RL Training of Inverted Singular Pendulums

Zhang Zhiyi
Zhuang Zikun
Southern University of Science and Technology)
Shenzhn, China
12111916@mail.sustech.edu.cn
12110418@mail.sustech.edu.cn

***Abstract*—This report explores strategies for training an inverted pendulum via reinforcement learning (RL). The inverted pendulum consists of a single rod fixed at one end and requires control input to keep it upright and balanced. The main goal of this project is to apply the basic policy gradient method (REINFORCE) to train the control policy. We define appropriate state and action spaces, design a reward function that encourages stable equilibrium, and use the REINFORCE algorithm to learn the control policy. The report explains the RL algorithm used in detail and discusses the comparison of the results.**

***Index Terms*—RL,inverted pendulum,RF,reward,action**

I. Introduction

A. Project Background

Reinforcement Learning (RL) has become a powerful tool in the field of robotics and control systems. It involves training an agent to make a sequence of decisions by rewarding desired actions and punishing undesired ones. One classic problem used to study and develop RL algorithms is the control of inverted pendulums, which are systems that are inherently unstable and require continuous balancing.

The primary objective of this problem is to apply the proper RL algorithm to train control policies that can successfully balance both types of inverted pendulums. This involves defining appropriate state and action spaces, designing a reward function that encourages stable balancing, and employing RL algorithms to learn the control policies.

B. Problem Introduction

For the inverted pendulum, we need to finish the following two sub-problems.

- The pendulum starts with a slight inclination. The goal is to stabilize the pendulum in the upright position by controlling the movement of the cart.
- The pendulum begins in the downward position. The objective is to swing the pendulum up and stabilize it in the upright position.

II. Theoretical Introduction

In this project, we used the basic policy gradient method (REINFORCE) to train the control policy. The theoretical part will be divided into two parts: Policy Network and Agent.

Identify applicable funding agency here. If none, delete this.

A. Policy Network

In this project, we used a feedforward neural network as the policy network, which receives state input and outputs the mean and standard deviation of the action, thereby parameterizing the probability distribution of the action. The structure of the policy network includes an input layer, two hidden layers, and an output layer:

- **input layer**Receive status information from the environment, including the position and speed of the car, the angle and angular velocity of the rod.
- **hidden layer**include two fully connected layers, each of which uses the ReLU activation function. The role of these layers is to extract state features through nonlinear transformation to generate appropriate action distribution parameters.
- **output layer**Generate the mean and standard deviation of the action. The mean determines the most likely action to be taken in the current state, and the standard deviation reflects the uncertainty of the strategy.

B. Agent

The agent is responsible for interacting with the environment, sampling actions, collecting data, and updating the parameters of the policy network through the policy gradient method (REINFORCE). The REINFORCE algorithm is a Monte Carlo-based policy gradient algorithm that aims to optimize the policy by maximizing the cumulative return. This idea mainly comes from the teaching assistant's PPT explanation.

- **Input Layer**: Receives state information from the environment, including the cart's position, velocity, the pole's angle, and angular velocity.
- **Hidden Layers**: Comprise two fully connected layers, each using ReLU activation functions. These layers extract state features through nonlinear transformations to generate appropriate action distribution parameters.
- **Output Layer**: Produces the mean and standard deviation of the action. The mean determines the most likely action to take in the current state, while the standard deviation reflects the uncertainty of the policy.

Using the logarithm of the standard deviation (log_std) helps improve numerical stability and ensures that the standard

deviation is always positive. During training, by optimizing these parameters, the policy network can gradually learn how to generate optimal control actions in different states, achieving the goal of balancing the inverted pendulum.

C. Agent

The agent interacts with the environment, samples actions, collects data, and updates the policy network's parameters using the policy gradient method (REINFORCE). The REINFORCE algorithm is a Monte Carlo-based policy gradient method that aims to optimize the policy by maximizing the cumulative reward.

The main components and steps of the agent include:

1) **Initialization**:
   - Create an instance of the policy network.
   - Set up the optimizer (e.g., Adam) to update the policy network's parameters.
   - Define the discount factor γ, used to calculate the discounted future rewards.
2) **Action Sampling**:
   - At each time step, use the policy network to generate the mean and standard deviation of the action based on the current state.
   - Sample the actual action from the generated probability distribution and calculate its log-probability $\log \pi_\theta(a_t|s_t)$, where θ represents the policy parameters, a_t is the action at time t, and s_t is the state at time t.
3) **Policy Update**:
   - Collect state, action, reward, and log-probability data for an episode.
   - Calculate the discounted return $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ for each time step, and normalize it to improve training stability and efficiency.
   - Compute the policy loss: using the discounted returns and the log-probabilities of the actions, calculate the policy gradient for each action and accumulate the loss $L(\theta) = -\mathbb{E}[\log \pi_\theta(a_t|s_t) G_t]$.
   - Update the policy network's parameters by backpropagation and the optimizer to maximize the future rewards.

III. InvertedPendulum-v4

The model we use is InvertedPendulum-v4,first,let's talk aboutthis model.

A. State Space

The state space includes the following four variables:

- x: Position of the cart on the horizontal track.
- $\dot{x}$: Velocity of the cart.
- θ: Angle of the pole relative to the vertical direction.
- $\dot{\theta}$: Angular velocity of the pole.

B. Action Space

The action space is the horizontal force F applied to the cart. the action space is continuous, allowing the application of any force within a certain range $[-3, 3]$.

C. step Function

The step function has three return values,they are obs, reward, terminated:

- **obs** This contains the state space of the model, x,$\dot{x}$,θ,$\dot{\theta}$.
- **reward**The reward function is designed to encourage the system to remain balanced and minimize excessive actions.However ,in this model, reward $= 1$,it is a constant.
- **terminated**This condition is used to determine when an episode should end. The main condition is `(np.abs(ob[1]) > 0.2)` checks if the absolute value of the pole's angle exceeds 0.2 radians. If it does, this part evaluates to `True`.

D. Reset Model

The `reset_model` function is responsible for initializing the state at the beginning of each episode. This method ensures that the pendulum and cart start from a slightly perturbed initial state.The steps are as follows:

- Generate the initial position `qpos` by adding a small random offset to the initial position `self.init_qpos`. The offset is sampled uniformly from the range $[-0.01, 0.01]$.
- Generate the initial velocity `qvel` by adding a small random offset to the initial velocity `self.init_qvel`. The offset is sampled uniformly from the range $[-0.01, 0.01]$.
- Call `self.set_state(qpos, qvel)` to set the state of the model with the generated position and velocity.
- Return the current observation by calling `self._get_obs()`.

IV. Stabilize the Pendulum with a Slight Inclination

First, we define the total number of training episodes as 5000.

```
total_num_episodes = 5000
```

A. Training Loop

The training process consists of the following steps:

1) **Reset the Environment**: At the beginning of each new training episode, reset the environment to get the initial observation and environment information.

```
    for episode in range(
        total_num_episodes):
        obs, info = wrapped_env.reset()
        done = False
        rewards = []
        log_probs = []
```

2) **Episode Loop**: Within each episode, the agent samples actions based on the current state, interacts with the environment, and receives new states and rewards until

the episode ends.This while loop is very important,let me explain this.

First, we must clarify two things again.

- **reward** reward $= 1$
- **terminated** The main condition is if `(np.abs(ob[1]) > 0.2)` this part evaluates to `True`.

Although on the surface, our reward value of 1 does not seem to be directly related to keeping the inverted pendulum upright, in fact, when the inverted pendulum falls, the loop will trigger the termination condition and end. This means that if the model is able to hold on to a nearly vertical position for longer, the loop will execute more steps. Therefore, the number of rewards stored in the rewards list will increase, and eventually when the update() method is called, a larger cumulative reward R will be calculated, making the model more accepting of this behavior. For this model, the default maximum number of steps is 1000, so in our drawing method, the maximum total reward is 1000.

```
while not done:
    action, log_prob = agent.
        sample_action(obs)
    obs, reward, terminated, truncated
        , _ = wrapped_env.step([action
        ])
    done = terminated or truncated
    rewards.append(reward)
    log_probs.append(log_prob)

agent.update(rewards, log_probs)
```

3) **Monitor Training Progress**: Every 100 episodes, calculate and print the average reward of the last 100 episodes to monitor the training progress.

```
if episode % 100 == 0:
    avg_reward = np.mean(
        rewards_per_episode[-100:])
    print(f"Episode {episode}: Average
         Reward: {avg_reward:.2f}")
```

B. Result

Finally, let's take a look at the effect display.

- **Initial Phase**: In the first 500 episodes, rewards are low and stable, indicating poor initial performance.
- **Reward Increase**: From episode 500 to 1500, rewards gradually increase, showing significant fluctuations as the agent learns basic balancing strategies.
- **Stabilization**: After episode 1500, rewards increase significantly and stabilize.
- **Max Reward**: After episode 2000, rewards frequently reach 1000, the maximum, showing near-perfect balance maintenance.
- **Fluctuations**: Despite high rewards, some episodes still show lower rewards due to occasional failures or randomness.

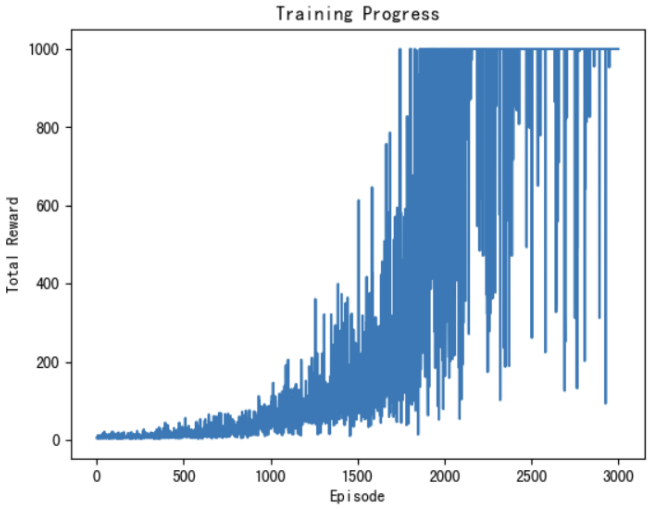


Fig. 1: Training Progress of the Inverted Pendulum Agent

V. Swing the Pendulum Up and Stabilize

In this project, we focus on the pendulum starting from the downward position. The goal is to swing the pendulum up and stabilize it in an upright position by controlling the movement of the car. Specifically, the pendulum is initially in a downward vertical position, and through reward learning, it gradually swings up and finally stabilizes in an upright state. There are two key points in this task, one is to be able to swing up, and the other is to stabilize it.

A. Configuration changes

Here we change two configurations of two files, the first one is about the track length, the second one is about the initial angle as follows:

- **track** The following operations are performed in `inverted_pendulum.xml` file.
  - **Slider Size**: The slider size is changed from $[-1, 1]$ to $[-5, 5]$. This modification allows the inverted pendulum to have sufficient space for the swing-up maneuver.
  - **Pole Swing Range**: The swing range of the pole is modified from $[-180°, 180°]$ to $[-180°, 180°]$. This change enables the pole to rotate without any restrictions.
- **initial angle** The following operation is performed is `inverted_pendulum_v4.py` file. In the `reset` method, we modified `obs[1]`, the angle value, to `np.pi`. This ensures that the pendulum starts in the vertically downward position each time the environment is reset.

B. Custom Environment Wrapper

First, we need to explain why we need to customize the environment wrapper. For the original InvertedPendulum-v4 environment, its reward mechanism and termination conditions do not fully meet our needs. Therefore, the benefit of a custom

environment wrapper is that we can easily use a custom reward mechanism and termination conditions.

It should be noted that the custom environment wrapper inherits from the InvertedPendulum-v4 instance, and all we have done is to rewrite some of its methods. This approach allows us to flexibly adjust and optimize specific functions to meet experimental needs while retaining the characteristics of the original environment.

C. Reward choice

1) Reward 1:

$$\text{reward} = 0.5 \cdot \cos(\theta) - \left(\frac{x}{2}\right)^2 - 0.01 \cdot u^2 - 0.1 \cdot \dot{\theta}^2$$

Explanation and Rationality:

- **Encourage Balance**: $0.5 \cdot \cos(\theta)$ rewards the pendulum being close to upright.
- **Control Position**: $-\left(\frac{x}{2}\right)^2$ penalizes the cart deviating from the center.
- **Reduce Action Magnitude**: $-0.01 \cdot u^2$ penalizes large control forces.
- **Reduce Angular Velocity**: $-0.1 \cdot \dot{\theta}^2$ penalizes high angular velocities.

2) Reward 2:

$$\text{reward} = -(\theta^2 + 0.1 \cdot \dot{\theta}^2 + 0.001 \cdot u^2)$$

Explanation and Rationality:

- **Direct Penalty for Deviation**: $-\theta^2$ penalizes the angle deviation.
- **Reduce Angular Velocity**: $-0.1 \cdot \dot{\theta}^2$ penalizes high angular velocities.
- **Reduce Action Magnitude**: $-0.001 \cdot u^2$ penalizes large control forces.

3) Reward 3:

$$\text{reward} = 2$$

You might be wondering why I use a constant as the reward. Don't worry, see below for the termination condition selection.

D. terminated choice

1) Terminated 1: Maximum Steps:

$$\text{max\_steps\_per\_episode} = x$$

In each episode, we set the maximum number of steps to x. The episode terminates when the length of the rewards list reaches or exceeds this maximum step count. This condition ensures that each episode has a finite length, preventing the agent from exploring indefinitely in a single episode.

```
done = len(rewards) >=
    max_steps_per_episode
```

2) Terminated 2: Angle Exceeding Range:

$$\text{terminated} = \text{bool}(|\text{obs}[1]| > 0.2)$$

This is actually the termination setting of the first question. You may know what I want to do here. It doesn't matter if you don't know. Let's continue reading.

```
terminated = bool(\left| \text{obs}[1] \
    right| > 0.2)
```

E. Training Mode

In training the agent for the inverted pendulum system, we adopt an iterative training mode. After each training session, we save the model and load it for further training in the next session. Here are the key points of our training mode:

1) Model Saving and Loading: We save the model after each training session and load it for the next session, avoiding the need to start from scratch.

2) Training Episodes: Each training session consists of 3000 to 4000 episodes, allowing for quick evaluation and adjustments.

3) Dynamic Adjustment of Maximum Steps per Episode: The maximum steps per episode are dynamically adjusted based on the training situation to accommodate different training phases.

4) Adjusting Reward and Termination Conditions: We adjust the reward function and termination conditions based on the model's performance to ensure the agent learns stable and effective strategies.

F. Training Process

During the training process, I divided the whole process into two stages: first, swinging the inverted pendulum from the vertical downward position to the highest point, and second, keeping it stable near the highest point. These two processes are the key to implementing the entire strategy, and I will explain their implementation in detail below.

1) To the highest point: In this stage, I trained a total of fifteen models. The process can be detailed as follows:

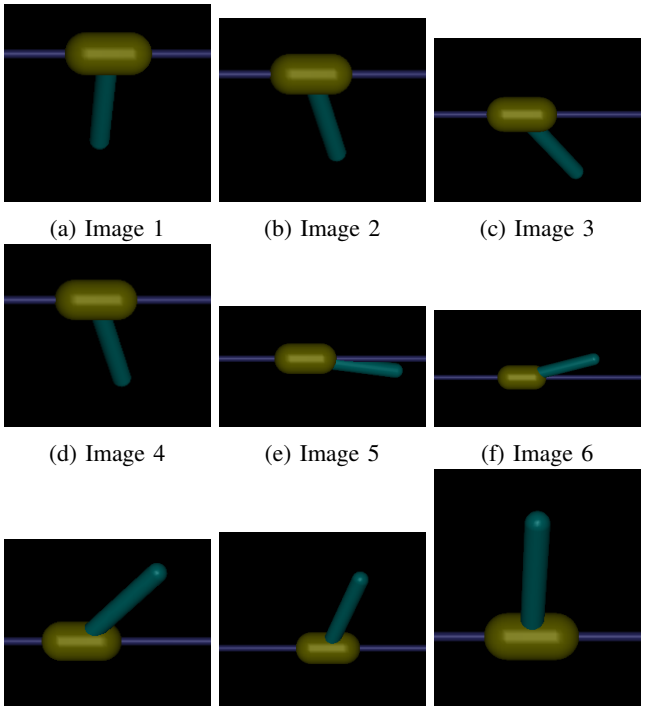
1) **First Eight Models**:
   - Utilized Reward 2 and Termination 1 .
   - Set `max_steps_per_episode` to 200.

   Reason for Setting `max_steps_per_episode` to 200:
   - During training, each step lasts 0.02 seconds, giving each episode a total of 4 seconds.
   - 4 seconds is sufficient for the pendulum to swing from the bottom to the top, although it may still fall to the other side after reaching the highest point.

   Increased Training Speed:
   - Setting `max_steps_per_episode` to 200 significantly increased the training speed, which is a major advantage.
2) **Last Seven Models**: The last seven models were trained using different reward mechanisms and termination conditions to further optimize performance.

- Utilized Reward 1 and Termination 1 .
- Set `max_steps_per_episode` to 500. This provides more time for training the behavior after reaching the highest point.

The effect after training these eight models is as follows:



(a) Image 1 (b) Image 2 (c) Image 3

(d) Image 4 (e) Image 5 (f) Image 6

(g) Image 7 (h) Image 8 (i) Image 9

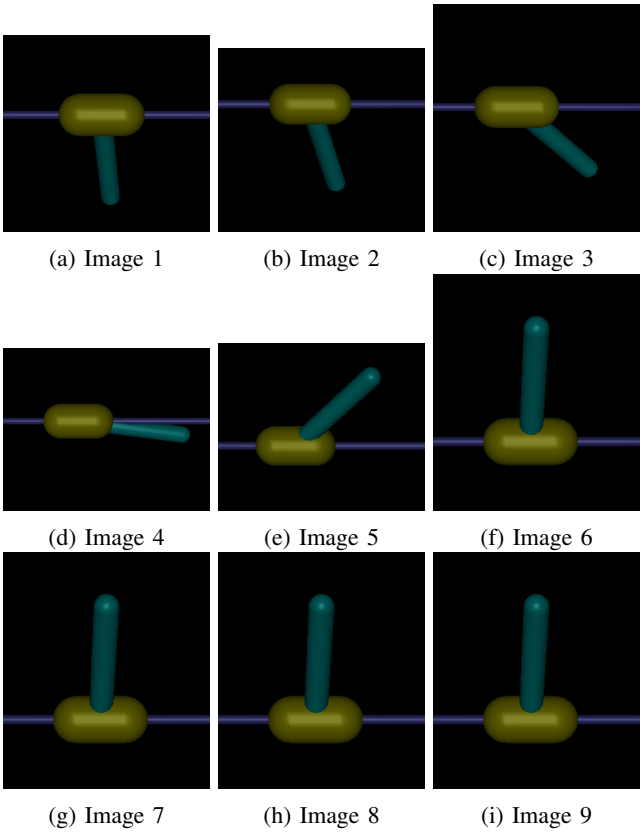
Fig. 2: Results of eight models

The model is capable of reaching the highest point, but it requires several swings back and forth. Additionally, after reaching the highest point, the pendulum continues to fall to the left.

2) Stable near the highest point: During the training process, the model not only needs to learn how to swing the inverted pendulum from the vertical downward position to the highest point, but also to maintain stability once it reaches the highest point. This stage of training is crucial as it determines whether the agent can successfully balance the inverted pendulum.

At this point, I recalled the reward mechanism used in the first phase. Although the reward in the first phase was a constant value of 1, unrelated to angle, position, or velocity, it worked very well in conjunction with the termination condition. Therefore, I decided to use the same reward and termination conditions from the first phase, specifically:

1) **Using Reward 3 and Termination 2**:
   - Reward 3 is a constant value of 1, unrelated to angle, position, or velocity, but effective when combined with the termination condition.
   - Termination 2 is more suitable for the stabilization phase.
2) **Setting `max_steps_per_episode` to 800**:
   - Although the termination condition is `(np.abs(ob[1]) > 0.2)`, we cannot continue training without terminating, otherwise we will not be able to update the learning.
   - The maximum number of steps per episode was set to 800 to accommodate the longer time needed for maintaining balance.



(a) Image 1 (b) Image 2 (c) Image 3

(d) Image 4 (e) Image 5 (f) Image 6

(g) Image 7 (h) Image 8 (i) Image 9

Fig. 3: Result of final model

G. conclusion

In this part, we successfully train an agent to control an inverted pendulum system from an initial vertical downward position to the highest point and to stabilize it near the highest point. The entire training process is divided into two main stages: First, we use reward 2 and termination 1, setting the maximum number of steps per round to 200 to quickly evaluate and adjust the model. Subsequently, we adjust to reward 1 and termination 1, increasing the maximum number of steps to 500 to accommodate more complex balance. Finally, when reaching the highest point and needing to maintain stability, we adopt reward 3 and termination 2, setting the maximum number of steps to 800 to further optimize the stability of the model.