Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2023

Project Title:	**Automated Web Application Testing Using Deep Reinforcement Learning**
Student:	**Timeo Schmidt**
CID:	**01707530**
Course:	**EIE4**
Project Supervisor:	**Dr A. Abu Ebayyeh**
Second Marker:	**Dr J. A. Barria**

Plagiarism Statement

- I affirm that I have submitted, or will submit, an electronic copy of my final year project report to the provided EEE link.
- I affirm that I have submitted, or will submit, an identical electronic copy of my final year project to the provided Blackboard module for Plagiarism checking.
- I affirm that I have provided explicit references for all the material in my Final Report that is not authored by me, but is represented as my own work.
- I have used ChatGPT v4 as an aid in the preparation of my report. I have used it to improve the quality of my English throughout, however all technical content and references come from my original text.
- I have used MidJourney to generate the graphic on the title page.

Abstract

Graphical User Interface (GUI) testing, despite its time-consuming nature, remains a dominantly manual task due to the combinatorial complexity and infinite unique states posed by user inputs. This thesis presents a novel approach to tackle this challenge by proposing an autonomous reinforcement learning (RL) agent designed to mimic a human tester. The agent learns to interact with a web application's GUI, thereby uncovering potential bugs and faults in the interface. The system uses the Soft-Actor-Critic algorithm and is trained solely on GUI screenshots and a reward signal, allowing it to iteratively learn a policy for effective exploration of the GUI. On average, the proposed system outperforms a Q-learning baseline by 91% and inexperienced human testers by 67% while approaching the ability of expert human testers in certain web apps. Additionally, the system demonstrated remarkable generalisability across different types of web applications and showed the efficacy of transfer learning. This enabled the setup of a custom web app testing agent within just one hour of training on a conventional laptop. By bypassing the need for time-consuming setup and maintenance of predefined test cases, this strategy significantly enhances the efficiency of automated GUI testing. The full project repository, including a standard OpenAI Gymnasium environment, is made publicly available as a supplementary outcome to facilitate reproduction and further research of autonomous web app testing.

Acknowledgements

I would like to express my sincere gratitude to my project supervisor, Dr Abu Ebayyeh. His guidance and motivation greatly contributed to the progress and success of this project. Sincere thanks go to him for passing his depth of experience with contributions in the area of machine learning on to me and guiding me through this scientific process.

I extend my appreciation to my family, who has continually supported my educational pursuits. The invaluable opportunity to go to school and university abroad in the UK, alongside their unwavering encouragement and support was the most important factor to my success to date.

Finally, I extend my gratitude to all who have been, are currently, or will in the future become part of my journey. It has been a privilege to create shared memories and experiences with each one of you. I am profoundly thankful for your influence in shaping me into the individual I am today.

Contents

1 Introduction

1.1 Problem Statement

The central objective of this project is to design and develop a fully automated testing system proficient in detecting faults within the Graphical User Interface (GUI) of web applications. Rather than relying on predefined test cases, the proposed system should utilise reinforcement learning to autonomously interact with the GUI in a manner similar to human users. The aim is to provoke and identify faults that would typically surface during human interaction. More specifically, by observing the state of the GUI and subsequently predicting click coordinates, the system should effectively explore the various states and areas within the GUI application. Hence, the challenge lies in developing an agent capable of learning effective exploration strategies for GUI testing, thereby maximising the detection of potential faults without manual effort. The final deliverable of this project will be a software package equipped with all necessary reinforcement learning algorithms and associated infrastructure, designed to facilitate the training and deployment of the automated testing system on any custom web application.

1.2 Motivation

Software testing serves as an irreplaceable component of the software development lifecycle, contributing significantly towards the production of high-quality software [6]. The inability to detect and correct software faults can have severe implications, even leading to substantial financial losses. It is estimated that in the US alone the cost of poor quality software amounts to an annual cost of over $2 trillion US [31].

The centrality of software testing in the development lifecycle is further emphasised by the considerable investment of time and financial resources it demands. In particular, developers are known to dedicate a large proportion of their available time to software testing. Estimates for this commitment range from 28% [20] up to 50% [11] of the total time of the developers working hours. Despite this large time commitment, almost 80% [20] of the testing undertaken in software development teams remains a manual, rather than an automated effort.

In terms of time devoted to different types of testing, the testing of GUIs is among the most time-consuming types of testing. Existing automation tools that serve the purpose to make testing more efficient face significant barriers to adoption, which include the high associated costs and required time [20] for the setup and maintenance of such tools. Even if successfully implemented, GUI test automation tools are not the 'magic' solution and bear a range of unsolved technical difficulties. The vast number of potential combinations of user inputs and resulting states mean that there are close to infinite paths [42] through a typical GUI application, which prohibits a naive exhaustive testing method. A common approach to automating test case execution for GUIs is to specify test scripts, which outline the step-by-step sequence of simulated user inputs to the GUI. Another common method is to build up a separate model of the GUI with the corresponding state transitions, from which test sequences can be generated automatically [5]. Another long-standing approach is to use record and replay techniques, where an expert first demonstrates a specific interaction sequence, which is then stored to be automatically replayed at test time [8]. One drawback that all these techniques have in common is that changes to the user interface, such as new features or changes to the application layout can disrupt existing test case sequences, necessitating manual updates to the test cases. Furthermore, highly dynamic and non-deterministic GUIs exist, which also pose their own distinct challenges, as exact test acceptance conditions might be unknown [42]. All of these outlined challenges, as well as the potential benefits of efficient test automation, make it an interesting topic of research.

Reinforcement learning (RL), a subclass of machine learning, is a method where an autonomous agent learns to make decisions by taking actions in an environment. The agent learns from the consequences of its actions, rather than from being explicitly taught, progressively adjusting its behaviour to maximise a given reward signal [52]. Reinforcement learning has been the underpinning technology behind many recent high-profile achievements in artificial intelligence (AI). Perhaps most notably, Google's DeepMind utilised deep reinforcement learning to train its AlphaGo program, which famously defeated a world champion Go player [50]. This was a major milestone in AI research, as Go is a complex and strategic board game with an enormous number of possible states, far too many to be solved by any brute force or exhaustive search method. RL methods generally excel in settings that face common challenges with automating GUI testing, such as dealing with large state spaces, where exhaustive sweeps of all the states are impractical [52]. Training reinforcement learning agents to automatically interact with software systems that are designed for human users has been proven successful previously, where reinforcement learning agents were trained to learn how to play ATARI games [40] solely by receiving pixel inputs of the screen and outputting user control inputs. In these experiments, a reward function was provided which would cause the agent to seek a game strategy that would maximise the score of the game. Importantly, the agent learned to outperform even human players solely from being provided high-level signals such as the score and observations of the environment, without any prior knowledge of the rules or strategy of the game. These interesting applications of RL spark the question: could an RL agent be trained to mimic a human tester, independently interacting with a GUI to identify potential faults? The agent could be given access to a GUI application with the goal to learn how to fully independently interact with the GUI to efficiently explore the GUI and uncover potential faults. An agent with this capability could be executed autonomously and any reported faults could be reported back to the developers without any manual effort required in the testing loop.

For this research, the focus has been intentionally placed on web applications as the system under test (SUT). In its early years, the web primarily consisted of simple static pages, serving primarily as a document-sharing platform. Today, it has evolved into a general-purpose application environment [53]. Examples that demonstrate the capability of modern web applications include the browser-based version of the popular photo editing software Photoshop [4], dynamic video content sharing platforms like YouTube [58], and even complex engineering CAD software like AutoCAD [7]. Unlike traditional binary applications that require download and installation, web applications like these can be deployed and accessed globally instantly, favouring practices like continuous deployment (CD) [48]. CD is a software engineering approach where new software versions are typically tested and deployed automatically on a continuous rolling basis which shortens development cycles. This quick pace underscores the need for robust automated testing strategies, as more frequent deployments demand frequent test executions. The demand for automated tests for web applications hence motivates the development of better test automation for this type of GUI application.

1.3 Project Objectives

A GUI testing system is required to perform two separate tasks. Firstly, it must provoke faults in the GUI through simulated user interaction, which should expose as many possible meaningful states of the GUI. The second task consists of the actual identification of faults within the discovered GUI states. Faults can range from simple errors to application crashes [37] [38] to more challenging issues such as visual and layout bugs [34]. This thesis will focus on the first task, which is the efficient generation of simulated user interactions with the interface and limit the fault detection to easy-to-detect, yet common JavaScript exceptions [44]. In summary, the ultimate goal of this project is a fully autonomous system that could be deployed to a desired web application deployment pipeline, where it autonomously emulates a human user to uncover and report faults back to the developers. To achieve this, the following high-level objectives have been identified:

- Conduct a comprehensive review of the existing automated GUI exploration methods, understanding their capabilities and identifying any limitations.
- Formulate the autonomous exploration of GUIs as an RL problem
- Develop a compatible RL environment that enables reinforcement learning agents to interact with the GUIs of browser-based web applications.
- Investigate various possibilities for defining suitable reward functions that encourage effective learning, and subsequently implement them.
- Explore, test and iteratively refine the architectural choices of the proposed autonomous testing system.
- Implement baseline models from existing literature. Establish a human baseline by collecting GUI interaction data from human testers.
- Evaluate the proposed system and identify the strengths and weaknesses of the proposed method.

1.4 Report Structure

Chapter 1: Introduction This chapter gave a high-level problem statement, provided motivation for carrying out the research and finally formulated a list of objectives to be addressed.

Chapter 2: Background The next chapter gives an overview of the existing literature and theoretical perspectives relevant to the topic. It explores previous studies, models, and concepts that lay the foundation for this research.

Chapter 3: Requirements Capture The third chapter captures the project requirements. It elaborates on the criteria that the solution must meet and specifies the technical and user requirements of the system.

Chapter 4: Analysis and Design The fourth chapter discusses the analysis of the collected requirements and how they informed the design of the solution. This chapter also presents the overall architecture of the proposed system.

Chapter 5: Implementation The fifth chapter covers the implementation of the proposed design. It presents the process, technologies used, and key challenges faced during the development phase.

Chapter 6: Experimental Setup The sixth chapter describes the research questions and the experiment setup aimed to gather empirical data on individual design choices and system performance.

Chapter 7: Experiment Results The seventh chapter presents the results, interpretation and discussion of the experiment outcomes.

Chapter 8: Evaluation The eighth chapter evaluates the system against the initial requirements and objectives. It assesses the implications of the results, the limitations, and the potential impact of the system.

Chapter 9: Conclusions and Further Work The final chapter concludes the report, reflecting on the findings and their implications. It also presents potential avenues for future research and development.

Appendix The Appendix includes the source code and a user guide to enable the use of the developed system.

2 Background

This chapter gives an overview of the existing literature and theoretical background relevant to the topic. It will start off by laying out the different types of faults that are commonly present in a GUI application. It will then proceed with outlining the traditional testing techniques that are employed in the industry to automate the detection and prevention of such issues. An overview of the testing frameworks specific in the context of browser-based web applications will follow. Afterwards, the machine learning background, which includes an introduction to reinforcement learning as well as a high-level overview of relevant RL algorithms will be provided. Alongside this, Convolutional Neural Networks will be introduced. Finally, the directly related work of reinforcement learning applied to the testing of GUIs will be laid out.

2.1 Types of GUI Faults

To fully understand the potential benefits of a testing system, it is important to understand the potential faults that are likely to be introduced through the absence of effective testing mechanisms. In this report, the term fault is used as a general term that includes bugs, errors and layout issues. From the reviewed literature, the most common types of faults can be divided into two separate categories: logic faults and appearance faults [22].

The first class of faults, namely logic or functional faults are a group of errors where the application under test does not behave as expected. The majority of dynamic interactions in a browser are facilitated through the use of JavaScript [41]. Therefore, faults in the GUI frequently materialise as JavaScript errors [44] which may cause the web application to become unusable, cause loss of user data or cause the application to 'hang'. As will be discussed later on, the detection of functional faults is straightforward, as such errors show up in the JavaScript logs.

The second class of faults are visual faults. These are errors in the GUI layout and its components, and different studies derived varying classifications. The two related studies Owl Eyes [34] and Nighthawk [33] classify visual faults into 5 categories. These are component occlusion, text overlap, missing images, NULL values displayed, and blurred screens. They arrived at this classification by manually finding and classifying 4470 faults in the 'Rico' data set [15], which contains screenshots of user interfaces. The detection of this second type of fault category proves to be significantly more challenging, as appearance-based faults tend to cause no detectable side effects, such as error logs, and tend to only cause visual changes and discomfort to a user.

As this project will focus exclusively on the detection of JavaScript exceptions, it is important to acknowledge the diversity of possible fault types beyond this, to allow for a thorough understanding of the system's limitations later. The detection of more advanced fault types is beyond the scope of this project and the addition of more sophisticated fault types will be left for future research.

2.2 Traditional Techniques for GUI Testing

This section serves the purpose of giving an overview of the existing traditional techniques that are commonly used in the industry today to test GUIs. By understanding the strengths and limitations of current GUI test automation, this project will be able to work towards overcoming problems with traditional methods.

2.2.1 Scripted Tests

Scripted GUI testing is a method of testing a software application by following a pre-defined test plan. The test plan contains sequential actions describing the interaction steps with the GUI. Such steps could be clicking on specific GUI elements, scrolling or inputting values into text fields. The test script can then check for certain expected properties or states in the GUI to determine if the test passes or fails. Generally, such test cases are manually defined and an example is given as pseudo-code:

```
Input: Web Application
$LaunchBrowser()$
$NavigateToLoginPage()$
$EnterUsername($"testuser"$)$
$EnterPassword($"testpass"$)$
$ClickLoginButton()$
$welcomeMessage \leftarrow GetWelcomeMessage()$
if $welcomeMessage ==$ "Welcome, testuser" then
    set $test\_passed$ to $false$
else
    set $test\_passed$ to $true$
end if
$CloseBrowser()$
Output: $test\_passed$
```

Figure 1: Example Pseudo-Code for a Scripted Test that tests a Login Process

Scripted test cases are a reliable method of testing key aspects of a GUI, however, they entail a range of drawbacks. One of the main disadvantages is the high cost to set up, as they require significant manual effort to implement. Additionally, the test cases have to be constantly adapted to changes in the GUI which adds a high cost of maintenance. Furthermore, scripted GUI testing may not represent actual user interactions, as it only tests the functionality based on pre-defined scenarios, which may not reflect the ways a real user would interact with the application. The automated system developed in this project aims to overcome these issues, as it requires no manual setup time and aims to behave more naturally by learning human-like behaviour when interacting with the GUI.

Another issue with traditional scripts is the limited application to complicated GUI interfaces, such as moving maps etc. Here, it may not be possible to pinpoint the coordinates and type of elements in advance, as the content of the GUI is highly dynamic. One study by Macchi et al. [35] developed a method that would identify and check for the existence of elements on a GUI visually by using computer vision algorithms and machine learning. Although this does improve the flexibility and range of applications, it still suffers from the existing limitations of scripted tests. Another similar study [59] also developed a technique to visually identify and interact with GUI elements based on template matching. In light of these limitations, the system developed in this project will pose the advantage that it aims to learn dynamic interactions with the GUI, without relying on any prior assumptions which enables the system to learn an adaptive exploration strategy.

A further group of techniques that fall under scripted tests are model-based techniques. Here, a human tester first creates a model of the application views and the transitions between the views that take the shape of a finite state machine. From this finite state machine, test scripts are automatically generated. One study [5] used finite state machines to generate test scripts specifically for web applications. This study however only focused on transitions between web pages and not on dynamic interactions on a single web page. Another study [36] overcame this limitation and focused on dynamic interactions within a single web page by also relying on a model-based approach. Both studies suffer from the limitation that a model needs to be set up manually, which is very time-consuming. Furthermore, continuous updates to a web application also require ongoing manual updates of the model which is very time-consuming and impractical for many applications. As before, the proposed method will pose the benefit of not requiring the manual definition of such models, thereby overcoming their associated limitations.

2.2.2 Record & Replay

Record and Replay is an alternative method of creating test scripts for GUI testing that can be used as an improvement over manually scripted tests. Although Capture-Replay-Tools have existed since the 1960s [8], they are still common practice in the industry and most state-of-the-art GUI test tools have a capture and replay functionality.

In a sample run-through, a human tester would interact with the software application's GUI, performing actions such as clicking on buttons, inputting values, and navigating through menus. These interactions would be recorded and saved as a test plan, which can then be replayed at a later time to automatically test the functionality of the application.

The main advantage of Record and Replay is that it is faster to define complex and long interactions, as it eliminates the need to manually write test scripts. It also doesn't require a skilled human tester as oftentimes no scripting knowledge is required [39] and the record & replay tools are simple and intuitive to use. However, the technique may still suffer from the limited range and realism of the scenarios that are tested. The problem with high maintenance costs and effort also remains, as changes in a GUI may still require manual edits or entirely new test recordings from a human tester. The adaptive and autonomous nature of the proposed method aims to overcome these issues.

2.2.3 Regression Testing

Regression testing is a type of software testing that ensures that parts of software remain working after changes have been made. It is typically done by re-running existing test cases to ensure that they are passing after an update. If a test case fails, this is called a regression.

Regression testing has also been adapted to test for visual differences in a GUI after an update. One approach is to generate various screenshots of the GUI and then use an algorithm to compare the screenshots to the previous version of the software. This can help to identify visual differences between the two versions of the software, such as changes in layout, colour, or font size. The advantage of this approach is that it can reduce the amount of manual review required after an update. Instead of having to manually review the entire GUI, only a much smaller subset of screenshots or areas in the app require manual review by a human. This can save a lot of time and cost, and help to ensure that any visual regressions are caught and fixed early on. It is worth noting that GUI regression testing is not a complete replacement for manual testing, but rather a complementary technique. One company that offers this as a commercial service to test web applications is BrowserStack, through their product Percy.io [26].

The algorithm required to check for differences in a GUI is not trivial, as a simple pixel-level difference will not work in cases where dynamic content is used, such as changing images or text on news pages or social media networks. Here, structural differences still need to be identified and literature exists that develops techniques to identify such structural changes. One study by Ivanova et al. [29] developed an algorithm based on simple machine learning models. The paper evaluated the K-means algorithm and the Mean-Shift algorithm to effectively segment structural and layout components on a web page.

In summary, regression testing may reduce the amount of manual effort, however, the review of identified regressions remains a manual effort. With regard to the proposed method in this project, regression testing may be a complementary, rather than an alternative method. The proposed method will be capable of producing test cases that can afterwards be used to identify regressions.

2.3 Web Testing Frameworks

GUI testing frameworks are an essential tool for test automation, as they provide many pre-built features and functionalities which are common to different types of testing. One of the earlier GUI testing frameworks that have been used and developed in academia is the framework GUITAR [43]. It offers many different types of testing and works across multiple platforms. Since then, more modern frameworks have been developed and two frameworks that are specialised in browser-based web app testing will be presented here. The first framework is the well-known 'Selenium' framework [13], which has over 170k users on GitHub and the second framework is 'Cypress' [27], with well over 600k users. Both are open-source frameworks that can be used for free.

Selenium is a widely-used open-source testing framework that enables the automation of web browsers across multiple platforms. It supports multiple programming languages such as Java, Python, C#, Ruby, and JavaScript. Selenium provides an API that enables interaction with different web browsers in a consistent way.

Cypress is a newer open-source testing framework that is specifically designed for testing web applications. It is built on top of JavaScript and provides a modern, easy-to-use API for interacting with web browsers. Cypress provides many interesting features and is particularly well suited for end-to-end testing, as it allows a simple set-up of full end-to-end test suites.

It is important to note that the usage of Selenium is significantly more flexible, as it doesn't specialise in test automation but instead in the programmatic interaction with a web browser in general. Furthermore, while Cypress only supports JavaScript, Selenium supports multiple programming languages to facilitate interaction with a web browser. Specifically, the integration with Python is attractive as a range of powerful machine-learning libraries are available. It was decided to choose the Selenium framework for the implementation of the systems in this project.

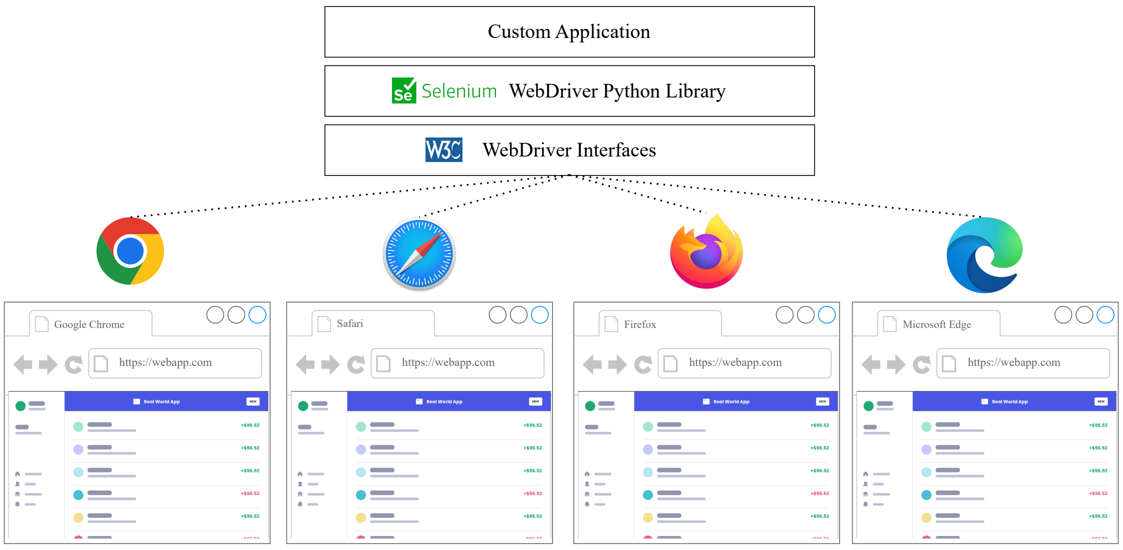

Figure 2: Overview of the software stack that enables programmatic interaction with a web browser. At the bottom, the web application is running in a given web browser (i.e.: Chrome, Safari, Firefox, Edge) and their corresponding WebDriver interfaces are sitting on top. These WebDriver interfaces are then accessed via the user-friendly Selenium WebDriver Python library.

To understand how Selenium fits into the whole software stack, a brief overview is provided: The World Wide Web Consortium specifies a standard WebDriver [55] interface, which is typically implemented for all major web browsers. For instance, the market-leading browser Google Chrome [51] offers the compatible ChromeDriver [12] as the implemented WebDriver interface. The WebDriver interface enables rich programmatic access to a web browser. Functionality such as navigating to web apps, performing clicks, manipulating browser cookies or taking screenshots is made available in a standardised, browser-independent interface. The framework Selenium provides the 'Selenium WebDriver', which exposes the

WebDriver interface in the form of a user-friendly Python library.

2.4 Machine Learning

Machine Learning (ML) is a subset of artificial intelligence (AI) that provides systems with the ability to automatically learn, improve, and make decisions from experience without being explicitly programmed to do so [46]. An important sub-class of ML is supervised learning, where algorithms learn a potentially complex relationship of inputs and outputs from labelled pairs of training data with the objective to afterwards predict outputs on previously unseen inputs. This approach typically requires large volumes of labelled training data, which may not always be readily available. Another algorithm sub-class of ML which overcomes this issue is Reinforcement Learning (RL). In RL, an agent learns from the environment by interacting with it and receiving rewards or penalties for the actions that it performs [30]. Unlike supervised learning, this approach does not require large volumes of labelled data sets.

The enabling technology that has driven the capabilities of ML methods are Artificial Neural Networks (ANN) [57]. ANNs are computational models inspired by the human brain's interconnected neural structures. These networks are composed of multiple interconnected artificial neurons or nodes, which are organised into layers: an input layer to receive the data, one or more hidden layers for computation, and an output layer for the final decision or prediction. Each artificial neuron applies a weighted sum of inputs and a nonlinear transformation, allowing the network to model nonlinear patterns in the data. The weights in the network are learned through optimisation procedures, typically involving backpropagation and gradient descent. In particular deep neural networks (DNNs), which are ANNs with more than one, oftentimes many more hidden layers further boost the performance and versatility of ML models [21].

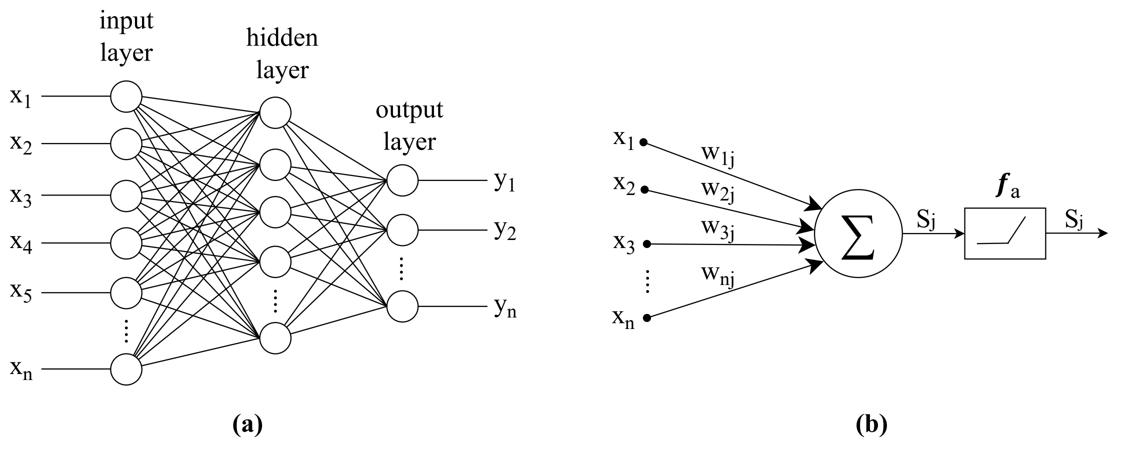

Figure 3: Diagram depicting the structure of a simple ANN adapted from [9]. **(a)** shows a simle ANN architecture, with one input layer, one hidden layer and one output layer. **(b)** shows a single perceptron from the left architecture in detail, where a weighted sum of inputs is passed through an activation function.

2.4.1 Reinforcement Learning

RL is a domain within the broader field of ML that centres on an agent learning to make optimal decisions through interactions with its environment [52]. In the context of RL, an *agent* refers to the learner or decision-maker that interacts with the environment, seeking to achieve a certain goal through its actions. On the other hand, the *environment* refers to everything outside the agent with which it interacts, providing the agent with feedback (rewards) and presenting new situations (states) in response to the agent's actions. The RL problem is formalised using a Markov Decision Process (MDP) [52]. MDPs formalise sequential decision-making problems where actions influence not just immediate rewards, but also future states and subsequent rewards. The MDP is best explained with the help of Figure 4.

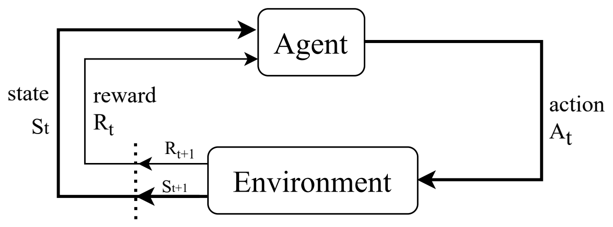

Figure 4: The agent–environment interaction in a Markov decision process, re-created from [52].

The state of the environment S_n as well as the associated reward is passed to the Agent, which produces an action A_t based on the observed state [52]. This action in turn modifies the state of the environment, which produces a new reward R_{t+1} and a new state S_{t+1}. The agent continuously uses the scalar-reward signal R to update the parameters θ of its policy π_θ, which is the strategy or rule that determines the agent's actions given the observed environment state. This policy maps the observed state of the environment to actions that the agent should take in a given state. As the agent continues to interact with the environment over time, it iteratively refines this policy based on the reward signal R, with the ultimate goal of maximising the sum of these rewards over time. In RL, interaction sequences with the environment can be limited to a maximum length, known as the horizon h, after which the environment is reset and a new interaction sequence is started. The weighted sum of the rewards is known as the return, which is discounted such that future rewards are given less weight over short-term, immediate rewards. It is written as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1}$$

where γ is an adjustable hyper-parameter, $0 < \gamma \leq 1$, called the discount rate.

The agent's task, in the reinforcement learning paradigm, is to learn the optimal policy π^* that maximises the expected return G from each state, S, thereby effectively solving the decision-making problem it's faced with.

2.4.2 RL Algorithms

There is a large number of different RL algorithms in existence, hence it is important to gather a high-level overview of this space. Figure 5 provides a loose taxonomy of various state-of-the-art RL algorithms.

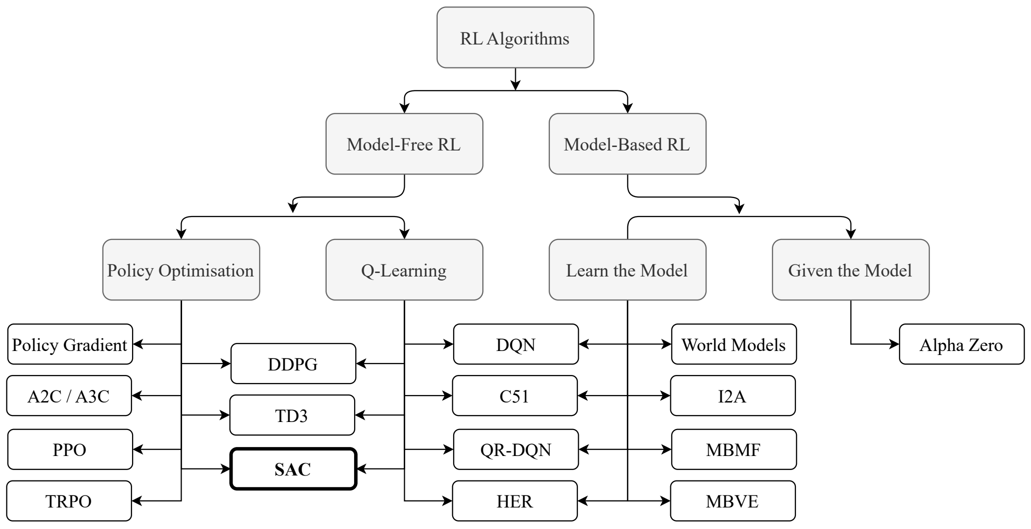

Figure 5: A loose taxonomy of RL algorithms re-created from [2].

The first important feature of RL algorithms is whether they are a model-based or a model-free method. In brief, model-based algorithms are able to build up a model of the environment, hence allowing them to plan ahead and predict the next state and expected rewards, thereby increasing their sample efficiency [2].

Model-Free RL algorithms, on the other hand, do not assume any knowledge of the environment or its model. As models are rarely available for real-life applications, model-free methods tend to be applied more frequently and are more popular [2]. As the problem that is discussed in this thesis is incompatible with model-based methods, the following will focus entirely on model-free methods.

Within the realm of model-free methods, one further sub-classification can be made into Policy-Optimisation and Q-Learning methods [2]. In Policy-Optimisation, the parameters θ of the policy π_θ are directly optimised by gradient ascent or descent through experiences that are collected by interacting with the environment [2]. Q-learning-based methods on the other hand follow a different approach that makes use of a value function. A value function estimates the expected return of a policy for a given state, given that the agent will proceed to forever act according to its policy. The objective here is to find the optimal action-value function, $Q^*(s, a)$, which yields the value of action a for an environment state s given that the agent will continue to act under the optimal policy π^*. It is apparent that the policy is not directly optimised and is only implicitly defined by the relationship:

$$a(s) = \underset{a}{\operatorname{argmax}}\, Q_\theta(s, a), \tag{2}$$

which states that the action a for a given policy, parameterised by θ is the one which leads to the maximum subsequent return for the current state s. With this background, it is now possible to understand a concrete algorithm variant.

Soft Actor-Critic (SAC) [23] is an advanced RL algorithm that falls between policy optimisation and Q-learning methods. It uses a stochastic policy and a separate target value network to improve stability. SAC employs a Gaussian distribution for action selection and applies a squashing function to ensure the actions remain within bounds. It uses entropy maximisation to stabilise training, which is particularly beneficial for complex tasks where hyperparameter tuning is challenging. SAC is known for its robust performance across various tasks, outperforming both existing on-policy and off-policy methods in many challenging scenarios. All these properties make SAC an attractive choice for this project. The implementation intricacies of the SAC algorithm are covered in the later sections of this report.

2.4.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) have emerged as the default model for extracting insights from data with grid-like topology, such as images [21]. Rather than directly feeding raw pixel values into an ANN, the CNN helps to translate the image to a rich feature-oriented representation that captures the significant aspects of the image.

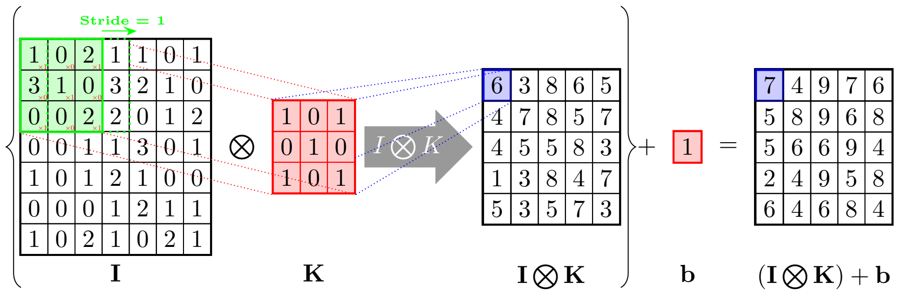

Figure 6: Convolutional operation of 3×3 kernel applied to a 7×7 input [1]

The functioning of a CNN is rooted in a mathematical procedure known as a convolution. An image, shown as a square matrix where each entry matches a pixel's brightness, is combined with a filter to produce a new output matrix. The filter, also referred to as a kernel, is a small square matrix comprising learned weights. By sliding this kernel across the image and taking a weighted sum at each step, the convolution is obtained. The mathematical operations are visualised in Figure 6. A bias term is added and the result is passed through an activation function such as ReLU (Rectified Linear Unit), which produces the final output that is passed to the subsequent network layers. After multiple convolutional layers, the final output of the CNN is a feature map of a more manageable dimension, which captures the meaningful information included in the original input image. This feature map is flattened from a 2D matrix to a 1D vector and then passed into the fully-connected ANN.

Besides the size of the kernel, another parameter to consider is the stride, which is the offset that is applied when sliding the kernel across the input. Additionally, many CNNs rely on pooling layers for downsampling, however as the simple architecture in this project does not include this operation, its discussion will be omitted.

In the context of this thesis, the ability to effectively extract relevant features from image inputs is particularly useful as screenshot images will be used to represent the state of the GUI. In this setting, CNNs will be an effective method to extract features from these screenshots that allow the RL agent to learn more efficiently.

2.5 Reinforcement Learning and GUI Testing

The usefulness of RL methods in the context of GUI testing has been previously demonstrated by a range of studies. This section will provide a brief overview of the literature that has applied RL methods to GUI testing.

A common theme across the literature is the application of RL techniques, specifically Q-learning, in the automated testing of Android mobile applications. Autodroid [3] is a proposed method for automated GUI testing of Android applications using Q-learning, where an RL agent interacts with the application through trial and error to identify actions that are likely to discover unexplored states and revisit partially explored states. This method achieved higher average code coverage across eight Android applications compared to random test generation. Similarly, another paper presented QDroid [54], an automated GUI testing tool for Android applications that uses a Deep Q-Network and semantic analysis of the GUI. The tool identifies the semantic meanings of GUI elements and uses them as input to a neural network. This network, through training, approximates the behavioural model of the application under test. The tool aims to generate test cases that test hard-to-reach states of the application, cover the most code possible, and reveal faults, all within a limited amount of time. A similar study proposed the method DinoDroid [60], which used Deep Q-Learning to find interactions which would lead to unexplored and/or partially explored states. Likewise, this study also claimed improvements over random exploration. Another study [37] also achieved good GUI exploration, again using Q-learning. Yet another paper [47] also developed a Q-learning method to generate GUI interactions and managed to achieve an average code coverage of >80% after only 500 steps of training.

It is obvious that most research that applied RL methods to explore user interfaces utilised a Q-learning approach. Despite the success of these methods, Q-learning approaches face certain drawbacks. The first difficulty is the requirement to 'hand-pick' a discrete set of features to represent the observation space. This makes these methods difficult to generalise between different types of applications, such as between Android and Web Apps. Using screenshots, as done in this project overcomes this difficulty and resembles the observation as the visual information on the screen, similar to human perception.

Furthermore, the Q-learning approaches outlined here all utilised a discrete action presentation, meaning that in any given GUI state, the agent was asked to choose one action from a fixed range of allowed actions. The difficulty here arises, as the number of allowed actions may differ between states, which makes it difficult to find a universally suitable action space representation that works across GUI states and across applications. Additionally, GUI interactions extend beyond only clicks on elements [14] and modern GUIs often use more sophisticated user interactions, such as scrolling, swiping, dragging or long-clicking/holding. Such interactions tend to be better represented with a continuous-valued action space, as they require action parameters. For example, an agent that is capable of mimicking an interaction such as swiping will not only need to predict the action "swipe" but will also need to predict the associated action parameters in the form of "swipe from point A to point B at speed V". Therefore, continuous action spaces offer more potential to model such complex actions.

The prior literature considering continuous action spaces in GUI testing is limited. One relevant paper titled "Automating GUI Testing with Image-Based Deep Reinforcement Learning" [18] presented a novel approach to GUI testing using image-based deep reinforcement learning (DRL). The authors propose a method that utilises an RL algorithm (A3C), where the observation is an RGB screenshot image of the current view of the GUI. The method predicts a continuous probability distribution for the predicted click coordinate in a continuous space. Pairing this continuous distribution with the element boundaries and the policy output values, the clicked element was predicted. Additionally, the approach utilised a CNN for efficiently extracting features from the image and used an LSTM to keep a memory of previously visited states. The method outperformed Q-learning-based and Random approaches by a large margin. Despite the good performance, the agent training took multiple days on a GPU, before it could be deployed on a real-world product and the source code was not publicised.

Table 1 provides an overview of the discussed methods with their key characteristics. Similarly to [18], the proposed method will also rely on a CNN, continuous actions and address web apps, however, the proposed method innovates on various architectural choices, including the reward signal and choice of RL algorithm among others.

			AUT			Observation		Action	
Study	**Algorithm/ Policy**	**Reward Signal**	Android App	Desktop App	Web App	Element-wise	Screenshots	Discrete	Continuous
Adamo et al. [3]	Q Table	$1/x$	✓			✓		✓	
Vuong et al. [54]	DQN	Δ Elements	✓			✓		✓	
Zhao et al. [60]	DQN	Code Coverage	✓			✓		✓	
Degott et al. [14]	MAB	Visual Change	✓			✓		✓	
Saber et al. [47]	DQN	$1/x$		✓		✓		✓	
Eskonen et al. [18]	A3C/CNN	URLs Visited			✓		✓		✓
(Proposed method)	**SAC/CNN**	**Δ New Elements**			✓		✓		✓

Table 1: Summary of properties of related studies that are using RL methods for automated GUI testing. Abbreviations: Deep Q Networks (DQN), Multi-Armed Bandit (MAB), Reward inverse to number of times a state was visited $1/x$, Change in (Δ), Application Under Test (AUT).

3 Requirements

This section outlines the high-level requirements of the deliverables that are produced in this project. The requirements will be broken down for the individual subsystems and will set the goals for the subsequent implementation section.

The high-level goal of this project is to produce a fully autonomous testing system for web applications. The objective of the system is to find interaction sequences within a given web application which induce faults, specifically JavaScript Exceptions. The output of the system are the discovered faults and the interaction sequences that provoked them in the form of a test report. This test report will be returned to the web app developers for investigation and fixing.

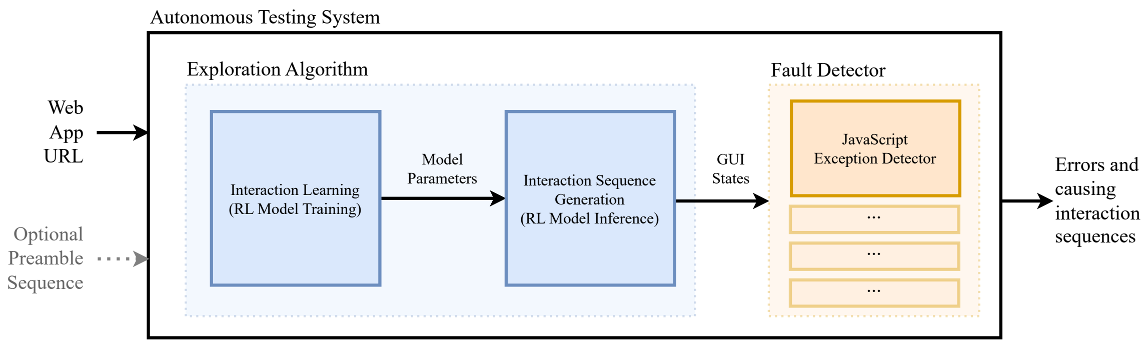
Autonomous Testing System
Exploration Algorithm
Fault Detector
Web App URL
Optional Preamble Sequence
Interaction Learning (RL Model Training)
Model Parameters
Interaction Sequence Generation (RL Model Inference)
GUI States
JavaScript Exception Detector
...
...
...
Errors and causing interaction sequences

Figure 7: High-level overview of the testing system showing system inputs and outputs as well as internal sub-components.

The testing system requirements can be broken down further into individual components. Figure 7 gives a high-level overview of the testing system including its sub-systems. There are two major subsystems: The first subsystem is the exploration algorithm, responsible for learning and then simulating user-like interactions with the GUI. The second subsystem is the fault detector, which will take the unique GUI states that were discovered through the interactions as input and potentially detect an array of faults. It can be seen that this is a full end-to-end system, which only requires the URL where the web application can be accessed and outputs faults that are present as the output, without requiring any human efforts in between.

3.1 Exploration Algorithm

The Exploration Algorithm is a fundamental component of the autonomous testing system, with its function being the discovery of as many unique states within the GUI as possible. This requirement ensures broad coverage of the web application, thus increasing the probability of uncovering interaction points that may conceal potential faults. The development of this exploration algorithm will be the research focus of this project. The following functionality is required for the successful delivery of an exploration algorithm:

- Development of an interface to the web application under test, which would need to provide a compatible interface for the RL agent to observe and manipulate the GUI state.

- Development of an RL agent capable of autonomously learning through interaction with the GUI, thereby generating parameters (including policy weights and biases) that can be repurposed at inference time.
- Implementation of a GUI interaction sequence generator that utilises the previously obtained parameters to generate extensive interaction sequences with the GUI. Generating interaction sequences should not require re-training and allow continuous testing of the system.

There are a range of desirable characteristics for the RL exploration algorithm:

- Fast and sample-efficient training of the algorithm: This implies that the RL agent should be trained using minimal learning interactions in the least possible time, thereby reducing both the computational and temporal resource footprints required for setting up the testing system. This will broaden the accessibility of the system, catering to a wider user base while making the system more user-friendly.
- The developed method should be as versatile as possible and be applicable across different applications without entailing any assumptions or dependencies that limit its application to a certain type of web application. This characteristic will also promote the generalisability of the developed methods in this project.

3.2 Fault Detector

The Fault Detector forms the second major subsystem of the autonomous testing system. Its purpose is to identify faults that may be present in any of the states that are visited by the exploration algorithm. Initially, the Fault Detector should be configured to detect and handle JavaScript Exceptions, a prevalent class of faults in web applications. However, the design of the Fault Detector must allow easy extensibility to allow future inclusions of more diverse fault detection types.

The simplicity of the Fault Detector does not diminish its importance. As the component that identifies faults instigated by the Exploration Algorithm's interactions, it plays a pivotal role in realising the final deliverable of this system: a comprehensive report of discovered faults and the specific interaction sequences that led to them. This output would provide invaluable insights to web application developers, enabling them to pinpoint potential problems in their applications, reproduce them, and ultimately rectify them.

4 Analysis and Design

This section delves into the analytical and design phases involved in creating the autonomous testing system, providing insight into the design decisions, such as important design trade-offs. The system is broken down into multiple layers, with each layer representing a group of related functional modules. Figure 8 visualises the different layers and the constituent sub-modules with arrows indicating the information flow at a high level.

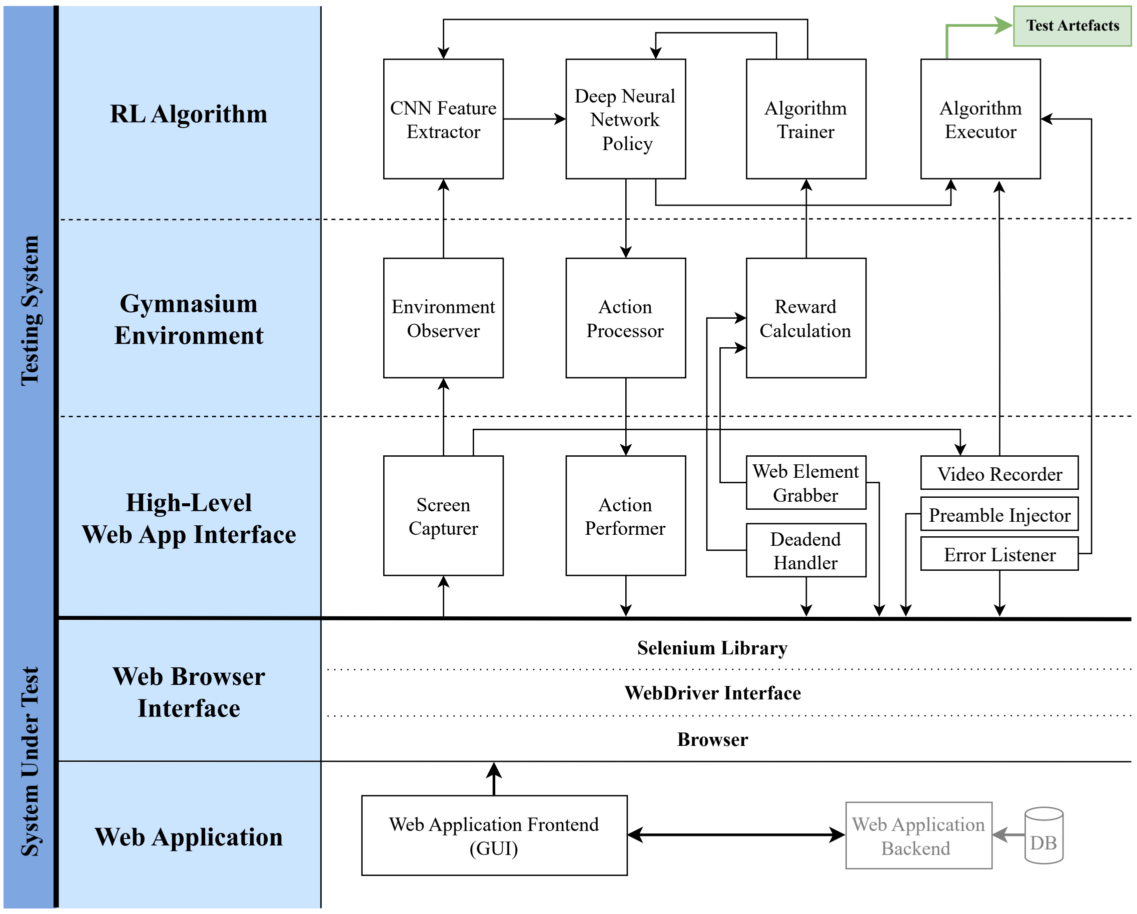

Figure 8: Detailed system design overview showing the different layers and the major sub-components therein. Arrows give an intuition about the flow of information.

The following subsections will discuss the design of each of the layers individually, starting from the bottom and working up to the highest layer. First, the web application and the browser interface, as well as the custom higher-level abstract interface will be described. The gymnasium environment, which builds on top of these interfaces will subsequently be introduced. Finally, a high-level introduction to the design process of the RL algorithm layer is provided. Acquiring a comprehensive understanding of the system architecture and appreciating the associated design choices will establish a solid foundation for the upcoming implementation section, which will illuminate the technical implementation nuances in detail.

4.1 Web Application

As can be seen in Figure 8, the web application (web app) is the lowest-level constituent of the system overview provided in this chapter. Although the design of the web app is not the purpose of this project, it is valuable to have a high-level understanding of the architecture of a modern web app before designing a testing system for it. Web apps are characterised by offering a more dynamic and interactive experience to the user, compared to traditional websites.

The frontend, also known as the client side, is the user-facing component of the web app responsible for rendering the GUI and managing user interactions. Built using technologies like HTML, CSS, and JavaScript, the front end processes certain user inputs directly, providing real-time visual updates to enable a responsive user experience. Form validations and dynamic visual updates and animations typically occur directly in the frontend. Such interactions can happen without requiring a full reload of the web page, allowing for more responsiveness without the need to wait for network requests.

The backend, also known as server-side, operates behind the scenes, handling vital and sensitive calculations or functionalities to maintain application integrity. It's responsible for server-side logic, database interactions, complex calculations, and application rule enforcement which cannot be left to the client side. Communication between the frontend and backend typically takes place through REST API calls, typically transferring data in JSON or XML formats.

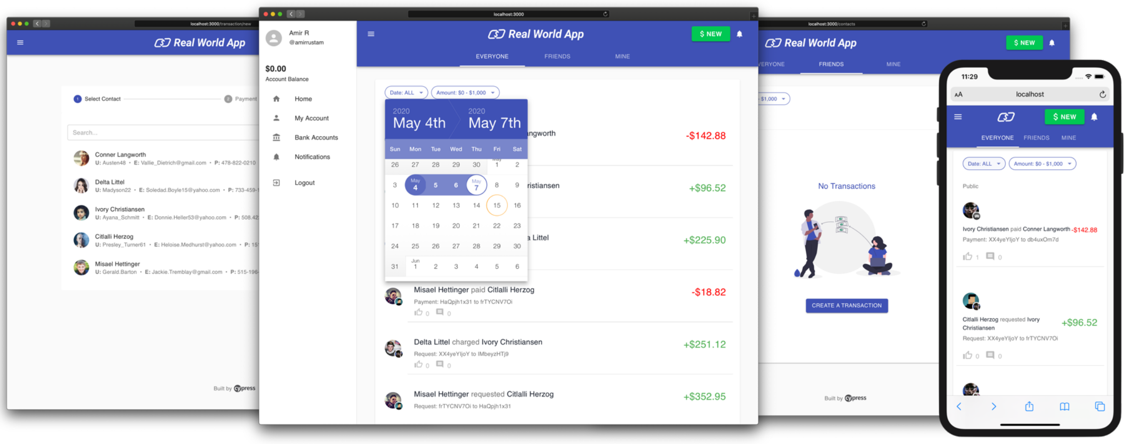

Figure 9: Example screenshots of the Cypress Real World App [25].

In order to evaluate the effectiveness of the autonomous testing system, the Cypress Real-World App (RWA) [25] serves as a suitable benchmark. This open-source dummy web app, licensed under the MIT license, emulates a modern payment application similar to online banking web apps like Venmo. Screenshots of the GUI and highlighted GUI elements can be seen in Figure 9. The RWA is a full-stack web app, embodying both frontend and backend components described previously. The frontend provides a rich user interface, featuring sophisticated components such as a date-picker and a user authentication system, enabling users to sign up for accounts, log in, and interact with the application. The backend complements the frontend by executing server-side operations, allowing users to perform complex tasks such as adding a bank account, sending, and receiving money between friends. This holistic application structure encapsulates various real-world functionalities, making the RWA an excellent candidate for algorithm evaluations. Its open-source nature permits reproducibility, facilitating future research with comparative performance analysis.

4.2 Web Browser Interface

The WebBrowser interface, designed to interact with the web app, utilises Web-Driver interfaces via the Selenium WebDriver Python library. A detailed introduction of WebDrivers has been provided in the Background section on Web Testing Frameworks. In summary, a WebDriver, specified by the W3C and implemented by all major web browsers, allows rich programmatic access to a web browser's functionalities. The Selenium WebDriver Python Library provides a user-friendly Python interface for this purpose.

In the context of this project, the Google Chrome browser has been chosen, utilising the corresponding ChromeDriver [12] for the WebDriver interface. This selection is based on the popularity of the browser and reliable WebDriver support.

4.3 High-Level Web App Interface

The High-Level Web App Interface serves as an abstraction layer for the Selenium WebDriver Python library, wrapping various functionality into more user-friendly and syntax-efficient operations. Its primary responsibilities encompass setting up the browser environment and orchestrating several sub-components:

- The 'Action Performer' module simplifies the interaction with the web application, offering high-level interactions, such as clicks or mouse movements, by wrapping WebDriver methods.
- The 'Screen Capturer' module streamlines screenshot acquisition, simplifying the visualisation process. The 'Video Recorder' combines the screenshots taken by the 'Screen Capturer' to create videos, optionally visualising and painting user interactions such as clicks on specific screen locations in the video recording. This feature enhances the traceability of the testing and debugging process, providing an intuitive overview of the actions performed.
- The 'Web Element Grabber' efficiently retrieves all web elements present on the webpage. The 'Dead end Handler' monitors the state of the web application, identifying and handling irreversible states such as being logged out. If the application enters a dead end state, it returns the application to a functional state, ensuring the testing process can continue.
- The 'Preamble Injector' provides users with the flexibility to define specific steps to be executed upon launching the web app, such as creating a new user account and logging into the web application, as in the case of the RWA.

This functionality allows for tailored and controlled starting conditions for each testing process.

- Lastly, the ‘Error Listener’ vigilantly keeps track of any JavaScript errors that occur during the test, recording them for later inclusion in the test report.

Collectively, these functionalities form a robust and reusable interface that simplifies the interaction with the web application beyond the WebDriver interface, streamlining the operations of all the system’s upper layers.

4.4 Gymnasium Environment

The Gymnasium Environment layer of the system architecture implements the standard Gymnasium API [19] (formerly OpenAI Gym [10]), which forms a strategic design decision for this project. Gymnasium is a Python library providing a standard interface between RL algorithms and various environments. It offers a flexible, standardised API that enables easy development and comparison of RL algorithms. Essentially, Gymnasium abstracts the complexities of the environments, enabling the developers to focus on crafting and refining the RL algorithms.

The most significant advantage of implementing a Gymnasium interface is the plug-and-play compatibility with a wide array of open-source RL algorithm implementations and libraries. Moreover, by adhering to this standard interface, one can leverage pre-implemented wrappers that facilitate reward and observation pre-processing, along with environment vectorisation. Vectorisation enables running multiple environments in parallel, which is a common approach to speeding up training in RL.

A Gymnasium environment contains two fundamental properties: the observation space and the action space. The observation space is a mathematical model that defines what an environment observation looks like, typically being either a set of discrete entities or a tensor of continuous real numbers. The same applies to action spaces, where the action space defines the mathematical representation of an action applied in the environment. This project defines the observation space as an image, more specifically an RGB colour screenshot downscaled to a size of 128×128. This type of observation is represented by a tensor $\boldsymbol{O}_{RGB} \in \boldsymbol{R}^{128\times128\times3}$. Alternatively, $\boldsymbol{O}_{GRAY} \in \boldsymbol{R}^{128\times128\times1}$ is also used and indicates a grayscale image of the same size.

The logical components of a Gymnasium environment can be broken down into the following:

- ‘Environment Observer’: This module captures a screenshot image from the high-level web app interface, applies pre-processing (like grayscale conversion or image resizing), and transforms it to the appropriate data type for the Gymnasium API.

- ‘Action Processor’: Given an action within the specified action space, this module scales and post-processes the predicted action from the raw action space representation to a usable action that is executed on the web app interface.

- 'Reward Calculation': Arguably the most important component, it generates a reward as a real-numbered scalar, guiding the RL algorithm. Positive rewards promote behaviour, while negative rewards discourage it. Zero rewards are neutral, indicating neither promotion nor discouragement. The choice of reward function may require knowledge of elements present on the webpage and can utilise information from the 'Web Element Grabber' component for this purpose. In addition, knowledge of whether the agent has provoked a dead end state may also be valuable, making it also reliant on the 'Dead end Handler' for this purpose.

Overall, the Gymnasium Environment serves as a structured and efficient way to link the high-level web app interface with RL algorithms, promoting an isolated and hence effective RL algorithm development process.

4.5 RL Algorithm

The RL Algorithm Layer implements the 'intelligent' aspect of the autonomous testing system. It involves training an RL algorithm to produce the algorithm's optimal parameters (weights and biases), a process that typically demands substantial computational effort and training time. However, once determined, these parameters can be reused and fully characterise the agent's behaviour. After training, the inference is almost instant, as small networks, like the ones used in this project, are computationally lightweight.

In the RL Algorithm Layer, a CNN serves as the Feature Extractor. CNNs are chosen due to their ability to efficiently learn from the high-dimensional, grid-like data that screenshots represent. The CNN takes the processed screenshot images as input and transforms these into a set of abstract features that summarise the state of the GUI. A simple architecture is preferred here to maintain sample-efficient training, as the acquisition of data in the form of experiences with the environment is computationally expensive.

Following the feature extraction, the features are fed into a DNN Policy. The DNN Policy acts as a function approximator, taking the high-level features and predicting the next action, which the RL agent should take. The DNN is composed of multiple fully connected layers and, like the CNN, is typically kept shallow to allow for training with limited data.

The Algorithm Trainer is a central component that uses the feature representation from the CNN and the DNN Policy's predictions to adjust the parameters of these networks. It does this based on the reward signal it receives from the RL Environment (Gymnasium). Through iterative optimisation, the Algorithm Trainer adjusts the weights and biases of the CNN and DNN to maximise the expected cumulative reward.

Lastly, the Algorithm Executor acts during the inference phase. It employs the optimal weights computed during the training phase and executes an RL inference loop, allowing the RL agent to interact autonomously with the GUI. The Algorithm Executor pays particular attention to any JavaScript errors that occur during this exploration and maintains a screen recording of all actions leading up to detected faults. These outputs are invaluable test artefacts that aid developers in debugging and rectifying any faults identified. As inference of the relatively small network architecture is very fast, the algorithm executor is suitable for integration into continuous deployment pipelines, offering the potential for ongoing and continuous test automation.

One fundamental design decision in this project was the implementation approach of the RL agents. The two considered options are discussed below:

(Option 1) Implement an RL algorithm from scratch: The advantage of this approach lies in the flexibility it offers. A custom implementation allows for easy adaptation, modification, and extension to cater to specific needs without requiring knowledge of potentially complex libraries. It also provides full control over implementation details and decreases third-party dependency, thereby mitigating risks associated with bugs in existing implementations. However, this option has its drawbacks: it is error-prone, demands a strong grasp of complex mathematical optimisation, and the notoriously complex nature of RL algorithms makes debugging and verification difficult. The significant effort of developing an RL algorithm from scratch would likely mean remaining confined to one working implementation within the scope of this project.

(Option 2) Use existing implementations for the RL algorithm: This option eases experimentation with different algorithms as many implementations already exist. It also provides access to standardised algorithms with established performance benchmarks that can be replicated. Using existing implementations avails much functionality, including vectorised environments and utilities for experiment management and hyperparameter tuning. Additionally, existing user bases and communities can offer support to address known issues. However, customisation of these algorithms may require a deep understanding of potentially convoluted codebases, and certain modifications may necessitate a considerable amount of boilerplate code. Design decisions made by these libraries, such as system compatibility (i.e. for GPU accelerated training), can also be challenging to resolve.

Several libraries were evaluated and analysed for implementing the RL Algorithm Layer, including CleanRL [24], Ray RLLib [32], and Stable Baselines 3 [45]. CleanRL offers single-file algorithm implementation for simplicity and extendibility but lacks functionality, such as vectorised environments, limiting training speed. Ray RLLib is a robust library with numerous algorithm implementations, however, it has poor and partly outdated documentation, is difficult to extend, and during the initial analysis of the library, multiple bugs were encountered. The library is targeted for large-scale production systems potentially deployed in the cloud rather than smaller-scale research use. The third library, Stable Baselines 3, provides straightforward, well-documented Python implementations of various algorithms, supports vectorised environments, and facilitates convenient local experiment tracking with Tensorboard. Despite requiring a moderate amount of boilerplate code to extend existing implementations, it serves as a user-friendly choice for research, albeit lacking implementations of RNN/LSTM wrappers for existing algorithms.

In the end, Stable Baselines 3 was selected as the library of choice for implementing the RL Algorithm Layer. Its user-friendly documentation, support for vectorised environments, convenient local experiment tracking, and straightforward Python implementations of various algorithms aligned best with the scope and goals of this project. The necessary extension was manageable despite requiring a moderate amount of boilerplate code. Although it currently lacks implementations of RNN/LSTM wrappers for existing algorithms, its advantages greatly outweighed its shortcomings, making it a practical and well-suited solution for this research-oriented project.

5 Implementation

5.1 RL Algorithm

This section outlines all the implementation details relating to the RL algorithm layer. The first section outlines implementation details for the SAC algorithm, which is the chosen algorithm for this project. Afterwards, details regarding the implementation of the policy will be discussed. Subsequently, the concept of frame-stacking is described. To compare the method's performance, baseline implementations are discussed and finally, the test executor, which forms the deployed test system will be described.

5.1.1 Soft Actor-Critic (SAC) Implementation

The SAC algorithm, as detailed in the paper by Haarnoja et al. [23], is a founded choice for the RL algorithm implementation in this project. Firstly, SAC is based on the maximum entropy reinforcement learning framework, which encourages the exploration of the action space by maximising entropy. This is particularly beneficial for the automatic GUI exploration agent, as it ensures a varied exploration of the web app under test. Secondly, SAC training is particularly stable and the algorithm is comparably insensitive to hyperparameters, making it robust and well-suited for real-world applications. Other algorithms often have a very narrow window of effective hyper-parameters where training is possible, while SAC training remains intact for a very wide range of hyperparameters. Finally, SAC is designed to be sample-efficient, achieving excellent results in fewer training time steps compared to other algorithms. This is particularly advantageous in situations where data acquisition is computationally expensive or time-consuming, as it is in the case of this project with the relatively slow web browser environment.

In summary, the SAC algorithm is made up of the following components:

1. **Replay Buffer ($\mathcal{D}$):** Being an off-policy algorithm, SAC can learn from past 'recycled' experiences. To do this, the replay buffer is used as a store of previous experiences with the environment. Each experience is a tuple (s, a, r, s', d), where s is the current state, a is the action taken, r is the reward received, s' is the next state, and d is a boolean flag indicating whether s' is a terminal state. The replay buffer allows the algorithm to learn from past experiences, which improves sample efficiency and stability.

2. **Target Functions:** SAC uses two target functions, which are used to compute the target values for the Q-function updates. These target functions are based on the current policy and the current Q-functions.

3. **Q-functions:** In line with this, SAC uses Q-functions to estimate the expected return of taking a particular action in a given state. The use of two Q-functions is a technique borrowed from Double Q-Learning, and it helps to mitigate the overestimation bias that can occur in Q-Learning. To prevent overestimation, the more conservative (lower) Q-value estimate of the two is picked at each instance.

4. **Policy:** The policy in SAC is a stochastic policy that outputs a probability distribution over actions. The policy is updated to maximise a trade-off between expected return and entropy, a measure of randomness. This encourages the policy to maintain a balance between exploration (choosing random actions) and exploitation (choosing the best-estimated action).

Algorithm 1 Soft Actor-Critic (SAC) Training

```
1: **Input:** initial policy parameters $\theta$, Q-function parameters $\phi_1, \phi_2$, empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\phi_{targ,1} \leftarrow \phi_1, \phi_{targ,2} \leftarrow \phi_2$
3: **repeat**
4:     Observe state $s$ and select action $a \sim \pi_\theta(\cdot|s)$
5:     Execute $a$ in the environment
6:     Observe the next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
7:     Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
8:     If $s'$ is terminal, reset environment state.
9:     **if** it's time to update **then**
10:        **for** $j$ in range (however many updates) **do**
11:            Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12:            Compute targets for the Q functions:
               $$y(r, s', d) = r + \gamma(1-d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}' \mid s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot \mid s')$$
13:            Update Q-functions by one step of gradient descent using
               $$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s,a) - y(r,s',d)\right)^2 \quad for \;\; i = 1,2$$
14:            Update policy by one step of gradient ascent using:
               $$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \left(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s)|s)\right),$$
               where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt $\theta$ via the reparameterisation trick
15:            Update target networks with
               $$\phi_{targ,i} \leftarrow \rho\phi_{targ,i} + (1-\rho)\phi_i \quad for \;\; i = 1,2$$
16:        **end for**
17:    **end if**
18: **until** convergence
```

Figure 10: Pseudo-Code of SAC Algorithm training re-created from [2]

The interplay between the aforementioned components is orchestrated through the actor-critic architecture. The 'actor' in this context refers to the policy, which is responsible for deciding which action to take in a given state. The 'critic', on the other hand, refers to the Q-functions, which evaluate the quality of the actions taken by the actor. The critic guides the actor's learning process by providing feedback on its decisions. An overview of the training algorithm of SAC, which is seen in Figure 10 will be provided in the following.

The SAC algorithm starts by initialising the policy parameters θ, the Q-function parameters ϕ_1, ϕ_2, and an empty replay buffer D (line 1). The parameters for the target Q-functions, $\phi_{\text{targ},1}$ and $\phi_{\text{targ},2}$, are then set to match the main Q-function parameters (line 2). The algorithm then enters a loop, which continues until convergence. In each iteration, the agent observes the current state s and selects an action a based on the policy $\pi_\theta(\cdot|s)$ (line 4). This action is executed in the environment, leading to a new state s', a reward r, and a signal d indicating whether s' is a terminal state (lines 5-6). The tuple (s, a, r, s', d) is then stored in the replay buffer D (line 7), and if s' is a terminal state, the environment is reset (line 8).

The policy and Q-function parameters are then updated if the conditions for doing so are met (line 9). This update process involves several steps, which are repeated for a specified number of updates (line 10). Firstly, a batch of transition tuples is sampled from the replay buffer (line 11). The target values for the Q-function updates are then computed based on these transitions (line 12). These target values are used to update the Q-functions (line 13) and the policy (line 14) using gradient descent and ascent, respectively. The target Q-function parameters are then updated to slowly track the main Q-function parameters (line 15). This iterative process continues until the parameters have converged, at which point the trained policy can be used to select actions in the environment.

The logarithmic terms present in the policy update step (line 14) and in the computation of target values for Q-functions (line 12) form the entropy regularisation of the policy. This entropy regularisation encourages high exploration of the action space. The entropy coefficient parameter α weighs the importance of the entropy term against the expected return in both the policy and Q-functions update processes. The modern implementation of SAC used for this project includes an algorithm to automatically adjust the entropy coefficient for optimal results throughout training.

The mathematically sophisticated optimisation procedure discussed here highlights the difficulty of implementing this algorithm from scratch. Therefore, for the implementation of this project, the implementation provided in Stable Baselines 3 (`stable_baselines3.sac.SAC`) [45] is relied upon. Stable Baselines 3 also provides tuned hyperparameters for different environments. The closest related problem is the Atari [40] environments, which also use image-based CNN policies of screens, somewhat related to the GUI screenshots discussed here. For that reason, the Atari hyperparameters are used as the starting point for this project.

5.1.2 Policy Implementation

While the previous section discussed the optimisation algorithm and mathematical framework of SAC, this section will outline the implementation of the policy that SAC is optimising. The policy network architecture is depicted in Figure 11.

The inputs to the policy are the preprocessed screenshots obtained from the environment. The exact preprocessing operations occur before reaching the policy within the environment. The only preprocessing step happening here is image normalisation, where pixel values originally ranging between 0 and 255 are adjusted to fall between 0 and 1.

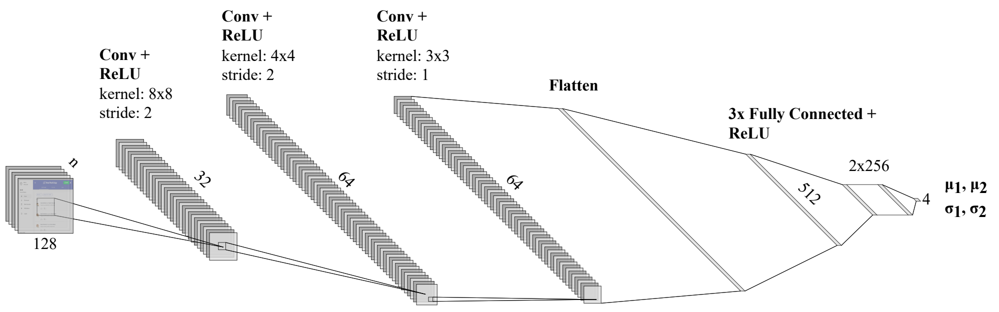

Figure 11: Policy Network Architecture with the 128×128 stack of image inputs on the left and action distribution parameter output on the right. CNN Feature extractors produce 1×512 feature space and subsequent fully connected layers produce the final prediction.

The structure of the policy can be separated into two main components: a CNN that serves as the feature extractor, and a fully connected DNN that produces the outputs based on the derived features. The CNN architecture used as the feature extractor aligns with the default architecture in Stable Baselines 3 [45]. This architecture, which was originally employed in the work by [40], has proven to be successful in the related RL problem domain of visually interacting with Atari Games. The exact layer architecture is given in Table 2.

Layer Type	Filters	Kernel Size	Stride	Activation
Conv2D	32	8	4	ReLU
Conv2D	64	4	2	ReLU
Conv2D	64	3	1	ReLU
Flatten	-	-	-	-

Table 2: CNN Architecture for Feature Extractor as proposed by [40]. The exact layer dimensions depend on chosen image size and number of stacked frames n and are dynamically determined at runtime.

The output of the CNN, a flattened feature vector with a dimension of 1×512, is then fed into the ANN. The ANN consists of two fully connected layers of size 256×256. The ultimate output of the ANN, and hence the policy, is a tensor of dimension 1×4. This tensor corresponds to the parameters of the action distribution — specifically, the mean and standard deviation of a 2-dimensional squashed Gaussian distribution. This distribution serves as a stochastic policy, providing the action selection mechanism of the SAC algorithm with a balance between exploration and exploitation.

To get a better intuition of how the parameters predicted by the policy relate to the real-world actions in the form of mouse clicks within the web applications, an explanation and visualisation are provided. The parameters that are output by the policy are not converted into actions directly. Instead, the action parameters parameterise a 2D Squashed Gaussian distribution and are the mean and standard deviation of this distribution. The action is then obtained by sampling from this distribution. This is known as a stochastic policy, and it allows for on-policy exploration, meaning that when receiving a given observation, the action will not always be the same, as it is randomly sampled from the distribution, which allows for exploration.

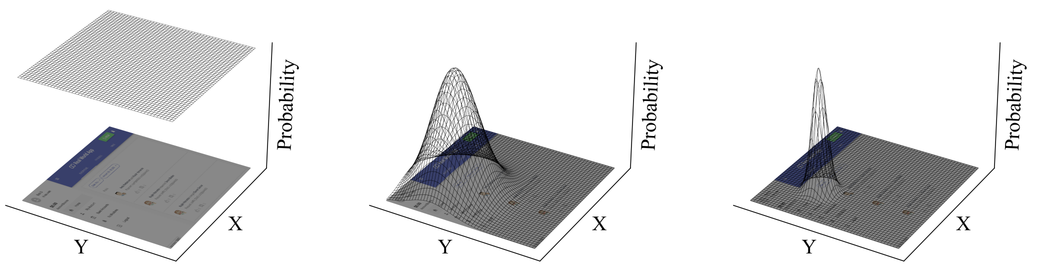

Figure 12: Visualisation of a stochastic policy as training progresses. Initially, actions are sampled from a uniform distribution (high exploration) and progressively the standard deviation of the Gaussian decreases (high exploitation).

Figure 12 shows a visualisation of this 2D distribution. The Y and X axes are the raw unscaled click coordinates and the Probability axis measures the likelihood of selecting a certain action for a given location. Initially, the policy will employ uniform sampling to pre-load the replay buffer, which means that all click coordinates are equally likely to be chosen (pure exploration). Over time, the policy will begin to learn a policy and produce an action distribution with a given mean and a high standard deviation (centre image). The agent will progressively learn a more effective policy and the standard deviation will likely decrease to exploit the learned policy (right image). Interestingly, this stochasticity does not break the optimisation procedure, as the 'reparameterisation trick' is made use of, which still allows the stochastic policy to be differentiated, which is a requirement for backpropagation.

5.1.3 Frame-Stacking

Frame-stacking is a technique commonly employed in RL tasks, particularly those where the state representation is an image, as it is in the problem domain considered in this project. The core idea behind frame-stacking is that instead of feeding the RL agent just a single image (or 'frame') as the current state observation, multiple consecutive frames are stacked together and provided as input. This stacking operation is typically done along the channel dimension, so for RGB images, a stack of four frames would have a shape of $H \times W \times (C \times n)$, where H and W are the height and width of the image, C is the number of channels in each image (3 for RGB, 1 for grayscale), and n is the number of stacked frames. Frame-stacking is advantageous as it allows the agent to perceive dynamic behaviour in the form of a recent history of observed states within the environment. This is particularly relevant when exploring a web app, where the previous actions are likely to influence the subsequent actions in the exploration sequence. The concept of frame-stacking was notably employed by [40] in their pioneering work on training RL agents to play Atari games.

In this project, frame-stacking is implemented using an environment wrapper. This wrapper intercepts the interactions between the RL agent and the environment, and takes care of stacking the frames before they are passed to the agent. The number of stacked frames, denoted as n, is a hyperparameter of the system, and it can be tuned to meet the balance of capturing a sufficient history of states to facilitate sophisticated GUI interactions, while at the same time not including an excessive number of frames that increase training difficulty. The choice of n is subject of an ablation study, described in the following sections.

One important aspect to note is that Stable Baselines 3, the library used for implementing the SAC algorithm in this project, does not have out-of-the-box support for Long Short-Term Memory (LSTM) architectures for SAC. LSTMs are a type of recurrent neural network that can inherently handle sequential data, like a series of images, by maintaining a form of internal 'memory'. Incorporating LSTM layers into the policy network could be another way to provide the agent with a sense of temporal context, similar to what frame-stacking accomplishes. The method developed by Eskonen et al. [18] relied on an LSTM to capture sequential information. As integrating LSTMs would require considerable modification to the Stable Baselines 3 library, it was avoided in this project and frame-stacking was utilised instead to serve as a simpler, yet effective, alternative to using LSTMs.

5.1.4 Baseline Implementations

In order to evaluate the effectiveness of the RL-based GUI testing developed in this project, three baseline methods are implemented. These existing methods for GUI testing enable a relative performance comparison. The three baseline methods selected are random testing, Q-Learning, and a human baseline.

Algorithm 4 Random Baseline Algorithm

```
 1: Input: Web Application
 2: Initialise $newElementsDiscovered$ to 0
 3: Initialise $i$ to 0
 4: while $i < 20$ do
 5:     Draw a sample $(x, y)$ from a 2D-uniform distribution representing the action space
 6:     Perform the click at $(x, y)$
 7:     Get the number of newly discovered elements, store as $newElements$
 8:     Add $newElements$ to $newElementsDiscovered$
 9:     Increment $i$ by 1
10: end while
11: Output: Return $newElementsDiscovered$
```

Figure 13: Pseudo-Code showing the implementation for the 'Random' Baseline.

Random: One common approach to automated GUI exploration is random testing [56]. The literature frequently refers to this class of testing also as 'monkey-testing'. To establish a baseline for random testing that can be used to evaluate the automated testing system, a compatible baseline implementation is created.

In summary, the algorithm for Random testing, presented in Figure 13 continuously clicks at a randomly sampled point on the user interface, potentially hitting clickable elements and triggering GUI interactions while doing so.

Algorithm 5 Tabular Q-Learning Algorithm

```
1: Input: Web Application
2: Initialise $elementsDiscovered = 0$
3: Initialise $i$ to 0
4: while $i < 20$ do
5:     $actions \leftarrow getAvailableActions()$
6:     for all $a$ in $actions$ do
7:         if $a$ has never been executed before then
8:             $setQValue(a, 500)$
9:         end if
10:    end for
11:    $A \leftarrow$ get the action with maximum Q value from $actions$
12:    Perform action $A$, i.e. click at x,y
13:    $elementsDiscovered$ += number of new elements discovered
14:    $newEvents \leftarrow getAvailableActions()$
15:    $\gamma \leftarrow 0.9 \times e^{-0.1 \times (len(actions)-1)}$
16:    $reward \leftarrow 1/$(number of times action $A$ was executed)
17:    $maxValue \leftarrow$ get the maximum Q value of $newEvents$
18:    $q \leftarrow reward + \gamma \times maxValue$
19:    $setQValue(A, q)$
20:    Increment $i$ by 1
21: end while
22: Output: Return $elementsDiscovered$
```

Figure 14: Pseudo-Code showing the implementation for the 'Tabular Q-Learning' Baseline adapted from [3].

Q-Learning: The second baseline that will be implemented is the Q-Learning approach, as proposed by [3]. This algorithm uses a Q-Table as the policy. Although other more sophisticated Q-learning approaches with DNN policies exist, they are not directly applicable to web apps, without making further assumptions, as they targeted Android Applications and used state representations that are not directly transferable to web apps.

The algorithm presented in Figure 14 is explained in brief: All the possible actions (i.e. clickable elements) are retrieved from the current web page and all the actions that have not been executed before receive an initial Q value of 500. Then, the action with the maximum Q value is selected and performed, which triggers a transition to a new state. In the new state, again all possible actions are obtained. The Q value of the executed action is calculated as the current reward and the future reward, where the current reward is inversely proportional to the times the action has been executed and the future reward is proportional to the maximum Q values in the state that is transitioned to. This ensures that areas in the web application that are unexplored are likely to be explored next, due to the high initial Q-value of 500.

One important thing to note about this baseline is that, unlike other baselines, it relies on a function that retrieves clickable elements on the web page. In this application, which is specific to web applications, these actions are defined as either buttons, links, or any elements that have a JavaScript `click` EventListener attached to it, indicating that the element provokes some kind of JavaScript execution.

Human Tester: To reliably facilitate the baseline experiments with human testers, a system was implemented, which would retrieve the current screenshot from the web application, display it to a human user and allow the user to click at any desired location while following the objective to explore as many unique states and elements of the web app GUI as possible. After the human tester has inputted a fixed-length sequence of clicks, the experiment ends automatically and records the cumulative number of unique elements discovered throughout the trial.

In summary, the technical details regarding the baseline experiments have been discussed. The precise experimental setup will be discussed in the following 'Experimental Setup' chapter, followed by the results in the subsequent section.

5.1.5 Test Executor

The Test Executor is the final package that is integrated into the testing process for the GUI application. Its purpose is to apply the learned policy parameters—namely the network weights and biases acquired during training—and effectively interact with the web app using the trained agent. The goal of the Test Executor is to monitor and log any JavaScript errors that occur during the GUI interactions. Once the test run is complete, the Executor generates test artefacts. These include a detailed log file with all the encountered JavaScript errors, and a video recording of the entire interaction process showing the click locations of the agent. In case the executor encounters any GUI faults, the produced test artefacts allow the developers to easily investigate and reproduce the error and pinpoint the cause.

Algorithm 2 Automatic GUI Test Algorithm

```
1: Input: Optimal policy weights $\theta$, number of steps $t$
2: Initialise empty error log $L$
3: Begin screen recording and write video to $R$
4: for $i$ less than $t$ do
5:     Observe state $s$ from the environment
6:     Predict action $a = \pi_\theta(s)$
7:     If any new errors occur, add them to log $L$
8: end for
9: Stop screen recording $R$
10: Output: Return artefacts $R, L$
```

Figure 15: GUI System Test Pseudo-Code (Inference Time).

Figure 15 provides the simplified pseudo-code of how the test executor performs a test run. The Stable Baselines 3 library provides methods to automatically save and load a policy. This implementation specifically uses the method stable_baselines3.sac.SAC.load() to simply read in the previously stored weights from a zip file. In summary, all that is needed to deploy a trained agent is a lightweight zip file with the weights and the Test Executor script, which allows developers to flexibly integrate this automated testing procedure into their development and testing setups. The tests can then be continuously executed in cloud CD pipelines or even in the local development setup, as the model inference can run on most common consumer hardware without requiring intensive computational resources.

5.2 Gymnasium Environment

To briefly recap, the purpose of the Gymnasium Environment layer is to create a standardised abstract representation of the environment, which is compatible with available RL algorithm implementations. The functionality that this layer implements can be grouped into three different modules, which are the Environment Observer, Action Processor and Reward Calculator. Collaboratively, these modules handle environmental observations, action executions, and reward feedback mechanisms. In the following, the important implementation details will be outlined.

5.2.1 Observations and Actions

The Gymnasium Environment is characterised by two fundamental properties, namely the action space and the observation space. These properties form the essential interface for the RL agent, dictating what it can observe from the environment and how it can interact with it.

Observations For the implementation of the observation space, the gymnasium.spaces.Box class was utilised. This class allows the construction of multi-dimensional boxes, and it is particularly useful for defining observation spaces that are sets of continuous real numbers. For this project, the observation space is defined as an image, specifically a screenshot of the web app interface, downscaled to a specified size.

This observation space is implemented as follows:

```
spaces.Box(low=0, high=255, shape=(downscale_height,
    downscale_width, n_img_channels), dtype=np.uint8)
```

In this implementation, low and high are the lowest and highest possible pixel values, respectively, which are set to 0 and 255 as typical for a standard 8-bit RGB image. The shape parameter is a tuple specifying the dimensions of the observation space. The dimensions downscale_height and downscale_width represent the height and width of the downscaled screenshot, respectively, and n_img_channels represents the number of channels in the image (3 for RGB, 1 for grayscale). The dtype parameter is set to np.uint8 which is the typical data type for images, representing 8-bit unsigned integer arrays. The downscale size, as well as the colour mode, can both be adjusted in the config files. The image downscale size is set to 128×128, as it was the smallest possible size that would still allow small, yet important details in the GUI to remain visible.

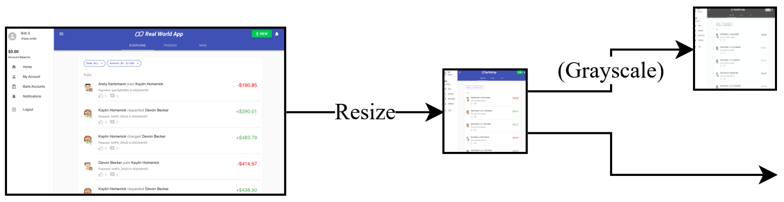

Figure 16: The observation pre-processing from the screenshot (left) to the observation space. Grayscaling optionally occurs if it has been enabled in the configuration.

Starting with the capture of a screenshot in the form of a `PIL.Image` (Python Pillow), the observation formation process, illustrated in Figure 16, involves several steps. The captured screenshot is downsized to the determined downscaling size using Bicubic interpolation. If enabled in the configuration, a colour space conversion transforms the RGB image into grayscale, through the following operations:

$$\begin{bmatrix} I \end{bmatrix} = \begin{bmatrix} 0.2989 & 0.5870 & 0.1140 \end{bmatrix}^T \cdot \begin{bmatrix} R\ G\ B \end{bmatrix} \tag{3}$$

These weights reflect the human eye's relative perception of luminance for each colour according to the ITU-R BT.601 standard [28]. Notably, image normalisation occurs separately outside of the environment prior to the CNN feature extractor.

Actions The action space is the second fundamental property of the Gymnasium Environment. The action space is defined as a two-dimensional continuous space where each dimension falls in the range between -1 and 1. The scaling of the action value between -1 and 1 is a standard RL convention that most continuous action space algorithms follow. Again, the class `gymnasium.spaces.Box` is utilised to implement this action space as it allows for the construction of continuous spaces:

```
spaces.Box(low=-1, high=1, shape=(2,), dtype=np.float32)
```

The `low` and `high` parameters define the range of the action space. Each action dimension corresponds to an action that the agent can perform in the web app's GUI. The `shape` parameter indicates that there are two dimensions in the action space. The data type `np.float32` is used to denote that the actions are continuous values.

The first dimension relates to the x-coordinate of a mouse click, and the second dimension to the y-coordinate. However, the actions in the action space first need to be converted to actual screen pixel coordinates. This is because the RL algorithms operate in the normalised action space, while the GUI interface requires absolute pixel positions for interactions. Hence, a post-processing step is included, which scales the normalised actions into screen coordinates as follows:

```
x = int((x_norm*0.5+0.5)*viewport_width)
y = int((y_norm*0.5+0.5)*viewport_height)
```

Here, `x_norm` and `y_norm` are the normalised x and y coordinates, respectively. These values are first scaled from the range [-1, 1] to [0, 1], then multiplied by the viewport's width and height. The continuous floating point values are then discretised by casting them into an integer. Finally, the resulting x and y coordinates are then passed to the GUI interaction functions.

5.2.2 Reward Calculation

The reward function in RL has a profound influence on the training process, and ultimately on the success of the agent. Given the overarching goal of the RL agent to effectively explore the GUI and uncover potential faults, it is crucial to devise a reward mechanism that accurately and beneficially encapsulates this objective.

One direct approach could be to simply design a reward function that incentivises the agent to maximise the number of faults discovered. However, such an approach is fraught with challenges. Firstly, the rewards would be sparsely distributed in the action space as bugs in an application are typically infrequent [17]. Sparse rewards make training difficult as the agent doesn't receive informative feedback for most of its actions. Secondly, such a scheme would be highly dependent on the specifics of what is classified as a 'bug', making it hard to generalise across different applications.

A more effective approach is to incentivise the agent to explore the user interface efficiently, under the assumption that effective exploration will likely uncover bugs. As there is not one right solution to an effective reward function, three possible implementations have been designed and implemented to allow for an empirical evaluation later on. In addition to the three reward 'Variants' outlined in the following, details describing two 'Add-Ons' are provided, which are optional post-processing steps applicable to each of the variants. The three reward implementations are outlined in the following:

Variant 1: Visual Differences This method assumes that actions causing significant state changes in the GUI will induce substantial visual differences. The reward is calculated based on the pixel-level difference between consecutive screenshots. A non-productive click, such as on an empty area, would not cause any changes to the GUI, resulting in a reward of zero. Conversely, a click that causes dynamic changes in the GUI will yield visibly differing screenshots, and hence a larger reward. The implementation is shown in 1.

```python
def visual_reward(previous_observation, observation):

    # Calculate the absolute difference between the frames
    diff = np.abs(prev_obs - obs)

    # Sum up all differences
    diff_sum = np.sum(diff)

    # Return the reward
    return diff_sum
```

Listing 1: Reward function to compute the visual reward given the current and previous observations. Adapted for improved readability.

Variant 2: Change in Elements The second method is similar to the visual approach but relies on the changes in HTML elements of the GUI, instead of visual differences. The principle here is that new or different elements appearing or disappearing from the GUI are an indicator of dynamic GUI changes and exploratory behaviour. Such changes are quantified by comparing the sets of HTML elements before and after an action. The reward calculation code for this method is as follows:

```python
def element_delta_reward():

    # Get the set of currently visible elements
    current_elements = set(get_visible_paths())

    # Length of set difference of prev. and current elements
    element_delta = len(current_elements - prev_elements)

    # Update the previous elements
    prev_elements = current_elements

    # Return the reward
    return element_delta
```

Listing 2: Reward function to compute the reward as the change in elements between two states. Adapted for improved readability.

This method compares the current visible HTML elements with those from the previous state. The set difference, denoted by `delta`, is the number of elements that appeared or disappeared as a result of the latest action.

Variant 3: New Elements Discovery The final reward implementation is designed based on the premise of incentivising the discovery of new, previously unseen HTML elements in the GUI. It is similar to the 'Change in Elements' method but focuses on novel discoveries rather than changes. The assumption is that discovering new elements indicates further exploration of the application and hence potentially uncovers more faults.

The reward in this method is calculated based on the number of new HTML elements that become visible as a result of the agent's action. These elements are identified by comparing the current set of visible elements with a running set of known elements, stored in `known_elements`. If an element is not in the known set, it is considered new and the agent receives a reward. The number of new elements found forms the basis of the reward. The code for this method is presented in the following.

```python
def new_unseen_element_reward():

    # Get the set of currently visible elements
    all_visible_elements = set(get_visible_paths())

    # Get the new elements that haven't been seen previously,
    # which is the following set difference
    new_unseen_elements = all_visible_elements - known_elements

    # Add the new elements to the set of known elements
    known_elements = known_elements.union(new_unseen_elements)

    # Get the number of previously unseen elements
    n_new_elements = len(new_unseen_elements)

    # Return the reward
    return reward
```

Listing 3: Reward function to compute the reward as the number of newly discovered previously unseen elements. Adapted for improved readability.

This implementation first gets the current set of visible elements and identifies the new ones by comparing them to `known_elements`. The newly discovered elements are appended to the `known_elements`, which prevents all elements that have previously been found to contribute to future rewards. This implementation operates under the hypothesis that it nurtures an incentive within the agent to discover and interact with previously unseen elements and regions of the graphical user interface. In doing so, it strategically avoids the agent from falling into a pattern of executing familiar actions, which, while they may cause large state changes within the environment, don't contribute to novel exploration. Hence, it mitigates the formation of possible high-reward loops, where the agent is unduly rewarded for repeated behaviour, detracting from the broader goal of thorough application exploration.

All three implementations will later be evaluated empirically. After the best-performing implementation has been selected, two further reward post-processing strategies will be investigated to further improve upon the implementations. The implementation details of these further 'add-ons' are described in the following:

Add-On 1: Logarithmic Reward Scaling Logarithmic reward scaling is an additional mechanism used to smooth the magnitude of the rewards provided to the agent. By applying the $x = ln(x + 1)$ operation to the raw reward, the spread between very large positive rewards and small positive rewards is reduced. As certain state transitions are associated with very large changes, these would be disproportionally reinforced. For example, some clicks may lead to the discovery of long lists of elements, which would produce unreasonably high rewards, making clicks that only provoke small changes, such as clicking a like button, diminish in importance. It is therefore hypothesised that applying logs incentivises a more balanced exploration by the RL agent.

Add-On 2: Negative Rewards Negative rewards are another auxiliary strategy used to shape the agent's behaviour more explicitly. In particular, negative rewards are used to discourage certain actions. For instance, if an action does not lead to the discovery of new states in the GUI (i.e., the reward is 0), a small negative reward could be given instead. This discourages the agent from performing ineffective actions, thereby encouraging the minimisation of 'wasted' clicks. Furthermore, actions that lead to a dead end state in the GUI, from which no further exploration is possible, could be met with a larger negative reward. This discourages the agent from pursuing actions that limit its potential for further exploration. This is the second universal 'add-on' that will be tested.

5.2.3 Gymnasium API

To the RL algorithm interacting with the environment, all that is visible is the abstracted gymnasium API that the algorithms use to interact with the environment. A custom Gymnasium environment is created by inheriting from the `Gymnasium.Env` class and overriding its methods to achieve custom functionality. The Gymnasium API has got two mandatory properties that are set in the class constructor, which are `observation_space` and `action_space`, as previously discussed. In addition to the constructor, there exist three methods that need to be implemented by a custom Gymnasium environment, which are `reset()`, `step()` and `close()`.

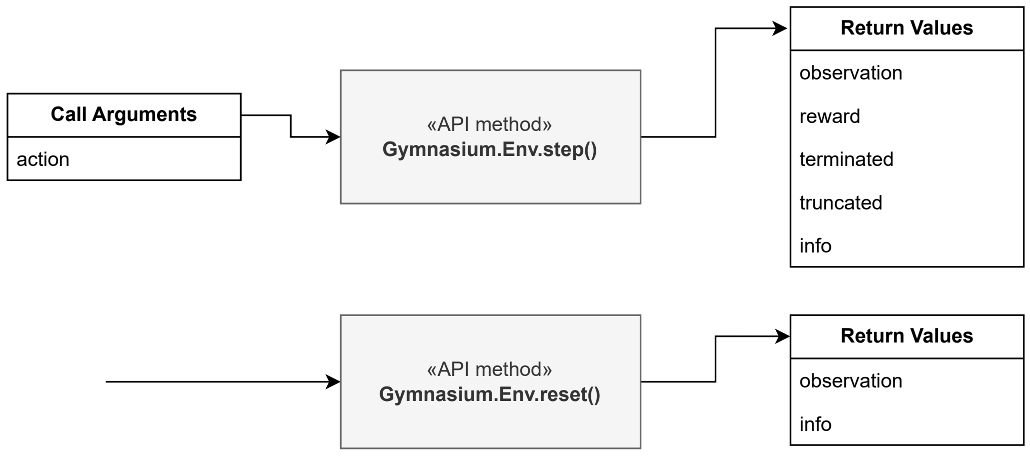

Figure 17: Important Gymnasium API Methods with call arguments and return values.

The `reset()` method is called at the beginning of each episode, setting the environment to an initial state and providing the agent with the first observation. This method is crucial as it prepares the environment for a new episode after the previous one has terminated. In this case, the starting state of the web app (home page) is restored and all the data structures relevant for the reward calculations, such as the set of known elements, are reset to their initial states. As seen in Figure 17, the reset method does not require any call arguments and it returns the initial observation and an optional 'info' value to pass additional information, which is not used for this implementation.

The `step()` method is where the main interaction with the environment occurs. This method accepts an action as input and performs this action in the environment, updating its internal state accordingly. It then returns the next observation that the agent perceives following the action, along with the reward for the action, a boolean flag indicating whether the episode has terminated, another boolean flag indicating whether the episode was truncated and again an info object, which is not used. In this implementation, an episode can terminate for either of two reasons. The first reason is termination due to reaching the maximum configured horizon, which is set to 20 steps. The value of 20 steps is configurable and was found to be a realistic number of actions to sufficiently explore smaller web apps. The second reason for termination is reaching an invalid 'dead end' state, which typically is the agent inadvertently leaving the web application to another website by following an external link or by logging out of the user account. Both of these dead end states make further exploration impossible leading to a direct termination of the episode. In the case of the second reason, the 'truncated' boolean flag will be set accordingly.

Lastly, the `close()` method is responsible for freeing any resources used by the environment. This method ensures that the browser application is properly closed and is usually called when the RL training or evaluation is completely finished.

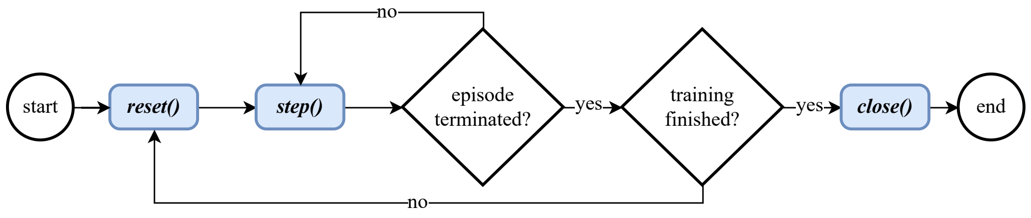

Figure 18: Typical flow-chart of interactions with a Gymnasium Environment.

These outlined methods are typically called in a standard sequence, depicted in Figure 18: Initially, the `reset()` method is called to set the initial state, in this case the home page view of the web app. Following action predictions from the agent, the `step()` method is called in a loop until the episode is terminated. Another outer loop resets the environment and starts a new episode until training is finished. Once training is finished, the environment is closed by calling the `close()` method and the flow ends.

In conclusion, the Gymnasium API facilitates the creation of an abstract layer that helps to simplify the complex web browsing environment for RL algorithms. By inheriting from the `Gymnasium.Env` class and appropriately defining the `observation_space`, `action_space`, `reset()`, `step()`, and `close()` members, a bridge between the intricate workings of a web application in a browser and the abstract mathematical models used in reinforcement learning was constructed.

5.3 Web App Interface

5.3.1 Screenshots and Recordings

The Web App Interface provides a critical link between the RL agent and the web application it is designed to interact with. A central aspect of this interface lies in the capture of screenshots and video recordings of the web application.

Screenshots Capturing screenshots of the web app forms a fundamental feature required for the operation of our system. These screenshots provide the observations for the RL agent, serving as the basis upon which the agent determines the actions it needs to execute. Furthermore, these screenshots are invaluable for debugging the system, as they offer visual snapshots of the application's state at various points during the interaction.

Typically, the browser operates in a headless mode, where no actual browser window is rendered. In this context, screenshots serve as the only visual representation of the application's current state. The implementation of the method `WebAppInterface.get_screenshot()` resembles these lines of code:

```python
def get_screenshot():
    # Get the screenshot as a bytes object
    screenshot_bytes = webdriver.get_screenshot_as_png()
    # Convert the screenshot to a PIL image
    image = Image.open(BytesIO(screenshot_bytes))
    return image
```

Listing 4: High-Level function to get a screenshot of the current browser window. Adapted for easier readibility.

Video Recordings In addition to screenshots, video screen recordings form another useful feature of the Web App Interface. Unlike screenshots, Selenium WebDriver does not inherently support screen recording, necessitating a custom implementation.

The recording functionality was developed from scratch and operates on a separate parallel thread. This thread continuously captures screenshots at a frequency of 30 frames per second (FPS). The individual frames, along with their respective recording timestamps, are stored in a buffer.

Once the screen recording is stopped, a video is constructed by compiling these frames using the Python OpenCV library. To enhance the interpretability of these videos, the screen recorder supports interaction overlays on the video, which are painted on top of the recorded frames and provide a visualisation of the interaction inputs to the GUI. For instance, mouse clicks are visualised as a red dot on the video. A selection of video frames with interactions is shown in Figure 19.

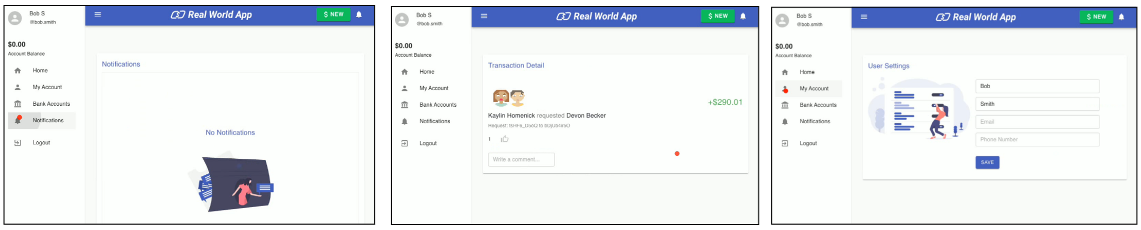

Figure 19: Selection of video recording frames with click interactions painted (red dots).

5.3.2 Action Interfaces

While the previously described screenshot interface is used to obtain observations, the action interface described in the following enables the agent to interact with the environment. Although interfaces for multiple action types, including mouse moves, clicks, and swipes, only clicks were actually used in the end. The action interfaces wrap the low-level WebDriver methods and expose a high-level easy-to-use interface. In the following, the implementation of the click interface is given:

```python
def click(x, y):
    # Clips the coordinates so they are within the browser window
    x,y = _clip_coordinates(x, y)
    # Moves the mouse pointer to the coordinates
    _move_mouse(x, y)
    # Records the timestamp of the mouse click
    t = time.time()
    # Performs the click in the web browser (Selenium)
    ActionChains(webdriver).click().perform()
    # Creates a new MouseClick object and
    # appends it to the action history
    action_history.append(MouseClick(t, x, y))
```

Listing 5: High-Level function for a click action, which wraps a sequence of lower-level Selenium calls. Adapted for easier readibility.

As can be seen from the code snippet, there are several steps involved when performing a simple mouse click, which are all abstracted away and wrapped in a simple `click(x,y)` method call. In summary, the implementation first ensures that the provided click coordinates are within the browser window and clips them to confine them into the valid range. The mouse pointer is then moved to the specified click location, through a call to another custom `move_mouse(x,y)` method. The time of the action is recorded and the click is performed, after which a new custom `MouseClick` object is created and appended to the action history. This action history is used by the video recorder to visualise the interactions on the screen recording.

5.3.3 Preambles and Dead Ends

The necessity to put the web app into a desired starting state for testing, as well as the ability to recover from inadvertently entering a dead end in the exploration process, have been discussed previously. As both dead ends and preambles are effectively custom action sequences, their implementation is very similar. The key difference is the point at which these interaction sequences are triggered: Preambles are injected during the construction of the `WebDriverInterface` class as soon as the web app is opened in the browser for the first time. Dead ends, on the other hand, are handled whenever a dead end state is encountered at any given time step. The implementation of additional custom preambles and dead end definitions is straightforward and allows the test system to be set up with little effort.

One noteworthy implementation detail is that the dead end handler returns a value, which indicates whether a dead end has been discovered or not. This allows for the early termination of the episode and can inform the reward function to allow for the penalisation of actions that provoke dead ends.

In the case of the Cypress Real World App (RWA), the preamble creates a new user account, by navigating to the sign-up page and filling in the sign-up form. After the user is created, it is logged in to the web application, which directs the web app to the home screen that presents the starting point of the exploration. For the RWA, a dead end is defined as either any URL that is outside the web app, reached by accidentally clicking a link that points to an external website, or the login window, indicating that the user has been logged out. In both cases, the web page home page is restored.

5.3.4 Error Listener

The Error Listener implements the fault detection capability of the testing system. It operates by retrieving all logs from the JavaScript Browser Console during the testing process. These logs, typically comprising errors, warnings, and informational messages generated during the execution of JavaScript code in the browser, provide valuable insights into potentially undesired behaviour during the automatic GUI testing procedure. At the end of the test run, the logs are written to a log file and are part of the test artefacts. Each entry in the log file comprises the severity level, the error message itself, and the timestamp at which the error occurred. This information can assist developers in diagnosing and resolving faults that the testing process uncovers.

The `webdriver.get_log("browser")` obtains the JavaScript console logs and returns a list of all log lines that have occurred during the browser session. The Error Listener then iterates through this list, writing each entry to the log file. Each log file entry provides information on the severity level, the actual log or error message, and the timestamp at which the error occurred. The severity level can vary and include 'DEBUG', 'INFO', 'WARNING', or 'SEVERE', each dictating the seriousness of the log entry. This helps in quickly identifying critical issues ('SEVERE') that require immediate attention.

5.3.5 Element Grabber

The reward calculations (Variant 2&3) used in this project rely on the functionality that is implemented by the 'Element Grabber' described in this section. More specifically, two key requirements need to be met by this component. The first requirement, used by Reward Variant 2 is the determination of change in elements between two consecutive states. To determine this, the count of unique visible elements needs to be available. The second requirement, used by Variant 3 is the extraction of new, previously unseen elements, which requires unique identification of elements across different states.

Elements on a web page, defined by their HTML tags do not always have an inherent unique ID associated with them. This makes identifying the same element across different web pages difficult. Although the Selenium WebDriver does offer a method to get all visible elements from the current web page and attaches a unique identifier to these elements, this identifier is not preserved across state transitions, making the re-identification of the same element on different web pages impossible.

For that reason, a custom method was developed that allowed for the unique re-identification of elements across different web pages and states. The developed method employs some heuristics, as there exists no foolproof method to attach unique identifiers to elements. The developed method relies on xpaths, which is essentially the path of all the parent elements in the XML-like structure of an HTML web page. The identification of elements is performed by the following algorithm:

Algorithm 3 Get identifiers of visible elements

```
1: Input: Web Page
2: Get a list of all elements in the web page and store as $allElements$
3: Initialise empty list $visibleXPaths$
4: for all e in $allElements$ do
5:    Check whether $e$ is visible
6:    if $e$ is visible then
7:       Compute the XPath identifier $xpath$ of $e$
8:       Add $xpath$ to $visibleXPaths$
9:    end if
10: end for
11: Output: Return $visibleXPaths$
```

Figure 20: Pseudo-Code of how unique identifiers are obtained of all visible elements on a web page.

In summary, the algorithm presented in Figure 20 first gets all the elements from the HTML document of the web app, checks which of these elements are currently visible and finally computes the ids in the form of xpaths of these visible elements, which is the returned list. To better explain what xpaths are, and how they are used to compute an element ID, the following example is provided.

```html
<html>
    <body>
        <h1>List Example</h1>
        <ul>
            <li>Item 1</li>
            <li>Item 2</li>
            <li>Item 3</li>
        </ul>
    </body>
</html}

```

Example HTML Document

$$\Downarrow$$

```
/html
/html/body
/html/body/h1
/html/body/ul
/html/body/ul/li[1]
/html/body/ul/li[2]
/html/body/ul/li[3]

```

Associated XPath IDs

Figure 21: Simple example of translating elements from an HTML document into unique XPath IDs.

As can be seen on the example provided in Figure 21, unique identifiers can be obtained by building the xpaths of all the elements. If there are multiple elements with the same xpath, as is the case with the list in the example, the index of the trailing element is simply appended, which allows sibling elements to be correctly treated as separate, distinct elements. This concludes the procedure for obtaining the element ids.

Next to the element id generation, there is a second core functionality that the algorithm in Figure 20 relies upon. This is the determination of wether a given element is visible. At any given time, there may be numerous HTML elements present in an HTML document, which are not actually visible. These elements must be excluded. Again, there is no straightforward property that states whether a given element is visible. Instead, each element is checked for a range of CSS (styling) properties, which are:

- **Opacity:** The opacity property determines the transparency of an element. If the opacity value is set to 0, the element is fully transparent and not visible. If the opacity value is greater than 0, the element is considered visible.
- **Visibility:** The visibility property determines whether an element is visible or hidden. If the visibility value is set to "hidden" or "collapse", the element is

not visible. If the visibility value is set to "visible", the element is considered visible.

- **Height and Width:** The height and width properties determine the size of an element. If either the height or width value is set to 0 or not specified, the element is considered not visible. If both the height and width values are greater than 0, the element is considered visible.
- **Location:** The location of the element is determined and it is checked whether the coordinates are within the viewport. If the element indicates that the element is outside the viewport, it is considered invisible.

Only if all four of the above checks succeed is an element determined as being visible.

One interesting detail regarding the implementation is that it does not rely on Selenium WebDriver API methods, as some checks are cumbersome, slow or simply impossible to do using Selenium. Instead, custom JavaScript code is injected into the web application, using `webdriver.execute_script()`. The injected JavaScript code performs all the operations outlined in this section natively in the browser's JavaScript runtime and only returns the visible element ids back to the Python script. Selenium does however provide this functionality of injecting custom JavaScript code at runtime.

5.4 Performance: Increasing Training Throughput

The speed at which experiences from the web browser environment can be collected is a critical factor in making this RL system practical. Without a high-throughput environment, the training process can become prohibitively slow. While not a separate component in its own right, the need for efficient training directed the design and implementation of various components in the system. Contrary to many machine learning systems where the optimisation procedure is the primary bottleneck, the central constraints in this context were the computationally heavy browser environments. Notably, the entire development process was conducted on a laptop, specifically an Apple MacBook Pro with an M1 Max 10-core CPU, 32GB of RAM, and an integrated GPU. This setup constituted a significant system constraint, necessitating various performance optimisations. The motivation for making the system more efficient, over simply using a more powerful machine is that it would allow the final system to be similarly trained and deployed on a regular web developer's computer without requiring powerful and expensive extra hardware.

Browser RAM Accumulation In the training process, around 10 concurrent threads were typically employed, each running an instance of the Google Chrome Browser. At startup, each browser instance required approximately 300MB of memory, aggregating to a total of about 3GB of RAM for the 10 concurrent browsers. However, over time, this memory consumption would increase linearly as session information accumulates in the browser, reaching up to 3GB per browser after a few hundred thousand interactions and causing the total consumption to peak at > 30GB for the ten web browsers alone. This level of memory use would exceed the available RAM, causing the system to utilise slower swap memory (hard disk), thereby significantly reducing training throughput.

To alleviate this issue, a browser-restart mechanism was introduced, wherein each web browser would be re-launched after a given number of steps. This approach effectively cleared all accumulated memory in the browser, resetting it back to its initial lightweight state. Given the hardware used, re-launch intervals of approximately 1000 steps proved optimal, allowing the 10 training environments to run indefinitely at a high speed without exceeding the available RAM.

Replay Buffer Accumulation The Soft Actor-Critic (SAC) algorithm used for the training process retains past experiences in a replay buffer. However, this replay buffer, which includes the observation framestacks, can accumulate a significant amount of memory after long training runs. In its initial implementation, the memory consumption of the Python script grew to more than 30GB after a few hundred thousand training steps, which again slowed down the training in the later stages.

This challenge was addressed by limiting the length of the replay buffer to 100k entries, thereby discarding older experiences. This strategy not only helped manage memory consumption but also contributed positively to training performance by removing outdated, low-reward experiences from the initial stages of training.

Native JavaScript During the implementation process, it was observed that certain functions implemented via the Selenium APIs were inefficient, significantly slowing down training. For instance, obtaining all elements from a webpage and invoking the `isDisplayed()` method on each element blocked execution at each time step for several seconds, which significantly slowed down training.

To overcome this bottleneck, the same functionality was implemented directly in JavaScript and executed within the web browser. This increase in efficiency was not necessarily because JavaScript is faster than Python, but because the data transfer between Python/Selenium and the browser posed a bottleneck. This issue was addressed by minimising back-and-forth calls between Python and the browser, allowing entire computations to be performed directly in JavaScript and only returning the final results once at the end. Compared to the previous methods, this approach substantially improved the speed of execution, increasing training throughput by a factor of 5.

In conclusion, the outlined optimisations were indispensable to the successful development and execution of the testing system. Without these performance enhancements, the computationally intense browser environments would have rendered the development and training process practically infeasible. Ultimately, the described optimisations yielded a highly efficient web browser environment, capable of generating experiences and simultaneously training at a rate of 10 samples per second. Consequently, extended training runs of 500,000 time steps can be executed on a laptop within a single day. This efficiency not only makes the methods developed here accessible to a broader range of users, including web developers who may not typically have access to powerful hardware but also offers significant cost savings for deployment in cloud environments.

5.5 Configuration Options

This section briefly outlines the implemented strategy of how the system can be configured. As the various components in the testing system (Web Browser Interface, Gymnasium Environment, RL Algorithm, Test Executor) all have configurable parameters that affect their behaviour, it is important to allow users of the system to configure the system without making changes to the code.

To achieve this, a comprehensive configuration file was created, which supplied all the components with the required configuration options automatically. This was done by loading the configuration file, creating a configuration object that included all the options and then passing this object through to all components. The configuration file uses the `.yaml` format, which is a popular choice for configuration files. As there are >50 configurable system parameters, an overview of the most important configuration parameters together with an explanation and suggested values can be found in Appendix: Configuration Options.

One important feature of the configuration system is that at every experiment run, a copy of the `config.yaml` file is created and saved together with the experiment artefacts. This allows for organised tracking of hyper-parameters and permits a reproduction of the experiment run using the exact system state at a later time.

6 Experimental Setup

This section outlines the testing methodology and experimental setup for the system developed in this project. First, an ablation study is performed to examine the impact of different key components that form part of the system. These components include frame stacking, image preprocessing, the CNN feature extractor, and the reward function. Afterwards, a generalisation study is performed, which tests the transferability of the proposed methods to different web apps. Finally, experiments with baseline implementations are performed.

6.1 Research Questions

The experiments are guided by the following research questions:

RQ1: How does the CNN feature extractor influence performance?

RQ2: How does frame stacking influence performance?

RQ3: How do image pre-processing techniques affect performance?

RQ4: How do different options for reward functions affect performance?

RQ5: How well does the system generalise to different web apps?

RQ6: To what extent can trained models be transferred across web apps?

RQ7: How does this system compare to a random approach?

RQ8: How does this system compare to a Q-learning approach?

RQ9: How does this system compare to human testers?

6.2 Ablation Study

The ablation study performed in the following will always modify a singular component on the base architecture. Unless subject to ablation or explicitly stated, all of the following experiments use the configuration summarised in 3:

Component	**Parameter / Configuration**
Frame Stacking	stack size 4, as used in [40]
Horizon length	20
Downscaling Size	128 x 128
Reward Function	Variant 3 with log-scaling, negative default=-0.01
Algorithm	SAC: learning rate=0.0003, gamma=0.99, replay buffer size=100k
Policy Type	CNN Policy from [40]
Web Application Under Test	Cypress Real World App

Table 3: Ablation baseline configuration, which shows the parameters and configuration of the key components.

All experiments in the ablation study will train the agent for 100,000 time steps. Although this is insufficient to reach convergence of the episodic reward, it was found that it is long enough to demonstrate the agent's long-term ability to learn.

6.2.1 CNN Feature Extractor

The proposed system uses a CNN to learn and extract features from the input image. To justify the necessity of this component and answer RQ1, the CNN feature extractor undergoes ablation. To do this, the CNN feature extractor from the ablation baseline in Table 3 is removed and replaced by a simple flattening layer, which flattens the input image into a column vector. This flattened layer serves as the input to the subsequent DNN policy.

6.2.2 Frame Stacking

A history of the n most recent frames is used as the input to the RL agent. To answer RQ2, again an ablation experiment is performed where frame stacking is deliberately omitted. To do this, instead of providing a stack of frames only the single most recent observation is used as the input.

The stack height n used in the ablation baseline (Table 3) was used, as it worked well in previous studies [40]. To validate this choice, a series of experiments are run, where the parameter n is modified.

In summary, the following values are tested: $n = [1, 2, 3, 4, 5, 10]$, noting that $n = 1$ means that no frame stacking is performed and $n = 4$ is the default value in the ablation baseline.

6.2.3 Image Preprocessing

Another potentially influential component of the system is the preprocessing stage that is applied to the screenshot images before they are inputted into the CNN. The ablation baseline (Table 3) performs a grayscaling operation, which is hypothesised to improve training efficiency by reducing the complexity of the input by discarding colour information.

To verify this hypothesis and answer RQ3, this preprocessing step is ablated and an experiment is run where instead the frame stack of grayscale images is replaced by a frame stack with the RGB image screenshots.

6.2.4 Reward Function

The reward function is a decisive component that determines the success of the RL agent. Previously, the implementations of three potential reward functions have been presented. To recap, Variant 1 uses pixel-wise differences between consecutive GUI states to define reward. Variant 2 on the other hand uses the change of elements between consecutive states, while Variant 3 keeps track of seen elements throughout the episode and rewards proportional to the number of new, previously unseen elements.

To answer RQ4, all three variants will be tested in isolation, leaving all other components unchanged. In addition, the best-performing variant will be tested with two reward add-ons, which have previously been described. Reward Add-On 1 applies logarithmic scaling to the reward, while Reward Add-On 2 replaces zero (neutral) rewards with a small negative reward to explicitly penalise useless clicks. These add-ons will be applied to the best-performing reward variant, yielding a total set of four experiments that will inform the design of the final, optimal reward function.

To aid the evaluation of the reward functions, a qualitative assessment of the different reward functions is performed, which allows judging the exploratory behaviour of the RL agent

6.3 Generalisation Study

As most of the testing and development of the system will be carried out on one example web app (RWA), the question arises of whether the methodologies generalise to different web applications. To demonstrate the applicability to another class of common web apps, an e-commerce system, specifically Shopware 6 [49] was chosen. The choice was again motivated by the open-source nature of the system to facilitate future reproducibility and research. The Shopware 6 system was run using an existing Docker image [16], which allowed for a simple set-up of the system with demo product data that resembled a real e-commerce system. A few screenshots from the web app can be seen on Figure 22:

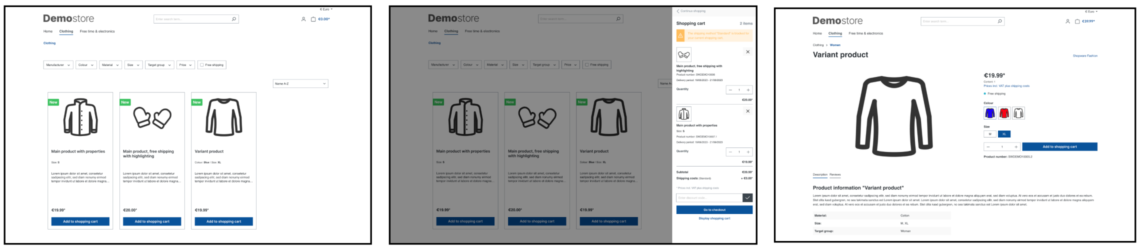

Figure 22: Screenshots from the E-Commerce Store based on Shopware 6, used to test generalisability to different web apps.

The first experiment, aimed to answer RQ5 simply trains the RL model on the Shopware 6 system to show the applicability of the system to a different type of web app. To allow the training to converge and to obtain a well-performing model that allows for effective exploration of the web app, the training run is extended to 500,000 steps. The second experiment sets out to answer RQ6, which is done by using the policy weights from a model that was trained on a different app as the initial weights. This tests whether the transferrable concepts of interacting with a different web app can allow the model to be trained more quickly on a new app. This later experiment is performed by training the model on each of the two apps (RWA and Shopware 6) and each time initialising the weights obtained from training on the other web app. The training runs that test transfer learning will be limited to only 100,000 steps, as this would be sufficient to demonstrate faster learning when compared to randomly initialised weights. This makes the generalisability study constitute three experiments.

6.4 Baselines

Having empirically tested the validity of the architectural choices through ablation, as well as having tested the generalisability of the method to different web apps, it is important to compare the overall system to external baselines to determine its relative performance to existing methods. Three baseline experiments will be performed, which will establish comparable performance scores for a random approach, a Q-learning method and finally for non-automated testing by human testers.

All baseline experiments are conducted with the maximum episode length set to 20. This means that each baseline is tasked to predict a sequence of 20 clicks on the web app with the objective to maximise exploration, measured by the number of unique discovered elements. The horizon length of 20 was chosen, as it was found that 20 clicks are sufficient to explore the major regions and functionality within the relatively simple web apps considered in these experiments.

6.4.1 Random

To address RQ7, the implementation of the random algorithm is run for 20 time steps, producing the baseline results used for comparison. To get more reliable performance results from the random testing, this run is repeated for a total of 20 trials and the mean number of elements discovered per trial is reported. The baseline is run for both web apps, the RWA and Shopware 6.

6.4.2 Q-Learning

The baseline implementation of the Q-Learning method undergoes the same experimental setup as the random baseline, thereby producing results that allow answering RQ8. To summarise, the Q-learning method is run for 20 time steps, repeated for 20 trials and all trials are run on the RWA and Shopware 6.

6.4.3 Human Testers

The final baseline tests the performance of the most challenging competitor: human testers. Albeit not an automated testing strategy, human testers are commonly used to manually test interactions with the GUI. The experiment relies on the implementation described previously. To answer RQ9, two pairs of experiments are run:

Both the RWA and the Shopware 6 web apps are tested. Two groups, each including five student colleagues were asked to take part in the experiment. The first group of students named the ‘expert’ group, was allowed to first take time to explore the web apps prior to the experiment to familiarise themselves with the different states and interactions of the web app. The other group was not allowed to access the web apps prior to the experiment and they are termed, who are referred to as the ‘unfamiliar’ or ‘inexperienced’ group.

In summary, this yielded 20 data points across two different web apps and ten human testers, which allows for an evaluation of the system against a human baseline.

7 Experiment Results

In this section, the results of the previously explained experiments are presented and analysed. First, the results of the ablation study are presented, followed by the experiments of the generalisation study. Finally, the proposed system undergoes a relative comparison against baselines and results together with an analysis are given.

7.1 Ablation Study Results

The ablation study experiments are evaluated by looking at the episodic reward throughout training. The episodic reward is the total cumulative reward that the agent is able to produce at a given time during training. Besides high absolute values, other desirable characteristics for the episodic reward learning curves are stability and the gradient at which they increase. A stable learning curve implies that the agent is steadily learning from its environment and making consistent decisions that contribute to the overall reward. On the other hand, the gradient at which the learning curves increase represents the rate of the agent's learning process; A steep learning curve means that the agent will be able to learn in fewer training steps.

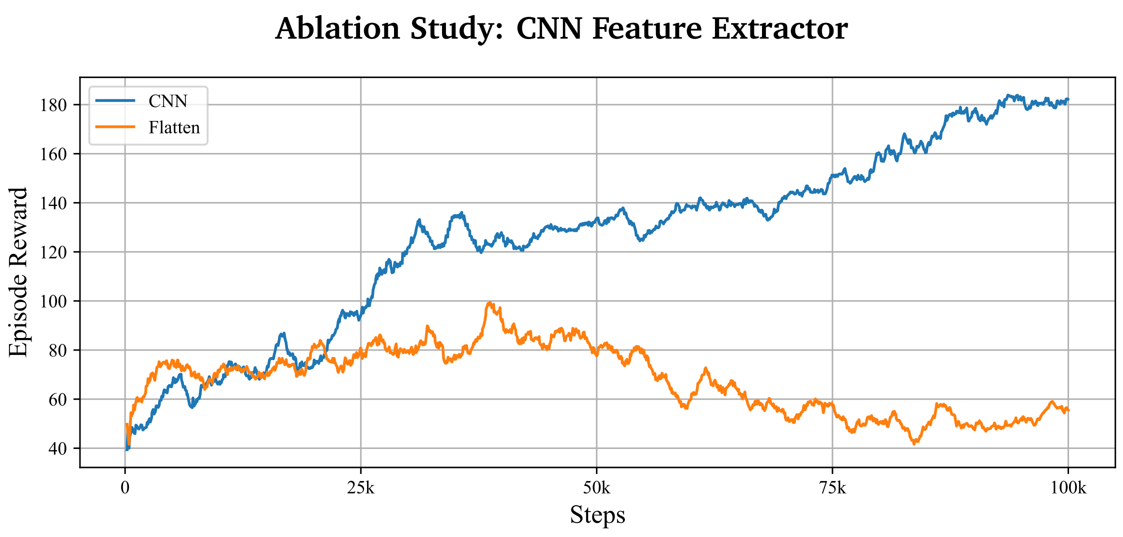

Figure 23: Ablation Study Results for the CNN feature extractor, replaced by a fully-connected 'flatten' layer.

The results of the CNN ablation study, depicted in Figure 23, strongly emphasise the critical role of the Convolutional Neural Network (CNN) as a feature extractor for the RL policy. The agent using the CNN exhibits an upward learning curve, resulting in a high cumulative reward of 180 after 100k steps. In stark contrast, substituting the CNN with a fully-connected 'flatten' layer leads to an unsatisfying performance, evidenced by a lower, plateaued reward under 60. This underscores the inefficiency of the flatten layer to extract meaningful features, thereby impacting the agent's ability to learn from the screenshot images. Thus, the CNN proves to be a vital component of the system, thereby answering RQ1.

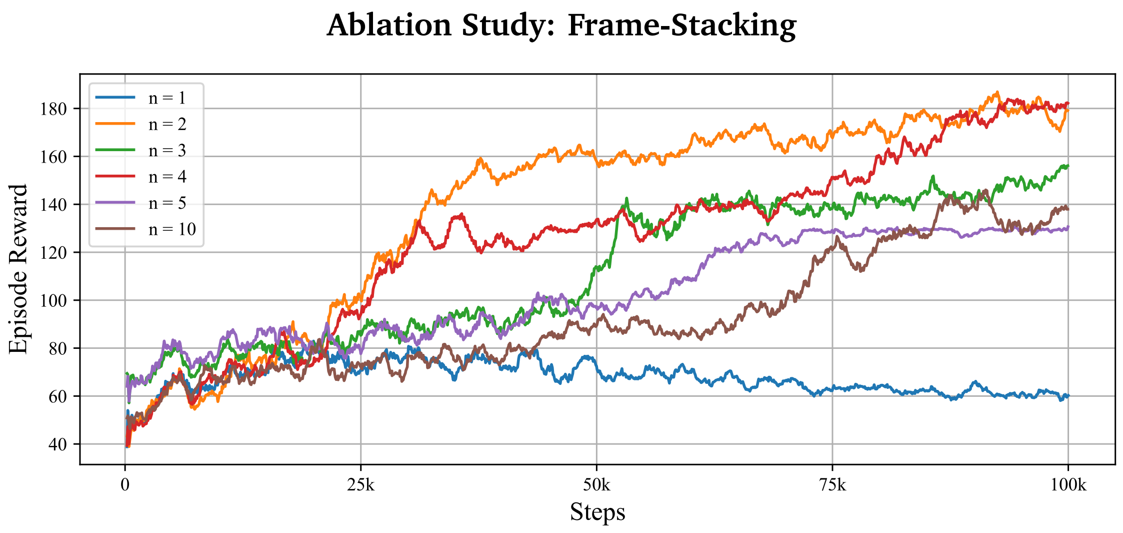

Figure 24: Ablation Study Results for different frame stack heights.

The effect of frame stacking, a technique used to provide a recent state history to the agent, is examined by varying the frame stack heights. Frame stack height represents the number of most recent observations provided to the agent, with different configurations tried: $n = 1$ (no frame-stacking), $n = 2$, $n = 3$, $n = 4$ (as used by prior works [40]), $n = 5$, and n=10. The results are displayed in Figure 24.

From the data, it becomes apparent that frame stacking has a substantial positive impact on the agent's learning capabilities. The agent with $n = 1$ shows poor learning efficacy, reaching a plateau around the cumulative reward score of 60 after 100k training steps. On the other hand, all other configurations result in at least twice the score (>120), showcasing the importance of temporal information for learning an effective GUI exploration strategy.

As for selecting an optimal stack height, a trade-off exists between providing sufficient temporal context for complex sequential GUI interactions and avoiding an excess of information that might inflate the input and hinder efficient learning. Additionally, larger frame stacks demand greater RAM for storing more extensive replay buffers.

Both $n = 2$ and $n = 4$ configurations yielded the best results, reaching a cumulative reward score of 180 after 100k steps. As the learning curve gradient of the $n = 4$ configuration is higher, it suggests that it may outperform $n = 3$ in extended training. Given its performance and precedent in the literature [40], the $n = 4$ configuration was chosen as the optimal parameter, providing an answer to RQ2.

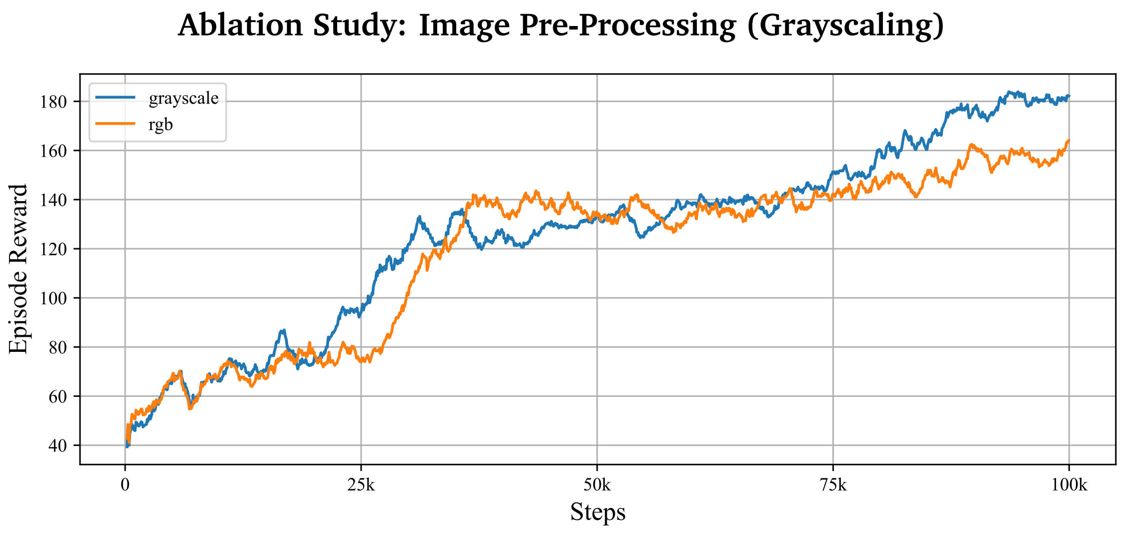

Figure 25: Ablation Study Results for using either RGB inputs or grayscaled images.

In the image pre-processing experiment (Figure 27), the effectiveness of the grayscaling technique is assessed by comparing the learning outcomes of agents receiving either RGB inputs or grayscale images. Grayscaling resulted in a marked performance improvement, with a final cumulative reward score of 180, as compared to 160 for RGB. This implies that grayscaling, which simplifies the input by reducing the number of input channels from 12 (as seen with RGB and frame stacking) to 4 (with stacked grayscale frames), supports the agent's learning process. Importantly, this simplification doesn't lead to the loss of any vital information that might be contained in the colour channels, ensuring effective learning. Moreover, it offers computational advantages, as it requires less processing and memory, with the size of the replay buffer also reduced to one-third. Thus, the grayscaling preprocessing stage proves beneficial, answering RQ3.

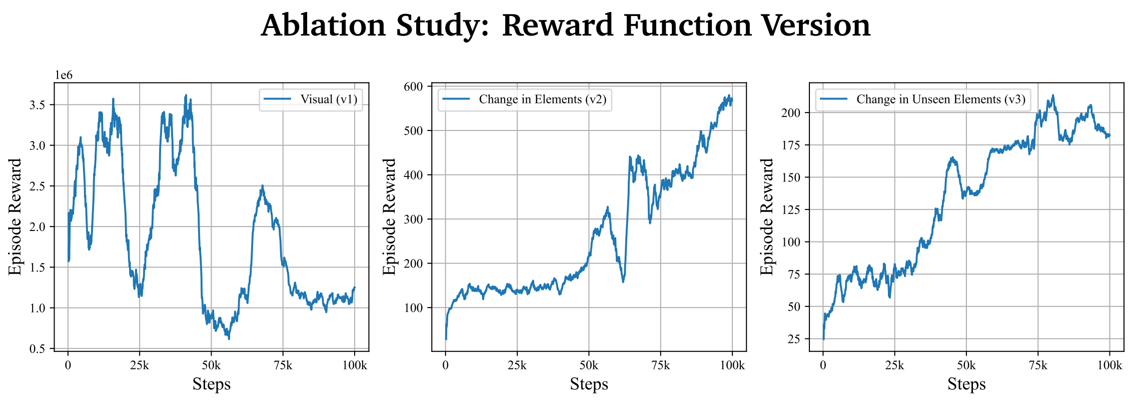

Figure 26: Ablation Study Results for the three different reward variants. Variant 1 (left) relies on visual pixel-level differences between consecutive states. Variant 2 (middle) relies on the difference in elements between consecutive states. Variant 3 (right) relies on addition of new, previously unseen elements.

Three reward function variants were tested in this experiment, with results depicted in Figure 27. Notably, each variant presents a vast difference in reward magnitudes: Variant 1 (v1) ranges up to 3.5×10^6, Variant 2 (v2) approaches 600, and Variant 3 (v3) reaches up to 200. This discrepancy can be disregarded, as reward normalisation is applied during training for each sampled batch, which normalises all rewards into the $[0, 1]$ range.

While reward functions can be challenging to evaluate since they often serve as the ground truth, these learning curves provide valuable insights into the training feasibility and stability of each variant. Variant 1 exhibits heavy fluctuation and lacks steady training progress, rendering it unsuitable. Both v2 and v3 show steady increases, indicating promising training potential.

A qualitative assessment of each reward function was conducted by observing the exploratory behaviour of agents in screen recordings after training for 100k steps. The agent using v2 fell into a repetitive loop of two different states. This loop, while changing large numbers of elements and thus continuously generating positive rewards, demonstrated suboptimal exploratory behaviour. Conversely, the agent using v3 exhibited highly desirable exploratory behaviour, as it could perform navigation and exploration through the GUI, evidently seeking previously unseen elements. In light of these results, Variant 3 was chosen as the most suitable reward function.

Ablation Study: Reward Function 'Add-Ons'

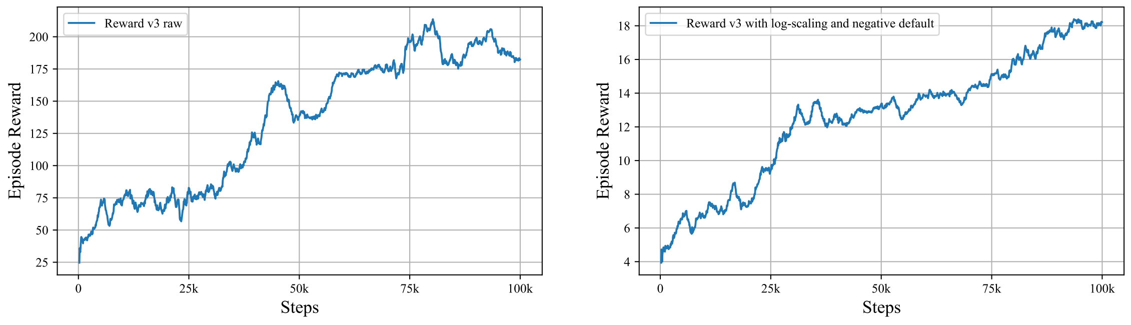

Figure 27: Ablation of the reward add-ons applied to reward variant 3. The applied add-ons perform log-scaling and set a small negative reward as the default when no elements are discovered, instead of zero.

The second reward function experiment examined the effect of applying 'add-ons' to the chosen reward function (v3), as shown in Figure 27. These add-ons comprised logarithmic scaling and replacing zero rewards with small negative rewards, with the aim to explicitly incentivise the agent to avoid unproductive clicks.

Again, a significant absolute scale difference is visible, this time due to the log scaling. This can be ignored, as reward normalisation is applied to all rewards. The introduction of logarithmic scaling and a small negative default reward noticeably stabilised training, as evidenced by fewer fluctuations and temporary drops in the episodic reward during the training process.

A qualitative analysis was again performed by examining the agents' behaviours in video recordings after 100k training steps. The agent trained with the reward add-ons visibly wasted fewer clicks compared to its counterpart. Furthermore, the raw reward function had a preference for states with long lists, as these states offered high raw rewards due to the discovery of numerous new elements. Conversely, simpler actions, such as clicking a 'like' button, which added only a single new element, were often ignored, leading to unbalanced exploration. Logarithmic scaling mitigated this issue effectively by reducing the spread between rewards, thereby promoting a more balanced exploration of the GUI environment. Consequently, this add-on was included in the reward function configuration for the system. With these results and analysis, an answer to RQ4 was provided.

In summary, this comprehensive ablation study was carried out to evaluate the significance and performance of each key component in the proposed system. Through a series of experiments, both numerical and qualitative evidence were gathered to assess their contribution to the system's overall performance. Each component - from feature extraction and frame stacking, to image preprocessing and reward functions - was thoroughly tested. The outcomes of this study not only reaffirmed the importance of each element within the context of the system but also led to informed decisions on optimal configurations. The final system constitutes the configuration options as outlined in Table 3.

7.2 Generalisation Study Results

This section presents the results of the generalisation study.

The first experiment in the generalisation study aimed to test the applicability of the developed system beyond the RWA (Financial Application). The reinforcement learning agent was trained from scratch on a different web application, the e-commerce platform Shopware 6, whether the agent is trainable, and hence applicable to different web apps.

Generalising: Trainability of the Agent across Apps

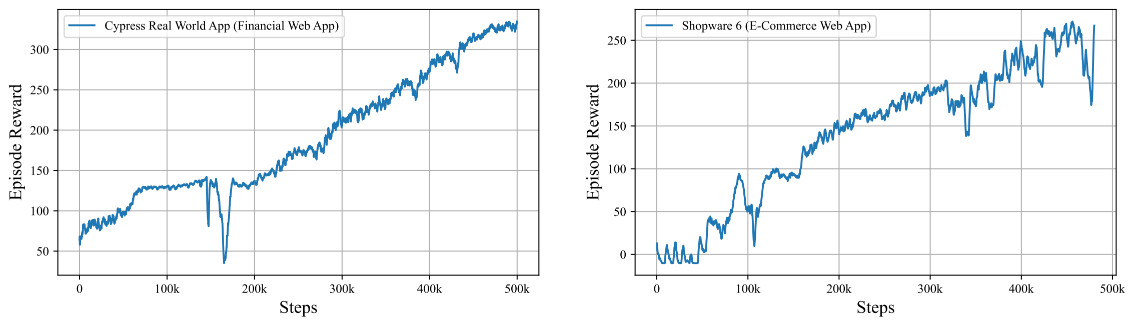

Figure 28: Plots showing training curve for the RWA on the left, which was used for most of the development, and the e-commerce system Shopware 6 on the right, showing that the RL agent is generally trainable on different web apps.

In the experiment, the system was trained for 500k steps. Figure 28 illustrates the side-by-side learning curves with the episode reward throughout training. As expected, the RWA training showed stable training progress and a steady increase in reward. The Shopware 6 application also exhibited effective training. As a side note, the reward values are not directly comparable across different web applications due to differences in the number of elements. An interesting observation was the initial difficulty encountered during the first 50k steps, where the Shopware 6 agent struggled to find rewarding actions. This could be attributed to the home page of the online store, which only hosts a few links leading to more complex areas of the app, thus requiring more time for the agent to identify actions to escape this state. Despite this initial hurdle, the agent successfully overcame this challenge and showed improving rewards thereafter. A qualitative evaluation, which looked at screen recordings of the agent exploring the web app further confirmed the agent's exploratory capabilities, thereby affirming the system's capacity for generalisation and its applicability across different web apps. This analysis provides an answer to RQ5.

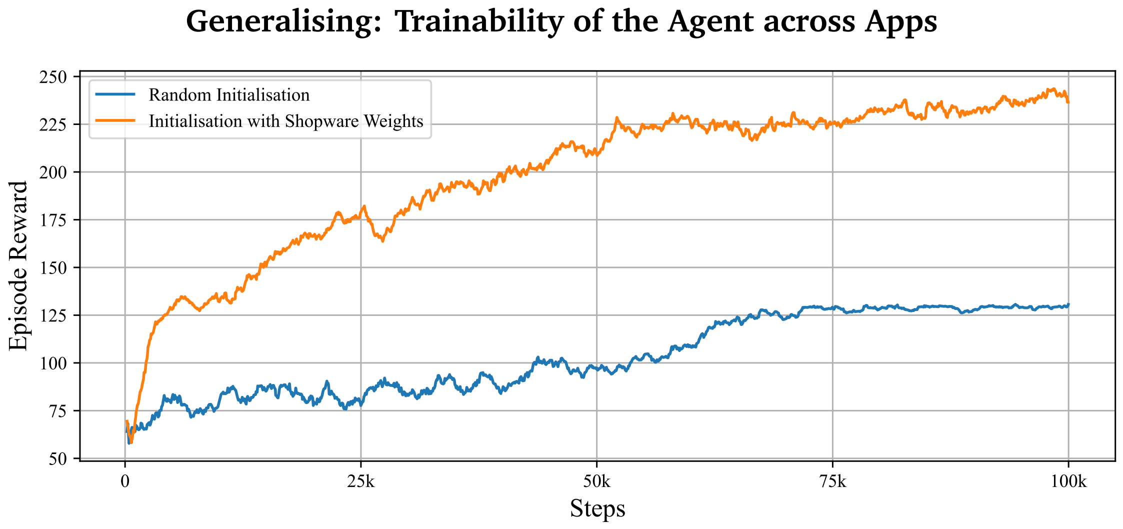

Figure 29: Training Progress for randomly initialised weights and transferred weights from another model. In this case, the weights from the Shopware 6 training run were transferred to the RWA.

The next step towards ensuring the generalisability of the system was to evaluate whether a pre-trained agent could be effectively applied across different web applications. To test this, an experiment was conducted where the initial weights of an agent were set as the existing weights obtained from training on one web app, as opposed to random initialisation. Specifically, the weights of an agent trained for 500k steps on the Shopware 6 e-commerce system were used as the initial weights for an agent trained on the RWA.

The results, shown in Figure 29, indicate a significantly steeper training progress compared to random initialisation. The episode reward after 100k steps is almost double that of the agent with randomly initialised weights. As a point of reference, a reward level of 200 (indicating that the agent is not fully trained but generally capable of exploring the web app) is achieved after roughly 40k steps with the pre-trained agent, while an agent trained with randomly initialised weights requires approximately 290k steps to reach the same reward level. This represents a sevenfold speed-up in training time, reducing it to just one hour using this fine-tuning technique.

These results showcase the potential of transfer learning in expediting the training process and indicate the feasibility of creating a general agent, trained on a variety of applications, that can generalise to various web apps without requiring significant upfront training. With these results, a favourable answer is given to RQ8.

7.3 Baseline Results

The baseline comparison contextualises the methods proposed in this project within the broader field of existing methods and literature. Four baseline experiments were conducted across two web applications. A random algorithm, a Q-learning method from [3], and human testers were all tasked with exploring the web applications under test. The success of each testing method was measured by the number of unique elements discovered over a sequence of 20 clicks within the web app.

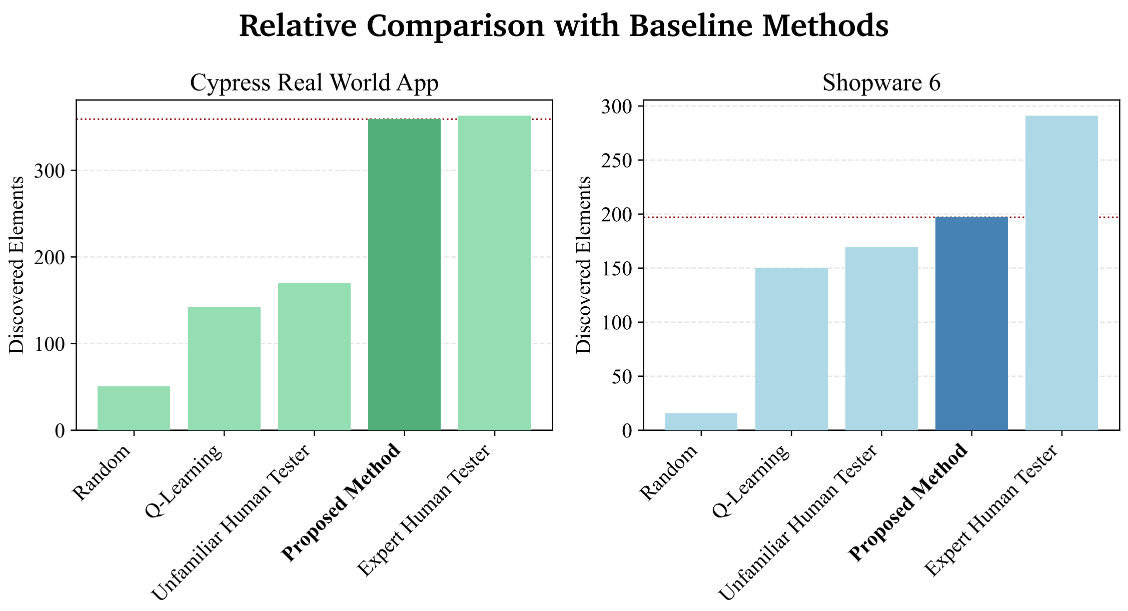

Figure 30: Relative Comparison of the proposed method against baselines. The success of each (automated) testing method is measured by the number of unique elements that are discovered over a sequence of 20 clicks within the web app.

In the case of the RWA, the random algorithm only managed to uncover an average of 51 unique elements. This performance was greatly enhanced with the use of Q-Learning, which discovered around 142 unique elements. An unfamiliar human tester was able to identify approximately 170 unique elements. Remarkably, the proposed method performed excellently, uncovering an average of 359 unique elements, nearly on par with an expert human tester who only managed to uncover an additional 4 elements (363).

For Shopware 6, the results were similarly patterned: The random algorithm was significantly less effective, identifying only about 15 unique elements. However, Q-Learning was more successful, uncovering an average of 150 elements. A tester unfamiliar with the system found 169 unique elements. The proposed method showed a strong performance, uncovering 197 unique elements, although an expert human tester managed to uncover an even larger number for this app, finding approximately 291 unique elements in total.

From these results, it is evident that random methods were ineffective with only 20 steps available, demonstrating poor exploratory performance. In contrast, the Q-Learning method showed improved results, approximating the level of performance of an inexperienced human tester. Remarkably, the proposed method performed on par with an expert tester for the RWA and significantly outperformed an inexperienced human tester in both applications. Finally, the analysis of the experimental results provides an answer to RQ7, RQ8 and RQ9.

8 Evaluation

This section evaluates the project with respect to the project objectives outlined in the introduction of this report (1.3). It does so by evaluating the achievements of this project while putting it in the wider context of existing literature.

The first objective was the formulation of autonomous GUI testing as an RL problem. The approach pursued in this project stands out from the predominant focus on Q-learning-based methods in prior work. Traditional Q-learning approaches usually demand an abstracted state representation that encapsulates detailed information about a discrete set of possible actions in the current GUI view. This requirement often imposes significant constraints on the transferability of these systems between different platforms, such as between Android Apps and Web Apps. It also leads to difficulties in hand-picking suitable information that should be incorporated into the state representation. For example, it might not be immediately clear whether a simple list of clickable elements suffices or what additional information, such as the colour or text of the element, should also be considered.

In contrast, the proposed method in this project utilises screenshots as a means to represent states. This approach mirrors how a human user perceives the GUI (visually) and provides a more elegant and universal solution to encapsulating the state of the GUI. By relying entirely on screenshots as the state representation, the transferability across platforms is enhanced.

Furthermore, the proposed method leverages a continuous action space that mimics natural human interaction with the GUI. Instead of selecting an interactable element from the screen, a user typically clicks at a specific coordinate based on their visual interpretation. This approach makes the system entirely end-to-end and doesn't rely on any assumptions about the internal state information, such as element locations, of the application under test.

The next objective, which demanded the development of a 'compatible RL environment that enables RL agents to interact with the GUIs of browser-based web apps' was also successfully addressed. By implementing a standardised Gymnasium environment, the development of RL algorithms was notably simplified. To the author's knowledge, the code released with this project is the only publicly available implementation of a web browser RL environment, thereby kickstarting future research in this area.

The objective of finding a suitable reward function was realised by implementing different options and performing experiments to empirically evaluate their respective suitability in the context of autonomously testing web apps. At its core, the reward function proposed in this system relies on finding as many possible unique elements while interacting with the GUI. Prior literature by [18] quantified reward by the maximisation of unique URLs visited, which disregards dynamic interactions within a single page of a web app, where the URL may not change. The approach of using a visual pixel-level difference, as utilised by [14] was determined to be ineffective for the problem at hand. Likewise, the change in elements between consecutive states was used by [54] and also proved to be ineffective as it allowed the agent to enter infinite back-and-forth loops without promoting exploration. Using code coverage, as done by [60] is against the philosophy of the presented approach in this project, as it would require access to the source code of the application, breaking the end-to-end nature of the method.

In addition to the reward function, an extensive ablation study was carried out to target the objective of determining suitable architectural choices for the autonomous testing system. In particular, the use of frame-stacking was a novel contribution, which was supported by empirical evidence and has been previously unseen in the related literature. While a related method [18] used RGB images, the proposed system performed grayscaling, which evidently increased performance and sped up training.

The last two objectives concerned the implementation of baseline methods and an overall system evaluation. A random and Q-learning algorithm was implemented and tested against the same set of web apps to establish comparable baseline results. Additionally, human testers were tasked to explore the web app, which yielded a baseline of how effectively humans - the target users of GUIs - are themselves at thoroughly testing a web app. The results allowed for a quantitative comparison and showed that the proposed method outperforms the alternative options for test automation. Furthermore, the RL algorithm surpassed the abilities of an inexperienced tester and reached the levels of an expert human tester in certain scenarios. In terms of exploratory performance, these findings are in line with the results produced by a related method [18]. However, the training speed, enabled through a high-throughput architecture paired with the ability to perform transfer learning from existing weights is unprecedented and heavily simplifies the time and computational effort required to set up the system for a custom application.

8.1 Limitations

Despite the significant achievements and positive results, it is essential to acknowledge the limitations inherent to the proposed method:

Firstly, the current implementation only supports click actions. In contrast, modern GUIs often demand more complex interaction types, including swipes, scrolls, and text inputs. This limitation could potentially restrict access to certain areas of the GUI interface, which might only be navigable via these more sophisticated action types.

Secondly, the conducted tests are entirely unguided. Although the exploration algorithm's primary goal is to cover as many different areas of the web application as possible, it doesn't guarantee the exploration of specific interaction sequences. For instance, in the context of an e-commerce system, testing key functionalities like the product purchasing process is crucial. However, when relying fully on the automated testing system proposed in this project, there is no guarantee that certain desirable interaction sequences get explored. This means that for certain interactions, it may be important to manually define critical GUI interaction sequences using scripted techniques, rather than relying on this approach to test such sequences. It is essential to note that this challenge isn't exclusive to the proposed method but remains a general issue with all known fully automated GUI testing methods.

Finally, the robustness of the trained agent to GUI changes presents an open question. Web applications are frequently subject to updates and alterations that could significantly modify the GUI. These changes might range from subtle visual tweaks to complete interface overhauls. It is currently unclear how well the trained agent can adapt to these drastic changes without undergoing retraining. While the proposed method demonstrates a capacity for transfer learning, which may somewhat alleviate this issue, extensive alterations to the GUI could still necessitate a new training phase. Such a limitation would impose a constraint on the method's practical application, as maintaining up-to-date models that can competently interact with the latest versions of web applications could require a continuous cycle of retraining and deployment.

9 Conclusions and Further Work

This project represents a significant contribution to the field of autonomous GUI testing, addressing several limitations of existing methods while introducing new approaches and insights. Herein, we reflect upon the key design choices, challenges, and iterative improvements that were instrumental to the project's success.

Executing a reinforcement learning algorithm on a real-world problem proved to be remarkably challenging, characterised by a delicate balance between numerous parameters and architectural choices. Poorly chosen hyperparameters or even slight errors in the reward function rendered learning impossible, underscoring the sensitivity of the RL training process. A significant portion of the project's duration was dedicated to fine-tuning this balance.

Importantly, the importance of software design became particularly evident in the early stages of the project. Initial versions of the web browser environment were practically unusable due to poor training throughput, which would have completely hindered training or made it prohibitively slow. Speeding up the training process, through code performance improvements and parallelised training, became a crucial prerequisite to effective experimentation and fine-tuning. Thus, the project has highlighted the necessity of considering computational efficiency alongside algorithmic design when implementing RL in practical settings.

The project's iterative nature, characterised by the formulation of hypotheses and their subsequent testing through training trial runs, proved instrumental in its ultimate success. This process of repeatable experimentation led to hundreds of trial runs performed throughout the project's duration, each contributing to the refined system that was eventually realised.

A number of novel contributions also stem from this project, each serving to enrich the landscape of autonomous GUI testing. A uniquely formulated reward scheme was developed to better align the agent's behaviour with the exploration and testing objectives, which allowed them to navigate modern web apps. In another first, the project saw the deployment of the potent SAC algorithm for automated GUI testing, leveraging its robustness and efficiency for the task at hand. Perhaps most importantly, this project represents the first attempt to investigate the generalisability and transferability of image-based deep RL methods in this context. This exploration opens up new avenues for developing more versatile and universally applicable testing systems, paving the way for further advancements in this field.

9.1 Further Work

While the project has made considerable strides in autonomous GUI testing, there remain several avenues for further exploration and improvement.

Firstly, temporal information could be better potentially integrated into the agent's state representation. While frame-stacking was used in this project, a potentially more expressive and efficient approach could be the use of Long Short-Term Memory networks (LSTMs) as done by [18]. By capturing temporal dependencies more effectively, LSTMs could help the agent to better understand the implications of its actions over time. A comparative study between LSTMs and frame-stacking could shed more light on their respective strengths and weaknesses, thereby informing more effective designs for future systems.

Secondly, the current action space could be expanded to cover a wider range of interactions. The current system is limited to click actions, but modern web applications often demand more complex interactions like scrolls, swipes, or even drag-and-drop actions. By adding support for these additional action types, the system could become even more effective at navigating and testing complex GUIs.

Finally, there is the challenge of handling text fields, which are ubiquitous in modern web applications. One possibility could be to integrate language models such as GPT into the system, which could then be used to generate context-appropriate text inputs dynamically. Such a feature would eliminate the need for manual text input definition, thus further enhancing the autonomy and effectiveness of the testing system. The integration of visual and textual understanding within the same RL agent represents an exciting area for future research.

References

[1] M Abd Al Rahman et al. “Waveguide quality inspection in quantum cascade lasers: A capsule neural network approach”. In: *Expert Systems with Applications* 210 (2022), p. 118421.

[2] Josh Achiam. *Open AI Spinning Up Blog*.

[3] David Adamo et al. “Reinforcement learning for Android GUI testing”. In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation* (Nov. 2018), pp. 2–8.

[4] Adobe Inc. *Photoshop Web Beta*.

[5] Anneliese A. Andrews, Jeff Offutt, and Roger T. Alexander. “Testing Web applications by modeling with FSMs”. In: *Software and Systems Modeling* 4.3 (July 2005), pp. 326–345.

[6] Pölzleitner Anton. *ON SOFTWARE TESTING*.

[7] Autodesk Inc. *AutoCAD*.

[8] N Bauerjee. “Planning, monitoring and evaluating software tests using TEST-PROFI, a multimedia information system”. In: *WIT Transactions on Information and Communication Technologies* 9 (1970).

[9] Svajone Bekesiene, Rasa Smaliukiene, and Ramute Vaicaitiene. “Using artificial neural networks in predicting the level of stress among military conscripts”. In: *Mathematics* 9.6 (2021), p. 626.

[10] Greg Brockman et al. “OpenAI Gym”. In: *arXiv preprint arXiv:1606.01540* (June 2016).

[11] Brooks Fred P. “The mythical man-month Addison-Wesley”. In: *Reading, Massaschusetts* (1975).

[12] *ChromeDriver - WebDriver for Chromium*.

[13] Software Freedom Conservancy. *Selenium*.

[14] Christian Degott, Nataniel P. Borges, and Andreas Zeller. “Learning user interface element interactions”. In: *ISSTA 2019 - Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (July 2019), pp. 101–111.

[15] Biplab Deka et al. “Rico: A mobile app dataset for building data-driven design applications”. In: *UIST 2017 - Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Oct. 2017), pp. 845–854.

[16] *Dockware Managed Shopware Setups*.

[17] Juha Eskonen. *Deep reinforcement learning in automated user interface testing*. 2019.

[18] Juha Eskonen, Julen Kahles, and Joel Reijonen. “Automating GUI testing with image-based deep reinforcement learning”. In: *Proceedings - 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2020* (Aug. 2020), pp. 160–167.

[19] Farama Foundation. *Gymnasium by Farama*.

[20] Vahid Garousi and Junji Zhi. "A survey of software testing practices in Canada". In: *Journal of Systems and Software* 86.5 (2013), pp. 1354–1376.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT press, 2016.

[22] Yuepu Guo and Sreedevi Sampath. "Web application fault classification-an exploratory study". In: *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement* (2008), pp. 303–305.

[23] Tuomas Haarnoja et al. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor". In: *Proceedings of the 35 th International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018* (Jan. 2018).

[24] Shengyi Huang et al. "Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms". In: *The Journal of Machine Learning Research* 23.1 (2022), pp. 12585–12602.

[25] Cypress.io Inc. *Cypress Real World App*.

[26] BrowserStack Inc. *Percy.io by BrowserStack*.

[27] Cypress.io Inc. *Cypress*.

[28] International Telecommunications Union. *Recommendation ITU-R BT.601-7*.

[29] Kateryna Ivanova et al. "Artificial Intelligence in Automated System for Web-Interfaces Visual Testing". In: *COLINS* (2020), pp. 1019–1031.

[30] L. P. Kaelbling, M. L. Littman, and A. W. Moore. "Reinforcement Learning: A Survey". In: *Journal of artificial intelligence research* 4 (Apr. 1996).

[31] Herb Krasner. *The Cost of Poor Software Quality in the US: A 2020 Report*. Tech. rep. CONSORTIUM FOR INFORMATION & SOFTWARE QUALITY, 2021.

[32] Eric Liang et al. "RLlib: Abstractions for Distributed Reinforcement Learning". In: *arXiv preprint arXiv:1712.09381v4* (Dec. 2017).

[33] Zhe Liu et al. "Nighthawk: Fully Automated Localizing UI Display Issues via Visual Understanding". In: *IEEE Transactions on Software Engineering* (May 2022).

[34] Zhe Liu et al. "Owl Eyes: Spotting UI Display Issues via Visual Understanding". In: *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2020), pp. 398–409.

[35] Federico Macchi et al. "Image-based approaches for automating GUI testing of interactive web-based applications". In: *Conference of Open Innovation Association, FRUCT* 2021-January (Jan. 2021).

[36] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. "State-based testing of Ajax Web applications". In: *Proceedings of the 1st International Conference on Software Testing, Verification and Validation, ICST 2008* (2008), pp. 121–130.

[37] Leonardo Mariani et al. "AutoBlackTest: Automatic Black-Box Testing of Interactive Applications". In: *2012 IEEE fifth international conference on software testing, verification and validation* (2012), pp. 81–90.

[38] Leonardo Mariani et al. "Automatic testing of GUI-based applications". In: *Software Testing Verification and Reliability* 24.5 (Aug. 2014), pp. 341–366.

[39] Scott McMaster and Atif M Memon. "An Extensible Heuristic-Based Framework for GUI Test Case Maintenance". In: *IEEE International Conference on Software Testing, Verification, and Validation Workshops* (2009), pp. 251–254.

[40] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *NIPS Deep Learning Workshop 2013* (Dec. 2013).

[41] Mozilla Foundation. *JavaScript MDN Docs*. 2023.

[42] Michel Nass, Emil Alégroth, and Robert Feldt. "Why many challenges with GUI test automation (will) remain". In: *Information and Software Technology* 138 (Oct. 2021).

[43] Bao N. Nguyen et al. "GUITAR: An innovative tool for automated testing of GUI-driven software". In: *Automated Software Engineering* 21.1 (2012), pp. 65–105.

[44] Frolin Ocariza et al. "An empirical study of client-side JavaScript bugs". In: *International Symposium on Empirical Software Engineering and Measurement* (2013), pp. 55–64.

[45] Antonin Raffin et al. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8.

[46] Stuart J Russell. *Artificial intelligence a modern approach*. Vol. Third Edition. Pearson Education, Inc., 2010.

[47] Salma Saber et al. "Autonomous GUI Testing using Deep Reinforcement Learning". In: *2021 17th International Computer Engineering Conference (ICENCO)* (Dec. 2021), pp. 94–100.

[48] Gerald Schermann et al. "An empirical study on principles and practices of continuous delivery and deployment". In: *PeerJ Preprints* (2016), p. 342.

[49] *Shopware 6*.

[50] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (Jan. 2016), pp. 484–489.

[51] Similarweb LTD. *Top Browsers Market Share April 2023*.

[52] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[53] Antero Taivalsaari and Tommi Mikkonen. "The web as an application platform: The Saga continues". In: *Proceedings - 37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011* (2011), pp. 170–174.

[54] Tuyet Vuong and Shingo Takada. "Semantic analysis for deep Q-network in android GUI testing". In: *Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE* 2019-July (2019), pp. 123–128.

[55] *WebDriver W3C Working Draft*. 2023.

[56] Thomas Wetzlmaier, Rudolf Ramler, and Werner Putschogl. "A Framework for Monkey GUI Testing". In: *Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016* (July 2016), pp. 416–423.

[57] Bayya Yegnanarayana. *Artificial neural networks*. PHI Learning Pvt. Ltd., 2009.

[58] *YouTube LLC*.

[59] Yu Zhengwei et al. "A Novel Automated GUI Testing Technology based on Image Recognition". In: *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)* (2016), pp. 144–149.

[60] Yu Zhao, Brent Harrison, and Tingting Yu. "DinoDroid: Testing Android Apps Using Deep Q-Networks". In: *arXiv preprint arXiv:2210.06307* (Oct. 2022).

Appendix

Code Repository

The full code repository is downloadable from GitHub:
`https://github.com/timeo-schmidt/gui-testing`

The repository contains the following:

- Configuration Files
- Training Scripts
- Inference Scripts (to test web apps and obtain JS Error Logs)
- Web Browser Environment
- High-Level Web App Interface
- Baseline Implementations
- Evaluation Scripts
- Experiment Orchestration Scripts

User Guide

The most up-to-date user-guide is published on the GitHub repository in the `README.md` file: `https://github.com/timeo-schmidt/gui-testing`

A brief guide on how to set-up, train and test the repository is outlined in the following steps:

1. Clone the repository
2. Install the python packages associated with the repository:
   (`pip install -e .`)
3. Set up your web application on a local development server
4. Adjust settings in the `config.yaml`, where necessary. A description of the the important configuration options is provided in the following Appendix section.
5. Run the training script: `python3 train.py` (this may take multiple hours, depending on your hardware and weight initialisation)
6. To execute your tested web app testing system, run `python3 test.py` The testing artefacts will be saved to the specified artefact directory, revealing JS Errors and an associated video recording.

Configuration Options

Configuration Option	Suggested Value	Description
`browser_reset_interval`	`1000`	Restart interval of the browser to limit memory build-up.
`viewport_dimension`	`[1280,720]`	Size of the viewport for the browser.
`headless_mode`	`true`	Allows the browser to invisibly run in the background without rendering a window.

Table 4: Overview of Important Web App Interface Configuration Options

Configuration Option	Suggested Value	Description
`test_url`	`"http://localhost"`	Accessibly URL for the web app
`n_envs`	`10`	Number of concurrent, vectorised environments
`frame_stack`	`4`	Height of frame stack, set to 1 for no stacking.
`downscale_size`	`[128,128]`	The downscaling resolution for the screenshots
`log_steps`	`false`	Whether all actions and rewards should be logged
`gray_scale`	`true`	Applies the grayscaling preprocessing stage
`reward_variant`	`3`	Set to 1,2 or 3 corresponding to reward variants
`logarithmic_scaling`	`true`	Whether rewards should be log-scaled
`negative_default`	`-0.01`	Replaces 0 rewards by this number. Set to 0 to disable

Table 5: Overview of Important Environment Configuration Options

Configuration Option	Suggested Value	Description
`max_buffer_size`	`100000`	The size of the replay buffer. Older entries are deleted.
`policy_type`	`"CNNPolicy"`	The type of feature extractor. Replace my "MlpPolicy" to use a flatten layer.
`total_timesteps`	`500000`	The total number of training time steps.
`seed`	`42`	The seed of the random number generators. Keep constant.
`save_freq`	`5000`	The frequency at which weights should be saved.
`log_tensorboard`	`true`	Enables TensorBoard Logging.
`artefact_base_path`	`"./experiments/"`	The save path of the training artefacts, including checkpoints.

Table 6: Overview of Important Algorithm Configuration Options

Configuration Option	Suggested Value	Description
`deterministic`	`true`	Disables noise in the policy by using the distribution mean.
`model_load_path`	`"weights.zip"`	Path of the model .zip file with the saved weight checkpoint
`wait_seconds`	`0.5`	The time to wait between consecutive actions
`log_errors`	`true`	Enables listening to Browser Console logs and saves them.
`record_video`	`true`	Enables video a video recording and saves it.
`artefact_base_path`	`"./test_results"`	The location at which to store the testing artefacts.

Table 7: Overview of Important Algorithm Executor (Inference) Configuration Options