DDPG Bipedal Walker

Harteley Sebastian
Department of Computer Science
University of Bath
Bath, BA2 7AY
hs2397@bath.ac.uk

Nattharika Sae Tang
Department of Computer Science
University of Bath
Bath, BA2 7AY
nst34@bath.ac.uk

Nisarnart Sinsuesatkul
Department of Computer Science
University of Bath
Bath, BA2 7AY
ns2176@bath.ac.uk

Saethawut Rochanavibhata
Department of Computer Science
University of Bath
Bath, BA2 7AY
sr2380@bath.ac.uk

Tzu-Ying Wang
Department of Computer Science
University of Bath
Bath, BA2 7AY
tyw47@bath.ac.uk

Yuk Ting Lam
Department of Computer Science
University of Bath
Bath, BA2 7AY
ytl81@bath.ac.uk

1 Problem Definition

The problem is 'BipedalWalker-v3', and the main task is to teach the bipedal walker to walk through rough terrain. The bipedal walking robots are a model of a humanoid robot that has sparked a great deal of interest in the past few decades[1], which is a challenging task as our agent needs to run fast without tripping itself, leading to a massive drop in rewards. Below is the discussion of states, actions, transition dynamics, and the reward function for this problem. More details of the problem domains can be found in 7

States: The size of the state space for this problem is 24, which is continuous. States consist of several parameters, including hull angle speed, angular velocity, horizontal speed, vertical speed, the position of joints and joints angular speed, legs contact with the ground, and ten lidar rangefinder measurements. There are no coordinates in the state vector. Figure 1 shows the list of observation spaces.

Actions: The size of the action space for this problem is 4. Figure 2 illustrates that the BipedalWalker has two legs, and each leg has two joints. These joints are hips and knees, each with the same action space in the range [-1, 1] as seen in Figure 3.

Transitions Dynamics: The agent terminates and restarts when the robot touches the ground. Since the environment is stochastic, the rewards are not specified to a set of numbers. The rewards are calculated based on the torque or how much the robot moves.

Reward Function: The maximum earning a reward is 300 points which increase proportionally to the distance the bipedal walker walks on the terrain. There are two negative rewards; one is proportional to the torque applied to the joint, and another is -100 for tumbling. Therefore, it is optimal to walk with minimal torque and avoid falling when the hull contacts the ground.

Num	Observation	Min	Max	Mean
0	hull_angle	0	2*pi	0.5
1	hull_angularVelocity	-inf	+inf	-
2	vel_x	-1	+1	-
3	vel_y	-1	+1	-
4	hip_joint_1_angle	-inf	+inf	-
5	hip_joint_1_speed	-inf	+inf	-
6	knee_joint_1_angle	-inf	+inf	-
7	knee_joint_1_speed	-inf	+inf	-
8	leg_1_ground_contact_flag	0	1	-
9	hip_joint_2_angle	-inf	+inf	-
10	hip_joint_2_speed	-inf	+inf	-
11	knee_joint_2_angle	-inf	+inf	-
12	knee_joint_2_speed	-inf	+inf	-
13	leg_2_ground_contact_flag	0	1	-
14-23	10 lidar readings	-inf	+inf	-

Figure 1: The bipedal walker's observation space [5].

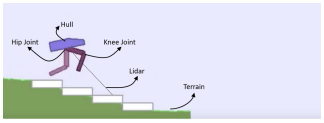

Figure 2: The bipedal walker's elements [5].

2 Background

Deep Q Networks (DQN), introduced in [3], are neural networks that utilise Q learning. Instead of having the Q function as a table, the DQN estimates Q-values by training the neural network with some parameters from experienced memory. The strengths of this algorithm are experience replay and the periodically updated target. Experience replay is the mechanism to store all the agent's experiences at each time step in a buffer called replay memory. The replay memory contains experience tuples of all episodes. It also correlates less with the target by adjusting the Q-value towards target values that are only periodically updated. In addition, it takes a long time to converge since Q-learning is an off-policy learner where the updated policy is not the same as the behaviour policy. With this constraint and the fact that the DQN agent is only compatible in the environment with a discrete action space, this method is not fully applicable to this problem. However, the strengths of DQN are worth exploring and considering as the core idea to solve this problem.

Deterministic Policy Gradient, or DPG, is the expected gradient of the action-value function, which can be estimated much more efficiently than the usual stochastic policy gradient [6]. According to David Silver [6], DPG uses the Actor-Critic method as its algorithm. The algorithm works by having an actor that updates the policy and a Critic that approximates the true action-state function by learning a set of parameters. One of the DPG's advantages is handling the continuous action space problems. It is simpler and can be computed more efficiently than the stochastic policy resulting in a more robust convergence than the value-based methods. However, this method has no target networks leading to the lower stability and converging on a local maximum rather than the global optimum [7].

Table 1: An example table

Method	Strength	Weakness
DQN	• Experience Replay • Updated Target	• Slowly converge • For discrete action space
DPG	• For continuous action space • Stronger convergence rate	• Lack of fixed target networks • Converge on a local maximum

Num	Name	Min	Max
0	Hip_1 (Torque / Velocity)	-1	+1
1	Knee_1 (Torque / Velocity)	-1	+1
2	Hip_2 (Torque / Velocity)	-1	+1
3	Knee_2 (Torque / Velocity)	-1	+1

Figure 3: The bipedal walker's action space [5].

From the above background, we see that both may not be the perfect solution for our chosen problem. Instead, the better way is to use the Deep Deterministic Policy Gradient method (DDPG). It combines the strength of both as it uses ideas from value-based and policy-based RL methods to increase the stability of the model and reduce variance.

3 Method

The authors in [2] introduced Deep Deterministic Policy Gradient or DDPG, which combines the ideas of DPG and DQN to make it more resilient and efficient. DDPG is a model-free off-policy actor-critic algorithm for learning continuous actions that Q-Learning cannot handle. It applies the Actor-Critic algorithm to Q-Learning, allowing DDPG to tackle continuous action spaces. Similar to DQN, there are two core mechanisms with a slight adaptation in DDPG: replay memory and the periodically updated target network. DDPG does not learn from the most recent state transition that the agent experiences for the replay memory. It keeps track of the total experiences and random samples from them at each timestep to get batches of memories and update the networks' weights. DDPG has two target networks since it is a type of Actor-Critic method. Each target network determines the value of the action specified by online networks, which are actor and critic networks. The values from target networks are then used to update the weights of the deep neural network. Therefore, this method involves four networks: actor, target actor, critic, and target critic networks.

The authors in [2] also provided the DDPG algorithm as seen in Figure 4. To initialise the actor and critic network, we set the number of nodes in fully connected layers to be 400, and 300 with the ReLU activation function as they recommended. While the actor network's output channel number is set according to the action number using tanh as the activation function, the critic network's final layer has only one node. The learning rate for actor and critic networks, alpha=0.0001 and beta=0.001, respectively, is set according to the suggestion. This algorithm has four primary points: the update rule for the actor network, the update rule for the critic network, and the target networks with the soft copy concept.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

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

    Update the target networks:

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

  end for
end for

Figure 4: DDPG pseudocode

The update rule for actor network

Actor network decides what to do based on the current state by outputting the action values. These values are deterministic as they will produce the same actions if the same states are passed into the network. The algorithm faces the explore-exploit dilemma, which forces the agent to take the off-policy action when obtaining the same action value repeatedly. Therefore, some random Gaussian noises N_t will be added to the actor network μ so that it can avoid this dilemma and explore more, as seen in Formula 1. Formula 2 is the update rule for the actor where J is the cost function, θ, μ, and Q denotes the parameter, actor, and critic respectively. First, it randomly samples states from memory that consists of S_t, A_t, R_t, S_{t+1} and then uses the actor to determine actions that it should take according to those states. Those actions and states from memory are then plugged into the critic function to get the Q value. Finally, we take the gradient of that value with respect to the actor network parameter.

$$a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t \tag{1}$$

$$\nabla\theta\mu J \approx \mathbb{E}_{s_t\ \rho\beta}[\nabla_\theta Q(s, a|\theta^Q|_{s=s_t, a=\mu(s_t|\theta^\mu)}] \tag{2}$$

$$= \mathbb{E}_{s_t\ \rho\beta}[\nabla_\theta Q(s, a|\theta^Q|_{s=s_t, a=\mu(s_t)\nabla_{\theta\,\mu}\mu(s|\theta^\mu)|_{s=s_t}}] \tag{3}$$

The update rule for critic network

Critic network evaluates the state and action pairs, whether that pair is good or bad. Formula 4 is the loss function essential in the critic's updating rule. y_i, $\mu^{'}$, $Q^{'}$ are target value, target actor, and target critic subsequently. First, it randomly samples states, new states, actions, and rewards. Then it uses the target actor network to determine the actions of new states. Those actions are substituted into the target critic network and multiplied those outputs with a discount factor before adding the reward sampled from that timestep to obtain the target value as seen in Formula 5. Lastly, current states and actions are plugged into the critic network to use its result to find the Mean Squared Error (MSE) between it and the target value.

$$L = \frac{1}{N}\sigma_i(y_i - Q(s_i, a_i|\theta^Q))^2 \tag{4}$$

$$y_i = r_i + \gamma Q^{'}(s_{i+1}, \mu^{'}(s_{i+1}|\theta^{\mu'})|\theta^{Q'}) \tag{5}$$

The update rule for target networks To update the target networks, we first randomly initialise actor network $\mu(s|\theta^\mu)$ and critic network $Q(s, a|\theta^Q)$ with random parameters (θ^Q and θ^μ). Then directly copy those parameters to the target actor and target critic network for the first time only. Then apply the soft copy concept where a constant of around 0.001 or τ is multiplied by the parameters to update the target network as seen in Formula 6 and 7. Since updating online and target networks are the structure of DDPG, it utilises the concept of soft copy from online to target networks where each timestep, the weights get updated with the stable learning.

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'} \tag{6}$$

$$\theta^{\mu'} \leftarrow +(1-\tau)\theta^{\mu'} \tag{7}$$

There are three reasons for choosing DDPG to train the bipedal walker. First, DDPG is a type of actor-critic method that allows it to store the memory as a buffer. The actor updates the policy distribution by following the critic's suggestions, such as policy gradients. The actor and critic networks randomly sample mini-batches of the transition from the replay buffer. The replay memory is helpful since they do not learn from the recent state transition that the agent experiences but from the total experiences to update the weights of their networks. The second is having the periodically updated target for both actor and critic networks. Therefore, two target networks are updated by soft copying the ctor and critic networks. Those target networks calculate the values of actions that are later used to update the networks' weights. The advantage of the target networks is that it provides a stable learning rate. Last, extending the Q-learning from discrete to continuous action spaces works well with this bipedal walker problem. The details of work experiment for the proposed DDPG agent are explained in 7 and the hyperparameter tuning in 7

4 Results

The results are categorised based on the DDPG agent, random agent, and DDPG agent with some hyperparameter tuning. Firstly, the DDPG agent learned for 10,000 episodes, and the average score of each 100 is shown as a graph in Figure 5. It can be seen that the highest average score the agent can reach is 210.4, whereas the highest score is 298.4 at the 8180th episode from Figure 6. After training for 10,000 episodes, the video shows that the bipedal walker mincing walk fastly can reach the goal in the end.

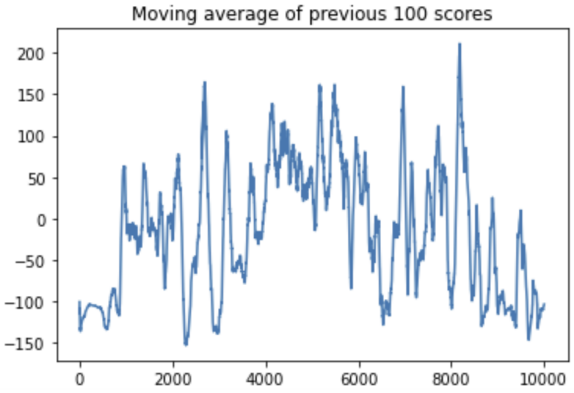

Figure 5: Average previous 100 scores of proposed DDPG

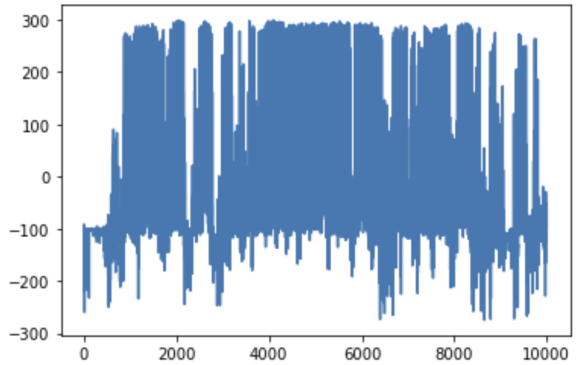

Figure 6: Scores from proposed DDPG

Secondly, the agent that chooses actions randomly manages to walk and gain the highest 100-average score of -83.3 for 10,000 episodes, as seen in Figure 7. From Figure 8, the random agent's highest score is -74.1, which comes from around the 30th episode. As in the video of the bipedal walker using a random agent, the bipedal walker can barely walk and fall before reaching the end.

Finally, two models were developed from the original DDPG using different numbers of nodes in the fully connected layers. These models run for 5000 episodes just for further discussion. According to Figure 9, the greatest average score of changing to 256 and 128 nodes in the first and second fully connected layers is 165.2. For another model, the experiment reduces the number of fully connected layers to one with 256 nodes. As shown in Figure 10, the highest average score of this model is -38.9. It is also worth mentioning the highest average score of the original DDPG agent that learned for the same 5000 episodes, which is 152.1 for further discussion.

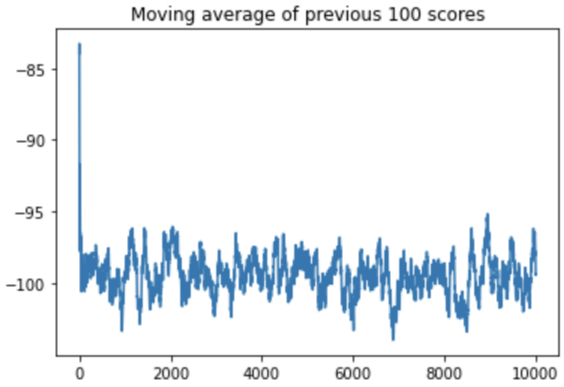

Figure 7: Average previous 100 scores of random agent

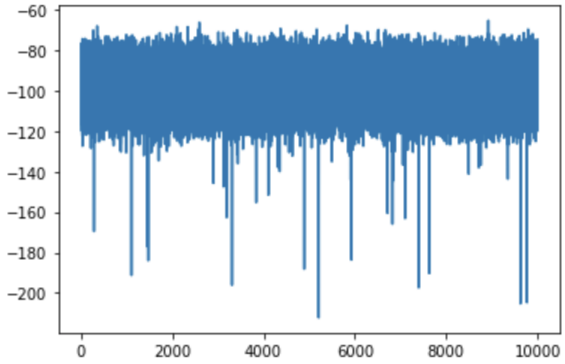

Figure 8: Scores from random agent

5 Discussion

This section will discuss the results from the above paragraph by first comparing the DDPG agent with the random agent and then analysing the differences after hyperparameters tuning. First, the random agent does not learn anything and always makes a random walk, leading to inconsistency in learning. The average score from the random agent never reached positive, while the average score of the DDPG agent did. The average scores from DDPG show that the agent learns to walk at a slow rate as it reaches the positive average at around the 1000th episode, as seen in Figure A1. Though the average score can only reach 210.4, it can learn to walk without falling down or almost reach the score of 300 at some episodes. It is sensible since the DDPG agent learns to converge slowly like the DQN agent. To evaluate the DDPG method in the bipedal walker problem, the agent learns to walk till the end, unlike the random one. The result is satisfactory since the bipedal walker uses the optimal walk to avoid moving the joints as much as possible. Second, compare the results of average scores from three models after training for 5000 episodes: the original DDPG, the DDPG of two fully connected layers with 256 and 128 nodes, and the DDPG of one dense layer with one dense layer 256 nodes. The results show that the two dense layers with 256 and 128 nodes DDPG produces the best average score at 165.2. The assumption that this model is better than others is that it has less complex neural networks. In addition, the experiment using only one layer with 256 nodes confirms that a too simple model will not produce the desired result.

6 Future Work

Our implemented DDPG agent on this project can't break the 300 score mark, but we aim to do that next. There are two potential tasks that we would like to perform. First is the extension of the original DDPG, which uses an adaptive parameter noise introduced by [4] instead of the random gaussian action noise. The learning curve of using the new type of noise would be better than the

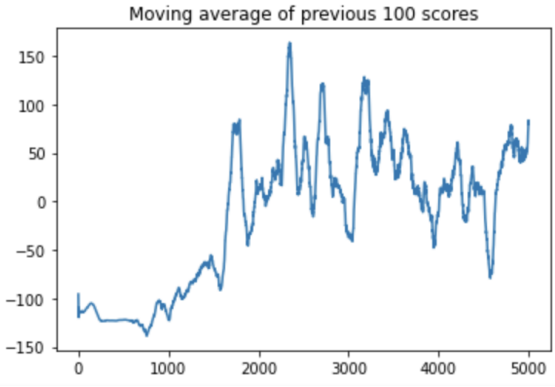

Figure 9: Average previous 100 scores of DDPG with 256 and 128 nodes

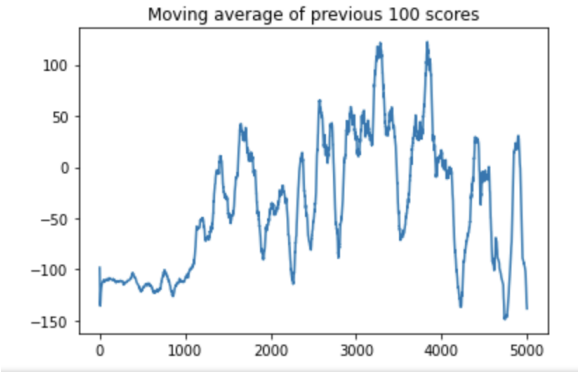

Figure 10: Average previous 100 scores of DDPG with 256 nodes

current one as it allows the model to explore the environment more. It is more effective as it is now correlated to the action space and its weights. The second is to try another method: Continuous Proximal Policy Optimization (PPO). From the article of PyLessons [5], after training the agent for 100 episodes, an average score exceeds 300. Although it is harder to implement the PPO, it is worth trying after knowing the results. PPO agent selects action randomly based on its estimated probability distribution. It uses the current policy to interact with the environment for multiple steps, then uses mini-batches to update actor-critic properties over multiple epochs. We believe it can match or outperform DDPG because it uses a stochastic policy. The policy's objective function is based on sampling from trajectories from a probability distribution that depends on the current policy. It will have more guarantee for convergence but, at the same time, take a longer time for training as it is not deterministic when choosing from the action space.

7 Personal Experience

The result of DDPG is surprisingly unstable. The reward fluctuates and almost reaches 300, but it could suddenly drop to a negative score. We tried to run DDPG for 10000 episodes, and it took us 1-week to finish the runtime. This long runtime also applies to other models where we run for 5000 episodes, and it took us around 24-hour. In addition, having to run the code for an extended period is our main difficulty as it will take us longer to find bugs, and we would have less time to improve the performance and discuss the result. However, in the end, our bipedal walker can smoothly walk and is much better than we expected.

References

[1] Arun Kumar, Navneet Paul, and SN Omkar. Bipedal walking robot using deep deterministic policy gradient. *arXiv preprint arXiv:1807.05924*, 2018.

[2] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2015.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[4] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter space noise for exploration. *arXiv preprint arXiv:1706.01905*, 2017.

[5] PyLessons. Bipedalwalker-v3 with continuous proximal policy optimization. 2020.

[6] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. PMLR, 2014.

[7] Thomas Simonini. An introduction to policy gradients with cartpole and doom, 2018.

Appendices

Appendix A: Details of problem domains

The 'BipedalWalker-v3', the selected problem, is a challenging task. We have to teach the robot to walk with its legs. The bipedal walker has two legs, each with two joints. The main goal of this problem is to let the robot walk until the end to get the reward of 300 points without falling and move the joints as less as possible to avoid getting the penalties. There are two versions of the bipedal walker's environment: the normal and the hardcore. The normal version, our selected problem, is the normal uneven terrain. The rough ground forces the robot to balance itself, which is the cause of the movement and the penalty the robot will get. The hardcore version adds ladders, stumps, and pitfalls, making it more difficult to balance and walk until the end. More descriptions of the problem domains will be explained below.

States: The state consists of 24 parameters which are:

1. Hull angle speed which ranges from 0 to 2 $\times$ the angle size
2. Angular velocity from $-\infty$ to ∞
3. Horizontal speed or the velocity on the x-axis from -1 to 1
4. Vertical speed or the velocity on the y-axis from -1 to 1
5. Position of joints and joints angular speed, which are two knees and two hips from $-\infty$ to ∞
6. Legs contact with the ground is a flag of 0 and 1 for contacting and not contacting the ground
7. 10 lidar rangefinder measurements to help the robot understand the next states. This ranges from $-\infty$ to ∞ as well.

Actions: There are four actions for this problem which are the first hip, the second hip, the first knee, and the second knee. These actions are calculated based on either controlling the torque(default) or the velocity, therefore the lower bound and upper bound of the action are -1 and 1 respectively.

Reward Function: The maximum earning reward is 300 points. The rewards increase proportionally to the distance the bipedal walker walks on the terrain. There are two negative rewards; one is proportional to the torque applied to the joint, and another is -100 for tumbling. Therefore, it is optimal to walk with minimal torque and avoid falling when the hull contacts the ground.

Appendix B: Experimental details

The DDPG model is created in a Jupyter notebook applying object oriented scheme to create classes and functions. The model also uses the tensorflow for the neural network parts. All parameters are set based on the suggestion of a successful DDPG model from [2]. This model consists of four classes: Critic network, Actor network, Buffer, and Agent classes.

The Critic network class is built based on the Keras TensorFlow model. The initial values are the number of nodes for the first and second fully connected layers. This class contains a call function that takes inputs as the state and action to determine the action value after going through the neural networks. After each dense layer, the activation layer uses the ReLU activation function. The action value is then used to calculate the Q-value. There is only one output or the Q-value from the last dense layer of this network. The Critic and target Critic networks are replicated from this class.

The Actor network class also consists of the number of neural nodes for the first and second dense layers. Another initial value for this network is the number of actions. The fully connected layers have the same properties as the Critic network except for the last layer, where the Actor network uses the tanh activation function to output the actions.

The buffer class is used to store the different values including:

1. States
2. Actions
3. Rewards
4. Next states

The transition function in this class is used to update the latest value when our agent performs an action. Using the modulo operator, we can update the latest values to replace the oldest memory in our replay buffer and continue to do this without overloading the memory size. The sample function in this class is used to sample randomly from the replay buffer using a self-defined batch size.

The agent class is where we initialise the networks and all the mechanisms behind the DDPG method. The initial value for the agent class consists of:

1. Input shape
2. Learning rates
3. Environment
4. Number of actions
5. Discount factor
6. Constant for soft copying (τ)
7. Number of neural nodes
8. Batch size
9. Initial noise

The input shape and number of actions are required to generate the replay buffer. The number of actions, minimum action, and maximum action can be retrieved from the environment. The learning rates, alpha and beta for the Actor and Critic networks, are set according to the suggestion, which are 0.0001 and 0.001, respectively. The discount factor will be included in the calculation of the Critic network. The constant (τ), which is a tiny number of 0.001, is a part of parameter updating. The number of actions and nodes are the required parameters for generating the networks. The number of nodes used in the default DDPG is 400 and 300 for the first and second fully connected layers. The main functions in this class are the updateParameters, chooseAction, and learn functions. The update function illustrates the concept of soft update. Soft copying occurs after the first directly copying of the weights. The chooseAction function takes the states as the inputs and selects the actions based on the given states. However, these actions are added by the random noises with an initial standard deviation of 0.1. This random noise is Gaussian distributed and is added to the action to solve the explore and exploit dilemma problem. The learn function consists of the gradient for the critic and

actor networks. The Critic loss is calculated using the mean square error. These numbers can be calculated using the help of Keras. The batch size for this DDPG is set to 64.

For the DDPG agent, the game was trained for 10,000 episodes. Inside each loop, the best average score of the previous 100 episodes are stored in the array and the actual score. The best average score is observed and updated to the checkpoint in order to save the model.

For the random agent, the game was trained for the same 10,000 episodes as the DDPG agent, storing the numerical data in the later plot as graphs for comparison. The random agent walks in the random action as the action is chosen from the uniform number from -1 to 1.

Appendix C: Hyperparameter tuning details

First Trial: We tried experimenting with a few tuning on the hyperparameter. Since we are trying to find the best hyperparameter to use in our case, we start with only running 2000 episodes. The first few trials we use for the number of neuron nodes in fully connected layer 1 and layer 2 are 400 and 300 (Figure 11), 512 and 512 (Figure 12), and 256 and 256 (Figure 13).

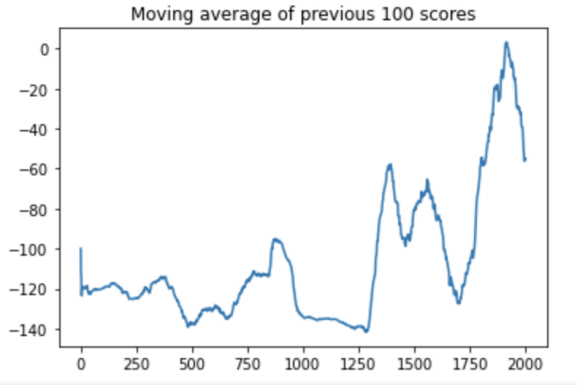

Figure 11: 2000 episode, fc1=512, fc2=512

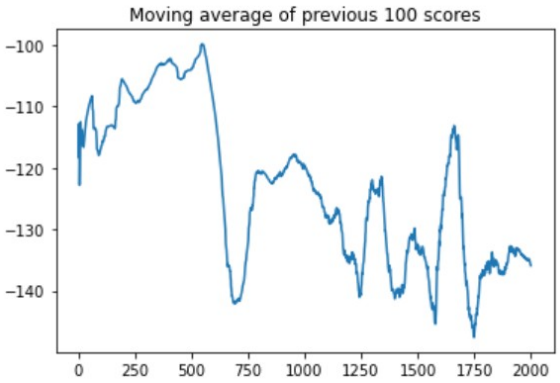

Figure 12: 2000 episode, fc1=512, fc2=512

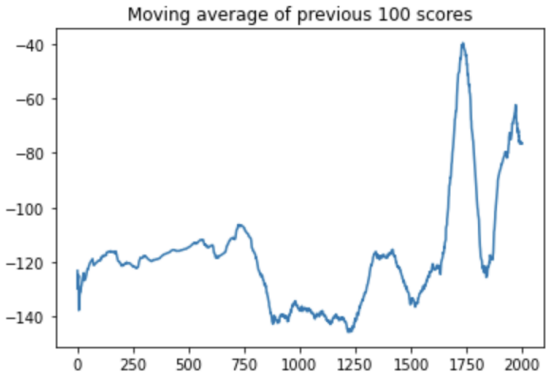

Figure 13: 2000 episode, fc1=256, fc2=256

Second Trial: From the results we were getting, we decided to continue doing 5000 episodes with the same hyperparameter tuning from the first trial, except for the tuning to 512 and 512 , since its average score never reached more than -100. Therefore, we tried training two models where the results are shown in Figure 10 and 14

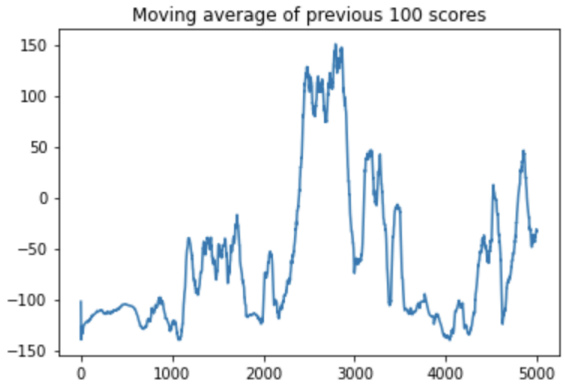

Figure 14: 5000 episode, fc1=400, fc2=300

Final trial: Looking from the results we were getting, we could see that the average score of hyperparameter tuning for fully connected layer 1 = 400, and fully connected layer 2 = 300 can reach more than 150 (Figure 5), while the other (fc1 = 256, fc2 =256) did not reach 150. So we decided to use this hyperparameter to run 10,000 episodes.

Extra attempt: Since we have more time and device to attempt another hyperparameter tuning to run for 5,000 episodes, we did try using 256 for the fully connected layer 1 and 128 for the fully connected layer 2 (Figure 15). We got better results than the one we ended up using. However, we did not have the time to run 10,000 episodes for this hyperparameter.

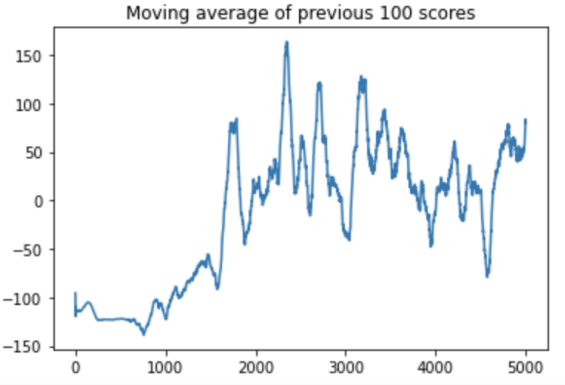

Figure 15: 5000 episode, fc1=256, fc2=128