ME5406: Deep Learning for Robotics

Part II Project Report

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

Student Name:
Arijit Dasgupta

Student Number:
A0182766R

Student Email:
arijit.dasgupta@u.nus.edu

November 18, 2020

1 Introduction

1.1 Problem Statement & Motivation

In this group project, I partnered with Chong Yu Quan (A0136286Y) to undertake a very challenging task of training our custom-design spider robot walk in a 3D virtual physics engine with deep reinforcement learning. Figure 1 introduces SpiderBot, an 8-legged robot inspired by a spider that we proudly designed in SolidWorks 2019. The robot has 8 legs, 4 revolute joints per leg, tabulating to a total of 32 joints. The aim of this project is to get the SpiderBot to walk to a target in a specified direction (positive x-axis) in a plane world with no obstacles. We set the target distance to 3 metres.

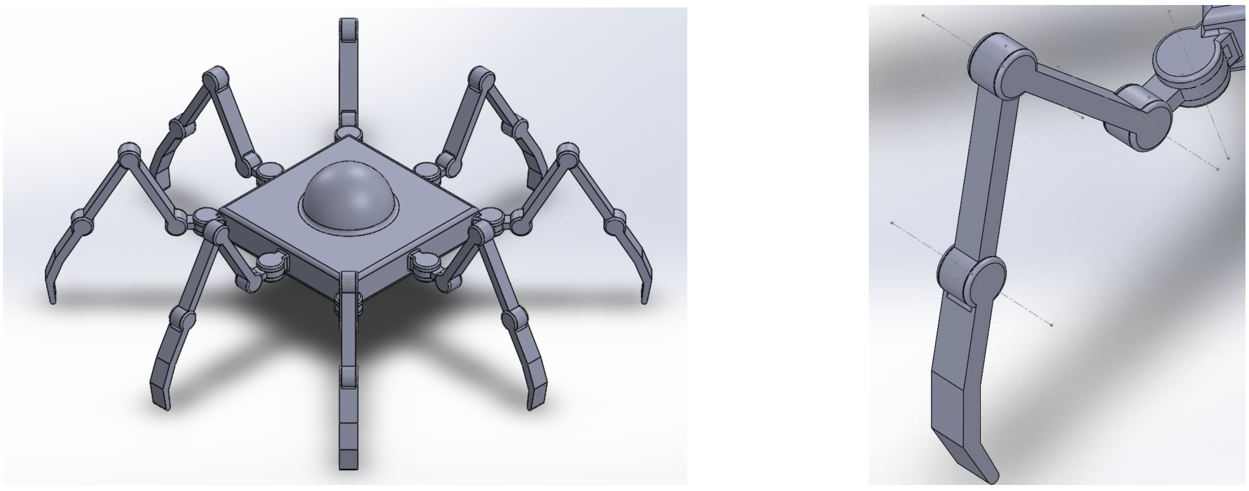

(a) Full Model (b) Closeup of Leg (4 Revolute Joints)

Figure 1: Our CAD model of SpiderBot

The motivation behind this robot comes from the increasing developments in insect robots for the purpose of surveillance and rescue. An interesting example would be that of RoboBee, a micro-robot developed by researchers at Harvard that is capable of flight. Small robots like these are capable of reaching areas like small vents and holes and they can navigate autonomously in a swarm. This inspired us to take up a spider design for our robot. Although our robot is not micro (it is 20cm by 20cm at the square base), we consider it a good start. We believe this can be potentially be scaled into larger and more complex environments in future studies. Imagine having a swarm of such spider robots traversing sites of destruction to locate survivors. The multiple legs and joints give the robot a lot agility and flexibility in navigating difficult terrains. In this example, we start simple with an infinite flat plane and no obstructions.

1.2 Existing Methods

There are a few approaches one can conventionally take to make the SpiderBot in Figure 1 walk. The most conventional way would be the "classical robotics" approach, which does not involve reinforcement learning. Unlike the end-to-end method that deep learning offers, a classical robotics approach would use low-level modularity in their implementation. This would make use of optimal control theory to control the actuation of the robot legs. Making a robot walk 'classically' could also use dynamics calculations while requiring good understanding of the mechanical system. Such a classical approach would usually use sensor data and many steps of filtering and data processing to tell the robot what to do. Some processes even require complex inverse kinematics calculations. A cool example of this would be the Spot robot from Boston Dynamics that uses a wide range of smart sensors and modules to do specific tasks like climbing stairs or navigating an uneven terrain. To summarise, there are many modules and steps that work together to make a decision on actuation. The advantage of such a method is that they are generally reliable. As the designers ground the robot mechanics on theory, it would be very likely that they would be able to make our SpiderBot walk with enough work.

The issue with the classical approach however, is that many aspects of the system are essentially hardcoded. Such hard coded logic is extremely cumbersome, difficult to optimise and untenable in terms of scalability to more complex actions in complex environments. Moreover, if any link is damaged, the mechanics-dependent movement of the robot may not allow the robot to recover well unless a lot of tedious work is done to prepare for such scenarios. With this, we come to the next branch of algorithms, into the dark yet wonderful field of reinforcement learning (RL). To put it crudely, RL algorithms don't have to know any rules or strategy of the walking mechanism. There is no hardcoded or modular aspect that is commonly present in classical robotics methods. Being an end-to-end method, an RL agent simply needs to have an understanding of its state, an interface to actuate the joints and a feedback of reward at every time step. It does not need to know what happens with actuation or go through a series of complicated modules. A very common approach is using Q-learning, which is a tabular method for learning Q, the action-state value function which determines the maximum expected rewards for an action taken in a particular state. The algorithm allows for online learning via bootstrapping and it learns a table of Q values, where $Q \in S \times A$. Assuming a fully trained model with Q learning, the walking spider could search up its current state in the table and greedily take the action with the highest Q value as an optimal policy.

Nonetheless, the Q learning approach would be completely unfeasible in the case of the SpiderBot. Imagine at the very minimum, the agent can take only a maximum of two actions per joint (rotate clockwise or anticlockwise), the size of the action space would be as large as $2^{32} = 4294967296$. Even if there was only one state in the Q table, the size of the table (assuming a numpy array) would be 34 Gb in size. Even the state space of the walking spider would be colossal in size should we choose to discretise it for a Q-table. This also means that searching through the Q table would be slow, slowing down the online learning. Therefore, a approach that can scale better would be the use of deep learning methods. Deep learning is a technique for universal approximation that uses batches of data for training. Hence a Deep Q Network would approximate the Q value of taking an action at a state by giving a state-action pair as input; $Q(S_t, A_t) = f(S_t, A_t)$ where f refers to the neural network. The network can take in a continuous space state representation and have a final layer of N neurons that correspond to the Q value of each one of the N actions. Two networks, a target network and a policy network can be used in conjunction to conduct double deep Q learning as shown in Equation 1. θ' refer to the neural network parameters of the target network while θ refers to the neural network parameters of the policy network. $Q'_\theta(S_t, A_t)$ on the LHS of Equation 1 is the value that is compared with the policy network output, $Q_\theta(S_t, A_t)$ for the loss function.

$$Q'_\theta(S_t, A_t) = Q_\theta(S_t, A_t) + \alpha[R_{t+1} + \gamma Q_{\theta'}(S_{t+1}, arg\ \max_{a'} Q_\theta(S_{t+1}, a')) - Q_\theta(S_t, A_t)] \tag{1}$$

Finally, another conventional approach in deep reinforcement learning would be to learn the policy of walking rather than approximating the Q values of every state and action. These policy approximators use policy gradient methods which attempt to learn a policy, $\pi_\theta(S_t, A_t)$, for any given state. π refers to the probability distribution of taking an action given the state and parameters θ. There are numerous of such algorithms ground on the mathematical relation in Equation 2 which describes the gradient of $J_Q(\theta)$, the cost function of the policy gradient optimisation problem to approximate π based on parameters θ.

$$-\nabla_\theta J_Q(\theta) = E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(S_t, A_t) \cdot Q_\pi(S_t, A_t)] \tag{2}$$

We can take advantage of the gradient relation in Equation 2 and the automatic differentiation capability of TensorFlow 2 in Python to conduct a step-wise approximation using deep learning. This way we can use deep learning to learn a policy for walking. Given the advantages that reinforcement learning and deep learning provide, the present report will focus on using deep reinforcement learning to tackle our problem statement.

2 Methodology

2.1 Algorithms

There are a multitude of algorithms to use in deep reinforcement learning along with many ways to approach the RL cast of the walking spider problem. For this project, we implemented a total of 5 approaches. The approaches have a mix of value approximation & policy gradient methods, single-agent & multi-agent casting, single-action policy & multiple-action policy and separate network learning & hybrid network learning. Table 1 summarises the characteristics of our five approaches.

Table 1: Characteristic of algorithms implemented in the present project

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

2.1.1 MAD3QN

The first approach is to use Multi-Agent Double Duelling Deep Q Networks (MAD3QN) to make the SpiderBot walk. In this approach, each leg in the spider is designated as an agent. We define an agent as an entity that can only take actions for itself. This means that one leg cannot make a decision for another leg. As an additional definition, each agent must have its own neural network for training. Thus for a spider with 8 legs, there are 8 separate agent Q networks. As there is no parameter sharing between any of the agents, the policy of the entire SpiderBot is decentralised to each one of the legs. However, we allow the state inputs into each one of the agent Q networks to be the same global states. Our justification for this approach is that having knowledge of the other legs may allow each leg to make better decisions as moving a leg in the same way may lead to different outcomes depending on the states of other legs. Additionally, this approach makes use of a double network approach to estimate the Q value. This is the same as the target and policy network approach shown in Equation 1. The reward from Equation 1 is the same for all agents, as this is a cooperative task. Moreover we also adopted the duelling network (as shown in Figure 2) that separates the stream into state values and advantage ($\tilde{A}$), which is defined as $\tilde{A}^{\pi}(S_t, A_t) = Q^{\pi}(S_t, A_t) - V^{\pi}(S_t)$. This way, the network can focus on states that are more important than others. To calculate the Q value from $V(S_t)$ and $\tilde{A}(S_t, A_t)$, Equation 3 is used. N_A refers to the number of actions/advantages in each network.

$$Q_\theta(S_t, A_t) = V(S_t) + \tilde{A}(S_t, A_t) - \frac{1}{N_A}\sum_a \tilde{A}(S_t, A_t) \tag{3}$$

Using a target and policy network as seen in Equation 1, the loss is calculated as $L(\theta) = \frac{1}{N}\sum_N (Q_\theta(S_t, A_t) - Q'_\theta(S_t, A_t))^2$. The weights of the target networks are updated softly with a τ of 0.005. Additionally, we use experience replay in this algorithm and batch learning to learn from a batch of experiences. Finally, an ϵ-greedy policy is used as the behaviour policy for this training.

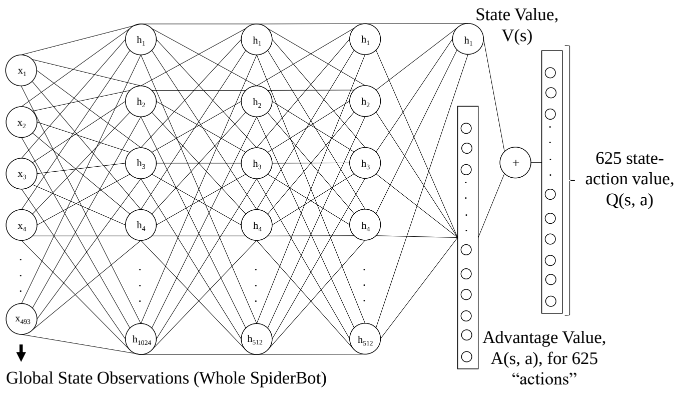
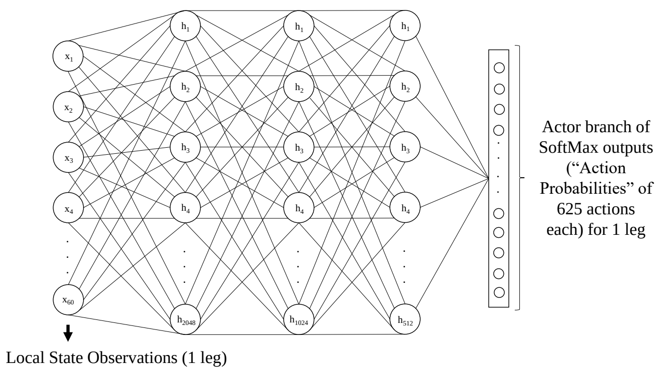

State Value, V(s)
625 state-action value, Q(s, a)
Advantage Value, A(s, a), for 625 "actions"
Global State Observations (Whole SpiderBot)

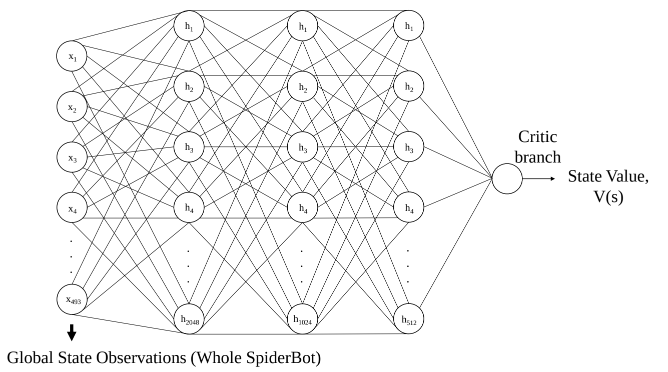
Figure 2: Neural Network Architecture for MAD3QN

2.1.2 MAA2C

In the next three algorithms, we used Advantage Actor Critic. In this algorithm, the Advantage ($\tilde{A}$) is the baseline to be calculated in the policy gradient update. We chose to adopt a policy gradient method as it is known to converge to a solution faster than value-based RL. In this approach, we adopted a multi-agent verison of the algorithm (MAA2C). Like MAD3QN, each leg of the spider is assigned as an agent and also has its own actor network (Figure 3a), leading to a decentralised policy. Each actor network outputs a probability distribution, $\pi_\theta(S_t, A_t)$, with a softmax layer. However, each actor network only takes in state observations locally, of its own agent. Another difference is that the network is an actor network, which does not try to predict a value function, but the policy instead. Other than the 8 actor networks, there is also a network for a critic (Figure 3b), which tries to predict $V(S_t)$ with the input of global state observations from the whole spider. There is only one critic network for the SpiderBot, which all actors networks share. The $\tilde{A}$ is estimated by the TD error of the state-value function, $R + \gamma V(S_{t+1}) - V(S_t)$. Hence the critic only needs to estimate $V(S_t)$. The critic loss $L(\theta_c)$ is calculated as $\tilde{A}^2$ and the actor loss $L(\theta_a)$ is calculated as $-\log \pi_\theta(S_t, A_t) * \tilde{A}$.


Actor branch of SoftMax outputs ("Action Probabilities" of 625 actions each) for 1 leg
Local State Observations (1 leg)

(a) Actor Network

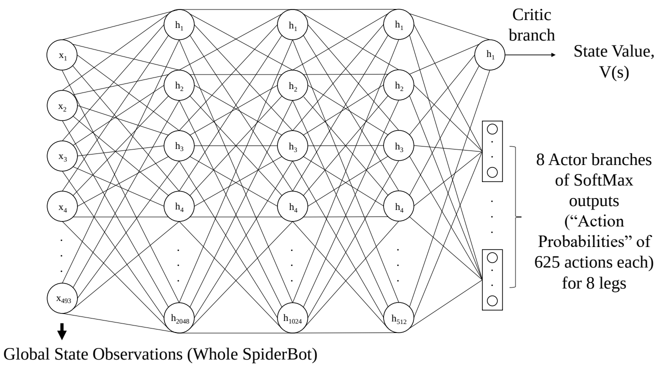


(b) Critic Network

Figure 3: Neural Network Architecture for MAA2C

2.1.3 A2CMA

The next approach, Advantage Actor Critic MultiAction (A2CMA) is extremely similar to MAA2C. The main difference is that A2CMA uses a combined hybrid neural network for all 8 actors and critic (Figure 4a). The hybrid neural network takes in global state observations of the whole SpiderBot, like each agent network from MAD3QN. By our definition, as there is a single network, we treat it as a single agent system. However, we decentralise the policy learnt by splitting the final layer of the network into 8 branches: 8 actor branches for each leg along with 1 branch for the critic. This means that each one of the output branches has a softmax layer to learn the policy, $\pi_\theta(S_t, A_t)$, for the specific leg. The reason we chose this hybrid method of learning is to contrast with the local observation state input of the actors in MAA2C. The hypothesis is similar to the reason for using a global state space input in each agent Q network from MAD3QN, to allow each leg to learn the policy while taking into account the states of all other legs; as moving a leg in the same manner in two different global state versions may lead to completely different outcomes. Each one of the actor losses from each branch and the critic loss are the same as defined for MAA2C, and they are summed together.


Critic branch
State Value, V(s)
8 Actor branches of SoftMax outputs ("Action Probabilities" of 625 actions each) for 8 legs
Global State Observations (Whole SpiderBot)
(a) MultipleAction (A2CMA)

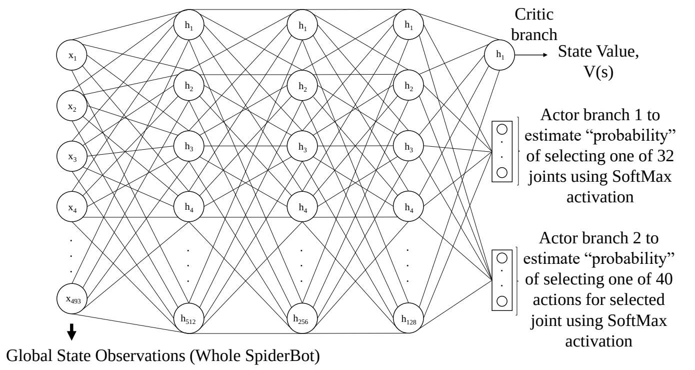

Critic branch
State Value, V(s)
Actor branch 1 to estimate "probability" of selecting one of 32 joints using SoftMax activation
Actor branch 2 to estimate "probability" of selecting one of 40 actions for selected joint using SoftMax activation
Global State Observations (Whole SpiderBot)
(b) SingleAction (A2CSA)

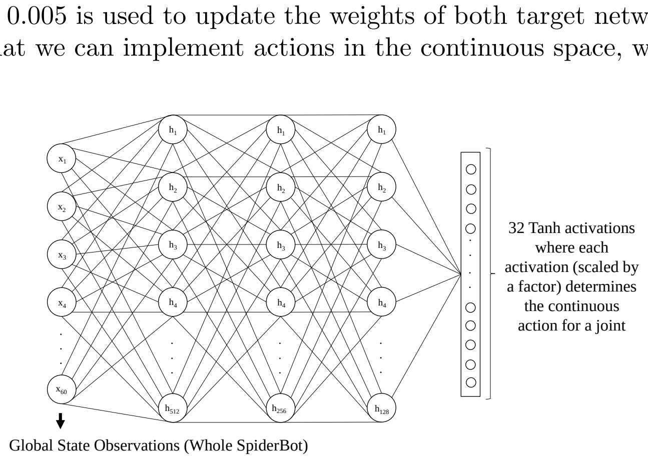
Figure 4: Neural Network Architectures for A2C Approaches

2.1.4 A2CSA

So far in all the algorithms mentioned, all joints and legs are actuated at every time step as described in Table 1. One concern we had at this point was that moving all joints at once will lead to a form of chaotic motion that does not allow the legs of the spider to work together. To create an alternative, we came up with the Advantage Actor Critic SingleAction (A2CSA) version. In this case, a hybrid network is also used, but not with 8 actors. As seen in Figure 4b, there are two branches at the end of the network that we consider as two special actors. The first branch outputs a probability distribution of each one of the 32 joints with a softmax activation and the joint with the highest probability is chosen. The actor in this branch is thus learning the policy of choosing a joint. The second branch outputs a policy $\pi_\theta(S_t, A_t)$ for that selected joint using softmax activation again. Hence the actor for this branch learns the policy of selecting actions given the joints. This way, in one time step, only one joint is selected for an action. The losses for the two actor branches are calculated in the same way ,$-\log \pi_\theta(S_t, A_t) * \tilde{A}$, as they are both learning a policy.

2.1.5 DDPG

The final approach we tested was the Deep Deterministic Policy Gradient (DDPG) algorithm. This algorithm is founded on the deterministic policy gradient theorem and tries to learn a deterministic target policy. Like A2C, this approach uses both an actor and critic, but with a separate network for both. The actor network (Figure 5a) takes in a global input state of the spider bot and outputs an action in a continuous space for all 32 joints. The output comes from a hyperbolic tangent activation, which can be scaled a larger range. The critic (Figure 5b) outputs a Q value based on the same global input state and the action. Like MAD3QN, DDPG uses experience replay and target networks for the actor and critic. The loss of the critic is captured by taking the TD error, using both the target critic and critic. The actor network is updated using the deterministic policy gradient update and the target actor output. A soft update of $\tau = 0.005$ is used to update the weights of both target networks. The motivation to use this algorithm came from the fact that we can implement actions in the continuous space, which allows for the SpiderBot to move in a smooth manner.


32 Tanh activations where each activation (scaled by a factor) determines the continuous action for a joint
Global State Observations (Whole SpiderBot)
(a) Actor Network

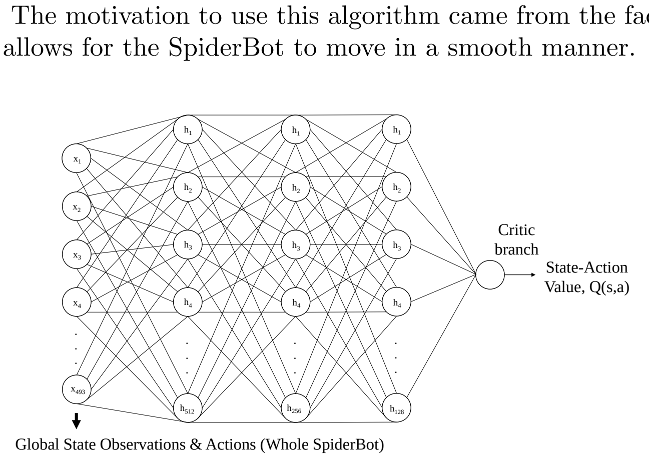

Critic branch
State-Action Value, Q(s,a)
Global State Observations & Actions (Whole SpiderBot)
(b) Critic Network

Figure 5: Neural Network Architecture for DDPG

2.2 RL Cast

It is extremely important to cast the RL framework onto the problem statement effectively. This refers to the design of the state space representation, action space and reward structure. In this project, the state space is represented by a vector of scalar values for the base link (centre square region of SpiderBot) and a joint as shown below. Many of the state values are taken with respect to the centre of mass (COM) in world coordinates.

State Representation for Base Link:

- Position of COM (x, y, z)
- Orientation in Quaternion $(a, b\mathbf{i}, c\mathbf{j}, d\mathbf{k})$
- Linear Velocity of COM (v_x, v_y, v_z)
- Angular Velocity $(\omega_x, \omega_y, \omega_z)$

State Representation for each Joint:

- Joint Position, θ
- Joint Angular Velocity, ω
- Position of local COM (x, y, z)
- Local Orientation in Quaternion $(a, b\mathbf{i}, c\mathbf{j}, d\mathbf{k})$
- Local Linear Velocity of COM (v_x, v_y, v_z)
- Local Angular Velocity $(\omega_x, \omega_y, \omega_z)$

The state of the base link can be represented by a vector of size 13 while each leg can be represented by a vector of size 15. Hence when we refer to the input of local state observations in the case of MAA2C, each actor network takes in a vector of size 15*4 (4 joints in a leg) = 60. In the other approaches where the critic, agent or actor takes in the global state observations, the input size of the neural network is 13 (base link) + 15*8 (8 legs) = 493. Note that as neural networks are used, the state space can be made continuous for all approaches. Moving on to the action space, we have the ability to set actions for each one of the 32 joints. We initially considered setting a target angular displacement, θ, but our initial observations showed us that none of the joints attached to the base were moving at all. We then took an alternative strategy to set a target angular velocity of each joint, ω in rad/s. Given that the actions theoretically exist in the continuous space, we discretised the ω to a few options for each joint, for the MAD3QN, A2CMA & MAA2C approaches. With our manual tuning, we decided to stick to the following action space (ω) of [-10, 5, 0, 5, 10] rad/s. This meant that one leg had a total of $5^4 = 625$ permutation of actions. In the case of A2CSA, as the second actor branch simply has to chose an action for one joint, we gave it a larger range from -20 rad/s to 20 rad/s in intervals of 1. Finally, as the DDPG algorithm allowed for a continuous action space, we scaled the final hyperbolic tangent activation for each joint by 10 to give a continuous action space from -10 rad/s to 10 rad/s.

The reward structure is a crucial part of the RL cast as its design can encourage or discourage the SpiderBot from taking certain actions. We took a multifaceted approach for the reward structure by including various components that affect the reward. There are 8 possible rewards the SpiderBot can get at every time step. The first is a reward proportional to the forward velocity of the spider with the second being a negative reward that gets smaller the closer SpiderBot gets to the target. The values are set to 500 & 250 respectively. The aim of these two rewards is to encourage the robot to move forward in the correct direction. The thirds and fourth are negative rewards that are proportional to the sideways velocity and sideways distance from the expected line of walk. They are set to -500 & -250 respectively. Naturally, the negative rewards mean that we are trying to dissuade the SpiderBot from moving sideways. The fifth reward is the time step penalty (set to -1) that has an increasingly negative reward proportional to the time of the episode thus far with the aim of pressuring the SpiderBot to move. The sixth to eight rewards come from 3 specific termination conditions. It received a positive reward when the goal is reached (500) and a negative reward for falling down (-500) or moving out of a specified range (-500) i.e. 1m sideways and 1m backwards. We had initially imposed a punishment if the SpiderBot han any rotation. However we observed that the spider did not want to move at all in the end, therefore we went without it.

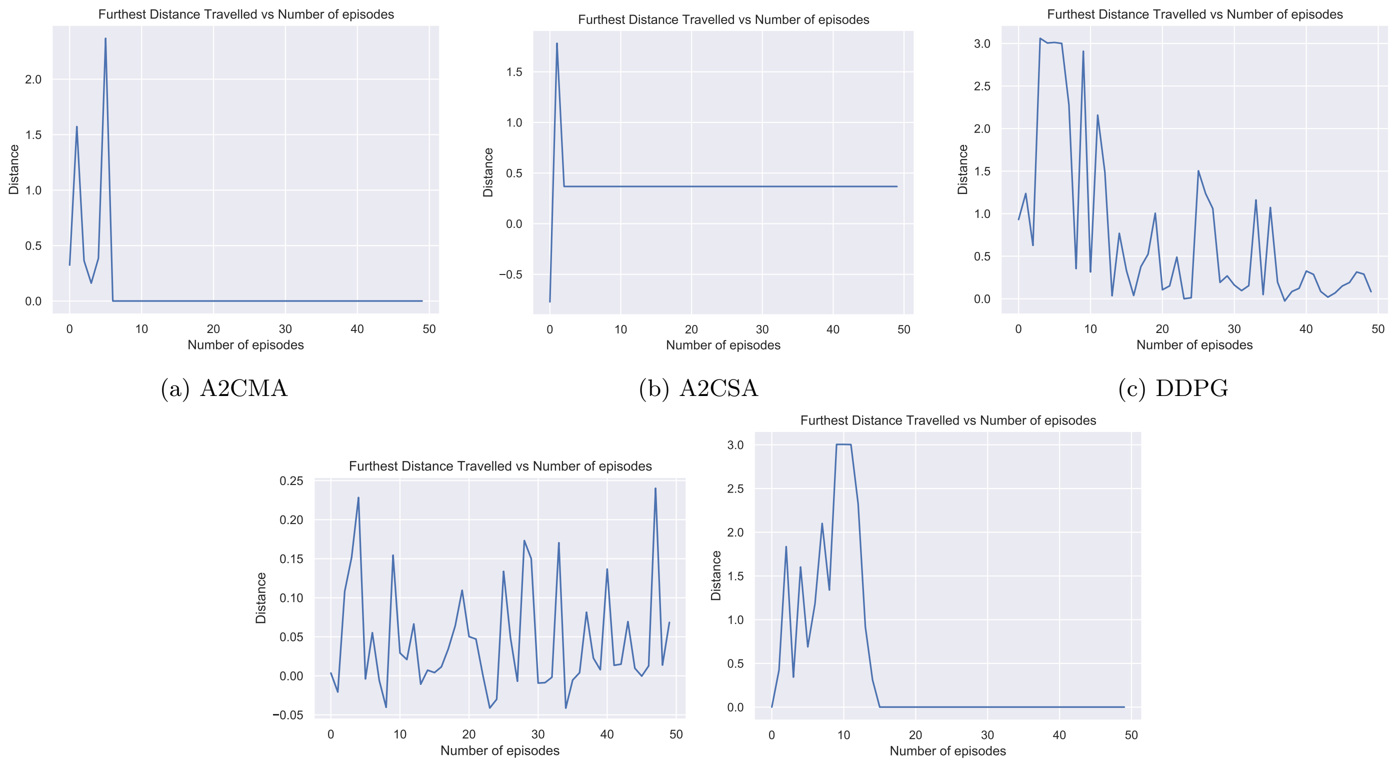
2.3 Environment & Training Process

The creation of the environment involves a few steps, starting with the CAD model of the SpiderBot. The CAD model is converted to a URDF file, which contains information of the links, joints & STL files of the CAD. This URDF is then uploaded to a physics engine in OpenGL, where we fully designed a gym environment to interact with the virtual model. The PyBullet API is adopted to aid us in interfacing with the SpiderBot. We also use the API to limit the range of all joints from -60° to 60°, where 0° is the starting position for each joint based on the URDF. The interfacing allows us to record state observations, set target velocities at joints and gather information to determine rewards. For instance, a robot is deemed to have fallen if any second link (where the first link is defined as the link normally in contact with the ground) has contact with the ground. Additional to the 3 termination conditions, an additional termination condition is added to limit an episode to 10 minutes in real time. The final aspect of the environment is the length of time given to allow for the SpiderBot to execute the action. If the time step is too low, the joints will barely be able to move due to the lack of time. A time step too large could mean that the joints have reached their limits, which would not be productive to movement and overall learning. Hence, we tuned the time step to about 0.1s when comparing across the algorithms.

For the training itself, we ran each one of the algorithms in TensorFlow 2, accelerated by an NVIDIA GeForce GTX 1060 GPU. Due to the lack of time and computational speed, we had to limit each run to 50 episodes. The learning rates for all actors, critics and the agents in MAD3QN were set to 0.001. A reasonable discount factor of 0.9 was used. A replay memory of 1 million with a batch size of 256 was used for DDPG & MAD3QN. All neural networks used a total of 3 fully connected hidden layers with a decoder-like structure. MAA2C & A2CMA used an architecture (number of neurons in each layer) of [2048,1024,512] while DDPG & A2CSA used [512,256,128] and MAD3QN used [1024, 512, 512]. An L2 regulariser and batch normalisation is used at every layer in all networks. All non-output neurons use a Rectified Linear Unit (ReLU) activation function in this project. To optimise the learning of the neural networks, an Adam optimiser is adopted. During training, all relevant information is logged and subsequently post-processed. The subsequent section will present the results for a comparison between the 5 approaches along with a reward structure analysis for the best performing approach and more.

3 Results

We first look at the comparison between the 5 approaches in tackling our problem task. Figure 6 illustrates the furthest distance in the x direction the SpiderBot was able to move throughout the episodes.



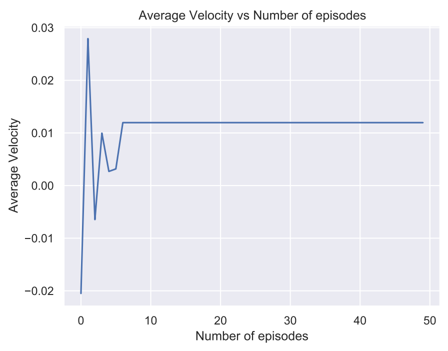
(a) A2CMA (b) A2CSA (c) DDPG

(d) MAD3QN (e) MAA2C

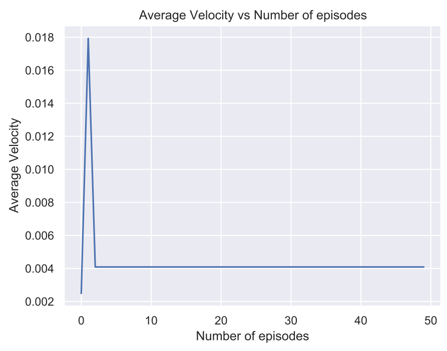
Figure 6: Furthest Distance reached per episode for different algorithms

The plots in Figures 6c & 6e show us that that MAA2C and DDPG are the only approaches that actually met the target distance of 3m a few times. The DDPG approach achieved successes rather early on, however it was not able to maintain that for the remainder of the episodes. In MAA2C, the SpiderBot was slowly improving towards the goal and once it reached success, it very immediately converged to a bad local minimum policy, where it always took actions to fall down at the start of every episode. The other approaches all showed to perform very poorly in comparison, with A2CMA & A2CSA converging to a bad local minimum policy quickly despite A2CMA almost the reaching the target once. Finally, Figure 6d showed that the MAD3QN approach did not converge into a policy & it never went very far during training (maximum of 0.24m).

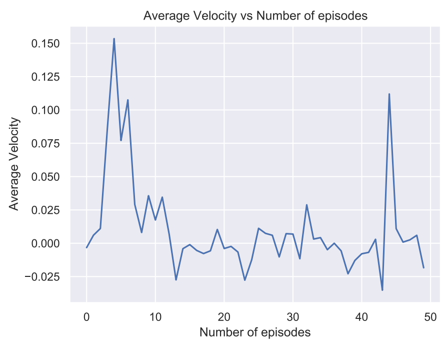
Other than the best distance metric, another important measure is the average velocity of the SpiderBot during each episode. This is because the SpiderBot can theoretically achieve a positive furthest distance travelled but still be moving backwards most of the time. The plots in Figure 7 for average velocity appear to be quite correlated with the furthest distance plots in Figure 6. A close look at Figures 7c & 7e show that the SpiderBot has an impressively high forward velocity of 0.16 m/s in the cases where the DDPG & MAA2C approaches reach the target. This shows that in those instances, the SpiderBot was able to walk forward to the goal without moving far sideways. With these results, we concluded that MAA2C and DDPG are our best performing approaches. However, to sieve out a best performing approach, we take a look at the training loss of both these approaches.



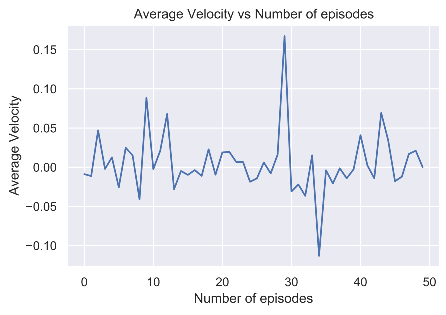
(a) A2CMA



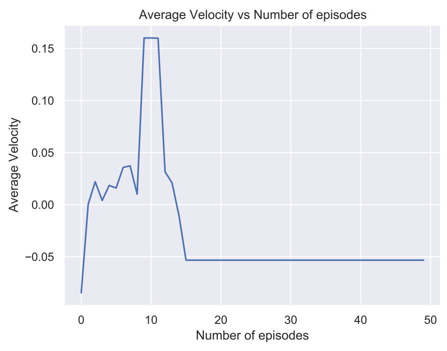
(b) A2CSA



(c) DDPG



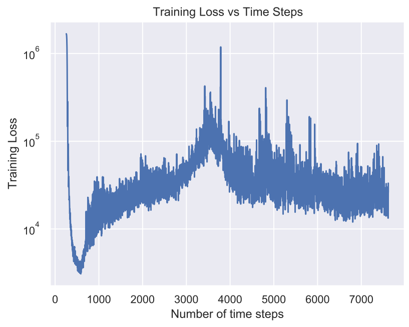
(d) MAD3QN



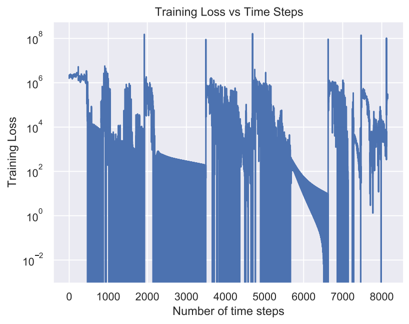
(e) MAA2C

Figure 7: Average Velocity per episode for different algorithms

Figure 8 compares the training loss between DDPG & MAA2C. Note that as the architectures for both approaches are different, we should not directly compare the scalar value of the loss, but rather the trends they show. Figure 8b shows that there is an oscillating trend in the loss for MAA2C. The loss in one time step can be as high as 10^6, drop to 10^-2 in the subsequent timesteps and then rise to 10^6 again. This is clearly very poor learning and the training process needs to be re-looked to understand why such a trend occurred. On the other hand, the loss from DDPG in Figure 8a is a lot more stable and shows a more reasonable trend. The loss dropped quickly in the early part of training it its minimum and then it started to increase afterwards. Interestingly, the region of lowest loss corresponds to the times the SpiderBot reached the goal. This is a sign that perhaps the SpiderBot is indeed learning to walk in the specified direction. With these observations, we concluded that DDPG is our best approach. However, the results are not satisfactory as the successes were not sustained for the rest of the training. Hence, we considered changing the reward structure of training DDPG.



(a) DDPG

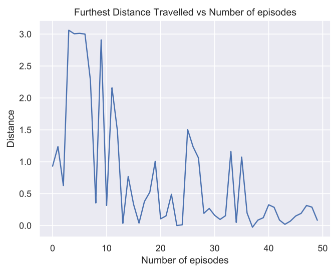


(b) MAA2C

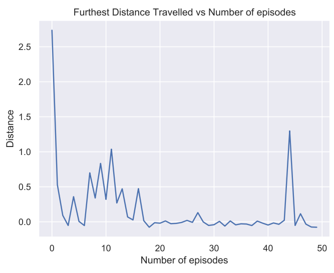
Figure 8: Neural Network Training Loss with time steps

To test different reward structures, we adopted three new philosophies. The first is to emphasise on reward for forward motion in hopes that the SpiderBot is mainly encouraged to move forward. We multiply the rewards for forward velocity and distance from target by 10 and call it Version A. The second philosophy is to emphasise the punishment for the SpiderBot from moving sideways by multiplying both sideways distance and velocity punishments by 10 and we call this Version B. The aim of this reward structure is to keep the SpiderBot focused on moving ahead. The final philosophy is to significantly increase the rewards for termination conditions by multiplying all rewards for termination by 100 and call it Version C. The aim of this approach is to encourage the SpiderBot to meet the goal more often and to not terminate by going sideways or backwards.

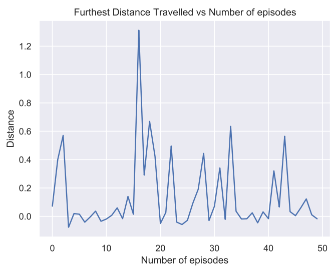
Figure 9 shows the comparison for the original and Versions A-C reward structures for the DDPG approach. What



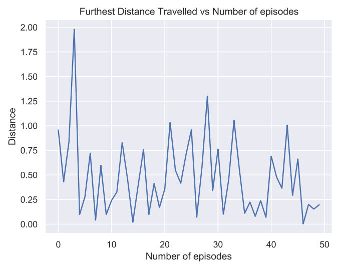
(a) Original



(b) Version A



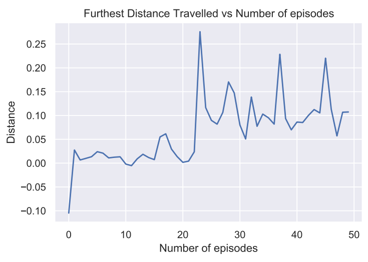
(c) Version B



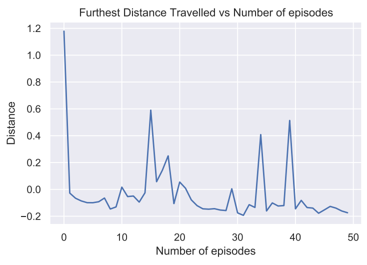
(d) Version C

Figure 9: Furthest distance travelled for different reward structures for DDPG

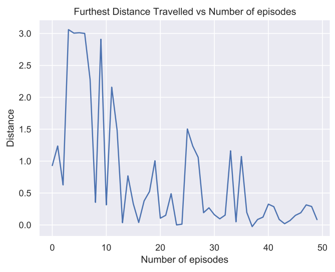
we find is that none of the alternative reward structures worked as well as the original reward structure. The worst performing was Version B in Figure 9c, where sideways movement was severely dissuaded and the SpiderBot only achieve 1.2m at the best. One interesting observation made was that in Version A (Figure 9b) the SpiderBot managed to reach a high 2.7m on the very first episode. This shows that the forward rewards did encourage it to move forward in the early stages of training. However this good performance quickly went away in subsequent episodes. Finally, we thought that perhaps the number of states were too high for the SpiderBot to learn effectively. Hence we decided to run the original reward structure and DDPG with SpiderBots of 4 & 6 legs to see how the agent performs with smaller state spaces. Figure 10 shows the comparison for SpiderBots with different number of legs. The plots show that as we reduce the number of legs, the performance of the SpiderBot only gets worse, with the 4-legged version not even reaching 10% of the target distance. With this, we came to the conclusion that with our set of hyperparameters and reward structure, the 8-legged SpiderBot was our best model to tackle our problem statement.



(a) 4 Legs



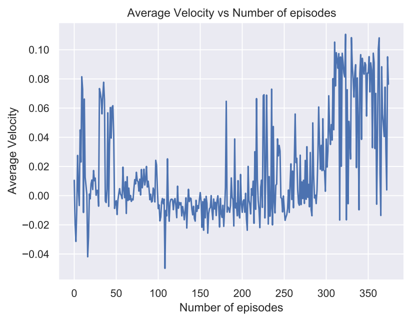
(b) 6 Legs



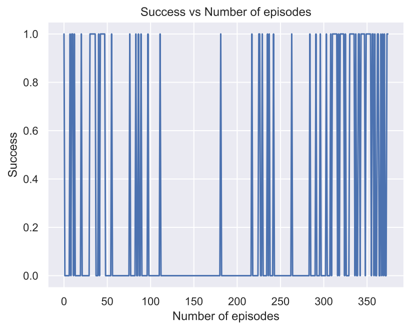
(c) 8 Legs

Figure 10: Furthest distance travelled for SpiderBots with different number of legs

To get a fully trained model for this project, we decided to run DDPG for a longer duration of 375 episodes. We also made a modification where we reduced the critic learning rate to 5×10^{-5} and actor learning rate to 1×10^{-4}. To our surprise, the model worked very well. Figure 11a shows a clear trend of the SpiderBot's average velocity increasing with the training. Figure 11b shows that at the end of training, there was a very dense region of constant successes. After saving the model, we realised that not only did the SpiderBot hit the target of 3m, it managed to reach 9m before terminating despite not having seen the 3m to 9m region before. We consider this a big success for our project.



(a) Average Velocity



(b) Success Rate

Figure 11: Fully Trained DDPG Model with 375 Episodes

4 Discussion of Results, Limitations & Advantages

To achieve a deeper understanding of the results of this project, we must discuss the what we saw the SpiderBot actually doing in the OpenGL physics engine GUI. Before commencing the experiments, we explored and tuned a few hyperparameters and observed the behaviour of the spider. Two of the most important hyperparameters we found were the length of each time step and the joint angle limits. When we set the time step to be very low, like that of 0.01s, we noticed that the SpiderBot was simply shaking and sort of vibrating on the spot. This is because it does not have enough time to make any action significant enough to cause a movement. When we set the time step too large, like a few seconds, we observed that the SpiderBot would start gliding away. Given that none of these scenarios are productive, we manually tuned the time step to around 0.1s. In the case of joint angle limits, when they were set too high, the SpiderBot action were almost violent-like. Likewise, a very low angle limit did not allow the SpiderBot to move very much. Therefore we also had to search the sweet spot for this hyperparameter, which was found to be 60°.

During the run of the experiments, the DDPG SpiderBot agent was seen to be very stable in its movement. On the other hand, the MAA2C SpiderBot was not at all stable and it was practically gliding most of the time, even to the target. After reaching the target a few times, it started constantly falling down at the origin where it spawned. In all honesty, we did not expect MAA2C to work well as we intentionally limited the state input to each actor network to local observations. Despite the fact that all actor networks share the same critic, they have no information on what the other legs are doing. Hence each one of the legs of the SpiderBot are not working together. The SpiderBot of the other multi-agent approach, MAD3QN, seemed to look like it was shaking near the origin and it would often move backwards. This is despite the fact that all agent Q networks had a global state input. One specific issue with MAD3QN was that it took an incredibly long time to conduct training, and it would often overload the CPU & GPU. This is because it has 16 networks (8 policy networks and 8 target networks) in total for training. This could mean that we need a much longer time for MAD3QN to train and converge to an effective policy. We conclude that perhaps it is an overkill to go with a multi agent approach for this problem given the resources available to us.

In the case of both A2C methods, the SpiderBot clearly failed to meet the target. The A2CMA shared a fundamental problem that all other algorithms (except A2CSA) shared. They all actuate all 32 joints at once, which leads to a very chaotic movement of the SpiderBot visually on the GUI. The fundamental issue stems from the fact that the reward is centralised and equal for all actors and agents in all our approaches. So imagine that the SpiderBot manages to inch forward mainly due to the action of joints in one leg. All the joints in that leg would be rightfully getting a positive reward. But what if there are other joints in other legs that took an action that inhibited the forward movement in that time step? Perhaps the net outcome may have been a forward velocity, but the unhelpful joints would have all gotten a positive reward. This is clearly not optimal as we want to avoid having those joint taking those adverse actions. We believe that this is a serious limitation of attempting to actuate all joints in one time step. The alternative to this would be the A2CSA option where only one joint takes an action at a time. This ensures that the reward given only applies to the joint in question. However, the downside that we observed with the SpiderBot using A2CSA is that the spider moves very slowly. Only taking one action at a time also means that the spider learns very slowly. It will take a significant amount more time steps for all the joints for the entire SpiderBot to be trained sufficiently. Something that was observed in the GUI was that a few slow actions from the A2CSA SpiderBot would quicky lead it sideways and lead the episode to terminate. This would mean that the episode would often terminate prematurely as many of the joints were not even used. A possible improvement to this would involve a larger boundary for the spider to walk sideways.

Moving on to the reward structure of the SpiderBot, we had a few interesting observations. When we heavily emphasised sideways punishments as in Version B, we realised that the SpiderBot suddenly became afraid to move. We thought this would encourage it to focus on moving forward but instead it became discouraged to move at all. An improvement to this would possibly require a finer tuning into the reward structures. On to the Version C reward structure, we realised a huge flaw in the approach. We are giving an incredibly huge reward simply for the time steps in which the SpiderBot took specific actions to lead to termination. There is too much emphasis placed on these actions as they may not have been the most appropriate for those rewards. For instance, imagine if the SpiderBot already had a forward momentum and in the last time step, the spider took actions to slow down but it then reached the goal. The rewards would not only be incorrectly assigned, but it would be incredibly high for those actions. Recall that most of the state representation involved velocity and angular velocity terms. Hence it is possible that the SpiderBot could repeat those actions near the origin which may not be optimal. This would explain the extremely poor performance of Version C in Figure 9d. The final limitation of this project comes not from the approaches, but from the PyBullet environment itself. After extensively using the physics engine, we realised that the 3D virtual can be very unpredictable and often not work the way expect it to. This means that the state transition probabilities are never equal to one as it would depend on the rendering of the physics environment. Moreover, we did not manage to set any friction settings in the environment, which may have led to unwanted consequences like gliding or slipping. Interestingly, the SpiderBot would sometimes learn to glide to get close to the target.

Despite being heavily critical of our approaches, we still recognise the advantages. DDPG in particular performed the best out of all the approaches, given that it exceeded our expectations and solved our problem statement. In particular, we believe that the continuous action space outputs of DDPG significantly helped with the learning as it allows for the SpiderBot to move smoothly for all its joints. Although we have not explicitly verified it, we also believe that the replay experience & target networks used in DDPG and MAD3QN contributed to better learning. The use of replay buffers mean that the agent is not biased towards a particular trajectory. The use of target networks meant that the DDPG critics/actors and MAD3QN Q networks did not over-estimate their rewards, which would have led to a situation of a dog chasing its own tail.

5 Conclusion

5.1 Challenges Faced

This has admittedly been one of the toughest projects I have ever done. With my group mate and myself being only third year undergraduates, we certainly did not expect a graduate model to be easy. Despite having little to no knowledge of robotics, we still chose to take on an extremely difficult project as we wanted to further challenge ourselves. To start with, we were only able to start on the project 3 weeks before the deadline as we were very busy with our industrial attachments. This meant that we had to manage our time well and work effectively with each other. Movong on, one of the greatest challenges of this project is the lack of computational resources to run a much wider range of experiments. Even though we had access to an NVIDIA GPU in a gaming laptop, it was not sufficient to conduct a more fine-tuned analysis. This was compounded by the fact that we had quite a few bugs in the code that we only realised a few days before the deadline. That meant that we had to discard some of our experiments and intuitions of the algorithms and hyperparameters and re-run them. This gave us even less time to deliver honest and proper results. Alongside that, there was a serious glitch in the physics engine when we tried to train the 4-legged and 6-legged SpiderBots, which kept unexplainably flipping on its side. We prematurely concluded that it was a physics engine issue. Later on, we found out that the spider kept flipping because the centre of mass was mistakenly set at the corner of the SpiderBot, hence the lack of fully surrounded 8 legs forced the physical model to topple. This was the result of a seemingly innocuous mistake in the CAD. Thankfully, everything was resolved in time. Another aspect of this project that made it increasingly challenging was that we decided to take on 5 full approaches, where most of them could be considered state-of-the-art algorithms. This meant that the task of coding out the entire neural network and agent classes was incredibly difficult and tedious. Nonetheless, with enough hard work and patience, the implementations were fully developed for all the approaches. Finally, the most complex part of the training process is managing the hyperparameters. With the multiple approaches that use many features like experience replay, target networks, hybrid networks, multiple agents, the hyperparameter space exponentially blows up. We don't have nearly enough time or computational resources to run an extensive hyperparameter search. To manage this, we manually tuned hyperparameters we found more important and adjusted them based on how the SpiderBot was behaving in the short term.

5.2 Learning points

One good consequence of going through a very challenging project is that I get to learn a lot. On the technical side, I learnt to use TensorFlow 2 for programming neural networks, especially to manually apply gradients using a gradient tape. I also learnt the fascinating theory and algorithms for deep RL. It is also my first time working with a physics engine, where it was satisfying to see my SpiderBot design come to life. However, more than the several technical implementation lessons of the project along with the findings from it, I learnt many other valuable lessons that I will take to my future projects. The first lesson being that simply throwing a fancy and complicated algorithm at a problem will not fix it. I must carefully analyse where the algorithm has worked before and hasn't and even consider using a simpler approach first. Second, it is important to manage expectations and resources for a project. Taking on a difficult project could mean that there would be many bouts of failure before getting it right. It also requires careful planning and it is important to consider the allocation of computational resources beforehand. Finally, it is extremely important to think and rethink the RL cast for RL problems. I can use a good model, but it is entirely dependent on the way I design my state & actions space, along with the reward structure to achieve success. I am very satisfied to have learnt so much from this project and I am glad this report is evidence of that. No doubt that I will take these ideas to my next project in RL.

5.3 Future Work

A good idea for further work would be to implement a form of distributed learning by using multiprocessing, so that the agent(s) can learn faster in parallel. Additionally it would be wise to reduce the state representation. Not all the local angular velocity and location of COM information on the SpiderBot would be useful in the case of global observation as the base link angular velocity and COM would be enough to learn the location and orientation of the SpiderBot. I would also consider using less joints per leg to reduce the complexity of the action space for one leg. One should also consider simpler algorithms to first see if they can solve the problem statement. Finally, a robust hyperparameter search is necessary to tune the performance of the learning agents. All in all, I am very pleased with the work we put into this project and glad that we succeeded our goals with a fully-trained DDPG.

5.4 Declarations

I declare that all code submitted was 100% written by my group. There was no form of copy-pasting or dishonesty. We received a lot of help from YouTube videos, blog posts & resources from ME5406, but nothing was copied or used as a starting place for more development. Even the SpiderBot was designed by us from scratch. With regards to the environment, we had the help of the physics engine and PyBullet library to interface with our physical model. For grading purposes, I declare that my group mate, Chong Yu Quan (A0136286Y) and I contributed 50%-50% each to this project