

End-to-end RPA-like testing using reinforcement learning

Ciprian Păduraru

University of Bucharest and
Institute for Logic and Data Science
Bucharest, Romania
ciprian.paduraru@unibuc.ro

Rareș Cristea

Research Institute of the University of Bucharest
and Institute for Logic and Data Science
Bucharest, Romania
rares.cristea@unibuc.ro

Alin Stefanescu

University of Bucharest and
Institute for Logic and Data Science
Bucharest, Romania
alin.stefanescu@unibuc.ro

Abstract—Even though test automation has an increased presence in industry nowadays, there is still room for improvement, especially in the area of end-to-end testing. Most testing methods in the literature focus on techniques that do not test these applications as a typical end user would, i.e., starting from the user interface (UI) level. Our work, done in collaboration with UiPath company, a leader in Robotic Process Automation (RPA), proposes deep reinforcement learning methods that can test applications from end to end at the UI level. In the current implementation of our prototype, abstractions and separation of concerns are considered so that methods can be reused between applications and algorithms can be used with minimal user effort. The testing process that results after training the agents is similar to that of a human tester going through the functions of the application. Empirical evaluation of these agents shows that, on the one hand, they can almost perfectly mimic the behavior of human testers and, on the other hand, they can exceed the human performance level.

Index Terms—reinforcement learning, autonomous, testing, RPA

I. INTRODUCTION

While using an application, end users interact with the user interface (UI) via a sequence of actions. Most of the testing methods at the level of UI in the literature are based on unit testing, functional testing, end-to-end testing, or a combination of these methods. There are also variations where a test solution interacts with the application, either using model-based testing [1], guided fuzzing techniques [2] etc., but usually this is not done at the UI level.

Our current work performs testing of applications where the test process is independent of the application type and starts from the UI level. This type of testing is challenging, however, as some interface errors may only occur under unusual conditions. [3]. Testing a wide range of scenarios is necessary to ensure the quality of the software. However, this approach could lead to a combinatorial explosion of tests (e.g., triggering modal windows, populating input fields with various values, or even altering the front-end code of the web page).

Reinforcement Learning (RL) [4] has proven to be a successful tool for improving testing procedures. On the one hand, it is a useful tool to prioritize test cases to promote tests that are expected to identify vulnerabilities [5]. On the other hand, RL has proven to be a capable tool to create new test cases by learning from tests created either manually or with other

automated tools [6]. A reinforcement learning-guided test system has proven useful in identifying new vulnerabilities, especially in areas for which the developer would not think of creating a test scenario. The next step in RL-based testing at the level of UI is to improve over the widely used Q-learning paradigm through a Deep Q-learning paradigm that bypasses the need for a traditional state-action table.

The main contributions of our paper are as follows:

- We propose three Deep Reinforcement Learning (DRL) based methods for end-to-end testing applications that start at the UI level and simulate the entire testing process of a human user. The first is based on Behavior Cloning (BC) methods [7], which trains the RL agent to mimic the demonstrations of human testers. The second method is based on Deep-Q-Network (DQN) [8] to train test agents without supervised data. Finally, the third method performs transfer learning from BC to DQL to start with a network trained on human behavior to incorporate the real behavior of testers and then enrich it. The dependencies between the software under test (SUT), its UI, and the RL environment are abstracted using interfaces and inversion of control [9].
- Our abstraction method for separating the UI and the application's internal functionalities is based on a generic crawling method that builds a graph (model) of the connections between the application's states. Each node in the graph represents a high-level state, while edges represent the triggers that change a state to another. We also provide an API that allows you to fine-tune the model (or write it from scratch) and add annotations (hints) to improve the testing process.
- In the RL domain, we first model the problem as a Partially-Observed-Markov Decision Process (POMDP) [4]. Our customized environment is derived from the OpenAI Gym interface [10], which provides separation between the test environment and the numerous open-source reinforcement learning algorithms and libraries such as [11]. The environment created also allows developers to customize reward functions, agent observations, and selected actions. These are sometimes needed to align test objectives with specific goals or use cases.

As a result, the developer of a SUT is able to try different techniques with minimal knowledge of the underlying RL domain. To wrap-up, the novelty of our paper is that the problem is modeled as a POMDP and then DRL and behavioral clone agents are trained to perform end-to-end testing of the applications.

The work presented in the paper has a direct connection to industry, since the requirements came from our industry partner UiPath¹. UiPath [12] offers enterprise solutions to automate repetitive tasks at the level of UI, aka Robotic Process Automation (RPA) [13]. UiPath is market leader in RPA, according to independent analysts² and they offers also solutions for testing³. The source code of our prototype implementation will be made available at the publication time.

II. RELATED WORK

According to the literature, testing at UI level is a challenging problem. One of the major problems is to connect to the UI of the applications, understand and abstract the model independently from the application's architecture, and then finally generate functional test cases. Efficient methods for correlating the application model with an instrumentation tool that can cover a large set of UI states is another challenge.

The highest order taxonomy for test solutions is the division into verification and validation. While the former ensures that the software works well, the latter *validates* software requirements. Regression testing is a useful test method for validation. For example, the authors in [14] define a useful classification for GUI regression testing solutions. The similarity between their work and ours is that they perform model extraction at the UI level. However, their testing methods focus on testing only specific components, while our work aims at testing end-to-end applications, without any human supervision.

End-to-end test methods have the advantage of validating multiple components of systems that are connected by real use cases. They also prove more relevant in crash detection for web [15]. However, their main drawback is that when such a test fails, it relies on either a good unit test definition or solid debugging strategies to quickly identify the system component that triggered the failure [16]. In addition, end-to-end tests suffer from both a combinatorial explosion of cases and a long execution time, making it difficult to run them in a daily development cycle or during regression testing and to maintain them in a scalar context [17].

Current testing methods in the literature in this area require either the description of an application model to provide the underlying business model [18] or the source code structure for the tested UI components, especially for web applications [19] [20] and Windows API-enabled applications [21].

Reinforcement learning is proving effective in testing applications in a variety of domains, such as fuzz testing [22].

Regarding testing at the level of UI, progress has been made in using image recognition as a method for a machine to understand a user interface [23]. Recent studies have also focused on improving locators, the elements that allow testing algorithms to understand where to look for the UI widget they need to interact with during testing [20], [24]. Other studies in the same field criticized the ability of automated test generation through empirical comparisons with manual testers [25]. However, the applications used are not representative of an industrial environment and the work does not discuss their efficiency in an agile developed app.

Research into the applications of RL in the context of UI testing has also focused on two complementary categories that are also of interest to our future work. First, it is used to prioritize UI tests. The work in [5] proposes several RL-based methods for figuring out which tests should be run first because they are likely to yield identifiable bugs. Second, RL can be used to generate new test cases by learning from previous manually designed test cases. Research on test generation methods has focused on platform-specific solutions, such as Android [6], [26], and web applications [19].

A work comparable in purpose to ours is *TESTAR* [27], which uses reinforcement learning on dynamically generated GUI models. In comparison, our work proposes improvements in the methods used, especially in the RL part, bypassing the need to build a special state-action table and replacing Q-learning with Deep Q-learning. The advantage of our method in this case is the possibility to encode larger state spaces and to include the historical context of the successive steps. In our view, this is a necessity nowadays, considering the complexity of applications.

III. THE TESTING RL ENVIRONMENT

All the components defined in this section, and later in Section IV, are encapsulated in a framework using as many as possible interfaces that allow customization and inversion of control [9]. We define two categories of users: Developers - the stakeholders that define the functionalities of the application and testing objectives, and End Users - the stakeholders that use the application.

This section describes the architecture of an environment capable of testing end-to-end applications independent of platform or Application-Under-Test's (AUT) architecture. The environment defined is also compatible with OpenAI Gym [10] in order to assure maximum compatibility with existing libraries and tools. The setup definition process is depicted in Fig. 1 and addressed in detail in the rest of this section. Formally, the framework maps the testing problem as a Markov Decision Process (MDP), more precisely, a Partially Observable Markov Decision Process (POMDP) given that the agent sees only a substate of the application state (at least by default) at any timestep [4].

A. Internal application specification

To solve the challenge of having an environment definition that is platform and application independent, at the architec-

¹<https://uiopath.com>

²<https://www.uiopath.com/resources/automation-analyst-reports/gartner-magic-quadrant-robotic-process-automation> and <https://www.uiopath.com/resources/automation-analyst-reports/forrester-wave-rpa>

³<https://www.uiopath.com/platform/operate/continuous-testing>

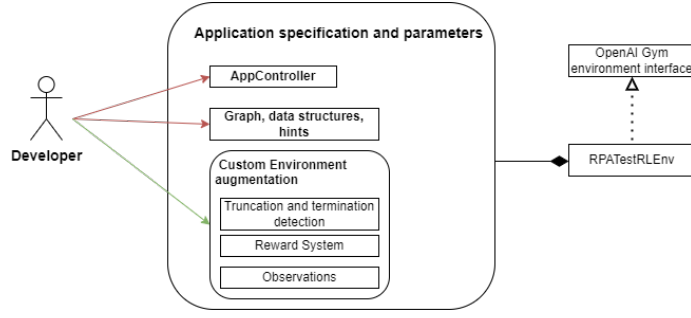


Fig. 1. The process of having the developer inject its knowledge and insights for the application under test. The red arrows (above) represent mandatory parameters, while the green arrow is optional. The *AppController* component is responsible for providing interfaces needed by the environment and agents to navigate through the application autonomously. The graph and data structure that provide the application’s insights can be created automatically by a crawler process provided and customized using our defined API. The environment customization component gives access to the developer to a set of fine details to augment the environment for testing efficiency. These specifications are aggregated inside the testing framework’s *RPAstREnv* component, representing the environment used by the agents to test the application. Note that this is derived and complies with the OpenAI Gym interface.

tural level we adopted a few strategies. Mainly, two mandatory components, and one optional, provided by the developer side need to be injected as parameters to the environment - Fig. 1.

ApplicationOrchestrator object (mandatory). It defines the interaction with the developer’s application. Our framework provides an interface that the developer needs to inherit and implement its own implementation for two functionalities:

- *setInputInController(ctrlId, value)*: sets the input given value at a given *ctrlId*, e.g., setting a text to an editable box, selecting an option from a list or dropdown, etc.
- *clickOnController(ctrlId)*: activates the control flow enabled by the given *ctrlId*, e.g., a click on it, keyboard input, etc.

This component abstracts different types of applications, for example, a video game or a web application. The former can have different kinds of input and processing methods (e.g., virtual reality kits, gestures, etc.), while for the latter the input and processing may involve more common controlling methods such as mice, keyboards, and touchscreens. This object interfaces the input method used by the application and the algorithm’s perception of the input.

Application definition (mandatory). It defines a graph and additional data structures that abstract the states and flows of the application.

Each application developer will provide first a graph of its application using a custom-defined API. This is needed to make the relationship between an application and the abstraction a decoupled environment that the reinforcement agents need to operate on.

This graph can be defined in two ways:

- 1) Manually designed using an API from our repository.
- 2) Automatically generated using a crawler component that we provide as an example which explores exhaustively the application creating possible states, actions, and links between them. The output of the crawler populates the graph.

There is also the possibility to mix the two approaches, either by manually directing the crawling process or by

allowing the crawler to write the draft graph and then enhance the result manually.

Furthermore, the framework offers the possibility of managing different contexts and states of the applications using two grouping concepts:

- *Pages*. In a web-based application, this could represent exactly a web page, while in a video game or simulation application, it could be the HUD (Heads-Up-Display) or a typical menu for choosing settings. Each page has a unique id, i.e., P_{id} . The set of all pages of the application, $AllPages = \{P_0, P_1, P_2, \dots, P_n\}$ are abstracted as nodes of the graph mentioned above.
- *Meta-states*. These represent a logical group of Pages from the applications. Formally, considering each meta-state as having a unique id, we denote the set of pages inside a meta state as $M_{id} = \{P_{id_0}, \dots, P_{id_{n_{id}}}\}$. Note that the same page can appear in different meta-states. At the graph level definition, a meta-state can be seen as a super node that aggregates a certain set of pages.

Concrete examples for a web-based application are the group of pages that are handling the authentication process, or one that describes the loan process in a bank application. Note that these are needed further to adopt the strategies used by Hierarchical and Feudal Reinforcement learning methods [28].

Each page P_{id} , contains a set of controllers $Controllers_{id} = \{C_{id_0}, \dots, C_{id_{n_{id}}}\}$ which are the set of objects supporting the interaction with the user. The framework divides these controllers further into two subsets:

- *Editable controllers*, representing which of the controllers on the page are user-editable, e.g., in a web-based application: checkboxes, text filling boxes, dropdowns, list selections, etc. In this case, the user can provide hints for the algorithm on how to fill these fields: $Hints(C_{id})$. Currently, the framework supports the following specifications as hints (extensible on the developer’s side): a range of possible values, a set of ranges, a set of concrete discrete values, and string patterns. There is

also the option to hint a controller as being mandatory for completion, or even more, completion with a certain pattern (e.g., mail addresses). While optional, if provided, these could boost the testing processes’ performance. One obvious example is providing hints for login controllers, e.g., credentials for a set of users that exists in the application’s database. Then, the agents running automatically will not need to guess combinations of correct usernames and passwords.

Note that all the editable content states, i.e., current values, are stored in a persistent state dictionary (*PersistentState*), such that agents can quickly access the current fillings when making a decision.

- *Flow controllers*, representing controllers that could change the flow and move from the current state (page, node) to a different one. In a web-based application, examples can be buttons or links. Thus, for each active controller at any time, the framework knows the parent page (starting node in the graph) and the link (edge and ending node). These objects help in organizing internal data structures such as the set of all controllers that can move from one page to another, $Links(P_{id_i}, P_{id_j}) = \{C_{ij_0}, \dots\}$, or ones to know from where and how a certain page can be opened, $InLinks(P_{id})$. These are used to inject internal knowledge of the application to the reinforcement learning agents acting on the environment can make the connection between its targets and actions more efficiently.

Environment augmentation

The typical operations and data structures used by reinforcement learning-based environments, i.e., observation, reward systems, and truncation detection mechanisms can be augmented outside our default implementation (Section III-B) using developers’ and application-specific needs. The augmentations are injected using inversion of control, giving developers a full range of customization capabilities of the operations.

For example, a developer can augment the observation space to add its own specific application such as server states, images, and persistent states that the application may use.

A customized reward system may be valuable for training agents who test certain aspects of an application. Concrete scenarios target testing a group of web pages and items, modules in video games (e.g., physics or gameplay systems), and certain APIs (e.g., databases or network management source code). Truncation and termination augmentation are highly used in training the reinforcement learning agents today for speeding up purposes. It helps by cutting the episodes earlier to prevent learning from non-sense trajectories or to restart the episodes sooner rather than later with different conditions. [29].

B. Customized environment implementation

As shown in Fig. 1, the application’s specifics are aggregated by the developer inside a customized OpenAI Gym

compatible environment, named *RPATestRLEnv*. This environment’s default subcomponents are further described. The developer is also able to augment these subcomponents. The interaction in the case of a full-step process taken by the agent in the environment is shown in Fig. 2.

Observations

The observation space contains what an agent would “see”. This is made up in the default implementation by the objects in equation 1, and detailed further in the text.

$$Obs = \{P_{id}, M_{id}, Ctrls_{Obs}, Ctrls_E, Ctrls_F, Ctrls_H\} \quad (1)$$

- P_{id} : the index of the page; represented as one-hot encoding.
- M_{id} : the index of the meta-state in the current state of the application, one-hot encoding. Note that this is needed for the agent in order to understand the global state or user intention since the same page id can be part of multiple meta-states.
- $Ctrls_{Obs}$: List of observable controllers that are available, i.e., not blocked or disabled by the application in the current state. Each entry is represented as a multi-hot encoding in the range $(0, num_max_controllers_per_page)$.
- $Ctrls_E$: the list of editable items, $Ctrls_E \subset Ctrls_{Obs}$. Each controller also contains the list of hints provided, if any.
- $Ctrls_F$: the list of flow items. Each entry contains an id of the next page’s id represented as one-hot encoding in the range $(0, num_max_controllers_per_page)$.
- $Ctrls_H$: the list of previously interacted controller ids, actions, and content stored in a deque of fixed size, 8. If there are less than 8, the invalid filling value is -1 .

This proved to be important in letting know the agent that different filling orders of the controllers result in bad or worse decisions. E.g., consider a web-based application with specific needs of selecting an item in a checkbox control before filling a particular editable control.

In the case that developer provided augmentation with its own set of customized observations, Obs_{dev} , a different set of values can be added to the default values: $Obs = Obs \cup Obs_{dev}$.

Step process

The step function, as described in Fig. 2, takes the agent’s action and executes it in the environment, performing the actions on the AUT through the registered *AppController* component. The result of the step process is the new observation, info object, and termination/truncation feedback.

Actions

The flow of executing an action in the environment is depicted in Fig. 2 with a sequence of 6 steps:

- Step 1.
When selecting an action, the agent first choses a controller to interact with $ctrlId$, then one of the defined actions, in our case:

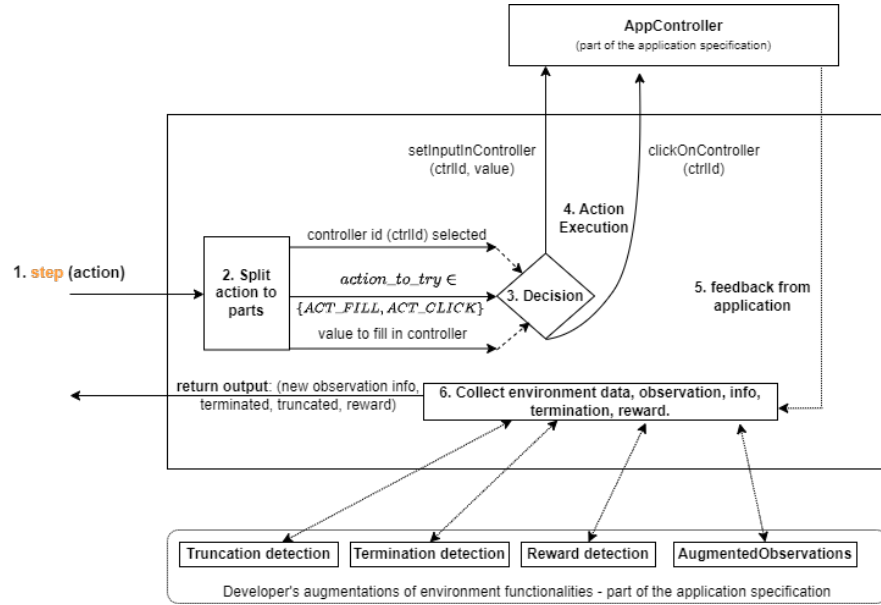


Fig. 2. The full activity flow for taking action in the customized environment and the interaction with components registered by the developer for augmentation and application control. The text in Section III-A (*Actions*) describes the steps in detail.

- *ACT_FILL*: the agent attempts to fill a value in an editable controller selected. In this case, the agent also selects the concrete value to fill in C_{ctrlId} , where it takes into account the hints given by the developer, if any.
- *ACT_CLICK*: if the item is valid, the agent will follow the transition to the next page represented by a state in the graph.

The action type, value to fill, and controller id are sent further in the form of a dictionary of values.

- Step 2 and 3.
This step takes the request described in the previous step and performs internal postprocessing depending on action types and concrete values selected.
- Step 4.
It performs the actions requested in the application under test using the object registered by the developer as the controller, i.e., *AppController*. As explained in III-A, depending on the selected action type, one of the two functions from the interface (*setInputInController* or *clickOnController*) is called and the control is inverted to the developer's implementation.
- Step 5.
After the actions are executed, the application provides feedback with details such as errors, logs, messages from the applications, the new *pageId* (node in the graph) if the transition was successful, and any violated constraints.
- Step 6.
Using the output from step 5, we build the final object used in the continuation of the RL feedback loop.
To explain this further, please note that it is also possible

to select an invalid controller for the interaction or an action type, e.g., if you try to enter a value into a controller of type button, fill in or use a disabled (not active in this state) controller, select an *ACT_CLICK* if the controller has selected an editable type instead of a flow item. In this case, the application feedback is returned (step 5 in Fig. 2), and the detailed information about invalid selections is described in an internal object *RestrictionsViolation*. This data structure is sent back to the reward system, where the developer can make further decisions, such as imposing a penalty on the agent. By default, our implementation of the reward system issues penalties to the agent with a constant factor.

Info object

In the OpenAI Gym interfaces, the *reset* and *step* functions also return an *info* object containing all other hidden (or not) feedback. In our case, this was adapted to send some other interesting values for the testing process that can either help with performance evaluation or debugging:

- Error messages, logs, and captured images.
- A *RestrictionsViolation* object: The violated restrictions after the last action taken (see the text above that describes the *Actions*).

Reset function

The reset function of *RPATestRLEnv* sets the environment to a default state, i.e., the initial page or a random sample from the initial pages/groups assigned by the developer in the application specification graph.

IV. AGENTS

In reinforcement learning methods, an agent's main task is to explore the environment and train the underlying decision

processes to obtain higher rewards over time. During the exploration period, the agent starts in an initial state defined with the function *reset* as mentioned in section III-B and creates *trajectories*, τ (2) represented as tuples of *states*, *actions* and *rewards* obtained after each decision. The states are the current observations, while the actions taken are from the decision process. This process is generally encapsulated in terms of a *policy*, where the actions are taken from a probability distribution conditioned on the current state: $a_t \sim \pi(a|s_t)$.

$$\tau = (s_0, a_0, r_1, s_1, a_1, r_2, s_2, \dots, s_T) \quad (2)$$

The testing agents in the framework all have the same interface and share the technical debt for logging, and metrics/evaluation functionalities. The different methods used behind by each agent are customized using different policies. There are 4 main policies implemented now, briefly mentioned below:

- **RandomPolicy**: makes a random decision at each time step.
- **HumanAgent**: used to record demonstrations of human experts testing the application.
- **BCPolicy**: the Behavioral Cloning Policy [7], which mimics the behavior of human testers recorded with the HumanAgent Policy.
- **DQNPolicy**: implements the Deep Q-Network [8], with capabilities to reuse BCPolicy knowledge.

Note that in our framework, the notions of *trajectory* and *episode* are interchangeable. This is common in practice, although in literature formally a trajectory could be only parts of an episode.

A. RandomPolicy

The decision-making process of this policy is just a uniform random selection from the set of possible actions in each state. It first chooses a random controller id to interact with and a random action, a value, to fill in. This is used to collect a small set of data that is used to debug the methods and verify that they are valid. In practice, however, it might be useful to switch between one of the other agents and this one to simulate abnormal user behavior as well.

B. HumanAgent

This agent type takes real expert demonstrations that come from human testers. It has no internal decision process. The human tester specifies the actions at each time step, and the sequence of states and actions is recorded for later reuse. However, rewards are given by the internal components, so demonstrations can be given a final score indicating how useful they are for a particular test objective. It is also possible to replace the human in this scenario with other automation software that provides user scenarios. For example, if the application testing team already has scenarios that they can run with RPA or other testing tools, they can automatically be used as test oracles as long as the actions can be recorded.

The output of each agent is a recorded dataset of demonstrations: $\mathcal{D}_i = \{\tau_0, \tau_1, \dots, \tau_N\}$, where each entry is a recorded trajectory from the user, Eq. (2). The complete dataset consists of the union of all such demonstrations $\mathcal{D} = \{\mathcal{D}_i\}_M$.

C. BCPolicy

The goal of Behavioral Cloning Policies is to mimic the experts' demonstrations as closely as possible [7]. In our case, the data used to train the policy decision process comes from the dataset \mathcal{D} captured by *HumanAgent*. The current model architecture is shown in Fig. 3. The input to the network contains the observations that the agents see in the current time step. These are composed of the default observations and the additional observations registered by the developer, if any. This data is smoothed, converted to floating point numbers, and sent to a sequence of multi-perceptron layers (MLP) [30]. The size of the MLP in-depth and the number of neurons on each layer can also be adjusted by the developer, depending on the expected results (section V describes the default values used by the framework and the challenges associated with this aspect).

The output of the network has a fixed size of $(Num_{Actions} \times Num_{Ctrls})$ neurons, where $Num_{Actions}$ represents the number of possible actions to choose from (fixed to two in the current framework, i.e., fill or click), while Num_{Ctrls} represents the maximum number of controllers in the current state *Page* of the application. This array of raw floating point numbers is further divided into two components:

- **Action type selection distribution parameters** (the first two floats): The mean and log standard deviation for a bivariate Gaussian distribution. Thus, at the time of inference, an action type is drawn from this distribution, i.e., $Act_{type} \sim \mathcal{N}(\mu_{act}, \log \sigma_{act})$.
- **Controller index to interact with and apply the action selected above** (the remaining Num_{Ctrls} float values). This time they represent the mean and log standard deviation for a multivariate Gaussian, as they represent the probability of selecting each of the possible controllers in the current state. Thus, at the time of inference, an action type is drawn from this distribution, i.e., $C_{id} \sim \mathcal{N}(\mu_{ctrl}, \log \sigma_{ctrl})$.

If the action type selected is of type *ACT_FILL*, the agent will try first to select a value sampled from the hints specified by the developer, i.e., $value \sim Hints(C_{id})$.

There are three main use cases for the proposed network, as shown in Fig. 3. The first is the inference process, which at each time step provides the observations returned by the environment and determines the actions and values to be executed (see the *Actions* and *Step* mechanisms described in Section III). This is reused by the other two use cases:

- The sampling processes to obtain new test data and create a corpus in the form of the dataset \mathcal{D} , similar to the one created by *HumanAgent*. In this case, the actions are performed according to the given trained policy.
- Training process to improve the parameters of the network (MLP and connection layers), performing back-

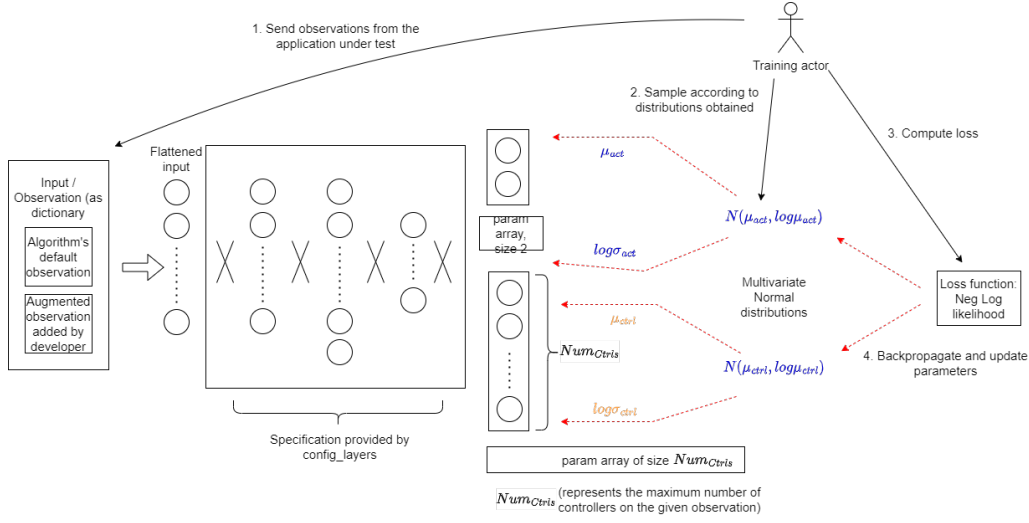


Fig. 3. The architecture of the DQN agent, use cases for sampling, adding test new test to the corpus, and training through backpropagation. More details in Section IV-C.

propagation and using the negative log-likelihood loss (NLL) for the probabilities of the actions performed by the human experts in the same states. This is used for both output heads, i.e., for selecting the action type and the controller to interact within a state. Intuitively, the loss is high if the current policy does not perform actions similar to those in the demonstrations \mathcal{D} , and approaches 0 if it matches almost perfectly in the same states.

Humans can also be part of the annotation and improve the model using the *dagger* method [31] by human-in-the-loop. In short, this method works by having the agent ask the human expert in certain steps what action should be taken in that state. Then, the policy is enforced to choose the same action. According to the literature, this can help significantly in cases where there are states that have not yet been visited in the \mathcal{D} dataset provided.

D. DQNPolicy

A DQN algorithm class attempts to train a predictor of how good a state and action are, $\hat{Q}(\text{state}, \text{action})$, using a deep neural network architecture. Our proposed architecture is shown in Fig. 4. In the first part of the network, the input and the MLPs used are similar to the previous ones in BCPolicy and explained above. However, the raw output is different this time. It represents a Cartesian product between the action type and the controller index to interact with. The size of the output layer is $(\text{NumActions} \times \text{NumCtrls})$ neurons. Thus, the output corresponding to a pair $(\text{Act}_{\text{type}}, \text{C}_{\text{id}})$ indicates the value that occurs when, in a current state, this particular action and this particular controller are chosen for interaction, i.e., $Q(\text{state}, (\text{Act}_{\text{type}}, \text{C}_{\text{id}}))$.

During training, tuples of states, actions, next states, and rewards are collected: $\mathcal{D}\{(s_i, a_i, s_{i+1}, r_i)\}$. This is used as the ground truth for outcome estimation $Q(\text{state}, \text{action})$. The goal is then to explore the environment, collect as many tuples

as possible and train the neural network-based predictor by backpropagation to minimize the difference between \hat{Q} and Q . We chose the Huber loss [30] as the loss function for this difference, which provided better stability against outliers in our evaluation.

As suggested in the literature, the *exploration vs exploitation* plays an important role in training this type of strategy. Briefly, the *exploration* means making a random decision (like the *RandomAgent*), while the *exploitation* uses the current network to make the decision. Intuitively, this is mainly used to discover new possible states, rather than just going through common paths from the start. As suggested in section V, we have found that it is important for the testing process to maintain a high exploration rate even in the final epochs of training.

Since the DQNPolicy and the BCPolicy share an important part of the architecture of a deep network, we evaluated and proved that transfer learning can help in this situation. This is also proposed in the literature [32]. The idea is that instead of starting *cold*, with an estimator network initialized with random values, we perform a *hot* start where the parameters are imported from the already trained BCPolicy. Intuitively, this transfer of parameters should also integrate human knowledge into the network, and the continuation of the training process should attempt to go beyond human understanding. The implementation of the framework combines the improvements of several methods from the existing DQN literature as summarized in this paper [8].

V. EVALUATION

The aim of our evaluation is to respond to the following three research questions:

- **RQ1:** Can the proposed agents be trained and evaluated to generate reasonable corpora of test cases considering the short time constraints of the agile development methodologies?

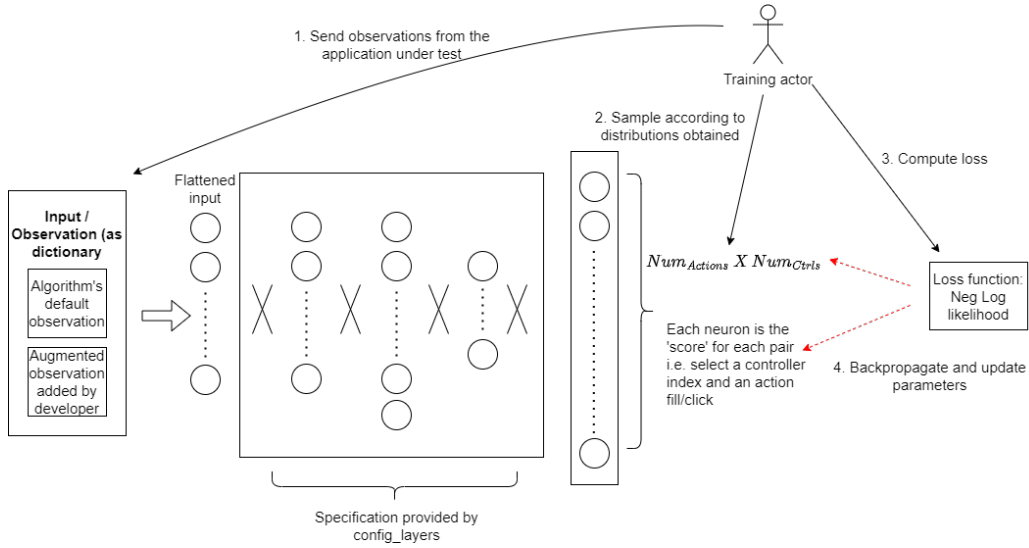


Fig. 4. The architecture of the DQN agent, use cases for sampling, adding test new test to the corpus, and training through backpropagation. More details in Section IV-D.

- **RQ2:** How efficient are human testers versus agents in generating test samples, in both numbers and their effectiveness?
- **RQ3:** What is the performance of individual agents ignoring the short time constraints?

Together with our partners at UiPath⁴, we applied the proposed methods and our prototype to the UiBank application⁵ (open to the public for testing) that emulates a banking application. The main application functionalities were credential management, loans, and other bank account operations. At the moment of writing, this is the only evaluated application that we can disclose, but we think that its complexity allows a good evaluation process that can be applied to many other applications. The scenarios that make up the expert demo datasets were recorded by various human testers using *Human Agent* in the background (cf. Section IV) and collected about 120 records for the initial training of the BC method. To identify all pages and interactive widgets in the web application, no manual tuning was required via the provided API.

Real and injected bugs. First, we note that RL agents were able to find two concrete problems that human testers had difficulty finding (or could not find without help): (a) the user can transfer money to and from the same account, inflating the total value of the account and (b) the user can view the details of a loan without being authenticated as the owner of the loan.

Aside from these problems, we manually inserted into the mock application built on top of UiBank other 8 demonstrative hard-to-find problems that could only occur due to abnormal (but possible) user behavior (e.g., filling in controllers on a particular order and then pressing Next or Back). Human

testers would be hard-pressed to find these situations in a short time, given the agile development environment.

A. Parameters and evaluation setup

Models architecture and hyperparameters The standard architecture of the MLP (Fig. 3 and 4) has three layers: 64, 128, and 64 neurons, respectively. For training, the Adam optimizer [33] was preferred due to its known effectiveness on RL problems, with stacks of 128 examples each. The prioritized version [34] with a maximum size of 10000 examples was used to implement the experience replay buffer. The preferred method for exploration during training is the temporally-extended ϵ -greedy exploration [35], starting with a value of $\epsilon_{start} = 0.9$, ending with $\epsilon_{end} = 0.05$ with a linear decay rate during these 20 steps. The discount factor used for the online policy network during training is $\Gamma = 0.99$. To update the target network weight from the online network, use the t-Soft update [36] with a $\tau = 0.005$. When building test corpora based on pre-trained agents, the default exploration rate is set to 0.9 (can be overridden by developers) to encourage small path deviations along the test plan. This prevents test duplication and better models a human model that could execute steps in a different order at some time steps, even in the same application.

Episodes configuration. Agents had 20 allowed steps per episode (customizable by developers) in both testing and training. A step means you see the observation and make a decision (fill in an editable checkbox or click a link/button). For example, on the login page, three steps are required: filling in the username and password, then clicking the *login* button.

The default termination system is configured to end the episode when a fatal assertion/error occurs or after 20 increments. However, with the ability to include developers' own systems, smarter systems can be introduced. For example, our

⁴<https://uipath.com>

⁵<https://uibank.uipath.com>

code repository has another implementation of a customized system that checks if the agent is just cycling between the same pair of pages (e.g., login and logout) and terminates/cancels the episode after a few cycles.

Rewards. By default, the reward system (the mechanism that gives scoring to agents per timestep; customizable by developers) has the following rules:

- +1 for each new page visited in the current episode.
- +2 for each of the 10 hard-to-find issues discovered.
- −0.5 for an actionable item violation (e.g., clicking on a disabled button).
- −0.2 for an editable control violation (same as above, but this time for an edit-like control).

B. Models under evaluation.

In our evaluations, we consider automatic methods for testing the application under agile development conditions. As for time constraints: testing must be performed in short iterations, usually during the day after each source code commit, at frequent fixed dates during a development day, and at night. It is also important to know that changes occur frequently during each iteration and even during a single day.

Four agent models are evaluated, as follows:

- The Behavioral Cloning (*BC*) model as described in Section IV-C.
- The Deep-Q-Network agent (*DQN*), Section IV-D.
- The *DQN* as above but this time reusing the pre-trained *BC* model using transfer learning (*DQNtf*), using those 120 provided expert demonstrations, Section IV-D.
- Same model as above, but this time the *BC* model was trained on a model of the application, while the *DQNtf* above was trained on an application with 25% modified content. We obtain this percentage by modifying the model graph by adding/subtracting nodes and or-edges. We refer to this model as *DQNtfMod*. The motivation behind this evaluation is that in most cases it is very difficult to collect behaviour data from human testers in a short time. Therefore, in agile development methodologies, it is useful to evaluate how efficient it would be to transfer the learning from an older application implementation to a newer one (the 25% number was determined empirically because it is very difficult, in terms of computational resources and time constraints, to evaluate a full graph of efficiency versus percent change in practice).

C. Metrics evaluations.

According to our tested application and its model graph, the maximum theoretical rewards an agent could receive are as follows: $MaxReward_{theor} = (10 \times 2 - the\ hard - to - detect\ issues) + 17 (number\ of\ pages\ in\ the\ app) = 37$. However, since agents are allowed 20 steps in each test episode, the practical maximum reward $MaxReward_{pract}$ is 27 per episode.

The first research question is whether agents can be trained given the time constraints imposed by agile devel-

TABLE I
RQ1. WALL TIME IN SECONDS TO TRAIN AGENTS CAPABLE OF REACHING THE SET THRESHOLD OF REWARD 16 (OUT OF 27, THE MAXIMUM VALUE IN PRACTICE).

Agent	Time (in sec and corresponding hours)
<i>BC</i>	738s $\sim 0.2h$
<i>DQN</i>	19894s $\sim 5.5h$
<i>DQNtf</i>	7188s $\sim 2h$
<i>DQNtfMod</i>	12960s $\sim 3.6h$

TABLE II
RQ2. COMPARISON BETWEEN A HUMAN AND AN RL AGENT. AVERAGE STATISTICS TIME OF HOW MUCH A TEST (~ 20 STEPS) TAKES FOR EACH, AND HOW MANY TESTS CAN BE RUN IN A 1H WINDOW.

Agent Type	Avg time per single test	Avg no. of tests per hour
Human	50s	72/h
RL-based	8.37s	430/h

opment. Table I shows the training time to achieve an average training performance exceeding a fixed threshold of $MaxReward_{target} = 16$ for each of the last 50 evaluation episodes. The thresholds were set empirically, based on what should be the minimum expected performance of a real test agent. In this context, the results were averaged over 20 simulations. The conclusion from this experiment is that the *DQNtf* is the fastest agent of all. Note, however, that in practice it is not suitable to (a) acquire another dataset containing expert demonstrations for the actual application created, train a *BC* agent (at least a few iterations until convergence), then (b) perform transfer learning and train a *DQNtf* agent again. The takeaway of this experiment and its results is that the *DQNtfMod* method, which performs transfer learning from the older application state with 25% modifications compared to the current state (as described above), is suitable in practice and can efficiently provide an adequate test corpus within temporal constraints. The worst result in the table, *BC*-agent, can be explained by the fact that the hard-to-detect problems were often not among the expert demonstrations. The agent managed to train to a reward threshold of 10 in a few minutes but reached the target score of 16 by switching and running for a long time using the small exploration factor provided. This is also an answer to the RQ2, which is that human testers face the challenge of finding these hard-to-spot problems in a short time given the *BC* result, but also the fact that a computer running the *BC* agent can produce tests at a much faster rate than a human user, as shown in Table II. For the statistical tests in this table, we used the times extracted from the dataset records. Note that at evaluation time, since the architecture of *BC* and *DQN* classes are mostly the same in the high-computational parts (MLP layers), the time needed to get through the models of a particular input is similar.

RQ3 is about how powerful the agents are overall, ignoring the short time constraints. In this case, a time window of 24 hours is set for the evaluation. We observed: (a) what is the maximum output reward per agent and (b) when this value is reached for the first time since the start. The results of the

TABLE III
RQ3. MAXIMUM REWARD AND THE TRAINING TIME NEEDED TO REACH
THE VALUE PER AGENT (CLAMPED TO A 24H WINDOW)

Agent	Max Reward	Time in hours
<i>BC</i>	17	13.2h
<i>DQN</i>	24	5.5h
<i>DQNtf</i>	25	16.1h
<i>DQNtfMod</i>	25	22.5h

evaluation are shown in Table III. We see in previous Table II that *BC* method obtains quicker the target reward of 16, but to reach its maximum reward of 17 it takes another 13 hours (13.2-0.2h) as seen in Table III.

The summarisation of the entire evaluation is given below:

- *BC* learns very quickly and proves to be the fastest way to perform quick tests between iterations, even if the application changes when a new dataset is provided for expert demonstration. It is the fastest way to train to achieve a moderate reward result but its performance is limited.
- *DQN* class can outperform *BC* in maximum reward terms, but it takes more time to train to achieve a reasonable reward. This could be suitable for overnight testing or regression testing. The method is important for detecting deep-seated bugs and achieving broader code coverage.
- *DQNtf* is the intermediate solution, able to reuse the results of *BC*, start with a hot pre-trained different base, and outperform the results of a *DQN* starting from scratch. This means that it may make sense to train a *BC* first, then *DQNtf*, and then use the results produced during day builds.
- *DQNtfMod* turns out to be an optimal way to reuse the efficiency of *DQNtf* without having to acquire a new dataset of expert demonstrations.

D. Challenges regarding models and technical aspects.

As in Fig. 3 and 4, agent classes *BC* and *DQN* share an important part of the architecture of the network. This is an important factor in our experiments on reusing a previously learned model (even for an older version of the application) and performing efficient transfer learning to another test model or one representing a new state of an application. The number of shared layers has a direct impact on the efficient transfer method. Since MLP layers are fully configurable, it is important to keep these factors in mind and perform an evaluation on the tested application.

Second, we mentioned that the architecture of *BC* originally had another output header that also predicted the meta-state. The agent was issued a penalty if the meta-state, i.e., the agent's overall intention, did not match that of the human tester (the information is contained in the recorded dataset, where the meta-state in this context represents the user story under test in the episode). At least in the tested applications, we found that since the intention is already used as input in the observation, the neural networks should automatically infer the meta-state and use it in decision making on the

local page/state. However, further experimentation with the developer's application is needed to be sure which version performs better.

Third, intuitively, this serves primarily to discover new possible states, rather than just following the usual paths from the beginning. We have found that it is important for the testing process to maintain a high rate of exploration even in the final epochs of training.

Another observation we made during the development of the framework regarding the efficiency of training the RL algorithms is that the real-world application must first be replicated in a simulator, rather than developing, tuning, and evaluating the models using the real-world application. The reason for this is that the real application always has significant delays after decisions are made, e.g., the button to submit the login page would normally be checked on the server side and then come back with a delayed response. If these delays are not simulated at least in the development and debugging phases, they are likely to cause too much overhead in most applications for the agents to be trained properly. Of course, there is a trade-off between mocking/simulating some of the functionalities that have a long response time and building a full new environment by hand.

The last observation we make concerns the debugging and tuning processes. Our tests have shown that it is important to have the appropriate debugging tools and to be able to understand the relationship between agent performance, current parameters and setup. In our case, the framework uses Tensorboard⁶ graphs and custom graphs to enable quick debugging. For example, we output the min/max/avg gradients and check the average number of dead neurons per MLP layer, statistics on loss functions, rewards, etc. Hyperparameter configuration and efficient model management methods have also been adopted [37] to understand how different parameters can affect performance (e.g., learning rates, depth/width of layers, exploration schedules/rules, etc.)

VI. CONCLUSIONS AND FUTURE WORK

This paper presents a prototype for testing end-user applications that mimics human tester behavior, generally at the UI level (i.e., RPA-like). The methods used are based on Deep Reinforcement Learning and are efficient in not only mimicking human tester behavior but also outperforming human testers. After technical discussions with our UiPath partners, we concluded that in the future we should focus on improving the abstraction levels between the understanding of the model of an application and the RL model of the environment used.

Acknowledgments: We thank UiPath for supporting this project, supplying real-world requirements from the RPA-like testing domain, validating internally the prototype, and providing concrete applications for testing. We also thank our student collaborator Vlad Calomfirescu who shared with us some technical parts of his BSc thesis.

⁶<https://www.tensorflow.org/tensorboard>

REFERENCES

- [1] P. Aho, M. Suarez, T. Kanstrén, and A. M. Memon, “Murphy Tools: Utilizing Extracted GUI Models for Industrial Software Testing,” in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, 2014, pp. 343–348.
- [2] P. Godefroid, M. Y. Levin, and D. Molnar, “SAGE: whitebox fuzzing for security testing,” *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012. [Online]. Available: <https://doi.org/10.1145/2093548.2093564>
- [3] I. Sommerville, *Software Engineering*, 10th ed. Pearson, Jul. 2021.
- [4] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed., ser. Adaptive Computation and Machine Learning. MIT Press, 2018. [Online]. Available: <http://incompleteideas.net/book/the-book-2nd.html>
- [5] H.-G. Nguyen, H.-D. Le, and V. Nguyen, “Prioritizing automated test cases of Web applications using reinforcement learning: an enhancement,” in *2021 13th International Conference on Knowledge and Systems Engineering (KSE)*, Nov. 2021, pp. 1–8, iSSN: 2694-4804.
- [6] Y. Koroglu and A. Sen, “Functional test generation from UI test scenarios using reinforcement learning for android applications,” *Software Testing, Verification and Reliability*, vol. 31, no. 3, p. e1752, 2021.
- [7] F. Torabi, G. Warnell, and P. Stone, “Behavioral Cloning from Observation,” May 2018, arXiv:1805.01954 [cs].
- [8] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, “Rainbow: Combining Improvements in Deep Reinforcement Learning,” 2017.
- [9] E. Razina and D. Janzen, “Effects of dependency injection on maintainability,” in *Proc. of the 11th IASTED International Conference on Software Engineering and Applications*, ser. SEA’07. ACTA Press, Nov. 2007, pp. 7–12.
- [10] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “OpenAI Gym,” 2016, arXiv:1606.01540 [cs].
- [11] A. Bou, M. Bettini, S. Dittert, V. Kumar, S. Sodhani, X. Yang, G. De Fabritiis, and V. Moens, “TorchRL: A data-driven decision-making library for PyTorch,” 2023, arXiv:2306.00577 [cs].
- [12] L. Dobrica, “Robotic process automation platform UiPath,” *Communications of the ACM*, vol. 65, no. 4, pp. 42–43, 2022.
- [13] W. M. P. Van Der Aalst, M. Bichler, and A. Heinzl, “Robotic Process Automation,” *Business & Information Systems Engineering*, vol. 60, no. 4, pp. 269–272, Aug. 2018.
- [14] P. Aho, R. A. P. Oliveira, E. Alégroth, and T. E. J. Vos, “Evolution of Automated Regression Testing of Software Systems Through the Graphical User Interface,” in *Proceedings of the 1st International Conference on Advances in Computation, Communications and Services*, 2016, pp. 16–21.
- [15] Y. Pan, F. Sun, Z. Teng, J. White, D. C. Schmidt, J. Staples, and L. Krause, “Detecting web attacks with end-to-end deep learning,” *Journal of Internet Services and Applications*, vol. 10, no. 1, p. 16, 2019.
- [16] “Google Testing Blog: Just Say No to More End-to-End Tests.” [Online]. Available: <https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html>
- [17] M. Leotta, B. García, F. Ricca, and J. Whitehead, “Challenges of End-to-End Testing with Selenium WebDriver and How to Face Them: A Survey,” in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Apr. 2023, pp. 339–350.
- [18] T. A. T. Vuong and S. Takada, “A reinforcement learning based approach to automated testing of Android applications,” in *Proc. of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2018. ACM, 2018, pp. 31–37.
- [19] N. Sunman, Y. Soydan, and H. Sözer, “Automated Web application testing driven by pre-recorded test cases,” *Journal of Systems and Software*, vol. 193, no. C, 2022.
- [20] V. Nguyen, T. To, and G.-H. Diep, “Generating and selecting resilient and maintainable locators for Web automated testing,” *Software Testing, Verification and Reliability*, vol. 31, no. 3, p. e1760, 2021.
- [21] L. Harries, R. S. Clarke, T. Chapman, S. V. P. L. N. Nallamalli, L. Ozgur, S. Jain, A. Leung, S. Lim, A. Dietrich, J. M. Hernández-Lobato, T. Ellis, C. Zhang, and K. Ciosek, “DRIFT: Deep Reinforcement Learning for Functional Software Testing,” 2020, arXiv:2007.08220 [cs, stat].
- [22] C. Paduraru, M. Paduraru, and A. Stefanescu, “RiverFuzzRL - an open-source tool to experiment with reinforcement learning for fuzzing,” in *14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 430–435, iSSN: 2159-4848.
- [23] J. Eskonen, J. Kahles, and J. Reijonen, “Automating GUI Testing with Image-Based Deep Reinforcement Learning,” in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2020, pp. 160–167.
- [24] M. Leotta, F. Ricca, and P. Tonella, “Sidereal: Statistical adaptive generation of robust locators for web testing,” *Software Testing, Verification and Reliability*, vol. 31, no. 3, p. e1767, 2021.
- [25] S. Di Martino, A. R. Fasolino, L. L. L. Starace, and P. Tramontana, “Comparing the effectiveness of capture and replay against automatic input generation for Android graphical user interface testing,” *Software Testing, Verification and Reliability*, vol. 31, no. 3, p. e1754, 2021.
- [26] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, “Reinforcement learning based curiosity-driven testing of Android applications,” in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. ACM, Jul. 2020, pp. 153–164.
- [27] T. E. J. Vos, P. Aho, F. Pastor Ricos, O. Rodriguez-Valdes, and A. Mulders, “TESTAR – scriptless testing through graphical user interface,” *Software Testing, Verification and Reliability*, vol. 31, no. 3, p. e1771, 2021.
- [28] A. S. Vezhnevets, S. Osindero, T. Schaul, N. Heess, M. Jaderberg, D. Silver, and K. Kavukcuoglu, “FeUdal Networks for Hierarchical Reinforcement Learning,” in *Proceedings of the 34th International Conference on Machine Learning*. PMLR, 2017, pp. 3540–3549, iSSN: 2640-3498.
- [29] H. Walke, J. Yang, A. Yu, A. Kumar, J. Orbik, A. Singh, and S. Levine, “Don’t Start From Scratch: Leveraging Prior Data to Automate Robotic Reinforcement Learning,” Jul. 2022, arXiv:2207.04703 [cs].
- [30] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: <https://www.deeplearningbook.org>
- [31] M. Kelly, C. Sidrane, K. Driggs-Campbell, and M. J. Kochenderfer, “HG-Dagger: Interactive Imitation Learning with Human Experts,” 2019, arXiv:1810.02890 [cs].
- [32] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He, “A Comprehensive Survey on Transfer Learning,” Jun. 2020, arXiv:1911.02685 [cs, stat].
- [33] Q. Lan, A. R. Mahmood, S. Yan, and Z. Xu, “Learning to Optimize for Reinforcement Learning,” 2023, arXiv:2302.01470 [cs].
- [34] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized Experience Replay,” 2016.
- [35] W. Dabney, G. Ostrovski, and A. Barreto, “Temporally-Extended $\{\epsilon\}$ -Greedy Exploration,” Jun. 2020, arXiv:2006.01782 [cs, stat].
- [36] T. Kobayashi and W. E. L. Ilboudo, “t-soft update of target network for deep reinforcement learning,” *Neural Networks*, vol. 136, pp. 63–71, 2021.
- [37] S. Schelter, F. Biessmann, T. Januschowski, D. Salinas, S. Seufert, and G. Szarvas, “On Challenges in Machine Learning Model Management,” *IEEE Data Eng. Bull.*, vol. 41, no. 4, pp. 5–15, 2018.