Data Challenge Feasibility Research

Data Driven Innovation Challenge

Thomas Van der Molen

|  |  |
| --- | --- |
| **Project Information** | |
| Project members | Thomas van der Molen |
| Project Name | Leveraging reinforcement learning for automated testing |

Table of Contents

[Introduction 3](#_Toc155796906)

[Glossary 3](#_Toc155796907)

[Web-page interactions 4](#_Toc155796908)

[Selenium 4](#_Toc155796909)

[Showcase 5](#_Toc155796910)

[Selenium to RL Environment 7](#_Toc155796911)

[Agent 8](#_Toc155796912)

[Environment 8](#_Toc155796913)

[Gym Web Environment 8](#_Toc155796914)

[Observation Space 8](#_Toc155796915)

[Action Space 10](#_Toc155796916)

# Introduction

This document contains summarizations for the set-up and results of feasibility studies performed for the separate components necessary, established in the [Project Plan](../DataDrivenInnovationChallenge%20Plan%20-%20Thomas%20Van%20der%20Molen.docx) (it is highly recommended to read the Project plan first).

To utilize a reinforcement learning agent to find anomalous behavior within a web application, several large road blocks will have to be proven feasible first. These feasibility tests are performed in accordance to the [DOT Framework](https://ictresearchmethods.nl/), and will span across many research strategies such as **Library** with documented research, **Workshop** via jupyternotebooks, and **Lab** utilizing an MVP.

## Glossary

|  |  |
| --- | --- |
| **Term** | **Definition** |
| Reinforcement learning (RL) | A machine learning technique were the model (agent) learns from positive and negative feedback for given actions. |
| Agent | Within reinforcement learning, an agent refers to the model that performs actions and updates its internal knowledge of them based on the feedback received. |
| Environment | A virtual instance of a web-page that will be interacted with. |
| Webdriver | The interface that allows code to interact with a browser. |

# Web-page interactions

To interact with a website two things have to happen; first, we need to obtain the page information from the website, secondly we need to interact with elements on the website.

Thankfully, I am not the first one to have these requirements, as web scrapers, hackers, and tools to allow website to run in an app (such as a mobile app), have all created great tools for this.

For this research, we will be using the chrome service with [their webdriver](https://sites.google.com/chromium.org/driver/), as google has very mature development support for automated usage of their browser, and chrome is the [most popular browser in use currently](https://gs.statcounter.com/) (so any anomalous behavior encountered on chrome, should impact a large portion of a general user base).

A pie chart with different colored numbers

Description automatically generated

## Selenium

To programmatically interact with the browser and thus webdriver provided by chrome (google), [Selenium](https://www.selenium.dev/) will be used, this Open-source framework is primarily used for automating functional testing of web browsers (e.g. automatically testing interactions) and had become very popular with websites such as [LinkedIn](https://www.linkedin.com/), [WordPress](https://wordpress.com/) and [Shopify](https://www.shopify.com/) using it for their automated testing.

Selenium has support for [many languages and webdrivers](https://www.kualitatem.com/blog/selecting-the-scripting-language-for-new-webdriver#:~:text=Selenium%20WebDriver%20supported%20languages%20include,PHPUnit%2C%20and%20FitNesse%2C%20etc.), allowing us the freedom to switch to a different technology stack completely if necessary.

Selenium will be able to take care of both obtaining page information and interacting with the page programmatically all through the webdriver.

### Showcase

As a short showcase and part of the feasibility testing, a Selenium notebook (<Selenium.ipynb>) has been created, that will interact with a very simple testing Environment ([BasicErrorLog.html](Environments/BasicErrorLog.html)). This testing environment is an HTML page that contains a single target button, with an eventlistener that will throw an error when clicked.

A screenshot of a computer

Description automatically generated

Figure BasicerrorLog.html

Using Selenium in python, only requires [Selenium to be installed via pip](https://pypi.org/project/selenium/) and the [chrome driver it connects with](https://googlechromelabs.github.io/chrome-for-testing/) (assuming chrome itself is [installed already](https://www.google.com/chrome/)).

After installing the prerequisites, a chrome service and webdriver can be initiated and sent to retrieve the required webpage.

A screenshot of a computer program

Description automatically generated

Figure Initializing environment

After this, the driver can be interacted with directly to obtain information of the browser/webpage and interact/modify this information (Rule of thumb: If you can do it in javascript, you can do it with the driver, and if you can’t with javascript you probably can still do it with the driver).

After initializing the driver and obtaining the correct environment information, interactions can be performed with the page, such as finding a button and clicking it.

A screenshot of a computer screen

Description automatically generated

Figure Interacting with environment

Any information of use to us can be retrieved from the driver as well, such as the browser logs where the error was thrown to.

A screenshot of a computer

Description automatically generated

Figure Reading browser logs

All functionality of the python API for Selenium is also [documented on their dev page](https://www.selenium.dev/selenium/docs/api/py/).

## Selenium to RL Environment

Reinforcement learning agents require knowledge of their current available actions, against the current state. It uses this to form a quality mapping of what actions will be best to perform in which scenario.

To make the agent interact with the web-page in a meaningful way (to find possible anomalous behavior), it will need to be “steered” in a direction that might lead to situation were mistakes can happen, such as [keywords of interest](https://en.wikipedia.org/wiki/Sentiment_analysis), or interactable objects (such as elements with [eventlisteners](https://www.w3schools.com/js/js_htmldom_eventlistener.asp)).

These different types of objects can be obtained from the webdriver to create the current state of the webpage and all possible actions we allow the agent to perform. Due to the [ever evolving single-page-applications of the modern day](https://www.bloomreach.com/en/blog/2018/what-is-a-single-page-application), these values are continuously changing, thankfully smart people have already come up with RL models that can handle continuous states such as [DQNs](https://paperswithcode.com/method/dqn).

# Agent

A grey and white person's profile

Description automatically generatedTo create a reinforcement model that learns from the actions taken in a web environment (as explored in [web-page interactions](#_Web-page_interactions)), an agent is used that serves as the actor of the model or the interface between the environment and the model.

## Environment

We have previously established that the environments the agent will be working within, will be websites/webpages, for allowing actors to interact with environments, standards have been created to (in a perfect world) for environments and actors to be switched interchangeably. A widely used and actively maintained solution is [Gym](https://gymnasium.farama.org/) (Gymnasium). This standard provides base functionality for interacting and obtaining data from an environment, allowing any agent to expect the same functionality throughout different environments, such as [standard API’s for performing an action, querying available actions, etc](https://gymnasium.farama.org/api/env/).

## Gym Web Environment

For this project, a custom environment has been created, based on the Gym standard. For this environment the [Selenium](#_Selenium) web driver is used either in headless (no UI) or in human rendering mode (with visual browser window), this is effectively used as the back-end of the environment and is started up at the start (or reset) of the environment.

### Observation Space

One of the standards of Gym is [spaces](https://gymnasium.farama.org/api/spaces/), these spaces refer to a data structure that represents the distinct (static possible values such as “first”, “second”, “third”) or continuous (such as any value between 0.000 – 1.000) values.

#### Dynamic Spaces

Dynamic spaces When considering how to represent a website in a meaningful manner, so that the agent can effectively traverse and learn it comes with a lot of caveats. One issue that Gym does not enforce is dynamically changing spaces. As an example, a valid observation space might be a list of all buttons on a page, these values would be distinct values with the list being of size n = amount of buttons. Within the given observation space the amount of buttons and therefore the size of the list represented in the space can change depending on the current state. Below a visual representation of this on YouTube (Fig 5, Fig 6)

A screenshot of a video

Description automatically generatedA screenshot of a computer

Description automatically generatedThis kind of observation space, while not explicitly wrong, does cause some issues due to a fundamental limitation of how Reinforcement models represent states in their quality tables, as they expect the shape of a state to stay the same with the values changing.

Figure 6 Youtube login page buttons (via Sign In button, n = 4)

Figure Youtube landing page buttons (n = 13 interactables)

During research, a [very interesting research paper](https://arxiv.org/pdf/1905.03970.pdf) was found that tries to cover this topic, however this solution is still very limited in application and outside of the scope for reproducibility in this project.

#### Static Spaces

As creating a completely new type of reinforcement model is not within the scope of this project, an observation space has to be chosen that [abides by the limitation of these models](#_Dynamic_Spaces) and therefore offers a statically shaped observation space/state.

To accomplish this, the initial idea of observation spaces and the corresponding states has to be considered. States in reinforcement learning are used to give the model an idea of what the environment looks like at the current moment, as this can change what actions are best to undertake. This kind of state in essence is just a way to uniquely identify the current page the agent is on in our case (as the agent should perform different actions depending on it being on the home screen or the login screen).

#### Implementation

When considering the limitations our environment has to work within ([see Observation Space](#_Observation_Space)), we want to give the agent/model an idea of what page it is on, and if it is an important state to be in (as some pages like an admin management page might have more critical bugs than a generic contact information page).

For this several data is obtained from the current page and put as distinct values into the static observation space to create the current state, these values consist of; the page url and page title, these two factors create a good indicator of what unique page the agent is on, and: Count of eventlisteners, count of error messages, count of warning messages, count of invalid html elements, and count of keyword hits (special words to pay attention to that are pre-defined, think of “password” or “admin”) are used to indicate to the agent that the current page is good to be on or not.

### Action Space

In the same vein as the [Observation Space](#_Observation_Space), every gym environment as an action space that represents the possible actions an agent can take in the current state. Sadly, the [same dynamic sizing issues apply to this space as well](#_Dynamic_Spaces), and thus will have to be account for too.

The space shape limitation of most modern reinforcement learning models, causes us to not be able to simply give the agent the ability to interact with every event listener on a page, instead a statically defined action set is used that allows the agent to navigate throughout all interactable objects and can decide to interact with it or not, this approach is very similar to the methodology used to create [Turing-complete systems](https://en.wikipedia.org/wiki/Turing_completeness).

#### Convergence Stagnation

With the very popular Q-learning structure for reinforcement learning, one problem that is often encountered is a problem with the model [never converging, or finding its end objective](https://towardsdatascience.com/convergence-of-reinforcement-learning-algorithms-3d917f66b3b7) (this can be due to many factors, such as local minimums, unclear reward structures, etc).

This is also an issue with the proposed statically scoped (turing-esq atomic steps) action space, as a single action does not achieve much in the environment as it might just be cycling through possible action options before choosing one.

Most implementations of reinforcement models, expect each action and reward to be relatively very important to the performance of the model, for us however this is not the case as a combination of actions will lead to a new state with possible rewards attached.

Two possible solutions for this, is to reduce the importance of a single action, however this does not actually solve the problem as it will make convergence that much harder, due to the model getting very small rewards that are hard to learn from. A second and preferred option however, is to take multiple actions into a single action scope, where an action scope is terminated by the defined interaction act (e.g. an action scope might be nextitem->nextitem->nextitem->interact) and give a calculated reward based on all the actions taken at once (which would also slightly speed up the model’s performance due to the environment state calculations being shorthanded as they do not change completely).

#### Implementation

Based on the limitations, and possible solutions found, a very straightforward/limited action space is used for the agent to interact with, this being; next element and interact. For the next element action, a list of all elements with pre-defined eventlisteners is created that the agent can navigate through (resetting back to the first element when end of list is reached), and the interact action triggering the relevant eventlistener (NOTE: This does leave out any kind of input interactable such as input fields, as the agent will not have any way to give key inputs without making the action space dynamically sized). A common turing-complete actionset might include an action to navigate through the elements backwards (e.g. previous item), but this is excluded on purpose as this causes extra entropy into the system where the agent will have the chance to perform more non action terminating operations which can slow down training significantly.