Spinning Up Documentation

Release

Joshua Achiam

Feb 07, 2020

User Documentation

CHAPTER 1

Introduction

Table of Contents

1.1 What This Is

Welcome to Spinning Up in Deep RL! This is an educational resource produced by OpenAI that makes it easier to learn about deep reinforcement learning (deep RL).

For the unfamiliar: reinforcement learning (RL) is a machine learning approach for teaching agents how to solve tasks by trial and error. Deep RL refers to the combination of RL with deep learning.

This module contains a variety of helpful resources, including:

- a short introduction to RL terminology, kinds of algorithms, and basic theory,
- an essay about how to grow into an RL research role,
- a curated list of important papers organized by topic,
- a well-documented code repo of short, standalone implementations of key algorithms,
- and a few exercises to serve as warm-ups.

1.2 Why We Built This

One of the single most common questions that we hear is

> If I want to contribute to AI safety, how do I get started?

At OpenAI, we believe that deep learning generally—and deep reinforcement learning specifically—will play central roles in the development of powerful AI technology. To ensure that AI is safe, we have to come up with safety strategies and algorithms that are compatible with this paradigm. As a result, we encourage everyone who asks this question to study these fields.

However, while there are many resources to help people quickly ramp up on deep learning, deep reinforcement learning is more challenging to break into. To begin with, a student of deep RL needs to have some background in math, coding, and regular deep learning. Beyond that, they need both a high-level view of the field—an awareness of what topics are studied in it, why they matter, and what's been done already—and careful instruction on how to connect algorithm theory to algorithm code.

The high-level view is hard to come by because of how new the field is. There is not yet a standard deep RL textbook, so most of the knowledge is locked up in either papers or lecture series, which can take a long time to parse and digest. And learning to implement deep RL algorithms is typically painful, because either

- the paper that publishes an algorithm omits or inadvertently obscures key design details,
- or widely-public implementations of an algorithm are hard to read, hiding how the code lines up with the algorithm.

While fantastic repos like garage, Baselines, and rllib make it easier for researchers who are already in the field to make progress, they build algorithms into frameworks in ways that involve many non-obvious choices and trade-offs, which makes them hard to learn from. Consequently, the field of deep RL has a pretty high barrier to entry—for new researchers as well as practitioners and hobbyists.

So our package here is designed to serve as the missing middle step for people who are excited by deep RL, and would like to learn how to use it or make a contribution, but don't have a clear sense of what to study or how to transmute algorithms into code. We've tried to make this as helpful a launching point as possible.

That said, practitioners aren't the only people who can (or should) benefit from these materials. Solving AI safety will require people with a wide range of expertise and perspectives, and many relevant professions have no connection to engineering or computer science at all. Nonetheless, everyone involved will need to learn enough about the technology to make informed decisions, and several pieces of Spinning Up address that need.

1.3 How This Serves Our Mission

OpenAI's mission is to ensure the safe development of AGI and the broad distribution of benefits from AI more generally. Teaching tools like Spinning Up help us make progress on both of these objectives.

To begin with, we move closer to broad distribution of benefits any time we help people understand what AI is and how it works. This empowers people to think critically about the many issues we anticipate will arise as AI becomes more sophisticated and important in our lives.

Also, critically, we need people to help us work on making sure that AGI is safe. This requires a skill set which is currently in short supply because of how new the field is. We know that many people are interested in helping us, but don't know how—here is what you should study! If you can become an expert on this material, you can make a difference on AI safety.

1.4 Code Design Philosophy

The algorithm implementations in the Spinning Up repo are designed to be

- as simple as possible while still being reasonably good,
- and highly-consistent with each other to expose fundamental similarities between algorithms.

They are almost completely self-contained, with virtually no common code shared between them (except for logging, saving, loading, and MPI utilities), so that an interested person can study each algorithm separately without having to dig through an endless chain of dependencies to see how something is done. The implementations are patterned so that they come as close to pseudocode as possible, to minimize the gap between theory and code.

Importantly, they're all structured similarly, so if you clearly understand one, jumping into the next is painless.

We tried to minimize the number of tricks used in each algorithm's implementation, and minimize the differences between otherwise-similar algorithms. To give some examples of removed tricks: we omit regularization terms present in the original Soft-Actor Critic code, as well as observation normalization from all algorithms. For an example of where we've removed differences between algorithms: our implementations of DDPG, TD3, and SAC all follow a convention of running gradient descent updates after fixed intervals of environment interaction. (By contrast, other public implementations of these algorithms usually take slightly different approaches from each other, making them a little bit harder to compare.)

All algorithms are "reasonably good" in the sense that they achieve roughly the intended performance, but don't necessarily match the best reported results in the literature on every task. Consequently, be careful if using any of these implementations for scientific benchmarking comparisons. Details on each implementation's specific performance level can be found on our benchmarks page.

1.5 Long-Term Support and Support History

Spinning Up is currently in maintenance mode. If there are any breaking bugs, we'll repair them to ensure that Spinning Up can continue to help people study deep RL.

Support history so far:

- **Nov 8, 2018:** Initial release!
- **Nov, 2018:** Release was followed by a three-week period of high-bandwidth support.
- **April, 2019:** Approximately six months after release, we conducted an internal review of Spinning Up based on feedback from the community. The review surfaced interest in a few key features:
  - **Implementations in Other Neural Network Libraries.** Several people expressed interest in seeing Spinning Up use alternatives to Tensorflow v1 for the RL implementations. A few members of the community even developed their own PyTorch versions of Spinning Up algorithms, such as Kashif Rasul's Fired Up, Kai Arulkumaran's Spinning Up Basic, and Misha Laskin's Torching Up. As a result, making this kind of "Rosetta Stone" for deep RL became a high priority for future work.
  - **Open Source RL Environments.** Many people expressed an interest in seeing Spinning Up use more open source RL environments (eg PyBullet) for benchmarks, examples, and exercises.
  - **More Algorithms.** There was some interest in seeing other algorithms included in Spinning Up, especially Deep Q-Networks.
- **Jan, 2020:** The PyTorch update to Spinning Up was released!
- **Future:** No major updates are currently planned for Spinning Up. In the event it makes sense for us to release an additional update, following what we found in the 6-month review, the next-highest priority features are to focus more on open source RL environments and adding algorithms.

Additionally, as discussed in the blog post, Spinning Up has been integrated into the curriculum for our Scholars and Fellows programs.

CHAPTER 2

Installation

Table of Contents

Spinning Up requires Python3, OpenAI Gym, and OpenMPI.

Spinning Up is currently only supported on Linux and OSX. It may be possible to install on Windows, though this hasn't been extensively tested.[1]

You Should Know

Many examples and benchmarks in Spinning Up refer to RL environments that use the MuJoCo physics engine. MuJoCo is a proprietary software that requires a license, which is free to trial and free for students, but otherwise is not free. As a result, installing it is optional, but because of its importance to the research community—it is the de facto standard for benchmarking deep RL algorithms in continuous control—it is preferred.

Don't worry if you decide not to install MuJoCo, though. You can definitely get started in RL by running RL algorithms on the Classic Control and Box2d environments in Gym, which are totally free to use.

[1] It looks like at least one person has figured out a workaround for running on Windows. If you try another way and succeed, please let us know how you did it!

2.1 Installing Python

We recommend installing Python through Anaconda. Anaconda is a library that includes Python and many useful packages for Python, as well as an environment manager called conda that makes package management simple.

Follow the installation instructions for Anaconda here. Download and install Anaconda3 (at time of writing, Anaconda3-5.3.0). Then create a conda Python 3.6 env for organizing packages used in Spinning Up:

```
conda create -n spinningup python=3.6
```

To use Python from the environment you just created, activate the environment with:

```
conda activate spinningup
```

You Should Know

If you're new to python environments and package management, this stuff can quickly get confusing or overwhelming, and you'll probably hit some snags along the way. (Especially, you should expect problems like, "I just installed this thing, but it says it's not found when I try to use it!") You may want to read through some clean explanations about what package management is, why it's a good idea, and what commands you'll typically have to execute to correctly use it.

FreeCodeCamp has a good explanation worth reading. There's a shorter description on Towards Data Science which is also helpful and informative. Finally, if you're an extremely patient person, you may want to read the (dry, but very informative) documentation page from Conda.

2.2 Installing OpenMPI

2.2.1 Ubuntu

```
sudo apt-get update && sudo apt-get install libopenmpi-dev
```

2.2.2 Mac OS X

Installation of system packages on Mac requires Homebrew. With Homebrew installed, run the follwing:

```
brew install openmpi
```

2.3 Installing Spinning Up

```
git clone https://github.com/openai/spinningup.git
cd spinningup
pip install -e .
```

You Should Know

Spinning Up defaults to installing everything in Gym **except** the MuJoCo environments. In case you run into any trouble with the Gym installation, check out the Gym github page for help. If you want the MuJoCo environments, see the optional installation section below.

2.4 Check Your Install

To see if you've successfully installed Spinning Up, try running PPO in the LunarLander-v2 environment with

```
python -m spinup.run ppo --hid "[32,32]" --env LunarLander-v2 --exp_name installtest -
↪-gamma 0.999
```

This might run for around 10 minutes, and you can leave it going in the background while you continue reading through documentation. This won't train the agent to completion, but will run it for long enough that you can see *some* learning progress when the results come in.

After it finishes training, watch a video of the trained policy with

```
python -m spinup.run test_policy data/installtest/installtest_s0
```

And plot the results with

```
python -m spinup.run plot data/installtest/installtest_s0
```

2.5 Installing MuJoCo (Optional)

First, go to the mujoco-py github page. Follow the installation instructions in the README, which describe how to install the MuJoCo physics engine and the mujoco-py package (which allows the use of MuJoCo from Python).

You Should Know

In order to use the MuJoCo simulator, you will need to get a MuJoCo license. Free 30-day licenses are available to anyone, and free 1-year licenses are available to full-time students.

Once you have installed MuJoCo, install the corresponding Gym environments with

```
pip install gym[mujoco,robotics]
```

And then check that things are working by running PPO in the Walker2d-v2 environment with

```
python -m spinup.run ppo --hid "[32,32]" --env Walker2d-v2 --exp_name mujocotest
```

CHAPTER 3

Algorithms

Table of Contents

3.1 What's Included

The following algorithms are implemented in the Spinning Up package:

- Vanilla Policy Gradient (VPG)
- Trust Region Policy Optimization (TRPO)
- Proximal Policy Optimization (PPO)
- Deep Deterministic Policy Gradient (DDPG)
- Twin Delayed DDPG (TD3)
- Soft Actor-Critic (SAC)

They are all implemented with MLP (non-recurrent) actor-critics, making them suitable for fully-observed, non-image-based RL environments, e.g. the Gym Mujoco environments.

Spinning Up has two implementations for each algorithm (except for TRPO): one that uses PyTorch as the neural network library, and one that uses Tensorflow v1 as the neural network library. (TRPO is currently only available in Tensorflow.)

3.2 Why These Algorithms?

We chose the core deep RL algorithms in this package to reflect useful progressions of ideas from the recent history of the field, culminating in two algorithms in particular—PPO and SAC—which are close to state of the art on reliability and sample efficiency among policy-learning algorithms. They also expose some of the trade-offs that get made in designing and using algorithms in deep RL.

3.2.1 The On-Policy Algorithms

Vanilla Policy Gradient is the most basic, entry-level algorithm in the deep RL space because it completely predates the advent of deep RL altogether. The core elements of VPG go all the way back to the late 80s / early 90s. It started a trail of research which ultimately led to stronger algorithms such as TRPO and then PPO soon after.

A key feature of this line of work is that all of these algorithms are *on-policy*: that is, they don't use old data, which makes them weaker on sample efficiency. But this is for a good reason: these algorithms directly optimize the objective you care about—policy performance—and it works out mathematically that you need on-policy data to calculate the updates. So, this family of algorithms trades off sample efficiency in favor of stability—but you can see the progression of techniques (from VPG to TRPO to PPO) working to make up the deficit on sample efficiency.

3.2.2 The Off-Policy Algorithms

DDPG is a similarly foundational algorithm to VPG, although much younger—the theory of deterministic policy gradients, which led to DDPG, wasn't published until 2014. DDPG is closely connected to Q-learning algorithms, and it concurrently learns a Q-function and a policy which are updated to improve each other.

Algorithms like DDPG and Q-Learning are *off-policy*, so they are able to reuse old data very efficiently. They gain this benefit by exploiting Bellman's equations for optimality, which a Q-function can be trained to satisfy using *any* environment interaction data (as long as there's enough experience from the high-reward areas in the environment).

But problematically, there are no guarantees that doing a good job of satisfying Bellman's equations leads to having great policy performance. *Empirically* one can get great performance—and when it happens, the sample efficiency is wonderful—but the absence of guarantees makes algorithms in this class potentially brittle and unstable. TD3 and SAC are descendants of DDPG which make use of a variety of insights to mitigate these issues.

3.3 Code Format

All implementations in Spinning Up adhere to a standard template. They are split into two files: an algorithm file, which contains the core logic of the algorithm, and a core file, which contains various utilities needed to run the algorithm.

The algorithm file always starts with a class definition for an experience buffer object, which is used to store information from agent-environment interactions. Next, there is a single function which runs the algorithm. The algorithm function follows a template that is roughly the same across the PyTorch and Tensorflow versions, but we'll break it down for each separately below. Finally, there's some support in each algorithm file for directly running the algorithm

in Gym environments from the command line (though this is not the recommended way to run the algorithms—we'll describe how to do that on the Running Experiments page).

3.3.1 The Algorithm Function: PyTorch Version

The algorithm function for a PyTorch implementation performs the following tasks in (roughly) this order:

1. Logger setup
2. Random seed setting
3. Environment instantiation
4. Constructing the actor-critic PyTorch module via the `actor_critic` function passed to the algorithm function as an argument
5. Instantiating the experience buffer
6. Setting up callable loss functions that also provide diagnostics specific to the algorithm
7. Making PyTorch optimizers
8. Setting up model saving through the logger
9. Setting up an update function that runs one epoch of optimization or one step of descent
10. Running the main loop of the algorithm:
    (a) Run the agent in the environment
    (b) Periodically update the parameters of the agent according to the main equations of the algorithm
    (c) Log key performance metrics and save agent

3.3.2 The Algorithm Function: Tensorflow Version

The algorithm function for a Tensorflow implementation performs the following tasks in (roughly) this order:

1. Logger setup
2. Random seed setting
3. Environment instantiation
4. Making placeholders for the computation graph
5. Building the actor-critic computation graph via the `actor_critic` function passed to the algorithm function as an argument
6. Instantiating the experience buffer
7. Building the computation graph for loss functions and diagnostics specific to the algorithm
8. Making training ops
9. Making the TF Session and initializing parameters
10. Setting up model saving through the logger
11. Defining functions needed for running the main loop of the algorithm (e.g. the core update function, get action function, and test agent function, depending on the algorithm)
12. Running the main loop of the algorithm:
    (a) Run the agent in the environment

(b) Periodically update the parameters of the agent according to the main equations of the algorithm

(c) Log key performance metrics and save agent

3.3.3 The Core File

The core files don't adhere as closely as the algorithms files to a template, but do have some approximate structure:

1. **Tensorflow only:** Functions related to making and managing placeholders
2. Functions for building sections of computation graph relevant to the `actor_critic` method for a particular algorithm
3. Any other useful functions
4. Implementations for an MLP actor-critic compatible with the algorithm, where both the policy and the value function(s) are represented by simple MLPs

CHAPTER 4

Running Experiments

Table of Contents

One of the best ways to get a feel for deep RL is to run the algorithms and see how they perform on different tasks. The Spinning Up code library makes small-scale (local) experiments easy to do, and in this section, we'll discuss two ways to run them: either from the command line, or through function calls in scripts.

4.1 Launching from the Command Line

Spinning Up ships with `spinup/run.py`, a convenient tool that lets you easily launch any algorithm (with any choices of hyperparameters) from the command line. It also serves as a thin wrapper over the utilities for watching trained policies and plotting, although we will not discuss that functionality on this page (for those details, see the pages on experiment outputs and plotting).

The standard way to run a Spinning Up algorithm from the command line is

```
python -m spinup.run [algo name] [experiment flags]
```

eg:

```
python -m spinup.run ppo --env Walker2d-v2 --exp_name walker
```

You Should Know

If you are using ZShell: ZShell interprets square brackets as special characters. Spinning Up uses square brackets in a few ways for command line arguments; make sure to escape them, or try the solution recommended here if you want to escape them by default.

Detailed Quickstart Guide

```
python -m spinup.run ppo --exp_name ppo_ant --env Ant-v2 --clip_ratio 0.1 0.2
    --hid[h] [32,32] [64,32] --act torch.nn.Tanh --seed 0 10 20 --dt
    --data_dir path/to/data
```

runs PPO in the `Ant-v2` Gym environment, with various settings controlled by the flags.

By default, the PyTorch version will run (except for with TRPO, since Spinning Up doesn't have a PyTorch TRPO yet). Substitute `ppo` with `ppo_tf1` for the Tensorflow version.

`clip_ratio`, `hid`, and `act` are flags to set some algorithm hyperparameters. You can provide multiple values for hyperparameters to run multiple experiments. Check the docs to see what hyperparameters you can set (click here for the PPO documentation).

`hid` and `act` are special shortcut flags for setting the hidden sizes and activation function for the neural networks trained by the algorithm.

The `seed` flag sets the seed for the random number generator. RL algorithms have high variance, so try multiple seeds to get a feel for how performance varies.

The `dt` flag ensures that the save directory names will have timestamps in them (otherwise they don't, unless you set `FORCE_DATESTAMP=True` in `spinup/user_config.py`).

The `data_dir` flag allows you to set the save folder for results. The default value is set by `DEFAULT_DATA_DIR` in `spinup/user_config.py`, which will be a subfolder `data` in the `spinningup` folder (unless you change it).

Save directory names are based on `exp_name` and any flags which have multiple values. Instead of the full flag, a shorthand will appear in the directory name. Shorthands can be provided by the user in square brackets after the flag, like `--hid[h]`; otherwise, shorthands are substrings of the flag (`clip_ratio` becomes `cli`). To illustrate, the save directory for the run with `clip_ratio=0.1`, `hid=[32,32]`, and `seed=10` will be:

```
path/to/data/YY-MM-DD_ppo_ant_cli0-1_h32-32/YY-MM-DD_HH-MM-SS-ppo_ant_cli0-1_h32-32_
↪seed10
```

4.1.1 Choosing PyTorch or Tensorflow from the Command Line

To use a PyTorch version of an algorithm, run with

```
python -m spinup.run [algo]_pytorch
```

To use a Tensorflow version of an algorithm, run with

```
python -m spinup.run [algo]_tf1
```

If you run `python -m spinup.run [algo]` without `_pytorch` or `_tf1`, the runner will look in `spinup/user_config.py` for which version it should default to for that algorithm.

4.1.2 Setting Hyperparameters from the Command Line

Every hyperparameter in every algorithm can be controlled directly from the command line. If `kwarg` is a valid keyword arg for the function call of an algorithm, you can set values for it with the flag `--kwarg`. To find out what keyword args are available, see either the docs page for an algorithm, or try

```
python -m spinup.run [algo name] --help
```

to see a readout of the docstring.

You Should Know

Values pass through `eval()` before being used, so you can describe some functions and objects directly from the command line. For example:

```
python -m spinup.run ppo --env Walker2d-v2 --exp_name walker --act torch.nn.ELU
```

sets `torch.nn.ELU` as the activation function. (Tensorflow equivalent: run `ppo_tf1` with `--act tf.nn.elu`.)

You Should Know

There's some nice handling for kwargs that take dict values. Instead of having to provide

```
--key dict(v1=value_1, v2=value_2)
```

you can give

```
--key:v1 value_1 --key:v2 value_2
```

to get the same result.

4.1.3 Launching Multiple Experiments at Once

You can launch multiple experiments, to be executed **in series**, by simply providing more than one value for a given argument. (An experiment for each possible combination of values will be launched.)

For example, to launch otherwise-equivalent runs with different random seeds (0, 10, and 20), do:

```
python -m spinup.run ppo --env Walker2d-v2 --exp_name walker --seed 0 10 20
```

Experiments don't launch in parallel because they soak up enough resources that executing several at the same time wouldn't get a speedup.

4.1.4 Special Flags

A few flags receive special treatment.

Environment Flag

--env, --env_name
string. The name of an environment in the OpenAI Gym. All Spinning Up algorithms are implemented as functions that accept `env_fn` as an argument, where `env_fn` must be a callable function that builds a copy of the RL environment. Since the most common use case is Gym environments, though, all of which are built through `gym.make(env_name)`, we allow you to just specify `env_name` (or `env` for short) at the command line, which gets converted to a lambda-function that builds the correct gym environment.

Shortcut Flags

Some algorithm arguments are relatively long, and we enabled shortcuts for them:

--hid, --ac_kwargs:`hidden_sizes`
list of ints. Sets the sizes of the hidden layers in the neural networks (policies and value functions).

--act, --ac_kwargs:`activation`
tf op. The activation function for the neural networks in the actor and critic.

These flags are valid for all current Spinning Up algorithms.

Config Flags

These flags are not hyperparameters of any algorithm, but change the experimental configuration in some way.

--cpu, --num_cpu
int. If this flag is set, the experiment is launched with this many processes, one per cpu, connected by MPI. Some algorithms are amenable to this sort of parallelization but not all. An error will be raised if you try setting `num_cpu` > 1 for an incompatible algorithm. You can also set `--num_cpu auto`, which will automatically use as many CPUs as are available on the machine.

--exp_name
string. The experiment name. This is used in naming the save directory for each experiment. The default is "cmd" + [algo name].

--data_dir
path. Set the base save directory for this experiment or set of experiments. If none is given, the `DEFAULT_DATA_DIR` in `spinup/user_config.py` will be used.

--datestamp
bool. Include date and time in the name for the save directory of the experiment.

4.1.5 Where Results are Saved

Results for a particular experiment (a single run of a configuration of hyperparameters) are stored in

```
data_dir/[outer_prefix]exp_name[suffix]/[inner_prefix]exp_name[suffix]_s[seed]
```

where

- `data_dir` is the value of the `--data_dir` flag (defaults to `DEFAULT_DATA_DIR` from `spinup/user_config.py` if `--data_dir` is not given),
- the `outer_prefix` is a `YY-MM-DD_` timestamp if the `--datestamp` flag is raised, otherwise nothing,
- the `inner_prefix` is a `YY-MM-DD_HH-MM-SS-` timestamp if the `--datestamp` flag is raised, otherwise nothing,
- and `suffix` is a special string based on the experiment hyperparameters.

How is Suffix Determined?

Suffixes are only included if you run multiple experiments at once, and they only include references to hyperparameters that differ across experiments, except for random seed. The goal is to make sure that results for similar experiments (ones which share all params except seed) are grouped in the same folder.

Suffixes are constructed by combining *shorthands* for hyperparameters with their values, where a shorthand is either 1) constructed automatically from the hyperparameter name or 2) supplied by the user. The user can supply a shorthand by writing in square brackets after the kwarg flag.

For example, consider:

```
python -m spinup.run ddpg_tf1 --env Hopper-v2 --hid[h] [300] [128,128] --act tf.nn.
↪tanh tf.nn.relu
```

Here, the `--hid` flag is given a **user-supplied shorthand**, `h`. The `--act` flag is not given a shorthand by the user, so one will be constructed for it automatically.

The suffixes produced in this case are:

```
_h128-128_ac-actrelu
_h128-128_ac-acttanh
_h300_ac-actrelu
_h300_ac-acttanh
```

Note that the `h` was given by the user. the `ac-act` shorthand was constructed from `ac_kwargs:activation` (the true name for the `act` flag).

4.1.6 Extra

You Don't Actually Need to Know This One

Each individual algorithm is located in a file `spinup/algos/BACKEND/ALGO_NAME/ALGO_NAME.py`, and these files can be run directly from the command line with a limited set of arguments (some of which differ from what's available to `spinup/run.py`). The command line support in the individual algorithm files is essentially vestigial, however, and this is **not** a recommended way to perform experiments.

This documentation page will not describe those command line calls, and will *only* describe calls through `spinup/run.py`.

4.2 Launching from Scripts

Each algorithm is implemented as a python function, which can be imported directly from the `spinup` package, eg

```
>>> from spinup import ppo_pytorch as ppo
```

See the documentation page for each algorithm for a complete account of possible arguments. These methods can be used to set up specialized custom experiments, for example:

```
from spinup import ppo_tf1 as ppo
import tensorflow as tf
import gym

env_fn = lambda : gym.make('LunarLander-v2')

ac_kwargs = dict(hidden_sizes=[64,64], activation=tf.nn.relu)

logger_kwargs = dict(output_dir='path/to/output_dir', exp_name='experiment_name')

ppo(env_fn=env_fn, ac_kwargs=ac_kwargs, steps_per_epoch=5000, epochs=250, logger_
↪kwargs=logger_kwargs)
```

4.2.1 Using ExperimentGrid

It's often useful in machine learning research to run the same algorithm with many possible hyperparameters. Spinning Up ships with a simple tool for facilitating this, called ExperimentGrid.

Consider the example in `spinup/examples/pytorch/bench_ppo_cartpole.py`:

```
from spinup.utils.run_utils import ExperimentGrid
from spinup import ppo_pytorch
import torch

if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('--cpu', type=int, default=4)
    parser.add_argument('--num_runs', type=int, default=3)
    args = parser.parse_args()

    eg = ExperimentGrid(name='ppo-pyt-bench')
    eg.add('env_name', 'CartPole-v0', '', True)
    eg.add('seed', [10*i for i in range(args.num_runs)])
    eg.add('epochs', 10)
    eg.add('steps_per_epoch', 4000)
    eg.add('ac_kwargs:hidden_sizes', [(32,), (64,64)], 'hid')
    eg.add('ac_kwargs:activation', [torch.nn.Tanh, torch.nn.ReLU], '')
    eg.run(ppo_pytorch, num_cpu=args.cpu)
```

(An equivalent Tensorflow example is available in `spinup/examples/tf1/bench_ppo_cartpole.py`.)

After making the ExperimentGrid object, parameters are added to it with

```
eg.add(param_name, values, shorthand, in_name)
```

where `in_name` forces a parameter to appear in the experiment name, even if it has the same value across all experiments.

After all parameters have been added,

```
eg.run(thunk, **run_kwargs)
```

runs all experiments in the grid (one experiment per valid configuration), by providing the configurations as kwargs to the function `thunk`. `ExperimentGrid.run` uses a function named call_experiment to launch `thunk`, and `**run_kwargs` specify behaviors for `call_experiment`. See the documentation page for details.

Except for the absence of shortcut kwargs (you can't use `hid` for `ac_kwargs:hidden_sizes` in `ExperimentGrid`), the basic behavior of `ExperimentGrid` is the same as running things from the command line. (In fact, `spinup.run` uses an `ExperimentGrid` under the hood.)

CHAPTER 5

Experiment Outputs

Table of Contents

In this section we'll cover

- what outputs come from Spinning Up algorithm implementations,
- what formats they're stored in and how they're organized,
- where they are stored and how you can change that,
- and how to load and run trained policies.

You Should Know

Spinning Up implementations currently have no way to resume training for partially-trained agents. If you consider this feature important, please let us know—or consider it a hacking project!

5.1 Algorithm Outputs

Each algorithm is set up to save a training run's hyperparameter configuration, learning progress, trained agent and value functions, and a copy of the environment if possible (to make it easy to load up the agent and environment simultaneously). The output directory contains the following:

Output Directory Structure	
`pyt_save/`	**PyTorch implementations only.** A directory containing everything needed to restore the trained agent and value functions. (*Details for PyTorch saves below.*)
`tf1_save/`	**Tensorflow implementations only.** A directory containing everything needed to restore the trained agent and value functions. (*Details for Tensorflow saves below.*)
`config.json`	A dict containing an as-complete-as-possible description of the args and kwargs you used to launch the training function. If you passed in something which can't be serialized to JSON, it should get handled gracefully by the logger, and the config file will represent it with a string. Note: this is meant for record-keeping only. Launching an experiment from a config file is not currently supported.
`progress.txt`	A tab-separated value file containing records of the metrics recorded by the logger throughout training. eg, `Epoch`, `AverageEpRet`, etc.
`vars.pkl`	A pickle file containing anything about the algorithm state which should get stored. Currently, all algorithms only use this to save a copy of the environment.

You Should Know

Sometimes environment-saving fails because the environment can't be pickled, and `vars.pkl` is empty. This is

known to be a problem for Gym Box2D environments in older versions of Gym, which can't be saved in this manner.

You Should Know

As of 1/30/20, the save directory structure has changed slightly. Previously, Tensorflow graphs were saved in the `simple_save/` folder; this has been replaced with `tf1_save/`.

You Should Know

The only file in here that you should ever have to use "by hand" is the `config.json` file. Our agent testing utility will load things from the `tf1_save/` or `pyt_save/` directory, and our plotter interprets the contents of `progress.txt`, and those are the correct tools for interfacing with these outputs. But there is no tooling for `config.json`—it's just there so that if you forget what hyperparameters you ran an experiment with, you can double-check.

5.1.1 PyTorch Save Directory Info

The `pyt_save` directory contains:

Pyt_Save Directory Structure	
`model.pt`	A file created with `torch.save`, essentially just a pickled PyTorch `nn.Module`. Loading it will restore a trained agent as an ActorCritic object with an `act` method.

5.1.2 Tensorflow Save Directory Info

The `tf1_save` directory contains:

TF1_Save Directory Structure	
`variables/`	A directory containing outputs from the Tensorflow Saver. See documentation for Tensorflow SavedModel.
`model_info.pkl`	A dict containing information (map from key to tensor name) which helps us unpack the saved model after loading.
`saved_model.pb`	A protocol buffer, needed for a Tensorflow SavedModel.

5.2 Save Directory Location

Experiment results will, by default, be saved in the same directory as the Spinning Up package, in a folder called `data`:

```
spinningup/
    data/
        ...
    docs/
        ...
    spinup/
        ...
    LICENSE
    setup.py
```

You can change the default results directory by modifying `DEFAULT_DATA_DIR` in `spinup/user_config.py`.

5.3 Loading and Running Trained Policies

5.3.1 If Environment Saves Successfully

For cases where the environment is successfully saved alongside the agent, it's a cinch to watch the trained agent act in the environment using:

```
python -m spinup.run test_policy path/to/output_directory
```

There are a few flags for options:

-l L, **--len**=L, **default**=0
: *int*. Maximum length of test episode / trajectory / rollout. The default of 0 means no maximum episode length—episodes only end when the agent has reached a terminal state in the environment. (Note: setting L=0 will not prevent Gym envs wrapped by TimeLimit wrappers from ending when they reach their pre-set maximum episode length.)

-n N, **--episodes**=N, **default**=100
: *int*. Number of test episodes to run the agent for.

-nr, **--norender**
: Do not render the test episodes to the screen. In this case, `test_policy` will only print the episode returns and lengths. (Use case: the renderer slows down the testing process, and you just want to get a fast sense of how the agent is performing, so you don't particularly care to watch it.)

-i I, **--itr**=I, **default**=-1
: *int*. This is an option for a special case which is not supported by algorithms in this package as-shipped, but which they are easily modified to do. Use case: Sometimes it's nice to watch trained agents from many different points in training (eg watch at iteration 50, 100, 150, etc.). The logger can do this—save snapshots of the agent from those different points, so they can be run and watched later. In this case, you use this flag to specify which iteration to run. But again: spinup algorithms by default only save snapshots of the most recent agent, overwriting the old snapshots.

  The default value of this flag means "use the latest snapshot."

  To modify an algo so it does produce multiple snapshots, find the following line (which is present in all of the algorithms):

```
logger.save_state({'env': env}, None)
```

and tweak it to

```
logger.save_state({'env': env}, epoch)
```

Make sure to then also set `save_freq` to something reasonable (because if it defaults to 1, for instance, you'll flood your output directory with one `save` folder for each snapshot—which adds up fast).

`-d, --deterministic`
Another special case, which is only used for SAC. The Spinning Up SAC implementation trains a stochastic policy, but is evaluated using the deterministic *mean* of the action distribution. `test_policy` will default to using the stochastic policy trained by SAC, but you should set the deterministic flag to watch the deterministic mean policy (the correct evaluation policy for SAC). This flag is not used for any other algorithms.

5.3.2 Environment Not Found Error

If the environment wasn't saved successfully, you can expect `test_policy.py` to crash with something that looks like

```
Traceback (most recent call last):
  File "spinup/utils/test_policy.py", line 153, in <module>
    run_policy(env, get_action, args.len, args.episodes, not(args.norender))
  File "spinup/utils/test_policy.py", line 114, in run_policy
    "and we can't run the agent in it. :( nn Check out the readthedocs " +
AssertionError: Environment not found!

 It looks like the environment wasn't saved, and we can't run the agent in it.
 ↪ :(

 Check out the readthedocs page on Experiment Outputs for how to handle this␣
 ↪situation.
```

In this case, watching your agent perform is slightly more of a pain but not impossible, as long as you can recreate your environment easily. Try the following in IPython:

```
>>> from spinup.utils.test_policy import load_policy_and_env, run_policy
>>> import your_env
>>> _, get_action = load_policy_and_env('/path/to/output_directory')
>>> env = your_env.make()
>>> run_policy(env, get_action)
Logging data to /tmp/experiments/1536150702/progress.txt
Episode 0    EpRet -163.830      EpLen 93
Episode 1    EpRet -346.164      EpLen 99
...
```

5.3.3 Using Trained Value Functions

The `test_policy.py` tool doesn't help you look at trained value functions, and if you want to use those, you will have to do some digging by hand. For the PyTorch case, load the saved model file with `torch.load` and check the documentation for each algorithm to see what modules the ActorCritic object has. For the Tensorflow case, load the saved computation graph with the restore_tf_graph function, and check the documentation for each algorithm to see what functions were saved.

CHAPTER 6

Plotting Results

Spinning Up ships with a simple plotting utility for interpreting results. Run it with:

```
python -m spinup.run plot [path/to/output_directory ...] [--legend [LEGEND ...]]
    [--xaxis XAXIS] [--value [VALUE ...]] [--count] [--smooth S]
    [--select [SEL ...]] [--exclude [EXC ...]]
```

Positional Arguments:

logdir
strings. As many log directories (or prefixes to log directories, which the plotter will autocomplete internally) as you'd like to plot from. Logdirs will be searched recursively for experiment outputs.

You Should Know

The internal autocompleting is really handy! Suppose you have run several experiments, with the aim of comparing performance between different algorithms, resulting in a log directory structure of:

```
data/
    bench_algo1/
        bench_algo1-seed0/
        bench_algo1-seed10/
    bench_algo2/
        bench_algo2-seed0/
        bench_algo2-seed10/
```

You can easily produce a graph comparing algo1 and algo2 with:

```
python spinup/utils/plot.py data/bench_algo
```

relying on the autocomplete to find both `data/bench_algo1` and `data/bench_algo2`.

Optional Arguments:

-l, --legend=[LEGEND ...]
strings. Optional way to specify legend for the plot. The plotter legend will automatically use the `exp_name` from the `config.json` file, unless you tell it otherwise through this flag. This only works if you provide a name for each directory that will get plotted. (Note: this may not be the same as the number of logdir args you provide! Recall that the plotter looks for autocompletes of the logdir args: there may be more than one match for a given logdir prefix, and you will need to provide a legend string for each one of those matches—unless you have removed some of them as candidates via selection or exclusion rules (below).)

-x, --xaxis=XAXIS, **default**='TotalEnvInteracts'
string. Pick what column from data is used for the x-axis.

-y, --value=[VALUE ...], **default**='Performance'
strings. Pick what columns from data to graph on the y-axis. Submitting multiple values will produce multiple graphs. Defaults to `Performance`, which is not an actual output of any algorithm. Instead, `Performance` refers to either `AverageEpRet`, the correct performance measure for the on-policy algorithms, or `AverageTestEpRet`, the correct performance measure for the off-policy algorithms. The plotter will automatically figure out which of `AverageEpRet` or `AverageTestEpRet` to report for each separate logdir.

--count
Optional flag. By default, the plotter shows y-values which are averaged across all results that share an `exp_name`, which is typically a set of identical experiments that only vary in random seed. But if you'd like to see all of those curves separately, use the `--count` flag.

-s, --smooth=S, **default**=1
int. Smooth data by averaging it over a fixed window. This parameter says how wide the averaging window will be.

--select=[SEL ...]
strings. Optional selection rule: the plotter will only show curves from logdirs that contain all of these substrings.

--exclude=[EXC ...]
strings. Optional exclusion rule: plotter will only show curves from logdirs that do not contain these substrings.

CHAPTER 7

Part 1: Key Concepts in RL

Table of Contents

Welcome to our introduction to reinforcement learning! Here, we aim to acquaint you with

- the language and notation used to discuss the subject,
- a high-level explanation of what RL algorithms do (although we mostly avoid the question of *how* they do it),
- and a little bit of the core math that underlies the algorithms.

In a nutshell, RL is the study of agents and how they learn by trial and error. It formalizes the idea that rewarding or punishing an agent for its behavior makes it more likely to repeat or forego that behavior in the future.

7.1 What Can RL Do?

RL methods have recently enjoyed a wide variety of successes. For example, it's been used to teach computers to control robots in simulation...

...and in the real world...

It's also famously been used to create breakthrough AIs for sophisticated strategy games, most notably Go and Dota, taught computers to play Atari games from raw pixels, and trained simulated robots to follow human instructions.

7.2 Key Concepts and Terminology

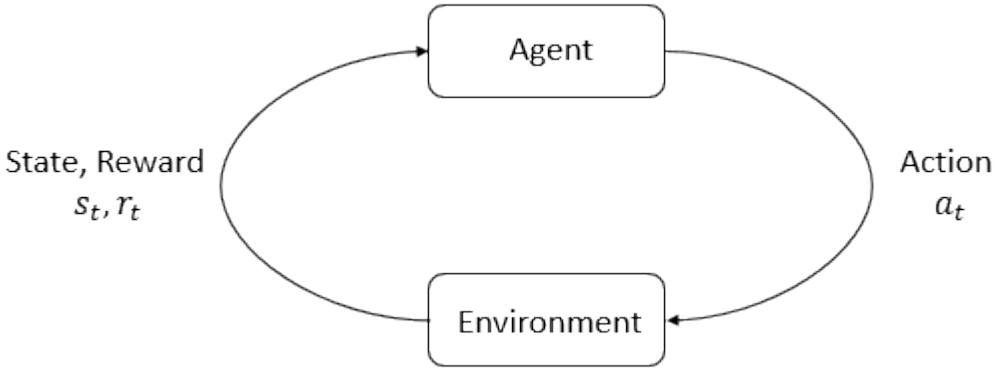

Fig. 7.1: Agent-environment interaction loop.

The main characters of RL are the **agent** and the **environment**. The environment is the world that the agent lives in and interacts with. At every step of interaction, the agent sees a (possibly partial) observation of the state of the world, and then decides on an action to take. The environment changes when the agent acts on it, but may also change on its own.

The agent also perceives a **reward** signal from the environment, a number that tells it how good or bad the current world state is. The goal of the agent is to maximize its cumulative reward, called **return**. Reinforcement learning methods are ways that the agent can learn behaviors to achieve its goal.

To talk more specifically what RL does, we need to introduce additional terminology. We need to talk about

- states and observations,
- action spaces,
- policies,
- trajectories,
- different formulations of return,
- the RL optimization problem,
- and value functions.

7.2.1 States and Observations

A **state** s is a complete description of the state of the world. There is no information about the world which is hidden from the state. An **observation** o is a partial description of a state, which may omit information.

In deep RL, we almost always represent states and observations by a real-valued vector, matrix, or higher-order tensor. For instance, a visual observation could be represented by the RGB matrix of its pixel values; the state of a robot might be represented by its joint angles and velocities.

When the agent is able to observe the complete state of the environment, we say that the environment is **fully observed**. When the agent can only see a partial observation, we say that the environment is **partially observed**.

You Should Know

Reinforcement learning notation sometimes puts the symbol for state, s, in places where it would be technically more appropriate to write the symbol for observation, o. Specifically, this happens when talking about how the agent decides an action: we often signal in notation that the action is conditioned on the state, when in practice, the action is conditioned on the observation because the agent does not have access to the state.

In our guide, we'll follow standard conventions for notation, but it should be clear from context which is meant. If something is unclear, though, please raise an issue! Our goal is to teach, not to confuse.

7.2.2 Action Spaces

Different environments allow different kinds of actions. The set of all valid actions in a given environment is often called the **action space**. Some environments, like Atari and Go, have **discrete action spaces**, where only a finite number of moves are available to the agent. Other environments, like where the agent controls a robot in a physical world, have **continuous action spaces**. In continuous spaces, actions are real-valued vectors.

This distinction has some quite-profound consequences for methods in deep RL. Some families of algorithms can only be directly applied in one case, and would have to be substantially reworked for the other.

7.2.3 Policies

A **policy** is a rule used by an agent to decide what actions to take. It can be deterministic, in which case it is usually denoted by μ:

$$a_t = \mu(s_t),$$

or it may be stochastic, in which case it is usually denoted by π:

$$a_t \sim \pi(\cdot|s_t).$$

Because the policy is essentially the agent's brain, it's not uncommon to substitute the word "policy" for "agent", eg saying "The policy is trying to maximize reward."

In deep RL, we deal with **parameterized policies**: policies whose outputs are computable functions that depend on a set of parameters (eg the weights and biases of a neural network) which we can adjust to change the behavior via some optimization algorithm.

We often denote the parameters of such a policy by θ or ϕ, and then write this as a subscript on the policy symbol to highlight the connection:

$$a_t = \mu_\theta(s_t)$$
$$a_t \sim \pi_\theta(\cdot|s_t).$$

Deterministic Policies

Example: Deterministic Policies. Here is a code snippet for building a simple deterministic policy for a continuous action space in PyTorch, using the `torch.nn` package:

```
pi_net = nn.Sequential(
              nn.Linear(obs_dim, 64),
              nn.Tanh(),
              nn.Linear(64, 64),
              nn.Tanh(),
              nn.Linear(64, act_dim)
            )
```

This builds a multi-layer perceptron (MLP) network with two hidden layers of size 64 and $\tanh$ activation functions. If `obs` is a Numpy array containing a batch of observations, `pi_net` can be used to obtain a batch of actions as follows:

```
obs_tensor = torch.as_tensor(obs, dtype=torch.float32)
actions = pi_net(obs_tensor)
```

You Should Know

Don't worry about it if this neural network stuff is unfamiliar to you—this tutorial will focus on RL, and not on the neural network side of things. So you can skip this example and come back to it later. But we figured that if you already knew, it could be helpful.

Stochastic Policies

The two most common kinds of stochastic policies in deep RL are **categorical policies** and **diagonal Gaussian policies**.

Categorical policies can be used in discrete action spaces, while diagonal Gaussian policies are used in continuous action spaces.

Two key computations are centrally important for using and training stochastic policies:

- sampling actions from the policy,
- and computing log likelihoods of particular actions, $\log \pi_\theta(a|s)$.

In what follows, we'll describe how to do these for both categorical and diagonal Gaussian policies.

Categorical Policies

A categorical policy is like a classifier over discrete actions. You build the neural network for a categorical policy the same way you would for a classifier: the input is the observation, followed by some number of layers (possibly convolutional or densely-connected, depending on the kind of input), and then you have one final linear layer that gives you logits for each action, followed by a softmax to convert the logits into probabilities.

Sampling. Given the probabilities for each action, frameworks like PyTorch and Tensorflow have built-in tools for sampling. For example, see the documentation for Categorical distributions in PyTorch, torch.multinomial, tf.distributions.Categorical, or tf.multinomial.

Log-Likelihood. Denote the last layer of probabilities as $P_\theta(s)$. It is a vector with however many entries as there are actions, so we can treat the actions as indices for the vector. The log likelihood for an action a can then be obtained by indexing into the vector:

$$\log \pi_\theta(a|s) = \log \left[P_\theta(s)\right]_a .$$

Diagonal Gaussian Policies

A multivariate Gaussian distribution (or multivariate normal distribution, if you prefer) is described by a mean vector, μ, and a covariance matrix, Σ. A diagonal Gaussian distribution is a special case where the covariance matrix only has entries on the diagonal. As a result, we can represent it by a vector.

A diagonal Gaussian policy always has a neural network that maps from observations to mean actions, $\mu_\theta(s)$. There are two different ways that the covariance matrix is typically represented.

The first way: There is a single vector of log standard deviations, $\log \sigma$, which is **not** a function of state: the $\log \sigma$ are standalone parameters. (You Should Know: our implementations of VPG, TRPO, and PPO do it this way.)

The second way: There is a neural network that maps from states to log standard deviations, $\log \sigma_\theta(s)$. It may optionally share some layers with the mean network.

Note that in both cases we output log standard deviations instead of standard deviations directly. This is because log stds are free to take on any values in $(-\infty, \infty)$, while stds must be nonnegative. It's easier to train parameters if you don't have to enforce those kinds of constraints. The standard deviations can be obtained immediately from the log standard deviations by exponentiating them, so we do not lose anything by representing them this way.

Sampling. Given the mean action $\mu_\theta(s)$ and standard deviation $\sigma_\theta(s)$, and a vector z of noise from a spherical Gaussian ($z \sim \mathcal{N}(0, I)$), an action sample can be computed with

$$a = \mu_\theta(s) + \sigma_\theta(s) \odot z,$$

where $\odot$ denotes the elementwise product of two vectors. Standard frameworks have built-in ways to generate the noise vectors, such as torch.normal or tf.random_normal. Alternatively, you can build distribution objects, eg through torch.distributions.Normal or tf.distributions.Normal, and use them to generate samples. (The advantage of the latter approach is that those objects can also calculate log-likelihoods for you.)

Log-Likelihood. The log-likelihood of a k -dimensional action a, for a diagonal Gaussian with mean $\mu = \mu_\theta(s)$ and standard deviation $\sigma = \sigma_\theta(s)$, is given by

$$\log \pi_\theta(a|s) = -\frac{1}{2}\left(\sum_{i=1}^{k}\left(\frac{(a_i - \mu_i)^2}{\sigma_i^2} + 2\log \sigma_i\right) + k \log 2\pi\right).$$

7.2.4 Trajectories

A trajectory τ is a sequence of states and actions in the world,

$$\tau = (s_0, a_0, s_1, a_1, ...).$$

The very first state of the world, s_0, is randomly sampled from the **start-state distribution**, sometimes denoted by ρ_0:

$$s_0 \sim \rho_0(\cdot).$$

State transitions (what happens to the world between the state at time t, s_t, and the state at $t+1$, s_{t+1}), are governed by the natural laws of the environment, and depend on only the most recent action, a_t. They can be either deterministic,

$$s_{t+1} = f(s_t, a_t)$$

or stochastic,

$$s_{t+1} \sim P(\cdot|s_t, a_t).$$

Actions come from an agent according to its policy.

You Should Know

Trajectories are also frequently called **episodes** or **rollouts**.

7.2.5 Reward and Return

The reward function R is critically important in reinforcement learning. It depends on the current state of the world, the action just taken, and the next state of the world:

$$r_t = R(s_t, a_t, s_{t+1})$$

although frequently this is simplified to just a dependence on the current state, $r_t = R(s_t)$, or state-action pair $r_t = R(s_t, a_t)$.

The goal of the agent is to maximize some notion of cumulative reward over a trajectory, but this actually can mean a few things. We'll notate all of these cases with $R(\tau)$, and it will either be clear from context which case we mean, or it won't matter (because the same equations will apply to all cases).

One kind of return is the **finite-horizon undiscounted return**, which is just the sum of rewards obtained in a fixed window of steps:

$$R(\tau) = \sum_{t=0}^{T} r_t.$$

Another kind of return is the **infinite-horizon discounted return**, which is the sum of all rewards *ever* obtained by the agent, but discounted by how far off in the future they're obtained. This formulation of reward includes a discount factor $\gamma \in (0, 1)$:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t.$$

Why would we ever want a discount factor, though? Don't we just want to get *all* rewards? We do, but the discount factor is both intuitively appealing and mathematically convenient. On an intuitive level: cash now is better than cash later. Mathematically: an infinite-horizon sum of rewards may not converge to a finite value, and is hard to deal with in equations. But with a discount factor and under reasonable conditions, the infinite sum converges.

You Should Know

While the line between these two formulations of return are quite stark in RL formalism, deep RL practice tends to blur the line a fair bit—for instance, we frequently set up algorithms to optimize the undiscounted return, but use discount factors in estimating **value functions**.

7.2.6 The RL Problem

Whatever the choice of return measure (whether infinite-horizon discounted, or finite-horizon undiscounted), and whatever the choice of policy, the goal in RL is to select a policy which maximizes **expected return** when the agent acts according to it.

To talk about expected return, we first have to talk about probability distributions over trajectories.

Let's suppose that both the environment transitions and the policy are stochastic. In this case, the probability of a T -step trajectory is:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t).$$

The expected return (for whichever measure), denoted by $J(\pi)$, is then:

$$J(\pi) = \int_{\tau} P(\tau|\pi)R(\tau) = \underset{\tau\sim\pi}{\mathrm{E}}\left[R(\tau)\right].$$

The central optimization problem in RL can then be expressed by

$$\pi^* = \arg\max_{\pi} J(\pi),$$

with π^* being the **optimal policy**.

7.2.7 Value Functions

It's often useful to know the **value** of a state, or state-action pair. By value, we mean the expected return if you start in that state or state-action pair, and then act according to a particular policy forever after. **Value functions** are used, one way or another, in almost every RL algorithm.

There are four main functions of note here.

1. The **On-Policy Value Function**, $V^\pi(s)$, which gives the expected return if you start in state s and always act according to policy π:

$$V^\pi(s) = \mathop{\mathrm{E}}_{\tau\sim\pi}\left[R(\tau)\,|s_0 = s\right]$$

2. The **On-Policy Action-Value Function**, $Q^\pi(s,a)$, which gives the expected return if you start in state s, take an arbitrary action a (which may not have come from the policy), and then forever after act according to policy π:

$$Q^\pi(s,a) = \mathop{\mathrm{E}}_{\tau\sim\pi}\left[R(\tau)\,|s_0 = s, a_0 = a\right]$$

3. The **Optimal Value Function**, $V^*(s)$, which gives the expected return if you start in state s and always act according to the *optimal* policy in the environment:

$$V^*(s) = \max_{\pi} \mathop{\mathrm{E}}_{\tau\sim\pi}\left[R(\tau)\,|s_0 = s\right]$$

4. The **Optimal Action-Value Function**, $Q^*(s,a)$, which gives the expected return if you start in state s, take an arbitrary action a, and then forever after act according to the *optimal* policy in the environment:

$$Q^*(s,a) = \max_{\pi} \mathop{\mathrm{E}}_{\tau\sim\pi}\left[R(\tau)\,|s_0 = s, a_0 = a\right]$$

You Should Know

When we talk about value functions, if we do not make reference to time-dependence, we only mean expected **infinite-horizon discounted return**. Value functions for finite-horizon undiscounted return would need to accept time as an argument. Can you think about why? Hint: what happens when time's up?

You Should Know

There are two key connections between the value function and the action-value function that come up pretty often:

$$V^\pi(s) = \mathop{\mathrm{E}}_{a\sim\pi}\left[Q^\pi(s,a)\right],$$

and

$$V^*(s) = \max_a Q^*(s,a).$$

These relations follow pretty directly from the definitions just given: can you prove them?

7.2.8 The Optimal Q-Function and the Optimal Action

There is an important connection between the optimal action-value function $Q^*(s,a)$ and the action selected by the optimal policy. By definition, $Q^*(s,a)$ gives the expected return for starting in state s, taking (arbitrary) action a, and then acting according to the optimal policy forever after.

The optimal policy in s will select whichever action maximizes the expected return from starting in s. As a result, if we have Q^*, we can directly obtain the optimal action, $a^*(s)$, via

$$a^*(s) = \arg\max_a Q^*(s,a).$$

Note: there may be multiple actions which maximize $Q^*(s,a)$, in which case, all of them are optimal, and the optimal policy may randomly select any of them. But there is always an optimal policy which deterministically selects an action.

7.2.9 Bellman Equations

All four of the value functions obey special self-consistency equations called **Bellman equations**. The basic idea behind the Bellman equations is this:

> The value of your starting point is the reward you expect to get from being there, plus the value of wherever you land next.

The Bellman equations for the on-policy value functions are

$$\begin{aligned} V^\pi(s) &= \mathop{\mathrm{E}}_{\substack{a\sim\pi \\ s'\sim P}}\left[r(s,a) + \gamma V^\pi(s')\right], \\ Q^\pi(s,a) &= \mathop{\mathrm{E}}_{s'\sim P}\left[r(s,a) + \gamma \mathop{\mathrm{E}}_{a'\sim\pi}\left[Q^\pi(s',a')\right]\right], \end{aligned}$$

where $s' \sim P$ is shorthand for $s' \sim P(\cdot|s,a)$, indicating that the next state s' is sampled from the environment's transition rules; $a \sim \pi$ is shorthand for $a \sim \pi(\cdot|s)$; and $a' \sim \pi$ is shorthand for $a' \sim \pi(\cdot|s')$.

The Bellman equations for the optimal value functions are

$$\begin{aligned} V^*(s) &= \max_a \mathop{\mathrm{E}}_{s'\sim P}\left[r(s,a) + \gamma V^*(s')\right], \\ Q^*(s,a) &= \mathop{\mathrm{E}}_{s'\sim P}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a')\right]. \end{aligned}$$

The crucial difference between the Bellman equations for the on-policy value functions and the optimal value functions, is the absence or presence of the $\max$ over actions. Its inclusion reflects the fact that whenever the agent gets to choose its action, in order to act optimally, it has to pick whichever action leads to the highest value.

You Should Know

The term "Bellman backup" comes up quite frequently in the RL literature. The Bellman backup for a state, or state-action pair, is the right-hand side of the Bellman equation: the reward-plus-next-value.

7.2.10 Advantage Functions

Sometimes in RL, we don't need to describe how good an action is in an absolute sense, but only how much better it is than others on average. That is to say, we want to know the relative **advantage** of that action. We make this concept precise with the **advantage function.**

The advantage function $A^\pi(s,a)$ corresponding to a policy π describes how much better it is to take a specific action a in state s, over randomly selecting an action according to $\pi(\cdot|s)$, assuming you act according to π forever after. Mathematically, the advantage function is defined by

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s).$$

You Should Know

We'll discuss this more later, but the advantage function is crucially important to policy gradient methods.

7.3 (Optional) Formalism

So far, we've discussed the agent's environment in an informal way, but if you try to go digging through the literature, you're likely to run into the standard mathematical formalism for this setting: **Markov Decision Processes** (MDPs). An MDP is a 5-tuple, $\langle S, A, R, P, \rho_0 \rangle$, where

- S is the set of all valid states,
- A is the set of all valid actions,
- $R : S \times A \times S \to \mathbb{R}$ is the reward function, with $r_t = R(s_t, a_t, s_{t+1})$,
- $P : S \times A \to \mathcal{P}(S)$ is the transition probability function, with $P(s'|s,a)$ being the probability of transitioning into state s' if you start in state s and take action a,
- and ρ_0 is the starting state distribution.

The name Markov Decision Process refers to the fact that the system obeys the Markov property: transitions only depend on the most recent state and action, and no prior history.

CHAPTER 8

Part 2: Kinds of RL Algorithms

Table of Contents

Now that we've gone through the basics of RL terminology and notation, we can cover a little bit of the richer material: the landscape of algorithms in modern RL, and a description of the kinds of trade-offs that go into algorithm design.

8.1 A Taxonomy of RL Algorithms

Fig. 8.1: A non-exhaustive, but useful taxonomy of algorithms in modern RL. *Citations below.*

We'll start this section with a disclaimer: it's really quite hard to draw an accurate, all-encompassing taxonomy of algorithms in the modern RL space, because the modularity of algorithms is not well-represented by a tree structure. Also, to make something that fits on a page and is reasonably digestible in an introduction essay, we have to omit quite a bit of more advanced material (exploration, transfer learning, meta learning, etc). That said, our goals here are

- to highlight the most foundational design choices in deep RL algorithms about what to learn and how to learn it,
- to expose the trade-offs in those choices,
- and to place a few prominent modern algorithms into context with respect to those choices.

8.1.1 Model-Free vs Model-Based RL

One of the most important branching points in an RL algorithm is the question of **whether the agent has access to (or learns) a model of the environment**. By a model of the environment, we mean a function which predicts state

transitions and rewards.

The main upside to having a model is that **it allows the agent to plan** by thinking ahead, seeing what would happen for a range of possible choices, and explicitly deciding between its options. Agents can then distill the results from planning ahead into a learned policy. A particularly famous example of this approach is AlphaZero. When this works, it can result in a substantial improvement in sample efficiency over methods that don't have a model.

The main downside is that **a ground-truth model of the environment is usually not available to the agent.** If an agent wants to use a model in this case, it has to learn the model purely from experience, which creates several challenges. The biggest challenge is that bias in the model can be exploited by the agent, resulting in an agent which performs well with respect to the learned model, but behaves sub-optimally (or super terribly) in the real environment. Model-learning is fundamentally hard, so even intense effort—being willing to throw lots of time and compute at it—can fail to pay off.

Algorithms which use a model are called **model-based** methods, and those that don't are called **model-free**. While model-free methods forego the potential gains in sample efficiency from using a model, they tend to be easier to implement and tune. As of the time of writing this introduction (September 2018), model-free methods are more popular and have been more extensively developed and tested than model-based methods.

8.1.2 What to Learn

Another critical branching point in an RL algorithm is the question of **what to learn.** The list of usual suspects includes

- policies, either stochastic or deterministic,
- action-value functions (Q-functions),
- value functions,
- and/or environment models.

What to Learn in Model-Free RL

There are two main approaches to representing and training agents with model-free RL:

Policy Optimization. Methods in this family represent a policy explicitly as $\pi_\theta(a|s)$. They optimize the parameters θ either directly by gradient ascent on the performance objective $J(\pi_\theta)$, or indirectly, by maximizing local approximations of $J(\pi_\theta)$. This optimization is almost always performed **on-policy**, which means that each update only uses data collected while acting according to the most recent version of the policy. Policy optimization also usually involves learning an approximator $V_\phi(s)$ for the on-policy value function $V^\pi(s)$, which gets used in figuring out how to update the policy.

A couple of examples of policy optimization methods are:

- A2C / A3C, which performs gradient ascent to directly maximize performance,
- and PPO, whose updates indirectly maximize performance, by instead maximizing a *surrogate objective* function which gives a conservative estimate for how much $J(\pi_\theta)$ will change as a result of the update.

Q-Learning. Methods in this family learn an approximator $Q_\theta(s,a)$ for the optimal action-value function, $Q^*(s,a)$. Typically they use an objective function based on the Bellman equation. This optimization is almost always performed **off-policy**, which means that each update can use data collected at any point during training, regardless of how the agent was choosing to explore the environment when the data was obtained. The corresponding policy is obtained via the connection between Q^* and π^*: the actions taken by the Q-learning agent are given by

$$a(s) = \arg\max_a Q_\theta(s,a).$$

Examples of Q-learning methods include

- DQN, a classic which substantially launched the field of deep RL,
- and C51, a variant that learns a distribution over return whose expectation is Q^*.

Trade-offs Between Policy Optimization and Q-Learning. The primary strength of policy optimization methods is that they are principled, in the sense that *you directly optimize for the thing you want.* This tends to make them stable and reliable. By contrast, Q-learning methods only *indirectly* optimize for agent performance, by training Q_θ to satisfy a self-consistency equation. There are many failure modes for this kind of learning, so it tends to be less stable.[1] But, Q-learning methods gain the advantage of being substantially more sample efficient when they do work, because they can reuse data more effectively than policy optimization techniques.

Interpolating Between Policy Optimization and Q-Learning. Serendipitously, policy optimization and Q-learning are not incompatible (and under some circumstances, it turns out, equivalent), and there exist a range of algorithms that live in between the two extremes. Algorithms that live on this spectrum are able to carefully trade-off between the strengths and weaknesses of either side. Examples include

- DDPG, an algorithm which concurrently learns a deterministic policy and a Q-function by using each to improve the other,
- and SAC, a variant which uses stochastic policies, entropy regularization, and a few other tricks to stabilize learning and score higher than DDPG on standard benchmarks.

8.1.3 What to Learn in Model-Based RL

Unlike model-free RL, there aren't a small number of easy-to-define clusters of methods for model-based RL: there are many orthogonal ways of using models. We'll give a few examples, but the list is far from exhaustive. In each case, the model may either be given or learned.

Background: Pure Planning. The most basic approach *never* explicitly represents the policy, and instead, uses pure planning techniques like model-predictive control (MPC) to select actions. In MPC, each time the agent observes the environment, it computes a plan which is optimal with respect to the model, where the plan describes all actions to take over some fixed window of time after the present. (Future rewards beyond the horizon may be considered by the planning algorithm through the use of a learned value function.) The agent then executes the first action of the plan, and immediately discards the rest of it. It computes a new plan each time it prepares to interact with the environment, to avoid using an action from a plan with a shorter-than-desired planning horizon.

- The MBMF work explores MPC with learned environment models on some standard benchmark tasks for deep RL.

Expert Iteration. A straightforward follow-on to pure planning involves using and learning an explicit representation of the policy, $\pi_\theta(a|s)$. The agent uses a planning algorithm (like Monte Carlo Tree Search) in the model, generating candidate actions for the plan by sampling from its current policy. The planning algorithm produces an action which is better than what the policy alone would have produced, hence it is an "expert" relative to the policy. The policy is afterwards updated to produce an action more like the planning algorithm's output.

- The ExIt algorithm uses this approach to train deep neural networks to play Hex.
- AlphaZero is another example of this approach.

Data Augmentation for Model-Free Methods. Use a model-free RL algorithm to train a policy or Q-function, but either 1) augment real experiences with fictitious ones in updating the agent, or 2) use *only* fictitous experience for updating the agent.

- See MBVE for an example of augmenting real experiences with fictitious ones.
- See World Models for an example of using purely fictitious experience to train the agent, which they call "training in the dream."

[1] For more information about how and why Q-learning methods can fail, see 1) this classic paper by Tsitsiklis and van Roy, 2) the (much more recent) review by Szepesvari (in section 4.3.2), and 3) chapter 11 of Sutton and Barto, especially section 11.3 (on "the deadly triad" of function approximation, bootstrapping, and off-policy data, together causing instability in value-learning algorithms).

Embedding Planning Loops into Policies. Another approach embeds the planning procedure directly into a policy as a subroutine—so that complete plans become side information for the policy—while training the output of the policy with any standard model-free algorithm. The key concept is that in this framework, the policy can learn to choose how and when to use the plans. This makes model bias less of a problem, because if the model is bad for planning in some states, the policy can simply learn to ignore it.

- See I2A for an example of agents being endowed with this style of imagination.

8.2 Links to Algorithms in Taxonomy

CHAPTER 9

Part 3: Intro to Policy Optimization

Table of Contents

In this section, we'll discuss the mathematical foundations of policy optimization algorithms, and connect the material to sample code. We will cover three key results in the theory of **policy gradients**:

- the simplest equation describing the gradient of policy performance with respect to policy parameters,
- a rule which allows us to drop useless terms from that expression,
- and a rule which allows us to add useful terms to that expression.

In the end, we'll tie those results together and describe the advantage-based expression for the policy gradient—the version we use in our Vanilla Policy Gradient implementation.

9.1 Deriving the Simplest Policy Gradient

Here, we consider the case of a stochastic, parameterized policy, π_θ. We aim to maximize the expected return $J(\pi_\theta) = \underset{\tau\sim\pi_\theta}{\mathrm{E}}[R(\tau)]$. For the purposes of this derivation, we'll take $R(\tau)$ to give the finite-horizon undiscounted return, but the derivation for the infinite-horizon discounted return setting is almost identical.

We would like to optimize the policy by gradient ascent, eg

$$\theta_{k+1} = \theta_k + \alpha \left.\nabla_\theta J(\pi_\theta)\right|_{\theta_k}.$$

The gradient of policy performance, $\nabla_\theta J(\pi_\theta)$, is called the **policy gradient**, and algorithms that optimize the policy this way are called **policy gradient algorithms.** (Examples include Vanilla Policy Gradient and TRPO. PPO is often referred to as a policy gradient algorithm, though this is slightly inaccurate.)

To actually use this algorithm, we need an expression for the policy gradient which we can numerically compute. This involves two steps: 1) deriving the analytical gradient of policy performance, which turns out to have the form of an expected value, and then 2) forming a sample estimate of that expected value, which can be computed with data from a finite number of agent-environment interaction steps.

In this subsection, we'll find the simplest form of that expression. In later subsections, we'll show how to improve on the simplest form to get the version we actually use in standard policy gradient implementations.

We'll begin by laying out a few facts which are useful for deriving the analytical gradient.

1. Probability of a Trajectory. The probability of a trajectory $\tau = (s_0, a_0, ..., s_{T+1})$ given that actions come from π_θ is

$$P(\tau|\theta) = \rho_0(s_0) \prod_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t).$$

2. The Log-Derivative Trick. The log-derivative trick is based on a simple rule from calculus: the derivative of $\log x$ with respect to x is $1/x$. When rearranged and combined with chain rule, we get:

$$\nabla_\theta P(\tau|\theta) = P(\tau|\theta)\nabla_\theta \log P(\tau|\theta).$$

3. Log-Probability of a Trajectory. The log-prob of a trajectory is just

$$\log P(\tau|\theta) = \log \rho_0(s_0) + \sum_{t=0}^{T} \Big(\log P(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t) \Big).$$

4. Gradients of Environment Functions. The environment has no dependence on θ, so gradients of $\rho_0(s_0)$, $P(s_{t+1}|s_t, a_t)$, and $R(\tau)$ are zero.

5. Grad-Log-Prob of a Trajectory. The gradient of the log-prob of a trajectory is thus

$$\begin{aligned}\nabla_\theta \log P(\tau|\theta) &= \cancel{\nabla_\theta \log \rho_0(s_0)} + \sum_{t=0}^{T} \Big(\cancel{\nabla_\theta \log P(s_{t+1}|s_t, a_t)} + \nabla_\theta \log \pi_\theta(a_t|s_t) \Big) \\ &= \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t).\end{aligned}$$

Putting it all together, we derive the following:

Derivation for Basic Policy Gradient

$$
\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}[R(\tau)] \\
&= \nabla_\theta \int_\tau P(\tau|\theta)R(\tau) && \text{Expand expectation} \\
&= \int_\tau \nabla_\theta P(\tau|\theta)R(\tau) && \text{Bring gradient under integral} \\
&= \int_\tau P(\tau|\theta)\nabla_\theta \log P(\tau|\theta)R(\tau) && \text{Log-derivative trick} \\
&= \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau|\theta)R(\tau)] && \text{Return to expectation form} \\
\therefore \nabla_\theta J(\pi_\theta) &= \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)\right] && \text{Expression for grad-log-prob}
\end{aligned}
$$

This is an expectation, which means that we can estimate it with a sample mean. If we collect a set of trajectories $\mathcal{D} = \{\tau_i\}_{i=1,\dots,N}$ where each trajectory is obtained by letting the agent act in the environment using the policy π_θ, the policy gradient can be estimated with

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau\in\mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau),$$

where $|\mathcal{D}|$ is the number of trajectories in $\mathcal{D}$ (here, N).

This last expression is the simplest version of the computable expression we desired. Assuming that we have represented our policy in a way which allows us to calculate $\nabla_\theta \log \pi_\theta(a|s)$, and if we are able to run the policy in the environment to collect the trajectory dataset, we can compute the policy gradient and take an update step.

9.2 Implementing the Simplest Policy Gradient

We give a short PyTorch implementation of this simple version of the policy gradient algorithm in `spinup/examples/pytorch/pg_math/1_simple_pg.py`. (It can also be viewed on github.) It is only 128 lines long, so we highly recommend reading through it in depth. While we won't go through the entirety of the code here, we'll highlight and explain a few important pieces.

You Should Know

This section was previously written with a Tensorflow example. The old Tensorflow section can be found here.

1. Making the Policy Network.

```
30 # make core of policy network
31 logits_net = mlp(sizes=[obs_dim]+hidden_sizes+[n_acts])
32
33 # make function to compute action distribution
34 def get_policy(obs):
35     logits = logits_net(obs)
36     return Categorical(logits=logits)
37
38 # make action selection function (outputs int actions, sampled from policy)
```

```
39 def get_action(obs):
40     return get_policy(obs).sample().item()
```

This block builds modules and functions for using a feedforward neural network categorical policy. (See the Stochastic Policies section in Part 1 for a refresher.) The output from the `logits_net` module can be used to construct log-probabilities and probabilities for actions, and the `get_action` function samples actions based on probabilities computed from the logits. (Note: this particular `get_action` function assumes that there will only be one `obs` provided, and therefore only one integer action output. That's why it uses `.item()`, which is used to get the contents of a Tensor with only one element.)

A lot of work in this example is getting done by the `Categorical` object on L36. This is a PyTorch `Distribution` object that wraps up some mathematical functions associated with probability distributions. In particular, it has a method for sampling from the distribution (which we use on L40) and a method for computing log probabilities of given samples (which we use later). Since PyTorch distributions are really useful for RL, check out their documentation to get a feel for how they work.

You Should Know

Friendly reminder! When we talk about a categorical distribution having "logits," what we mean is that the probabilities for each outcome are given by the Softmax function of the logits. That is, the probability for action j under a categorical distribution with logits x_j is:

$$p_j = \frac{\exp(x_j)}{\sum_i \exp(x_i)}$$

---

2. Making the Loss Function.

```
42 # make loss function whose gradient, for the right data, is policy gradient
43 def compute_loss(obs, act, weights):
44     logp = get_policy(obs).log_prob(act)
45     return -(logp * weights).mean()
```

In this block, we build a "loss" function for the policy gradient algorithm. When the right data is plugged in, the gradient of this loss is equal to the policy gradient. The right data means a set of (state, action, weight) tuples collected while acting according to the current policy, where the weight for a state-action pair is the return from the episode to which it belongs. (Although as we will show in later subsections, there are other values you can plug in for the weight which also work correctly.)

You Should Know

Even though we describe this as a loss function, it is **not** a loss function in the typical sense from supervised learning. There are two main differences from standard loss functions.

1. The data distribution depends on the parameters. A loss function is usually defined on a fixed data distribution which is independent of the parameters we aim to optimize. Not so here, where the data must be sampled on the most recent policy.

2. It doesn't measure performance. A loss function usually evaluates the performance metric that we care about. Here, we care about expected return, $J(\pi_\theta)$, but our "loss" function does not approximate this at all, even in expectation. This "loss" function is only useful to us because, when evaluated at the current parameters, with data generated by the current parameters, it has the negative gradient of performance.

But after that first step of gradient descent, there is no more connection to performance. This means that minimizing this "loss" function, for a given batch of data, has *no* guarantee whatsoever of improving expected return. You can send this loss to $-\infty$ and policy performance could crater; in fact, it usually will. Sometimes a deep RL researcher

might describe this outcome as the policy "overfitting" to a batch of data. This is descriptive, but should not be taken literally because it does not refer to generalization error.

We raise this point because it is common for ML practitioners to interpret a loss function as a useful signal during training—"if the loss goes down, all is well." In policy gradients, this intuition is wrong, and you should only care about average return. The loss function means nothing.

You Should Know

The approach used here to make the `logp` tensor–calling the `log_prob` method of a PyTorch `Categorical` object–may require some modification to work with other kinds of distribution objects.

For example, if you are using a Normal distribution (for a diagonal Gaussian policy), the output from calling `policy.log_prob(act)` will give you a Tensor containing separate log probabilities for each component of each vector-valued action. That is to say, you put in a Tensor of shape `(batch, act_dim)`, and get out a Tensor of shape `(batch, act_dim)`, when what you need for making an RL loss is a Tensor of shape `(batch,)`. In that case, you would sum up the log probabilities of the action components to get the log probabilities of the actions. That is, you would compute:

```
logp = get_policy(obs).log_prob(act).sum(axis=-1)
```

---

3. Running One Epoch of Training.

```
# for training policy
def train_one_epoch():
    # make some empty lists for logging.
    batch_obs = []          # for observations
    batch_acts = []         # for actions
    batch_weights = []      # for R(tau) weighting in policy gradient
    batch_rets = []         # for measuring episode returns
    batch_lens = []         # for measuring episode lengths

    # reset episode-specific variables
    obs = env.reset()       # first obs comes from starting distribution
    done = False            # signal from environment that episode is over
    ep_rews = []            # list for rewards accrued throughout ep

    # render first episode of each epoch
    finished_rendering_this_epoch = False

    # collect experience by acting in the environment with current policy
    while True:

        # rendering
        if (not finished_rendering_this_epoch) and render:
            env.render()

        # save obs
        batch_obs.append(obs.copy())

        # act in the environment
        act = get_action(torch.as_tensor(obs, dtype=torch.float32))
        obs, rew, done, _ = env.step(act)

        # save action, reward
        batch_acts.append(act)
```

```
83          ep_rews.append(rew)
84
85          if done:
86              # if episode is over, record info about episode
87              ep_ret, ep_len = sum(ep_rews), len(ep_rews)
88              batch_rets.append(ep_ret)
89              batch_lens.append(ep_len)
90
91              # the weight for each logprob(a|s) is R(tau)
92              batch_weights += [ep_ret] * ep_len
93
94              # reset episode-specific variables
95              obs, done, ep_rews = env.reset(), False, []
96
97              # won't render again this epoch
98              finished_rendering_this_epoch = True
99
100             # end experience loop if we have enough of it
101             if len(batch_obs) > batch_size:
102                 break
103
104     # take a single policy gradient update step
105     optimizer.zero_grad()
106     batch_loss = compute_loss(obs=torch.as_tensor(batch_obs, dtype=torch.float32),
107                               act=torch.as_tensor(batch_acts, dtype=torch.int32),
108                               weights=torch.as_tensor(batch_weights, dtype=torch.
↪float32)
109                               )
110     batch_loss.backward()
111     optimizer.step()
112     return batch_loss, batch_rets, batch_lens
```

The `train_one_epoch()` function runs one "epoch" of policy gradient, which we define to be

1. the experience collection step (L67-102), where the agent acts for some number of episodes in the environment using the most recent policy, followed by
2. a single policy gradient update step (L104-111).

The main loop of the algorithm just repeatedly calls `train_one_epoch()`.

You Should Know

If you aren't already familiar with optimization in PyTorch, observe the pattern for taking one gradient descent step as shown in lines 104-111. First, clear the gradient buffers. Then, compute the loss function. Then, compute a backward pass on the loss function; this accumulates fresh gradients into the gradient buffers. Finally, take a step with the optimizer.

9.3 Expected Grad-Log-Prob Lemma

In this subsection, we will derive an intermediate result which is extensively used throughout the theory of policy gradients. We will call it the Expected Grad-Log-Prob (EGLP) lemma.[1]

[1] The author of this article is not aware of this lemma being given a standard name anywhere in the literature. But given how often it comes up, it seems pretty worthwhile to give it some kind of name for ease of reference.

EGLP Lemma. Suppose that P_θ is a parameterized probability distribution over a random variable, x. Then:

$$\underset{x\sim P_\theta}{\mathrm{E}}\left[\nabla_\theta \log P_\theta(x)\right] = 0.$$

Proof

Recall that all probability distributions are *normalized*:

$$\int_x P_\theta(x) = 1.$$

Take the gradient of both sides of the normalization condition:

$$\nabla_\theta \int_x P_\theta(x) = \nabla_\theta 1 = 0.$$

Use the log derivative trick to get:

$$\begin{aligned} 0 &= \nabla_\theta \int_x P_\theta(x) \\ &= \int_x \nabla_\theta P_\theta(x) \\ &= \int_x P_\theta(x) \nabla_\theta \log P_\theta(x) \\ \therefore 0 &= \underset{x\sim P_\theta}{\mathrm{E}}\left[\nabla_\theta \log P_\theta(x)\right]. \end{aligned}$$

9.4 Don't Let the Past Distract You

Examine our most recent expression for the policy gradient:

$$\nabla_\theta J(\pi_\theta) = \underset{\tau\sim\pi_\theta}{\mathrm{E}}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\right].$$

Taking a step with this gradient pushes up the log-probabilities of each action in proportion to $R(\tau)$, the sum of *all rewards ever obtained*. But this doesn't make much sense.

Agents should really only reinforce actions on the basis of their *consequences*. Rewards obtained before taking an action have no bearing on how good that action was: only rewards that come *after*.

It turns out that this intuition shows up in the math, and we can show that the policy gradient can also be expressed by

$$\nabla_\theta J(\pi_\theta) = \underset{\tau\sim\pi_\theta}{\mathrm{E}}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1})\right].$$

In this form, actions are only reinforced based on rewards obtained after they are taken.

We'll call this form the "reward-to-go policy gradient," because the sum of rewards after a point in a trajectory,

$$\hat{R}_t \doteq \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}),$$

is called the **reward-to-go** from that point, and this policy gradient expression depends on the reward-to-go from state-action pairs.

You Should Know

But how is this better? A key problem with policy gradients is how many sample trajectories are needed to get a low-variance sample estimate for them. The formula we started with included terms for reinforcing actions proportional to past rewards, all of which had zero mean, but nonzero variance: as a result, they would just add noise to sample estimates of the policy gradient. By removing them, we reduce the number of sample trajectories needed.

An (optional) proof of this claim can be found **'here'**_, and it ultimately depends on the EGLP lemma.

9.5 Implementing Reward-to-Go Policy Gradient

We give a short PyTorch implementation of the reward-to-go policy gradient in `spinup/examples/pytorch/pg_math/2_rtg_pg.py`. (It can also be viewed on github.)

The only thing that has changed from `1_simple_pg.py` is that we now use different weights in the loss function. The code modification is very slight: we add a new function, and change two other lines. The new function is:

```
17 def reward_to_go(rews):
18     n = len(rews)
19     rtgs = np.zeros_like(rews)
20     for i in reversed(range(n)):
21         rtgs[i] = rews[i] + (rtgs[i+1] if i+1 < n else 0)
22     return rtgs
```

And then we tweak the old L91-92 from:

```
91                 # the weight for each logprob(a|s) is R(tau)
92                 batch_weights += [ep_ret] * ep_len
```

to:

```
98                 # the weight for each logprob(a_t|s_t) is reward-to-go from t
99                 batch_weights += list(reward_to_go(ep_rews))
```

You Should Know

This section was previously written with a Tensorflow example. The old Tensorflow section can be found here.

9.6 Baselines in Policy Gradients

An immediate consequence of the EGLP lemma is that for any function b which only depends on state,

$$\underset{a_t \sim \pi_\theta}{\mathrm{E}} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) b(s_t) \right] = 0.$$

This allows us to add or subtract any number of terms like this from our expression for the policy gradient, without changing it in expectation:

$$\nabla_\theta J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{\mathrm{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t) \right) \right].$$

Any function b used in this way is called a **baseline**.

The most common choice of baseline is the on-policy value function $V^\pi(s_t)$. Recall that this is the average return an agent gets if it starts in state s_t and then acts according to policy π for the rest of its life.

Empirically, the choice $b(s_t) = V^\pi(s_t)$ has the desirable effect of reducing variance in the sample estimate for the policy gradient. This results in faster and more stable policy learning. It is also appealing from a conceptual angle: it encodes the intuition that if an agent gets what it expected, it should "feel" neutral about it.

You Should Know

In practice, $V^\pi(s_t)$ cannot be computed exactly, so it has to be approximated. This is usually done with a neural network, $V_\phi(s_t)$, which is updated concurrently with the policy (so that the value network always approximates the value function of the most recent policy).

The simplest method for learning V_ϕ, used in most implementations of policy optimization algorithms (including VPG, TRPO, PPO, and A2C), is to minimize a mean-squared-error objective:

$$\phi_k = \arg\min_{\phi} \underset{s_t, \hat{R}_t \sim \pi_k}{\mathrm{E}} \left[\left(V_\phi(s_t) - \hat{R}_t \right)^2 \right],$$

where π_k is the policy at epoch k. This is done with one or more steps of gradient descent, starting from the previous value parameters ϕ_{k-1}.

9.7 Other Forms of the Policy Gradient

What we have seen so far is that the policy gradient has the general form

$$\nabla_\theta J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{\mathrm{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \Phi_t \right],$$

where Φ_t could be any of

$$\Phi_t = R(\tau),$$

or

$$\Phi_t = \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}),$$

or

$$\Phi_t = \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t).$$

All of these choices lead to the same expected value for the policy gradient, despite having different variances. It turns out that there are two more valid choices of weights Φ_t which are important to know.

1. On-Policy Action-Value Function. The choice

$$\Phi_t = Q^{\pi_\theta}(s_t, a_t)$$

is also valid. See this page for an (optional) proof of this claim.

2. The Advantage Function. Recall that the advantage of an action, defined by $A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)$, describes how much better or worse it is than other actions on average (relative to the current policy). This choice,

$$\Phi_t = A^{\pi_\theta}(s_t, a_t)$$

is also valid. The proof is that it's equivalent to using $\Phi_t = Q^{\pi_\theta}(s_t, a_t)$ and then using a value function baseline, which we are always free to do.

You Should Know

The formulation of policy gradients with advantage functions is extremely common, and there are many different ways of estimating the advantage function used by different algorithms.

You Should Know

For a more detailed treatment of this topic, you should read the paper on Generalized Advantage Estimation (GAE), which goes into depth about different choices of Φ_t in the background sections.

That paper then goes on to describe GAE, a method for approximating the advantage function in policy optimization algorithms which enjoys widespread use. For instance, Spinning Up's implementations of VPG, TRPO, and PPO make use of it. As a result, we strongly advise you to study it.

9.8 Recap

In this chapter, we described the basic theory of policy gradient methods and connected some of the early results to code examples. The interested student should continue from here by studying how the later results (value function baselines and the advantage formulation of policy gradients) translate into Spinning Up's implementation of Vanilla Policy Gradient.

CHAPTER 10

Spinning Up as a Deep RL Researcher

By Joshua Achiam, October 13th, 2018

Table of Contents

If you're an aspiring deep RL researcher, you've probably heard all kinds of things about deep RL by this point. You know that it's hard and it doesn't always work. That even when you're following a recipe, reproducibility is a challenge. And that if you're starting from scratch, the learning curve is incredibly steep. It's also the case that there are a lot of great resources out there, but the material is new enough that there's not a clear, well-charted path to mastery. The goal of this column is to help you get past the initial hurdle, and give you a clear sense of how to spin up as a deep RL researcher. In particular, this will outline a useful curriculum for increasing raw knowledge, while interleaving it with the odds and ends that lead to better research.

10.1 The Right Background

Build up a solid mathematical background. From probability and statistics, feel comfortable with random variables, Bayes' theorem, chain rule of probability, expected values, standard deviations, and importance sampling. From multivariate calculus, understand gradients and (optionally, but it'll help) Taylor series expansions.

Build up a general knowledge of deep learning. You don't need to know every single special trick and architecture, but the basics help. Know about standard architectures (MLP, vanilla RNN, LSTM (also see this blog), GRU, conv layers, resnets, attention mechanisms), common regularizers (weight decay, dropout), normalization (batch norm, layer norm, weight norm), and optimizers (SGD, momentum SGD, Adam, others). Know what the reparameterization trick is.

Become familiar with at least one deep learning library. Tensorflow or PyTorch would be a good place to start. You don't need to know how to do everything, but you should feel pretty confident in implementing a simple program to do supervised learning.

Get comfortable with the main concepts and terminology in RL. Know what states, actions, trajectories, policies, rewards, value functions, and action-value functions are. If you're unfamiliar, Spinning Up ships with an introduction to this material; it's also worth checking out the RL-Intro from the OpenAI Hackathon, or the exceptional and thorough overview by Lilian Weng. Optionally, if you're the sort of person who enjoys mathematical theory, study up on the math of monotonic improvement theory (which forms the basis for advanced policy gradient algorithms), or classical RL algorithms (which despite being superseded by deep RL algorithms, contain valuable insights that sometimes drive new research).

10.2 Learn by Doing

Write your own implementations. You should implement as many of the core deep RL algorithms from scratch as you can, with the aim of writing the shortest correct implementation of each. This is by far the best way to develop an understanding of how they work, as well as intuitions for their specific performance characteristics.

Simplicity is critical. You should organize your efforts so that you implement the simplest algorithms first, and only gradually introduce complexity. If you start off trying to build something with too many moving parts, odds are good that it will break and you'll lose weeks trying to debug it. This is a common failure mode for people who are new to deep RL, and if you find yourself stuck in it, don't be discouraged—but do try to change tack and work on a simpler algorithm instead, before returning to the more complex thing later.

Which algorithms? You should probably start with vanilla policy gradient (also called REINFORCE), DQN, A2C (the synchronous version of A3C), PPO (the variant with the clipped objective), and DDPG, approximately in that order. The simplest versions of all of these can be written in just a few hundred lines of code (ballpark 250-300), and some of them even less (for example, a no-frills version of VPG can be written in about 80 lines). Write single-threaded code before you try writing parallelized versions of these algorithms. (Do try to parallelize at least one.)

Focus on understanding. Writing working RL code requires clear, detail-oriented understanding of the algorithms. This is because **broken RL code almost always fails silently,** where the code appears to run fine except that the agent never learns how to solve the task. Usually the problem is that something is being calculated with the wrong equation, or on the wrong distribution, or data is being piped into the wrong place. Sometimes the only way to find these bugs is to read the code with a critical eye, know exactly what it should be doing, and find where it deviates from the correct behavior. Developing that knowledge requires you to engage with both academic literature and other existing implementations (when possible), so a good amount of your time should be spent on that reading.

What to look for in papers: When implementing an algorithm based on a paper, scour that paper, especially the ablation analyses and supplementary material (where available). The ablations will give you an intuition for what parameters or subroutines have the biggest impact on getting things to work, which will help you diagnose bugs. Supplementary material will often give information about specific details like network architecture and optimization hyperparameters, and you should try to align your implementation to these details to improve your chances of getting it working.

But don't overfit to paper details. Sometimes, the paper prescribes the use of more tricks than are strictly necessary, so be a bit wary of this, and try out simplifications where possible. For example, the original DDPG paper suggests a complex neural network architecture and initialization scheme, as well as batch normalization. These aren't strictly necessary, and some of the best-reported results for DDPG use simpler networks. As another example, the original

A3C paper uses asynchronous updates from the various actor-learners, but it turns out that synchronous updates work about as well.

Don't overfit to existing implementations either. Study existing implementations for inspiration, but be careful not to overfit to the engineering details of those implementations. RL libraries frequently make choices for abstraction that are good for code reuse between algorithms, but which are unnecessary if you're only writing a single algorithm or supporting a single use case.

Iterate fast in simple environments. To debug your implementations, try them with simple environments where learning should happen quickly, like CartPole-v0, InvertedPendulum-v0, FrozenLake-v0, and HalfCheetah-v2 (with a short time horizon—only 100 or 250 steps instead of the full 1000) from the OpenAI Gym. Don't try to run an algorithm in Atari or a complex Humanoid environment if you haven't first verified that it works on the simplest possible toy task. Your ideal experiment turnaround-time at the debug stage is <5 minutes (on your local machine) or slightly longer but not much. These small-scale experiments don't require any special hardware, and can be run without too much trouble on CPUs.

If it doesn't work, assume there's a bug. Spend a lot of effort searching for bugs before you resort to tweaking hyperparameters: usually it's a bug. Bad hyperparameters can significantly degrade RL performance, but if you're using hyperparameters similar to the ones in papers and standard implementations, those will probably not be the issue. Also worth keeping in mind: sometimes things will work in one environment even when you have a breaking bug, so make sure to test in more than one environment once your results look promising.

Measure everything. Do a lot of instrumenting to see what's going on under-the-hood. The more stats about the learning process you read out at each iteration, the easier it is to debug—after all, you can't tell it's broken if you can't see that it's breaking. I personally like to look at the mean/std/min/max for cumulative rewards, episode lengths, and value function estimates, along with the losses for the objectives, and the details of any exploration parameters (like mean entropy for stochastic policy optimization, or current epsilon for epsilon-greedy as in DQN). Also, watch videos of your agent's performance every now and then; this will give you some insights you wouldn't get otherwise.

Scale experiments when things work. After you have an implementation of an RL algorithm that seems to work correctly in the simplest environments, test it out on harder environments. Experiments at this stage will take longer—on the order of somewhere between a few hours and a couple of days, depending. Specialized hardware—like a beefy GPU or a 32-core machine—might be useful at this point, and you should consider looking into cloud computing resources like AWS or GCE.

Keep these habits! These habits are worth keeping beyond the stage where you're just learning about deep RL—they will accelerate your research!

10.3 Developing a Research Project

Once you feel reasonably comfortable with the basics in deep RL, you should start pushing on the boundaries and doing research. To get there, you'll need an idea for a project.

Start by exploring the literature to become aware of topics in the field. There are a wide range of topics you might find interesting: sample efficiency, exploration, transfer learning, hierarchy, memory, model-based RL, meta learning, and multi-agent, to name a few. If you're looking for inspiration, or just want to get a rough sense of what's out there, check out Spinning Up's key papers list. Find a paper that you enjoy on one of these subjects—something that inspires you—and read it thoroughly. Use the related work section and citations to find closely-related papers and do a deep dive in the literature. You'll start to figure out where the unsolved problems are and where you can make an impact.

Approaches to idea-generation: There are a many different ways to start thinking about ideas for projects, and the frame you choose influences how the project might evolve and what risks it will face. Here are a few examples:

Frame 1: Improving on an Existing Approach. This is the incrementalist angle, where you try to get performance gains in an established problem setting by tweaking an existing algorithm. Reimplementing prior work is super helpful here, because it exposes you to the ways that existing algorithms are brittle and could be improved. A novice will find this the most accessible frame, but it can also be worthwhile for researchers at any level of experience. While some

researchers find incrementalism less exciting, some of the most impressive achievements in machine learning have come from work of this nature.

Because projects like these are tied to existing methods, they are by nature narrowly scoped and can wrap up quickly (a few months), which may be desirable (especially when starting out as a researcher). But this also sets up the risks: it's possible that the tweaks you have in mind for an algorithm may fail to improve it, in which case, unless you come up with more tweaks, the project is just over and you have no clear signal on what to do next.

Frame 2: Focusing on Unsolved Benchmarks. Instead of thinking about how to improve an existing method, you aim to succeed on a task that no one has solved before. For example: achieving perfect generalization from training levels to test levels in the Sonic domain or Gym Retro. When you hammer away at an unsolved task, you might try a wide variety of methods, including prior approaches and new ones that you invent for the project. It is possible for a novice to approch this kind of problem, but there will be a steeper learning curve.

Projects in this frame have a broad scope and can go on for a while (several months to a year-plus). The main risk is that the benchmark is unsolvable without a substantial breakthrough, meaning that it would be easy to spend a lot of time without making any progress on it. But even if a project like this fails, it often leads the researcher to many new insights that become fertile soil for the next project.

Frame 3: Create a New Problem Setting. Instead of thinking about existing methods or current grand challenges, think of an entirely different conceptual problem that hasn't been studied yet. Then, figure out how to make progress on it. For projects along these lines, a standard benchmark probably doesn't exist yet, and you will have to design one. This can be a huge challenge, but it's worth embracing—great benchmarks move the whole field forward.

Problems in this frame come up when they come up—it's hard to go looking for them.

Avoid reinventing the wheel. When you come up with a good idea that you want to start testing, that's great! But while you're still in the early stages with it, do the most thorough check you can to make sure it hasn't already been done. It can be pretty disheartening to get halfway through a project, and only then discover that there's already a paper about your idea. It's especially frustrating when the work is concurrent, which happens from time to time! But don't let that deter you—and definitely don't let it motivate you to plant flags with not-quite-finished research and over-claim the merits of the partial work. Do good research and finish out your projects with complete and thorough investigations, because that's what counts, and by far what matters most in the long run.

10.4 Doing Rigorous Research in RL

Now you've come up with an idea, and you're fairly certain it hasn't been done. You use the skills you've developed to implement it and you start testing it out on standard domains. It looks like it works! But what does that mean, and how well does it have to work to be important? This is one of the hardest parts of research in deep RL. In order to validate that your proposal is a meaningful contribution, you have to rigorously prove that it actually gets a performance benefit over the strongest possible baseline algorithm—whatever currently achieves SOTA (state of the art) on your test domains. If you've invented a new test domain, so there's no previous SOTA, you still need to try out whatever the most reliable algorithm in the literature is that could plausibly do well in the new test domain, and then you have to beat that.

Set up fair comparisons. If you implement your baseline from scratch—as opposed to comparing against another paper's numbers directly—it's important to spend as much time tuning your baseline as you spend tuning your own algorithm. This will make sure that comparisons are fair. Also, do your best to hold "all else equal" even if there are substantial differences between your algorithm and the baseline. For example, if you're investigating architecture variants, keep the number of model parameters approximately equal between your model and the baseline. Under no circumstances handicap the baseline! It turns out that the baselines in RL are pretty strong, and getting big, consistent wins over them can be tricky or require some good insight in algorithm design.

Remove stochasticity as a confounder. Beware of random seeds making things look stronger or weaker than they really are, so run everything for many random seeds (at least 3, but if you want to be thorough, do 10 or more). This is really important and deserves a lot of emphasis: deep RL seems fairly brittle with respect to random seed in a lot of

common use cases. There's potentially enough variance that two different groups of random seeds can yield learning curves with differences so significant that they look like they don't come from the same distribution at all (see figure 10 here).

Run high-integrity experiments. Don't just take the results from the best or most interesting runs to use in your paper. Instead, launch new, final experiments—for all of the methods that you intend to compare (if you are comparing against your own baseline implementations)—and precommit to report on whatever comes out of that. This is to enforce a weak form of preregistration: you use the tuning stage to come up with your hypotheses, and you use the final runs to come up with your conclusions.

Check each claim separately. Another critical aspect of doing research is to run an ablation analysis. Any method you propose is likely to have several key design decisions—like architecture choices or regularization techniques, for instance—each of which could separately impact performance. The claim you'll make in your work is that those design decisions collectively help, but this is really a bundle of several claims in disguise: one for each such design element. By systematically evaluating what would happen if you were to swap them out with alternate design choices, or remove them entirely, you can figure out how to correctly attribute credit for the benefits your method confers. This lets you make each separate claim with a measure of confidence, and increases the overall strength of your work.

10.5 Closing Thoughts

Deep RL is an exciting, fast-moving field, and we need as many people as possible to go through the open problems and make progress on them. Hopefully, you feel a bit more prepared to be a part of it after reading this! And whenever you're ready, let us know.

10.6 PS: Other Resources

Consider reading through these other informative articles about growing as a researcher or engineer in this field:

Advice for Short-term Machine Learning Research Projects, by Tim Rocktäschel, Jakob Foerster and Greg Farquhar.

ML Engineering for AI Safety & Robustness: a Google Brain Engineer's Guide to Entering the Field, by Catherine Olsson and 80,000 Hours.

10.7 References

CHAPTER 11

Key Papers in Deep RL

What follows is a list of papers in deep RL that are worth reading. This is *far* from comprehensive, but should provide a useful starting point for someone looking to do research in the field.

Table of Contents

11.1 1. Model-Free RL

11.1.1 a. Deep Q-Learning

11.1.2 b. Policy Gradients

11.1.3 c. Deterministic Policy Gradients

11.1.4 d. Distributional RL

11.1.5 e. Policy Gradients with Action-Dependent Baselines

11.1.6 f. Path-Consistency Learning

11.1.7 g. Other Directions for Combining Policy-Learning and Q-Learning

11.1.8 h. Evolutionary Algorithms

11.2 2. Exploration

11.2.1 a. Intrinsic Motivation

11.2.2 b. Unsupervised RL

11.3 3. Transfer and Multitask RL

11.4 4. Hierarchy

11.5 5. Memory

11.6 6. Model-Based RL

11.6.1 a. Model is Learned

11.6.2 b. Model is Given

11.7 7. Meta-RL

11.8 8. Scaling RL

11.9 9. RL in the Real World

11.10 10. Safety

11.11 11. Imitation Learning and Inverse Reinforcement Learning

11.12 12. Reproducibility, Analysis, and Critique

CHAPTER 12

Exercises

Table of Contents

12.1 Problem Set 1: Basics of Implementation

Exercise 1.1: Gaussian Log-Likelihood

Path to Exercise:

- PyTorch version: `spinup/exercises/pytorch/problem_set_1/exercise1_1.py`
- Tensorflow version: `spinup/exercises/tf1/problem_set_1/exercise1_1.py`

Path to Solution:

- PyTorch version: `spinup/exercises/pytorch/problem_set_1_solutions/exercise1_1_soln.py`
- Tensorflow version: `spinup/exercises/tf1/problem_set_1_solutions/exercise1_1_soln.py`

Instructions. Write a function that takes in the means and log stds of a batch of diagonal Gaussian distributions, along with (previously-generated) samples from those distributions, and returns the log likelihoods of those samples. (In the Tensorflow version, you will write a function that creates computation graph operations to do this; in the PyTorch version, you will directly operate on given Tensors.)

You may find it useful to review the formula given in this section of the RL introduction.

Implement your solution in `exercise1_1.py`, and run that file to automatically check your work.

Evaluation Criteria. Your solution will be checked by comparing outputs against a known-good implementation, using a batch of random inputs.

Exercise 1.2: Policy for PPO

Path to Exercise:

- PyTorch version: `spinup/exercises/pytorch/problem_set_1/exercise1_2.py`
- Tensorflow version: `spinup/exercises/tf1/problem_set_1/exercise1_2.py`

Path to Solution:

- PyTorch version: `spinup/exercises/pytorch/problem_set_1_solutions/exercise1_2_soln.py`
- Tensorflow version: `spinup/exercises/tf1/problem_set_1_solutions/exercise1_2_soln.py`

Instructions. Implement an MLP diagonal Gaussian policy for PPO.

Implement your solution in `exercise1_2.py`, and run that file to automatically check your work.

Evaluation Criteria. Your solution will be evaluated by running for 20 epochs in the InvertedPendulum-v2 Gym environment, and this should take in the ballpark of 3-5 minutes (depending on your machine, and other processes you are running in the background). The bar for success is reaching an average score of over 500 in the last 5 epochs, or getting to a score of 1000 (the maximum possible score) in the last 5 epochs.

Exercise 1.3: Computation Graph for TD3

Path to Exercise.

- PyTorch version: `spinup/exercises/pytorch/problem_set_1/exercise1_3.py`
- Tensorflow version: `spinup/exercises/tf1/problem_set_1/exercise1_3.py`

Path to Solution.

- PyTorch version: `spinup/algos/pytorch/td3/td3.py`
- Tensorflow version: `spinup/algos/tf1/td3/td3.py`

Instructions. Implement the main mathematical logic for the TD3 algorithm.

As starter code, you are given the entirety of the TD3 algorithm except for the main mathematical logic (essentially, the loss functions and intermediate calculations needed for them). Find "YOUR CODE HERE" to begin.

You may find it useful to review the pseudocode in our page on TD3.

Implement your solution in `exercise1_3.py`, and run that file to see the results of your work. There is no automatic checking for this exercise.

Evaluation Criteria. Evaluate your code by running `exercise1_3.py` with HalfCheetah-v2, InvertedPendulum-v2, and one other Gym MuJoCo environment of your choosing (set via the `--env` flag). It is set up to use smaller neural networks (hidden sizes [128,128]) than typical for TD3, with a maximum episode length of 150, and to run for only 10 epochs. The goal is to see significant learning progress relatively quickly (in terms of wall clock time). Experiments will likely take on the order of ~10 minutes.

Use the `--use_soln` flag to run Spinning Up's TD3 instead of your implementation. Anecdotally, within 10 epochs, the score in HalfCheetah should go over 300, and the score in InvertedPendulum should max out at 150.

12.2 Problem Set 2: Algorithm Failure Modes

Exercise 2.1: Value Function Fitting in TRPO

Path to Exercise. (Not applicable, there is no code for this one.)

Path to Solution. Solution available here.

Many factors can impact the performance of policy gradient algorithms, but few more drastically than the quality of the learned value function used for advantage estimation.

In this exercise, you will compare results between runs of TRPO where you put lots of effort into fitting the value function (`train_v_iters=80`), versus where you put very little effort into fitting the value function (`train_v_iters=0`).

Instructions. Run the following command:

```
python -m spinup.run trpo --env Hopper-v2 --train_v_iters[v] 0 80 --exp_name ex2-1 --
↪epochs 250 --steps_per_epoch 4000 --seed 0 10 20 --dt
```

and plot the results. (These experiments might take ~10 minutes each, and this command runs six of them.) What do you find?

Exercise 2.2: Silent Bug in DDPG

Path to Exercise.

- PyTorch version: `spinup/exercises/pytorch/problem_set_2/exercise2_2.py`
- Tensorflow version: `spinup/exercises/tf1/problem_set_2/exercise2_2.py`

Path to Solution. Solution available here.

The hardest part of writing RL code is dealing with bugs, because failures are frequently silent. The code will appear to run correctly, but the agent's performance will degrade relative to a bug-free implementation—sometimes to the extent that it never learns anything.

In this exercise, you will observe a bug in vivo and compare results against correct code. The bug is the same (conceptually, if not in exact implementation) for both the PyTorch and Tensorflow versions of this exercise.

Instructions. Run `exercise2_2.py`, which will launch DDPG experiments with and without a bug. The non-bugged version runs the default Spinning Up implementation of DDPG, using a default method for creating the actor and critic networks. The bugged version runs the same DDPG code, except uses a bugged method for creating the networks.

There will be six experiments in all (three random seeds for each case), and each should take in the ballpark of 10 minutes. When they're finished, plot the results. What is the difference in performance with and without the bug?

Without referencing the correct actor-critic code (which is to say—don't look in DDPG's `core.py` file), try to figure out what the bug is and explain how it breaks things.

Hint. To figure out what's going wrong, think about how the DDPG code implements the DDPG computation graph. For the Tensorflow version, look at this excerpt:

```
# Bellman backup for Q function
backup = tf.stop_gradient(r_ph + gamma*(1-d_ph)*q_pi_targ)

# DDPG losses
pi_loss = -tf.reduce_mean(q_pi)
q_loss = tf.reduce_mean((q-backup)**2)
```

How could a bug in the actor-critic code have an impact here?

Bonus. Are there any choices of hyperparameters which would have hidden the effects of the bug?

12.3 Challenges

Write Code from Scratch

As we suggest in the essay, try reimplementing various deep RL algorithms from scratch.

Requests for Research

If you feel comfortable with writing deep learning and deep RL code, consider trying to make progress on any of OpenAI's standing requests for research:

- Requests for Research 1
- Requests for Research 2

CHAPTER 13

Benchmarks for Spinning Up Implementations

Table of Contents

We benchmarked the Spinning Up algorithm implementations in five environments from the MuJoCo Gym task suite: HalfCheetah, Hopper, Walker2d, Swimmer, and Ant.

13.1 Performance in Each Environment

13.1.1 HalfCheetah: PyTorch Versions

Fig. 13.1: 3M timestep benchmark for HalfCheetah-v3 using **PyTorch** implementations.

13.1.2 HalfCheetah: Tensorflow Versions

Fig. 13.2: 3M timestep benchmark for HalfCheetah-v3 using **Tensorflow** implementations.

13.1.3 Hopper: PyTorch Versions

Fig. 13.3: 3M timestep benchmark for Hopper-v3 using **PyTorch** implementations.

13.1.4 Hopper: Tensorflow Versions

13.1.5 Walker2d: PyTorch Versions

13.1.6 Walker2d: Tensorflow Versions

13.1.7 Swimmer: PyTorch Versions

13.1.8 Swimmer: Tensorflow Versions

13.1.9 Ant: PyTorch Versions

13.1.10 Ant: Tensorflow Versions

13.2 Experiment Details

Random seeds. All experiments were run for 10 random seeds each. Graphs show the average (solid line) and std dev (shaded) of performance over random seed over the course of training.

Performance metric. Performance for the on-policy algorithms is measured as the average trajectory return across the batch collected at each epoch. Performance for the off-policy algorithms is measured once every 10,000 steps by running the deterministic policy (or, in the case of SAC, the mean policy) without action noise for ten trajectories, and reporting the average return over those test trajectories.

Network architectures. The on-policy algorithms use networks of size (64, 32) with tanh units for both the policy and the value function. The off-policy algorithms use networks of size (256, 256) with relu units.

Batch size. The on-policy algorithms collected 4000 steps of agent-environment interaction per batch update. The off-policy algorithms used minibatches of size 100 at each gradient descent step.

Fig. 13.4: 3M timestep benchmark for Hopper-v3 using **Tensorflow** implementations.

Fig. 13.5: 3M timestep benchmark for Walker2d-v3 using **PyTorch** implementations.

All other hyperparameters are left at default settings for the Spinning Up implementations. See algorithm pages for details.

Learning curves are smoothed by averaging over a window of 11 epochs.

You Should Know

By comparison to the literature, the Spinning Up implementations of DDPG, TD3, and SAC are roughly at-parity with the best reported results for these algorithms. As a result, you can use the Spinning Up implementations of these algorithms for research purposes.

The Spinning Up implementations of VPG, TRPO, and PPO are overall a bit weaker than the best reported results for these algorithms. This is due to the absence of some standard tricks (such as observation normalization and normalized value regression targets) from our implementations. For research comparisons, you should use the implementations of TRPO or PPO from OpenAI Baselines.

13.3 PyTorch vs Tensorflow

We provide graphs for head-to-head comparisons between the PyTorch and Tensorflow implementations of each algorithm at the following pages:

- VPG Head-to-Head
- PPO Head-to-Head
- DDPG Head-to-Head
- TD3 Head-to-Head
- SAC Head-to-Head

Fig. 13.6: 3M timestep benchmark for Walker2d-v3 using **Tensorflow** implementations.

Fig. 13.7: 3M timestep benchmark for Swimmer-v3 using **PyTorch** implementations.

Fig. 13.8: 3M timestep benchmark for Swimmer-v3 using **Tensorflow** implementations.

Fig. 13.9: 3M timestep benchmark for Ant-v3 using **PyTorch** implementations.

Fig. 13.10: 3M timestep benchmark for Ant-v3 using **Tensorflow** implementations.

CHAPTER 14

Vanilla Policy Gradient

Table of Contents

14.1 Background

(Previously: Introduction to RL, Part 3)

The key idea underlying policy gradients is to push up the probabilities of actions that lead to higher return, and push down the probabilities of actions that lead to lower return, until you arrive at the optimal policy.

14.1.1 Quick Facts

- VPG is an on-policy algorithm.
- VPG can be used for environments with either discrete or continuous action spaces.
- The Spinning Up implementation of VPG supports parallelization with MPI.

14.1.2 Key Equations

Let π_θ denote a policy with parameters θ, and $J(\pi_\theta)$ denote the expected finite-horizon undiscounted return of the policy. The gradient of $J(\pi_\theta)$ is

$$\nabla_\theta J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{\mathrm{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t) \right],$$

where τ is a trajectory and A^{π_θ} is the advantage function for the current policy.

The policy gradient algorithm works by updating policy parameters via stochastic gradient ascent on policy performance:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_{\theta_k})$$

Policy gradient implementations typically compute advantage function estimates based on the infinite-horizon discounted return, despite otherwise using the finite-horizon undiscounted policy gradient formula.

14.1.3 Exploration vs. Exploitation

VPG trains a stochastic policy in an on-policy way. This means that it explores by sampling actions according to the latest version of its stochastic policy. The amount of randomness in action selection depends on both initial conditions and the training procedure. Over the course of training, the policy typically becomes progressively less random, as the update rule encourages it to exploit rewards that it has already found. This may cause the policy to get trapped in local optima.

14.1.4 Pseudocode

Algorithm 1 Vanilla Policy Gradient Algorithm

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

7: Compute policy update, either using standard gradient ascent,

$$\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k,$$

or via another gradient ascent algorithm like Adam.
8: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
9: **end for**

14.2 Documentation

You Should Know

In what follows, we give documentation for the PyTorch and Tensorflow implementations of VPG in Spinning Up. They have nearly identical function calls and docstrings, except for details relating to model construction. However, we include both full docstrings for completeness.

14.2.1 Documentation: PyTorch Version

`spinup.vpg_pytorch`(*env_fn*, *actor_critic=<MagicMock spec='str' id='140475680208208'>*, *ac_kwargs={}*, *seed=0*, *steps_per_epoch=4000*, *epochs=50*, *gamma=0.99*, *pi_lr=0.0003*, *vf_lr=0.001*, *train_v_iters=80*, *lam=0.97*, *max_ep_len=1000*, *logger_kwargs={}*, *save_freq=10*)

Vanilla Policy Gradient

(with GAE-Lambda for advantage estimation)

Parameters

- **`env_fn`** – A function which creates a copy of the environment. The environment must satisfy the OpenAI Gym API.

- **actor_critic** – The constructor method for a PyTorch Module with a `step` method, an `act` method, a `pi` module, and a `v` module. The `step` method should accept a batch of observations and return:

Symbol	Shape	Description
`a`	(batch, act_dim)	Numpy array of actions for each observation.
`v`	(batch,)	Numpy array of value estimates for the provided observations.
`logp_a`	(batch,)	Numpy array of log probs for the actions in `a`.

The `act` method behaves the same as `step` but only returns `a`.

The `pi` module's forward call should accept a batch of observations and optionally a batch of actions, and return:

Symbol	Shape	Description
`pi`	N/A	Torch Distribution object, containing a batch of distributions describing the policy for the provided observations.
`logp_a`	(batch,)	Optional (only returned if batch of actions is given). Tensor containing the log probability, according to the policy, of the provided actions. If actions not given, will contain `None`.

The `v` module's forward call should accept a batch of observations and return:

Symbol	Shape	Description
v	(batch,)	Tensor containing the value estimates for the provided observations. (Critical: make sure to flatten this!)

- **ac_kwargs** (*dict*) – Any kwargs appropriate for the ActorCritic object you provided to VPG.
- **seed** (*int*) – Seed for random number generators.
- **steps_per_epoch** (*int*) – Number of steps of interaction (state-action pairs) for the agent and the environment in each epoch.
- **epochs** (*int*) – Number of epochs of interaction (equivalent to number of policy updates) to perform.
- **gamma** (*float*) – Discount factor. (Always between 0 and 1.)
- **pi_lr** (*float*) – Learning rate for policy optimizer.
- **vf_lr** (*float*) – Learning rate for value function optimizer.
- **train_v_iters** (*int*) – Number of gradient descent steps to take on value function per epoch.
- **lam** (*float*) – Lambda for GAE-Lambda. (Always between 0 and 1, close to 1.)
- **max_ep_len** (*int*) – Maximum length of trajectory / episode / rollout.
- **logger_kwargs** (*dict*) – Keyword args for EpochLogger.
- **save_freq** (*int*) – How often (in terms of gap between epochs) to save the current policy and value function.

14.2.2 Saved Model Contents: PyTorch Version

The PyTorch saved model can be loaded with `ac = torch.load('path/to/model.pt')`, yielding an actor-critic object (`ac`) that has the properties described in the docstring for `vpg_pytorch`.

You can get actions from this model with

```
actions = ac.act(torch.as_tensor(obs, dtype=torch.float32))
```

14.2.3 Documentation: Tensorflow Version

`spinup.`**vpg_tf1**(*env_fn*, *actor_critic=<function mlp_actor_critic>*, *ac_kwargs={}*, *seed=0*, *steps_per_epoch=4000*, *epochs=50*, *gamma=0.99*, *pi_lr=0.0003*, *vf_lr=0.001*, *train_v_iters=80*, *lam=0.97*, *max_ep_len=1000*, *logger_kwargs={}*, *save_freq=10*)

Vanilla Policy Gradient

(with GAE-Lambda for advantage estimation)

Parameters

- **env_fn** – A function which creates a copy of the environment. The environment must satisfy the OpenAI Gym API.
- **actor_critic** – A function which takes in placeholder symbols for state, `x_ph`, and action, `a_ph`, and returns the main outputs from the agent's Tensorflow computation graph:

Symbol	Shape	Description
`pi`	(batch, act_dim)	Samples actions from policy given states.
`logp`	(batch,)	Gives log probability, according to the policy, of taking actions `a_ph` in states `x_ph`.
`logp_pi`	(batch,)	Gives log probability, according to the policy, of the action sampled by `pi`.
`v`	(batch,)	Gives the value estimate for states in `x_ph`. (Critical: make sure to flatten this!)

- **ac_kwargs** (*dict*) – Any kwargs appropriate for the actor_critic function you provided to VPG.
- **seed** (*int*) – Seed for random number generators.
- **steps_per_epoch** (*int*) – Number of steps of interaction (state-action pairs) for the agent and the environment in each epoch.
- **epochs** (*int*) – Number of epochs of interaction (equivalent to number of policy updates) to perform.
- **gamma** (*float*) – Discount factor. (Always between 0 and 1.)
- **pi_lr** (*float*) – Learning rate for policy optimizer.
- **vf_lr** (*float*) – Learning rate for value function optimizer.
- **train_v_iters** (*int*) – Number of gradient descent steps to take on value function per epoch.
- **lam** (*float*) – Lambda for GAE-Lambda. (Always between 0 and 1, close to 1.)

- **max_ep_len** (*int*) – Maximum length of trajectory / episode / rollout.
- **logger_kwargs** (*dict*) – Keyword args for EpochLogger.
- **save_freq** (*int*) – How often (in terms of gap between epochs) to save the current policy and value function.

14.2.4 Saved Model Contents: Tensorflow Version

The computation graph saved by the logger includes:

Key	Value
`x`	Tensorflow placeholder for state input.
`pi`	Samples an action from the agent, conditioned on states in `x`.
`v`	Gives value estimate for states in `x`.

This saved model can be accessed either by

- running the trained policy with the test_policy.py tool,
- or loading the whole saved graph into a program with restore_tf_graph.

14.3 References

14.3.1 Relevant Papers

- Policy Gradient Methods for Reinforcement Learning with Function Approximation, Sutton et al. 2000
- Optimizing Expectations: From Deep Reinforcement Learning to Stochastic Computation Graphs, Schulman 2016(a)
- Benchmarking Deep Reinforcement Learning for Continuous Control, Duan et al. 2016
- High Dimensional Continuous Control Using Generalized Advantage Estimation, Schulman et al. 2016(b)

14.3.2 Why These Papers?

Sutton 2000 is included because it is a timeless classic of reinforcement learning theory, and contains references to the earlier work which led to modern policy gradients. Schulman 2016(a) is included because Chapter 2 contains a lucid introduction to the theory of policy gradient algorithms, including pseudocode. Duan 2016 is a clear, recent benchmark paper that shows how vanilla policy gradient in the deep RL setting (eg with neural network policies and Adam as the optimizer) compares with other deep RL algorithms. Schulman 2016(b) is included because our implementation of VPG makes use of Generalized Advantage Estimation for computing the policy gradient.

14.3.3 Other Public Implementations

- rllab
- rllib (Ray)

CHAPTER 15

Trust Region Policy Optimization

Table of Contents

15.1 Background

(Previously: Background for VPG)

TRPO updates policies by taking the largest step possible to improve performance, while satisfying a special constraint on how close the new and old policies are allowed to be. The constraint is expressed in terms of KL-Divergence, a measure of (something like, but not exactly) distance between probability distributions.

This is different from normal policy gradient, which keeps new and old policies close in parameter space. But even seemingly small differences in parameter space can have very large differences in performance—so a single bad step can collapse the policy performance. This makes it dangerous to use large step sizes with vanilla policy gradients, thus hurting its sample efficiency. TRPO nicely avoids this kind of collapse, and tends to quickly and monotonically improve performance.

15.1.1 Quick Facts

- TRPO is an on-policy algorithm.
- TRPO can be used for environments with either discrete or continuous action spaces.
- The Spinning Up implementation of TRPO supports parallelization with MPI.

15.1.2 Key Equations

Let π_θ denote a policy with parameters θ. The theoretical TRPO update is:

$$\theta_{k+1} = \arg\max_\theta \mathcal{L}(\theta_k, \theta) \\ \text{s.t. } \bar{D}_{KL}(\theta||\theta_k) \leq \delta$$

where $\mathcal{L}(\theta_k, \theta)$ is the *surrogate advantage*, a measure of how policy π_θ performs relative to the old policy π_{θ_k} using data from the old policy:

$$\mathcal{L}(\theta_k, \theta) = \underset{s,a\sim\pi_{\theta_k}}{\mathrm{E}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a)\right],$$

and $\bar{D}_{KL}(\theta||\theta_k)$ is an average KL-divergence between policies across states visited by the old policy:

$$\bar{D}_{KL}(\theta||\theta_k) = \underset{s\sim\pi_{\theta_k}}{\mathrm{E}}\left[D_{KL}\left(\pi_\theta(\cdot|s)||\pi_{\theta_k}(\cdot|s)\right)\right].$$

You Should Know

The objective and constraint are both zero when $\theta = \theta_k$. Furthermore, the gradient of the constraint with respect to θ is zero when $\theta = \theta_k$. Proving these facts requires some subtle command of the relevant math—it's an exercise worth doing, whenever you feel ready!

The theoretical TRPO update isn't the easiest to work with, so TRPO makes some approximations to get an answer quickly. We Taylor expand the objective and constraint to leading order around θ_k:

$$\mathcal{L}(\theta_k, \theta) \approx g^T(\theta - \theta_k) \\ \bar{D}_{KL}(\theta||\theta_k) \approx \frac{1}{2}(\theta - \theta_k)^T H(\theta - \theta_k)$$

resulting in an approximate optimization problem,

$$\theta_{k+1} = \arg\max_\theta g^T(\theta - \theta_k) \\ \text{s.t. } \frac{1}{2}(\theta - \theta_k)^T H(\theta - \theta_k) \leq \delta.$$

You Should Know

By happy coincidence, the gradient g of the surrogate advantage function with respect to θ, evaluated at $\theta = \theta_k$, is exactly equal to the policy gradient, $\nabla_\theta J(\pi_\theta)$! Try proving this, if you feel comfortable diving into the math.

This approximate problem can be analytically solved by the methods of Lagrangian duality[1], yielding the solution:

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g.$$

If we were to stop here, and just use this final result, the algorithm would be exactly calculating the Natural Policy Gradient. A problem is that, due to the approximation errors introduced by the Taylor expansion, this may not satisfy the KL constraint, or actually improve the surrogate advantage. TRPO adds a modification to this update rule: a backtracking line search,

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g,$$

where $\alpha \in (0, 1)$ is the backtracking coefficient, and j is the smallest nonnegative integer such that $\pi_{\theta_{k+1}}$ satisfies the KL constraint and produces a positive surrogate advantage.

Lastly: computing and storing the matrix inverse, H^{-1}, is painfully expensive when dealing with neural network policies with thousands or millions of parameters. TRPO sidesteps the issue by using the conjugate gradient algorithm to solve $Hx = g$ for $x = H^{-1}g$, requiring only a function which can compute the matrix-vector product Hx instead of computing and storing the whole matrix H directly. This is not too hard to do: we set up a symbolic operation to calculate

$$Hx = \nabla_\theta \left(\left(\nabla_\theta \bar{D}_{KL}(\theta || \theta_k) \right)^T x \right),$$

which gives us the correct output without computing the whole matrix.

15.1.3 Exploration vs. Exploitation

TRPO trains a stochastic policy in an on-policy way. This means that it explores by sampling actions according to the latest version of its stochastic policy. The amount of randomness in action selection depends on both initial conditions and the training procedure. Over the course of training, the policy typically becomes progressively less random, as the update rule encourages it to exploit rewards that it has already found. This may cause the policy to get trapped in local optima.

[1] See Convex Optimization by Boyd and Vandenberghe, especially chapters 2 through 5.

15.1.4 Pseudocode

Algorithm 2 Trust Region Policy Optimization

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: Hyperparameters: KL-divergence limit δ, backtracking coefficient α, maximum number of backtracking steps K
3: **for** $k = 0, 1, 2, ...$ **do**
4: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
5: Compute rewards-to-go $\hat{R}_t$.
6: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
7: Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

8: Use the conjugate gradient algorithm to compute

$$\hat{x}_k \approx \hat{H}_k^{-1} \hat{g}_k,$$

where $\hat{H}_k$ is the Hessian of the sample average KL-divergence.
9: Update the policy by backtracking line search with

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k,$$

where $j \in \{0, 1, 2, ...K\}$ is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraint.
10: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
11: **end for**

15.2 Documentation

You Should Know

Spinning Up currently only has a Tensorflow implementation of TRPO.

spinup.**trpo_tf1**(*env_fn*, *actor_critic=<function mlp_actor_critic>*, *ac_kwargs={}*, *seed=0*, *steps_per_epoch=4000*, *epochs=50*, *gamma=0.99*, *delta=0.01*, *vf_lr=0.001*, *train_v_iters=80*, *damping_coeff=0.1*, *cg_iters=10*, *backtrack_iters=10*, *backtrack_coeff=0.8*, *lam=0.97*, *max_ep_len=1000*, *logger_kwargs={}*, *save_freq=10*, *algo='trpo'*)

Trust Region Policy Optimization

(with support for Natural Policy Gradient)

Parameters

- **env_fn** – A function which creates a copy of the environment. The environment must satisfy the OpenAI Gym API.
- **actor_critic** – A function which takes in placeholder symbols for state, `x_ph`, and action, `a_ph`, and returns the main outputs from the agent's Tensorflow computation graph:

Symbol	Shape	Description
`pi`	(batch, act_dim)	Samples actions from policy given states.
`logp`	(batch,)	Gives log probability, according to the policy, of taking actions `a_ph` in states `x_ph`.
`logp_pi`	(batch,)	Gives log probability, according to the policy, of the action sampled by `pi`.
`info`	N/A	A dict of any intermediate quantities (from calculating the policy or log probabilities) which are needed for analytically computing KL divergence. (eg sufficient statistics of the distributions)
`info_phs`	N/A	A dict of placeholders for old values of the entries in `info`.
`d_kl`	()	A symbol for computing the mean KL divergence between the current policy (`pi`) and the old policy (as specified by the inputs to `info_phs`) over the batch of states given in `x_ph`.
`v`	(batch,)	Gives the value estimate for states in `x_ph`. (Critical: make sure

- **ac_kwargs** (*dict*) – Any kwargs appropriate for the actor_critic function you provided to TRPO.
- **seed** (*int*) – Seed for random number generators.
- **steps_per_epoch** (*int*) – Number of steps of interaction (state-action pairs) for the agent and the environment in each epoch.
- **epochs** (*int*) – Number of epochs of interaction (equivalent to number of policy updates) to perform.
- **gamma** (*float*) – Discount factor. (Always between 0 and 1.)
- **delta** (*float*) – KL-divergence limit for TRPO / NPG update. (Should be small for stability. Values like 0.01, 0.05.)
- **vf_lr** (*float*) – Learning rate for value function optimizer.
- **train_v_iters** (*int*) – Number of gradient descent steps to take on value function per epoch.
- **damping_coeff** (*float*) – Artifact for numerical stability, should be smallish. Adjusts Hessian-vector product calculation:

  $$Hv \rightarrow (\alpha I + H)v$$

  where α is the damping coefficient. Probably don't play with this hyperparameter.
- **cg_iters** (*int*) – Number of iterations of conjugate gradient to perform. Increasing this will lead to a more accurate approximation to $H^{-1}g$, and possibly slightly-improved performance, but at the cost of slowing things down.

  Also probably don't play with this hyperparameter.
- **backtrack_iters** (*int*) – Maximum number of steps allowed in the backtracking line search. Since the line search usually doesn't backtrack, and usually only steps back once when it does, this hyperparameter doesn't often matter.
- **backtrack_coeff** (*float*) – How far back to step during backtracking line search. (Always between 0 and 1, usually above 0.5.)
- **lam** (*float*) – Lambda for GAE-Lambda. (Always between 0 and 1, close to 1.)
- **max_ep_len** (*int*) – Maximum length of trajectory / episode / rollout.
- **logger_kwargs** (*dict*) – Keyword args for EpochLogger.
- **save_freq** (*int*) – How often (in terms of gap between epochs) to save the current policy and value function.
- **algo** – Either 'trpo' or 'npg': this code supports both, since they are almost the same.

15.2.1 Saved Model Contents

The computation graph saved by the logger includes:

Key	Value
`x`	Tensorflow placeholder for state input.
`pi`	Samples an action from the agent, conditioned on states in `x`.
`v`	Gives value estimate for states in `x`.

This saved model can be accessed either by

- running the trained policy with the test_policy.py tool,
- or loading the whole saved graph into a program with restore_tf_graph.

15.3 References

15.3.1 Relevant Papers

- Trust Region Policy Optimization, Schulman et al. 2015
- High Dimensional Continuous Control Using Generalized Advantage Estimation, Schulman et al. 2016
- Approximately Optimal Approximate Reinforcement Learning, Kakade and Langford 2002

15.3.2 Why These Papers?

Schulman 2015 is included because it is the original paper describing TRPO. Schulman 2016 is included because our implementation of TRPO makes use of Generalized Advantage Estimation for computing the policy gradient. Kakade and Langford 2002 is included because it contains theoretical results which motivate and deeply connect to the theoretical foundations of TRPO.

15.3.3 Other Public Implementations

- Baselines
- ModularRL
- rllab

CHAPTER 16

Proximal Policy Optimization

Table of Contents

16.1 Background

(Previously: Background for TRPO)

PPO is motivated by the same question as TRPO: how can we take the biggest possible improvement step on a policy using the data we currently have, without stepping so far that we accidentally cause performance collapse? Where TRPO tries to solve this problem with a complex second-order method, PPO is a family of first-order methods that use a few other tricks to keep new policies close to old. PPO methods are significantly simpler to implement, and empirically seem to perform at least as well as TRPO.

There are two primary variants of PPO: PPO-Penalty and PPO-Clip.

PPO-Penalty approximately solves a KL-constrained update like TRPO, but penalizes the KL-divergence in the objective function instead of making it a hard constraint, and automatically adjusts the penalty coefficient over the course of training so that it's scaled appropriately.

PPO-Clip doesn't have a KL-divergence term in the objective and doesn't have a constraint at all. Instead relies on specialized clipping in the objective function to remove incentives for the new policy to get far from the old policy.

Here, we'll focus only on PPO-Clip (the primary variant used at OpenAI).

16.1.1 Quick Facts

- PPO is an on-policy algorithm.
- PPO can be used for environments with either discrete or continuous action spaces.
- The Spinning Up implementation of PPO supports parallelization with MPI.

16.1.2 Key Equations

PPO-clip updates policies via

$$\theta_{k+1} = \arg\max_{\theta} \underset{s,a \sim \pi_{\theta_k}}{\mathrm{E}} \left[L(s, a, \theta_k, \theta) \right],$$

typically taking multiple steps of (usually minibatch) SGD to maximize the objective. Here L is given by

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \;\; \text{clip} \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1 - \epsilon, 1 + \epsilon \right) A^{\pi_{\theta_k}}(s, a) \right),$$

in which ϵ is a (small) hyperparameter which roughly says how far away the new policy is allowed to go from the old.

This is a pretty complex expression, and it's hard to tell at first glance what it's doing, or how it helps keep the new policy close to the old policy. As it turns out, there's a considerably simplified version[1] of this objective which is a bit easier to grapple with (and is also the version we implement in our code):

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s, a)) \right),$$

where

$$g(\epsilon, A) = \begin{cases} (1 + \epsilon) A & A \geq 0 \\ (1 - \epsilon) A & A < 0. \end{cases}$$

To figure out what intuition to take away from this, let's look at a single state-action pair (s, a), and think of cases.

Advantage is positive: Suppose the advantage for that state-action pair is positive, in which case its contribution to the objective reduces to

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1 + \epsilon) \right) A^{\pi_{\theta_k}}(s, a).$$

[1] See this note for a derivation of the simplified form of the PPO-Clip objective.

Because the advantage is positive, the objective will increase if the action becomes more likely—that is, if $\pi_\theta(a|s)$ increases. But the min in this term puts a limit to how *much* the objective can increase. Once $\pi_\theta(a|s) > (1+\epsilon)\pi_{\theta_k}(a|s)$, the min kicks in and this term hits a ceiling of $(1+\epsilon)A^{\pi_{\theta_k}}(s,a)$. Thus: *the new policy does not benefit by going far away from the old policy*.

Advantage is negative: Suppose the advantage for that state-action pair is negative, in which case its contribution to the objective reduces to

$$L(s,a,\theta_k,\theta) = \max\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1-\epsilon)\right) A^{\pi_{\theta_k}}(s,a).$$

Because the advantage is negative, the objective will increase if the action becomes less likely—that is, if $\pi_\theta(a|s)$ decreases. But the max in this term puts a limit to how *much* the objective can increase. Once $\pi_\theta(a|s) < (1-\epsilon)\pi_{\theta_k}(a|s)$, the max kicks in and this term hits a ceiling of $(1-\epsilon)A^{\pi_{\theta_k}}(s,a)$. Thus, again: *the new policy does not benefit by going far away from the old policy*.

What we have seen so far is that clipping serves as a regularizer by removing incentives for the policy to change dramatically, and the hyperparameter ϵ corresponds to how far away the new policy can go from the old while still profiting the objective.

You Should Know

While this kind of clipping goes a long way towards ensuring reasonable policy updates, it is still possible to end up with a new policy which is too far from the old policy, and there are a bunch of tricks used by different PPO implementations to stave this off. In our implementation here, we use a particularly simple method: early stopping. If the mean KL-divergence of the new policy from the old grows beyond a threshold, we stop taking gradient steps.

When you feel comfortable with the basic math and implementation details, it's worth checking out other implementations to see how they handle this issue!

16.1.3 Exploration vs. Exploitation

PPO trains a stochastic policy in an on-policy way. This means that it explores by sampling actions according to the latest version of its stochastic policy. The amount of randomness in action selection depends on both initial conditions and the training procedure. Over the course of training, the policy typically becomes progressively less random, as the update rule encourages it to exploit rewards that it has already found. This may cause the policy to get trapped in local optima.

16.1.4 Pseudocode

Algorithm 3 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

16.2 Documentation

You Should Know

In what follows, we give documentation for the PyTorch and Tensorflow implementations of PPO in Spinning Up. They have nearly identical function calls and docstrings, except for details relating to model construction. However, we include both full docstrings for completeness.

16.2.1 Documentation: PyTorch Version

`spinup.ppo_pytorch`(*env_fn*, *actor_critic=<MagicMock spec='str' id='140475683269208'>*, *ac_kwargs={}*, *seed=0*, *steps_per_epoch=4000*, *epochs=50*, *gamma=0.99*, *clip_ratio=0.2*, *pi_lr=0.0003*, *vf_lr=0.001*, *train_pi_iters=80*, *train_v_iters=80*, *lam=0.97*, *max_ep_len=1000*, *target_kl=0.01*, *logger_kwargs={}*, *save_freq=10*)

Proximal Policy Optimization (by clipping),

with early stopping based on approximate KL

Parameters

- **`env_fn`** – A function which creates a copy of the environment. The environment must satisfy the OpenAI Gym API.
- **`actor_critic`** – The constructor method for a PyTorch Module with a `step` method, an `act` method, a `pi` module, and a `v` module. The `step` method should accept a batch of observations and return:

Symbol	Shape	Description
`a`	(batch, act_dim)	Numpy array of actions for each observation.
`v`	(batch,)	Numpy array of value estimates for the provided observations.
`logp_a`	(batch,)	Numpy array of log probs for the actions in `a`.

The `act` method behaves the same as `step` but only returns `a`.

The `pi` module's forward call should accept a batch of observations and optionally a batch of actions, and return:

Symbol	Shape	Description
`pi`	N/A	Torch Distribution object, containing a batch of distributions describing the policy for the provided observations.
`logp_a`	(batch,)	Optional (only returned if batch of actions is given). Tensor containing the log probability, according to the policy, of the provided actions. If actions not given, will contain `None`.

The `v` module's forward call should accept a batch of observations and return:

Symbol	Shape	Description
v	(batch,)	Tensor containing the value estimates for the provided observations. (Critical: make sure to flatten this!)

- **ac_kwargs** (*dict*) – Any kwargs appropriate for the ActorCritic object you provided to PPO.
- **seed** (*int*) – Seed for random number generators.
- **steps_per_epoch** (*int*) – Number of steps of interaction (state-action pairs) for the agent and the environment in each epoch.
- **epochs** (*int*) – Number of epochs of interaction (equivalent to number of policy updates) to perform.
- **gamma** (*float*) – Discount factor. (Always between 0 and 1.)
- **clip_ratio** (*float*) – Hyperparameter for clipping in the policy objective. Roughly: how far can the new policy go from the old policy while still profiting (improving the objective function)? The new policy can still go farther than the clip_ratio says, but it doesn't help on the objective anymore. (Usually small, 0.1 to 0.3.) Typically denoted by ϵ.
- **pi_lr** (*float*) – Learning rate for policy optimizer.
- **vf_lr** (*float*) – Learning rate for value function optimizer.
- **train_pi_iters** (*int*) – Maximum number of gradient descent steps to take on policy loss per epoch. (Early stopping may cause optimizer to take fewer than this.)
- **train_v_iters** (*int*) – Number of gradient descent steps to take on value function per epoch.
- **lam** (*float*) – Lambda for GAE-Lambda. (Always between 0 and 1, close to 1.)
- **max_ep_len** (*int*) – Maximum length of trajectory / episode / rollout.
- **target_kl** (*float*) – Roughly what KL divergence we think is appropriate between new and old policies after an update. This will get used for early stopping. (Usually small, 0.01 or 0.05.)
- **logger_kwargs** (*dict*) – Keyword args for EpochLogger.
- **save_freq** (*int*) – How often (in terms of gap between epochs) to save the current policy and value function.

16.2.2 Saved Model Contents: PyTorch Version

The PyTorch saved model can be loaded with `ac = torch.load('path/to/model.pt')`, yielding an actor-critic object (`ac`) that has the properties described in the docstring for `ppo_pytorch`.

You can get actions from this model with

```
actions = ac.act(torch.as_tensor(obs, dtype=torch.float32))
```

16.2.3 Documentation: Tensorflow Version

spinup.**ppo_tf1**(*env_fn*, *actor_critic=<function mlp_actor_critic>*, *ac_kwargs={}*, *seed=0*, *steps_per_epoch=4000*, *epochs=50*, *gamma=0.99*, *clip_ratio=0.2*, *pi_lr=0.0003*, *vf_lr=0.001*, *train_pi_iters=80*, *train_v_iters=80*, *lam=0.97*, *max_ep_len=1000*, *target_kl=0.01*, *logger_kwargs={}*, *save_freq=10*)

Proximal Policy Optimization (by clipping),

with early stopping based on approximate KL

Parameters

- **env_fn** – A function which creates a copy of the environment. The environment must satisfy the OpenAI Gym API.
- **actor_critic** – A function which takes in placeholder symbols for state, `x_ph`, and action, `a_ph`, and returns the main outputs from the agent's Tensorflow computation graph:

Symbol	Shape	Description
`pi`	(batch, act_dim)	Samples actions from policy given states.
`logp`	(batch,)	Gives log probability, according to the policy, of taking actions `a_ph` in states `x_ph`.
`logp_pi`	(batch,)	Gives log probability, according to the policy, of the action sampled by `pi`.
`v`	(batch,)	Gives the value estimate for states in `x_ph`. (Critical: make sure to flatten this!)

- **ac_kwargs** (*dict*) – Any kwargs appropriate for the actor_critic function you provided to PPO.
- **seed** (*int*) – Seed for random number generators.
- **steps_per_epoch** (*int*) – Number of steps of interaction (state-action pairs) for the agent and the environment in each epoch.

- **epochs** (*int*) – Number of epochs of interaction (equivalent to number of policy updates) to perform.
- **gamma** (*float*) – Discount factor. (Always between 0 and 1.)
- **clip_ratio** (*float*) – Hyperparameter for clipping in the policy objective. Roughly: how far can the new policy go from the old policy while still profiting (improving the objective function)? The new policy can still go farther than the clip_ratio says, but it doesn't help on the objective anymore. (Usually small, 0.1 to 0.3.) Typically denoted by ϵ.
- **pi_lr** (*float*) – Learning rate for policy optimizer.
- **vf_lr** (*float*) – Learning rate for value function optimizer.
- **train_pi_iters** (*int*) – Maximum number of gradient descent steps to take on policy loss per epoch. (Early stopping may cause optimizer to take fewer than this.)
- **train_v_iters** (*int*) – Number of gradient descent steps to take on value function per epoch.
- **lam** (*float*) – Lambda for GAE-Lambda. (Always between 0 and 1, close to 1.)
- **max_ep_len** (*int*) – Maximum length of trajectory / episode / rollout.
- **target_kl** (*float*) – Roughly what KL divergence we think is appropriate between new and old policies after an update. This will get used for early stopping. (Usually small, 0.01 or 0.05.)
- **logger_kwargs** (*dict*) – Keyword args for EpochLogger.
- **save_freq** (*int*) – How often (in terms of gap between epochs) to save the current policy and value function.

16.2.4 Saved Model Contents: Tensorflow Version

The computation graph saved by the logger includes:

Key	Value
`x`	Tensorflow placeholder for state input.
`pi`	Samples an action from the agent, conditioned on states in `x`.
`v`	Gives value estimate for states in `x`.

This saved model can be accessed either by

- running the trained policy with the test_policy.py tool,
- or loading the whole saved graph into a program with restore_tf_graph.

16.3 References

16.3.1 Relevant Papers

- Proximal Policy Optimization Algorithms, Schulman et al. 2017
- High Dimensional Continuous Control Using Generalized Advantage Estimation, Schulman et al. 2016
- Emergence of Locomotion Behaviours in Rich Environments, Heess et al. 2017

16.3.2 Why These Papers?

Schulman 2017 is included because it is the original paper describing PPO. Schulman 2016 is included because our implementation of PPO makes use of Generalized Advantage Estimation for computing the policy gradient. Heess 2017 is included because it presents a large-scale empirical analysis of behaviors learned by PPO agents in complex environments (although it uses PPO-penalty instead of PPO-clip).

16.3.3 Other Public Implementations

- Baselines
- ModularRL (Caution: this implements PPO-penalty instead of PPO-clip.)
- rllab (Caution: this implements PPO-penalty instead of PPO-clip.)
- rllib (Ray)

CHAPTER 17

Deep Deterministic Policy Gradient

Table of Contents

17.1 Background

(Previously: Introduction to RL Part 1: The Optimal Q-Function and the Optimal Action)

Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy.

This approach is closely connected to Q-learning, and is motivated the same way: if you know the optimal action-value function $Q^*(s,a)$, then in any given state, the optimal action $a^*(s)$ can be found by solving

$$a^*(s) = \arg\max_a Q^*(s,a).$$

DDPG interleaves learning an approximator to $Q^*(s,a)$ with learning an approximator to $a^*(s)$, and it does so in a way which is specifically adapted for environments with continuous action spaces. But what does it mean that DDPG is adapted *specifically* for environments with continuous action spaces? It relates to how we compute the max over actions in $\max_a Q^*(s,a)$.

When there are a finite number of discrete actions, the max poses no problem, because we can just compute the Q-values for each action separately and directly compare them. (This also immediately gives us the action which maximizes the Q-value.) But when the action space is continuous, we can't exhaustively evaluate the space, and solving the optimization problem is highly non-trivial. Using a normal optimization algorithm would make calculating $\max_a Q^*(s,a)$ a painfully expensive subroutine. And since it would need to be run every time the agent wants to take an action in the environment, this is unacceptable.

Because the action space is continuous, the function $Q^*(s,a)$ is presumed to be differentiable with respect to the action argument. This allows us to set up an efficient, gradient-based learning rule for a policy $\mu(s)$ which exploits that fact. Then, instead of running an expensive optimization subroutine each time we wish to compute $\max_a Q(s,a)$, we can approximate it with $\max_a Q(s,a) \approx Q(s,\mu(s))$. See the Key Equations section details.

17.1.1 Quick Facts

- DDPG is an off-policy algorithm.
- DDPG can only be used for environments with continuous action spaces.
- DDPG can be thought of as being deep Q-learning for continuous action spaces.
- The Spinning Up implementation of DDPG does not support parallelization.

17.1.2 Key Equations

Here, we'll explain the math behind the two parts of DDPG: learning a Q function, and learning a policy.

The Q-Learning Side of DDPG

First, let's recap the Bellman equation describing the optimal action-value function, $Q^*(s,a)$. It's given by

$$Q^*(s,a) = \underset{s' \sim P}{\mathrm{E}}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a')\right]$$

where $s' \sim P$ is shorthand for saying that the next state, s', is sampled by the environment from a distribution $P(\cdot|s,a)$.

This Bellman equation is the starting point for learning an approximator to $Q^*(s,a)$. Suppose the approximator is a neural network $Q_\phi(s,a)$, with parameters ϕ, and that we have collected a set $\mathcal{D}$ of transitions (s,a,r,s',d) (where d

indicates whether state s' is terminal). We can set up a **mean-squared Bellman error (MSBE)** function, which tells us roughly how closely Q_ϕ comes to satisfying the Bellman equation:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}}\left[\left(Q_\phi(s,a) - \Big(r + \gamma(1-d)\max_{a'} Q_\phi(s',a')\Big)\right)^2\right]$$

Here, in evaluating $(1-d)$, we've used a Python convention of evaluating `True` to 1 and `False` to zero. Thus, when `d==True`—which is to say, when s' is a terminal state—the Q-function should show that the agent gets no additional rewards after the current state. (This choice of notation corresponds to what we later implement in code.)

Q-learning algorithms for function approximators, such as DQN (and all its variants) and DDPG, are largely based on minimizing this MSBE loss function. There are two main tricks employed by all of them which are worth describing, and then a specific detail for DDPG.

Trick One: Replay Buffers. All standard algorithms for training a deep neural network to approximate $Q^*(s,a)$ make use of an experience replay buffer. This is the set $\mathcal{D}$ of previous experiences. In order for the algorithm to have stable behavior, the replay buffer should be large enough to contain a wide range of experiences, but it may not always be good to keep everything. If you only use the very-most recent data, you will overfit to that and things will break; if you use too much experience, you may slow down your learning. This may take some tuning to get right.

You Should Know

We've mentioned that DDPG is an off-policy algorithm: this is as good a point as any to highlight why and how. Observe that the replay buffer *should* contain old experiences, even though they might have been obtained using an outdated policy. Why are we able to use these at all? The reason is that the Bellman equation *doesn't care* which transition tuples are used, or how the actions were selected, or what happens after a given transition, because the optimal Q-function should satisfy the Bellman equation for *all* possible transitions. So any transitions that we've ever experienced are fair game when trying to fit a Q-function approximator via MSBE minimization.

Trick Two: Target Networks. Q-learning algorithms make use of **target networks**. The term

$$r + \gamma(1-d)\max_{a'} Q_\phi(s',a')$$

is called the **target**, because when we minimize the MSBE loss, we are trying to make the Q-function be more like this target. Problematically, the target depends on the same parameters we are trying to train: ϕ. This makes MSBE minimization unstable. The solution is to use a set of parameters which comes close to ϕ, but with a time delay—that is to say, a second network, called the target network, which lags the first. The parameters of the target network are denoted ϕ_{targ}.

In DQN-based algorithms, the target network is just copied over from the main network every some-fixed-number of steps. In DDPG-style algorithms, the target network is updated once per main network update by polyak averaging:

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi,$$

where ρ is a hyperparameter between 0 and 1 (usually close to 1). (This hyperparameter is called `polyak` in our code).

DDPG Detail: Calculating the Max Over Actions in the Target. As mentioned earlier: computing the maximum over actions in the target is a challenge in continuous action spaces. DDPG deals with this by using a **target policy network** to compute an action which approximately maximizes $Q_{\phi_{\text{targ}}}$. The target policy network is found the same way as the target Q-function: by polyak averaging the policy parameters over the course of training.

Putting it all together, Q-learning in DDPG is performed by minimizing the following MSBE loss with stochastic gradient descent:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}}\left[\left(Q_\phi(s,a) - \Big(r + \gamma(1-d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))\Big)\right)^2\right],$$

where $\mu_{\theta_{\text{targ}}}$ is the target policy.

The Policy Learning Side of DDPG

Policy learning in DDPG is fairly simple. We want to learn a deterministic policy $\mu_\theta(s)$ which gives the action that maximizes $Q_\phi(s,a)$. Because the action space is continuous, and we assume the Q-function is differentiable with respect to action, we can just perform gradient ascent (with respect to policy parameters only) to solve

$$\max_\theta \underset{s\sim\mathcal{D}}{\mathrm{E}}\left[Q_\phi(s,\mu_\theta(s))\right].$$

Note that the Q-function parameters are treated as constants here.

17.1.3 Exploration vs. Exploitation

DDPG trains a deterministic policy in an off-policy way. Because the policy is deterministic, if the agent were to explore on-policy, in the beginning it would probably not try a wide enough variety of actions to find useful learning signals. To make DDPG policies explore better, we add noise to their actions at training time. The authors of the original DDPG paper recommended time-correlated OU noise, but more recent results suggest that uncorrelated, mean-zero Gaussian noise works perfectly well. Since the latter is simpler, it is preferred. To facilitate getting higher-quality training data, you may reduce the scale of the noise over the course of training. (We do not do this in our implementation, and keep noise scale fixed throughout.)

At test time, to see how well the policy exploits what it has learned, we do not add noise to the actions.

You Should Know

Our DDPG implementation uses a trick to improve exploration at the start of training. For a fixed number of steps at the beginning (set with the `start_steps` keyword argument), the agent takes actions which are sampled from a uniform random distribution over valid actions. After that, it returns to normal DDPG exploration.

17.1.4 Pseudocode

Algorithm 4 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

    end for
  end if
until convergence

17.2 Documentation

You Should Know

In what follows, we give documentation for the PyTorch and Tensorflow implementations of DDPG in Spinning Up. They have nearly identical function calls and docstrings, except for details relating to model construction. However, we include both full docstrings for completeness.

17.2.1 Documentation: PyTorch Version

spinup.**ddpg_pytorch**(*env_fn*, *actor_critic=<MagicMock spec='str' id='140475680855376'>*, *ac_kwargs={}*, *seed=0*, *steps_per_epoch=4000*, *epochs=100*, *replay_size=1000000*, *gamma=0.99*, *polyak=0.995*, *pi_lr=0.001*, *q_lr=0.001*, *batch_size=100*, *start_steps=10000*, *update_after=1000*, *update_every=50*, *act_noise=0.1*, *num_test_episodes=10*, *max_ep_len=1000*, *logger_kwargs={}*, *save_freq=1*)

Deep Deterministic Policy Gradient (DDPG)

Parameters

- **env_fn** – A function which creates a copy of the environment. The environment must satisfy the OpenAI Gym API.
- **actor_critic** – The constructor method for a PyTorch Module with an `act` method, a `pi` module, and a `q` module. The `act` method and `pi` module should accept batches of observations as inputs, and `q` should accept a batch of observations and a batch of actions as inputs. When called, these should return:

Call	Output Shape	Description
`act`	(batch, act_dim)	Numpy array of actions for each observation.
`pi`	(batch, act_dim)	Tensor containing actions from policy given observations.
`q`	(batch,)	Tensor containing the current estimate of Q* for the provided observations and actions. (Critical: make sure to flatten this!)

- **ac_kwargs** (*dict*) – Any kwargs appropriate for the ActorCritic object you provided to DDPG.
- **seed** (*int*) – Seed for random number generators.
- **steps_per_epoch** (*int*) – Number of steps of interaction (state-action pairs) for the agent and the environment in each epoch.
- **epochs** (*int*) – Number of epochs to run and train agent.
- **replay_size** (*int*) – Maximum length of replay buffer.
- **gamma** (*float*) – Discount factor. (Always between 0 and 1.)
- **polyak** (*float*) – Interpolation factor in polyak averaging for target networks. Target

networks are updated towards main networks according to:

$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta$$

where ρ is polyak. (Always between 0 and 1, usually close to 1.)

- **pi_lr** (*float*) – Learning rate for policy.
- **q_lr** (*float*) – Learning rate for Q-networks.
- **batch_size** (*int*) – Minibatch size for SGD.
- **start_steps** (*int*) – Number of steps for uniform-random action selection, before running real policy. Helps exploration.
- **update_after** (*int*) – Number of env interactions to collect before starting to do gradient descent updates. Ensures replay buffer is full enough for useful updates.
- **update_every** (*int*) – Number of env interactions that should elapse between gradient descent updates. Note: Regardless of how long you wait between updates, the ratio of env steps to gradient steps is locked to 1.
- **act_noise** (*float*) – Stddev for Gaussian exploration noise added to policy at training time. (At test time, no noise is added.)
- **num_test_episodes** (*int*) – Number of episodes to test the deterministic policy at the end of each epoch.
- **max_ep_len** (*int*) – Maximum length of trajectory / episode / rollout.
- **logger_kwargs** (*dict*) – Keyword args for EpochLogger.
- **save_freq** (*int*) – How often (in terms of gap between epochs) to save the current policy and value function.

17.2.2 Saved Model Contents: PyTorch Version

The PyTorch saved model can be loaded with `ac = torch.load('path/to/model.pt')`, yielding an actor-critic object (`ac`) that has the properties described in the docstring for `ddpg_pytorch`.

You can get actions from this model with

```
actions = ac.act(torch.as_tensor(obs, dtype=torch.float32))
```

17.2.3 Documentation: Tensorflow Version

spinup.**ddpg_tf1**(*env_fn*, *actor_critic=<function mlp_actor_critic>*, *ac_kwargs={}*, *seed=0*, *steps_per_epoch=4000*, *epochs=100*, *replay_size=1000000*, *gamma=0.99*, *polyak=0.995*, *pi_lr=0.001*, *q_lr=0.001*, *batch_size=100*, *start_steps=10000*, *update_after=1000*, *update_every=50*, *act_noise=0.1*, *num_test_episodes=10*, *max_ep_len=1000*, *logger_kwargs={}*, *save_freq=1*)

Deep Deterministic Policy Gradient (DDPG)

Parameters

- **env_fn** – A function which creates a copy of the environment. The environment must satisfy the OpenAI Gym API.

- **actor_critic** – A function which takes in placeholder symbols for state, `x_ph`, and action, `a_ph`, and returns the main outputs from the agent's Tensorflow computation graph:

Symbol	Shape	Description
`pi`	(batch, act_dim)	Deterministically computes actions from policy given states.
`q`	(batch,)	Gives the current estimate of Q* for states in `x_ph` and actions in `a_ph`.
`q_pi`	(batch,)	Gives the composition of `q` and `pi` for states in `x_ph`: q(x, pi(x)).

- **ac_kwargs** (*dict*) – Any kwargs appropriate for the actor_critic function you provided to DDPG.
- **seed** (*int*) – Seed for random number generators.
- **steps_per_epoch** (*int*) – Number of steps of interaction (state-action pairs) for the agent and the environment in each epoch.
- **epochs** (*int*) – Number of epochs to run and train agent.
- **replay_size** (*int*) – Maximum length of replay buffer.
- **gamma** (*float*) – Discount factor. (Always between 0 and 1.)
- **polyak** (*float*) – Interpolation factor in polyak averaging for target networks. Target networks are updated towards main networks according to:

$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1-\rho)\theta$$

  where ρ is polyak. (Always between 0 and 1, usually close to 1.)
- **pi_lr** (*float*) – Learning rate for policy.
- **q_lr** (*float*) – Learning rate for Q-networks.
- **batch_size** (*int*) – Minibatch size for SGD.
- **start_steps** (*int*) – Number of steps for uniform-random action selection, before running real policy. Helps exploration.
- **update_after** (*int*) – Number of env interactions to collect before starting to do gradient descent updates. Ensures replay buffer is full enough for useful updates.
- **update_every** (*int*) – Number of env interactions that should elapse between gradient descent updates. Note: Regardless of how long you wait between updates, the ratio of env steps to gradient steps is locked to 1.

- **act_noise** (*float*) – Stddev for Gaussian exploration noise added to policy at training time. (At test time, no noise is added.)
- **num_test_episodes** (*int*) – Number of episodes to test the deterministic policy at the end of each epoch.
- **max_ep_len** (*int*) – Maximum length of trajectory / episode / rollout.
- **logger_kwargs** (*dict*) – Keyword args for EpochLogger.
- **save_freq** (*int*) – How often (in terms of gap between epochs) to save the current policy and value function.

17.2.4 Saved Model Contents: Tensorflow Version

The computation graph saved by the logger includes:

Key	Value
`x`	Tensorflow placeholder for state input.
`a`	Tensorflow placeholder for action input.
`pi`	Deterministically computes an action from the agent, conditioned on states in `x`.
`q`	Gives action-value estimate for states in `x` and actions in `a`.

This saved model can be accessed either by

- running the trained policy with the test_policy.py tool,
- or loading the whole saved graph into a program with restore_tf_graph.

17.3 References

17.3.1 Relevant Papers

- Deterministic Policy Gradient Algorithms, Silver et al. 2014
- Continuous Control With Deep Reinforcement Learning, Lillicrap et al. 2016

17.3.2 Why These Papers?

Silver 2014 is included because it establishes the theory underlying deterministic policy gradients (DPG). Lillicrap 2016 is included because it adapts the theoretically-grounded DPG algorithm to the deep RL setting, giving DDPG.

17.3.3 Other Public Implementations

- Baselines
- rllab
- rllib (Ray)

- TD3 release repo

CHAPTER 18

Twin Delayed DDPG

Table of Contents

18.1 Background

(Previously: Background for DDPG)

While DDPG can achieve great performance sometimes, it is frequently brittle with respect to hyperparameters and other kinds of tuning. A common failure mode for DDPG is that the learned Q-function begins to dramatically

overestimate Q-values, which then leads to the policy breaking, because it exploits the errors in the Q-function. Twin Delayed DDPG (TD3) is an algorithm that addresses this issue by introducing three critical tricks:

Trick One: Clipped Double-Q Learning. TD3 learns *two* Q-functions instead of one (hence "twin"), and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.

Trick Two: "Delayed" Policy Updates. TD3 updates the policy (and target networks) less frequently than the Q-function. The paper recommends one policy update for every two Q-function updates.

Trick Three: Target Policy Smoothing. TD3 adds noise to the target action, to make it harder for the policy to exploit Q-function errors by smoothing out Q along changes in action.

Together, these three tricks result in substantially improved performance over baseline DDPG.

18.1.1 Quick Facts

- TD3 is an off-policy algorithm.
- TD3 can only be used for environments with continuous action spaces.
- The Spinning Up implementation of TD3 does not support parallelization.

18.1.2 Key Equations

TD3 concurrently learns two Q-functions, Q_{ϕ_1} and Q_{ϕ_2}, by mean square Bellman error minimization, in almost the same way that DDPG learns its single Q-function. To show exactly how TD3 does this and how it differs from normal DDPG, we'll work from the innermost part of the loss function outwards.

First: **target policy smoothing**. Actions used to form the Q-learning target are based on the target policy, $\mu_{\theta_{\text{targ}}}$, but with clipped noise added on each dimension of the action. After adding the clipped noise, the target action is then clipped to lie in the valid action range (all valid actions, a, satisfy $a_{Low} \leq a \leq a_{High}$). The target actions are thus:

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \qquad \epsilon \sim \mathcal{N}(0, \sigma)$$

Target policy smoothing essentially serves as a regularizer for the algorithm. It addresses a particular failure mode that can happen in DDPG: if the Q-function approximator develops an incorrect sharp peak for some actions, the policy will quickly exploit that peak and then have brittle or incorrect behavior. This can be averted by smoothing out the Q-function over similar actions, which target policy smoothing is designed to do.

Next: **clipped double-Q learning**. Both Q-functions use a single target, calculated using whichever of the two Q-functions gives a smaller target value:

$$y(r, s', d) = r + \gamma(1-d) \min_{i=1,2} Q_{\phi_{i,\text{targ}}}(s', a'(s')),$$

and then both are learned by regressing to this target:

$$L(\phi_1, \mathcal{D}) = \mathop{\text{E}}_{(s,a,r,s',d)\sim\mathcal{D}}\left[\left(Q_{\phi_1}(s,a) - y(r,s',d)\right)^2\right],$$

$$L(\phi_2, \mathcal{D}) = \mathop{\text{E}}_{(s,a,r,s',d)\sim\mathcal{D}}\left[\left(Q_{\phi_2}(s,a) - y(r,s',d)\right)^2\right].$$

Using the smaller Q-value for the target, and regressing towards that, helps fend off overestimation in the Q-function.

Lastly: the policy is learned just by maximizing Q_{ϕ_1}:

$$\max_{\theta} \mathop{\text{E}}_{s\sim\mathcal{D}}\left[Q_{\phi_1}(s, \mu_\theta(s))\right],$$

which is pretty much unchanged from DDPG. However, in TD3, the policy is updated less frequently than the Q-functions are. This helps damp the volatility that normally arises in DDPG because of how a policy update changes the target.

18.1.3 Exploration vs. Exploitation

TD3 trains a deterministic policy in an off-policy way. Because the policy is deterministic, if the agent were to explore on-policy, in the beginning it would probably not try a wide enough variety of actions to find useful learning signals. To make TD3 policies explore better, we add noise to their actions at training time, typically uncorrelated mean-zero Gaussian noise. To facilitate getting higher-quality training data, you may reduce the scale of the noise over the course of training. (We do not do this in our implementation, and keep noise scale fixed throughout.)

At test time, to see how well the policy exploits what it has learned, we do not add noise to the actions.

You Should Know

Our TD3 implementation uses a trick to improve exploration at the start of training. For a fixed number of steps at the beginning (set with the `start_steps` keyword argument), the agent takes actions which are sampled from a uniform random distribution over valid actions. After that, it returns to normal TD3 exploration.

18.1.4 Pseudocode

Algorithm 5 Twin Delayed DDPG

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute target actions

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \qquad \epsilon \sim \mathcal{N}(0, \sigma)$$

      Compute targets

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', a'(s'))$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      if $j \mod \texttt{policy\_delay} = 0$ **then**
        Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_{\phi_1}(s, \mu_\theta(s))$$

        Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$
$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$$

      end if
    end for
  end if
until convergence

18.2 Documentation

You Should Know

In what follows, we give documentation for the PyTorch and Tensorflow implementations of TD3 in Spinning Up. They have nearly identical function calls and docstrings, except for details relating to model construction. However,

we include both full docstrings for completeness.

18.2.1 Documentation: PyTorch Version

spinup.**td3_pytorch**(*env_fn*, *actor_critic=<MagicMock spec='str' id='140475680326936'>*, *ac_kwargs={}*, *seed=0*, *steps_per_epoch=4000*, *epochs=100*, *replay_size=1000000*, *gamma=0.99*, *polyak=0.995*, *pi_lr=0.001*, *q_lr=0.001*, *batch_size=100*, *start_steps=10000*, *update_after=1000*, *update_every=50*, *act_noise=0.1*, *target_noise=0.2*, *noise_clip=0.5*, *policy_delay=2*, *num_test_episodes=10*, *max_ep_len=1000*, *logger_kwargs={}*, *save_freq=1*)

Twin Delayed Deep Deterministic Policy Gradient (TD3)

Parameters

- **env_fn** – A function which creates a copy of the environment. The environment must satisfy the OpenAI Gym API.
- **actor_critic** – The constructor method for a PyTorch Module with an `act` method, a `pi` module, a `q1` module, and a `q2` module. The `act` method and `pi` module should accept batches of observations as inputs, and `q1` and `q2` should accept a batch of observations and a batch of actions as inputs. When called, these should return:

Call	Output Shape	Description
`act`	(batch, act_dim)	Numpy array of actions for each observation.
`pi`	(batch, act_dim)	Tensor containing actions from policy given observations.
`q1`	(batch,)	Tensor containing one current estimate of Q* for the provided observations and actions. (Critical: make sure to flatten this!)
`q2`	(batch,)	Tensor containing the other current estimate of Q* for the provided observations and actions. (Critical: make sure to flatten this!)

- **ac_kwargs** (*dict*) – Any kwargs appropriate for the ActorCritic object you provided to TD3.
- **seed** (*int*) – Seed for random number generators.
- **steps_per_epoch** (*int*) – Number of steps of interaction (state-action pairs) for the agent and the environment in each epoch.
- **epochs** (*int*) – Number of epochs to run and train agent.
- **replay_size** (*int*) – Maximum length of replay buffer.
- **gamma** (*float*) – Discount factor. (Always between 0 and 1.)
- **polyak** (*float*) – Interpolation factor in polyak averaging for target networks. Target networks are updated towards main networks according to:

$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1-\rho)\theta$$

  where ρ is polyak. (Always between 0 and 1, usually close to 1.)
- **pi_lr** (*float*) – Learning rate for policy.
- **q_lr** (*float*) – Learning rate for Q-networks.
- **batch_size** (*int*) – Minibatch size for SGD.
- **start_steps** (*int*) – Number of steps for uniform-random action selection, before running real policy. Helps exploration.
- **update_after** (*int*) – Number of env interactions to collect before starting to do gradient descent updates. Ensures replay buffer is full enough for useful updates.
- **update_every** (*int*) – Number of env interactions that should elapse between gradient descent updates. Note: Regardless of how long you wait between updates, the ratio of env steps to gradient steps is locked to 1.
- **act_noise** (*float*) – Stddev for Gaussian exploration noise added to policy at training time. (At test time, no noise is added.)
- **target_noise** (*float*) – Stddev for smoothing noise added to target policy.
- **noise_clip** (*float*) – Limit for absolute value of target policy smoothing noise.
- **policy_delay** (*int*) – Policy will only be updated once every policy_delay times for each update of the Q-networks.
- **num_test_episodes** (*int*) – Number of episodes to test the deterministic policy at the end of each epoch.
- **max_ep_len** (*int*) – Maximum length of trajectory / episode / rollout.
- **logger_kwargs** (*dict*) – Keyword args for EpochLogger.
- **save_freq** (*int*) – How often (in terms of gap between epochs) to save the current policy and value function.

18.2.2 Saved Model Contents: PyTorch Version

The PyTorch saved model can be loaded with `ac = torch.load('path/to/model.pt')`, yielding an actor-critic object (`ac`) that has the properties described in the docstring for `td3_pytorch`.

You can get actions from this model with

```
actions = ac.act(torch.as_tensor(obs, dtype=torch.float32))
```

18.2.3 Documentation: Tensorflow Version

spinup.**td3_tf1**(*env_fn*, *actor_critic=<function mlp_actor_critic>*, *ac_kwargs={}*, *seed=0*, *steps_per_epoch=4000*, *epochs=100*, *replay_size=1000000*, *gamma=0.99*, *polyak=0.995*, *pi_lr=0.001*, *q_lr=0.001*, *batch_size=100*, *start_steps=10000*, *update_after=1000*, *update_every=50*, *act_noise=0.1*, *target_noise=0.2*, *noise_clip=0.5*, *policy_delay=2*, *num_test_episodes=10*, *max_ep_len=1000*, *logger_kwargs={}*, *save_freq=1*)

Twin Delayed Deep Deterministic Policy Gradient (TD3)

Parameters

- **env_fn** – A function which creates a copy of the environment. The environment must satisfy the OpenAI Gym API.
- **actor_critic** – A function which takes in placeholder symbols for state, `x_ph`, and action, `a_ph`, and returns the main outputs from the agent's Tensorflow computation graph:

Symbol	Shape	Description
`pi`	(batch, act_dim)	Deterministically computes actions from policy given states.
`q1`	(batch,)	Gives one estimate of Q* for states in `x_ph` and actions in `a_ph`.
`q2`	(batch,)	Gives another estimate of Q* for states in `x_ph` and actions in `a_ph`.
`q1_pi`	(batch,)	Gives the composition of `q1` and `pi` for states in `x_ph`: q1(x, pi(x)).

- **ac_kwargs** (*dict*) – Any kwargs appropriate for the actor_critic function you provided to TD3.
- **seed** (*int*) – Seed for random number generators.

- **steps_per_epoch** (*int*) – Number of steps of interaction (state-action pairs) for the agent and the environment in each epoch.
- **epochs** (*int*) – Number of epochs to run and train agent.
- **replay_size** (*int*) – Maximum length of replay buffer.
- **gamma** (*float*) – Discount factor. (Always between 0 and 1.)
- **polyak** (*float*) – Interpolation factor in polyak averaging for target networks. Target networks are updated towards main networks according to:

$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta$$

  where ρ is polyak. (Always between 0 and 1, usually close to 1.)
- **pi_lr** (*float*) – Learning rate for policy.
- **q_lr** (*float*) – Learning rate for Q-networks.
- **batch_size** (*int*) – Minibatch size for SGD.
- **start_steps** (*int*) – Number of steps for uniform-random action selection, before running real policy. Helps exploration.
- **update_after** (*int*) – Number of env interactions to collect before starting to do gradient descent updates. Ensures replay buffer is full enough for useful updates.
- **update_every** (*int*) – Number of env interactions that should elapse between gradient descent updates. Note: Regardless of how long you wait between updates, the ratio of env steps to gradient steps is locked to 1.
- **act_noise** (*float*) – Stddev for Gaussian exploration noise added to policy at training time. (At test time, no noise is added.)
- **target_noise** (*float*) – Stddev for smoothing noise added to target policy.
- **noise_clip** (*float*) – Limit for absolute value of target policy smoothing noise.
- **policy_delay** (*int*) – Policy will only be updated once every policy_delay times for each update of the Q-networks.
- **num_test_episodes** (*int*) – Number of episodes to test the deterministic policy at the end of each epoch.
- **max_ep_len** (*int*) – Maximum length of trajectory / episode / rollout.
- **logger_kwargs** (*dict*) – Keyword args for EpochLogger.
- **save_freq** (*int*) – How often (in terms of gap between epochs) to save the current policy and value function.

18.2.4 Saved Model Contents: Tensorflow Version

The computation graph saved by the logger includes:

Key	Value
`x`	Tensorflow placeholder for state input.
`a`	Tensorflow placeholder for action input.
`pi`	Deterministically computes an action from the agent, conditioned on states in `x`.
`q1`	Gives one action-value estimate for states in `x` and actions in `a`.
`q2`	Gives the other action-value estimate for states in `x` and actions in `a`.

This saved model can be accessed either by

- running the trained policy with the test_policy.py tool,
- or loading the whole saved graph into a program with restore_tf_graph.

18.3 References

18.3.1 Relevant Papers

- Addressing Function Approximation Error in Actor-Critic Methods, Fujimoto et al, 2018

18.3.2 Other Public Implementations

- TD3 release repo

CHAPTER 19

Soft Actor-Critic

Table of Contents

19.1 Background

(Previously: Background for TD3)

Soft Actor Critic (SAC) is an algorithm that optimizes a stochastic policy in an off-policy way, forming a bridge between stochastic policy optimization and DDPG-style approaches. It isn't a direct successor to TD3 (having been published roughly concurrently), but it incorporates the clipped double-Q trick, and due to the inherent stochasticity of the policy in SAC, it also winds up benefiting from something like target policy smoothing.

A central feature of SAC is **entropy regularization.** The policy is trained to maximize a trade-off between expected return and entropy, a measure of randomness in the policy. This has a close connection to the exploration-exploitation trade-off: increasing entropy results in more exploration, which can accelerate learning later on. It can also prevent the policy from prematurely converging to a bad local optimum.

19.1.1 Quick Facts

- SAC is an off-policy algorithm.
- The version of SAC implemented here can only be used for environments with continuous action spaces.
- An alternate version of SAC, which slightly changes the policy update rule, can be implemented to handle discrete action spaces.
- The Spinning Up implementation of SAC does not support parallelization.

19.1.2 Key Equations

To explain Soft Actor Critic, we first have to introduce the entropy-regularized reinforcement learning setting. In entropy-regularized RL, there are slightly-different equations for value functions.

Entropy-Regularized Reinforcement Learning

Entropy is a quantity which, roughly speaking, says how random a random variable is. If a coin is weighted so that it almost always comes up heads, it has low entropy; if it's evenly weighted and has a half chance of either outcome, it has high entropy.

Let x be a random variable with probability mass or density function P. The entropy H of x is computed from its distribution P according to

$$H(P) = \underset{x\sim P}{\mathrm{E}}\left[-\log P(x)\right].$$

In entropy-regularized reinforcement learning, the agent gets a bonus reward at each time step proportional to the entropy of the policy at that timestep. This changes the RL problem to:

$$\pi^* = \arg\max_{\pi} \underset{\tau\sim\pi}{\mathrm{E}}\left[\sum_{t=0}^{\infty} \gamma^t \Big(R(s_t, a_t, s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right) \Big)\right],$$

where $\alpha > 0$ is the trade-off coefficient. (Note: we're assuming an infinite-horizon discounted setting here, and we'll do the same for the rest of this page.) We can now define the slightly-different value functions in this setting. V^π is changed to include the entropy bonuses from every timestep:

$$V^\pi(s) = \underset{\tau\sim\pi}{\mathrm{E}}\left[\sum_{t=0}^{\infty} \gamma^t \Big(R(s_t, a_t, s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right) \Big) \,\middle|\, s_0 = s\right]$$

Q^π is changed to include the entropy bonuses from every timestep *except the first*:

$$Q^\pi(s,a) = \underset{\tau\sim\pi}{\mathrm{E}}\left[\sum_{t=0}^{\infty}\gamma^t R(s_t,a_t,s_{t+1}) + \alpha\sum_{t=1}^{\infty}\gamma^t H\left(\pi(\cdot|s_t)\right)\middle| s_0 = s, a_0 = a\right]$$

With these definitions, V^π and Q^π are connected by:

$$V^\pi(s) = \underset{a\sim\pi}{\mathrm{E}}\left[Q^\pi(s,a)\right] + \alpha H\left(\pi(\cdot|s)\right)$$

and the Bellman equation for Q^π is

$$\begin{aligned} Q^\pi(s,a) &= \underset{\substack{s'\sim P\\ a'\sim\pi}}{\mathrm{E}}\left[R(s,a,s') + \gamma\left(Q^\pi(s',a') + \alpha H\left(\pi(\cdot|s')\right)\right)\right] \\ &= \underset{s'\sim P}{\mathrm{E}}\left[R(s,a,s') + \gamma V^\pi(s')\right]. \end{aligned}$$

You Should Know

The way we've set up the value functions in the entropy-regularized setting is a little bit arbitrary, and actually we could have done it differently (eg make Q^π include the entropy bonus at the first timestep). The choice of definition may vary slightly across papers on the subject.

Soft Actor-Critic

SAC concurrently learns a policy π_θ and two Q-functions Q_{ϕ_1}, Q_{ϕ_2}. There are two variants of SAC that are currently standard: one that uses a fixed entropy regularization coefficient α, and another that enforces an entropy constraint by varying α over the course of training. For simplicity, Spinning Up makes use of the version with a fixed entropy regularization coefficient, but the entropy-constrained variant is generally preferred by practitioners.

You Should Know

The SAC algorithm has changed a little bit over time. An older version of SAC also learns a value function V_ψ in addition to the Q-functions; this page will focus on the modern version that omits the extra value function.

Learning Q. The Q-functions are learned in a similar way to TD3, but with a few key differences.

First, what's similar?

1. Like in TD3, both Q-functions are learned with MSBE minimization, by regressing to a single shared target.
2. Like in TD3, the shared target is computed using target Q-networks, and the target Q-networks are obtained by polyak averaging the Q-network parameters over the course of training.
3. Like in TD3, the shared target makes use of the **clipped double-Q** trick.

What's different?

1. Unlike in TD3, the target also includes a term that comes from SAC's use of entropy regularization.
2. Unlike in TD3, the next-state actions used in the target come from the **current policy** instead of a target policy.
3. Unlike in TD3, there is no explicit target policy smoothing. TD3 trains a deterministic policy, and so it accomplishes smoothing by adding random noise to the next-state actions. SAC trains a stochastic policy, and so the noise from that stochasticity is sufficient to get a similar effect.

Before we give the final form of the Q-loss, let's take a moment to discuss how the contribution from entropy regularization comes in. We'll start by taking our recursive Bellman equation for the entropy-regularized Q^π from earlier, and rewriting it a little bit by using the definition of entropy:

$$\begin{aligned} Q^\pi(s,a) &= \mathop{\mathrm{E}}_{\substack{s'\sim P\\ a'\sim\pi}}\left[R(s,a,s') + \gamma\left(Q^\pi(s',a') + \alpha H\left(\pi(\cdot|s')\right)\right)\right] \\ &= \mathop{\mathrm{E}}_{\substack{s'\sim P\\ a'\sim\pi}}\left[R(s,a,s') + \gamma\left(Q^\pi(s',a') - \alpha \log \pi(a'|s')\right)\right] \end{aligned}$$

The RHS is an expectation over next states (which come from the replay buffer) and next actions (which come from the current policy, and **not** the replay buffer). Since it's an expectation, we can approximate it with samples:

$$Q^\pi(s,a) \approx r + \gamma\left(Q^\pi(s',\tilde{a}') - \alpha \log \pi(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi(\cdot|s').$$

You Should Know

We switch next action notation to $\tilde{a}'$, instead of a', to highlight that the next actions have to be sampled fresh from the policy (whereas by contrast, r and s' should come from the replay buffer).

SAC sets up the MSBE loss for each Q-function using this kind of sample approximation for the target. The only thing still undetermined here is which Q-function gets used to compute the sample backup: like TD3, SAC uses the clipped double-Q trick, and takes the minimum Q-value between the two Q approximators.

Putting it all together, the loss functions for the Q-networks in SAC are:

$$L(\phi_i, \mathcal{D}) = \mathop{\mathrm{E}}_{(s,a,r,s',d)\sim\mathcal{D}}\left[\left(Q_{\phi_i}(s,a) - y(r,s',d)\right)^2\right],$$

where the target is given by

$$y(r,s',d) = r + \gamma(1-d)\left(\min_{j=1,2} Q_{\phi_{\text{targ},j}}(s',\tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s').$$

Learning the Policy. The policy should, in each state, act to maximize the expected future return plus expected future entropy. That is, it should maximize $V^\pi(s)$, which we expand out into

$$\begin{aligned} V^\pi(s) &= \mathop{\mathrm{E}}_{a\sim\pi}\left[Q^\pi(s,a)\right] + \alpha H\left(\pi(\cdot|s)\right) \\ &= \mathop{\mathrm{E}}_{a\sim\pi}\left[Q^\pi(s,a) - \alpha \log \pi(a|s)\right]. \end{aligned}$$

The way we optimize the policy makes use of the **reparameterization trick**, in which a sample from $\pi_\theta(\cdot|s)$ is drawn by computing a deterministic function of state, policy parameters, and independent noise. To illustrate: following the authors of the SAC paper, we use a squashed Gaussian policy, which means that samples are obtained according to

$$\tilde{a}_\theta(s,\xi) = \tanh\left(\mu_\theta(s) + \sigma_\theta(s) \odot \xi\right), \quad \xi \sim \mathcal{N}(0, I).$$

You Should Know

This policy has two key differences from the policies we use in the other policy optimization algorithms:

1. The squashing function. The $\tanh$ in the SAC policy ensures that actions are bounded to a finite range. This is absent in the VPG, TRPO, and PPO policies. It also changes the distribution: before the $\tanh$ the SAC policy is a factored Gaussian like the other algorithms' policies, but after the $\tanh$ it is not. (You can still compute the log-probabilities of actions in closed form, though: see the paper appendix for details.)

2. The way standard deviations are parameterized. In VPG, TRPO, and PPO, we represent the log std devs with state-independent parameter vectors. In SAC, we represent the log std devs as outputs from the neural network, meaning that they depend on state in a complex way. SAC with state-independent log std devs, in our experience, did not work. (Can you think of why? Or better yet: run an experiment to verify?)

The reparameterization trick allows us to rewrite the expectation over actions (which contains a pain point: the distribution depends on the policy parameters) into an expectation over noise (which removes the pain point: the distribution now has no dependence on parameters):

$$\mathop{\mathrm{E}}_{a\sim\pi_\theta}[Q^{\pi_\theta}(s,a) - \alpha \log \pi_\theta(a|s)] = \mathop{\mathrm{E}}_{\xi\sim\mathcal{N}}[Q^{\pi_\theta}(s,\tilde{a}_\theta(s,\xi)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s,\xi)|s)]$$

To get the policy loss, the final step is that we need to substitute Q^{π_θ} with one of our function approximators. Unlike in TD3, which uses Q_{ϕ_1} (just the first Q approximator), SAC uses $\min_{j=1,2} Q_{\phi_j}$ (the minimum of the two Q approximators). The policy is thus optimized according to

$$\max_\theta \mathop{\mathrm{E}}_{\substack{s\sim\mathcal{D}\\ \xi\sim\mathcal{N}}}\left[\min_{j=1,2} Q_{\phi_j}(s,\tilde{a}_\theta(s,\xi)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s,\xi)|s)\right],$$

which is almost the same as the DDPG and TD3 policy optimization, except for the min-double-Q trick, the stochasticity, and the entropy term.

19.1.3 Exploration vs. Exploitation

SAC trains a stochastic policy with entropy regularization, and explores in an on-policy way. The entropy regularization coefficient α explicitly controls the explore-exploit tradeoff, with higher α corresponding to more exploration, and lower α corresponding to more exploitation. The right coefficient (the one which leads to the stablest / highest-reward learning) may vary from environment to environment, and could require careful tuning.

At test time, to see how well the policy exploits what it has learned, we remove stochasticity and use the mean action instead of a sample from the distribution. This tends to improve performance over the original stochastic policy.

You Should Know

Our SAC implementation uses a trick to improve exploration at the start of training. For a fixed number of steps at the beginning (set with the `start_steps` keyword argument), the agent takes actions which are sampled from a uniform random distribution over valid actions. After that, it returns to normal SAC exploration.

19.1.4 Pseudocode

Algorithm 6 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \qquad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \Big(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta \left(\tilde{a}_\theta(s)| \, s\right) \Big),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
      Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$

    end for
  end if
until convergence

19.2 Documentation

You Should Know

In what follows, we give documentation for the PyTorch and Tensorflow implementations of SAC in Spinning Up. They have nearly identical function calls and docstrings, except for details relating to model construction. However, we include both full docstrings for completeness.

19.2.1 Documentation: PyTorch Version

spinup.**sac_pytorch**(*env_fn*, *actor_critic=<MagicMock spec='str' id='140475680567136'>*, *ac_kwargs={}*, *seed=0*, *steps_per_epoch=4000*, *epochs=100*, *replay_size=1000000*, *gamma=0.99*, *polyak=0.995*, *lr=0.001*, *alpha=0.2*, *batch_size=100*, *start_steps=10000*, *update_after=1000*, *update_every=50*, *num_test_episodes=10*, *max_ep_len=1000*, *logger_kwargs={}*, *save_freq=1*)

Soft Actor-Critic (SAC)

Parameters

- **env_fn** – A function which creates a copy of the environment. The environment must satisfy the OpenAI Gym API.
- **actor_critic** – The constructor method for a PyTorch Module with an `act` method, a `pi` module, a `q1` module, and a `q2` module. The `act` method and `pi` module should accept batches of observations as inputs, and `q1` and `q2` should accept a batch of observations and a batch of actions as inputs. When called, `act`, `q1`, and `q2` should return:

Call	Output Shape	Description
`act`	(batch, act_dim)	Numpy array of actions for each observation.
`q1`	(batch,)	Tensor containing one current estimate of Q* for the provided observations and actions. (Critical: make sure to flatten this!)
`q2`	(batch,)	Tensor containing the other current estimate of Q* for the provided observations and actions. (Critical: make sure to flatten this!)

Calling `pi` should return:

Symbol	Shape	Description
`a`	(batch, act_dim)	Tensor containing actions from policy given observations.
`logp_pi`	(batch,)	Tensor containing log probabilities of actions in `a`. Importantly: gradients should be able to flow back into `a`.

- **ac_kwargs** (*dict*) – Any kwargs appropriate for the ActorCritic object you provided to SAC.
- **seed** (*int*) – Seed for random number generators.
- **steps_per_epoch** (*int*) – Number of steps of interaction (state-action pairs) for the agent and the environment in each epoch.
- **epochs** (*int*) – Number of epochs to run and train agent.
- **replay_size** (*int*) – Maximum length of replay buffer.
- **gamma** (*float*) – Discount factor. (Always between 0 and 1.)
- **polyak** (*float*) – Interpolation factor in polyak averaging for target networks. Target networks are updated towards main networks according to:

  $$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1-\rho)\theta$$

  where ρ is polyak. (Always between 0 and 1, usually close to 1.)
- **lr** (*float*) – Learning rate (used for both policy and value learning).
- **alpha** (*float*) – Entropy regularization coefficient. (Equivalent to inverse of reward scale in the original SAC paper.)
- **batch_size** (*int*) – Minibatch size for SGD.
- **start_steps** (*int*) – Number of steps for uniform-random action selection, before running real policy. Helps exploration.
- **update_after** (*int*) – Number of env interactions to collect before starting to do gradient descent updates. Ensures replay buffer is full enough for useful updates.
- **update_every** (*int*) – Number of env interactions that should elapse between gradient descent updates. Note: Regardless of how long you wait between updates, the ratio of env steps to gradient steps is locked to 1.
- **num_test_episodes** (*int*) – Number of episodes to test the deterministic policy at the end of each epoch.
- **max_ep_len** (*int*) – Maximum length of trajectory / episode / rollout.
- **logger_kwargs** (*dict*) – Keyword args for EpochLogger.
- **save_freq** (*int*) – How often (in terms of gap between epochs) to save the current policy and value function.

19.2.2 Saved Model Contents: PyTorch Version

The PyTorch saved model can be loaded with `ac = torch.load('path/to/model.pt')`, yielding an actor-critic object (`ac`) that has the properties described in the docstring for `sac_pytorch`.

You can get actions from this model with

```
actions = ac.act(torch.as_tensor(obs, dtype=torch.float32))
```

19.2.3 Documentation: Tensorflow Version

spinup.**sac_tf1**(*env_fn*, *actor_critic=<function mlp_actor_critic>*, *ac_kwargs={}*, *seed=0*, *steps_per_epoch=4000*, *epochs=100*, *replay_size=1000000*, *gamma=0.99*, *polyak=0.995*, *lr=0.001*, *alpha=0.2*, *batch_size=100*, *start_steps=10000*, *update_after=1000*, *update_every=50*, *num_test_episodes=10*, *max_ep_len=1000*, *logger_kwargs={}*, *save_freq=1*)

Soft Actor-Critic (SAC)

Parameters

- **env_fn** – A function which creates a copy of the environment. The environment must satisfy the OpenAI Gym API.
- **actor_critic** – A function which takes in placeholder symbols for state, `x_ph`, and action, `a_ph`, and returns the main outputs from the agent's Tensorflow computation graph:

Symbol	Shape	Description
`mu`	(batch, act_dim)	Computes mean actions from policy given states.
`pi`	(batch, act_dim)	Samples actions from policy given states.
`logp_pi`	(batch,)	Gives log probability, according to the policy, of the action sampled by `pi`. Critical: must be differentiable with respect to policy parameters all the way through action sampling.
`q1`	(batch,)	Gives one estimate of Q* for states in `x_ph` and actions in `a_ph`.
`q2`	(batch,)	Gives another estimate of Q* for states in `x_ph` and actions in `a_ph`.

- **ac_kwargs** (*dict*) – Any kwargs appropriate for the actor_critic function you provided to SAC.
- **seed** (*int*) – Seed for random number generators.
- **steps_per_epoch** (*int*) – Number of steps of interaction (state-action pairs) for the agent and the environment in each epoch.
- **epochs** (*int*) – Number of epochs to run and train agent.
- **replay_size** (*int*) – Maximum length of replay buffer.
- **gamma** (*float*) – Discount factor. (Always between 0 and 1.)
- **polyak** (*float*) – Interpolation factor in polyak averaging for target networks. Target

networks are updated towards main networks according to:

$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta$$

where ρ is polyak. (Always between 0 and 1, usually close to 1.)

- **lr** (*float*) – Learning rate (used for both policy and value learning).
- **alpha** (*float*) – Entropy regularization coefficient. (Equivalent to inverse of reward scale in the original SAC paper.)
- **batch_size** (*int*) – Minibatch size for SGD.
- **start_steps** (*int*) – Number of steps for uniform-random action selection, before running real policy. Helps exploration.
- **update_after** (*int*) – Number of env interactions to collect before starting to do gradient descent updates. Ensures replay buffer is full enough for useful updates.
- **update_every** (*int*) – Number of env interactions that should elapse between gradient descent updates. Note: Regardless of how long you wait between updates, the ratio of env steps to gradient steps is locked to 1.
- **num_test_episodes** (*int*) – Number of episodes to test the deterministic policy at the end of each epoch.
- **max_ep_len** (*int*) – Maximum length of trajectory / episode / rollout.
- **logger_kwargs** (*dict*) – Keyword args for EpochLogger.
- **save_freq** (*int*) – How often (in terms of gap between epochs) to save the current policy and value function.

19.2.4 Saved Model Contents: Tensorflow Version

The computation graph saved by the logger includes:

Key	Value
`x`	Tensorflow placeholder for state input.
`a`	Tensorflow placeholder for action input.
`mu`	Deterministically computes mean action from the agent, given states in `x`.
`pi`	Samples an action from the agent, conditioned on states in `x`.
`q1`	Gives one action-value estimate for states in `x` and actions in `a`.
`q2`	Gives the other action-value estimate for states in `x` and actions in `a`.
`v`	Gives the value estimate for states in `x`.

This saved model can be accessed either by

- running the trained policy with the test_policy.py tool,
- or loading the whole saved graph into a program with restore_tf_graph.

Note: for SAC, the correct evaluation policy is given by `mu` and not by `pi`. The policy `pi` may be thought of as the exploration policy, while `mu` is the exploitation policy.

19.3 References

19.3.1 Relevant Papers

- Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor, Haarnoja et al, 2018
- Soft Actor-Critic Algorithms and Applications, Haarnoja et al, 2018
- Learning to Walk via Deep Reinforcement Learning, Haarnoja et al, 2018

19.3.2 Other Public Implementations

- SAC release repo (original "official" codebase)
- Softlearning repo (current "official" codebase)
- Yarats and Kostrikov repo

CHAPTER 20

Logger

Table of Contents

20.1 Using a Logger

Spinning Up ships with basic logging tools, implemented in the classes Logger and EpochLogger. The Logger class contains most of the basic functionality for saving diagnostics, hyperparameter configurations, the state of a training run, and the trained model. The EpochLogger class adds a thin layer on top of that to make it easy to track the average, standard deviation, min, and max value of a diagnostic over each epoch and across MPI workers.

You Should Know

All Spinning Up algorithm implementations use an EpochLogger.

20.1.1 Examples

First, let's look at a simple example of how an EpochLogger keeps track of a diagnostic value:

```
>>> from spinup.utils.logx import EpochLogger
>>> epoch_logger = EpochLogger()
>>> for i in range(10):
        epoch_logger.store(Test=i)
>>> epoch_logger.log_tabular('Test', with_min_and_max=True)
>>> epoch_logger.dump_tabular()
---------------------------------------
|     AverageTest |             4.5 |
|         StdTest |            2.87 |
|         MaxTest |               9 |
|         MinTest |               0 |
---------------------------------------
```

The `store` method is used to save all values of `Test` to the `epoch_logger`'s internal state. Then, when `log_tabular` is called, it computes the average, standard deviation, min, and max of `Test` over all of the values in the internal state. The internal state is wiped clean after the call to `log_tabular` (to prevent leakage into the statistics at the next epoch). Finally, `dump_tabular` is called to write the diagnostics to file and to stdout.

Next, let's look at a full training procedure with the logger embedded, to highlight configuration and model saving as well as diagnostic logging:

```
import numpy as np
import tensorflow as tf
import time
from spinup.utils.logx import EpochLogger


def mlp(x, hidden_sizes=(32,), activation=tf.tanh, output_activation=None):
    for h in hidden_sizes[:-1]:
        x = tf.layers.dense(x, units=h, activation=activation)
    return tf.layers.dense(x, units=hidden_sizes[-1], activation=output_activation)


# Simple script for training an MLP on MNIST.
def train_mnist(steps_per_epoch=100, epochs=5,
                lr=1e-3, layers=2, hidden_size=64,
                logger_kwargs=dict(), save_freq=1):

    logger = EpochLogger(**logger_kwargs)
    logger.save_config(locals())

    # Load and preprocess MNIST data
    (x_train, y_train), _ = tf.keras.datasets.mnist.load_data()
    x_train = x_train.reshape(-1, 28*28) / 255.0

    # Define inputs & main outputs from computation graph
    x_ph = tf.placeholder(tf.float32, shape=(None, 28*28))
    y_ph = tf.placeholder(tf.int32, shape=(None,))
    logits = mlp(x_ph, hidden_sizes=[hidden_size]*layers + [10], activation=tf.nn.
↪relu)
    predict = tf.argmax(logits, axis=1, output_type=tf.int32)

    # Define loss function, accuracy, and training op
    y = tf.one_hot(y_ph, 10)
```

```
33      loss = tf.losses.softmax_cross_entropy(y, logits)
34      acc = tf.reduce_mean(tf.cast(tf.equal(y_ph, predict), tf.float32))
35      train_op = tf.train.AdamOptimizer().minimize(loss)
36
37      # Prepare session
38      sess = tf.Session()
39      sess.run(tf.global_variables_initializer())
40
41      # Setup model saving
42      logger.setup_tf_saver(sess, inputs={'x': x_ph},
43                                  outputs={'logits': logits, 'predict': predict})
44
45      start_time = time.time()
46
47      # Run main training loop
48      for epoch in range(epochs):
49          for t in range(steps_per_epoch):
50              idxs = np.random.randint(0, len(x_train), 32)
51              feed_dict = {x_ph: x_train[idxs],
52                           y_ph: y_train[idxs]}
53              outs = sess.run([loss, acc, train_op], feed_dict=feed_dict)
54              logger.store(Loss=outs[0], Acc=outs[1])
55
56          # Save model
57          if (epoch % save_freq == 0) or (epoch == epochs-1):
58              logger.save_state(state_dict=dict(), itr=None)
59
60          # Log info about epoch
61          logger.log_tabular('Epoch', epoch)
62          logger.log_tabular('Acc', with_min_and_max=True)
63          logger.log_tabular('Loss', average_only=True)
64          logger.log_tabular('TotalGradientSteps', (epoch+1)*steps_per_epoch)
65          logger.log_tabular('Time', time.time()-start_time)
66          logger.dump_tabular()
67
68  if __name__ == '__main__':
69      train_mnist()
```

In this example, observe that

- On line 19, logger.save_config is used to save the hyperparameter configuration to a JSON file.
- On lines 42 and 43, logger.setup_tf_saver is used to prepare the logger to save the key elements of the computation graph.
- On line 54, diagnostics are saved to the logger's internal state via logger.store.
- On line 58, the computation graph is saved once per epoch via logger.save_state.
- On lines 61-66, logger.log_tabular and logger.dump_tabular are used to write the epoch diagnostics to file. Note that the keys passed into logger.log_tabular are the same as the keys passed into logger.store.

20.1.2 Logging and PyTorch

The preceding example was given in Tensorflow. For PyTorch, everything is the same except for L42-43: instead of `logger.setup_tf_saver`, you would use `logger.setup_pytorch_saver`, and you would pass it a PyTorch module (the network you are training) as an argument.

The behavior of `logger.save_state` is the same as in the Tensorflow case: each time it is called, it'll save the latest version of the PyTorch module.

20.1.3 Logging and MPI

You Should Know

Several algorithms in RL are easily parallelized by using MPI to average gradients and/or other key quantities. The Spinning Up loggers are designed to be well-behaved when using MPI: things will only get written to stdout and to file from the process with rank 0. But information from other processes isn't lost if you use the EpochLogger: everything which is passed into EpochLogger via `store`, regardless of which process it's stored in, gets used to compute average/std/min/max values for a diagnostic.

20.2 Logger Classes

class `spinup.utils.logx.`**Logger**(*output_dir=None*, *output_fname='progress.txt'*, *exp_name=None*)

A general-purpose logger.

Makes it easy to save diagnostics, hyperparameter configurations, the state of a training run, and the trained model.

__init__(*output_dir=None*, *output_fname='progress.txt'*, *exp_name=None*)

Initialize a Logger.

Parameters

- **output_dir** (*string*) – A directory for saving results to. If `None`, defaults to a temp directory of the form `/tmp/experiments/somerandomnumber`.
- **output_fname** (*string*) – Name for the tab-separated-value file containing metrics logged throughout a training run. Defaults to `progress.txt`.
- **exp_name** (*string*) – Experiment name. If you run multiple training runs and give them all the same `exp_name`, the plotter will know to group them. (Use case: if you run the same hyperparameter configuration with multiple random seeds, you should give them all the same `exp_name`.)

dump_tabular()

Write all of the diagnostics from the current iteration.

Writes both to stdout, and to the output file.

log(*msg*, *color='green'*)

Print a colorized message to stdout.

log_tabular(*key*, *val*)

Log a value of some diagnostic.

Call this only once for each diagnostic quantity, each iteration. After using `log_tabular` to store values for each diagnostic, make sure to call `dump_tabular` to write them out to file and stdout (otherwise they will not get saved anywhere).

save_config(*config*)

Log an experiment configuration.

Call this once at the top of your experiment, passing in all important config vars as a dict. This will serialize the config to JSON, while handling anything which can't be serialized in a graceful way (writing as informative a string as possible).

Example use:

```
logger = EpochLogger(**logger_kwargs)
logger.save_config(locals())
```

save_state(*state_dict*, *itr=None*)
Saves the state of an experiment.

To be clear: this is about saving *state*, not logging diagnostics. All diagnostic logging is separate from this function. This function will save whatever is in `state_dict`—usually just a copy of the environment—and the most recent parameters for the model you previously set up saving for with `setup_tf_saver`.

Call with any frequency you prefer. If you only want to maintain a single state and overwrite it at each call with the most recent version, leave `itr=None`. If you want to keep all of the states you save, provide unique (increasing) values for 'itr'.

Parameters

- **state_dict** (*dict*) – Dictionary containing essential elements to describe the current state of training.
- **itr** – An int, or None. Current iteration of training.

setup_pytorch_saver(*what_to_save*)
Set up easy model saving for a single PyTorch model.

Because PyTorch saving and loading is especially painless, this is very minimal; we just need references to whatever we would like to pickle. This is integrated into the logger because the logger knows where the user would like to save information about this training run.

Parameters **what_to_save** – Any PyTorch model or serializable object containing PyTorch models.

setup_tf_saver(*sess*, *inputs*, *outputs*)
Set up easy model saving for tensorflow.

Call once, after defining your computation graph but before training.

Parameters

- **sess** – The Tensorflow session in which you train your computation graph.
- **inputs** (*dict*) – A dictionary that maps from keys of your choice to the tensorflow placeholders that serve as inputs to the computation graph. Make sure that *all* of the placeholders needed for your outputs are included!
- **outputs** (*dict*) – A dictionary that maps from keys of your choice to the outputs from your computation graph.

class spinup.utils.logx.**EpochLogger**(**args*, ***kwargs*)
Bases: *spinup.utils.logx.Logger*

A variant of Logger tailored for tracking average values over epochs.

Typical use case: there is some quantity which is calculated many times throughout an epoch, and at the end of the epoch, you would like to report the average / std / min / max value of that quantity.

With an EpochLogger, each time the quantity is calculated, you would use

```
epoch_logger.store(NameOfQuantity=quantity_value)
```

to load it into the EpochLogger's state. Then at the end of the epoch, you would use

```
epoch_logger.log_tabular(NameOfQuantity, **options)
```

to record the desired values.

get_stats(*key*)
Lets an algorithm ask the logger for mean/std/min/max of a diagnostic.

log_tabular(*key*, *val=None*, *with_min_and_max=False*, *average_only=False*)
Log a value or possibly the mean/std/min/max values of a diagnostic.

Parameters

- **key** (*string*) – The name of the diagnostic. If you are logging a diagnostic whose state has previously been saved with `store`, the key here has to match the key you used there.
- **val** – A value for the diagnostic. If you have previously saved values for this key via `store`, do *not* provide a `val` here.
- **with_min_and_max** (*bool*) – If true, log min and max values of the diagnostic over the epoch.
- **average_only** (*bool*) – If true, do not log the standard deviation of the diagnostic over the epoch.

store(***kwargs*)
Save something into the epoch_logger's current state.

Provide an arbitrary number of keyword arguments with numerical values.

20.3 Loading Saved Models (PyTorch Only)

To load an actor-critic model saved by a PyTorch Spinning Up implementation, run:

```
ac = torch.load('path/to/model.pt')
```

When you use this method to load an actor-critic model, you can minimally expect it to have an `act` method that allows you to sample actions from the policy, given observations:

```
actions = ac.act(torch.as_tensor(obs, dtype=torch.float32))
```

20.4 Loading Saved Graphs (Tensorflow Only)

spinup.utils.logx.**restore_tf_graph**(*sess*, *fpath*)
Loads graphs saved by Logger.

Will output a dictionary whose keys and values are from the 'inputs' and 'outputs' dict you specified with logger.setup_tf_saver().

Parameters

- **sess** – A Tensorflow session.
- **fpath** – Filepath to save directory.

Returns A dictionary mapping from keys to tensors in the computation graph loaded from `fpath`.

When you use this method to restore a graph saved by a Tensorflow Spinning Up implementation, you can minimally expect it to include the following:

Key	Value
`x`	Tensorflow placeholder for state input.
`pi`	Samples an action from the agent, conditioned on states in `x`.

The relevant value functions for an algorithm are also typically stored. For details of what else gets saved by a given algorithm, see its documentation page.

CHAPTER 21

Plotter

See the page on plotting results for documentation of the plotter.

CHAPTER 22

MPI Tools

Table of Contents

22.1 Core MPI Utilities

spinup.utils.mpi_tools.**mpi_avg**(*x*)
Average a scalar or vector over MPI processes.

spinup.utils.mpi_tools.**mpi_fork**(*n*, *bind_to_core=False*)
Re-launches the current script with workers linked by MPI.

Also, terminates the original process that launched it.

Taken almost without modification from the Baselines function of the same name.

Parameters

- **n** (*int*) – Number of process to split into.
- **bind_to_core** (*bool*) – Bind each MPI process to a core.

spinup.utils.mpi_tools.**mpi_statistics_scalar**(*x*, *with_min_and_max=False*)
Get mean/std and optional min/max of scalar x across MPI processes.

Parameters

- **x** – An array containing samples of the scalar to produce statistics for.

- **with_min_and_max** (*bool*) – If true, return min and max of x in addition to mean and std.

spinup.utils.mpi_tools.**num_procs**()
Count active MPI processes.

spinup.utils.mpi_tools.**proc_id**()
Get rank of calling process.

22.2 MPI + PyTorch Utilities

spinup.utils.mpi_pytorch contains a few tools to make it easy to do data-parallel PyTorch optimization across MPI processes. The two main ingredients are syncing parameters and averaging gradients before they are used by the adaptive optimizer. Also there's a hacky fix for a problem where the PyTorch instance in each separate process tries to get too many threads, and they start to clobber each other.

The pattern for using these tools looks something like this:

1. At the beginning of the training script, call `setup_pytorch_for_mpi()`. (Avoids clobbering problem.)
2. After you've constructed a PyTorch module, call `sync_params(module)`.
3. Then, during gradient descent, call `mpi_avg_grads` after the backward pass, like so:

```
optimizer.zero_grad()
loss = compute_loss(module)
loss.backward()
mpi_avg_grads(module)    # averages gradient buffers across MPI processes!
optimizer.step()
```

spinup.utils.mpi_pytorch.**mpi_avg_grads**(*module*)
Average contents of gradient buffers across MPI processes.

spinup.utils.mpi_pytorch.**setup_pytorch_for_mpi**()
Avoid slowdowns caused by each separate process's PyTorch using more than its fair share of CPU resources.

spinup.utils.mpi_pytorch.**sync_params**(*module*)
Sync all parameters of module across all MPI processes.

22.3 MPI + Tensorflow Utilities

The spinup.utils.mpi_tf contains a a few tools to make it easy to use the AdamOptimizer across many MPI processes. This is a bit hacky—if you're looking for something more sophisticated and general-purpose, consider horovod.

class spinup.utils.mpi_tf.**MpiAdamOptimizer**(***kwargs*)
Adam optimizer that averages gradients across MPI processes.

The compute_gradients method is taken from Baselines MpiAdamOptimizer. For documentation on method arguments, see the Tensorflow docs page for the base AdamOptimizer.

apply_gradients(*grads_and_vars*, *global_step=None*, *name=None*)
Same as normal apply_gradients, except sync params after update.

compute_gradients(*loss*, *var_list*, ***kwargs*)
Same as normal compute_gradients, except average grads over processes.

spinup.utils.mpi_tf.**sync_all_params**()
Sync all tf variables across MPI processes.

CHAPTER 23

Run Utils

Table of Contents

23.1 ExperimentGrid

Spinning Up ships with a tool called ExperimentGrid for making hyperparameter ablations easier. This is based on (but simpler than) the rllab tool called VariantGenerator.

class `spinup.utils.run_utils.`**`ExperimentGrid`**(*name=''*)

Tool for running many experiments given hyperparameter ranges.

`add`(*key*, *vals*, *shorthand=None*, *in_name=False*)

Add a parameter (key) to the grid config, with potential values (vals).

By default, if a shorthand isn't given, one is automatically generated from the key using the first three letters of each colon-separated term. To disable this behavior, change `DEFAULT_SHORTHAND` in the `spinup/user_config.py` file to `False`.

Parameters

- **`key`** (*`string`*) – Name of parameter.
- **`vals`** (*`value or list of values`*) – Allowed values of parameter.
- **`shorthand`** (*`string`*) – Optional, shortened name of parameter. For example, maybe the parameter `steps_per_epoch` is shortened to `steps`.
- **`in_name`** (*`bool`*) – When constructing variant names, force the inclusion of this parameter into the name.

print()
Print a helpful report about the experiment grid.

run(*thunk*, *num_cpu=1*, *data_dir=None*, *datestamp=False*)
Run each variant in the grid with function 'thunk'.

Note: 'thunk' must be either a callable function, or a string. If it is a string, it must be the name of a parameter whose values are all callable functions.

Uses `call_experiment` to actually launch each experiment, and gives each variant a name using `self.variant_name()`.

Maintenance note: the args for ExperimentGrid.run should track closely to the args for call_experiment. However, `seed` is omitted because we presume the user may add it as a parameter in the grid.

variant_name(*variant*)
Given a variant (dict of valid param/value pairs), make an exp_name.

A variant's name is constructed as the grid name (if you've given it one), plus param names (or shorthands if available) and values separated by underscores.

Note: if `seed` is a parameter, it is not included in the name.

variants()
Makes a list of dicts, where each dict is a valid config in the grid.

There is special handling for variant parameters whose names take the form

```
'full:param:name'.
```

The colons are taken to indicate that these parameters should have a nested dict structure. eg, if there are two params,

Key	Val
`'base:param:a'`	1
`'base:param:b'`	2

the variant dict will have the structure

```
variant = {
    base: {
        param : {
            a : 1,
            b : 2
            }
        }
    }
```

23.2 Calling Experiments

spinup.utils.run_utils.**call_experiment**(*exp_name*, *thunk*, *seed=0*, *num_cpu=1*, *data_dir=None*, *datestamp=False*, ***kwargs*)
Run a function (thunk) with hyperparameters (kwargs), plus configuration.

This wraps a few pieces of functionality which are useful when you want to run many experiments in sequence, including logger configuration and splitting into multiple processes for MPI.

There's also a SpinningUp-specific convenience added into executing the thunk: if `env_name` is one of the kwargs passed to call_experiment, it's assumed that the thunk accepts an argument called `env_fn`, and that the `env_fn` should make a gym environment with the given `env_name`.

The way the experiment is actually executed is slightly complicated: the function is serialized to a string, and then `run_entrypoint.py` is executed in a subprocess call with the serialized string as an argument. `run_entrypoint.py` unserializes the function call and executes it. We choose to do it this way—instead of just calling the function directly here—to avoid leaking state between successive experiments.

Parameters

- **exp_name** (*string*) – Name for experiment.
- **thunk** (*callable*) – A python function.
- **seed** (*int*) – Seed for random number generators.
- **num_cpu** (*int*) – Number of MPI processes to split into. Also accepts 'auto', which will set up as many procs as there are cpus on the machine.
- **data_dir** (*string*) – Used in configuring the logger, to decide where to store experiment results. Note: if left as None, data_dir will default to `DEFAULT_DATA_DIR` from `spinup/user_config.py`.
- ****kwargs** – All kwargs to pass to thunk.

spinup.utils.run_utils.**setup_logger_kwargs**(*exp_name*, *seed=None*, *data_dir=None*, *datestamp=False*)

Sets up the output_dir for a logger and returns a dict for logger kwargs.

If no seed is given and datestamp is false,

```
output_dir = data_dir/exp_name
```

If a seed is given and datestamp is false,

```
output_dir = data_dir/exp_name/exp_name_s[seed]
```

If datestamp is true, amend to

```
output_dir = data_dir/YY-MM-DD_exp_name/YY-MM-DD_HH-MM-SS_exp_name_s[seed]
```

You can force datestamp=True by setting `FORCE_DATESTAMP=True` in `spinup/user_config.py`.

Parameters

- **exp_name** (*string*) – Name for experiment.
- **seed** (*int*) – Seed for random number generators used by experiment.
- **data_dir** (*string*) – Path to folder where results should be saved. Default is the `DEFAULT_DATA_DIR` in `spinup/user_config.py`.
- **datestamp** (*bool*) – Whether to include a date and timestamp in the name of the save directory.

Returns logger_kwargs, a dict containing output_dir and exp_name.

CHAPTER 24

Acknowledgements

We gratefully acknowledge the contributions of the many people who helped get this project off of the ground, including people who beta tested the software, gave feedback on the material, improved dependencies of Spinning Up code in service of this release, or otherwise supported the project. Given the number of people who were involved at various points, this list of names may not be exhaustive. (If you think you should have been listed here, please do not hesitate to reach out.)

In no particular order, thank you Alex Ray, Amanda Askell, Ben Garfinkel, Christy Dennison, Coline Devin, Daniel Zeigler, Dylan Hadfield-Menell, Ge Yang, Greg Khan, Jack Clark, Jonas Rothfuss, Larissa Schiavo, Leandro Castelao, Lilian Weng, Maddie Hall, Matthias Plappert, Miles Brundage, Peter Zokhov, and Pieter Abbeel.

We are also grateful to Pieter Abbeel's group at Berkeley, and the Center for Human-Compatible AI, for giving feedback on presentations about Spinning Up.

CHAPTER 25

About the Author

Spinning Up in Deep RL was primarily developed by Josh Achiam, a research scientist on the OpenAI Safety Team and PhD student at UC Berkeley advised by Pieter Abbeel. Josh studies topics related to safety in deep reinforcement learning, and has previously published work on safe exploration.

CHAPTER 26

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

Symbols

A

C

D