ME5418 - Machine Learning in Robotics
Group 16

Ji-Dog Quadrupedal Robots Locomotion via Reinforcement Learning
Gym Report

Group members:

Bai Yue (ID: A0304233U)

Ji Shuchen (ID: A0305152U)

Zhang Binlin (ID: A0304090U)

October 18, 2024

1 Project Introduction

In this project, we aim to train a quadruped robot to perform locomotion using reinforcement learning. Our first step involves setting up an OpenAI Gym environment. For this, we developed a quadruped robot locomotion simulation in Isaac Sim. Isaac Sim provides a highly accurate physics simulation platform, well-suited for modeling complex robot dynamics and ideal for training quadruped locomotion. The environment we built is single-agent, static, deterministic, and partially observable, with a continuous state space and a continuous action space.

In the *ji_dog_env_create.py* file, we define the *Ji_Dog_Env* class, which inherits from the Env class in OpenAI Gym. *Ji_Dog_Env* establishes a connection with the Isaac Sim simulation engine and retrieves the agent's state information through this environment.

2 Code Explanation

Using the interfaces provided by Isaac Sim, the *Ji_Dog_Env* class can obtain the physical state of the quadruped robot, including:

- *self.robot.get_world_pose()*: The world pose position and orientation (quaternion form) of the robot.
- *self.robot.get_linear_velocity(), dog.get_angular_velocity()*: The linear velocity and angular velocity of the robot.
- *dog.get_mass()*: The mass of the robot.
- *self.joint_view.get_joint_positions())*: The position of the robot's legs. And check whether the legs are in contact with the ground using the condition if $|position\_leg*| <= 0.02$.
- *self.joint_view.get_joint_velocities()*: The velocities of the robot's legs.

2.1 setup_scene():

This function is used to set up the physical simulation environment. It adds the ground, loads the robot model, adds articulations, gets the articulations' controller, gets the physical interface, and initializes contact points.

- *scene.add_default_ground_plane()*: Create a ground plane.
- *add_reference_to_stage(usd_path, prim_path)*: Load the Ji-dog robot from USD file.
- *scene.get_object('my_robot')*: Get the loaded robot.
- *ArticulationView(prim_paths_expr, name="joint_view")*: Wrap all articulations.
- *scene.add(self.joint_view)*: Add the articulations into the world scene.
- *get_articulation_controller()*: Get the articulation controller.
- *get_physx_interface()*: Get physical interface.
- *subscribe_physics_on_step_events(self.on_step, pre_step, order)*: Subscribe physical interface.
- *self._world.scene.add(self.robot)*: Add the robot into the world scene.
- *slope_flag*: Slope_flag is used to decide whether there is a slope on the ground. When slope_flag is true, this function will be activated to establish a slope whose gradient and scale can be adjusted.

2.2 get_observation():

This function is responsible for collecting the agent's current state information during the simulation. It retrieves key data such as the agent's centroid position, orientation (in Euler angles), linear velocity, angular velocity, joint positions, joint velocities, and the contact state of each leg with the ground.

- $get\_world\_pose()$: The centroid position and orientation of the agent in the environment. The position provides the coordinates of the agent's center of mass while the orientation is represented as a quaternion.
- $get\_linear\_velocity()$: Translational velocities of the center of mass along the x, y, and z axes.
- $calculate\_rpy(robot\_orientation0)$: It converts a quaternion into Euler angles, specifically roll (r), pitch (p), and yaw (y).
- $get\_angular\_velocity()$: Angular velocities of the center of mass around the x, y, and z axes.
- $get\_joint\_states()$: It gets both the angles and the velocities of all 4 joints.
- $get\_contact\_state()$: The state of each foot's contact with the ground (Boolean values) where $c_i = 1$ indicates that the i-th foot is in contact with the ground and $c_i = 0$ indicates no contact.

2.3 step():

This function initializes or resets the agent and then moves it forward by a step. It retrieves sensor data, calculates rewards, and checks if the step or episode is completed.

We added a control flag, *gait_flag*, to switch between two modes of controlling the agent's gait:

- If *gait_flag* is True: The agent follows a predefined cyclic alternating gait, where each pair of legs (left-front and right-hind, or right-front and left-hind) moves alternately.
- If *gait_flag* is False: The agent's actions are randomly sampled from the defined action space.

2.4 calculate_reward():

This reward function gets the agent's state information from API including the position and orientation of the mass centre and contact points of four legs. And it will calculate the corresponding reward accordingly.

Our reward function can be divided into sparse and dense parts. The basic sparse rewards/penalties include goal reward, fall penalty and contact point penalty. The basic dense rewards/penalties include progress reward, mass centre reward and stability penalty. These rewards/penalties are given by comparing state values with the expected states or set thresholds. The final reward is the weighted sum of the mentioned rewards and penalties.

- Goal reward: A reward of 100 is given when the agent successfully reaches the goal.
- Fall penalty: A penalty of -10 is applied if the agent falls.
- Contact penalty: A penalty of -5 is applied each time the agent takes unsuitable gaits.
- Progress reward: Progress reward is always present which is inversely proportional to the agent's distance from the goal.
- Mass reward: Mass reward is always present which is inversely proportional to the difference between the centroid height of the agent and the expected centroid height.
- Stability reward: When the roll or pitch angle exceeds the 10-degree angle threshold, the agent will receive a stability penalty proportional to the extra angle.

3 Existing Codes/Libraries

3.1 Custom Quadruped Robot Model

We created a four-legged robot model with four joints, specifically designed for our simulation environment. The entire modeling process is documented in the `Ji_dog_model_create.py` file, where we defined the agent's structure, including the joints and limbs. After completing the design, the model was saved in the `ji_dog.usd` file, which is used within the simulation to represent the agent's physical characteristics and articulations.

3.2 Isaac Sim Simulation Environment

We chose Isaac Sim due to its ability to simulate complex robot dynamics and physics with high fidelity. We used USD files to import the robot model and configured the environment using Isaac Sim's core libraries. Through the World class, we define a physics scene, add objects such as ground planes, and manage the simulation loop.

3.3 Utilized Libraries

We utilized libraries from *omni.isaac.core*, such as *World, DynamicCuboid, Robot, ArticulationAction*, and *ArticulationView*. They facilitate the definition of the simulation environment, the creation of dynamic objects, the control over robot articulations, and the real-time retrieval of joint states.

- *World* is used to define the simulation environment, which includes elements such as ground planes and other physical objects.
- *DynamicCuboid* is utilized to create dynamic objects within the scene.
- *ArticulationAction* is used to define and control the actions performed by the robot's joints.
- *ArticulationView* provides a way to interact with the robot's articulations and gets joint positions and velocities.

4 Reflections/Lessons Learned

So far, we have utilized the four-joint quadrupedal robot designed by ourselves, and we plan to train the basic four-joint model first, and then if possible we will complete the training of an eight-joint robot.

When designing the model, we noticed that objects were bouncing off, and we deeply considered the impact of material properties on the stability of the simulation environment. By adjusting the friction and restitution coefficients, we resolved this issue.

While setting up the joints, we encountered difficulties with the physical engine interfaces, particularly in controlling the robot's gait. We found that randomly applying joint velocities to the model could not effectively drive the robot, which not only made it difficult for the robot to move but also added complexity to the following reinforcement learning training. To address this, we adopted a periodic gait control method.