Deep Reinforcement Learning in Automated User Interface Testing

Juha Eskonen

School of Science

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

Espoo, Finland 4.4.2019

Supervisor

Prof. Alexander Jung

Advisor

D.Sc. (Tech.) Jukka Ylitalo

Aalto University
School of Science

Aalto University
School of Science

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

Author Juha Eskonen	
Title Deep Reinforcement Learning in Automated User Interface Testing	
Degree programme Computer, Communication and Information Sciences	
Major Computer Science	**Code of major** SCI3042
Supervisor Prof. Alexander Jung	
Advisor D.Sc. (Tech.) Jukka Ylitalo	
Date 4.4.2019 **Number of pages** 57	**Language** English

Abstract

This thesis introduces deep reinforcement learning methods for finding problems in user interfaces. To find as many problems as possible the algorithms are trained to efficiently explore the user interface. The thesis proposes a method that takes a screenshot of the UI as input and outputs what to click next. In this new method, the algorithm learns to explore 4 times as efficiently as a random algorithm and 25% more efficiently than an inexperienced user. The exploration efficiency is almost on par with an experienced user.

Keywords deep reinforcement learning, software testing, exploratory testing, machine learning, user interface

Aalto University
School of Science

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

Tekijä Juha Eskonen		
Työn nimi Deep Reinforcement Learning in Automated User Interface Testing		
Koulutusohjelma Computer, Communication and Information Sciences		
Pääaine Computer Science		**Pääaineen koodi** SCI3042
Työn valvoja ja ohjaaja Prof. Alexander Jung		
Päivämäärä 4.4.2019	**Sivumäärä** 57	**Kieli** Englanti

Tiivistelmä

Tämä diplomityö esittää deep reinforcement learning menetelmiä ongelmien löytämiseen käyttöliittymissä. Jotta ongelmia löydetään mahdollisimman paljon, algoritmit opetetaan tutkimaan käyttöliittymää tehokkaasti. Diplomityö esittää menetelmän, joka hyödyntää käyttöliittymän näyttökaappausta ja kertoo mitä seuraavaksi klikataan. Tämä uusi menetelmä oppii navigoimaan neljä kertaa yhtä tehokkaasti kuin satunnaisalgoritmi ja 25% tehokkaammin kuin kokematon käyttäjä. Navigoinnin tehokkuus on melkein samalla tasolla kuin kokeneella käyttäjällä.

Avainsanat deep reinforcement learning, software testing, exploratory testing, machine learning, user interface

Preface

Thanks to Alex Jung, Juha Törrönen, Jukka Ylitalo, and Julen Kahles for all the help with the thesis process and ideas through fruitful discussions. Thanks to Adam Peltoniemi for proofreading my thesis.

I want to thank my parents for supporting me throughout my studies. Thanks to them, I also learned to think like a programmer and program in Visual Basic when I was young.

This thesis would not have been possible without four Juhas. I want to thank Juha Eloranta, Juha Törrönen, and Juha Kallioinen for great guidance. Thanks to Ericsson for providing the hardware to make experimenting faster and easier.

Espoo, Finland, 4.4.2019

Juha Eskonen

Contents

Abstract 3

Abstract (in Finnish) 4

Preface 5

Contents 6

Symbols and abbreviations 8

1 Introduction 9
- 1.1 Problem setting . . . 10
- 1.2 Research methods . . . 11
- 1.3 Structure of the system under test . . . 12
- 1.4 Related Work . . . 13
- 1.5 Structure of the thesis . . . 14

2 Background 16
- 2.1 Supervised learning . . . 16
- 2.2 Reinforcement learning . . . 17
- 2.3 Artificial Neural Networks . . . 18
  - 2.3.1 Convolutional Neural Networks . . . 19
  - 2.3.2 Recurrent Neural Networks . . . 20
- 2.4 Deep Reinforcement Learning . . . 21
  - 2.4.1 Actor-Critic . . . 22
  - 2.4.2 A3C . . . 23
- 2.5 Existing UI navigation methods . . . 25
  - 2.5.1 Q-learning-based testing . . . 25
  - 2.5.2 Workflow-Guided Exploration . . . 26

3 Proposed methods 28
- 3.1 Base deep reinforcement learning algorithm . . . 29
- 3.2 Element-based approach . . . 31
  - 3.2.1 Problems of the Element-based approach . . . 33
  - 3.2.2 Advantages of the Element-based approach . . . 34
- 3.3 Image-based approach . . . 34
  - 3.3.1 Relative Movement . . . 35
  - 3.3.2 Absolute Movement . . . 36
  - 3.3.3 Click probability image . . . 37
  - 3.3.4 Paint in the area of the clicked element . . . 38
  - 3.3.5 Means of elements in the prediction image . . . 38

4 Numerical results **42**
4.1 Evaluation criteria . . . 42
4.2 Baseline algorithms . . . 43
4.2.1 Random Agent . . . 43
4.2.2 Baseline Q-learning algorithm . . . 43
4.3 Test environments . . . 44
4.3.1 Simple site . . . 45
4.3.2 Single-page application . . . 45
4.3.3 Real product . . . 46
4.4 Exploration: Q-learning versus random search . . . 46
4.5 Testing the algorithm on a real product . . . 47
4.6 Performance and robustness . . . 48
4.7 Curiosity in UI exploration . . . 49

5 Conclusion **52**
5.1 For Future Studies . . . 53
5.1.1 Generalization . . . 53
5.1.2 Content . . . 53
5.1.3 Reward function . . . 53

Symbols and abbreviations

Symbols

t	Time step
s_t	State at time step t
a_t	Action at time step t
$Q(s_t, a_t)$	Action-value function
$V(s_t)$	State-value function
r_t	Reward at time step t
G_t	Return at time step t
π	Policy function
γ	Discount factor
$A(a_t, s_t)$	Advantage function
H	Entropy
$\in$	Belongs to (a set)
$\subset$	Is a subset of (a set)

Abbreviations

A2C	Synchronous Advantage Actor Critic
A3C	Asynchronous Advantage Actor Critic
API	Application Programming Interface
CNN	Convolutional Neural Network
DOM	Document Object Model
DNN	Deep Neural Network
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTML	Hypertext Markup Language
LSTM	Long Short Term Memory
MDP	Markov Decision Process
NN	Neural Network
PPO	Proximal Policy Optimization
REST	Representional state transfer
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SPA	Single Page Application
SUT	System Under Test
UI	User Interface
URL	Uniform Resource Locator
WGE	Workflow-Guided Exploration

1 Introduction

To ensure software quality it is essential to test the software during the development process. There are two main approaches to testing: manual and automated. Manual tests are performed by human testers, who go through the steps of a test case and verify that it works as intended. In automated testing, however, the steps and the verification are scripted. Therefore, automated tests can be run multiple times without human effort. On the other hand, manual testing is adaptive, because it is performed by the testers, but automated tests need to be rewritten when the software is modified. Changes are made rapidly in agile software testing environments, hence rewriting the tests takes a large portion of the development time. Nevertheless, manual testing is also very time-consuming. This motivates the research of novel testing methods that are both adaptive and automated.

Machine learning methods have shown an ability to learn directly from examples and generalize to previously unseen data. Applications of machine learning include the capability to detect what is in a picture, as shown in [21] [23], classifying reviews as positive or negative [28], and predicting stock prices [9]. After training, these methods can perform the taught task automatically without human effort.

Many successful machine learning methods are supervised. For training, supervised learning approaches require examples of input data and wanted outcomes for each data point. These methods, especially the deep learning-based ones, require large amounts of collected data to perform well. For example, accurate image classification algorithms have been trained using large datasets, such as ImageNet [10], which contains millions of images with desired output labels. While these methods work well, the downside is that collecting and labeling the data can be very manual and laborious work.

Reinforcement learning (RL) is an area of machine learning that, in contrast to supervised learning, does not require datasets in the form of correct input and output pairs. [36] In reinforcement learning an "agent" learns in an environment by choosing actions and obtaining rewards. The agent learns to maximize the reward by choosing the right actions in each state of the environment. While traditional reinforcement learning methods, such as tabular Q-learning, perform well on small scale problems, they struggle to scale up. In fact, user interfaces are almost always too complex for those methods.

Deep reinforcement learning (deep RL) combines deep neural networks (DNNs) with reinforcement learning methods. The use of DNNs allows scaling to larger problems. For example, Deep Q-Learning replaces the table of tabular Q-learning with a deep neural network, allowing faster and more memory-efficient learning. Recent advancements in deep RL have shown that agents can learn to play games and navigate in game environments. Algorithms such as DeepMind's AlphaGo [35] and DQN [27] are able to outperform human players in the games they were taught to play.

Reinforcement learning methods are appealing for software testing since they could learn directly by interacting with the software. An additional benefit over supervised learning methods is that instead of requiring human demonstrations, only a reward function is needed. The reward function is a way to define when the agent gets rewarded and how large the reward is. The reward can be positive or negative. Regarding UI testing, one way to design a reward function for the RL algorithm would be to give a positive reward for each problem that is found. Thus the RL algorithm should learn to maximize the number of problems. However, it is not useful to find the same problem again and it is more beneficial if the agent broadly tests the functionality of the system that is being tested. Furthermore, if the problems are difficult to find, the agent receives too little or no reward at all and thus learns nothing. Therefore, this thesis discusses methods that can be used to efficiently explore the user interface. The idea is to detect problems in the software automatically while exploring with an efficient algorithm.

This thesis focuses on web-based UIs, but the exploration and problem detection could be implemented for native desktop and mobile applications using the applicable APIs. Web applications are often separated into two parts: the frontend and the backend. The frontend contains the user interface and logic for handling user actions such as button clicks and communication with the backend. The backend stores the needed information and provides it to the frontend. In the context of a web application, there are existing exploratory testing methods for finding errors and security flaws in the backend. One such method is fuzz testing [11]. In fuzz testing a program automatically tests how the application behaves with different fuzzed, i.e. modified, inputs. Fuzz testing can be used to find problems in APIs, such as inputs that cause server errors.

However, many frontend testing tools require either app-specific programming or recording actions. The tests can be written using actions such as *find element*, *click* and *input text*. An alternative way is to record a real user using the application and then replay the actions when the test is run. These tests require manual work and can break easily when the software changes. This raises the question; is it possible to eliminate the need for manual work in user interface testing with the help of deep reinforcement learning?

1.1 Problem setting

The goal of this thesis is to create a system that can automatically explore the user interface of a web application and detect problems concurrently. UI-related problems can be categorized as functionality-related, usability-related, and appearance-related problems. Functional problems are those that prevent the application from working the way it is intended. This is the case, for instance, when clicking a button crashes the application. Usability problems are related to how easy it is for the user to perform the wanted task, e.g. whether it is easy to navigate to a certain page or not. The complications that affect the ability to see the important content are called visual problems. For example, text can be hard to read because of the background color. This thesis focuses on functional problems, but detection of other problem types could also be added alongside the exploration algorithm.

We are especially interested in testing web applications that have a JavaScript-based frontend, which communicates with a backend over a REST API. The functional problems we can find in these kinds of applications are JavaScript errors and errors related to communication with the backend. A request to the backend can cause a server error that is received by the frontend.

The main component of the automated testing system is an algorithm that can efficiently navigate the user interface. This thesis discusses how we can train such an algorithm using deep reinforcement learning. Deep reinforcement learning algorithms are able to adapt to changes because they can be readjusted or trained again from scratch after the software is modified. This thesis uses the number of unique UI states as the reward function of the Deep RL algorithms. Thus, they should learn to choose what to click so that it takes them to states that have not been seen before, in other words, learn to explore.

Many deep reinforcement learning methods are tested with video games due to the ease of use, diversity, availability to everyone, and ability to run an arbitrary number of environments simultaneously. Therefore it makes sense to compare the UI exploration problem to video games in order to predict what kinds of challenges there are and how well the problem can be expected to be solved. The video games that have been used as benchmarks for deep reinforcement learning usually have a small number of actions. In contrast, user interfaces can have a large number of elements, such as links, input elements, and buttons. Thus, rather than requiring a lot of correct consequent actions to earn rewards, such as going through a maze in Doom or finding a specific room or completing a level in Mario [29], this UI exploration task has a potentially small number of steps to get a reward, but the number of possible actions to choose from is large. The solutions for this problem are discussed in Section 3.

This thesis will answer the following research questions:

- How can we use deep reinforcement learning to automate exploratory software testing?
- How should we design the action space and states so that the agent is able to learn patterns?
- Can we use supervised learning for initial training of the agent?

1.2 Research methods

The following three methods were used to build the system for UI testing using deep reinforcement learning:

- Discussing with frontend and backend developers to find out what kind of testing tool is useful and how it could work
- Exploring existing algorithms for this specific problem and testing them
- Experimenting with algorithms that have been proven to work well for similar problems

The development of each version of the exploration algorithm followed the iterative cycle below:

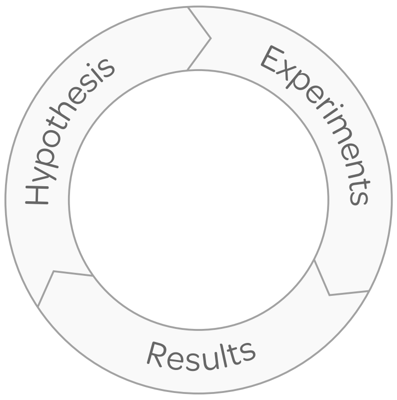


Figure 1: Iterative research cycle.

1. Hypothesis: An idea of what could work, what should be tested, how it could work
2. Experiments: Implementing the idea, testing how it works
3. Results: Looking at the performance of the method based on chosen metrics, comparing it to the other methods and baselines

Training the RL algorithms on the real product turned out to be slow. So, to make experimentation fast, many environments with varying levels of difficulty were created for performance testing. It was assumed that if the RL algorithm did not perform well on a simpler environment then a better algorithm was required to proceed to the more difficult environments. The different test environments are explained in detail in Section 4.3.

1.3 Structure of the system under test

System under test (SUT) refers to the system, in this thesis a web application, which is tested to ensure it functions correctly. This thesis uses test environments with varying capabilities to test the performance of the algorithms before moving to the real product. The test environments were designed to be simple, but still behave similarly to the actual testing target. The target system is a complex web application with a JavaScript-based frontend and a microservices backend.

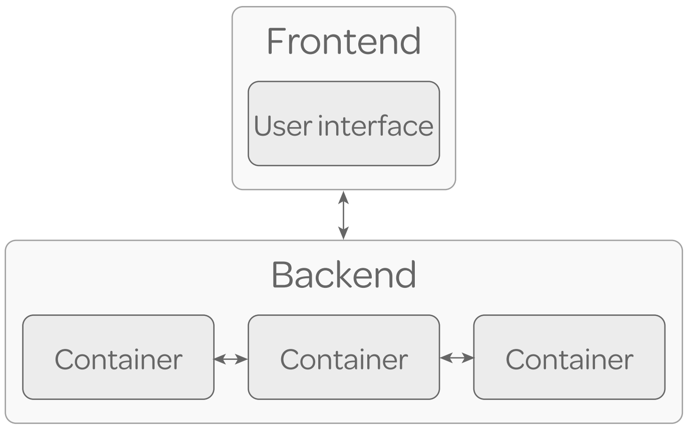


Figure 2: Frontend-backend architecture with containers.

The backend consists of microservices [8], where each microservice lies in its own container [25]. The microservices communicate with each other over REST API. This thesis focuses on testing the frontend, but parts of the functionality of the backend containers is also tested by checking if the requests sent by the frontend result in server errors on the backend side.

1.4 Related Work

The use of reinforcement learning to test user interfaces is not broadly explored in the research community. This thesis focuses on deep reinforcement learning methods that do not use training data at all, with the goal of building a solution that can be run with a single command. There are existing solutions for navigating user interfaces, but most of them use precollected human demonstrations. Furthermore, these solutions are also aimed at the use cases where some kind of web task is to be automated, rather than software testing. For example, in one use case the methods are used to automate booking of plane tickets. As training data they have existing demonstrations that they either use to create new similar action sequences [22] or are able to adapt the demonstrations to changes [24].

Related work can be divided to three categories. One category is the set of methods that use reinforcement learning to navigate a user interface. The methods in this category are the most relevant to solving the UI exploration problem. The second category contains methods that use reinforcement learning in software testing, but to test parts of the software other than the UI. Finally, in the third category, there are the general purpose deep reinforcement learning methods. The performance of these algorithms is usually tested by teaching them to play video games. Since the methods in the first and second categories have their difficulties, this thesis applies the third category algorithms for UI testing and compares the performance against those in the other categories.

There are existing methods that use reinforcement learning to control a graphical user interface, such as Q-learning-based testing [4] and Workflow-Guided Exploration (WGE) [22]. While choosing what to click randomly is a simple way to test a user interface, it can be inefficient. Q-learning-based testing improves on this by utilizing Q-learning to choose actions more wisely than choosing randomly. WGE uses human demonstrations to accelerate learning and to make the exploration more focused. In addition to those two, there are methods that utilize machine learning to make automated tests recover from problems as shown in [24]. Q-learning-based testing and WGE are discussed in detail in Sections 2.5.1 and 2.5.2, respectively.

Deep reinforcement learning has also been used to generate input for tests. Kim et al. [19] used Deep RL to create learned metaheuristic algorithms that achieve up to 100% branch coverage. Böttinger et al. used deep reinforcement learning to find security vulnerabilities [6].

There are many deep reinforcement learning algorithms that have shown great performance in video games such as: Deep Q Network (DQN) [27], Rainbow [14], Asynchronous Advantage Actor Critic (A3C) [26], synchronous Advantage Actor Critic (A2C) and sample efficient method ACKTR [38], and Proximal Policy Optimization (PPO) [34]. These algorithms learn to play games using just the current game frame and a reward function. Graphical user interfaces can be also rendered as images like games and a reward function can be designed. Based on this idea, this thesis evaluates how well Deep RL methods work in UI exploration.

Writing deep reinforcement learning algorithms from scratch is difficult due to the fact that small bugs in the code are difficult to track, but may still cause significant drops in performance. Therefore, an existing A3C GPU implementation [13] was chosen as the base for the methods proposed in this thesis. The implementation is attractive due to its parallelizability and possibility to use a graphics processing unit (GPU) for speeding up the inference of convolutional neural networks (CNNs) and recurrent neural networks (RNNs). The A3C implementation shows fast training. It learns many Atari games in under 30 minutes and achieves state of the art results for some games. More details about the implementation can be found in Section 2.4.2.

This thesis introduces a novel way to use deep reinforcement learning in user interface testing. The introduced deep RL methods learn to navigate the UI in a way that maximizes a provided reward function. This allows them to work more efficiently than the previous methods. Additionally, the reward function can be changed to test specific parts of the user interface.

1.5 Structure of the thesis

The rest of the thesis is structured as follows. Section 2 discusses the main building blocks used for the development of UI testing methods. In particular, we will give a brief introduction to supervised learning, reinforcement learning, and artificial neural networks. Moreover, we will discuss how deep learning can be combined with reinforcement learning to obtain deep reinforcement learning methods. The proposed methods are described in Section 3. There are two different approaches: element-based and image-based. The section starts by explaining the deep reinforcement learning algorithm that both of the approaches are based on. Then it explores in further detail how the two approaches work and how they differ. Section 4 shows the results by starting with baseline algorithms that the deep reinforcement learning algorithms are compared against and then shows the improved performance of the proposed methods. Finally, Section 5 concludes this thesis.

2 Background

This section will cover the theory behind the exploration algorithms that are described in the proposed methods in Section 3. Supervised learning is introduced first. After that we will discuss reinforcement learning, which is, in contrast to supervised learning, a way to teach machine learning algorithms by interaction with the environment rather than imitating the training data. Then in the next section, artificial neural networks are described. Finally, in the deep reinforcement learning section, we show how neural networks and reinforcement learning are combined to form efficient learning algorithms.

2.1 Supervised learning

Many successful machine learning methods use supervised learning, which means that they use labelled datasets to learn patterns. Datasets consist of observations (also known as data points). If the dataset is for example an image classification dataset, then an observation is a single image and a label pair.

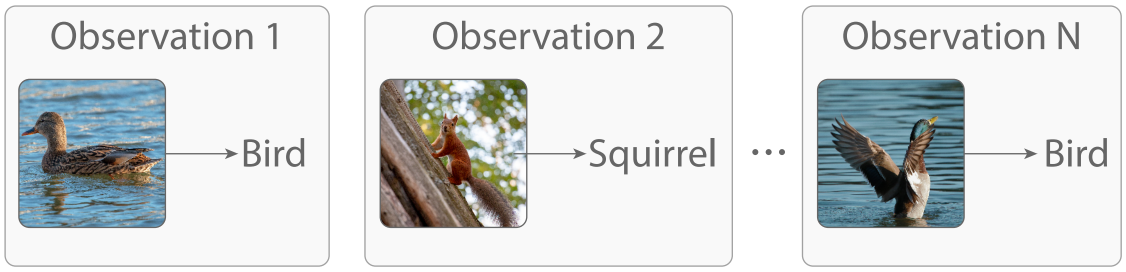


Figure 3: An example of an image classification dataset with N observations.

Supervised learning problems can be described using the components: feature space, label space, hypothesis space and a loss function [18]. Feature space is the space of all possible feature vectors. Feature vectors are d-dimensional vectors that contain attributes. The attributes can be, for example, information about whether text has been entered to a username input box. Label space is the set of possible outputs. An example of an output is "click login" or "type password". Hypothesis space contains the functions that are mappings from the feature to the label space. A simple hypothesis function h is:

$$h(x) = \begin{cases} \text{"Click login"} & \text{if } x = \text{"credentials written"} \\ \text{"Type credentials"}, & \text{otherwise} \end{cases} \tag{1}$$

Hypothesis functions can be very complex. For example, in this thesis they are deep neural networks that do millions of multiplications and additions.

The quality of the hypothesis function can be measured using loss functions. After a loss function is defined, the hypothesis functions can be optimized by finding the parameters that minimize the loss. The parameter tuning is done by using optimization algorithms, such as stochastic gradient descent [32].

2.2 Reinforcement learning

The idea of reinforcement learning (RL) is to maximize rewards by choosing correct actions in different situations [36]. The main components of RL are the agent and the environment. An agent observes, acts, and learns in an environment. The agent interacts with the environment by choosing actions. The actions may change the state of the environment and give the agent a positive or negative reward. Alternatively, the reward can also be zero.

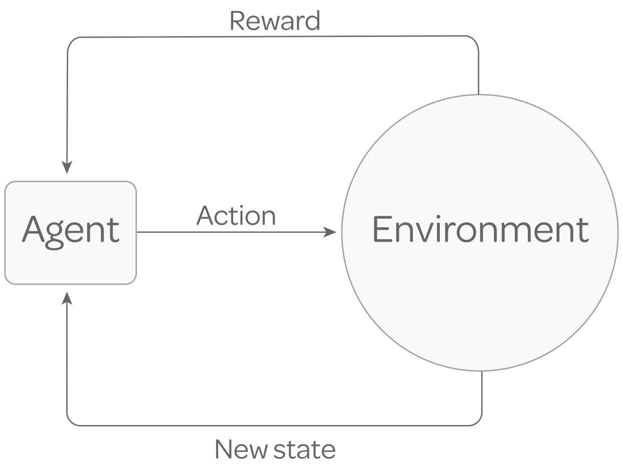


Figure 4: Reinforcement learning

In contrast to supervised learning, where the correct output is provided for each training sample, in reinforcement learning the reward can be delayed and may be the result of many actions taken earlier. In addition, reinforcement learning methods also collect the training data themselves by acting in the environment. Nevertheless, there are also similarities. Both methods learn a mapping from feature to label space; in reinforcement learning the labels are, for example, action values in the case of deep Q-learning or state values when learning a value function.

Markov Decision Processes (MDPs) are mathematical representations of sequential decision making problems [36]. Actions in MDPs have an effect on the subsequent states as well as immediate and future rewards.

MDPs are useful because they provide a mathematical representation that can be used to prove the convergence of reinforcement learning algorithms. Being able to represent an RL problem as an MDP means that the algorithms that can be used to solve MDPs are usable for the RL problem. Also algorithms and their convergence properties in MDPs can be compared.

The main components of MDPs are states, actions, dynamics, and reward functions. State defines the current situation. In UI exploration the state describes on what page the agent currently is and what the elements on that page are. Actions are the agent's way of interacting with the environment. Example actions are "click logout" and "enter username". Dynamics tell how the environment changes based on the current state and the action the agent has chosen and a reward function is used to incentivize or discourage the agent.

The agent receives reward r_t at time step t. The discounted return R_t, for time step t, using discount factor γ, is defined as follows:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2}$$

The goal of the agent is to maximize the return.

A policy π is a function that defines how an agent chooses actions. The policy takes in a state and returns the action. The goal of reinforcement learning is then to modify the policy so that the actions it outputs result in a large return.

The value function gives the expected return for a state, starting from state s and then following policy π:

$$v_\pi(s) = \mathbb{E}_\pi[R_t | S_t = s] \tag{3}$$

Intuitively, the value of a state s, $v_\pi(s)$, tells how good it is for the agent to be in that state.

A core problem in reinforcement learning is the trade-off between exploration and exploitation [36]. During learning, the agent needs to choose the actions that are known to be good in order to gain high reward. This is known as exploitation. However, since there could also be better actions to take, the agent also needs to try out new things. This is known as exploration. Finding the balance between the two is challenging. The difficulty of this problem is well described by Sutton et al. [36]: "The exploration–exploitation dilemma has been intensively studied by mathematicians for many decades, yet remains unresolved.".

Another problem is the credit assignment problem [36]. It means that it is difficult for the reinforcement learning algorithm to know which actions to blame when a bad reward is given or which actions resulted in a good reward. For example when controlling a UI with a reinforcement learning algorithm, it can initially be difficult for the agent to know which of the randomly-chosen "click" and "type text" actions in the end allowed the agent to login to the application.

Advanced deep reinforcement learning methods, such as A3C, have found ways to balance exploration and exploitation as well as deal with the credit assignment problem. This has allowed the use of Deep RL in new problems that were not solvable before.

2.3 Artificial Neural Networks

Neural networks can be used as function approximators. Furthermore, multi-layer networks can be trained to approximate almost any function [16]. In this thesis neural networks are used to learn functions that predict what is the best action to choose and what is the estimated value of being in a certain state (i.e. value function).

Artificial neural networks (ANNs) consist of layers of neurons (also known as units). Neurons are connected to other neurons in different layers. These connections carry activation signals to the next layer. The neurons are then activated based on the input from other neurons and the activation function of the neuron [5]. If a layer is fully connected, it means that all of the neurons of the layer in question are connected to all of the neurons in the previous layer.

Each connection has a weight that is adjusted in training to fulfill some objective. These weights are usually initialized randomly and then optimized with a variant of stochastic gradient descent [32]. When computing the activation value of a neuron, the values of the input neurons in the previous layer and the corresponding weights are multiplied. The weighted inputs are then summed together and finally the sum is passed through an activation function.

Neural networks can approximate more complex functions if they have many layers. However, adding layers with linear activation (or no activation function at all) does not make the network more complex. This is because a sequence of linear operations is equivalent to a single linear operation. Therefore we need nonlinear activation functions that are applied to the values of each neuron. One commonly-used nonlinear function is called rectified linear unit, ReLU for short. It is defined as follows:

$$ReLU(x) = \max(0, x) \tag{4}$$

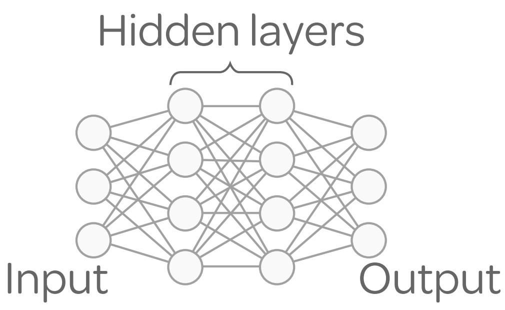


Figure 5: An example of an artificial neural network with two hidden layers, and input and output layers. All layers are fully connected.

2.3.1 Convolutional Neural Networks

In this thesis convolutional neural networks (CNNs) are used to extract features from a screenshot of the UI. These features define the current state of the user interface of the application that is being tested.

The name convolutional neural networks comes from the use of convolution operation [12]. In convolutional networks, kernels (also called filters) slide over the input image and produce an output value for each position. The kernels contain numeric values called weights. The output value of each position is calculated by multiplying the weights with the corresponding values in the image and summing those together, see Figure 6.

CNNs are invariant to certain input transformations [5]. For example, two images with the same button in different positions can produce the same output. The main differences of convolutional networks compared to fully-connected networks (cf. Figure 5) is that the neurons have local receptive fields, i.e. they are only connected to a region of neurons in the previous layer rather than all of them. In addition, the kernel weights are shared between locations. Subsampling is also used between layers to reduce the number of units. A smaller number of units allows faster computation.

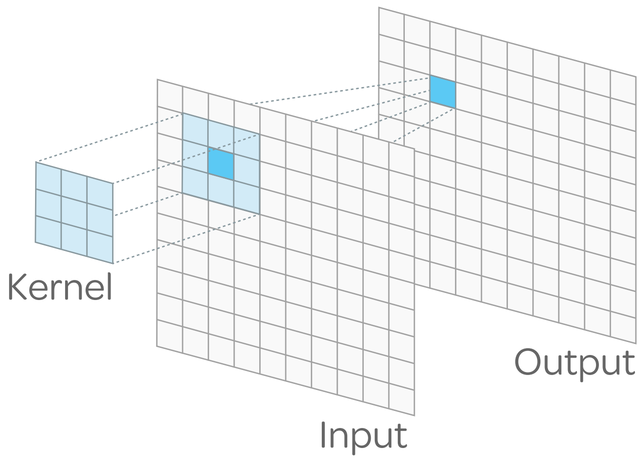


Figure 6: Convolution operation. Image adapted from [7].

Convolutional neural networks have been used successfully for image processing and other structured data tasks such as sound classification [30]. One of the first well-performing image classification neural networks, AlexNet [21], used CNNs. Since then, most of the top classifiers have been using convolutional neural networks [33].

2.3.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) can remember previous inputs and use the memory in decision making. RNNs are used, for example, in speech recognition and machine translation. In this thesis they are utilized to implement the memory of the reinforcement learning agent. With RNNs the agent can remember what states it has visited before instead of just knowing the current state.

The idea of RNNs is to redirect the output back to the network as input. By doing this, the networks can combine the previous output and the current input to produce the next output. This is useful, for instance, in text understanding, since the previous characters and words affect the meaning of the next characters and words.

Simply connecting the output of a fully-connected network back to itself is not enough, because it causes problems, such as vanishing or exploding gradients. Gradients are used to update the weights of the neurons in neural networks. Gradients can be used to find the direction that improves the optimization objective, e.g. minimizes a loss function. Optimization algorithms, such as stochastic gradient descent, take a step in that direction. Vanishing gradients cause the updates to fade and exploding gradients can cause the network to break due to the weights changing too much. As a solution for this, long short term memory (LSTM) cells enable gradients to stay effective over long sequences [12].

The main components of LSTM cells are cell state, input gate, forget gate, and output gate. The cell state flows through the cells and it can be used to remember information over many inputs. Each gate has its own task. The input gate decides what values are changed in the cell state. The forget gate allows the cell to forget information from its cell state. The output gate is used to define what will be the output of the cell. It can differ from the cell state.

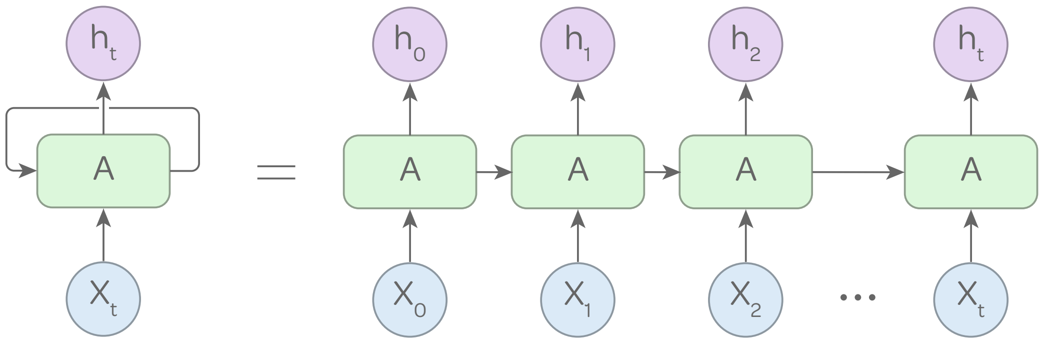


Figure 7: LSTM cells are recurrently connected. An example of recurrent connection is shown on the left. On the right the connection is unrolled. This is done for ease of computation. Adapted from [2] .

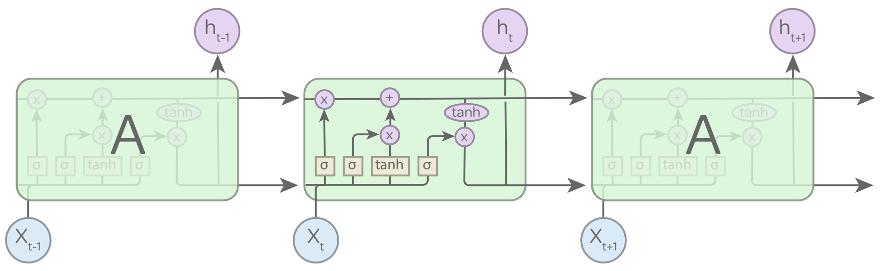


Figure 8: Three LSTM cells. Each cell consists of input, forget, and output gates. Adapted from [2] .

2.4 Deep Reinforcement Learning

Deep reinforcement learning combines reinforcement learning and deep neural networks, in other words, ANNs with multiple hidden layers. The memory and efficiency issues of tabular reinforcement learning methods are avoided with the use of neural networks.

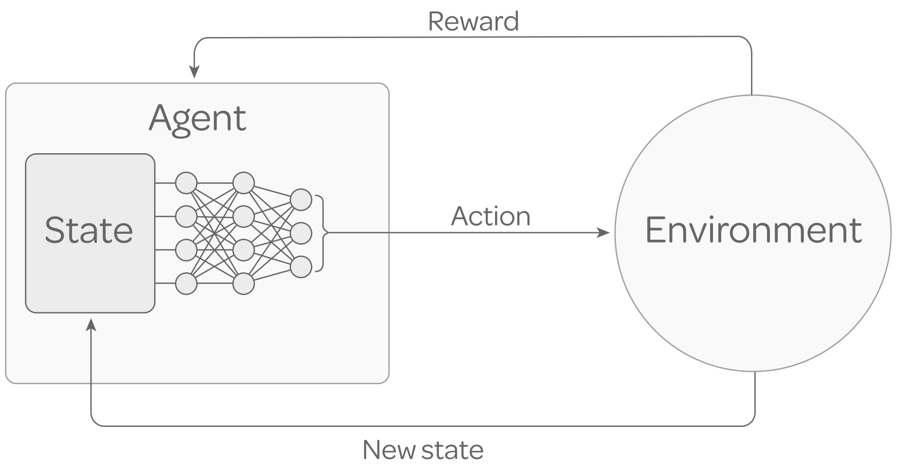


Figure 9: Main components of deep reinforcement learning algorithms. Adapted from [1]

Earlier, deep neural networks were thought to be unstable when used with reinforcement learning. However, Mnih et al. [27] managed to use the combination of Convolutional Neural Networks and Q-learning to train reinforcement learning agents in 2013. They introduced the first end-to-end solution for reinforcement learning using images as input. One key feature of their solution was the use of experience replay memory. Experience replay memory stores experience tuples (s_t, a_t, r_t, s_{t+1}). Using experience replay and random sampling they trained action value functions that could then be used to construct the optimal policy.

In tabular Q-learning (See Section 2.5.1) action-state values are stored in a table. In Deep Q-network (DQN) Q-values are approximated by using an artificial neural network. The use of an artificial neural network instead of a table allows the algorithm to work with larger state and action spaces. In addition, information between different states is shared and therefore the values can be learned faster and without visiting every state. For example, tabular Q-learning is unfeasible for the UI exploration problem studied in this thesis, but deep reinforcement learning methods are suitable.

2.4.1 Actor-Critic

In actor-critic methods the reinforcement learning agent consists of two parts, the actor and the critic. An actor represents the policy and a critic represents the value function. This means that the actor outputs probabilities for each action and the critic outputs the value for the state. As an allegory, it can be thought of as two people interacting with each other. The actor is a dancer choosing their moves and the critic will give feedback whether the dancer did well or not.

Actor-critic uses bootstrapping, i.e., it updates the value estimate of a state using the estimated values of the next states ([36] Section 13.5). This reduces the variance, but introduces some bias. Since variance is reduced, the actor-critic method often accelerates learning.

2.4.2 A3C

A3C, introduced by Mnih et al. [26], stands for Asynchronous Advantage Actor Critic. It extends the actor-critic method introduced in the last section by using an advantage function and multiple agents that update the policy asynchronously.

Advantage is defined as the difference between the return R_t and the baseline $b(s_t)$:

$$A(a_t, s_t) = R_t - b(s_t) \tag{5}$$

In A3C, the critic learns the value function $V(s)$, which is then used as the baseline. Thus the equation becomes:

$$A(a_t, s_t) = R_t - V(s_t) \tag{6}$$

The parameters of the critic are optimized based on the loss:

$$(R_t - V(s_t; \theta_c))^2 \tag{7}$$

This means that the critic will learn to predict the return R_t.

The actor in A3C is trained using the following gradient:

$$\nabla_{\theta'_a} \log \pi(a_t|s_t; \theta'_a)(R_t - V(s_t; \theta_c)) + \beta \nabla_{\theta'_a} H(\pi(s_t; \theta'_a)) \tag{8}$$

Where $\log \pi(a_t|s_t; \theta'_a)$ is the log probability for taking an action a_t in state s_t using policy π, $(R_t - V(s_t; \theta_c))$ is the advantage using $V(s_t; \theta_c))$ as the baseline, θ'_a is the set of parameters of the actor, θ_c is the set of parameters of the critic, $H(\pi(s_t; \theta'_a))$ is the entropy term, and β is a hyperparameter that determines the effect of the entropy term.

The entropy H is added to the objective function, because it helps A3C to explore better. Adding entropy means that the optimization algorithm tries to keep entropy high, and therefore the underlying probability distribution should not converge to a non-optimal deterministic policy.

A3C actor-learner thread pseudocode, adapted from [26], is shown in Algorithm 1. The algorithm has a global model and multiple actor-learners, each with their own local models. The individual actor-learners act independently in their own instances of the environment. At the beginning of each iteration the learners update their own local copy to match the global model. The learners learn how to maximize rewards by trying out different actions. After enough experience is collected, each learner updates the global model asynchronously and immediately at the end of each iteration. The global model parameters are updated using gradient descent with a chosen learning rate α. To find the optimal learning rate, Mnih et al. performed hyperparameter search.

Algorithm 1 A3C actor-learner thread

```
// We have global actor parameters $\theta_a$ and critic parameters $\theta_c$ and thread local versions $\theta'_a$ and $\theta'_c$
Initialize global shared counter $T \leftarrow 1$
Thread local step counter $t \leftarrow 1$
// $t_{max}$ is the maximum number of steps per update loop
// We have defined a learning rate $\alpha$
repeat
    Reset gradients: $d\theta_a \leftarrow 0$ and $d\theta_c \leftarrow 0$
    // Update thread local parameters to match global parameters:
    $\theta'_a \leftarrow \theta_a$ and $\theta'_c \leftarrow \theta_c$
    $t_{start} \leftarrow t$
    Observe state $s_t$
    repeat
        Perform action $a_t$ given by the policy $\pi(a_t|s_t;\theta'_c)$
        Observe state $s_{t+1}$ and receive reward $r_t$
        $t \leftarrow t+1$
        $T \leftarrow T+1$
    until terminal $s_t$ is reached or $t - t_{start} == t_{max}$
    $\text{R} = \begin{cases} 0, & \text{if } s_t \text{ is terminal} \\ V(s_t,\theta'_c), & \text{otherwise} \end{cases}$
    for $i \in (t-1, ..., t_{start})$ do
        $R \leftarrow r_i + \gamma R$
        // Accumulate gradients:
        $d\theta_a \leftarrow d\theta_a + \nabla_{\theta'_a} \log \pi(a_i, s_i; \theta'_a)(R - V(s_i;\theta'_c))$
        $d\theta_c \leftarrow d\theta_c + \partial(R - V(s_i;\theta'_c))^2/\partial\theta'_c$
    // Update global parameters asynchronously using a variant of gradient descent and learning rate $\alpha$:
    Update $\theta_a$ with $d\theta_a$
    Update $\theta_c$ with $d\theta_c$
until $T > T_{max}$
```

2.5 Existing UI navigation methods

In this section, we will discuss how Q-learning-based testing and WGE approach the task of controlling a GUI with a reinforcement learning algorithm.

2.5.1 Q-learning-based testing

Sebastian Bauersfeld and Tanja E. J. Vos have proposed a method for GUI robustness testing using tabular Q-learning [4] [3]. Q-learning is an off-policy temporal difference learning method [36]. Off-policy means that a different policy is used for exploration than the policy that is being optimized. In the case of Q-learning a greedy policy is used in the updates to choose the optimal next action by maximizing $Q(s, a)$, but the exploration policy may choose actions differently. Temporal difference means that the new estimates are based on samples and previous estimates. Tabular Q-learning uses a table to represent the state-action values, $Q(s, a)$.

Q-learning is used to calculate and update expected rewards for each state-action pair. The reward function is defined as shown below.

$$r_t(s, a) = \begin{cases} r_{\text{init}}, & \text{if } x_a = 0 \\ \frac{1}{x_a}, & \text{otherwise} \end{cases} \tag{9}$$

Where r_{init} is a large positive number and x_a is the number of times an action a has been taken in state s.

The algorithm learns state-action values or Q values, $Q(s, a)$. These are functions of the state and the action, and they output the expected return. The values are updated as follows:

$$Q(s_t, a_t) \leftarrow r_t + \gamma \cdot \max_a Q(s_{t+1}, a))$$

Their algorithm acts greedily, i.e., it chooses the action with the highest Q value. The idea is that the reward for each action in each state decreases when the action has been taken many times. Thus the algorithm should favor the actions that have been tried the least frequently.

In this thesis this tabular Q-learning method and random action selection were compared. More details of about the random action selection can be found in Section 4.2.1. It turned out that tabular Q-learning is notably better. The results can be seen in Figure 25. While the algorithm is substantially better than random action selection, it has some difficulties. To distinguish between states the algorithm creates a hash that is based on the elements in the GUI and their attributes. The performance therefore depends on what is chosen to be included in the state. Due to the use of hashes and tabular Q-learning, it cannot share information between states. Since information is not shared, the algorithm may try out more actions than necessary. For instance, for a human user it is easy to see that the menu bar works the same way on every page. The links that have the same text direct to the same page even if the page where the links are changes. However, since a new page has a new state, the tabular Q-learning algorithm will treat it as completely different page and will need to try clicking those links again.

In addition, it can be difficult to choose what information should be included when calculating the state hash. It makes sense to include what elements are on the page and what their attributes are, such as type (e.g. button or text box). However, including too much information makes the state change very easily and thus the algorithm may find many states while actually staying on a single page. An example of this is the choice of whether to include the text that has been entered or not. If the text is included as-is and the algorithm is able to enter any text, it can generate an infinite number of states with just a single text box. On the other hand, if entered text is not included, the algorithm cannot tell the difference between a form with all details filled versus an empty form.

Since the rewards are initialized with the constant r_{init} it is possible to have state-action pairs with the same expected reward. This makes the action selection unclear. What should be done if the expected values are the same? Should the algorithm choose the first one or choose randomly? In this thesis both ways were tested and choosing the first one was effective, because, for example, on the login page, username, password, and the login button happened to be in the right order. However, if they were not in the right order, the algorithm might have never been able to login. Thus choosing randomly is a safer choice. In this thesis the tabular Q-learning method uses random selection for solving ties.

This thesis tries to solve these difficulties by using a state representation that allows shared knowledge and a Deep Reinforcement Learning algorithm that is able to make use of that information. Additionally, this thesis experiments with an image-based state representation, which is essentially a screenshot of the application's current state. It does not have the same difficulty in choosing what information to include. Also, the underlying Deep Reinforcement Learning algorithm handles action selection by using a neural network that outputs a probability value for each action and the action is then sampled from that probability distribution. Therefore, the Deep RL algorithm can learn what actions are worth trying and keep their probabilities high while minimizing others. In this way, it can concentrate exploration on the most promising actions.

2.5.2 Workflow-Guided Exploration

Learning with random exploration can be slow when the task we want to learn is complicated. The goal of Workflow-Guided Exploration (WGE) [22], introduced by Liu et al., is to use a small number of human demonstrations to make the agent learn faster. The human demonstrations are used to constrain the exploration of the agent rather than directly training the agent to do what was shown.

Exploration is constrained by utilizing so-called workflows. Workflows are higher level instructions and they are induced from the human demonstrations. For example in an email forwarding task, the demonstration shows which specific email was clicked, but the workflow does not contain that information, rather it contains the information that these steps were required to forward an email. WGE works very well in various web tasks, such as clicking certain checkboxes and replying to emails, reaching 100% success rate in many tasks.

Web pages are often represented with a document object model (DOM) [37]. DOM tells what elements (buttons, links, images, etc.) are on a page and how they are related. To embed the DOM elements of a web page, Liu et al. use a model called DOMNet. DOMNet is a neural network that encodes the spatial and hierarchical structure of the web page. Elements have a base embedding, which contains the attributes of the element such as tag name, class names and text. Spatial information is taken into account by summing up the base embeddings of all elements that are closer than 30 pixels away from the element. Hierarchical information is added by including the base embeddings that are fed to a learnable function. The highest values of the processed base embeddings are included at each depth in the DOM tree. The authors used depths 3, 4, 5, and 6.

Since WGE is intended to improve personal assistants to do normal web tasks, such as sending email, it may not be usable for UI testing as-is. However, it could be used to detect problems by training agents to perform normal tasks in the SUT and check if something went wrong while executing the task. Thus the method could work for verification of intended functionality, but in the context of this thesis explorative testing is more interesting.

Because this thesis tests a complex web-based application, acquiring even a small number of human demonstrations for each feature of the application can be time consuming. Since this thesis is focused on methods that do not use training data, WGE was not used. There is still a possibility that the proposed methods would perform even better by utilizing techniques similar to WGE.

3 Proposed methods

In order to apply deep reinforcement learning for UI testing, this thesis will address the following questions:

- How to represent the states and actions?
- How to make the agent explore the user interface on its own?
- How to prevent the agent from getting stuck?

The problem of UI testing can be modelled as a Markov Decision Process in the following way. The state s_t represents the current state of the UI. It contains the information about the contents of the current page. The state can be encoded in multiple ways and the two ways presented in this thesis are detailed in Section 3.2 and Section 3.3. Additionally, the MDP is partially observable, because the whole state of the SUT is not visible from a single web page. For example, the contents of the backend database and the pending API requests cannot be seen from the screenshot or document object model (DOM).

For each state there is a finite set of available actions $A(s_t)$. The actions contain all different ways to interact with the current page. Interactions can be clicking a button or a link, or typing text into a text box. The two approaches, image-based and element-based, described in the following sections use different action representations.

The reward function $R(s, a, s')$ defines what behavior is wanted from the reinforcement learning agent. This thesis experiments with two reward functions: a URL-based reward and a DOM state reward. These functions give reward for finding unique URLs (web addresses) or states.

A discount factor γ is used to discount rewards when calculating the return. Discounting is used to make intermediate rewards matter more, but also take into account future rewards. This thesis uses a discount factor 0.99. If the factor were 0, only intermediate rewards would matter and if it were 1, all rewards would be equal regardless of the time the reward was obtained.

The goal of the deep reinforcement learning algorithm is to maximize the expected discounted return R_t (See Equation 2). By maximizing the expected return, the deep RL algorithm maximizes the number of unique URLs or unique states, depending on the reward function used.

One might expect that the transitions in a UI are deterministic, since clicking a link should navigate to the same page with 100% probability. However, since the MDP is partially observable, a state can appear to be the same for the agent, but an action may have a different result. Depending on the state of the backend database, clicking a button can navigate to a different page. Because of this the transitions have to be modelled as probabilistic.

Overview of the proposed methods:

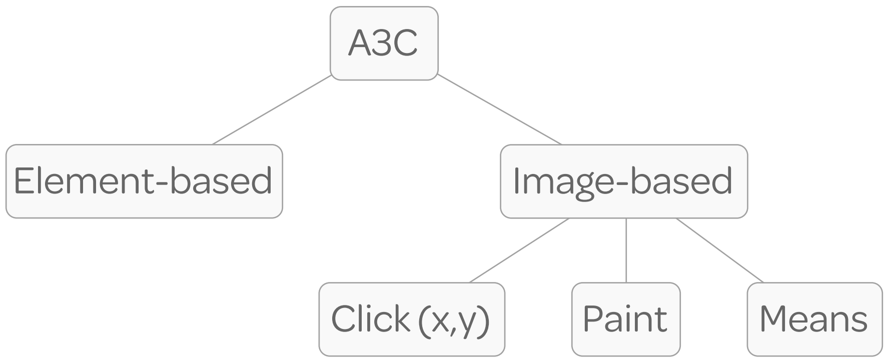


Figure 10: Proposed methods and their relations.

While deciding how to encode the state of the UI, two different approaches seemed the most reasonable. These approaches are called element-based and image-based. The element-based approach uses the DOM element information of the web page to construct a vector representing what elements are on the web page, and what their attributes, such as type and id, are. The image-based approach takes a screenshot of the rendered web page and encodes it as a three-dimensional tensor (multidimensional array).

The following sections will first cover the base deep reinforcement learning algorithm, A3C implementation, upon which both proposed methods are based. Then the element-based and image-based approaches for the state and action representations will be discussed. In addition, the different ways to utilize the action output of the image-based approach are described.

3.1 Base deep reinforcement learning algorithm

The base deep RL algorithm is based on David Griffis' implementation [13]. An already-tested implementation was chosen as the base, because deep reinforcement learning is difficult to get working and implementations require a lot of tuning to achieve the desired results [17].The contribution of this thesis is in experimenting with different input and output methods for the learning algorithm as well as designing a reward function and setting up an environment for controlling a browser with a reinforcement learning agent. The implementation uses A3C and utilizes GPUs. The neural network consists of convolution layers, LSTM cells, and fully-connected layers. In this thesis the model was split into three parts: feature extractor, policy, and action output modules.

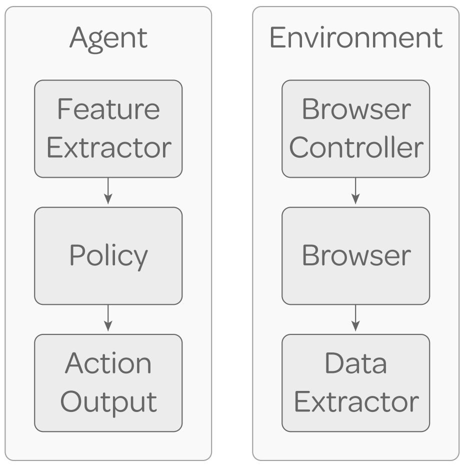


Figure 11: The main components of the solution.

In the case of the image-based approach, the feature extractor is a CNN (c.f. Section 2.3.1) and the element-based approach uses an ANN with all layers fully connected. The policy is the same for both models, an RNN with an LSTM cell, where the input, the output, and the hidden state are vectors. The action output module takes the output of the policy and applies some operations on it to produce the final actions. Actor and critic share all layers except for the last fully-connected layer.

Compared to video games, the UI exploration task adds three additional challenges: variable state size, large and variable action space, and slower execution speed. The number of elements differs between pages. For example, a login page may contain just three elements: username, password text boxes, and a login button, but a settings page may contain dozens of elements.

The execution slowness comes from the loading of the web pages and waiting for responses from the backend. Some of the actions are instantaneous, such as typing the password, but others can take a variable time to finish. The pages may have animations that smoothly transform from one view to another. While the animations are pleasing for the end user, they pose a challenge for the reinforcement learning agent. Some actions require data from the backend, which is fetched over the REST API. The time it takes to process the request depends on the server load and the complexity of the request.

We can define the task of finding errors using sets. After choosing how we define a UI state, we have a state space X which represents the space of all theoretically possible states. The real states $x_p \in X_p \subset X$ are the states that actually exist. Then we have states that contain errors $x_e \in X_e \subset X_p$. Our goal is to find the error states. However, we do not know beforehand which states cause errors and we do not know how many states there are in total. Thus by exploring and finding as many states x_p as possible we maximize our chances of finding the states x_e.

This thesis uses the Algorithm 1 to train the element-based and image-based methods. The training hyperparameters are the following. The algorithm uses the Adam optimizer [20] with shared statistics and a learning rate $\alpha = 0.0001$. The

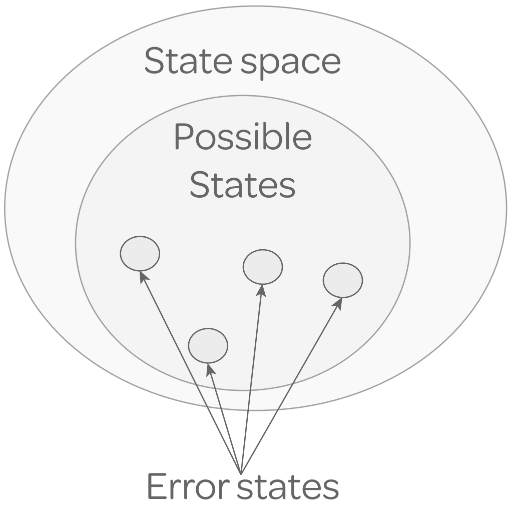


Figure 12: The state sets and their relations visualized.

discount factor is chosen to be 0.99. General advantage estimation is used with $\tau = 1$.

3.2 Element-based approach

The idea of the element-based approach is to find all interesting elements of the web page and encode them so that they can be fed to a neural network. The difficulty for the implementation arises from the fact that the number of elements varies between pages. Neural networks usually have a constant-size input, thus fitting a variable number of elements to a constant-size vector is challenging. One solution is to create vectors that are long enough so that the number of elements is never larger than what would fit in the vector. Then if the number of elements is less than the maximum, zeros will be added so that the size of the vector remains the same.

Some of the important information for the HTML elements is stored in their attributes. In this thesis the attributes that were extracted from the elements were the type of the element, its id, and the text it contains. The type of the element tells whether it is a link or a button or a text input box. The id of the element gives information about its use. For example the username and the password fields have different ids, so the algorithm can detect from the encoded representation which is which. The contained text is used to tell the algorithm when text has been typed into input boxes. Therefore the algorithm can, for instance, understand to click the login button after typing the username and the password.

To encode the elements into the real valued vector, elements need to be represented as numbers. One common way of coding classes to vectors is using one-hot encoding. One-hot encoding means that a vector of size x is created and initialized with zeros, where x is the number of classes. Then the classes are ordered so that each class has its own position in the vector. The final encoding is created by setting the element of the vector to one that corresponds to the class.

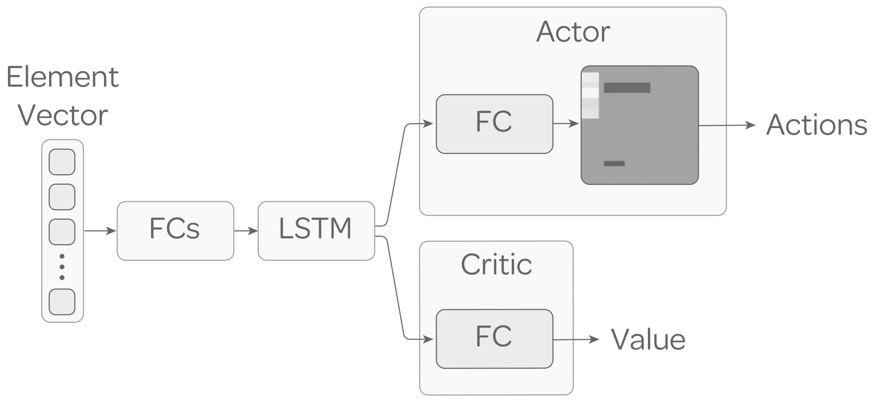


Figure 13: Element-based A3C.

To construct a fixed size vector representing the state of a web page assume that the page can have at most n elements (such as buttons, text boxes and links). For each element the state vector contains a one-hot encoded element type, a one-hot encoded identifier, and a number that is one or zero depending on whether the element contains text or not. Assume that we have at most t different element types and at most i different element identifiers. Now we can fit all possible web pages under these assumptions to a state vector with a size of $n \times (t + i + 1)$.

We can model the problem as a Markov Decision Process as follows.

- The state space S is $\{0, 1\}^{n\times(t+i+1)}$, which is the space of binary vectors of size $n \times (t + i + 1)$.
- The action space A is a set of actions where the set size is n. So there is one action for each element on the web page.
- The discount factor $\gamma = 0.99$.
- The reward function R is:

$$R(s, a) = \begin{cases} 1, & \text{if A new URL or state was found} \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

Next we will shown an example of the element representation vector when there are two element types: buttons and text boxes and element identifiers 0 and 1.

A button with id 0 has the following information in the state vector:

$$\begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{11}$$

The first two values encode the element type, the 3rd and 4th values encode the element identifier and the last value tells if the element contains text.

A text box with id 1 has the following information in the state vector:

$$\begin{bmatrix}0\\1\\0\\1\\0\end{bmatrix} \tag{12}$$

Because the last value in the vector is zero it means that the text box is empty. If the agent takes the action 1, which means enter text to the text box, then the state of the text box will be updated to:

$$\begin{bmatrix}0\\1\\0\\1\\1\end{bmatrix} \tag{13}$$

This example showed the element representation vector for each element separately. However, in the actual implementation, the element vectors are concatenated to one long vector.

3.2.1 Problems of the Element-based approach

The element-based approach has the following problems:

- It is unclear how the elements should be ordered. The trained neural network model may learn that certain elements are in specific positions in the input vector. However, if the order suddenly changes, for example if a new element is added as the first element in the vector due to some change in the UI, the neural network may start to output wrong actions.
- Elements that are invisible can be found unless a sophisticated element processor is implemented. There can be many reasons why an element that exists in the DOM is not actually visible. The element can be behind another element, the hidden attribute can be true, or the style of the element can make it transparent.
- The element representation contains no spatial information. Spatial information can be very important. For example related elements are usually close to each other, large elements are the most commonly used, and small, infrequently used elements are on the edges of the page.
- We need to choose the maximum number of elements, element types, and ids. If too small a number is chosen for any of the three, the representation vector cannot be formed and learning fails. If the number is too large, training becomes slow.

3.2.2 Advantages of the Element-based approach

The advantages of the element-based approach are:

- Compared to the image-based solution there are no problems in finding elements. The visual size of the elements does not matter, thus even tiny buttons can be found.
- All elements are equally likely to be clicked.
- Action space is limited to the number of elements.
- The input vector is smaller than the image that is fed to the image-based approach. Thus it is faster to process.

3.3 Image-based approach

The image-based approach uses the same learning algorithm, A3C, as the element-based approach. Instead of taking an element representation vector as the input, the image-based approach uses the screenshot of the current web page. The screenshot is a color (RGB) image with a resolution of 128 by 128 pixels. Thus the input is $128 \times 128 \times 3 = 49152$ numbers.

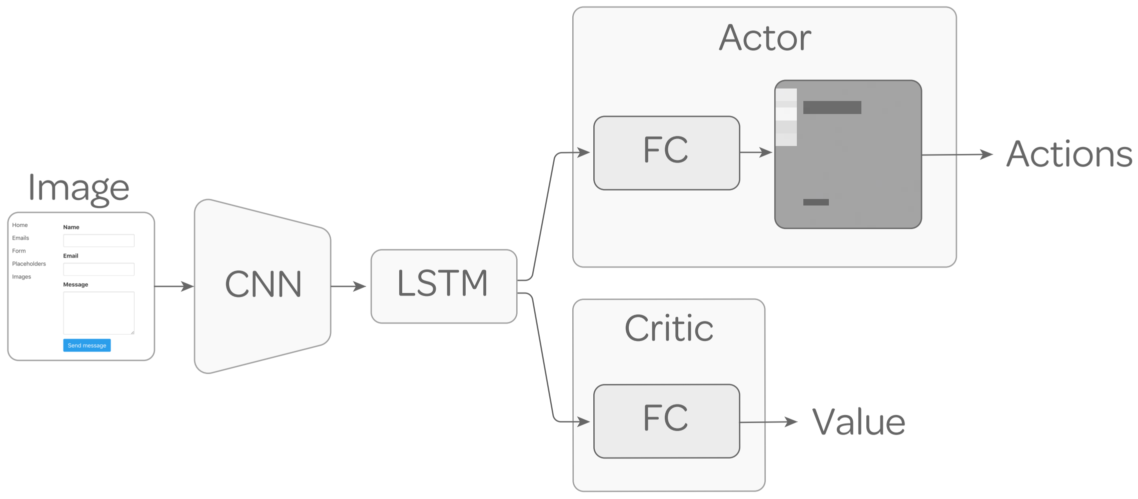


Figure 14: Image-based A3C.

When getting the state of the UI, a screenshot of the current web page is taken. The screenshot is resized to a lower resolution so that it is faster to process with a neural network. The resized image is then fed to a Convolutional Neural Network (the feature extractor). CNN extracts features out of the image and these features can be then used to make the decision what to click.

The main advantages of the image-based approach compared to the element-based approach are that the screenshot should contain all useful information about the current state of the UI. After all, that is the same as what a human user would see when using the web application. In addition to that, no other information should be required. The element-based approach has a variable number of elements, but an image maintains the same size. If the size of the browser window changes, the screenshot image can be resized. Additionally, if the output is presented as an image with each pixel telling the probability of clicking, the output is also constant size.

The image-based approach output is also easy to visualize. It can be depicted as a grayscale image where there is a pixel for each possible click location. The brightness of the pixel tells how high the probability is for clicking that pixel, white means high probability, black low probability. An example output visualization can be seen in Figure 14.

One of the difficulties of the image-based approach is that the feature extractor needs to be more complex than with the element-based approach, because the input is a raw screenshot image. Also, when outputting the click probability of each pixel, the number of actions is very large. Thus it is slow to go through all the actions in all states and find out which is the best.

The following sections will focus on solving the problem of having too many possible actions by discussing different ways to output actions. The cursor can be moved in two different ways. The neural network may output how many pixels to move in each direction. This is discussed in the Section 3.3.1 Relative Movement. Another way is to directly tell where to click (Section 3.3.2 Absolute Movement). The click location can be modelled as a probability distribution or the model can predict the probability of clicking each pixel. Since there are a lot of pixels, there are a lot of actions to try when using the latter method. A large number of actions means that exploring and thus also learning can be slow. The sections 3.3.3, 3.3.4, 3.3.5 will discuss how this problem can be addressed.

3.3.1 Relative Movement

One of the ways to solve the problem of having too many possible actions was to have a set of actions that move the cursor in different directions by n pixels. In fact, this simulates human users better than absolute clicking. There are two possibilities for this, discrete and continuous action spaces. In the discrete case, we can choose a set of actions such as move left, move right, move up, move down, and click. Then the policy network has to output five values, one for each action (we assume that there is no use for a "do nothing"-action). In the continuous case the network can output one value for movement delta along the x axis, one value for movement delta along the y axis, and a probability for clicking.

This method reduces the size of the action space, but on the other hand, doing the same tasks may require that the agent chooses the action more times. With the click probability map, a single action is enough to click a button anywhere on the screen. Relative movement requires the actions it takes to move to the button and an additional action to click the button.

Deep reinforcement learning has been found to be problematic when the reward function is sparse [31]. That is, when the reward is mostly zero and only rarely positive or negative. This case is no exception, and since the relative movement makes the rewards even more sparse, learning becomes too difficult.

The efficacy of this method was tested in a simple website where there was a single page with one button that was randomly initialized anywhere on the page. A reward of one is given when the agent clicks the button and otherwise the reward is zero. When the button is clicked the episode ends and the page is reset, causing the button to move to a new randomly-chosen location. In these tests, the algorithm is able to learn to locate the button and click, but the training takes a long time and the agent spends most of the time clicking the background. If a negative reward is given for clicking the background, the agent learns quickly to not click at all and never succeeds in the task. Even though this method is more realistic, it had to be abandoned, because the training was too slow for a single button task, making it unfeasible for larger websites.

3.3.2 Absolute Movement

Absolute movement means that the algorithm directly outputs the click location. In addition to the click probability map, a method that outputs the parameters for two normal distributions was tested as a way to learn what to click. By outputting two mean values and two standard deviation values, it was possible to create one normal distribution for the vertical axis and one for the horizontal axis. These two marginal distributions can be then combined to form a probability distribution for the click location as shown in Figure 15.

However, it turned out in the tests that the two normal distributions output did not work well. The author believes that this is due to the fact that when there are two or more buttons far away from each other and they have a similar action value, a single normal distribution does not work. This is because the mean of the distributions would be visually halfway between the two buttons and therefore the standard deviation needs to be high so that the buttons have a high probability of being clicked. As a side product, the probability of clicking increases all around the web page.

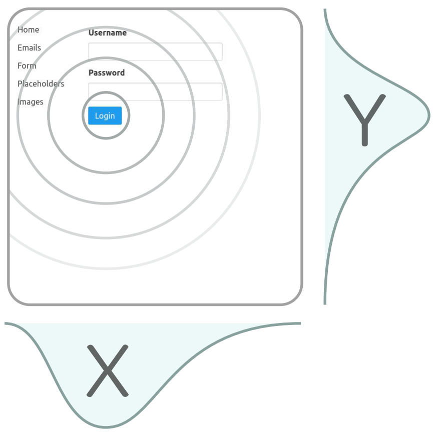


Figure 15: Two normal distributions output. The final click position distribution is formed from the marginal distributions for the horizontal (x) and the vertical (y) axes.

Even though the action space is large, with some adjustments to how the action is chosen based on the click probability map, the algorithms were found to learn well. The following three sections will discuss how the probability map is calculated and how it can be processed so that the learning is faster.

3.3.3 Click probability image

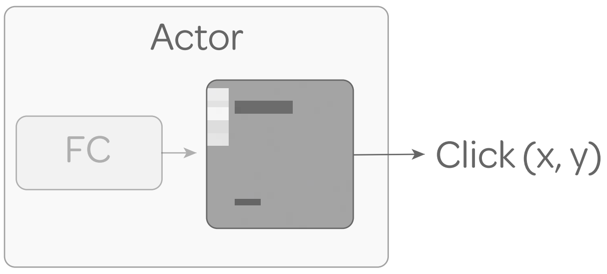


Figure 16: Image-based method with click position (x,y) as output.

The click probability image method outputs the probability of clicking for each pixel. The policy network outputs an image of size pw x ph. The size of the output can be different from the input image size. The smaller the size, the fewer actions there are to try, but also the size has to be large enough to have at least one pixel for each element on the page. For example if the input image is 128x128, and in that image there is a button that has a width of 4, the output should be at least 32x32 to make it possible to click that button.

3.3.4 Paint in the area of the clicked element

Painting the area of the clicked element means using the visual boundaries of the clicked element to simulate clicks to the other parts of the element. Since in the click probability image method the click locations need to be tried one by one even if the results are the same, this method improves the sample efficiency by telling the deep RL algorithm that all these other actions would work the same way.

The idea for this came from the fact that clicking an element will have the same effect regardless of which part of the element was clicked. It does not matter if the top or bottom part of the login button is clicked, both will trigger the same action: logging in the user.

3.3.5 Means of elements in the prediction image

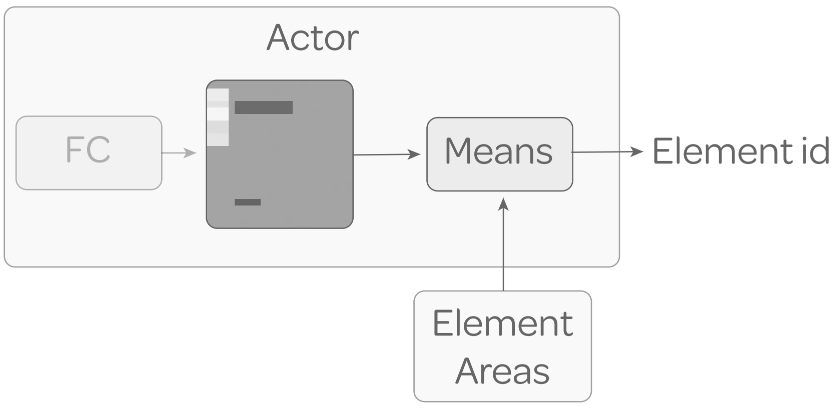


Figure 17: Image-based with element id output that is calculated by taking the mean over the element area.

Means of elements in the prediction image takes the predicted click locations from the actor and the areas of each element and calculates the mean click probability for each element.

This method has the advantage compared to click probability and the painting that the actions can be restricted to just the elements and there is only one action for each element. Entropy is used to encourage trying out actions other than the one that worked the best [39]. Entropy is subtracted from the loss and since small entropy means there are a small number of actions that are often selected, minimizing the loss causes entropy to increase and therefore the actor will explore other actions.

This is a problem with the click probability and paint methods, because increasing the entropy means trying out not only the actions that click on other elements but also the actions that land in the area of the same element and thus "exploring" the same action.

Since the entropy in the means method is calculated after computing the element area means, only the probabilities for the click locations that fall outside the clicked element will be increased.

Algorithm 2 Element-based algorithm

```
// For every episode
repeat
    $t_{start} \leftarrow t$
    // For each time step in an episode
    repeat
        Initialize an empty element representation vector $s_t$
        Get all elements $E$ in the current view
        for each element $e \in E$ do
            Get the type, id, and contained text of the element $e$
            One-hot encode the type and id
            Create a boolean telling whether the element contains text or not
            Combine the encoded attributes into the element representation vector
        Feed $s_t$ to the A3C model and get per element probabilities
        Choose an element from the per element probability distribution
        Interact with the chosen element
    until $t - t_{start} == t_{max}$
until stopping criteria is met
```

Algorithm 3 Image-based means algorithm

```
// For every episode
repeat
    $t_{start} \leftarrow t$
    // For each time step in an episode
    repeat
        $s_t \leftarrow$ screenshot of the current view
        $b_t \leftarrow$ visual boundaries of the DOM elements in the current view
        Feed $s_t$ to the A3C model and get per pixel probabilities
        Combine per pixel probabilities and $b_t$ to compute per element probabilities
        Choose an element from the per element probability distribution
        Interact with the chosen element
    until $t - t_{start} == t_{max}$
until stopping criteria is met
```

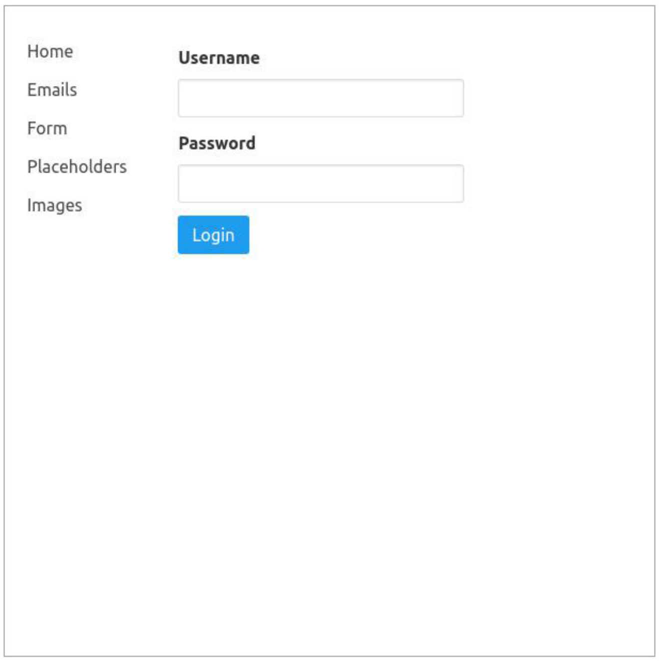


Figure 18: Sample login page

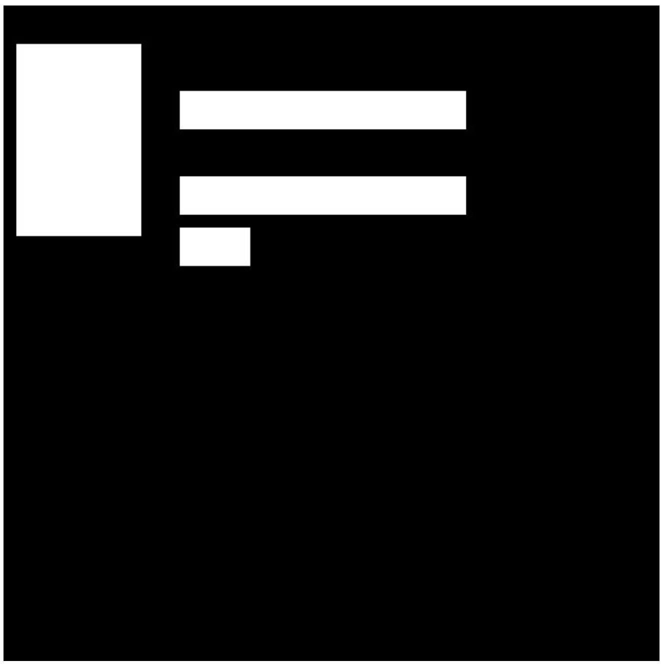


Figure 19: The areas of the elements on the sample login page

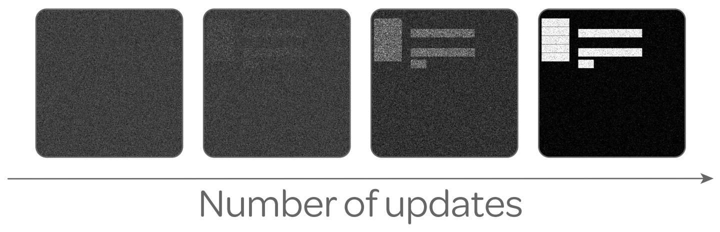


Figure 20: Click probability method update visualization.

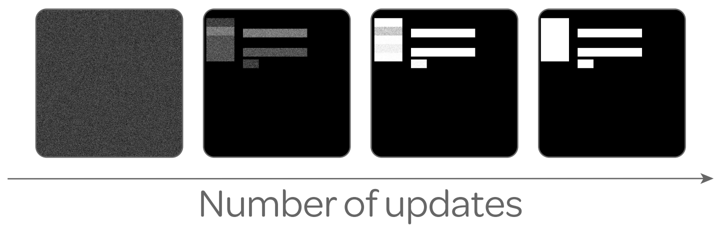


Figure 21: Paint method update visualization.

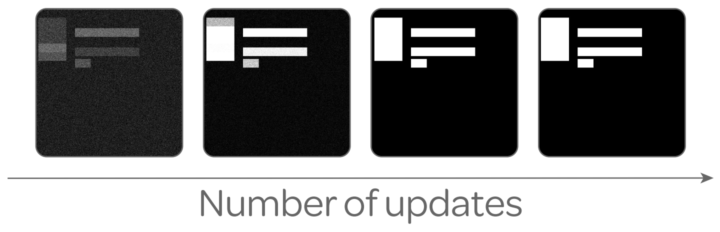


Figure 22: Means method update visualization.

4 Numerical results

This section describes the executed experiments and their results. The experiments were carried out to find out how well the proposed methods work. This section starts by presenting the evaluation criteria, after which the baseline methods are detailed. Then the test environments are described. The remaining subsections are about the different experiments and their results.

4.1 Evaluation criteria

The reward functions that are discussed in this thesis are:

- Reward for finding bugs
- Reward for unique URLs
- Reward for unique states

There are no standardized metrics for measuring the performance of each algorithm in UI exploration. In this thesis these metrics are: number of unique URLs and number of unique states (web pages) found during the exploration session. Reward for found problems was also considered, but not used in the experiments, because it was expected to be too sparse. The number of unique URLs was chosen as a measure of how diverse the exploration is: the more different URLs are found, the more web pages are explored. The number of unique states was chosen to measure how well the algorithm tests the functionality of the UI. This is thought to be closely related to discovering problems. The assumption is that the more different things are tried the more likely problems are found.

So why should we give reward for finding unique states or URLs? Why not use number of found problems as reward? Initially it was considered to give the agent reward for every problem it can find. However, it turns out that this may not be what is wanted because of the following reasons. The amount of reward given depends significantly on the system that is being tested. Since deep reinforcement learning methods are difficult to train when rewards are sparse, a system that does not have many problems would make the learning process hard. The deep RL methods, such as the ones used in this thesis, use random exploration to find out what states give rewards. Thus the deep RL agent would be somewhat equivalent to random search until it starts to find bugs.

Also, finding the same bug again is not useful and detecting whether a bug is new or not is challenging. Is it the same bug if the error message is the same? What if the state and actions taken were entirely different, is it still the same bug?

Additionally, giving reward for exploring as many states or URLs as possible makes the agent try to cover as much of the functionality of the system as possible. It may be more helpful to know that the agent covered this many states and URLs, and that these bugs were found or that there were no bugs.

4.2 Baseline algorithms

This section will discuss the already existing algorithms that the proposed methods will be compared against. Since there are no common benchmarks for UI testing, two baseline algorithms were implemented to provide a comparison to the proposed methods. Using the baseline algorithms, random agent, and tabular Q-learning it can be seen whether the proposed methods are better or not. First, the random agent algorithm will be described and then the tabular Q-learning algorithm, that is based on the algorithm introduced in Section 2.5.1, will be outlined.

4.2.1 Random Agent

One way to explore without a separate learning phase is to simply choose a random action out of all available actions. Theoretically, this method should be able to explore the web application entirely if it is run long enough. Since the actions are chosen randomly, all possible action sequences should be generated when the algorithm is run enough times. However the probability of specific action sequences is small. For example, if the UI has 20 links on each page and reaching a new page requires clicking the correct link on five consecutive pages, the probability of reaching the page is $\frac{1}{20^5} = 3.125e - 7$. Thus the algorithm may not be practical, but will be included in the experiments for comparison.

Algorithm 4 Random agent

```
repeat
    Get all elements in the current view
    Choose one of the elements uniformly at random
    Interact with the chosen element
until stopping criteria met
```

4.2.2 Baseline Q-learning algorithm

As a baseline for UI exploration, a tabular Q-learning algorithm was implemented based on the solution proposed by Sebastian Bauersfeld and Tanja E. J. Vos [4]. More information about the solution can be found in Section 2.5.1.

In the original paper, the algorithm creates a string from the fetched elements and their attributes, but in this thesis a constant-size vector is used. The reason why a constant-size vector is used instead is to make it easy to feed the same input to neural networks later on. To get a fixed-size vector with a variable number of elements, the algorithm requires a few parameters that set the maximum number of items. The vector has to be large enough to fit all elements up to the max number and leave all unused parts as zeros.

The algorithm works as follows. First all button, input, and link (tag name "a") type html elements are fetched from the current page. For each element the algorithm finds their attributes: id, value, type. The element type and id are one-hot encoded. Value is either 0 or 1 depending on whether the element value is a string longer than 0. Finally the algorithm combines all element information to a constant-size vector based on the hyperparameters. The size of the vector is $(\text{max\_elems} \times (\text{max\_ids} + \text{element\_types} + 1))$. The vector is then hashed with $sha256$, because the algorithm only needs to detect unique states, not how similar the states are. A hash lets us store the state without using a lot of memory and detecting if a state has been visited is fast to compute. Then, based on the stored states, state-action values, and action counts, the algorithm chooses the action that is expected to lead to a part of the UI that has not been tested before.

How does this method compare to the deep reinforcement learning based solutions? This method does not need a separate learning phase and is therefore faster to take into use. But, this tabular Q-learning algorithm can only recognize if states are different, but not the differences or how similar the states are. Deep RL algorithms use neural networks as function approximators. This eliminates the need for storing states and enables the agent to make use of the similarities of the states.

The pseudo-code for the algorithm can be written as follows (adapted from [4]).

Algorithm 5 Q-Learning baseline

```
$Q(s,a) = r_{init}\ \forall s \in S, a \in A$
**repeat**
    obtain current state $s$ and available actions $A_s$
    $a^* \leftarrow argmax_a Q(s,a), a \in A_s$
    execute action $a^*$
    obtain next state $s'$ and available actions $A_{s'}$
    $Q(s,a^*) \leftarrow R(a^*,s,s') + \gamma \max_{a \in A_{s'}} Q(s',a)$
**until** stopping criteria met
```

The r_{init} is chosen to be 10000.

4.3 Test environments

The test environments are browser-based applications that are controlled programmatically. Numerous test environments were created to rapidly test the performance of the different algorithms.

The browser is controlled by using Selenium [15]. Selenium is an open-source project for testing browser-based applications. Selenium allows the testing program to find what elements are on the web page and what their attributes are. It can also be used to interact with the page. This is done by calling simple commands, for example "click element" or "type text".

Since training the agent on the real product usually took a few days, multiple environments were used to find out early if an algorithm or setup works or not. Also in the beginning, it was not known whether or not the agent is able to navigate the real product, so it was first tested on smaller applications.

For example, to test how feasible relative movement is, a site with only a single button was created and the algorithm was given reward every time the button was clicked. The three most important environments are described in the next three sections.

4.3.1 Simple site

This environment is a traditional website with login. Pages are loaded fully when a button is pressed. There is a home page that has links to three pages. Each of those three pages has a link back to the home page. The site also has a login page. The agents need to learn to input the username and the password and press the login button in the right order to gain access to the website. The website also includes buttons for generating javascript and server errors to test the error detection and the algorithm's ability to find the states and actions that cause them. The goal is to visit all pages in as few clicks as possible.

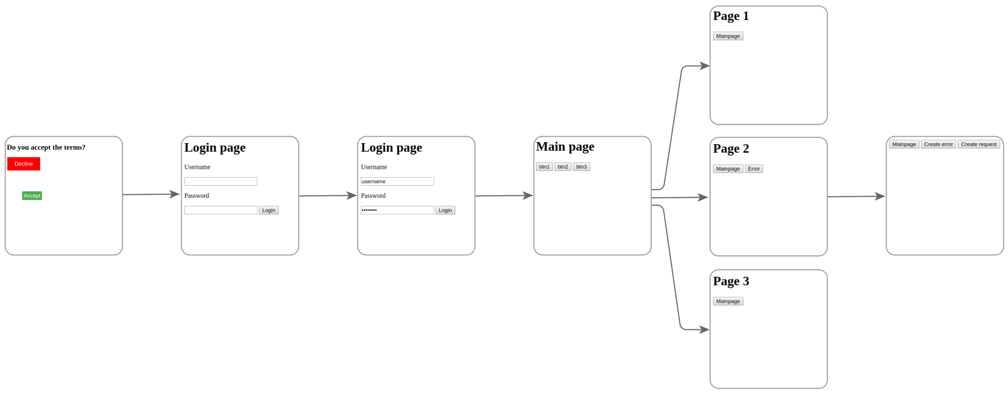
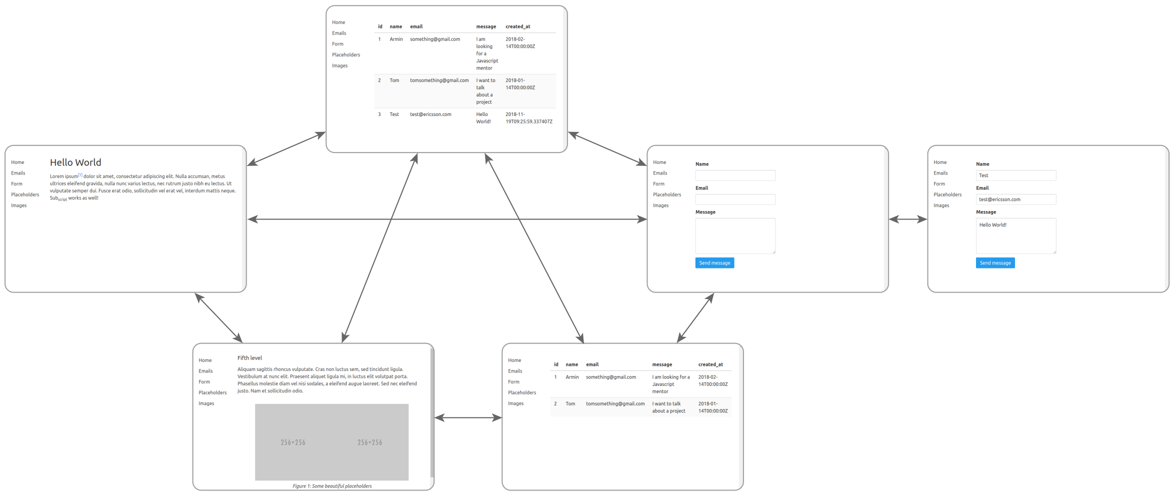


Figure 23: A visualization of the pages of the simple site environment.

4.3.2 Single-page application

This test environment is used to test that the algorithms and the system for controlling the browser works well with single-page applications. Single-page applications (SPAs) differ from traditional websites in the sense that the pages are not loaded fully; only the content of the page is loaded. The use of SPA usually means that the web application responses faster, but it is harder to detect whether the page is ready or not. Selenium waits for the page to load in the case of traditional websites, but it does not by default wait for the asynchronous API requests that the single-page applications send to finish.

This environment has a few pages that can be changed using the menu bar. The menu bar is visible on all pages, thus the algorithm needs to learn to recognize which pages it has already visited to be able to choose a link from the menu bar that leads to a new page.



id	name	email	message	created_at
1	Armin	something@gmail.com	I am looking for a Javascript mentor	2018-02-14T00:00:00Z
2	Tom	tomsomething@gmail.com	I want to talk about a project	2018-01-14T00:00:00Z
3	Test	test@ericsson.com	Hello World!	2018-11-19T09:25:59.337407Z

Hello World

Lorem ipsum[1] dolor sit amet, consectetur adipiscing elit. Nulla accumsan, metus ultrices eleifend gravida, nulla nunc varius lectus, nec rutrum justo nibh eu lectus. Ut vulputate semper dui. Fusce erat odio, sollicitudin vel erat vel, interdum mattis neque. Sub$_{\text{script}}$ works as well!

Name: Test
Email: test@ericsson.com
Message: Hello World!

Fifth level

Aliquam sagittis rhoncus vulputate. Cras non luctus sem, sed tincidunt ligula. Vestibulum at nunc elit. Praesent aliquet ligula mi, in luctus elit volutpat porta. Phasellus molestie diam vel nisi sodales, a eleifend augue laoreet. Sed nec eleifend justo. Nam et sollicitudin odio.

Figure 1: Some beautiful placeholders

id	name	email	message	created_at
1	Armin	something@gmail.com	I am looking for a Javascript mentor	2018-02-14T00:00:00Z
2	Tom	tomsomething@gmail.com	I want to talk about a project	2018-01-14T00:00:00Z

Figure 24: Visualization of the single-page application.

4.3.3 Real product

Real product is the actual system that we want to test. In this thesis it is a web-based application with a single-page application frontend and it has a backend for processing API requests. The product also has a login page, but in the experiments it is scripted so that the algorithms do not need to learn to login. This was done to prevent the algorithm getting stuck on the login.

4.4 Exploration: Q-learning versus random search

In this section, the exploration speed of the tabular Q-learning algorithm is compared to that of the random search algorithm. As an experiment, tabular Q-learning and random search were run in the real product environment. By running the experiment for both methods and recording unique states over number of actions taken it can be seen that the Q-learning based algorithm is substantially faster at finding unique states than random search. Figure 25 shows the results of this experiment.

The states were calculated by fetching the information of html elements on the page, creating a vector containing all the information and calculating a sha256 hash of the vector. Thus the number of unique states is the number of unique hashes calculated so far. Both methods were run for 10,000 actions and the browser was reset to the home page of the product every 1000 actions. Running 10,000 actions took about 6 minutes for each of the methods. The number of unique states was calculated as an average of 10 runs, because the algorithms are probabilistic.

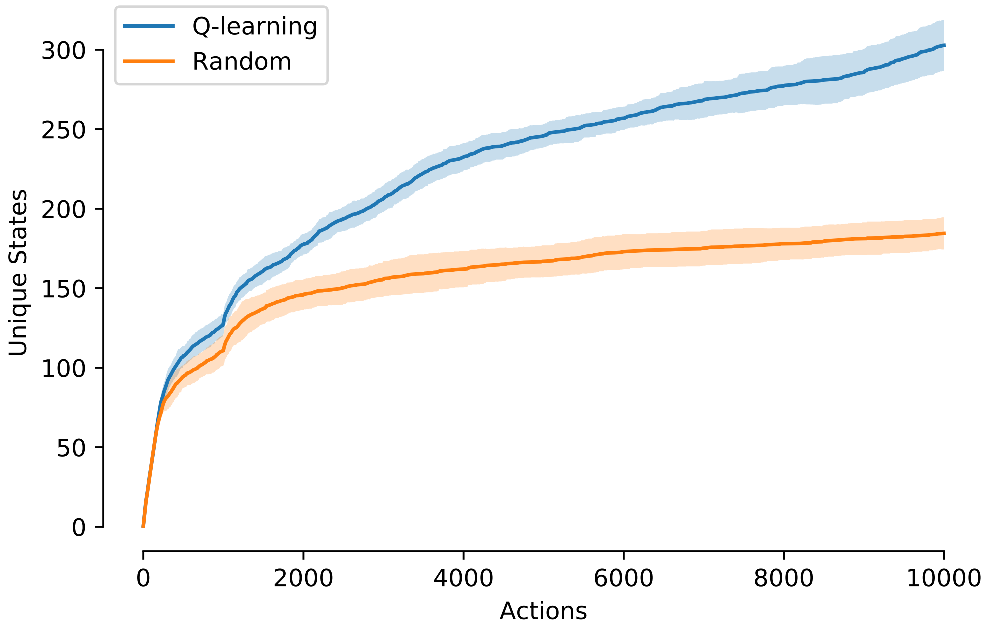


Figure 25: Number of unique states explored. Q-learning (Algorithm 5) shown with the blue line and random search (Algorithm 4) with the orange. Both methods were run 10 times. The darker color shows the mean and lighter color the standard deviation of the obtained results. Q-learning is more efficient at finding new states.

4.5 Testing the algorithm on a real product

Since the image-based means algorithm turned out to be the best proposed method, this section concentrates on comparing it against other algorithms. In terms of learning speed, the image-based means algorithm learns quickly to outperform the tabular Q-learning method. The untrained image-based means algorithm is slow and works similarly to the random agent. However, after only 8 minutes of training, the means algorithm outperforms Q-learning in the number of unique states per 1000 actions. Furthermore, after training for two days, the means algorithm finds substantially more unique states than Q-learning in 10,000 actions.

Image-based means method was compared against people performing the same task. The task was to find as many unique URLs as possible within 100 clicks. The image-based algorithm learned a deterministic policy that was able to find 52 unique URLs. As a comparison, the same task was first given to two human users. The users were only informed to navigate the website so that they find as many different URLs as possible. They were not told how the application works and what can be done with it. The task was performed by using a browser as usual, but with a program running in the background to keep track of how many unique URLs were found and how many clicks are left. During the task, the users were able to see how many URLs were found and how many times they can click. Performing the task took about 5 minutes, including the time to click and thinking about what to click to maximize URL count. As a result the two users found 37 and 38 unique URLs. The image-based means method performed the task in under 10 seconds.

The time difference is not very significant in such a short time, but if it were 1000 or 10,000 clicks, for example, it would be quite unfeasible for the human user.

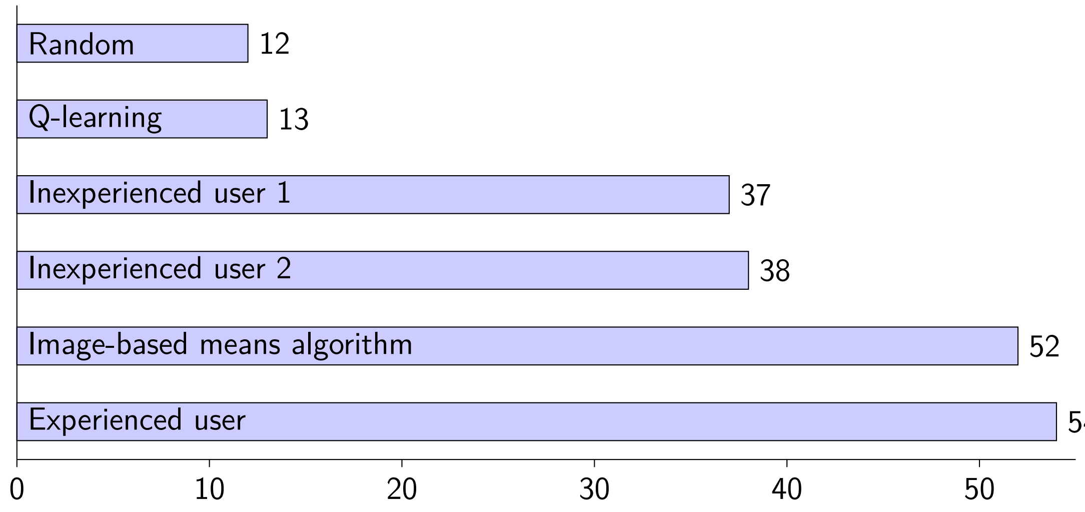


Chart 1: 100-click URL maximization task results

As shown in Figure 26, the image-based means algorithm outperforms the others and the element-based method is also notably better than the Q-learning and random algorithms. The interesting part is that both the image and element based methods were trained to maximize the number of unique states in 1000 clicks. But since the action selection is stochastic, both methods are able to find substantially more states when run longer. The algorithms have learned action probability distributions that favor those actions that lead to finding many unique states.

4.6 Performance and robustness

To verify that the image-based means solution works properly two additional experiments were executed. The method was tested by training it to navigate both the simple site (4.3.1) and the single page application (4.3.2). The environment in this experiment worked so that the site was chosen randomly with equal probability for both websites. It turned out that the agent is able to learn to navigate both sites efficiently and find all pages. This experiment shows that the policy can be trained to navigate many different sites and therefore it could potentially generalize. However, testing how well the policy generalizes is out of the scope of this thesis. Generalization would mean that the method could be only trained once and then be used for many different websites.

The ability to make decisions based on the state was also tested in the simple site login. By clearing the username and password fields in the login page after the agent had entered those, it was verified that the agent actually makes the decisions based on the screenshot rather than only memorizing the correct action order. The

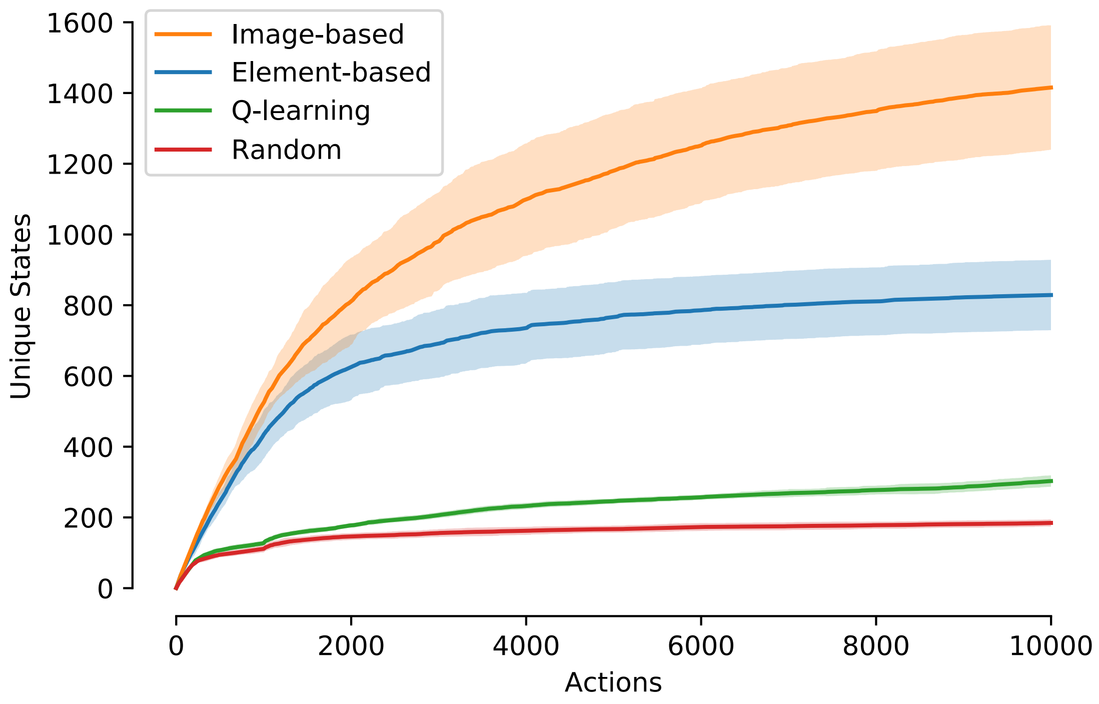


Figure 26: Comparison of the random (Algorithm 4), Q-learning (Algorithm 5), element-based (Algorithm 2) and image-based (Algorithm 3) algorithms testing their ability to find unique states. Each method was run 10 times. The darker color shows the mean and lighter color the standard deviation of the obtained results.

agent was able to recognize that the username or password was missing and enter them again before clicking the login button.

4.7 Curiosity in UI exploration

It is difficult to know what kind of reward function is the best to encourage the agent to explore. There have been multiple ways to improve exploration by creating a curiosity-based reward instead of using an environment-provided reward in the research community. One of these is curiosity-driven exploration, proposed by Pathak et al. [29]. Their solution uses an Intrinsic Curiosity Module (ICM) to give reward for finding states that cannot be predicted well. In the beginning the agent is bad at predicting the next state, but during training the agent improves and thus tries out things that lead to new states.

Since curiosity-based reward sounded promising, it was implemented for the UI exploration problem in this thesis. Figure 27 shows the training performance of the image based means algorithm on the real product environment with and without ICM. The reward is in this experiment the number of unique URLs. ICM seems to provide noise to the reward function in this case. Thus, the one with ICM is slower

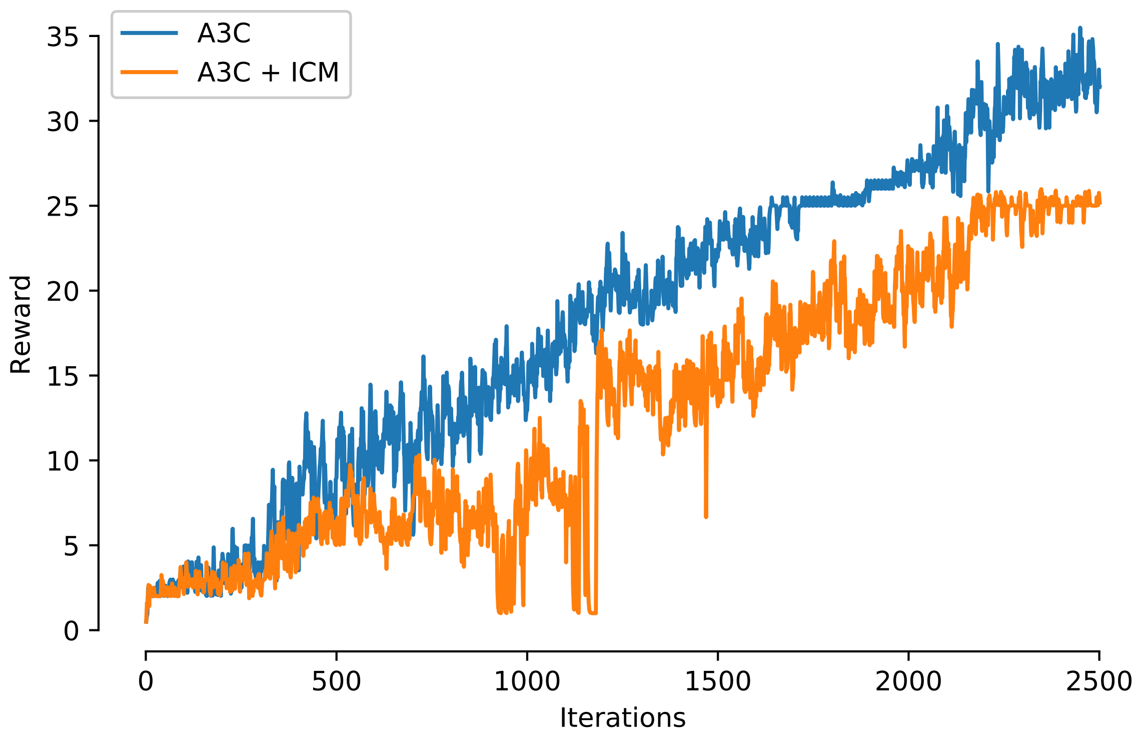


Figure 27: The performance of the image based algorithm with and without intrinsic curiosity module (ICM). The lines were smoothed using exponential moving average with discount coefficient 0.5. The reward is the number of unique URLs found.

to learn and has significant drops in performance. This can be explained by the fact that web pages are quite static; the pages change fairly little after the actions. Thus, the model quickly learns what comes next and therefore the curiosity reward is small.

The intrinsic curiosity module implementation that was done by the author of this thesis did not seem to work in cases where the camera does not move. For example in the Atari video game Montezuma's Revenge, the model quickly learned to predict mostly correct next state features and thus the prediction loss became very small. The same happened in the simple site environment (4.3.1) where the agent tries to type username and password and then click login. The prediction loss started high, but after not so many iterations, the prediction loss became very low and the agent was not curious anymore.

Curiosity as the only reward has a lot of potential in GUI testing. If the agent is able to learn with just the intrinsic curiosity, it could be used in environments where we have no access to the application or browser. It would suffice to get a screenshot of the current screen and be able to click and type. The agent would give itself reward based on how surprised it is for its actions. At first it would be surprised for everything and later learn to predict what is going to happen and thus it would move towards new functionality in the SUT. The method could be then used to test desktop applications, mobile applications, the graphical user interfaces of operating systems and more, assuming that taking screenshots and clicking and typing can be done programmatically. However, it requires more research to get a working curiosity-based solution.

5 Conclusion

We will now answer the research questions. In this thesis we saw that deep reinforcement learning can be used for exploratory software testing by designing a reward function that encourages trying out new things. We found that the state of the user interface can be represented as an element vector or as an image and we may use one action per element or probability for clicking each location on the UI. Supervised pretraining, i.e. training the agents with human demonstrations before applying reinforcement learning, was not needed, because the algorithms are able to learn from scratch. Generally, the problem of supervised pretraining is that the algorithm learns to imitate what was done in the demonstration, but the performance of the algorithm could be limited by the data. In addition, generalization to different scenarios can be difficult. Also, the training process may require a lot of data, thus the proposed methods are easier to use. If the system changes or a new system is tested and the trained policy does not work anymore, the proposed methods can be trained from scratch without requiring any demonstrations.

The presented methods can already be used to find problems in user interfaces. During the testing of the methods, novel problems were found in the development version of a real product. However, the methods could be improved in many aspects. The learning could potentially be sped up by using a more stable deep reinforcement learning algorithm such as proximal policy optimization (PPO). Also, more UI problems could be detected such as visual problems and readability of the text. The solution saves the taken actions (clicks and typed text) into a log file, but the taken actions could be saved as a runnable test. The test could be then used when fixing the problem to verify that it is actually fixed.

One of the most difficult factors of this thesis was that the learning process of the proposed methods took days on a real product. And since it was not known whether or not the algorithm could learn to maximize the number of unique states or URLs it was challenging to know when to stop the learning process and note that the algorithm did not work properly.

Luckily, since the same deep reinforcement learning algorithms had been used to train agents to play video games, it was possible to roughly estimate how many samples it would take. Many papers, such as [26], [34] and [14], have shown that it often requires millions of steps to reach good performance in games. And it turned out that it took around 8-15 million steps to reach the highest number of unique states per episode for the image-based means method.

Since the methods are able to find problems that are otherwise difficult to find, it is intriguing to start using them. We look forward to integrating the methods as part of the normal development pipeline.

5.1 For Future Studies

5.1.1 Generalization

The proposed methods are used to train deep-reinforcement-learning-based agents that can explore a specific UI efficiently. However, training takes a few days, thus it may not be feasible to retrain the methods often. Therefore it would be useful to find out how well the method generalizes to different versions of the same UI and to entirely different UIs. Do we need retraining if the UI changes slightly? Can we train a generic policy that can efficiently navigate a large number of different websites or web applications without training?

5.1.2 Content

Currently the proposed methods enter randomly-generated text to the text boxes. How could we design a model so that it can choose what type of input to put in a form input box?

Another open area for future research is how to handle more action types. These include for example email, password, number, textarea, and dropdown elements. How can we add support for drag and drop? Is it possible to expand the action space to include the possibility to choose these actions?

5.1.3 Reward function

This thesis used the number of unique URLs or states as the reward functions. However, maximizing unique URLs makes the agent navigate around many pages, but does not incentivize it to try different things, such as pressing buttons or typing text to text boxes, on those pages. On the other hand, depending on the application that is being tested, maximizing unique states may make the agent find a page where it can create a lot of new states and stay there. Thus, a better reward function could help the agent explore better and focus on more important parts of the application. Different reward functions could be tested by comparing them in their ability to find problems, how well they find new pages and how many things they try on each page.

References

[1] PVoodoo - deep reinforcement learning for trading. https://pvoodoo.blogspot.com/2017/09/deep-reinforcement-learning-for-trading.html. Accessed: 2019-02-04.

[2] Colah's blog - understanding lstm networks. https://pvoodoo.blogspot.com/2017/09/deep-reinforcement-learning-for-trading.html. Accessed: 2019-02-04.

[3] Francisco Almenar, Anna I Esparcia-Alcázar, Mirella Martínez, and Urko Rueda. Automated testing of web applications with testar. In *International Symposium on Search Based Software Engineering*, pages 218–223. Springer, 2016.

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

[5] C. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[6] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. Deep reinforcement fuzzing. *arXiv preprint arXiv:1801.04589*, 2018.

[7] Abdelwahhab Boudjelal, Zoubeida Messali, and Abderrahim Elmoataz. A novel kernel-based regularization technique for pet image reconstruction. *Technologies*, 5(2), 2017.

[8] L. Chen. Microservices: Architecting for continuous delivery and devops. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 39–397, April 2018.

[9] Rohit Choudhry and Kumkum Garg. A hybrid machine learning system for stock market forecasting. *World Academy of Science, Engineering and Technology*, 39(3):315–318, 2008.

[10] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, June 2009.

[11] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. *CoRR*, abs/1701.07232, 2017.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[13] David Griffis. A3C LSTM atari with pytorch plus a3g design. https://github.com/dgriff777/rl_a3c_pytorch, 2018.

[14] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Daniel Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017.

[15] Antawan Holmes and Marc Kellogg. Automating functional tests using selenium. In *Proceedings of the Conference on AGILE 2006*, AGILE '06, pages 270–275, Washington, DC, USA, 2006. IEEE Computer Society.

[16] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251 – 257, 1991.

[17] Riashat Islam, Peter Henderson, Maziar Gomrokchi, and Doina Precup. Reproducibility of benchmarked deep reinforcement learning tasks for continuous control. *CoRR*, abs/1708.04133, 2017.

[18] Alexander Jung. Machine Learning: Basic Principles. *ArXiv e-prints*, page arXiv:1805.05052, May 2018.

[19] Junhwi Kim, Minhyuk Kwon, and Shin Yoo. Generating test input with deep reinforcement learning. In *2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST)*, pages 51–58. IEEE, 2018.

[20] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[22] Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. Reinforcement learning on web interfaces using workflow-guided exploration. *CoRR*, abs/1802.08802, 2018.

[23] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[24] Rajesh Mathur, Scott Miles, and Miao Du. Adaptive automation: Leveraging machine learning to support uninterrupted automated testing of software applications. *arXiv preprint arXiv:1508.00671*, 2015.

[25] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[26] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q.

Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR.

[27] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[28] Bo Pang, Lillian Lee, et al. Opinion mining and sentiment analysis. *Foundations and Trends® in Information Retrieval*, 2(1–2):1–135, 2008.

[29] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International Conference on Machine Learning (ICML)*, 2017.

[30] K. J. Piczak. Environmental sound classification with convolutional neural networks. In *2015 IEEE 25th International Workshop on Machine Learning for Signal Processing (MLSP)*, pages 1–6, Sept 2015.

[31] Martin A. Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degrave, Tom Van de Wiele, Volodymyr Mnih, Nicolas Heess, and Jost Tobias Springenberg. Learning by playing - solving sparse reward tasks from scratch. *CoRR*, abs/1802.10567, 2018.

[32] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[33] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[34] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[35] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[36] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction.* MIT press, 1998.

[37] Lauren Wood, Arnaud Le Hors, Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Ian Jacobs, Gavin Nicol, Jonathan Robie, Robert Sutor, et al. Document object model (dom) level 1 specification. *W3C recommendation*, 1, 1998.

[38] Yuhuai Wu, Elman Mansimov, Shun Liao, Roger B. Grosse, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. *CoRR*, abs/1708.05144, 2017.

[39] Yuxin Wu and Yuandong Tian. Training agent for first-person shooter game with actor-critic curriculum learning. 2017.