Mälardalen University
School of Innovation Design and Engineering
Västerås, Sweden

Thesis for the Degree of Master of Science in Computer Science with
Specialization in Software Engineering 30.0 credits

REINFORCEMENT LEARNING
ASSISTED LOAD TEST GENERATION
FOR E-COMMERCE APPLICATIONS

Golrokh Hamidi
ghi19001@student.mdh.se

Examiner: Wasif Afzal
Mälardalen University, Västerås, Sweden

Supervisors: Mahshid Helali Moghadam
Mälardalen University, Västerås, Sweden

Company supervisor: Mehrdad Saadatmand,
RISE - Research Institutes of Sweden, Västerås, Sweden

June 17, 2020
Abstract

Background: End-user satisfaction is not only dependent on the correct functioning of the software systems but is also heavily dependent on how well those functions are performed. Therefore, performance testing plays a critical role in making sure that the system responsively performs the intended functionality. Load test generation is a crucial activity in performance testing. Existing approaches for load test generation require expertise in performance modeling, or they are dependent on the system model or the source code.

Aim: This thesis aims to propose and evaluate a model-free learning-based approach for load test generation, which doesn’t require access to the system models or source code.

Method: In this thesis, we treated the problem of optimal load test generation as a reinforcement learning (RL) problem. We proposed two RL-based approaches using q-learning and deep q-network for load test generation. In addition, we demonstrated the applicability of our tester agents on a real-world software system. Finally, we conducted an experiment to compare the efficiency of our proposed approaches to a random load test generation approach and a baseline approach.

Results: Results from the experiment show that the RL-based approaches learned to generate the effective workloads with smaller sizes and in fewer steps. The proposed approaches led to higher efficiency than the random and baseline approaches.

Conclusion: Based on our findings, we conclude that RL-based agents can be used for load test generation, and they act more efficiently than the random and baseline approaches.
# Table of Contents

1. Introduction .................................................. 1

2. Background ................................................ 3
   2.1. Performance ........................................... 3
   2.1.1 Performance Analysis ................................. 3
   2.2. Machine Learning ...................................... 4
   2.2.1 Reinforcement Learning ............................... 5
   2.2.2 Q-learning ........................................... 10
   2.2.3 Deep RL ............................................... 10

3. Related Work ............................................... 13

4. Problem Formulation ....................................... 15
   4.1. Motivation and Problem ............................... 15
   4.2. Research Goal and Questions .......................... 16

5. Methodology ............................................... 18
   5.1. Research Process ..................................... 18
   5.2. Research Methodology ................................ 18
   5.3. Tools for the Implementation ......................... 19

6. Approach .................................................. 21
   6.1. Defining the environment and RL elements ............ 21
   6.2. Reinforcement Learning Method ....................... 22
   6.2.1 Q-Learning .......................................... 22
   6.2.2 Deep Q-Network .................................... 24

7. Evaluation ................................................. 26
   7.1. System Under Test Setup ............................. 26
   7.1.1 Server Setup ....................................... 26
   7.1.2 Website Setup ..................................... 27
   7.2. Implementation ....................................... 27
   7.2.1 Workload Generation ............................... 27
   7.2.2 Q-Learning Implementation ......................... 32
   7.2.3 DQN Implementation ................................ 32
   7.3. Experiment Procedure ................................ 34

8. Results .................................................... 37

9. Discussion ................................................ 40
   9.1. Threats to Validity .................................. 40

10. Conclusions .............................................. 42

References .................................................. 46
1. Introduction

The industry is continuously finding ways to make software services accessible to more and more customers. One way to reach such customers (distributed over the globe) is the use of Enterprise Applications (EAs) delivering services over the internet. Inefficient and time-wasting software applications lead to customer dissatisfaction and financial losses [1, 2]. Performance problems are costly and waste resources. Furthermore, nowadays, using internet services and web-based applications have been extremely widespread among people and the industry. The significant role of internet services in people's daily life, and the industry is undeniable. Users around the world are dependent on internet services more than ever. Consequently, software success depends not only on the correct functioning of the software system but also on how well those functions are performed (non-functional properties). Responsiveness and efficiency are primitive requirements for any web-application due to the high expectations of users. For example, Google reported a 0.5 seconds increased delay in generating the search page resulted in 20% decrease in traffic by users [1]. Amazon also reported a 100 mili seconds delay in a web-page costed 1% loss in sales [2]. Accordingly, performance is a key success factor of software products, and it is of paramount importance to the industry, and a critical subject for user satisfaction. Tools allow companies to test software performance in both the development and design phases or even after the deployment phase.

Performance describes how well the system accomplishes its functionality. Typically, the performance metrics of a system are response time, error rate, throughput, utilization, bandwidth, and data transmission time. Finding and resolving performance bottlenecks of a system is an important challenge during the development and maintenance of a software [3]. The issues reported after project release are often performance degradation rather than system failures or incorrect response [4]. Two common approaches to performance analysis are performance modeling and performance testing. Performance models can be analyzed mathematically, or they could be simulated in case of having complex models [5]. Measuring and evaluating performance metrics of the software system through executing the software under various conditions by simulating concurrent multi-users with tools is the core of performance testing. One type of performance testing is load testing. Load testing evaluates the system’s performance (e.g., response time, error rate, resource utilization) by applying extreme loads on the system [6]. The load testing approaches usually generate workloads in multiple steps by increasing the workload in each step until a performance fault occurs in the system under test. The performance faults are triggered due to a higher error rate or response time than expected by the performance requirements [6]. Different approaches have been proposed for generating the test workload. Over the years, many approaches have been focused on testing for performance using system models or source code [6, 7]. These approaches require expertise in performance modeling, and the source code of the system is not always available. Various machine learning methods are also used in performance testing [8, 9]. However, these approaches require a significant amount of data for training. On the other hand, model-free Reinforcement Learning (RL) [10] is one of the machine learning techniques which does not require any training data set. Unlike other machine learning approaches, RL can be used in load testing to generate effective\(^1\) workloads without any training data-set.

As mentioned before, in software systems, performance bottlenecks could cause violations of performance requirements [11, 12]. Performance bottlenecks in the system will change during the time due to the changes in their source code. Load testing is a kind of performance testing in which the aim is to find the breaking points (performance bottlenecks) of the system by generating and applying workloads on the system. Manual approaches for test workload generations consume human resources; they are dependent on many uncontrolled manual factors and are highly prone to error. A possible solution to this problem is automated approaches for load testing. Existing automated approaches are dependent on the system model and may not be applicable when there is no access to the model or source code. There is a need for a model-free approach for load testing, which is independent of source code, system model, and requires no training data.

\(^1\)Effective, in terms of causing the violation of performance requirements (error rate and response time thresholds).
Contributions. In this thesis, our purpose is to generate efficient workload-based test conditions for a system under test without access to source code or system models, based on using an intelligent RL load tester agent. Intelligent here means that the load tester tries to learn how to generate an efficient workload. The contributions of this thesis are as follows.

1. Proposed model-free RL approach for load testing.

2. An evaluation of the applicability of the proposed approach on a real case.

3. An experiment for evaluating the two RL-based methods used in the approach i.e., q-learning and Deep Q-Network (DQN), against a baseline and a random approach for load test generation.

Method. In our proposed model-free RL approach, the intelligent agent can learn the optimal policy for generating test workloads to meet the performance analysis’s intended objective. The learned policy might also be reused in further stages of the testing. In this approach, the workload is selected in an intelligent way in each step instead of just increasing the workload size. We explain our mapping of the real-world problem of load test generation into an RL problem. We also presented the RL methods that we use in our approach i.e., q-learning and deep q-network (DQN). Then we present our approach with two variations of RL methods in detail. To evaluate the applicability of our proposed approach, we implement our RL-based approaches using open source libraries in Java. We use JMeter to generate our desired workload and apply the workload on an e-commerce website, deployed on a local server.

In addition, we conduct an experiment to evaluate the efficiency of the RL-based approaches. We execute the RL-based approaches, a baseline approach, and a random approach separately for comparison. We then compare the results of all approaches based on the efficiency (i.e., final workload size that violates the performance requirements and the number of workload increment steps for generating the workload).

Results The experiment results show that, in comparison to the other approaches, the baseline approach generates workloads with bigger sizes. Thus the baseline approach is not as efficient as the other approaches. The random approach performs better than the baseline approach since the average workload size generated by the random approach is lower than the baseline approach. However, the proposed RL-based approaches perform better than the random and baseline approaches. The results show that in both q-learning and DQN approaches, efficient workload size and the number of steps taken for generating workload in each episode converges to a lower value over time. The q-learning approach converges faster than the DQN. However, the DQN approach converges to lower values for the workload sizes. Our conclusion of the results is that both of the proposed RL approaches learn an optimal policy to generate optimal workloads efficiently.

Structure: The remainder of this thesis is structured as follows. In Section 2., we describe the basic knowledge and terms in performance testing and reinforcement learning. In Section 3., we introduce different approaches for load testing. In Section 4., we describe the motivation and problem, research goal, and research questions. In Section 5., we present the scientific method we use in this thesis and the tools we used. In Section 6., we provide our approach for generating load test and explain our RL-based load testers in detail. In Section 7., we provide an overview of the SUT setup, the process of applying workload using JMeter, and the implementation of our load tester. In Section 8., we describe the outcome of executing the implemented load testers on the SUT. We also explain the experiment procedure in this section. In Section 9., we present an interpretation of the results. Finally, in Section 10., we summarize the thesis report and presents conclusions and future directions.

\footnote{Efficient, in terms of optimal workload (workload size and number of steps for generating the workload).}
2. Background

In this section, we provide basic knowledge, terms, and notations in performance testing and reinforcement learning. The terms explained here will be used for describing the problem, approach, and solution in the following sections.

2.1. Performance

In this section, we discuss the terms related to performance and performance testing.

**Non-functional Quality Attributes of Software** Non-functional properties of a software system define the physiognomy of the system. These non-functional properties are often achieved by realizing some constraints over the functional requirements. Performance, security, availability, usability, interoperability, etc., are often classified under the term of run-time non-functional requirements. Then, modifiability, portability, reusability, integrability, testability, etc., are considered as non-runtime non-functional requirements. The run-time non-functional requirements can be verified by performance modeling in the development phase or by performance testing in execution.

**Performance** Performance is of paramount importance in connected systems and is a key success factor of software products. For example, EAs [1] such as e-commerce providing services to the customer over the globe, their success is subjected to the performance. Performance describes how well the system accomplishes its functionality. Efficiency is another term that is used in place of performance in some classifications of quality attributes [13, 14, 15]. Some performance metrics or performance indicators are:

- **Response Time**: The time between sending a request and beginning to receive the response.
- **Error Rate**: The proportion of erroneous units of transmitted data.
- **Throughput**: The number of processes that a system can handle per second.
- **Utilization on computer resources**: e.g., processor usage and memory usage.
- **Bandwidth**: The maximum rate of the data transferred in a given amount of time.
- **Data Transmission Time**: The amount of time that it takes for the transmitting node to put all the data on the wire.

Performance is one of the important factors that should also be taken into consideration in the design, development, and configuration phase of a system [5].

2.1.1 Performance Analysis

The performance of a system could be evaluated through measurements manually in a user environment or under controlled benchmark conditions [5]. Two conventional approaches to performance analysis are performance modeling and performance testing.

**Performance Modeling** It is not always feasible to measure the performance of the system or component, for example, in the design and development phase. In this case, the performance could be predicted based on models. Performance modeling is used during the design and development, and for configuration tuning and capacity planning. Other than quantitative predictions, performance modeling will give us insight into the structure and behavior of the system during the system design. To acquire performance measures, performance models can be analyzed mathematically, or they can be simulated in case of having complex models [5]. Some of the well-known modeling notations are queuing networks, Markov processes, and Petri nets, which are used together with analysis techniques to address performance modeling. [16, 17, 18].
**Performance Testing**  The IEEE standard definition of performance testing is: “Testing conducted to evaluate the compliance of a system or component with specified performance requirements” [19]. Measuring and evaluating the response time, error rate, throughput, and other performance metrics of the software system through executing the software under various conditions by simulating concurrent multi-users with tools is the core of performance testing. Performance testing could be performed on the whole system or on some parts of the system. Performance testing can also validate the efficiency of the system architecture, the system configurations, and the algorithms used by the software [20]. Some types of performance testing are load testing, stress testing, endurance testing, spike testing, volume testing, and scalability testing.

**Performance Bottlenecks**  Performance bottlenecks will result in violating performance requirements [11, 12]. The definition of a performance bottleneck is any system, component, or a resource that restricts the performance and prevents the whole system from operating properly as required [21]. The source of performance anomalies and bottlenecks are [11]:

- **Application Issues**: Issues in the application-level like incorrect tuning, buggy codes, software updates, and incorrect application configuration
- **Workload**: Application loads can effect in congested queues and resource and performance issues.
- **Architectures and Platforms**: For example, the behavior and effects of the garbage collector, the location of the memory and the processor, etc. can affect the system’s performance.
- **System Faults**: Faults in system resources and components such as software bugs, operator error, hardware faults, environmental issues, and security violations.

**Load Testing**  The load is the rate of different requests that are submitted to a system [22]. Load testing is the process of applying load on software to observe the software behavior and detect issues caused because of the load [20]. Load testing is applied through simulating multiple users to access the software at the same time.

**Regression testing**  Testing the software after new changes in the software is called regression testing. The aim of regression testing is to ensure the previous functionality of the software has not been violated, and it still meets the functional and non-functional requirements.

**Performance Testing Tools**  There are a variety of Performance Testing tools for measuring web application performance and load stress capacity. Some of these tools are open-source, and some have free trials. Some of the most popular performance testing tools are Apache JMeter, LoadNinja, WebLOAD, LoadUI, LoadView, NeoLoad, LoadRunner, etc.

**2.2. Machine Learning**

Nowadays, Machine Learning plays an important role in software engineering and is widely used in computer technology. Some well-known applications of machine learning algorithms in software engineering are:

- **Test data generation**: Transforming speech to text.
- **Drive autonomous vehicles**: For example, google self-driving cars.
- **Image Recognition**: Detecting an object in a digital image.
- **Sentiment Analysis**: Determining the attitude or opinion of the speaker or the writer.
- **Prediction**: For example, traffic prediction and weather prediction.
- **Information Extraction**: Extracting information from unstructured data.
- **Medical diagnoses**: Medical diagnoses based on clinical parameters.
Machine learning algorithms are a set of methods and algorithms in which the computer program learns to improve a task with respect to a performance measure based on experience. Machine learning uses techniques and ideas from artificial intelligence, probability and statistics, computational complexity theory, control theory, information theory, philosophy, psychology, neurobiology, and other fields [23]. Three major categories in learning problems are:

- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning

**Supervised Learning** In supervised learning, the training data set provides an output variable corresponding to each input variable. Supervised learning predicts the classification of other unlabeled data in the test data set based on the labeled training data in the training data set. Regression and classification are two types of supervised learning. The target is to minimize the expected output and the actual output of the learning system. Figure 1

![Figure 1: Supervised Learning](image)

**Unsupervised Learning** In unsupervised learning, unlike supervised learning, the training data set does not contain the output value of each input set, i.e., the training data set is not labeled. Unsupervised learning algorithms take unlabelled data as input and cluster the data in the same group based on their attributes.

### 2.2.1 Reinforcement Learning

In reinforcement learning, the agent tries to learn the best policy by experimenting and trial and error interaction with the environment. Reinforcement learning is goal-directed learning in which the goal of the agent is to maximize the reward [23]. In reinforcement learning problems, there is no training data set. In this case, the agent itself explores the environment to collect data and update its policy to maximize its expected cumulative reward over time (illustrated in Figure 2 [10]). "Trial-and-error search and delayed reward are the two most important distinguishing features of reinforcement learning"[10]. The agent is the learner and decision-maker, and everything outside the agent is the environment. The state is the current situation that is returned by the environment. Each action results in a new state and gives a reward corresponding to the state (or state-action). In a reinforcement learning problem, the reward function specifies the goal of the problem [10]. It is not specified for the agent which action to take in each state, and instead, the agent should discover taking which action leads to the most reward by trying them.

In the following, we discuss the main concepts in reinforcement learning:

**Agent and Environment** Everything except the agent is the environment; everything that the agent can interact with directly or indirectly. When the agent performs actions, the environment changes. This change is called the state-transition. As shown in Figure 2 [10], At each step $t$, the agent executes action $A_t$, receives a representation of the environment's state $S_t$ based on the observations from the environment, and receives a reward $R_t$. 

```
State
State contains the information used to determine what happens next. History is a sequence of states, actions, and rewards:

$$H_t = S_0, A_0, R_1, ..., S_t, A_t, R_{t+1}$$  \hspace{1cm} (1)

The agent state is a function of the history:

$$S_t = f(H_t)$$  \hspace{1cm} (2)

Action
Actions are the agent’s decision, which leads to a next state and provides a reward from the environment. Actions affect the immediate reward and can also affect the next state of the agent and consequently, the future rewards (delayed reward). So the actions may have long term consequences. The policy determines which action should be taken in each step.

Reward and Return
A reward $R_t$ is a scalar feedback signal that shows how well the agent is operating at step $t$. The learning agent tries to reach the goal of maximizing cumulative reward in the future. The Reward may be delayed, and it may be better to sacrifice immediate reward to gain more long-term reward. Reinforcement learning is based on the reward hypothesis: ”All goals can be described by the maximization of expected cumulative reward”. $R_a^s$ shown in equation 3 \cite{10} is the expected value of the reward of taking action $a$ from state $s$.

$$R_a^s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$$  \hspace{1cm} (3)

The return $G_t$ in equation 4 \cite{10} is the total discounted reward from time-step $t$.

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$  \hspace{1cm} (4)

Discount Factor
Discount factor $\gamma$ is a value in the interval $(0,1]$. A reward that occurs $k + 1$ steps in the future is multiplied by $\gamma^k$, which means the value of receiving reward $R$ after $k + 1$ time-steps is decreased to $\gamma^k R$. The discount factor indicates how much we value the future rewards. The more we trust our model, the discount factor would be nearer to 1, but if we are not certain about our model, the discount factor would be near to 0.

Markov Decision Process (MDP)
An MDP is an environment represented by a tuple $(S, A, P, R, \gamma)$. Where $S$ is a countable set of states, $A$ is a countable set of actions, $P$ is the state-transition probability function in equation 5 \cite{10}, $R$ is the reward function in equation 3, and $\gamma$ is the discount factor \cite{10}. The state-transition probability $P_{ss'}^{a}$ is the probability of going to state $s'$ by taking the action $a$ from state $s$. Almost all reinforcement learning problems can be formalised as MDPs.

$$P_{ss'}^{a} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$$  \hspace{1cm} (5)

In a MDP, the state is fully observable i.e. the current state completely characterises the process. A state $S_t$ (current state in time $t$) is Markov if and only if it follows the rule in equation 6 \cite{10},
meaning that the future state is only dependent of the present and it is independent of the past. A Markov state contains every relevant information from the history. So when the state is specified, the history may be thrown away.

**Partially Observable Markov Decision Process (POMDP)** In POMDP, the agent is not able to directly observe the environment, meaning the environment is partially observable to the agent. So unlike MDP, the agent state is not equal to the environment state. In this case, the agent must construct its own state representation.

**Policy** The policy \( \pi \) is the agent’s behavior function. It is a function from a state to action. A deterministic policy specifies which action should be taken in each state; it takes a state as an input and its output is an action:

\[
a = \pi(s)
\]

A stochastic policy (equation 8 [10]) determines the probability of the agent taking a specific action in a specific state:

\[
\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]
\]

**Value function** The value function is a prediction of future reward that is used to evaluate how good a state is. The value function \( v_\pi(s) \) of a state \( s \) under policy \( \pi \) is the expected return of following policy \( \pi \) starting from the state \( s \). The value function for MDPs is shown in equation 9 [10]:

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

\( v_\pi(s) \) is called the state-value function for policy \( \pi \). If terminal states exist in the environment, there value is zero.

The value of taking action \( a \) in state \( s \) under policy \( \pi \) is \( q_\pi(s,a) \) which is called the action-value function for policy \( \pi \) or the q-function shown in equation 10 [10]:

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

\( q_\pi(s,a) \) is the expected return starting from state \( s \), taking the action \( a \) and future actions based on policy \( \pi \).

**Bellman Equation** The Bellman equation explains the relation between the value of a state or state-action with its successors. The Bellman equation for \( v_\pi \) is shown in equation 11 [10]:

\[
v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[ r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s'] \right]
\]

\[
= \sum_a \pi(a|s) \sum_{s'} p(s'|s, a) \left[ r + \gamma v_\pi(s') \right]
\]

Where \( p(s', r|s, a) \) is the probability of going to state \( s' \) and receiving the reward \( r \) by taking the action \( a \) from state \( s \). Figure 3 [10] helps explaining the equation. Based on this equation, the value of a state is the average of its successor states’ values plus the reward of reaching them, weighting each state value by the probability of its occurrence. This recursive relation of states is a fundamental property value function in reinforcement learning.

The Bellman equation for q-value (action value) \( q_\pi(s,a) \) is shown in equation 12 [10]:
Golrokh Hamidi

Reinforcement Learning Assisted Load Test Generation

Figure 3: Backup diagram for $v_\pi$

$$q_\pi(s, a) = \sum_{s', r} p(s', r|s, a) [r + \gamma \sum_{a'} \pi(a'|s')q_\pi(s', a')]$$ (12)

This equation is clarified in Figure 4 [10].

Figure 4: Backup diagram for $q_\pi$

**Episode**  A sequence of states starting from an initial state and finishing in a terminal state is named episode. Different episodes are independent from each other. Figure 5 gives an overview of an episode.

Episodic and Continuous tasks  There are two kinds of tasks in reinforcement learning: episodic and continuous. Unlike continuous tasks, episodic tasks are when the interaction of the agent with the environment is broken down into separate episodes.

Policy Iteration  Policy iteration is the process of achieving the goal of the agent, which is finding the optimal policy $\pi^*$. Policy iteration consists of two parts; policy evaluation and policy iteration, which are executed iteratively. Policy evaluation is the iterative computation of the value functions for a given policy while the agent interacts with the environment. And policy improvement is enhancing the policy by choosing actions greedily with respect to the recently updated value function:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} ... \xrightarrow{E} \pi_* \xrightarrow{E} v_{\pi_*}.$$ (13)

Value Iteration  Value iteration is finding optimal value function iteratively. When the value function is optimal, then the policy out of it is also optimal. Unlike policy iteration, there is no explicit policy in value iteration, and the actions are chosen directly based on the optimal (converged) value function. Finding optimal value function is a combination of policy improvement and truncated policy evaluation.
Exploration and Exploitation  The reinforcement learning agent should choose the actions that have tried before, which have the highest return; this is exploitation. On the other hand, the agent should try new actions that have not selected before to find these best actions; this is exploration. There is a trade-off between exploration and exploitation in the learning process and making a balance between them is one of the challenges in reinforcement learning problems.

$\varepsilon$-Greedy Policy  An $\varepsilon$-greedy policy allows performing both exploration and exploitation during the learning. $\varepsilon$ is a number in the range of $[0,1]$ is chosen. In each step, the probability of selecting the best action (best action based on the main policy which is extracted from the q-table) is $1-\varepsilon$, and a random action is selected by the probability of $\varepsilon$.

Monte Carlo  Monte Carlo methods are a class of algorithms that repeat random sampling to achieve a result. One of the methods used in reinforcement learning is the Monte Carlo method to estimate value functions and find the optimal policy by averaging the returns from sample episodes. In this method, each episodic task is considered as an experience, which is a sample sequence of states, actions, and rewards. By using this method, we only need a model that generates sample transitions, and there is no need for the model to have complete probability distributions of all possible transitions and rewards. A simple Monte Carlo update rule is shown in equation 14 [10]:

$$V(S_t) \leftarrow V(S_t) + \alpha [G_t - V(S_t)] \quad (14)$$

Where $G_t$ is the return starting from time $t$ and $\alpha$ is the step-size (learning rate).

Temporal-Difference (TD) learning  Temporal-difference learning is another learning method in reinforcement learning. The TD method is an alternative to the Monte Carlo method for updating the estimation of the value function. The update rule for the value function is shown in equation 15 [10]:

$$V(S_t) \leftarrow V(S_t) + \alpha [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \quad (15)$$

Unlike Monte Carlo, TD learns from incomplete episodes. TD can learn after each step and does not need to wait for the end of the episode. The algorithm 1 explains $TD(0)$ [10]:

Algorithm 1: TD(0) for estimating $v_\pi$

Input: the policy $\pi$ to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialise $V(s)$, for all $s \in$ state space, arbitrarily except that $V(terminal) = 0$

for each episode do
  Initialize $S$
  for each step of episode do
    $A \leftarrow$ action given by $\pi$ for $S$
    Take action $A$, observe $R, S'$
    $V(S) \leftarrow V(S) + \alpha [R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$
  end
  until $S$ is terminal
end

Experience Replay  In a reinforcement learning algorithm, the RL agent interacts with the environment and updates the policy, value functions, or model parameters iteratively based on the observed experiment in each step. The data collected from the environment would be used once for updating the parameters, but it would be discarded in the future steps. This approach is wasteful because some experiences may be rare but useful in the future. Lin et al. [24] introduced experience replay as a solution to this problem. An experience (state-transition) in their definition [24] is a tuple of $(x, a, y, r)$ which means taking action $a$ from state $x$ and going to state $y$ and getting the reward $r$. In the experience replay method, a buffered window of N experiences is saved in the memory, and the parameters are updated with a batch of transitions in the experience replay, which are chosen based on different approaches e.g., randomly [25] or prioritized experiences.
Experience replay allows the agent to reuse the past experiences in an effective way and use them in more than one single update as if the agent experiences what it has experienced before again and again. Experience replay will speed up the learning of the agent, which leads to quicker convergence of the network. In addition, faster learning leads to less damage to the agent (the damage is when the agent takes actions based on bad experiences; therefore, it experiences a bad experience again and so on). Experience replay consumes more computing power and more memory but reduces the number of experiments for learning and the interaction of the agent with the environment, which is more expensive. Schau et al. [26] explain many stochastic gradient-based algorithms which have the i.i.d. assumption which is violated by strongly correlated updates in the RL algorithm and experience replay will break this temporal correlation by applying recent and former experiences in each update. Using experience replay has been effective in practice, for example Mnih et al. [25] applied experience replay in the DQN algorithm to stabilize the value function’s training. Google DeepMind also significantly improved the performance of the "Atari" game by using experience replay with DQN.

2.2.2 Q-learning

Q-learning is one of the basic reinforcement learning algorithms. Q-learning is an off-policy TD control algorithm. Methods in this family learn an approximator q-function for the optimal action-value function $Q^*$. In this algorithm the q-values of every possible state-action pairs are stored in a table named q-table. The q-table is updated based on the Bellman equation 16 [10]:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$  \hspace{1cm} (16)

The action is usually selected by an $\epsilon$-greedy policy. But the q-value is updated independent of the policy being followed (off-policy) algorithm, and based on the next action which has the maximum q-value. The q-learning algorithm is shown in Algorithm 4.

2.2.3 Deep RL

Deep reinforcement learning refers to the combination of RL with deep learning. Deep RL is nonlinear function approximation methods like artificial neural network (ANN) using SGD [10].

Value Function Approximation  Function approximation is used in RL because in large environments there are too many states and actions to be stored in the memory, also it is too slow to learn the value of each state/state-action individually. So the idea is to generalize from the visited states to the states which have not been visited yet. Hence the value function is estimated with function approximation:

$$\hat{v}(s, w) \approx v_\pi(s)$$
$$\hat{q}(s, a, w) \approx q_\pi(s, a)$$

Where $w$ is the weight vector, for example, $w$ is the feature weights in q linear function approximator, which returns the estimated value of each state by multiplying in the state’s feature vector. The dimensionality of $w$ is much less than the number of states and changing the weight vector, changes the estimated value of many states, therefor when $w$ is updated after each action from a single state, not only the value of that specific state will update, many states’ values will be updated too. This generalization makes learning faster and more powerful. Moreover, using function approximation makes reinforcement learning applicable to problems with partially observable environments.

There are many function approximators, e.g., linear function of features, artificial neural network, decision tree, nearest neighbor, Fourier/wavelet bases, and etc. For value function, approximation differentiable function approximators are used e.g., linear function and neural networks.
Stochastic Gradient Descent  Stochastic Gradient Descent or SGD is an optimization algorithm. This algorithm is used in machine learning algorithms, like training artificial neural networks used in deep learning. In this method, the goal is to find some model parameters which optimize an objective function by updating a model iteratively over multiple discrete steps. Optimizing an objective function is minimizing a loss function or maximizing a reward function (fitness function). In each step, the model makes some predictions based on the samples in the training data set, and based on the set of current internal parameters; then the predictions are compared to the real expected outcomes in the data set by calculating performance measures like mean square error. Then the gradient of the error is calculated and used to update the internal model’s parameters to decrease the error. Sample size, batch size, and epoch size are some hyperparameters in SGD [27]:

- Sample: A training data set contains many samples. A sample could be referred to as an instance, observation, input vector, or a feature vector. A sample is a set of inputs and an output. The inputs are fed into the algorithm, and the output is compared to the prediction by calculating the error.

- Batch: The model’s internal parameters would get updated after applying a batch of samples to the model. At the end of applying each batch of samples to the model, the error is computed. The batch size can be equal to the training data set size (Batch Gradient Descent), it can be equal to 1 meaning each batch is a sample in the data set (Stochastic Gradient Descent), and it can be between 1 and the training set size (Mini-Batch Gradient Descent). 32, 64, and 128 are popular batch sizes in mini-batch gradient descent.

- Epoch: The whole training data set is fed to the model once in each epoch. In every epoch, each sample will update the internal model parameters for one time. So in an SGD algorithm, there are two for-loops; the outer loop is over the number of epochs, and the inner loop iterates over the batches in each epoch.

There is no specific rule for configuring these parameters. The best configuration differs for each problem and is obtained by testing different values.

Deep Q-Network (DQN)  Deep Q-Network is a more complex version of q-learning. In this version, instead of using the q-table for accessing q-values, the q-values are approximated using an ANN.

Double Q-learning  Simple q-learning has a positive bias in estimating the q-values; it can overestimate q-values. Double q-learning is an extension of q-learning which overcomes this problem. It uses two q-functions, and in each update, one of the q-functions is updated based on the next state’s q-value from the other q-function [28]. The double q-learning algorithm is shown in Algorithm 2 [28]:

![Diagram of q-value function approximation](image.png)
**Algorithm 2:** Double Q-learning

Initialize $Q^A, Q^B, s$

repeat
- Choose $a$, based on $Q^A(s,.)$ and $Q^B(s,.)$, observe $r, s'$
- Choose (e.g. random) either UPDATE(A) or UPDATE(B)
  - if UPDATE(A) then
    - Define $a^* = \max_a Q^A(s', a)$
    - $Q^A(s, a) \leftarrow Q^A(s, a) + \alpha [r + \gamma Q^B(s', a^*) - Q^A(s, a)]$
  - else if UPDATE(B) then
    - Define $b^* = \max_a Q^B(s', a)$
    - $Q^B(s, a) \leftarrow Q^B(s, a) + \alpha [r + \gamma Q^A(s', b^*) - Q^B(s, a)]$
  - end
- $s \leftarrow s'$
until end;

**Double Deep Q Networks (DDQN)** The idea of double q-learning can be used in DQN [29]. There is an online network, a target network, and the online network gets updated based on the q-value from the target network. The target network is freezeed and gets updated from the online network after N steps. The other way is to smoothly average for every N number of last updates. N is the "target DQN update frequency".
3. Related Work

As mentioned before, in this study, we aim to detect certain workloads that cause performance issues in the software. To accomplish this objective, we use a reinforcement learning approach that applies workloads on the system and learns how to generate efficient workloads by measuring the performance metrics. Measuring performance metrics (e.g., response time, error rate, resource utilization) by applying various loads on the system under different execution conditions and different platform configurations is a common approach in performance testing [30, 31, 32]. Also discovering performance problems like performance degradation and violation of performance requirements that appear under specific workloads or resource configurations is a usual task in different types of performance testing [33, 6, 34].

Different methods have been introduced for load test generation, e.g., analyzing system model, analyzing source code, modeling real usage, declarative methods, and machine learning-assisted methods. We provide a brief overview of these approaches in the following:


Analyzing source code  Zhang et. al. [6] present a symbolic execution-based approach using the source code for generating load tests. Yang and Pollock [39] introduced a method for stress testing, limiting the stress test to parts of the modules that are more vulnerable to workloads. They used static analysis of the module’s code to find these parts.

Modeling real usage  Draheim et. al. [40] presents an approach for load testing of based on stochastic models of user behavior. Lutteroth and Weber [41] provide a stochastic form-oriented load testing approach. Shams et. al. [42] uses an application model-based approach that is an extension of Finite State Machines and models the user’s behaviour. Vögele et. al. [43] use Markov Chain for modeling user behaviour in workload generation. All the named papers here proposed approaches for generating realistic workloads.

Declarative methods  Ferme and Pautasso [44] conduct performance tests using their model-driven framework that is programmed by a declarative domain-specific language (DSL) provided by them. Ferme and Pautasso [45] also use BenchFlow that is a declarative performance testing framework, to provide a tool for performance testing. This tool uses DSL for the test configuration. Schulz et. al. [46] generate load test using a declarative behavior-driven approach where load test specification is in natural language.

Machine learning-assisted methods  Some approaches in load testing context, use machine learning techniques for analyzing the data collected from load testing. For example, Malik et al. [47] use and compare supervised and unsupervised approaches for analyzing the load test data (resource utilization data) in order to detect performance deviation. Syer et al. [8] use the clustering method for detecting anomalies (threads with performance deviations) in the system based on the resource usage of the system. Koo et al. [9] provides a RL-based symbolic execution to detect worst-case execution paths in a program. Note that symbolic execution is mostly used in more computational programs manipulating integers and booleans. Grechanik et al. [48] presents a feedback-directed method for finding performance issues of a system by applying workloads on a SUT and analyzing the execution traces of the SUT to learn how to generate more efficient workloads.
### Table 1: Overview of Related Work

<table>
<thead>
<tr>
<th>Reference</th>
<th>Required Input</th>
<th>General Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7, 35, 36, 37, 38]</td>
<td>System model</td>
<td>Generate performance test cases using Petri nets, usage pattern model, and UML model</td>
</tr>
<tr>
<td>[39, 6]</td>
<td>Source Code</td>
<td>Finding performance requirements violation via static analysis and symbolic execution</td>
</tr>
<tr>
<td>[40, 41, 42, 43]</td>
<td>User behaviour model</td>
<td>User behaviour simulation-based load testing</td>
</tr>
<tr>
<td>[44, 45, 46]</td>
<td>Instance Model of Domain-Specific Language</td>
<td>Propose Declarative methods for performance modeling and testing</td>
</tr>
<tr>
<td>[8, 47]</td>
<td>Training set</td>
<td>Uses Machine learning-assisted methods for load test generation</td>
</tr>
<tr>
<td>[9, 48, 49]</td>
<td>System/program inputs</td>
<td>Finding worst-case performance issues using RL</td>
</tr>
<tr>
<td>This Thesis</td>
<td>List of available transactions</td>
<td>Generate optimal workloads that violates the performance requirements, using RL</td>
</tr>
</tbody>
</table>

Ahmad et al. [49] try to find the performance bottlenecks of the system using an RL approach named PerfXRL, which uses a DDQN algorithm. This is one of the more similar approaches to our approach recently published. In their approach, each test scenario is a sequence of three constant requests to a web application. These requests have four variables in total, and the research aim is to find combinations of these four variables, which cause a performance violation. So the performance testing is done by executing test cases in which each test case is a sequence of three constant requests, and unlike our approach, no load testing is performed in this paper. They evaluate their approach by comparing the number of performance bottleneck request scenarios found by the PerfXRL approach with the number of performance bottleneck request scenarios found by a random approach. This comparison is made for different sizes of input value spaces. They show that for input value spaces bigger than a certain size (150000) the PerfXRL approach identifies more performance bottlenecks than the random approach.

Unlike most of the mentioned approaches, our approach is model-free and does not require access to the source code or a system model for generating load tests. On the other hand, unlike many of the machine learning approaches, our proposed approach does not need previously collected data, and it learns to generate workload while interacting with the system.
4. Problem Formulation

The objective of this thesis is to propose and evaluate a load testing solution that is able to generate an efficient test workload, which results in meeting the intended objective of the testing, e.g., finding a target performance breaking point without access to system model or source code.

4.1. Motivation and Problem

With the increase of dependence on software in our daily lives, the correct functioning and efficiency of Enterprise Applications (EAs) delivering services over the internet are crucial to the industry. Software success not only depends on the correct functioning of the software system but is also dependent on how well these functions performed i.e., non-functional properties like performance requirements. Performance bottlenecks can affect and harm performance requirements [11, 12]. Therefore, recognizing and repairing these bottlenecks are crucial.

The source of performance anomalies and bottlenecks can be application issues (i.e., source code, software updates, incorrect application configuration), workload, the systems architecture and platforms, and system faults in systems resources and component (e.g. software bugs, environmental issues, and security violations.) [11]. The source code would change during the continuous integration/delivery (CI/CD) process and software updates. The workload on the system is constantly changing, also the environmental issues and security conditions do not remain the same during the software’s life cycle. Therefore the performance bottlenecks in the system will change during time, and it is not easy to follow the model-driven approaches for performance analysis. To perform performance analysis that can consider all mentioned causes of performance bottlenecks we can use model-free performance testing approaches.

In addition, an important activity in performance testing is the generation of suitable load scenarios to find the breaking point of the software under test. Manual workload generation approaches are heavily dependent on the tester’s experience and are highly prone to error. Such approaches for performance testing also consume substantial human resources and are dependent on many uncontrolled manual factors. The solution to this matter is using automated approaches. However, existing automated approaches for finding breaking points of the system heavily rely on the system’s underlying performance model to generate load scenarios. In cases where the testers have no access to the underlying system models (describing the system), such approaches might not be applicable.

One other problem with existing automated approaches is that they do not reuse the data collected from previous load test generation for future similar cases, i.e. when the system should be tested again because of the changes made in the system during the time for maintenance, and scalability etc. There is a need for an automated, model-free approach for load scenario generation which can reuse learned policies and heuristics in similar cases.

Many model-free approaches for load generation, just keep increasing the load until performance issues appear in the system. The workload size is one factor that affects the performance, although the structure of the workload is another important factor. Selecting a certain combination of loads in the workload can lead to a violation of performance requirements and detecting performance anomalies with a smaller workload. A well-structured smaller workload can more accurately detect the performance breaking points of the system with lower resources for simulating workloads. In addition, a well-structured smaller workload can result in increase coverage at the system-level. Finding these specific workloads are difficult because it requires an understanding of the system’s model. [6]

Using model-free machine learning techniques such as model-free reinforcement learning [10] could be a solution to the problems mentioned above. In this approach, an intelligent agent can learn the optimal policy for performance analysis and load test scenarios that violate system performance. This method can be used independently of the system’s and environment’s state in
different conditions, and it does not need to access the source code or system model. The learned policy could also be reused in further stages of the testing (e.g., regression testing).

4.2. Research Goal and Questions

We intend to formulate a new method for load test generation using reinforcement learning and evaluate it by comparing it with random and baseline methods. Our technical contribution in this thesis is the formulation and development of an RL based agent, that will learn the optimal policy for load generation. We aim to evaluate the applicability and efficiency of our approach using an experiment research method.

The object of the study is an RL based load test scenario generation approach. The purpose is proposing and evaluating an automated, RL-based load test scenario generation tool. The quality focus is the well-structured efficient test scenario, the final size of its workload, and the number of steps for generating the workload. The perspective is from the researcher’s and tester’s point of view. The experiment is run using an e-commerce website as a system under test. Based on the GQM template for goal definition, presented by Basili and Rombach [50] our goal in this study is:

Formulate and analyze an RL-based load test approach
for the purpose of efficient\(^3\) load test generation
with respect to the structure and size of the effective\(^4\) workload, and the number of steps to generate it
from the point of view of a tester/researcher
in the context of an e-commerce website as a system under test

Based on our research goal we define the following research questions:

**RQ1: How can the load test generation problem be formulated as an RL problem?**

To solve the problem of load generation with reinforcement learning, a mapping should be done from the real-world problem to an RL problem environment and elements. The elements are the states, actions, and reward function (Figure 8). The aim of this research question is to find suitable definition of states, actions, and reward function in this problem.

---

\(^3\)Efficient, in terms of optimal workload (workload size and number of steps for generating the workload).

\(^4\)Effective, in terms of causing the violation of performance requirements (error rate and response time thresholds).
RQ2: *Is the proposed RL-based approach\(^5\) applicable for load generation?* After formulating the problem into an RL context, it is essential to evaluate the applicability of the approach on a real-world SUT. Answering this research question requires implementing the approach and setting up a SUT on which the generated load scenarios can be executed (see Section 7.1.).

RQ3: *What RL-based method is more efficient in the context of load generation?* Reinforcement learning can be applied using various algorithms like q-learning, SARSA (State-Action-Reward-State-Action), DQN, and Deep Deterministic Policy Gradient (DDPG). The aim of this research question is to choose at least two RL methods and find the most efficient (in terms of optimal) among them. In our case, we chose q-learning (a very basic RL algorithm) and DQN (an extended q-learning method). In addition, we also compare the results of the RL-based methods with a baseline and a random load generation methods.

\(^5\)derived from RQ1
5. Methodology

A research method guides the research process in a step-by-step iterative manner. We use well-established research methods to realize our research goals. The core of our research method is the research process illustrated in Figure 5.1. The research process we used (to guide our research method) is a modification of the four steps research framework proposed by Holz et al. [51]. In the rest of this section, we presented our research process (in Section 5.1.) followed by a discussion on the research method used in Section 5.2.. Finally, we present the tools used for implementation in this thesis, in Section 5.3..

5.1. Research Process

In this subsection, we outline the research process that we are following throughout this thesis.

Our research process started with forming a suitable research goal and research questions (as formulated in Section 4.2.). As discussed, the objective of our research is to propose and evaluate an automated model-free solution for load scenario generation. The main objective and research goal were identified in collaboration with our industrial partner (RISE Research Institutes of Sweden AB) by reviewing their needs. We then identified specific challenges of the adoption of performance testing approaches with our industrial partner. We realized that existing approaches require knowledge of performance modeling and access to source code, which limits the adoption of such approaches. We conducted a state-of-the-art review (some parts of it is presented in Section 2.) to identify the gaps in the literature. In the next step, we formulated an initial version of the problem which produced our thesis proposal. In the next step of our research process, we formulated and initial RL based solution that does not require any underlying model of the system and can reuse the learned policy in the future. This formulated solution helped in realizing our primary research goal. We then conducted an experiment to evaluate our solution on an e-commerce software system. Note that our research process was iterative and incremental.

5.2. Research Methodology

We conducted an experiment for answering our RQ3 following the guidelines presented by Wohlin et al. [52]. An experiment is a systematic formal research method in which the effects of all involved variables can be investigated in a controlled way. Thus, we can investigate the effect of our treatments (the different load test generation methods) on the outcome (size of workload generated which hit the thresholds, i.e., violates the performance requirements and the number of steps taken for generating this workload). Since our experiment’s goal is to answer RQ3 (which requires quantitative data to answer), the experiment research method is helpful in obtaining quantitative data about the objectively measurable phenomenon. In our case, the nature of the experiment is quantitative, i.e., comparing our RL-based load test generation approaches with a baseline and a random approach. The comparison is made based on the size of the workload generated that hits
the defined error rate and response time thresholds. In addition, the comparison is also made based on the number of workload increment steps required for each approach to generate a workload that hits the thresholds.

**Experiment Design** The procedure of our experiment is explained in Section 7.3. Here we provide the standard definition of experiment terminologies in the guidelines [52], and we define them in our experiment:

- **Independent variables:** “all variables in an experiment that are controlled and manipulated.” [52] In this experiment, the independent variables are the client machine generating workload, the client machine configuration, the network, the SUT server machine, and the SUT server configurations, and the parameters in Table 5.

- **Dependent variables:** “Those variables that we want to study to see the effect of the changes in the independent variables are called dependent variables.” [52] The dependent variables in this experiment are:
  - size of the final workload generated that hits the defined error rate and response time thresholds.
  - number of workload increment steps required to generate a workload that hits the thresholds.

- **Factors:** one or more independent variables that the experiment studies the effect of changing them. The Factor, in our case, is the load test generation method.

- **Treatment:** “one particular value of a factor.” [52] The treatments in our experiment are a baseline method, a random method, a q-learning method, and a DQN method for our factor load test generation method.

- **Subjects:** the subject, in our case, is the client machine generating workload. The properties of this machine are shown in Table 4.

- **Objects:** Instances that are used during the study. The object in our case is the SUT. The SUT is an e-commerce website explained in Section 7.1.2.

5.3. **Tools for the Implementation**

Here we introduce the tools we used in our implementation and the reason for selecting them.

**Apache Jmeter** Apache JMeter is an open-source performance testing java application. It can test performance on static and dynamic resources. Apache JMeter can simulate heavy loads on a server, group of servers, network or object to test and measure performance metrics of the system under different load types. It is written in Java, and it allows us to use its libraries for executing our desired workloads in the implementation of our approach, which is written in Java. Additionally, JMeter has a simple and user-friendly GUI, which helps us easily generate JMX files containing the basic configurations needed for the workloads generated and executed in our load tester.

**WordPress and WooCommerce** WordPress is a free and popular open-source content management system. It is written in PHP and paired with a MySQL database. We set up a website on WordPress as the SUT in the evaluation phase of our load testing approach. WordPress is very flexible and could be extended by using different plugins. WooCommerce is an open-source e-commerce plugin for WordPress to create and manage online stores. We use WooCommerce to turn the website into an e-commerce store.

**XAMPP** XAMPP is one of the most common desktop servers. It is a lightweight Apache distribution for deploying local web servers for testing purposes. We create the WordPress website (SUT) using XAMPP.
**RL4J** In order to avoid possible implementation errors in implementing the DQN in one of our proposed approaches for load testing, we use an open-source library RL4J [53]. RL4J is a deep reinforcement learning library that is a part of the Deeplearning4j project [54] and released under an Apache 2.0 open-source license. Eclipse Deeplearning4j is a deep learning project written in Java and Scala. It is open-source, and it is integrated with Hadoop and Apache Spark and could be used on distributed GPUs and CPUs. Deeplearning4j is compatible with all java virtual machine language e.g., Scala, Clojure, or Kotlin. It includes deep neural network implementations with lots of parameters to be set by the users when training a network [54]. RL4j contains libraries for implementing DQN (Deep Q-learning with double DQN) and Async RL (A3C, Async NStepQlearning).
6. Approach

In this section, we propose our approach for intelligent load test generation using reinforcement learning methods. We answer RQ1 here and present the mapping of the real-world problem to an RL problem. We provide the details of our approach and the learning procedure for generating load test. In section 6.1., we provide the mapping of the optimal load generation problem to an RL problem, how we define the environment and the RL elements in the problem. Then in section 6.2., we present the RL methods that we use in our approach, which are q-learning and DQN, we also present the operating workflow for each method.

6.1. Defining the environment and RL elements

In this section, we map the load test scenario elements to reinforcement learning elements and define the environment.

Agent and Environment  As mentioned before, the goal of the agent is to attain the optimal policy, which is to find the most efficient workloads for testing the system’s performance. For applying an RL-based approach to a problem, it is generally supposed that the environment is non-deterministic and also stationary upon transitions between the states of the system. The environment here is a server (the system under test) that is unknown to the agent. The agent interacts with the SUT continuously, and the only information that the agent knows about the SUT is gained by the agent’s observations from this interaction. The interactions are actions taken by the agent and the SUT’s responses to these actions in the form of observations for the agent. In other words, the actions that our agent takes affects the SUT as the environment, and the SUT returns metrics to the agent, which affects the agent’s next action.

States  We define the states according to performance metrics. Error rate and response time are two performance metrics in load testing. These two are considered as the agent’s observations of the environment. The two metrics define the agent’s state; the average error rate and average response time returned from the environment (SUT) after the agent took the last action. The terminal states are the states with average response time or average error rate higher than a threshold. The average error rate range is 0 to error_rate_threshold and the average response time range is 0 to response_time_threshold are divided into sections, each section determines one state.

Actions  The action that the agent takes in each step is increasing the workload and applying it to the SUT (environment). The workload is generated based on the policy and the workload in the previous action. The workload contains several transactions in which each transaction has a specific workload, i.e., a specific number of threads executes each transaction. A transaction consists of multiple requests. A single thread represents a user (client) running the transaction and sending requests to the server (SUT). The action space is discrete, and the set of actions is the same for all the states. Each action increases the last workload applied to the SUT by increasing the workload of exactly one of the transactions. The workload of a transaction is increased by multiplying the previous workload in a constant ratio. The definition of actions is shown in equation 17 and equation 18:

\[
\text{Actions} = \{ \cup action_k, 1 \leq k \leq |\text{Transactions}| \} \tag{17}
\]

\[
action_k = \{ \cup (W_{i}^{T_j}) \mid W_{i}^{T_j} = W_{n-1}^{T_j}, \text{if } j \neq k, \\
W_{i}^{T_j} = \alpha W_{n-1}^{T_j}, \text{if } j = k, \\
T_j \in \text{Transactions}, \\
1 \leq j \leq |\text{Transactions}| \} \tag{18}
\]
Golrokh Hamidi

Reinforcement Learning Assisted Load Test Generation

Where \( T_j \) indicates transaction number \( j \) among the set of transactions, \( t \) is the current learning time step (iteration), \( W^T_t \) is the workload of the transaction \( T_j \) at time step \( t \), and \( \alpha \) is the constant increasing ratio.

**Reward Function** The reward function takes an average error rate and average response time as input. The reward will increase as the average error rate and average response time increase. Consequently, the probability of the agent choosing actions which lead to a higher error rate and response time will increase. We define the reward function in equation 19.

\[
R_t = \left( \frac{RT_t}{RT_{\text{threshold}}} \right)^2 + \left( \frac{ER_t}{ER_{\text{threshold}}} \right)^2
\]  

(19)

Where \( R_t \) is the reward in time step \( t \), \( RT_t \) is the average response time and \( ER_t \) is the average error rate in time step \( t \). And \( RT_{\text{threshold}} \) and \( ER_{\text{threshold}} \) indicate the response time and error rate threshold.

### 6.2. Reinforcement Learning Method

In this section, we propose our RL solution to adaptive load test generation. We present our approach and explain the reinforcement learning algorithms that we chose for the approach, which are simple q-learning and DQN. We formulate the load test scenario in a reinforcement learning context and provide the architecture of our approach for each of the q-learning and DQN methods.

Algorithm 3 shows a general overview of the RL method. We use two methods q-learning and DQN for the learning phase in the Algorithm 3 explained in sections 6.2.1 and 6.2.2.

**Algorithm 3:** Adaptive Reinforcement Learning-Driven load Testing

| Required: \( S, A, \alpha, \gamma \);                |
| Initialize q-values, \( Q(s,a) = 0 \) \( \forall s \in S \), \( \forall a \in A \) and \( \epsilon = \nu , 0 < \nu < 1 \); |
| while Not (initial convergence reached) do        |
| Learning (with initial action selection strategy, e.g. \( \epsilon \)-greedy, initialized \( \epsilon \)); |
| end Store the learned policy;                     |
| Adapt the action selection strategy to transfer learning, i.e. tune parameter \( \epsilon \) in \( \epsilon \)-greedy; |
| while true do                                      |
| Learning with adapted strategy (e.g., new value of \( \epsilon \)); |
| end                                               |

#### 6.2.1 Q-Learning

As mentioned in section 2.2., q-learning is one of the basic reinforcement learning methods. Like other RL algorithms, q-learning seeks to find the policy which maximizes the total reward. The optimal policy here is extracted from the optimal q-function that is learned through the learning process by updating the q-table in each step. As mentioned before, q-tables store q-values, which get updated continuously. The q-value \( q_\pi(s,a) \) of a state-action shows how good is to take action \( a \) from state \( s \). In each step, the agent is in a state and can perform one of the available actions from that state. In q-learning, the agent will take action with the maximum q-value among the available actions. As mentioned in section 2.2.1, choosing the action with the maximum q-value would satisfy the exploitation criteria. However, we also have to take random actions to satisfy the exploration criteria and be able to experience the actions with lower q-values, which have not been chosen before (therefore, their q-value is not updated and is low). Consequently, we use the decaying \( \epsilon \)-greedy policy in which the \( \epsilon \) is big at the beginning of the learning and decays during the process. As mentioned before in Section 2.2.1, \( \epsilon \) is a number in the range of 0 to 1. In each step, the probability of selecting the best actions is \( 1-\epsilon \), and a random action is selected by the probability of \( \epsilon \). After the action selection, we will observe the environment (in our case...
the SUT), and we will detect the next state and compute the reward then update the q-table with a new q-value for the previous state and the taken action. The q-learning algorithm is shown in Algorithm 4 [10]:

**Algorithm 4: Q-learning (off-policy TD control) for estimating $\pi \approx \pi^*$**

- Algorithm parameter: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
- Initialise $Q(s, a)$, for all $s \in$ state space, $a \in A(s)$ arbitrarily except that $Q(terminal, .) = 0$

**for each episode do**

**for each step of episode do**

- Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
- Take action $A$, observe $R, S'$
- $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

- $S \leftarrow S'$

**end**

**end**

Figure 10 illustrates the learning procedure in our approach:

1. **Agent.** The purpose of the agent is to learn the optimum policy for generating load test scenarios that accomplish the objectives of load testing. The agent has four components: Policy, State Detection, Reward Computation, and Q-Table.

1.1. **Policy.** The policy which determines the next action is extracted from the Q-table based on the decaying $\epsilon$-greedy approach; in each step, one action is selected among the available actions in the current state. As mentioned before each action is: increasing the workload of one of the transactions by a constant ratio, then applying the total workload of all transactions on the SUT concurrently.

1.2. **State Detection** The state detection unit will detect the states based on the observations from the environment (i.e. SUT). The observations here are the error rate and response time. Each state is indicated by a range of average error rates and average response time. As Figure 11 shows,
we define six states, each one covering a specific range in error rate and response time. We divided the \([0, \text{error} \_ \text{rate} \_ \text{threshold}]\) range into two sections and the \([0, \text{response} \_ \text{time} \_ \text{threshold}]\) range into three sections.

![Diagram of States in the q-learning approach](image)

Figure 11: States in the q-learning approach

1.3 **Reward Computation** The reward computation unit takes the error rate and response time as an input and calculates the reward based on them.

1.4 **Q-Table** The q-table is where the q-values are stored. Each state-action has a q-value which will get updated by the gained reward after taking action from the state.

2 **SUT** The environment in our case is the SUT in which the actions would apply to it, and it would react to the actions (i.e., applied workload). Then the agent receives observations from SUT, which are error rate and response time, and determine the state and reward based on them.

6.2.2 Deep Q-Network

As mentioned in section 2.2., Deep Q-Network or DQN is an extension of q-learning. This method uses a function approximator instead of using a q-table. The function approximator, in this case, is a neural network. It approximates the q-values and refines this approximation (base on the rewards received each time after the agent taking action) instead of saving and retrieving the q-values from a q-table. Approximating q-values are beneficial when the state-action space is big. In this case, filling the q-table is not feasible and takes a long time. The benefit of using DQN is that it speeds up the learning process because 1) There is no need to store a big amount of data in the memory when the problem contains a large number of states and actions, 2) There is no need to learn the q-value of every single state-action and the learned q-values are generalized from the visited state-actions to the unvisited ones.

There are many function approximators (e.g., Linear combination of features, Neural Networks, Decision Tree, Nearest neighbor, Fourier/wavelet bases). Among the function approximators, neural networks are one of the function approximators which use gradient descent. Gradient descent is suitable for our data, which is not iid (Independent and Identically Distributed). The data is not iid because unlike supervised learning, in reinforcement learning values of the states near each other or the q-values of the state-action near each other are probably similar and the previous state is highly correlated with the previous state.

The DQN that we chose in our approach uses an ANN which takes a state as input and estimates the q-values of all the actions available from that state (Figure 12).

The architecture of our intelligent load runner approach with the DQN method is shown in Figure 13. The approach is the same as the q-learning approach except that it contains a \(^1\)\(^4\) DQN unit instead of the Q-Table unit. Also, in this approach, each state corresponds to a single response time,
and error rate, and thus the number of states is equal to $\text{error\_rate\_threshold} \times \text{response\_time\_threshold}$. In each iteration, after receiving the reward, the DQN gets updated then the policy unit chooses an action based on the actions’ q-value approximated by the DQN unit.
7. Evaluation

In this section, we explain the implementation setup of our proposed RL approaches for load test scenario generation, which answers RQ2. We explain the preparations for executing the implementation, i.e., the setup for the SUT, and also the procedure of how our experiment was setup. In addition, we evaluate our RL approaches by comparing them against a baseline and a random approach for generating the load tests in an experiment.

7.1. System Under Test Setup

Page loading time is an important factor in website’s user experience, and page delays can result in big sale loss in e-commerce stores (online shops). Page loading time is dependent on performance requirements. Therefore performance requirements play a key role in e-commerce stores. We intend to use an open-source e-commerce store as a SUT to apply our proposed load testing method on it. Note that the e-commerce application is already being used in production by many users and in the real-world. Using an e-commerce store as the SUT makes it possible to send a variety of requests to the website as the workload. Requests like registering, logging in, visiting product pages, buying products online, etc. We cannot apply our approach to a running e-commerce store that provides real services to customers because load testing on the website will affect the website’s performance and result in real sales loss. Therefore we will build our own e-commerce store.

In this section, we explain the implementation of the system under test in detail. We use XAMPP to deploy the SUT server and build a local e-commerce website using WordPress and WooCommerce.

7.1.1 Server Setup

We deployed the SUT on a local server on a computer dedicated to this mean. We used a local server to avoid load testing through proxies. Otherwise, we will end up load testing the proxy server too, and the proxy may fail before the SUT server. Using a local server we can avoid possible effects of any in-between network equipment or server which may influence the test results.

We deployed the SUT server using the XAMPP application on Ubuntu 16.04 operating system. As mentioned before XAMPP is a lightweight Apache distribution for deploying local web servers for testing purposes.

Figure 14: XAMPP application

To allocate our desired amount of resources to the SUT server, we use cgroups. Cgroups or control groups, are a feature in Linux kernel that makes it possible for the user to allocate resources
e.g., CPU time, system memory, network bandwidth, or combinations of these resources among the collection of processes running on a system, and manage and put restrictions on the resources.

1. We create a cgroup named "rlsutgroup" in the /etc/cgconfig.conf file:

```bash
group rlsutgroup {
    cpuset {
        cpuset.cpus = 0;
        cpuset.mems = 0;
    }
    memory{
        memory.limit_in_bytes = 2G;
    }
}
```

We specify CPU number 0 and the memory node 0 to be accessed by the cgroup. We also set the maximum amount of user memory (including file cache) to 2 gigabytes (GB) for the cgroup.

2. To move the SUT server process to the cgroup we created, we write the line below in /etc/cgrules.conf file:

```
*: /opt/lampp/manager-linux-x64.run cpuset, memory rlsutgroup
```

Where /opt/lampp/manager-linux-x64.run is the command for starting the Xampp server.

3. To apply changes in cgconfig.conf and cgrules.conf we enter the commands below in ubuntu terminal:

```
sudo cgconfigparser -l /etc/cgconfig.conf
sudo cgrulesendd
```

### 7.1.2 Website Setup

We set up an e-commerce website (Figure 15) on WordPress using WooComerce. As mentioned before, WordPress is an open-source content management system, and WooComerce is an e-commerce plugin for WordPress to create and manage online stores and is being used by millions of users across the globe. A client can view products on the website, register or login to the website, add products to her cart, and checkout and order the products in her cart using PayPal or other options.

### 7.2. Implementation

In this section, we will first explain the structure of a workload and how they are generated using JMeter. Then we will provide the implementation details of our RL load tester.

#### 7.2.1 Workload Generation

We use Apache JMeter as a load generation/execution tool in our implementation. As mentioned before, JMeter is a testing tool for generating and applying workload on servers. It generates the recommended workload by the tester agent, applies it on the SUT and, captures and measures the performance metrics of the SUT.

In each load test scenario, we execute a workload with eleven number of different transactions, in which each transaction has a specific size of workload executed by the JMeter threads. Each thread in JMeter indicates one user, and it is responsible for sending HTTP requests of one transaction to the SUT. While applying a workload on the SUT, for each transaction we will generate a number of JMeter threads equal to the size of that transactions specific workload (which is a variable we change during the workload generation process), and we will execute all the threads of every transaction in parallel in a specific ramp-up time.
TRANSACTIONS

We considered eleven different operations in the SUT shown in Table 2 for generating workloads. A transaction is an operation and may have some functional prerequisite transactions. When a transaction is executed in the test, all of its prerequisite transactions would be executed sequentially in the specific order before the execution of the main transaction. Table 3 shows the prerequisite transactions for each transaction. Each thread is responsible for executing a transaction and its prerequisite transactions sequentially. Nevertheless, all threads are executed in parallel.

Table 2: Common operations in an online shop

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
<td>Access to home page</td>
</tr>
<tr>
<td>Sign up page</td>
<td>Access to Sign up page</td>
</tr>
<tr>
<td>Sign up</td>
<td>Register and add a new user</td>
</tr>
<tr>
<td>Login page</td>
<td>Access to login page</td>
</tr>
<tr>
<td>Login</td>
<td>Sign in at the system</td>
</tr>
<tr>
<td>Search page</td>
<td>Access to search page</td>
</tr>
<tr>
<td>Select product</td>
<td>See the details of the selected product</td>
</tr>
<tr>
<td>Add to cart</td>
<td>Add the selected product to the cart</td>
</tr>
<tr>
<td>Payment</td>
<td>Access to payment page</td>
</tr>
<tr>
<td>Confirm</td>
<td>Confirm the order (payment)</td>
</tr>
<tr>
<td>Log out</td>
<td>Log out</td>
</tr>
</tbody>
</table>

JMeter Configuration

We used apache-jmeter-5.2.1 for applying the workload on the SUT. JMeter could be run in a GUI mode or CLI (command line) mode. JMeter Test Plans can also be created and executed through a java program.

A JMeter projects could be saved in a JMX file in the XML format. JMX or Java Management Extension is a standard framework for managing applications in java. It could be defined how to start, monitor, manage, and stop software components in a JMX file.
Generating the Test Plan  We use the GUI mode to generate a test; we setup JMeter to record user activities browsing the SUT. We do each of the transactions in table 3 step by step and record the requests sent to the SUT using the JMeter.

Each test consists of some elements. All tests in JMeter should contain a Test Plan and Thread Groups. The steps for setting the elements are (Figure 16):

1. Set the Test Plan element: In the Test Plan element, we check the option "Run tearDown Thread Groups after the shutdown of main threads"

2. Add Thread Group elements: In the Test Plan, we create a Thread Group element for one of the transactions:
   right click on the Test Plan → Add → Threads (Users) → Thread Group
   Each thread in a Thread Group simulates a user that sends requests to a server.

3. Add HTTP Request Defaults element: HTTP Request Defaults is another JMeter element. The user can set default values for HTTP Request Samplers using HTTP Request Defaults. We add a HTTP Request Defaults to the Thread Group:
   right click on the Thread Group → Add → Config Element → HTTP Request Defaults

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Prerequisite Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
<td>home page</td>
</tr>
<tr>
<td>Sign up page</td>
<td>home page → my account page</td>
</tr>
<tr>
<td>Sign up</td>
<td>home page → my account page → register</td>
</tr>
<tr>
<td>Login page</td>
<td>home page → my account page</td>
</tr>
<tr>
<td>Login</td>
<td>home page → my account page → login</td>
</tr>
<tr>
<td>Search page</td>
<td>home page</td>
</tr>
<tr>
<td>Select product</td>
<td>home page → select product</td>
</tr>
<tr>
<td>Add to cart</td>
<td>select product → add to cart</td>
</tr>
<tr>
<td>Payment</td>
<td>select product → add to cart → checkout</td>
</tr>
<tr>
<td>Confirm</td>
<td>select product → add to cart → checkout → PayPal page</td>
</tr>
<tr>
<td>Log out</td>
<td>my account page → logout</td>
</tr>
</tbody>
</table>
We enter the website name under test in the "Server Name or IP" field in the HTTP Request Defaults control panel. In the "Timeout" box we set the "Connect" timeout to 30000 ms and the "Response" timeout to 120000 ms.

4. Add Recording controller: JMeter can record the user activity and store it in a Recording controller. We add the Recording controller to the Thread Group:

right click on the Thread Group → Add → Logic Controller → Recording Controller

5. Add HTTP(S) Test Script Recorder: HTTP(S) Test Script Recorder can record all the requests sent to a server. We add this element to the Test Plan:

right click on the Test Plan → Add → Non-Test Elements → HTTP(S) Test Script Recorder

We set the "Target Controller" field to "Test Plan > Thread Group" where the recorded scripts will be added.

After building the Test Plan;

1. We change the proxy configuration of the browser and set the "HTTP Proxy" to "localhost" and the "Port" to the same port number in the HTTP(S) Test Script Recorder.

2. Click the "Start" button in the HTTP(S) Test Script Recorder panel.

3. Do the transaction step by step using the browser.

4. When the transaction is finished, we click the "Stop" button in the HTTP(S) Test Script Recorder panel.

5. Save the JMeter project as a JMX file.

We do all the steps above for each transaction in table 3, and in the end, we integrate all of the thread groups in a single JMX file (Figure 17) to be used for applying the workload by the agent. When executing the final JMX file, all the Thread Groups will start the same time and will execute concurrently.

Figure 17: JMeter Thread Groups
Executing the Test Plan  We do not use JMeter GUI or CLI for executing the generated Test Plan. Instead, the Test Plan is executed from java code. The java program is the implementation of the RL load test generation wherein each step of the agent’s learning process, the Test Plan is executed.

To run the JMeter Test Plan, first we increase the JMeter heap size to be able to generate larger workloads described as below:

- Go to the apache-jmeter-5.2.1/bin directory
- Open JMeter startup script
- Find the line HEAP=”-Xms1g -Xmx1g”
- Change the maximum value to -Xmx4g

We load the JMX file in the java program by importing Apache JMeter packages:

```java
testPlanTree = SaveService.loadTree(new File("myTest.jmx"));
```

And reset the Thread Group parameters for each step of applying the workload after an action is taken by the agent (i.e., one of the transactions workload is increased). There are three parameters to set for each Thread Group:

- number of threads: The number of threads (workload) in the Thread Group.
- number of loops: The number of times that the Thread Group is executed.
- ramp-up time: the time that takes for all threads of the Thread Group to get up and running.

For each Thread Group, we set the number of threads equal to the workload of that transaction in the RL agent, and the ramp-up time equal to the workload divided by a ratio "threadPerSecond" (which we set it equal to 10). The number of loops is set to 1 for all Thread Groups.

```java
threadGroup.setNumThreads(transaction[i].workLoad);
threadGroup.setRampUp(transaction[i].workLoad/threadPerSecond);
((LoopController) threadGroup.getSamplerController()).setLoops(1);
```

Then we run the Test Plan:

```java
jmeter.run();
```

To be able to apply a large number of concurrent requests, we should execute the program on a device with high memory/CPU resources. Table 4 shows the properties of the device we used.

Table 4: Hardware Overview of the Machine used for executing Load Generation Client

<table>
<thead>
<tr>
<th>Model Name</th>
<th>MacBook Pro</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Identifier</td>
<td>MacBookPro12,1</td>
</tr>
<tr>
<td>Processor Name</td>
<td>Dual-Core Intel Core i7</td>
</tr>
<tr>
<td>Processor Speed</td>
<td>3.1 GHz</td>
</tr>
<tr>
<td>Number of Processors</td>
<td>1</td>
</tr>
<tr>
<td>Total Number of Cores</td>
<td>2</td>
</tr>
<tr>
<td>L2 Cache (per Core)</td>
<td>256 KB</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>4 MB</td>
</tr>
<tr>
<td>Hyper-Threading Technology</td>
<td>Enabled</td>
</tr>
<tr>
<td>Memory</td>
<td>16 GB</td>
</tr>
</tbody>
</table>
7.2.2 Q-Learning Implementation

We implemented the agent in a java program based on the approach explained in Section 6.2.1. As shown in Figure 18, we have a module for each of Q-Table, Policy (action selection), Reward Computation, and State Detection. How the modules communicate with each other is explained in Section 7.3..

![Q-learning implementation architecture of Intelligent Load Runner](image)

7.2.3 DQN Implementation

We use the library RL4J [53] in our DQN approach implementation. As mentioned before, RL4J is a deep reinforcement learning library that contains libraries for implementing DQN (Deep Q-learning with double DQN).

**Prerequisites for deeplearning4j**

Prerequisites:

- Java: JDK 1.7 or later should be installed. (Only 64-Bit versions are supported)
- Apache Maven: Maven is a dependency manager for Java application.
- IntelliJ or Eclipse: IntelliJ and Eclipse are Integrated Development Environments (IDE) that help to work with deeplearning4j and configuring neural networks easier. IntelliJ is recommended for using the deeplearning4j library.
- Git: to clone deeplearning4j examples.

**Configuring and training a DQN agent using RL4J**

1. Create an action space for the mission:

   ```java
   DiscreteSpace actionSpace = new DiscreteSpace(numberOfTransactions);
   ```

2. Create an observation space for the mission:
3. Create an MDP wrapper:

```java
SUTObservationSpace observationSpace = new SUTObservationSpace(maxResponseTimeThreshold, maxErrorRateThreshold);
```

4. Create a DQN:

```java
public static DQNFactoryStdDense.Configuration LOAD_TEST_NET = DQNFactoryStdDense.Configuration.builder().l2(0.01).updater(new Adam(learningRate)).numLayer(3).numHiddenNodes(16).build();
```

5. Create a Q-learning configuration by specifying hyperparameters:

```java
public static QLearning.QLConfiguration LOAD_TEST_QL = new QLearning.QLConfiguration(seed, maxEpochStep, maxStep, expRepMaxSize, batchSize, targetDqnUpdateFreq, updateStart, rewardFactor, gamma, errorClamp, minEpsilon, epsilonNbStep, doubleDQN);
```

6. Create the DQN:

```java
Learning<QualityMeasures, Integer, DiscreteSpace, IDQN> dql = new QLearningDiscreteDense<QualityMeasures>(mdp, LOAD_TEST_NET, LOAD_TEST_QL, manager);
```

7. Train the DQN

```java
dql.train();
```

**Q-learning hyperparameters of the DQN**  The Q-learning configuration hyperparameters are [55, 56]:

- **maxEpochStep**: Each epoch is equivalent to an episode in the learning algorithm. `maxEpochStep` is the maximum number of steps allowed in each episode (epoch).

- **maxStep**: The maximum number of total iterations (the summation of steps in all episodes) in the learning. Training will finish when the number of steps exceeds `maxStep`.

- **expRepMaxSize**: The maximum size of experience replay. The number of past transitions that the agent will take the next action based on them is experience replay. Experience replay is explained in detail in section 2.

- **batchSize**: The number of steps which the neural network would update its weights after executing them. We choose the batch size equal to 1 because each sample in RL is dependent on the previous sample, so the network should be updated per sample (in our case, each learning step).

- **targetDqnUpdateFreq**: In double DQN the target network is frozen for `targetDqnUpdateFreq` number of steps and it would update after `targetDqnUpdateFreq` steps from the online network. The state-action values are computed (the evaluation) based on the target network to stabilize the learning.
- **updateStart**: The number of no-operation (do nothing) moves before starting the learning to make the learning start with a random configuration. The agent will conduct the same sequence of actions at the beginning of each episode instead of learning to take the next action based on the current state if it starts with the same configuration each time.

- **rewardFactor**: Reward factor is an important hyperparameter that should be considered carefully. It significantly affects the efficiency of learning. This factor scales the rewards, so the Q-values will be lower (if the range is [-1; 1] it is similar to normalization).

- **gamma**: The discount factor.

- **errorClamp**: This parameter will clip (bound between two limit values) the loss function (TD-error) based on the output in the backpropagation. For example if \( \text{errorClamp} = 1 \), then the gradient is bounded to the range (-1,1).

- **minEpsilon**: Epsilon is the derivative of the loss function with respect to the activation function’s output. The epsilon is used to compute the gradients for every activation node in backpropagation.

- **epsilonNbStep**: After \( \text{epsilonNbStep} \) number of steps, the epsilon will be decreased to \( \text{minEpsilon} \).

- **doubleDQN**: This value should be set to \text{True} to enable double DQN.

We set the values of the hyperparameters as shown in Table 6.

![Figure 19: DQN implementation architecture of Intelligent Load Runner](image)

### 7.3. Experiment Procedure

We separately executed our RL approaches for load generation (shown in Figure 18 and Figure 19) on the SUT. We also executed a baseline load generation approach and a random load generation approach on the SUT to evaluate the efficiency of the proposed RL approach against them. All approaches were executed in several episodes, each episode consisting of several steps. In the baseline approach, in each step, the workload size of all transactions was incremented. We chose...
a baseline approach to compare the final size of the generated workload (that hit the error rate or response time thresholds) of the baseline approach with our proposed RL approaches. In the random approach, in each step, a random transaction was chosen, and the size of its workload was increased (unlike the RL approaches where the transaction was selected based on the policy). The reason behind choosing a random approach was that random testing is found robust [57], [58] among many other systematic testing approaches and is a good criterion.

The SUT was deployed on a local server, an ASUS K46 computer with a Ubuntu 16.04 operating system, dedicated 1 CPU, and 2 GB memory to the SUT (as mentioned in section 7.1..1). During the execution of the methods, the system was logging necessary data for the evaluation metrics. An overview of the procedure is shown in Figure 20. We further explain the evaluation metrics used and the procedure for executing each approach.

**Evaluation metrics** The evaluation metrics are the average error rate, average response time, size of the final effective workload, and number of steps for generating the effective workload. We can not conclude from a quick response time that the SUT is operating fine and fast, and we should consider the error rate too. Since the servers are quick at delivering error pages, we may get low response times with high error rates in some situations. Consequently, not only we put a threshold on the response time, but also we put a threshold on the error rate.

**General configuration** We execute each approach for about 40 episodes. Each episode consisted of several steps. In each step, a workload was generated and applied to the SUT. The workload, response time, and error rate were logged for each step. The episodes continued until the observed error rate and response time hit the threshold. Every two continuous episodes were executed with a 5-minute delay between them, which allowed the server to go back to its normal state. Table 5 shows the configuration values used for all the approaches.

**Table 5: Load Tester Configuration Values**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>average response time threshold</td>
<td>1500ms</td>
</tr>
<tr>
<td>average error rate threshold</td>
<td>0.2</td>
</tr>
<tr>
<td>delay between executing two continuous episodes</td>
<td>5</td>
</tr>
<tr>
<td>number of started threads per second</td>
<td>10</td>
</tr>
<tr>
<td>initial workload per transaction</td>
<td>3</td>
</tr>
<tr>
<td>transaction workload increasing step ration</td>
<td>1/3</td>
</tr>
</tbody>
</table>
Baseline approach We executed the baseline approach for 40 episodes. In each step of an episode in this approach:

1. The workload of all transitions were increased by $1/3$ of their current workload.

2. The Test Plan was loaded from the JMX file, and after setting the new values for each transaction workload, the Test Plan was executed, and the workload was applied on the SUT.

Random approach We executed the random approach for 40 episodes. In each step of an episode in this approach:

1. A transition was chosen randomly, and its workload was increased by $1/3$ its current workload.

2. The Test Plan was loaded from the JMX file, and after setting the new values for each transaction workload, the Test Plan was executed, and the workload was applied on the SUT.

Q-learning and DQN approaches We executed the q-learning approach for 40 episodes. However, the DQN approach was executed for 47 episodes. The reason behind this is that the number of episodes was not configurable in the DQN implemented using the RL4J library, and instead, the number of steps was a configurable parameter. Therefore we configured the number of steps equal to a value (shown in table 6) that we approximated to be executed in around 40 episodes. In each episode, the agent started from an initial state, which was detected by applying an initial workload on the SUT and observing the average error rate and response time. The initial q-values in the q-table/q-network were set to 0. Each episode consisted of several learning steps. In each learning step:

1. An action was chosen according to the policy; one of the transactions workload would increase by $1/3$ its current workload.

2. The Test Plan was loaded from the JMX file, and after setting the new values for each transaction workload, the Test Plan was executed, and the workload was applied on the SUT.

3. Based on the observations (average error rate and average response time), the new reward and new state were detected, and the q-table or q-network got updated.

In the q-learning approach, we set the learning rate and discount factor both equal to 0.5, which are the q-learning attributes. In the DQN approach, we set the values of the hyperparameters, as shown in Table 6.
Table 6: Hyperparameters Configuration for DQN

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxEpochStep</td>
<td>30</td>
</tr>
<tr>
<td>maxStep</td>
<td>450</td>
</tr>
<tr>
<td>expRepMaxSize</td>
<td>450</td>
</tr>
<tr>
<td>batchSize</td>
<td>1</td>
</tr>
<tr>
<td>targetDqnUpdateFreq</td>
<td>10</td>
</tr>
<tr>
<td>updateStart</td>
<td>1</td>
</tr>
<tr>
<td>rewardFactor</td>
<td>0.1</td>
</tr>
<tr>
<td>gamma</td>
<td>0.5</td>
</tr>
<tr>
<td>errorClamp</td>
<td>10.0</td>
</tr>
<tr>
<td>minEpsilon</td>
<td>0.1f</td>
</tr>
<tr>
<td>epsilonNbStep</td>
<td>400</td>
</tr>
<tr>
<td>doubleDQN</td>
<td>true</td>
</tr>
</tbody>
</table>

8. Results

This section presents the results of the experiment conducted to evaluate the efficiency of the baseline, random, q-learning, and DQN approaches. This section is focused on answering RQ3.

Results of the Baseline Approach The baseline approach was executed for 40 episodes. In Figure 21a, the episodes are plotted on X-Axis, and the Y-Axis shows the number of steps in each episode that is needed for generating the workload that hit the response time or error rate threshold. As it can be seen from Figure 21a, the trend line for the baseline approach stays between zero to five. It means that the baseline approach took fewer steps in generating the final workload that hits the thresholds. However, the increment of the workload size in each step was very high because it was applied to all transactions (as shown in Figure 21b).

In Figure 21b, the episodes of baseline approach are shown on the X-Axis, and the size of the final workload that hits the response time or error rate threshold is shown on the Y-Axis. As it can be seen in Figure 21b, the trend line for the baseline approach consistently stays between 50 to 60. This means that in the majority of the episodes, the baseline approach was mostly able to hit the threshold with bigger workload sizes.

Results of the Random Approach The random approach was executed for 40 episodes. Figure 22a shows that in the random approach, the number of steps for increasing the workload is high. The trend line is between 10 and 15 steps, and it is constant (as expected from a random
method). It indicates that on average, no extreme change happened overtime in the state of the system and that the SUT remains stable.

As shown in Figure 22b, in the random approach, the size of the final workload, which hit the thresholds in each episode, is generally between 40 and 50. This size is smaller than the general size of the final workload in the baseline approach because here, the increments in the workload size were applied to just one transaction per step and not all of them. Thus this approach could find smaller workloads that hit the threshold.

![Figure 22: Random Approach Results of the Q-learning Approach](image)

The q-learning approach was executed for 40 episodes. Figure 23a shows that the number of steps in each episode is, on average, between 5 and 15 steps. We can see that the diversity of the number of steps is high in the first episodes, which is similar to the results of the random approach as shown in Figure 22a. But the diversity decreases over time, and after episode 25 i.e., the last 16 episodes, the number of steps converge and appear in the range of 8 to 10. This indicates that the agent has developed a policy, and it is following it. The policy that has been learned and converged over time and is not changing drastically after episode 25.

Another practical factor of the convergence in this approach is because of using the decaying $\varepsilon$-greedy method. However, the q-learning method is expected to converge to the optimal policy with the probability of one [59], using decaying $\varepsilon$-greedy can accelerate the convergence. As mentioned in Section 6.2.1, at the beginning of the learning, the actions are chosen more randomly to allow the agent to explore different actions and learn the consequences of taking them (by receiving the reward from taking that action). But over the time, the probability of choosing random actions decreases, also the probability of choosing the best action due to the main policy increases (policy derived from the q-table). Therefore, in each step, the actions (i.e., the transactions that their workloads are incremented) are chosen more randomly at the beginning of the learning (first episodes), but they are chosen less randomly and more intelligently based on the learned policy in the last episodes. In the final episodes, the process of generating the workload is following the same policy which leads to convergence of the number of steps in each episode.

In addition, the trendline in Figure 23a decreases over the time, which means that the number of steps for generating a workload that hits the thresholds, is decreasing. This indicates that the agent has learned to take more efficient actions and to choose the best candidate transitions to increment its workload in each step. Choosing the best actions intelligently in the last episodes leads to generating an effective workload in fewer steps. The convergence and degradation in the number of steps shows that the agent has found the optimal policy for generating effective workload which hits the threshold in fewer steps.

Figure 23b shows that in the q-learning approach, the size of the final workload in each episode is small, and it is, on average, between 40 and 50. The diversity of the final workload size is high in
the first 20 episodes, but it decreases over time and converges to the range of 42 to 46 after episode 22 (i.e., last 19 episodes). Also, the trendline shows, the final workload size decreases during the time.

![Graph of Q-learning Approach](image1)

Figure 23: Q-learning Approach

**Results of the DQN Approach** The DQN approach was executed for 47 episodes. As shown in Figure 24a, The number of steps are between 5 and 15. After episode 38 i.e., the last 10 episodes, the number of steps are staying in the range of 6 to 8. Like the q-learning approach and because of the same reason (i.e., using decaying \( \varepsilon \)-greedy method in the policy), the number of steps are converging in the last episodes. The steps are staying in the range of 6 to 8 after episode 38. Also, the slope of the trend line show that the number of steps for generating the effective workload decreases over time. Based on the convergence and the decrease in the number of steps, we can see that the agent has learned the optimal policy, and it is following that policy. We can also see that in comparison to the q-learning approach, the convergence is occurring after more episodes, but the convergence range is less than the convergence range in the q-learning approach.

In Figure 24b, the trendline shows that the final workload is generally between 40 and 50. We can see in the figure that after episode 40 i.e., the last 8 episodes, the size of the effective workload is altering in the range of 40 to 43. This shows that it is converging after episode 40, and the size of the final workload has reduced after this episode. Comparing to the q-learning method, the convergence and the decrease in the size of effective workload are happening in later episodes. But the convergence range is less than the range in the q-learning method (which is 8 to 10).

![Graph of DQN Approach](image2)

Figure 24: DQN Approach
9. Discussion

In this section, we highlight the answers to our research questions. We review the solutions and discuss the results (important bits in italic). We also discuss the threats to the validity of our results.

**RQ1.** Answering RQ1 required formulating an RL solution for the test load generation problem. In order to apply RL based solution, a mapping of the real-world problem into an RL problem is a pre-requisite. We formulated and presented our mapping in Section 6, and provided our RL approach for load test generation in detail. We defined the agent, environment, and the learning principles including states, actions, observations, and reward function in our RL problem. As discussed, we considered the SUT as the environment. We declared each state base on the last average error rate and average response time observed from the SUT. The actions were defined as choosing a transaction, increasing its workload, and applying it to the SUT. The average error rate and average response time were the agent’s observations of the SUT. And the reward was calculated according to the observations of the SUT. We also chose two different RL methods for in our approach namely q-learning and DQN.

**RQ2.** To demonstrate the applicability of our proposed RL-based load testing approaches, which is RQ2, we implemented and applied the q-learning and DQN approaches on an SUT in Section 7. We set up an e-commerce store on a local server using open-source and heavily maintained CMS and plugin i.e. WordPress and the WooCommerce plugin. We used JMeter for generating the load scenarios (produced by the RL agent), applying them on the SUT, and recording the error rate and response time. Results show that RL-based test load generation approaches are applicable for performance testing of real-world applications.

**RQ3.** To answer the third research question, we conducted an experiment. In the experiment, we executed four approaches (treatments) for load test generation, which were a baseline, a random, and the proposed q-learning and DQN approaches. The results of the experiment are provided in Section 8. Results from our experiment show that the baseline approach (in which the workload was increased per all transactions in each step) generally produced a larger effective workload. Thus we can conclude that, in comparison to other approaches, the baseline approach for workload generation is not efficient in terms of the size of generated workload.

In contrast, the random approach did perform better than the baseline approach. In comparison with RL-based approaches, the random approach was generally not attaining an optimal workload size, and the diversity of the applied workload sizes remained high.

The results show that the number of steps per episode and the size of the effective workload in each episode of both of the RL approaches are converging to a lower number in the last episodes. Results indicate that the q-learning approach converges faster (in terms of number of steps and optimal workload size) than the DQN approach. This is expected since the q-learning approach only has six states while the DQN approach has much more states (error_rate_threshold×response_time_threshold states).

Findings also revealed that the DQN approach converges to lower values in both metrics. Meaning that in comparison to q-learning-based approach, the DQN approach took more time to converge. However, the DQN was more efficient in terms of finding the optimal workload sizes. Based on our results, we believe that the DQN approach can perform even better after extensive tuning of the hyper-parameters. Finally, we can conclude that both of the proposed RL approaches for load test generation converges to the optimal policy and performs better than the baseline and random approach.

9.1. Threats to Validity

Load scenario generation is heavily dependent on the hardware of the SUT and its execution environment. External factors (such as running the SUT on a shared hosting server) might alter
the results. In this section, we address the potential threats to validity based on the classification presented by Runeson and Höst [60].

Construct validity: This aspect of validity reflects to what extent the studied operational measures really represent what the researcher has in mind. A misunderstood question is an example of a potential construct validity threat. To tackle potential threats to our construct validity, we have benefited the guides from multiple researchers in the problem formulation and used well-established guidelines for conducting our study.

Internal validity: This aspect of validity is of concern with the validity and credibility of the obtained results. We tackle the potential threats to internal validity by executing the experiment on a dedicated local server with no other processes running on it. However, there were several uncontrolled factors (such as the operating system’s processes) that might have affected our results.

External validity: This aspect of validity is concerned with to what extent it is possible to generalize the findings, and to what extent the findings are of interest to other people outside the investigated case. In our case, the results are obtained by executing the experiment on one SUT, and we, therefore, do not claim that the results can be generalized to other cases. However, we chose open-source and heavily maintained SUT, so that the results can be generalized to other similar cases (WooCommerce-based e-commerce applications). In addition, our results can also be of interest to performance testing researchers and practitioners working with load testing via JMeter.

Reliability: This aspect is concerned with to what extent the data and the analysis are dependent on the specific researchers. Hypothetically, if another researcher, later on, conducted the same study, the result should be the same. Threats to this aspect of validity are tackled by receiving feedback from multiple researchers in experiment planning and execution. In addition, we provided enough details on our experiment setup for replication.
10. Conclusions

One critical activity in performance testing is the generation of load tests. Existing load testing approaches heavily rely on the system models or source code. This thesis aims to propose and evaluate a model-free and an intelligent approach for load test generation.

In this thesis, we formulated the problem of efficient test generation for load testing into an RL problem. We presented an RL-driven model-free approach for generating effective workloads in performance testing. We mapped the real-world problem of test load generation into the RL context and discussed our approach in detail. We evaluate the applicability of our proposed RL based test load generation on a real-world software system. In addition, we conducted an experiment to compare the efficiency of two different proposed RL-based, q-learning, and DQN methods for effective workload generation. For the experiment, we implemented our proposed approach and prepared the requirements for testing it. We set up an e-commerce store on a local server using WordPress and its Woocommerce plugin. Then we applied the workloads generated by our approach to it. The workloads consisted of a different number of various transactions (operations) generated using JMeter. We performed a one factor-four treatment experiment on the same SUT where the factor for the experiment was “test load generation method” with treatments of baseline method, a random method, the q-learning method, and the DQN method. We executed each treatment for around 40 episodes. Each episode contained several load generation steps, and it would be finished when the generated workload produced an error rate or response time larger than the defined threshold.

The results indicated that, in general, the baseline approach was not efficient in terms of the size of the generated effective workload per episode. In the random approach, the average size of generated effective workload was smaller than the baseline approach, which means that the random approach performed better than the baseline approach in test load generation. In addition, the results showed that both of the RL-based methods performed better than the random and baseline approaches. The results show that effective workload size and the number of steps taken for generating workload in each episode converges to a lower value in both q-learning and DQN approaches. The q-learning approach converged faster than the DQN. However, the DQN approach converged to lower values for the workload sizes. We can conclude from the results that both of the proposed RL approaches learned an optimal policy to generate optimal workloads efficiently.

The RL-based approaches performed better in our experiment and do not require access to the system models or source code. In addition, the learned policy can be reused in further similar situations (stages) of testing, e.g., regression testing and in the process of Development and Operations (DevOps) incremental release testing.

In the future, we plan to extend our approach to support performance testing for software product lines. In software product lines, the derived products are variants of the standard product, and the learned policy for load generation can be reused on multiple derived products. This can significantly reduce the performance testing time for the software product line.
References


Golrokh Hamidi

Reinforcement Learning Assisted Load Test Generation


