OpenAI Spinning Up

latest

Search docs

USER DOCUMENTATION
Introduction
Installation
Algorithms
Running Experiments
Experiment Outputs
Plotting Results

INTRODUCTION TO RL
Part 1: Key Concepts in RL
Part 2: Kinds of RL Algorithms
Part 3: Intro to Policy Optimization

RESOURCES
Spinning Up as a Deep RL Researcher
Key Papers in Deep RL
Exercises
Benchmarks for Spinning Up Implementations

ALGORITHMS DOCS
⊟ Vanilla Policy Gradient
⊞ Background
⊞ Documentation
⊞ References
Trust Region Policy Optimization
Proximal Policy Optimization
Deep Deterministic Policy Gradient
Twin Delayed DDPG
Soft Actor-Critic

UTILITIES DOCS
Logger
Plotter
MPI Tools
Run Utils

ETC.
Acknowledgements
About the Author

Read the Docs v: latest

Vanilla Policy Gradient

Table of Contents

- Vanilla Policy Gradient
  - Background
    - Quick Facts
    - Key Equations
    - Exploration vs. Exploitation
    - Pseudocode
  - Documentation
    - Documentation: PyTorch Version
    - Saved Model Contents: PyTorch Version
    - Documentation: Tensorflow Version
    - Saved Model Contents: Tensorflow Version
  - References
    - Relevant Papers
    - Why These Papers?
    - Other Public Implementations

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.

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.

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.

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.

Pseudocode

OpenAI Spinning Up

latest

USER DOCUMENTATION

Introduction

Installation

Algorithms

Running Experiments

Experiment Outputs

Plotting Results

INTRODUCTION TO RL

Part 1: Key Concepts in RL

Part 2: Kinds of RL Algorithms

Part 3: Intro to Policy Optimization

RESOURCES

Spinning Up as a Deep RL Researcher

Key Papers in Deep RL

Exercises

Benchmarks for Spinning Up Implementations

ALGORITHMS DOCS

⊟ Vanilla Policy Gradient

⊞ Background

⊞ Documentation

⊞ References

Trust Region Policy Optimization

Proximal Policy Optimization

Deep Deterministic Policy Gradient

Twin Delayed DDPG

Soft Actor-Critic

UTILITIES DOCS

Logger

Plotter

MPI Tools

Run Utils

ETC.

Acknowledgements

About the Author

Read the Docs v: latest

Algorithm 1 Vanilla Policy Gradient Algorithm

```
1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: for $k = 0, 1, 2, ...$ 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 $V_{\phi_k}$.
6:     Estimate policy gradient as
           $$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left. \nabla_\theta \log \pi_\theta(a_t|s_t)\right|_{\theta_k} \hat{A}_t.$$
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
```

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.

Documentation: PyTorch Version

`spinup.vpg_pytorch`*(env_fn, actor_critic=<MagicMock spec='str' id='140554319865336'>, 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

OpenAI Spinning Up
latest

USER DOCUMENTATION
Introduction
Installation
Algorithms
Running Experiments
Experiment Outputs
Plotting Results

INTRODUCTION TO RL
Part 1: Key Concepts in RL
Part 2: Kinds of RL Algorithms
Part 3: Intro to Policy Optimization

RESOURCES
Spinning Up as a Deep RL Researcher
Key Papers in Deep RL
Exercises
Benchmarks for Spinning Up Implementations

ALGORITHMS DOCS
Vanilla Policy Gradient
Background
Documentation
References
Trust Region Policy Optimization
Proximal Policy Optimization
Deep Deterministic Policy Gradient
Twin Delayed DDPG
Soft Actor-Critic

UTILITIES DOCS
Logger
Plotter
MPI Tools
Run Utils

ETC.
Acknowledgements
About the Author

Read the Docs v: latest

`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.

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))
```

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.)

OpenAI Spinning Up

latest

USER DOCUMENTATION

Introduction
Installation
Algorithms
Running Experiments
Experiment Outputs
Plotting Results

INTRODUCTION TO RL

Part 1: Key Concepts in RL
Part 2: Kinds of RL Algorithms
Part 3: Intro to Policy Optimization

RESOURCES

Spinning Up as a Deep RL Researcher
Key Papers in Deep RL
Exercises
Benchmarks for Spinning Up Implementations

ALGORITHMS DOCS

⊟ Vanilla Policy Gradient
⊞ Background
⊞ Documentation
⊞ References
Trust Region Policy Optimization
Proximal Policy Optimization
Deep Deterministic Policy Gradient
Twin Delayed DDPG
Soft Actor-Critic

UTILITIES DOCS

Logger
Plotter
MPI Tools
Run Utils

ETC.

Acknowledgements
About the Author

- **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.

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.

References

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)

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.

Other Public Implementations

- rllab
- rllib (Ray)

Previous   Next

Read the Docs   v: latest