Training a Quadruped Walker

A

Synopsis submitted

in the partial fulfillment of the requirements for the award of

the degree of Bachelor of Technology

In

Information Technology

By

Shweta Pandey 1805213057

Geeta 1805213021

Swati Singh 1805213060

Under the guidance of

Prof. Manish Gaur Dr. Jaswant Kumar

Department of Computer Science and Engineering

Institute of Engineering and Technology, Lucknow

Dr. A.P.J. Abdul Kalam Technical University, Lucknow, Uttar Pradesh.

May, 2022

Contents

Declaration

The work presented in project entitle “Training of a Quadruped Walker” submitted to the Department of Computer Engineering, Institute of Engineering and Technology, Lucknow, for the award of the degree of Bachelor of Technology in Information Technology, during the session 2021-22, is our original work. I have neither plagiarized nor submitted the same work for the award of any degree.

Submitted by:- Date: 17/05/2022

1. Name: Shweta Pandey
   Roll no.: 1805213057
   Branch: IT
   Signature: Shweta Pandey
2. Name: Geeta
   Roll no.: 1805213021
   Branch: IT
   Signature: Geeta
3. Name: Swati Singh
   Roll no.: 1805213060
   Branch: IT
   Signature: [illegible]

<u>Certificate</u>

This is to certify that the Project Report entitled “Training of a Quadruped Walker”, being submitted by Shweta Pandey (1805213057), Geeta (1805213021), Swati Singh (1805213060) in partial fulfilment of the requirement for the award of degree B. Tech. in Department of Computer Science and Engineering of Dr. APJ Abdul Kalam Technical University, is a record of the candidate own work carried out by him under our supervision. The matter embodied in this thesis is original and has not been submitted for the award of any other degree.

It is also certified that this project has not been submitted at any other Institute for the award of any other degrees to the best of my knowledge.

[illegible] 1/6/22

Prof. Manish Gaur
Department of Computer Science and Engineering
Institute of Engineering and Technology, Lucknow

[illegible] 1/6/2022

Dr. Jaswant Kumar
Department of Computer Science and Engineering
Institute of Engineering and Technology, Lucknow

Acknowledgment

It gives us a great sense of pleasure to present the report of the B. Tech Project undertaken during B. Tech. Final Year. We owe special debt of gratitude to Prof. Manish Gaur and Dr. Jaswant Kumar, Department of Computer Science, Institute of Engineering and Technology, Lucknow for their constant support and guidance throughout the course of our work. Their sincerity, thoroughness and perseverance have been a constant source of inspiration for us. It is only their cognizant efforts that our endeavours have seen light of the day.

We also take the opportunity to acknowledge the contribution of Prof D.S. Yadav, Head of Department of Computer Science & Engineering, Institute of Engineering and Technology, Lucknow for her full support and assistance during the development of the project.

We also do not like to miss the opportunity to acknowledge the contribution of all faculty and staff members of the department for their kind assistance and cooperation during the development of our project.

1. Name: Shweta Pandey
   Roll no.: 1805213057
   Signature: Shweta Pandey
2. Name: Geeta
   Roll no.: 1805213021
   Signature: Geeta
3. Name: Swati Singh
   Roll no.: 1805213060
   Signature: [illegible]

Abstract

Nowadays, design, development, and motion planning of a mobile robot explore research areas in the field of robotics. The design and development of a mobile robot is a crucial part and reinforcement learning is one of the most popular technique for training mobile robots. Among all the mobile robot, the quadrupedal robot is a legged robot, which is superior to wheeled and tracked robot due to its potential to explore in all the terrain like the human and animal.

Reinforcement learning is a versatile paradigm of Machine Learning that finds its origins in disparate fields of study ranging from animal learning, which is concerned with learning by trial and error, to the study of optimal control that relies on value functions and dynamic programming. Reinforcement learning delivers decisions. By creating a simulation of an entire business or system, it becomes possible for an intelligent system to test new actions or approaches, change course when failures happen (or negative reinforcement), while building on successes (or positive reinforcement). As a result, reinforcement learning finds application in multiple domains i.e.; from mastering the highly competitive game of Go to navigating the sparse-reward and incomplete state-based game of DOTA2, to allowing optimal control over a Polypod agents moving in a 3D environment.

For the purposes of this project, we have leveraged various reinforcement learning algorithms to train a Quadruped agent to navigate an environment built using the Unity Game Engine. With this project, we seek to extend the applicability of reinforcement learning to a realistic scenario of mobile robotic navigation. For that, we will use some of the basic python libraries like pyBullet, NumPy, Torch and we are also going to use matplotlib, neural networks. Here in our project we will be creating the simulation of ant, half cheetah and half humanoid. Mobile robots have an extensive area of applications in various fields like space exploration, military application, industrial use, and many more.

.

Acronyms

RL	Reinforcement Learning
DRL	Deep Reinforcement Learning
DDPG	Deep Deterministic Policy Gradient
TD3	Twin Delayed DDPG
PPO	Proximal Policy Optimization
TRPO	Trust Region Policy Optimization
A2C	Advantage Actor Critic
GAN	Generative Adversarial Networks
DQL	Deep Q-Learning
SGD	Stochastic Gradient Descent
MSE	Mean Square Error

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Motivation

Deep Reinforcement Learning has allowed computers to solve problems at a level that surpasses human ability. Despite the limited scope of these challenges, DRL has shown that it can handle pre-existing issues in unique ways that maximise a reward signal and converge to an optimum on a short time scale. This is not the case for an agent educated through imitation learning, i.e. supervised learning, which can only do as well as the agent it is attempting to emulate by definition. Consider the Atari game of pong, in which an agent watches hundreds of frames from a skilled human player and attempts to map the player's up-down movement to the correct frame. A new study named "Controllable Agent with Reinforcement Learning(CARL) for Quadruped Locomotion" highlights the efficiency of imitation learning combined with GANs to adapt high-level control and reinforcement learning for fine-tuning quadruped locomotion. This research advances robotic locomotion, but it is limited in scope because it can only navigate as effectively as the real-life organism on which it is based. Another drawback is that hyperparameter adjustment for such a model can have significant effects on the accuracy of the animal's locomotion. The inextensibility of such a concept when it comes to other Polypods is a last deterrent.

1.2 Objectives and Scope

For our project, we are training a custom-built Quadruped Walker using reinforcement learning algorithms. The walker agent is incentivized to navigate towards a target while being restricted to two degrees of freedom along each of its limbs. With this project, we aim to capture similar results from the CARL publication, albeit on a simpler character-model with fewer points of articulation.

1.3 Organization

In the following chapter, headed "Literature Review," we present an overview of Reinforcement Learning and the more current methods used in this project. In the next chapters, we will break down our project into two phases. In Phase 1, we use a prebuilt environment called "Ant-v2" that was created by OpenAI for the Gym library. To the best of our knowledge, this training environment was initially used in the publication "High-Dimensional Continuous Control Using Generalized Advantage Estimation."

Chapter 2
Literature Review

2.1 Introduction to Reinforcement Learning

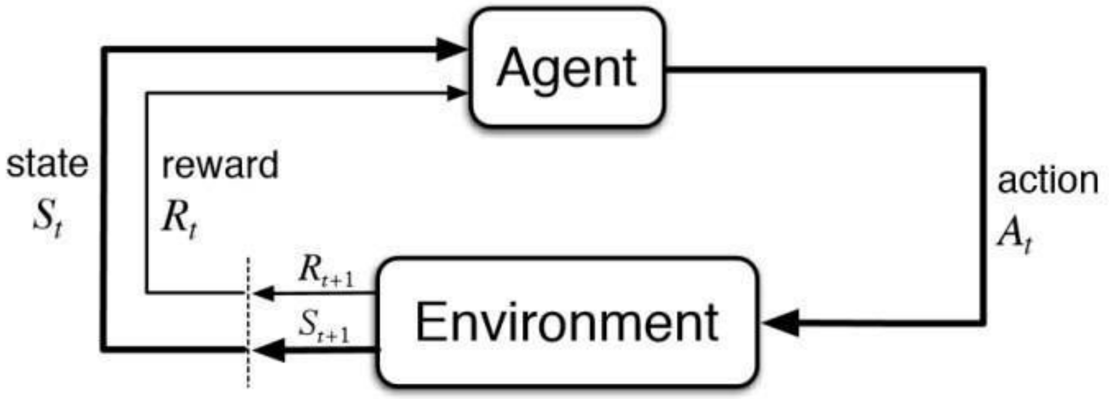


Figure 2.1: The reinforcement learning framework [Sutton and Barto, 1998]

The learner or decision-maker is referred to as the agent in [3] reinforcement learning, while everything around the agent is referred to as the environment. As demonstrated above, the learning process can be described as an interactive loop. The environment reached a new state St+1 when the agent at state St took an action At from a set of actions A, and the agent received a reward of Rt. The agent will be rewarded Rt+1 for further actions in state St+1. The loop can be programmed to continue until the end of an episode or to stop once the rewards converge to a certain maximum. The [7] cumulative reward for action taken at each time step t is represented as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\; where\; \gamma \in [0, 1)$$

Here γ is called the discount factor, for lower values of γ the future rewards are discounted and greedier actions is taken by the agent.

A mapping from state to action is defined as *policy*, while the probability of taking an action from a given state represents a *stochastic policy*. The *stochasti*

$$Q(s, a) = E[\sum_{n=0}^{N} \gamma^n r_n]$$

c policy π_π can be

represented as:

$$\pi_\theta(a|s) = P[a|s]$$

where γ represents a vector of parameters that have to be fine tuned inorder to select the best action for a given state. In order to select the best action we determine the expected total reward from state s and action a under a policy γ. The Action Value Function Qγ(s,a) is given by:

This method of determining value for a state-action pair is employed in Q-Learning; however, an alternate method relies on the value function of each state, i.e., the reward is the expected benefit from simply existing in a particular state s before any action is performed.

The models presented in sections 2.2 and 2.3 are built on the foundation of the subsections that follow.

2.3 Q Learning

Let's say we know the expected reward of each action at every step. This would essentially be like a cheat sheet for the agent! Our agent will know exactly which action to perform.

It will perform the sequence of actions that will eventually generate the maximum total reward. This total reward is also called the Q-value and we will formalise our strategy as:

$$Q(s,a) = r(s,a) + \gamma \max_a Q(s',a)$$

The above equation states that the Q-value yielded from being at state s and performing action a is the immediate reward r(s,a) plus the highest Q-value possible from the next state s'. Gamma here is the discount factor which controls the contribution of rewards further in the future.

Q(s',a) again depends on Q(s",a) which will then have a coefficient of gamma squared. So, the Q-value depends on Q-values of future states as shown here:

$$Q(s,a) \to \gamma Q(s',a) + \gamma^2 Q(s'',a) \dots\dots\dots \gamma^n Q(s''^{\dots n},a)$$

Adjusting the value of gamma will diminish or increase the contribution of future rewards. Since this is a recursive equation, we can start with making arbitrary assumptions for all q-values. With experience, it will converge to the optimal policy. In practical situations, this is implemented as an update:

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

where alpha is the learning rate or step size. This simply determines to what extent newly acquired information overrides old information

2.3 Deep Q-Learning

Conventional Q-Learninig is simple to use in contexts with a minimal number of states and actions. Because the number of state-action pairs in more complicated systems is substantially higher, the amount of time necessary to explore each state and construct Q-tables becomes unfeasible.

We can utilise neural networks to estimate these Q- values in Deep-Q Learning. The states are given as input, and the output is the Q-value of all feasible actions.

In deep Q-learning, we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output. The comparison between Q-learning & deep Q-learning is wonderfully illustrated below:

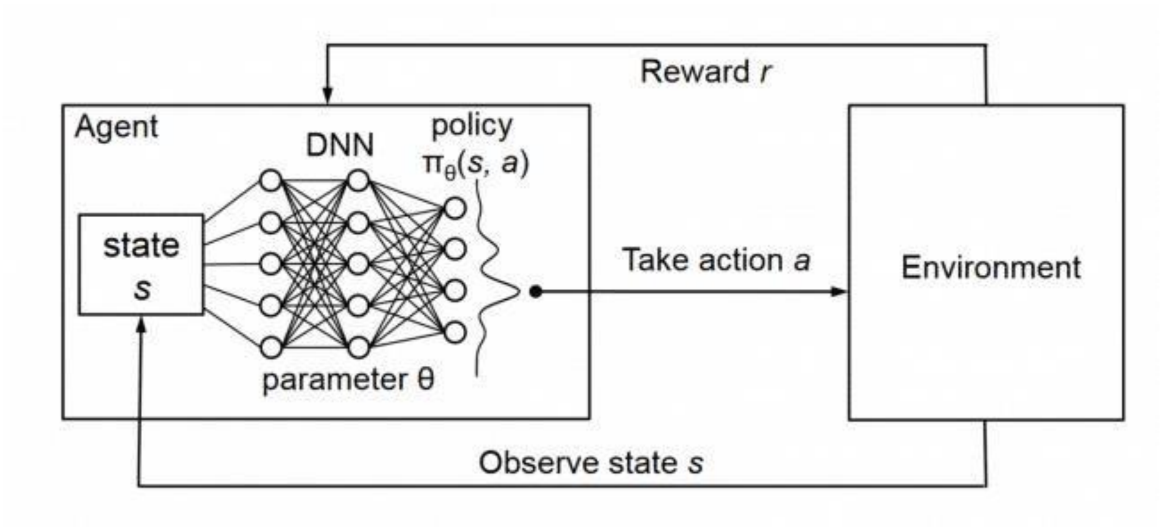


Figure 2.2: Deep Q-learning framework

Following is the steps procedure involved in Deep Q-Learning

Initialization:

1. The memory of the Experience Relay is initialized to an empty list M.
2. We choose a maximum size of the memory.

At each time *t*, we repeat the following process until the end of the epoch:

1. Predict the Q-values of the current state state s_t
2. We play the action that has the highest Q-value: *at = argmax{Q(st, a)}*
3. We get the reward *R(st, at)*
4. We reach the next state state *st+1*
5. We append the transition (*st, at, rt, st+1)* in the memory M.
6. We take a random batch $B \subset M$ of transitions. For all the transitions $s_{tB,}$ *atB, rtB, stB+1* of the random batch *B:*
    a. We get the predictions: *Q(stB, atB)*

b. We get the targets: *R(stB, atB) + γmax(Q(stB+1, a))*

c. We compute the loss between the predictions and the atargets over the whole batch *B:*

$$Loss = \frac{1}{2}\sum_{B}(R(s_{t_B}, a_{t_B}) + \gamma \max_{a}(Q(s_{t_B+1}, a)) - Q(s_{t_B}, a_{t_B}))^2 = \frac{1}{2}\sum_{B} TD_{t_B}(s_{t_B}, a_{t_B})^2$$

d. We backpropagate this loss error back into the neural network, and through stochastic gradient descent we update the weights according to how much they contribute to the loss error.

2.3.1 Why 'Deep' Q-Learning?

To develop a cheat sheet for our agent, we used Q-learning, a simple but powerful technique. This aids the agent in determining which action to take.
But what if this cheat sheet is very lengthy? Consider a world with 10,000 states and 1,000 actions for each state. This would result in a 10 million cell table. Things are about to spiral out of control!
It's evident that we can't infer the Q-value of new states from those that have already been studied. This raises two issues:
First, as the number of states grows, the amount of memory required to save and update that table grows.
Second, the time required to investigate each state in order to generate the required Q-table would be unfeasible. What if we used machine learning models such as a neural network to approximate these Q-values? This was the concept of DeepMind's algorithm, which led to its $500 million acquisition by Google.

2.4 Policy Gradients

The goal of [4] Policy Gradient methods is to directly improve the policy to maximize the RL objective and solve reinforcement learning problems by using parameterized policy γ, which is to maximize the cumulative reward from time step t=0 to time step t=T, where T is either finite for episodic problems or infinite for infinite horizon problems.
The return is given by:

$$R_t = \sum_{i=t}^{T} \gamma^{i-t} r(s_i, a_i)$$

The *expected return* is given by:

$$J(\phi) = E_{s_i \sim p_\pi, a_i \sim \pi}[R_0]$$

Here, expected return begins at R0 and follows the probability distribution given by p across all states, as well as the policy for actions a. The symbol represents all of the policy's parameters. We compute the gradient of the expected return with respect to and use gradient ascent to update the parameters:

$$\phi_{t+1} = \phi_t + \alpha \nabla_\phi J(\pi_\phi)|_{\phi_t}$$

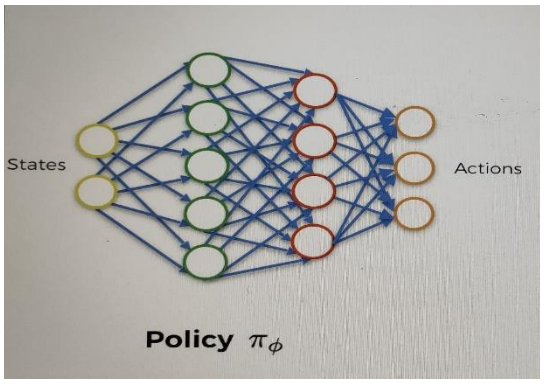


Figure 2.3 Policy Gradient

2.5 Actor Critic Model

Q-Learning and DQL are two examples of Value-Based approaches that attempt to identify the approximate optimal value function. We also looked at Policy Gradients, a Policy-Based technique that aims to identify the best policy without using Q-values. While each method has advantages, such as being better for continuous and stochastic situations and having a faster convergence, value-based methods are more sample efficient and steady but lack in specific areas.

The Actor-Critic model tries to maximize the benefits of both value-based and policy-based algorithms while minimizing their disadvantages. The model is divided into two parts: one for computing an action based on a state and another for producing the action's Q-values. As a result, the total design will learn to outperform the two methods individually. The critic model returns a reward for each action performed by the actor, given by

$$R_t = \sum_{i=t}^{T} \gamma^{i-t} r(s_i, a_i)$$

The goal of the critic is same as in section 2.1.2, i.e.; to maximize the expected reward. In actor critic model, the policy, known as the actor, can be updated through the deterministic policy gradient algorithm:

$$\nabla_\phi J(\phi) = E_{s \sim p_\pi}[\nabla_a Q^\pi(s,a)|_{a=\pi(s)} \nabla_\phi \pi_\phi(s)]$$

The parameters of the actor(i.e. policy) are updated through gradient ascent:

$$\phi_{t+1} = \phi_t + \alpha \nabla_\phi J(\pi_\phi)|_{\phi_t}$$

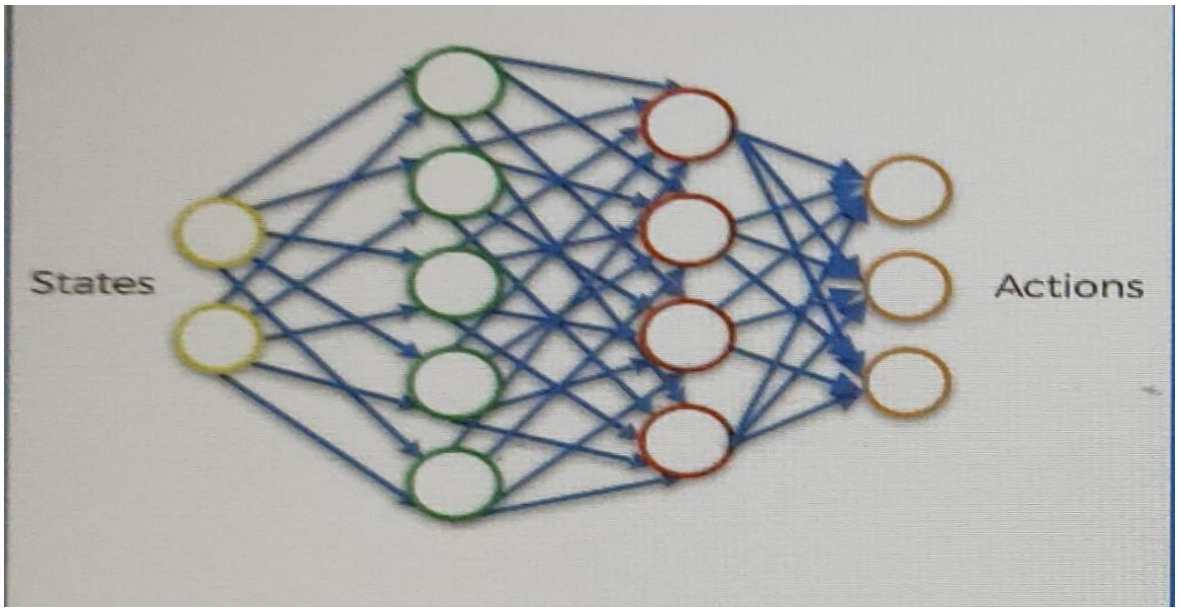


Figure 2.4: Actor

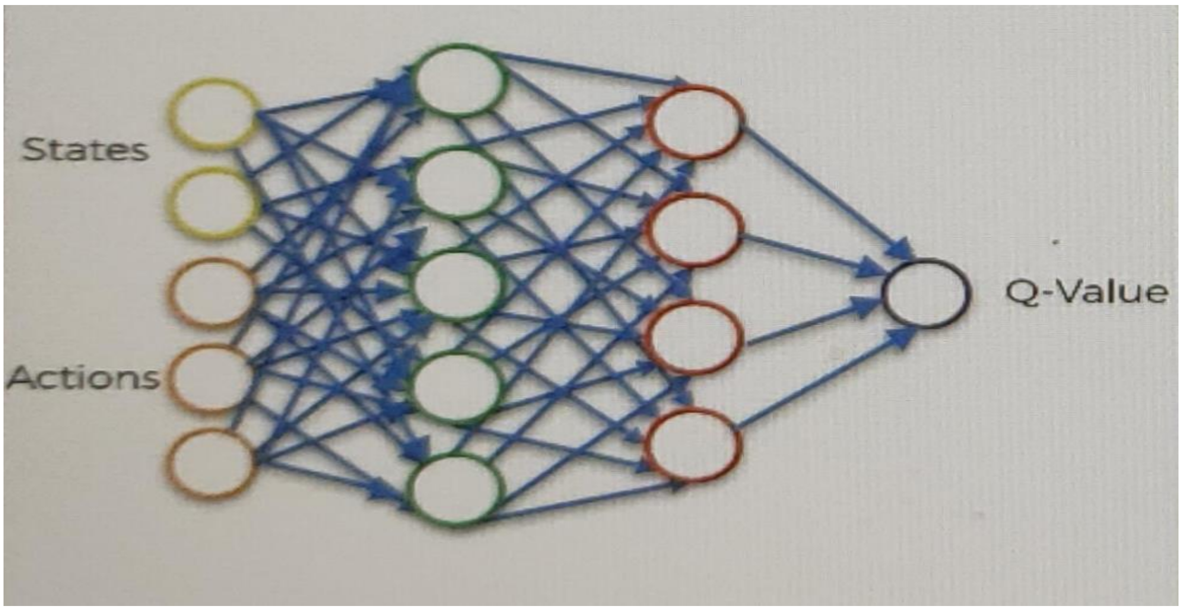


Figure 2.5: Critic

2.6 Artificial Neural Network

The term "artificial neural network" refers to a biologically inspired artificial intelligence sub-field that is designed after the brain. A computational network based on biological neural networks that construct the structure of the human brain is known as an artificial neural network. Artificial neural networks, like human brains, have neurons that are coupled to each other in various layers of the networks. Nodes are the name for these neurons.

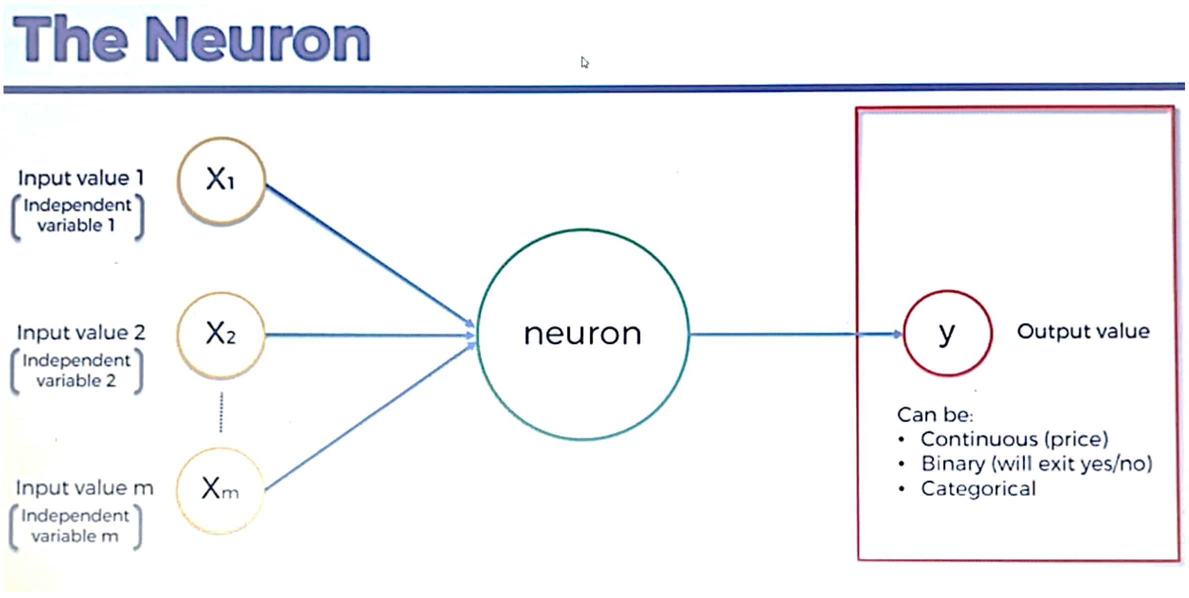


Figure 2.6: Internal Structure of Neuron

2.6.1 Activation Function

The activation function calculates a weighted total and then adds bias to it to determine whether a neuron should be activated or not. The activation function's goal is to introduce non-linearity into a neuron's output.

We know that neurons in a neural network work in accordance with their weight, bias, and activation function. We would change the weights and biases of the neurons in a neural network based on the output error. Back-propagation is the term for this procedure. Because the gradients are supplied simultaneously with the error to update the weights and biases, activation functions enable back-propagation.

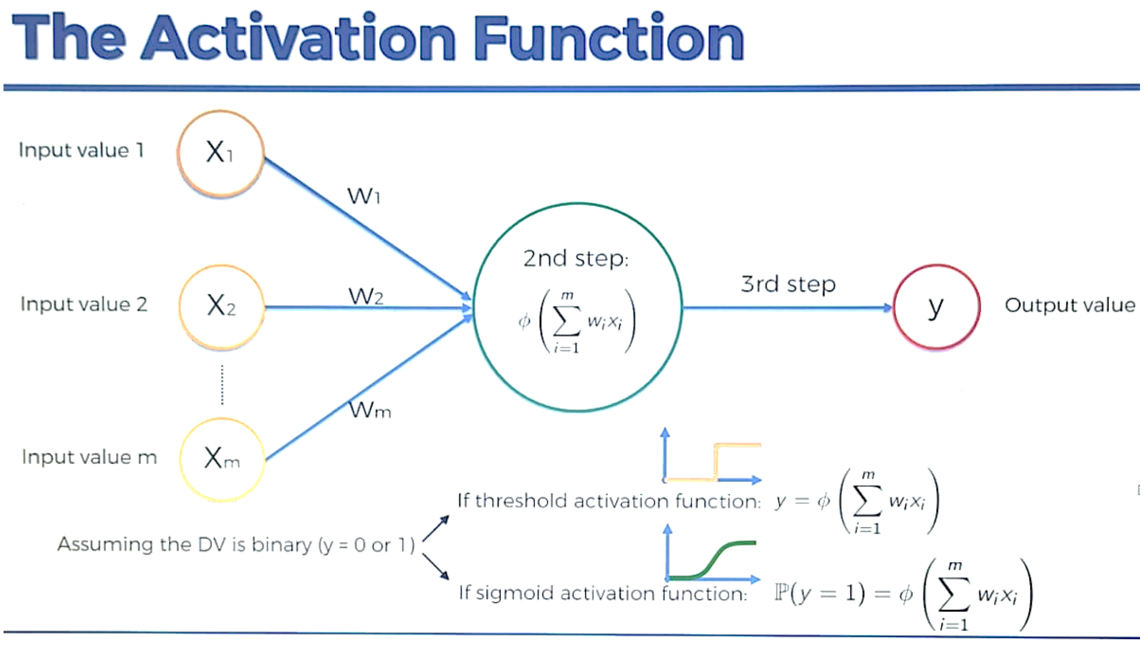

The Activation Function
Input value 1
Input value 2
Input value m
X_1
X_2
X_m
W_1
W_2
W_m
2nd step:
$\phi\left(\sum_{i=1}^{m} w_i x_i\right)$
3rd step
y
Output value
Assuming the DV is binary (y = 0 or 1)
If threshold activation function: $y = \phi\left(\sum_{i=1}^{m} w_i x_i\right)$
If sigmoid activation function: $\mathbb{P}(y=1) = \phi\left(\sum_{i=1}^{m} w_i x_i\right)$

Figure 2.7: Working of Activation Function

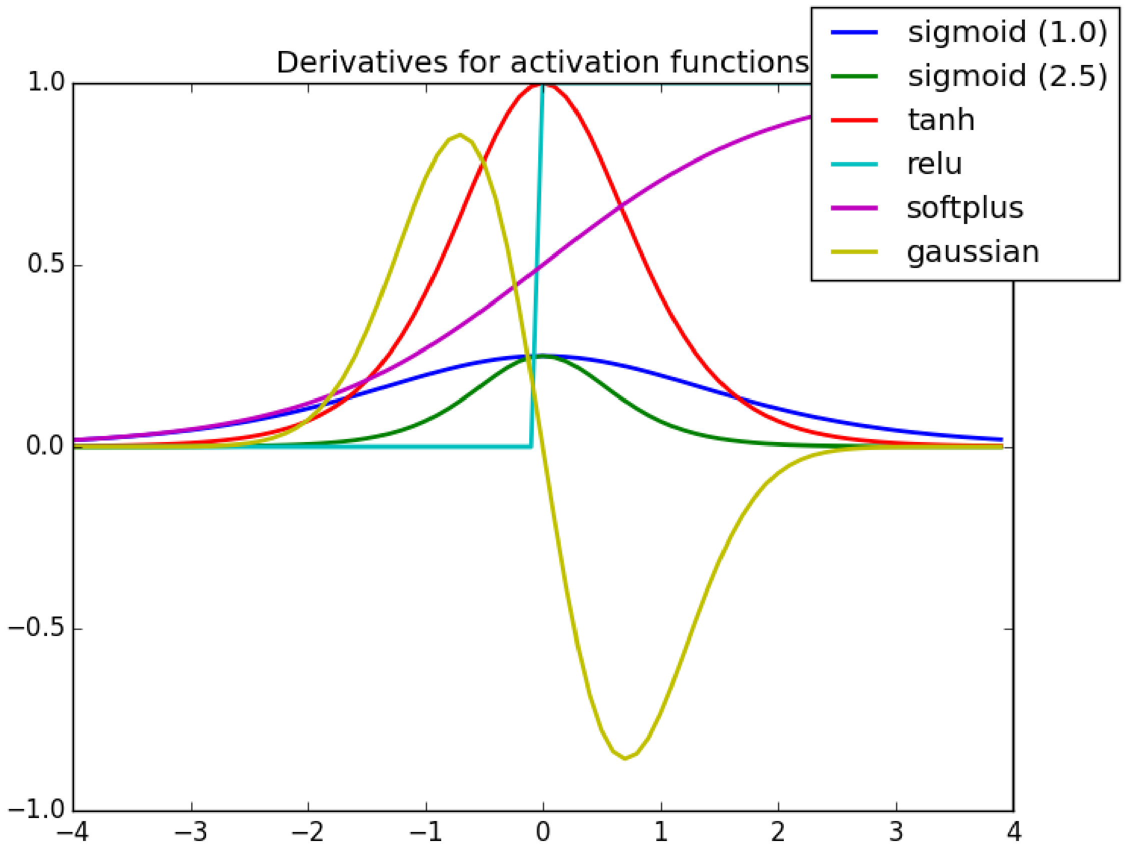


Figure 2.8: Types of activation Function

2.6.2 Working of Neural Network:

Consider each node as a separate linear regression model, with input data, weights, a bias (or threshold), and an output. This is what the formula would look like:

$$\sum_{i=1}^{m} w_i x_i + bias = w_1 x_1 + w_2 x_2 + w_3 x_3 + bias$$

Weights are assigned after an input layer is defined. These weights are used to determine the importance of each variable, with larger ones contributing more to the output than smaller ones. The sum of all inputs is then multiplied by their corresponding weights. The output is then run through an activation function to determine the output. If the output reaches a certain threshold, the node "fires" (or activates), sending data to the network's next tier. As a result, one node's output becomes the input for the next node.
We'll use a cost (or loss) function to evaluate the model's accuracy as we train it. The mean squared error (MSE) is another term for this (MSE).

- I is the sample index
- y-hat is the anticipated outcome
- y is the actual value
- m is the number of samples in the equation below

$$\text{Cost Function} = MSE = \frac{1}{2m}\sum_{i=1}^{m}(\hat{y} - y)^2$$

Finally, we want to minimize our cost function to ensure that any observation fits correctly. The model employs the cost function and reinforcement learning to approach the point of convergence, or the local minimum, as it adjusts its weights and bias. Gradient descent is the method by which the algorithm modifies its weights, allowing the model to discover the best path to minimize mistakes (or minimize the cost function). The parameters of the model adjust with each training case to progressively converge at the minimum.

2.6.3 Gradient Descent

[8] Gradient Descent is a generic function for minimizing a function, such as the Mean Squared Error cost function in this example.
Gradient Descent essentially does the same thing we did by hand: it incrementally changes the theta values, or parameters, until we reach a minimum.
We begin by setting theta0 and theta1 to any two values, such as 0 for both, and then proceed. The algorithm is written as follows:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \qquad (\text{for } j = 0 \text{ and } j = 1)$$

where α, alpha, is the learning rate, or how quickly we want to move towards the minimum. If α is too large, however, we can overshoot.

2.7 Deep Deterministic Policy Gradient

DDPG is a model-free off-policy actor-critic algorithm that combines Deep-Q Learning (DQN) and DPG. Original DQN works in a discrete action space and DPG extends it to the continuous action space while learning a deterministic policy.
As it is an off-policy algorithm, it uses two separate policies for the exploration and updates. It uses a stochastic behavior policy for the exploration and deterministic policy for the target update.
DDPG is an actor-critic algorithm; it has two networks: actor and critic. Technically, the actor produces the action to explore. During the update process of the actor, TD error from a critic is used. The critic network gets updated based on the TD error similar to Q-learning update rule.
We did learn the fact that the instability issue that can raise in Q-learning with the deep neural network as the functional approximator. To solve this, experience replay and target networks are being used.

2.8 Problems in Deep Deterministic Policy Gradient

- In DDPG, function approximation errors are known to lead to overestimated value estimates, suboptimal policies and divergent behavior.
- Also, due to the slow-changing policy in an actor-critic setting, the current and target value estimates remain too similar to avoid maximization bias. So we improved the DDPG, to an actor-critic format by using a pair of independently trained critics and to address the coupling of value and policy, we delayed policy updates until the value estimate has converged which is called Twin-Delayed Deep Deterministic Policy Gradient.
- Hence, here the word "Twin" represents the two independent critic model and target model and the "Delayed" represents the delaying of policy updates.

2.9 Twin Delayed Deep Deterministic Policy Gradient

The Twin Delayed Deep Deterministic Policy Gradient (TD3) Model is a new Reinforcement Learning innovation that produces consistent results in a continuous action space.
Deep Q-Learning is effective in discrete action space but not in continuous action space. Each state's award in DQL is set by

$$R(s_{t_B}, a_{t_B}) + \gamma \underset{a}{max}(Q(s_{t_B+1}, a))$$

Here the max term is easy to calculate for a discrete action space but is infeasible for a continuous action space with infinite state-action pairs.
The initial approach to this problem is to use simultaneous actor and critic models. The Actor-Critic model, on the other hand, suffers functional [4]approximation errors (Fujimoto et al. 2018), which means it overestimates values and generates suboptimal policies. To address this issue, the Actor-Critic Model's critic model is replaced with two critic models, hence the term "Twin." These two models begin similarly but diverge with time, yielding different Q-values for each state-action pair.
Following is the step procedure for implementing the TD3 Algorithm as explained by Fujimoto et. al:

Step1: Initialize the Replay memory

- Initialize the Experience Replay memory. This will store past transitions from which the critic models will learn the Q-values.
- Each transition is composed of *(s, s', a, r)* - the current state, the next state, the action, and the reward.

Step2: Build one neural network for the Actor model and one for the Actor target

- The actor target is initialized the same as the actor model, and the weights will be updated over the iterations by returning optimal actions.
- For our implementation we have used the architecture as mentioned in Appendix C of [1] *"Addressing Function Approximation Error in Actor- Critic Methods"* by Fujimoto et. al.

Step3: Build two neural networks for the two Critic models, and two neural networks for the two Critic targets

- The architecture for these networks is also based on Appendix C of the aforementioned publication.
- The actors learn the policy, while the critics learn the Q-values
- In total we have 2 Actor neural networks, and 4 critic neural networks.
- The neural networks work together following this sequence:
  - Actor Target → Critic Target → Critic Target → Critic Model → Critic Model→ Actor Model
  - The actor target outputs an action from a given state, this state- action pair is fed into the two critic targets which then produce a Q- value. The critic models use these Q-values to generate an output Q-value which is then used to adjust the weights of the actor model in-order to select the best action for a given state.

Training Process:

Step4: Sample a batch of transitions *(s,s',a,r)* from the memory

- A number of actions need to played randomly in-order to populate the experience replay memory before sampling can take place.

Step5: From the next state *s',* the Actor target plays the next action a'.

Step6: Add Gaussian noise to the next action a' and put the values in a range of values supported by the environment.

- Adding the Gaussian noise encourages exploration, and avoids the agent from getting stuck to one state, and potentially avoiding a bad state.

Q-Learning Process:

Step7: The two Critic Targets each take the couple *(s',a')* as input and return two Q-values: *Qt1(s',a')* and *Qt2(s',a')*

Step8: The minimum of these two Q-values is kept

- The original problem in DQL when applied to a continuous action-space was it's inability to approximate the Q-value of the next state given by:
  $Q(s_{t_B+1})$
- To over come this issue the minimum of the two Q-values act as a substitute for the next states Q-value, thereby allowing Q-Learning on a continuous action-space.
- The Actor-Critic model relied on overly optimistic Q-values of next state hence ran into approximation errors, this new approach for estimating Q-value of s_{t+1} does away with that and provides a conservative update, which adds to the stability of the model.

Step9: We use $min(Q_{t1}, Q_{t2})$ to get the final target of the two Critic Models:

$$Q_t = R + \gamma(min(Q_{t1}, Q_{t2}))$$

Step10: The two Critic models take each the couple *(s,a)* as input and return two Q-values *$Q_1(s,a)$* and *$Q_2(s,a)$* as outputs

- We now compare the minimum Critic target with the two Q-values from the Critic Models

Step11: Compute the loss between the two Critic models

$$CriticLoss = MSELoss(Q_1(s,a), Q_t) + MSELoss(Q_2(s,a), Q_t)$$

Step12: We backpropagate this Critic Loss and update the parameters of the two Critic models with SGD Optimizer

- To reduce the Critic Loss we update the parameters of the two Critic Models by backpropagation. The SGD Optimizer used is the Adam Optimizer.
- This is the final step of the Q-Learning part of the training process.

Policy Learning Process:

The purpose of policy learning is to determine the best Actor Model (primary policy) parameters in order to do the best actions in each state to maximize the expected return. Instead of having an explicit formula for expected return, we have a Q-value that is positively connected with it. By differentiating their output with respect to the weights of the Actor model, we will perform gradient ascent using the Q-values from the Critic Models. The Actor model's weights are then changed in the direction of increasing the Q-value. As a result, the agent returns a better action, increasing the state-action pair's Q-value and bringing the agent closer to the optimal return.

Step13: Every 2 interactions we update the Actor Model by performing gradient ascent on the output of the first Critic Model:

$$\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s,a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$$

where γ and $\gamma 1$ are the weights of the Actor and the Critic

Step14: We update the weights of the Actor Target every 2 iterations by Polyak averaging

$$\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$$

here γ'i represents the parameters of the Actor Target, γi represents the parameters of the Actor Model, and γ is a number equal to ~0.005. Polyak averaging the Actor Target gets closer to the actor model over multiple iterations, this slow update allows for the actor model to learn from the actor target - thereby stabilizing the learning process.

Step15: We update the weights of the Critic Targets every 2 iterations by Polyak averaging.

$$\phi' \leftarrow \tau\phi + (1-\tau)\phi'$$

here γ' represents the parameters of the critic target, and γ represents the parameters of the critic model.

The word "Delayed" refers to the fact that every other iteration, the update of the Actor Model, Actor Target, and the two Critic Targets is postponed, i.e., an update is only made every two iterations. In comparison to the DDPG model, this delay results in an improvement.

Chapter 3
Methodology

We used Scott Fujimoto's implementation of the TD3 method from his publication [1] "Addressing Function Approximation Error in Action Methods" as well as his Github repository for our implementation.

In contrast to Fujimoto et alMuJoCo .'s implementation, we have chosen to train a Quadruped Walker using Pybullet Physics Engine, a free to use Python toolkit.

Below is a layout of the architecture for the Twin Delayed Deep Deterministic Policy Gradient Model.

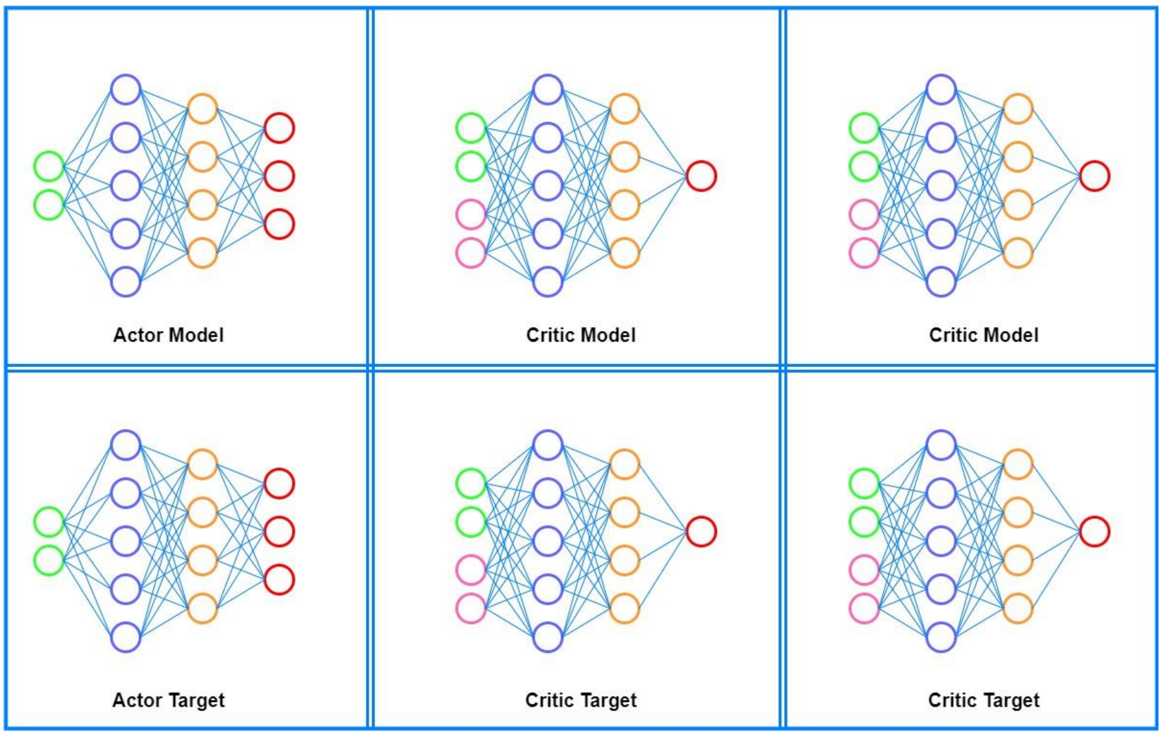


Figure 3.1: Architecture of TD3

This diagram will be expanded upon at the end of the section 3.1

3.1 Implementation

We implement the step-by-step procedure mentioned in Section 2.2.

Step1: The Experience Replay Memory Class with a maximum size of 10^6 is initialized

- The class populates a storage list with new transitions. Every element added after the storage limit has been reached is moved to the top of the list.
- Transitions can be sampled from memory in four batches at random

Step2: We initialize the Actor class for Actor Model and Actor Target.

- From state to 400 hidden neurons, 400 hidden neurons to 300 hidden neurons, and then to actions, both this actor model and actor target have three transformations.
- The class has a method to forward propagate a state to a set of actions.
- The max_action parameter is used to rescale the tanh output values.

Step3: We initialize the Critic class for twin Critic Models and Twin Critic Targets
- The class has a method to forward propagate state-action pair to obtain Q- values for the twin networks.
- There is a separate function to forward propagate a single Critic Model in case of gradient ascent calculation

Training Process:
- Create a TD3 training class.
- Instantiate Actor Model and Actor Target from Actor Class.
- Instantiate Twin Critic Models and Twin Critic Targets from the Critic Class.
- Create select_action() method to forward propagate actions from state
- Create train method

Step4: Sample a batch of transitions from replay_buffer Class
- Convert batch_states, batch_next_states, batch_actions, batch_rewards into torch tensors in order to work with the neural network architecture

Step5: From the next_state the Actor Target plays the next_action.
- next_action = self.actor_target(next_state)

Step6: Add Gaussian noise to the next_action and clamp noise in a range supported by the environment

Step7: From the next_state, and next_action the two Critic Targets generate target Q-values
- target_Q1, target_Q2 = self.critic_target(next_state, next_action)

Step8: We retrieve the minimum of the two Q-values
- target_Q = torch.min(target_Q1, target_Q2)

Step9: We get the final target of the two Critic Models, given by $Q_t = r + \gamma * \min(Q_{t1}, Q_{t2})$
- The term $\min(Q_{t1}, Q_{t2})$ is used to account for future rewards generated in a continuous action-space. In the case when we’re executing the last state in an episode we need to remove this term as the future reward will be zero.
- To account for this last state we use a bool named *“done”* which is 1 when the episode is over
    - target Q = reward + ((1 - done) * discount * target_Q).detach()
    - detach() is used to separate the target_Q from the Torch Computational Graph, this is done to allow addition with the reward tensor.

Step10: From the state, and action the two Critic Models generate Q-values
- current_Q1, current_Q2 = self.critic(state, action)

Step11: We compute the loss from the twin Critic Models

- The ms_loss() function from the Functional Module in PyTorch is used.
- critic_loss = F.mse_loss(current_Q1, target_Q) + F.mse_loss(current_Q2, target_Q)

Step12: We backpropagate the Critic loss and update the parameters of the two Critic Models with an SGD optimizer

- First we initiliaze the Critic Adam-Optimizer by setting the gradient to zero
  self.critic_optimizer.zero_grad()
- Differentiate the critic_loss with respect to the weights in the critic model
  critic_loss.backward()
- Update the weights inside the twin Critic Models self.critic_optimizer.step()

Step13: Once every two iterations we update the Actor Model by gradient ascent

- Perform stochastic gradient descent on the negative of Q-value output from the first Critic Model
  actor_loss = -self.critic.Q1(state, self.actor(state)).mean()
- The following steps correspond to the Deterministic Policy Gradient:
  self.actor_optimizer.zero_grad()
  actor_loss.backward()
  self.actor_optimizer.step(
)

Step14: Once every two iterations we update the Actor Target by Polyak averaging

- To account for the tau-based update of the Actor Target we loop over the weights
  for param, target_param in zip(self.actor.parameters(), self.actor_target.parameters()):
  target_param.data.copy_(tau*param.data+(1-tau)*target_param.data)

Step15: Once every two iterations we update the Critic Target by Polyak averaging

- To account for the tau-based update of the Actor Target we loop over the weights
  for param, target_param in zip(self.critic.parameters(), self.critic_target.parameters()):
  target_param.data.copy_(tau*param.data+(1-tau)*target_param.data)

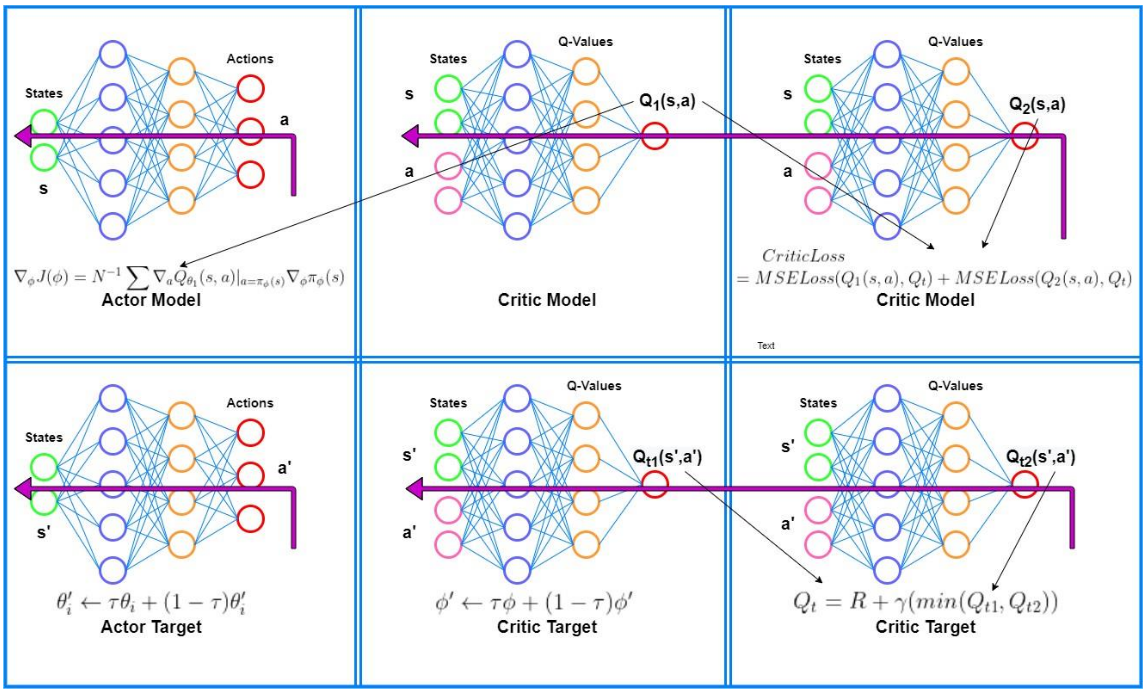

States
s
Actions
a
$\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s,a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
Actor Model
States
s
a
Q-Values
$Q_1(s,a)$
Critic Model
States
s
a
Q-Values
$Q_2(s,a)$
$CriticLoss = MSELoss(Q_1(s,a), Q_t) + MSELoss(Q_2(s,a), Q_t)$
Critic Model
Text
States
s'
Actions
a'
$\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$
Actor Target
States
s'
a'
Q-Values
$Q_{t1}(s',a')$
$\phi' \leftarrow \tau\phi + (1-\tau)\phi'$
Critic Target
States
s'
a'
Q-Values
$Q_{t2}(s',a')$
$Q_t = R + \gamma(min(Q_{t1}, Q_{t2}))$
Critic Target

Figure 3.2: Implementation of TD3

In this above diagram we can visualize the process from Step 1 through 15. The purple arrows indicate backpropagation.

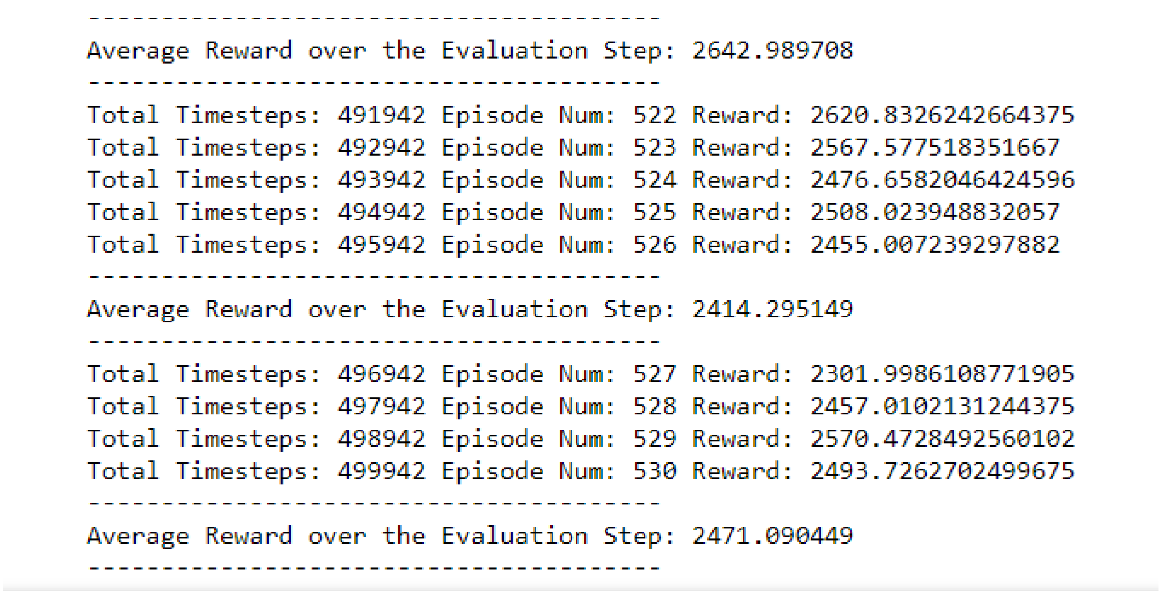
3.2 Training

For training we have run 500,000 timesteps. We start by checking if an episode is done. Because the TD3 is an [5] off-policy model, we train the policy on previous data and transitions after each episode. The previous episode's policy is then evaluated using the evaluate policy() function, which provides the average score acquired across 10 episodes. The evaluate policy() function's outcome is subsequently added to the evaluations list. The pybullet environment and episode settings are reset for a new episode at the end of the assessment process.

We run 10,000 timesteps of random actions to fill the replay buffer memory before starting a new episode. We add a small bit of noise to the generated action values at the conclusion of each timestep to provide a bias toward exploration.

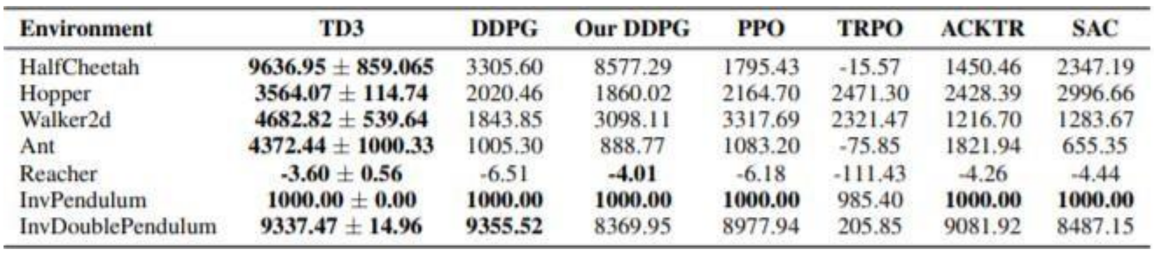
3.3 Inference

We achieved a maximum average return of 2493.73 after training the agent in the "Antv-2" environment for 500,000 timesteps, which is in line with the maximum average score achieved in the publication for 1,000,000 timesteps, i.e. 4372.44, which is roughly double our score for twice the timesteps.



```
---------------------------------------
Average Reward over the Evaluation Step: 2642.989708
---------------------------------------
Total Timesteps: 491942 Episode Num: 522 Reward: 2620.8326242664375
Total Timesteps: 492942 Episode Num: 523 Reward: 2567.577518351667
Total Timesteps: 493942 Episode Num: 524 Reward: 2476.6582046424596
Total Timesteps: 494942 Episode Num: 525 Reward: 2508.023948832057
Total Timesteps: 495942 Episode Num: 526 Reward: 2455.007239297882
---------------------------------------
Average Reward over the Evaluation Step: 2414.295149
---------------------------------------
Total Timesteps: 496942 Episode Num: 527 Reward: 2301.9986108771905
Total Timesteps: 497942 Episode Num: 528 Reward: 2457.0102131244375
Total Timesteps: 498942 Episode Num: 529 Reward: 2570.4728492560102
Total Timesteps: 499942 Episode Num: 530 Reward: 2493.7262702499675
---------------------------------------
Average Reward over the Evaluation Step: 2471.090449
---------------------------------------
```

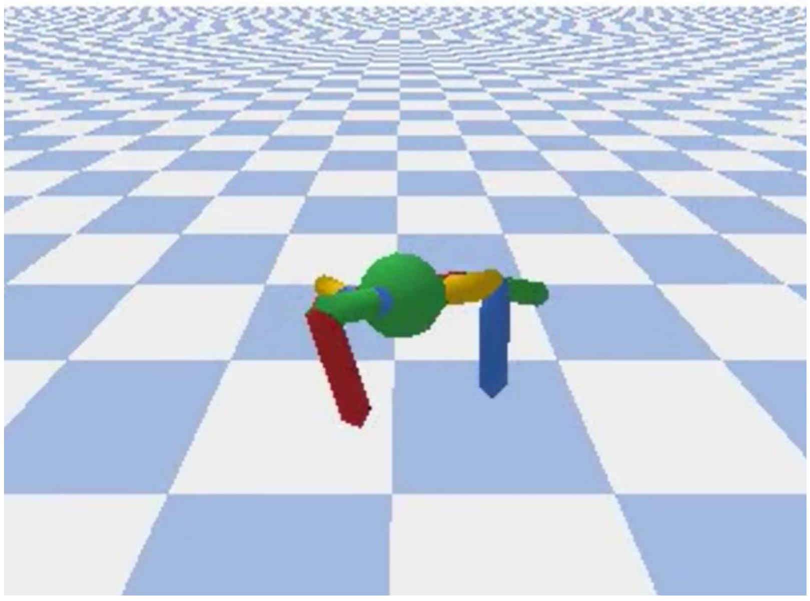
Figure 3.3: Reward Inference



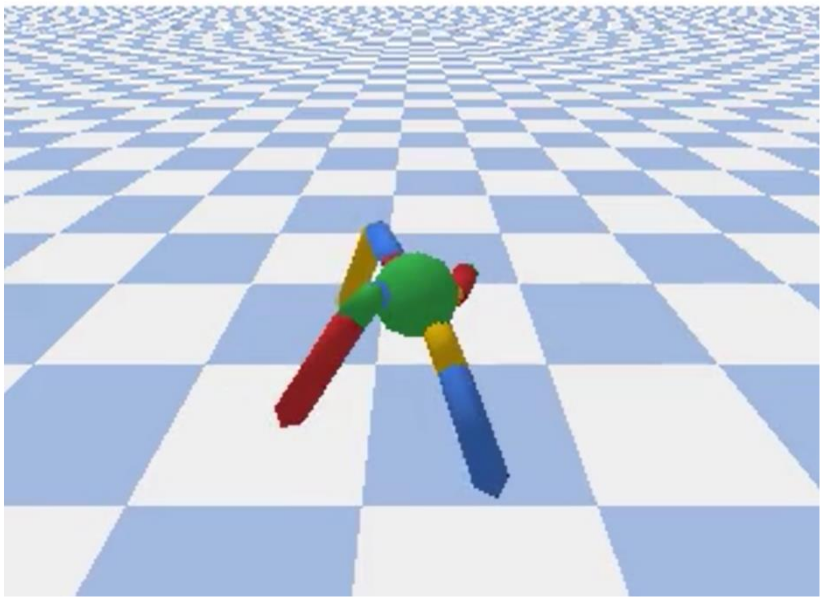
Environment	TD3	DDPG	Our DDPG	PPO	TRPO	ACKTR	SAC
HalfCheetah	**9636.95 ± 859.065**	3305.60	8577.29	1795.43	-15.57	1450.46	2347.19
Hopper	**3564.07 ± 114.74**	2020.46	1860.02	2164.70	2471.30	2428.39	2996.66
Walker2d	**4682.82 ± 539.64**	1843.85	3098.11	3317.69	2321.47	1216.70	1283.67
Ant	**4372.44 ± 1000.33**	1005.30	888.77	1083.20	-75.85	1821.94	655.35
Reacher	**-3.60 ± 0.56**	-6.51	**-4.01**	-6.18	-111.43	-4.26	-4.44
InvPendulum	**1000.00 ± 0.00**	**1000.00**	**1000.00**	**1000.00**	985.40	**1000.00**	**1000.00**
InvDoublePendulum	**9337.47 ± 14.96**	**9355.52**	8369.95	8977.94	205.85	9081.92	8487.15

Table 3.1: Max Average Return

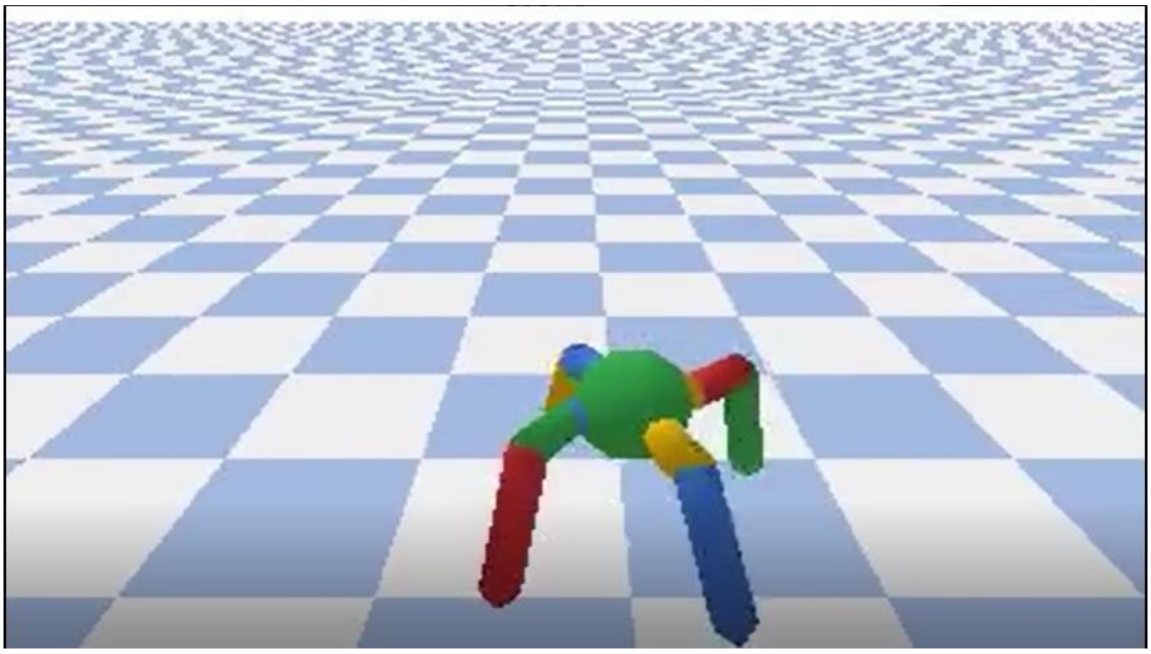
Below is the snapshot of walker after execution of 10,000 timesteps:



Below is the snapshot of walker after 50,000 timesteps:



Based on the training results the agent was able to successfully navigate the environment. Below is a snapshot of walker after 5,00,000 timesteps:



.

Figure 3.4: PyBullet Simulation of Ant-v2

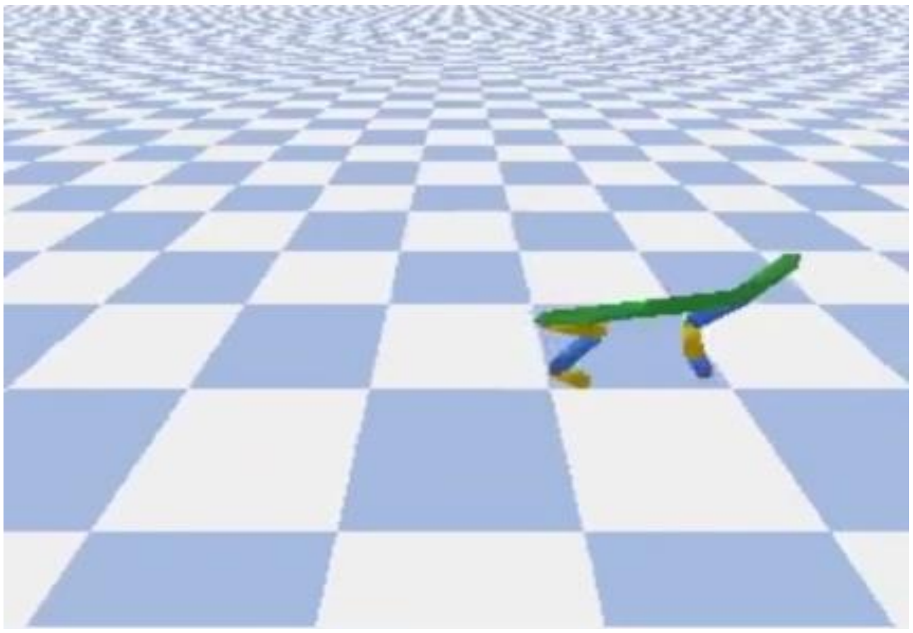


Figure 3.5: PyBullet Simulation of Half-Cheetah

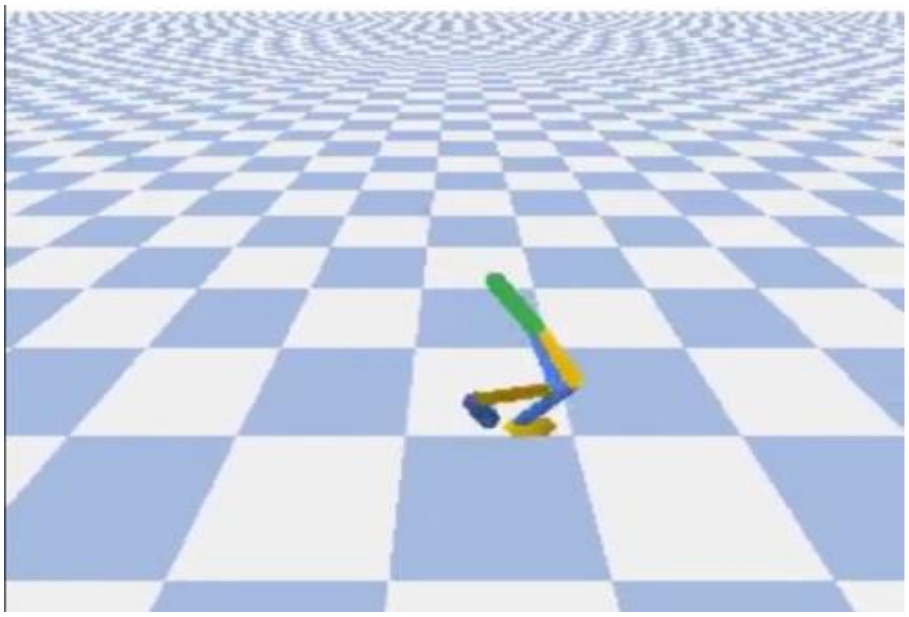


Figure 3.6: PyBullet Simulation of Half-Humanoid

The simulation above is rather simplistic and does not reflect a real-life situation.

3.4 Experimental Result

Below is the graph depicting the average reward gained so far after finishing the training with a particular number of episodes:

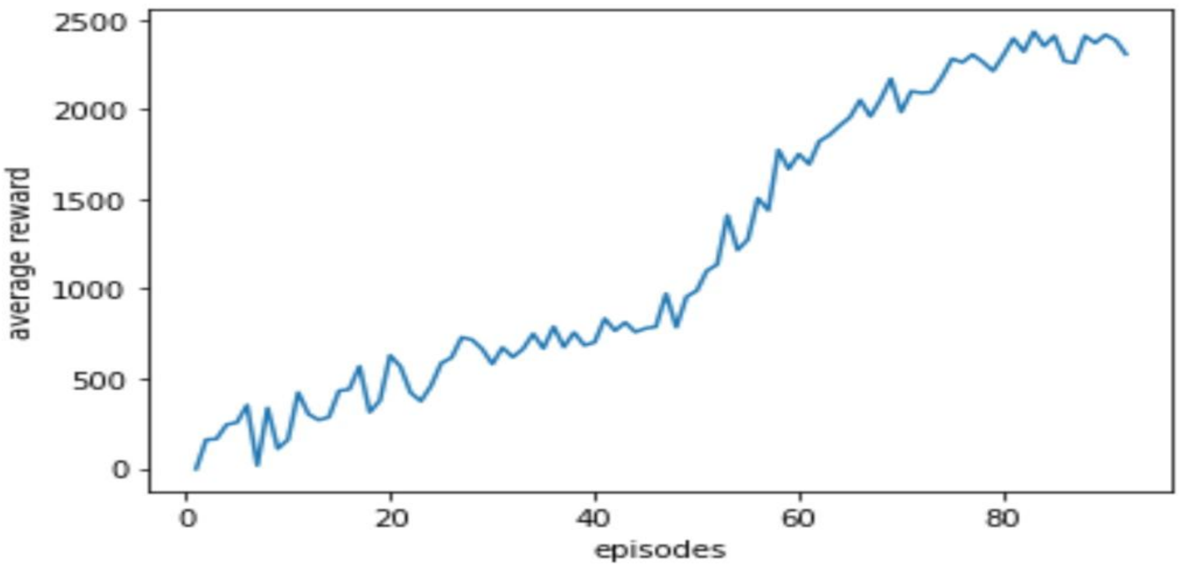


Fig 3.7

Below is the graph of reward achieved after completing the training with a particular number of episodes:

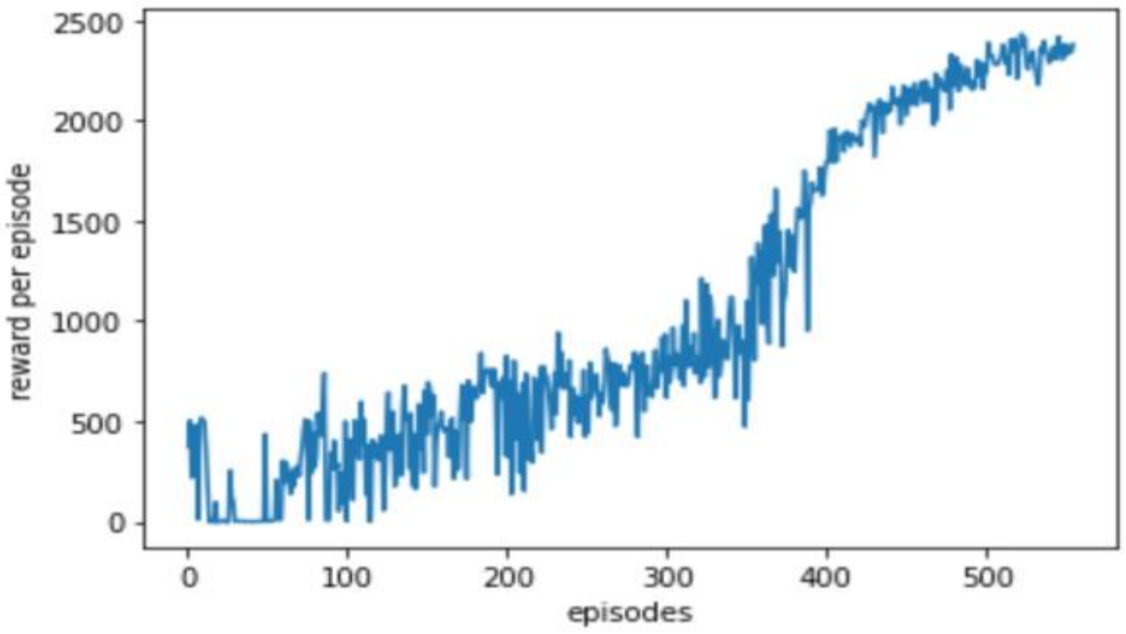


Fig: 3.8

Chapter 4
Conclusions

4.1 Conclusions

With the implementation in Section 3.1 we can conclude that we have fulfilled our objective, that is, we have successfully trained our walker in Ant-v1, Half-cheetah-v1, and Half-humanoid-v1 environment. Based on the training results the agent was able to successfully navigate the environment.

The agent has no specific objectives in mind and is simply walking towards an off-screen destination. However, we confirmed the usefulness of the TD3 algorithm in coordinating the motion of a Quadruped Walker in a continuous action-space with this implementation.

The approximate value of the average reward received after training the walker is equivalent to the return value mentioned in the publication.

The walker can be programmed with appropriate limitations to permit realistic mobility in our implementation. After that, it's only a matter of capturing data in the environment using

RayCasts (which may be used in place of LiDAR in real-life) and setting up the reward-penalty systems. We've avoided the requirement for imitation learning and all the costs and time that entails by completing this initial setup.
This new paradigm also tackles the issue of training other PolyPod walkers, such as adding limbs to a five- or six-legged walker digitally and then teaching the resulting agent to accomplish the desired goals.

4.2 Future Work

We attempted to apply the state of the art in Reinforcement Learning to the problem of smart locomotion for a Quadruped Walker during the course of this research. The field's reach is constantly expanding, with many experts speculating on how this technology could be used to create smart construction workers or assistance, such as Boston Dynamics' Spot bot, which can lift large loads and negotiate all terrains. While Spot mostly relies on hard-coded information to traverse its environment, the thought of a legion of autonomous robots might spur research into smart nanobot design, drone delivery, or autonomous driving in a vast city or community of constantly interacting vehicles.
We intend to build on this notion of swarm learning in the future by adopting more sophisticated environments that need numerous agents to work together to achieve a task.

References

[1] Fujimoto, S., van Hoof, H., and Meger, D. Addressing function approximation error in actor-critic methods. arXiv preprint arXiv:1802.09477, 2018

[2] Juliani, Arthur & Berges, Vincent-Pierre & Vckay, Esh & Gao, Yuan & Henry, Hunter & Mattar, Marwan & Lange, Danny. (2018). Unity: A General Platform for Intelligent Agents.

[3] Sutton, R.S.,Barto, A.G. , *Reinforcement Learning : An Introduction*, The MIT Press, 2018

[4] Schulman, John & Wolski, Filip & Dhariwal, Prafulla & Radford, Alec & Klimov, Oleg. (2017). Proximal Policy Optimization Algorithms.

[5] Luo, Ying-Sheng & Soeseno, Jonathan & Chen, Trista & Chen, Wei-Chao. (2020). CARL: Controllable Agent with Reinforcement Learning for Quadruped Locomotion.

[6] Mao, Hongzi & Alizadeh, Mohammad & Menache, Ishai & Kandula, Srikanth. (2016). Resource Management with Deep Reinforcement Learning. 50-56. 10.1145/3005745.3005750.

[7] Karpathy A. [Blog], “Deep Reinforcement Learning: Pong from Pixels”, May 31, 2016.

[8] Schulman, John & Moritz, Philipp & Levine, Sergey & Jordan, Michael & Abbeel, Pieter. (2015). High-Dimensional Continuous Control Using Generalized Advantage Estimation.