Bipedal Robot Walking on Complex Terrain using Soft Actor-Critic (SAC) Algorithm

ME5406 Deep Learning for Robotics Project II

Jia Yansong
A0263119H
jiayansong@u.nus.edu

Department of Mechanical Engineering
National University of Singapore
Singapore
2023.4.20

Contents

1 Introduction **1**

2 Environment Description **1**
2.1 State Space . . . 2
2.2 Action Space . . . 3
2.3 Reward Structure . . . 3
2.3.1 Default Reward Structure . . . 3
2.3.2 Refined Reward Structure . . . 3

3 Method **3**
3.1 Soft Actor-Critic . . . 3
3.1.1 Entropy-Regularized Reinforcement Learning . . . 3
3.1.2 Soft Actor-Critic . . . 4
3.1.3 Learning Q . . . 5
3.1.4 Learning the Policy . . . 5
3.2 First-In-First-Out (FIFO) Replay Buffer . . . 5

4 Experiments and Results **5**
4.1 Random Seed Environment . . . 6
4.1.1 Batch Size = 100 . . . 6
4.1.1.1 Training Results . . . 6
4.1.1.2 Testing Results . . . 6
4.1.2 Batch Size = 256 . . . 7
4.1.2.1 Training Results . . . 7
4.1.2.2 Testing Results . . . 7
4.2 Deterministic Seed Environment . . . 7
4.2.1 Seed = 0 . . . 8
4.2.1.1 Training Results . . . 8
4.2.1.2 Testing Results . . . 8
4.2.2 Seed = 2 . . . 9
4.2.2.1 Training Results . . . 9
4.2.2.2 Testing Results . . . 9

5 Compared with Conventional Method **10**
5.1 Conventional Method . . . 10
5.2 Advantages of Reinforcement Learning Methods . . . 10

6 Conclusion **10**
6.1 Limitation . . . 10
6.2 Future Work . . . 10

Abstract

This project is for NUS ME5406 Deep Learning for Robotics Part II. This is a group project to train the Bipedal Walker in OpenAI Gym[1] using different deep reinforcement learning methods. In this paper, Soft Actor-Critic (SAC)[2] reinforcement learning algorithm is used to train the bipedal robot (agent) within the complex road environment built by OpenAI Gym[1] and make the robot able to walk through the complex road environment. Several training and test experiment results with different hyperparameters of SAC[2] are shown in this paper. More experiments with different sets of hyperparameters can be tested following the instruction of *README.md* file or on GitHub: https://github.com/JYS997760473/NUS-ME5406-Project2 (Not public yet).

1 Introduction

A bipedal robot is a type of robot that has two legs and is designed to walk on two feet, just like a human. Bipedal robots are used in various applications, from industrial automation and manufacturing to space exploration and search and rescue missions. They are often designed to mimic the movements and balance of a human, and some are even capable of running, jumping, and performing other complex actions. Bipedal robots can be autonomous, meaning they can operate on their own without human intervention, or they can be teleoperated, meaning they are controlled by a human operator.

Bipedal robots can be used for emergency rescue, disaster relief, exploration, and other tasks. In the real world, bipedal robots must move in complex and unstable environments, such as uneven ground, slopes, steps, dust, gravel-covered ground, etc., which may cause the robot to lose balance and fall. However, traditional control methods have difficulties in dealing with complex nonlinear systems. We mainly try to train the walking control algorithm through deep reinforcement learning, which can make the robot realize stable walking in an unstable environment, so as to play a role in various practical application scenarios.

In this project, we set up a bipedal robot (with 2 degrees of freedom for each leg) and set up a scene with a ladder, stumps, and traps. The bipedal robot will maintain its balance and continue to move forward in this scene. In the event that the robot experiences a fall while traversing the road, it will be regarded as a failure to achieve the objectives of the current mission. Conversely, if the robot successfully reaches the designated right endpoint within the environment, it will be deemed as a successful episode. The bipedal robot model used in this project is shown in Figure 1.

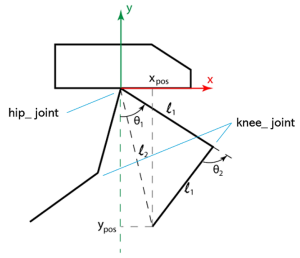


Figure 1: Model of a biped robot

In this project, Soft Actor-Critic (SAC)[2] reinforcement learning algorithm is used to train the bipedal robot (agent) within the complex road environment built by OpenAI Gym[1], and finally, we tested the trained model and found that the bipedal robot is capable of walking through the complex road environment. Several experiments with different hyperparameters are shown in this paper.

2 Environment Description

The environment is based on the OpenAI Gym[1] Bipedal Walker environment. One example frame of the environment is shown in Figure 2. From Figure 2 we can see that there are ladders, stumps, and pitfalls in the environment, which make the terrain complex.

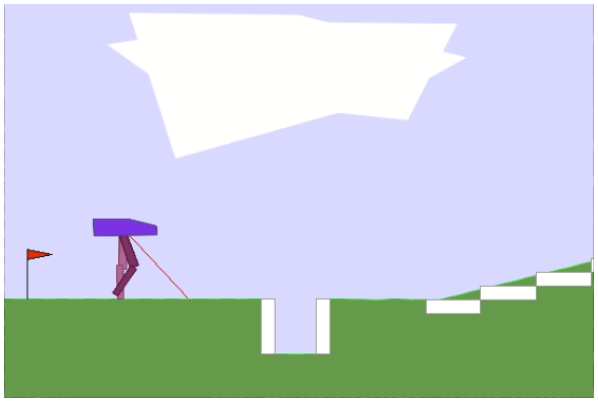


Figure 2: One example frame of the environment

2.1 State Space

The state is a 1×24 vector, which is shown in Table 1. The state space of the Bipedal Walker mainly includes the velocity of the base of the robot in the x and y directions, which gives an indication of the robot's overall speed and direction, the angular velocity of the robot's body, which gives an indication of how much it is tilting or rotating, the angles of the joints of the robot, which include the hip, knee, indicating the current position of the joints and give an indication of the robot's current posture. and the angular velocities of the joints of the robot, which indicate how fast the joints are moving.

Table 1: State Space

Index	State	Min Value	Max Value
0	hull angle	0	2π
1	hull angular velocity	-Inf	Inf
2	velocity x	-1	1
3	velocity 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	lidar readings	-Inf	Inf

2.2 Action Space

The action is a 1×4 continuous vector, whose each element represents the force applied to a specific joint of the robot, containing every action $\frac{Torque}{Velocity}$, which is shown in Table 2.

Table 2: Action Space

Index	Action
0	hip1
1	knee1
2	hip2
3	knee2

The first number corresponds to the force applied to the robot's hip joint in the x-direction. The second number corresponds to the force applied to the robot's knee joint in the x-direction. The third number corresponds to the force applied to the robot's hip joint in the y-direction. The fourth number corresponds to the force applied to the robot's knee joint in the y-direction.

2.3 Reward Structure

2.3.1 Default Reward Structure

Each fall is awarded -100 points, a small number of negative points for driving the joint to rotate, and positive points for moving forward. A total of 300 points is enough to win.

2.3.2 Refined Reward Structure

The default reward structure is from OpenAI Gym[1], in order to expedite the convergence of the training process in this project, each instance of failure has been assigned a reward of -10, while all other rewards have been increased by a factor of 10 from their original values.

3 Method

3.1 Soft Actor-Critic

Soft Actor-Critic (SAC)[2] is a reinforcement learning algorithm that optimizes a stochastic policy in an off-policy manner, which serves as a connection between stochastic policy optimization and approaches similar to DDPG[3]. Although SAC was published around the same time as TD3[4] and is not a direct successor, it still uses the clipped double-Q technique found in TD3[4]. Additionally, since SAC's policy is inherently stochastic, it benefits from a technique similar to target policy smoothing.

SAC[2] relies on entropy regularization as a key component of its training process. The objective is to find a balance between the expected return and the level of entropy in the policy. The latter reflects the amount of uncertainty or randomness in the policy's decisions. This is directly related to the classic exploration-exploitation dilemma, as increasing entropy encourages the policy to explore more and take risks, which can facilitate faster learning and avoid getting trapped in suboptimal solutions. By prioritizing exploration through entropy regularization, SAC[2] can also prevent the policy from prematurely converging to suboptimal local maxima. The pseudocode of SAC[2] is shown in Figure 3.

3.1.1 Entropy-Regularized Reinforcement Learning

In entropy-regularized reinforcement learning, the agent gets a bonus reward at each time step proportional to the entropy of the policy at that timestep. This changes the RL problem to:

$$\pi^* = \arg\max_{\pi} \underset{\tau \sim \pi}{E} \left[\sum_{t=0}^{\infty} \gamma^t \Big(R(s_t, a_t, s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right) \Big) \right] \tag{1}$$

Algorithm 1 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

      Update Q-functions by one step of gradient descent using

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

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \Big(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right)\Big),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
      Update target networks with

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

    end for
  end if
until convergence

Figure 3: Pseudocode of SAC

where $\alpha > 0$ is the trade-off coefficient.

V^π is changed to include the entropy bonuses from every timestep:

$$V^\pi(s) = \underset{\tau \sim \pi}{E}\left[\sum_{t=0}^{\infty} \gamma^t \Big(R(s_t, a_t, s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right)\Big) \,\middle|\, s_0 = s\right] \tag{2}$$

Q^π is changed to include the entropy bonuses from every timestep except the first:

$$Q^\pi(s, a) = \underset{\tau \sim \pi}{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) + \alpha \sum_{t=1}^{\infty} \gamma^t H\left(\pi(\cdot|s_t)\right) \,\middle|\, s_0 = s, a_0 = a\right] \tag{3}$$

3.1.2 Soft Actor-Critic

SAC[2] is a reinforcement learning algorithm that simultaneously learns a policy π_θ and two Q-functions Q_{ϕ_1} and Q_{ϕ_2}. There are two standard versions of SAC[2]. The first version uses a fixed entropy regularization coefficient α to encourage exploration, while the second version enforces an entropy constraint by adapting α during training, and we use the latter for simplicity.

3.1.3 Learning Q

SAC[2] sets up the MSBE loss for each Q-function using this kind of sample approximation for the target. The loss functions for the Q-networks in SAC[2] are:

$$L(\phi_i, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}} \left[\left(Q_{\phi_i}(s,a) - y(r,s',d) \right)^2 \right] \tag{4}$$

where the target is given by

$$y(r,s',d) = r + \gamma(1-d) \left(\min_{j=1,2} Q_{\phi_{\mathrm{targ},j}}(s',\tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s') \right), \qquad \tilde{a}' \sim \pi_\theta(\cdot|s') \tag{5}$$

3.1.4 Learning the Policy

To optimize the policy, we employ the reparameterization trick, which involves generating a sample from the policy distribution $\pi_\theta(\cdot|s)$ by computing a deterministic function of the state, policy parameters, and independent noise. Specifically, we use a squashed Gaussian policy, where the samples are obtained by applying a squashing function to a Gaussian distribution:

$$\tilde{a}_\theta(s,\xi) = \tanh\left(\mu_\theta(s) + \sigma_\theta(s) \odot \xi\right), \qquad \xi \sim \mathcal{N}(0, I) \tag{6}$$

We can use the reparameterization trick to transform the expectation over actions into an equivalent expression in terms of the expectation over the noise, thereby enabling us to calculate the integral using a simpler distribution and facilitating optimization using stochastic gradient descent:

$$\underset{a\sim\pi_\theta}{E}\left[Q^{\pi_\theta}(s,a) - \alpha \log \pi_\theta(a|s) \right] = \underset{\xi\sim\mathcal{N}}{E}\left[Q^{\pi_\theta}(s,\tilde{a}_\theta(s,\xi)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s,\xi)|s) \right] \tag{7}$$

The final step to compute the policy loss involves replacing the Q^{π_θ} term with a function approximator, and in the case of SAC[2], the minimum of the two Q-value function approximators Q_{ϕ_1} and Q_{ϕ_2} is used, whereas in TD3[4], only the first Q-value function approximator Q_{ϕ_1} is used. The policy is thus optimized according to

$$\max_\theta \underset{s\sim\mathcal{D}\xi\sim\mathcal{N}}{E}\left[\min_{j=1,2} Q_{\phi_j}(s,\tilde{a}_\theta(s,\xi)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s,\xi)|s) \right] \tag{8}$$

3.2 First-In-First-Out (FIFO) Replay Buffer

A Simple FIFO (First-In-First-Out) Experience Replay Buffer is a data structure used in this project. It is used to store the experiences of the agent as it interacts with the environment so that it can learn from past experiences and improve its performance over time.

The FIFO buffer is a queue of fixed size, where new experiences are added to the back of the queue, and old experiences are removed from the front of the queue. When the buffer is full and a new experience is added, the oldest experience is removed to make room for the new one.

In SAC[2], an experience is a tuple (s, a, r, s', d), where s is the current state, a is the action taken by the agent, r is the reward received, s' is the next state, and d is a flag indicating whether the episode has ended.

During training, the SAC[2] algorithm samples a batch of experiences randomly from the FIFO buffer and uses these experiences to update the actor and critic networks. By sampling experiences randomly, the algorithm can avoid overfitting to recent experiences and learn from a more diverse set of experiences.

4 Experiments and Results

There are several hyperparameters in this project and in this paper, the training and testing results of different seeds of the environment and *batch_size* are tuned and discussed in this section.

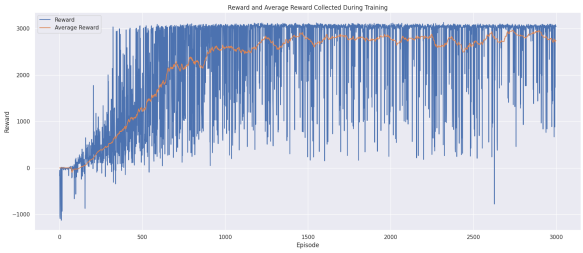
There are several similar complex terrains environment in OpenAI Gym[1], the agent is trained both in random seed environments, whose adjacent episodes during training are different, and deterministic seed environments, whose adjacent episodes during training are the same.

4.1 Random Seed Environment

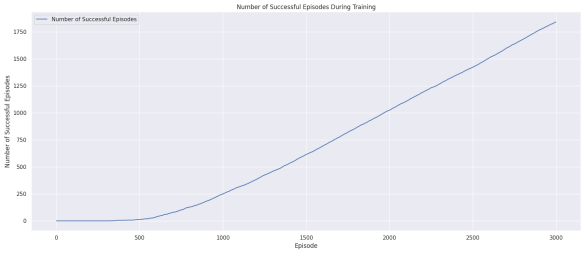
The *batch_size* is one of the hyperparameters of SAC[2], and in this part, two *batch_size*, 100 and 256 are trained and tested.

4.1.1 Batch Size = 100

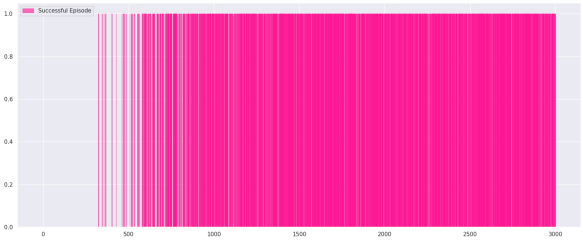
4.1.1.1 Training Results The training result is shown in Figure 4, and we can see that in the later stage of training, the policy has converged, the average reward reaches about 2800, and almost every episode is successful.



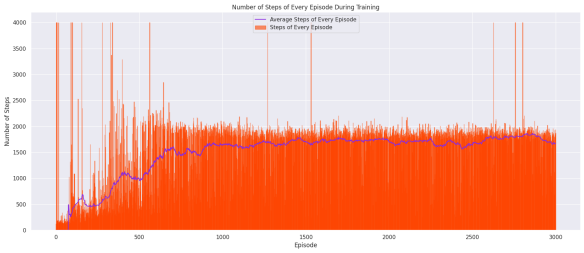
(a) Reward and average reward versus episodes during training.



(b) Number of total successful episodes during the training.



(c) Distribution of successful episodes during the training.

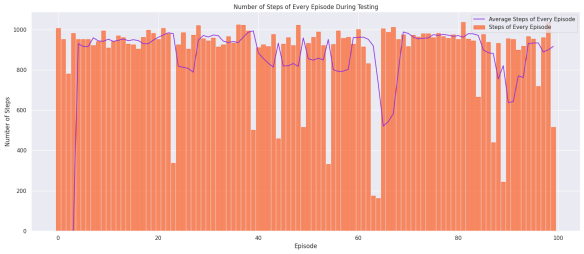


(d) Number of steps of each episode during the training.

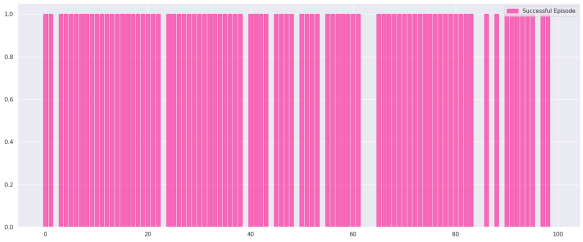
Figure 4: Training results of random seed environment with batch size = 100.

4.1.1.2 Testing Results The trained model is tested by 100 episodes, and the test result is shown in Figure 5. We can see that most of the tested episode is successful, and the average award of successful episodes is around 3000.

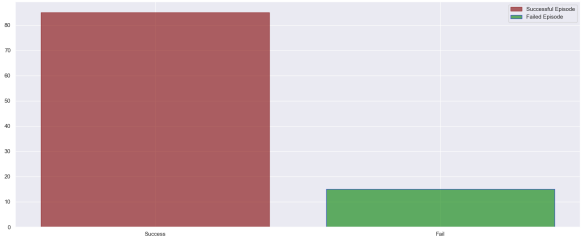
(a) Reward and average reward during the test.



(b) Number of steps and average steps during the test.



(c) Distribution of successful episodes during the test.

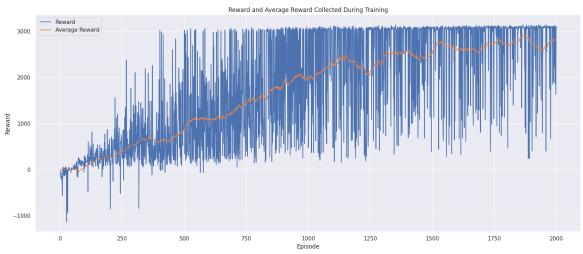


(d) Number of successful and failed episodes during the test.

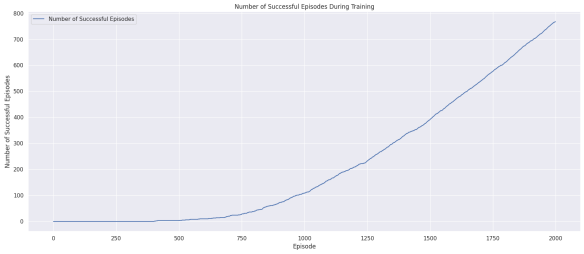
Figure 5: Test results of random seed environment with batch size = 100.

4.1.2 Batch Size = 256

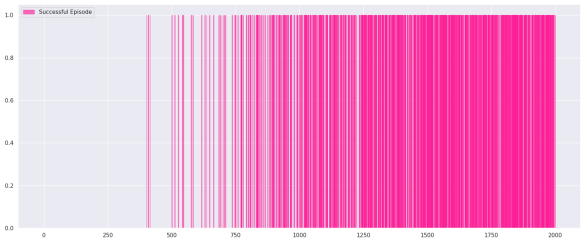
4.1.2.1 Training Results The training result is shown in Figure 6, and we can see that in the later stage of training, the policy has converged, the average reward reaches about 2700, and almost every episode is successful.



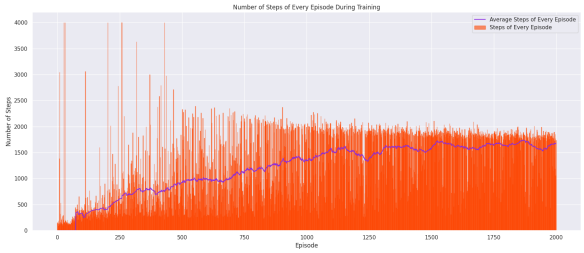
(a) Reward and average reward versus episodes during training.



(b) Number of total successful episodes during the training.



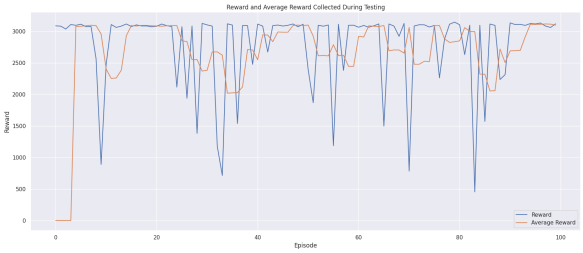
(c) Distribution of successful episodes during the training.



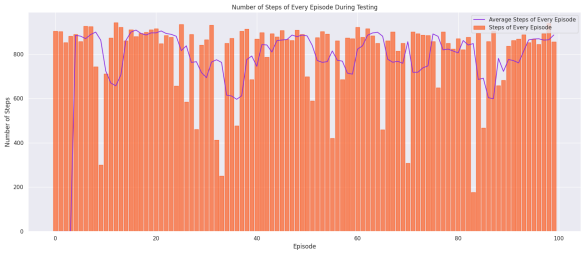
(d) Number of steps of each episode during the training.

Figure 6: Training results of random seed environment with batch size = 256

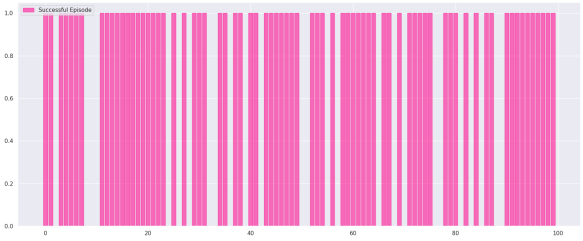
4.1.2.2 Testing Results The trained model is tested by 100 episodes, and the test result is shown in Figure 7. We can see that most of the tested episode is successful, and the average award of successful episodes is around 3000.



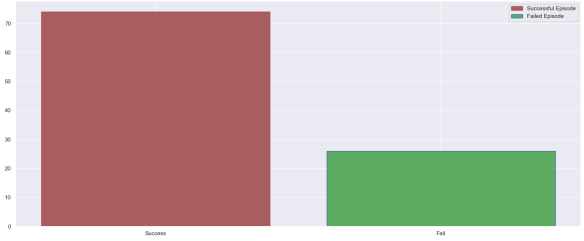
(a) Reward and average reward during the test.



(b) Number of steps and average steps during the test.



(c) Distribution of successful episodes during the test.



(d) Number of successful and failed episodes during the test.

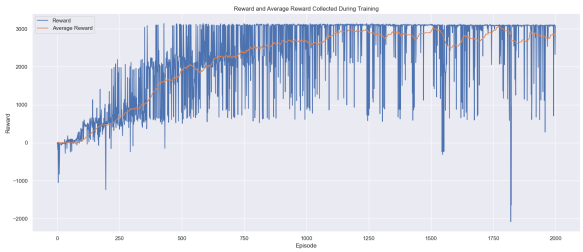
Figure 7: Test results of random seed environment with batch size = 256.

4.2 Deterministic Seed Environment

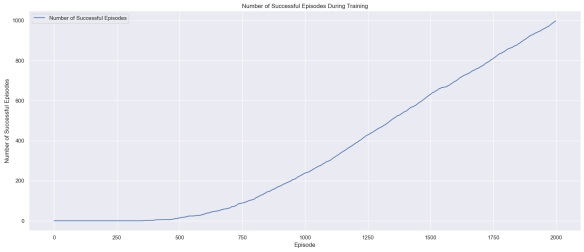
In this part, the agent is trained in several different seeds but deterministic environments (the environment keeps the same during agent training).

4.2.1 Seed = 0

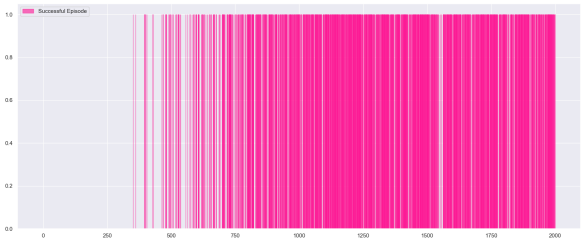
4.2.1.1 Training Results The training result is shown in Figure 8, and we can see that after 300 episodes, the agent is able to reach the goal, and the policy begins to converge. After the policy converges to the optimal policy, the average reward reaches about 2900, and almost every episode is successful.



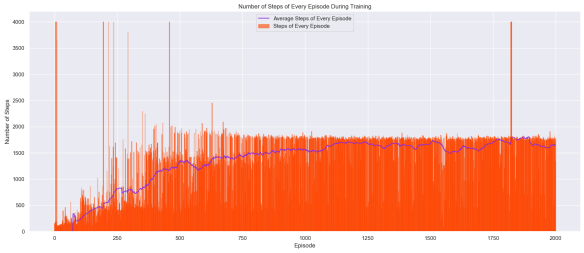
(a) Reward and average reward versus episodes during training.



(b) Number of total successful episodes during the training.



(c) Distribution of successful episodes during the training.

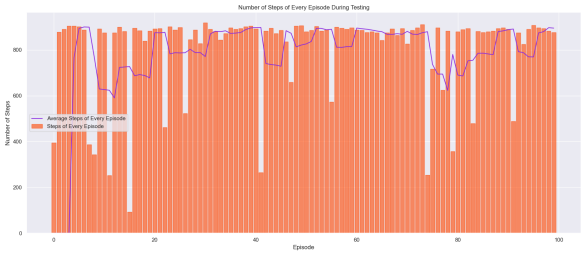


(d) Number of steps of each episode during the training.

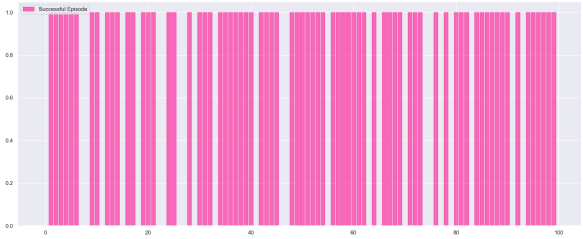
Figure 8: Training results of deterministic seed environment with seed = 0.

4.2.1.2 Testing Results The trained model is tested by 100 episodes, and the test result is shown in Figure 9. We can see that though we train the agent in the specific environment, it still can not reach the goal every episode, but most of the tested episode is successful, and the average award is around 3000.

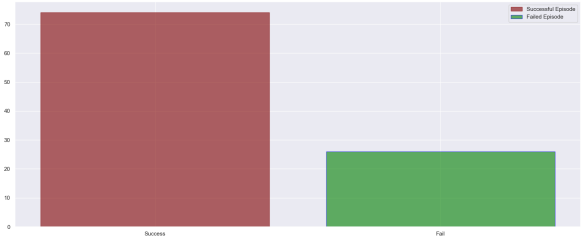
(a) Reward and average reward during the test.



(b) Number of steps and average steps during the test.



(c) Distribution of successful episodes during the test.

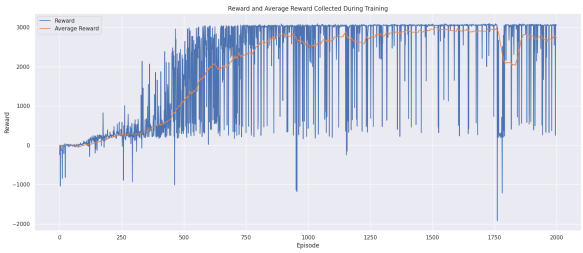


(d) Number of successful and failed episodes during the test.

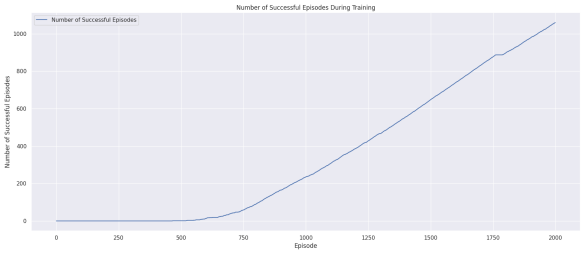
Figure 9: Test results of deterministic seed environment with seed = 0.

4.2.2 Seed = 2

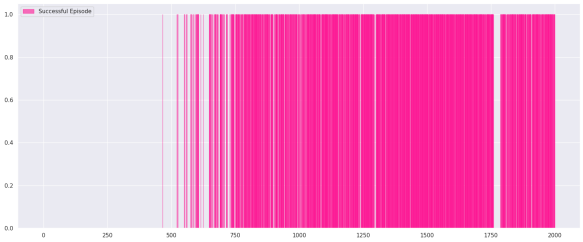
4.2.2.1 Training Results The training result is shown in Figure 10, and we can see that after about 500 episodes, the agent is able to reach the goal, and the policy begins to converge. After the policy converges to the optimal policy, the average reward reaches about 2900, and almost every episode is successful.



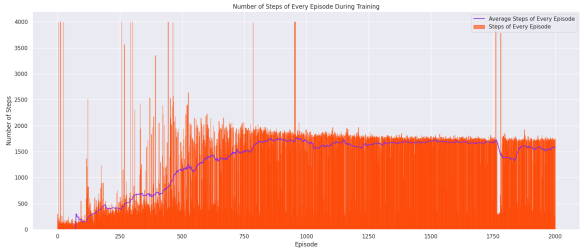
(a) Reward and average reward versus episodes during training.



(b) Number of total successful episodes during the training.



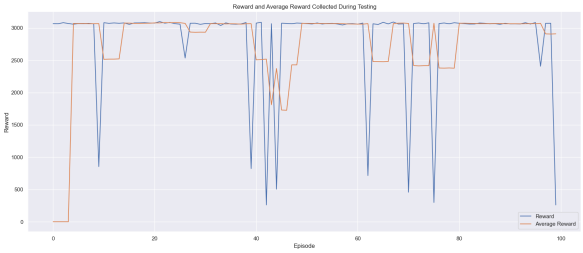
(c) Distribution of successful episodes during the training.



(d) Number of steps of each episode during the training.

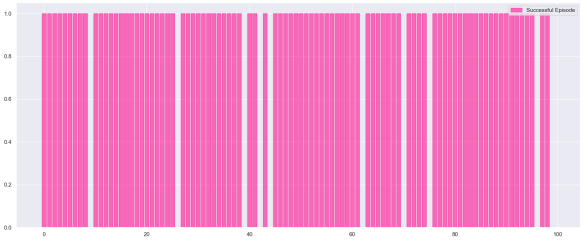
Figure 10: Training results of deterministic seed environment with seed = 2.

4.2.2.2 Testing Results The trained model is tested by 100 episodes, and the test result is shown in Figure 11. We can see that though we train the agent in the specific environment, it still can not reach the goal every episode, but most of the tested episode is successful, and the average award is around 3000.

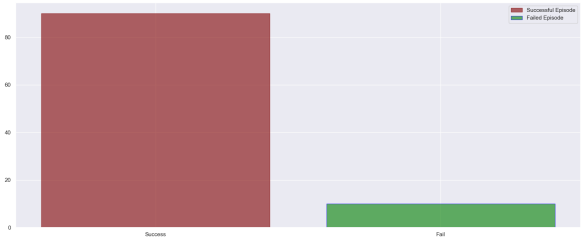


(a) Reward and average reward during the test.

(b) Number of steps and average steps during the test.



(c) Distribution of successful episodes during the test.



(d) Number of successful and failed episodes during the test.

Figure 11: Test results of deterministic seed environment with seed = 2.

5 Compared with Conventional Method

5.1 Conventional Method

Currently, the conventional method to control the walker to move is by Linear Quadratic Regulator (LQR) modern control theory. The simplified dynamic model of the bipedal walker is:

$$\ddot{x}_c = \frac{g}{h_0}(x_c - x_a) + \frac{1}{mh_0}(\tau_a - \tau_h) + \frac{F(t)}{m} \tag{9}$$

where x_c is the projection of the joint on the ground, x_a is the foot point, m is the mass of the robot, h_0 is the height of the joint, τ_a and τ_h are the torques of the robot leg, and $F(t)$ is the force applied on the joint.

LQR solves control problems by minimizing a quadratic cost function, a combination of the current state cost. After linearizing the system, we construct a discrete-time state space model and define the Q and R matrix, which are the gain of the cost function of the state vector and the gain of the cost function of the input respectively. The gain of the whole State Feedback System, K can be calculated by the Q and R matrix calculated previously. Finally, the whole robotic closed-loop system can stabilize the disturbances of the road surface, and make the robot walk stably.

5.2 Advantages of Reinforcement Learning Methods

Compared with using reinforcement learning algorithms like SAC[2], it can be quite difficult for us to establish a nonlinear system and control the system to be stable. By using reinforcement learning methods, we just need to set the reward structure and train the agent to interact with the environment and find the optimal policy, finally, the agent can move stably in the environment, so reinforcement learning methods can make the task much easier.

6 Conclusion

In this paper, Soft Actor-Critic (SAC)[2] algorithm is used to train the Bipedal Walker agent to be able to walk in complex terrain environments. The experiments with different hyperparameters and different modes (random seed or deterministic seed) of the environment are tested and it is found that all the sets of hyperparameters can train the agent successfully, and the results of the deterministic environment do not improve the test results compared with those experiments with random seed environments. More experiments can be tested following the instructions of *README.md* file or in this GitHub repository: https://github.com/JYS997760473/NUS-ME5406-Project2 (Not public yet).

6.1 Limitation

The most significant issue I am facing during the development, training, and testing of my project is the substantial amount of time it takes. On average, it takes around 5 hours to run 2000 episodes, which is a considerable amount of time.

6.2 Future Work

As mentioned in the limitation section, the most difficult problem is that the time consumption is quite large, so in the future, we should try to optimize the algorithm to decrease the time complexity to make our code more efficient.

The current version of this project is developed using the CPU version of PyTorch[5]. In the future, it would be beneficial to develop a new version based on the GPU version of PyTorch[5], to determine if it can improve the speed of the project.

References

[1] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[2] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," 2018.

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

[4] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," 2018.

[5] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.