Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science & Statistics

Software Testing of Reinforcement Learning Agents

Christiana Popoola

Supervisor: Associate Prof. Ivana Dusparic

April 15, 2024

A dissertation submitted in partial fulfilment
of the requirements for the degree of
MSci (Computer Science)

Declaration

Abstract

The field of Reinforcement Learning, and the use of Reinforcement Learning agents has been growing increasingly in popularity in the most recent years. Despite this, there is still an insufficient number of established testing techniques which are designed for Reinforcement Learning problems. There are currently a number of proposals to address this gap, but many are still insufficiently tested, and not widely used.

In the thesis, we aim to adapt an existing proposed technique - Mutation Testing - which was originally adapted from its general use in traditional Supervised Learning, for application on Reinforcement Learning agents. We will be further adapting this approach to work on a wider range of algorithms and environments, in this field. The existing work, can currently only facilitate discrete action space-based environments, and three algorithms: Deep Q-Network, Proximal Policy Optimization, and Advantage Actor Critic. Therefore, we will be expanding on this, with particular focus on the implementing the Deep Deterministic Policy Gradient algorithm, and the CartPoleContinuous environment - a continuous action space.

This implementation will then be assessed using two evaluation metrics: AVG and R, which were designed specifically for Mutation Testing on Reinforcement Learning problems. The results will be analysed under the evaluation objectives: the effectiveness of Mutation Testing overall, the effectiveness of Mutation Testing on a continuous environment in comparison to a discrete environment, and finally the effectiveness of the evaluation metrics being utilised.

We find that the proposed technique - Mutation Testing - is effective, dependent on the mutation introduced to system (agent, environment, policy). We found that some mutants had more of a significant impact on the system, than others, which was evidenced by certain mutants which were consistently detected and killed, and others that remained undetected. We find that based on the results, the effectiveness of Mutation Testing is virtually the same on both continuous, and discrete action space-based environments. Finally, we conclude that both evaluation metrics proved to be useful and effective, and when used in conjunction with one another, produce convincing and reliable conclusions.

Acknowledgements

I would first like, to thank God for all He has done throughout my academic career, I wouldn't have made it to this year without Him, and I owe all my accomplishments to His faithfulness and love to me. I could do this all because You strengthened me.

I would like to thank Ivana Dusparic, who was with me every step of the way. Your guidance and expertise were extremely helpful, thank you.

I want thank my family for their support throughout this year especially. Special mention to my sister Mary, brother-by-law Isaac, and my dearest Captain Levi.

Finally, I would like to thank the person I spent majority of my academic career with, Tumi for listening to all my ranting about Reinforcement Learning, and also a shout-out to the Oxford Scholar, who provided me with further insight and advice.

Thank you all :)

Contents

List of Figures

List of Tables

1 Introduction

This thesis focusing on expanding upon an existing proposed testing technique for Reinforcement Learning agents. The chosen technique is Mutation Testing, and the aim is to facilitate the testing of a wider range of Reinforcement Learning algorithms, as well as continuous action space based environments. This chapter introduces the problem statement, which provides insight into why this is an area of concern, the thesis aim/objective, its contribution to the field of software testing and finally the structure of the remainder of the thesis.

1.1 Problem Statement

As the popularity of Artificial Intelligence continues to grow across the world, and into every industry, so does the interest in Machine Learning (ML), and its branches. Thus far, there has been extensive research and work carried out in the field traditional Supervised Learning, however the branch of Reinforcement Learning (RL) has somewhat lacked in that regard. With the increase in use of Reinforcement Learning agents across the industries, it becomes increasingly more important to ensure that these agents are being sufficiently tested, to prevent against bugs and information leaks.

However, the current testing techniques used in traditional ML cannot be applied to RL problems, due to their fundamental differences, and there is a lack of adequate testing techniques which can be utilised. Although there are proposals for new RL-specific testing techniques, they are still insufficiently tested, and have yet to reach wide-stream use.

In the field of Reinforcement Learning, there are a range of algorithms which are widely and frequently used. They each fall under different categorises such as Q-Learning or Policy Optimisation, on-policy or off-policy, etc... Commonly used Q-Learning algorithms include Q-Learning, and Deep Q-Network (DQN), and commonly used Policy Optimisation algorithms include Deep Deterministic Policy Gradient (DDPG), Proximal Policy Optimisation (PPO), and Advantage Actor Critic (A2C). Although these algorithms have the same aim - i.e., to train the agent on the optimal policy - their individual design and architecture differ significantly depending on their categorisation. Therefore, it is imperative that any RL-specific testing technique account for, and be able to facilitate these differences, if it is to be widely-accessible and effective.

Therefore, this thesis aims to help to address the gap in the field, by exploring the current proposed techniques, and expanding upon a chosen proposal to improve its effectiveness and robustness. We also aim, during this process, to bring further awareness, and incite further research into this field.

1.2 Thesis Aim/Objective

The main objective of this thesis is to expand upon the existing work done in a proposed software testing technique adapted for Reinforcement Learning. The chosen technique is Mutation Testing (MT), and we will be adapting it for application on a wider range of algorithms, and environments. This thesis aims to achieve this by adapting an existing implementation of Mutation Testing [1], to facilitate the testing of the DDPG algorithm, and continuous action space-based environment, such as CartPoleContinuous.

The proposed MT approach has been designed to be functional for discrete action spaces only, and can facilitate the testing of the DQN, PPO, and A2C algorithms. Therefore, the further adaptation of the existing work will facilitate the testing of a wider range of algorithms such as DDPG algorithm, and allow for the use of continuous action space-based environment such as CartPoleContinuous. We hope that by the work carried out through this thesis, more research will be carried out to help address the gap in field of software testing of RL agents.

1.3 Thesis Contribution

This thesis, will build upon an existing work to adapt a proposed software testing method, - Mutation Testing - for Reinforcement Learning agents. This will be achieved by adapting the original implementation to be functional on a continuous action space, and on the DDPG algorithm. The existing work, is currently only able to facilitate discrete action space-based environments, and has been designed to facilitate the test of the DQN, PPO and A2C algorithms. Therefore, we will be expanding, and increasing the robustness of the proposed testing technique by extending it to a wider range of algorithms and environments.

1.4 Thesis Structure

The remainder of thesis will be as follows:

Chapter two, begins by providing background on the field of Reinforcement Learning (RL), introducing terms and concepts relevant to the foundation of RL problems. It will then continue into RL algorithms, touching on Q-Learning algorithms such as Q-Learning and DQN, as well as Policy Optimisation algorithms such as Policy Gradient, DDPG, PPO, and A2C. Finally, the area of software testing in the context of Machine Learning will be introduced, expanding into its applications in Reinforcement Learning, concluding with proposed RL-specific testing techniques.

Chapter three, will introduce and provide insight into the design on the experiment, expanding on the specifics of the testing approach to be utilised. This chapter will also introduce the algorithms, environment and mutants under test, as well as their specific parameters.

Chapter four, will provide details on the implementation on the experiment. This chapter will expand on, and provide specific details on the frameworks, and tools used to implement the experiment - this will also include code snippets to provide further clarity.

Chapter five, will focus on the evaluation of the results obtained from the experiment. The results will be presented and evaluated using specific metrics which will also be introduced in this chapter. Finally these results will be analysed and evaluated under a number of objectives, stated at the beginning of the chapter.

Chapter six, will conclude and summarise the work that was done throughout the thesis, highlighting the important notes from each chapter. Before finally touching on some avenues for potential future works in this field.

2 Background

In this chapter, we will be introducing Reinforcement Learning as a branch of Machine Learning, expanding on the various terms and concepts relevant to this paper. We will also be introducing various RL algorithms along with their defining features. We will introduce the area of software testing in the context of Machine Learning, and extend that to the area of Reinforcement Learning. Finally, we will introduce some proposed testing methods tailored specifically for RL problems.

2.1 Reinforcement Learning

Reinforcement Learning (RL) [2] is a sub-field/branch of Machine Learning (ML), which utilises a trial-and-error approach to train an AI agent. Inspired by human learning, an RL agent is placed in an environment, within which it learns how to achieve an optimal decision-making policy. Through this trial-and-error approach, the agent learns which actions are the most rewarding, considering its current state in the environment, thus enabling the agent to select the optimal sequence of actions that will maximise its total cumulative reward.

Dissimilar to traditional supervised Machine Learning, RL does not require any prior information or data, to begin the learning process. Rather than being provided with a training dataset, the agent must learn from its own actions, the resulting reward, and the input from its environment. An RL agent's ultimate goal is to maximise its final cumulative reward. The learning process can be explained by the diagram below.

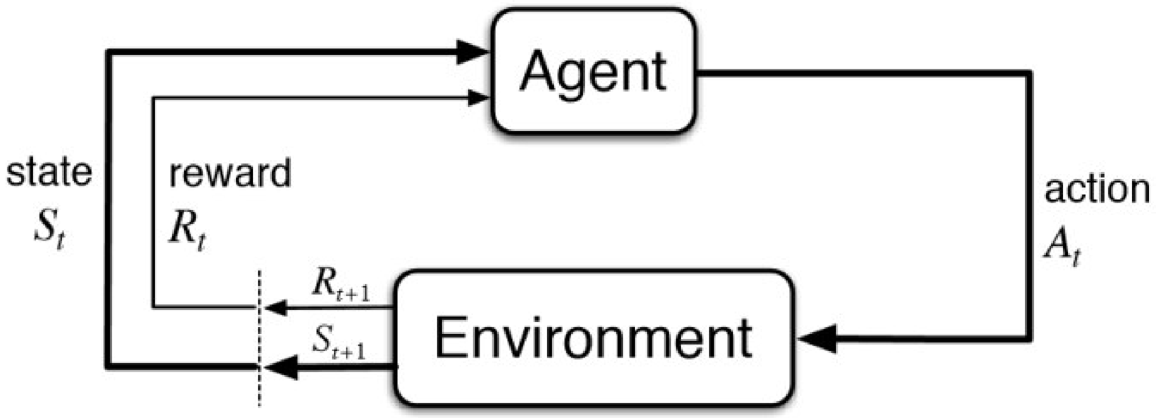


Figure 2.1: Reinforcement Learning Process

Components of Reinforcement Learning Process

To explain further, we will breakdown each component of the process pictured above.

Agent

The agent, is an object, which could be an algorithm or neural network, which makes the decision of what action will be taken within the environment. It works by continuously receiving input - also known as an observation - in the form of state and reward, from the environment, and utilises the input to make a decision for its next action.

Environment

The environment, can be described as the scene or the world-context, in which the RL agent operates. The environment contains all the necessary elements for the learning process (i.e. initial state), and then continuously receives feedback in the form of actions from the RL agent. Based on this feedback, it will return the resulting state and reward, as an observation, to the agent.

Action

The action, is the operation an agent can carry out, which causes a change in the current state. The action taken will be reported to the environment, which results in an observation containing, the next state, and a reward, being return to the agent. Depending on the environment, the action might include moving left or right, up or down, etc... The set of all possible actions can be referred to as the action space, A. Therefore, we can express a taken action as $a \in A$, where A is the action space, and a is the given action. Actions can be either discrete or continuous dependent on the environment.

State

The state, is a snapshot which describes the environment at a given moment, or time step. It has a major influence on the next action to be taken by the agent. Depending on the state, the same action could result in a differing reward. In an episode, the starting state is often referred to as the initial state, and the final state is referred to as the terminal state. [3]The set of all possible states can be describes as S. Therefore, we can express the given state as $s \in S$, where S is the set of all possible states, and s is the given state. Similar to actions, states can also be either discrete or continuous dependent on the environment.

Reward

The reward, can be described as the instant feedback that the agent receives from the environment after an action is taken. The feedback is a direct indication of the quality of the last action taken. The reward is arguable the main idea or most important aspect of Reinforcement Learning, as it influences both the short- and long-term outcome of the strategy used by the agent. The main goal of an RL agent is to obtain the best policy in order to maximise the cumulative reward value. We can express a reward that comes from taking an action, in the current state, and results in the moving to the next state, as $R(s, a, s')$. Where s is the current state, a is the action taken, and s' is the next state.

Other important definitions...

Policy

The policy, is the strategy used by an RL agent to decide which actions will be taken in the varying states. A policy can be either deterministic or stochastic.

A **deterministic policy**, works such that for a given state, the policy has a predefined action to be taken. This means that the policy will always select the same action for a given state, making it predictable.

A **stochastic policy**, works such that for a given state, the policy provides a set of probabilities for choosing the next action from a set of available actions. This type of policy is useful for capturing the uncertainty in an environment. However the randomness makes it unpredictable.

The overall aim of RL is to identify the optimal policy which maximises the cumulative reward. Therefore, depending on the situation, either type could produce the optimal policy.

Episodes

An episode, can be described as an agent's run through an environment, that starts in the initial state, and concludes when the agent arrives at the terminal state. An episode follows the agent as it aims to achieve a specified goal or until a specific condition is met. All of the states, actions taken, policy employed and reward accumulated is contained in an episode.

2.1.1 Markov Decision Process

Reinforcement Learning problems can be formally mathematically be expressed using the Markov Decision Process (MDP) model, which is defined by:

- the **state space** S, which contains all possible states an agent can be in
- the **action space** A, which contains all possible actions an agent can take
- the **time step** t, which is the time step at a given moment
- the **transition function** $T(s_t, a_t, s_{t+1})$, which is the probability that an action a_t performed in state s_t leads to state s_{t+1}
- the **reward function** $R(s_t, a_t, s_{t+1})$, which is the reward received after taking an action a_t, in state s_t, which leads to state s_{t+1}

The aim of the RL process is for the agent to obtain the optimal control of an [4]incompletely-known MDP, where the transition function and the reward function are initially unknown. They are learned by the agent during the RL training process.

To summarise, we can say that at a given time t, the agent receives state s_t from the environment. The agent performs the (current) optimal action a_t, which it passes to the environment for execution. The agent receives the reward R_t, and moves into the next state s_{t+1}. The agent's goal to maximise the long-term reward R over a given period of time (i.e. an episode). This process is visualised in Figure 2.1.

2.1.2 Reinforcement Learning Algorithms

As mentioned in Section 2.1.1 an RL agent can be defined as an algorithm or neural network, which makes the decision of what action will be taken within a learning environment. There two main types of RL algorithm: model-free and model-based.

Model-based algorithms: these algorithms know the transition and reward function, and utilise it to estimate the optimal policy and create the model.

Model-free algorithms: these algorithms learn the consequences of their actions through experience, without prior knowledge of the transition function or the reward function.

All RL algorithms fall under one of the two types above, however the following specifications also apply to model-free algorithms:

Value-based: these algorithms train the value function to learn which state is more valuable and take an action.

Policy-based: these algorithms train the policy directly to learn which action to take in a given state.

Actor-Critic: these algorithms combine the techniques from both value-based and policy-based algorithms.

Off-policy: these algorithms evaluate and update a policy different to the one currently being used to take the action.

On-policy: these algorithms evaluate and update the same policy being used.

2.2 Q-Learning Algorithms

Q-Learning algorithms are a model-free, off-policy and value-based. They are based on the Q-Learning algorithm, which will be introduced below. We will also introduce the Deep Q-Network (DQN) algorithm, which is based on the Q-Learning algorithm.

2.2.1 Q-Learning

Q-learning [5] - where the 'Q' stands for quality - is an algorithm that aims to find the optimal action, for an RL agent's current state. This means dependent on where the agent currently is in the environment, it will determine the next optimal action to be taken.

In Q-Learning, an agent maintains a 2-D Q-table, which keeps track of the action utility function - also known as the Q-value - of each action at a given state. These Q-values indicate how good or bad it is to perform a given action in a specified state, based on the past experiences of the agent in the environment. The Q-table is originally initialised with estimates for all Q-values, which are then updated as the agents proceeds through, and interacts with the environment. The update function of the Q-value can be expressed using the Bellman equation:

$$Q(s,a) = Q(s,a) + \alpha(R(s,a) + \gamma max(Q(s',a')) - Q(s,a)) \tag{2.1}$$

This equation is used to update the existing Q-value of a taken state-action pair, where, $Q(s,a)$ is the current Q-value of the state-action pair, α is the learning rate that controls how much the difference between previous and new Q-value is considered, $R(s,a)$ is the reward for the state-action pair, γ is the discount factor which controls the weight given to long-term strategies versus immediate benefits, and $max(Q(s',a')) - Q(s,a)$ is the maximum expected future reward of all actions at the next state.

Both the learning rate, and the discount factor aid in balancing the trade-off between exploration and exploitation.

Exploration is the process of the agent purposely choosing an action for the sole purpose of gathering information to understand its surroundings/environment.

Exploitation is the process of the agent purposely choosing an action it believes is optimal to obtain the best reward.

Exploration is necessary to ensure that information is gathered on the all possible actions in all states. However, as an agent must prefer actions that produce the highest reward based on the actions it has taken in the past, a dilemma arises regarding the trade-off between exploration and exploitation.

Balancing exploration and exploitation is vital to the agent's learning process; too much exploration hinders the access the agent has to the current reward, which defeats its purpose, while too much exploitation leads to lack of knowledge which hinders the agent from obtaining the globally optimal policy.

2.2.2 Deep Q-Network

In Q-Learning, the Q-table and the containing Q-values are central to overall learning process. However, as the size of state and action spaces grow larger, the Q-table faces what is known as the "curse of dimensionality" [6], as it becomes inefficient and impractical to represent large or continuous spaces in the table format. Thus, Deep Q-Networks (DQN) were born. DQNs are artificial neural networks trained with a variant of Q-Learning, therefore they are also model-free, off-policy and value-based.

They are far more scalable, as DQNs perform dimension compression on a Q-table and utilise value function approximation [7]. This means they make use of deep neural network to approximate the Q-value function, rather than storing the Q-value for each state-action pair in a table.

The value function approximation is as follows:

$$Q\phi(s, a) = f(s, a, k) \tag{2.2}$$

Where k is the input parameters of the function f, which is used to approximate the distribution of Q-values and function f is the neural network.

DQNs take the current state as an input to the neural network, and output a vector of Q-values for all possible actions from that state. A variant of stochastic gradient descent is used to train the neural network, which helps to minimise the difference between the approximated Q-values and the true Q-values.

2.3 Policy Optimisation Algorithms

Policy Optimisation algorithms utilise a policy-based approach, within which the RL agent directly learns the policy function that maps states to actions. This contrasts against the Q-Learning algorithms which trains a Q-value function to obtain the policy. However, similar to Q-learning, they are model-free, meaning that there is no prior knowledge of the transition or reward functions.

2.3.1 Policy Gradient

The Policy Gradient algorithm calculates the probability distribution of the random policy output actions from previously obtained observation. It then chooses to perform the action with the highest probability from a range of continuous distributions.

The goal of Policy Gradient is to approximate the optimal policy utilising gradient ascent on the expected return, which is defined as the sum of rewards from the initial state to the terminal state. The policy is usually modelled with a parameterized function respect to θ, $\pi_\theta(a|s)$. The value of objective function depends on this policy, as θ is used to regulate the probability of occurrence of each action. Policy Gradient aims to utilise θ to get the gradient for the objective function, $J(\pi_\theta)$, which is then used to maximise the expected return [8]. The objective function is as follows:

$$J(\pi_\theta) = E_{\pi_\theta}[\sum_{t=0}^{T-1} R(s_t, a_t)] = E_{\pi_\theta}[R(\tau)] \tag{2.3}$$

We can rewrite our policy gradient expression in the context of Monte-Carlo sampling:

$$\nabla J(\theta) = 1/N \sum_{\tau \in N} \sum_{t=0}^{T-1} \nabla_\theta log \pi_\theta(a_t, s_t) R(\tau) \tag{2.4}$$

2.3.2 Deep Deterministic Policy Gradient

One short-coming of DQN is that it is only suitable for discrete action spaces, this is because of its intrinsic greedy search in the target value. Deep Deterministic Policy Gradient (DDPG), was designed as an extension of DQN which would no longer be limited to the discrete action space, but would be able to facilitate for continuous action spaces.

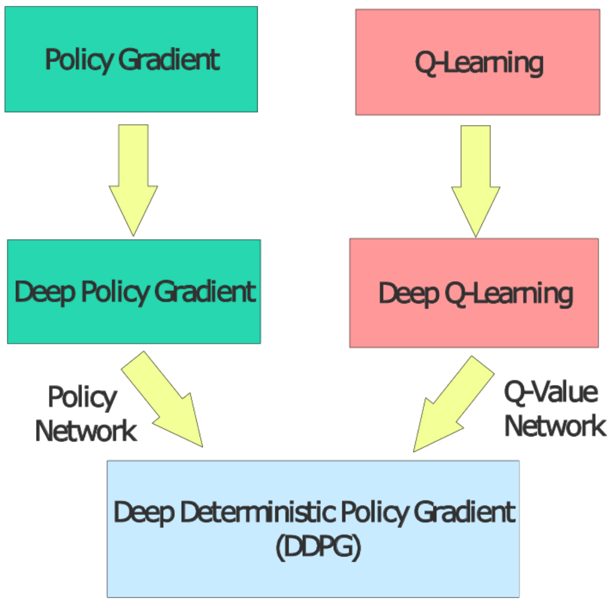


Figure 2.2: DDPG merges both DQN and DPG

DDPG [9] is a combination of both DQN and DPG, it inherits the off-policy aspect from Q-Learning, but also incorporates the actor-critic approach based on DPG. The algorithm utilises 2 neural networks, one for the actor and one for the critic. Consequently DDPG concurrently learns a Q-function, and a policy, using off-policy data and the Bellman equation to learn the Q-function, and using the Q-function to learn the policy.

The deterministic aspect of DDPG means that it maps specific action to a given state. Dissimilar to a value function, where we are return the probability function for all states, for deterministic policies, we are given a specified action to take, based on the state provided, there is no uncertainty.

Each time the agent interacts with the environment, we iteratively tweak and adjust the parameter θ of the neural network, to increase the likelihood of a good action being used in the future. This process is continued until the policy network converges to the optimal policy. The goal of Policy Gradient is to maximise the return (total expected rewards), so by adjusting the θ parameter of the policy network, the reward is maximised.

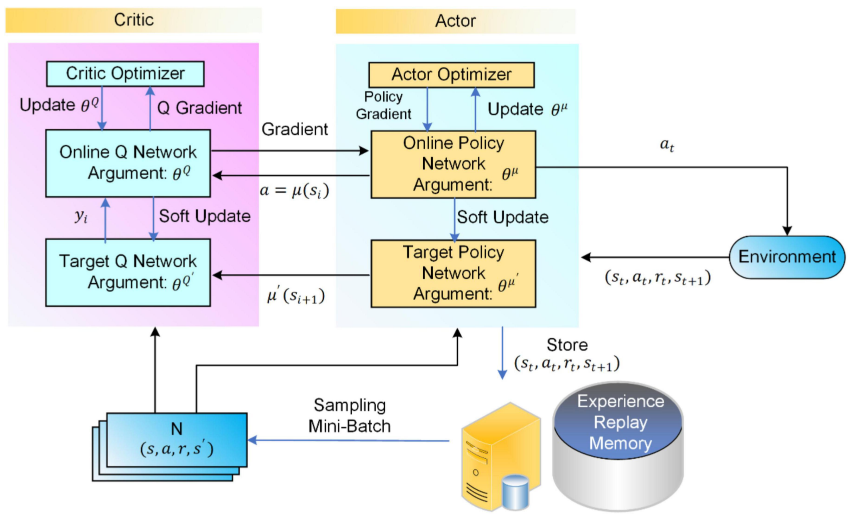


Figure 2.3: Design of the DDPG Algorithm

Key Components of DDPG

Its design comprises of three main aspects: the two neural networks: the actor network, the critic network, and the experience replay buffer [10].

Actor Network: one of two neural networks used in the DDPG algorithm. Its role is to learn the policy function, and map the relevant states to given actions. It is updated using a deterministic policy gradient theorem, which computes the gradient of the expected return in relation to parameters of the actor network. The gradient is then used to update the actor network's parameters in the direction that maximises the expected return.

Critic Network: the other of the two neural networks used in the DDPG algorithm. Its role is to evaluate the actions suggested by the actor network by estimating the state-action value function, also known as the Q-function. It then provides feedback to the actor network in the form of a Q-value - the value of taking certain actions in given states. It is updated using the temporal difference error between the predicted Q-values and the target Q-values. The Temporal Difference (TD) error is calculated using Bellman's equation, and is then used to minimise the mean square error loss between the predicted Q-value and the target Q-value.

In addition to the Actor and Critic...

DDPG uses target networks in conjunction with the main networks of the actor and critic. These target networks are copies of the main networks, which aim to stabilise training through the use of soft updates. These soft updates involve gradually adjusting the parameters of the target network towards the parameters of the main networks, which makes the target network updates more stable.

An optimizer is also used in conjunction with the main network and target network, to help each neural network achieve its specific targets. In the case of the actor network, an optimiser is used to help it maximise the expected return. In the case of the critic network, an optimiser is used to help it minimise the mean square error loss. Commonly used optimizers include Adam, SGD, and RMSprop.

Experience Replay Buffer: similar to its use in DQN, it stores experiences in the form - state, action, reward, next state - encountered by the agent during interactions throughout the environment. The experiences are then sampled from randomly through the training process which helps to reduce the variance of updates, and de-correlate the data.

2.3.3 Proximal Policy Optimization

The Proximal Policy Optimization [6] (PPO) algorithm is another policy-based algorithm within which the agent learns the policy which will result in the maximum cumulative reward for a given environment. It was developed for use on both continuous and discrete action spaces, though it is primarily used in discrete spaces. Dissimilar to DDPG, PPO learns a stochastic policy, meaning that rather mapping a given action to a specific state (the deterministic policy approach), it outputs a probability distribution over the possible actions in a given state.

One of the major disadvantages of using policy optimisation methods is their hypersensitivity to the policy's hyper-parameter tuning. PPO addresses this by employing the use of a proximal policy optimization strategy to iteratively improve/update the policy parameters. This strategy aims to constrain the policy updates to be close to the previous policy, which is important partially because PPO is an on-policy algorithm, meaning that the same policy is being updated.

The constraining of the policy updates is achieved by imposing a clipping mechanism or penalty term in the objective function, and by doing so PPO is able to maintain a stable and efficient learning process, enabling the agent to learn effective policies in a reliable manner.

Key Components of PPO

Its design comprises of five main aspects: policy, objective function, clipping mechanism, value function, and multiple epochs.

Policy: PPO learns stochastic policy, which returns a probability distribution over all possible action for a given state.

Objective Function: PPO optimises an objective function to approximate policy improvement. It is the surrogate for the expected improvement in the policy performance, and is generally used in its clipped version.

Clipping Mechanism: PPO uses a clipped surrogate version of the objective function to prevent large policy updates which could have a negative effect on the outcomes. It utilises a mechanism which clips the ratio between the probability of actions under the new and old policy, thus maintaining policy updates within a safe and reliable range.

Value Function: PPO incorporates value function estimation to reduce variance in the gradient estimates. This improves stability during training and improves sample efficiency overall.

Multiple Epochs: PPO performs multiple epochs of optimisation on the data collected from the agent's interactions with the environment. From this trajectories are collected and used to compute surrogate objective function, which is then optimised using mini-batch updates.

2.3.4 Advantage Actor Critic

Advantage Actor Critic (A2C) is the final policy optimisation algorithm we will be touching on. It is policy-based algorithm which utilises policy gradient, and is designed for discrete and continuous action spaces [11]. Following in the footsteps of PPO, A2C is on-policy and learns a stochastic policy, which it uses alongside advantage estimates.

A2C calculates the advantage function to update/improve its policy. It represents the advantage of taking a specific action in a given state, over the average action value. The advantage is generally estimated using the difference between the observed return and the value function estimate.

Advantage then indicates whether taking a specific action is "worth it" - essentially whether taking the action in the current state would be better or worse than the average expectation. A positive advantage indicates the action is better than the average, whereas a negative advantage indicates the action is worse than the average.

Key Components of A2C

Its design comprises of three main aspects: the two neural networks: the actor network, the critic network, the advantage function.

Actor Network: [12] one of the two neural networks used in the A2C algorithm. Its role is to learn the policy. It takes in a state as an input and returns a probability distribution over all possible actions. The actor's main aim is to maximise the expected return by adjusting the policy parameters based on the advantage estimations provided by the critic network.

Critic Network: the other of the two neural networks used in the A2C algorithm. Its role is to estimate the value function, which is the expected cumulative reward from a given state.

It takes in a state as input and returns a scalar value representing the expected return. The critic's main aim is to evaluate the quality of the current policy in different states by calculating the advantages.

Advantage Function: the advantage function (as briefly explained above) represents the advantage of taking a specific action in a given state in comparison to the average value of the specified state. The function is as follows [13]:

$$A(s,a) = Q(s,a) - V(s) \tag{2.5}$$

Where A is the advantage, s is the given state, a is an action being considered, Q is the q-value for action a in state s, and V is the average value of state s.

However as this function is reliant on two value functions, the Temporal Difference (TD) error is often used as a good estimator of the advantage function, adapting the function to be as follows:

$$A(s,a) = r + \gamma V(s') - V(s) \tag{2.6}$$

Where A is the advantage, s is the given state, a is an action being considered, r is the reward obtained, γ is the discount factor, V is the average value of the next state s', and V is the average value of state s.

2.4 Software Testing

The importance of software testing cannot be overstated. It plays an essential role in ensuring the quality, effectiveness, and reliability of software systems. Without sufficient testing, software bugs, and defects are likely to create more significant problems later down the line. There are a plethora of testing techniques and metrics in the software field, and even more specifically in the world of Machine Learning. We will be introducing three examples of these techniques: mutation testing, fuzzing, and adversarial attack testing.

2.4.1 Mutation Testing

Mutation testing [14] involves designing a set of mutant operators to inject faults into the training data, which is then passed into the ML model. The mutated training data is used to train the model, generating a mutated model. The mutated model is then tested using a test suite/dataset. The quality of the test suite can then be evaluated based on its performance in detecting the injected faults.

This testing technique's aim is to evaluate the quality of the test suite's ability to detect potential faults. By systematically and deliberately introducing mutations, developers can easily identify weaknesses in their testing processes, and improve the overall reliability of the software.

2.4.2 Fuzzing

Fuzzing [15] involves generating large amounts of invalid, malformed or random input data which is then passed into an ML system to reveal software defects and vulnerabilities. A fuzzing tool injects the faulty input data into the system and proceeds to monitor for exceptions such as abnormal runtime behaviour, information leaks and program crashes.

This testing technique's aim is to evaluate how the system performs in scenarios of unexpected inputs. The system is observed to see if it has any negative reactions or even lack of reaction to the faulty inputs which would indicate security or quality gaps. [16] The developer can then use the observations/results of fuzzing to improve the system's vulnerabilities and prevent against potential catastrophic events such as information leakage.

2.4.3 Adversarial Attack Testing

Adversarial Attack testing involves generating adversarial examples, and passing them into the ML system under test. Adversarial examples are inputs that have been subtly manipulated before being passed into the ML system [17]. These deceptive inputs would appear "normal" to humans but could have massive potential ramifications in the ML system.

Adversarial attack testing directly mimics adversarial attacks which are a commonly used method of attacking and exploiting AI systems [18]. These attacks aim to exploit systems but passing in seemingly valid inputs which appear "normal" to humans but can manipulate an AI system into nefarious acts. These attacks generally aim to exploit the training phase or predictive models of a given ML model. Therefore, using this method as a form of testing can help developers directly safeguard and mitigate against attacks of this nature, thus better protecting the system and its data.

2.5 Software Testing of Reinforcement Learning

As was mentioned at the beginning of the section, the importance of software testing cannot be overstated, especially in the world of Machine Learning. However, although there are many ML testing techniques/methods available, majority of them are not applicable to Reinforcement Learning problems. This is due to various differences in the structure of RL models in comparison to traditional ML models. These are just a few of the major differences, which hinder the use of traditional testing techniques on RL problems:

1. Dynamic Environment Interactions: RL agents interact with dynamic environments, where the response to their actions directly influences subsequent states, actions and rewards. This contrasts heavily against traditional supervised learning, where fixed training data is fed to the agent. As a consequence of this, traditional testing methods have not been designed to account for the dynamic nature of RL problems and therefore cannot be directly utilised [19].

2. Dynamic Environments: RL environments can be non-stationary in nature, meaning that the dynamics are subject to change at any given time step. Traditional testing methods generally assume stationary or predictable environments, and have thus, not been designed to account for dynamic environments. RL problems would require a testing method that has more of an adaptive testing strategy [20].

3. Temporal Dependencies: RL involves sequential decision-making over a period of time, where an action taken at a specific time step will affect future states and rewards. Due to the nature of Supervised Learning, traditional testing techniques typically focus on individual inputs/output, and do not account for temporal dependencies. Therefore, many traditional testing methods cannot be utilised for RL problems, as they fail to account for the agent's history and evolving state of the environment, which directly impacts the effectiveness of the actions taken.

4. Stochastic Environments: RL environments can be stochastic, which means that the outcomes of actions taken are not deterministic. Again many traditional testing methods assume deterministic inputs and outputs, meaning that they are unable to capture the inherent uncertainty and variability in RL environments. This once again, poses a hindrance for utilising these traditional methods.

5. Exploration-Exploitation Trade-off: RL agents must strike a competent balance between exploration - trying to new actions to learn about the environment - and exploitation - taking actions known to yield high reward. This adds complexity to the testing process, and as such, a traditional testing method may detect the unexpected behaviour as faults in the agent's policy rather than simply exploration-driven behaviour.

Due to these reasons and more, there is a significant scarceness of adequate testing techniques and metrics which are tailored specifically for the area. As Reinforcement Learning is becoming increasingly popular, and more widely used, it proves more evidently than ever, how crucial it is to develop suitable testing methods.

Therefore, in this section we will introduce some proposed testing methods specifically tailored towards RL problems. These methods will include adaptions of the traditional testing methods, as well as newly proposed methods. The methods to be touched on include: mutation testing, search-based testing, search-based with surrogate model , and simulated users testing.

2.5.1 Mutation Testing

Tambon et al [1] proposes a mutation testing method which modifies and adapts traditional mutation testing to be applicable on RL problems. The method proposes to introduce mutations to the RL system directly through injecting faults into the environment, agent or policy - rather than injecting the faults into the training data, as is done in traditional implementations. The changes are introduced to the learning algorithm, reward structure, exploration strategy, policy or agent's parameters. These faults mimic real potential faults that could occur during a training process.

Once these faults are introduced and the system is considered faulty or 'mutated'. The agent is then assessed in a test suite, to evaluate how effective/robust the test suite is at detecting the injected mutations. The image below depicts the usual steps involved in the mutation testing process.

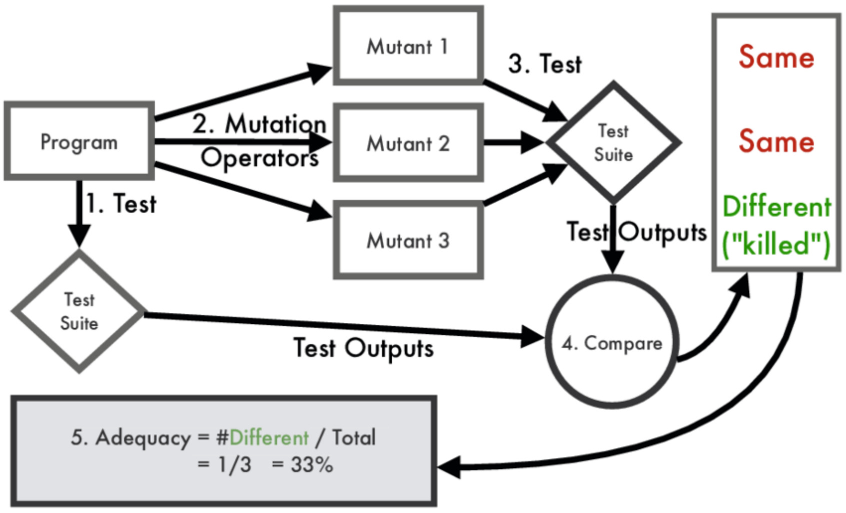


Figure 2.4: Mutation Testing Process

1. Execute the program P, in the test suite
2. Use mutation operators to generate mutant programs $P_1, ...P_n$ from program P
3. Execute all mutant programs P_i in the test suite
4. "Kill" or flag, each mutant program P_i that behaves differently to the program P, the rest get to survive
5. Evaluate the effectiveness of the test suite using the mutation score.
   $score = \#mutants_{killed} / \#mutants_{total}$, (1 is the highest score, 0 is the lowest)

A mutant operator can be defined as a rule or transformation that takes the original program and modifies it, and creates a variant with valid changes.

The proposed method of adapting Mutation Testing for RL problems, is highly effective and useful. It takes the originally successful method used in traditional learning, and modifies it for use in RL problems. The employment of mutants that mimic likely real faults proves to be highly effective, as it evaluates how well the test suite would be able to detect those faults if they were to occur.

2.5.2 Search-Based Testing - STARLA

Zolfagharian et al [21] proposes STARLA, a Search-based Testing Approach for RL Agents. Its aim is to efficiently test the policies of Deep Reinforcement Learning (DRL) agents in safety-critical environments. Dissimilar to adversarial attacks, STARLA focuses on identifying failing executions of the agent by utilizing Machine Learning models and a genetic algorithm.

STARLA's main aim is to generate and identify episodes with high fault probabilities to evaluate the safety of deploying an RL agent. Its approach uses a genetic algorithm, which takes in a diverse subset of episodes as an initial population. Through crossover and mutation, offspring are generated to form a new population. The fitness function generates values for the new population, and the process continues iteratively until all fitness functions are satisfied (or the maximum number of generations is reached).

The algorithm continues to iterate through this process of creating offspring, calculating fitness, and updating populations until termination conditions are met. Once the termination conditions are met, they are left with a set of solutions which satisfy at least one fitness function.

Overall the proposed approach represents a significant advancement in testing methodologies for DRL agents. By using a search-based approach, STARLA was able to effectively detect faulty episodes. The use of a genetic algorithm and state abstraction also seemed to enhance the accuracy and efficiency of the fault detection. STARLA seems to be a highly promising and efficient method of testing for DRL agents.

2.5.3 Search-based Approach with Surrogate Models

Biagiola et al [22] propose a search-based testing approach utilising surrogate models for assessing RL agents. Their objective was to test RL agents by training a classifier on environment configurations where failures occur during the training process. The classifier is trained of two types of environment configurations: failure (agent does not successfully accomplish task), and non-failure (agent successfully accomplishes task). When testing the trained classifier acts as a surrogate model for the RL agent's behaviour in the environment.

The surrogate model predicts the likelihood of failure for a given environment configuration, and this prediction is then used as a fitness function during the search process. This will aid and guide the next new iteration of environment configuration towards configurations more likely to induce failures in the RL agent. The approach's use of the classifier for prediction, saves computational time by only executing the RL agents in an environment whose configuration is likely to expose errors.

Overall, the proposed approach efficiently guides the search for environment configurations that stress-test RL agents, thereby improving the effectiveness of testing, while also reducing the overall computational overhead.

2.5.4 Simulated Users

Bignold et al [23] propose a testing method which utilises simulated users instead of human trainers. These simulated users, mimic human knowledge, bias and interaction, and enable the development and testing of RL agents to occur in a more affordable and efficient manner than involving actual humans. They simulate human interaction such as providing feedback and guidance to the RL agent. These simulated users can then be used to evaluate the performance of interactive RL models. The method involves three main steps: construction of interaction model, implementation of the model, and evaluation of the model.

Construction of the Interactive Model: Identify simulated user requirements, create models representing interaction characteristics, and generate users with varying characteristics to cover a range of possibilities.

Implementation of the Interactive Agent: Develop the agent considering elements like advice interpretation and modification, aligning with the specific interactive RL field and the role of the human in the interaction process.

Evaluation of the Interactive Approach: Test different agents (similarly to human trials), control user characteristics using simulated users, and reset/alter simulated users to minimize bias.

Overall, the proposed method utilises a systematic approach to assessing interactive RL agents using simulated users. This approach enables controlled testing of the agents' performance under varying human interaction scenarios. This is a more efficient and cost-effective method of testing, in comparison to human trials.

2.6 Summary

In this chapter, we introduced Reinforcement Learning as a branch of Machine Learning, explaining important terms and concepts that are relevant to the paper. We went into detail introducing the different types of RL algorithms and the further classification under the relevant types. We also introduced the topic of software testing, giving examples of testing techniques used in Machine Learning, while expanding on why these techniques are not applicable to RL problems. Finally we introduce the proposed testing techniques/methods tailored specifically to RL problems.

3 Design

In this chapter, we will be introducing the design of the experiment. The experiment will be expanding upon, and building off of the adapted mutation testing method proposed by Tambon et al [1], by adapting their method to be functional for CartPoleContinuous which is a continuous environment, and for the DDPG algorithm. We will introduce the design specifications of the experiment in relation to three algorithms: DDPG, PPO and A2C, as well as the CartPoleContinuous environment which will also be implemented in the experiment. Finally, we will introduce the mutations that will be implemented and as well as their specifications.

3.1 Algorithms

We will be implementing the DDPG, PPO and A2C algorithms for the purposes of this experiment. Tambon et al originally implemented and adapted their approach for the DQN, PPO and A2C algorithms. Therefore, we will be using their same implementation of PPO and A2C, and include our own implementation DDPG.

All three algorithms will be implemented using Stable-Baselines3 - same as the original source material. The hyper-parameters will be tuned in accordance with the tuned parameters [24], which the source framework recommends.

10 agents for each algorithm will be generated and run for the training phase. During the testing phase, averages of the agents will be calculated and used for the final evaluation.

3.2 Environment

We will be implementing the CartPoleContinuous environment which is continuous action space-based environment, for the experiment. The CartPole environment is a popular benchmark used in Reinforcement Learning. The concept, though fairly simple, creates an optimal learning environment for an RL agent. The most widely used implementation of the CartPole environment is gym's [25] implementation, which corresponds to the cart-pole problem described in "Neuron-like Adaptive Elements That Can Solve Difficult Learning Control Problem", by Barto, Sutton and Anderson [26].

The CartPole environment consists of a cart with a pole connected to its middle through a pivot [27]. The aim of the agent is to stabilise, and balance the pole, to prevent it from falling. It achieves this by appropriately shifting the cart, left or right. Since the goal is to keep the pole upright, for as long as possible, a reward is given for every step taken by the agent, until the reward limit or terminal state is achieved.

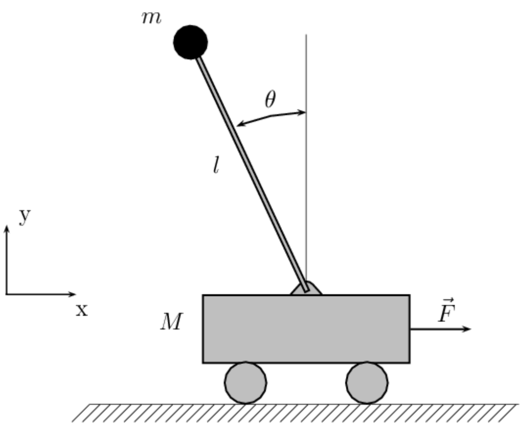
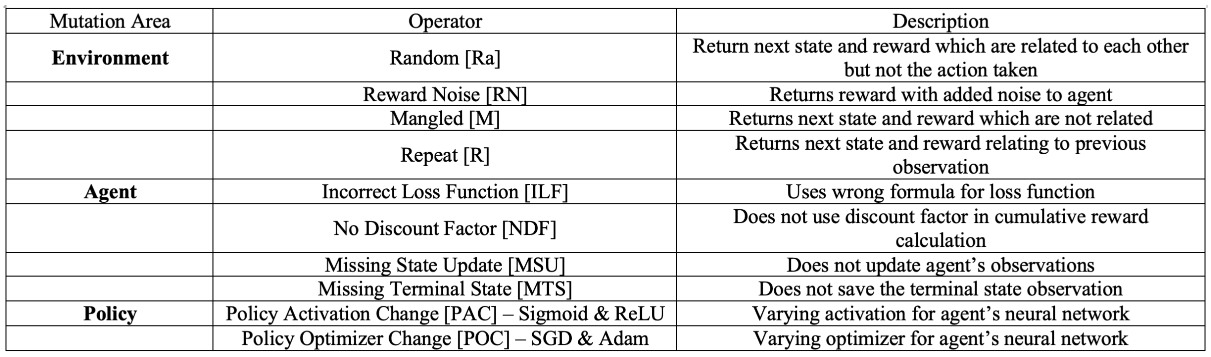


Figure 3.1: CartPole Problem

CartPole was original designed as for a discrete action space, however many implementations have now expanded to facilitate a continuous action space. The implementation proposed by Penkin et al [28] achieves the continuous action space, by overriding the base code from the regular discrete gym implementation of the CartPole environment, and changing the action space to [-1, 1].

3.3 Mutations

We will be implementing a set of 10 different mutations for the purposes of this experiment. The mutations used, simulate real faults that could occur on an agent, environment or policy level. These faults could be as a result of human error, or mechanical failure, regardless it is important to investigate the influence each mutation could have on the performance of a given agent, so it is possible to mitigate against them in the future. Below is a table briefly describing each mutation.



Mutation Area	Operator	Description
Environment	Random [Ra]	Return next state and reward which are related to each other but not the action taken
	Reward Noise [RN]	Returns reward with added noise to agent
	Mangled [M]	Returns next state and reward which are not related
	Repeat [R]	Returns next state and reward relating to previous observation
Agent	Incorrect Loss Function [ILF]	Uses wrong formula for loss function
	No Discount Factor [NDF]	Does not use discount factor in cumulative reward calculation
	Missing State Update [MSU]	Does not update agent's observations
	Missing Terminal State [MTS]	Does not save the terminal state observation
Policy	Policy Activation Change [PAC] – Sigmoid & ReLU	Varying activation for agent's neural network
	Policy Optimizer Change [POC] – SGD & Adam	Varying optimizer for agent's neural network

Figure 3.2: Mutations with brief descriptions

3.3.1 Environment Mutations

The following mutations are implemented on an environment-level. They directly affect the output returned from the environment to the agent, thus impacting the learning process of the agent in the environment. The aim of these mutations is to see the effect that distorting the output provided by the environment has on the agent's performance.

Random : The Random (Ra) mutation, involves the environment returning a next state and reward based on a random observation (from the buffer) and returning them to the agent. In this case, the next state and reward are related to each other, but not related to the current observation.

Reward Noise : The Reward Noise (RN) mutation, involves the environment returning the reward with added (Gaussian) noise to the agent. In this case, the returned reward would be distorted, which would in turn, influence the learning process.

Mangled : The Mangled (M) mutation, involves the environment returning a next state and reward individually based on two random observations (from the buffer), and returning them to the agent. In this case, the next state and reward are not related to each other, nor are they related to the current observation.

Repeat : The Repeat (R) mutation, involves the environment returning a next state and reward based on the previous observation. In this case, the next state and reward are related to each other, and related to the previous observation but not to the current observation.

3.3.2 Agent Mutations

The following mutations are implemented on an agent-level. They directly affect the agent via its neural network. The aim of these mutations is to see the effect that the absence or distortion of various vital aspects of the neural network has on the agent's performance.

Incorrect Loss Function : The Incorrect Loss Function (ILF) mutation, involves a wrongly defined loss function being used in the neural network of the agent. This can be achieved by adding additional variables to the function or by inverting the signs.

No Discount Factor : The No Discount Factor (NDF) mutation, involves removing the discount factor γ, from the reward function used in calculations for the agent. The removal of the γ means that no discount parameter will be used in the reward calculations for the agent.

Missing State Update : The Missing State Update (MSU) mutation, involves not updating the observation passed to the agent. This can be achieved by setting the current observation to be equivalent to the previous observation, rather than deriving it in the correct manner.

Missing Terminal State : The Missing Terminal State (MTS) mutation, involves not saving the terminal state of an episode during the training process.

3.3.3 Policy Mutations

The following mutations are implemented on a policy network-level. They directly affect the policy network utilised by the agent. The aim of these mutations is to see the effect that varying the activation and optimizer functions has on the agent's performance.

Policy Activation Change - Sigmoid & ReLU : The Policy Activation Change (PAC) mutation, involves changing the default policy activation function used in the policy network of the agent. In this case, the activation functions to be used will be Sigmoid and ReLU.

Policy Optimizer Change - SGD & Adam : The Policy Optimizer Change (POC) mutation, involves changing the default policy optimizer function used in the policy network of the agent. In this case, the optimizer function to be used will be SGD and Adam.

3.4 Summary

In this chapter we established the design details of the experiment. We introduced the specific algorithms we would be experimenting with: DDPG, PPO, and A2C. We also introduced CartPoleContinuous, the environment we will be implementing for the experiment, which is an adaptation of the original discrete CartPole environment. Finally, we introduced the mutations that would be used in the experiment, along with a short description of their designs.

4 Implementation

In this chapter, we will be describing the implementation of the experiment carried out for the purposes of this thesis. The entire experiment was implemented using Python 3.8 in a virtual environment, and was adapted from the existing source code [29], provided by Tambon et al from their original implementation. We will introduce the CartPoleContinuous environment which was adapted from the original gym CartPole environment, the Stable Baselines3 framework which was used to implement the DDPG algorithm, as well as the PPO and A2C algorithms which were used as a comparative baseline, and finally, the various changes made to implement the mutations.

4.1 Environment

4.1.1 CartPoleContinuous

The CartPoleContinuous environment was used for this experiment to investigate how the Mutation Testing process would perform on a continuous environment. This environment is an adapted implementation of the CartPole environment from gym [25]. The creator of CartPoleContinuous made one main change which enabled CartPole to function as a continuous environment. This change was to the action space, which has now been set to be continuous from [-1 to 1] [28]. The specific version CartPoleContinuous-v1 was used in this experiment, to match the original implementation which used CartPole-v1.

4.2 Algorithms

As mentioned above, the experiment was carried out on Python 3.8. The algorithms were implemented using Stable Baselines3. Stable Baselines [30] is set of improved implementations of Reinforcement Learning algorithms, originally based on OpenAI Baselines. Using this framework ensured an even playing field for the experiment. This is due to the fact we were able to implement the algorithms using a standardised base code, and only override and modify relevant parts of the code, which ensures the code is less likely to run into unintended errors. Overall, use of the Stable Baselines framework provided better control over our implementation choices.

The experiment involved running 10 RL agents, for each of the three algorithms: DDPG, PPO and A2C, on the CartPoleContinuous-v1 environment. The agents all had *seed* $= 0$, and were run for a range of 10,000-50,000 timesteps.

4.2.1 DDPG

The DDPG algorithm was implemented using the Stable Baselines3 implementation, and was tuned to the following hyper-parameters:

Hyper-parameter	Value
learning_rate	0.001
buffer_size	1,000,000
learning_starts	1,000
batch_size	256
tau	0.005
gamma	0.99
train_freq	1
gradient_step	1
action_noise	none
policy_kwargs	none
tensorboard_log	tensorboard_directory
seed	1

Table 4.1: Hyper-parameters used in DDPG algorithm implementation

These hyper-parameters were adapted from the ones present in the RL-Zoo source code [24]. The main parameter that was adjusted, was the 'learning_starts' parameter, which was increased from 100, to 1000. This was due to number of timesteps chosen for the experiment.

4.2.2 PPO & A2C

The PPO and A2C algorithms were also implemented using the Stable Baselines3 implementation, and the hyper-parameters were set the same as was used in the original implementation [1].

4.3 Mutations

As described in section 3.3, three types of mutations were implemented in this experiment: environment, agent and policy. To implement this mutations, modifications were made to override the base code provided by Stable Baselines3. The modifications that were made simulate real potential faults that could occur when using RL algorithms.

4.3.1 Environment Mutations

Random

To implement the Random mutation, a random observation is used to generate the next state and reward, to return to the agent.

```
if "random" in mutation:
    random_observation = buffer.sample()[0]
    next_state = random_observation["next_state"]
    reward = random_observation["reward"]
```

Reward Noise

To implement the Reward Noise mutation, a random floating point number is generated and added to the reward, before returning it to the agent.

```
if "reward_noise" in mutation:
    reward += random.normalvariate(0, 0.1 * reward)
```

Mangled

To implement the Mangled mutation, a random next state, and random reward are generated and returned to the agent.

```
if "mangled" in mutation and total_steps > 0:
    batch_size = min(10, total_steps)
    observation_pool = buffer.sample(batch_size=batch_size)
    next_state = random.choice(observation_pool)["next_state"]
    reward = random.choice(observation_pool)["reward"]
```

Repeat

To implement the Repeat mutation, the previous observation is used to generate the next state and reward, which is returned to the agent.

```
if "repeat" in mutation and total_steps > 0:
    prev_observation = buffer[-2]
    next_state = prev_observation["next_state"]
    reward = prev_observation["reward"]
```

4.3.2 Agent Mutations

In Stable Baselines3, the DDPG algorithm is considered a special case of the TD3 algorithm (it's successor), this is because they share the same policies and implementation. Stable Baselines have thus made it so that the DDPG algorithm is simply an extension of the TD3 algorithm, so majority of the implementation remains in the TD3 source code. Due to this, the agent-level mutations had to made to the TD3 algorithm implementation, which the DDPG algorithm would then inherit.

Incorrect Loss Function (ILF)

To implement the ILF mutation, the formula used to calculate the target q-values was adjusted as can be seen below.
Original:

```
target_q_values = replay_data.rewards + (1 - replay_data.dones) * gamma
* next_q_values
```

Mutation:

```
target_q_values = replay_data.rewards - (1 - replay_data.dones) * gamma
* next_q_values
```

No Discount Factor (NDF)

To implement the NDF mutation, the discount factor γ (gamma) was removed from the formula used to calculate the target q-values.

```
target_q_values = replay_data.rewards + (1 - replay_data.dones) *
next_q_values
```

Missing State Update (MSU)

To implement the MSU mutation, the previous observation of the agent is used again, rather than updating.

```
if "missing_state_update" in mutation:
    observation = np.array(prev_observation).copy()
```

Missing Terminal State (MTS)

To implement the MTS mutation, the terminal state position is set to 0.

```
if "missing_terminal_state" in mutation:
    dones[self.pos] = 0
```

4.3.3 Policy Mutations

Implementing the policy mutations, focused on varying the different activation or optimizer function used. For the policy activation function, we varied between "Sigmoid" and "ReLU", and for the policy optimizer function, we varied between "SGD" and "Adam".

Policy Activation Change (PAC) - Sigmoid & ReLU

To implement the PAC mutation, the policy activation function is changed, dependent on the specification, i.e. "Sigmoid" or "ReLU"

```
if "policy_activation_change" in mutation:
    if mutation["policy_activation_change"] == "Sigmoid":
        policy_kwargs["activation_fn"] = th.nn.Sigmoid
    elif mutation["policy_activation_change"] == "ReLU":
        policy_kwargs["activation_fn"] = th.nn.ReLU
```

Policy Optimizer Change (POC) - SGD & Adam

To implement the mutation, the policy optimizer function is changed, dependent on the specification, i.e. "SGD" or "Adam"

```
if "policy_optimizer_change" in mutation:
    if mutation["policy_optimizer_change"] == "SGD":
        policy_kwargs["optimizer_class"] = th.optim.SGD
    elif mutation["policy_optimizer_change"] == "Adam":
        policy_kwargs["optimizer_class"] = th.optim.Adam
```

4.4 Summary

In this chapter, we established all relevant details of the implementation of the experiment. We specified the details of how CartPoleContinuous was adapted from the original discrete environment. We introduced Stable-Baselines3, the framework that was used to implement each of the algorithms, alongside the hyper-parameters for the algorithms. Finally, we explained and provided the Python code implementation for each of the mutations used in the experiment.

5 Evaluation

In this chapter, we will be carrying out an evaluation of the results obtained from the experiment as described in chapter four. We will be introducing the main objectives for the evaluation, the metrics used during the process, as well as presenting the results obtained before finally carrying out an analysis based on the stated objectives.

5.1 Objectives

Going into the evaluation we have three main objectives:

- **O1.** Analyse the overall effectiveness of the Mutation Testing process, e.g. how effectively were the mutants killed, which mutants were consistently killed, which mutants consistently went undetected.
- **O2.** Compare the performance of the Mutation Testing process on the PPO and A2C algorithms on the CartPoleContinuous environment versus the CartPole discrete environment.
- **O3.** Discuss the effectiveness of both evaluation metrics.

5.2 Evaluation Metrics

We will be evaluating the results of the experiment using two of the metrics proposed in the original source material: AVG - RL adapted mutation score, and R - reward-based statistical test.

5.2.1 AVG

The AVG metric, is an adaptation of the original mutation score metric used in traditional mutation testing. The original mutation score metric was simply defined as the number of killed mutants over the total number of mutants. The AVG variation was proposed Lu et al [31] as a modification of the original metric that would be applicable to Reinforcement

Learning problems.

$$\frac{avg(r)_{healthy}}{n} : \frac{avg(r)_{mutant}}{n} < \theta \tag{5.1}$$

Where r is the reward, n is the number of episodes.

This metric involves calculating the average total reward obtained by the healthy agent, as a ratio against the average total reward obtained by a given mutated agent. The result is then compared against θ, the 'killing' threshold. If the result is less than θ - in this case $\theta = 0.9$ - then the mutant is considered not killed. However, if the result is greater than or equal to θ, the mutant is considered killed.

As this metric does not account for multiple agents, the number of times the ratio for each agent is below the threshold θ will be recorded, and if the cumulative number is below 70% (7/10), then we will consider the mutant "not sufficiently detected or killed".

5.2.2 R

The R metric was proposed by Tambon et al [1], as a simpler evaluation metric to the first, as it would allow for more mutants to be killed, and is not as limited due to stochasticity of the RL problems as the first metric.

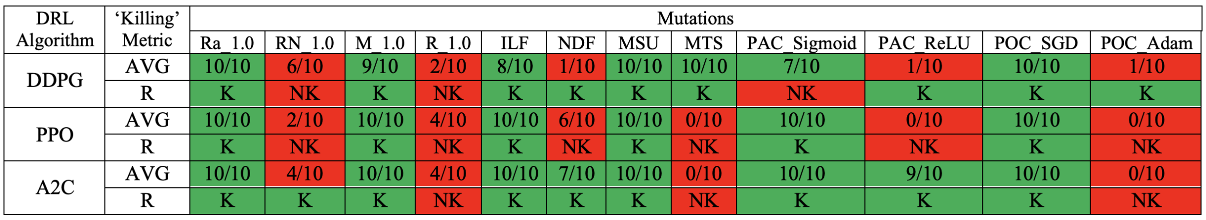
$$\frac{\sum_{n}^{i=0} r_i}{n} \tag{5.2}$$

Where r is the reward, n is the number of episodes.

This metric involves calculating the total reward over the number of episodes, and extracting the p-value. If the p-value < 0.8, inconclusive is returned, as the statistical power of the test was insufficient. If the p-value < 0.05, it is considered significant, and 'Killed' is returned, otherwise it is insignificant and 'Not Killed' is returned. In the case of this experiment, inconclusive results will be consider 'Not Killed'

5.3 Results & Analysis

Below is a table detailing the results of the experiment carried out.



DRL Algorithm	'Killing' Metric	Mutations											
		Ra_1.0	RN_1.0	M_1.0	R_1.0	ILF	NDF	MSU	MTS	PAC_Sigmoid	PAC_ReLU	POC_SGD	POC_Adam
DDPG	AVG	🟩 10/10	🟥 6/10	🟩 9/10	🟥 2/10	🟩 8/10	🟥 1/10	🟩 10/10	🟩 10/10	🟩 7/10	🟥 1/10	🟩 10/10	🟥 1/10
	R	🟩 K	🟥 NK	🟩 K	🟥 NK	🟩 K	🟩 K	🟩 K	🟩 K	🟥 NK	🟩 K	🟩 K	🟩 K
PPO	AVG	🟩 10/10	🟥 2/10	🟩 10/10	🟥 4/10	🟩 10/10	🟥 6/10	🟩 10/10	🟥 0/10	🟩 10/10	🟥 0/10	🟩 10/10	🟥 0/10
	R	🟩 K	🟥 NK	🟩 K	🟥 NK	🟩 K	🟥 NK	🟩 K	🟥 NK	🟩 K	🟥 NK	🟩 K	🟥 NK
A2C	AVG	🟩 10/10	🟥 4/10	🟩 10/10	🟥 4/10	🟩 10/10	🟩 7/10	🟩 10/10	🟥 0/10	🟩 10/10	🟩 9/10	🟩 10/10	🟥 0/10
	R	🟩 K	🟩 K	🟩 K	🟥 NK	🟩 K	🟩 K	🟩 K	🟥 NK	🟩 K	🟩 K	🟩 K	🟥 NK

Figure 5.1: Performance of Mutations on each RL Algorithm on CartPoleContinuous

***Green** indicates that the mutant was detected and killed. **Red** indicates that the mutant* *was not detected and killed.*

As explained above in the evaluation metrics section, the number of times the ratio for each agent is below the threshold θ were recorded, and can be seen in the table above. When the cumulative number is below 70% (7/10), we considered the mutant "not sufficiently detected or killed", denoted by the red marker.

5.3.1 Overall Effectiveness of Mutation Testing

As introduced previously, **O1** focuses on the analysis of the overall effectiveness of the Mutation Testing process.

We found that for each algorithm, the detection of mutants performance was fairly average when evaluated with the metrics introduced previously.

When considering both metrics, i.e. both metrics consider the mutant killed:

Algorithm	**AVG & R**
DDPG	50% [6/12]
PPO	50% [6/12]
A2C	67% [8/12]

Table 5.1: No. of mutants considered killed by both metrics

Based on the results above, on average only 56% of mutants were killed when considered under both metrics. It can also be seen when looking at Figure 5.1, that in majority of cases both metrics agree on whether a mutant is killed.

When considering the metric individually, i.e. how many mutants each metric considered killed:

Algorithm	**AVG**	**R**
DDPG	58% [7/12]	75% [9/12]
PPO	50% [6/12]	50% [6/12]
A2C	67% [8/12]	75% [9/12]

Table 5.2: No. of mutants considered killed by each metric

Based on the results above, on average AVG considered 59% of the mutants killed, while R considered 67% of the mutants killed, meaning on average more mutants were considered killed under the R metric in comparison to the AVG metric.

This was an expected result, as the Tambon et al, who originally proposed the R metric, intended it to allow for more mutants to be killed, basing it solely on the reward value.

Looking specifically at the mutants from the results, it can be seen that a number of mutants were consistently killed under both metrics and for all algorithms, these were:

- **Environment Mutants:** Random and Mangled
- **Agent Mutants:** Incorrect Loss Function and Missing State Update
- **Policy Mutants:** POC - SGD

Based on the results above, it can be seen that there does not seem to be any bias in the type of mutant killed. However, below are the most probable reasons as to why these specific mutants were able to be detected and killed.

Random [Ra]: Return next state and reward which are related to each other but not the action taken. Mangled [M]: Returns next state and reward which are not related.

Random & Mangled: both the Ra and M mutants have a direct impact on the learning process of the agent. By manipulated and corrupting the observation data passed back to the agent from the environment, they disrupt and distort the decisions the policy makes, and thus affect the agent's next moves. Therefore, it follows that these mutants would be detected immediately and killed, especially when compared against the healthy agent.

Incorrect Loss Function [ILF]: Uses wrong formula for loss function. Missing State Update [MSU]: Does not update agent's observations.

Incorrect Loss Function & Missing State Update: both the ILF and MSU mutants have a direct impact on the learning process of the agent. Changing the loss function (ILF), can lead to significant changes such as corrupted q-values etc... Whereas not updating the agent's observation (MSU), disrupts the agent's learning process, by not passing in the acquired information from each action taken. Therefore, it follows that these mutants would be detected immediately and killed, especially when compared against the healthy agent.

Policy Optimizer Change [POC] - SGD: Varying optimizer for agent's neural network

Policy Optimizer Change - SGD: the use of the SGD optimizer was easily detected in comparison to the healthy agent because, the default optimizer on the Stable-Baselines3 implementation of PPO, A2C, and DDPG is the Adam optimizer. Therefore, it follows that using a policy optimizer like SGD which converges at a much slower rate than the Adam optimizer (among other differences), would result in the mutant being detected immediately and killed.

A number of mutants were also frequently not killed under both metrics and for all algorithms:

- **Environment Mutants:** Repeat and Reward Noise
- **Agent Mutants:** Missing Terminal State
- **Policy Mutants:** POC - Adam

Based on the results in Figure 5.1, it can be see that the Repeat mutant goes undetected in all situations, followed shortly by Reward Noise and POC - Adam, which were only killed in one instance respectively, and Missing Terminal State which was killed only in two instances. Below are the most probable reasons as to why these specific mutants were able to go undetected.

Repeat [R]: Returns next state and reward relating to previous observation

Repeat: returning the previous observation's next state and reward may not have a drastic effect on the learning process for these specific algorithms. This is because DDPG, PPO, and A2C are all policy-based algorithms which employ the use of replay/rollout buffers to store past experiences, which are then used to adjust the policy parameters through each iteration. As a result these algorithms are less affected by the most recent state-action-reward data, and thus, the agent would still be able to the learn the task from the past experiences stored in the buffer. Therefore, it follows that this mutant would not necessarily be detected.

Reward Noise [RN]: Returns reward with added noise to agent

Reward Noise: the additional of noise to the reward signal may not have a drastic effect on the agent's learning process, especially is the noise level is too low. The agent may still be able to learn the overall task, despite the slight difference in reward estimations. Therefore, it follows that this mutant would not necessarily be detected.

Missing Terminal State[MTS]: Does not save the terminal state observation

Missing Terminal State: not saving the terminal state observation of an episode, may not have a drastic effect on the agent's learning process. This is because, the agent would still be able to learn from all other observation acquired through the episode but would simply not be able to recognise the terminal state efficiently. Therefore, it follows that this mutant would not necessarily be detected.

Policy Optimizer Change [POC] - Adam: Varying optimizer for agent's neural network

Policy Optimizer Change - Adam: the use of the Adam optimizer was not detected in majority of cases in comparison to the healthy agent because, the default policy optimizer on the Stable-Baselines3 implementation of PPO, A2C, and DDPG is the Adam optimizer.

Therefore, it follows that there would be no major difference between the mutant and healthy agent. However, it can be seen that in one case, the mutant was killed, but this was due to a discrepancy in the R metric's statistical test (which we will touch on in the *Effectiveness of Evaluation Metrics* section).

Our final response for **O1** can be summarised as follows:

Overall, it can be seen that the detection and killing of mutants was fairly average across all of the algorithms and under both evaluation metrics. Though there are specific mutants which were consistently killed, i.e., Random, Incorrect Loss Function, and specific mutants which were consistently not killed, i.e., Repeat, Missing Terminal State, there does not seem to be any bias in relation to the type of the mutants. It can also be seen that on average both evaluation metrics agree that a mutant is considered killed.

5.3.2 Effectiveness of Mutation Testing on PPO & A2C on Continuous vs Discrete Environments

Continuing on to **O2**, which focuses on the performance of Mutation Testing on the PPO and A2C algorithm on CartPoleContinuous vs CartPole classic (discrete).

DRL Algorithm	'Killing' Metric	Mutations – CartPoleContinuous										
		Ra_1.0	RN_1.0	M_1.0	R_1.0	ILF	NDF	MSU	MTS	PAC_Sigmoid	PAC_ReLU	POC_SGD
PPO	AVG	10/10	2/10	10/10	4/10	10/10	6/10	10/10	0/10	10/10	0/10	10/10
	R	K	NK	K	NK	K	NK	K	NK	K	NK	K
A2C	AVG	10/10	4/10	10/10	4/10	10/10	7/10	10/10	0/10	10/10	9/10	10/10
	R	K	K	K	NK	K	K	K	NK	K	K	K

Figure 5.2: PPO & A2C on CartPoleContinuous

DRL Algorithm	'Killing' Metric	Mutations - CartPole										
		Ra_1.0	RN_1.0	M_1.0	R_1.0	ILF	NDF	MSU	MTS	PAC_Sigmoid	PAC_ReLU	POC_SGD
PPO	AVG	9.5/10 (19/20)	0/10 (0/20)	10/10 (20/20)	0/10 (0/20)	10/10 (20/20)	6.5/10 (13/20)	10/10 (20/20)	0.5/10 (1/10)	2.5/10 (5/20)	0.5/10 (1/20)	10/10 (20/20)
	R	K	NK	K	NK	K	K	K	NK	K	NK	K
A2C	AVG	10/10 (20/20)	2.5/10 (5/20)	9.5/10 (19/20)	1.5/10 (3/20)	10/10 (20/20)	3.5/10 (7/20)	10/10 (20/20)	0.5/10 (1/20)	2/10 (4/20)	10/10 (20/20)	10/10 (20/20)
	R	K	K	K	NK	K	K	K	NK	K	K	K

Figure 5.3: PPO & A2C on CartPole*

*The original source material trains 20 agents to obtain their results, whereas this experiment train 10 agents, therefore we had to adapt their results by changing the ratios to be a proportion of 10 rather than 20.

Looking at the results above, it can be seen the PPO and A2C perform virtually the same on both the discrete and continuous implementations of the CartPole environment - aside from some occasional variance in the ratios. However there are a few significant differences:

1. No Discount Factor (NDF) - PPO: using the R metric on the PPO algorithm, the NDF mutant is not considered killed on the continuous environment, but is considered killed on the discrete environment. Looking at the values in the AVG metric on both environments, they seem to hover around the same value (6/10). Therefore, is it fair to assume that this is most likely a matter of results being on the borderline, which could result in a positive response in one case and a negative in another. Therefore, we do not necessarily deem this significant.

2. No Discount Factor (NDF) - PPO: using the AVG metric on the A2C algorithm, the NDF mutant is considered killed on the continuous environment, but is not considered killed on the discrete environment. Once again, by looking at the AVG result on the continuous environment, it can be seen that the result hovers on the cut-off point of the AVG ratio (70%), so again we would say that it is fair to assume that this is most likely a matter of results being on the borderline. Therefore, we do not necessarily deem this significant.

3. Policy Activation Change - Sigmoid (PAC-Sigmoid) - A2C & PPO: using the AVG metric on both the PPO and A2C algorithms, the PAC-Sigmoid mutant is considered killed on the continuous environment, but is not considered killed on the discrete environment. This could be due to the nature of the policy network used in a continuous action spaces. Generally functions such as ReLU are used as they provide unbounded real-valued outputs, which can represent continuous actions without the use of additional transformations. However, since Sigmoid constricts the output range to [0,1], this can have a significant impact on the output. Therefore, we would say it is fair to assume that this difference is significant [32].

Our final response for **O2** can be summarised as follows:

The performance of the Mutation Testing on the PPO and A2C algorithms, on both the discrete and continuous implementations of CartPole, was virtually the same, other than a few small discrepancies. We found that upon further inspection, these differences in performance, were mainly due to split decisions which could have gone in either direction, due to the results being on the borderline of the cut-off point. We did found however, that the discrepancy of policy activation mutant performance, could be due to the nature of how policy networks are applied in continuous environments in comparison to discrete environments, thus explaining the results. Overall, it does seem that Mutation Testing is as effective on continuous environments, as it is on discrete environments.

5.3.3 Effectiveness of Evaluation Metrics

Finally, we conclude on **O3**, which focuses on the effectiveness of the two metrics utilised during the evaluation process. As we mentioned at the beginning of the section, Tambon et al, propose the use of the two evaluation metrics: AVG - RL adapted mutation score, and R

- reward-based statistical test. The AVG metric was originally proposed by Lu et al [31], which was then used by Tambon et al in their paper. They then went on to propose the R metric as an alternative which would allow for more mutants to be killed and would not be as limited by the stochasticity of the RL problems in comparison to AVG.

Through the use of both metrics throughout this experiment, we found them both to be highly effective and useful in determining whether a mutant should be considered killed. Just as Tambon et al suggests, the R metric did allow for a higher number of mutants to be considered killed, in comparison to AVG. However, we believe it was not excessive in this fact, as in majority of the situations the AVG and R metric were in agreement about whether a mutant should be considered killed. The main difference in their classifications, lied whenever the AVG ratio value was near the cut-off point. Although both metrics are highly effective, they do both have their own shortcomings.

The use of the threshold θ in the AVG metric, is highly important, as it is the main deciding factor as to whether a mutant is considered killed or not. As a result of this, the AVG metric could be rendered useless if an unsuitable threshold value is chosen. Therefore, it is imperative that when using the metric an appropriate threshold is selected.

Due to the nature of the R metric, it relies solely on the statistical analysis of the reward distribution of an agent - healthy or mutant. However, this can be problematic in situations where the reward distribution may not exhibit statistically significant differences between healthy and mutant agents. This can then lead to inconclusive results which in the case of this experiment were considered as "Not Killed". Therefore, although this problem may be inconsequential in majority of situations, in the cases of outliers it could be fairly significant.

Our final response for **O3** can be summarised as follows.

We believe that both metrics were highly effective and useful during the evaluation process. Despite their shortcomings, we believe that if used with the proper parameters, they can be used to aid further mutation testing evaluations. We believe that the two metrics work best in conjunction with one another, as often the results support the final conclusion. We also believe that the use of the two metrics makes the final conclusion of the evaluation more reliable and convincing overall.

5.4 Summary

In this chapter we explained and detailed the evaluation of the experiment. We introduced the objectives we would be focusing on, and the metrics for the evaluation. We provided the results of the experiment, and analysed them under the objectives stated.

O1, focused on the analysis of the performance of the Mutation Testing process across all algorithms. Based on the results, we were able to conclude that the performance was fairly average across the board. We were able to identify particular mutants which had a significant impact of the agent's learning process - the mutants that were consistently killed, and the mutants which had the least significant impact - the mutants that went consistently not killed. We were able to conclude that although MT is an effective testing technique, there are potential faults that could go undetected in an RL system, under its use.

O2, focused on the comparison of the performance of Mutation Testing on the PPO and A2C algorithms on a continuous environment, against a discrete environment. Based on the results, we were able to conclude that MT is just as effective in a continuous action spaces, as it is in discrete action spaces. Despite the appearance of a few outliers, the performance overall was virtually same. These outliers could also be reasonably attributed to borderline split results, and differences in how different action spaces utilise the policy network.

O3, focused on the effectiveness of the evaluation metrics used: AVG - RL adapted mutation score, and R - reward-based statistical test. Based on the results, we were able to conclude, that both metrics despite their individual shortcomings are highly effective and of great use in the area of Mutation Testing of RL agents. We further stated, that we believe the best use of these metrics would be in conjunction with one another, as in majority of cases they arrive at the same conclusion. Therefore, using both metrics could provide increased confidence in the results, and make the conclusions appear more reliable.

To conclude, through this evaluation period we were able to carry out an analysis of the results of the experiment under the stated objectives. The conclusion of these evaluation objectives contributed to the main objective of the thesis, which was to expand upon the existing work done in adapting the Mutation Testing technique for application on a wider range of algorithms, and environments in RL. By achieving the evaluation objectives, we were able to verify that the adaptation of the MT method was effective in the wider areas, this thesis aimed to reach. This then opens the door for further work and research to be carried out not only in Mutation Testing, but into all testing techniques applicable to Reinforcement Learning.

6 Conclusion & Future Works

In this chapter, we will be reiterating the main points of this thesis, which will include a summary of each chapter. We will also expand on potential future works based on the experiment and this thesis.

6.1 Conclusion

Through this thesis, we explored the Software Testing of Reinforcement Learning Agents, with a specific focus on Mutation Testing. In chapter one, we introduced the paper, alongside the problem statement, the main aim/objective, the specific contribution this work has made, finally concluding with the structure of the thesis.

We continue into chapter two, where we introduced some background on the thesis' field of study, specifically Reinforcement Learning and software testing. We provided detail on the structure of RL problems, the commonly used RL algorithms, as well algorithms that are relevant to this thesis. We also introduced various software testing methods used in Machine Learning, and how that expands to Reinforcement Learning, before finally concluding on some proposed RL specific software testing methods.

In chapter three, we specified the design of the experiment, including the algorithms, environment and mutations used - also providing further details relating to the parameters utilised in each component. This is then followed up in chapter four, where we presented the specific details on the implementation including the frameworks, libraries, and code used in executing the experiment in accordance with the design previously specified.

Finally, we presented the results and carried out their evaluation in chapter four. We introduced the main objectives of the evaluation, alongside the evaluation metrics being utilised. We analysed the results under the objectives and added personal insights to provide further clarification.

6.2 Future Works

This thesis builds upon the work originally proposed by Tambon et al, which focused on the 'Mutation Testing of Reinforcement Learning Agents Based on Real Faults'. Expanding on their work, we were able to apply their approach to a new RL algorithm which they had not previously tested. We were also able to adapt their existing environment processing to facilitate a continuous action space, in comparison to their original implementation which only allowed for discrete action spaces.

Based on this, we believe that future works in this area, can expand on the work that has been carried out. This could be achieved by implementing and testing using different continuous action space based environments. As we used an adaptation of the CartPole environment which is originally discrete in the experiment, there may be some added complexity in environments which are designed to be continuous - i.e., MountainCar, Pendulum - which we did not encounter. Therefore, we believe it would be a good idea to implement further continuous environments to see how they perform in comparison to the discrete environment which have already been tested.

Implementing other algorithms could also be a potential avenue to pursue, as this would mean that the Mutation Testing approach is more extensively tested on a wider range of algorithms, which would enable any discrepancies or bugs to be identified and addressed early on.

Overall, there are a lot exciting avenues to explore in the area of software testing of Reinforcement Learning agents, and we believe that as more work and further research is being carried out, this area will become well-documented and easily accessible to all those who are interested.

Bibliography

[1] F. Tambon, V. Majdinasab, A. Nikanjam, F. Khomh, and G. Antonio, "Mutation testing of deep reinforcement learning based on real faults," 2023.

[2] R. S. Sutton and A. Barto, *Reinforcement learning: An introduction*. The MIT Press, 2020.

[3] C. Leo, "Reinforcement learning 101: Building a rl agent," Feb 2024. [Online]. Available: https://towardsdatascience.com/reinforcement-learning-101-building-a-rl-agent-0431984ba178#04c4

[4] S. Shoham and Y. Vizel, *Computer aided verification 34th international conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings*. Springer, 2022.

[5] C. J. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3/4, 1992.

[6] D. Kumar, "Ppo algorithm," Feb 2024. [Online]. Available: https://medium.com/@danushidk507/ppo-algorithm-3b33195de14a

[7] S. Dhumne, "Deep q-network (dqn)," Jun 2023. [Online]. Available: https://medium.com/@shruti.dhumne/deep-q-network-dqn-90e1a8799871

[8] D. Karunakaran, "Reinforce-a policy-gradient based reinforcement learning algorithm," Jun 2020. [Online]. Available: https://medium.com/intro-to-artificial-intelligence/reinforce-a-policy-gradient-based-reinforcement-learning-algorithm-84bde440c816

[9] A. Marekar, "How ddpg (deep deterministic policy gradient) algorithms works in reinforcement learning ?" Jun 2022. [Online]. Available: https://medium.com/@amaresh.dm/how-ddpg-deep-deterministic-policy-gradient-algorithms-works-in-reinforcement-learning-117e6a932e68

[10] "Deep deterministic policy gradient algorithm: A systematic review," Nov 2023. [Online]. Available: https://www.researchsquare.com

[11] M. Wang, "Advantage actor critic tutorial: Mina2c," Jun 2023. [Online]. Available: https://towardsdatascience.com/advantage-actor-critic-tutorial-mina2c-7a3249962fc8

[12] T. Simonini, "Advantage actor critic (a2c)," Jul 2022. [Online]. Available: https://huggingface.co/blog/deep-rl-a2c

[13] A. D. Tovar, "Advantage actor critic (a2c) implementation," Jan 2020. [Online]. Available: https://medium.com/deeplearningmadeeasy/advantage-actor-critic-a2c-implementation-944e98616b

[14] D. Marijan and A. Gotlieb, "Software testing for machine learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 13 576–13 582, 04 2020.

[15] Synopsys, "What is fuzz testing and how does it work?" [Online]. Available: https://www.synopsys.com/glossary/what-is-fuzz-testing.html

[16] "Fuzz testing," Jul 2021. [Online]. Available: https://www.lakera.ai/blog/fuzz-testing

[17] M. Duffin, "7 types of adversarial machine learning attacks," Feb 2024. [Online]. Available: https://rareconnections.io/adversarial-machine-learning-attacks/

[18] "Adversarialmachinelearning?" Apr 2024. [Online]. Available: https://viso.ai/deep-learning/adversarial-machine-learning/

[19] F. G. Zambon, "Differences between supervised, unsupervised and reinforcement learning." Feb 2024. [Online]. Available: https://medium.com/@zambonfrancescogiorgio/the-main-differences-between-supervised-learning-unsupervised-learning-and-reinforcement-learning-be

[20] P. Pedamkar, "Supervised learning vs reinforcement learning: 7 valuable differences," Jun 2023. [Online]. Available: https://www.educba.com/supervised-learning-vs-reinforcement-learning/

[21] A. Zolfagharian, M. Abdellatif, L. C. Briand, M. Bagherzadeh, and R. S, "A search-based testing approach for deep reinforcement learning agents," *IEEE Transactions on Software Engineering*, vol. 49, no. 7, p. 3715–3735, Jul. 2023. [Online]. Available: http://dx.doi.org/10.1109/TSE.2023.3269804

[22] M. Biagiola and P. Tonella, "Testing of deep reinforcement learning agents with surrogate models," *Association for Computing Machinery*, vol. 33, no. 3, mar 2024. [Online]. Available: https://doi.org/10.1145/3631970

[23] A. Bignold, F. Cruz, R. Dazeley, P. Vamplew, and C. Foale, "An evaluation methodology for interactive reinforcement learning with simulated users," *Biomimetics*, vol. 6, no. 1, 2021. [Online]. Available: https://www.mdpi.com/2313-7673/6/1/13

[24] "Dlr-rm/rl-baselines3-zoo," 2019. [Online]. Available: https://github.com/DLR-RM/rl-baselines3-zoo

[25] "Cart pole - gym documentation," 2022. [Online]. Available: https://www.gymlibrary.dev/environments/classic_control/cart_pole/

[26] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-13, no. 5, pp. 834–846, 1983.

[27] S. Nagendra, N. Podila, R. Ugarakhod, and K. George, "Comparison of reinforcement learning algorithms applied to the cart-pole problem," in *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, Sep. 2017. [Online]. Available: http://dx.doi.org/10.1109/ICACCI.2017.8125811

[28] A. Penkin, "Cartpolecontinuous," Apr 2018. [Online]. Available: https://github.com/SSS135/pytorch-rl-kit/blob/04cd026116bfbd7353274f8dbb4951cddfc66e6b/ppo_pytorch/common/cartpole_continuous.py

[29] "Flowss/rlmutation," 2022. [Online]. Available: https://github.com/FlowSs/RLMutation

[30] "Stable baselines 2.10.3a0 documentation," 2018. [Online]. Available: https://stable-baselines.readthedocs.io/en/master/

[31] Y. Lu, W. Sun, and M. Sun, "Towards mutation testing of reinforcement learning systems," *Journal of Systems Architecture*, vol. 131, p. 102701, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762122001977

[32] D. Lee, "Comparison of reinforcement learning activation functions to improve the performance of the racing game learning agent," Oct 2020. [Online]. Available: https://doi.org/10.3745/JIPS.02.0141