EKF Estimation for the Biped Robot and Examination for Reinforcement Learning

- SDM366 Report for Final Project -

July 4, 2024

Zhiling Li
Department of System Design and Intelligent Manufacturing
Shenzhen, China
12112748@mail.sustech.edu.cn

Jinda Dong
Department of System Design and Intelligent Manufacturing
Shenzhen, China
12111829@mail.sustech.edu.cn

Zixuan Zhong
Department of System Design and Intelligent Manufacturing
Shenzhen, China
12112707@mail.sustech.edu.cn

1 Biped Robot State Estimation

In the first project, we need to deploy an Extended Kalman filter algorithm[1] that uses information from the motor and IMU to predict the body speed of the biped robot.

1.1 State Space Model of a biped robot

1.1.1 Filter State Definition

First, we need to define the state variable. According to the description of the paper[2], the state variables of the robot are defined as:

$$^0P_{com} \in \mathbb{R}^3, \quad ^0V_{com} \in \mathbb{R}^3, \quad ^0P_l \in \mathbb{R}^3, \quad ^0P_r \in \mathbb{R}^3 \tag{1}$$

The variables above represent: the position of the robot's center of mass in the world coordinates, the velocity of the robot's center of mass in the world coordinates, the position of the robot's left foot in the world coordinates, and the position of the robot's right foot in the world coordinate, sequentially.

The state of the robot is defined as x:

$$x := (^0P_{com}, \quad ^0V_{com}, \quad ^0P_l, \quad ^0P_r) \in \mathbb{R}^{12} \tag{2}$$

1.1.2 Prediction Model

In the process of moving forward, we first need to determine the relationship between the global position and speed of the robot. In the case of discrete time, if the time interval is δt, then the position at the next time is:

$$^0P_{com}(t+\delta t) = {}^0P_{com}(t) + {}^0V_{com}(t)\delta t \tag{3}$$

We can get the acceleration of the robot in body coordinates from the IMU:

$$^{0}V_{com}(t+\delta t) = {}^{0}V_{com}(t) + {}^{0}a(t)\delta t \tag{4}$$

where $^{0}a(t)$ is the acceleration term on the robot in the global coordinate. It can be measured with gravity acceleration, IMU acceleration, and rotation matrix from global to body coordinate:

$$^{0}a(t) = {}^{0}_{B}R(t)({}^{B}a(t) + g(t)) \tag{5}$$

In equation (5), $^{0}_{B}R(t)$ is the rotation matrix from global coordinate to body coordinate. $^{B}a(t)$ is the body acceleration which got from IMU, and $g(t)$ is the gravity acceleration.

For the feet' position in the global coordinate, we only use the leg, which stands on the ground, to do the estimation. In this case, let the tiptoe motionless:

$$^{0}P_{l}(t+\delta t) = {}^{0}P_{l}(t) \tag{6}$$

$$^{0}P_{r}(t+\delta t) = {}^{0}P_{r}(t) \tag{7}$$

According to equation(3)-equation(7), we can rewrite these equations to matrix form as prediction state:

$$x(t+\delta t) = Ax(t) + Bu(t) \tag{8}$$

$$A = \begin{bmatrix} \mathbf{I}_3 & \mathbf{I}_3\delta t & \mathbf{0}_3 & \mathbf{0}_3 \\ \mathbf{0}_3 & \mathbf{I}_3 & \mathbf{0}_3 & \mathbf{0}_3 \\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{I}_3 & \mathbf{0}_3 \\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{I}_3 \end{bmatrix}, \quad B = \begin{bmatrix} \mathbf{I}_3 \\ \mathbf{I}_3\delta t \\ \mathbf{0}_3 \\ \mathbf{0}_3 \end{bmatrix}, \quad u(t) = {}^{0}a(t) \tag{9}$$

1.1.3 Observation Model

Then, we need to use the information from the motor position and velocity to observe the body position. Firstly, we can use the feet' position under global coordinates and the motor's position to calculate the body's position under global coordinates. According to the coordinates transfer equation:

$$^{0}P_{l}(t) = {}^{0}P_{com}(t) + {}^{0}_{B}R(t)\,{}^{B}P_{l}(t) \tag{10}$$

Rewrite it in observation form:

$$-{}^{0}_{B}R(t)\,{}^{B}P_{l}(t) = {}^{0}P_{com}(t) - {}^{0}P_{l}(t) \tag{11}$$

Similarly, the right foot can measure too:

$$-{}^{0}_{B}R(t)\,{}^{B}P_{r}(t) = {}^{0}P_{com}(t) - {}^{0}P_{r}(t) \tag{12}$$

We also know the velocity of the motors, which can be used to observe the velocity of the body. Derive the equation(10):

$$^{0}V_{l}(t) = {}^{0}V_{com}(t) + {}^{0}_{B}\dot{R}(t)\,{}^{B}P_{l}(t) + {}^{0}_{B}R\,{}^{B}V_{l}(t) \tag{13}$$

The robot foot stands on the ground, which means ${}^0V_l$ equals to zero, and rewrite the equation as the observation form:

$$-{}^0_BR(t)({}^0\omega_B(t) \times {}^BP_l(t) + {}^BV_l(t)) = {}^0V_{com}(t) \tag{14}$$

There is the same equation on the right foot:

$$-{}^0_BR(t)({}^0\omega_B(t) \times {}^BP_r(t) + {}^BV_r(t)) = {}^0V_{com}(t) \tag{15}$$

In the equation above, ${}^0\omega_B(t)$ stands for the angular velocity of the body in terms of global coordinates, ${}^BP_l(t)$ stands for the position of the left foot under body coordinate, and ${}^BV_l(t))$ stands for velocity of the left foot under body coordinate.

Lastly, if the foot is on the ground, we can easily get that the height of the foot position, the third term in ${}^0P_l(t)$ and ${}^0P_r(t)$ are constant:

$$\begin{bmatrix}0 & 0 & 1\end{bmatrix} {}^0P_l(t) = \begin{bmatrix}0 & 0 & 1\end{bmatrix} {}^0P_r(t) = h \tag{16}$$

And we can get h from robot URDF file:

$$h = 0.037062 \tag{17}$$

Combining the equation(10)-equation(17), we write it in matrix form:

$$z(t) = Hx(t) \tag{18}$$

$$z(t) = \begin{bmatrix} -{}^0_BR\,{}^BP_l \\ -{}^0_BR\,{}^BP_l \\ -{}^0_BR({}^0\omega_B\,{}^BP_l + {}^B\dot{P}_l) \\ -{}^0_BR({}^0\omega_B\,{}^BP_r + {}^B\dot{P}_r) \\ h \\ h \end{bmatrix} \tag{19}$$

$$H = \begin{bmatrix} \mathbf{I}_3 & \mathbf{0}_3 & -\mathbf{I}_3 & \mathbf{0}_3 \\ \mathbf{I}_3 & \mathbf{0}_3 & \mathbf{0}_3 & -\mathbf{I}_3 \\ \mathbf{0}_3 & \mathbf{I}_3 & \mathbf{0}_3 & \mathbf{0}_3 \\ \mathbf{0}_3 & \mathbf{I}_3 & \mathbf{0}_3 & \mathbf{0}_3 \\ \mathbf{0}_{1x3} & \mathbf{0}_{1x3} & \begin{bmatrix}0 & 0 & 1\end{bmatrix} & \mathbf{0}_{1x3} \\ \mathbf{0}_{1x3} & \mathbf{0}_{1x3} & \mathbf{0}_{1x3} & \begin{bmatrix}0 & 0 & 1\end{bmatrix} \end{bmatrix} \tag{20}$$

1.2 Implement EKF as State Estimator

We have established the motion model and observation model of the robot. And now, we can implement the Extended Kalman Filter to estimate the body velocity.

prediction step

$$\hat{x}(t) = Ax(t) + Bu(t) \tag{21}$$

$$\hat{P(t)} = AP(t)A^\top + Q(t) \tag{22}$$

observation step

$$K(t) = \hat{P}(t)H^\top(H\hat{P}(t)H^\top + R(t))^{-1} \tag{23}$$

$$x(t+\delta t) = \hat{x}(t) + K(t)(z(t) - H\hat{x}(t)) \tag{24}$$

$$P(t+\delta t) = (I - K(t)H)\hat{P}(t) \tag{25}$$

It is worth mentioning that both the motion model and the observation model here are based on the assumption that the foot is standing on the ground. When this assumption is satisfied, we can use the above model to predict the current body speed. But when a foot is off the ground, we give a large value to the variance of the corresponding terms of the Q and R matrices. In this case, the foot on the ground will have a slight effect on estimating velocity, while the model will tend to use the foot on the ground to do estimation.

1.3 Experiment Result and Extras

1.3.1 Experiment Results

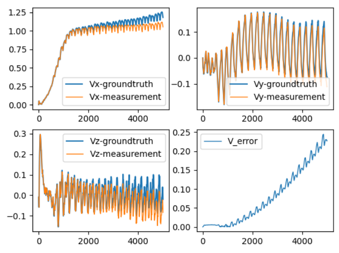
Since the foot has a certain speed in the world coordinate system when it just lands and is about to lift up, the observation of the body speed is inaccurate. Therefore, in practice, we focus on using IMU information for measurement, which means the Q matrix is smaller than the R matrix.

$$Q(t) = diag(I_3\delta t, 9.8I_3\delta t, I_3\delta t * noise_{Ql}, I_3\delta t * noise_{Qr}) \tag{26}$$

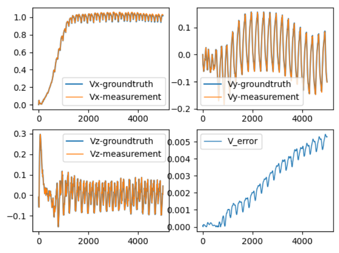
$$R(t) = diag(I_6 * noise_{pfoot}, I_6 * noisevfoot, I_1 * noise_{Rl}, I_1 * noise_{Rr}) \tag{27}$$

In the equations above, to make the IMU measurement more credible, we take the $noise_{Ql}, noise_{Qr} < noise_{Rl}, noise_{Rr}$.

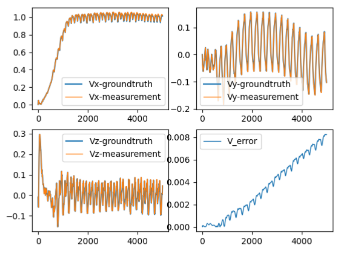
Here are the experiment results of different Q and R matrix. With all figures shows below, yellow line stands for the real body velocity under global coordinate, while the blue line stands for estimate velocity under global coordinate. The figures demonstrate the x-velocity, y-velocity, z-velocity, and error among velocities, from top left, top right, bottom left, bottom right, sequentially.



(a) $noise_R = 1e5 * noise_Q$



(b) $noise_R = 1e7 * noise_Q$



(c) $noise_R = 1e10 * noise_Q$

Figure 1: Comparison of Measurements with Different Noise Levels

We can see that with R gets larger, the error of the estimate gets smaller, and the predicted speed gets closer to the real speed.

This is due to the fact that estimation step requires information about the robot's foot contact with the ground compared to prediction step. However, in this simulation, the information of the robot's foot touching the ground is only 1 and 0, and there is no strength information. The speed of the motor is not accurate between the time when the robot is about to lift its feet and the time when it just lands, but since the legs have already landed, this inaccurate information will be used to estimate the speed of the body. In addition, the estimation step has large noise compared to prediction step, so the error gets smaller with R gets larger.

This trend has not always held true. Since the IMU information of the prediction step itself has errors, the total error of the system will gradually increase with the increase of time, that is, the predicted position will become more and more inaccurate with the real position. The velocity of the system divergence mainly depends on the size of the IMU noise and the value of the Q&R matrix.

In the above experiments, the robot could walk normally at the beginning, but when the distance gradually increased and the prediction of speed and position was no longer accurate, the robot's posture gradually deformed and eventually fell to the ground. In the experiment, the distance traveled by the robot is inversely proportional to the error of speed prediction. The smaller the error, the farther the robot will go. In the above three experiments, the robot went the closest in the first experiment, and the robot went the farthest in the third experiment, which also proves this conclusion.

1.3.2 Extra

There is a bug in the demo, that is, the time steps of the predicted results do not match. In the source code you want to take an action after each `self.decimation` time step. However, the measurement step is also incorrectly written in the judgment of `self.decimation`, that is, the measurement will also be performed after the `self.decimation` time step. Therefore, if this measurement is followed, the actual time step of the measurement needs to be changed to the `self.decimation` of the defined time step. Another solution is to remove this line of code from the `self.decimation` judgment.

```python
while self.data.time < 5.0 and self.viewer.is_running():
    step_start = time.time()
    proprioception_obs = self.compute_obs() # correct position
    if self.iter_ % self.decimation == 0:
        # proprioception_obs = self.compute_obs() # wrong position
```

Another thing is that, given the simplicity of directly using the pinocchio library to calculate the forward kinematics, we hand-modeled the DH table using our knowledge of the robot arm kinematics and the robot URDF file. After testing, the position can also be predicted using our DHtable, but due to the large number of design matrices during calculation and the calculation is time-consuming, it is still recommended to use the pinocchio library.

You could set `withPinocchio` term in the code to `False` to check the difference.

2 Reinforcement Learning in Inverted Pendulum

2.1 Problem Formulation

This project involves the application of Reinforcement Learning (RL) to control inverted pendulum systems. The inverted pendulum is a classic problem in control theory and robotics, characterized by its inherent instability and requirement for continuous balancing. The project is divided into two main tasks: controlling a single inverted pendulum and a double inverted pendulum.

2.1.1 Single Inverted Pendulum

The single inverted pendulum consists of a rod pivoted at one end and a cart that can move horizontally. The objective is to control the movement of the cart to keep the pendulum balanced in the upright position.

In task 1a: **Upright Stabilization**, the pendulum starts with a slight inclination from the upright position. The goal is to stabilize the pendulum by controlling the cart's motion. This involves applying forces to the cart to counteract the pendulum's tendency to fall.

In task 1b: **Swing-up and Stabilization**, the pendulum starts from the downward position. The objective is to swing the pendulum up and stabilize it in the upright position. This requires a more complex control strategy as the pendulum needs to gain sufficient momentum to reach the upright position and then be stabilized.

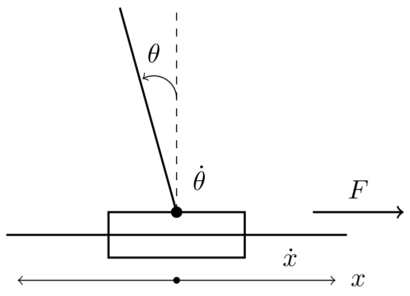


Figure 2: Block diagram of the single inverted pendulum system. The diagram shows the cart, the pendulum rod, and the forces applied to control the system.

2.1.2 Double Inverted Pendulum (Bonus)

The double inverted pendulum consists of two rods connected end-to-end, creating a system with higher degrees of freedom and more complex dynamics. The goal is to **swing and stabilize** the double pendulum from a downward initial position to an upright position.

The objective is to control the movement of the cart to swing both rods up and maintain them in a given state. The increased complexity of the double pendulum system makes this task more challenging due to its non-linear dynamics and the need to coordinate the motion of both rods.

2.1.3 Objective and Challenges

The primary objective for both single and double inverted pendulums is to develop RL-based control policies that can achieve stable balancing. The challenges include:

- Defining appropriate state and action spaces.
- Designing a reward function that encourages stable balancing.
- Selecting and implementing effective RL algorithms.

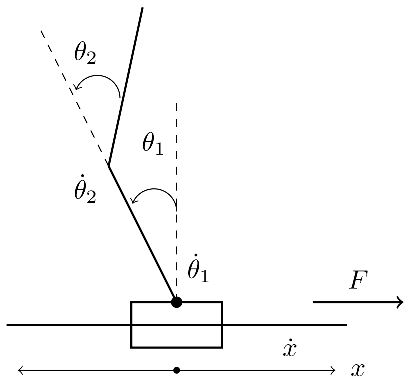


Figure 3: Block diagram of the double inverted pendulum system. The diagram shows the cart, the pendulum rods, and the forces applied to control the system.

- Training the RL agent to handle the non-linear dynamics of the pendulum systems.

To solve these tasks, the following steps are necessary:

1. **State and Action Spaces**: Define the observations and actions for the RL agent. For the inverted pendulum, the state typically includes the angle and angular velocity of the pendulum and the position and velocity of the cart. The action space involves the forces applied to the cart.
2. **Reward Function Design**: Design a reward function that provides positive feedback for actions that lead to stable balancing and penalizes actions that lead to instability.
3. **Algorithm Selection**: Choose suitable RL algorithms such as PPO, DDPG, A2C or others to train the control policies.
4. **Training Procedure**: Implement the RL algorithms and train the agent using simulations of the inverted pendulum systems.
5. **Evaluation and Tuning**: Evaluate the performance of the trained policies and fine-tune the parameters to improve stability and robustness.

2.1.4 State and Action Spaces Description

In reinforcement learning (RL), defining the state and action spaces is critical for the agent to learn and perform the required tasks effectively. The state space provides the RL agent with the necessary information about the environment, while the action space defines the set of possible actions the agent can take.

For the inverted pendulum systems, the state and action spaces are defined as follows:

Single Inverted Pendulum For the single inverted pendulum, the state space typically includes:

- **Cart Position** (x): The horizontal position of the cart.
- **Pendulum Angle** (θ): The angle of the pendulum from the vertical position.
- **Cart Velocity** ($\dot{x}$): The velocity of the cart.
- **Pendulum Angular Velocity** ($\dot{\theta}$): The angular velocity of the pendulum.

These state variables provide a comprehensive description of the system's dynamics, allowing the RL agent to make informed decisions.

The action space for the single inverted pendulum is `Box(-3.0, 3.0, (1,), float32)`, and usually consists of:

- **Force (F)**: The horizontal force applied to the cart. This force can be positive or negative, corresponding to pushing the cart to the right or left.

Double Inverted Pendulum (Bonus) For the double inverted pendulum, the state space is more complex due to the additional degree of freedom. The state space includes:

- **Cart Position (x)**: The horizontal position of the cart.
- **Cart Velocity ($\dot{x}$)**: The velocity of the cart.
- **First Pendulum Angle (θ_1)**: The angle of the first pendulum from the vertical position.
- **First Pendulum Angular Velocity ($\dot{\theta}_1$)**: The angular velocity of the first pendulum.
- **Second Pendulum Angle (θ_2)**: The angle of the second pendulum from the vertical position.
- **Second Pendulum Angular Velocity ($\dot{\theta}_2$)**: The angular velocity of the second pendulum.

Additionally, we add two dims of observation, the `sin` and `cos` values of each θ, to better observe the state. Next, the action space for the double inverted pendulum is `Box(-1.0, 1.0, (1,), float32)`, and also consists of:

- **Force (F)**: The horizontal force applied to the cart. This force controls the movement of both pendulums indirectly through the cart's motion.

Defining these state and action spaces accurately is essential for the RL agent to learn effective control policies. These spaces capture the system's dynamics and provide the necessary feedback for the agent to optimize its actions and achieve stable balancing.

2.2 Reinforcement Learning Approach

In classic reinforcement learning, the environment is formulated as a Markov decision process described by the tuple (S, A, p, r, γ) with S to be the state space, A to be the action space, p to be the state transition probability, r to be the obtained reward and γ to be the discount factor [3]. The process of the Markov is shown in (28), where $P_{ss'}$ is the state transition probability matrix.

$$P_{ss'} = P[S_{t+1} = s'|S_t = s] \tag{28}$$

The Bellman equation reveals the recursive relationship between cost function and strategy in reinforcement learning [4], described as (29), where $\pi(s)$ represents the probability distribution in state s choosing strategy a, and $R(s, a)$ stands for the current reward, s' is the new state add γ is the discount factor.

$$V(s) = E_{a\ \pi(s)}[R(s, a) + \gamma V(s')] \tag{29}$$

In order to find the policy $\pi(a|s) = P[A_t = a|S_t = s]$ that can maximize the reward, we follow the basic MDP (Markov Decision Process) process [5] to implement the reinforcement learning in our model. Q-learning is a model-free method widely used in solving the MDP problems in reinforcement learning [6], using the state-action value function (Q function). We follow the Q-learning method to choose strategies that can maximize the final reward in (30) to calculate the expected rewards Q in a given state s to s' corresponds to taking the action a. Here α is the learning rate, a hyper-parameter.

$$\begin{aligned} Q^{new}(s, a) =& (1 - \alpha)Q(s, a) \\ &+ \alpha(R(s', a) + \gamma \max Q(s', a)) \end{aligned} \tag{30}$$

2.3 Training Methods

2.3.1 Algorithm Selection and 'Throw-Catch' Method

In this project, we tested various reinforcement learning algorithms to train the control policies for the inverted pendulum systems. The algorithms can be grouped based on their suitability for discrete or continuous action spaces, with PILCO being a special case due to its unique approach.

Algorithms for Discrete Action Spaces

- **Deep Q-Network (DQN)**: DQN approximates the Q-values using a neural network, employing experience replay and target networks for stability. It is effective for discrete action spaces and well-studied. However, DQN is not directly applicable to continuous action spaces and may require discretization of the action space. Therefore, it is less suitable for our continuous-time environment without significant modifications.

For algorithms for discrete action spaces, Deep Q-Network (DQN) is specifically designed, where the Q-values are approximated using a neural network. To apply DQN in a continuous action space, we discretized the action space and assigned discrete values to approximate continuous actions.

While this approach allowed us to use DQN for our experiments, we observed that the performance was suboptimal. Specifically, during the stabilization process, the control exhibited noticeable oscillations and instability. This is because the discretized actions cannot capture the precise control needed for smooth operation in continuous environments, leading to less effective performance compared to algorithms natively designed for continuous action spaces.

Algorithms for Continuous Action Spaces

- **Advantage Actor-Critic (A2C)**: A2C is a synchronous version of the Actor-Critic method, where the actor updates the policy, and the critic estimates the value function. The advantage function helps to reduce variance and stabilize training. A2C is simple to implement and stable in training, making it suitable for continuous action spaces. However, synchronous updates can be slower compared to asynchronous methods. It is well-suited for continuous-time environments without the need for action space modification.
- **Asynchronous Advantage Actor-Critic (A3C)**: A3C runs multiple instances of the environment in parallel with separate actors, which asynchronously update the global policy and value function. This approach allows for better exploration and faster learning, making it effective for large-scale problems. However, it has a more complex implementation and higher computational requirements. A3C is effective for continuous-time environments without needing action space modifications.
- **Deep Deterministic Policy Gradient (DDPG)**: DDPG combines the actor-critic architecture with deterministic policy gradients, using experience replay and target networks to stabilize training. It handles continuous action spaces well and is efficient in learning through experience replay. However, DDPG can be sensitive to hyperparameters and may require careful tuning. It is highly suitable for continuous-time environments with continuous action spaces.
- **Proximal Policy Optimization (PPO)**: PPO uses a clipped objective function to prevent large policy updates, improving training stability and efficiency. It is robust, high-performing, and simpler than TRPO. While it requires on-policy updates, which can be less data-efficient, it is well-suited for continuous-time environments and works well with continuous action spaces.

For algorithms with continuous action spaces, A2C, DDPG, and PPO demonstrated effective control performance. However, each algorithm showed different strengths and weaknesses depending on the specific tasks in `'throw-catch'` process.

In our project, we transformed the `'swing-up'` task into a `'throw-catch'` process using two agents working in collaboration. Each agent was responsible for a different subtask. We tested various models and hyperparameters under the same reward function to evaluate their performance.

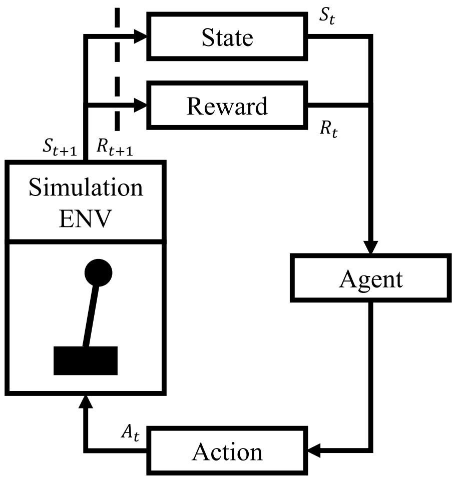
The results indicated that different algorithms excelled in different sub-tasks. For instance, DDPG provided smooth control for the initial throw phase, while PPO excelled in the catch phase due to its robust policy updates. By systematically comparing the performance of each algorithm, we selected the most suitable agent for each subtask, ensuring optimal overall performance for the throw-catch model.

Special Algorithm

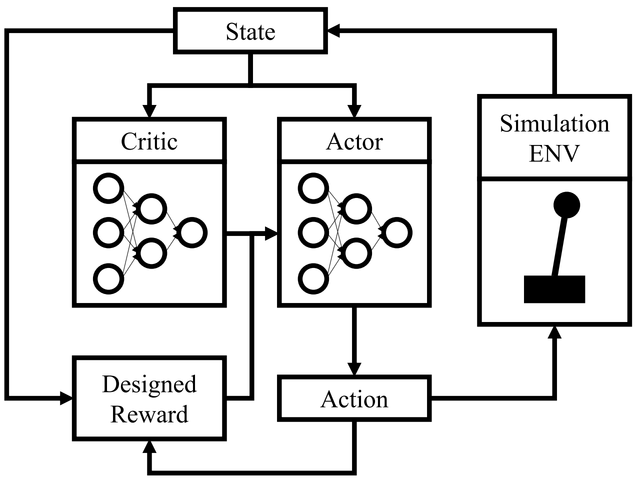
- **Probabilistic Inference for Learning Control (PILCO)**: PILCO uses Gaussian processes to **model the system dynamics** and performs probabilistic inference to optimize the policy, achieving high data efficiency. It is highly data-efficient and **effective** with limited training data. However, PILCO is computationally intensive and has a complex implementation, for we have to simulate relatively precise model, or we have to use supervised learning method to reflect the relationship between the current input and the corresponding output state.Generally, it is suitable for continuous-time environments and handles continuous action spaces well.

All of the algorithm are implemented manually by ourselves based on pytorch, including a lot of parameter adjustment and trial and error links.

Traditionally, a model-free training process for reinforcement learning can be described in Fig. 4a. To improve performance in the continuous domain, the actor-critic method has been proven to be highly efficient and low-cost as one of the model-free algorithms in reinforcement learning.[7] Based on the actor-critic model, several models have been tried by us, including the A2C, A3C, DDPG, etc.



(a) Reinforcement learning.



(b) Actor-Critic algorithm.

Figure 4: Flow chart for the reinforcement learning framework and the actor-critic algorithm. **(a) Reinforcement Learning Framework:** This diagram shows the reinforcement learning process where the agent interacts with the environment. The agent receives the current state S_t and selects an action A_t. The environment returns a new state S_{t+1} and a reward R_{t+1}. This feedback loop continues, allowing the agent to learn optimal actions over time. **(b) Actor-Critic Algorithm:** This diagram illustrates the actor-critic algorithm. The actor selects actions based on the current state, while the critic evaluates these actions by estimating future rewards. The critic's feedback helps the actor adjust its policy for better performance. The agent interacts with the environment iteratively, refining its actions through this dual-component system.

2.4 Strategies

In this section, we will discuss the various strategies employed in our project to achieve optimal performance in controlling the inverted pendulum systems. These strategies include the creation of a custom RL agents toolkit, the implementation of a collaborative model for the swing-up task, and the design of effective methods for unwrapping, initializing the starting state, and defining episode end conditions. By detailing these strategies, we aim to provide a comprehensive overview of the techniques and methodologies that contributed to the success of our project.

2.4.1 Swing-Up Task and Stabilizing Task

To tackle the challenging swing-up task, we implemented a collaborative `'throw-catch'` process using two agents working in coordination. In this model, the first agent (the thrower) is responsible for generating the required state and swinging the pendulum up to the high at a certain velocity within the constraints. The second agent (the catcher) takes over to stabilize the pendulum in the upright position. This approach allowed us to decompose the complex task into manageable sub-tasks, each handled by a specialized agent.

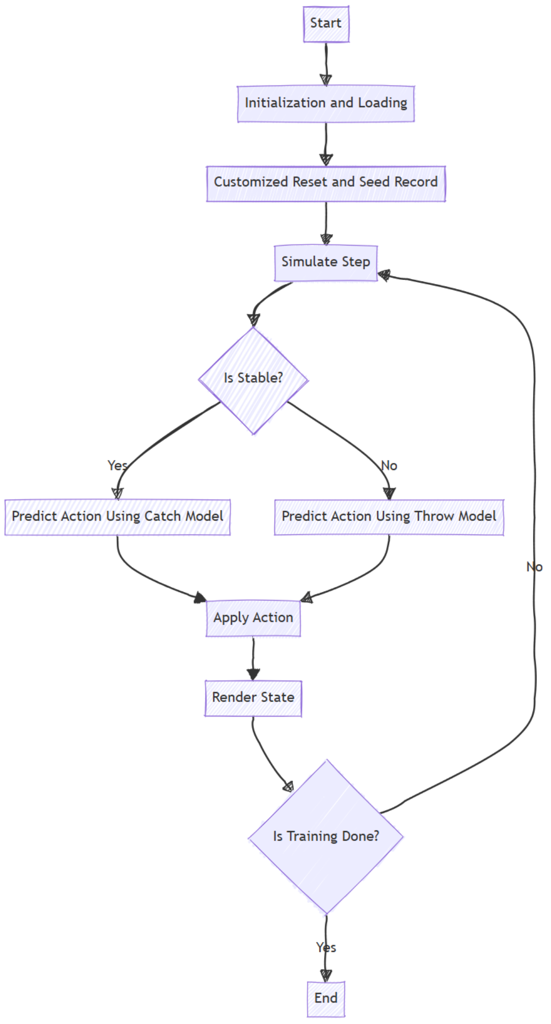

Start
Initialization and Loading
Customized Reset and Seed Record
Simulate Step
Is Stable?
Yes
No
Predict Action Using Catch Model
Predict Action Using Throw Model
Apply Action
Render State
Is Training Done?
No
Yes
End

Figure 5: Throw-catch method for swing-up task.

The thrower agent was trained to maximize the probability of swing-up high, ensuring that the pendulum gained sufficient height and maintained relative stationarity. Various reinforcement learning algorithms, such as DDPG and PPO, were tested to determine the most effective approach for this phase. The catcher agent, on the other hand, focused on stabilizing the pendulum once it reached the upright position. We found that algorithms with robust policy updates, such as PPO, performed exceptionally well in maintaining stability during this critical phase.

By systematically testing different models and hyperparameters, we were able to fine-tune each agent's performance. The collaborative effort between the two agents resulted in a successful swing-up and stabilization process, demonstrating the effectiveness of the method in handling complex dynamic tasks.

2.4.2 Agents Toolkit Preparation and Design

In this project, we developed a custom RL agents toolkit instead of using existing libraries. This allowed us to have full control over the implementation and fine-tuning of various RL models. The toolkit includes the following components:

First, we implemented several agent classes for different RL algorithms such as DDPG, A2C, and DQN. Each agent class includes the necessary methods for initialization, model loading and saving, resetting, updating, and predicting actions. The initialization method sets up the neural network architecture, optimizers, and any other necessary parameters. Methods to load pre-trained models and save the current state of the model for future use were also included. The reset method resets the agent's internal state and environment to start a new episode. The update method applies the RL algorithm's update rules to train the model based on observed data. Finally, the `predict` method generates actions based on the current state of the environment using the trained model.

```python
# Agent classes for different RL algorithms
class A2CAgent # Or other RL algorithms
    def __init__(self, obs_space_dims, action_space_dims, lr, gamma):
        # Initialize neural networks, optimizers, parameters
        pass

    def load_model(self, path):
        pass

    def save_model(self, path):
        pass

    def update(self, rewards, log_probs, states):
        # Update model based on observed data
        pass

    def sample_action(self, state):
        # Generate actions based on current state
        pass
```

For each agent, we developed corresponding network classes. These classes define the architecture of the neural networks used by the agents, such as policy networks, value networks, and Q-networks. The network classes also include methods for forward propagation and loss computation.

```python
# Network classes for defining neural network architectures
class PolicyNetwork:
    def __init__(self):
        # Define policy network architecture
        pass
    def forward(self, x):
        # Forward propagation
        pass

class ValueNetwork:

class QNetwork:

# And other classes
```

We created functions to handle data observations from the environment. These functions preprocess the raw data, normalize it if necessary, and format it into a suitable shape for the neural networks.

```
# Data observation functions
def get_obs(raw_data):
    # Preprocess and normalize raw data
    pass
```

Functions to initialize the Mujoco environment were also developed. These functions include setting up the simulation parameters, loading the model, and configuring the environment for interaction with the agents.

```
# Mujoco initialization functions
def init_mujoco(path):
    # Set up simulation parameters and load model
    pass
```

Additionally, we implemented functions to record training data, such as rewards, losses, and episode lengths. These functions help in tracking the training progress and analyzing the performance of the agents.

Finally, visualization tools were developed to plot training metrics, such as reward curves, loss curves, and other relevant performance indicators. These tools are crucial for monitoring the training process and identifying any issues or areas for improvement.

The custom toolkit provided us with the flexibility to experiment with different RL algorithms and tailor them to the specific requirements of our project.

2.4.3 Starting State

The starting State varies in single or double inverted pendulum, `thrower` agent or `catcher` agent situations:

Single Inverted Pendulum: For the `thrower` agent, after resetting, all observations start in the state (0.0, 0.0, 0.0, 0.0) with a uniform noise in the range of [-0.1, 0.1] added to the positional values (cart position and pole angles) and standard normal force with a standard deviation of 0.1 added to the velocity values for stochasticity. After resetting and adding noise, the theta will be subtracted by `np.pi` to simulate the slumping state. For the `catcher` agent, after resetting data, all states are changed. A random number is generated for each value within the respective limits to mimic the range of reception that the agent needs to undertake, such as position, angle, velocity, angular velocity, etc.

Double Inverted Pendulum: Similarly to the single-bar inverted pendulum, for models that take on different tasks, a reset is performed, followed by a random initialization over a range of different settings. During this initialization, a random seed can be specified.

2.4.4 Episode End

In reinforcement learning, defining the episode's end conditions is crucial for the training process. An episode ends when certain criteria are met, and these criteria can significantly impact the learning efficiency and effectiveness. For the inverted pendulum systems, we established specific conditions to mark the end of an episode:

First, for the `stabling` task, an episode ends if the pendulum falls beyond a certain angle threshold from the vertical position. This prevents the agent from continuing to make futile attempts at recovery when the pendulum is too far off balance, thus focusing the learning on more likely successful scenarios.

Second, an episode might be ended if the cart reaches the boundary of the track. The inverted pendulum operates on a finite track, and reaching the boundary indicates that the agent needs to learn to stabilize the pendulum without excessive horizontal movement.

Third, we introduced a maximum time step limit for each episode. This ensures that the agent's training is not dominated by a few long episodes but instead by many shorter, diverse attempts. This time-step limit helps maintain a balance between exploration and exploitation, allowing the agent to experience a variety of states and actions within a controlled timeframe.

These episode end conditions are designed to provide meaningful feedback to the agent, encouraging learning strategies that maintain the pendulum's balance and control within the given constraints. By clearly defining the end of an episode, we ensure that the agent receives consistent and relevant experiences, leading to more stable and effective learning outcomes.

2.4.5 Unwrapping

In particular, in order to allow the model to rotate freely, we introduced customized models and environments, and in order to unwrap the angles, we defined an extended observation space, adding sine-cosine quantities for each angle in order to compute the value of the angle over the course of a week, which is used to compute the rewards correctly.

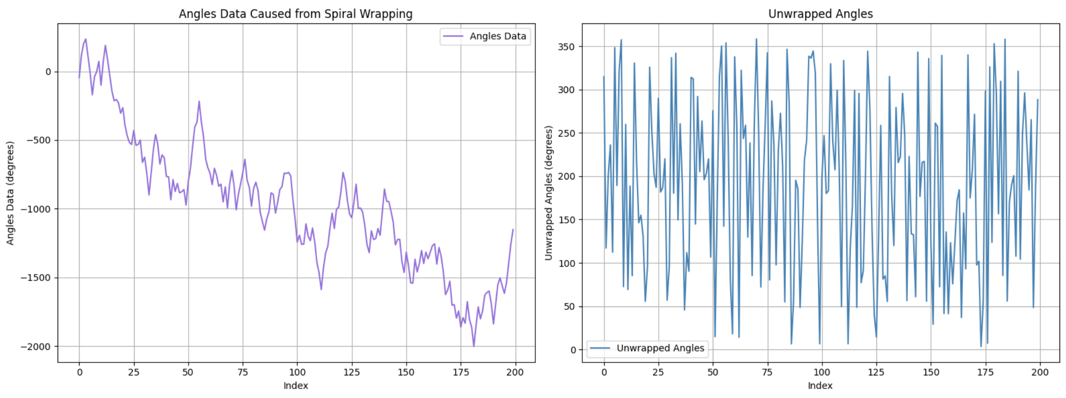


Figure 6: Unwrap the spiral angles.

2.5 Results and Discussion

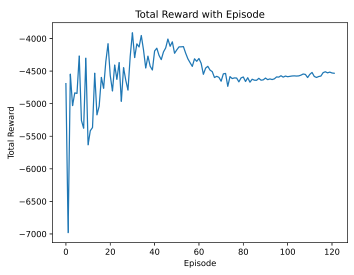
2.5.1 Single Inverted Pendulum

To complete the tasks, we tried to divide the tasks into sub-tasks.

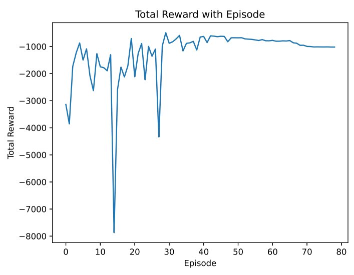
In fact, swing up the pendulum and then keep balance is really a hard task, so we divide the task into 2 parts: the swing-up task and stabilizing task. In the swing-up task, our goal is to swing up the pendulum; and another task is to keep balance of the pendulum. Hence, in the swing-up task, we only need to swing it up, and try to make it less unstable when it reaches its peak, and the apply the stabilizing task to keep balance.

In the training process of the swing-up task, we divide the task into to part: the first task is to swing the pendulum as high as it can, and the second one is to lower the speed when reaching the highest point. We define 2 reward functions for the whole swing-up task, and the change of the reward is shown in Fig. 7.

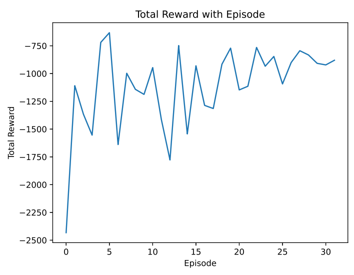
And for the stabilization task, because in the swing-up task, the pendulum often reaches the top with a relatively high velocity, to achieve better performance, we increased the initial condition noise in the stabilization inverted pendulum training task. However, directly adopting high noise levels for training might not yield perfect fitting results for the inverted pendulum due to the excessive noise. Therefore, we divided the task into three sub-tasks, progressively increasing the training noise. The initial conditions for the three tasks are provided in the first three rows of Table. 4, all using the same reward function. The total training comprises 50,000 episodes, with each episode simulating up to a maximum of 15 seconds. The relationship between the reward function and the training episodes is shown in Fig. 7.



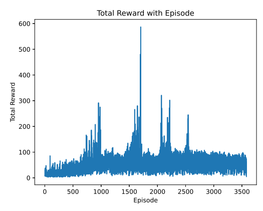
(a)



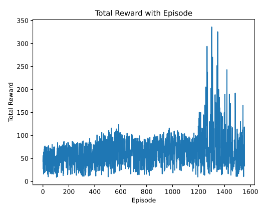
(b)



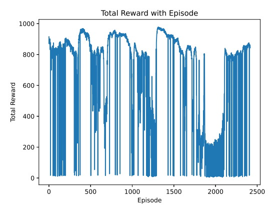
(c)



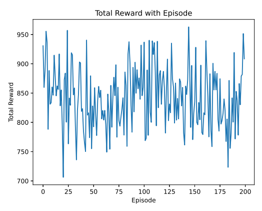
(d)



(e)



(f)



(g)

Figure 7: Rewards for inverted pendulum swing-Up and stabilization tasks. **The first row** illustrates the reward function variations during the swing-up task. Figure (a) shows the trend of the reward function over the initial 120 training episodes for the swing-up task. Figure (b) presents the reward function trend over the subsequent 80 training episodes for the swing-up task. Figure (c) displays the reward function trend after modifying the reward function to reduce the speed at which the pendulum reaches the highest point. **The second row**, from left to right, depicts the reward function variations during the stabilization task training. Figure (d) shows the reward function trend at the beginning of the stabilization task training. Figure (e) and Figure (f) illustrate the reward function trends during the intermediate stages of the stabilization task training. Figure (g) presents the reward function trend towards the end of the stabilization task training.

Task 1a: Upright Stabilization Results Using the A2C method, our inverted pendulum was successfully stabilized and remained predominantly centered on the rod. For each episode, the agent we trained using the algorithmic class we wrote ourselves was able to control the pendulum to be almost motionless and not fall over.

Additionally, we finished the task by testing the DQN method and PPO agent. As mentioned before, the action output from the DQN method is affected by the frame rate and produces some jitter, while the PPO training time is rapid and effective.

Task 1b: Swing-up and Stabilization Results Using the DDPG algorithm, we successfully swung the inverted pendulum from a stationary position to the top. Subsequently, we applied our stabilization model, achieving a transition from swing-up to stabilization in under **5 seconds**.

We conducted 20 tests of the inverted pendulum swing-up from a **stationary state**, with a simulation time of 10 seconds. In each test, the pendulum successfully swung up and stabilized with a success rate of **100%**. Additionally, we introduced some noise to the pendulum, with noise parameters detailed in Table 1. Under these conditions, 18 out of 20 tests successfully stabilized, achieving a success rate of 90%. When we elongate the simulation time to 30 seconds, the swing-up success rate under noisy conditions reached **100%**.

We provide videos for the task.

Table 1: Noise for the swing-up process.

x	θ	$\dot{x}$	$\dot{\theta}$
U(-0.5,0.5)	U($-\pi$,π)	U(-2.0,2.0)	U(-2.0,2.0)

2.5.2 Double Inverted Pendulum (Bonus)

Although the model-free reinforcement learning shows a good adaptability for solving simple problems, for the inverted double pendulum which is an underactuated system with deterministic chaotic behavior[8], it is not easy to perform well when applying simple models. Aiming to better the performance of the model, we try to apply model-based reinforcement learning, to better the performance of the swing-up task.

The same as the inverted pendulum, we divide the whole task into swing-up task and stabilization task. We use A2C for stabilization, and DDPG for swing-up. The training results are shown in Fig. 8.

Swing-up and Stabilization Results

2.6 Conclusion

Both of the basic tasks are finished successfully, and perform extremely well. And for the inverted double pendulum, we successfully stabilize it. And we provide videos for the results. However, we fail to swing up the inverted double pendulum based on the model-free learning algorithm. How to solve it?

2.7 Challenging and Enhancement

2.7.1 Learning to Control: A Model-Based Strategy

In the previous portions of this report, we provide reinforcement learning models like DQN, A2C, and DDPG that are model-free. However, these reinforcement learning models sacrifice applicability to specific scenarios in order to have broad universality. So what if we learn how to control the inverted double pendulum by studying the observation state? Initially, when I came to this idea, I'd like to call it an "Ethy Algorithm." However, after examining the information for a long, this algorithm, called the Probabilistic Inference for Learning Control (PILCO) algorithm, shows an extremely good performance in the inverted double pendulum system.[8]

For the double inverted pendulum system, we can write the universal state transmission equation (31):

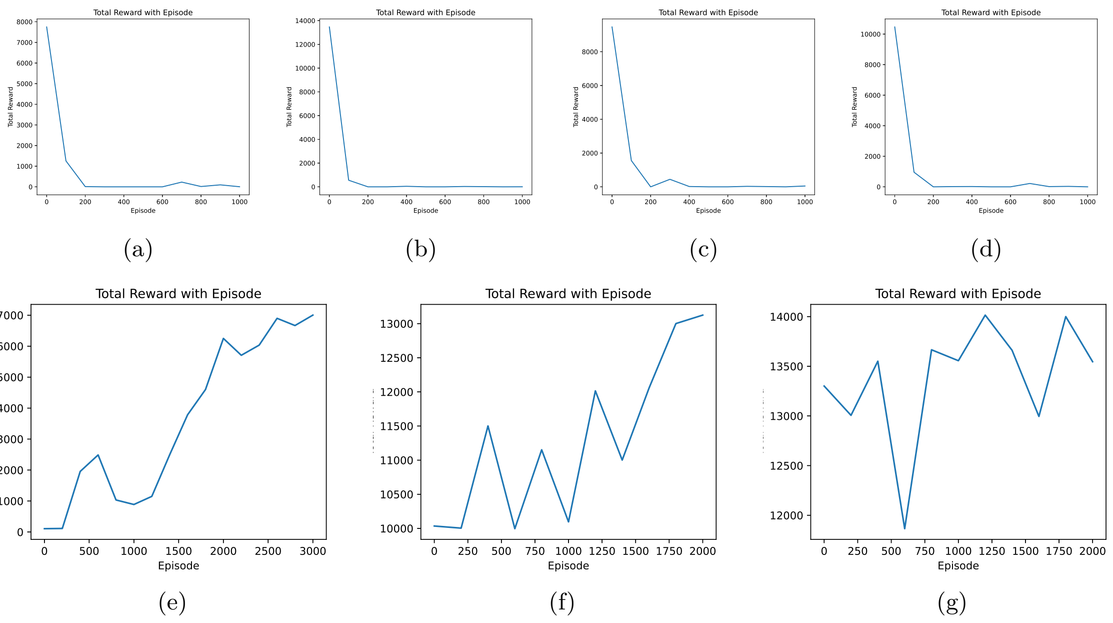


Figure 8: Reward for swing-Up and stabilization tasks of the inverted double pendulum. **The first row** of four images (a), (b), (c), and (d) demonstrates the results of using the DDPG algorithm for the swing-up task. The reward functions in these plots ultimately decline to the lowest point, indicating that the swing-up task was not successfully achieved using the DDPG algorithm. **The second row**, from left to right, shows the reward function trends during the training process for stabilizing the second-order inverted pendulum. Image (e) represents the beginning of the training, image (f) shows the intermediate stages of training, and image (g) illustrates the reward function trend towards the end of the training.

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned} \tag{31}$$

but it is really hard to establish a model for the inverted double pendulum mathematically like (31), and the simulation environment is not the same as the real word, so we're going to use a multi-layer perceptron to fit the relationship between `next_state` and `state, action`, which is also a Gaussian Process.

And we only use the state for calculating the reward, so we directly pass the reward function to the model, so that it will not need to fit the relationship between `state` and `reward`. The design of the inner model is shown in Fig. 9. To train the multi-layer perceptron, we randomized some states and actions and used `mujoco.mj_step` to calculate the new states as the labels. The total sample is 2×10^5 in size, with 0.001 to be the learning rate and 2000 to be the number of the iteration.

And here, we provide our training rewards in Fig. 10. The graph demonstrates a satisfactory performance of the reward function, as it ultimately converges to approximately -200. Notably, our observations indicate that, although the inverted double pendulum did not achieve a stable state, it reached its peak position with relative frequency and at a comparatively low speed. This suggests that the swing-up task was successfully completed.

Finally, we combined the swing-up task with the stabilization task to evaluate the overall performance. After seven rounds of simulations, each lasting 25 seconds, we observed that the best result (also the only result), that the inverted double pendulum successfully swung to a stable position ($y > 1.0$) and remained stable for nearly 7 seconds.

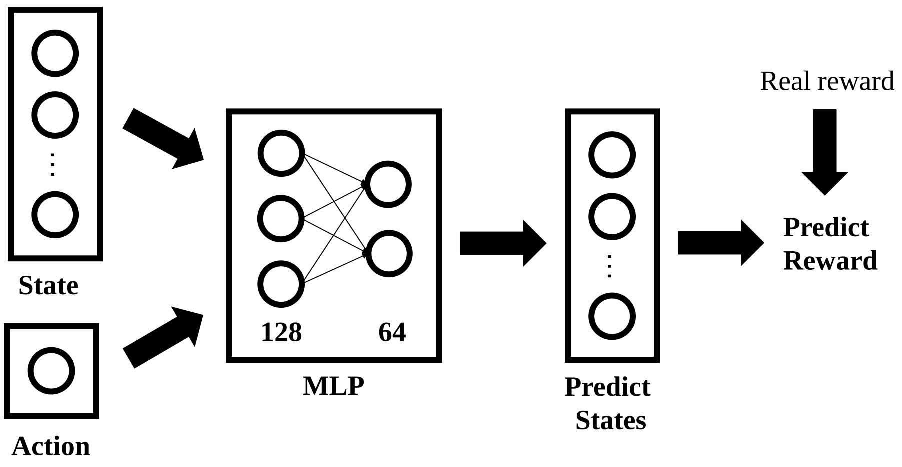


Figure 9: The design of our PILCO reinforcement learning framework for the inverted double pendulum. The process begins with the current state and action, which are fed into a Multi-Layer Perceptron (MLP) consisting of layers with 128 and 64 neurons. The MLP, which has been trained, predicts the next state of the system, which is then used to compute the predicted reward. The real reward is obtained from the actual system's response and is compared with the predicted reward to update the MLP model. This iterative process refines the model to optimize the control policy for the inverted double pendulum problem. This model illustrates the relationship among state, action and reward clearly, so we apply this model to the reinforcement algorithm to better predict future rewards, and decrease the variance of the RL model.

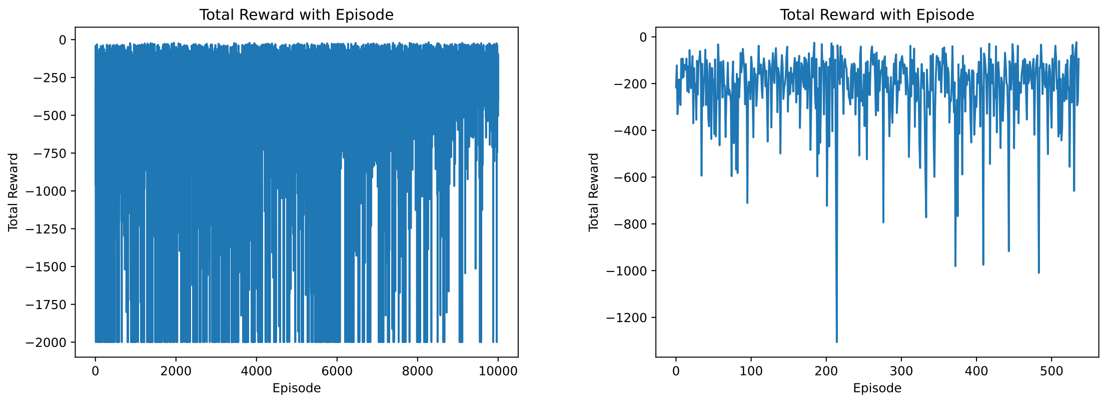


Figure 10: Reward for the swing-up task of the inverted double pendulum. **Left side** shows the rewards at the beginning of the training. **Right side** demonstrates the rewards at the end of the training.

References

[1] M. Blösch, M. Hutter, M. A. Höpflinger, S. Leutenegger, C. Gehring, C. D. Remy, and R. Y. Siegwart, "State estimation for legged robots - consistent fusion of leg kinematics and imu," in *Robotics: Science and Systems*, 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:1367000

[2] M. Bloesch, C. Gehring, P. Fankhauser, M. Hutter, M. A. Hoepflinger, and R. Siegwart, "State estimation for legged robots on unstable and slippery terrain," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013, pp. 6058–6064.

[3] J. R. Norris, *Markov chains.* Cambridge university press, 1998, no. 2.

[4] R. Bellman, "Dynamic programming," *science*, vol. 153, no. 3731, pp. 34–37, 1966.

[5] D. T. Hoang, N. V. Huynh, D. N. Nguyen, E. Hossain, and D. Niyato, *Markov Decision Process and Reinforcement Learning*, 2023, pp. 25–36.

[6] T. G. Dietterich, "Hierarchical reinforcement learning with the maxq value function decomposition," *Journal of artificial intelligence research*, vol. 13, pp. 227–303, 2000.

[7] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," *Advances in neural information processing systems*, vol. 12, 1999.

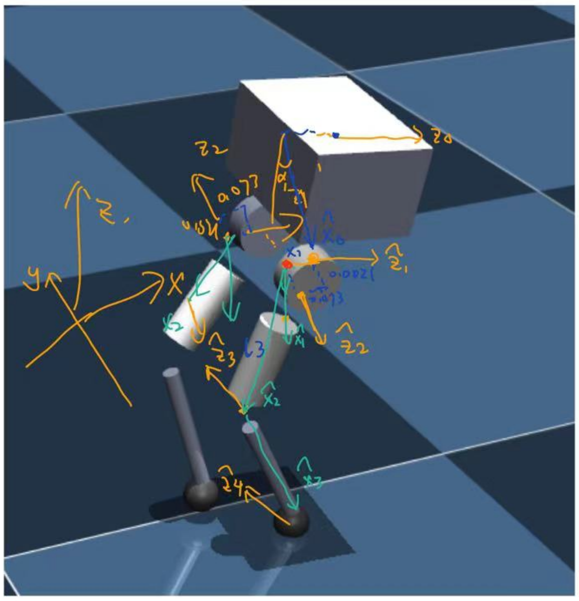
[8] M. Hesse, J. Timmermann, E. Hüllermeier, and A. Trächtler, "A reinforcement learning strategy for the swing-up of the double pendulum on a cart," *Procedia Manufacturing*, vol. 24, pp. 15–20, 2018.

Appendix

A Project File Structure

We provide a `readme.md` in the folder where we described our models and presentation videos. Besides, you can also access our model on Optima_Estimation.

B Derivation of DH Table



	a_{i-1}	α_{i-1}	d_i	θ_i
1	$-\sqrt{0.034^2+0.07976^2}$	0	-0.073	$\theta_1 + \alpha_1$
2	0	90°	0	θ_2
3	$\sqrt{0.15^2+0.25981^2}$	180°	0	θ_3
4	$\sqrt{0.15^2+0.25566^2}$	0	0	0

Figure 11: Derive the DH Table Manually (right leg)

The Left DH Table is:

```
args = (
    np.arctan2(0.0095, 0.7976),
    np.arctan2(0.15, 0.25981),
    np.arctan2(0.15, 0.25981) + np.arctan2(0.15, 0.25566)
)
DH_TABLE_L = [
    (0.0, 0.0, 0.0, args[0]),
    (np.sqrt(0.25981**2+0.15**2), 0.0, 0.34, -args[0] + theta1),
    (0.0, np.pi/2, 0.0, theta2 + args[1]),
    (np.sqrt(0.15**2+0.25981**2), np.pi, 0.0, theta3 + args[2]),
    (np.sqrt(0.15**2+0.25566**2), 0.0, 0.0, 0.0)
]
```

The Right DH Table is:

```
args = (
    np.arctan2(0.0095, 0.7976),
    np.arctan2(0.15, 0.25981),
    np.arctan2(0.15, 0.25981) + np.arctan2(0.15, 0.25566)
)
DH_TABLE_R = [
    (0.0, 0.0, 0.0, -args[0]),
    (np.sqrt(0.25981**2+0.15**2), 0.0, 0.34, args[0] + theta4),
    (0.0, np.pi/2, 0.0, theta5 - args[1]),
    (np.sqrt(0.15**2+0.25981**2), np.pi, 0.0, theta6 - args[2]),
    (np.sqrt(0.15**2+0.25566**2), 0.0, 0.0, 0.0)
]
```

C Rewards and Restrictions

Here we provide our reward functions for the training.

Table 2: Rewards and restrictions.

Task	Formula	Notes
Balance Inverted Pendulum	$1 - 0.6x^2 - 0.01\theta^2 - 0.01\dot{x}^2 - 0.01\dot{\theta}^2$	Stop if $\lvert\theta\rvert < 0.15$
Swing-up Inverted Pendulum (1)	$2e^{(\cos(\theta))}$ - θ^2	/
Swing-up Inverted Pendulum (2)	$-0.2x^2 - 1.8\theta^2 - 0.002\dot{\theta}^2$	if $\lvert x\rvert > 0.95$, $r = r - 20$, if $\lvert\theta\rvert < 0.2$, $r = r + 3.5$
Balance Inverted Double Pendulum	$10 - 0.08x^2 - 10(y-2)^2 - 0.008\dot{\theta}_1^2 - 0.04\dot{\theta}_2^2$	Stop if $y < 1.02$
Swing-up Inverted Double Pendulum (1)	-1	Stop if $y > 1.1$
Swing-up Inverted Double Pendulum (2)	$-\theta_1^2 - (\theta_1 + \theta_2)^2$	/

y: The vertical displacement of the top of the second-order pendulum relative to the pendulum rod.

D Initial States for Training

Here we provide some of the initial states our training process.

Table 3: Initial states for our training process.

Task	x	θ/θ_1	θ_2	$\dot{x}$	$\dot{\theta}/\dot{\theta}_1$	$\dot{\theta}_2$
Balance Inverted Pendulum (1)	U(-0.1,0.1)	U(-0.1,0.1)	/	U(-0.1,0.1)	U(-0.1,0.1)	/
Balance Inverted Pendulum (2)	U(-0.4,0.4)	U(-0.2,0.2)	/	U(-0.8,0.8)	U(-0.6,0.6)	/
Balance Inverted Pendulum (3)	U(-0.6,0.6)	U(-0.3,0.3)	/	U(-1.8,1.8)	U(-1.8,1.8)	/
Swing-up Inverted Pendulum	0	$-\pi$		0	0	
Balance Inverted Double Pendulum	U(-0.1,0.1)	U(-0.1,0.1)	U(-0.1,0.1)	U(-0.1,0.1)	U(-0.1,0.1)	U(-0.1,0.1)
Swing-up Inverted Double Pendulum	0	$-\pi$	0	0	0	0

`U(a, b)`: Uniform distribution from a to b.

E Simulation Environment

We provide our environment in the following table:

Table 4: Simulation environments.

ID	CPU	GPU	OS
1	Intel(R) Core(TM) i9-12900H	/	Windows 11 Home
2	AMD RYZEN 6800H	NVIDIA GEFORCE RTX 3050 ti	Windows 11 Home
3			Ubuntu 22.04