Using LLM-Based Deep Reinforcement Learning Agents to Detect Bugs in Web Applications

Yuki Sakai, Yasuyuki Tahara, Akihiko Ohsuga and Yuichi Sei
The University of Electro-Communications, Japan
s2230061@edu.cc.uec.ac.jp, {tahara, ohsuga, seiuny}@uec.ac.jp

Keywords: Black-Box GUI Testing, Web Applications, Deep Reinforcement Learning, Large Language Model, Automated Testing.

Abstract: This paper presents an approach to automate black-box GUI testing for web applications by integrating deep reinforcement learning (DRL) with large language models (LLMs). Traditional GUI testing is often inefficient and costly due to the difficulty in generating comprehensive test scenarios. While DRL has shown potential in automating exploratory testing by leveraging GUI interaction data, such data is browser-dependent and not always accessible in web applications. To address this challenge, we propose using LLMs to infer interaction information directly from HTML code, incorporating these inferences into the DRL's state representation. We hypothesize that combining the inferential capabilities of LLMs with the robustness of DRL can match the accuracy of methods relying on direct data collection. Through experiments, we demonstrate that LLM-inferred interaction information effectively substitutes for direct data, enhancing both the efficiency and accuracy of automated GUI testing. Our results indicate that this approach not only streamlines GUI testing for web applications but also has broader implications for domains where direct state information is hard to obtain. The study suggests that integrating LLMs with DRL offers a promising path toward more efficient and scalable automation in GUI testing.

1 INTRODUCTION

In software development, testing is a crucial process. Particularly in web applications (web apps), black-box GUI testing can be costly (Bertolino, 2007). As a result, research efforts are underway to automate the creation and execution of test scenarios (Sneha and Malle, 2017). Additionally, exploratory testing, which does not rely on predefined scenarios, has been proposed as a testing methodology. Exploratory testing leverages the intuition and experience of testers to discover bugs, and it is considered an effective means for bug detection (Itkonen and Rautiainen, 2005).

The primary approach to automating exploratory testing is through deep reinforcement learning (DRL). Recent studies have shown that leveraging interaction information of GUI elements, rather than focusing solely on their states, can enhance performance (Romdhana et al., 2022). However, in web apps, the interaction information of HTML elements depends on the browser, and some browsers cannot retrieve this information. Thus, large language models (LLMs) can be used to infer interaction information from HTML and incorporate these inferences into the state, in order to verify whether accuracy remains comparable to using actual interaction data (Brown et al., 2020).

DRL is known for its robustness (Carlini and Wagner, 2017), whereas LLM inference results are probabilistic (Xia et al., 2024). We hypothesize that these characteristics are compatible. If this hypothesis is confirmed, it could have applications in various fields. Since the extensive knowledge of LLMs is not limited to web apps (Chang et al., 2024), they can be utilized when obtaining states in DRL is difficult, or serve as clues during agent training.

2 RELATED RESEARCH

2.1 Black-Box GUI Testing with DRL

Various approaches have been proposed for automating black-box GUI testing (Wetzlmaier et al., 2016) (Adamo et al., 2018). Recently, methods utilizing DRL have also been proposed. Eskonen et al. proposed a method for web apps that uses GUI screenshots as input for DRL, achieving higher exploration

Sakai, Y., Tahara, Y., Ohsuga, A. and Sei, Y.
Using LLM-Based Deep Reinforcement Learning Agents to Detect Bugs in Web Applications.
DOI: 10.5220/0013248800003890
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 17th International Conference on Agents and Artificial Intelligence (ICAART 2025) - Volume 3*, pages 1001-1008
ISBN: 978-989-758-737-5; ISSN: 2184-433X
Proceedings Copyright © 2025 by SCITEPRESS – Science and Technology Publications, Lda.

accuracy than random search and Q-learning-based methods (Eskonen et al., 2020).

Andrea Romdhana et al. proposed ARES, a DRL-based Android app testing framework (Romdhana et al., 2022). GUI is retrieved in XML format via Appium. A vector consisting of the visibility of GUI elements and the effectiveness of interactions is used as the state. Error discovery and new element exploration are used as positive rewards. As a result, it achieved higher exploration accuracy than Q-learning-based methods. In recent years, besides ARES, further research targeting mobile apps has been conducted (Cai et al., 2021) (Tao et al., 2024).

Research focusing on web apps is also important, and there are two reasons for this. The first reason is the difference in release spans. Typically, releasing a mobile app requires store review, which can take from several days to up to seven days (Apple, 2024) (Google, 2024). Because releasing takes time, so does releasing bug fixes, so thorough testing is required before introducing new features. By contrast, web apps can be released simply by uploading files to the server. Through automated testing, the testing process can be streamlined, enabling more frequent releases. The second reason is differences in frameworks and types of interactions. During automated test execution, Appium is used for mobile apps, while Selenium is used for web apps. However, the information that can be obtained and the operations that can be performed differ based on the framework. For these reasons, applying methods proposed for mobile apps to web apps is valuable.

2.2 GUI Test Specialized LLM Agent

Yoon et al. proposed a GUI testing framework utilizing LLMs (Yoon et al., 2023). The framework consists of four types of agents: Planner, Actor, Observer, and Reflector. First, the Planner generates high-level test cases considering diversity, realism, difficulty, and importance. Next, the Actor determines and executes specific actions to achieve the generated test cases. The Observer monitors the post-action state of the GUI and outputs it. Finally, the Reflector reviews the execution and provides feedback to the Planner. This approach demonstrated significant results in exploration and functional coverage. Yoon et al. identified monetary cost as a challenge because their framework relies on LLMs accessed through the OpenAI API.

2.3 LLMs as a Reward Function

Kwon et al. proposed a method that utilizes LLMs as reward functions in reinforcement learning (Kwon et al., 2023). Designing reward functions in reinforcement learning is challenging because it is difficult to specify desired behaviors through reward functions, and creating effective reward functions requires specialized knowledge. Therefore, by using LLMs, Kwon et al. enabled the use of natural language as an interface, successfully reducing the difficulty of designing reward functions. Users provide examples or descriptions of desired behaviors as text prompts to the LLM. The LLM outputs reward signals based on these prompts to update the behavior of the reinforcement learning agent. In multiple tasks, the proposed method demonstrated superior performance compared to conventional methods. This approach uses LLMs as reward functions of DRL. In reinforcement learning, executing actions and obtaining states can also require specialized knowledge, which significantly impacts learning. Therefore, the application of LLMs is anticipated in these areas.

3 PROPOSED METHOD

3.1 Reinforcement Learning Method

In this study, we use Proximal Policy Optimization (PPO) as the DRL algorithm (Schulman et al., 2017). For implementing the DRL algorithm, we use Stable Baselines3 (Raffin et al., 2021). The system overview is shown in Figure 1. We defined the action space as the indices in the dictionary that stores GUI elements, and the state space as a one-hot vector consisting of the visibility and clickability of GUI elements. The reward assigns a numerical value based on the cumulative number of new states discovered in the web page. The aim of this research is not to improve performance through changes in the reinforcement learning method, but to examine how substituting LLM inference results affects performance. Consequently, we adopt PPO for its stability in training. PPO is chosen as OpenAI's default reinforcement learning algorithm due to its ease of use and excellent performance(OpenAI, 2024b).

3.2 Utilization of Inference Results by LLMs

Web apps are generally operated through a browser. However, even with the same source code, behavior

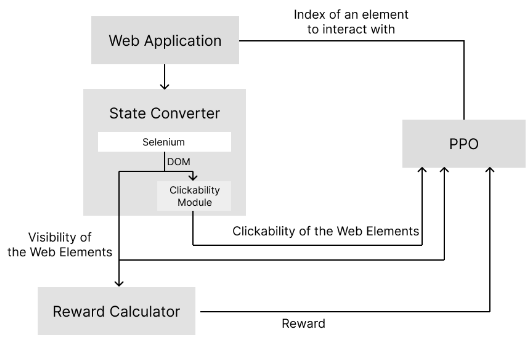
Web Application
Index of an element to interact with
State Converter
Selenium
DOM
Clickability Module
PPO
Visibility of the Web Elements
Clickability of the Web Elements
Reward Calculator
Reward

Figure 1: The System Overview.

can vary across different browsers. Therefore, it is advisable to conduct black-box GUI testing for each browser. Selenium is a well-known automated testing framework for web apps. It provides drivers for each browser, which allows tests to be conducted individually.

In this study, we use interaction information, specifically whether HTML elements are clickable, in our learning process. Because we employ Selenium as the test framework, the interaction information obtainable differs from that in Appium, which is used for mobile apps. Furthermore, determining whether an element is clickable is a runtime process that must be performed through the browser. For Chrome, Selenium provides an interface to execute the necessary Chrome DevTools Protocol commands. Meanwhile, although such data can be obtained via the developer tools in Safari or Firefox, Selenium does not provide an interface for these browsers. Some browsers may not support retrieving interaction information at all. Therefore, we propose a method using LLMs to infer whether HTML elements are clickable and incorporate this inference into the DRL state. Although the appearance of buttons and other GUI elements varies across sites, the act of inferring clickability from HTML is largely unaffected. If the inference results prove sufficient as a substitute, more efficient black-box GUI testing can be conducted across various browsers.

3.3 Robustness of Machine Learning Models and LLMs

In this study, we utilize the LLM's inference results as part of the DRL state. This approach is based on the hypothesis that the imperfect inference accuracy of LLMs is compensated by the robustness of DRL models, making them an effective combination. The inference accuracy of LLMs is not 100% due to factors such as the incompleteness of training data, the probabilistic nature of LLMs, and the ambiguity of

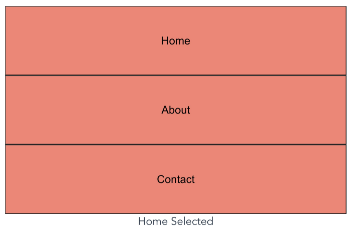

Figure 2: Test Application.

natural language. On the other hand, DRL models are characterized by robustness, meaning they are resilient to noise. This implies that even if some noise is present in the input, the model's output remains stable. By leveraging these characteristics, we expect that incorporating LLMs' inference results into the DRL state will enhance performance.

4 EXPERIMENTS

In this study, we first verify that utilizing the interaction information of HTML elements in web applications can enhance the efficiency of learning. Next, we use an LLM to infer the interaction information of HTML elements and incorporate these results as part of the state in DRL. We then verify whether incorporating these inferred results allows us to achieve accuracy comparable to that when the inferred results are not used.

4.1 Original Test Application Creation

In this study, we created and used a custom web application as the test subject. There are two reasons for this. The first reason is that test apps in research on automating black-box GUI testing of web apps are not generalized. The second reason is that to verify the use of LLMs as part of the DRL state. The created web application is shown in Figure 2. It has three states, and there are three buttons at the top of the screen. Clicking any of these buttons switches the state, which is then displayed as text at the bottom of the screen. The web app was developed using Vue.js (You, 2024). All buttons are implemented using the button tag and are clickable. The current state display is implemented using a div tag. This app is created as a single-page application, so the URL does not change.

4.2 Exp. 1: Application to Web Apps

First, we trained an automated test agent on the custom web application. Through this training, we

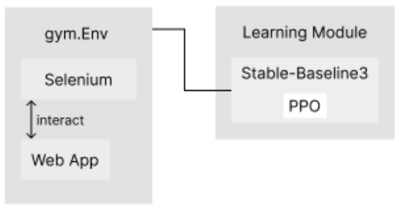

Figure 3: Exp. 1: System Architecture.

verified that utilizing interaction information in web apps accelerates learning and improves accuracy of an DRL model. The system overview is shown in Figure 3. The web app was operated and obtained state through Selenium, and converted into an OpenAI Gym environment (OpenAI, 2024a). We then performed training using PPO with Stable Baselines3.

Selenium is a browser automation framework commonly used in automated testing of web apps. OpenAI Gym is an open-source toolkit provided by OpenAI to facilitate the development and comparison of DRL algorithms. Stable Baselines3 uses OpenAI Gym environments as the interface between the reinforcement learning algorithms and the environment. Therefore, to use the web app as a learning environment, we adapted it to conform to the OpenAI Gym interface using Selenium.

The parameters for training were set as follows. For parameters not listed below, the default values of Stable Baselines3 were used.

Environments Concurrency. 20 envs

Update the Network. Every 512 steps per environment, i.e., every 10,240 steps in total.

Test Cycle. At the end of each epoch.

I will explain the custom environment.

Episode. 1 episode consists of 3 steps. One HTML element is clicked per step. Information is reset at the end of the episode.

Observation. $2 \times n$-dimensional matrix. The first element of each row indicates whether the web element is on the screen, and the second element indicates whether it is clickable.

Action. The index of the clickable HTML element in the HTML element dictionary.

Reward. If the first state is discovered, give $+0.1$; if the second is discovered, give $+0.2$; and if all three are discovered, give $+1.0$.

Because the first column in the action space indicates whether each HTML element is on screen, we determine new states by comparing those conditions. The list of discovered states were reset at the end of the episode.

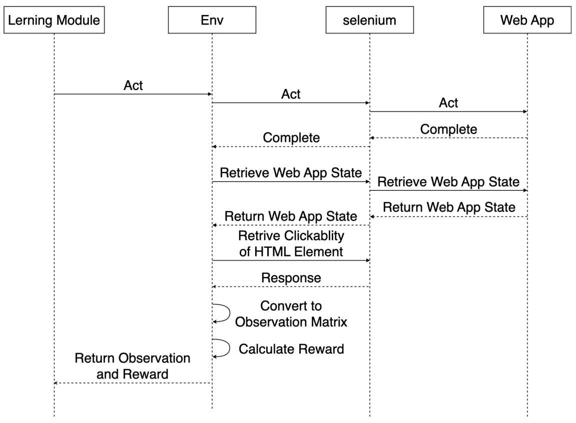

Figure 4: Step Sequence.

In this study, we determined whether an HTML element is clickable by checking if a click event listener is attached. The DOM elements obtained via Selenium do not provide a direct method to determine clickability. Although it is possible to generally infer clickability based on HTML tags like button or div, this method is not straightforward. Buttons can be disabled, making them non-clickable, and div tags can have click events attached, making them clickable. Therefore, we decided that an element is clickable if it has a click event listener attached. Event listeners attached to HTML elements are runtime information that need to be obtained via the browser's developer tools. Selenium provides drivers for each browser, but the ability to use developer tool APIs varies by browser. The Chrome driver has an interface for calling developer tool APIs, but Firefox and Safari do not. Therefore, we used the Chrome driver in this study.

The step-by-step flow is shown in Figure 4. First, the action determined by the learning module is executed on the web application using Selenium. Once the action is complete, the HTML elements of the web app are retrieved via Selenium. Then, event listeners attached to each element are obtained using the Chrome DevTools Protocol. If the element has a click event listener, it is determined to be clickable. Based on this information, the state is updated and the reward is calculated. Finally, the state and reward are returned to the learning module, and various networks are updated.

In this system, the uniqueness of HTML elements was determined using the hash value of the DOM element's outerHTML. While there is a possibility of hash collisions in complex web apps, the web app used in this study only has elements with identical outerHTML. Therefore, the hash value of outerHTML was used as a unique ID for each element.

4.3 Exp. 2: Preliminary Experiment with Random Noise

In this study, we use LLMs to infer whether HTML elements are clickable, but the inference accuracy of LLMs is not 100%. This is because LLMs are probabilistic models. While using high-performance models or refining prompts can improve accuracy, it can never reach 100%. Therefore, we introduced random noise into the clickability portion of the state space in Exp. 1 to determine the level of inference accuracy that can be tolerated. Since LLM inference errors differ from random noise, we used random noise only as a rough guideline. In this experiment, the random noise ratios were set at 0%, 10%, and 20%. We compared learning efficiency and accuracy for each ratio.

4.4 Exp. 3: Inference of Clickability by LLMs

In Exp. 2, we examined how different ratios of random noise affect learning efficiency and accuracy. Having established a benchmark for the accuracy that should be achieved by LLMs' inference, In Exp. 3, we aimed to improve inference accuracy through enhancements to the model and prompts.

In this study, we use inference results as the state in reinforcement learning, so we need to perform inference at each step. Using paid services such as the OpenAI API is not cost-effective, so we built an inference environment using a local LLM. Based on the results of Exp. 2, we set a benchmark and ended the experiment once we found a model-prompt combination that exceeded it. We also manually extracted 40 HTML elements from multiple websites. The target websites are GitHub, YouTube, Count Characters, and LetterFan. To mirror the conditions of actual inference in web apps, we included both clickable and non-clickable elements.

4.5 Exp. 4: Substitution Using LLMs Inference Results

In Exp. 4, we replaced the vector indicating clickability in the state space used in Exp. 1 with LLM inference results for training. Figure 5 shows the system overview for Exp. 4. The LLM inference was implemented as an API accessible on a server independent of the learning module and environment. Since inference takes time, executing it at every step would increase the simulation time. Therefore, we reduced inference time by caching responses, which shortened the total training time.

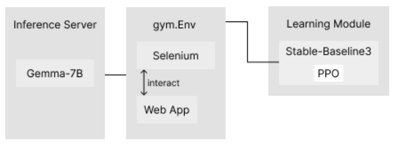

Figure 5: Exp. 4: System Architecture.

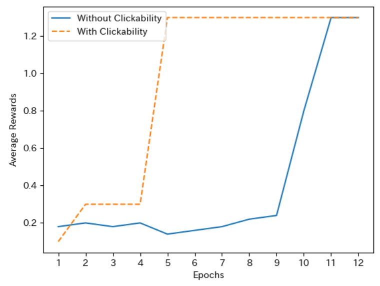

Figure 6: Exp. 1: Average Reward Transition.

5 RESULTS

5.1 Exp. 1: Application to Web Apps

In Exp. 1, we examined changes in learning efficiency and accuracy by utilizing click information. Figure 6 shows the transition of the average episode rewards after each epoch, based on the average reward obtained over 10 episodes. The solid line represents the case without click information, while the dashed line indicates the case with click information. When the clickability of HTML elements was included in the action space, both learning efficiency and accuracy improved compared to when click information was not included. In the reinforcement learning environment used in this study, the maximum reward per episode is 1.3. With click information, it took only four epochs to achieve a reward of 1.3, whereas without click information, it took 10 epochs. These results confirm that, even in web apps, including the interaction information of GUI elements in the state improves learning efficiency and accuracy.

5.2 Exp. 2: Preliminary Experiment with Random Noise

In Exp. 2, we aimed to determine a benchmark for inference accuracy by substituting part of the action space with the inference results of an LLM. Figure 7 shows the transition of the average episode rewards when random noise is included at multiple ratios. Evaluation was carried out at the end of each epoch. Each line represents results with 0%, 10%, or 20% noise. The dotted line indicates 0%, the dashed line

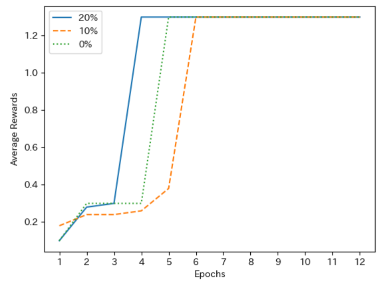

Figure 7: Exp. 2: Average Reward Transition by Noise Ratio.

indicates 10%, and the solid line indicates 20%. With 0% noise, the agent reached a reward of 1.3 by the 5th epoch; with 10% noise, it took 6 epochs; and with 20% noise, it took 4 epochs. Based on these results, in Exp. 3, we aimed for an LLM and prompts capable of achieving at least 80% inference accuracy.

5.3 Exp. 3: Inference of Clickability by LLMs

In Exp. 3, we explored a combination of an LLM and prompts capable of inferring clickability of HTML elements with over 80% accuracy. When selecting the model, we prioritized running on a local machine and inference performance. As a result, we chose google/gemma-7b. Figure 8 shows the prompt, which takes HTML as input and returns 0 or 1 to indicate whether an element is clickable. The prompt structure included sections for [INSTRUCTION] to give commands, [ADVICE] for inference advice, [THINKING STEPS] to outline the thought process, and [EXAMPLES] to provide concrete examples.

As a result, the inference results on the manually extracted validation data averaged 81.5% over five trials. Additionally, the inference accuracy in the custom web application averaged 100.0% over five trials.

5.4 Exp. 4: Substitution Using LLMs Inference Results

In Exp. 4, the LLM and prompts selected in Exp. 3 were used to infer whether HTML elements are clickable, and these results were incorporated into the state space for DRL.

Figure 9 shows the training results. The graph represents the average reward obtained across 10 episodes after each epoch. The solid line indicates the scenario without click information, the dashed line shows the scenario using LLM-based inference, and the dotted line corresponds to the scenario without LLM-based inference. Although it is not clearly visible in the figure, the dashed and dotted lines overlap, indicating no observable difference in learning performance up to the 12th epoch, regardless of LLM usage. In this environment, the maximum achievable reward per episode is 1.3. With click information (whether using LLM or not), the agent reached 1.3 by the 5th epoch, whereas without click information, it took until the 11th epoch. These results confirm that including LLM inference as part of the state attains learning efficiency and accuracy comparable to using direct click information.

```
[INSTRUCTION]
Determine whether the following HTML element is clickable
       or not.
Answer 1 if the element is clickable, and 0 if it is not.
Please answer with 0 or 1, and answer only at the
      beginning of your response. Do not include any
      explanations.
Think with following the steps.

[ADVICE]
- Do not consider child elements in your judgment.

[THINKING STEPS]
1. Check if the given HTML element's tag is inherently
      clickable like <button> tags, <select> and so on.
      If so, the given html element is clickable.
      Therefore return 1, and then finish thinking
      sequence.
2. If not inherently clickable, it become clickable due
      to an attribute. Examples include <a> tags with
      href attributes or <div> tags with onclick
      attributes and so on. If the element is clickable
      due to this, return 1.
3. When elements such as class names or text content
      suggest that they are clickable, they should be
      treated as clickable.

[EXAMPLES]
Here are some examples:
# Example 1:
Given HTML: <a href="https://www.example.com">Link</a>
Answer: 1
# Example 2:
Given HTML: <a>Link</a>
Answer: 0
# Example 3:
Given HTML: <p>Text</p>
Answer: 0
# Example 4:
Given HTML: <div><button>Click Me</button></div>
Answer: 0
# Example 5:
Given HTML: <button>Click Me</button>
Answer: 1
# Example 6:
Given HTML: <div onclick="alert('Clicked!')">Click Me</
      div>
Answer: 1

[ACTUAL]
GIVEN HTML:
```

Figure 8: Prompt for Inference.

6 DISCUSSION

6.1 Improving Accuracy

The aim of this study is to demonstrate that LLM inference results can be utilized as states in DRL. By simplifying the web application to a minimal config-

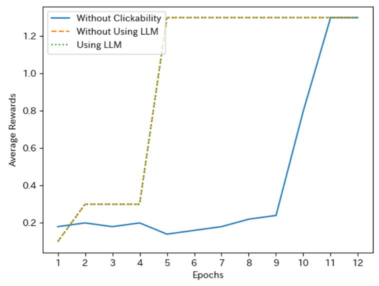

Figure 9: Exp. 4: Average Reward Transition.

uration, we confirmed that substituting with LLM inference is effective. The web app used in this study has a simple structure and uses representative tags like button and div. However, typical web apps are more complex, so the system developed in this study may not achieve sufficient inference accuracy, potentially hindering successful learning. Therefore, when applying this approach to more general web apps, it is necessary not only to refine the LLM and prompts but also to improve inference accuracy through model fine-tuning and partially supervised learning. In addition to improving inference accuracy, enhancing the robustness of DRL is also important. DRL is known for its robustness to noise, although the level of robustness varies among models. In this study, we selected PPO for its stability in training, but by using a model that is more resistant to noise, it will be possible to build a system that is even more robust to LLM noise.

6.2 Generalization

In this study, we conducted training using an original web app as the target. Furthermore, we demonstrated that including the values inferred by the LLM into the state space improves learning efficiency and accuracy. In prior research focusing on mobile apps (Romdhana et al., 2022), 68 open-source apps selected from Su et al.'s paper were used. On the other hand, for web apps, no generalized test set exists. Therefore, to validate general apps, it is necessary to start from the creation of a test set.

One of the factors that improved learning efficiency and accuracy by utilizing the inference results of the LLM in this study is that the learning target was a simple, original web app. When the web app being learned becomes more complex, the number of possible states increases, leading to an expansion of the state space. Since the inference accuracy of the LLM is not 100%, there is a risk that learning efficiency and accuracy may decrease when the state space becomes bloated. Therefore, it is necessary to conduct training on complex web apps to verify how effective the proposed method is. Though not included in this paper due to time constraints, we are currently conducting the experiment.

There are two issues in generalizing the proposed method. The first is whether the values inferred by the LLM and added to the state contribute to improving learning efficiency and accuracy. In this study, we inferred clickability and added it to the state because previous research had shown that adding it to the state improves learning efficiency and accuracy. However, in environments other than web apps, it is often unclear what kind of values contribute to improving learning efficiency and accuracy. Therefore, it is necessary to establish a method to evaluate how important the inferred values are. The second issue is determining the degree of accuracy required for the inferred values added to the state. In this study, preliminary experiments led us to judge that an inference accuracy of 80% or higher is sufficient. However, this threshold is likely to vary depending on the environment and the values to be inferred. Therefore, an evaluation method is needed to set guidelines for inference accuracy.

6.3 Application to Other Fields

Possible applications of this study include black-box environments like consumer games and domains that handle complex environmental information, such as autonomous driving. In consumer games, there is typically no API to obtain environment information; however, by using an LLM to infer environmental data from images, these games can be more easily utilized as DRL tasks. In autonomous driving, which often involves high-dimensional camera footage, extracting summary information from images or videos with an LLM and incorporating it into the state space is expected to improve learning efficiency. For example, by inferring danger levels and including them in the state space, the system can be guided toward safer actions. The extensive knowledge contained in LLMs is anticipated to support learning in these environments.

7 CONCLUSIONS

In this study, we demonstrated that in DRL-based GUI black-box testing for web applications, learning efficiency and accuracy can be improved by using inferred clickability from HTML as part of the state. Clickability is restrictive information that depends on the browser. First, we used click information obtained via the Chrome DevTools Protocol and trained with the Chrome driver, resulting in more efficient learning and higher accuracy. Next, we set up an environment to infer clickability from HTML and added the inference results to the state, which led to equally efficient learning and improved accuracy compared to not using the LLM-inferred values. This shows that in DRL, incorporating LLM inference results as part of the state is effective. As future work, it will be necessary to validate on more complex web applications and to verify other types of information beyond clickability.

ACKNOWLEDGEMENTS

This paper uses ChatGPT's o1-preview and 4o for translations from Japanese to English. This work was supported by JSPS KAKENHI Grant Numbers JP22K12157, JP23K28377, JP24H00714.

REFERENCES

Adamo, D., Khan, M. K., Koppula, S., and Bryce, R. (2018). Reinforcement learning for android gui testing. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, page 2–8, New York, NY, USA. Association for Computing Machinery.

Apple (2024). App review. https://developer.apple.com/jp/distribute/app-review. Access Date: 2024-05-23.

Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE '07)*, pages 85–103.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners.

Cai, L., Wang, J., Cheng, M., and Wang, J. (2021). Automated testing of android applications integrating residual network and deep reinforcement learning.

Carlini, N. and Wagner, D. (2017). Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57, Los Alamitos, CA, USA. IEEE Computer Society.

Chang, Y., Wang, X., Wang, J., Wu, Y., Yang, L., Zhu, K., Chen, H., Yi, X., Wang, C., Wang, Y., Ye, W., Zhang, Y., Chang, Y., Yu, P. S., Yang, Q., and Xie, X. (2024). A survey on evaluation of large language models. *ACM Trans. Intell. Syst. Technol.*, 15(3).

Eskonen, J., Kahles, J., and Reijonen, J. (2020). Automating gui testing with image-based deep reinforcement learning. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 160–167.

Google (2024). Play console help. https://support.google.com/googleplay/android-developer/answer/9859751?hl=en. Access Date: 2024-05-23.

Itkonen, J. and Rautiainen, K. (2005). Exploratory testing: a multiple case study. In *2005 International Symposium on Empirical Software Engineering, 2005.*, pages 10 pp.–.

Kwon, M., Xie, S. M., Bullard, K., and Sadigh, D. (2023). Reward design with language models. In *The Eleventh International Conference on Learning Representations*.

OpenAI (2024a). Openai gym. https://github.com/openai/gym.

OpenAI (2024b). Proximal policy optimization. https://openai.com/index/openai-baselines-ppo/.

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N. (2021). Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8.

Romdhana, A., Merlo, A., Ceccato, M., and Tonella, P. (2022). Deep reinforcement learning for black-box testing of android apps. *ACM Trans. Softw. Eng. Methodol.*, 31(4).

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms.

Sneha, K. and Malle, G. M. (2017). Research on software testing techniques and software automation testing tools. In *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, pages 77–81.

Tao, C., Wang, F., Gao, Y., Guo, H., and Gao, J. (2024). A reinforcement learning-based approach to testing gui of mobile applications. *World Wide Web*, 27(2).

Wetzlmaier, T., Ramler, R., and Putschögl, W. (2016). A framework for monkey gui testing. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 416–423.

Xia, T., Yu, B., Wu, Y., Chang, Y., and Zhou, C. (2024). Language models can evaluate themselves via probability discrepancy.

Yoon, J., Feldt, R., and Yoo, S. (2023). Autonomous large language model agents enabling intent-driven mobile gui testing.

You, E. (2024). Vue.js. https://ja.vuejs.org/.