Build a Balancing Bot with OpenAI Gym, Pt II: The Robot and Environment

This is part II of the tutorial series on building a Balancing Bot environment in OpenAI Gym, discussing implementation details of the `Env` class. You can reach the first part here.

The OpenAI Gym defines an *environment* specification, which is implemented in a python class called `Env`. Agents send actions to, and receive observations and rewards from this class. The class includes a number of attributes and methods that aid in bookkeeping and management. We will be subclassing the `Env` class and implementing the working logic of our environment as part our subclass methods.

Before going into implementation details, we will briefly go through a few essentials including a short discussion of the balancing task and the robot model. We will also have a look at the `Space` class, an essential component for creating observations.

The Balancing Bot

A balancing bot is a robot in an inverted pendulum configuration with two wheels on the same axis. Inverted pendulum is a naturally unstable configuration, thus the aim is to keep the body of the robot upright and optionally allow movement at a desired velocity. The robot controller accepts orientation and angular velocity information from an IMU and is also aware of the angular velocity of the wheels (either indirectly or using an encoder). The controller's output consists of commands to either increase or decrease the wheel angular velocity in either direction, so as to maintain balance.

The Robot Model

pyBullet accepts robot models defined in the URDF format, which is an XML format used by ROS. I have prepared a basic balancing robot model that I am sharing below:

```
<?xml version="1.0"?>
<robot name="balance">

  <material name="white">
    <color rgba="1 1 1 1"/>
  </material>

  <material name="black">
    <color rgba="0.2 0.2 0.2 1"/>
  </material>
```

```
<link name="torso">
  <visual>
    <geometry>
      <box size="0.2 0.05 0.4"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0 0.0 0.3"/>
    <material name="white"/>
  </visual>
  <collision>
    <geometry>
```

The balancing bot comprises a rectangular body with two cylindrical wheels no each side. Total height is around 50cm (19.6 inch). You can paste this snippet into a file named `balancebot_simple.xml` and save it in the same folder as the `balancebot_env.py` file.

The Space Class

The Space class provides a standardized way of defining action and observation spaces. There are many subclasses of Space included in the Gym, but in this tutorial we will deal with just two: `space.Box` and `space.Discrete`. The first is a generic class for n-dimensional continuous domains. Think of it as an n-dimensional numpy array. The `Discrete` space represents a finite set of values. In fact, the `Discrete` space itself does not hold any values, rather it only represents a single positive index, corresponding to the selected value.

In this tutorial, we will be describing the observation space using a 3-dimensional `Box` space corresponding to three continuous values: robot inclination (pitch), angular velocity and wheel angular velocity. The action space will be a `Discrete` space of nine values, each corresponding to progressively greater (negative and positive) changes in commanded wheel angular speed.

We'll discuss usage of the Space class later on when implementing the reward, observation and reset routines.

Attributes of the Env class

Environments in OpenAI Gym are subclasses of the `gym.Env` Python class. According to the `Env` class documentation, there are three attributes that need to be set in every `Env` subclass:

`action_space`: The `Space` object corresponding to valid actions.

`observation_space`: The `Space` object corresponding to valid observations.

`reward_range`: A tuple corresponding to the min and max possible rewards.

We will be setting these during the initialization of our `Env` subclass. Our implementation of the `__init__` method will be as follows:

```
def __init__(self):
    self._observation = []
```

```
    self.action_space = spaces.Discrete(9)
    # pitch, gyro, commanded speed
    self.observation_space = spaces.Box(np.array([-math.pi, -math.pi, -5
                                    np.array([math.pi, math.pi, 5])
    self.physicsClient = p.connect(p.GUI)
    p.setAdditionalSearchPath(pybullet_data.getDataPath())  # used by l
    self._seed()
```

Methods of the Env class

Taking a look at the Env class code hints that there are five methods that we need to override, whose signatures are as folows:

```
def _step(self, action)
def _reset(self)
def _render(self, mode='human', close=False)
def _seed(self, seed=None)
```

The `_step` method

The `_step` method is where the magic happens. `_step` is called once for each time step of the simulation, accepts an action from the agent, and has to return a tuple with the following four items:

- An observation
- A reward
- A boolean flag indicating whether an episode has ended
- An info dictionary.

Of those, only the first three are essential, the last one is optional and a blank dict may be returned in place. The observation is a numpy array object that needs to conform to the dimensions specified when assigning the `observation_space` attribute during instance initialization. So for instance if your observation space is a box as follows:

```
self.observation_space = spaces.Box(np.array([-1, -1, -1]),
                                    np.array([1, 1, 1]))
```

then your returned observation should be a numpy array containing three values, one for each dimension:

```
observation_array = [, , ]
```

The reward is a scalar value that represents the reward an agent gets for performing this particular action at the current state of the environment.

The third option is a boolean flag that indicates the end of an episode. Returning `True` here will trigger a call to `_reset`, which depending on the implementation, will have the environment reset itself.

The last option is really up to you. I haven't yet tried it, but returning a blank dictionary is ok.

Here is a made up example of what a typical `_step` implementation return statement would look like:

```
return np.array([0.5, -1.0, 0.22]), 0.6, False, {}
```

Obviously instead of fixed values your observations and rewards should be computed from the current state of the environment and the action being passed as an argument to `_step`.

Our implementation of the _step method is as follows:

```
def _step(self, action):
    self._assign_throttle(action)
    p.stepSimulation()
    self._observation = self._compute_observation()
    reward = self._compute_reward()
    done = self._compute_done()

    self._envStepCounter += 1

    return np.array(self._observation), reward, done, {}
```

The `_step` method itself has a few high-level calls to other methods that perform specific actions, and which we'll be getting into right away.

The `_assign_throttle` method is as follows:

```
def _assign_throttle(self, action):
    dv = 0.1
    deltav = [-10.*dv,-5.*dv, -2.*dv, -0.1*dv, , 0.1*dv, 2.*dv,5.*dv, 10
    vt = self.vt + deltav
    self.vt = vt
    p.setJointMotorControl2(bodyUniqueId=self.botId,
                            jointIndex=,
                            controlMode=p.VELOCITY_CONTROL,
                            targetVelocity=vt)
    p.setJointMotorControl2(bodyUniqueId=self.botId,
                            jointIndex=1,
                            controlMode=p.VELOCITY_CONTROL,
                            targetVelocity=-vt)
```

In this method, the following happen: First, the `deltav`, an indicator of the decided wheel speed change, is determined by selecting the element at the index specified by the action. The array contents are arbitrary and could be different. Then, `self.vt` is adjusted according to `deltav`. Finally a pyBullet method, `setJointMotorControl2()`, is called that updates the angular velocity of the robot wheels with the new value.

Read also: Wanhao i3 Mini: Essential Mods & Printing More Materials

The next call in the _step method is to `stepSimulation()`, which advances the simulation by one step. Next, three methods are called in sequence that compute and return the observation, reward and done flag, respectively. The content of each one follows:

```
def _compute_observation(self):
    cubePos, cubeOrn = p.getBasePositionAndOrientation(self.botId)
    cubeEuler = p.getEulerFromQuaternion(cubeOrn)
    linear, angular = p.getBaseVelocity(self.botId)
    return [cubeEuler[],angular[],self.vt]
```

In _compute_observation, pyBullet specific methods `getBasePositionAndOrientation`, `getEulerFromQuaternion` and `getBaseVelocity` are used to get the robot pitch and angular velocity.

```
def _compute_reward(self):
    _, cubeOrn = p.getBasePositionAndOrientation(self.botId)
    cubeEuler = p.getEulerFromQuaternion(cubeOrn)
    # could also be pi/2 - abs(cubeEuler[0])
    return (1 - abs(cubeEuler[])) * 0.1 -  abs(self.vt - self.vd) * 0.01
```

`_compute_reward` uses the same methods `getBasePositionAndOrientation`, `getEulerFromQuaternion` in order to determine the pitch. It then returns a sum of the pitch value (0 being upright) and the absolute difference between the current and desired wheel speeds. (`self.vt - self.vd`). `self.vd` is initially set to zero, and at this point it is not modifiable.

```
def _compute_done(self):
    cubePos, _ = p.getBasePositionAndOrientation(self.botId)
    return cubePos[2] < 0.15 or self._envStepCounter >= 1500
```

`_compute_done` checks two things: The first is whether the center of mass of the robot is below 15cm (5.9 inch). If so, it will mean the robot has fallen over, and so the episode has ended. In addition, it measures the environment step counter so as to limit the duration of each episode.

The final steps in the `_step` method implementation is to increase the environment step counter by one, and return the observation, reward and done flag.

The `_reset` method

The reset method is responsible both for resetting as well as initializing the environment, as it is also called once after the class is initialized. Our implementation of `_reset` is as follows:

```
def _reset(self):
    self.vt =
    self.vd =
    self._envStepCounter =

    p.resetSimulation()
    p.setGravity(,,-10) # m/s^2
    p.setTimeStep(0.01) # sec

    planeId = p.loadURDF("plane.urdf")
    cubeStartPos = [,,0.001]
    cubeStartOrientation = p.getQuaternionFromEuler([,,])

    path = os.path.abspath(os.path.dirname(__file__))
    self.botId = p.loadURDF(os.path.join(path, "balancebot_simple.xml"),
                           cubeStartPos,
                           cubeStartOrientation)

    self._observation = self._compute_observation()
    return np.array(self._observation)
```

The `_render` method

The render method is where visualization-related actions take place. In our case the pyBullet environment takes care of all visualization, so there isn't anything to do in this method. We will leave it blank.

The `_seed` method

Finally, the seed method sets the seed for this environment's random number generator:

```
def _seed(self, seed=None):
    self.np_random, seed = seeding.np_random(seed)
    return [seed]
```

Putting it all together

The complete `balancebot_env.py` file contents are as shown below:

```
import os
import math
import numpy as np

import gym
from gym import spaces
from gym.utils import seeding

import pybullet as p
import pybullet_data

class BalancebotEnv(gym.Env):
    metadata = {
        'render.modes': ['human', 'rgb_array'],
        'video.frames_per_second' : 50
    }

    def __init__(self):
        self._observation = []
        self.action_space = spaces.Discrete(9)
        self.observation_space = spaces.Box(np.array([-math.pi, -math.pi
                                            np.array([math.pi, math.pi,
```

You can paste the snippet above into your empty `balancebot_env.py`, and you'll be good to go.

Running the balancing bot simulation

It's now time to put the script we wrote at the very beginning of this tutorial to good use. Here it is presented again for convenience:

```
import gym
from baselines import deepq
import balance_bot

def callback(lcl, glb):
    # stop training if reward exceeds 199
    is_solved = lcl['t'] > 100 and sum(lcl['episode_rewards'][-101:-1])
    return is_solved

def main():
    # create the environment
    env = gym.make("balancebot-v0") # <-- this we need to create

    # create the learning agent
    model = deepq.models.mlp([16, 16])

    # train the agent on the environment
```

```
    act = deepq.learn(
        env, q_func=model, lr=1e-3,
        max_timesteps=200000, buffer_size=50000, exploration_fraction=0
        exploration_final_eps=0.02, print_freq=10, callback=callback
    )

    # save trained model
    act.save("balance.pkl")

if __name__ == '__main__':
    main()
```

Paste the contents to a file named `balancebot_task.py` just outside your `balancebot-env` folder, and run it:

```
python balancebot_task.py
```

You should see the the pyBullet viewer popping up and the robot should already be hitting the floor by the time you read this! Something like this:

But don't worry! It will improve quickly. After around 2 minutes (depending on the speed of your machine) the bot should be able to stand upright for most of the time. You can use `Ctrl+C` in the terminal to quit the simulation, or let it complete the predefined number of cycles, after which it will save the best model and quit.

What's next?

This post just scratches the surface of what is possible with Gym, Baselines and pyBullet. Regarding the balancing bot task in particular, there are a few ideas for advancing it:

- Add sensor noise: The values that we get currently are squeaky clean, as they come from software. In real life, reading sensors would involve some level of noise, so one idea is to mix some of the noise in the values of the observations. Don't worry, this will not reduce the performance of the balancing bot, on the contrary it will make it more robust.
- Add bumps and obstacles: A flat plain is good for starters, but how will the balancing bot behave in an uneven ground filled with obstacles?
- Build more advanced robot models: Boston Dynamics' Handle is a balancing bot at heart but it is so much more agile due to all the articulations and suspension it includes! Control of a complex robot like the Handle would be an ideal scenario for a reinforcement learning algorithm!

Conclusion

In this post we completed implementation of the balancing bot task and saw in detail what each part of the `Env` subclass does, getting a working Gym+Baselines+pyBullet experiment at the end. It's good fun to watch the little bot learning to balance, but there are so much more to explore!

The code for this tutorial series is now available on Github.

Do you have any questions or comments regarding this tutorial? Ask and share in the comments below!

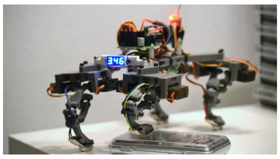
Happy hacking!

For more exciting experiments and tutorials, subscribe to our email:

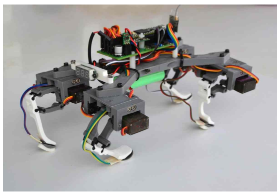
Email address

Sign me up!

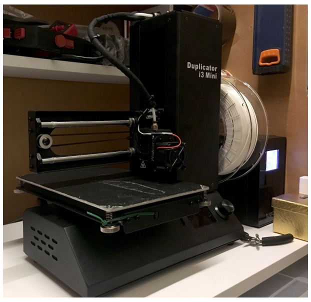
You may also like



Quadruped Robot Part II: Assembly, Electronics



Quadruped Robot Part III: Movement, Balance and Testing



Wanhao i3 Mini: Essential Mods & Printing More Materials

15 replies on " Build a Balancing Bot with OpenAI Gym, Pt II: The Robot and Environment "

Sandipan Haldar
December 19, 2017 at 5:17 am

You said to keep the balancebot_task.py file outside balancebot-env folder. But no such folder is mentioned in the directory tree in the first part of the tutorial
Also I am getting a import error :No module name 'balance-bot' while running the balancebot_task.py script

REPLY

yconst
December 20, 2017 at 12:20 am

Hi and thanks for your comment. You're right, I've amended the post with instructions on making a project folder as a first step. Regarding your error, have you run pip install -e inside your balancebot-env folder? Also, are you running all of these commands inside a Conda environment?

REPLY

Sandipan Haldar
December 25, 2017 at 11:31 pm

I ran pip install -e . inside the balance-bot folder (where setup.py is present) and it successfulled installed balance-bot. But now I am getting "ImportError:cannot import the name 'balancebotEnv' " while running balancebot_task.py. I am running all the commands inside the conda environment .

REPLY

yconst
December 27, 2017 at 3:23 pm

Only thing I can think of is your`balancebotEnv` should be with capital B, i.e.`BalancebotEnv`. I'll be publishing the code in this tutorial to Github once holidays are over and I get back to my computer 🙂

REPLY

yconst
January 12, 2018 at 6:55 pm

Hi again, the code is now available on Github:https://github.com/yconst/balance-bot

REPLY

Sandipan Haldar
January 15, 2018 at 9:32 am

Thanks its working fine. But after the bot has learnt to balance itself quite properly, its performance goes down again in the end .i.e the mean 100 episode reward increase at the beginning , reaches a max value and then decreases.
How to avoid that?

yconst
January 15, 2018 at 10:40 pm

It seems that this is caused by the buffer_size parameter being much smaller than the max_timesteps parameter. I've since found that setting them to the same value eliminates this problem.

yuta
June 6, 2018 at 9:28 am

Just to point out, a lot of '0's seem to be missing from balancebot_env.py in the code listed on this web page (for example line 44), so it would not compile if copy pasted from here, and there are differences between the code on GitHub.

REPLY

yconst
June 6, 2018 at 4:21 pm

There are indeed some differences with github code (which was updated even after this post was published).

Can you elaborate on the issue you are facing running the code? It seems line 44 of balancebot_env.py does not contain any 0s.

Thanks

REPLY

rickjhee
August 20, 2018 at 5:48 pm

Code is working great, pybullet for some reason is not opening though? I am just getting results in the command prompt window. What could be the issue?

Thanks

REPLY

yconst
August 20, 2018 at 7:55 pm

Thanks for the comment! The window is an OpenGL viewport AFAIK so if you have any special OpenGL or screen setup/settings this may be bollocksing the display. I don't have much experience on OpenGL to help you, but I've googled around and found this https://superuser.com/questions/1241137/accessing-an-opengl-gui-through-x11-forwarding?rq=1 which may be of help.

REPLY

rickjhee
August 21, 2018 at 7:22 pm

Thank you, I added the following line to the balancebot_task.py and this did the job!

import pybullet as p; p.connect(p.GUI)

REPLY

yconst
August 22, 2018 at 9:30 pm

It would also be helpful to assign the client returned by connect() to a variable like shown in the post, i.e. self.physicsClient = p.connect(p.GUI)

REPLY

reinforcementlearner
December 7, 2018 at 1:57 pm

Thank you for this great post. Just a small question, in the control part why is one joint getting vt, while the other -vt as the target velocities. I am a bit confused on the -vt part.

REPLY

yconst
December 11, 2018 at 9:11 pm

One joint is rotated 180 degrees around the z axis, so it needs the opposite throttle setting to rotate in the same direction.

REPLY

Leave a Reply

Enter your comment here...

Share this: 11

Next Post
Training Pipeline for Autonomous Driving

Previous Post
Build a Balancing Bot with OpenAI Gym, Pt I: Setting up

Subscribe to our email:

Email Address

Sign me up!