The Penguin Project

File legend

- full_robot.xml - position-controlled robot model with no hip actuation.
- torque_robot.xml - torque-controlled robot with no hip actuation.
- test_robot.xml - torque-controlled robot with hip actuation (actual robot)
- Meshes - all cad files
- gym_PPO.py - PPO implementation from GitHub repo(s) for gym environments
- PPO_chkpts - folder to save models for actor and critic networks
- video_progress - folder to save videos of an agent in action during testing
- saved_models - folder to save checkpoints while training
- sac - custom implementation of SAC from GitHub repos for penguin env
- mcts - custom implementation of Monte Carlo Tree search for penguin env
- gym_mujoco.py - gym environment for penguin's mujoco model with stable baseline3's SAC agent.
- reward_func_trials.py - types of reward functions I've tried so far.

Files for training the RL model

gym_mujoco.py

This script has code for creating a Gym environment using the XML file of the penguin robot. The action space is a Box of size 2 with a minimum value of -0.078 and a maximum of 0.078. The observation space I've used is a box of size 16 [COM position; Euler angles(XYZ) of COM; COM velocity; COM angular velocity, right hip and right motor angular velocities, left hip and left motor angular velocities]. I've explained the reward function separately under the 'reward_func_trails.py'.

The test_model() function runs the trained model for 1000 time steps (3 seconds), generates a video and saves it in the video_progress folder. It records the COM trajectory(x,y,z) and the actions history and displays a graph plotting actions vs time.

The next part of the code deals with using Stable Baseline3's algorithms. It involves, importing the necessary libraries, vectorising the environment and running the training loop. The argument 'n_envs' is used to specify the number of environments parallel training.

Heads-up: Sometimes, when running the whole code, the test_model() function saves a video with nothing but noise. If it happens, comment out the training loop, load the trained model and run the test function separately.

reward_func_trials.py

The net reward is a sum of rewards that incentivizes the desired behaviour and penalizes undesirable actions. I have used 3 major types of reward functions:

1) Discontinuous rewards - A constant healthy reward was staying alive (i.e. if the COM z-position of the robot is greater than 0.6) and a negative reward for falling down. A forward reward that includes forward position (COM y position) and the sum of COM velocities along the x and y directions, multiplied by a scaling factor. Finally, a small control loss penalizes huge control values and hip motor accelerations. Ideally, for algorithms like a soft-actor critic, it is preferable to use a continuous reward.
2) Continuous rewards- Here the healthy reward is a function of the COM z position. So, if the robot stays at the same height it'll receive a constant reward and will the rewards will drop as the robot falls down.
3) Time-varying cost- This function includes a time-varying penalty for not walking as time progresses. The cost is calculated by dividing the total time by the total COM distance travelled along the y-axis. The idea is to have a time-varying reward function that incorporates the significance of time.
4) Reward function with sigmoid scaling- The idea is to scale each of the reward terms between 0 and 1 before calculating the net reward. This helps to prioritize one reward over another and speeds up the training.

Note on observation space: Adding acceleration to the observation space causes the agent to take only the highest and least actions (-0.078 and 0.078).

Note on reward function: If I prioritize the forward reward, then the robot ignores the healthy reward and covers the distance at the cost of falling down and using high control

values. Also, sometimes the robot stands still and moves front and back like a rocking duck to collect rewards without taking a step forward.

To replicate the results, just run the ‘gym_mujoco.py’ with all the cad files in the same folder. The best result (video and model) I’ve got so far is saved with the suffix ‘_gg’.

sac

This folder has files to run the custom implementation of SAC algorithm. Run the ‘main.py’ file to start the training.

mcts

This folder has files to run the Monte-Carlo Tree Search algorithm on the penguin environment. The goal is to do a tree search on the action space to find the optimal control. The tree search process involves copying the environment along with the states at each stage to perform rollouts. Ideally, deepcopy is used to make a copy of the gym environments with the state information but, deepcopy doesn’t work with custom created environments because of some missing private functions that can pickle the state information. So, I created a custom function that creates a new environment and loads the states.

Also, tree search algo needs a discrete action space. So, I created a dictionary that stores combinations of action pairs. The ‘utils.py’ file stores the functions for discretizing the action space. Run the ‘main_mcts.py’ script to start the tree search.

Issue: For each step, the search algorithm creates a copy of the whole environment and hence runs out of space very quickly. Should try saving the environment states locally or run the code outside WSL.