CS 348E Project 5: Sink or Swim

 Due Date: June 1 2022, 11:59 PM

1 Preview

In this project, your job is to "teach" the humanoid character to "swim" and keep its head above water in the swimming pool we built earlier in this course. The simulated character will learn the motor skill through trial-and-error with the state-of-the-art deep reinforcement learning (DRL) algorithm. In each trial, the environment will first be reset such that the character starts at a same pose in the pool. Then the character will perform a series of actions and get rewards/penalties based on how good the motion resulted from its actions is for it to not sinking. The character will improve its policy for selecting actions in later trials based on the reward it has got. Hopefully it will get better at this motor skill after many iterations!

1.1 Requirements

1. Understand how DRL problems are formulated using the framework provided by OpenAI Gym: (https://stackoverflow.com/questions/45068568/how-to-create-a-new-gym-environment-in-openai). Don't worry – we provide much help in the starter code.

2. Understand how the character is defined, and design the observation (state) space and the action space for this character.

3. Translate the task into a well-shaped reward function (Try several candidates!)

4. Set-up and run the DRL toolchain for training and testing. Training may take over 10 hours depend on how powerful your CPU is. Plan ahead! If you think you need computing resources to complete this project, please contact the CA. In general, if your computer is running at a FPS less than 100, that is probably too slow (see Section 7).

1.2 Overview of files to change

There are many files in the starter code of this project. To finish the requirements you will only need to change the "TODO" sections in "my_pybullet_envs/humanoid_swimmer.py" and "my_pybullet_envs/humanoid_swimmer_env.py". That said, since this project is semi-open, you are free to modify the starter code, as long as you clarify what your changes are and why you change them in a short write-up.

2 DRL Toolchain Set-up

Refer to README.txt after you downloaded the starter code. The starter code is adapted from `https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail` and `https://github.com/openai/baselines`. We will be using the DRL algorithm Proximal Policy Optimization (PPO) for this assignment (`https://arxiv.org/abs/1707.06347`).

3 Humanoid Swimmer in pyBullet

We use a humanoid model very similar to the one in Project 3. In "my_pybullet_envs /assets/humanoid_box.urdf", you can find the exact definition of the humanoid (Figure 1). It has 23 Degrees of Freedom (DoFs) from joint #0 to #22. A final fixed joint #23 is added between torso and head, meaning the character does not move its neck.

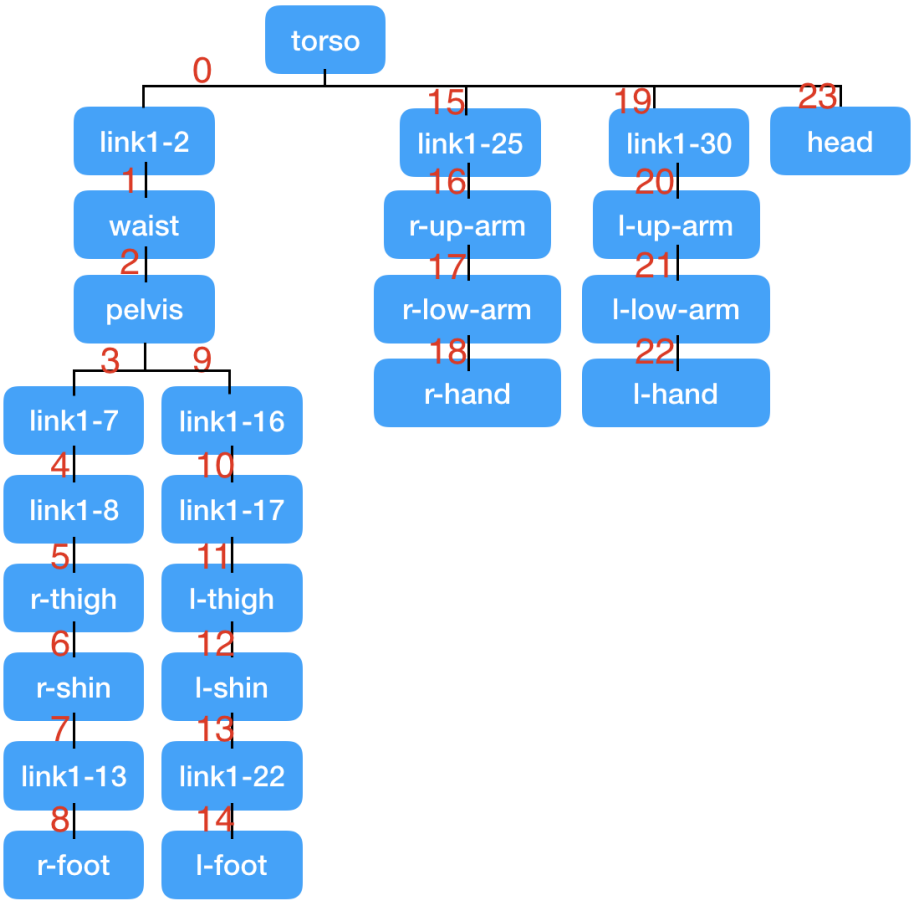

Figure 1: The URDF humanoid model.

We will control all 23 Degrees of Freedom (DoFs) as DRL can handle high-dimensional control space relatively easily. "my_pybullet_envs/humanoid_swimmer.py" contains (incomplete) utility functions to reset, control and get current state of the character. Check the top of this file for a comment section about additional details of humanoid model.

4 Observation Space Design

In "my_pybullet_envs/humanoid_swimmer_env.py", you should design your observation vector in the `get_extended_observation` function. The vector returned by this function will be the input to the policy neural network, thus it should include all the features (information) needed for the policy to select the next action.

For example, the observation should include the current state of the character such as joint angles $\boldsymbol{q}$ and velocities $\dot{\boldsymbol{q}}$, as well as current position, orientation, and velocities of the base link. You could also consider doing some feature engineering to facilitate neural network training. For example, practitioners sometimes also append the world positions of the two feet to the observation vector. This information is redundant to what we already have (why? Hint: forward kinematics), but in practice feature engineering like this could accelerate training.

The observation can also include interaction information between the character and the environment, such as whether certain link is currently in contact with the pool bottom, or the fluid force currently applied at certain link.

One final reminder is to keep all dimensions of the observation roughly normalized, or at least with the same order of magnitude. The reason is a standard machine learning one – we do not want to have one feature that is for example 100x larger than the other one (which easily gets ignored by the neural network).

To make our code a bit cleaner, you could pack the first part of observation that is only related to the character in the `get_robot_observation` function in "humanoid_swimmer.py", and call this function in `get_extended_observation` with the second part of observation added there. But again, it is up to you to re-structure the starter code.

5 Action Space Design

The action vector, the output of the policy neural network, usually has a small magnitude per element (around 1) due to weight initialization. We need to transform / scale the vector so that they represent physically plausible control signals to the humanoid. This will be done in the `apply_action` function in "humanoid_swimmer.py"

We suggest two approaches here, but feel free to invent your own! The first one is to apply target position control as we learned in Project 3. `HumanoidSwimmer.tar_q` stores the current target joint angles in the starter code. We obtain the next target angles by `tar_q += a * delta_tar_q_scale`, where `a` is the action vector and `delta_tar_q_scale` is a small constant (0.03 in starter code) so that `tar_q` do not change too much in one single control step. Afterwards, use `setJointMotorControlArray` with `POSITION_CONTROL` mode. Remember to constrain the maximum motor forces to `HumanoidSwimmer.max_forces` to make simulation more stable (see PyBullet Start Guide). If you use this definition of action, you should append `tar_q` to the observation vector to make the problem Markovian (Why?).

The second approach would be direct torque control. Similar to the PD control used in Project 3, we would calculate and set the motor forces by ourselves rather than let PyBullet calculate them for us from target joint angles. In this case we can simply scale

`a` up with `HumanoidSwimmer.max_forces` and feed it into `setJointMotorControlArray` with `TORQUE_CONTROL` mode. Remember to first cap `a` to the range of $[-1, 1]$ so that the scaled motor forces do not exceed the maximum force.

Using torque control may sometimes create unstable motions with very large velocities (i.e. simulation almost explodes). You could decrease `HumanoidSwimmer.max_forces`, or make `max_forces` a variable vector depending on the current magnitude of the joint velocities – for example, when $\dot{\boldsymbol{q}}$ becomes large, `max_forces` should be smaller to slow down further increase of $\dot{\boldsymbol{q}}$.

6 Reward Function Design

Providing reward to evaluate the policy's performance after each control step is arguably the most crucial component of setting up successful DRL training. The `step` function in "humanoid_swimmer_env.py" advances one control step, which consists of `control_skip` number of Bullet simulation steps. Here, you will need to first roll the simulation forward `control_skip` steps with the same `a` (action) applied at each simulation step.

Applying Fluid Forces: Naively, you should also apply the fluid forces (defined in rigidBodySento.py) to the humanoid at every simulation step. But this will make the training slow. Since fluid forces tend to not change much in short periods of time, we can make a simplification here by only updating fluid forces every 5 simulation steps (`control_skip / 5` times per control step). In other words, we apply the same amount of force for 5 simulation steps and then recompute. The API of `apply_fluid_force` should give you a hint on this.

In the `step` function, you also need to provide a reward value. A large reward value means that the state after this control step is more preferable. You might want to give a high reward if the head is currently above the water surface; you might also want to provide some "reward shaping" to make reward less sparse: even if the head is not above the water, it should still get some reward based on how "close" it is to good states, instead of getting no reward at all. Finally, you might also want to provide some "regularization": you might want to penalize (i.e. give low reward) when the character's joint velocities are too high, or its joint angles are too close to the joint limits, or it is using too much torque (motor force). This could prevent the policy from exploring unfavorable or unnatural-looking regions of the state space – after all, we are searching for a good policy in a pretty high-dimensional state-action space.

You will need to tune the weights of each reward term so that no single term dominates the reward value. One additional tip is that you would want each term to be bounded above and below - if certain term is unbounded and can sometime be 1 sometimes be 1000, it would then be really difficult to assign a good weight to it.

Again, do not expect your first reward function to work perfectly, and feel free to add more terms which you think will lead the character to success (what kind of behavior is promising even if it is not success yet?). Be careful and creative!

7 Training

Now we can start the training! The following command uses the default parameters of PPO to train the policy (use "`--using-torque-ctrl 0`" if you are implementing position control):

```
  python -m a2c_ppo_acktr.train_policy --env-name "HumanoidSwimmerEnv-v1"
--num-steps N/P --num-processes P
--lr 3e-4 --entropy-coef 0 --ppo-epoch 10
--num-mini-batch 32 --num-env-steps 5000000
--use-linear-lr-decay --clip-param 0.2
--save-dir trained-models-swim-1st-trial
--seed 12345 --using-torque-ctrl 1
```

N here is the total number of control steps we roll out and collect as experience before we update our policy neural network once. For example, we can set $N = 19200$. Since each trajectory is 240 control steps long (check `my_pybullet_envs/__init__.py`), this would mean that we collect 80 trajectories before each policy update. These N steps will be simulated in parallel with P threads. It is not a bad idea to set P to 1x $\sim$ 2x the number of cores/hyper-threads you have on your machine.

`num-env-steps` is the total number of control steps we roll out for the entire training. When you start running this command above, `FPS` will be printed to your console. For example, if your `FPS` is 100, then training would take 5000000/100/3600 = 13.9 hours on your machine. **Contact us if `FPS` on your machine is less than** 100**.**

Feel free to increase `num-env-steps` if your machine can afford. Though this assignment will not be graded by how good your character's motion looks, training for longer with more collected data will in general improve task success and quality of motion.

You can look up what the other parameters mean in the original PPO paper; in general tuning them is less important than the observation, action, and reward designs mentioned above. The meanings of logging parameters are mentioned in "a2c_ppo_acktr/argumemnts.py".

You can visualize the trained policy using the following command
(use "`--using-torque-ctrl 0`" if you are not using Bullet torque control mode):

```
  python -m a2c_ppo_acktr.test_policy
--env-name "HumanoidSwimmerEnv-v1"
--load-dir trained-models-swimmer-1st-trial/ppo
--render 1 --using-torque-ctrl 1 --non-det 0 --num-trajs 10
```

7.1 Debugging tips

At the very beginning of the training, the actions outputted by the policy are essentially random. Visualizing a random policy can help you understand if there are issues with your action space design. For example, if the simulation almost explodes, then the motor forces applied might be too large; if the character instead only does minor movements,

then the scaling factor of the action might be too small. You can visualize, for example the policy after 20th update, using:

```
  python -m a2c_ppo_acktr.test_policy
--env-name "HumanoidSwimmerEnv-v1"
--load-dir trained-models-swimmer-1st-trial/ppo
--render 1 --using-torque-ctrl 1 --non-det 1 --num-trajs 10 --iter 20
```

If your average reward value is not improving in the first 20 updates, first think if there is a bug in your `step` function, and think if some necessary information is missing from your observation. After that, think whether your reward function is too sparse. If your average reward has improved a lot over iterations, but when visualizing the policy it does not seem to improve at all, then maybe there is a bug in your reward function implementation. Finally, be patient and trust yourself.

8 Deliverables and Grading (85%)

There are several deliverables required for this project besides your code.

1. Please include a video (screen recording is good enough) of the motion of your final policy. An example video is included in the project folder, but you are not required to create a motion like that!

2. Please include the final policy model file (.pt), and the console_output.log file stored in your `--save-dir` path. (You could delete the intermediate policy files.)

3. Please provide a short write-up, explaining your final action space, observation space, reward function, and why you decide on them. Please also include what you tried but seemed worse than your final choice in the write-up, if any. If you modify the starter code, briefly discuss your changes as well. Finally, include the commands you use to train and visualize your policy.

60% will be allocated to whether you implemented the action, observation, and reward function correctly. The remaining 25% will be allocated to your write-up and graded based on your effort, understanding and creativity on the problem.

9 Tasks (15%+2% extra credit)

1. Start the humanoid from close to the surface of pool. Instead of keeping the head above the water, can you train the agent to move forward in the pool? You may pick an arbitrary direction as "forward". (15%)

2. If the character gains 5% of its current weight, will your policy still work, if so, you automatically complete this challenge! If not, train another policy that works for the character with larger density (+1%).

3. (Open ended) Any idea that could make this project better and/or more interesting, either in the form of a text description or starter code change. If we like it, you will get 1% extra credit!