Application of Reinforcement Learning in Ball and Plate Balance Systems

Author: Xia Wenxuan
Robotics
National University of Singapore
Student ID: A0304020H
E1373108@u.nus.edu

***Abstract*—This report tackles the classical ball and plate balance control problem using reinforcement learning techniques. A detailed physical model of the ball and plate system is constructed with PyBullet to simulate accurate physical parameters. The Soft Actor-Critic (SAC) algorithm is chosen for control, and both actor and critic network models are defined. By integrating optimized parameters into a customized simulation environment, the model is trained to effectively balance the ball. The trained policy is then saved for potential deployment or further analysis.**

***Index Terms*—SAC, Pybullet, ball and plate system, neural network**

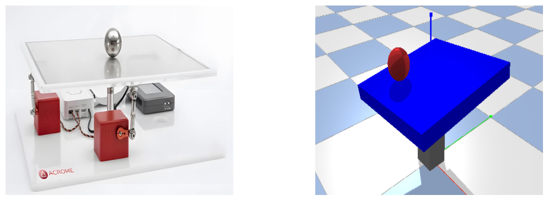
(a) Real Ball-Plate System (b) Simplified Simulation Model

Fig. 1. Ball-plate system

I. Introduction

The plate balancing problem is a classic control challenge that entails keeping a ball stable at the center of a plate by adjusting the plate's tilt angle. This task exemplifies nonlinear, underactuated systems with complex dynamics, making it an essential testbed for robotic control strategies. In practical robotics applications, the significance of the ball-and-plate balancing problem emerges in fields like precision manipulation, balanced robot design, and adaptive control systems—particularly in scenarios where maintaining system stability under dynamic conditions is crucial.

Traditional control methods such as proportional-integral-derivative (PID) controllers [1] and linear-quadratic regulators (LQRs) [2] have been widely employed to address ball-and-plate balancing issues. These approaches are favored for their simplicity, computational efficiency, and strong performance in linear or near-linear systems. However, they often require accurate mathematical modeling and precise parameter tuning, and they have limited capacity to handle nonlinearities and uncertainties within the system. When confronted with highly nonlinear dynamic behaviors, traditional methods may struggle to meet performance expectations.

Reinforcement Learning (RL), especially model-free algorithms like Soft Actor-Critic (SAC), offers a fresh perspective on control problems without the necessity for precise system modeling. RL enables agents to learn optimal policies directly through interactions with the environment, exhibiting self-learning and adaptive capabilities. Its strengths lie in managing nonlinear and high-dimensional control problems and adapting to dynamically changing environments. However, RL methods often face challenges such as low sample efficiency and slow convergence rates. Deploying RL algorithms in real robotic systems also requires careful consideration of safety and reliability during the training phase.

The following pictures show the existing ball-plate system that could keep balance by throw the ball onto the board from any direction, which could be simplified to the simulation model in Pybullet as shown on the right in Figure 1.

The goal of this study is to explore the application of reinforcement learning techniques to train a ball-and-plate balance controller, aiming to surpass the limitations of traditional methods and capitalize on the adaptive advantages of RL algorithms. By comparing the performance of RL approaches with conventional control strategies, we seek to highlight the potential benefits of RL in real-world robotic control systems and identify the challenges involved in their implementation. This research aspires to contribute novel insights and methodologies for advancing intelligent control technologies and solving practical engineering problems.

II. METHOD

The Fig. 2 shows the whole process of how to train the agent. This image on the left represents getting information from a customized environment, via the reset, sample, step function $(s_t, a_t, r, s_{t+1}, done)$ and pass this data into the replaybuffer. On the right is the schematic diagram of the SAC algorithm used in this report. **The algorithms presented in this report build upon those described in the previous learning agent report, incorporating significant improvements and simplifications**. Enhancements include modifications to the actor-critic network and updates to the entropy generation algorithm. The previously utilized network, due to its high complexity, often resulted in gradient explosions, slow convergence, and inefficient training. These issues have been effectively addressed through the improvements detailed in this report. A comprehensive discussion of these enhancements is provided in the following sections.

A. Nerual Network Design

I simplified and improved network structure compared to previous neural network reports. **In previous network designs, adding residual blocks and attentional mechanisms to Actor and Critic networks resulted in poor training results.** I think in this project SAC is based on a continuous control task and the

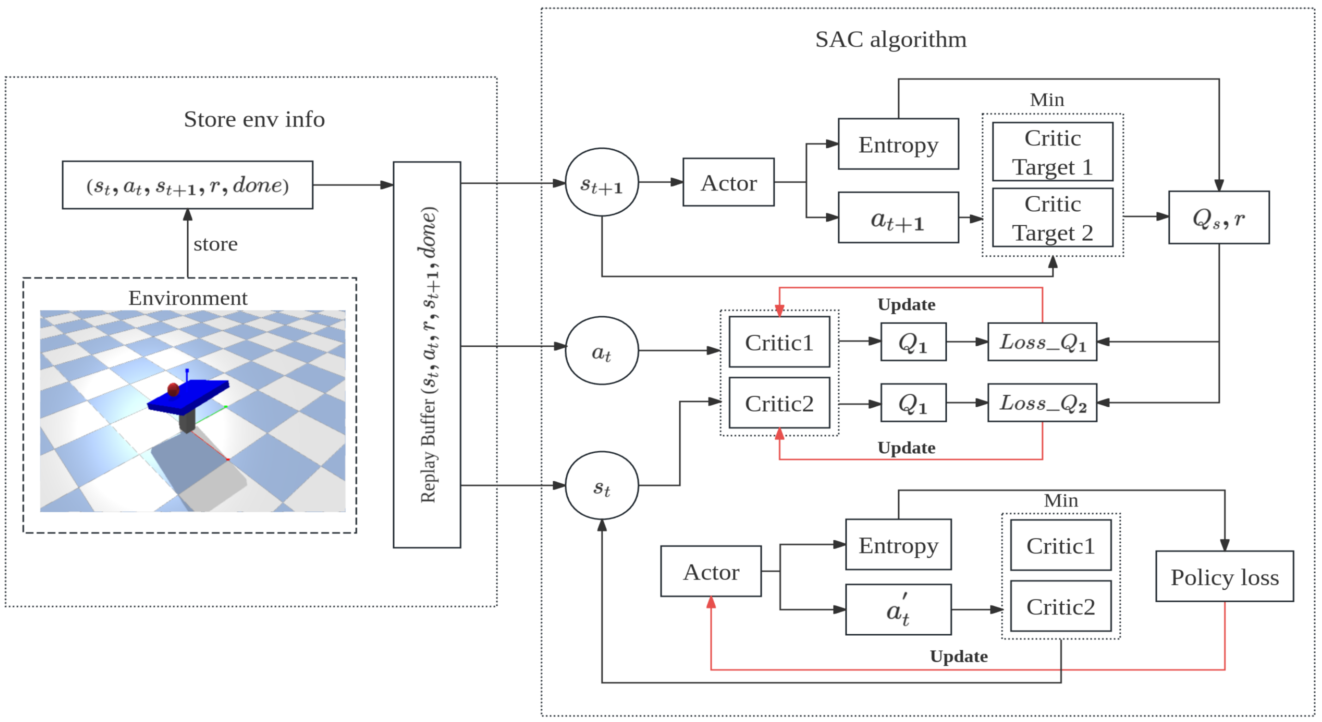

Fig. 2. The flow chart of the whole training agent.

environment is usually noisy. If the model is too complex, it may overfit to the training data and perform poorly in testing or real-world environments. And the increase in model complexity may introduce more gradient instability, especially in critic network, because its objective function depends on the Actor's output and environment feedback, and the complex network structure may amplify the error propagation.

So in the new version, I used a simple MLP network as a common network for Actor and Critic which only have the different input and output dims as is shown in Fig. 3.

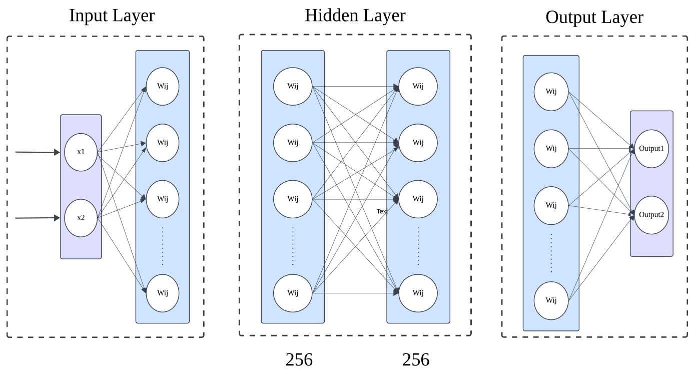

Fig. 3. The Structure of Neural Network.

The following table shows the important parameters of the Actor and Critic networks.

First, a generic multilayer perceptron (MLP) class is defined, which is the basis of the Actor and Critic networks. The input dimension, which is the dimension of the state or state action; the output dimension, which is the dimension of the value to be predicted; and the number of neurons in the hidden layer,

TABLE I
IMPORTANT PARAMETERS IN ACTOR AND CRITIC NETWORKS

Parameter	Actor	Critic
Input dim	8	10
Hidden state 1	256	256
Hidden state 2	256	256
Ouput dim	4	1
Batch size	256	256
Opitimizer	Adam	Adam
Learning rate	$3e^{-4}$	$3e^{-4}$

which is set to 256 here, are defined. and a two-layer fully-connected network is constructed using nn.Sequential, with the ReLU activation function attached to the back of each layer. The output layer is defined to represent the mapping of the output of the hidden layer to the desired output dimension. In addition the _init_weights method is called to initialize the network weights. In the forward method the input x is passed through the network self.net and then through the output layer self.output_layer to get the final output.

The Actor network is used to output the probability distribution parameters of the actions based on the current state and is responsible for policy generation.

First initialize the dimension of the state, the dimension of the action, and the range of values for the action (for scaling the action to what the environment allows). Define self.net, using the MLP class defined earlier, with an output dimension of action_dim * 2, since it is necessary to output the mean (mean) and log standard deviation (log_std) of the action**(state_dim=8, action_dim=2, output_dim=action_dim*2=4)**. The goal of the forward propagation method is to compute the mean $\mu(s)$ and standard deviation $\sigma(s)$ of the action distribution, given the state

s. First, the hidden representation is computed through the first few layers of the network (excluding the output layer):

$$h = f_{net}(s) \tag{1}$$

where f_{net} denotes the first few layers of the network (self.net.net) and h is the hidden layer output vector.

Through the output layer, a spliced vector of the mean and log standard deviation of the action distribution is obtained:

$$\begin{bmatrix} \mu(s) \\ log\ \sigma(s) \end{bmatrix} = W_{out}h + b_{out} \tag{2}$$

where: W_{out} is the weight matrix of the output layer. b_{out} is the bias vector of the output layer, and the dimension of the output vector is 2×action_dim. The output vector is subsequently sliced into mean and log standard deviation in the last dimension by the torch.chunk function:

$$\mu(s) = [\mu_1(s), \mu_2(s), ..., \mu_n(s)]^T \tag{3}$$

$$log\ \sigma(s) = [log\ \sigma_1(s), log\ \sigma_2(s), ..., log\ \sigma_n(s)]^T \tag{4}$$

where n=action_dim. To maintain numerical stability, the log standard deviation is trimmed (truncated):

$$log\ \sigma(s) = clip(log\ \sigma(s), -20, 2) \tag{5}$$

The function of the clip is to limit the value of $log\ \sigma(s)$ between -20 and 2. Trimming is performed to maintain numerical stability and prevent numerical overflow or underflow during calculations. When the standard deviation is too large ($log\ \sigma(s)$ is very large): it may lead to sampling actions that are too random, making it difficult to learn effective strategies. When the standard deviation is too small ($log\ \sigma(s)$ is very small or negative infinity): the standard deviation tends to be close to zero, leading to strategies that are too deterministic and unexploratory, and that can result in numerical underflow or divide-by-zero errors in the calculation of log probabilities. Next, the log standard deviation is converted to standard deviation by means of an exponential function:

$$\sigma(s) = exp(log\ \sigma(s)) \tag{6}$$

In the sampling process, a modification was made to the methodology for the calculation of log probability. The goal of the sampling **method (self.actor. sample)** is to sample action a from the strategy and compute its log probability $log\ p(a|s)$. The forward method is called to obtain the parameters of the action distribution, i.e., the mean($\mu(s)$) and standard deviation($\sigma(s)$). Then **the reparameterization trick(normal.resample)** was used to sample from a Gaussian distribution [4]:

$$\mathbf{z} = \mu(s) + \sigma(s) \odot \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I}) \tag{7}$$

where z is the intermediate variables not processed by the activation function. $\boldsymbol{\epsilon}$: random noise sampled from a standard normal distribution. $\odot$ denotes element-by-element multiplication . The operation of sampling directly from a probability distribution is not microscopic and cannot be optimized for policy networks. The reparameterization technique converts the random sampling process into a combination of a deterministic function about a differentiable parameter and independent random noise, making the sampling process microscopic. The sampled values are then passed through the $tanh$ function to ensure that the action is in the range (-1,1). Then calculate the logarithmic probability of the action. The logarithmic probability of the original Gaussian distribution is:

$$log\ p_{\mathcal{N}}(\mathbf{z}|s) = -\sum_{i=1}^{n}(\frac{1}{2}(\frac{(z_i - \mu_i(s))^2}{\sigma_i(s)^2}) + 2log\ \sigma_i(s) + log\ 2\pi) \tag{8}$$

Since the $tanh$ transform is used, the logarithmic probability of the transform needs to be calculated. According to the transformed probability density formula:

$$log\ p(\mathbf{a}|s) = log\ p_{\mathcal{N}}(\mathbf{z}|s) - \sum_{i=1}^{n} log|\frac{\partial a_i}{\partial z_i}| \tag{9}$$

where the diagonal elements of the Jacobi determinant are:

$$\frac{\partial a_i}{\partial z_i} = 1 - tanh^2(z_i) = 1 - a_i^2 \tag{10}$$

Therefore, the log probability correction term is $\sum_{i=1}^{n} log\ (1 - a_i^2 + \boldsymbol{\epsilon})$, where $\boldsymbol{\epsilon} = 1 \times 10^{-6}$ is a very small constant that prevents the value from being unstable. Then mapping the action to an actual action range (assuming the action range is $[-a_{bound}, a_{bound}]$):

$$a = a \times a_{bound} \tag{11}$$

Here a_{bound} is the maximum value obtained from the environment, which in our environment means the maximum angle of each rotation of the plate. The environment specifies a maximum angle of 0.08 rad for a single movement.

In conclusion, the Actor network uses a Gaussian distribution as an approximation of the strategy, i.e:

$$\pi(\mathbf{a}|s) = \mathcal{N}(\mu(s), \sigma(s)) \tag{12}$$

When optimizing the strategy, the logarithmic probabilities of the actions need to be calculated in order to update the parameters of the strategy network. Due to the $tanh$ transform, the original Gaussian log probabilities need to be corrected.

The Critic network consists of two Q-networks (q1_net, q2_net) created with the MLP class, where the inputs are action-space and state-space splices(state_dim + action_dim = 10)and the outputs are the corresponding Q-values, i.e., the expected cumulative returns.The Critic part uses two independent Q-networks, which reduces the bias in the estimation of the values, improves the stability of the training, and the outputs are used to compute the gradient of the policy, which guides the The output is used to compute the strategy gradient, which guides the Actor network to update its strategy parameters.

B. System environment construction

Since the ball and plate balance system is a physical model, the state and action space are continuous. The physical model and environment used in this report is essentially the same as that used in the previous report, **with changes to some of the values in the state and action spaces and changes to the reward function In order to be more relevant**. This is described in more detail below.

The state space mainly contains the ball's for displacement, velocity and the current angle of the plate:

$$state = [x, y, z, \dot{x}, \dot{y}, \dot{z}, \theta_x, \theta_y] \tag{13}$$

Now calculate the upper and lower limits of the z-axis displacement of the sphere based on the maximum deflection angle of the plate:

$$z_{min} = h_{base} + \frac{1}{2}\sin(\theta_{min}) \cdot w_{plate} \tag{14}$$

$$z_{max} = h_{base} + \frac{1}{2}\sin(\theta_{max}) \cdot w_{plate} \tag{15}$$

where $\theta_{min} = -45°$, $\theta_{max} = 45°$, which means the uper and lower limits of plate rotation angle. h_{base} is the height of center support cylinder. w_{plate} is the width of the plate. The final calculation gives $z_{min} = 0.4$, $z_{max} = 1.0$.

The upper and lower bounds of the velocity of the ball in the state space are also changed, and **the upper and lower bounds of the velocity along the three directions are converted from [-5, 5] to [-1, 1]**, in order to make the simulation process smoother and to prevent it from ending prematurely due to physical factors.

Similarly, during the simulation we found that the training process was difficult to converge due to the large plate deflection angle set in the action space for each movement. Therefore, we reduced the motion amplitude in the action space by setting the upper and lower limits of each deflection to [-0.1, 0.1] (in radian system, which is $[-5.73°, 5.73°]$). It represents the amplitude of each rotation of the plate along the x and y axes.

In addition, I modified part of the reward function. Termination penalties have been removed, as **it has been found in experiments that termination penalties tend to be larger resulting in a steady decline in the average reward of the rounds**. as shown in the curve below:

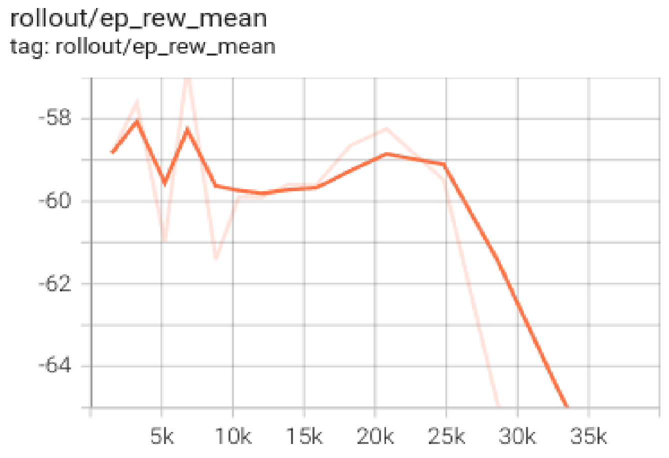

Fig. 4. The original reward curve.

So I remove the termination penalty and use the sum of the squares of the angles of rotation in both directions as the movement penalty to Intelligentsia are encouraged to use smooth, moderate control actions to avoid over-manipulation and reduce energy consumption and shock to the system:

$$angle\_reward = -0.5 \cdot (|\theta_x|^2 + |\theta_y|^2) \tag{16}$$

It has been shown experimentally that the training effect converges faster in this case. In addition, a reward condition was added:

$$reward = reward + 1, \quad if \quad d < 0.05 \tag{17}$$

where $d = \sqrt{x^2 + y^2}$, representing the distance from the ball to the center of the plate; $reward = distance\_reward + angle\_reward + velocity\_reward$. **The improved reward curve will be discussed in detail in the experimental section.**

C. Soft Actor-Critic Algorithm

The basic framework of the SAC algorithm used in this report is roughly the same as that used in the previous Learning Agent report and described in detail in the previous report. The basic framework is shown below, which is also the right part in Figure 1 [3]. I will briefly describe the SAC process below: The SAC algorithm inputs the current state s t of the

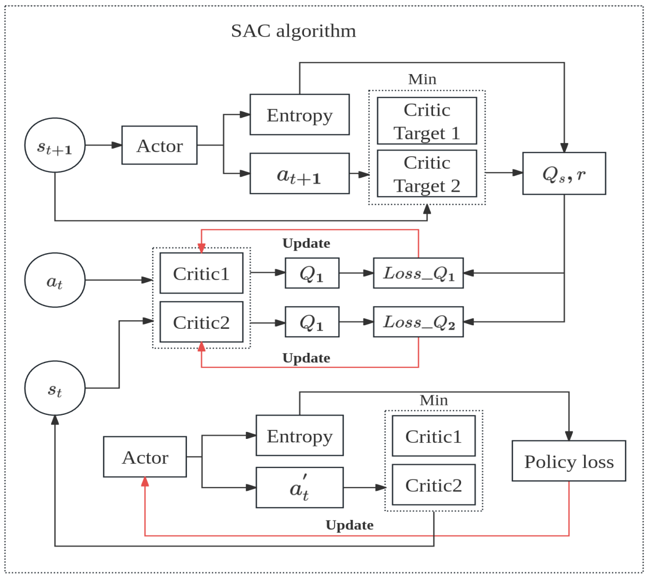

Fig. 5. The SAC Framework.

robot into the Actor network , the Actor network gets the mean and variance of the Gaussian distribution based on the input state, samples the output action a_t according to the probability distribution, obtains the reward r_t, the state changes to s_{t+1}, and puts the currently generated experience $(s_t, a_t, r_t, s_{t+1}, done)$ into the replay buffer. When the experience reaches a certain number, sample a batch of experience from the experience pool, calculate the minimum of the average value of the 2 Critic target networks, update the 2 current Critic networks, then update the Actor network, and use soft update to update the Critic target networks. Repeat the above process until convergence.

The improvement in this report is the addition of an update to the entropy coefficients α. The entropy coefficient α is a key hyperparameter that controls the stochasticity of the strategy, i.e., the trade-off between exploration and exploitation. Updating α helps the algorithm to dynamically adjust the degree of exploration during the training process, thus improving the learning efficiency and stability of the strategy.

First define a target entropy value, $\mathcal{H}_{target}$, indicating the desired level of strategy randomness. It is usually set to a negative value of the action dimension:

$$\mathcal{H}_{target} = -action\_dim = -4 \tag{18}$$

Then α is updated by minimizing the following loss function:

$$J(\alpha) = E_{a_t \sim \pi} = [-\alpha(\mathcal{H}(\pi(\cdot|s_t)) + \mathcal{H}_{targrt})] \tag{19}$$

The goal of this loss function is to bring the actual entropy of the strategy close to the target entropy. Where $\mathcal{H}(\pi(\cdot|s_t)$ is computed through sample method in Actor network. An estimate of the entropy is obtained by sampling the action a from the strategy and computing the corresponding log probability $log\ \pi(a|s_t)$, then taking the negative expected value.

D. Training Agent

This section will discuss in detail how to train the intelligences and show the important training parameters, starting with a training loop diagram: It is clear that there are two loops (step loop

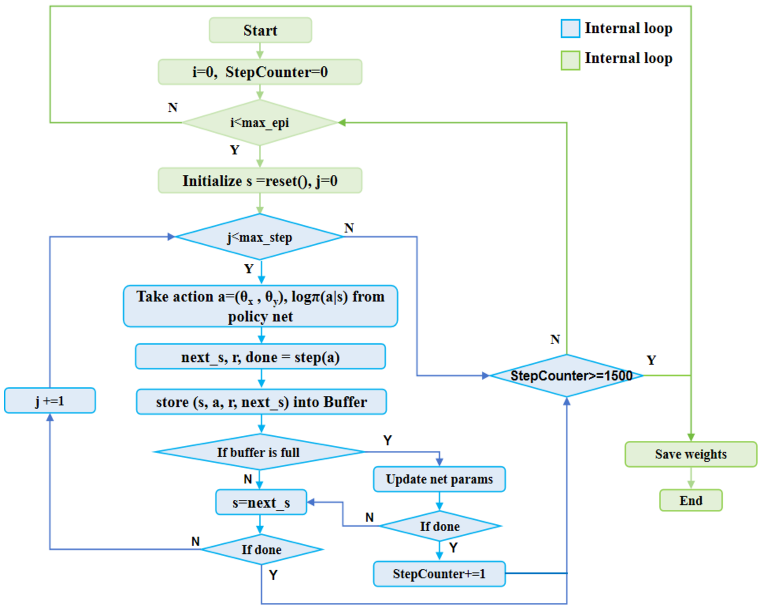

Fig. 6. Flowchart of the Main Training Loop (learning agent).

and episode loop) inside. The important condition for breaking the internal loop is that the agent meets the done condition or j reaches the maximum step number. The important condition for breaking the episode loop is that after the memory is full, the number of steps is more than or equals to 1500 or i reaches the maximum episode number. When initializing the environment, the command $env.reset()$ will be used and it will generate one random state for agent to train. That's the main part of the learning agent (train_sac function). Some of the key parameters regarding the SAC algorithm are shown in the following table:

TABLE II
RELATED PARAMETERS IN SAC

Parameters	Value
Discount factors γ	0.99
Soft update arameter τ	0.005
Learning rate	3×10^{-4}
Batch size	256
Replay buffer size	1×10^{6}
Episodes	400
Action bound	0.1rad
Device	GPU

III. Experiments and Results

This section focuses on the experimental procedure and discussion of the results of the ball-plate system. The training process is visualized in Pybullet by setting render=True, and set the ball to fall anywhere from 0.5m above the board (size of the board is 0.5×0.5, with the center of the board as the origin, and the ball's point of fall is within [-0.4, 0.4]). The training environment is Ubuntu 22.04, and the GPU used for simulation is NVIDIA GeForce RTX 2060/PCIe/SSE2.

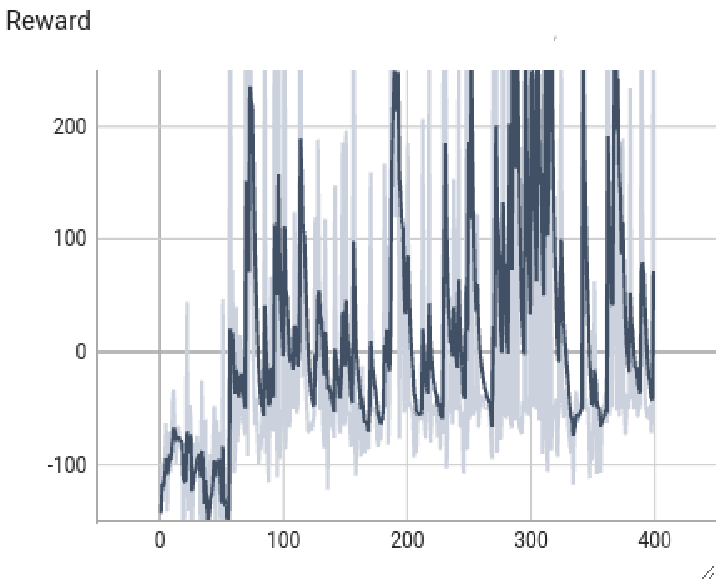

Fig. 7. The average episode reward.

Figure 7 demonstrates the change in the average reward for system training after improving the neural network structure and the log-probability generation algorithm. In general, the Balance Board system improves the reward value given by the environment by constantly learning. This results in greater bonus values. Comparing Fig. 4 and Fig. 7, it is obvious that The original network showed a decreasing trend in getting rewards per round and was poorly trained. As can be seen in Fig. 7, the system shows an upward trend in the average reward after about 60 rounds of iterations under the modified SAC algorithm. After that, the reward value fluctuates due to the high-dimensional continuity of the system's action and state space, but it has been maintained at a high level, and the reward value can reach up to 1000, which indicates that the system learns a better strategy in the process. In terms of the reward function, there is a great improvement compared to the original.

Fig. 8 shows the loss of the policy network. Initially, Actor Loss rises rapidly to a high value (about 10) and then begins to fall, gradually stabilizing. After stabilization, it fluctuates somewhat but generally varies between 5 and 6. In the early stages of training, the strategy may output very randomized actions with increased uncertainty in the plate rotation angle, resulting in a very low Q, along with high entropy and a consequent increase in loss. As training proceeds, the policy becomes more optimized and is able to find near-optimal actions, thus reducing the loss. After training stabilization, the fluctuations in Actor Loss reflect the learning dynamics of the strategy in different states. **Overall, Actor Loss stabilizes and remains moderately volatile, indicating that the policy network is being effectively optimized.**

Fig. 9 illustrates the variation of temperature parameter α loss. Initially the Alpha Loss decreases rapidly, even to large negative values. Subsequently, the Alpha Loss fluctuates up and down around 0 to a lesser extent. In the early stages of training, the strategy may produce high-entropy actions (i.e., the actions

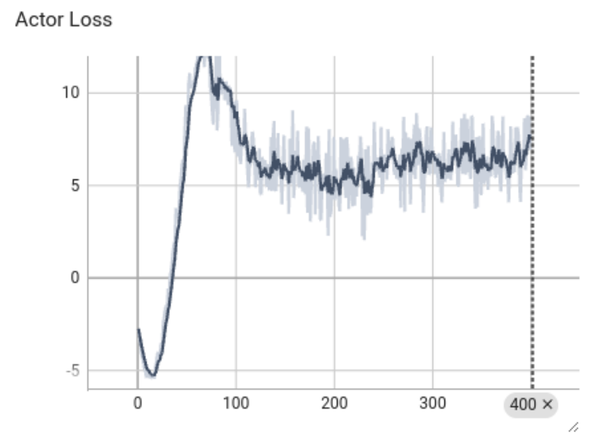

Fig. 8. Actor loss

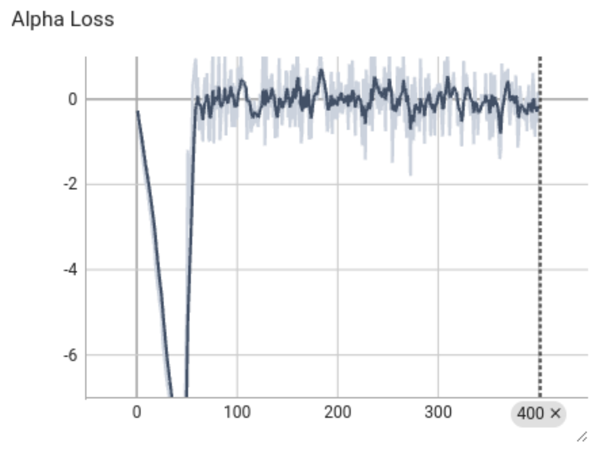

Fig. 9. alpha loss

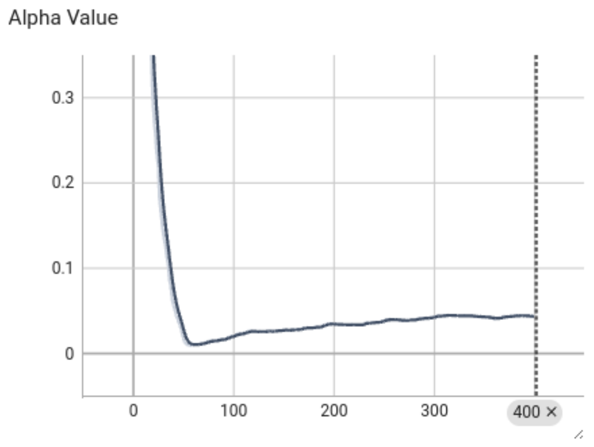

Fig. 10. Alpha value

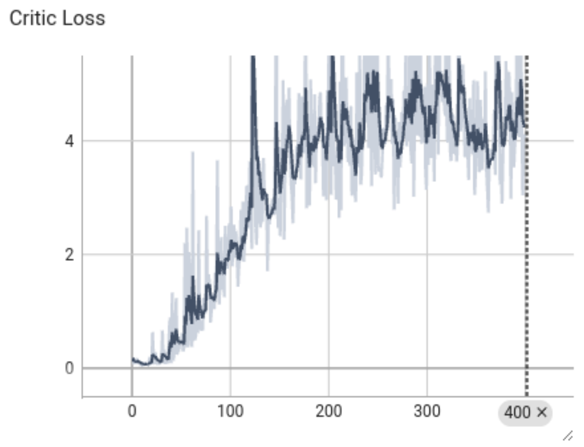

Fig. 11. Critic loss

are highly randomized), and Alpha Loss reduces the entropy by adjusting alpha. After that the loss fluctuates around 0 alpha changes level off and the entropy regulation mechanism works properly.

Fig. 10 shows the variation in the values of the temperature parameter; initially, the Alpha Value rapidly decreases from a large initial value (about 0.35) to a lower range (close to 0.05), and then oscillates around the low value.The entropy of the Alpha control strategy is such that the higher the value, the more the strategy tends to explore, and the lower the value, the more the strategy tends to utilize the optimal action. At the beginning of training, the randomness of the strategy actions is large, and the Alpha value needs to decrease rapidly to reduce exploration and prompt the strategy to converge. After that, the strategy is close to optimal and the Alpha value is kept in a low range, allowing limited exploration. **The images show that the exploration strategy of this system is effectively controlled**. It is normal for Critic Loss to remain fluctuating in the later stages, but in this system, the later stages fluctuate slightly more, probably related to the learning rate and other hyperparameters of the network.

Fig. 11 shows the Critic loss.Initially, the Critic Loss is low, then it gradually increases and stabilizes within a certain range in the middle and late stages. In the early stages of training, the current Q-network and the target Q-network are not fully optimized due to the fact that they use the same weights, resulting in predictions that are closer to the initial target value. As the policy network is optimized, the actions input to the Q network become more complex and the Q-valued network needs to capture these changes, making optimization more difficult.

What's more, during the training process, the weight parameters of the network with the largest average reward in all rounds were stored in the file "best_model.pth", and a test script was written. For the established physical environment, the model was inducted for 100 rounds by loading the original network structure and weight parameters, and the distribution of the reward function was obtained. as follows:

It can be seen that the average reward was basically at a high level of fluctuation during the test, and the training effect proved to be effective!

Overall, the training process is effective and the trend of each loss term is as expected. Both the strategy network and the value network are gradually being optimized, and the Alpha adjustment mechanism is functioning properly, indicating that the model is learning the dynamics of the environment and converging to a reasonable strategy.

IV. Conclusion and reflection

At first, the network was designed to be very complex, the effect is counterproductive, and then the network to remove the attention mechanism and residual block, the effect of training under the greatly improved, and according to the training effect to improve the reward function, so that it is as smooth as possible;

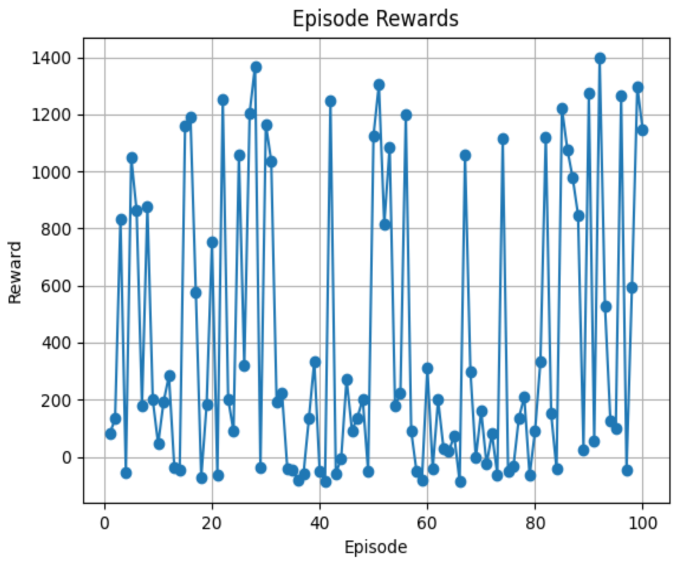

Fig. 12. The test average reward.

in addition, according to references to amend the algorithm of the logarithmic probability, so that the problem can be resolved.

A. *Advantages*

The ball-and-plate balancing system is a typical nonlinear, underdriven control problem, and traditional control methods (e.g., PID or LQR) may have difficulty in handling the complex nonlinear dynamics. SAC, as a model-independent deep reinforcement learning method, can directly learn the near-optimal policy by interacting with the environment without explicit modeling. The ball and board balancing problem requires continuous adjustment of the board angle to control the ball position, while SAC supports learning continuous action strategies directly, avoiding the loss of accuracy caused by discretization. Compared with the discretized action space approach, SAC can control the inclination of the board more finely, thus achieving a smoother balancing process.

B. *Limitations*

In this project, this algorithm also demonstrates some limitations, when the ball enters the table from the edge position of the table, as soon as many rounds are trained, the whole system is difficult to maintain the balance (to control the ball not to fall off the table). I think it may be that the SAC algorithm is a model-free reinforcement learning method whose learning relies heavily on sampled experience. If there are fewer samples of edge states in the training data, the strategy will be less capable of learning these rare states. It is also possible that the SAC's automatic temperature regulation mechanism may lead to under-exploration, and that the strategy may not attempt enough actions to find the optimal response in the edge state. It is also possible that the reward function lacks a clear guide to the direction of the action, resulting in a strategy that is not precise enough in the edge state. The strategy attempts to directly minimize the distance between the ball and the center, but fails to effectively account for slowing down and orienting the ball.

C. *Reflection and Lessons learned*

The first challenge encountered in completing the project was setting up the physical simulation environment, as pybullet was completely new to the environment. For this reason I consulted the relevant interface documentation and wrote the physical model modeled after the urdf format. The second challenge was the algorithm for the reward function, after many attempts I finally chose to use the sum of squares to make the reward calculation smoother. The last and most important challenge was the reproduction of the SAC algorithm, there was a doubt on how to calculate the log probability part of the action, I reviewed the paper and the related open source code, and finally revised the algorithm to improve the experimental results.

D. *Future work*

We can design additional reward functions for complex states (e.g., ball-on-the-edge or high-speed states) encourages learning of the strategy in these states. Improve the physics model to make it more consistent with the actual physical environment, and use active exploration methods (e.g., strategies based on uncertainty estimation) to guide sampling and ensure that the training data covers both edge and high-speed states. Increasing the randomness at the beginning of training, in particular randomly initializing the position and velocity of the ball.

References

[1] O'Dwyer, A. (2009). "Handbook of PI and PID Controller Tuning Rules." Imperial. College Press

[2] Anderson, B. D. O., & Moore, J. B. (2007). "Optimal Control: Linear QuadraticMethods." Dover Publications.

[3] Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. In Proceedings of the 35th International Conference on Machine Learning (ICML), pp. 1861–1870.

[4] Kingma, D. P., & Welling, M. (2014). Auto-Encoding Variational Bayes. In Proceedings of the 2nd International Conference on Learning Representations (ICLR).