Automated Test Case Generation for Web Applications Using Machine Learning

Riku Lehtonen

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 29.12.2023

Supervisor

Prof. Joni Pajarinen

Advisor

MSc Anna-Liisa Lepistö

Aalto University
School of Electrical Engineering

Aalto University
School of Electrical Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Riku Lehtonen

Title Automated Test Case Generation for Web Applications Using Machine Learning

Degree programme Automation and Electrical Engineering

Major Control, Robotics and Autonomous Systems **Code of major** ELEC3025

Supervisor Prof. Joni Pajarinen

Advisor MSc Anna-Liisa Lepistö

Date 29.12.2023 **Number of pages** 59+4 **Language** English

Abstract

In recent years, web software development has gained significant prevalence. Consequently, the resources and costs required for verifying the functionality of web applications have also increased substantially. Thus, automation in the testing process, such as test case generation, enhances testing efficiency and reduces testing costs. Automatic test case generation employs gathered knowledge of the software to create test steps without direct control by a tester.

In web software development, test generation is challenging as applications frequently consist of multiple complex systems. Therefore, machine learning algorithms have been implemented in test case generation to replicate the manual testing traditionally performed by humans. Recent research has created test cases by exploring the application using search algorithms and directly converting the source code to test cases utilizing language processing. However, previous work has not suggested a generation framework for widely used test automation libraries and machine learning algorithms.

This thesis proposes a framework for transmitting information, such as visible elements and actions, between the machine learning algorithm and the software. For the framework, two machine learning algorithms, Proximal Policy Optimization (PPO) and Online Decision Transformer (ODT), are implemented to benchmark search-based test generation performance. The algorithms optimize test steps for achieving user-provided test objectives, such as logging into a website.

Results indicate that the framework can support the algorithms for exploration-based test generation for web applications. The PPO can optimize the test steps towards various test objectives. The ODT efficiently clones the behavior from collected trajectories, for example, previously created test cases. This thesis also analyzes solutions to address potential scalability challenges in the algorithms used, particularly as the number of available actions increases in larger applications. Furthermore, the future aim for these algorithms is simultaneous and rapid test case generation across multiple applications.

Keywords Testing, Test Generation Framework, Machine Learning, Proximal Policy Optimization, Online Decision Transformer

Aalto University
School of Electrical Engineering

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä	Riku Lehtonen
Työn nimi	Automaattinen testitapausten luonti web-sovelluksille koneoppimista hyödyntäen
Koulutusohjelma	Automation and Electrical Engineering
Pääaine	Control, Robotics and Autonomous Systems **Pääaineen koodi** ELEC3025
Työn valvoja	Prof. Joni Pajarinen
Työn ohjaaja	MSc Anna-Liisa Lepistö
Päivämäärä	29.12.2023 **Sivumäärä** 59+4 **Kieli** Englanti

Tiivistelmä

Web-sovellusten kehittämisen suosio on kasvanut viime vuosina, mikä on lisännyt sovellusten testaamiseen käytettyä työmäärää merkittävästi. Testausprosessin työmäärän vähentämiseksi automaatiota on hyödynnetty esimerkiksi testitapausten luomisessa. Testejä voidaan suunnitella sovelluksesta kerättyjä tietoja hyödyntämällä.

Testien automaattinen luominen web-sovelluksille on haastavaa, sillä sovellukset koostuvat yleensä useista järjestelmistä. Koneoppimista on hyödynnetty testien laatimisessa, koska algoritmit voidaan opettaa testaamaan sovellusta ihmisen tavoin. Viimeisimmät tutkimukset ovat käyttäneet sovelluksen tutkimiseen perustuvaa testien luontia ja tuottaneet testejä myös suoraan sovelluksen lähdekoodista. Aikaisemmat tutkimukset eivät kuitenkaan ole ehdottaneet ohjelmistokehystä testitapausten laatimiseen, jota voidaan käyttää yleisten testiautomaatiokirjastojen ja koneoppimisalgoritmien kanssa.

Diplomityössä kehitettiin ohjelmistokehys koneoppimisalgoritmeja varten, mikä mahdollistaa sovelluksen tilan seuraamisen ja toimintojen suorittamisen testattavassa järjestelmässä. Kehyksen toiminta varmistettiin toteuttamalla Proximal Policy Optimization (PPO) ja Online Decision Transformer (ODT) -algoritmit, ja vertaamalla testitapausten luonnin tehokkuutta web-sovelluksessa. Algoritmien tavoitteena on löytää parhaat testiaskeleet, jotka saavuttavat käyttäjän määrittelemän testitavoitteen. Tavoite voi olla esimerkiksi kirjautuminen sovellukseen.

Tulokset osoittivat, että kehys pystyy tuottamaan testitapauksia web-sovellukselle käyttämällä molempia algoritmeja. PPO-algoritmin avulla voidaan oppia testiaskeleet, jotka saavuttavat käyttäjän asettaman testaustavoitteen. ODT-algoritmi puolestaan pystyy luomaan testitapauksia yhtä tehokkaasti käyttämällä kerättyjä testitapauksia. Tulosten perusteella algoritmeja on jatkossa kehitettävä eteenpäin, jotta niitä voidaan käyttää sovelluksissa, jotka sisältävät suuren määrän eri toimintoja. Lisäksi tulevaisuudessa on tavoitteena, että algoritmeja voidaan hyödyntää samanaikaisesti ja tehokkaasti monessa eri web-sovelluksessa.

Avainsanat Testaus, Testitapausten luominen, Ohjelmistokehys, Koneoppiminen, Proximal Policy Optimization, Online Decision Transformer

Preface

I extend my gratitude to Tietoevry and Anna-Liisa Lepistö for granting me the opportunity to pursue a thesis in the field of machine learning, which closely aligns with my interests and is particularly relevant in software testing. I would also like to thank my supervisor, Joni Pajarinen, for his valuable comments that have improved the thesis. I also appreciate the feedback from my friends Jaakko and Esa. Lastly, I'm grateful for the ongoing support from Siiri Lapila and from my mother, Hannele Lehtonen.

Tämä diplomityö on omistettu Sinikka Lehtosen muistolle.

Helsinki, 29.12.2023

Riku Lehtonen

Contents

Symbols and Abbreviations

Symbols

S	State Space
A	Action Space
S_t	State at Time t
A_t	Action at Time t
R_t	Reward at Time t
G_t	Return at Time t
$\boldsymbol{x}_i$	Feature Vector
y_i	Label
N	Number of Data Points
γ	Discount Rate
θ	Policy Parameter Vector
α	Learning Rate
π	Policy
w	Weight Vector
b	Bias

Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
BERT	Bidirectional Encoder Representations from Transformers
DNN	Deep Neural Network
GPT	Generative Pre-trained Transformer
GUI	Graphical User Interface
HTML	HyperText Markup Language
LLM	Large Language Model
MDP	Markov Decision Process
ML	Machine Learning
MSE	Mean Squared Error
NLP	Natural Language Processing
ODT	Online Decision Transformer
PPO	Proximal Policy Optimization
SDLC	Software Development Lifecycle
SUT	System Under Test
UI	User Interface

1 Introduction

Testing is an integral part of the lifecycle of software development. To prevent defects as early as possible, testing is often performed at multiple levels and different stages of software development [1]. Accordingly, software testing has frequently been regarded as a time-consuming and labor-intensive process. Thus, testing activities have been estimated to account for up to 50% of software project costs [2]. These costs can further increase when testing critical applications, such as safety-critical real-time systems [3]. Testing critical systems, including a control interface in a vehicle, is essential since a failure or malfunction could compromise user safety.

As a subset of software development, web development has become increasingly prevalent in recent years, as demonstrated by the rapid growth in cloud computing markets [4]. Subsequently, testing has become increasingly difficult as the amount and complexity of web-based software grows. Web applications often require repetitious testing on multiple platforms. To address the challenge, tests could be automated, thus significantly reducing the time and effort needed for repetitive testing while increasing its consistency [1].

However, test-automation has its limitations often demanding significant effort to achieve benefits. Developing and maintaining tests not only requires time but can also distract from any testing objectives by forcing the focus to be on the automation rather than on executing the tests [5]. Moreover, test-automation does not replace the more important task of manually exploring the application to find new bugs. Rather than searching for new bugs, the purpose of test automation is to effectively detect unintended changes in an updated system [1]. Eventually, repeating the same test cases does not help find new defects.

To overcome these difficulties, research in test case generation has been increasingly popular [6]. The generation process selects the actions similarly to a human manually testing the application then combines the actions into a test case. By commonly utilizing machine learning, test case generation aims to reduce the challenges of test-automation by autonomously exploring the test environment and creating test cases [7]. The generation approaches can be categorized by the test levels, such as component and system testing, or the test types, including functional and non-functional testing [1].

Another way to divide the testing approaches is access to and knowledge of the internal structure of the application. White-box or structural test generation creates the test cases using the structure or model of the software, such as source code and documentation, and it often targets high code coverage [8]. Recent advances in white-box test generation include using large language models by Tufano et al. [9] to create test cases with improved readability by mining the source material of the software. Black-box test generation often utilizes search-based methods without knowing the software structure [8]. Black-box methods have been successfully used in desktop [10], mobile [11], and web applications [12].

Despite vast research in test generation, a common framework for executing multiple machine learning algorithms for test-automation has not been suggested. Moreover, the recent approaches have not provided a simple way to expand the algorithm for numerous test-automation libraries and platforms. An ideal test generation framework should be capable of covering multiple machine learning algorithms and should be uncomplicated to scale up.

1.1 Objectives and Scope

This thesis aims to develop an easy-to-use framework for test generation to support multiple machine learning algorithms. Two machine learning algorithms, Proximal Policy Optimization (PPO) and Online Decision Transformer (ODT), are selected and implemented for test generation to verify the functionality of the framework. The thesis limits the algorithms to black-box test generation by exploring the application by the test objective set by the user. In addition, generation is confined to GUI-based web application testing. The scope has been selected for two reasons. Firstly, to reduce required testing efforts in the growing domain of web application development. Secondly, to develop black-box test generation methods since white-box testing of web applications is only sometimes possible.

The test generation is evaluated in a benchmarking application designed to resemble a fully featured online store. The secondary goal is to compare the algorithms to understand the effectiveness of the selected black-box generation methods. With the evaluation, the aim is to answer three research questions.

RQ1: How can machine learning models be optimized to generate script-based test cases using a black-box approach?

RQ2: How does the performance of black-box test generation compare between the selected machine learning algorithms?

RQ3: What is the coverage of the test cases generated by the selected machine learning algorithms?

The questions aim to identify the factors affecting the test generation when using the selected machine learning algorithms to generate test cases for web applications. In addition, the thesis compares the test generation performance and coverage to assess the benefits and potential areas for future improvement of both algorithms.

1.2 Structure

The rest of the chapters are structured as follows. The second chapter explains the background of software testing, the theory behind machine learning algorithms, and previous research on test generation. The third chapter describes the research material and methods employed in the test generation approach. The fourth chapter presents the benchmarking results, which include the training and the evaluation of the test generation. The fifth chapter focuses on discussing the thesis results reflecting the research questions. The sixth and final chapter concludes the thesis and summarizes the results of the work.

2 Background

The chapter will cover the background of software testing, the theory behind machine learning related to the utilized algorithms, and previous research on test generation. The background of testing, machine learning, and test generation is a broad topic. Therefore, the chapter is limited to only fundamental and the most significant information about the methods used.

2.1 Software Testing

Software testing is a process that ensures the software functions as intended and aims to prevent any unexpected behavior [13]. The process consists of several parts that not only verify the correct behavior but also validate the requirements set by users and stakeholders [14]. In short, testing evaluates the quality of the system under test (SUT) and reduces the risk of failure. Software failures can cause effects ranging from insignificant, such as bad user experience, to significant, including injury or death. Although testing reduces the chances of failure, it's impossible to prove that software is error-free in most cases [14]. Therefore, a good mindset for testing is to find as many errors as possible [13].

Testing is an essential part of software development. A software development lifecycle (SDLC) model describes the software development process on a high level [2]. SDLC models include different phases, such as planning, development, and testing, that are organized into a systematic development process [15]. The relation and order of these phases often varies between different models. However, testing should be integrated into every development activity to ensure comprehensive quality control [2].

The popularity of iterative SDLC models such as Scrum has continuously risen in the last decade [16]. These agile development models divide the process into small parts where the software is created in increments and improved by customer feedback [17]. In Agile development, the incremental modifications and additions to existing software require repetitive regression tests that have also been updated based on the new requirements. Consequently, various levels of testing need to be considered according to the areas impacted by these modifications.

2.1.1 Test Levels and Types

Testing can be divided into different test levels. Test levels describe the stages of testing from individual components to the whole system [14]. Commonly used test levels are component or unit testing, component integration testing, system testing, system integration testing, and acceptance testing [1]. The component testing focuses on the individual components, and the component integration tests the connections between different units in the system. When moving to higher levels, system testing and system integration testing assess the complete system and integrations between system interfaces [14]. At the highest level, acceptance testing validates the system's behavior before deployment. Test levels are tied to different deployment phases and related to activities within the SDLC [1].

In addition to test levels, different test types are needed to describe the test process. Testing can be categorized in many ways, but the usual types are functional and non-functional testing. Functional testing evaluates the behavioral characteristics and components of a system [14]. In contrast, non-functional testing covers other software quality characteristics such as performance, reliability, and security [1]. Integrating functional and non-functional testing approaches gives a more comprehensive and robust testing process, ensuring the proper function of the systems and overall quality.

Testing can be also categorized as white- and black-box testing. White-box or model-based testing uses the implementation and structure of the system to derive the test cases [13]. White-box methods can utilize the software source code to test while aiming to reach as high code coverage as possible. Black-box testing relies only on the system specifications without knowledge of the internal structure of the system [13]. In short, white-box testing is structured testing, and black-box is specification-based testing [1].

Combining the test levels and types forms a robust testing approach to ensure software quality and functionality in the SDLC. Testing should also focus on previously tested software features if the software is updated. Therefore, testing should include confirmation and regression testing. Confirmation testing ensures that new or fixed features meet the specifications [14]. Regression testing confirms that the features tested by confirmation tests in previous iterations are working correctly [2]. The repetition of the test cases in regression testing makes it often a candidate for test automation [1].

2.1.2 Test Automation

Test automation reduces the cost and time spent on testing by utilizing software to perform the tests [18]. Automating tests has become a frequently used method in agile software development. Automation enables rapid feedback to developers and can improve the efficiency of test-driven development and acceptance testing [19]. For instance, tests could be scheduled or automatically initiated after software modifications to provide updates to stakeholders, who are the people affected by the software project. Additionally, investing resources in building automated tests improves test coverage even with fewer testers engaged in manual testing [20].

In test automation, the main questions are about which tests to automate and selecting the methods for automating these tests. Planning which test to automate occurs before the test design. Before choosing the strategy, the tester must consider factors such as benefits, risks, and costs [5]. The SUT, testing tools, and the existing testing process can affect the required effort to implement the test automation [21]. Therefore, planning is crucial before moving to test implementation.

Test automation can be viewed as an iterative process of test generation, definition, and execution [5]. Test generation is the layer that manually or automatically designs test cases. The generation layer is followed by the definition phase that implements the tests. Finally, the execution layer runs the created test cases. When designing the automated tests, the tester can consider the test types and levels. Test automation can account for different test types, such as non-functional performance testing or requirement-based functional testing utilizing black-box test design [21]. Furthermore, the tester could consider the test levels including integration and system tests [5]. For example, in web applications, testing could be split into user interface (UI) and application programming interface (API) tests while taking into account the functional and non-functional requirements.

The definition phase of the automation process involves the implementation of the test cases. Multiple methods exist for developing the test script and setting the test data. A basic form of test automation is linear scripting, where the manual test steps, for example, clicking a button and typing text, are transformed into a script and played back by the automation [21]. A tester can further structure a script. In data-driven testing, the input data, such as generated user details, are usually separated from the test step sequence, improving the reusability of the test cases [21]. A keyword-driven testing technique often builds on the data-driven approach by splitting the test steps into detailed keyword instructions that a control script executes. For example, in web application testing, a keyword could be a function that clicks a selected element. The control script uses the click keywords to navigate in the web application.

Although test automation reduces the need for manual regression testing, the stakeholders must consider the associated costs and challenges of implementing automation. Building automated tests might require high investment costs, and maintaining the test cases is often critical for the development lifecycle [18]. Testers must develop and refactor test cases when changes occur in the SUT [22]. As a result, the test cases require modification to accommodate the new requirements.

Another reason why tests must be regularly developed relates to the basic principles of software testing. While running existing regression tests aids in identifying bugs caused by recent changes, it may leave new, unrelated bugs undetected [14]. In short, the testers must create new tests to find new bugs. As these challenges raise costs for test automation, the amount of research for tackling the challenges has increased over the years [7]. As a result, machine learning methods have garnered significant interest in automated testing [8].

2.2 Machine Learning

Machine learning is a subfield of artificial intelligence (AI) that focuses on problems that are particularly challenging to encapsulate within human-designed algorithms [23]. Machine learning can be described as a process where a dataset is collected, and a statistical model is built from the dataset [24]. The model-building process is called learning since the model is learned from the collected dataset. When learning is complete, the statistical model is used to solve the initial problem, for example, by estimating values.

A dataset can be considered a collection of data points representing text documents, signals, images, and numerous other types depending on the application domain [25]. Many of the machine learning methods try to create an estimate for a quantity of interest using data points. Therefore, the properties of the data point can be divided into different groups, for example, features and labels. Features are the properties of the data that are easy to measure or compute, and labels are the high-level properties that the machine learning algorithm tries to predict [25]. When forecasting housing prices, a couple of features can be the house size and location. The label is then the housing price.

After collecting a dataset, the next crucial step in training a machine learning model is selecting an appropriate learning method. The methods can be classified by the type of supervision and how the features and labels are used. [26] Different methods are used for different kinds of tasks, such as classification and regression problems [27]. Machine learning can also be utilized for learning the best policy, which is the strategy for decision-making [28].

2.2.1 Types of Machine Learning

The learning process of machine learning can be categorized into four different types: supervised, unsupervised, semi-supervised, and reinforcement learning [24]. In supervised learning, a dataset has features that have been associated with a label [28]. Therefore, the dataset can be thought of as a collection of labeled examples $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N}$ where each element $\boldsymbol{x_i}$ in N is called a feature vector [24]. In short, the target value is known for each data point. Supervised learning is commonly applied to regression and classification including tasks such as spam filtering [26].

In unsupervised learning, the dataset is a collection of unlabelled examples $\{(\boldsymbol{x}_i)\}_{i=1}^{N}$ [24]. Since each feature vector's labels are unknown, the model must find patterns and structures in the data without explicit guidance from labeled examples. Unsupervised machine learning can be used for clustering, which groups the data points based on the feature vectors [26].

Supervised and unsupervised learning are not entirely separated; their boundaries can be blurry. Many machine learning techniques can be utilized for both tasks interchangeably [27]. Sometimes, supervised problems can be converted to unsupervised problems and vice versa. Other variants exist, such as semi-supervised learning, where the dataset contains labeled and unlabeled data points [24]. Overall, the categories are not strictly defined, but they represent different approaches.

2.2.2 Reinforcement Learning

Reinforcement learning stands out from supervised and unsupervised learning since it employs a learning agent. The agent makes choices in a particular environment to reach a defined objective. The objective is often the desired outcome, such as finding as many bugs as possible, when the agent interacts with the environment. While interacting with the environment, the agent senses the state of the environment and selects an action based on the learned strategy for decision-making, also called a policy π [28]. A state describes the possible configuration of the system or environment.

Fundamentally, the agent tries to capture a real-world problem by observing the state, selecting an action, and optimizing the agent's behavior toward the objective described by a reward signal. The reward signal assesses how successfully the objective is achieved. The interaction is called a Markov decision process (MDP), which is displayed in Figure 1. An MDP models a decision-making process in which the optimization of the policy is referred to as learning or training [28].

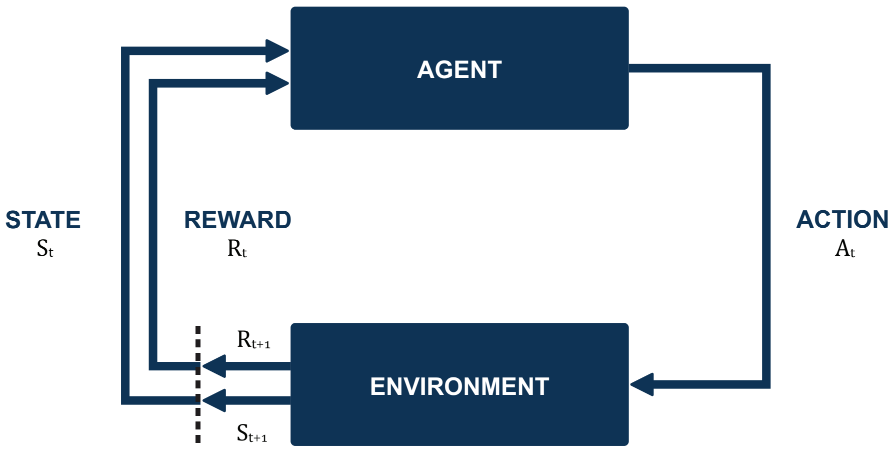


Figure 1: Markov decision process [28]

The agent observes the system state S_t as a vector of features [24]. In the context of web applications, the feature vector could include binary flags of the visibility of website elements. Unlike other machine learning methods that use labels, RL relies on a reward signal R_t, aiming to maximize the reward by selecting the best possible action A_t in a given state [28]. Similarly to supervised learning, reinforcement learning aims to create an optimal policy [24]. In contrast, the optimal step in RL is determined by trial and error in a dynamic environment. Therefore, a finite MDP creates a trajectory of states, actions, and rewards.

$$S_0, A_0, R_1, S_1, A_1, R_2, \; ... \; S_{t-1}, A_{t-1}, R_t. \tag{1}$$

In reinforcement learning the agent aims to maximize the sum of the rewards collected in a trajectory. A reward discounting can be added to increase the preference of current rewards compared to the future rewards [29]. The return of of a trajectory is then

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+2} = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{2}$$

where γ is called a discount rate. When the discounting factor is closer to one, the agent is more patient to wait for future rewards. When reducing the factor towards zero, the future rewards are viewed as more insignificant [29].

In an MDP, the agent makes decisions based on the expected return from a state, which indicates the advantage of transitioning to that state. The estimations of the expected return can be achieved using a value function under a policy π [28]. The policy maps the probabilities of all possible actions to each state. It answers a question of $\pi(a|s)$, which denotes the probability of an action given a state. The state-value function is then

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\middle|S_t = s\right], \tag{3}$$

where $\mathbb{E}_\pi$ denotes the expected value at any time step t when the agent follows policy π [28]. Furthermore, the function can be expanded into actions. The expected return can be calculated using an action-value function

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\middle|S_t = s, A_t = a\right], \tag{4}$$

which depends on an action taken in a given state using a policy [28]. The value functions $v_\pi(s)$ and $q_\pi(s,a)$ can be estimated by exploring the environment. Simply by keeping the average return of the visited states, the average will converge to the state value $v_\pi(s)$ when the state visitation count approaches infinity [28]. In general, finding the value function often requires dynamic programming, which recursively breaks the problem into smaller sections and calculates the optimal solution of the sections [29]. Importantly, the objective is to find optimal policies that lead to the highest expected value. The optimal policies share the same state-value function

$$v_*(s) = \max_\pi v_\pi(s) \tag{5}$$

and also the same optimal action-value function

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{6}$$

for all states and actions in the state [28]. One of the algorithms for estimating the optimal action-value function is Q-learning defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]. \tag{7}$$

Here, $Q(S_t, A_t)$ estimates $q_*(s,a)$ and α is the learning rate [29]. The algorithm selects actions from the $Q(S_t, A_t)$ and updates the function in each iteration [28]. The function $Q(S_t, A_t)$, representing the collected values for state-action pairs, can be stored in a two-dimensional lookup table called a Q-table.

2.2.3 Policy Gradient Methods

Sometimes, it is not feasible to learn the policy by memorizing the values of all the states. The state-action tables required for calculating value functions, which might also include transition probabilities to states, are size $O(|S||A|)$ [29]. Here, S is all possible states called a state space, and A is all possible actions called an action space. For instance, if a feature vector contains ten binary flags and action space includes ten actions, the table size for storing the values for state-action pairs is $2^{10} * 10 = 10240$. The sizes will rapidly increase with larger states and actions. The tables are often sparse, containing many undefined state-action pairs. For larger state and action spaces, even the tables with infrequently defined values become excessively memory-consuming [29].

To resolve this problem, it's not necessary to consult a value function for the action selection. Instead, it's possible to learn a parameterized policy for the action selection and use the value functions only to learn the policy parameter vector θ [28]. The learning is possible with an optimization method, such as a gradient ascent, which aims to maximize the total reward input. With policy gradient methods the parameter vector is optimized by executing the policy and observing the return. In gradient ascent, each of the parameters is updated in increments while moving towards the maximum total reward value [29]. The update equation is expressed as

$$\theta_{t+1} = \theta_t + \alpha \nabla \hat{J}(\theta_t), \tag{8}$$

where θ_t denotes the vector of policy parameters at iteration t, α is the learning rate, and $\nabla \hat{J}(\theta_t)$ signifies the estimated gradient of the performance function J with respect to θ_t [28]. The equation describes the update rule for the policy parameters using gradient ascent to maximize the performance measure. The policy gradient methods often use the expected return as a performance measure [29].

The policy gradient methods use commonly stochastic policy $\pi(a|s,\theta)$, which specifies the probabilities of actions taken in a given state [28]. The policy is dependent on the policy parameter vector. In discrete action spaces, a frequently used method of converting the parametrized numerical preferences into action probabilities is a soft-max function

$$\pi(a|s,\theta) = \frac{e^{h(s,a,\theta)}}{\sum_b e^{h(s,b,\theta)}}, \tag{9}$$

where $h(s,b,\theta)$ is the numerical preferences [28]. Action preferences can be parametrized with methods such as neural networks where the vector θ contains network weights.

The benefit of a stochastic policy is that it enables a transition from exploration to exploitation. The transition means switching from probability-based action selection to selecting only the best actions [28]. In a discrete action space where only one action is selected, the action can be sampled from categorical distribution. During the learning process, the policy often exhibits a progressive shift towards determinism focusing more on exploitation. Stochasticity is also possible in continuous actions by sampling a normal distribution and controlling the mean and variance [30]. Balancing exploration and exploitation remains an open problem in reinforcement learning. Multiple algorithms have been developed in response to the challenge, including ϵ-Greedy, Simulated annealing, and Probability matching [31].

2.2.4 Feedforward Neural Networks

Artificial neural networks (ANN) are used for non-linear function approximation, for example, in the case of a value function [28]. A feedforward neural network, also known as a multi-layer perceptron (MLP), is the basic artificial neural network architecture [27]. The structure of the network consists of layers of nodes or neurons. The number of neurons in a layer can vary, and deeper networks with more layers lead to what is known as deep learning. The information flows through the layers from the input layer through hidden layers and finally reaches the output layer, hence the name feedforward [27]. Each neuron in a layer has direct connections to the neurons of the subsequent layer. An example of a feedforward neural network is displayed in Figure 2.

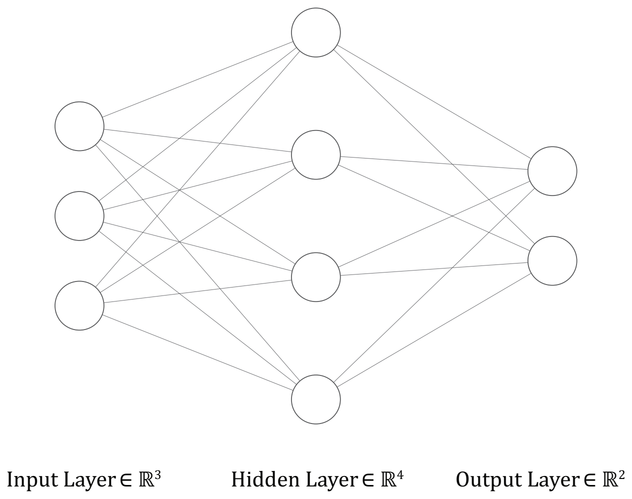


Figure 2: Example of a feedforward neural network with one input, output, and hidden layer.

In a feedforward neural network, each connection between neurons includes a real-value weight, which is adjusted during the learning process [28]. These weights are adjusted based on the error of the output compared to the expected result. The neuron can be a linear function

$$f(x; w, b) = x^T w + b, \tag{10}$$

where x is input vector, w is weight vector and b is a selected bias [27]. w and b belong to the policy parameters θ_t. Typically, a non-linear activation function such as rectifier nonlinearity or logistic function is used to produce the output of the neuron [28]. In the final layer, the neurons form a chain of functions $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$ [27]. An optimization method such as the gradient ascent is used to update the policy parameters in θ_t. The learning process typically uses a technique called backpropagation with the optimization method [28]. The loss is determined by progressing forward in the network, that is, by forward propagation. The loss can be then used to calculate the gradients using the chain rule from the output layer backward to the input layer [27].

2.2.5 Transformer Model

The transformer model, introduced by Vaswani et al. [32], is a type of neural network architecture that is commonly used for sequence transduction tasks such as translation, text summarization, and question answering [33]. The transformer architecture allows for more significant use of parallelization due to attention mechanisms, which enables the model to process different parts of the input data simultaneously [32]. Consequently, transformers are highly scalable and can be trained on large datasets. In addition, attention allows focusing on different parts of the input sequence, meaning that the dependencies on longer inputs can be efficiently connected [32]. The transformed model is displayed in Figure 3.

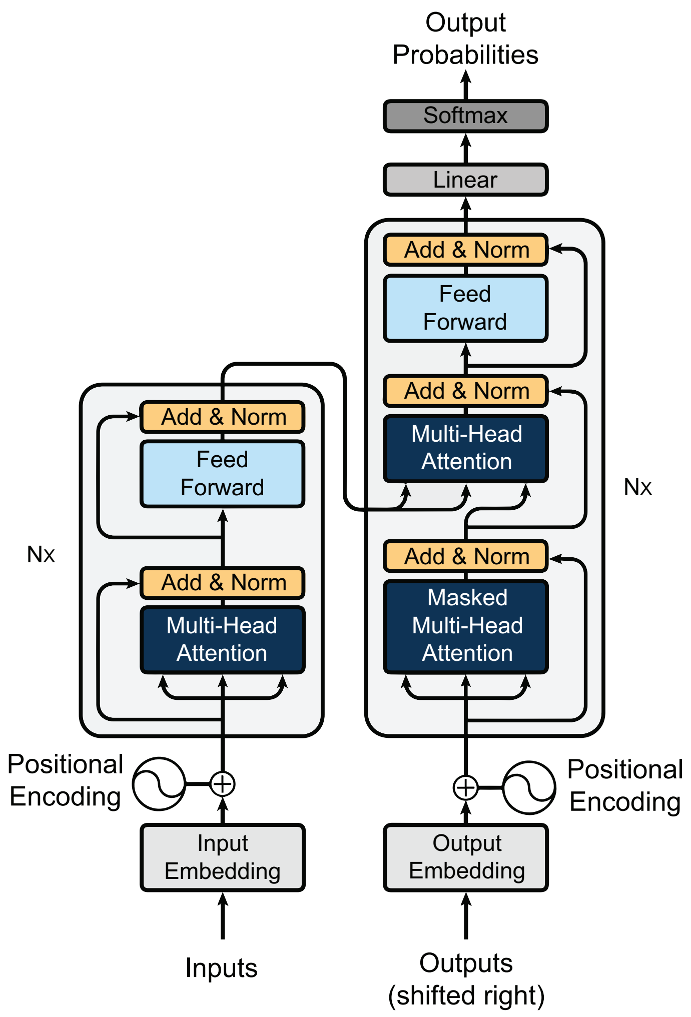


Figure 3: Transformer model introduced by Vaswani et al. [32]

At its core, the decision transformer model is based on an encoder-decoder structure. The encoder and decoder function is usually done by having the encoder extract a fixed-length vector representation from the input sequence, and the decoder translates the vector representation into a correct variable-length output sequence [34]. In the transformer model, the input and output sequences are first transformed into continuous token vectors, into which the positional information of the token locations is embedded[32]. The input sequence is then moved to the encoder layer and the output sequence to the decoder layer. The encoder layer consists of a stack of N identical layers, each containing two sub-layers, a multi-head self-attention mechanism, and a position-wise fully connected feed-forward network [32]. The decoder layer includes N identical layers but also has an extra attention layer for the encoder output. There are also additional residual connections and layer normalization after each sub-layer inside the encoder and the decoder. The decoder also uses masking and output embedding offsetting to ensure that the predictions are made only from the preceding data [32]. The model outputs the next-token probabilities using a Softmax layer.

2.3 Test Generation via Machine Learning

In recent years, the methods of improving testing efficiency and automating the testing process have been a frequent research topic. Hence, multiple machine learning solutions and tools have been created to support the testing process. The most popular research areas have included test case design, evaluation, prioritization, and refinement [6]. The generation phase of test automation has been one of the most explored topics in software testing [7]. The methods often utilize machine learning in transforming the collected paths and data to test cases [8].

The different approaches can be categorized into black-box and white-box test generation. Both methods are frequently used in areas such as system, unit, and GUI test generation and different platforms such as mobile and web applications [8]. Black-box methods often employ exploration and stochastic testing to generate the test steps [8]. In stochastic testing, the difference from traditional deterministic scripts is the probability of actions. Stochastic testing creates a probabilistic model of the SUT and uses the model to produce the test steps [35]. White-box methods use software-related information, such as the codebase, to build the test cases [7]. Recent advancements in large language models and language processing have led to new approaches in white-box test generation [9].

Although both test-generation approaches have been thoroughly explored, a framework for generating tests that can run multiple machine learning algorithms has not yet been suggested. Particularly in black-box test generation, the algorithm often needs information on the action and states of the SUT. In addition, the information on the reached states could benefit white-box generation.

2.3.1 Test Case Generation Approaches

The simplest methods in black-box test generation include random testing [7]. Random testing creates inputs by randomly sampling the program's input space and using the resulting trajectories to build error-revealing test cases [36]. Random testing is suitable when the source code or other documentation is unavailable. Variants of random testing have been created to enhance the performance, including feedback-directed and adaptive random testing [7]. In feedback-directed testing, the test cases are generated or adapted based on feedback from prior test executions to improve test effectiveness [37]. Arteca et al. [38] used feedback-directed random testing for asynchronous APIs to solve challenges in determining callback expectations.

Adaptive random testing differs from feedback-directed testing. Adaptive methods ensure that the created test cases are evenly distributed across the input domain rather than at random, enhancing the likelihood of finding defects [7].

Another black-box generation method, which is related to random testing, is search-based testing. The search-based testing uses optimization methods to search the test steps that maximize the test objectives and minimize the test cost [39]. The searching concept relies on having a fitness function directed towards a testing objective, such as maximizing code coverage. The algorithms used in the search use evolutionary techniques such as mutation and machine learning agents [40, 41]. Alshahwan et al. [42] used a search-based method for web application testing by mutating the input and using a heuristic to pick the best option.

Machine learning has been increasingly applied to the test generation process and has shown efficiency improvements [8]. Yasin et al. [43] have created a test case generation tool with Q-learning using states and an external heuristic to select the best action. Recently, search-based methods have been combined with Curiosity-Driven Q-learning by Zheng et al. [44] and Pan et al. [45] with approaches aiming to maximize code coverage. Curiosity-driven methods achieved complex test trajectories, higher code coverage, and improved failure detection rate compared to traditional search-based methods [44]. Storing the policy in simple Q-tables might limit the state and action space size. Therefore, function approximations, such as neural networks, have been used in state exploration techniques. For example, Li et al. [46] used human interaction traces to imitate the user's behavior in the application by training a deep neural network (DNN) model.

Some test generation methods go beyond exploring the application without knowledge of the software. White-box or model-based test generation and symbolic execution use the codebase for building the test cases. White-box test generation derives the test cases from a formal model that describes the system's behavior or specifications [7]. The formal models have various ways to describe the system, including scenario, state, and process-oriented notations [7]. Symbolic execution differs from the model-based approaches. Symbolic execution is a software analysis technique that operates on program code, treating its inputs as symbolic values rather than concrete ones [7]. The method allows for the exploration of multiple execution paths simultaneously, enabling the identification of possible code vulnerabilities, bugs, and undesired behaviors under various input conditions. Besides White-box generation and symbolic execution, natural language processing (NLP) using machine learning methods has been a prevalent approach for defining test cases. The NLP methods will be in section 2.3.2.

Machine learning has been used widely in test generation for different testing levels, types, and platforms. Studies have broadly used white and black-box methods for end-to-end testing [8]. End-to-end tests validate the entire workflow of a system, from individual components to system level, ensuring comprehensive coverage of the system's functionality. State-based GUI test generation has been a popular research topic in end-to-end testing. Evaluation platforms have mostly covered mobile applications but also included web applications [8]. Despite the evaluation environment, the common goal for the test generation algorithms has been increasing the test coverage metrics [8].

2.3.2 Natural Language Processing in Test Generation

Natural Language Processing (NLP) is a subfield of AI that focuses on enabling computers to interpret and respond to a human language [47]. In test case generation, NLP techniques could be advantageous to understand the testing needs and the context from sources such as the requirements, codebase, and documentation. As a result, NPL algorithms have been used in black-box and white-box testing [48, 49].

In black-box testing, Khaliq et al. [48] used a language transformer to translate UI descriptions into test cases. Although black-box generation can be successful, the recent methods have focused on white-box testing while gathering more information from the application to increase the test coverage [49]. The methods have used large language models (LLM). LLMs represent deep learning architectures, often utilizing transformer models trained with large datasets [50].

Schäfer et al. created a unit test generation tool called Testpilot that utilizes LLMs for automatic test generation [49]. Testpilot uses a generative pre-trained transformer (GPT) language model Codex to generate unit tests for JavaScript testing framework without necessitating further training or few-shot learning from existing test cases [49]. The generation process of the Testpilot tool is described in figure 4.

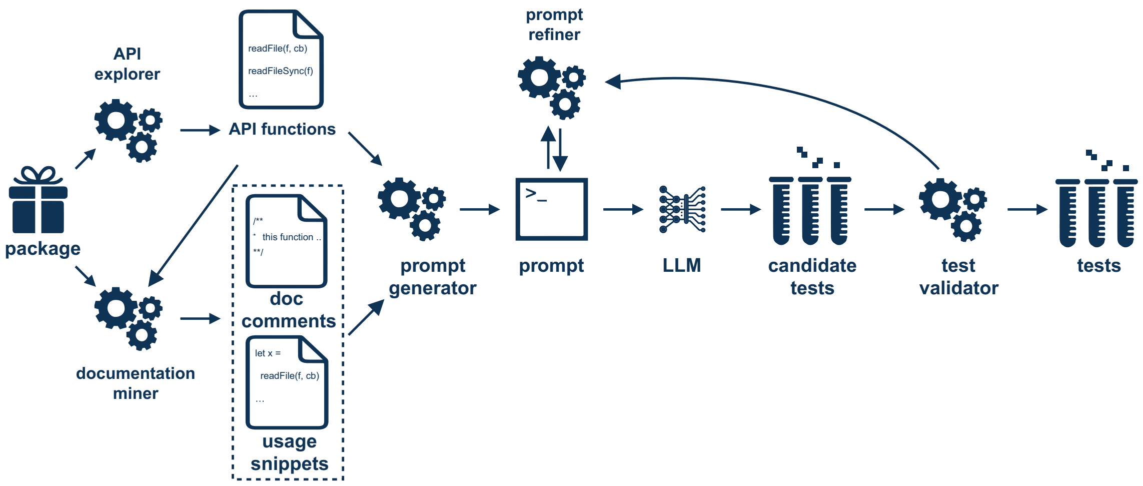


Figure 4: Test generation process used in Testpilot [49].

Testpilot works by forming a prompt for the Codex model. In the given prompt, the Codex model not only receives the source code but also harnesses information from extracted snippets, comments, and the signature of the software contained within the documentation. Furthermore, failed tests are corrected using a modified prompt, which tries to fix the invalid tests. When implemented for JavaScript, Testpilot was evaluated on 25 npm packages with 1,684 API functions [49]. The tests produced for the functions reached up to 93.1 % statement coverage with highly unique tests [49].

Crucially, LLMs can produce test cases that are more human-readable compared to conventional white-box methods since the models have been trained with human language and codebases [49].

A similar solution to Testpilot, Tufano et al. [9] presented Athenatest, an approach using a transformer model to create unit tests. Compared to Testpilot, Athenatest is trained with actual Java methods combined with developer-written test cases. For the training phase, a large dataset of mapped test cases has been drawn from open-source Java projects on GitHub [9]. The process is shaped as a sequence-to-sequence learning task and undergoes a two-phase training procedure with training large code corpus and finetuning for test generation [9]. The trained model can then be used to create test cases for the focal methods. When used for Java test generation, Athenatest reached similar test coverage compared to EvoSuite, a Java unit test generation tool, with improved readability and efficacy of its generated test cases [9].

2.3.3 Challenges of a Test Oracle

Automating the testing process, from test generation to execution, is a complex problem with several challenges that must be overcome. One of the challenges lies in the principle of testing, which is that exhaustively testing software is only sometimes possible [14]. Hence, there are no guarantees that the software is free of defects. Therefore, the test design process aims for high coverage of the functional and non-functional requirements.

If the target is to automate the complete testing process, creating the test steps is only part of the challenge. During the generation, the automation process must validate the correct behavior of the system. Verification of the outcome is called an oracle problem, and it has been a major part of test generation research [6]. The oracle problem is complex since correct behavior is not always well-defined and easily determined.

Test oracle is a significant part of the costs in test generation [7]. Even if the test generation is automatic, determining the outcome might require manual labor. For this reason, test oracles have been created for specific applications. Test oracles have often used supervised methods to determine the test verdict or the predicted expected output of the system [8]. The test verdict describes whether the test steps cause a passed or failed result.

Furthermore, NLP algorithms have been leveraged as the test oracle. Dinella et al. [51] proposed a transformer-based test verdict, which leverages two primary components: a classifier to discern whether a developer-intended exception should be raised and an assertion ranker to rank and select the most appropriate assertions based on the context. Even though progress has been made with LLMs in approaches such as by Schafer et al. [49], the test verdict often depends on the source code assertions. A generic test oracle for black box testing that determines the outcome for produced test cases has not yet been proposed. A black-box test oracle would be particularly beneficial for web applications and GUI testing.

3 Research Material and Methods

The chapter presents the approach to test generation, which includes the creation of a test generation framework, the implementation of the algorithms, as well as the training and evaluation of machine learning models. Additionally, chapter explains the benchmarking setup for evaluating algorithm performance in two test scenarios. Overall, the chapter offers a method for employing reinforcement learning and sequence modeling in test generation for large-scale web applications.

3.1 Approach to Test Generation

The selected approach for the thesis is search-based test generation. Search-based testing formulates the input generation as an optimization problem [52]. From a set of inputs, the objective is to find the ones that lead to the desired objective [8]. The thesis examines the black-box search approach for the generation method since white-box testing is only sometimes possible for complex web applications. For instance, the source code might be unavailable or unsuitable for the generation algorithm. The approach follows stochastic testing by creating a policy with action probabilities and generating test steps by sampling from the distribution. The steps for the generation approach are presented in Figure 5.

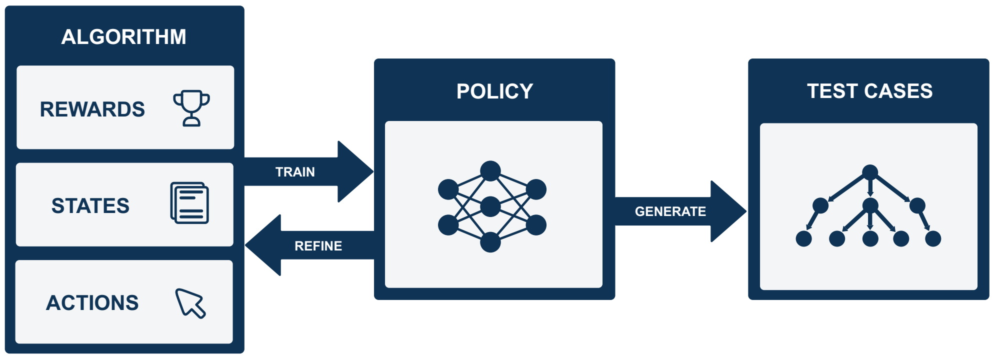


Figure 5: Test generation approach used in thesis

The search-based test generation often relies on a fitness function [52]. The fitness function describes a generation goal, which could be as simple as finding exceptions in execution [53] or aiming for as high code coverage as possible [45]. The thesis approach utilizes a user-defined reward signal to describe the testing objective, which can be formulated based on the software requirements. The reward signal allows the user to control the direction of the generation.

In the selected approach, the generation requires a pre-defined state and action space to use a policy gradient algorithm [54]. In the context of websites, the state can be a feature vector of website elements, and the actions can include a set of possible steps such as click and type. After selecting the state and action space, the algorithm can train a policy, which is evaluated by generating the test steps by sampling actions in the environment. The user can refine the selected reward signal, states, and actions if the policy fails to produce the desired outcome. Furthermore, the algorithm has training hyperparameters that can be tuned for better performance [55].

The previous research has covered several machine learning techniques in different state-based GUI applications. However, most research has focused on mobile testing and doesn't use test libraries designed for web application testing [8]. Web applications present testing challenges due to their potential to be accessed across various devices, resolutions, and browsers. Therefore, the thesis focuses on demonstrating the generation of web applications using an open-source test automation library.

3.2 Test Generation Framework

The machine learning algorithm requires a connection to a test library to generate test steps. Therefore, the thesis aims to create a framework for test generation and benchmark machine learning algorithms using the developed framework. The framework outputs keyword-based test steps that a test automation library can use. The created tests can then be used in regression testing or templates for creating more complex keywords. Test generation can reduce the required time for manual test development and aid with exploratory testing.

The test generation process is described in Figure 6. The approach consists of four parts: the system under test (SUT), the test library, the integration layer, and the machine learning algorithm. The framework, which includes the algorithm and the integration layer, is the core part of the test generation. One of the design principles was that the framework should be able to adapt to multiple test libraries and SUTs. The aim is to simplify the architecture from recent test generation approaches Yasin et al. [43] and Zheng et. al [44] to create a highly flexible framework. The framework forms an MDP that optimizes the test steps towards the testing objective given by the user.

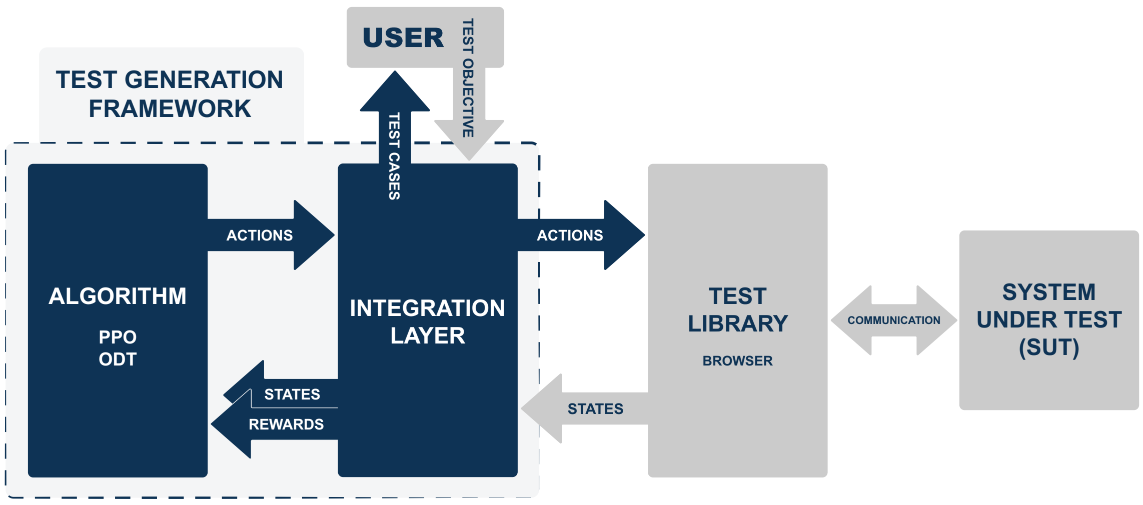


Figure 6: A high level diagram of the test generation framework.

1. System under test (SUT). Tests are generated for test objects, which could be web, mobile, or desktop applications, databases, or even embedded systems. In this thesis, web applications are used for benchmarking. The SUT communicates with the test library by returning the requested information from the system and executing the action keywords.

2. Test library. Test automation libraries exist for automating testing tasks for the system. The library is usually responsible for communication with the SUT. Test libraries often build upon structured scripting techniques by having reusable scripts that could be used in test cases [5]. In web applications, the scripts could consist of keywords such as clicking an element or typing text into the input field. The test library dynamically receives instructions, actions, and information requests as keywords from the framework.

3. Integration layer. The search-based method used in the thesis receives the test objective from the user to control the test creation process. Thus, an integration layer forwards information between the machine learning algorithm and the test library controlling the SUT. The integration layer receives website data and converts it to a state representation for the ML algorithm. In addition, the layer calculates the reward for the ML algorithm using the test objective given by the user. The layer also plays a part in forwarding the selected actions to the test library. Thus, the integration layer controls the whole test generation process by the objectives set by the user.

4. Machine learning algorithm. The ML algorithm receives system states and rewards from the integration layer. Using the states and rewards the algorithm optimizes towards a testing objective given by a user. Thus, the algorithm estimates the best possible step in a given state toward the test objective and returns it to the integration layer for execution. The integration layer and the algorithm are designed to be modular, which allows running multiple ML algorithms for test generation. Modularity ensures that the framework is suitable for new algorithms and test environments.

Besides generating tests, the goal of the framework is to provide a testing tool that is effortless to set up and has a conservative usage of system resources, allowing algorithms to be trained in a regular consumer laptop. In addition, it should be able to expand to larger models that can cover entire applications and systems. The framework should be capable of performing stochastic testing by creating models of software behavior [35]. The generation should be uncomplicated to expand to all types of testing.

3.2.1 State and Action Space for a Web Application

The integration layer converts the responses from the test library and the action keywords into corresponding state and action spaces. The ML algorithm requires a vector of features visible in state s, represented by $x(s)$, to output the action probabilities $\pi(a|s)$ for the state [28]. Furthermore, it is essential to know the vector of actions for selecting an action based on the probability distribution output. In the thesis, the generation approach uses a preselected feature and action vectors that could modified based on the training results.

The central question in web application test generation is how to represent the application states. The previous research by Zheng et al. [45] and Pan et al. [45] scans the application at every step. If the current page differs from previous observations, it is added to memory as a new state. The studies have used abstraction methods to simplify the GUI content into a state that the generation algorithm can compare with other states. Yasin et al. [43] uses information including the application element types, positions, and the parent-child relationships to distinguish between the states.

The thesis proposes an abstraction method for translating the HyperText Markup Language (HTML) page into state representation. A pure HTML document often contains redundant information, which can hinder learning efficiency. In the abstraction method, the HTML document is scanned for elements that are differentiated by types, attributes, and text content. The elements form a vector of feature flags, indicating if the elements are present on the page. Although this approach loses some information, including the page structure, the feature vector can store the state compactly. The state representation in the framework can be modified to support complex ML algorithms if the algorithm needs additional information from the environment.

The table 1 shows an example of two scanned elements of a webpage. The algorithm scans the webpage and compares the found elements to the selected features to form the feature vector $x(s)$. The features without tracking are updated to a secondary vector, which can be added to the state space by restarting the training. Unlike Q-learning, which could update new states and actions to Q-tables during training [44, 45], the algorithms employed in this thesis necessitate retraining when there are updates to the action or state space. The feature selection used in training can be further refined using feature selection techniques, including filtering, wrapper, and embedded methods [56]. In this thesis, the features are filtered manually.

Table 1: State representation of the system under test involves comparing the elements on a website with the feature vector. The state is then represented as a vector of binary feature flags, indicating the visibility of these elements.

State as a Feature Vector	
Feature	Description

`{"tag": "A",`
`"text": "Cart (0)",`
`"attributes": []}`
	`Shopping cart link with zero items is visible`

`{"tag": "INPUT",`
`"text": "",`
`"value": "test",`
`"attributes": [{"key": "type",`
`"value": "email"}]}`
	`A input field with value test is visible`

The table 2 shows an example of the actions available for the ML agent. Similarly to state space, the action space is selected before training, and the new actions are saved to a separate vector. The algorithm collects actions from interactive elements similarly to the proposed method by Yasin [43]. The action vector enables training the ML model to estimate the optimal action probabilities $\pi(a|s)$. In the training phase, the action is selected based on the action probabilities. As the probability increases, the likelihood of choosing the action rises. The ML agent can take any action, including unsuitable ones, as it is not always possible to predict all correct actions based solely on the state. If the training succeeds, the agent selects only suitable actions when generating the test cases.

Table 2: An action vector is used in the test generation framework. Actions are selected based on a probability distribution provided by the trained model.

Actions	
Action	Description
`{"keyword": "click", "args": ["xpath= //A[contains(text(),'Cart')]"]}`	`Click the shopping cart link`
`{"keyword": "type_text", "args": ["xpath= //input[@type='email']", "test"]}`	`Type text "test" to a field with attribute type='email'`

3.2.2 Reward Signal

In reinforcement learning, a reward signal indicates the objective that needs to be accomplished. In test generation, the reward signal could describe how well the test steps have reached the planned objective in a test scenario. Common objectives in test generation are different coverages such as state [57], transition [58], and code coverage [59]. Other metrics include input diversity [60] and curiosity [44]. In the thesis, the test generation is focused on covering the individual software requirements instead of maximizing the overall test coverage. The approach gives the control of the generated test cases to the user.

The reward signal for a step in Equation 11 is divided into rewards and costs, which are determined by the current state and action.

$$\textit{step reward} = \textit{test objective reward} - \textit{action cost} - \textit{stagnation cost} \tag{11}$$

The test objective reward is the core of the reward signal controlled by the user. Its purpose is to define the goal of testing without knowing the exact steps to get there [28]. The testing objective can be specific, such as reaching a message after submitting a form, or more generic, for example, locating as many error messages in the application as possible. Designing the test objective reward is not a trivial task that often requires trial-and-error [28]. The reward signal could achieve the desired signal by simply returning a high positive value after reaching the objective. However, rewarding the algorithm after completing the objective could lead to sparse rewards if the correct steps are challenging to find. Design can account for the scattered rewards by guiding the learning with a more detailed reward signal. However, the design should not make assumptions about the actions taken towards achieving the main objective [28].

Since the ML agent can take any action, the reward signal penalizes unsuitable actions. The action cost returns a negative reward if an action is selected, which cannot be taken in the state. For example, clicking a link that is not visible. The cost is a smaller negative reward if the ML agent performs a suitable action. The reason for the cost is to prevent incorrect actions and guide the agent to complete the task in as few steps as possible.

The stagnation cost is defined to prevent the plateau problem where the ML agent wanders between the same states [28]. The cost is a negative value assigned when the current state has been visited earlier in the trajectory. Discouraging repetition can reduce stagnation, but it might create path preferences. Therefore, the user must balance the stagnation cost against the other components of the reward signal.

The ML agent can perform different styles of tests by adjusting the reward parameters. Before training, selecting an appropriate reward signal is a critical aspect of successful test generation. Balancing the rewards might also require multiple trials. The total reward for the test case is the sum of rewards over the test steps. The generated test cases can then be ranked by the total reward and used for optimization.

3.3 Algorithms

The test generation benchmarking will be performed with two machine learning algorithms. These are proximal policy optimization (PPO) [55] and online decision transformer (ODT) [61]. With both algorithms, the aim is to create an optimal policy towards the test objective. By sampling actions from the policy, the ML agent should reach the testing objective. The thesis applies two methods for learning the policy. First, through search-based testing with the PPO algorithm, and second, by sequence modeling with ODT using the collected PPO training data, including the predicted action probabilities. Searching and sequence modeling offer viable solutions for test generation, as paths can be discovered by exploring the application and building upon previous trajectories. The PPO algorithm was selected for search-based testing due to its stability and sample efficiency, which are crucial in web environments with slow response times [55]. The ODT was the choice for sequence modeling since the algorithm can efficiently learn the behavior from the collected trajectories with transformer architecture [62]. Additionally, transformers can utilize the order of perceived states, in contrast to PPO training. Implementation details of both algorithms are explained in Sections 3.3.1 and 3.3.2.

3.3.1 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a policy gradient algorithm for reinforcement learning [55]. It's an on-policy algorithm, which means that the policy used to interact with the environment is the same as the one being optimized [63]. The algorithm addresses issues in traditional policy gradient methods of scalability, robustness, and sample efficiency by a clipped surrogate objective function that limits the size of policy updates, thus preventing drastic changes in the policy [64]. In addition, The PPO implementation used for test generation is displayed in Algorithm 1.

Algorithm 1 Proximal Policy Optimization

```
 1: Initialize actor and critic networks
 2: for each iteration do
 3:     for each episode step in the environment do
 4:         Collect trajectory using current policy
 5:     end for
 6:     Compute rewards-to-go $R_t$ and advantage estimate $\hat{A}_t$ using GAE
 7:     for each epoch do
 8:         for each minibatch do
 9:             Compute ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$
10:             Compute policy (actor) loss using PPO-Clip objective: $L^{CLIP}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)\right]$
11:             Compute value (critic) loss by mean-squared error
12:             Update actor and critic using gradient ascent with gradient clipping
13:         end for
14:     end for
15: end for
```

The PPO implementation uses an actor-critic method with independent feedforward neural networks for estimating, both the actor and the critic [55]. The actor outputs the probability distribution over actions and the critic the value for each state. In each iteration, a new trajectory is collected and added to the batch. After collecting the trajectory, the rewards-to-go and Generalized Advantage Estimated (GAE) are calculated for the batch. GAE is selected to balance the variance and bias in the policy gradient [65]. The batch is then split into several mini-batches and used to update the actor and the critic over multiple epochs. In one epoch the agent updates the policy based on collected data. The actor loss is calculated using the PPO-clip objective and the critic using mean squared error (MSE) loss [63].

The architecture used for both actor and critic is a feedforward neural network with one hidden layer with 128 neurons. The input and output sizes are determined by the feature vector and the number of actions, respectively. The network configuration was similar to OpenAI Gym PPO benchmarks using a ReLu activation function after the first linear layer [66]. The softmax activation function outputs the action probabilities after the hidden layer. During experimentation, the network became overfitted, causing high probabilities for unsuitable actions. Ten percent dropout, suggested by Srivastava et al. [67], was added after the hidden layer to prevent overfitting. Dropout randomly drops the neurons along with the connections from the network [67]. Additionally, temperature scaling was added to modify the balance of the action probability output [68].

Hyperparameters control the learning process. In the PPO algorithm, the parameters can direct factors such as gradient step sizes with learning rate and the discount factor γ [55]. The benchmarking process applied a manual parameter search starting from the suggested ranges for PPO training [69]. The parameters reaching the highest rewards were selected for the final model. In addition to manual searching, automatic hyperparameter optimization was also employed to optimize the learning rate. Exploring the web environment is often slower compared to simulated environments. Therefore, the manual search reduced parameters such as the batch size to lower values. The selection process and the hyperparameters are in Appendix A.1.

3.3.2 Online Decision Transformer (ODT)

Decision transformer, introduced by Chen et al., [62] converts reinforcement learning to a sequence modeling problem. The algorithm abstracts the decision-making process as a trajectory of states and actions, optimizing rewards over the trajectory [62]. The decision transformer leverages the transformer architecture, which has reached superior performance in applications such as text generation using language models such as Generative pre-trained transformer (GPT) and Bidirectional Encoder Representations from Transformers (BERT) [33, 70]. The decision-making process, which involves selecting an action based on the state, is a similar problem to sequential data translation tasks where transformer models are frequently used. The algorithm architecture is visualized in Figure 7.

The decision transformer starts by embedding the inputs, including the states, actions, and rewards-to-go, with positional encoding to retain the trajectory order. The trajectory is then passed through a causal transformer, which uses GPT-2 architecture [62]. Finally, the linear decoder translates the transformer output into the action probabilities. The decision transformer is trained offline from collected trajectories such as random explorations in the environment [62].

In short, the algorithm aims to optimize actions for maximum reward using the patterns in the trajectories. Compared to reinforcement learning with policy gradient approaches, the decision transformer doesn't only utilize the current state but also information on the previous steps. The memorization of previous steps is limited by context length variable K [62]. With Atari benchmarks, the study by Chen et al. [62] noticed that the context length $K = 1$ reduced the performance of the decision transformer and increased with higher values. In web applications, memorizing the previous states might be even more essential since these environments often contain various loops between the same states.

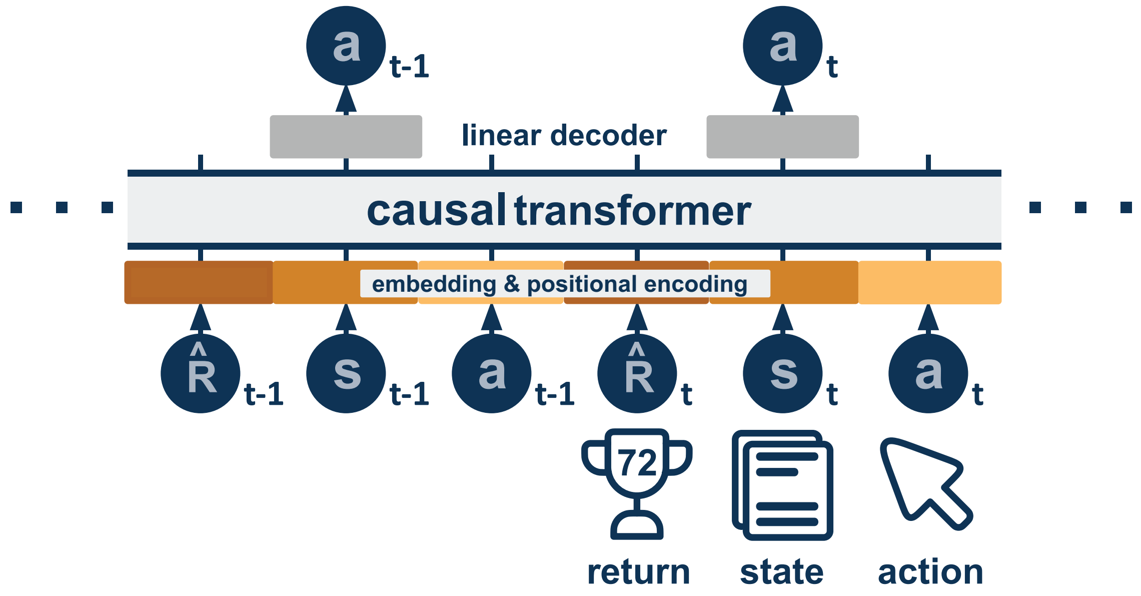


Figure 7: Decision Transformer architecture[62]

Another benefit of decision transformers for test generation is the option to select the target reward for the test generation. The algorithm can be used for behavior cloning training the model from trajectories and evaluating using a targeted reward [62]. By selecting the reward signal, the test generation can behave differently and create various test cases.

For benchmarking the test generation in this thesis, the original decision transformer algorithm by Chen et al. [62] with continuous action space was modified for discrete action selection. Furthermore, the algorithm was extended to include online fine-tuning. The implementation is displayed in Algorithm 2.

Algorithm 2 Decision Transformer for Discrete Actions [62]

```
1: **Input:** $R, s, a, t$: returns-to-go, states, actions, or timesteps
2: **Input:** $embed\_s,\ embed\_a,\ embed\_R$: linear embedding layers
3: **Input:** $embed\_t$: learned episode positional embedding
4: **Input:** $pred\_a$: linear action prediction layer with softmax activation for probabilities
5: **Input:** $transformer$: transformer with causal masking (GPT)
6: **function** DecisionTransformer($R, s, a, t$)
7:     $pos\_embedding \leftarrow embed\_t(t)$
8:     $s\_embedding \leftarrow embed\_s(s) + pos\_embedding$
9:     $a\_embedding \leftarrow embed\_a(a) + pos\_embedding$
10:    $R\_embedding \leftarrow embed\_R(R) + pos\_embedding$
11:    $input\_embeds \leftarrow \text{stack}(R\_embedding, s\_embedding, a\_embedding)$
12:    $hidden\_states \leftarrow transformer(input\_embeds = input\_embeds)$
13:    $a\_hidden \leftarrow \text{unstack}(hidden\_states).\text{actions}$
14:    **return** $\text{softmax}(pred\_a(a\_hidden))$
15: **end function**
16: # Training the decision transformer
17: **for** each $(R, s, a, t)$ in $dataloader$ **do**
18:    $a\_probs \leftarrow \text{DecisionTransformer}(R, s, a, t)$
19:    $loss \leftarrow \text{cross\_entropy}(a\_probs, a)$
20:    $optimizer.zero\_grad(); loss.backward(); optimizer.step()$
21: **end for**
22: # Evaluating the decision transformer
23: $target\_return \leftarrow 1$
24: $R, s, a, t, \text{done} \leftarrow [target\_return], [\text{env.reset}], [], [1], \text{False}$
25: **while** not done **do**
26:    $action\_probs \leftarrow \text{DecisionTransformer}(R, s, a, t)[-1]$
27:    $action \leftarrow \text{sample}(action\_probs)$
28:    $new\_s, r, \text{done}, \_ \leftarrow \text{env.step}(action)$
29:    $R \leftarrow R + [R[-1] - r]$
30:    $s, a, t \leftarrow s + [new\_s], a + [action], t + [\text{len}(R)]$
31:    $R, s, a, t \leftarrow R[-K :], \ldots$
32: **end while**
```

Compared to the original decision transformer, the softmax activation function is added to the action prediction for categorical probability distribution output. The framework samples the action from the distribution similar to the PPO implementation. The original decision transformer included only offline training [62]. Therefore, the algorithm used for benchmarking has an option for online fine-tuning. The thesis uses an online decision transformer (ODT) algorithm, first formulated by Zheng et al. [61]. In online training, evaluation trajectories are collected and included in training data in the subsequent training iterations [61]. Experimental details of ODT implementation, including the hyperparameters, are listed in Appendix A.2.

3.3.3 Training and Evaluation

Before test generation, the implemented algorithms train a model for a specified test scenario. The benchmarking trains models for PPO and ODT algorithms and evaluates the performance by comparing the achieved rewards and test coverage metrics. For the PPO model, the first step is pre-training the model. In the PPO pre-training, the reward signal doesn't contain the test objective reward. The objective is not included since the model is trained to learn the state-specific suitable actions to reduce the required training time in the fine-tuning phase. After the model is pre-trained, the agent no longer needs to explore and learn which actions are unavailable in each training round. The pre-training phase is also used for collecting the feature vector and action space by restarting the training after the model has explored states for a user-specified number of episodes. An episode is the trajectory from start to finish.

After pre-training the PPO model, the model is fine-tuned with the test objective reward included in the reward signal. In this phase, the model is trained to generate the test cases for specified test scenarios. When the model is trained, the trajectories and total rewards are collected. After a model has been trained with PPO, the collected data is used for training a model with ODT. The ODT training is conducted in two parts: offline pre-training with the collected data and online fine-tuning in the online environment while generating new trajectories to the dataset.

The created models are evaluated in the original and modified test environment for a test scenario. The evaluation generates 100 test cases, from which performance data, including the mean reward and the completion percentage of the test objectives, are collected. The research in test generation often evaluates the algorithms by coverage metrics such as code coverage [8]. The test generation approach in the thesis doesn't aim for the highest possible code coverage across the application due to the test objective set by the user. Therefore, the evaluation focuses on the covered paths for the test scenario by examining the test generation graphs.

3.4 Benchmarking Environment

Machine learning algorithms are often trained and evaluated in simulated environments such as OpenAI's Gym [71]. Therefore, a benchmarking environment is essential for evaluating the performance of test creation algorithms and verifying the functionality of the framework. Research in GUI-based test generation has often focused on mobile [43] or web applications [44] for test generation. For benchmarking in this thesis, a web application has been designed to run in a local environment. The local application ensures greater stability and eliminates any latency that might occur while testing the application over the internet. A screenshot of the benchmarking application is shown in Figure 8.

The web application has been designed to mimic a generic web store where users can purchase items. It includes common functionality used in many websites, such as navigation, modals, and forms. The application is divided into software features that need to fill specified requirements. The software features are product browsing, shopping cart, and user login. In Agile software development, the requirements are used by a tester to verify that the application meets the specifications [72]. High-level requirements designed for the software features are listed from 1 to 4:

1: A user should be able to browse different product categories and search for specific products

2: The website includes a shopping cart page where the products can be added

3: The purchase can be completed by filling out a form

4: A user can log in to the website with a username and password

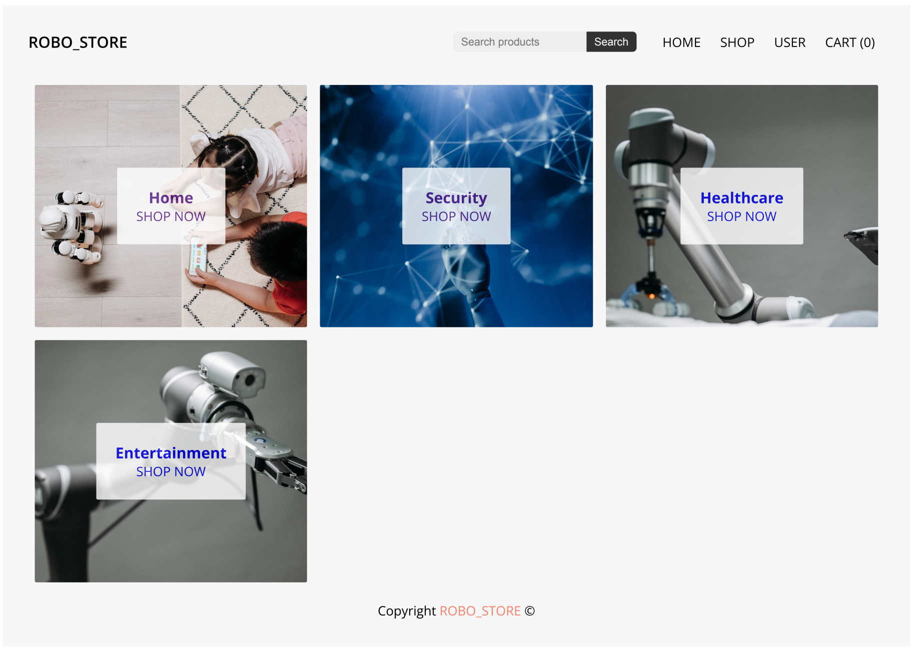


Figure 8: Screenshot of the benchmarking web application

The requirements can be translated into test objectives by creating a reward signal. The signal returns the combined rewards and costs of the step taken in the test environment, along with a boolean value that indicates whether the final test objective has been achieved. Translating a test to a reward signal manually removes the need for natural language processing. Although, converting the application-related data into test cases is already demonstrated by Schafer et al. in white-box test generation [49].

From the requirements, multiple test objectives could be defined for guiding the ML agent through the application. The reward signal gives freedom to control the behavior of the ML agent. The design should still monitor the learning behavior to avoid the plateau problem [28]. For the benchmarking application, the reward signal will be designed to increase the rewards towards the final objectives to avoid stagnation.

3.4.1 Test Objectives as Reward Signals

The reward signal is one of the critical aspects of the generation process. Before initiating the training process, the software requirements are transformed into two test scenarios, each incorporating specific test objectives. Figure 3 shows the chosen static rewards and costs for the scenarios.

Designing a reward signal requires often trial-and-error [28]. First, the range of the rewards was set approximately from -1000 to 1000 with a maximum of 20 steps in an episode. Negative costs from actions will decrease, and the test objective reward will usually increase the total reward. After selecting the ranges, the action costs were adjusted by observing the performance while training an initial policy. After setting the failed and passed action costs, the stagnation cost was added to avoid looping through the same states.

Table 3: Rewards and costs for the test scenarios

Reward signal	
Name	Value
Test objective reward	Computed by a function
Passed action cost	-5.0
Failed action cost	-25.0
Stagnation cost	-15.0

The test objective reward is specific to a test scenario. Two test scenarios can cover the software requirements. The scenarios are logging in to the page and buying products from the website. The first scenario is divided into two test objectives, correct and incorrect logins, which cover the 4th requirement. The objectives are transformed into rewards simply by returning a positive reward from a login attempt and a greater value from a successful login. A test case is complete when the successful login is reached. During experimentation with the reward signal, it became apparent that returning small negative rewards when the agent explores pages other than the login page enhances the efficiency of the learning process. The test objective reward for the first test scenario is displayed in Algorithm 3.

Algorithm 3 Test objective reward for the first test scenario

```
1: $firstObjectiveReached \leftarrow False$
2: function REWARD(loginPage, loginFailed, loginSucceeded)
3:     $reward \leftarrow 0$
4:     $done \leftarrow False$
5:     if not loginPage then
6:         $reward \leftarrow reward - 30$
7:     end if
8:     if not firstObjectiveReached and loginFailed then
9:         $firstObjectiveReached \leftarrow True$
10:        $reward \leftarrow reward + 500$
11:    end if
12:    if loginSucceeded then
13:        $reward \leftarrow reward + 1000$
14:        $done \leftarrow True$
15:    end if
16:    return $reward, done$
17: end function
```

The second test scenario includes the objectives of adding products to the shopping cart and completing the purchase. The scenario covers requirements 1 to 3, as the ML agent can learn the paths through category pages or the search function. The objectives are transformed into the test objective reward by returning high positive values from reaching both objectives. In the second test scenario, an additional cost is not added to avoid affecting the exploration before the second objective. The test objective reward for the first test scenario is displayed in Algorithm 4.

Algorithm 4 Test objective reward for the second test scenario

```
1: $firstObjectiveReached \leftarrow False$
2: function REWARD(shoppingPageNotEmpty, purchaseComplete)
3:     $reward \leftarrow 0$
4:     $done \leftarrow False$
5:     if not firstObjectiveReached and shoppingPageNotEmpty then
6:         $firstObjectiveReached \leftarrow True$
7:         $reward \leftarrow reward + 800$
8:     end if
9:     if purchaseComplete then
10:        $reward \leftarrow reward + 1000$
11:        $done \leftarrow True$
12:    end if
13:    return $reward, done$
14: end function
```

Since there are no definite guidelines for designing the reward signal, the user could modify the test objective reward, and training could achieve similar or even better behavior. The training process could use techniques such as shaping to update the reward signal as learning proceeds [28]. For the benchmarking, the two reward signals achieve coverage of the functional requirements. Even though these cover the functionality in the benchmarking application, test generation could be expanded to include security or broad defect finding in production environments.

3.4.2 Selected Features and Actions

The feature vector was created by exploring the application during the pre-training phase. The page is scanned with an automatic script in each iteration, and the elements are collected into a feature vector and saved for the training phase. A feature includes the element name, text content, and every attribute. All content within the division (div) and paragraph (p) elements were automatically filtered to remove redundant features. Additionally, the shopping element was excluded since it contains updating numerical values. The final feature vector contained 186 elements.

The actions were created and added to a vector, similar to the features. Two keywords, "Click" and "Type Text", were used to cover the actions in the application. The automatic script selected all link (a) and button elements and created click actions. After selecting the elements, the script assigned type actions for the text fields. For each field, two test input options were provided, including correct input values for both test scenarios. After automatic inputs had been created, manual additions were made to broaden the action space. Manual filtering also had to be performed to improve the learning process. Filtering dropped a few "Add to Cart" actions generated by the script to balance the number of available actions in each view. The resulting action array contains 62 actions.

4 Results

The results chapter contains the training of the machine learning models and evaluation by generating the test cases. The training and evaluation are performed for the two designed test scenarios executed in the same test environment. The test environment is further modified in the evaluation section to test the generation performance in a changing environment.

4.1 Training with PPO

The aim of benchmarking the algorithms within the framework was to determine the effectiveness of objective-directed test generation and how to optimize the machine learning model to create test cases. As planned, the training process is divided into two phases. The model is pre-trained for 600 episodes in the first training phase without defining a testing objective. Thus, the test objective reward in Equation 11 equals zero. With pre-training, the model learns the basic steps in the environment and reduces training time in the fine-tuning phase. Figure 9 displays the mean reward from pre-training with three random seeds.

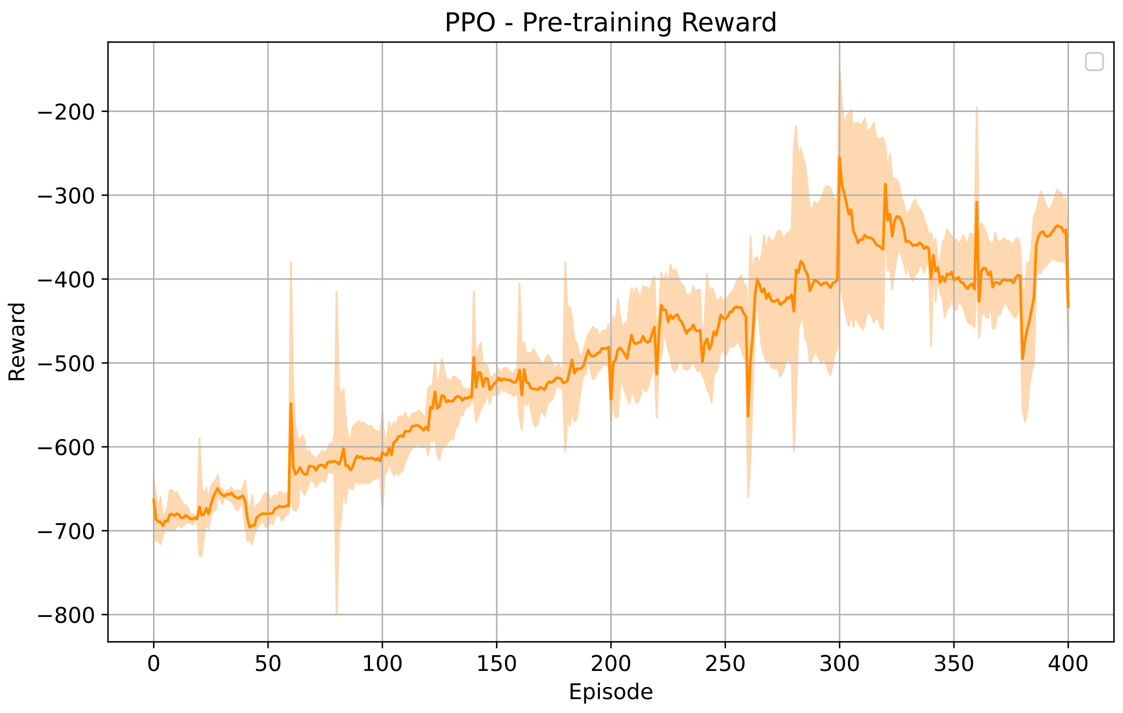


Figure 9: Mean episode rewards result from pre-training a model in the benchmarking environment using the PPO algorithm. During the pre-training phase, a test objective reward is zero.

The pre-training reaches maximum mean reward across three training runs after 300 episodes. While pre-training without a test objective, the model overfits after accomplishing the highest rewards, lowering the probability of suitable actions. Overfitting is a problem in the fine-tuning phase since an overfitted model doesn't generalize well to unseen data [73]. The pre-training phase utilizes early stopping to reduce overfitting. Therefore, the pre-trained model is selected for the next training phase at the 300-episode mark when training has reached the maximum reward.

After pre-training, the model undergoes fine-tuning for both test scenarios. In fine-tuning, the scenario-specific test objective reward determines the value of the ML agent's path. In other words, the reward signal assigns a score to the generated test cases according to the objectives in the test scenario. The fine-tuning is performed three times for both test scenarios. Figure 10 and 11 display the mean reward of the training runs and the mean test scenario completion signal for each run. The training uses the optimized hyper-parameters shown in Appendix A.1.

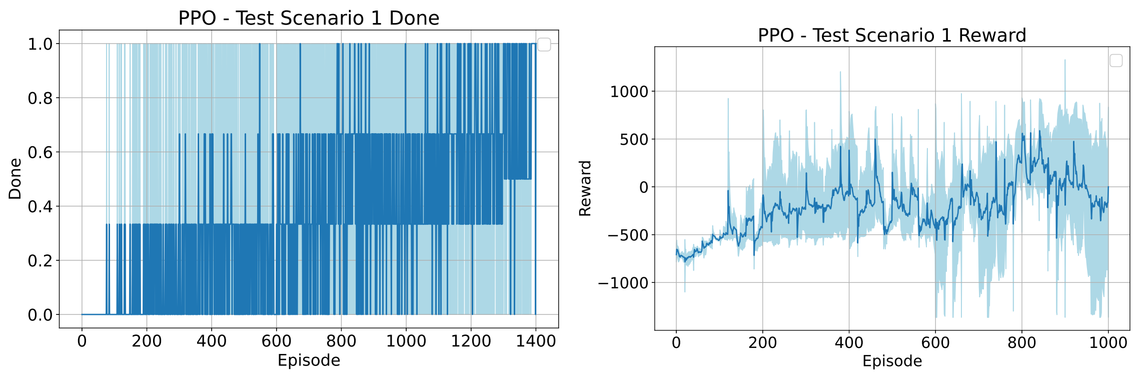


Figure 10: Test Scenario 1 - PPO Fine-Tuning. On the left is the mean of the scenario completion signal $[0, 1]$, and on the right are the mean episode rewards.

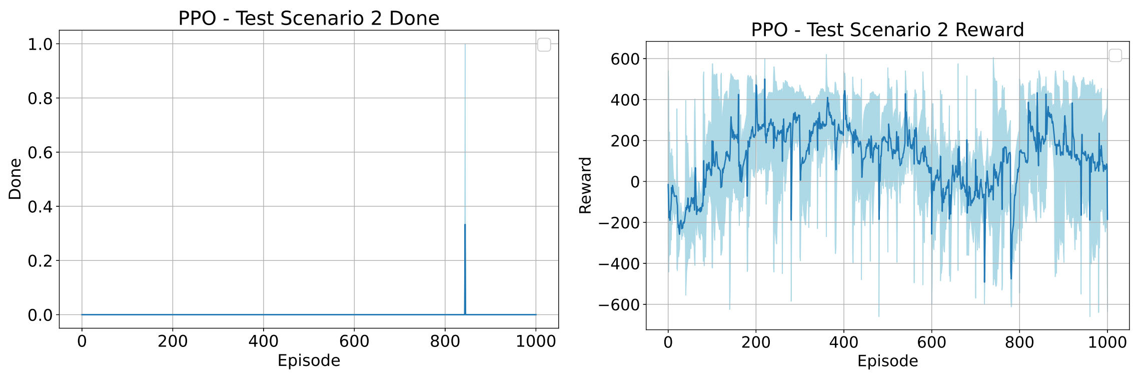


Figure 11: Test Scenario 2 - PPO Fine-Tuning. On the left is the mean of the scenario completion signal $[0, 1]$, and on the right are the mean episode rewards.

The PPO algorithm reached a test complete condition of the test objectives during the fine-tuning phase, although the model did not learn to generate all the expected test cases. Furthermore, the performance of the training in test scenarios shows noticeable differences. The differences were expected since the scenarios require distinct test steps.

Besides training performance, fine-tuning exhibits a more significant reward variance than pre-training. The higher variance is caused by a sparse test objective reward signal, which returns a high value when the objective is reached and a lower value when the ML agent selects inadequate test steps. Since the algorithm optimizes the probability of actions, the agent might randomly choose the correct steps even though the agent has not learned the objective. The training graphs show that the frequency of high rewards increases when the model learns the test objective.

The first test scenario, login to the system, is in a test-done state when the agent logs in with the correct details. Positive rewards were also awarded for login attempts with incorrect information. The training plots show that the test scenario is first completed after 70 episodes. Furthermore, the completion rate and the mean reward increase through training. The agent also discovered the trajectory with the highest possible rewards by first trying the wrong login details and completing the login afterward.

The second test scenario differs in length and the amount of potential trajectories. The test scenario has two objectives: adding a product to the shopping cart and completing the purchase. The first test objective is shorter and has multiple routes through shopping, category, and search functions. In contrast, the second objective requires specific test steps to complete the form. Based on the reward signal, the agent learns the first test objective quickly after 200 episodes. However, the second objective is achieved only once in the training iterations. Upon attaining the initial test objective, the agent searches for the optimal competition path, resulting in training plateauing and later declining.

When comparing the two test scenarios, the PPO algorithm performed better with shorter trajectories and test objectives with multiple paths. Additionally, the test objective significantly affects the learning performance. In the first test scenario, the reward signal encourages the agent to stay on the login page by adding a cost when the agent leaves the page. In contrast, in the second scenario, the agent doesn't receive additional rewards for completing the purchase form. As a result, staying on the shopping cart page and trying to fill out a form doesn't benefit the agent in the short term. Therefore, the second test scenario is challenging to complete.

4.2 Training with ODT

The subsequent target is to train an online decision transformer with the collected PPO training trajectories. The models are initially offline pre-trained for both test scenarios using test data from PPO training. In the second phase, the models are fine-tuned through online training while collecting new trajectories for the training data. The training is divided into multiple iterations of training steps that update the model parameters. The pre-training contains five iterations with 100 steps and fine-tuning ten iterations with 1000 steps in one iteration. After each iteration, the algorithm evaluates the model for 30 episodes in the online environment. The mean training loss is displayed in Figure 12, while the mean episode rewards in Figure 13.

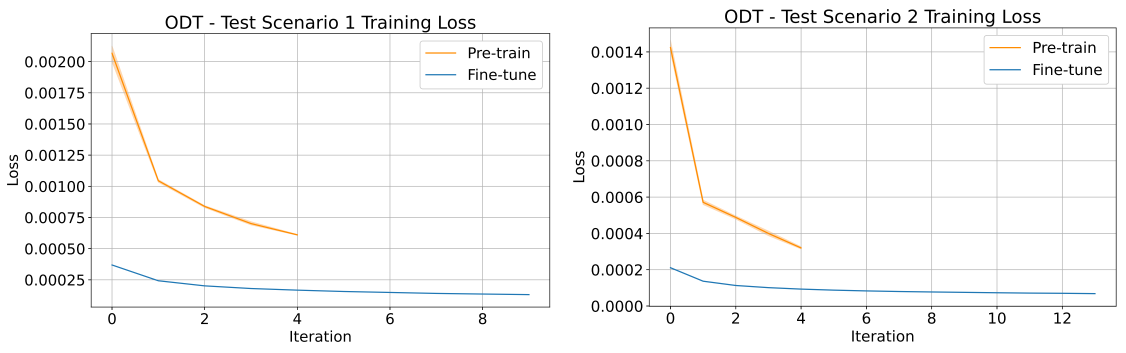


Figure 12: ODT pre-training and fine-tuning mean loss for both test scenarios. Pre-training iteration contained 100 steps and fine-tuning 1000. The training was executed three times.

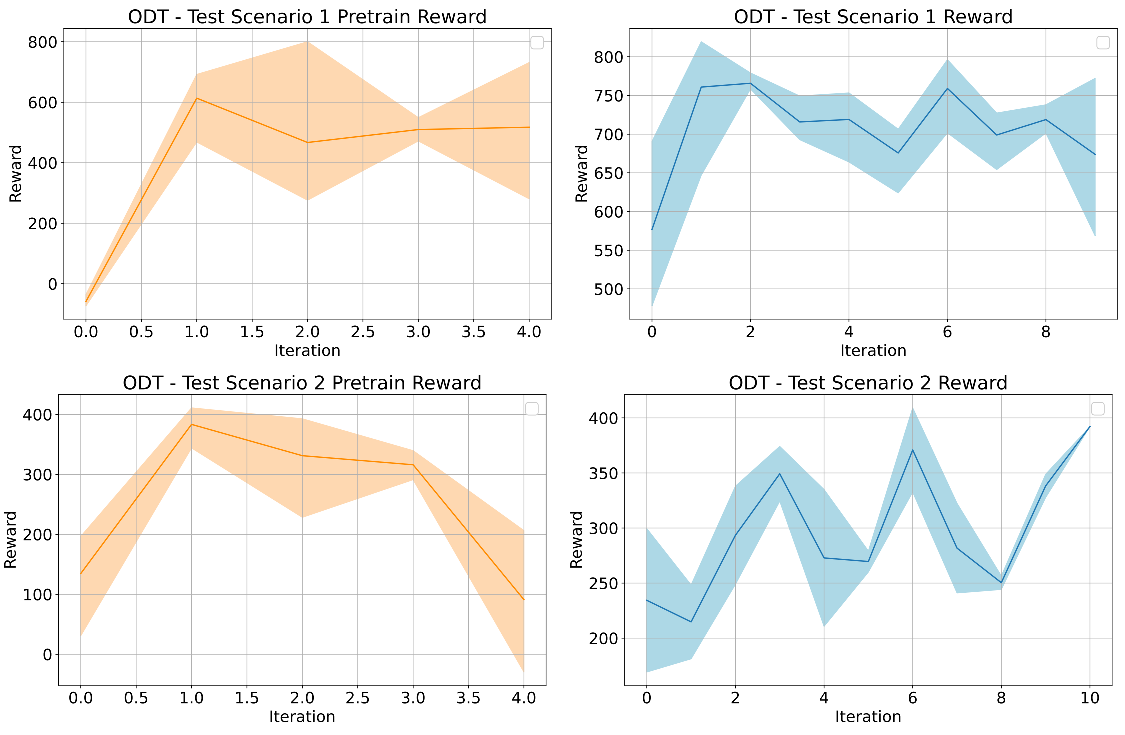


Figure 13: ODT mean reward for pre-training and fine-tuning phases. The evaluation was performed for 30 episodes after each training iteration.

Using the PPO training data, both models reached high positive rewards in 200 training steps in the pre-training phase. In the duration, the model learned to output action probabilities, which resulted in a similar outcome as the steps generated with the PPO models. However, the rewards had high variations in the pre-training for the second test scenario. The training loss converged after 500 steps of offline and 3000 steps of online training.

In the online fine-tuning phase, the models were evaluated in a benchmarking environment after a training iteration. The algorithm appended the evaluation trajectories to the training data. Although the algorithm explored new paths in the online training, the maximum rewards didn't further improve compared to the PPO training. Based on the training graph, the models reached the same test objectives as PPO. The reward variance indicates that the models don't explore the environment as much as in the PPO training. Overall, the ODT training achieved comparable maximum rewards to the PPO models.

4.3 Test Generation Evaluation

The second research question focused on an analysis of the trained models by generating a series of test cases. The best-performing PPO and ODT models, according to episode rewards, are selected for test generation. The ODT model for the same test scenario has been pre-trained with the collected action probabilities during PPO training. Thus, the null hypothesis is that the ODT model should be able to exhibit as high mean rewards as the PPO model. In each evaluation round, 100 test cases are generated in total. The evaluation is run in the original environment and modified environments explained in Appendix A.3. Mean reward, standard deviation, maximum reward, and success rate for test objectives are collected and displayed in Table 4 and 5.

Table 4: The table presents the outcomes of 100 evaluation rounds for the models in the first test scenario. The environments consist of both original and modified benchmarking applications. In the modified application, the user page is accessible only through the product pages. The table displays the completion percentages for Test Objective 1 (TO1) and Test Objective 2 (TO2).

Test Scenario 1						
Environment	Algorithm	Mean Reward	STD	Max Reward	TO1	TO2
Original	PPO	59.2	643.3	1195.0	46%	40%
Original	ODT	764.3	263.4	980.0	2%	99%
Modified 1	PPO	-708.4	749.0	1200	23%	16%
Modified 1	ODT	-1366.1	42.2	-1140.0	0%	0%

Table 5: The table presents the outcomes of 100 evaluation rounds for the models in the second test scenario. The environments consist of both original and modified benchmarking applications. In the modified application, access to some of the category pages are disabled. The table displays the completion percentages for Test Objective 1 (TO1) and Test Objective 2 (TO2).

Test Scenario 2						
Environment	Algorithm	Mean Reward	STD	Max Reward	TO1	TO2
Original	PPO	369.1	266.0	585	92%	0%
Original	ODT	253.1	347.4	565.0	83%	0%
Modified 2	PPO	-499.45	114.7	355	1%	0%
Modified 2	ODT	-538.1	131.9	305.0	2%	0%

Although the algorithms used equivalent training trajectories, the models have differences in the evaluation rewards. Therefore, the alternative hypothesis is that there is a difference in the mean rewards between the algorithms. The significance of the results can be checked by an independent two-sample t-test in Table 6. Independent t-test compares two means by producing a p-value to describe the confidence of the results [74]. For the t-test, the data are assumed to be approximately normally distributed with variance differences. Welch's t-test variant is used to take into account the variance differences [74].

Table 6: Independent two-sample t-test results for the mean reward difference in the test scenario and environment combinations

Scenario	Environment	T-Statistic	P-Value
1	Original	-10.14	$3.30 * 10^{-18}$
1	Modified	8.77	$5.08 * 10^{-14}$
2	Original	2.65	0.0087
2	Modified	2.21	0.0282

The null hypothesis of no statistical difference can be rejected, as each p-value is significantly lower than the typical significance level of 0.05 [74]. Therefore, the differences in mean rewards between the Proximal Policy Optimization (PPO) and Online Decision Transformer (ODT) algorithms are statistically significant.

The lower mean rewards observed in ODT are likely attributable to multiple factors. The algorithmic differences can partially explain the differences in mean rewards. As the PPO uses the current state of the application, the ODT follows the previous states, actions, and rewards. Just the approach of using the previous data can create vastly different training results. As the ODT is pre-trained with the PPO action probabilities, the PPO training data limits the ODT performance. For instance, if there are only a few example trajectories with high rewards, as in the case of login training, the ODT model might not learn the optimal action probabilities. Consequently, failing to reach the optimal test steps will result in lower mean rewards. Furthermore, fine-tuning in an online setting failed to improve the mean rewards. The lower performance might also be attributed to a limited number of new trajectories explored during the online training phase.

The algorithms also show differences when examining the individual evaluations and maximum rewards. The PPO algorithm successfully trained a model that achieved the login attempt objectives in the first test scenario. Similarly, the ODT model reached both objectives. The most notable aspect is the behavior between the models. The trained PPO model has more variance in the selected actions while submitting the wrong user details first and then the correct ones. When the login form receives correct and incorrect information, evaluation has increased maximum rewards and higher reward variance. Contrarily, the ODT model exhibits lower variance and aims specifically for successful logins. Login succeeds nearly every round, reaching a higher mean reward than the PPO model. However, the maximum reward is lower since the model doesn't attempt to input incorrect login details.

In the second test scenario, 80% of the generated test cases achieved the first test objective of adding a product to the shopping cart for both models. As observed in the training phase, the second objective of completing the purchase is not reached with the trained models. The ODT achieved a slightly lower mean and higher reward variation in the second scenario. The evaluation further outlined that the rewards for the scenario are two sparse, and the model continues to explore the environment until reaching the step limit.

Modifying the environments reduces the test generation performance. The environment was modified so that the most commonly used route to the test objective is no longer available, forcing the model to use an alternative path. Reduction in performance is expected since the models trained in the original environment haven't adjusted to states where some elements are missing. In the first test scenario, the PPO model reached the objectives with lower percentages, and the ODT model didn't reach the objectives at all. The second test scenario also had lower completion percentages. The main observation from evaluating the modified applications is that the changes or bugs can be detected by test generation. A modified path will lead to lower completion percentages.

For examining the test coverage required by the third research question, Figure 14 and 15 contain directed graphs of 10 test cases for both scenarios. The section focuses on the individual paths covered by the test generations, as the interest lies in the coverage around the test objective. The graphs present the system state as nodes where a level corresponds to a successful test step. The nodes include information about the reached objective and the visit count in the state at the current level.

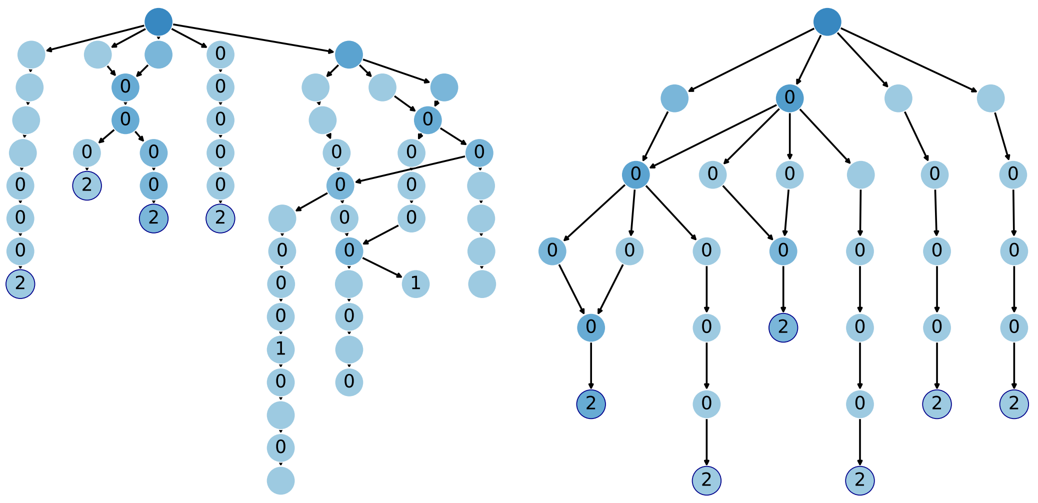


Figure 14: Generated test steps represented in a tree graph for Test Scenario 1. A darker color indicates a higher number of visits to that state. Markings in the diagram are as follows: '0': the agent is on the login page, '1': Test Objective 1 has been reached, and '2': Test Objective 2 has been reached. Nodes where the process is complete are marked with a blue circle.

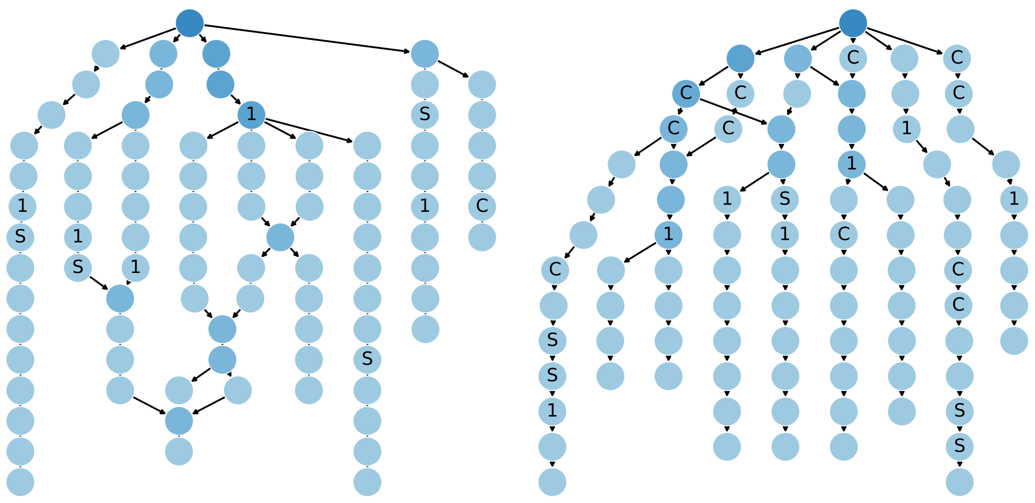


Figure 15: Generated test steps are represented in a tree graph for Test Scenario 2. A darker color indicates a higher number of visits to that state. Markings in the diagram: 'C': the agent is on the category page, 'S': the agent is on the search page, and '1': Test Objective 1 has been reached.

The plotted test cases are consistent with the observations from evaluating the test generation in table 4. The PPO model explores the application with widely spread action probabilities, which usually leads to either a login attempt or a successful login. In the other graph, the ODT model trained with the same data has stricter paths following the near-optimal routes to successful login with some alternative orders of filling the text fields. When comparing the paths in the first test scenario, the model trained with PPO covers more of the functionality related to the requirements. The ODT model covers fewer paths but is consistent with successful logins.

The second test scenario is even more interesting since there are multiple paths to the test objectives. There are three main paths: search tool, category pages, and all products page to reach the first test objective of adding the products to the shopping cart. The PPO model mainly uses the all-products page as the ODT navigates the category pages to reach the test objective. In addition, the ODT model uses all paths to achieve higher coverage than PPO. After reaching test objective 1, both models explore the application, aiming to reach the final objective of completing the order.

Although the coverage differed between the models, the training parameters can control the probability distribution for the actions. For instance, modifying the temperature hyperparameter can increase or decrease the stochasticity. In addition, the pre-training process can manipulate the training data for the ODT algorithm to change the behavior.

The evaluation showed that the test generation approach produces various test cases with carefully planned training. The pre-selected actions and states work well when the number of paths is large enough. The ODT algorithm also enabled the behavior cloning from the collected data to similar or even better rewards and coverage to the PPO model.

5 Discussion

The discussion section will focus on interpreting and comparing the results to other test generation methods. In addition, the section outlines the future research needed for improving the test generation performance in different metrics.

5.1 The Framework

The primary aim of the thesis was to create a testing framework that could support multiple machine learning algorithms and testing libraries. With the framework, test generation should be able to run effortlessly by setting up the test environment including the training objective and parameters. As demonstrated by the results, the framework could generate test cases and multiple algorithms can be run using the framework.

For future use cases, the framework is expandable for multiple test libraries and areas such as API and database testing. In addition, removing the need for manual filtering of the generated features and actions could improve the state and action space creation. As a result, converting the website to states and actions can be optimized and automized. The automatic translation process is required for fully autonomous test generation using software requirements and documentation. The framework provides a basis to develop new algorithms and more complex models for test generation.

5.2 Test Generation Optimization

In prior studies, the PPO algorithm has not been applied to test generation [7, 8]. Compared to other search-based methods utilizing Q-learning including mobile testing by Adamo et al. [75], the PPO test generation with the test objective reward signal gives more control over the generated test cases. By implementing a reward signal for the software requirements, the test generation can reach high test coverage. Additionally, the created model can be used to detect defects in the test environment by observing the test objective completion rate and the paths as demonstrated by the modified environments in the evaluation section. The PPO search-based test generation method, with its pre-selected state and action space, was simple to implement. However, the generation performance, particularly in single-path test cases, is improved with more complex curiosity-driven approaches by Zheng et al. [44] and Pan et al. [45]. These methods, however, require more memory for saving the state-action pairs and more complex pathfinding algorithms [45].

Using the collected trajectories, the ODT algorithm was able to create a model with high positive rewards and matching coverage to the trained PPO policy. In the second test scenario, ODT reached higher test coverage compared to the PPO test generation. The benefit of using ODT over the traditional policy gradient methods is the ability to remember the order of states and selected actions with context length hyperparameter [62]. The context length affects the generation performance since the shortest path might sometimes repeat identical states. A potential use case for

ODT is learning the model from manually developed test cases. Although Khaliq et al. [48] used transformers to generate black-box test data, manually created test cases have not been utilized for generating new tests. In general, research on using transformers in black-box test generation is in its early stages.

A limitation of the approach used for test generation is the state and action space that must be selected before training. In addition, the model doesn't have information on the incorrect actions for a state. As observed in the evaluation phase, the second objective in test scenario 2 was not learned. The behavior can be partially attributed to a phenomenon called the curse of dimensionality introduced by Bellman [76]. When the action space grows, the combinations of possible trajectories and computational requirements increase exponentially [28]. Therefore, the test objectives with multiple available actions tend to perform better compared to a single action combination. Carefully selecting the reward signal can help as with the first test scenario. However, longer single-path test objectives are still difficult to reach.

The challenge for the user is designing suitable test scenarios for the test generation. The training performance depends on several factors including the reward signal, actions, state, and hyperparameter selections such as the learning rate. Therefore, the goal of test generation should be creating a generic model for multiple applications. Nevertheless, expanding the framework to build a generic model and using the model to generate test cases across different applications is left for future research.

5.3 Future Research for Black-Box Test Generation

In the broader context, the target of the test generation is to have the complete pipeline in Figure 5 fully automated. Successfully overcoming the challenges of automating each section could help with anomaly detection in autonomous self-healing systems [77]. When focusing only on the limitations found in benchmarking, the focus of improvement should be on overcoming the problems with scaling up the state and action space.

As mentioned, one approach is to have a more advanced algorithm that learns the available actions in a state, such as the Q-learning-based approach used by Pan et al. [45]. In the approach, the actions were limited only to the legal ones in a state, unlike the thesis framework, which allowed the execution of any listed action at every step. However, extracting suitable state-action pairs might not be trivial, even with web applications. The situation gets more complicated when testing systems that don't automatically indicate the appropriate actions. Even though this poses a challenge for test generation, more advanced feature and action extraction methods are possible with machine learning. For instance, Khaliq et al. [48] used object detection for extracting features from mobile application screenshots.

The thesis demonstrated that the black-box test generation by sequence modeling is achievable from collected trajectories. Although ODT can generate test cases, the method has action and state space scalability issues similar to the PPO algorithm. Since transformers have been utilized for complex text-based decision-making in driving, as demonstrated by Sha et al. [78], and in interactive text environments, as shown by Gontier et al. [79], it is possible to resolve the limitations of state and action spaces. Similarly, the ODT algorithm could be adapted for text-to-text decision-making. The algorithm could utilize a large language model by directly inputting the HTML code as a state. The test objective could also take the form of plain text, thereby eliminating the need for a reward signal. The actions could consist of combinations of keywords and attributes in text format, which removes the need for preselected actions. Since the transformers use supervised and unsupervised methods, trajectories must be collected using the search methods for training these transformer-based language models. Moreover, transformers require an efficient method to store the website state to conserve tokens. Therefore, the algorithm must abstract the HTML document to a simple form.

6 Conclusion

The thesis explored methods around test generation and proposed a framework for executing the machine learning algorithms for test generation. The framework was designed to execute different types of algorithms for multiple test automation libraries. The aim was to be able to create a stochastic model of the available actions and generate test cases with the model trained for a specific test scenario.

With the framework, the thesis aimed to answer questions about the test generation. The first question concerned the optimization of the machine learning algorithms for generating script-based test cases. The critical aspect was finding the components of test generation that must be considered for successful test generation. The second and third questions aimed to compare the effectiveness of two algorithms, Proximal Policy Optimization (PPO) and Online Decision Transformer (ODT), in the context of test generation. The factors for comparison were the training performance and coverage. The thesis also considered future improvements for the presented test generation methods.

The two algorithms, PPO and ODT, were implemented and benchmarked in a local web application. The benchmarking phase had two testing scenarios, one with a simple login objective and another with a more complicated task of buying products. A machine learning policy was trained for each algorithm and scenario pair. The PPO algorithm learned the policy by exploring the application, whereas the ODT algorithm utilized the data gathered from the PPO training process. The evaluation phase used the resulting machine learning models to generate test cases in original and modified test environments for observing performance differences. In addition, the coverage was checked by plotting ten generated test cases.

The results demonstrated that achieving a carefully planned test objective is feasible with the framework by constructing a stochastic model and generating test steps based on this model. The main observation was that the PPO method was as efficient in creating the stochastic model by exploring the application while offering control over the generated test cases through the reward signal. In addition, the ODT algorithm was able to achieve similar test coverage using the trajectories collected from PPO training. However, selecting a proper reward signal, action space, and state space is critical for successful test generation. The tester must also be aware that the ODT training depends on the training data collected from the PPO training.

Based on the results, further research is needed in search-based testing and sequence modeling approaches. Although objectives with multiple paths were uncomplicated to explore, the test cases leading to objectives with only a few correct paths proved challenging to generate. Recently, curiosity-based test generation has shown progress in efficiently navigating websites. The algorithms benchmarked in the thesis could be expanded to include dynamic state-based actions instead of a fixed number of actions. In addition, the ODT algorithm could perform the decision-making as a text-to-text task while removing the need for state and action spaces. The developed models can then be trained and utilized for test generation across various applications.

In conclusion, test case generation provides an approach that expands beyond traditional regression testing. The created test framework offers a foundation for further exploration into machine learning methods for black-box and white-box testing. In addition, the approach has significant potential to enhance the efficiency of the testing process.

References

[1] D. Graham, R. Black, and E. Veenendaal, Foundations of Software Testing: ISTQB Certification, 4th ed. Hampshire, UK: Cengage Learning EMEA, 2019.

[2] P. Ammann and J. Offutt, Introduction to software testing, 2nd ed. Cambridge, UK: Cambridge University Press, 2016.

[3] M. J. Harrold, "Testing: a roadmap," in Proceedings of the Conference on the Future of Software Engineering, 2000, pp. 61-72.

[4] Gartner, Inc., "Gartner Forecasts Worldwide Public Cloud End-User Spending to Reach Nearly $600 Billion in 2023", Gartner.com, 2022. Accessed: Jun 26, 2023. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2022-10-31-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-600-billion-in-2023

[5] International Software Testing Qualifications Board, "Certified Tester Advanced Level Test Analyst (CTAL-TA) Syllabus, v3.1.1," May 15, 2021. Accessed: May 6, 2023. [Online]. Available: https://astqb.org/assets/documents/ISTQB_CTAL-TA_Syllabus_v3.1.1.pdf

[6] V.H. Durelli et al., "Machine learning applied to software testing: A systematic mapping study," in IEEE Transactions on Reliability, vol. 68, no. 3, pp. 1189-1212, 2019.

[7] S. Anand et al., "An orchestrated survey of methodologies for automated software test case generation," in Journal of Systems and Software, vol. 86, no. 8, pp. 1978–2001, 2013.

[8] A. Fontes and G. Gay, "The integration of machine learning into automated test generation: A systematic mapping study," in Software Testing, Verification and Reliability, p. e1845, May. 2023. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/stvr.1845

[9] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit Test Case Generation with Transformers and Focal Context," May, 2021, [Online]. Available: https://arxiv.org/abs/2009.05617

[10] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "Automatic testing of GUI-based applications," in Software Testing, Verification and Reliability, vol. 24, no. 5, pp. 341-366, 2014.

[11] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI ripping for automated testing of Android applications," in Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, 2012, pp. 258–261.

[12] D. P. Nguyen and S. Maag, "Codeless web testing using selenium and machine learning," in ICSOFT 2020: 15th International Conference on Software Technologies, 2020, pp. 51-60.

[13] G. J. Myers, T. Badgett, and C. Sandler, The Art of Software Testing, 3rd ed. Hoboken, NJ, USA: Wiley, 2011. doi: 10.1002/9781119202486.

[14] A. Spillner and T. Linz, Software Testing Foundations, 5th Edition, 5th ed. Rocky Nook, 2021.

[15] N. B. Ruparelia, "Software development lifecycle models," in ACM SIGSOFT Software Engineering Notes, vol. 35, no. 3, pp. 8-13, 2010. doi: 10.1145/1764810.1764814

[16] R. Hoda, N. Salleh, and J. Grundy, "The rise and evolution of agile software development," in IEEE Software, vol. 35, no. 5, pp. 58–63, 2018.

[17] Y. B. Leau, W. K. Loo, W. Y. Tham, and S. F. Tan, "Software development life cycle AGILE vs traditional approaches," in Proc. Int. Conf. Inf. Netw. Technol., vol. 37, no. 1, pp. 162–167, 2012.

[18] K. Wiklund, S. Eldh, D. Sundmark, and K. Lundqvist, "Impediments for software test automation: A systematic literature review," Software Testing, Verification and Reliability, vol. 27, no. 8, Art. no. e1639, 2017, doi: 10.1002/stvr.1639

[19] B. Haugset and G. K. Hanssen, "Automated acceptance testing: A literature review and an industrial case study," in Agile 2008 Conference, 2008, pp. 27-38.

[20] J. Kasurinen, O. Taipale, and K. Smolander, "Software test automation in practice: empirical observations," Advances in Software Engineering, vol. 2010, 2010. doi: 10.1155/2010/620836

[21] M. Fewster and D. Graham, Software Test Automation: Effective Use of Test Execution Tools. ACM Press/Addison-Wesley Publishing Co., 1999.

[22] O. Taipale, J. Kasurinen, K. Karhu, and K. Smolander, "Trade-off between automated and manual software testing," International Journal of System Assurance Engineering and Management, vol. 2, pp. 114-125, 2011. doi: 10.1007/s13198-011-0065-6

[23] E. Alpaydin, Introduction to Machine Learning, 4th ed. Cambridge, Massachusetts: The MIT Press, 2020.

[24] A. Burkov, The Hundred-Page Machine Learning Book, vol. 1. Quebec City, QC, Canada: Andriy Burkov, 2019.

[25] A. Jung, Machine Learning: The Basics, Springer, Singapore, 2022. doi: 10.1007/978-981-16-8193-6

[26] A. Géron, Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 3rd ed. Sebastopol, CA: O'Reilly Media, Inc., Oct. 2022.

[27] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[28] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction. MIT Press, 2018.

[29] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, 4th US ed. University of California, Berkeley, 2021.

[30] P. W. Chou, D. Maturana, and S. Scherer, "Improving stochastic policy gradients in continuous control with deep reinforcement learning using the beta distribution," in Proc. of the International Conference on Machine Learning, 2017, pp. 834-843.

[31] M. Yogeswaran and S. G. Ponnambalam, "Reinforcement learning: Exploration–exploitation dilemma in multi-agent foraging task," Opsearch, vol. 49, pp. 223-236, 2012.

[32] A. Vaswani et al., "Attention is all you need," in Advances in Neural Information Processing Systems, vol. 30, 2017.

[33] L. Floridi and M. Chiriatti, "GPT-3: Its nature, scope, limits, and consequences," Minds and Machines, vol. 30, pp. 681-694, 2020.

[34] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," arXiv preprint arXiv:1409.1259, 2014.

[35] J. A. Whittaker, "Stochastic software testing," Annals of Software Engineering, vol. 4, no. 1, pp. 115-131, 1997.

[36] C. Pacheco, S. K. Lahiri, and T. Ball, "Finding errors in. net with feedback-directed random testing," in Proceedings of the 2008 International Symposium on Software Testing and Analysis, 2008, pp. 87-96.

[37] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for Java," in Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion, 2007, pp. 815-816.

[38] E. Arteca, S. Harner, M. Pradel, and F. Tip, "Nessie: automatically testing JavaScript APIs with asynchronous callbacks," in Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 1494-1505.

[39] M. Harman and B. F. Jones, "Search-based software engineering," Information and Software Technology, vol. 43, no. 14, pp. 833-839, 2001.

[40] M. Harman, F. Islam, T. Xie, and S. Wappler, "Automated test data generation for aspect-oriented programs," in Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development, 2009, pp. 185-196.

[41] C. D. Nguyen et al., "Evolutionary testing of autonomous software agents," Autonomous Agents and Multi-Agent Systems, vol. 25, pp. 260-283, 2012.

[42] N. Alshahwan and M. Harman, "Automated web application testing using search based software engineering," in 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), 2011, pp. 3-12.

[43] H. N. Yasin, S. H. A. Hamid, and R. J. Raja Yusof, "Droidbotx: Test case generation tool for android applications using Q-learning," Symmetry, vol. 13, no. 2, p. 310, 2021.

[44] Y. Zheng et al., "Automatic web testing using curiosity-driven reinforcement learning," in Proc. 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 423–435.

[45] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of Android applications," in Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020, pp. 153-164.

[46] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box android app testing," in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 1070-1073.

[47] K. R. Chowdhary, "Natural language processing," in Fundamentals of Artificial Intelligence, Springer, 2020, pp. 603-649.

[48] Z. Khaliq, D. A. Khan, and S. U. Farooq, "Using deep learning for selenium web UI functional tests: A case-study with e-commerce applications," Engineering Applications of Artificial Intelligence, vol. 117, p. 105446, 2023.

[49] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "Adaptive test generation using a large language model," arXiv preprint arXiv:2302.06527, 2023.

[50] NVIDIA, 'What are Large Language Models?,' NVIDIA Glossary, 24 March 2023. Accessed: Dec 14, 2023. [Online]. Available: https://www.nvidia.com/en-us/glossary/data-science/large-language-models/

[51] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "Toga: A neural method for test oracle generation," in Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 2130-2141.

[52] H. Almulla and G. Gay, "Learning how to search: generating exception-triggering tests through adaptive fitness function selection," in 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), 2020, pp. 63-73.

[53] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, "Combining multiple coverage criteria in search-based unit test generation," in Search-Based Software Engineering: 7th International Symposium, 2015, pp. 93-108.

[54] D. Silver et al., "Deterministic policy gradient algorithms," in Proceedings of the International Conference on Machine Learning, 2014, pp. 387-395.

[55] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," arXiv preprint arXiv:1707.06347, 2017.

[56] A. Zheng and A. Casari, Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists. Sebastopol, CA, United States: O'Reilly Media, Inc., 2018.

[57] T. A. T. Vuong and S. Takada, "A reinforcement learning based approach to automated testing of android applications," in Proc. 9th ACM SIGSOFT Int. Workshop on Automating TEST Case Design, Selection, and Evaluation, 2018, pp. 31-37.

[58] T. Shu, C. Wu, and Z. Ding, "Boosting input data sequences generation for testing EFSM-specified systems using deep reinforcement learning," Information and Software Technology, vol. 155, p. 107114, 2023.

[59] M. Esnaashari and A. H. Damia, "Automation of software test data generation using genetic algorithm and reinforcement learning," Expert Systems with Applications, vol. 183, p. 115446, 2021.

[60] S. Reddy, C. Lemieux, R. Padhye, and K. Sen, "Quickly generating diverse valid test inputs with reinforcement learning. In 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)," 2020.

[61] Q. Zheng, A. Zhang, and A. Grover, "Online decision transformer," in Proc. Int. Conf. Machine Learning, 2022, pp. 27042-27059.

[62] L. Chen et al., "Decision transformer: Reinforcement learning via sequence modeling," in Advances in Neural Information Processing Systems, vol. 34, pp. 15084-15097, 2021.

[63] OpenAI, "Proximal Policy Optimization," Spinning Up in Deep RL. [Online]. Accessed: Nov 11, 2023. Available: https://spinningup.openai.com/en/latest/algorithms/ppo.html

[64] Y. Wang, H. He, and X. Tan, "Truly proximal policy optimization," in Proc. Uncertainty in Artificial Intelligence, 2020, pp. 113-122.

[65] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-Dimensional Continuous Control Using Generalized Advantage Estimation," 2018, arXiv:1506.02438

[66] OpenAI, "Experiment Details," Spinning Up Documentation, [Online]. Accessed: Nov. 22, 2023. Available: https://spinningup.openai.com/en/latest/spinningup/bench.html

[67] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," in J. Mach. Learn. Res., vol. 15, no. 1, pp. 1929-1958, 2014.

[68] A. Shih, D. Sadigh, and S. Ermon, "Long Horizon Temperature Scaling," 2023, arXiv preprint arXiv:2302.03686.

[69] AurelianTactics. "PPO Hyperparameters and Ranges." Medium, 25 July 2018. [Online]. Accessed: Jun 8, 2023. Available: https://medium.com/aureliantactics/ppo-hyperparameters-and-ranges-6fc2d29bccbe

[70] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," 2019, arXiv preprint arXiv:1810.04805.

[71] G. Brockman et al., "OpenAI Gym," 2016, arXiv preprint arXiv:1606.01540.

[72] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta, "Agile Software Development Methods: Review and Analysis," 2017. [Online]. Available: arXiv:1709.08439.

[73] X. Ying, "An overview of overfitting and its solutions," in Journal of Physics: Conference Series, vol. 1168, 022022, 2019.

[74] A. Field, Discovering Statistics Using IBM SPSS Statistics, 5th ed. Los Angeles, CA: SAGE, 2018.

[75] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, "Reinforcement learning for android GUI testing," in Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, 2018, pp. 2-8.

[76] R. Bellman, "The theory of dynamic programming," Bulletin of the American Mathematical Society, vol. 60, no. 6, pp. 503-515, 1954.

[77] M. A. Naqvi, M. Astekin, S. Malik, and L. Moonen, "Adaptive Immunity for Software: Towards Autonomous Self-healing Systems," in 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2021, pp. 521-525.

[78] H. Sha et al., "LanguageMPC: Large Language Models as Decision Makers for Autonomous Driving," 2023. [Online]. Available: arXiv:2310.03026.

[79] N. Gontier, P. Rodriguez, I. Laradji, D. Vazquez, and C. Pal, "Long-Context Language Decision Transformers and Exponential Tilt for Interactive Text Environments," arXiv preprint arXiv:2302.05507, 2023.

[80] D. Lawson, "online-decision-transformer," GitHub repository, [Online]. Accessed on: Sep 10, 2023. Available: https://github.com/daniellawson9999/online-decision-transformer

A Experimental Details

The test generation framework and implemented algorithms are available from: https://github.com/rikulehtonen/Test-Case-Generation-Masters-Thesis

All of the models created in thesis were trained with consumer-grade laptop with NVIDIA T500 and 4 GB of GPU memory.

A.1 Proximal Policy optimization

The PPO hyperparameters follow ranges from a post by AurelianTactics [69]. The parameters were manually searched, by selecting the best values for the pre-training. Furthermore, the learning rate was optimized with Optuna for both test scenarios. The selected hyperparameters are listed in Table A1

Table A1: Hyperparameters for training with PPO. Pre-training used the parameters from test scenario 1.

Hyperparameters		
Hyperparameter	Test Scenario 1	Test Scenario 2
learning rate	1e-3	5.558e-4
gamma	0.99	0.99
gae lambda	0.95	0.95
clip	0.2	0.2
max timesteps	20	20
batch timesteps	20	20
minibatches	3	3
episode max timesteps	20	20
iteration epochs	4	4

A.2 Online Decision Transformer

The online decision transformer was implemented and the hyperparameters were selected based on the approach by Chen et al. [62]. Furthermore, the final parameters were partly selected from implementation by Lawson [80]. The ODT used the GPT-2 model for the benchmarking. The selected hyperparameters are listed in Table A2

Table A2: Hyperparameters for training with ODT. Pre-training used the parameters from test scenario 1.

Hyperparameters for both test scenarios		
Hyperparameter	Offline Pre-training	Online Fine-tuning
learning rate	1e-4	1e-4
weight decay	5e-4	5e-4
batch size	64	256
max iters	5	10
num steps per iter	100	1000
max ep len	20	20
reward target	1600	1600
pct traj	1.0	1.0
embed dim	512	512
n layer	4	4
n head	4	4
warmup steps	500	1000
num eval episodes	30	30
K (context length)	20	20
dropout	0.1	0.1
online buffer size	1000	1000

A.3 Benchmarking Environment

A custom benchmarking application has been built to empirically validate the test generation. Figures A1 and A2 display the 4 pages of the benchmarking application.

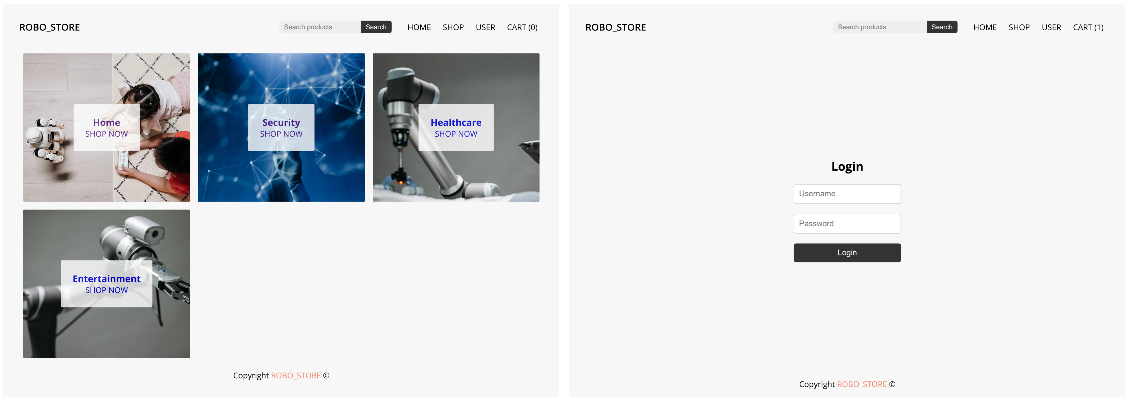
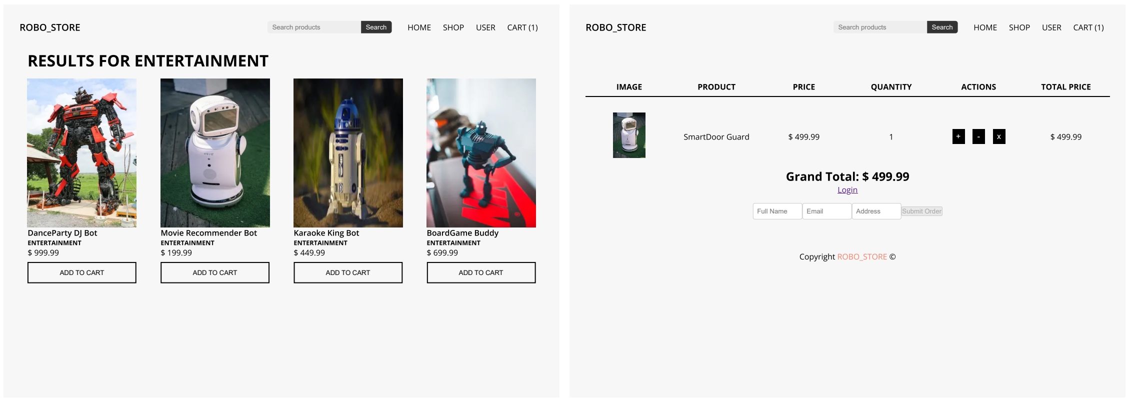


Figure A1: Screenshots of the benchmarking application. On the left is the front page, and on the right is the login page.



RESULTS FOR ENTERTAINMENT

DanceParty DJ Bot
ENTERTAINMENT
$ 999.99

Movie Recommender Bot
ENTERTAINMENT
$ 199.99

Karaoke King Bot
ENTERTAINMENT
$ 449.99

BoardGame Buddy
ENTERTAINMENT
$ 699.99

IMAGE	PRODUCT	PRICE	QUANTITY	ACTIONS	TOTAL PRICE
	SmartDoor Guard	$ 499.99	1	+ - x	$ 499.99

Grand Total: $ 499.99

Figure A2: Screenshots of the benchmarking application. On the left is the results page, which selects the products based on the query: category, search, or all products. On the right is the shopping cart page, which displays the total sum of products added to the shopping cart.

The original environment is also modified to evaluate the model performance to simulate application with bugs or new updates. The modified environment with removed login page link for first test scenario in Figure A3. Figure A4 displays the benchmarking application with removed links for the second test scenario.

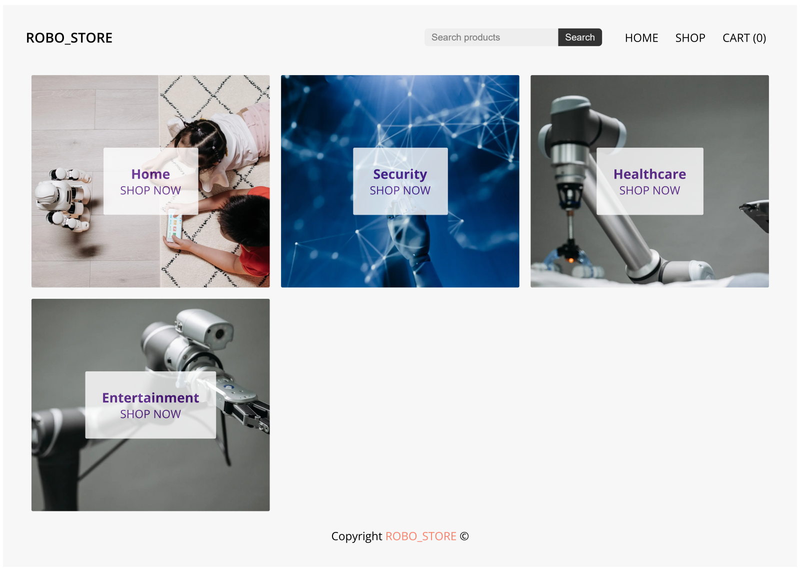


Figure A3: Modified application 2 for the second test scenario. The login page link "USER" is only available on product and shopping cart pages. The link is removed from the front page.

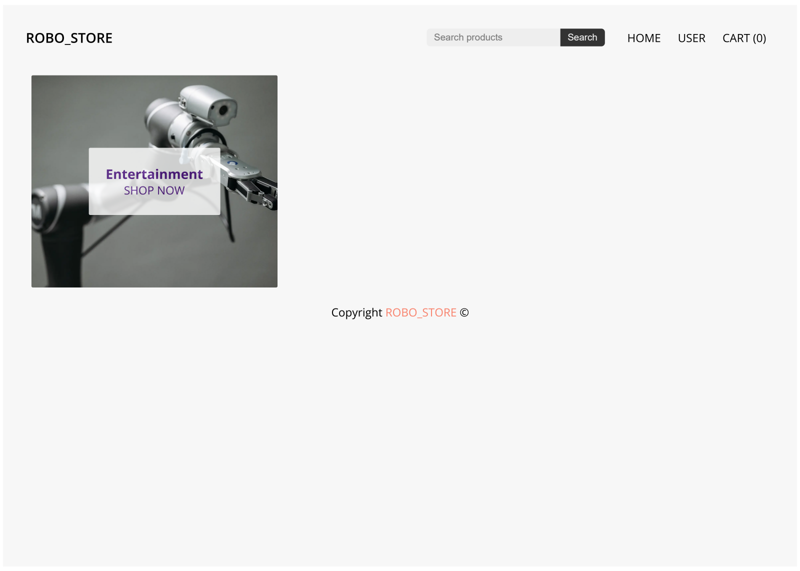


Figure A4: Modified application 2 for the second test scenario. On the front page, 3 category links and all products link is removed to evaluate performance in the updated environment.