University of Essex
Department of Mathematical Sciences

MA981: DISSERTATION

Bipedal Robot Walking using Reinforcement Learning

Jayani Dipalkumar Bhatwadiya

Supervisor: Dr. Alexei Vernitski

September 10, 2021
Colchester

ABSTRACT

Reinforcement Learning is a framework for creating controllers for dynamical system that can be used in a variety of situations. A policy is learned by trial and error. This makes it ideal for systems like walking robots that are difficult to handle using traditional control methods. The usage of Deep Neural Networks with reinforcement learning has demonstrated incredible results for management of high system controls. This method is known as *Deep Reinforcement Learning*. (Rastogi, 2017) The purpose of this dissertation is to use a deep reinforcement learning algorithm for a bipedal robot, such as the Deep Deterministic Policy Gradient (DDPG) and Twin-Delayed Deep Deterministic Policy Gradient (TD3) and both algorithms' performance was compared and evaluated. Training with TD3 was, in general, faster than training with DDPG, as well as also far more reliable for such kind of motion tasks. Using OpenAI Gym's built-in *Box2D* environment, I conducted trials and successfully trained an agent to walk on a flat surface.

KEYWORDS : Reinforcement Learning, Deep Deterministic Policy Gradient, Twin-Delayed Deep Deterministic Policy Gradient, OpneAI, BipedalWalker.

ACKNOWLEDGMENTS

I would want to give my deepest appreciation to Dr. Alexei Vernitski for his kindness, as well as his helpful advice and guidance over the period of my research. I feel myself extremely pleased to get him as my supervisor.

I would want to express my gratitude to our Mathematical Department for all of their help and support during my graduate studies. I am grateful to all of my teachers for their constant support and advice over the course of the year.

Thank you to my parents and siblings for their consistent assistance and motivation throughout my graduate studies.

Jayani Bhatwadiya

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

One or more of our regular household duties may be taken over by robots in the near future. Humanoid robots must have stable walking movements in order to blend in with humans. The way humans walk is quite efficient. Because of their energy-efficient movement patterns they can traverse any terrain. Humanoid robots have made great advances in recent years, still they have difficulty walking and it is a major drawback for the field of robotics. (Benbrahim, 1996; Rastogi, 2017) The recent survey of **DARPA Robotics Challenge**, demonstrated that humanoid robos are still unable to walk effectively over plain surface. (Rastogi, 2017)

So, this is why in my thesis I want to use the Reinforcement Learning approach to address the challenge of bipedal walking. To tackle this problem, I would prefer to take an Actor Critic approach. For my research, I proposed two key algorithms that produced outstanding results in a continuous task.

The chapter will provide an introduction to the topic, what the actual goal of this study is, as well as the best strategy to dealing with this problem and its suitable consequences. Finally, Thesis structure was created at the end of this section.

1.1 Problem Statement

It has long been an aim in the field of Artificial Intelligence to create a machine that is clever enough to do every activities that a human can complete. Artificial General Intelligence (AGI)

is a term used to describe this. AGI simulates human intelligence in software, allowing it to find a solution to such an unknown issue. Human control activities like as having to walk, opening doors, and so on are accomplished easily by the human brain, although it still remain amongst the most difficult jobs to simulate for practical implementation in robotics. (Garg, 2018) New attention in and developments with in subject of Deep Reinforcement Learning, on the other hand, have become a huge benefit to field of robotics. (Garg, 2018) The objective of this project is to teach an agent to walk on a flat surface using Deep Reinforcement Learning. Humans will be able to construct more capable robots. Such robot could be used to do risky activities such as mountainous climbing and rescue operations.

The Google DeepMind research released a paper and video depicting virtual agents taught to negotiate a series of difficult terrains. (Garg, 2018; Heess, TB, Sriram, Lemmon, Merel, Wayne, Tassa, Erez, Wang, Eslami, Riedmiller, and Silver, 2017) The reinforcement learning method was used to train these agents. In the beginning, my actual objective was to train a humanoid robot to effectively walk in an advanced simulator in a similar fashion. However, given the existing limits on compute techniques and support available, it became clear that this task was unfeasible. (Garg, 2018) As a result, I narrow down the project scope to teach an agent to walk in the OpenAI Gym's simulator.

1.2 Research Goal

In order to achieve the main objective of this thesis, numerous supplemental goals have to be designed and completed. The below list summarises the subgoals which I established :

- Learn about classical reinforcement learning and the principles that go along with my thesis.
- Discover the benefits of deep reinforcement learning like a preliminary to deployment.
- Learn how to use the OpenAI Gym toolkit, its requirements and its different environments.
- Use deep reinforcement learning approach to successfully establish walking actions in the *Bipedal Walker-v2* of OpenAI Gym environment.

1.3 Approach

My thesis' important aim is that I can walk a bipedal robot across a specified terrain. In my thesis, I explore a variety of techniques for training the agent, including both value-based and policy-based approaches, but at last I offer an actor-critic mechanism that enables for both value and policy problems.

In this thesis, I introduced a model-free and actor-critic technique using *Deep Deterministic Policy Gradient* and *Twin-Delayed Deep Deterministic Policy Gradient* to improve efficiency in continuous control tasks. The method described is general and may be used to a wide range of robotic activities.

1.4 Contribution

- As part of this research, I have employed OpenAI's present environments of reinforcement learning and successfully applied one of its environment from the Gym framework.
- The significant contribution with this research is to assess the impact of two Deep Reinforcement Learning algorithms: Deep Deterministic Policy Gradient (DDPG) and Twin-Delayed Deep Deterministic Policy Gradient (TD3) in training an agent can discover the movement actions required to succeed in a given environment.
- Upon success of this work, I aim to release the program code for the implementations as it involve distinct action plans.

1.5 Thesis Structure

Chapter 2 –Background

This chapter provides background information about the present research that is necessary to comprehend before beginning the dissertation.

Chapter 3 –The Reinforcement Learning Problem

This chapter consists of a detailed explanation to Reinforcement Learning by explaining the mathematical fundamentals associated with it. This chapter also covers basic terms that are related with Reinforcement Learning.

Chapter 4 –Deep Reinforcement Learning

This chapter contains information about deep reinforcement learning, This chapter also discuss about the proposed technique and its algorithms with their mathematical explanation.

Chapter 5 –Experimental Design and Implementation

This chapter describes the concept and implementation of the research, as well as the experimental setup I deployed.

Chapter 6 –Results and Discussion

This chapter evaluates the effectiveness of both recommended algorithms and summarizes the outcomes.

Chapter 7 –Conclusions

This chapter summarises and concludes the work presented in the thesis, as well as providing suggestions for future studies.

CHAPTER 2

Background

2.1 Introduction

The adaptation of reinforcement learning towards robots is a new subject in the robotic field. (Ribeiro, 2019) The background section approached on three perspectives:

1. Machine learning and its various relevant branches.
2. Reinforcement learning in robotics
3. Simulators and frameworks

2.2 Machine Learning

Artificial Intelligence is an area of computer science which deals with the creation of intelligent machines that operate and behave similarly to humans. Machine Learning is a sub field of Artificial Intelligence that allows computers to understand new tasks without having to be pre-programmed. It is also the study of algorithms concepts that computers utilise to do tasks successfully without explicit instructions. Without any direct programming, a mathematical analysis was conducted based on the information, termed as "training data", to make predictions or judgments. These algorithms further teach themselves when new data is provided. (Ribeiro, 2019)

The use of machine learning technology is allowing a transition in problem-solving from

analytic to a powerful data-driven approach. Computer programs can train models from training patterns and evaluate new data's outcomes. (Huang and Ma, 2018)

Machine Learning (ML) is gaining popularity these days, with numerous applications ranging from big data pattern detection to automation and entertainment. (Mellatshahi, 2020)

One of the most significant elements of robotics research right now is machine learning. Moreover, in the creation of learning systems, the more complex idea of machine learning linked with optimization has been proven to be beneficial. (Mosavi and Varkonyi, 2017)

Machine learning is mainly separated into three branches given in the (Figure 2.1 from (Shewan, 2021)), each of these sub-branches can be further split.

- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning

2.2.1 Supervised Learning

Supervised learning is a sort of machine learning for which machines are trained with labelled training data and then used to predict output.(Jaiswal, 2015) The objective of supervised learning in machine learning is to train a function that translates an input to an output using sample input-output pairs. (Han, Kamber, and Pei, 2011)

A supervised learning algorithm examines the training data before generating a function that may be used to classify.(Ribeiro, 2019) The training data is analysed and an inferred function is constructed using a supervised learning approach. Additional samples can be mapped or predicted using the training model, or inferred function. (Alpaydin, 2014)

In the great majority of real-world machine learning applications, supervised machine learning is used. (Brownlee, 2016) Use supervised machine learning to transfer the function from the input to the output whenever input data (x) and target data (Y) are available. (Brownlee, 2016) The main aim of supervised learning is to develop model of the type $y = f(x)$ that can predicts outputs(Y) from inputs (x). (Siebel, 2012a)

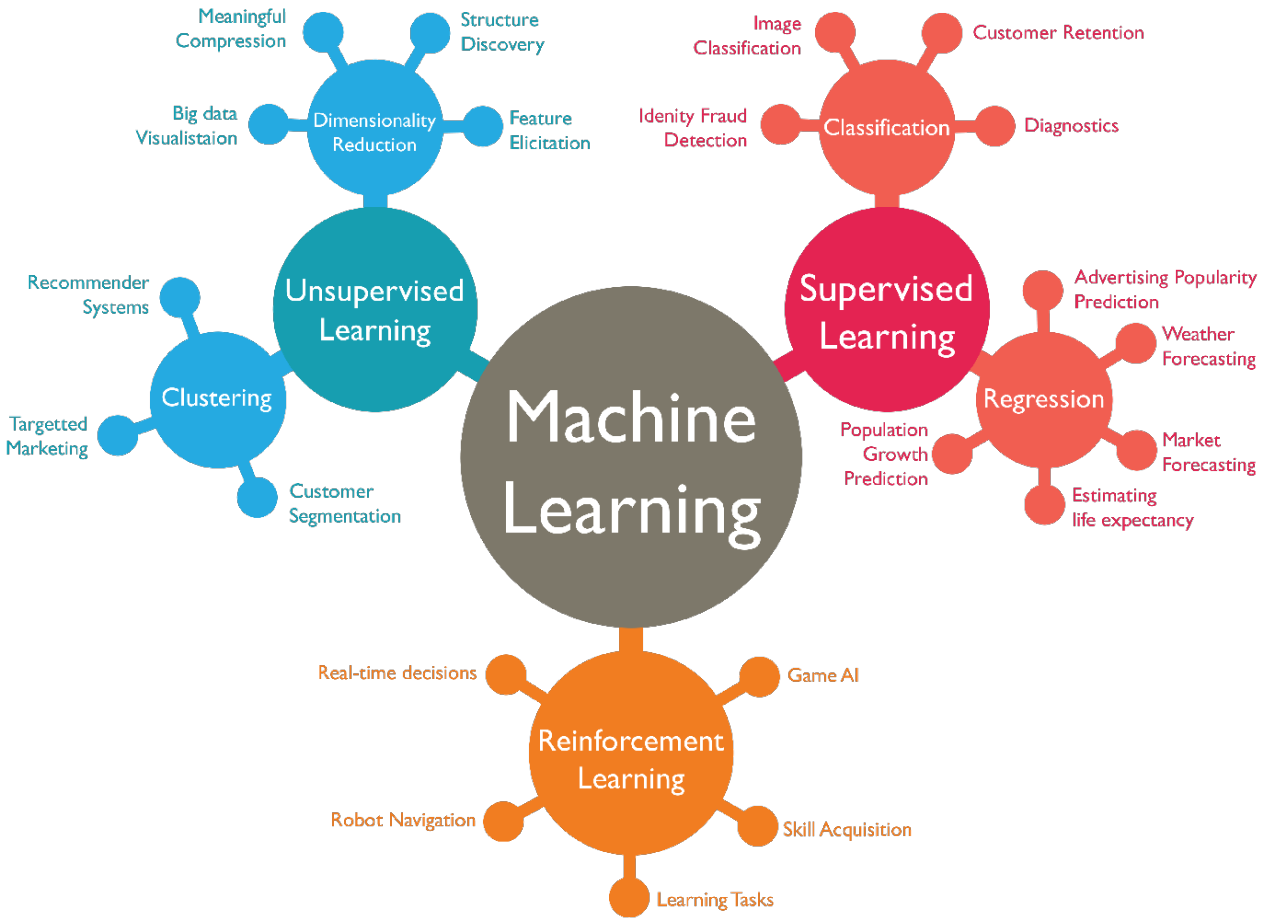

Figure 2.1: Machine Learning branches and respective sub-branches

Supervised learning approaches are divided into two categories. (Siebel, 2012a) Classification is the first type. Techniques for classifying objects predict categorical outputs, such as whether or not an object is a cat or dog. Regression is the second type. Regression algorithms are used to predict continuous quantities, such as sales forecasts for the coming week. (Siebel, 2012a)

2.2.2 Unsupervised Learning

Unsupervised learning model is trained using unlabelled data without any help of human. (Han, Kamber, and Pei, 2011) So, this algorithm is able to categorise and group the data without any external information to perform some task. (Burns, 2010)

When there is no sufficient data to train a supervised model, unsupervised learning can be applied to tackle the problem. (Siebel, 2012b) Only input data (x) and no outcome variables (Y) are provided in unsupervised learning.

Unsupervised learning tries to better explain the data's fundamental mechanism or distribution by modelling it. (Brownlee, 2016) Unsupervised learning has more in common with data mining than with actual learning. There is really no right or wrong output with unsupervised learning method. (Bell, 2014; Xiao, 2015) It indicates we are more interested with whatever patterns and outcomes appear in general and what does not after the machine learning model is used. (Acharyya, 2008; Geoffrey Hinton, 1999; Hastie et al., 2004; Xiao, 2015) Unsupervised learning enables the system to identify content on its own from large data sets.

Compared to supervised learning systems, unsupervised learning algorithms can handle more complicated processing tasks. (Burns, 2010) Clustering and dimensionality reduction are two types of unsupervised learning issues.

2.2.3 Reinforcement Learning

Machine learning is the use of set of data or past data to increase a performance metrics. Model learning is the process of optimising the parameters of a partially specified model using the training data or prior observations. (Shantia, 2011) The system's output in applications like navigating, grabbing, and exploring is a sequence of decisions. Single acts are really not essential in this situation; what matters is having an overall strategy on how to get to the objective given the existing circumstances. This type of algorithmic learning is termed reinforcement learning. (Alpaydin, 2014; Kaelbling, Littman, and Moore, 1996; Shantia, 2011)

Real-time learning enables an agent that discover optimum actions within the environment via multiple rounds of try and error. (Shantia, 2011; Zhu, Yu, Gupta, Shah, Hartikainen, Singh, Kumar, and Levine, 2020) The goal of reinforcement learning is to develop a specific formula known as a *policy* that accepts the agent's current state as input and produces the best possible action.

Throughout training, an agent gets rewarded depending on how near the policy's action will be to the target action. This policy is made depending upon those rewards across several rounds in order to better identify appropriate actions. (Agarwal, Schuurmans, and Norouzi, 2019; Shantia, 2011)

The usage of behavior-based models is among the most well-known techniques of executing

tasks in robotic. (Arkin, Arkin, et al., 1998; Shantia, 2011) Every action needs the completion of a number of activities, thus reinforcement learning is perhaps the most appropriate method for these kind of systems. (Shantia, 2011)

2.3 Reinforcement Learning in Robotics

In order for a robot to function effectively in a given environment, it has to be capable of understanding it, design and implement its activities, and receive feedback to help that everything is going as planned. Recognizing things, conversing, anticipating, human intents, and manipulating numerous items are all complex things for a machine to complete. Deep Learning has gained significant popularity for a variety of computer vision problems. (Pinto, 2019; Ribeiro, 2019) Recently, several academics have developed novel techniques to operate robots through reinforcement learning. (Ribeiro, 2019)

In response to the surprise success of deep reinforcement learning in video games, board games, and simulated control issues, researchers began to focus their attention on robotics. As a result of such simulation's performance, an agent can train in fewer episodes when doing the previously explained tasks. (Ribeiro, 2019)

When it comes to sample efficiency, model-based and model-free reinforcement learning are the two most prevalent techniques. In model-free learning, a policy is learned only from the rewards it receives as a result of engaging with the environment. Model-based learning is the process through which an agent attempts to identify an environment's pattern then use it to develop and enhance policies. However, this technique has a poorer performance compared to model-free learning. (Ribeiro, 2019)

A new study demonstrating a substantial sample efficiency for a model-free learning method was published. In this study, an off-policy actor-critic algorithm known as Deep Deterministic Policy Gradient and Twin-delayed Deep Deterministic Policy Gradient used to teach robots to walk in specified environment (Ribeiro, 2019)

2.4 Simulators and Frameworks

In 2016, OpenAI Gym, a new simulator for reinforcement learning algorithms featuring a robotics branch, was released. 'Gym' is a open source toolkit for developing and comparing reinforcement learning algorithms. Any numerical computation library like TensorFlow or Theano can be used. A key component of OpeanAI's Gym is the usage of platform to test reinforcement learning algorithms and standardise environment enabling results comparability. (Ribeiro, 2019) OpenAI Gym provides readymade enviroment of (1) Algorithms, (2) Atari, (3) Box2D, (4) Classic Controls, (5) MujoCo, (6) Robotics and (7) Toy Text. In this study, a *Box2D* simulator is used to move forward a bipedal walker, which is inbuilt simulation provided by OpenAI Gym.

2.5 Summary

In this chapter, supervised learning, unsupervised learning, and reinforcement learning are discussed. Reinforcement Learning depends purely upon trial-and-error procedures, making it ideal for robots to understand how to do new tasks without any human influence. (Ribeiro, 2019) Simulation environments are necessary to attain outcomes in consistent way. A detailed description about the environment which is going to be used in this research, is explained in chapter-5. A mathematical basis for reinforcement learning is given in the next chapter.

CHAPTER 3

The Reinforcement Learning Problem

3.1 Introduction

In this chapter, I discuss a theoretical explanation about reinforcement learning. The chapter is organised as follows:

1. I begin by discussing "traditional" Reinforcement Learning and how it can be mathematically represented as a Markov Decision Process. (section 3.2)
2. Afterwards, I go through some formal Reinforcement Learning ideas. (section 3.3)
3. Finally, I discuss about Q-Learning and the Bellman equation at the end of this chapter. (section 3.4)

3.2 Reinforcement Learning

Reinforcement learning (RL) (Kober et al., 2013; Sutton and Barto, 2018) is a common approach in which a system tries to choose the best actions to maximise rewards.(Isele, 2018)

Reinforcement learning is used when there is minimal information about the system to be managed. In this type of learning a certain task is assigned to the controller, When controller completes its task successfully, it receives positive rewards, and if it fails then it receives negative rewards or penalty. (Benbrahim, 1996)

It is used by a variety of software and computers to determine the best feasible action or path in a given circumstance.

Reinforcement learning is different from the supervised learning where supervised learning uses labelled data to train the supervise model, but reinforcement learning requires the agent to determine what to do in order to accomplish the task. (Bajaj, 2020) Reinforcement learning differs from unsupervised learning in terms of objectives. Unsupervised learning aims to find commonalities and differences across data sets, whereas reinforcement learning aims to create a suitable action model that maximises the agent's rising total reward. (Bhatt, 2018)

As illustrated in (Figure 3.1 from (Bhatt, 2018)), reinforcement learning approaches follow a fundamental Reinforcement learning (RL) paradigm.

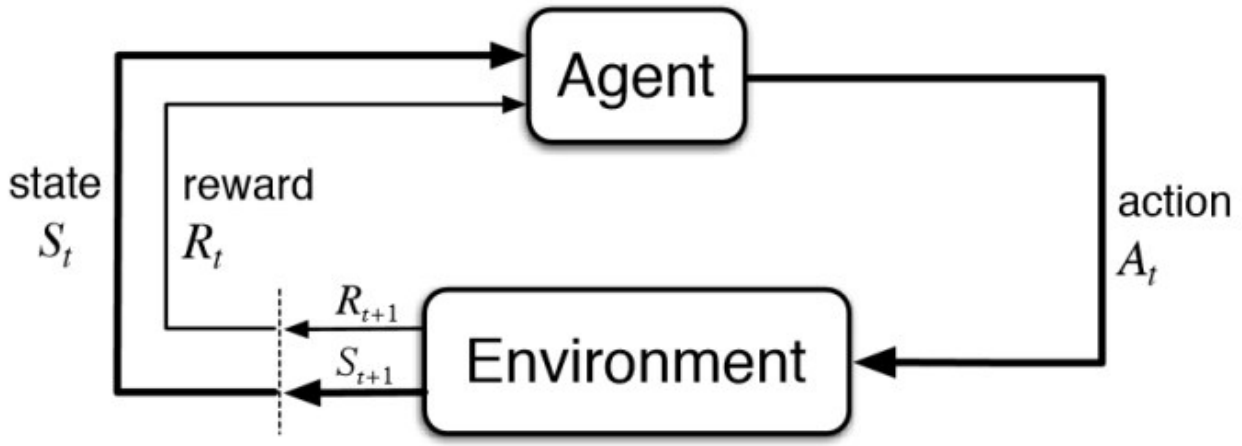

Figure 3.1: The basic reinforcement learning model

Reinforcement Learning (RL) is a way of acquiring knowledge through interacting with one's environment. (Woergoetter and Porr, 2008) The environment is seen as a Markov Decision Process (MDP). (Garg, 2018) MDP is comprise of a collection of environment states S and group of alternative actions $A(s)$. (Bhatt, 2018)

In Reinforcement Learning, we consider that we have no prior knowledge about the alternative actions or the collection of states for any specific environment. The agent must investigate the environment and learn about states and actions. As the training proceeds, the agent has a better understanding of the environment and is able to apply this information to conduct more positive actions. This is referred to as exploitation. (Garg, 2018)

The authors suggest that approaches to Reinforcement learning (RL) issues be divided into two classes, each characterised by a set of methods. One group utilises statistical approaches and dynamic programming techniques to assess the value of performing actions in specific states of the world, while the other searches the space of actions for one which works well in the environment.

Based on the prior state, the agent performs a specific action in dealing with internal action plan, and afterwards it interacts to discover the current state and appropriate rewards in the environment. This is considered as a transition. (Chen, 2018)

The pseudo code for REINFORCE Algorithm as below : (reference from (Fussell, 2018)

Algorithm 1 The REINFORCE Algorithm

Initialise: policy network π_θ with output μ and σ^2
while not converged **do**
  for e in epochs **do**
    start new environment E_e
    sample trajectory $\{\mathbf{s}_0, \mathbf{a}_0, r_1, \ldots, \mathbf{s}_{T-1}, \mathbf{a}_{T-1}, r_T\}$
    for r_t in sampled $\{r_T, \ldots, r_1\}$ **do**
      if t == T **then**
        $G(t) = r_t$
      else
        $G(t) \leftarrow r_t + G(t+1)$
      end if
    end for
    $\nabla J(\theta) = \frac{1}{T}\sum_{t=1}^{N}\sum_{a_i \in A}\frac{a_i - \mu_{a_i}(s_t))\nabla\mu_{a_i}(s_t)}{\sigma^2_{a_i}(s_t)}G(t)$
    backpropagate π_θ with $\nabla J(\theta)$
  end for
end while

3.2.1 Markov Decision Process

The focus of reinforcement learning is on how an agent interacts with environment in order to maximise reward. Furthermore, the entire procedure is subjected by Markov property as well as Markov decision processes (MDPs). (Tian, 2019)

In Markov decision process, a fixed states and actions be considered (MDPs).The agent observes a state and action at each moment. In Markov property referred as controlled dynamic system whose state transfers only based upon current state s and action a. (Edelkamp and

Schrodl, 2012)

$$\mathbb{P}(X_n = x_n | X_{n-1} = x_{n-1}, \ldots, X_0 = x_0) = \mathbb{P}(X_n = x_n | X_{n-1} = x_{n-1})$$

Markov decision process can be used to address optimization issues that aren't easily solved with dynamic programming or reinforcement learning. (Tian, 2019)

A Markov decision process has a five tuple, $\langle S, A, R, P, \rho_0 \rangle$ where,

- S denotes the set of all possible states,
- A denotes the set of all possible actions,
- $R : S \times A \times S \to \mathbb{R}$ stands for reward function, with

$$R(s, a, s^{'}) = \mathbb{E}(r_t | s_t = s, a_t = a, s_{t+1} = s^{'})$$

- $P : S \times A \to \mathcal{P}(S)$ would be the transition probability function, having $P(s'|s, a)$ the probability of transitioning in state $s^{'}$ when you begin from state s as well as take an action a,
- ρ_0 is denoted by initial state distribution.

3.3 Formal Definitions

State Space : Let the set of different agent states in the environment be $S = \{s_1, s_2, \ldots s_N\}$. This is referred to as *State Space*.

Action Space : Let the set of activities that the agent does be $A = \{a_1, a_2, \ldots a_N\}$. This is referred to as *Action Space*.

Transition Probability : Let the agent is in state $s \in S$ and performs some action $a \in A$. This *Transition Probability* referred to as $T(s^{'} | s, a)$.

Environment : This dynamically defines the outer world. The *Environment* involves everything the agent(s) engages with. It is designed for agent should appear as though it is a real-life situation. It is important to demonstrate an agent's performance, i.e., if it will work effectively in such a real-world application. (Odemakinde, 2019)

Reward Function and Reward : The *Reward Function* is a function that represents the reward values for a specific environment. This function is often context-sensitive and depends on which task is performed in the environment. The agent gains knowledge through the outcomes of its actions. A metric is required to compare the result of the actions, and this metric is referred as *Reward*. The reward earned from this environment is represented by $R(s, a)$ if an agent performs action $a \in A$ in state $s \in S$. (Garg, 2018)

Return : The agent also collects a total reward from it's own environment. Agent's aim is to maximise its total reward, often known as *return* G_t, and it is written as, (Rastogi, 2017)

$$G_t = r_t + r_{t+1} + r_{t+2} + \cdots + r_{N-1}$$

where N represents the episode's completion and t denotes the time index. The discounted returns is provided for ongoing activities, and it is obtained by, (Rastogi, 2017)

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

in which the discount factor is $\gamma \in [0, 1)$.

Policy : A *Policy* is represented by π_θ, is a strategy that connects each state $s \in S$ to the best action $a \in A$ in the particular environment. If we implement the actions specified by such a policy, then the expected reward will be maximised, this policy is considered as *optimal policy*. A *stochastic policy* provides a probability distribution for next best action, and it defined as $\pi_\theta\ (a|s) = P\ [a|s; \theta]$ while A *deterministic policy* specifies a single action to be performed. (Garg, 2018)

Value and Q-Value : A *Value*, commonly indicated by the $V^\pi(S)$, which informs about the long-term benefits of staying in state S. (Garg, 2018) The value equation for policy π an is as follows:

$$V_\pi(s) = \mathbb{E}_\pi(G_t|s_t = s) = \mathbb{E}_\pi\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k}|s_t = s\right)$$

The *Q-Value* reflects the total amount of reward the agent may expect for completing action a in state s, if the policy π is followed by an agent to determine its upcoming actions. (Garg,

2018) It is represented as $Q^{\pi}\ (s, a)$. The Q-value equation for policy π an is as follows:

$$Q_{\pi}(s,a) = \mathbb{E}_{\pi}(G_t|s_t = s, a_t = a) = \mathbb{E}_{\pi}\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k}|s_t = s, a_t = a\right)$$

Model-Free and Model-Based : A Reinforcement Learning (RL) algorithm is divided in two categories : (1) Model-Free and (2) Model-Based.

An algorithm that calculates the optimum policy without utilising or predicting the environment's dynamics (transition and reward functions) is known as a *Model-Free* algorithm. In fact, a model-free algorithm alternatively calculates a "value function" or a "policy" straight from experience (the agent's interaction with the environment) without utilising the transition or reward functions.

A *Model-Based* algorithm is one that estimates the best policy using the transition function (T) and the reward function (R). To construct a model of the particular environment, a model-based algorithm learns the transition and the reward functions.(Garg, 2018) Only an approximate of the transition and reward functions may be available to the agent, which can be learnt as the agent acts with the environment or provided to the agent.

On-policy and Off-policy : A policy-estimating reinforcement learning algorithm can either be off-policy or on-policy. As previously stated, the agent must examine the environment while being trained. For exploration and exploitation, the *On-policy* algorithm will take the same approach. An *Off-policy* algorithm, on the other hand, will pick actions from a distinct behavioral policy than the one being trained on. (Garg, 2018)

3.4 Q-Learning

Q-learning is off-policy reinforcement learning algorithm that tries to figure out which optimal form of action to choose in given current scenario. Here the "Q" represents Quality. (Violante, 2019)

Q-learning is a learning algorithm that is based on values. Value-based algorithms have used an equation to change a value function (like Bellman equation). (Shyalika, 2019)

Following are multiple form of Q-learning Definition.

Definition 3.4.1 *$Q^*(s, a)$ is the estimated value of taking action a into state s and then adopting that optimal policy.*

Definition 3.4.2 *Temporal Differences (TD) are used in Q-learning to calculate the value of $Q^*(s, a)$. In TD, agent acquiring knowledge from environment through episodes having no earlier understandings of that environment. (Shyalika, 2019)*

Definition 3.4.3 *The agent keeps a table called $Q[S, A]$, where S stands for states and A stands for actions.*

Definition 3.4.4 *The initial prediction of $Q^*(s, a)$ is $Q[s, a]$.*

Applying a Q-function instead of a state-value function has the benefit of providing an implicit representation of such a policy. Assume that π^* is the best policy and that Q^* is the best Q-function. An ideal deterministic strategy is one that chooses the action that maximises the Q-value at a particular state in time. (Peng, 2017)

$$\pi^*(s) = \arg\max_a Q^*(s, a)$$

where the best Q-function meets the requirement

$$\begin{aligned} Q_*(s, a) &= \mathbb{E}_{r,s'}[r + \gamma \max_{a'} Q^*(s', a')] \\ &= \mathbb{E}_{r,s'}[r + \gamma Q^*(s', \pi^*(s'))] \end{aligned}$$

The Bellman solution is being used by the Q-function, which takes two parameters: state s and action a. (Shyalika, 2019)

$$Q^\pi(s_t, a_t) = E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | s_t, a_t]$$

The Q-learning algorithm are among the most basic reinforcement learning algorithms,(Shantia, 2011; Watkins, 1989; Watkins and Dayan, 1992) which is given below : (refrance taken from (Castro, 2020)

Algorithm 2 Q-Learning Algorithm

Result: Off-policy control for estimating $\pi \simeq \pi_*$
Parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $Q(s, a) \forall s \in S^+, a \in \mathcal{A}(s)$ arbitrarily:
Set Q (terminal,$\cdot$) $= 0$
for each episode **do**
    Initialize S;
    for step = 0 **to** T **do**
        Choose A from S using policy derived from Q (e.g., ϵ-greedy);
        Take action A, observe R, S';
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$;
        $S \leftarrow S'$;
    end for
end for

3.4.1 Bellman Equation

The Bellman equation provides the foundation for addressing reinforcement learning problems. It facilitates in the selection of the most appropriate policy and value functions. (Tanwar, 2019)

The optimal value function $V_*(s)$ which contains maximum value, and it is represented as below :

$$V_*(s) = \max_{\pi} V_\pi(s), \forall \ \ s \in S$$

Likewise, $Q_*(s, a)$ could be described as the best action value function.

$$Q_*(s, a) = \max_{\pi} Q_\pi(s, a), \forall \ \ s \in S, a \in A$$

In addition, the below formula may be constructed for optimal policy:

$$V_*(s) = \max_{a \in A(s)} Q_{\pi^*}(s, a)$$

Combining both above equation,

$$\begin{aligned} V_*(s) &= \max_a \mathbb{E}_{\pi^*}(G_t|s_t = s, a_t = a) \\ &= \max_a \mathbb{E}_{\pi^*}\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k}|s_t = s, a_t = a\right) \\ &= \max_a \sum_{s'} p(s'|s,a)[R(s,a,s') + \gamma V_*(s')] \end{aligned}$$

A Bellman optimality formula of $V_*(s)$ is defined by above derived equation. For Q_*, a Bellman optimality formula is,

$$\begin{aligned} Q_*(s,a) &= \mathbb{E}(r_t + \gamma \max_{a'} Q_*(s_{t+1}, a')|s_t = s, a_t = a) \\ &= \sum_{s'} p(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q_*(s', a')] \end{aligned}$$

If such transition probabilities as well as reward functions are available, then Bellman optimally formulas could be addressed successively and t his technique can be considered as dynamic programming. (Rastogi, 2017)

3.5 Summary

This chapter explained the fundamentals of Reinforcement Learning by explaining a number of key topics. A number of popular techniques inside classic Reinforcement Learning were also covered in this section. (Rastogi, 2017) In any case, this knowledge was either directly used to build the algorithm for this study or was necessary to comprehend the approach presented in this thesis.(Bayiz, 2014) I explained difference about Model-free and Model-based method in formal concepts of Reinforcement Learning. In thesis I use Model-free techniques which are recommended for simulated systems since they do not require any environment knowledge.

CHAPTER 4

Deep Reinforcement Learning

4.1 Introduction

Deep Reinforcement Learning has sparked a huge amount of attention in the Artificial Intelligence field in recent years. (Rastogi, 2017) It is a way to approximate value functions or policy in a Reinforcement Learning framework by using Deep Neural Networks. Deep Reinforcement Learning has exceeded human performance in Atari games (Mnih, Kavukcuoglu, Silver, Rusu, Veness, Bellemare, Graves, Riedmiller, Fidjeland, Ostrovski, Petersen, Beattie, Sadik, Antonoglou, King, Kumaran, Wierstra, Legg, and Hassabis, 2015) and 3D locomotion tasks with a large state space. (Lillicrap et al., 2019; Rastogi, 2017; Schulman et al., 2018) When it comes to managing a biped robot's walking motion, DRL's high-dimensional state-action space makes it ideal. (Rastogi, 2017)

Deep Reinforcement Learning has been effectively used to Q-learning (Gu et al., 2016; Mnih et al., 2015; Rastogi, 2017) and actor critic techniques (Lillicrap et al., 2019; Rastogi, 2017; Wang et al., 2017) as well as policy gradient methods (Rastogi, 2017; Schulman et al., 2017, 2018). This chapter will provide a thorough description of Actor-Critic methods with its architecture, and explain deep actor critic algorithms like Deep Deterministic Policy Gradient algorithm, and Twin Delayed Deep Deterministic Policy Gradient.

4.2 Actor-Critic Learning

Actor-Critic is a Temporal Difference (TD) variant of policy gradient. It is divided into two networks which are Actor and Critic. The *actor* chose whatever action to do, and the *critics* told him how excellent it was and how it needed to be improved. (Karunakaran, 2020b)

4.2.1 Actor-Critic Methods

Actor-Critic (AC) techniques consist of a combination of value-based and policy-based approaches. (Fussell, 2018)

Compared to earlier TD approaches, actor critic methods have stronger convergence characteristics. It also facilitates action computation, which is especially useful for ongoing activities. Actor critic techniques may also be used to understand stochastic policies.(Rastogi, 2017)

As with actor-only techniques, actor-critic strategies are able to construct continuous actions, however the wide range in policy gradients with actor-only approaches is addressed by including a critic. (Grondman, 2015)

Actor-Critic method is divided into two groups based on how such functions being distributed in the techniques: (1) Actor-only and (2) Critic-only methods.

4.2.1.1 Actor-Only Methods

Actor-only techniques interact with such a set of policies that are parameterized. As a result of simulation, the actor parameters are changed in a manner that improves the performance of the actor. (Konda and Tsitsiklis, 2000)

Most of the policy is modelled using a function approximator, which has its own parameters. A policy gradient, is a popular method for updating the policy parameter.(Bayiz, 2014)

Though actor-only techniques perform well enough in continuous action spaces, the calculated policy gradient has a significant amount of variation. (Bayiz, 2014; Bhatnagar, Sutton, Ghavamzadeh, and Lee, 2009; Sutton and Barto, 2018)

A variance of gradient estimation is reduced by training a value function, which is necessary

enabling fast learning. (Bayiz, 2014)

4.2.1.2 Critic-Only Methods

Q-learning and *SARSA* are examples of critic-only techniques that apply a state-action value function rather than a policy function.(Grondman, 2015)

When action or state spaces both constant then function approximators are often used to estimate the value function. (Bayiz, 2014)

One of most interesting feature of critic-only techniques would be that they generate low-variance estimates of expected returns, which accelerates learning. (Bayiz, 2014; Hastie, Tibshirani, Friedman, and Franklin, 2004)

Critic only methods have not an advantage of continuous action space because it generates computational complexities , so that this method uses distributed action spaces. (Bayiz, 2014)

4.2.2 Actor-Critic Architecture

A basic Actor-Critic network design is shown in 4.1. In this architecture we have two models named Actor and Critic, for both we need two pair of weights (θ for Actor and w for Critic) as per following:

$$Actor : \pi(s, a, \theta)$$

$$Critic : \hat{Q}(s, a, w)$$

This pair of weights should be maintained and modified independently (Karagiannakos, 2018) as indicated in the following equations:

$$\mathcal{P} : \theta = \alpha \nabla_\theta \Big(\log \pi_\theta(s, a) \Big) \hat{Q}_w(s, a)$$

$$\mathcal{V} : w = \beta \Big(R + \gamma \hat{Q}_w(s_{t+1}, a_{t+1} - \hat{Q}_w(s_t, a_t) \Big) \nabla_w \hat{Q}_w(s_t, a_t)$$

As time goes on, the actor improves his ability to create progressively better action, and the critic improves his ability to assess those actions. (Karagiannakos, 2018)

There are two learning rates represented by α, β, the current policy by π_θ, and an approximation to the value function for the action $\hat{a}$ and state s by $\hat{Q}_w(s, a)$. The Temporal Difference

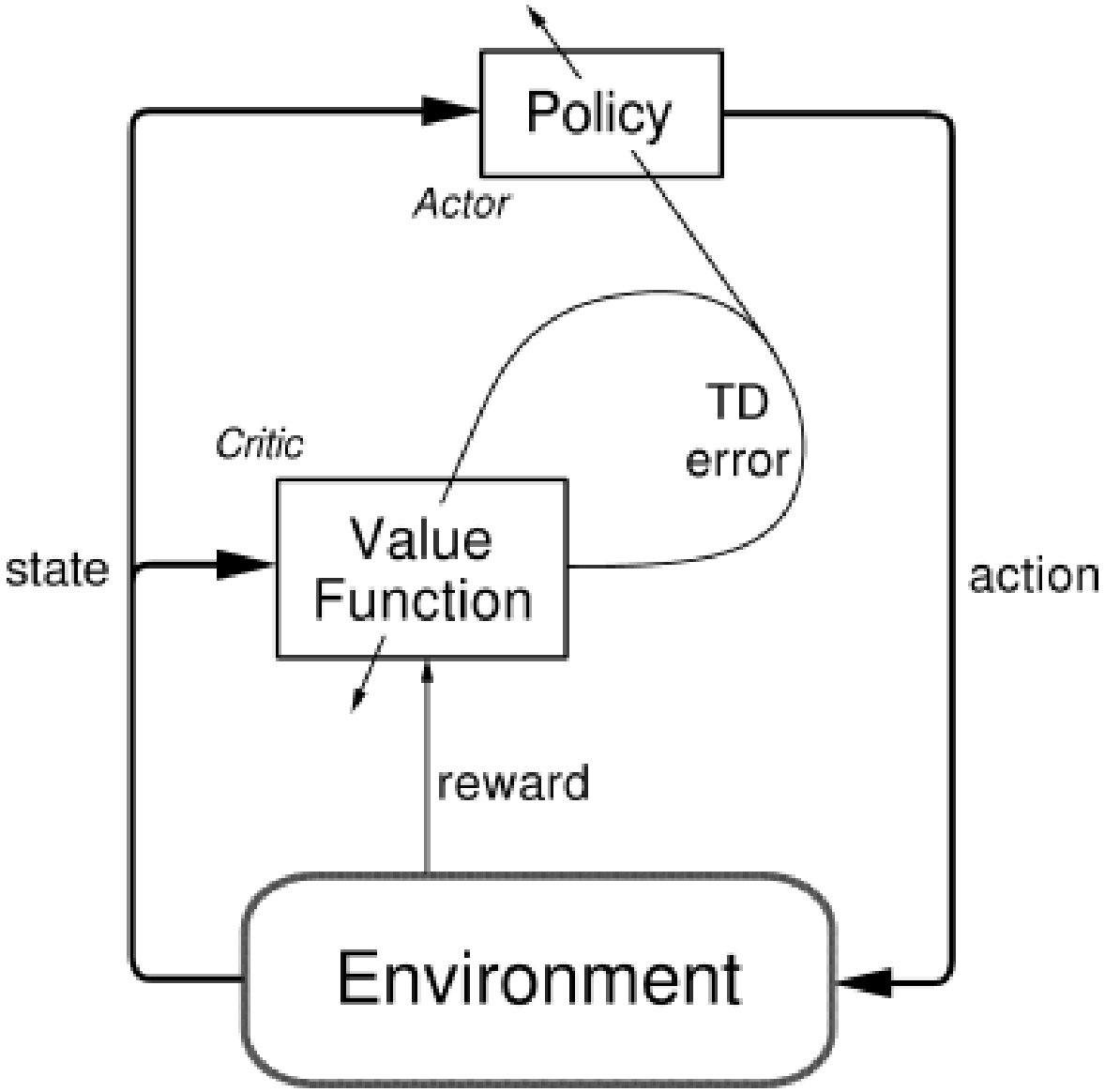

Figure 4.1: The actor-critic architecture

(TD) is the difference in $\hat{Q}$-value estimate between states s_t and s_{t+1}. (Kyziridis, 2017) The particular TD formula $(R + \gamma \hat{Q}_w(s_{t+1}, a_{t+1} - \hat{Q}_w(s_t, a_t))$ given in the above equation is known as $TD(0)$, and it is a version of the generic procedure known as $TD(\lambda)$. (Kyziridis, 2017; Sutton and Barto, 2018)

The pseudo code for Actor- Critic Algorithm as below : (reference from (Fussell, 2018))

Algorithm 3 General Actor Critic Algorithm

Initialise: actor π, critic Q
while not converged **do**
  for e in epochs **do**
    start new environment E_e
    sample $\{\mathbf{s}_0, \mathbf{a}_0, r_0, \mathbf{s}_1, \mathbf{a}_1, r_1, \ldots, \mathbf{s}_{T-1}, \mathbf{a}_{T-1}, r_{T-1}, \mathbf{s}_T\} \sim \pi$
    Run critic: compare critic predictions of the returns $Q(\mathbf{s}_0, \mathbf{a}_0), Q(\mathbf{s}_1, \mathbf{a}_1), \ldots, Q(\mathbf{s}_T, \mathbf{a}_T)$ to the actual returns $G_0, G_1, \ldots, G_T$ according to some error metric J
    Update critic: update parameters of Q using the error metric J
    Update actor: update the parameters of π using the new critic Q
  end for
end while

4.3 Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) is a model-free, off-policy and actor critic algorithm which integrate concepts from DPG **(Deterministic Policy Gradient)** and DQN **(Deep Q-Network)**. (Garg, 2018; Silver, Lever, Heess, Degris, Wierstra, and Riedmiller, 2014)

It trains the Q-function using off-policy data as well as the Bellman equation, then applies the Q-function to determine the policy. (Tian, 2019) It is a policy gradient approach that enables deterministic continuous-action control as a particular case. (Benbrahim, 1996)

It works in the same way as DQN (Deep Q-Network), in that it employs a target network to make the learning quite reliable. (Tian, 2019)

Furthermore, memory is used to verify that the training data is distributed independently and similarly. (Tian, 2019) It employs two different policies during exploration and upgrades because it is an off-policy algorithm. (Karunakaran, 2020a) Again for exploration, it employs a *Stochastic Behaviour Strategy*, while for the target upgrade, it employs a *Deterministic Policy*. (Karunakaran, 2020a)

As it follows actor-critic, the actor defines the policy function $\pi(s) = a$ and provides the action to be taken based on the state of a environment while the critic is the Q value function that describes the outcome reward and generates a signal error that evaluate the actor's activities. (Garg, 2018) The critic is the responsible for finding the maximum target Q value with given pair of state and action. (Bach, Melnik, Schilling, Korthals, and Ritter, 2020)

This concept is normally related to Q-learning and for optimum action-value function $Q^*(s, a)$, you can find the best action $a^*(s)$ in every specific state by analyzing it. Consider the given equation for it (OpenAI, 2018a) :

$$Q^*(s_t, a_t) = \mathbb{E}|r(s_t, a_t) + \gamma \max_{a_{t+1}} Q^{'*}(s_{t+1}, \pi^{'}(s_t))]$$

This equation, however, is not applicable to continuous action spaces. We may apply the equation with a deterministic policy (Garg, 2018) :

$$\max_{\theta} \mathop{\mathrm{E}}_{s\sim\mathcal{D}} [Q_\phi(s, \mu_\theta(s))]$$

The loss function L provided by actor and critic networks is as below : (Bach, Melnik, Schilling, Korthals, and Ritter, 2020)

$$L(\theta^Q) = \mathbb{E}[(Q(s_t, a_t|\theta^Q) - Y_t)^2]$$

where Y_t seems to be the supervised learning goal and therefore is calculated using the Bellman-equation. (Bach, Melnik, Schilling, Korthals, and Ritter, 2020)

$$Y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1}|\theta^Q)$$

where θ^Q are function parameters for the Q value function and policy μ.

Exploration Noise : A random noise technique $\mathcal{N}$ is also included of policy network μ to validate enough exploration in continuous space (Bach, Melnik, Schilling, Korthals, and Ritter, 2020) . The **Ornstein-Uhlenbeck** noise mechanism was used for our investigations. The technique upgrades its networks at the completion from every time interval to use a chosen at random compact out from replay buffer. (Garg, 2018)

$$\mu^{'}(s_t) = \mu(s_t) + \mathcal{N}$$

Target value and policy networks : To compute the goal Y_t , DDPG utilizes restored clones of the value as well as policy functions. These target networks get changed using the equations below once the value plus policy networks have been modified. (Bach, Melnik, Schilling,

Korthals, and Ritter, 2020)

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

The value network and policy network factors are θ^{Q} and θ^{μ} respectively, whereas the target value and policy network factors are $\theta^{Q'}$ and $\theta^{\mu'}$.

Experience replay buffer R would be used to take episodes from related to existing paths. This information is then utilised for training purposes. (Garg, 2018) A typical experience sample comprised of the tuple $s = s_n, a, r, d, s_{n+1}$, where s_n is the present state and s_{n+1} would be the next state, a is indeed the action and r seems to be the reward, and d would be a boolean representing either or not an episode is complete. (Bach, Melnik, Schilling, Korthals, and Ritter, 2020)

The pseudo code for **Deep Deterministic Policy Gradient (DDPG)** is given below (the reference taken from (OpenAI, 2018a)) The method is executed for a number of $M \times T$ time steps in this case. The algorithm creates paths or transitioning depending on the current policy for every time interval. (Garg, 2018) For this the agent takes action depending on the specific actor policy μ plus exploration noise $\mathcal{N}$, then saves the subsequent changes in a replay buffer R. (Garg, 2018)

DDPG employs four different neural networks: (1) **Q Network,** (2) **Deterministic Policy Network,** (3) **Target Q Network and** (4) **Target Policy Network**. (Tian, 2019)

Algorithm 4 Deep Deterministic Policy Gradient algorithm

Input: initial policy parameters θ, Q-function parameters ϕ , empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{targ} \leftarrow \theta$, $\phi_{targ} \leftarrow \phi$
repeat
Observe state s and select action $a = clip(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
Execute a in the environment
Observe next state s', reward r, and done signal d to indicate whether s' is terminal
Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
If s' is terminal, reset environment state.
if it's time to update **then**
for however many updates **do**
Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
Compute targets

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{targ}}(s', \mu_{\phi_{targ}}(s'))$$

Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_\phi(s, a) - y(r, s', d))^2$$

Update policy by one step of gradient ascent using

$$\nabla_\phi \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

Update target networks with

$$\phi_{targ} \leftarrow \rho\phi_{targ} + (1 - \rho)\phi$$

$$\theta_{targ} \leftarrow \rho\theta_{targ} + (1 - \rho)\theta$$

end for
end if
until convergence

4.4 Twin Delayed Deep Deterministic Policy Gradient

The **twin-delayed deep deterministic policy gradient (TD3)** algorithm is an *off-policy* reinforcement learning technique that is *model-free* and *online*. (The MathWorks, 2017) A TD3 agent is a reinforcement learning agent with actor-critic which finds out the best policy for increasing the predicted long-term reward. (The MathWorks, 2017)

The Twin-Delayed Deep Deterministic Policy Gradient algorithm (TD3) was developed to enhance the accuracy and efficiency by taking into account function approximation error. (Dankwa and Zheng, 2019; Fujimoto, Hoof, and Meger, 2018)

The TD3 method is special in that it combines three main Deep Reinforcement Learning approaches, including continous Double Deep Q-Learning. Policy Gradient as well as Actor-Critic. (Dankwa and Zheng, 2019; Silver, Lever, Heess, Degris, Wierstra, and Riedmiller, 2014) DDPG has certain disadvantages, despite its ability to provide good outcomes. Train DDPG, like several other reinforcement learning algorithms, may be unreliable and largely dependant on determining the proper hyper parameters for the present task. This is due to the algorithm's constant overestimation of the critic (value) network's Q values. (Science, 2019) This problem is addressed by TD3, which concentrates on decreasing the overestimation bias observed in prior algorithms.(Science, 2019)

The Twin Delayed DDPG (TD3) technique solves this problem by using *three* key strategies: (Science, 2019)

- **Clipped Double-Q Learning :** Instead of understanding one Q-function, it studies two (thus the name "twin"), and the lower including its two different Q-values is used to construct the objectives. (OpenAI, 2018b)
- **Delayed Policy Updates :** The strategy and objectives of a TD3 agent are modified less frequently than the Q functions. (OpenAI, 2018b)
- **Target Policy Smoothing :** A TD3 agent introduces sound towards the target action while upgrading the policy, making it less probable for the policy to utilize actions having large Q-value estimations. (OpenAI, 2018b)

TD3 simultaneously trains two different Q-functions, Q_{ϕ_1} and Q_{ϕ_2}, using mean square Bellman error reduction just like Deep Deterministic Policy Gradient algorithm trains one Q-function. (OpenAI, 2018b)

Smoothing actions relevant to the target policy $\mu_{\theta_{\text{targ}}}$ are utilised to construct the Q-learning target, but with some noise introduced across each level of the action. (OpenAI, 2018b) The target action is edited to fit inside the acceptable action limit (all acceptable actions range between $a_{Low} \leq a \leq a_{High}$) just after noise has been applied. (OpenAI, 2018b) The following are indeed the target actions:

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \qquad \epsilon \sim \mathcal{N}(0, \sigma)$$

Each Q-functions utilise only one target with clipped double-Q learning, to check which of the two different Q-functions provides a lower target value: (OpenAI, 2018b)

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{i,\text{targ}}}(s', a'(s'))$$

Then both of these are trained for the target:

$$L(\phi_1, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{E}\left[\left(Q_{\phi_2}(s, a) - y(r, s', d)\right)^2\right]$$

$$L(\phi_2, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{E}\left[\left(Q_{\phi_2}(s, a) - y(r, s', d)\right)^2\right]$$

Finally, appropriate policy is discovered simply via increasing Q_{ϕ_1}:

$$\max_{\theta} \underset{s\sim\mathcal{D}}{\text{E}}\left[Q_{\phi_1}(s, \mu_\theta(s))\right]$$

It is essentially the same as Deep Deterministic Policy Gradient algorithm. In TD3, though, this policy is modified less frequently than with the Q-functions. (OpenAI, 2018b)

Exploration Noise : Adding **Gaussian** noise to the actions of TD3 policies during training period will allow them to explore more effectively. Furthermore, you can lower the size of the noise during the training for acquiring relatively high training data. (OpenAI, 2018b)

The loss factor is proposed as for critic networks as continues to follow : (Zhang, Li, An, Man, and Zhang, 2020)

$$J(\theta_i) = \frac{1}{N}\sum_{j=1}^{N}(y - Q_{\theta_i}(s,a))^2$$

The deterministic approach is being used to solve the optimization problem for the actor network, as well as the its loss function is described as : (Zhang, Li, An, Man, and Zhang, 2020)

$$\nabla_\phi J(\phi) = \frac{1}{N}\sum_{j=1}^{N}\nabla_a Q_{\theta 1}(s,a)|_{a=\pi_\phi(s)}\nabla_\phi \pi_\phi(s)$$

TD3 offers a total of six neural networks, including ***(1)*** *Actor Network* (π_ϕ), ***(2)*** *Actor Target Network* ($\pi_{\phi'}$), ***(3)*** *Critic Network* (Q_{θ_1}), ***(4)*** *Critic Network* (Q_{θ_2}), ***(5)*** *Critic Target Network* ($Q_{\theta_1'}$) *and* ***(6)*** *Critic Target Network* ($Q_{\theta_2'}$). (Zhang, Li, An, Man, and Zhang, 2020) The roles and responsibility of every network would be as below:

1. **Actor Network** (π_ϕ) : Accountable for upgrading actor network parameters ϕ and identify the current action a based on the current state s, that can be used to explore the environment to produce the immediate state $s^{'}$ and reward R. (Zhang, Li, An, Man, and Zhang, 2020)

2. **Actor Target Network** ($\pi_{\phi'}$) : Decide best next action $a^{'}$ to take based on the next state $s^{'}$ observed with in experience replay buffer. (Zhang, Li, An, Man, and Zhang, 2020)

3. **Critic Networks** (Q_{θ_i}) : It estimates the present value Q by considering minimum Q value of both critic networks ($Q_{\theta_1}, Q_{\theta_2}$) and on basis of that upgrades the Q network parameters. (Zhang, Li, An, Man, and Zhang, 2020) To calculate the minimum value of Q network it follows below equation :

$$y = r + \gamma \min_{i=1,2} Q_{\theta_i'}(s', a')$$

4. **Critic Target Networks** ($Q_{\theta_i'}$) : It is responsible to estimate the target Q value of $Q_{\theta_i'}(s', a')$. The parameters of the network θ_i are updated through θ_i' on a regular basis. (Zhang, Li, An, Man, and Zhang, 2020)

The pseudo code for Twin Delayed DDPG algorithm Algorithm as below : (reference from (OpenAI, 2018b))

Algorithm 5 Twin Delayed DDPG algorithm

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2 , empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{targ} \leftarrow \theta$, $\phi_{targ,1} \leftarrow \phi$, $\phi_{targ,2} \leftarrow \phi$
repeat
Observe state s and select action $a = clip(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
Execute a in the environment
Observe next state s', reward r, and done signal d to indicate whether s' is terminal
Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
If s' is terminal, reset environment state.
if it's time to update **then**
for j in range (however many updates) **do**
Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
Compute target actions

$$a'(s') = clip(\mu_{\phi_{targ}}(s') + clip(\epsilon, -c, c), a_{Low}, a_{High}), \qquad \epsilon \sim \mathcal{N}(0, \sigma)$$

Compute targets

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{targ}}(s', a'(s'))$$

Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_\phi(s, a) - y(r, s', d))^2 \qquad for\ i = 1, 2$$

if j mod policy delay = 0 **then**
Update policy by one step of gradient ascent using

$$\nabla_\phi \frac{1}{|B|} \sum_{s \in B} Q_{\phi_1}(s, \mu_\theta(s))$$

Update target networks with

$$\phi_{targ,i} \leftarrow \rho\phi_{targ,i} + (1 - \rho)\phi_i \qquad for\ i = 1, 2$$

$$\theta_{targ} \leftarrow \rho\theta_{targ} + (1 - \rho)\theta$$

end if
end for
end if
until convergence

4.5 Summary

The most important takeaway from this chapter is actor-critic techniques, the actor and critic are discussed individually. I have proposed an actor-critic architecture that aims to combine the benefits of actor-only and critic-only techniques. In addition, I have introduced two essential actor-critic algorithms for continuous space that will be used to train biped robo's to walk in my research. The implementation of both algorithms will be described in the next chapter.

CHAPTER 5

Experimental Design and Implementation

5.1 Introduction

This chapter contains information about design and implementation of current study. This chapter is divided in two parts: Section - 5.2 explains the OpenAI Gym library, which is an open source toolkit that provides ready-made infrastructures for training agents. Moreover, this section provides further information on how to install the toolkit and how to configure it in the system. At the end of this section, I describe the brief detail about an environment that I utilise in this study.

In this Section - 5.3, I go through the suggested techniques in depth, including the parameters used to train the model and the procedures required to execute it.

5.2 OpenAI Gym

Elon Musk founded OpenAI, a non-profit company dedicated to putting the OpenAI open source framework into practise. This framework aims to make AI algorithms more understandable by offering a user-friendly environment for developers to build customized models. (del Corral Tercero and Macias, 2018)

Gym is a toolbox developed by OpenAI, for building and testing reinforcement learning algorithms. The goal of OpenAI is to find and create a route to artificial general intelligence that is both safe and effective. (Laivamaa, 2019)

Gym eliminates the need for the user to construct complex settings and adds to standardisation, allowing researchers to work with a similar assumptions. Python programming language is used to create Gym. Many other environments can be found in this toolkit, including robot simulations and Atari. (Chen, 2018)

Gym is a library that includes test scenarios, sometimes known as environments, that may be used to evaluate reinforcement learning systems. The environments have a similar interface, allowing the user to develop generic algorithms. (Laivamaa, 2019)

Many environments are not bundled in Gym, such as 3 dimensional environments, robotics with ROS and Gazebo, and so on. As a result, various gym integration tools and packages, such as Parallel Game Engine, Gym-Gazebo, and Gym-Maze, are created in order to integrate Gym with those other simulator and visualization tools. (Chen, 2018)

The essential methods exist within every environment class : (1) step, (2) (3) reset, (4) render, (5) close, and (6) seed. The reset function returns the environment to its starting state after the end of every episode. (Hajash, 2018)

5.2.1 Library Setup

To get started with the OpenAI Gym, you will require Python 3.5 or higher version on your machine.(Elon Musk and Schulman, 2018) The Python 3.5+ Tutorials contains a beginners instruction article that explains how to install and use Python. Installing Gym is as simple as using Python's package manager *pip* command. (Laivamaa, 2019)

```
pip install gym
```

I have started by creating the very simplest code to get everything functioning. This will help to better understand OpenAI Gym's fundamental structure. *CartPole-v0-problem* environment is executed for 1,000 time-steps in OpenAI Gym manual, and the results are presented at each step. (Elon Musk and Schulman, 2018; Laivamaa, 2019)

```
import gym
env = gym.make('CartPole-v0')
```

```
env.reset()
for _ in range(1000):
    env.render()
    env.step(env.action_space.sample()) # take a random action
env.close()
```

As shown in the above code, after importing the OpenAI Gym library, which has all the environments and dependencies that I required, I can start work with the toolkit. My next step is to create the Environment Variable (ENV) to the environment I am using, and then reset it back to its original state. I construct a for-loop for 1000 timesteps after the specified environment is established. During each loop, I display the existing state of the environment, and then perform a random action in it. (Laivamaa, 2019)

When I execute the code, a popup window with a visual representation of the process is appeared like below:

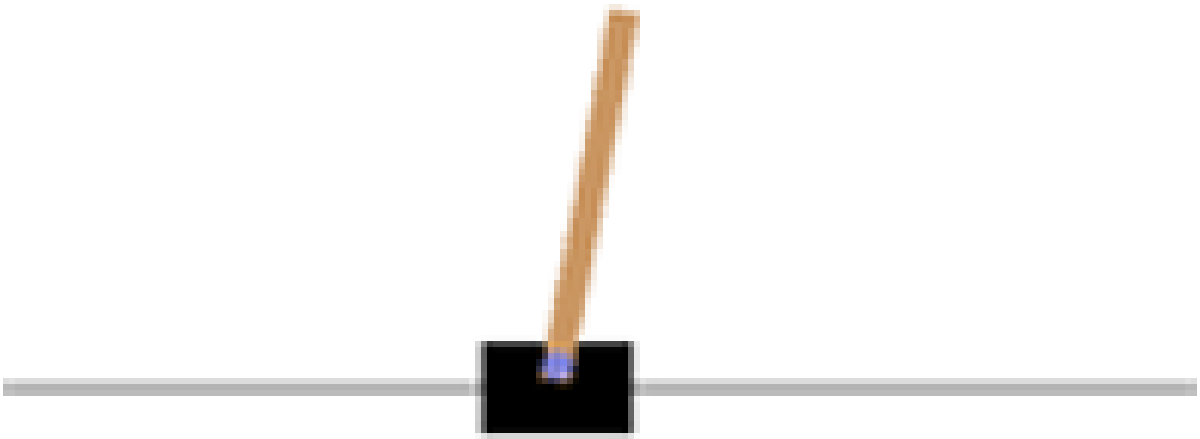

Figure 5.1: CartPole-v0 (OpenAI Gym Documentation, n.d.)

Note that, this is just basic example to check whether all the functionality is working properly. I will use **"BipedalWalker-v2"** for my further research.

5.2.2 Main elements

OpenAI Gym Documentation (n.d.) introduces environment's step -function, which provides following four values: (Elon Musk and Schulman, 2018)

- **Observation (object)** is an environment-specific entity which describes the agent's information of such environment. (Elon Musk and Schulman, 2018; Laivamaa, 2019)
- **Reward (float)** term is refereed as the degree of reward given by the past action (Elon Musk and Schulman, 2018; Laivamaa, 2019)
- **done (boolean)** which indicates the final state of an environment and notifies an algorithm if it is time to update it. Most Gym objectives are divided into episodes, and when **done = True**, it signifies that now the episode has reached its end. (Elon Musk and Schulman, 2018; Laivamaa, 2019)
- **Info (dict)** is a debugging tool. It could be useful for learners in some circumstances. However, you cannot utilise for learning purposes in agent's assessment techniques. (Elon Musk and Schulman, 2018)

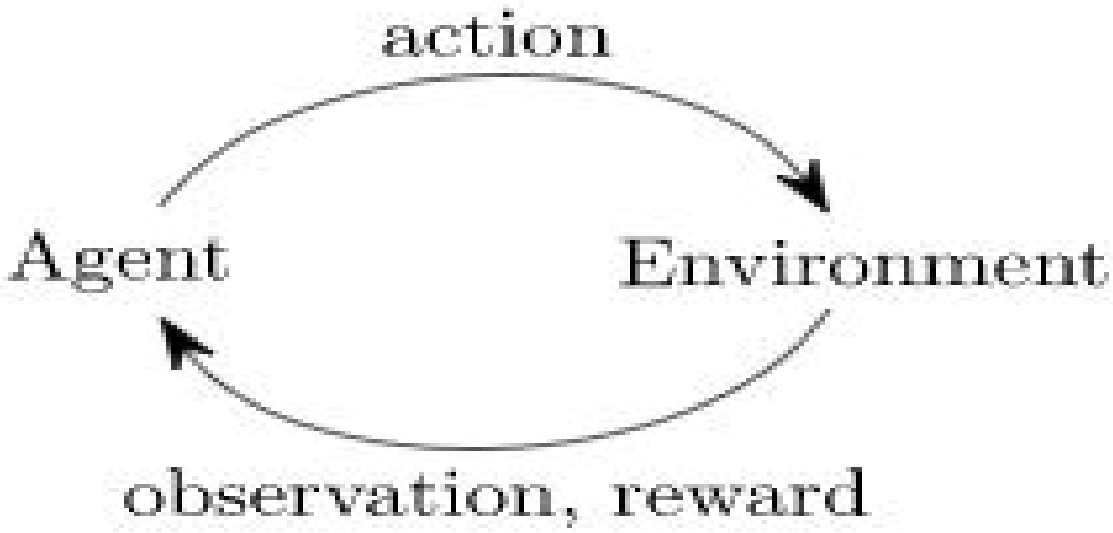

Figure 5.2: Observation step function (OpenAI Gym Documentation, n.d.)

According to OpenAI Gym, the step-function is simply an application of the standard Reinforcement Learning "agent-environment" loop. As soon as the agent performs an action, the environment responds including an observation as well as a numerical reward for that action. (Elon Musk and Schulman, 2018; Laivamaa, 2019) The procedure begins with a call to reset(), which receives the very first observation. (Elon Musk and Schulman, 2018)

5.2.3 Environment

The **Env** class is the most basic component of OpenAI Gym. It is a Python class that acts as a simulator, running the environment in which you wish to train your agent. Open AI Gym has a variety of environments, including one in which you may drive a car up a hill, balance a hanging pendulum, and do well on Atari games, and others. Gym also gives you the option of creating your own unique environments. (Kathuria, 2020)

In my research, I have used the **"BipedalWalker-v2"** environment, which comprises of a two-dimensional robot in a horizontal terrain. The agent is subjected to circumstances comparable to those found on Earth, such as gravity and friction. (Fussell, 2018) The aim is to go ahead; If the robot moves forward, it receives 300+ points; if the robot collapses, it receives -100 points.

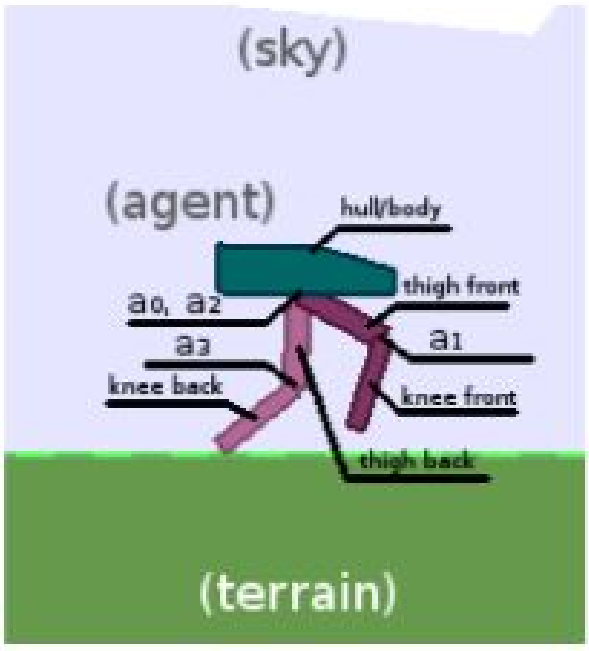

Figure 5.3: Labeled anatomy of the parts of the agent

The robot's anatomy is made up of a 5-sided polygon termed the *hull*, which is linked to two similar legs. The *back* leg has lighter colour than *front* leg. A single leg is made up of two identical rectangles: a thigh rectangle is attached to the hull at a connection called the *waist*, and a shin rectangle is joined to the *distal* position of the thigh at a joint called the *knee*. (Fussell, 2018) This is seen in Figure

5.2.3 (Fussell, 2018)

To experiment **"BipedalWalker-v2"** environment, one need to install *Box2D* simulator which is designed by OpenAI Gym for continuous control tasks.

```
pip install Box2D
```

The environment creates observations based on the robot's activities. Most specifically, the robot performs a particular action $a_t \in A$, generated from the environment's action space, $A = [-1, 1]$, at every iteration $t \in T$ and in state $s_t \in S$. After that, the environment generates observations like reward $R(s_t) \in R$ plus new state $s_{t+1} \in S$. After every action, the environment generates different states and rewards. Rewards indicate how effective the performed action a_t was at time interval t, as measured from the present state towards the next state, $s_t \rightarrow s_{t+1}$. (Kyziridis, 2017)

A four-length vector representing the speed and direction of a four robot joints is used to indicate actions $a_t \in A$. The speed as well as degree of a joints are represented by each integer with in action vector a_t , which would be a continuous value inside the range $[-1, 1]$. (Kyziridis, 2017)

As previously mentioned, the environment accepts the action vector a_t as input for every state s_t and produces the existing reward $R(s_t)$, as well as the next state s_{t+1}. The robot is likewise moved from the previous state s_t to the new state s_{t+1} by the environment. Streaming of a game in real time is possible using OpenAI's environment. (Kyziridis, 2017) Figure-5.4 depicts an arbitrary robot position as seen via the compositing environment.

The environment interactions is illustrated in Figure 5.5 (reference from the (Kyziridis, 2017)) using Algorithm 6. In this case, we may offer an action is input and receive the new observation straight again from environment as a result of that action. A mathematical formulation of the vectors, action plus state as below :

$$Action = \langle H_1, K_1, H_2, K_2 \rangle$$

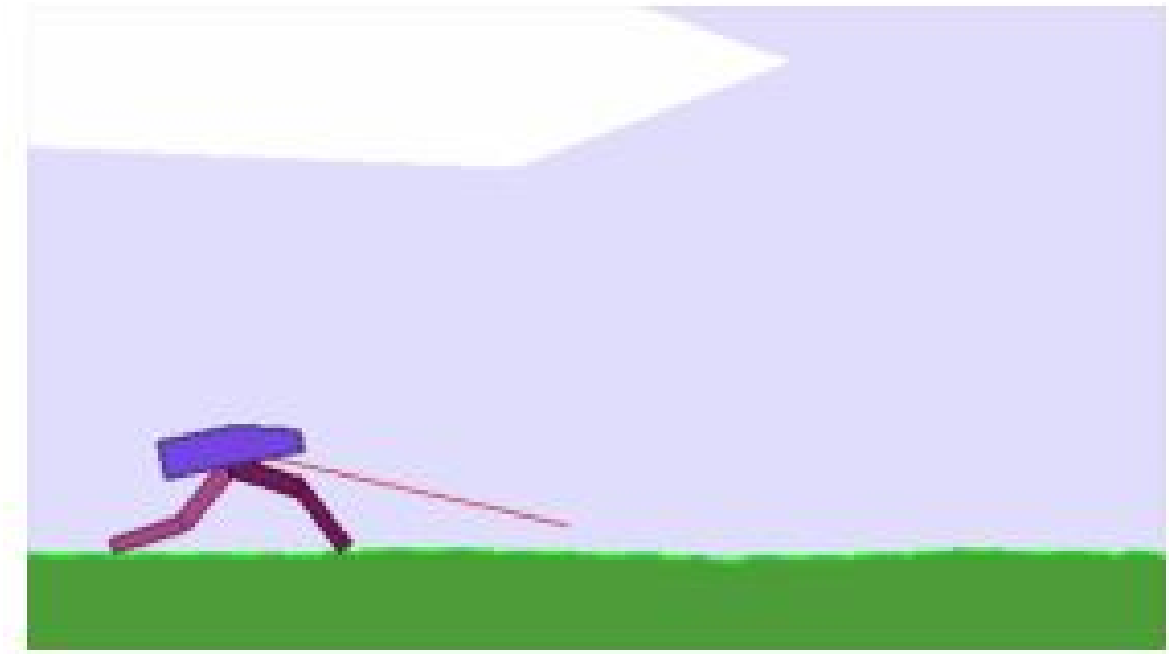

Figure 5.4: BipedalWalker screenshot

and

$$State = \langle h_0, \frac{\partial h_0}{\partial t}, \frac{\partial x}{\partial t}, \frac{\partial y}{\partial t}, h_1, \frac{\partial h_1}{\partial t}, k_1, \frac{\partial k_1}{\partial t}, h_2, \frac{\partial h_2}{\partial t}, k_2 \frac{\partial k_2}{\partial t}, \iota_l eg1, \iota_l eg2, \textbf{lidar} \rangle$$

A hull angle is represented by h_0, while the hips with knees angles are represented by h_1, h_2, k_1, k_2. Moreover, $lidar \in R$ is a vector that indicates the lidar measurements which the robot monitors at every time stamp t_i. As you can see in the *action* vector, H_1, H_2, K_1, K_2 show the hip and knee torques and speed, respectively. (Kyziridis, 2017)

The **"BipedalWalker-v2"** pseudo code is as described in the following (Kyziridis, 2017):

Algorithm 6 Environment Interaction Example

```
procedure ENVIRONMENT
    Load the environment
    env = OpenaiMake('BipedalWalker-v2')
    Reset get initial State S_t
    state = ResetEnvironment()
    Generate random Action A_t
    action = Random (x_min = −1, x_max = 1, size = 4)
    Interact: feed Action A_t, get new observation
    state_new, reward = envStep(action)
end procedure
```

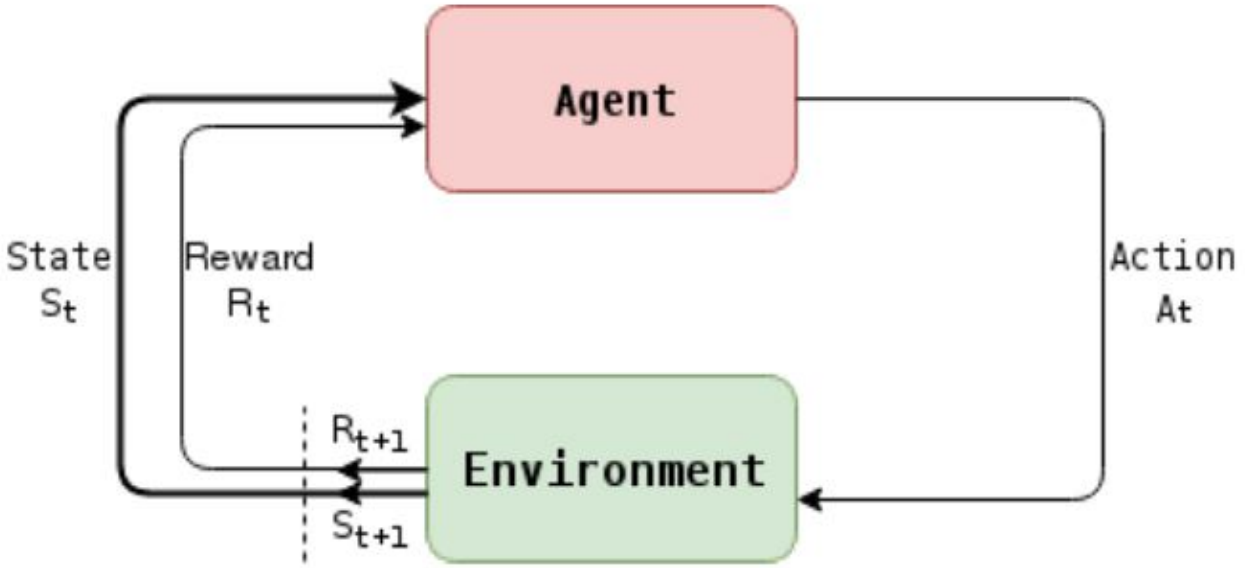

Figure 5.5: Agent-Environment feedback loop

5.3 Proposed Method

In order to overcome the continuous task challenges in my research, I propose two deep learning algorithms (1) Deep Deterministic Policy Gradient and (2) Twin Delayed Deep Deterministic Policy Gradient, which are model-free, off-policy and follows actor-critic approach. In Chapter-4 there is detailed information about both algorithms in sections 4.3 and 4.4 respectively.

As a programming language, I utilised Python version 3.5 in this research. I used the Jupyter notebook to write and run the code. And all of the experiments were carried out on central processing unit (CPU).

Other libraries are employed in this study, including **numpy** for working with arrays, **pytorch** for deep learning, and **matplotlib** for producing graphical representations. I used the Box2D environment of OpenAI Gym to connect the continuous control task of BipedalWalker-v2.

In this section, the experiments with different parameters and implementation of both algorithms is described briefly :

5.3.1 Experiment Details of DDPG

The Actor-network consist of two hidden layers that are activated by batch-norm and relu. The hidden layers are 600 and 300 in size, respectively. Because the action is restricted to the range $(-1, 1)$, tanh activation was applied on the last layer. (Verma, 2019)

The Critic-network consist of two hidden state layers and one hidden action layer. In this case, batch-norm as well as relu activation are used. With the use of convolution layers, hidden layers of both action and space is combined. (Verma, 2019)

The following parameters were described and used to build the model: (1) Environment name is **BipedalWalker-v2**, (2) Number of episodes **2000**, (3) Random seed number **0**, (4) Maximum timesteps **2000**, (5) Exploration noise **0.1**, (6) Batch size **100**, (7) Discount factor **0.99**, (8) Target network update rate **0.001**, (9) Weights of network **0.001**, (10) Buffer size **1000000**, (11) Save model is **true** (12) Actor learning rate **0.0001**, (13) Critic learning rate **0.001** and **Adam optimizer** was used to update network parameters.

Table 5.1 display the hyperparameters and its associated values which were used during training of DDPG algorithm. (Dankwa and Zheng, 2019)

5.3.1.1 Steps Used in Implementing the DDPG Algorithm

To implement DDPG algorithm in my current research, I follow below procedure :

Step 1: Create an Experience Replay Buffer, that will be updated with new transitions.

Step 2: Construct a neural net for both the Actor model and another for the Actor target.

Step 3: Create a neural net for both the Critic model and another for the Critic target.

Step 4: Take a set of transition (s, a, r, s') form memory, where s indicates state, a indicates action, r indicates reward and s' indicates next state.

Step 5: The Actor target performs a next action a when next state s' will come.

Step 6: The Critic target accept the pair of (s', a') as input and then generate Q-value.

$Q_t(s', a')$ is the output.

Table 5.1: The Parameters of the DDPG model

Parameter	**Value**	**Description**
Env_name	BipedalWalker-v2	The environment's name
seed	0	Random seed number
Max_timeste ps	2000	Number of iterations
Save_model	true	If you want to store the pre-trained model then this value is true
Batch_size	100	The batch's size
discount factor	0.99	Discount factor (γ)
Buffer_size	1000000	Replay buffer size
Weight_decay	0.001	A neural network's weights
Tau	0.001	For target update parameters
LR_Actor	0.0001	The learning rate of the actor
LR_Critic	0.001	The learning rate of the critic

Step 7: The final aim of critic model is $Q_t = r + \gamma^* Q_t$, where γ is the discount factor.

Step 8: Critic model accept (s, a) as input and produce one Q-value.

$Q_t(s, a)$ is the output.

Step 9: Calculate the Loss of critic model.

Critic Loss = MSE_Loss$(Q(s, a), Q_t)$

Step 10: To calculate critic loss stochastic Gradient Descent (SGD) optimizer is used.

Step 11: Upgrade an Actor model by conducting Gradient Ascent on the result of an initial Critic model:

$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|s_i$

where Q and μ are Actor and Critic weights, respectively.

Step 12: Updating the Actor target's weights using Polyak Averaging.

$\phi_{targ} \leftarrow \rho\phi_{targ} + (1 - \rho)\phi$

Step 13: Updating the Critic target's weights using Polyak Averaging.

$$\theta_{targ} \leftarrow \rho\theta_{targ} + (1 - \rho)\theta$$

5.3.2 Experiment Details of TD3

The TD3 model was developed using two-layer feedforward neural net with 256 and 256 input layers, accordingly. Rectified Linear Units (ReLU) were utilised as such an input signal among every layer both for Actor as well as Critics, with a tanh unit after the Actor's output. (Dankwa and Zheng, 2019)

The following parameters were specified and used in the model: (1) Environment name is **BipedalWalker-v2**, (2) Number of episodes **2000**, (3) Random seed number **10**, (4) Maximum timesteps **2000**, (5) Exploration noise **0.1**, (6) Batch size **100**, (7) Discount factor **0.99**, (8) Target network update rate **0.005**, (9) Policy noise **0.2**, (10) Clip noise **0.5**, (11) Policy frequency **2**, (12) Buffer size **50000**, (13) Save model is **true** (14) Actor learning rate **0.001**, (15) Critic learning rate **0.001** and **Adam optimizer** was used to update both network parameters. (Dankwa and Zheng, 2019; Kingma and Ba, 2014)

Table 5.2 summarises the components and associated values that were employed while TD3 algorithm. (Dankwa and Zheng, 2019)

Table 5.2: The Parameters of the TD3 model

Parameter	Value	Description
Env_name	BipedalWalker-v2	The environment's name
seed	10	Random seed number
Max_timeste ps	2000	Number of iterations
Save_model	true	If you want to store the pre-trained model then this value is true
Expl_noise	0.1	Exploration noise
Batch_size	100	The batch's size
discount factor	0.99	Discount factor (γ)
Noise_clip	0.5	The Gaussian noise's maximum value
Policy_freq	2	Total number of iterations must pass before network was created
LR_Actor	0.001	The learning rate of the actor
LR_Critic	0.001	The learning rate of the critic

5.3.2.1 Steps Used in Implementing the TD3 Algorithm :

To implement TD3 algorithm, I follow below procedure :

Step 1: Create an Experience Replay Buffer, that will be updated with new transitions as they are added.

Step 2: Construct a neural net for both the Actor model and another for the Actor target.

Step 3: Create neural nets for each of two critic models, as well as two neural nets for each of two critic target models.

Step 4: Take a set of transition (s, s', a, r) form memory, where s denotes state, s' denotes next state, a denotes action, and r denotes reward. Then, for every batch element, repeat the process. (Dankwa and Zheng, 2019)

Step 5: The Actor target performs a next action a when next state s' is reached.

Step 6: Insert Gaussian noise to such following action a' and assign this to a range of possible values that the environment supports.

Step 7: These two Critic target accept both of the (s', a') pairs as input and then generate two Q-values.

$Q_{t1}(s', a')$ and $Q_{t2}(s', a')$ are the outputs.

Step 8: Select minimum of two Q-Value outputs as $\min(Q_{t1}, Q_{t2})$ which indicates the estimated value for immediate state.

Step 9: $Q_t = r + \gamma^* \min(Q_{t1}, Q_{t2})$ is the end goal of both critic model, where γ is the discount factor.

Step 10: Both critic models accept (s, a) as input and output two Q-values.

$Q_{t1}(s, a)$ and $Q_{t2}(s, a)$ are the outputs.

Step 11: Calculate the Loss resulting from the both Critic models.

Critic Loss = MSE_Loss$(Q_1(s, a), Q_t)$ + MSE_Loss $(Q_2(s, a), Q_t)$

Step 12: Stochastic Gradient Descent (SGD) optimization is used to backprobagate Critic Loss.

Step 13: Modify an Actor model after two stages by conducting Gradient Ascent on the result of an initial Critic model:

$\nabla_\phi J(\phi) = \frac{1}{N} \sum_{j=1}^{N} \nabla_a Q_{\theta 1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$

where ϕ and θ_1 are Actor and Critic weights, respectively.

Step 14: Updating the Actor target's weights over two iterations using Polyak Averaging.

$\phi_{targ,i} \leftarrow \rho \phi_{targ,i} + (1 - \rho)\phi_i$

Step 15: Updating the Critic target's weights over two iterations using Polyak Averaging.

$\theta_{targ} \leftarrow \rho \theta_{targ} + (1 - \rho)\theta$

5.4 Summary

In this chapter, we learned about OpenAI gym, its installation, observations and the *BipedalWalker-v2* environment. Additionally, alternative hyper parameters and steps for implementing both the Deep Deterministic Policy Gradient and Twin-Delayed Deep Deterministic Policy Gradient algorithms were provided to address this scenario.

CHAPTER 6

Experiment Results

The purpose of this experiment was to familiarise myself with both the libraries and the training method. (Garg, 2018) In my present research study, I tested a bipedal robot to walk and run over a field while interacting with a constant control environment (Dankwa and Zheng, 2019) using a Deep Deterministic Policy Gradient and Twin-Delayed Deep Deterministic Policy Gradient algorithm. The major goals were to illustrate the development of a DDPG and TD3 algorithms, build a model based on the DDPG and TD3 algorithms, and assess their efficiency.

In order to do this, the DDPG algorithm was used for **500** episodes and TD3 algorithm was used to train for **400** episodes and the combined training time was just over 10 hours. (Garg, 2018) Both trials were carried out on an laptop with an Intel Core(TM) i5-6300HQ CPU @2.30GHz x 4, 8GB of DDR3 RAM, and Jupiter Notebook was used for coding and graphical depiction of the results.

First and foremost, I taught the agent how to walk within **200** episodes. After training 200 episodes of DDPG algorithm, I got a score of around **-100**, however in TD3, I got the same result after only **100** episodes as shown in figures 6.1 and 6.2 respectively.

---

https://github.com/Jayani/BipedalWalker-v2-using-OpenAI-Gym

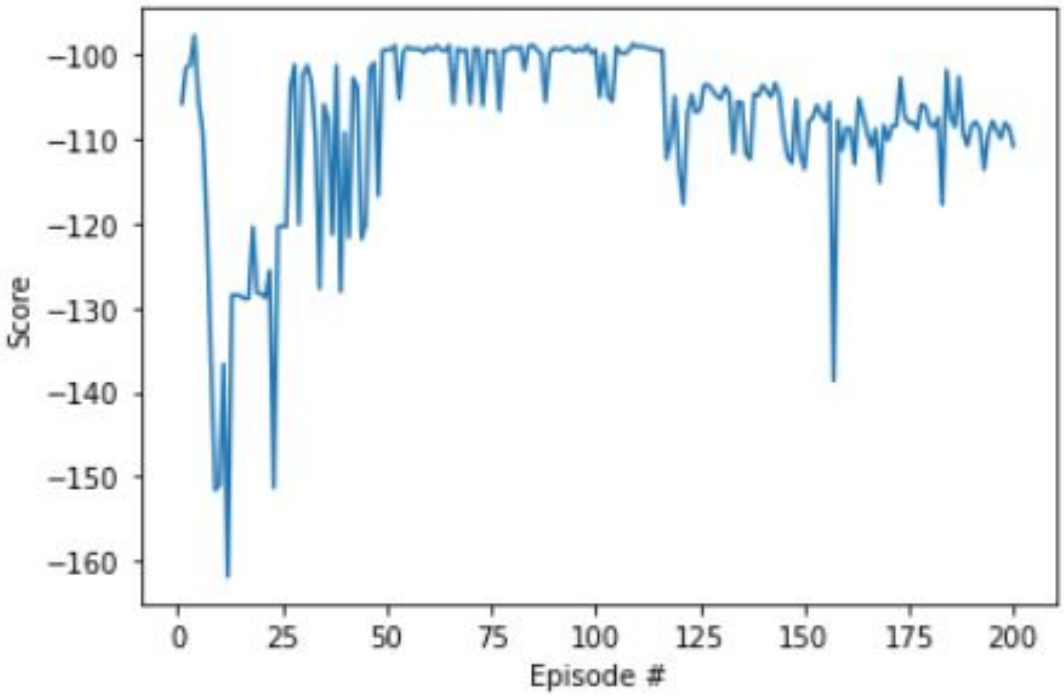

Figure 6.1: DDPG Algorithm score after 200 Episodes

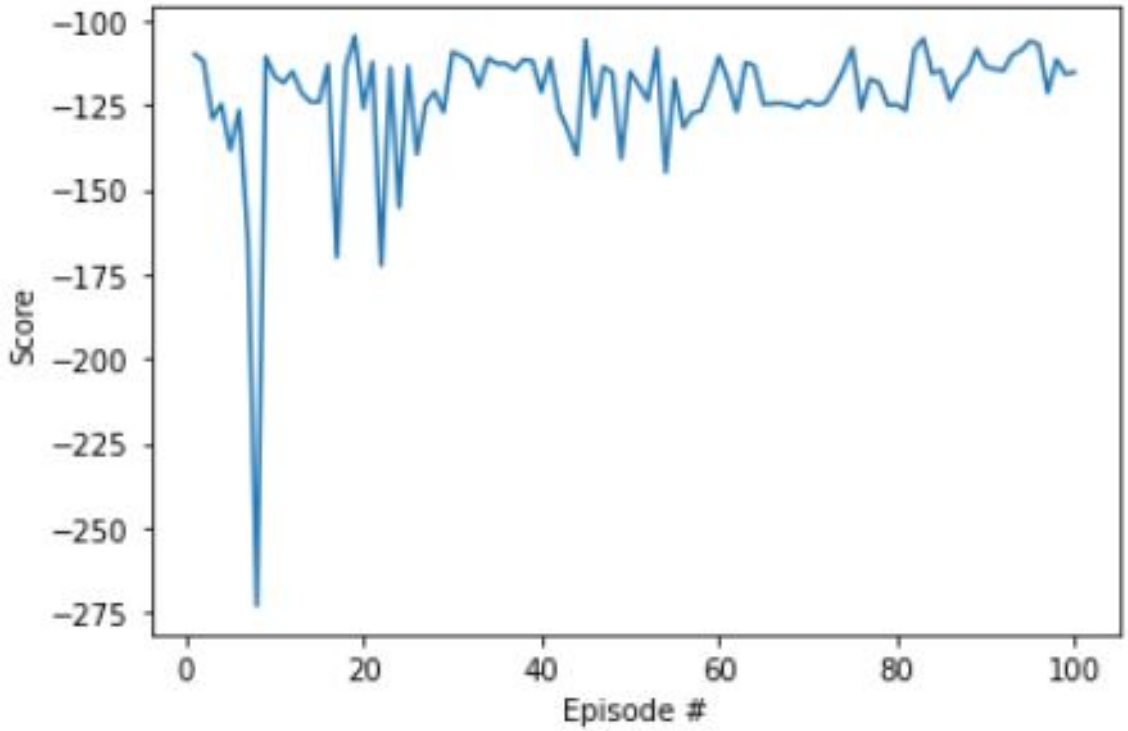

Figure 6.2: TD3 Algorithm score after 100 Episodes

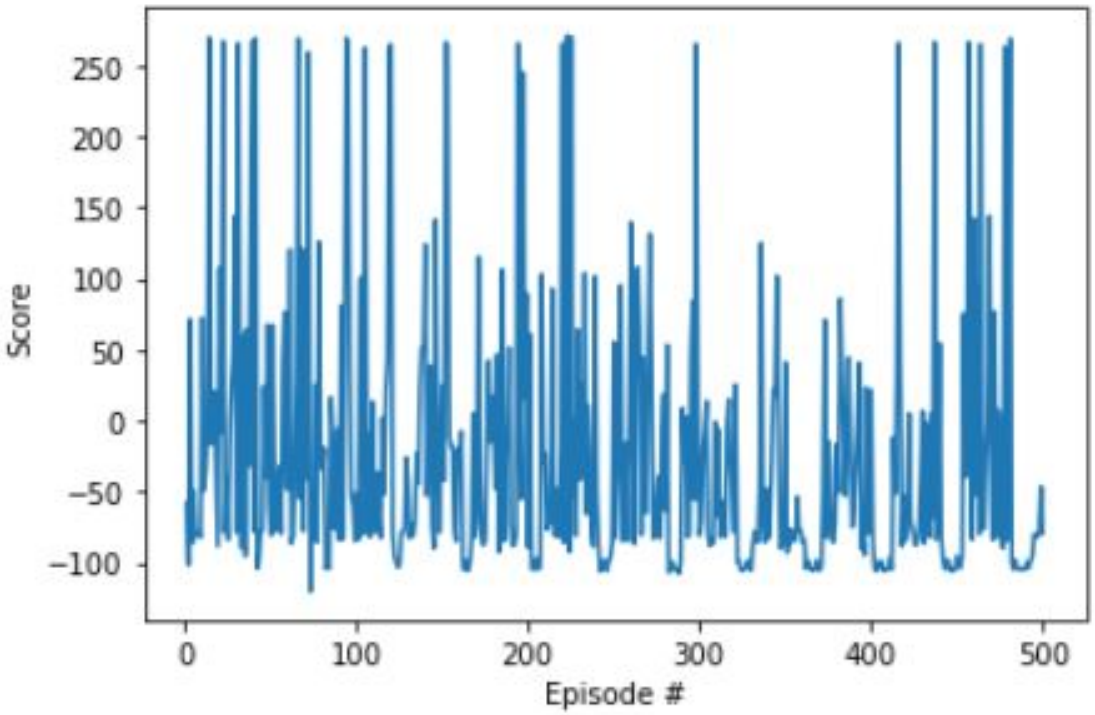

Figure 6.3: DDPG Algorithm score after 500 Episodes

The Agent then learnt from the model's previous pattern after being exposed to the training. After training of **1500** episodes, the agent increased its performance. (Zamora, Lopez, Vilches, and Cordero, 2016)

After saving the weights of previously trained **1500 episodes**, I again retrained my agent for **500 episodes** using previously trained model's weights, I stopped the program's execution since it had reached **300** rewards, and I observed the trained agent behaviours of both algorithms. I found that DDPG managed to achieve **250** score in **500 episodes** while TD3 achieve **300** in **400 episodes** in the given environment.

Figure shows a comparison of the average episodes vs. rewards graphs for both models. Both models were well-trained and demonstrated a significant position reward. It has been found in figures 6.3 and 6.4 accordingly.

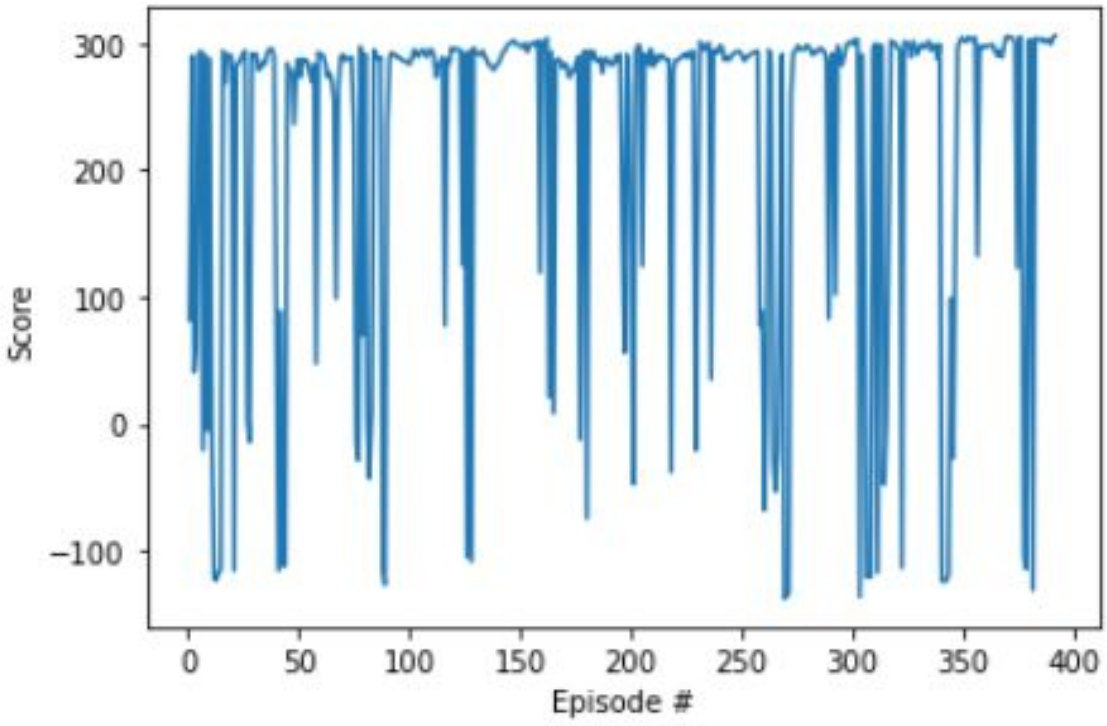

Figure 6.4: TD3 Algorithm score after 400 Episodes

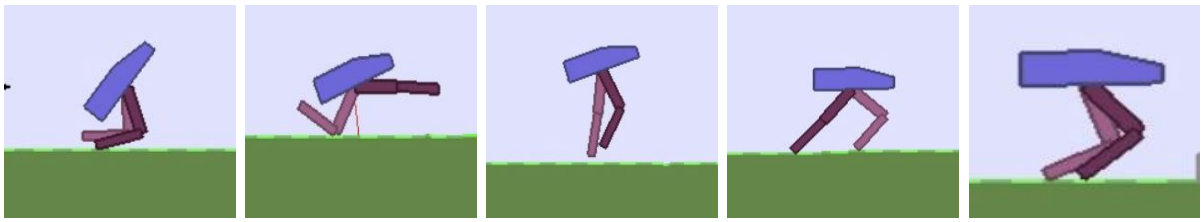

Figure 6.5: Walking movement of Bipedal Walker

The model's result was a motion clip demonstrating a simulated AI system in which a biped robot walks and runs over a field in such a continuous controls, just as in real life. (Zamora, Lopez, Vilches, and Cordero, 2016) The movement was made in a left-to-right manner. The simulated results for an episode following training through using both algorithms are shown in 6.5. (Garg, 2018)

Both models were found to have effectively trained the simulations to move ahead. We can observe from the graphs of both models that TD3 is more efficient since it achieves 300 awards in 400 episodes, whereas DDPG only achieve average 200 rewards in 500 episodes. In addition, training with the DDPG algorithm was found to be slower, since it only covered 400 episodes in 7 hours, compared to 500 episodes with the TD3 algorithm. (Garg, 2018)

CHAPTER 7

Conclusion and Future Work

This chapter summarises all of my previous work and suggests some basic recommendation for future work. (Chen, 2018)

The purpose of this research was to see how well Reinforcement Learning could be used to teach a robot simulation to walk in specific circumstances. (Garg, 2018) I selected and investigated two Deep Reinforcement Learning algorithms, DDPG and TD3, for this goal. To teach the agents for such a task, I investigated the effectiveness and feasibility of both algorithms. (Garg, 2018) I tested both algorithms by putting them together in the OpenAI Gym. I concluded that training with TD3 performed significantly faster compared training with DDPG overall. The Twin-Delayed Deep Deterministic Policy Gradient (TD3) method significantly enhanced the DDPG's learning performance and reliability in a difficult continuous control challenge. (Dankwa and Zheng, 2019)

Both algorithms are off-policy methods that utilise a deterministic targeted strategy including an actor-critic approach. The *"BipedalWalker-v2"* environment was used to test the algorithm's ability to operate with constant action-space as well as state-space. The implementation was highly effective, as shown by all the graphs, because the algorithms addressed the issue with such a wide variety of hyperparameter combinations. The continuous state, and the consistent objective strategy, are both necessary properties for robotic programs. (Ribeiro, 2019)

The code used in this dissertation is open source and available on GitHub. (Ribeiro, 2019)

The work presented in this dissertation is intended to serve as a foundation for future advancements in robotics.Both conceptual and applied elements may act as a base for future research by utilising the provided theoretical background and the specified solutions. (Ribeiro, 2019)

The following are the enhancements which have been chosen and would be researched in the future: (Ribeiro, 2019)

- Using a higher-performance computer, particularly one with graphics processing units (GPUs) or tensor processing units (TPUs).(Chen, 2018)
- Transfer learning can be used to increase an agent's performance. (Pilnan, 2019)
- Evaluating performance of this study with simulator like Mujoco, Gazebo and Pybullet.
- The study of other algorithms linked with recent accomplishments, such as Soft Actor Critic (SAC), Trust Region Policy Optimization (TRPO), and Proximal Policy Optimization (PPO) (Ribeiro, 2019)
- Experiment this study using OpenAI Baseline algorithms, which seem to be greater implementation of reinforcement learning.

CHAPTER 8

Appendix

The following necessary libraries are use to execute the code and environment is common for both algorithms.

```
import gym
import numpy as np
import random
import copy
import matplotlib.pyplot as plt
import pandas as pd
from collections import namedtuple, deque
import time
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

Listing 8.1: Libraries

Note : The OpenAI environment *BipedalWalker-v2* is deprecated, thus it declares it as *BipedalWalker-v3* in practical implementation, even though I specified it as *BipedalWalker-v2* in my entire thesis.

```
env_id = 'BipedalWalker-v3'
env = gym.make(env_id)
```

Listing 8.2: Environment

Code for Deep Deterministic Policy Gradient Algorithm

```
class Actor(nn.Module):

    def __init__(self, state_size, action_size, seed, fc_units=600, fc1_units=300):

        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc_units)
        self.fc2 = nn.Linear(fc_units, fc1_units)
        self.fc3 = nn.Linear(fc1_units, action_size)

        self.bn1 = nn.BatchNorm1d(fc_units)
        self.bn2 = nn.BatchNorm1d(fc1_units)
        self.reset_parameters()

    def reset_parameters(self):

        self.fc2.weight.data.uniform_(-1.5e-3, 1.5e-3)
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state):
        """Build an actor (policy) network that maps states -> actions."""
        x = F.relu((self.bn1(self.fc1(state))))
        x = F.relu((self.bn2(self.fc2(x))))
        return F.torch.tanh(self.fc3(x))
```

Listing 8.3: Actor Architecture

```
class Critic(nn.Module):

    def __init__(self, state_size, action_size, seed, fcs1_units=600, fcs2_units=300, fca1_units=300):

        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fcs1 = nn.Linear(state_size, fcs1_units)
```

```
        self.fcs2 = nn.Linear(fcs1_units, fcs2_units)
        self.fca1 = nn.Linear(action_size, fca1_units)
        self.fc1 = nn.Linear(fcs2_units, 1)
        self.bn1 = nn.BatchNorm1d(fcs1_units)
        self.reset_parameters()

    def reset_parameters(self):

        self.fcs2.weight.data.uniform_(-1.5e-3, 1.5e-3)
        self.fc1.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state, action):
        """Build a critic (value) network that maps (state, action) pairs -> Q-values."""
        xs = F.relu((self.bn1(self.fcs1(state))))
        xs = self.fcs2(xs)
        xa = self.fca1(action)
        x = F.relu(torch.add(xs, xa))
        return self.fc1(x)
```

Listing 8.4: Critic Architecture

```
BUFFER_SIZE = 1000000      # replay buffer size
BATCH_SIZE = 100           # minibatch size
GAMMA = 0.99               # discount factor
TAU = 0.001                # for soft update of target parameters
LR_ACTOR = 0.0001          # learning rate of the actor
LR_CRITIC = 0.001          # learning rate of the critic
WEIGHT_DECAY = 0.001       # L2 weight decay
```

Listing 8.5: Hyperparameter of DDPG

```
class OUNoise:
    """Ornstein-Uhlenbeck process."""

    def __init__(self, size, seed, mu=0., theta=0.4, sigma=0.2):
        """Initialize parameters and noise process."""
        self.mu = mu * np.ones(size)
        self.theta = theta
        self.sigma = sigma
        self.seed = random.seed(seed)
```

```
        self.reset()

    def reset(self):
        """Reset the internal state (= noise) to mean (mu)."""
        self.state = copy.copy(self.mu)

    def sample(self):
        """Update internal state and return it as a noise sample."""
        x = self.state
        dx = self.theta * (self.mu - x) + self.sigma * np.array([random.random() for i in range(len(x))])
        self.state = x + dx
        return self.state

class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""

    def __init__(self, action_size, buffer_size, batch_size, seed):
        """Initialize a ReplayBuffer object.
        Params
        ======
            buffer_size (int): maximum size of buffer
            batch_size (int): size of each training batch
        """
        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)  # internal memory (deque)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state", "action", "reward", "next_state", "done"])
        self.seed = random.seed(seed)

    def add(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)

    def sample(self):
        """Randomly sample a batch of experiences from memory."""
        experiences = random.sample(self.memory, k=self.batch_size)
```

```
47
48          states = torch.from_numpy(np.vstack([e.state for e in experiences if e
    is not None])).float().to(device)
49          actions = torch.from_numpy(np.vstack([e.action for e in experiences if e
     is not None])).float().to(device)
50          rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e
     is not None])).float().to(device)
51          next_states = torch.from_numpy(np.vstack([e.next_state for e in
    experiences if e is not None])).float().to(device)
52          dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is
    not None]).astype(np.uint8)).float().to(device)
53
54          return states, actions, rewards, next_states, dones
55
56      def __len__(self):
57          """Return the current size of internal memory."""
58          return len(self.memory)
```

Listing 8.6: Utils Classes

```
1  class Agent():
2      """Interacts with and learns from the environment."""
3
4      def __init__(self, state_size, action_size, random_seed):
5          """Initialize an Agent object.
6
7          Params
8          ======
9              state_size (int): dimension of each state
10             action_size (int): dimension of each action
11             random_seed (int): random seed
12         """
13         self.state_size = state_size
14         self.action_size = action_size
15         self.seed = random.seed(random_seed)
16
17         # Actor Network (w/ Target Network)
18
19         self.actor_local = Actor(state_size, action_size, random_seed).to(device
    )
```

```
20        self.actor_target = Actor(state_size, action_size, random_seed).to(
    device)
21        self.actor_optimizer = optim.Adam(self.actor_local.parameters(), lr=
    LR_ACTOR)
22
23        # Critic Network (w/ Target Network)
24        self.critic_local = Critic(state_size, action_size, random_seed).to(
    device)
25        self.critic_target = Critic(state_size, action_size, random_seed).to(
    device)
26        self.critic_optimizer = optim.Adam(self.critic_local.parameters(), lr=
    LR_CRITIC, weight_decay=WEIGHT_DECAY)
27
28        # Noise process
29        self.noise = OUNoise(action_size, random_seed)
30
31        # Replay memory
32        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE,
    random_seed)
33
34    def step(self, state, action, reward, next_state, done):
35        """Save experience in replay memory, and use random sample from buffer
    to learn."""
36        # Save experience / reward
37        self.memory.add(state, action, reward, next_state, done)
38
39        # Learn, if enough samples are available in memory
40        if len(self.memory) > BATCH_SIZE:
41            experiences = self.memory.sample()
42            self.learn(experiences, GAMMA)
43
44    def act(self, state, add_noise=True):
45        """Returns actions for given state as per current policy."""
46        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
47        self.actor_local.eval()
48        with torch.no_grad():
49            action = self.actor_local(state).cpu().data.numpy()
50        self.actor_local.train()
51        if add_noise:
```

```
52              action += self.noise.sample()
53          return action
54
55      def reset(self):
56          self.noise.reset()
57
58      def learn(self, experiences, gamma):
59
60          states, actions, rewards, next_states, dones = experiences
61
62          # ---------------------------- update critic
    ---------------------------- #
63          # Get predicted next-state actions and Q values from target models
64          actions_next = self.actor_target(next_states)
65          Q_targets_next = self.critic_target(next_states, actions_next)
66          # Compute Q targets for current states (y_i)
67          Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
68          # Compute critic loss
69          Q_expected = self.critic_local(states, actions)
70          critic_loss = F.mse_loss(Q_expected, Q_targets)
71          # Minimize the loss
72          self.critic_optimizer.zero_grad()
73          critic_loss.backward()
74          self.critic_optimizer.step()
75
76          # ---------------------------- update actor ----------------------------
     #
77          # Compute actor loss
78          actions_pred = self.actor_local(states)
79          actor_loss = -self.critic_local(states, actions_pred).mean()
80          # Minimize the loss
81          self.actor_optimizer.zero_grad()
82          actor_loss.backward()
83          self.actor_optimizer.step()
84
85          # ----------------------- update target networks -----------------------
     #
86          self.soft_update(self.critic_local, self.critic_target, TAU)
87          self.soft_update(self.actor_local, self.actor_target, TAU)
```

```
88
89      def soft_update(self, local_model, target_model, tau):
90
91          for target_param, local_param in zip(target_model.parameters(),
        local_model.parameters()):
92              target_param.data.copy_(tau*local_param.data + (1.0-tau)*
        target_param.data)
93
94  def ddpg(episodes, step, pretrained, noise):
95
96      if pretrained:
97          agent.actor_local.load_state_dict(torch.load('DDPG/pretrained/1
        checkpoint_actor.pth', map_location="cpu"))
98          agent.critic_local.load_state_dict(torch.load('DDPG/pretrained/1
        checkpoint_critic.pth', map_location="cpu"))
99          agent.actor_target.load_state_dict(torch.load('DDPG/pretrained/1
        checkpoint_actor_t.pth', map_location="cpu"))
100         agent.critic_target.load_state_dict(torch.load('DDPG/pretrained/1
        checkpoint_critic_t.pth', map_location="cpu"))
101
102     reward_list = []
103
104     for i in range(episodes):
105         state = env.reset()
106         score = 0
107
108         for t in range(step):
109
110             env.render()
111
112             action = agent.act(state, noise)
113             next_state, reward, done, info = env.step(action[0])
114             # agent.step(state, action, reward, next_state, done)
115             state = next_state.squeeze()
116             score += reward
117
118             if done:
119                 # print('Reward: {} | Episode: {}/{}'.format(score, i, episodes))
120                  print('Episode ', i,'finished with reward:', score)
```

```
                print('Finished at timestep ', t)
                break

        reward_list.append(score)

        if score >= 300:
            print('Task Solved')
            torch.save(agent.actor_local.state_dict(), 'DDPG/checkpoint_actor.
    pth')
            torch.save(agent.critic_local.state_dict(), 'DDPG/checkpoint_critic.
    pth')
            torch.save(agent.actor_target.state_dict(), 'DDPG/checkpoint_actor_t
    .pth')
            torch.save(agent.critic_target.state_dict(), 'DDPG/
    checkpoint_critic_t.pth')
            break

    torch.save(agent.actor_local.state_dict(), 'DDPG/checkpoint_actor.pth')
    torch.save(agent.critic_local.state_dict(), 'DDPG/checkpoint_critic.pth')
    torch.save(agent.actor_target.state_dict(), 'DDPG/checkpoint_actor_t.pth')
    torch.save(agent.critic_target.state_dict(), 'DDPG/checkpoint_critic_t.pth')

    print('Training saved')
    return reward_list
```

Listing 8.7: DDPG Agent

```
fig = plt.figure()
plt.plot(np.arange(1, len(scores) + 1), scores)
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.show()
```

Listing 8.8: Graph

```
time.sleep(1) #human is slow.
state = env.reset()
agent.reset()
total_reward = 0
ep_len = 0
noise = 1
```

```
while True:
    action = agent.act(state, noise)
    env.render()
    next_state, reward, done, info = env.step(action[0])
    state = next_state.squeeze()
    total_reward += reward
    if done:
        print(ep_len)
        break
    ep_len += 1
print(f"Reward:{total_reward}")
```

Listing 8.9: Test trained Agent

Code for Twin-Delayed Deep Deterministic Policy Gradient Algorithm

```
class Actor(nn.Module):
    def __init__(self, state_size, action_size, max_action, fc1=256, fc2=256):
        """
        Initializes actor object.
        @Param:
        1. state_size: env.observation_space.shape[0].
        2. action_size: env.action_space.shape[0].
        3. max_action: abs(env.action_space.low), sets boundary/clip for policy approximation.
        4. fc1: number of hidden units for the first fully connected layer, fc1. Default = 256.
        5. fc2: number of hidden units for the second fully connected layer, fc1. Default = 256.
        """
        super(Actor, self).__init__()

        #Layer 1
        self.fc1 = nn.Linear(state_size, fc1)
        #Layer 2
        self.fc2 = nn.Linear(fc1, fc2)
        #Layer 3
        self.mu = nn.Linear(fc2, action_size)

        #Define boundary for action space.
        self.max_action = max_action

    def forward(self, state):
        """Peforms forward pass to map state--> pi(s)"""
        #Layer 1
        x = self.fc1(state)
        x = F.relu(x)
        #Layer 2
        x = self.fc2(x)
        x = F.relu(x)
        #Output layer
```

```
        mu = torch.tanh(self.mu(x))#set action b/w -1 and +1
        return self.max_action * mu
```

Listing 8.10: Actor Architecture

```
class Critic(nn.Module):
    def __init__(self, state_size, action_size, fc1=256, fc2=256):
        """
        Initializes Critic object, Q1 and Q2.
        Architecture different from DDPG. See paper for full details.
        @Param:
        1. state_size: env.observation_space.shape[0].
        2. action_size: env.action_space.shape[0].
        3. fc1: number of hidden units for the first fully connected layer, fc1. Default = 256.
        4. fc2: number of hidden units for the second fully connected layer, fc1. Default = 256.
        """
        super(Critic, self).__init__()

        #---------Q1 architecture---------

        #Layer 1
        self.l1 = nn.Linear(state_size + action_size, fc1)
        #Layer 2
        self.l2 = nn.Linear(fc1, fc2)
        #Output layer
        self.l3 = nn.Linear(fc2, 1)#Q-value

        #---------Q2 architecture---------

        #Layer 1
        self.l4 = nn.Linear(state_size + action_size, fc1)
        #Layer 2
        self.l5 = nn.Linear(fc1, fc2)
        #Output layer
        self.l6 = nn.Linear(fc2, 1)#Q-value

    def forward(self, state, action):
        """Perform forward pass by mapping (state, action) --> Q-value"""
```

```
        x = torch.cat([state, action], dim=1) #concatenate state and action such that x.shape = state.shape + action.shape

        #---------Q1 critic forward pass---------
        #Layer 1
        q1 = F.relu(self.l1(x))
        #Layer 2
        q1 = F.relu(self.l2(q1))
        #value prediction for Q1
        q1 = self.l3(q1)

        #---------Q2 critic forward pass---------
        #Layer 1
        q2 = F.relu(self.l4(x))
        #Layer 2
        q2 = F.relu(self.l5(q2))
        #value prediction for Q2
        q2 = self.l6(q2)

        return q1, q2
```

Listing 8.11: Critic Architecture

```
class ReplayBuffer():
    """
    Implementation of a fixed size replay buffer as used in DQN algorithms.
    The goal of a replay buffer is to unserialize relationships between sequential experiences, gaining a better temporal understanding.
    """
    def __init__(self, buffer_size=BUFFER_SIZE, batch_size=MINI_BATCH):
        """
        Initializes the buffer.
        @Param:
        1. action_size: env.action_space.shape[0]
        2. buffer_size: Maximum length of the buffer for extrapolating all experiences into trajectories. default - 1e6 (Source: DeepMind)
        3. batch_size: size of mini-batch to train on. default = 64.
        """
        self.replay_memory = deque(maxlen=buffer_size) #Experience replay memory object
```

```
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state", "action", "reward", "next_state", "done"]) #standard S,A,R,S',done

    def add(self, state, action, reward, next_state, done):
        """Adds an experience to existing memory"""
        trajectory = self.experience(state, action, reward, next_state, done)
        self.replay_memory.append(trajectory)

    def sample(self):
        """Randomly picks minibatches within the replay_buffer of size mini_batch"""
        experiences = random.sample(self.replay_memory, k=self.batch_size)

        states = torch.from_numpy(np.vstack([e.state for e in experiences if e is not None])).float().to(device)
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not None])).float().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not None])).float().to(device)
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if e is not None])).float().to(device)
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None]).astype(np.uint8)).float().to(device)

        return states, actions, rewards, next_states, dones

    def __len__(self):#override default __len__ operator
        """Return the current size of internal memory."""
        return len(self.replay_memory)
```

Listing 8.12: Util Class

```
BUFFER_SIZE = 50_000 #max number of experiences in a buffer
MINI_BATCH = 100 #number of samples to collect from buffer
discount=0.99,  # discount factor
tau=0.005,      # for soft update of target parameters
exploration_noise = 0.1,
policy_noise=0.2,
noise_clip=0.5,
```

```
policy_freq=2
```

Listing 8.13: Hyperparameter of TD3

```
class Agent():
    """Agent that plays and learn from experience. Hyper-paramters chosen from paper."""
    def __init__(
            self,
            state_size,
            action_size,
            max_action,
            discount=0.99,
            tau=0.005,
            policy_noise=0.2,
            noise_clip=0.5,
            policy_freq=2
        ):
        """
        Initializes the Agent.
        @Param:
        1. state_size: env.observation_space.shape[0]
        2. action_size: env.action_size.shape[0]
        3. max_action: list of max values that the agent can take, i.e. abs(env.action_space.high)
        4. discount: return rate
        5. tau: soft target update
        6. policy_noise: noise reset level, DDPG uses Ornstein-Uhlenbeck process
        7. noise_clip: sets boundary for noise calculation to prevent from overestimation of Q-values
        8. policy_freq: number of timesteps to update the policy (actor) after
        """
        super(Agent, self).__init__()

        #Actor Network initialization
        self.actor = Actor(state_size, action_size, max_action).to(device)
        self.actor.apply(self.init_weights)
        self.actor_target = copy.deepcopy(self.actor) #loads main model into target model
        self.actor_optimizer = torch.optim.Adam(self.actor.parameters(), lr
```

```
=0.001)

        #Critic Network initialization
        self.critic = Critic(state_size, action_size).to(device)
        self.critic.apply(self.init_weights)
        self.critic_target = copy.deepcopy(self.critic) #loads main model into target model
        self.critic_optimizer = torch.optim.Adam(self.critic.parameters(), lr=0.001)

        self.max_action = max_action
        self.discount = discount
        self.tau = tau
        self.policy_noise = policy_noise
        self.noise_clip = noise_clip
        self.policy_freq = policy_freq
        self.total_it = 0

    def init_weights(self, layer):
        """Xaviar Initialization of weights"""
        if(type(layer) == nn.Linear):
            nn.init.xavier_normal_(layer.weight)
            layer.bias.data.fill_(0.01)

    def select_action(self, state):
        """Selects an automatic epsilon-greedy action based on the policy"""
        state = torch.FloatTensor(state.reshape(1, -1)).to(device)
        return self.actor(state).cpu().data.numpy().flatten()

    def train(self, replay_buffer:ReplayBuffer):
        """Train the Agent"""

        self.total_it += 1

        # Sample replay buffer
        state, action, reward, next_state, done = replay_buffer.sample()#sample 256 experiences

        with torch.no_grad():
```

```
            # Select action according to policy and add clipped noise
            noise = (
                torch.randn_like(action) * self.policy_noise
            ).clamp(-self.noise_clip, self.noise_clip)


            next_action = (
                self.actor_target(next_state) + noise #noise only set in training to prevent from overestimation
            ).clamp(-self.max_action, self.max_action)

            # Compute the target Q value
            target_Q1, target_Q2 = self.critic_target(next_state, next_action) #Q1, Q2
            target_Q = torch.min(target_Q1, target_Q2)
            target_Q = reward + (1 - done) * self.discount * target_Q #TD-target

        # Get current Q estimates
        current_Q1, current_Q2 = self.critic(state, action) #Q1, Q2

        # Compute critic loss using MSE
        critic_loss = F.mse_loss(current_Q1, target_Q) + F.mse_loss(current_Q2, target_Q)

        # Optimize the critic
        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        self.critic_optimizer.step()

        # Delayed policy updates (DDPG baseline = 1)
        if(self.total_it % self.policy_freq == 0):

            # Compute actor loss
            actor_loss = -self.critic(state, self.actor(state))[0].mean()

            # Optimize the actor
            self.actor_optimizer.zero_grad()
            actor_loss.backward()
            self.actor_optimizer.step()
```

```
104
105             # Soft update by updating the frozen target models
106             for param, target_param in zip(self.critic.parameters(), self.
    critic_target.parameters()):
107                 target_param.data.copy_(self.tau * param.data + (1 - self.tau) *
     target_param.data)
108
109             for param, target_param in zip(self.actor.parameters(), self.
    actor_target.parameters()):
110                 target_param.data.copy_(self.tau * param.data + (1 - self.tau) *
     target_param.data)
111
112
113     def save(self, filename):
114         """Saves the Actor Critic local and target models"""
115         torch.save(self.critic.state_dict(), "TD3/models/checkpoint/"+ filename
    + "_critic")
116         torch.save(self.critic_optimizer.state_dict(), "TD3/models/checkpoint/"
    + filename + "_critic_optimizer")
117
118         torch.save(self.actor.state_dict(), "TD3/models/checkpoint/" + filename
    + "_actor")
119         torch.save(self.actor_optimizer.state_dict(), "TD3/models/checkpoint/" +
     filename + "_actor_optimizer")
120
121
122     def load(self, filename):
123         """Loads the Actor Critic local and target models"""
124         self.critic.load_state_dict(torch.load("TD3/models/checkpoint/" +
    filename + "_critic", map_location='cpu'))
125         self.critic_optimizer.load_state_dict(torch.load("TD3/models/checkpoint/
    " + filename + "_critic_optimizer", map_location='cpu'))#optional
126         self.critic_target = copy.deepcopy(self.critic)
127
128         self.actor.load_state_dict(torch.load("TD3/models/checkpoint/" +
    filename + "_actor", map_location='cpu'))
129         self.actor_optimizer.load_state_dict(torch.load("TD3/models/checkpoint/"
     + filename + "_actor_optimizer", map_location='cpu'))#optional
130         self.actor_target = copy.deepcopy(self.actor)
```

```
131
132     #Set exploration noise for calculating action based on some noise factor
133 exploration_noise = 0.1
134
135 #Define observation and action space
136 state_space = env.observation_space.shape[0]
137 action_space = env.action_space.shape[0]
138 max_action = float(env.action_space.high[0])
139
140 #Create Agent
141 policy = Agent(state_space, action_space, max_action)
142
143 try:
144     policy.load("final")
145 except:
146     raise IOError("Couldn't load policy")
147
148 #Create Replay Buffer
149 replay_buffer = ReplayBuffer()
150
151
152 #Train the model
153 max_episodes = 500
154 max_timesteps = 2000
155
156 ep_reward = [] #get list of reward for range(max_episodes)
157
158 for episode in range(1, max_episodes+1):
159     avg_reward = 0
160     state = env.reset()
161     for t in range(1, max_timesteps + 1):
162         env.render()
163         # select action and add exploration noise:
164         action = policy.select_action(state) + np.random.normal(0, max_action *
    exploration_noise, size=action_space)
165         action = action.clip(env.action_space.low, env.action_space.high)
166
167         # take action in env:
168         next_state, reward, done, _ = env.step(action)
```

```
169         replay_buffer.add(state, action, reward, next_state, done)
170         state = next_state
171 
172         avg_reward += reward
173 
174         #Renders an episode
175         # env.render()
176 
177         if(len(replay_buffer) > 100):#make sure sample is less than overall
    population
178             policy.train(replay_buffer) #training mode
179 
180         # if episode is done then update policy:
181         if(done or t >=max_timesteps):
182             print(f"Episode {episode} reward: {avg_reward} | Rolling average: {
    np.mean(ep_reward)}")
183             print(f"Current time step: {t}")
184 
185             ep_reward.append(avg_reward)
186             break
187 
188     if(np.mean(ep_reward[-10:]) >= 300):
189         policy.save("final")
190         break
191 
192     if(episode % 100 == 0 and episode > 0):
193         #Save policy and optimizer every 100 episodes
194         policy.save(str("%02d" % (episode//100)))
195 env.close()
```

Listing 8.14: TD3 Agent

Bibliography

R. Acharyya. *A New Approach for Blind Source Separation of Convolutive Sources: Wavelet Based Separation Using Shrinkage Function*. VDM, Verlag Dr. Muller, 2008.

R. Agarwal, D. Schuurmans, and M. Norouzi. Striving for simplicity in off-policy deep reinforcement learning. *CoRR*, abs/1907.04543, 2019. URL http://arxiv.org/abs/1907.04543.

E. Alpaydin. *Introduction to Machine Learning*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, 3 edition, 2014. ISBN 978-0-262-02818-9.

R. C. Arkin, R. C. Arkin, et al. *Behavior-based robotics*. MIT press, 1998.

N. Bach, A. Melnik, M. Schilling, T. Korthals, and H. Ritter. Learn to move through a combination of policy gradient algorithms: Ddpg, d4pg, and td3. In G. Nicosia, V. Ojha, E. La Malfa, G. Jansen, V. Sciacca, P. Pardalos, G. Giuffrida, and R. Umeton, editors, *Machine Learning, Optimization, and Data Science*, pages 631–644, Cham, 2020. Springer International Publishing. ISBN 978-3-030-64580-9.

P. Bajaj. Reinforcement learning, May 2020. URL https://www.geeksforgeeks.org/what-is-reinforcement-learning/.

Y. E. Bayiz. Multi-agent actor-critic reinforcement learning for cooperative tasks. Master's thesis, Delft University of Technology, 2014.

J. Bell. *Machine Learning: Hands-On for Developers and Technical Professionals*. Wiley, Indianapolis, IN, 2014. ISBN 978-1-118-88906-0. URL http://my.safaribooksonline.com/9781118889060.

H. Benbrahim. Biped dynamic walking using reinforcement learning. Master's thesis, University of New Hampshire, 1996.

S. Bhatnagar, R. S. Sutton, M. Ghavamzadeh, and M. Lee. Natural actor-critic algorithms. *Automatica*, 45(11):2471â2482, Nov. 2009. ISSN 0005-1098. doi: 10.1016/j.automatica.2009.07.008. URL https://doi.org/10.1016/j.automatica.2009.07.008.

S. Bhatt. 5 things you need to know about reinforcement learning, March 2018. URL https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html.

J. Brownlee. Supervised and unsupervised machine learning algorithms, March 2016. URL https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/.

E. Burns. unsupervised learning. *In-depth guide to machine learning in the enterprise*, 2010. (to appear).

M. G. Castro. Q-learning, August 2020. URL https://www.eecs.tufts.edu/~mguama01/post/q-learning/.

S. Chen. Comparing deep reinforcement learning methods for engineering applications. Master's thesis, Otto-von-Guericke-University, 2018.

S. Dankwa and W. Zheng. 08 2019.

G. del Corral Tercero and M. Macias. Training a drone using ros and openai gym. 04 2018.

S. Edelkamp and S. Schrodl. Chapter 1 - introduction. In S. Edelkamp and S. Schrodl, editors, *Heuristic Search*, pages 3–46. Morgan Kaufmann, San Francisco, 2012. ISBN 978-0-12-372512-7. doi: https://doi.org/10.1016/B978-0-12-372512-7.00001-8. URL https://www.sciencedirect.com/science/article/pii/B9780123725127000018.

I. S. G. B. W. Z. Elon Musk, Sam Altman and J. Schulman. Getting started with gym, February 2018. URL https://gym.openai.com/docs/.

S. Fujimoto, H. Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1587–1596. PMLR, 2018.

L. Fussell. Exploring bipedal walking through machine learning techniques. Master's thesis, University Of Edinburgh, 2018.

U. Garg. Virtual robot climbing using reinforcement learning. Master's thesis, San Jose State University, 2018.

T. J. S. Geoffrey Hinton. *Unsupervised Learning Foundations of Neural Computation*. The MIT Press, One Rogers Street Cambridge, MA 02142-1209, 1999.

I. Grondman. *Online Model Learning Algorithms for Actor-Critic Control*. PhD thesis, 01 2015.

S. Gu, T. Lillicrap, I. Sutskever, and S. Levine. Continuous deep q-learning with model-based acceleration, 2016.

K. S. Hajash. Learning to collaborate robots building together. Master's thesis, Massachusetts Institute of Technology, 2018.

J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, Amsterdam, 3 edition, 2011. ISBN 978-0-12-381479-1. URL http://www.sciencedirect.com/science/book/9780123814791.

T. Hastie, R. Tibshirani, J. Friedman, and J. Franklin. The elements of statistical learning: Data mining, inference, and prediction. *Math. Intell.*, 27:83–85, 11 2004. doi: 10.1007/BF02985802.

N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. M. A. Eslami, M. Riedmiller, and D. Silver. Emergence of locomotion behaviours in rich environments, 2017.

L. Huang and K.-S. Ma. Introducing machine learning to first-year undergraduate engineering students through an authentic and active learning labware. In *2018 IEEE Frontiers in Education Conference (FIE)*, pages 1–4, 2018. doi: 10.1109/FIE.2018.8659308.

D. Isele. Lifelong reinforcement learning on mobile robots. Master's thesis, University of Pennsylvania, 2018.

S. Jaiswal. Supervised machine learning, November 2015. URL https://www.javatpoint.com/supervised-machine-learning.

L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *CoRR*, cs.AI/9605103, 1996. URL https://arxiv.org/abs/cs/9605103.

S. Karagiannakos. The idea behind actor-critics and how a2c and a3c improve them, November 2018. URL https://theaisummer.com/Actor_critics/.

D. Karunakaran. Deep deterministic policy gradient(ddpg) â an off-policy reinforcement learning algorithm, November 2020a. URL https://medium.com/intro-to-artificial-intelligence/deep-deterministic-policy-gradient-ddpg-an-off-policy-reinforcement-learn

D. Karunakaran. The actor-critic reinforcement learning algorithm, September 2020b. URL https://medium.com/intro-to-artificial-intelligence/the-actor-critic-reinforcement-learning-algorithm-c8095a655c14.

A. Kathuria. Getting started with openai gym: The basic building blocks, December 2020. URL https://blog.paperspace.com/getting-started-with-openai-gym/.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

J. Kober, J. A. Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013. doi: 10.1177/0278364913495721. URL https://doi.org/10.1177/0278364913495721.

V. Konda and J. Tsitsiklis. Actor-critic algorithms. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 2000. URL https://proceedings.neurips.cc/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

G. Kyziridis. Reinforcement learning algorithms in the bipedalwalker-v2 environment. Master's thesis, LIACS, Leiden University, 2017.

J. Laivamaa. Reinforcement q-learning using openai gym. Bachelor Thesis, March 2019.

T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning, 2019.

S. N. Mellatshahi. Learning control of robotic arm using deep q-neural network. Master's thesis, University of Windsor, 2020.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb 2015.

A. Mosavi and A. Varkonyi. Learning in robotics. *International Journal of Computer Applications*, 157(1):8–11, January 2017. doi: 10.5120/ijca2017911661. URL https://eprints.qut.edu.au/127556/.

E. Odemakinde. Model-based and model-free reinforcement learning â pytennis case study, July 2019. URL https://neptune.ai/blog/model-based-and-model-free-reinforcement-learning-pytennis-case-study.

OpenAI. Deep deterministic policy gradient, November 2018a. URL https://spinningup.openai.com/en/latest/algorithms/ddpg.html.

OpenAI. Twin delayed ddpg, November 2018b. URL https://spinningup.openai.com/en/latest/algorithms/td3.html?highlight=TD3.

X. B. Peng. Developing locomotion skills with deep reinforcement learning. Master's thesis, The University of British Columbia, 2017.

B. Pilnan. Exploring dynamic locomotion of a quadruped robot: a study of reinforcement learning for the anymal robot. Master's thesis, University of Edinburgh, 2019.

T. A. B. Pinto. Object detection with artificial vision and neural networks for service robots. Master's thesis, Minho University, 2019.

D. Rastogi. Deep reinforcement learning for bipedal robots. Master's thesis, Delft University of Technology, 2017.

T. A. Ribeiro. Deep reinforcement learning for robot navigation systems. Master's thesis, Universidade do Minho Escola de Engenharia, 2019.

J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. Trust region policy optimization, 2017.

J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation. *arXiv:1506.02438 [cs]*, Oct. 2018. URL http://arxiv.org/abs/1506.02438. arXiv: 1506.02438.

T. D. Science. Td3: Learning to run with ai, June 2019. URL https://towardsdatascience.com/td3-learning-to-run-with-ai-40dfc512f93.

A. Shantia. Automatic robot navigation using reinforcement learning. Master's thesis, University of Groningen, 2011.

D. Shewan. 10 companies using machine learning in cool ways, 2021. URL https://www.wordstream.com/blog/ws/2017/07/28/machine-learning-applications.

C. Shyalika. A beginners guide to q-learning, November 2019. URL https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c.

T. Siebel. Supervised learning, November 2012a. URL https://c3.ai/introduction-what-is-machine-learning/supervised-learning/.

T. Siebel. Unsupervised learning, November 2012b. URL https://c3.ai/introduction-what-is-machine-learning/unsupervised-learning/.

D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, page Iâ387âIâ395. JMLR.org, 2014.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

S. Tanwar. Bellman equation and dynamic programming, January 2019. URL https://medium.com/analytics-vidhya/bellman-equation-and-dynamic-programming-773ce67fc6a7.

I. The MathWorks. Twin-delayed deep deterministic policy gradient agents, October 2017. URL https://www.mathworks.com/help/reinforcement-learning/ug/td3-agents.html.

Y. Tian. Model free reinforcement learning with stability guarantee. Master's thesis, Delft University of Technology, 2019.

S. Verma. Teach your ai how to walk | solving bipedalwalker | openaigym, May 2019. URL https://towardsdatascience.com/teach-your-ai-how-to-walk-5ad55fce8bca.

A. Violante. Simple reinforcement learning: Q-learning, March 2019. URL https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56.

Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas. Sample efficient actor-critic with experience replay, 2017.

C. Watkins. *Learning from Delayed Rewards*. PhD dissertation, King's College, University of Cambridge, May 1989.

C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992. ISSN 1573-0565. doi: 10.1007/BF00992698. URL https://doi.org/10.1007/BF00992698.

F. Woergoetter and B. Porr. Reinforcement learning, March 2008. URL http://www.scholarpedia.org/article/Reinforcement_learning.

C. Xiao. Using machine learning for exploratory data analysis and predictive models on large datasets. Master's thesis, Universidade do Minho Escola de Engenharia, 2015.

I. Zamora, N. G. Lopez, V. M. Vilches, and A. H. Cordero. Extending the openai gym for robotics: a toolkit for reinforcement learning using ROS and gazebo. *CoRR*, abs/1608.05742, 2016. URL http://arxiv.org/abs/1608.05742.

Z. Zhang, X. Li, J. An, W. Man, and G. Zhang. Model-free attitude control of spacecraft based on pid-guide td3 algorithm. *International Journal of Aerospace Engineering*, 2020:8874619, Dec 2020. ISSN 1687-5966. doi: 10.1155/2020/8874619. URL https://doi.org/10.1155/2020/8874619.

H. Zhu, J. Yu, A. Gupta, D. Shah, K. Hartikainen, A. Singh, V. Kumar, and S. Levine. The ingredients of real-world robotic reinforcement learning. *CoRR*, abs/2004.12570, 2020. URL https://arxiv.org/abs/2004.12570.