Exploring the Gym and its Features

Now that you have a working setup, we will start exploring the various features and options provided by the Gym toolkit. This will walk you through some of the commonly used environments, the tasks they solve, and what it would take for your agent to master a task.

In this , we will explore the following topics:

- Exploring the various types of Gym environment
- Understanding the structure of the reinforcement learning loop
- Understanding the different observation and action spaces

Exploring the list of environments and nomenclature

Let's start by picking an environment and understanding the Gym interface. You may already be familiar with the basic function calls to create a Gym environment from the previous chapters, where we used them to test our installations. Here, we will formally go through them.

Let's activate the `rl_gym_book` conda environment and open a Python prompt. The first step is to import the Gym Python module using the following line of code:

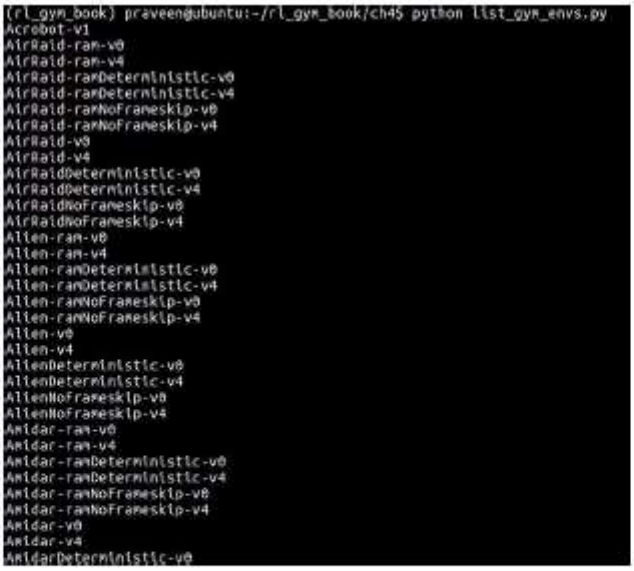
```
import gym
```

We can now use the `gym.make` method to create an environment from the available list of environments. You may be asking how to find the list of Gym environments available on your system. We will create a small utility script to generate the list of environments so that you can refer to it later when you need to. Let's create a script named `list_gym_envs.py` under the `~/rl_gym_book/ch4` directory with the following contents:

```
#!/usr/bin/env python
from gym import envs
env_names = [spec.id for spec in envs.registry.all()]
for name in sorted(env_names):
 print(name)
```

This script will print the names of all the environments available through your Gym installation, sorted alphabetically. You can run this script using the following command to see the names of the environments installed and available in your system:

```
(rl_gym_book) praveen@ubntu:~/rl_gym_book/ch4$python list_gym_envs.py
```



From the list of environment names, you may note that there are similar names, with some variations. For example, there are eight different variations for the Alien environment. Let's try to understand the nomenclature before we pick one and start using it.

Nomenclature

The presence of the word *ram* in the environment name means that the observation returned by the environment is the contents of the **Random Access Memory** (**RAM**) of the Atari console on which the game was designed to run.

The presence of the word *deterministic* in the environment names means that the actions sent to the environment by the agent are performed repeatedly for a *deterministic/fixed* duration of four frames, and then the resulting state is returned.

The presence of the word *NoFrameskip* means that the actions sent to the environment by the agent are performed once and the resulting state is returned immediately, without skipping any frames in-between.

By default, if *deterministic* and *NoFrameskip* are not included in the environment name, the action sent to the environment is repeatedly performed for a duration of *n* frames, where *n* is uniformly sampled from {2,3,4}.

The letter *v* followed by a number in the environment name represents the version of the environment. This is to make sure that any change to the environment implementation is reflected in its name so that the results obtained by an algorithm/agent in an environment are comparable to the results obtained by another algorithm/agent without any discrepancies.

Let's understand this nomenclature by looking at the Atari Alien environment. The various options available are listed with a description as follows:

Version name	**Description**
`Alien-ram-v0`	Observation is the RAM contents of the Atari machine with a total size of 128 bytes and the action sent to the environment is repeatedly performed for a duration of *n* frames, where *n* is uniformly sampled from {2,3,4}.
	Observation is the RAM contents of the Atari machine with a total size of 128 bytes and the action sent to the

`Alien-ram-v4`	environment is repeatedly performed for a duration of *n* frames, where *n* is uniformly sampled from {2,3,4}. There's some modification in the environment compared to v0.
`Alien-ramDeterministic-v0`	Observation is the RAM contents of the Atari machine with a total size of 128 bytes and the action sent to the environment is repeatedly performed for a duration of four frames.
`Alien-ramDeterministic-v4`	Observation is the RAM contents of the Atari machine with a total size of 128 bytes and the action sent to the environment is repeatedly performed for a duration of four frames. There's some modification in the environment compared to v0.
`Alien-ramNoFrameskip-v0`	Observation is the RAM contents of the Atari machine with a total size of 128 bytes and the action sent to the environment is applied, and the resulting state is returned immediately without skipping any frames.
`Alien-v0`	Observation is an RGB image of the screen represented as an array of shape (210, 160, 3) and the action sent to the environment is repeatedly performed for a duration of *n* frames, where *n* is uniformly sampled from {2,3,4}.
`Alien-v4`	Observation is an RGB image of the screen represented as an array of shape (210, 160, 3) and the action sent to the environment is repeatedly performed for a duration of *n* frames, where *n* is uniformly sampled from {2,3,4}. There's some modification in the environment compared to v0.
`AlienDeterministic-v0`	Observation is an RGB image of the screen represented as an array of shape (210, 160, 3) and the action sent to the environment is repeatedly performed for a duration of four frames.
`AlienDeterministic-v4`	Observation is an RGB image of the screen represented as an array of shape (210, 160, 3) and the action sent to the environment is repeatedly performed for a duration of

	four frames. There's some modification in the environment compared to v0.
`AlienNoFrameskip-v0`	Observation is an RGB image of the screen represented as an array of shape (210, 160, 3) and the action sent to the environment is applied, and the resulting state is returned immediately without skipping any frames.
`AlienNoFrameskip-v4`	Observation is an RGB image of the screen represented as an array of shape (210, 160, 3) and the action sent to the environment is applied, and the resulting state is returned immediately without skipping any frames. any frames. There's some modification in the environment compared to v0.

This summary should help you understand the nomenclature of the environments, and it applies to all environments in general. The RAM may be specific to the Atari environments, but you now have an idea of what to expect when you see several related environment names.

Exploring the Gym environments

To make it easy for us to visualize what an environment looks like or what its task is, we will make use of a simple script that can launch any environment and step through it with some randomly sampled actions. You can download the script from this book's code repository under `ch4` or create a file named `run_gym_env.py` under `~/rl_gym_book/ch4` with the following contents:

```
#!/usr/bin/env python

import gym
import sys

def run_gym_env(argv):
    env = gym.make(argv[1]) # Name of the environment supplied as 1st argument
    env.reset()
    for _ in range(int(argv[2])):
        env.render()
        env.step(env.action_space.sample())
    env.close()

if __name__ == "__main__":
    run_gym_env(sys.argv)
```

This script will take the name of the environment supplied as the first command-line argument and the number of steps to be run. For example, we can run the script like this:

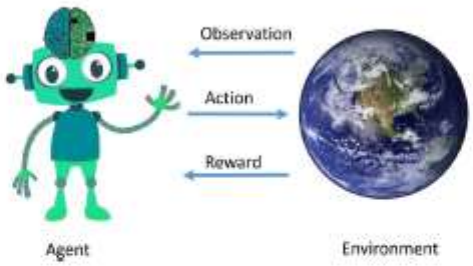
```
(rl_gym_book) praveen@ubntu:~/rl_gym_book/ch4$python run_gym_env.py Alien-ram-v0 2000
```

This command will launch the `Alien-ram-v0` environment and step through it 2,000 times using random actions sampled from the action space of the environment.

You will see a window pop up with the `Alien-ram-v0` environment, like this:

Understanding the Gym interface

Let's continue our Gym exploration by understanding the interface between the Gym environment and the agents that we will develop. To help us with that, let's have another look at the picture



Did the picture give you an idea about the interface between the agent and the environment? We will make your understanding secure by going over the description of the interface.

After we `import gym`, we `make` an environment using the following line of code:

```
env = gym.make("ENVIRONMENT_NAME")
```

Here, `ENVIRONMENT_NAME` is the name of the environment we want, chosen from the list of the environments we found installed on our system. From the previous diagram, we can see that the first arrow comes from the environment to the agent, and is named Observation. we understand the difference between partially observable environments and fully observable environments, and the difference between state and observation in each case. We get that first observation from the environment by calling `env.reset()`. Let's store the observation in a variable named `obs` using the following line of code:

```
obs = env.reset()
```

Now, the agent has received the observation (the end of the first arrow). It's time

for the agent to take an action and send the action to the environment to see what happens. In essence, this is what the algorithms we develop for the agents should figure out! We'll be developing various state-of-the-art algorithms to develop agents in the next and subsequent chapters. Let's continue our journey towards understanding the Gym interface.

Once the action to be taken is decided, we send it to the environment (second arrow in the diagram) using the `env.step()` method, which will return four values in this order: `next_state`, `reward`, `done`, and `info`:

1. The `next_state` is the resulting state of the environment after the action was taken in the previous state.

Some environments may internally run one or more steps using the same action before returning the `next_state`. We discussed deterministic and NoFrameskip types in the previous section, which are examples of such environments.

2. The `reward` (third arrow in the diagram) is returned by the environment.
3. The `done` variable is a Boolean (true or false), which gets a value of true if the episode has terminated/finished (therefore, it is time to reset the environment) and false otherwise. This will be useful for the agent to know when an episode has ended or when the environment is going to be reset to some initial state.

4. The `info` variable returned is an optional variable, which some environments may return with some additional information. Usually, this is not used by the agent to make its decision on which action to take.

Here is a consolidated summary of the four values returned by a Gym environment's `step()` method, together with their types and a concise description about them:

Returned value	**Type**	**Description**
`next_state` (or observation)	`Object`	Observation returned by the environment. The object could be the RGB pixel data from the screen/camera, RAM contents, join angles and join velocities of a robot, and so on, depending on the

		environment.
`reward`	`Float`	Reward for the previous action that was sent to the environment. The range of the `Float` value varies with each environment, but irrespective of the environment, a higher reward is always better and the goal of the agent should be to maximize the total reward.
`done`	`Boolean`	Indicates whether the environment is going to be reset in the next step. When the Boolean value is true, it most likely means that the episode has ended (due to loss of life of the agent, timeout, or some other episode termination criteria).
`info`	`Dict`	Some additional information that can optionally be sent out by an environment as a dictionary of arbitrary key-value pairs. The agent we develop should not rely on any of the information in this dictionary for taking action. It may be used (if available) for debugging purposes.

Note that the following code is provided to show the general structure and is not ready to be executed due to the `ENVIRONMENT_NAME` and the `agent.choose_action()` not being defined in this snippet.

Let's put all the pieces together and look at them in one place:

```
import gym
env = gym.make("ENVIRONMENT_NAME")
obs = env.reset() # The first arrow in the picture
# Inner loop (roll out)
action = agent.choose_action(obs) # The second arrow in the picture
next_state, reward, done, info = env.step(action) # The third arrow (and more)
obs = next_state
# Repeat Inner loop (roll out)
```

I hope you got a good understanding of one cycle of the interaction between the environment and the agent. This process will repeat until we decide to terminate the cycle after a certain number of episodes or steps have passed. Let's now have a look at a complete example with the inner loop running for `MAX_STEPS_PER_EPISODE` and the outer loop running for `MAX_NUM_EPISODES` in a `Qbert-v0` environment:

```
#!/usr/bin/env python
import gym
env = gym.make("Qbert-v0")
MAX_NUM_EPISODES = 10
MAX_STEPS_PER_EPISODE = 500
for episode in range(MAX_NUM_EPISODES):
    obs = env.reset()
    for step in range(MAX_STEPS_PER_EPISODE):
        env.render()
        action = env.action_space.sample()# Sample random action. This will be replaced
by our agent's action when we start developing the agent algorithms
        next_state, reward, done, info = env.step(action) # Send the action to the
environment and receive the next_state, reward and whether done or not
        obs = next_state

        if done is True:
            print("\n Episode #{} ended in {} steps.".format(episode, step+1))
            break
```

When you run this script, you will notice a Qbert screen pop up and Qbert taking random actions and getting a score, as shown here:

You will also see print statements on the console like the following, depending on when the episode ended. Note that the step numbers you get might be different because the actions are random:

```
(rl_gym_book) praveen@ubuntu:~/rl_gym_book/ch4$ python rl_gym_boilerplate_code.py
Episode #0 ended in 375 steps.
Episode #1 ended in 363 steps.
Episode #3 ended in 495 steps.
Episode #4 ended in 437 steps.
Episode #5 ended in 355 steps.
Episode #6 ended in 443 steps.
Episode #7 ended in 407 steps.
Episode #8 ended in 400 steps.
Episode #9 ended in 370 steps.
```

The boilerplate code is available in this book's code repository under the `ch4` folder and is named `rl_gym_boilerplate_code.py`. It is indeed boilerplate code, because the overall structure of the program will remain the same. When we build our intelligent agents in subsequent chapters, we will extend this boilerplate code. It is worth taking a while and going through the script line by line to make sure you understand it well.

You may have noticed that in the example code snippets provided we used `env.action_space.sample()` in place of `action` in the previous code. `env.action_space` returns the type of the action space (`Discrete(18)`, for example, in the case of Alien-v0), and the `sample()` method randomly samples a value from that `action_space`. That's all it means!

We will now have a closer look at the spaces in the Gym to understand the state space and action spaces of environments.

Spaces in the Gym

We can see that each environment in the Gym is different. Every game environment under the Atari category is also different from the others. For example, in the case of the `VideoPinball-v0` environment, the goal is to keep bouncing a ball with two paddles to collect points based on where the ball hits, and to make sure that the ball never falls below the paddles, whereas in the case of `Alien-v0`, which is another Atari game environment, the goal is to move through a maze (the rooms in a ship) collecting *dots*, which are equivalent to destroying the eggs of the alien. Aliens can be killed by collecting a pulsar dot and the reward/score increases when that happens. Do you see the variations in the games/environments? How do we know what types of actions are valid in a game?

In the VideoPinball environment, naturally, the actions are to move the paddles up or down, whereas in the Alien environment, the actions are to command the player to move left, right, up, or down. Note that there is no "move left" or "move right" action in the case of VideoPinball. When we look at other categories of environment, the variations are even greater. For example, in the case of continuous control environments such as recently release robotics environments with the fetch robot arms, the action is to vary the continuous valued join positions and joint velocities to achieve the task. The same discussion can be had with respect to the values of the observations from the environment. We already saw the different observation object types in the case of Atari (RAM versus RGB images).

This is the motivation for why the *spaces* (as in mathematics) for the observation and actions are defined for each environment. At the time of the writing of this book, there are six spaces (plus one more called `prng` for random seed) that are supported by OpenAI Gym. They are listed in this table, with a brief description of each:

Space type	Description	Usage Example
	A box in the $\mathbb{R}^n$ space (an n-dimensional	

`Box`	box) where each coordinate is bounded to lie in the interval defined by [low,high]. Values will be an array of *n* numbers. The shape defines the *n* for the space.	`gym.spaces.Box(low=-100, high=100, shape=(2,))`
`Discrete`	Discrete, integer-value space in the interval [0,n-1]. The argument for `Discrete()` defines *n*.	`gym.spaces.Discrete(4)`
`Dict`	A dictionary of sample space to create arbitrarily complex space. In the example, a Dict space is created, which consists of two discrete spaces for positions and velocities in three dimensions.	`gym.spaces.Dict({"position": gym.spaces.Discrete(3), "velocity": gym.spaces.Discrete(3)})`
`MultiBinary`	*n*-dimensional binary space. The argument to `MultiBinary()`defines *n*.	`gym.spaces.MultiBinary(5)`
`MultiDiscrete`	Multi-dimensional discrete space.	`gym.spaces.MultiDiscrete([-10,10], [0,1])`
`Tuple`	A product of simpler spaces.	`gym.spaces.Tuple((gym.spaces.Discrete(2), spaces.Discrete(2)))`

`Box` and `Discrete` are the most commonly used action spaces. We now have a basic understanding of the various space types available in the Gym. Let's look at how to find which observation and action spaces an environment uses.

The following script will print the observation and the action space of a given environment, and also optionally print the lower bound and upper bound of the values in the case of a `Box Space`. Additionally, it will also print a description/meaning of the possible action in the environment if it is provided by the environment:

```
#!/usr/bin/env python
import gym
from gym.spaces import *
import sys

def print_spaces(space):
    print(space)
    if isinstance(space, Box): # Print lower and upper bound if it's a Box space
        print("\n space.low: ", space.low)
        print("\n space.high: ", space.high)


if __name__ == "__main__":
    env = gym.make(sys.argv[1])
    print("Observation Space:")
    print_spaces(env.observation_space)
    print("Action Space:")
    print_spaces(env.action_space)
    try:
        print("Action description/meaning:",env.unwrapped.get_action_meanings())
    except AttributeError:
        pass
```

This script is also available for download in this book's code repository under `ch4`, named `get_observation_action_space.py`. You can run the script using the following command, where we supply the name of the environment as the first argument to the script:

```
(rl_gym_book) praveen@ubuntu:~/rl_gym_book/ch4$ python get_observation_action_space.py
CartPole-v0
```

The script will print an output like this:

```
(rl_gym_book) praveen@ubuntu:~/rl_gym_book/ch4$ python get_observation_action_space.py CartPole-v0
WARN: gym.spaces.Box autodetected dtype as <class 'numpy.float32'>. Please provide explicit dtype.
Observation Space:
Box(4,)

 space.low:  [ -4.80000019e+00  -3.40282347e+38  -4.18879032e-01  -3.40282347e+38]

 space.high:  [  4.80000019e+00   3.40282347e+38   4.18879032e-01   3.40282347e+38]
Action Space:
Discrete(2)
```

In this example, the script prints that the observation space for the `CartPole-v0` environment is `Box(4,)`, which corresponds to `cart position`, `cart velocity`, `pole angle`, and `pole velocity` at the tip for the four box values.

The action space is printed out to be `Discrete(2)`, which corresponds to *push cart to left* and *push cart to right* for the discrete values `0` and `1`, respectively.

Let's have a look at another example that has a few more complex spaces. This time, let's run the script with the `BipedalWalker-v2` environment:

```
(rl_gym_book) praveen@ubuntu:~/rl_gym_book/ch4$ python get_observation_action_space.py
BipedalWalker-v2
```

That produces an output like this:

```
(rl_gym_book) praveen@ubuntu:~/rl_gym_book/ch4$ python get_observation_action_space.py BipedalWalker-v2
WARN: gym.spaces.Box autodetected dtype as <class 'numpy.float32'>. Please provide explicit dtype.
WARN: gym.spaces.Box autodetected dtype as <class 'numpy.float32'>. Please provide explicit dtype.
Observation Space:
Box(24,)

 space.low:  [-inf -inf -inf -inf -inf -inf -inf -inf -inf -inf -inf -inf -inf -inf -inf
 -inf -inf -inf -inf -inf -inf -inf -inf -inf]

 space.high:  [ inf  inf  inf  inf  inf  inf  inf  inf  inf  inf  inf  inf  inf  inf  inf
  inf  inf  inf  inf  inf  inf  inf  inf  inf]
Action Space:
Box(4,)

 space.low:  [-1. -1. -1. -1.]

 space.high:  [ 1.  1.  1.  1.]
```

A detailed description of the state space of the Bipedal Walker (v2) environment is tabulated here for your quick and easy reference:

Index	**Name/description**	**Min**	**Max**
0	hull_angle	0	2*pi
1	hull_angularVelocity	-inf	+inf
2	vel_x	-1	+1
3	vel_y	-1	+1

4	hip_joint_1_angle	-inf	+inf
5	hip_joint_1_speed	-inf	+inf
6	knee_joint_1_angle	-inf	+inf
7	knee_joint_1_speed	-inf	+inf
8	leg_1_ground_contact_flag	0	1
9	hip_joint_2_angle	-inf	+inf
10	hip_joint_2_speed	-inf	+inf
11	knee_joint_2_angle	-inf	+inf
12	knee_joint_2_speed	-inf	+inf
13	leg_2_ground_contact_flag	0	1
14-23	10 lidar readings	-inf	+inf

The state space, as you can see, is quite complicated, which is reasonable for a complex bipedal walking robot. It more or less resembles an actual bipedal robot system and sensor configuration that we can find in the real world, such as Boston Dynamics' (part of Alphabet) Atlas bipedal robot, who stole the limelight during the DARPA Robotics Challenge in 2015.

Next, we will look into and understand the action space. A detailed description of the action space for the Bipedal Walker (v2) environment is tabulated here for your quick and easy reference:

Index	Name/description	Min	Max
0	Hip_1 (torque/velocity)	-1	+1
1	Knee_1 (torque/velocity)	-1	+1
2	Hip_2 (torque/velocity)	-1	+1
3	Knee_2 (torque/velocity)	-1	+1

Action

The torque control is the default control method, which controls the amount of torque applied at the joints.