ME5406: Deep Learning for Robotics

Part II Report

Implementation of Single and Multi-Agent Deep Reinforcement Learning Algorithms for a Walking Spider Robot

---

Student Name:
Chong Yu Quan

Student Number:
A0136286Y

Student Email:
chong.yuquan@u.nus.edu

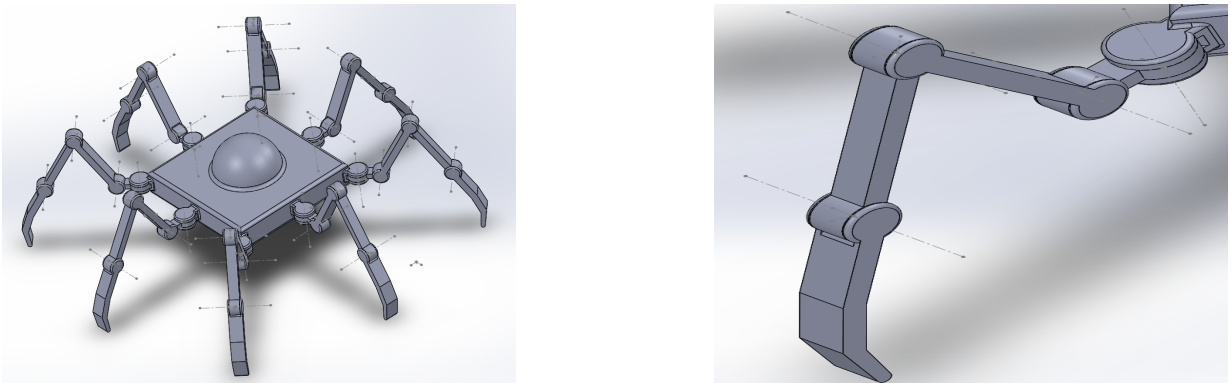
November 18, 2020

1 Introduction

The key inspiration for the project are the spider-like robots in the movie, Minority Report, that are highly mobile and capable of locomotion in complex environments. In reality, locomotion can be addressed with conventional algorithms based on the fundamental mechanical principles of actuating the joints in the legs of the robot. Such an algorithm would involve inverse kinematics calculations coupled with hard coded logic based on sensory data inputs (e.g. joint angles) for locomotion. However, hard coded logic algorithms are: 1) Tedious to implement; 2) Difficult to optimise; 3) Not scalable towards increasing number of joints with increasingly complex environments and tasks. On the other hand, deep reinforcement learning techniques are not hard-coded, optimises towards a task and scales wells to complex environments and tasks. Hence, the implementation of deep reinforcement learning techniques to "teach" a spider-like robot to locomote is elaborated in this report. For the purposes of this project, a similar spider-like robot will be designed and trained to locomote in a simple virtual environment without obstructions.

2 Problem Statement

2.0.1 SpiderBot



(a) SpiderBot

(b) Leg of SpiderBot

Figure 1: Design of SpiderBot

Figure 1 above highlights the spider-like robot (SpiderBot) designed for this project using a solid modelling computer-aided design (CAD) software, SolidWorks. SpiderBot has of a main body, comprising of a cuboid with a square cross-section and dome attached at the top, that serves as a housing for the critical components of a robot in reality (e.g. battery, sensors). On the main body, eight legs, equally spaced apart, are attached on the perimeter of body as show in Figure 1a. Each leg are comprises of four component connected through revolute joints constraint about a specific axis of rotation. The first joint connecting to the main body revolves about the axis perpendicular to the ground plane while the remaining joints revolve about an axis parallel to the ground plane as shown in Figure 1b.

2.0.2 PyBullet Gym

The virtual environment for which SpiderBot will be learning to locomote would be PyBullet, a real-time collision detection and multi-physics simulation for virtual reality, games, visual effects, robotics, machine learning etc. In the environment, a simple plane with no obstacles is imported as the ground plane. The SpiderBot is exported as the Unified Robot Description Format (URDF) from SolidWorks and loaded on the origin of the environment. In PyBullet, the simulated SpiderBot as described in the URDF file has a base, i.e. the main body of the SpiderBot, and optional links, i.e. components of the SpiderBot's legs, connected by joints. Each joint connects one parent link to a child link, following the convention where the joint frames are expressed relative to the parents center of mass inertial frame, which is aligned with the principle axis of inertia. Given that the joints are revolute joints, upper and lower joint limits of $\pm 60°$ are deployed to ensure that the joint positions remains within a reasonable fixed range. With regards to the dynamics of the environment, gravity is set to $9.81\, ms^{-2}$. Lateral, rolling and joint friction are set to default coefficients in PyBullet and there are no joint friction and joint damping constraints for SpiderBot. The specified task in training the SpiderBot to locomote is simply for it to reach a specific target location on the ground plane along the x-axis.

In PyBullet, state observations are continuous in nature and can be obtained by calling specific functions querying on the state of the every component in the SpiderBot. For the base of the SpiderBot, its state is defined by the following: 1) Position is represented by the Cartesian world coordinates of its centre of mass, (x_m, y_m, z_m), 2) Orientation is represented using a quaternion, (x, y, z, w), 3) Linear velocity in world coordinates is represented by a 3-tuple, (v_x, v_y, v_z) and 4) Angular velocity in world coordinates is represented by a 3-tuple, (w_x, w_y, w_z). The state of the links are represented similarly, with the only difference being that its position and orientation are localised measurements relative to its joint frame rather than the world frame. This is due to fact that it is simpler to derive features with regards to the SpiderBot's

posture using localised measurements rather than absolute world coordinates. Lastly, the state of the revolute joints can be defined from its angular position and angular velocity about the axis it is rotating about.

All the joints of SpiderBot are actuated by motors in PyBullet, for which position velocity control motors are utilised for the purposes of this project. The motors allows SpiderBot to actuate in the PyBullet environment by setting target positions and/or velocities for the joints and running the simulation over a specified discrete time step, where the default unit time step is $\frac{1}{240}$ second. The physics engine in PyBullet resolves the actuation by minimising the error between the target position and the current position as well as between the target velocity and the current velocity. For the purposes of this project, only target velocities are set for the joints and are the "actions" by the agent.

A few terminating conditions were coded in the gym as well. After every simulation over a specified time step, the gym would firstly check if the SpiderBot is "flipped". The SpiderBot is considered to be flipped when any other links or the base touches the ground plane other than the end links of each leg. Secondly, the gym checks if the centre of mass of the base of the SpiderBot have crossed a specific target location along the x-axis, from which it achieves its stipulated task. Thirdly, the gym checks if the SpiderBot strays out of a specified range in the ground plane. Lastly, the gym checks if the length of the episode is longer than a specified timeout in real time. All stated checks raises different terminal flags for the purposes of the reward function and resets the PyBullet environment for the next episode.

After every simulation for a specified time step, the reward function takes in the SpiderBot's base velocity and position. It rewards the Spider for its x-position relative to target location along the x-axis and its velocity, v_x, along the x-axis relative to an ideal velocity. The relative position and velocity ensures that the rewards are always negative to prevent the scenario where the SpiderBot receives positive rewards without locomotion. Negative rewards are given for the SpiderBot's absolute position and velocity, v_y, along the y-axis to punish straying off course. The reward function also takes in the terminal flags and accords necessary rewards for the various flags (e.g. negative reward of -200 for moving out of range) should the episode terminate.

3 RL Cast

In this project, an agent is defined as an entity capable of generating a policy or policies to select actions for a specific entity in the gym, such as the SpiderBot (i.e 32 joints) or a single leg of the SpiderBot (i.e 4 joints from the leg). The policy of the algorithm is defined to be centralised if there is a single policy output defining the entire policy of the SpiderBot and decentralised if there are multiple policy outputs. In this project, a total of five deep learning algorithms are explored, of which three are single-agent algorithms and two are multi-agent algorithms. The single agent algorithms are as follows: 1) Advantage Actor-Critic Multi-Action (A2CMA), 2) Advantage Actor-Critic Single-Action (A2CSA) and 3) Deep Deterministic Policy Gradients (DDPG). The multi-agent algorithms are as follows: 1) Multi-Agent Advantage Actor-Critic (MAA2C) and 2) Multi-Agent Dueling Double Deep Q-Learning (MAD3QN). Table 1 below summarises the key characteristics of the algorithms to be further elaborated in the following sections.

Table 1: Characteristics of algorithms implemented

Algorithm	Agent (Actor)	Policy	Learning Network	Actions per Time-Step	Action Space	State Space
MAD3QN	Multiple (Decentralised)	Decentralised	Separate	Multiple	Discrete	Continuous
MAA2C	Multiple (Decentralised)	Decentralised	Separate	Multiple	Discrete	Continuous
A2CMA	Single (Centralised)	Decentralised	Hybrid	Multiple	Discrete	Continuous
A2CSA	Single (Centralised)	Centralised	Hybrid	Single	Discrete	Continuous
DDPG	Single (Centralised)	Centralised	Separate	Multiple	Continuous	Continuous

3.1 Multi-Agent Advantage Actor-Critic

Given the high variance of Monte-Carlo (MC) policy gradient (PG) algorithms, Actor-Critic (AC) PG algorithms are introduced. ACPG algorithms maintains two sets of learned parameters as follows: 1) Critic: Updates ω parameters to estimate a value function (e.g. $Q_{\pi_\theta}(s,a) \approx Q_\omega(s,a)$), 2) Actor: Updates θ parameters to estimate policy $\pi_\theta(s,a)$. ACPG approaches follow an approximate policy gradient as shown in equation 1 below.

$$\begin{aligned} -\nabla_\theta J(\theta) &\approx \mathbb{E}[\nabla_\theta \log \pi_\theta(s,a) * Q_\omega(s,a)] \\ \Delta\theta &= \alpha * \nabla_\theta \log \pi_\theta(s,a) * Q_\omega(s,a)) \end{aligned} \tag{1}$$

However, while ACPG may reduce PG variance which may slow down learning, it introduces biases that may cause convergence to the wrong solution. However, if the value function is chosen appropriately, the introduction of bias can be avoided while reducing PG variance. This can achieved by subtracting the state-action value $Q_{\pi_\theta}(s,a)$ with the state value, $V_{\pi_\theta}(s)$ to obtain the advantage, $A_{\pi_\theta}(s,a)$ as shown in equation 2 below, where α is a learning rate.

$$A_{\pi_\theta}(s,a) = Q_{\pi_\theta}(s,a) - V_{\pi_\theta}(s) \tag{2}$$

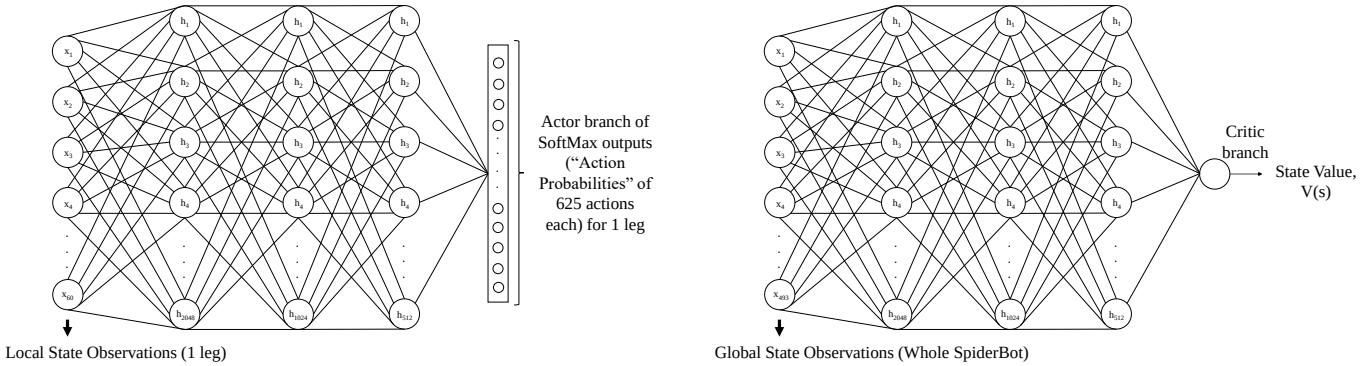
This subtraction reduces PG variance without changing the expectation value (not elaborated). Instead of approximating $Q_{\pi_\theta}(s,a)$ and $V_{\pi_\theta}(s)$ with two sets of critic parameters, it is observed that for the true value function $V_{\pi_\theta}(s)$, the

temporal difference (TD) error, δ_{π_θ}, is an unbiased estimate of $A_{\pi_\theta}(s,a)$ as shown in equation 3 below, where γ is the discount rate.

$$\begin{aligned} \delta_{\pi_\theta} &= r + \gamma V_{\pi_\theta}(s') - V_{\pi_\theta}(s) \\ \mathbb{E}[\delta_{\pi_\theta}|s,\, a] &= \mathbb{E}[r + \gamma V_{\pi_\theta}(s')|s,\, a] - V_{\pi_\theta}(s) = Q_{\pi_\theta}(s,a) - V_{\pi_\theta}(s) = A_{\pi_\theta}(s,a) \end{aligned} \tag{3}$$

Hence, the approximate TD error, δ_ω, can be used to the compute the policy gradient, which only require one set of critic parameters ω, shown in equation 4 below.

$$\begin{aligned} -\nabla_\theta J(\theta) &\approx \mathbb{E}[\nabla_\theta \log \pi_\theta(s,a) * \delta_\omega] \qquad (actor) \\ \delta_\omega &= r + \gamma V_\omega(s') - V_\omega(s) \qquad (critic) \end{aligned} \tag{4}$$



(a) MAA2C actor architecture

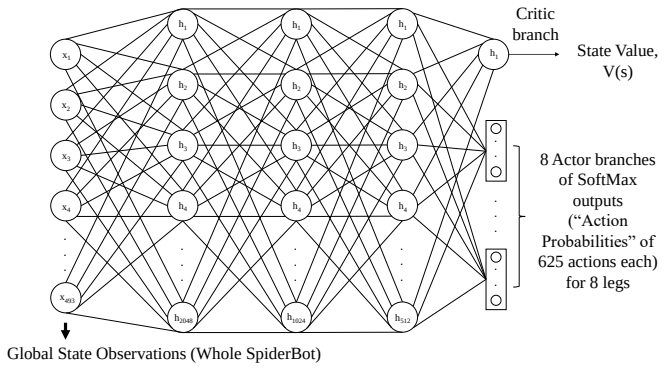
(b) MAA2C critic architecture

Figure 2: Separate Advantage Actor-Critic

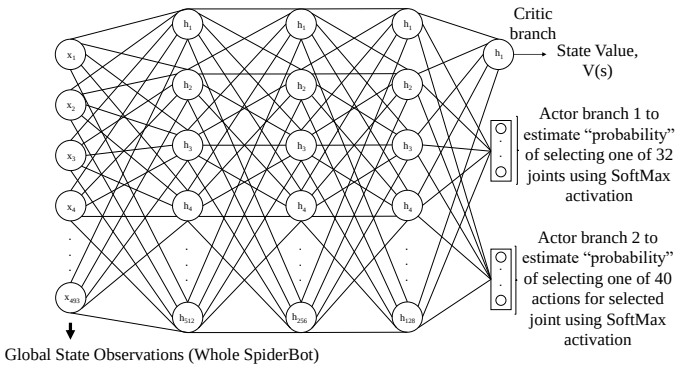
Figure 2 above highlights the fully-connected deep neural network (FCDNN) architecture of the actor and critic models. In MAA2C, there are a total of nine FCDNNs, with one critic FCDNN and eight actors FCDNNs representing each leg of the SpiderBot, hence multiple agent with separate learning networks. All FCDNNs have three hidden layers with 512, 256, 128 hidden units for each respective layer for the critic and 2048, 1024, 512 hidden units for each respective layer of the actor. The critic FCDNN takes in global state observations (493 features) from the all joints, links and the base and gives a single output of the state value, $V_\omega(s)$ as shown in Figure 2b. The critic losses for which the FCDNN trains to minimise is the squared of the TD error, δ_ω, shown in equation 4. On the other hand, each actor takes in a local state observations (60 features) from the base as well as all the joints, links of the specified leg. Naturally, the assumption is that each leg does not have access to the information of the other legs, but share the same task of locomotion. Before elaborating on the output of each actor FCDNN, it must first be stated that the action space for MAA2C is defined to be a discrete action space. Hence, each joint have a specified finite list of velocities ([-10, -5, 0, 5, 10]) stored in the PyBullet gym class for which it can select as target velocities for actuation in PyBullet within a specified time step. Given that each leg comprises of four joints, a list of permutations of target velocities, where each permutation is a list of 4 possible target velocities indexed for each joint for a specific leg (length = $5^4 = 625$), is generated in the PyBullet gym class. Hence, the actor FCDNN gives softmax outputs for each index in the permutations list as "actions" as shown in Figure 2a, for which the index of the maximum softmax output is the corresponding of index of a specific permutation of target velocities to be applied to the corresponding joints of the specific leg. This process is repeated for each actor FCDNN for each leg, hence having a decentralised policy with multiple actions per time step. The actor losses are the negative of the expected value of the logarithmic probability density function for the "actions" multiplied by the TD error, δ_ω, from the critic FCDNN as shown in equation 4.

3.2 Advantage Actor-Critic Single Action / Multi-Action

Like MAA2C, A2CMA and A2CSA are based on the same AC principles elaborated in the previous section. However, instead of separate actor and critic FCDNNs, the actor and critic are combined to a single hybrid AC FCDNN as shown in Figure 3 below, hence considered single agent algorithms. Both A2CMA and A2CSA take in global state observations (493 features) from the all joints, links and the base in its single FCDNN. A2CMA has hidden units of 2048, 1024, 512 while A2CSA has hidden units of 512, 256, 128 for each respective hidden layer. Both models output the state value, $V_\omega(s)$, for the critic aspect, but have different outputs for the actor aspect. For A2CMA, the output layer for the actor aspect has eight actor branches representing eight different policies per leg, hence a decentralised policy with multiple actions per time step. Each actor branch gives Softmax outputs to select "actions" from the list of permutations of target velocities in the exact same manner as elaborated in the previous section for MAA2C. On the other hand, the output layer for A2CSA has two actor branches. The first branch gives Softmax outputs for each joint in a list of joints in the SpiderBot and the second branch gives Softmax outputs for a specified list of target velocities. The policy is then the selection of the a specific joint followed by a selection of a specific target velocity for the selected joint. Hence, the policy is a centralised policy with only a single action per time step. The losses for the FCDNN for both A2CMA and A2CSA



(a) A2CMA architecture

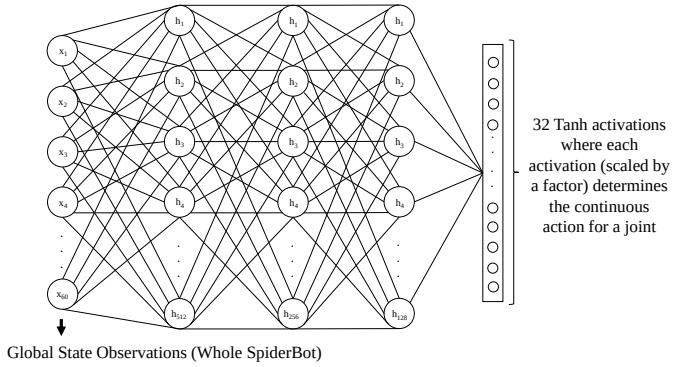


(b) A2CSA architecture

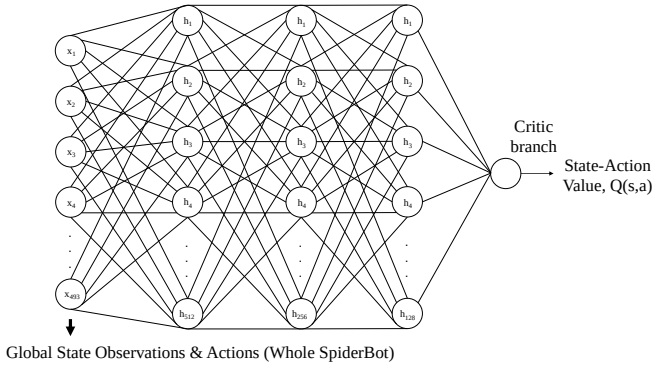
Figure 3: Hybrid Advantage Actor-Critic neural network

is summation of the critic losses and actor losses according to equation 4 computed in the same manner as elaborated for MAA2C.

3.3 Deep Deterministic Policy Gradients



(a) DDPG actor architecture



(b) DDPG critic architecture

Figure 4: DDPG architecture

DDPG consists of four FCDNNs, namely the actor FCDNN, critic FCDNN, target actor FCDNN and the target critic FCDNN, and a replay buffer. The replay buffer serves as a form of memory that stores the experiences of the agent during training. It stores the state (s), action (a), new state (s') after taking action, a, from state, s, reward from the reward function (r) as well as the terminal flag. The memory space has a finite space (e.g. 1000000) and is based on a first in first out principle in replacing experiences stored in memory. In training the FCDNNs, a specified batch (e.g. 256) of the experiences stored in the replay buffer will be randomly sampled. The actor FCDNN takes in global state observations (493 features) from the all joints, links and the base, with three hidden layers of hidden units 512, 256, 128 respectively as shown in Figure 4a above. The global state observations are sampled from the replay buffer during training and from the PyBullet environment during experience gathering. The global state observations naturally assumes that state information of the other legs are readily accessible to each leg. However, unlike previous algorithms in the report, DDPG works with a continuous action space instead of a discrete action space in this project. Hence, the outputs of the actor FCDNN are 32 outputs with Tanh activation functions for each joint in the SpiderBot. Given that the Tanh activation function is bounded between ± 1, the 32 Tanh outputs are scaled by a factor such that the resultant outputs, which is the respective target velocity for the 32 joints of the SpiderBot, are bounded between a specified range (e.g. ± 10). Therefore, it is apparent that DPPG follows a centralised policy with multiple actions per specified time step. The critic FCDNN follows the same architecture as the critic FCDNN for MAA2C, with three hidden layers of hidden units 512. 256, 128 respectively. It takes in global state observations (493 features) from the all joints, links and the base concatenated with the corresponding actions taken by each joint of the SpiderBot (32 actions) sampled from the replay buffer. It gives a single output of the state-action value, $Q_\omega(s, a)$ as shown in Figure 4b above. The target actor FCDNN and target critic FCDNN follows an identical architecture with the actor FCDNN and critic FCDNN respectively. The two target FCDNNs are not utilised during experience gathering but for training of the actor and critic FCDNNs. The weights of each hidden unit in the target FCDNNs with are updated with based on a parameter, τ, with respect to its corresponding FCDNN counterpart as shown in equation 5 below.

$$\theta'_{target\ actor} = \theta_{actor} * \tau + \theta_{target\ actor} * (1 - \tau)$$
$$\omega'_{target\ critic} = \omega_{critic} * \tau + \omega_{target\ critic} * (1 - \tau) \tag{5}$$

On initialisation of the FCDNNs, the target FCDNNs are a hardcopy of their original counterparts, i.e $\tau = 1$. After every subsequent training of the actor and critic FCDNNs, the target FCDNNs undergoes a softcopy update (e.g. $\tau = 0.005$). The training losses of the actor and critic FCDNNs are stated in equation 6 below.

$$-\nabla_\theta J(\theta) \approx Q_{\omega_{critic}}(s, a_{actor}) \qquad (actor)$$
$$\delta_{\omega_{critic}} = r + \gamma Q_{\omega_{target\ critic}}(s', a'_{target\ actor}) - Q_{\omega_{critic}}(s, a) \qquad (critic) \tag{6}$$

For the critic losses with experiences from the replay buffer, the action of the target actor FCDNN from s', $a'_{target\ actor}$, is first obtained. From s' and $a'_{target\ actor}$, the state-action value, $Q_{\omega_{target\ critic}}(s', a'_{target\ actor})$, can be obtained from the target critic FCDNN, from which added to r, gives the TD target. The critic loss is hence the mean squared error, δ^2_ω, between the TD target and the state-action value, $Q_{\omega_{critic}}(s, a)$, from the critic FCDNN. For the actor losses, the action selected with an updated policy from the actor FCDNN, a_{actor} is obtained. The actor losses are then the negative of the state-action value, $Q_{\omega_{critic}}(s, a_{actor})$, using s and a_{actor} from the critic FCDNN, which is coupled with the actor FCDNN by virtue of a_{actor}.

3.4 Multi-Agent Dueling Double Deep Q-Learning

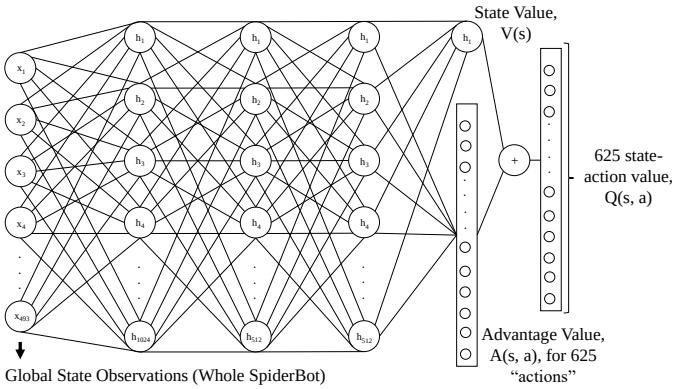


Figure 5: MAD3QN architecture

MAD3QN consists of 8 online FCDNNs, representing each leg of the SpiderBot (multiple separate agents), 8 target FCDNNs for each online FCDNNs and a replay buffer identical to that implemented in DDPG. Each online FCDNN takes in global state observations (493 features) from the all joints, links and the base, with three hidden layers of hidden units 1024, 512, 512 respectively, and outputs 625 state-action value, $Q(s, a)$ as shown in Figure 5 above. The generated $Q(s, a)$ stems from the sum of state value, $V(s)$, and the advantage value, $A(s, a)$ from equation 2. The 625 $Q(s, a)$ corresponds to the value of the 625 "actions" from a given state, where the "actions" are the index of the list of permutations of target velocities ([-10, -5, 0, 5, 10]) to be applied to the four joints of a specific leg as elaborated in MAA2C and A2CMA. With a decentralised policy and working in a discrete action space, the selection of the action is based on the "action" that gives the largest $Q(s, a)$. Eight of such "actions" is hence selected from each FCDNN and applied to all 32 joints of the SpiderBot, hence multiple actions per specified time step. As with DDPG, the target FCDNNs follow an identical architecture as the online FCDNNs and are utilised for training rather than during experience gathering. The weights of each hidden unit in the target FCDNNs with are updated with based on a parameter, τ, with respect to its corresponding FCDNN counterpart as shown in equation 7 below (equivalent to equation 5).

$$\theta'_{target} = \theta_{online} * \tau + \theta_{target} * (1 - \tau) \tag{7}$$

Like DDPG, on initialisation of the FCDNNs, the target FCDNNs are a hardcopy of their original counterparts, i.e $\tau = 1$. After every subsequent training of the online FCDNNs, the target FCDNNs undergoes a softcopy update (e.g. $\tau = 0.005$). The training losses for the FCDNNs are shown in equation 8 below.

$$\delta_{\theta_{online}} = r + \gamma Q_{\theta_{target}}(s', \underset{a}{\operatorname{argmax}}\, Q_{\theta_{online}}(s', a)) - Q_{\theta_{online}}(s, a) \tag{8}$$

During training, a batch of experiences is sampled from the replay buffer, where each experience consists of the state (s), action (a), new state (s') after taking action, a, from state, s, reward from the reward function (r) as well as the terminal flag. The maximal actions for the state-action value of the online FCDNN at s' are used to select the state-action value of the target FCDNN at s', for which added to r gives the TD target. The TD error, $\delta_{\theta_{online}}$, for the online FCDNN is the TD target subtracted by the state-action value of the online FCDNN, $Q_{\theta_{online}}(s, a)$, for which squared gives the training losses.

4 Results

4.1 Standardised Test

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ HYPERPARAMETER CONFIG ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#

# set the time length (in seconds) for each time step to split the continuous time space into discrete steps
time_step_size = 120./240
# Set the target location for the SpiderBot to walk to
target_location = 3
# Upper and Lower Joint limits in angle in degrees
upper_angle = 60
lower_angle = -60
# Specify learning rates for actors and critics, for MAD3QN, only the actor learning rate is used
lr_actor = 0.001
lr_critic = 0.001
# Specify the discount rate
discount_rate = 0.9
# number of time_steps to do a hard copy for target networks in MAD3QN and DDPG
# Otherwise enter None for Softcopy
update_target = None
# Tau value for soft copy of online network weights to target networks for MAD3QN and DDPG
tau = 0.005
# Replay memory size and batch size for MAD3QN and DDPG
max_mem_size = 1000000
batch_size = 256
# Continuous action space range in units of rad/s of joints and noise stddev for DDPG.
max_action = 10
min_action = -10
noise = 3
# epsilon value for starting, minimum and decay for MAD3QN (as it is a value approximator)
epsilon = 1
epsilon_decay = 0.0001
epsilon_min = 0.01
```

(a) Hyperparameters configuration

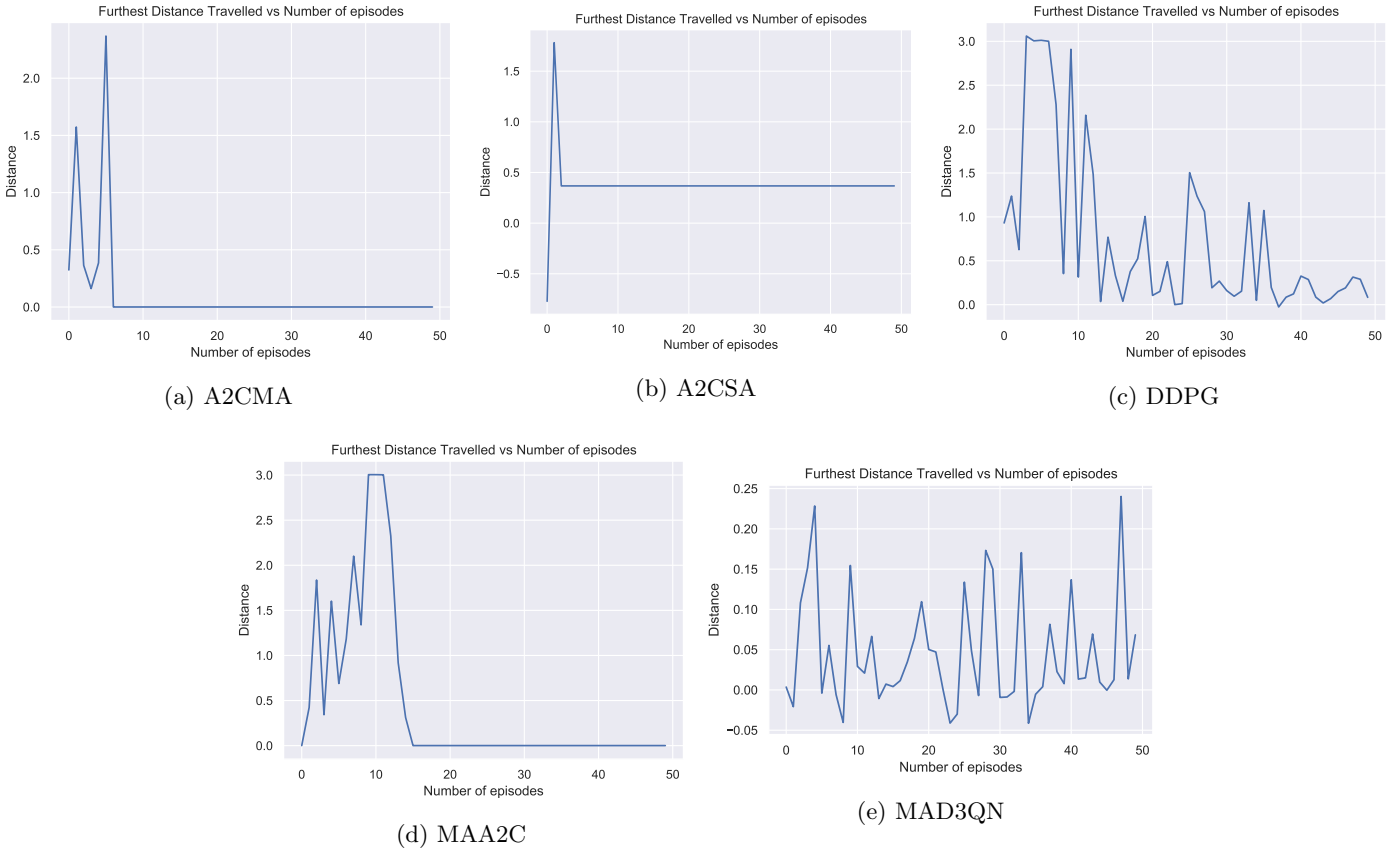
```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ REWARD STRUCTURE CONFIG ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#

# reward for forward velocity in the correct direction
forward_motion_reward = 500
# reward for being close to the target location
forward_distance_reward = 250
# penalty for having sideways velocity
sideways_velocity_punishment = 500
# penalty for moving sideways from axis of walk
sideways_distance_penalty = 250
# penalty applied multiplied to the total number of seconds passed during training
time_step_penalty = 1
# penalty for terminating by flipping
flipped_penalty = 500
# reward for reaching the Target
goal_reward = 500
# penalty for terminating by going out of range (more than 1m sideways and backward from origin)
out_of_range_penalty = 500

############################## END OF CONFIGURATION ################################
```

(b) Reward structure configuration

Figure 6: Configuration for standardized test



(a) A2CMA

(b) A2CSA

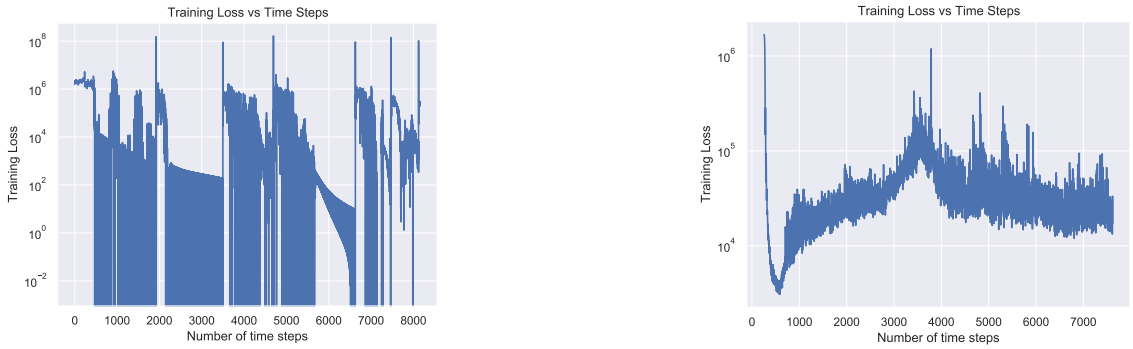
(c) DDPG

(d) MAA2C

(e) MAD3QN

Figure 7: Furthest distance travelled each episode for the algorithms

To estimate the performance of the five algorithms, a standardised test is where the hyperparameters and reward structure are fixed, as shown above in Figure 6 above, is performed to select the best algorithm to focus on. Figure 7 shown above are the results of the test, evaluated based on the furthest distance travelled by the SpiderBot in each episode, from which it can be seen that A2CMA, A2CSA and MAD3QN have performed poorly. Both A2CMA and A2CSA had respectable spikes in furthest distance travelled in early episodes (between 1.5 to 2.5 metres). However, upon reaching the highest furthest distance travelled, both algorithms collapses to a baseline with negligible furthest distance travelled. On the other hand, MAD3QN has an extremely poor furthest distance travelled in general of less than 0.25 metres for all episodes. Only MAA2C and DDPG have shown glimpses of success in the test by reaching the target location several times. Eliminating A2CMA, A2CSA and MAD3QN, the training losses of MAA2C and DDPG are inspected as shown in Figure 8 below. From Figure 8a, it can be observed that training loss of the MAA2C can fluctuates wildly between 10^{-2} and 10^{8} for small differences in time steps with no signs of convergence, highlighting that training is highly unstable. On the other hand, the training losses for DDPG are generally more stable, between 10^{4} and 10^{6} despite no signs of convergence. In addition, the fluctuations of the training losses are also more well constrained, within approximately an order of magnitude. Hence, it can be concluded that the DDPG algorithm is best algorithm ahead.



(a) MAA2C

(b) DDPG

Figure 8: Training losses of MAA2C and DDPG

4.2 Reward Structure Tuning

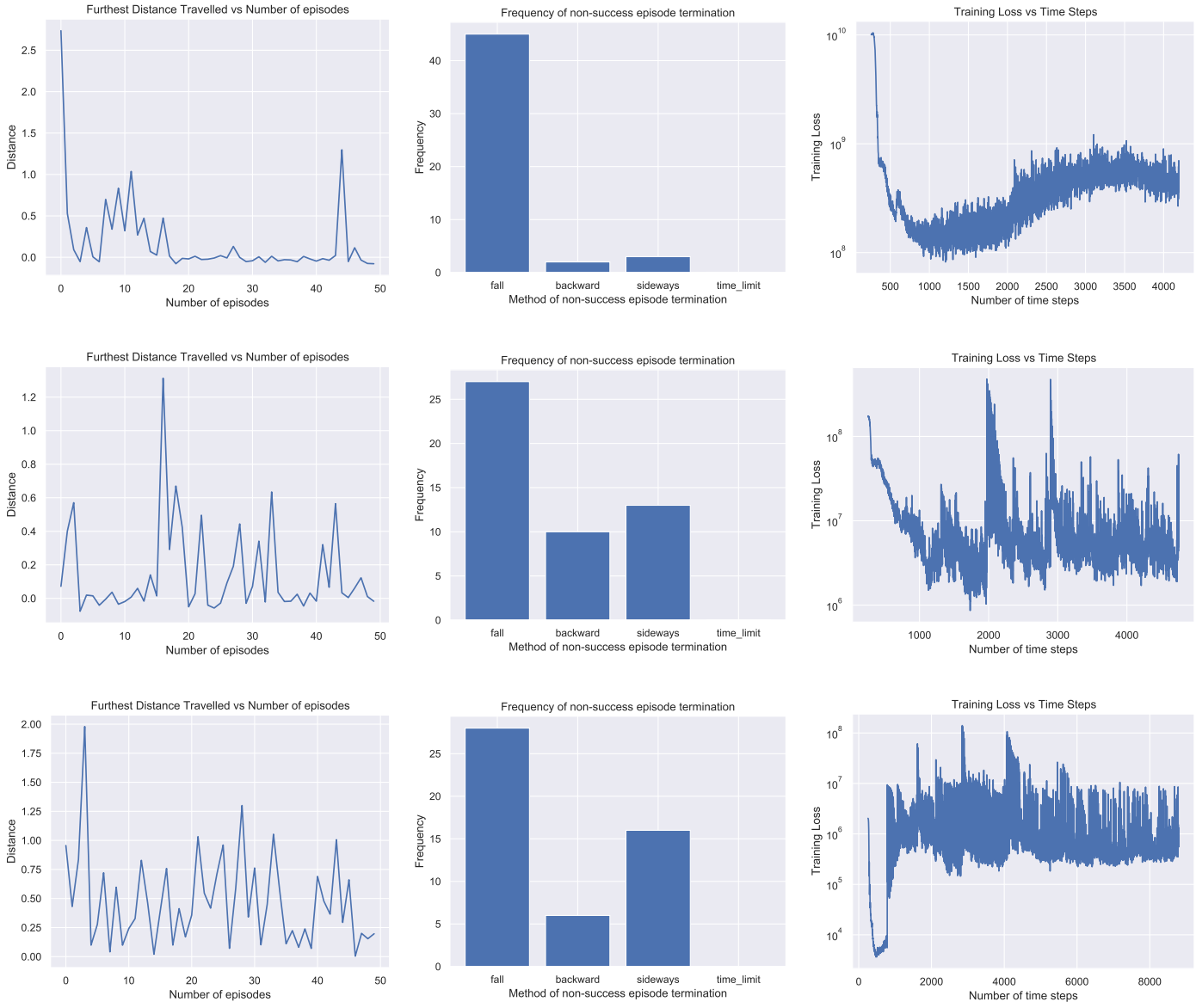
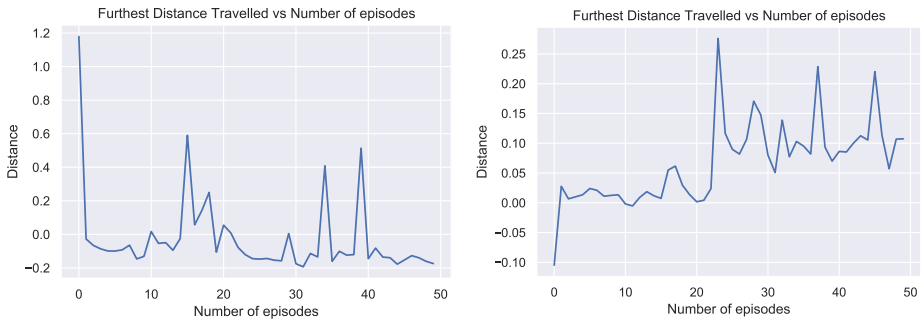


Figure 9: Results for scaling forward, side and terminal constants in the first, second and third row respectively

The reward structure is an important hyperparameter for any deep learning algorithm. In this section, the reward structure from Figure 6b is adjusted while keeping the hyperparameter configuration of Figure 6a to test its effects on performance for DDPG. The adjustments are as follows: 1) Scale rewards for forward velocity and position relative to target location by 100, 2) Scale rewards for sideways velocity and position by 100, 3) Scale reward for terminal flags by 100. From the results shown in Figure 9, it can be observed that the scaling of the forward variables has deteriorated

the performance, in terms of furthest distance travelled, of the SpiderBot from relative to Figure 7c in the standardised test. By observing the simulation from the graphical interface, it can be inferred that by scaling the reward for forward velocity and position significantly in proportion to the other reward variables, the SpiderBot tends to take aggressive, lurching actions that increases its forward velocity and position at the expense of its stability, hence "flipping" often (approximately 45 out of 50 episodes) as corroborated by the terminal flags plot. Given that the training losses do not appear to be converging as well, such a reward structure is non-ideal for training. Similarly, the scaling of the punishment for sideways velocity and position significantly in proportion to the other reward variables has deteriorated performance as well. By observing the simulation, the harsh penalties for straying off the optimal course has made the SpiderBot learn to swivel, which resulted in an inefficient rotational motion that ironically strays off course or leads to the Spiderbot being "flipped" as shown in the terminal flags plot. With the training losses fluctuating wildly with no signs of convergence, the stated reward structure is non-ideal for training as well. Lastly, the scaling of the terminal rewards significantly in proportion to the other reward variables has deteriorated performance as well. No significant trends can be observed from the simulations and with non-converging training losses, the stated reward structure is non-ideal for training as well. Given the results, it is concluded that the excessive scaling of any reward variables generally does not lead to any improvement in performance.

4.3 SpiderBot Iterations



(a) SpiderBot with 6 legs

(b) SpiderBot with 4 legs

Figure 10: Results for SpiderBot with different number of legs

Given the constraints of the report, further hyperparameter tuning are not elaborated. Nevertheless, while tuning the hyperparameters of DDPG, other iterations of the SpiderBot with different number of legs (6 and 4) were also designed and tested using DDPG with the standardised configuration. The motivation behind the implementation is to test if convergence to a solution could be brought about by reducing the number of degree of freedom, i.e. number of joints for the SpiderBot. From the results shown in Figure 7c and 10, it can be concluded that the reduction in number of legs does not appear to bring any significant benefits given a deterioration in performance in terms of furthest distance travelled relative to the original SpiderBot.

4.4 Successful Model

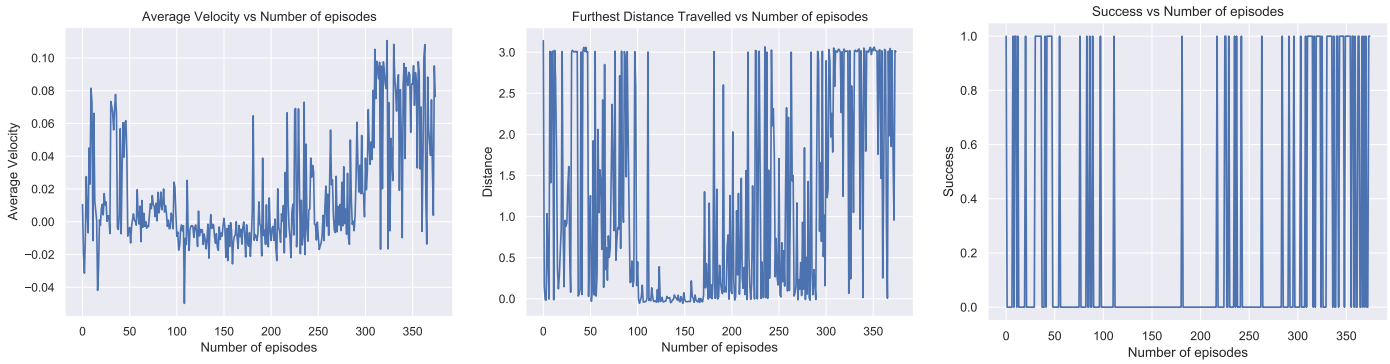


Figure 11: Results from successful DDPG model

With numerous trials, the successful model with the original SpiderBot is generated with the following alterations to the hyperparameters and rewards configuration from Figure 6: 1) lr_actor = 0.00005, 2) lr_critic = 0.0001, 3) batch_size = 512, 4) noise = 1, 5) time_step_penalty = 0.05, 6) goal_reward = 0. It was found that a significantly reduced learning rate for both the actor and critic FCDNN, by approximately a factor of 100 from the standardised configuration, along

with a higher critic learning rate relative to the actor helped improved DDPG's performance in training significantly. A hypothesis would be that the learning rate for the critic should be larger than the actor in order for the critic to lead the actor towards an optimum policy. Batch size is also doubled to broaden variety of experiences sampled from replay buffer. In addition, a lower Gaussian noise variable helped the SpiderBot train more consistently as well as improve the performance of successful saved models during evaluation, where the Gaussian noise is not factored in the selected target velocities generated by the DDPG actor FCDNN. The reduced time step penalty, by a factor of 20 from the standardised configuration, ensured that the rewards based on positions and velocities are not drowned out by the time step penalty when the time step is large during simulations, hence making the training process more stable. Lastly, the goal reward upon reaching the target location is reduced to zero as it was suspected to be contributing to instability in training by giving a disproportionately large reward to the terminating set of experience. This was inferred through numerous trials where the performance (in terms of furthest distance travelled) of the SpiderBot was observed deteriorate generally (e.g. being "flipped" almost immediately) after a successful episode as the FCDNNs learn from that specific terminal set of experience randomly sampled from the replay buffer.

The combination of the alterations is believed to have led to the eventual convergence to a solution by the DDPG algorithm. This is corroborated by the success plot shown in Figure 11, where the sparse successes in earlier episodes became significantly more consistent at approximately episode 300 and above, from which the successful model are saved and extracted from. In addition, the average velocity and furthest distance travelled plots also show corresponding rises at approximately 300 episodes. Upon evaluating the best saved model on the graphical interface, it was observed that the SpiderBot does not locomote in the manner from which one would expect a spider in reality would. While the motions of the SpiderBot remains indescribable, it has shown that is capable of making towards the target location at a reasonable and consistent speed. Upon straying off course, it also exhibits course correction motions to attempt to remain on an optimal straight line course towards the target location. In addition, despite the fact the target location during training was 3 metres, the model was found to be capable of reaching target locations up to 8 metres, which is technically unknown grounds for the models. Hence, it can be inferred that the saved model can be used for transfer learning in training to reach further target locations, from which it should be capable of reaching with minimal training.

5 Discussion

5.1 Advantages

Given that the advantages of the advantage based AC algorithm have already been elaborated in introducing MAA2C, the purported advantages of the subsequent algorithms building on MAA2C will be elaborated. Closely tied to MAA2C is A2CMA, which is essentially the MAA2C algorithm, but with the actor and critic combined into a centralised FCDNN instead of being two separate FCDNNs. The rationale behind the combination was to exploit the commonality between the actor and critic FCDNNs initial few hidden layers in feature extraction from the state observations. Given that the FCDNN of both A2CMA is shallow with 3 hidden layers, the networks are just combined. As a result, there is no need to train nine FCDNNs like in MAA2C for one critic and eight actors per leg. The inherent benefit is naturally a reduction of computational resources and consequentially a potentially earlier convergence to a sufficiently optimal policy. Closely tied to A2CMA is A2CSA, which is in essence A2CMA with a different take of a centralised policy output. The motivation behind the A2CSA is to experiment if taking a single action per specified time step would allow the rewards be most accurately attributed causally to the selected action rather than the 32 different actions per specified time step for A2CMA. Hence, it may potentially give an earlier convergence to a sufficiently optimal policy.

For DDPG and MAD3QN, two additional deep reinforcement learning features were implemented, namely target networks and replay buffer. The motivation for introducing target FCDNNs is to increase the stability of the algorithm during training. This is due to the fact that the essence of deep reinforcement learning is inherently linked to updating a guess with another guess. Consider equation 4 for MAA2C, where the TD error, δ_ω, used to update the weights of the critic FCDNN, ω, is using estimates of state values, $V_\omega(s)$ and $V_\omega(s')$ from the same critic FCDNN. The issue here lies in the fact that the differences between s and s' are of a specific small time step apart, which from the perspective of the FCDNN, that serves as a function approximator, is extremely difficult to differentiate. As a result, by updating ω to better approximate $V_\omega(s)$, the FCDNN inevitable affects the values generated for $V_\omega(s')$ and other nearby states nearby by virtue of correlations. Given that such states are usually within the neighbourhood of where the SpiderBot might likely to be in the next few time steps, such correlations can be extremely harmful to stability in training. Hence, to address this problem, the target FCDNNS essentially capture the weights of the online FCDNN that is being trained at a specified earlier time step by periodically synchronising with the online network (e.g. hardcopy/softcopy). The target FCDNN then uses its output as the target instead of that from the online FCDNN to train the online FCDNN, hence helping to mitigate the correlation between the target (e.g $r + V_\omega(s')$) and the predicted value (e.g. $V_\omega(s)$).

By using the replay buffer, the training phase, where the weights of the FCDNNs are updated, is decoupled from the experience gathering phase, where experiences consisting of the state (s), action (a), new state (s') after taking action, a, from state, s, reward from the reward function (r) as well as the terminal flag are gathered. This gives benefit of an efficient use of previous experiences, as the stored experiences can be reused multiple times for learning, rather than being discarded immediately after every specified time step like in MAA2C, A2CMA and A2CSA. This benefit is especially valuable in the context for this project as it is computation time-wise extremely expensive to generate specific experiences (e.g. experiences of episodes of success) given the countless of possible states the SpiderBot with 32 joints. As a result, the utility of the replay buffer would greatly aid the convergence to a sufficiently optimal policy.

In addition, a dueling FCDNN, where the state-action value, $Q_{\pi_\theta}(s,a)$, is split into the state value, $V_{\pi_\theta}(s)$ and the advantage value, $A_{\pi_\theta}(s,a)$ as shown in equation 2 is implemented for MAD3QN. As a result, the state value, $V_{\pi_\theta}(s)$ functions as a form of baseline that affects all actions in the same state, hence helping to speed up training and inference.

5.2 Limitations

As the advantages of certain algorithms, of which highlights the limitations of other algorithms (e.g. lack of replay buffer and target network for MAA2C, A2CMA and A2CSA), have been discussed above, this section will focus on other limitations that the algorithms face in general. One of the possible limitations faced by the algorithms would be working with a discrete action space (not applicable to DDPG). This is a significant limitation given that the implementation of "actions" in the form of permutations of an finite list of target velocities being applied grows exponentially with increasing number of joints or with an increasing list of target velocities. For example, the implementation of discrete actions cannot be applied to the entire SpiderBot through a single actor FCDNN as the minimal case of two opposing velocities already has untenable 2^{32} possible permutations of target velocities given 32 joints. Hence, by working with a discrete action space, the possible actions of SpiderBot is inevitably constrained by the computational resources available. Given that there is no prior intuition to the the values for the target velocities nor the size of the list of the target velocities, the selection of the stated variables inevitable becomes part of the numerous hyperparameters to be tuned. This is due to the fact that it is very possible that certain lists of target velocities makes the SpiderBot inherently unstable by the laws of physics (e.g unsuitably large velocities), for which the algorithms cannot be faulted for its failure to converge to a solution. Hence, it would be most ideal to be able to have implement algorithms that works with a continuous state and action space.

Another general problem faced by all algorithms would be the fact the value based updates (e.g. state value, state-action value) for the critic FCDNNs, which eventually affects the actor FCDNNs, or all the FCDNNs for MAD3QN follows a one step look-ahead for the TD error. This is problematic as the one-step look-ahead is inherently shortsighted given that it does not reward actions taken at states numerous time steps back that could have contributed significantly to the SpiderBot's ability for forward motion at the current time step. An example of such a macro action would be the act of lifting the leg up and moving the leg forward in air. Under the current implementation of the algorithms, the series of actions for a given state that constructs this entire macro action would be have a reward of zero throughout when used to update the weights of the FCDNNs given that it does not give the base of the SpiderBot any gain in forward position or velocity. However, such an action would actually be vital in propelling a spider forward in reality. Hence, some form of eligibility traces to update the rewards of past experiences indexed by time steps from the rewards of the current time step during experience gathering could potentially be implemented for the replay buffer in future work for DDPG. The eligibility trace serves track the extent to which the past set of experiences is "responsible" for the current reward based on its recency in terms of time steps. As a result, such a reward update every time step for the replay buffer would holistically better represent the value of actions taken in past states, hence potentially lead to a quicker convergence to an optimal policy.

6 Conclusion

On hindsight, this project was more challenging and ambitious than what was envisioned during the proposal stage. A large amount of time was invested in learning how to utilise the PyBullet gym given the relatively sparse documentation, non-existent community support and bugs. An important lesson learned would be the value of good CAD practices. This stems from the experience where the geometric center of a square in a sketch to be extruded for the base of the SpiderBot is not co-incident with the origin of the sketch in SolidWorks. As a result, upon exporting the finalised SpiderBot to URDF, the centre of mass of the base of the SpiderBot is skewed towards a side of the base instead of being the geometric centre. Hence, the SpiderBot, as well as it different legged versions, were inherently unbalanced for numerous trials before the discovery of the error further into the project. However, the most challenging part of the project was the learning and implementation of the five algorithms in the context of the problem in this project. Given that this is first deep reinforcement learning project embarked by us, the learning curve was extremely steep. This is especially when the algorithms utilised initially were not working despite numerous trials and errors with hyperparamter tuning. As a result, a significant amount of time and effort was invested in understanding the limitations of the existing algorithms in order to research on more state-of-the-art algorithms to address the problems faced, of which the implementation is another big challenge on its own. In addition, by implementing new algorithms with new features (e.g replay buffer, target networks), a whole new series of hyperparameters were gained as well. Naturally, the hyperparameter tuning posed another great challenge for us as we have no prior intuition for the values for the parameters given our lack of experience. As a result, the intuition had to be gained hard way, through numerous trials and errors. Nevertheless, despite the challenges faced, we reaped numerous gains within a short month and I am grateful for the experience and humbled by vastness of the field of deep reinforcement learning.

6.1 Declaration

I declare that all the code submitted is written 100% from scratch. Online resources (e.g. Youtube tutorials, blog posts) were referred to, but with no copy and paste. For grading purposes, I declare that my group mate, Arijit Dasgupta (A0182766R) and I contributed 50% each to the project.