Conference on Software Engineering*, in ICSE '05. New York, NY, USA: Association for Computing Machinery, 2005, pp. 402–411. doi: 10.1145/1062455.1062530.
[33] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, Aug. 2006, doi: 10.1109/TSE.2006.83.
[34] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Guided Mutation Testing for JavaScript Web Applications," *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 429–444, May 2015, doi: 10.1109/TSE.2014.2371458.
[35] S. Sherin, M. Z. Iqbal, M. U. Khan, and A. A. Jilani, "Comparing coverage criteria for dynamic web application: An empirical evaluation," *Comput. Stand. Interfaces*, p. 103467, 2020, doi: https://doi.org/10.1016/j.csi.2020.103467.