Documentation

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. It makes no assumptions about the structure of your agent, and is compatible with any numerical computation library, such as TensorFlow or Theano. You can use it from Python code, and soon from other languages.

OpenAI Gym consists of two parts:

1. The gym open-source library: a collection of test problems — environments — that you can use to work out your reinforcement learning algorithms. These environments have a shared interface, allowing you to write general algorithms.
2. The OpenAI Gym service: a site and API allowing people to meaningfully compare performance of their trained agents.

Getting started

To get started, you'll need to have Python 2.7 or Python 3.5. Fetch the `gym` code using:

```
git clone https://github.com/openai/gym
cd gym
pip install -e . # minimal install
```

You can later run `pip install -e .[all]` to do a full install (this requires `cmake` and a recent `pip` version).

`pip install gym` and `pip install gym[all]` will also work if you prefer to fetch `gym` as a package.

Running an environment

Here's a bare minimum example of getting something running. This will run an instance of the `CartPole-v0` environment for 1000 timesteps, rendering the environment at each step. You should see a window pop up rendering the classic cart-pole problem:

```
import gym
env = gym.make('CartPole-v0')
env.reset()
for _ in range(1000):
    env.render()
    env.step(env.action_space.sample()) # take a random action
```

It should look something like this:

Normally, we'll end the simulation before the cart-pole is allowed to go off-screen. More on that later.

If you'd like to see some other environments in action, try replacing `CartPole-v0` above with something like `MountainCar-v0`, `MsPacman-v0` (requires the Atari dependency), or `Hopper-v1` (requires the MuJoCo dependencies). Environments all descend from the `Env` base class.

Note that if you're missing any dependencies, you should get a helpful error message telling you what you're missing. (Let us knowif a dependency gives you trouble without a clear instruction to fix it.) Installing a missing dependency is generally pretty simple. You'll also need a MuJoCo license for `Hopper-v1`.

Observations

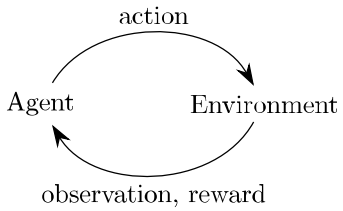
If we ever want to do better than take random actions at each step, it'd probably be good to actually know what our actions are doing to the environment.

The environment's `step` function returns exactly what we need. In fact, `step` returns four values. These are:

- `observation` (object): an environment-specific object representing your observation of the environment. For example, pixel data from a camera, joint angles and joint velocities of a robot, or the board state in a board game.

- `reward` (float): amount of reward achieved by the previous action. The scale varies between environments, but the goal is always to increase your total reward.
- `done` (boolean): whether it's time to `reset` the environment again. Most (but not all) tasks are divided up into well-defined episodes, and `done` being `True` indicates the episode has terminated. (For example, perhaps the pole tipped too far, or you lost your last life.)
- `info` (dict): diagnostic information useful for debugging. It can sometimes be useful for learning (for example, it might contain the raw probabilities behind the environment's last state change). However, official evaluations of your agent are not allowed to use this for learning.

This is just an implementation of the classic "agent-environment loop". Each timestep, the agent chooses an `action`, and the environment returns an `observation` and a `reward`.



The process gets started by calling `reset`, which returns an initial `observation`. So a more proper way of writing the previous code would be to respect the `done` flag:

```
import gym
env = gym.make('CartPole-v0')
for i_episode in range(20):
    observation = env.reset()
    for t in range(100):
        env.render()
        print(observation)
        action = env.action_space.sample()
        observation, reward, done, info = env.step(action)
        if done:
            print("Episode finished after {} timesteps".format(t+1))
            break
```

This should give a video and output like the following. You should be able to see where the resets happen.

```
[-0.061586   -0.75893141  0.05793238  1.15547541]
[-0.07676463 -0.95475889  0.08104189  1.46574644]
[-0.0958598  -1.15077434  0.11035682  1.78260485]
[-0.11887529 -0.95705275  0.14600892  1.5261692 ]
[-0.13801635 -0.7639636   0.1765323   1.28239155]
[-0.15329562 -0.57147373  0.20218013  1.04977545]
Episode finished after 14 timesteps
[-0.02786724  0.00361763 -0.03938967 -0.01611184]
[-0.02779488 -0.19091794 -0.03971191  0.26388759]
[-0.03161324  0.00474768 -0.03443415 -0.04105167]
```

Spaces

In the examples above, we've been sampling random actions from the environment's action space. But what actually are those actions? Every environment comes with first-class `Space` objects that describe the valid actions and observations:

```
import gym
env = gym.make('CartPole-v0')
print(env.action_space)
#> Discrete(2)
print(env.observation_space)
#> Box(4,)
```

The `Discrete` space allows a fixed range of non-negative numbers, so in this case valid `action` s are either 0 or 1. The `Box` space represents an n-dimensional box, so valid `observations` will be an array of 4 numbers. We can also check the `Box` 's bounds:

```
print(env.observation_space.high)
#> array([ 2.4       ,         inf,  0.20943951,         inf])
```

```
print(env.observation_space.low)
#> array([-2.4       ,        -inf, -0.20943951,        -inf])
```

This introspection can be helpful to write generic code that works for many different environments. `Box` and `Discrete` are the most common `Space`s. You can sample from a `Space` or check that something belongs to it:

```
from gym import spaces
space = spaces.Discrete(8) # Set with 8 elements {0, 1, 2, ..., 7}
x = space.sample()
assert space.contains(x)
assert space.n == 8
```

For `CartPole-v0` one of the actions applies force to the left, and one of them applies force to the right. (Can you figure out which is which?)

Fortunately, the better your learning algorithm, the less you'll have to try to interpret these numbers yourself.

Environments

`gym`'s main purpose is to provide a large collection of environments that expose a common interface and are versioned to allow for comparisons. You can find a listing of them as follows:

```
from gym import envs
print(envs.registry.all())
#> [EnvSpec(DoubleDunk-v0), EnvSpec(InvertedDoublePendulum-v0), EnvSpec(BeamRider-v0), EnvS
pec(Phoenix-ram-v0), EnvSpec(Asterix-v0), EnvSpec(TimePilot-v0), EnvSpec(Alien-v0), EnvSpec
(Robotank-ram-v0), EnvSpec(CartPole-v0), EnvSpec(Berzerk-v0), EnvSpec(Berzerk-ram-v0), EnvS
pec(Gopher-ram-v0), ...
```

This will give you a list of `EnvSpec`s. These define parameters for a particular task, including the number of trials to run and the maximum number of steps. For example, `EnvSpec(Hopper-v1)` defines an environment where the goal is to get a 2D simulated robot to hop; `EnvSpec(Go9x9-v0)` defines a Go game on a 9x9 board.

These environment IDs are treated as opaque strings. In order to ensure valid comparisons for the future, environments will never be changed in a fashion that affects performance, only replaced by newer versions. We currently suffix each environment with a `v0` so that future replacements can naturally be called `v1`, `v2`, etc.

It's very easy to add your own enviromments to the registry (and thus be creatable by `make`). Just `register` them at load time. (If you implement your own environments, please let us know! We're accepting pull requests for new environments.)

Recording and uploading results

Gym makes it simple to record your algorithm's performance on an environment, as well as to take videos of your algorithm's learning. Just wrap your environment with a `Monitor` Wrapper as follows:

```
import gym
from gym import wrappers
env = gym.make('CartPole-v0')
env = wrappers.Monitor(env, '/tmp/cartpole-experiment-1')
for i_episode in range(20):
    observation = env.reset()
    for t in range(100):
        env.render()
        print(observation)
        action = env.action_space.sample()
        observation, reward, done, info = env.step(action)
        if done:
            print("Episode finished after {} timesteps".format(t+1))
            break
```

This will log your algorithm's performance to the provided directory. The `Monitor` is fairly sophisticated, and supports multiple instances of an environment writing to a single directory.

You can then `upload` your results to OpenAI Gym:

```
import gym
gym.upload('/tmp/cartpole-experiment-1', api_key='YOUR_API_KEY')
```

The output should look like the following:

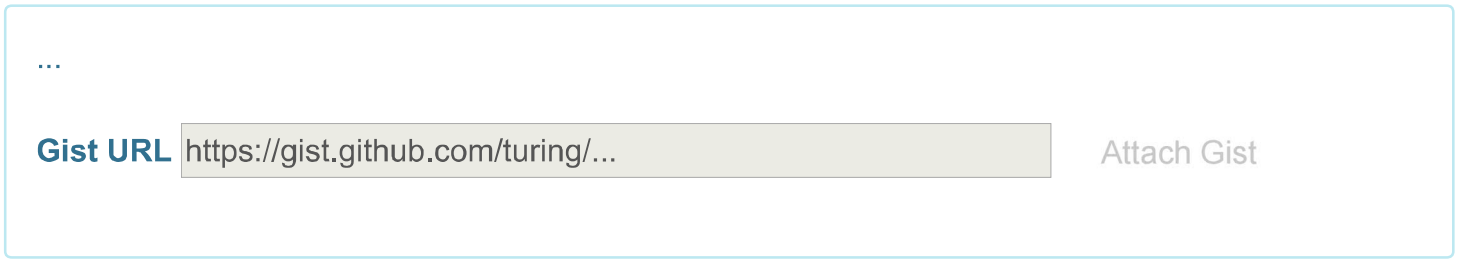
```
[2016-04-22 23:16:03,123] Uploading 20 episodes of training data
[2016-04-22 23:16:04,194] Uploading videos of 2 training episodes (6306 bytes)
[2016-04-22 23:16:04,437] Creating evaluation object on the server with learning curve and
 training video
[2016-04-22 23:16:04,677]
****************************************************
You successfully uploaded your agent evaluation to
OpenAI Gym! You can find it at:

    https://gym.openai.com/evaluations/eval_tmX7tssiRVtYzZk0PlWhKA

****************************************************
```

Evaluations

Each `upload` results in an `evaluation` object on our servers. You should then create a Gist showing how to reproduce your result. Your evaluation page will have a box like the following into which you can past your Gist's URL:



Alternatively, you can provide your Gist at `upload` time by passing a `writeup` parameter:

```
import gym
gym.upload('/tmp/cartpole-experiment-1', writeup='https://gist.github.com/gdb/b6365e79be605
2e7531e7ba6ea8caf23', api_key='YOUR_API_KEY')
```

Your evaluation will be automatically scored, and have a nice page that you can show off to others.

On most environments, your goal is to minimize the number of steps required to achieve a threshold level of performance. (The threshold is defined per environment.) On some environments (especially very difficult ones), it's not yet clear what that threshold should be. There, your goal is instead to maximize final performance.

Reviewed evaluations

(This process will be a work-in-progress for the foreseeable future. The more usage we see, the more we can arrive at something great!)

For each environment, we maintain a list of reviewed evaluations. This is inspired by the mechanics of peer review: we want to make OpenAI Gym ideal for research and collaboration, rather than make it into a leaderboard-based competition. (Currently only OpenAI team members have the ability to curate this list, but out of necessity rather than desire — we'd love to have community moderators once we've gotten our bearings.)

The reviewed list is intended to contain solutions which made a new contribution to OpenAI Gym at the time of creation. At least for now, we require all results to be reproduced by the community. (That way, over time we'll have an open library of correct results and reusable implementations!) New contributions include:

- Coming up with a new algorithm that works much better than anything else.
- Modifying someone else's solution to get meaningfully better performance than they did.
- Implementing an algorithm from the literature not yet on the reviewed list.

It's not just about maximizing score; it's about finding solutions which will generalize well. Solutions which involve task-specific hardcoding or otherwise don't reveal interesting characteristics of learning algorithms are unlikely to pass review.

How to get reviewed

First, keep in mind that all solutions must be reproduced before passing review. So your Gist writeup should make it very easy for others to reproduce your results, or else it's unlikely anyone will go through the effort. Everyone reproducing solutions is a volunteer, so it's always your responsibility to get people excited to do so.

It's probably a good starting point to try to get another community member (friends and coworkers are fine) to reproduce your solution. You can also ask in chat.

How to help

There's a lot you can do to help!

- Solve environments and submit your evaluations!
- Reproduce others' solutions, and let the solution author (constructively) know how it goes by commenting on their Gist. Depending on demand, we'll release tools to help you find solutions to reproduce. Until then you can hang around in chat.
- Integrate new environments into gym. Please show that you have run several well-implemented RL algorithms on it, and you have good reason to believe that it is tractable but not too easy. (If there's demand, we may introduce a registry for sharing environment collections without pulling them into the main repository.)