arXiv:2007.08220v1 [cs.LG] 16 Jul 2020

DRIFT: Deep Reinforcement Learning for Functional Software Testing

Luke Harries*, Rebekah Storan-Clarke*, Timothy Chapman, Swamy V. P. L. N. Nallamalli,

Levent Ozgur, Shuktika Jain, Alex Leung, Steve Lim, Aaron Dietrich,

José Miguel Hernández-Lobato, Tom Ellis, Cheng Zhang**, Kamil Ciosek****

Abstract

Efficient software testing is essential for productive software development and reliable user experiences. As human testing is inefficient and expensive, automated software testing is needed. In this work, we propose a Reinforcement Learning (RL) framework for functional software testing named DRIFT. DRIFT operates on the symbolic representation of the user interface. It uses Q-learning through Batch-RL and models the state-action value function with a Graph Neural Network. We apply DRIFT to testing the Windows 10 operating system and show that DRIFT can robustly trigger the desired software functionality in a fully automated manner. Our experiments test the ability to perform single and combined tasks across different applications, demonstrating that our framework can efficiently test software with a large range of testing objectives.

1 Introduction

Testing computer software is a crucial element of modern software engineering practice. The push towards continuous integration and continuous delivery (CI/CD) of software requires efficient testing to ensure the builds are stable [1]. Otherwise, software delivery may be delayed or bugs may result in poor user experience.

While this is true for all software, tests are especially important for operating systems, where bugs could impair core system functions and introduce security vulnerabilities.

The most time-consuming part of software testing is testing through interactions with the Graphical User Interface (GUI), which was traditionally done manually. However, creating GUI tests manually is a time-consuming and expensive process. In particular, testing large numbers of interacting components takes many hours and small changes in the software can easily break many of these tests. [2]. As such, companies often outsource this testing to humans through quality assurance (QA) companies and/or beta users.

Alternatively, large numbers of automated agents may be deployed to interact with the software. The current generation of these agents commonly act using a fixed random policy. This results in poorly-targeted tests, compared to those performed by a human user. To improve efficiency, heuristics have been added to random agents [3]. However, a better automated solution for efficient software testing is clearly needed.

As system complexity increases, two features of software tests become critically important. First, since it is infeasible to manually specify a huge number of tests, they have to be fully automatic.

* Equal contribution. Completed during the AI residency at Microsoft Research, Cambridge, UK.
** Equal contribution from senior authors. All authors are affiliated with Microsoft.

Second, the testing framework needs to be *sample-efficient*, meaning the ability to complete the tests with a reasonable amount of interactions. As discussed, existing approaches do not meet these requirements [4, 5].

In this work, we propose an efficient software testing framework addressing these requirements, by exploiting insights from Deep Reinforcement Learning [6–8]. Our goal is to train an RL agent which can perform efficient software testing with specified properties, such as testing specific functionalities with different coverage. We name our proposed framework DRIFT, which stands for Deep ReInforcement learning for Functional software-Testing.

Contributions We design a novel Batch Reinforcement learning framework, DRIFT, for software testing. We use the tree-structured symbolic representation of the GUI as the state, modelling a generalizeable Q-function with Graph Neural Networks (GNN). We introduce a fully modular and automated setup to train agents to perform desired tasks. Additionally, the programmer can designate, in a language-agnostic way, which functionality should be tested. Afterwards, we evaluate DRIFT on the Windows 10 operating system, showing that trained agents can learn single tasks as well as multiple tasks with different coverage requirements. These agents outperform a random baseline by two orders of magnitude and can successfully generalize in settings where hash-based methods failed.

2 Background

To formally define our testing framework, we require several concepts that we will now introduce.

Markov Decision Process We formalize the interaction between the agent and the environment as a family of Markov Decision Processes [9] (MDP), indexed by the objective, o. An MDP is defined as a tuple $(S, A, T, R_o, \gamma, \perp_o)$, where S is the set of states, A is the set of actions, T is the transition function, R_o is the reward function corresponding to an objective o, γ is the discount factor and $\perp_o$ is the set of absorbing states. The transition function $T(s_{t+1}, s_t, a_t) = \mathbb{P}(s' = s_{t+1}|s = s_t, a = a_t)$ models the transition to the next state given the current state and action. The reward function for a given objective $R_o(s_{t+1}, s_t, a_t) = \mathbb{E}_\tau[r_t^o|s = s_t, a = a_t, s' = s_{t+1}]$ (we skip the superscript o in the remainder of the paper where it is clear). Each step in the MDP can be described with a $\text{transition} := (s_t, a_t, r_t, s_{t+1})$. An episode is a sequence of transitions until termination $\text{episode} := [s_0, a_0, r_0, s_1, a_1, r_1 ...]$.

Reinforcement Learning In reinforcement learning, the agent is faced with a sequential decision-making problem. At each time step t, the agent receives the state $s_t \in S$ from the environment. The agent has a policy π which selects an action given the state $\pi(a|s) = P[A = a|S = s]$. The action is passed to the environment, and the state is updated using the transition function T. The environment then returns the new state s_{t+1} and a scalar reward $r_{t+1}^o \in \mathbb{R}$ determined by the (objective-dependent) reward function. This interaction continues until a goal is achieved and thus the episode is completed. The task for the agent is to learn a policy π which maximizes the total discounted reward $J_t^o = \mathbb{E}_\tau[\sum_t \gamma^t r_t^o]$ received from the environment where the discount $\gamma \in (0, 1]$ [10, 11].

Trees and Graph Neural Networks A graph is a tuple $G = (V, E)$ containing a set of vertices V and edges $E \subseteq \{(x, y)|(x, y) \in V^2 \wedge x \neq y\}$. A tree is an acyclic graph with a designated root node. Graph Neural Networks (GNNs) are differentiable parameterized functions $f : G \to \mathbb{R}^m$ where $v \in V, v \in \mathbb{R}^n$. Each layer in a GNN updates the representation of each node using its neighboring nodes [12–14].

Off-Policy Reinforcement Learning and Batch Reinforcement Learning Off-policy Reinforcement Learning is often deployed in settings where the policy collecting the experience is different from the policy being learned. This setting is known as off-policy reinforcement learning [15]. Batch Reinforcement Learning (Batch-RL) describes a subset of off-policy learning where a fixed set of transitions is used, with no further interaction with the environment [16–18]

3 Method

We now describe the components of DRIFT. Abstractly, the framework works by exploiting the operating system API to interact with the software under test. It learns a policy which, given a symbolic representation of the interface, selects desired GUI interactions.

```
{
    "Identifier": "94d29a9543c9c...",
    "UIProperties": [
        {
            "AutomationID": "23423",
            "ClassName": "MainMenu"
            "ControlType": "Panel",
            "ProcessName": "StartMenu"
        }
    ],
    "Children": [...]
}
```

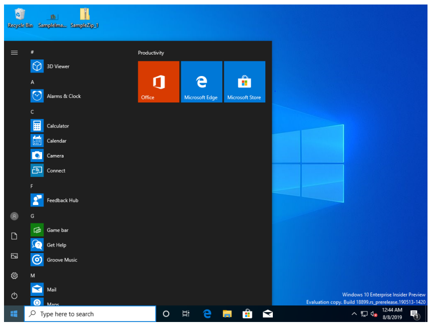


Figure 1: On left subfigure is the UITree for the StartMenu, shown by the large black rectangular box in the screenshot the right subfigure. On average, each state has 36 UI elements, although some graphs have up to 800 UI elements.

In this section, we provide details of how this policy is obtained. We do this by first specifying software testing as a Markov Decision Process and then describing the training process that computes the policy for the MDP.

3.1 The Testing MDP

States The state s_t is a tree that corresponds to a hierarchy of GUI elements, known as a UITree. Each node in the UITree represents a UI element such as the "Start Menu" as shown in Figure 1. Each node has four properties: the *AutomationID*, an optional string which is used for manual UI testing; the *ClassName*, a string which determines the visual properties of the element; the *ControlType*, an enum describing the type of element; and the *ProcessName*, the name of the process which created the element. Additionally, the element may have children which correspond GUI elements that it contains. The root of the tree is the Desktop node.

The UITree is obtained using Microsoft UI Automation Tree Tool [19] which is traditionally used for accessibility tools such as screen readers. Alternative tree representations of the interface may be used to test different systems, for example the Document Object Model (DOM) for testing websites.

An alternative to the UITree would be to use screenshots of the interface [20]. However, using tree representations has several advantages over screenshots. Firstly, the features of each element are encoded directly into the node properties such as ClassName Button, Label, ScrollBar, etc. which eliminates the need to learn to visually recognize each element. Secondly, the UITrees are invariant to the location of the windows on the screen. Thirdly, the UITrees have a smaller memory footprint of a few KBs compared to several MBs for a screenshot. Fourthly, we have access to a very large number of historical trajectories where only the UITrees are stored. Combined, these features mean that a testing framework that makes use of UITrees is much more efficient and well worth the small amount of computation the API has to perform to obtain them.

Actions The actions in the testing MDP reflect the possible interactions the user might have with the software such as clicking the mail icon to open to mail application. We extract the actions from the UITree. In particular, each possible action is a tuple consisting of a hash of the UIProperties of the node, known as the node identifier, and the action type, for example ("94d29a9543c9c...", "LeftClick"). As different windows may have a variable number of items the user can interact with, the number of actions that are available at each step is dynamic. The number of available actions varies from 2 to 842, with the mean being 248.

Reward Function For important systems, there often exists a list of key functions that should be verified as operational before a build is released. This process of verification is known as smoke testing [21]. In Windows 10, these key functions span many different apps. For example, adding a website to favorites in Edge or clicking "Add Bluetooth or other device" in System Settings.

We use several reward functions, one for each objective o that our framework is trying to achieve. The reward function computes a scalar reward from a state-action pair. To know when a particular objective has indeed been achieved, we use an API function that triggers the generation of logs. The logs are converted into a scalar reward by counting the number of times the event related to a particular function has occurred. Our testing setup is fully modular, allowing the programmer to easily mark, in a language-agnostic way, which functionality should be tested. The agent adapts both to the objective and to any changes in the software that might have changed the best path towards the existing objectives.

3.2 Training

The Simulator We developed a training environment by interfacing with the Windows operating system. We followed OpenAI's gym specification [22]. However, since GUI events are processed in real-time, the time required to complete each step (such as opening Outlook) can be up to several seconds. This is very different from step lengths of several milliseconds, typical of most typical RL environments. This, combined with the often complex sequence of actions needed to perform a particular task, impairs the ability to directly learn from the simulator in a reasonable length of time. Therefore, we trained our agent using a large cache of historical data. The simulator was then only used to evaluate the agents. The simulator was reset when the desired task was achieved.

Historical Data To overcome the limitations of the relatively slow simulator, we learn solely from historical data. In our case, the historical data are the episodes collected by random agents during previous testing runs. We query the historical data for the particular episodes where a particular objective is achieved, helping to alleviate the problem of sparse rewards. Our testing framework is generic with respect to the source of historical data. For alternative systems under test, interactions recorded from real users could be used as well.

Q-learning For training, we use a variant of Q-learning known as DQN [15, 7]. As Q-learning is an off-policy algorithm, the training data does not have to be collected from the policy being trained. In Q-learning, we learn the Q-function which allows us to estimate the expected cumulative reward of a state-action pair under the optimal policy $\pi^\star$. By default, the policy π of the agent is *greedy* with respect to the current estimate of the Q-function, selecting the action with the highest expected reward $\pi(s) = \arg\max_{a \in A} Q_\pi(s, a)$. Q-learning iteratively performs the update $Q'_\pi(s, a) \leftarrow r + \max_{a \in A} Q(s', a)$ for tuples (s, a, r, s').

Due to the large possible number of combinations of state-action pairs, and to achieve the ability to generalize across state-action pairs, we approximate the function using a neural network and optimize the network parameters θ by using stochastic gradient descent to minimize the loss function L. The stochastic gradient descent update is shown below where β represents the learning rate. In practice, the learning rate is dynamically set using the Adam optimiser [23] with an initial value of 1×10^{-2}.

The combination of approximation (our use of neural networks), bootstrapping, and off-policy learning results in instability during training known as the deadly triad [15]. We use an experience replay buffer and dual Q networks to aid stability [7]. The loss function L is based on [7], D represents the dataset of transitions (s, a, r, s') contained in the experience replay, θ represents the parameters of the policy network, and θ^- represents the values of the target network

$$L(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim D}[(r + \gamma \max_{a' \in A} Q(s', a'; \theta_i^-) - Q_\pi(s, a; \theta_i))^2] \tag{1}$$

An additional parameter $\eta \in \mathbb{R}^+$ determines the frequency at which the target network parameters are updated using the policy network parameters. η and γ are chosen using a hyperparameter sweep. Algorithm 1 gives an overview of our training procedure.

Modeling the state using Graph Neural Networks The state is initially returned from the environment as a UITree. We first convert the nodes of the UITree to vector form and then apply the graph neural network.

For a graph with $n \in \mathbb{N}^+$ nodes, we set the representation of each node as the concatenation of the one hot encoding of the node's *UIProperties*. Specifically, we use a variant of one hot encoding where the rarely seen/unseen properties are grouped into an "Other" value. This ensures that the network can handle unseen values at evaluation time. We represent each node embedding $\mathbf{v} \in \{0, 1\}^z$. We define the vectorized action $\mathbf{a} \in (\mathbf{a}_e, \mathbf{a}_i)$ as the concatenation of the one hot encoding of the action type $\mathbf{a}_e\{\text{LeftClick}, \text{RightClick}, ...\}$ and $\mathbf{a}_i$ the one hot encoding of the node index $[0..n]$.

We use a graph neural network (GNN) to approximate the action-value function where $s = (V_s, E_s)$ and $Q(s, a) = \text{GNN}(V_s, E_s) \cdot \mathbf{a}$. The GNN is applied to the state and outputs a matrix containing a vector representation for each node. The expected cumulative reward for performing each of the

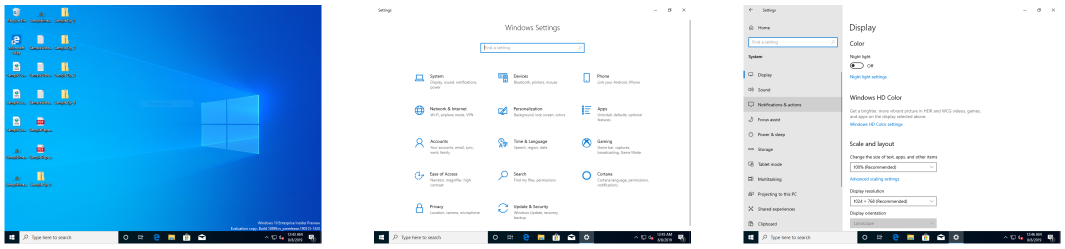

Settings
Windows Settings
Find a setting
System
Display, sound, notifications, power
Devices
Bluetooth, printers, mouse
Phone
Link your Android, iPhone
Network & Internet
Wi-Fi, airplane mode, VPN
Personalization
Background, lock screen, colors
Apps
Uninstall, defaults, optional features
Accounts
Your accounts, email, sync, work, family
Time & Language
Speech, region, date
Gaming
Game bar, captures, broadcasting, Game Mode
Ease of Access
Narrator, magnifier, high contrast
Search
Find my files, permissions
Cortana
Cortana language, permissions, notifications
Privacy
Location, camera, microphone
Update & Security
Windows Update, recovery, backup
Type here to search
Home
Display
Sound
Notifications & actions
Focus assist
Power & sleep
Storage
Tablet mode
Multitasking
Projecting to this PC
Shared experiences
Clipboard
Color
Night light
Off
Night light settings
Windows HD Color
Get a brighter, more vibrant picture in HDR and WCG videos, games, and apps on the display selected above.
Windows HD Color settings
Scale and layout
Change the size of text, apps, and other items
100% (Recommended)
Advanced scaling settings
Display resolution
1024 × 768 (Recommended)
Display orientation
Landscape
12:46 AM
8/8/2019

Figure 2: States seen whilst navigating to the notifications panel. At the start of each episode, the System Settings process is started by the simulator.. To achieve the reward the agent must select the "System" element followed by the "Notifications Panel" element.

corresponding actions, followed by the agent acting greedily, is calculated by performing a dot product with the vectorized action. Here, we want a network architecture which takes into account the hierarchy within the tree and is able to differentiate between nodes with identical properties but different locations within the tree. We chose the GNN architecture as it is specially designed to operate on graph structures [14].

Algorithm 1 DRIFT Batch-RL

```
Input: The desired objective $o$, the application processes relevant to the reward *process*, the historical data *D*
Output: Trained GNN *net*

transitions ← list()

# Extract relevant transitions
for episode in D do
    if episode_meets_objective(episode, o) then
        cropped_episode ← crop(episode, o)
        transitions += cropped_episode

# Vectorize the states and actions
for t in transitions do
    for s in ['state', 'next_state'] do
        t[s] ← get_process(t[s], process)
        t[s] ← vectorise(t[s])
    t['r'] ← get_reward(o, t)
    t['a'] ← vectorise_action(t['a'])

net ← GNN()
target_net ← net.copy()
update_frequency ← 100
steps ← 0
# Train the GNN
for batch in DataLoader(transitions) do
    net ← train(batch, net, target_net)
    if update_frequency mod steps = 0 then
        target_net ← net.copy()
```

At each time step or each layer, the GNN updates the representation of each node based on the current node representation and its neighboring nodes. We chose GNN architecture by using a toy supervised learning task (classifying whether the node ClassName is Button). We selected the Graph Attention Network (GAT) [24] as it achieved the best performance. GAT is a convolution style architecture which uses self-attention to determine the relative weighting of the neighboring nodes. Each layer of the GAT updates each node representation

$$\mathbf{v}'_i = \alpha_{i,i}\mathbf{v}_i + \sum_{j \in \mathcal{N}(i)} \alpha_{i,j}\mathbf{v}_j. \qquad (2)$$

where $\alpha_{i,j}$ represents the self-attention with i being the index of the query node and j being the index of the key node.

We implemented the GAT architecture using PyTorch Geometric [25]. Our final architecture consisted of two GAT layers separated by a Rectified Linear Unit (ReLU) [26]. The first layer has 1080 input channels (the length of the vectorized node representation) and 80 outputs channels with 8 self-attention heads. The second layer outputs 1 channel (the number of action types) and 1 self-attention head. Only 1 action type was used as all the key functions could be performed with only the *LeftClick*.

Efficiency vs Coverage We introduce two variants of DRIFT: DRIFT-Greedy and DRIFT-Sampler. DRIFT-Sampler focuses on performing one given objective most efficiently. It uses a greedy policy, which selects the action with the highest expected reward $\pi(s,a) = \arg\max_{a \in A} Q(s,a)$. However, when testing software, we may also want to maximize state coverage in addition to achieving the desired objective. To do this, we use DRIFT-Sampler, which has a stochastic policy $\pi(s,a) = a \sim \text{Categorical}(\text{Softmax}([Q(s,a)/m | a \in A]))$ where the temperature parameter $m \in (0, \infty]$. Smaller

Task	DRIFT-Greedy (Train, Eval)	Q-hash Agent (Train, Eval)	Random Agent (Evaluation)
Navigate to notifications panel (Settings)	2 ± 0, $\mathbf{2} \pm 0$	2 ± 0, Fail	443 ± 1235
Add Bluetooth or other device (Settings)	2 ± 0, $\mathbf{2} \pm 0$	2 ± 0, Fail	852 ± 1681
Add a page to favorites (Edge)	2 ± 0, $\mathbf{2} \pm 0$	2 ± 0, Fail	444 ± 1372

Figure 4: The number of steps required for each agent to complete a specific task on a simulator. For DRIFT-Greedy and Q-hash, the evaluation was performed using 1000 evaluation steps by the agent in the simulator. The result shown is the mean and standard deviation of 5-fold cross-validation with 4 random seeds. The number of steps for the Random Agent is calculated using the mean historical data and includes the standard deviation. DRIFT-Greedy was able to successfully learn to achieve all the tasks following the optimal route. Although the random agent was able to achieve all the tasks, it was very *sample-inefficient*, requiring a large number of steps. The Q-hash based agent failed to achieve any of the tasks on the unseen evaluation environments as it was unable to generalize.

temperature values bias the sampled actions towards the action with the largest predicted reward while higher temperature values bias the sampled actions toward a uniform distribution across actions.

Multiple Tasks To support multiple tasks, DRIFT is trained train on episodes where the tasks are performed individually. Since the rewards are normalized across tasks, this gives a combined policy that targets all tasks. During training, we monitor performance on each task. During testing, we deploy the multi-objective agent using DRIFT-Sampler so that the agent explores more efficiently and is more robust to imperfections in the learned policy.

4 Experiments

We use DRIFT to perform a variety of different software tests within Windows 10. In particular, we verify key functionality of the System Settings App and the Edge browser. DRIFT-Greedy finds the most efficient path to the objective, outperforming the random agent baseline by two orders of magnitude, and consistently achieving the reward, unlike the Q-hash baseline which failed to generalize. We show that the temperature parameter of DRIFT-Sampler can be varied to effectively trade-off between efficiency at completing the task and coverage of the possible states. Additionally, we demonstrate that DRIFT-Sampler can learn to accomplish multiple testing objectives.

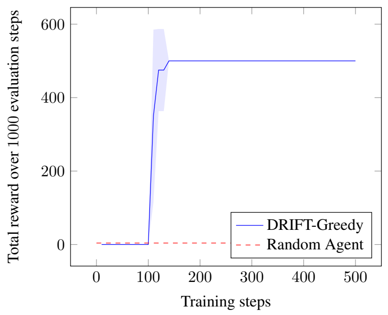
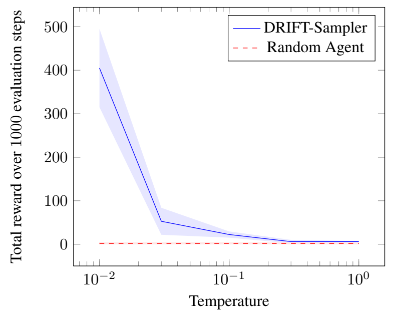


Figure 3: The cumulative reward achieved by an agent on a single task for a given amount of training. The task is to navigate to the notifications page within the settings app for which it is given a reward of 1. The shaded area represents the standard deviation of the DRIFT-Greedy agent over 5 folds and 4 seeds.

Experimental setup The agents were trained on historical data as described in the Methods section. For each reward, 20 episodes of training data were used, each with an average of 44 transitions. We used a batch size of 128 transitions, where each batch is known as a step. We used random search to determine which hyperparameters were able to most efficiently learn to navigate to the notifications page. The best frequency of the target network updates was found to be 100 and the optimal discount factor γ was set to 0.1. All experiments are repeated using 5-fold cross-validation, holding out a subset of the historical data used for training, with 4 different seeds. We implemented the algorithm using PyTorch and PyTorch Geometric [27, 25]

As the naive agent required a huge number of samples to complete the tests it was very expensive to evaluate, taking on average 443 steps to 852 steps with a very large standard deviation to achieve one desired goal state. Rather than let it timeout, we performed the evaluation off-policy.



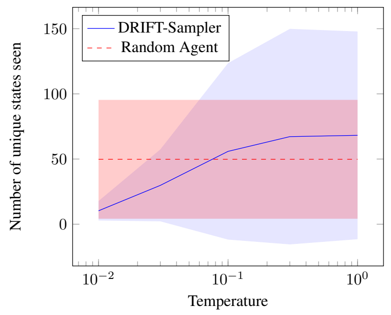


Figure 5: Number of rewards achieved and unique pages visited given different temperature. The task is to navigate to the notifications panel within the settings app. As the temperate increases, the distribution from which the actions are sampled becomes more uniform and so more states are visited. The shading represents the standard deviation across 5 folds and 4 seeds.

We use an additional baseline, which we call Q-hash, where the representation of a state-action pair was obtained by hashing. Q-hash learned the correct path on the training data, however, it failed on both held-out data and the evaluation simulator. This was caused by the fact that slight variations in the state greatly change the hash representation, which meant the algorithm was unable to generalize. Although method was inspired by [28], we do not use the same feature engineering, or a process of selecting stable elements, which is the reason our Q-hash agent was unable to generalize.

4.1 Solving a specific task

We evaluated the ability of DRIFT-Greedy to perform tasks by measuring the number of times a particular task is achieved in 1000 steps. For example, the "Navigate to notifications panel" task is shown in Figure 2. Figure 3 demonstrates that all runs of DRIFT, regardless of which seed or folds of data were used for training, were able to learn the most efficient route to the notifications panel (2 steps). In contrast, by analyzing the historical data we can see that the random agent took on average 443 steps with a very large standard deviation. The Q-hash approach failed to perform the task due to an inability to generalize to slight differences in the state of the simulator compared with the training data. Results for a larger number of different tasks are shown in Figure 4.

4.2 Trading off efficiency and state coverage

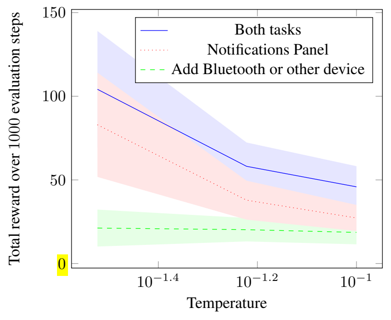


Figure 6: The cumulative reward achieved by DRIFT-Sampler on multiple tasks given different temperature. The agent is given a reward of 1 for each of the tasks: firstly, navigating to the notifications panel; secondly, navigating to and clicking the button "Add Bluetooth or other device".

An additional consideration when performing automated software testing is covering a large number of possible states. We evaluate DRIFT-Sampler's ability to trade-off coverage and ability to achieve the desired reward by investigating the effects of the temperature parameter. The left panel of Figure 5 shows that as we increase the temperature the number of steps required to achieve the reward decreases. The right panel shows that as the temperature increases the number of unique states visited increases as well. Together, these panels show that the temperature can be used to effectively trade-off between task efficiency and state coverage given a trained value function.

4.3 Multiple Testing Objectives

We evaluated our testing framework where there were multiple functions to test, as is often the case in practice. To do this, we trained a single agent on a mixture of episodes containing a mixture of several reward signals.

We trained the agent to perform two tasks within the Settings application. The first task was to navigate to the notifications panel, as shown in Figure 2, requiring two steps. The second task was to navigate to the "Devices" page and then click "Add Bluetooth or other device", similarly requiring two steps. A reward of 1 was given for completing either task.

As our agent has no memory, learning to achieve the tasks sequentially was not possible. Instead, we used the DRIFT-Sampler so that at each step the agent would sample from the likely actions and, hopefully, on different runs, randomly choose between the different objectives. As shown in Figure 6 the agent was able to solve both tasks at least some of the time over 1000 evaluation steps. However, the "Add Bluetooth or other device" task was completed less often. On average, it was accomplished 21 times using a temperature of 0.03 and 18 times using a temperature of 0.1. This was due to the fact that some of the training episodes for the Bluetooth task also contained successful solutions to the Notifications panel task, while the opposite was not the case, biasing the agent to one of the tasks.

5 Related Work

Our framework has the same underlying objective as other tools for automated software testing — using the GUI to find the kind of bugs that are easily reachable by the end-user [29, 28, 5]. However, many of these tools use a policy without a feedback loop, simply generating a sequence of inputs from a fixed distribution. For example, Monkey uses a random policy to test android apps with the action space being a combination of UI interactions and system events such as turning airplane mode off and on [29]. GUITest is a Java application which uses a fixed random policy to test MacOSX apps [30], similarly using the accessibility interface of the operating system. Such fixed policies are often combined with heuristics. For example, the DynoDroid policy is biased towards least recently used actions [3]. The common limitation of these tools is that they require large amounts of time to complete the tests.

Several other tools have been produced to deal with this limitation. Sapienz uses genetic algorithms to optimize the sequences generated by the random policy [5]. Sapienz learns to perform a sequence of events and motifs (hand-crafted sequence of events) depending on the current screenshot. Humanoid [20] learns to generate human-like actions by using a convolutional neural network to learn a mapping from the screenshot to the actions selected by end-users. It has been theorized that the improved coverage compared to other tools was the result of learning to prioritize more critical UI elements.

The system [31] proposed a Q-learning based approach to finding bugs within MacOSX applications. They suggested using a Q-table, where a Q-value is learned for each state-action pair. To do this, they propose hand-engineering the representation of the state and actions which can then be used to lookup the Q-table for the corresponding Q-value. This approach was developed by [28] resulting in the "Testar" tool, where the state and action representations are created by their respective hashes on hand-selected, application-specific stable elements. This paper inspired our Q-hash baseline. However, we found that our model, which lacked feature-engineering of stable elements, was unable to generalize to any small variations in the state and actions, suggesting that this method is not robust.

6 Conclusion

We propose DRIFT, a novel efficient software testing framework. We first formalize the software testing task as an MDP and solve it using deep RL. Our agent operates on a symbolic representation of the GUI and uses a graph neural network to model the state-action value function. To amortize the cost of data collection, it is trained from existing data using the batch-RL paradigm. We demonstrate our framework on several testing tasks on the Windows 10 platform. Our agents outperform the baseline, a fuzzing tool with a policy independent of the state, by two orders of magnitude. Moreover, we compare to a simpler Q-learning approach, which represents the state-action tuples using a hash function, that failed to generalize. We introduce a sampling agent and demonstrate that we can trade-off efficiency and coverage in addition to learning to perform multiple tasks. In future work, we hope to improve exploration by incorporating agent memory, which would allow more complex tests, where the current action depends on the whole history of interactions between agent and environment rather than just the current state.

References

[1] Sean Stolberg. Enabling agile testing through continuous integration. *2009 Agile Conference*, 2009.

[2] Yauhen Leanidavich Arnatovich and Lipo Wang. A systematic literature review of automated techniques for functional GUI testing of mobile applications. *CoRR*, abs/1812.11470, 2018.

[3] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.

[4] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. Understanding the test automation culture of app developers. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015.

[5] Ke Mao, Mark Harman, and Yue Jia. Sapienz: multi-objective automated testing for android applications. *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*, 2016.

[6] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[8] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[9] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[10] Richard S. Sutton and Andrew G Barto. *Reinforcement Learning: an introduction*. MIT Press, 2018.

[11] Rahul Gupta, Aditya Kanade, and Shirish K. Shevade. Deep reinforcement learning for programming language correction. *CoRR*, abs/1801.10467, 2018.

[12] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

[13] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[14] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Graph neural networks: A review of methods and applications. *CoRR*, abs/1812.08434, 2018.

[15] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*. MIT press Cambridge, 1998.

[16] Sascha Lange, Thomas Gabel, and Martin Riedmiller. Batch reinforcement learning. *Adaptation, Learning, and Optimization Reinforcement Learning*, page 45–73, 2012.

[17] Scott Fujimoto, David Meger, and Doina Precup. Off-policy deep reinforcement learning without exploration. *CoRR*, abs/1812.02900, 2018.

[18] Rishabh Agarwal, Dale Schuurmans, and Mohammad Norouzi. Striving for simplicity in off-policy deep reinforcement learning. *CoRR*, abs/1907.04543, 2019.

[19] UI Automation Team. Using ui automation for automated testing, Mar 2017.

[20] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. A deep learning based approach to automated android app testing. *CoRR*, abs/1901.02633, 2019.

[21] A.m. Memon and Q. Xie. Studying the fault-detection effectiveness of gui test cases for rapidly evolving software. *IEEE Transactions on Software Engineering*, 31(10):884–896, 2005.

[22] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[23] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *3rd International Conference for Learning Representations, San Diego, 2015*, 2015.

[24] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018. accepted as poster.

[25] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[26] Richard Hahnloser, Rahul Sarpeshkar, Misha A. Mahowald, Rodney Douglas, and H Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405:947–51, 07 2000.

[27] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. *NIPS 2017 Workshop Autodiff*, 2017.

[28] Tanja E.j. Vos, Peter M. Kruse, Nelly Condori-Fernández, Sebastian Bauersfeld, and Joachim Wegener. Testar. *International Journal of Information System Modeling and Design*, 6(3):46–83, 2015.

[29] Google. Android monkey, 2018.

[30] Sebastian Bauersfeld and Tanja E. J. Vos. Guitest: a java library for fully automated gui robustness testing. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, 2012.

[31] Sebastian Bauersfeld and Tanja E. J. Vos. A reinforcement learning approach to automated gui robustness testing. 2012.