garage Documentation

Release 0.0.1

garage contributors

May 30, 2019

Contents

garage is a framework for developing and evaluating reinforcement learning algorithms.

garage is a work in progress, input is welcome. The available documentation is limited for now.

CHAPTER 1

User Guide

The garage user guide explains how to install garage, how to run experiments, and how to implement new MDPs and new algorithms.

1.1 Installation

1.1.1 Express Install

The fastest way to set up dependencies for garage is via running the setup script.

Clone our repo (https://github.com/rlworkgroup/garage) and navigate to its directory.

A MuJoCo key is required for installation. You can get one here: https://www.roboti.us/license.html

Make sure you run these scripts from the root directory of the repo, not from the scripts directory.

- On Linux, run the following:

```
./scripts/setup_linux.sh --mjkey path-to-your-mjkey.txt --modify-bashrc
```

- On macOS, run the following:

```
./scripts/setup_macos.sh --mjkey path-to-your-mjkey.txt --modify-bashrc
```

The script sets up a conda environment, which is similar to `virtualenv`. To start using it, run the following:

```
source activate garage
```

Optionally, if you would like to run experiments that depends on the MuJoCo environment, you can set it up by running the following command:

```
./scripts/setup_mujoco.sh
```

and follow the instructions. You need to have the zip file for Mujoco v1.50 and the license file ready.

1.1.2 Manual Install

Anaconda

`garage` assumes that you are using Anaconda Python distribution. You can download it from *https://www.continuum.io/downloads<https://www.continuum.io/downloads*. Make sure to download the installer for Python 2.7.

System dependencies for pygame

A few environments in garage are implemented using Box2D, which uses pygame for visualization. It requires a few system dependencies to be installed first.

On Linux, run the following:

```
sudo apt-get install swig
sudo apt-get build-dep python-pygame
```

On macOS, run the following:

```
brew install swig sdl sdl_image sdl_mixer sdl_ttf portmidi
```

System dependencies for scipy

This step is only needed under Linux:

```
sudo apt-get install build-dep python-scipy
```

Install Python modules

```
conda env create -f environment.yml
```

1.1.3 GPU Support

To enable GPU support, you need to run the express installation script with the argument `--gpu`. This options installs GPU-supported Tensorflow and modules needed by Theano.

Before you run garage, you need to specify the directory for the CUDA library in environment variable `LD_LIBRARY_PATH`. You may need to replace the directory conforming to your CUDA version accordingly.

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda-9.0/lib64
```

You should now be able to use GPU in Tensorflow. For Theano, two additional steps are needed.

- Specify CUDA root in `~/.theanorc` (Create the file if it doesn't exist)

```
[cuda]
root = /usr/local/cuda-9.0
```

- Enable GPU for theano by

```
export THEANO_FLAGS=device=cuda,floatX=float32,force_device=True
```

1.2 Running Experiments

We use object oriented abstractions for different components required for an experiment. To run an experiment, simply construct the corresponding objects for the environment, algorithm, etc. and call the appropriate train method on the algorithm. A sample script is provided in `examples/trpo_cartpole.py`. The code is also pasted below for a quick glance:

```
from garage.algos.trpo import TRPO
from garage.baselines.linear_feature_baseline import LinearFeatureBaseline
from garage.envs.box2d.cartpole_env import CartpoleEnv
from garage.envs.normalized_env import normalize
from garage.policies.gaussian_mlp_policy import GaussianMLPPolicy

env = normalize(CartpoleEnv())

policy = GaussianMLPPolicy(
    env_spec=env.spec,
    # The neural network policy should have two hidden layers, each with 32 hidden␣
↪units.
    hidden_sizes=(32, 32)
)

baseline = LinearFeatureBaseline(env_spec=env.spec)

algo = TRPO(
    env=env,
    policy=policy,
    baseline=baseline,
    batch_size=4000,
    whole_paths=True,
    max_path_length=100,
    n_itr=40,
    discount=0.99,
    step_size=0.01,
)
algo.train()
```

Running the script for the first time might take a while for initializing Theano and compiling the computation graph, which can take a few minutes. Subsequent runs will be much faster since the compilation is cached. You should see some log messages like the following:

```
using seed 1
instantiating garage.envs.box2d.cartpole_env.CartpoleEnv
instantiating garage.policy.mean_std_nn_policy.MeanStdNNPolicy
using argument hidden_sizes with value [32, 32]
instantiating garage.baseline.linear_feature_baseline.LinearFeatureBaseline
instantiating garage.algo.trpo.TRPO
using argument batch_size with value 4000
using argument whole_paths with value True
using argument n_itr with value 40
using argument step_size with value 0.01
using argument discount with value 0.99
```

(continues on next page)

(continued from previous page)

```
using argument max_path_length with value 100
using seed 0
0%                          100%
[##############################] | ETA: 00:00:00
Total time elapsed: 00:00:02
2016-02-14 14:30:56.631891 PST | [trpo_cartpole] itr #0 | fitting baseline...
2016-02-14 14:30:56.677086 PST | [trpo_cartpole] itr #0 | fitted
2016-02-14 14:30:56.682712 PST | [trpo_cartpole] itr #0 | optimizing policy
2016-02-14 14:30:56.686587 PST | [trpo_cartpole] itr #0 | computing loss before
2016-02-14 14:30:56.698566 PST | [trpo_cartpole] itr #0 | performing update
2016-02-14 14:30:56.698676 PST | [trpo_cartpole] itr #0 | computing descent direction
2016-02-14 14:31:26.241657 PST | [trpo_cartpole] itr #0 | descent direction computed
2016-02-14 14:31:26.241828 PST | [trpo_cartpole] itr #0 | performing backtracking
2016-02-14 14:31:29.906126 PST | [trpo_cartpole] itr #0 | backtracking finished
2016-02-14 14:31:29.906335 PST | [trpo_cartpole] itr #0 | computing loss after
2016-02-14 14:31:29.912287 PST | [trpo_cartpole] itr #0 | optimization finished
2016-02-14 14:31:29.912483 PST | [trpo_cartpole] itr #0 | saving snapshot...
2016-02-14 14:31:29.914311 PST | [trpo_cartpole] itr #0 | saved
2016-02-14 14:31:29.915302 PST | ------------------------  -------------
2016-02-14 14:31:29.915365 PST | Iteration                    0
2016-02-14 14:31:29.915410 PST | Entropy                      1.41894
2016-02-14 14:31:29.915452 PST | Perplexity                   4.13273
2016-02-14 14:31:29.915492 PST | AverageReturn               68.3242
2016-02-14 14:31:29.915533 PST | StdReturn                   42.6061
2016-02-14 14:31:29.915573 PST | MaxReturn                  369.864
2016-02-14 14:31:29.915612 PST | MinReturn                   19.9874
2016-02-14 14:31:29.915651 PST | AverageDiscountedReturn     65.5314
2016-02-14 14:31:29.915691 PST | NumTrajs                  1278
2016-02-14 14:31:29.915730 PST | ExplainedVariance            0
2016-02-14 14:31:29.915768 PST | AveragePolicyStd             1
2016-02-14 14:31:29.921911 PST | BacktrackItr                 2
2016-02-14 14:31:29.922008 PST | MeanKL                       0.00305741
2016-02-14 14:31:29.922054 PST | MaxKL                        0.0360272
2016-02-14 14:31:29.922096 PST | LossBefore                  -0.0292939
2016-02-14 14:31:29.922146 PST | LossAfter                   -0.0510883
2016-02-14 14:31:29.922186 PST | ------------------------  -------------
```

1.2.1 Pickled Mode Experiments

`garage` also supports a "pickled" mode for running experiments, which supports more configurations like logging and parallelization. A sample script is provided in `examples/trpo_cartpole_pickled.py`. The content is pasted below:

```
from garage.algos.trpo import TRPO
from garage.baselines.linear_feature_baseline import LinearFeatureBaseline
from garage.envs.box2d.cartpole_env import CartpoleEnv
from garage.envs.normalized_env import normalize
from garage.misc.instrument import run_experiment
from garage.policies.gaussian_mlp_policy import GaussianMLPPolicy


def run_task(*_):
    env = normalize(CartpoleEnv())

    policy = GaussianMLPPolicy(
```

(continues on next page)

(continued from previous page)

```
        env_spec=env.spec,
        # The neural network policy should have two hidden layers, each with 32␣
↪hidden units.
        hidden_sizes=(32, 32)
    )

    baseline = LinearFeatureBaseline(env_spec=env.spec)

    algo = TRPO(
        env=env,
        policy=policy,
        baseline=baseline,
        batch_size=4000,
        max_path_length=100,
        n_itr=1000,
        discount=0.99,
        step_size=0.01,
        # Uncomment both lines (this and the plot parameter below) to enable plotting
        # plot=True,
    )
    algo.train()


run_experiment(
    run_task,
    # Number of parallel workers for sampling
    n_parallel=1,
    # Only keep the snapshot parameters for the last iteration
    snapshot_mode="last",
    # Specifies the seed for the experiment. If this is not provided, a random seed
    # will be used
    seed=1,
    # plot=True,
)
```

Note that the execution of the experiment (including the construction of relevant objects, like environment, policy, algorithm, etc.) has been wrapped in a function call, which is then passed to the *run_experiment* method, which serializes the fucntion call, and launches a script that actually runs the experiment.

The benefit for launching experiment this way is that we separate the configuration of experiment parameters and the actual execution of the experiment. *run_experiment* supports multiple ways of running the experiment, either locally, locally in a docker container, or remotely on ec2 (see the section on *Running jobs on EC2*). Multiple experiments with different hyper-parameter settings can be quickly constructed and launched simultaneously on multiple ec2 machines using this abstraction.

Another subtle point is that we use Theano for our algorithm implementations, which has rather poor support for mixed GPU and CPU usage. This might be handy when the main process wants to use GPU for the batch optimization phase, while multiple worker processes want to use the CPU for generating trajectory rollouts. Launching the experiment separately allows the worker processes to be properly initialized with Theano configured to use CPU.

Additional arguments for *run_experiment* (experimental):

- *exp_name*: If this is set, the experiment data will be stored in the folder *data/local/{exp_name}*. By default, the folder name is set to *experiment_{timestamp}*.
- *exp_prefix*: If this is set, and if *exp_name* is not specified, the experiment folder name will be set to *{exp_prefix}_{timestamp}*.

1.2.2 Running Experiments with TensorFlow and GPU

To run experiments in the TensorFlow tree of garage with the GPU enabled, set the flags use_tf and use_gpu to True when calling *run_experiment*, as shown in the code below:

```
run_experiment(
    run_task,
    # Number of parallel workers for sampling
    n_parallel=1,
    # Only keep the snapshot parameters for the last iteration
    snapshot_mode="last",
    # Specifies the seed for the experiment. If this is not provided, a random seed
    # will be used
    seed=1,
    # Always set to True when using TensorFlow
    use_tf=True,
    # Set to True to use GPU with TensorFlow
    use_gpu=True,
    # plot=True,
)
```

It's also possible to run TensorFlow with only the CPU by setting use_gpu to False, which is the default behavior when use_tf is enabled.

1.3 Implementing New Environments

In this section, we will walk through an example of implementing a point robot environment using our framework.

Each environment should implement at least the following methods / properties defined in the file `garage/envs/base.py`:

```
class Env(object):
    def step(self, action):
        """
        Run one timestep of the environment's dynamics. When end of episode
        is reached, reset() should be called to reset the environment's internal␣
↪state.
        Input
        -----
        action : an action provided by the environment
        Outputs
        -------
        (observation, reward, done, info)
        observation : agent's observation of the current environment
        reward [Float] : amount of reward due to the previous action
        done : a boolean, indicating whether the episode has ended
        info : a dictionary containing other diagnostic information from the previous␣
↪action
        """
        raise NotImplementedError

    def reset(self):
        """
        Resets the state of the environment, returning an initial observation.
        Outputs
        -------
```

(continues on next page)

(continued from previous page)

```
        observation : the initial observation of the space. (Initial reward is␣
↪assumed to be 0.)
        """
        raise NotImplementedError

    @property
    def action_space(self):
        """
        Returns a Space object
        """
        raise NotImplementedError

    @property
    def observation_space(self):
        """
        Returns a Space object
        """
        raise NotImplementedError
```

We will implement a simple environment with 2D observations and 2D actions. The goal is to control a point robot in 2D to move it to the origin. We receive position of a point robot in the 2D plane $(x, y) \in \mathbb{R}^2$. The action is its velocity $(\dot{x}, \dot{y}) \in \mathbb{R}^2$ constrained so that $|\dot{x}| \leq 0.1$ and $|\dot{y}| \leq 0.1$. We encourage the robot to move to the origin by defining its reward as the negative distance to the origin: $r(x, y) = -\sqrt{x^2 + y^2}$.

We start by creating a new file for the environment. We assume that it is placed under `examples/point_env.py`. First, let's declare a class inheriting from the base environment and add some imports:

```
from garage.envs.base import Env
from garage.envs.base import Step
from garage.spaces import Box
import numpy as np


class PointEnv(Env):

    # ...
```

For each environment, we will need to specify the set of valid observations and the set of valid actions. This is done by implementing the following property methods:

```
class PointEnv(Env):

    # ...

    @property
    def observation_space(self):
        return Box(low=-np.inf, high=np.inf, shape=(2,))

    @property
    def action_space(self):
        return Box(low=-0.1, high=0.1, shape=(2,))
```

The `Box` space means that the observations and actions are 2D vectors with continuous values. The observations can have arbitrary values, while the actions should have magnitude at most 0.1.

Now onto the interesting part, where we actually implement the dynamics for the MDP. This is done through two methods, `reset` and `step`. The `reset` method randomly initializes the state of the environment according to some

initial state distribution. To keep things simple, we will just sample the coordinates from a uniform distribution. The method should also return the initial observation. In our case, it will be the same as its state.

```
class PointEnv(Env):

    # ...

    def reset(self):
        self._state = np.random.uniform(-1, 1, size=(2,))
        observation = np.copy(self._state)
        return observation
```

The `step` method takes an action and advances the state of the environment. It should return a `Step` object (which is a wrapper around `namedtuple`), containing the observation for the next time step, the reward, a flag indicating whether the episode is terminated after taking the step, and optional extra keyword arguments (whose values should be vectors only) for diagnostic purposes. The procedure that interfaces with the environment is responsible for calling `reset` after seeing that the episode is terminated.

```
class PointEnv(Env):

    # ...

    def step(self, action):
        self._state = self._state + action
        x, y = self._state
        reward = - (x**2 + y**2) ** 0.5
        done = abs(x) < 0.01 and abs(y) < 0.01
        next_observation = np.copy(self._state)
        return Step(observation=next_observation, reward=reward, done=done)
```

Finally, we can implement some plotting to visualize what the MDP is doing. For simplicity, let's just print the current state of the MDP on the terminal:

```
class PointEnv(Env):

    # ...

    def render(self):
        print 'current state:', self._state
```

And we're done! We can now simulate the environment using the following diagnostic script:

```
python scripts/sim_env.py garage.envs.point_env --mode random
```

It simulates an episode of the environment with random actions, sampled from a uniform distribution within the defined action bounds.

You could also train a neural network policy to solve the task, which is probably an overkill. To do so, create a new script with the following content (we will use stub mode):

```
from garage.algos.trpo import TRPO
from garage.baselines.linear_feature_baseline import LinearFeatureBaseline
from garage.envs.point_env import PointEnv
from garage.envs.normalized_env import normalize
from garage.policies.gaussian_mlp_policy import GaussianMLPPolicy

env = normalize(PointEnv())
```

(continues on next page)

(continued from previous page)

```
policy = GaussianMLPPolicy(
    env_spec=env.spec,
)
baseline = LinearFeatureBaseline(env_spec=env.spec)
algo = TRPO(
    env=env,
    policy=policy,
    baseline=baseline,
)
algo.train()
```

Assume that the file is `examples/trpo_point.py`. You can then run the script:

```
python examples/trpo_point.py
```

1.4 Implementing New Algorithms (Basic)

In this section, we will walk through the implementation of the classical REINFORCE[1] algorithm, also known as the "vanilla" policy gradient. We will start with an implementation that works with a fixed policy and environment. The next section *Implementing New Algorithms (Advanced)* will improve upon this implementation, utilizing the functionalities provided by the framework to make it more structured and command-line friendly.

1.4.1 Preliminaries

First, let's briefly review the algorithm along with some notations. We work with an MDP defined by the tuple $(\mathcal{S}, \mathcal{A}, P, r, \mu_0, \gamma, T)$, where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition probability, $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, $\mu_0 : \mathcal{S} \to [0, 1]$ is the initial state distribution, $\gamma \in [0, 1]$ is the discount factor, and $T \in \mathbb{N}$ is the horizon. REINFORCE directly optimizes a parameterized stochastic policy $\pi_\theta : \mathcal{S} \times \mathcal{A} \to [0, 1]$ by performing gradient ascent on the expected return objective:

$$\eta(\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t)\right]$$

where the expectation is implicitly taken over all possible trajectories, following the sampling procedure $s_0 \sim \mu_0$, $a_t \sim \pi_\theta(\cdot|s_t)$, and $s_{t+1} \sim P(\cdot|s_t, a_t)$. By the likelihood ratio trick, the gradient of the objective with respect to θ is given by

$$\nabla_\theta \eta(\theta) = \mathbb{E}\left[\left(\sum_{t=0}^{T} \gamma^t r(s_t, a_t)\right)\left(\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\right)\right]$$

We can reduce the variance of this estimator by noting that for $t' < t$,

$$\mathbb{E}\left[r(s_{t'}, a_{t'}) \nabla_\theta \log \pi_\theta(a_t|s_t)\right] = 0$$

Hence,

$$\nabla_\theta \eta(\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} \gamma^{t'} r(s_{t'}, a_{t'})\right]$$

[1] Williams, Ronald J. "Simple statistical gradient-following algorithms for connectionist reinforcement learning." Machine learning 8.3-4 (1992): 229-256.

Often, we use the following estimator instead:

$$\nabla_\theta \eta(\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}, a_{t'})\right]$$

where $\gamma^{t'}$ is replaced by $\gamma^{t'-t}$. When viewing the discount factor as a variance reduction factor for the undiscounted objective, this alternative gradient estimator has less bias, at the expense of having a larger variance. We define $R_t := \sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}, a_{t'})$ as the empirical discounted return.

The above formula will be the central object of our implementation. The pseudocode for the whole algorithm is as below:

- Initialize policy π with parameter θ_1.
- For iteration $k = 1, 2, \ldots$:
  - Sample N trajectories $\tau_1, \ldots, \tau_n$ under the current policy θ_k, where $\tau_i = (s_t^i, a_t^i, R_t^i)_{t=0}^{T-1}$. Note that the last state is dropped since no action is taken after observing the last state.
  - Compute the empirical policy gradient:

$$\widehat{\nabla_\theta \eta(\theta)} = \frac{1}{NT} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t^i|s_t^i) R_t^i$$

  - Take a gradient step: $\theta_{k+1} = \theta_k + \alpha \widehat{\nabla_\theta \eta(\theta)}$.

1.4.2 Setup

As a start, we will try to solve the cartpole balancing task using a neural network policy. We will later generalize our algorithm to accept configuration parameters. But let's keep things simple for now.

```
from __future__ import print_function
from garage.envs.box2d.cartpole_env import CartpoleEnv
from garage.policies.gaussian_mlp_policy import GaussianMLPPolicy
from garage.envs.normalized_env import normalize
import numpy as np
import theano
import theano.tensor as TT
from lasagne.updates import adam

# normalize() makes sure that the actions for the environment lies
# within the range [-1, 1] (only works for environments with continuous actions)
env = normalize(CartpoleEnv())
# Initialize a neural network policy with a single hidden layer of 8 hidden units
policy = GaussianMLPPolicy(env.spec, hidden_sizes=(8,))

# We will collect 100 trajectories per iteration
N = 100
# Each trajectory will have at most 100 time steps
T = 100
# Number of iterations
n_itr = 100
# Set the discount factor for the problem
discount = 0.99
# Learning rate for the gradient update
learning_rate = 0.01
```

1.4.3 Collecting Samples

Now, let's collect samples for the environment under our current policy within a single iteration.

```
paths = []

for _ in xrange(N):
    observations = []
    actions = []
    rewards = []

    observation = env.reset()

    for _ in xrange(T):
        # policy.get_action() returns a pair of values. The second one returns a␣
↪dictionary, whose values contains
        # sufficient statistics for the action distribution. It should at least␣
↪contain entries that would be
        # returned by calling policy.dist_info(), which is the non-symbolic analog of␣
↪policy.dist_info_sym().
        # Storing these statistics is useful, e.g., when forming importance sampling␣
↪ratios. In our case it is
        # not needed.
        action, _ = policy.get_action(observation)
        # Recall that the last entry of the tuple stores diagnostic information about␣
↪the environment. In our
        # case it is not needed.
        next_observation, reward, terminal, _ = env.step(action)
        observations.append(observation)
        actions.append(action)
        rewards.append(reward)
        observation = next_observation
        if terminal:
            # Finish rollout if terminal state reached
            break

    # We need to compute the empirical return for each time step along the
    # trajectory
    returns = []
    return_so_far = 0
    for t in xrange(len(rewards) - 1, -1, -1):
        return_so_far = rewards[t] + discount * return_so_far
        returns.append(return_so_far)
    # The returns are stored backwards in time, so we need to revert it
    returns = returns[::-1]

    paths.append(dict(
        observations=np.array(observations),
        actions=np.array(actions),
        rewards=np.array(rewards),
        returns=np.array(returns)
    ))
```

Observe that according to the formula for the empirical policy gradient, we could concatenate all the collected data for different trajectories together, which helps us vectorize the implementation further.

```
observations = np.concatenate([p["observations"] for p in paths])
actions = np.concatenate([p["actions"] for p in paths])
```

(continues on next page)

(continued from previous page)

```
returns = np.concatenate([p["returns"] for p in paths])
```

1.4.4 Constructing the Computation Graph

We will use Theano for our implementation, and we assume that the reader has some familiarity with it. If not, it would be good to go through some tutorials first.

First, we construct symbolic variables for the input data:

```
# Create a Theano variable for storing the observations
# We could have simply written `observations_var = TT.matrix('observations')` instead␣
↪for this example. However,
# doing it in a slightly more abstract way allows us to delegate to the environment␣
↪for handling the correct data
# type for the variable. For instance, for an environment with discrete observations,␣
↪we might want to use integer
# types if the observations are represented as one-hot vectors.
observations_var = env.observation_space.new_tensor_variable(
    'observations',
    # It should have 1 extra dimension since we want to represent a list of␣
↪observations
    extra_dims=1
)
actions_var = env.action_space.new_tensor_variable(
    'actions',
    extra_dims=1
)
returns_var = TT.vector('returns')
```

Note that we can transform the policy gradient formula as

$$\widehat{\nabla_\theta \eta(\theta)} = \nabla_\theta \left(\frac{1}{NT} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \log \pi_\theta(a_t^i | s_t^i) R_t^i \right) = \nabla_\theta L(\theta)$$

where $L(\theta) = \frac{1}{NT} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \log \pi_\theta(a_t^i | s_t^i) R_t^i$ is called the surrogate function. Hence, we can first construct the computation graph for $L(\theta)$, and then take its gradient to get the empirical policy gradient.

```
# policy.dist_info_sym returns a dictionary, whose values are symbolic expressions␣
↪for quantities related to the
# distribution of the actions. For a Gaussian policy, it contains the mean and (log)␣
↪standard deviation.
dist_info_vars = policy.dist_info_sym(observations_var, actions_var)

# policy.distribution returns a distribution object under garage.distributions. It␣
↪contains many utilities for computing
# distribution-related quantities, given the computed dist_info_vars. Below we use␣
↪dist.log_likelihood_sym to compute
# the symbolic log-likelihood. For this example, the corresponding distribution is an␣
↪instance of the class
# garage.distributions.DiagonalGaussian
dist = policy.distribution

# Note that we negate the objective, since most optimizers assume a
# minimization problem
```

(continues on next page)

(continued from previous page)

```
surr = - TT.mean(dist.log_likelihood_sym(actions_var, dist_info_vars) * returns_var)

# Get the list of trainable parameters.
params = policy.get_params(trainable=True)
grads = theano.grad(surr, params)
```

1.4.5 Gradient Update and Diagnostics

We are almost done! Now, you can use your favorite stochastic optimization algorithm for performing the parameter update. We choose ADAM[2] in our implementation:

```
f_train = theano.function(
    inputs=[observations_var, actions_var, returns_var],
    outputs=None,
    updates=adam(grads, params, learning_rate=learning_rate),
    allow_input_downcast=True
)
f_train(observations, actions, returns)
```

Since this algorithm is on-policy, we can evaluate its performance by inspecting the collected samples:

```
print('Average Return:', np.mean([sum(path["rewards"]) for path in paths]))
```

The complete source code so far is available at `examples/vpg_1.py`.

1.4.6 Additional Tricks

Adding a Baseline

The variance of the policy gradient can be further reduced by adding a baseline. The refined formula is given by

$$\widehat{\nabla_\theta \eta(\theta)} = \frac{1}{NT} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t^i | s_t^i)(R_t^i - b(s_t^i))$$

We can do this since $\mathbb{E}\left[\nabla_\theta \log \pi_\theta(a_t^i | s_t^i) b(s_t^i)\right] = 0$

The baseline is typically implemented as an estimator of $V^\pi(s)$. In this case, $R_t^i - b(s_t^i)$ is an estimator of $A^\pi(s_t^i, a_t^i)$. The framework implements a few options for the baseline. A good balance of computational efficiency and accuracy is achieved by a linear baseline using state features, available at `garage/baselines/linear_feature_baseline.py`. To use it in our implementation, the relevant code looks like the following:

```
# ... initialization code ...

from garage.baselines.linear_feature_baseline import LinearFeatureBaseline
baseline = LinearFeatureBaseline(env.spec)

# ... inside the loop for each episode, after the samples are collected

path = dict(
    observations=np.array(observations),
```

(continues on next page)

[2] Kingma, Diederik P., and Jimmy Ba Adam. "A method for stochastic optimization." International Conference on Learning Representation. 2015.

(continued from previous page)

```
    actions=np.array(actions),
    rewards=np.array(rewards),
)

path_baseline = baseline.predict(path)
advantages = []
returns = []
return_so_far = 0
for t in xrange(len(rewards) - 1, -1, -1):
    return_so_far = rewards[t] + discount * return_so_far
    returns.append(return_so_far)
    advantage = return_so_far - path_baseline[t]
    advantages.append(advantage)
# The advantages are stored backwards in time, so we need to revert it
advantages = np.array(advantages[::-1])
# And we need to do the same thing for the list of returns
returns = np.array(returns[::-1])
```

Normalizing the returns

Currently, the learning rate we set for the algorithm is very susceptible to reward scaling. We can alleviate this dependency by whitening the advantages before computing the gradients. In terms of code, this would be:

```
advantages = (advantages - np.mean(advantages)) / (np.std(advantages) + 1e-8)
```

Training the baseline

After each iteration, we use the newly collected trajectories to train our baseline:

```
baseline.fit(paths)
```

The reason that this is executed after computing the baselines along the given trajectories is that in the extreme case, if we only have one trajectory starting from each state, and if the baseline could fit the data perfectly, then all the advantages would be zero, giving us no gradient signals at all.

Now, we can train the policy much faster (we need to change the learning rate accordingly because of the rescaling). The complete source code so far is available at `examples/vpg_2.py`

1.5 Implementing New Algorithms (Advanced)

In this section, we will walk through the implementation of vanilla policy gradient algorithm provided in the algorithm, available at `garage/algos/vpg.py`. It utilizes many functionalities provided by the framework, which we describe below.

1.5.1 The `BatchPolopt` Class

The `VPG` class inherits from `BatchPolopt`, which is an abstract class inherited by algorithms with a common structure. The structure is as follows:

- Initialize policy π with parameter θ_1.

- Initialize the computational graph structure.
- For iteration $k = 1, 2, \ldots$:
  - Sample N trajectories $\tau_1, \ldots, \tau_n$ under the current policy θ_k, where $\tau_i = (s_t^i, a_t^i, R_t^i)_{t=0}^{T-1}$. Note that the last state is dropped since no action is taken after observing the last state.
  - Update the policy based on the collected on-policy trajectories.
  - Print diagnostic information and store intermediate results.

Note the parallel between the structure above and the pseudocode for VPG. The `BatchPolopt` class takes care of collecting samples and common diagnostic information. It also provides an abstraction of the general procedure above, so that algorithm implementations only need to fill the missing pieces. The core of the `BatchPolopt` class is the `train()` method:

```
def train(self):
    # ...
    self.init_opt()
    for itr in xrange(self.start_itr, self.n_itr):
        paths = self.obtain_samples(itr)
        samples_data = self.process_samples(itr, paths)
        self.optimize_policy(itr, samples_data)
        params = self.get_itr_snapshot(itr, samples_data)
        logger.save_itr_params(itr, params)
        # ...
```

The methods `obtain_samples` and `process_samples` are implemented for you. The derived class needs to provide implementation for `init_opt`, which initializes the computation graph, `optimize_policy`, which updates the policy based on the collected data, and `get_itr_snapshot`, which returns a dictionary of objects to be persisted per iteration.

The `BatchPolopt` class powers quite a few algorithms:

- Vanilla Policy Gradient: `garage/algos/vpg.py`
- Natural Policy Gradient: `garage/algos/npg.py`
- Reward-Weighted Regression: `garage/algos/erwr.py`
- Trust Region Policy Optimization: `garage/algos/trpo.py`
- Relative Entropy Policy Search: `garage/algos/reps.py`

To give an illustration, here's how we might implement `init_opt` for VPG (the actual code in `garage/algos/vpg.py` is longer due to the need to log extra diagnostic information as well as supporting recurrent policies):

```
from garage.misc.ext import extract, compile_function, new_tensor

# ...

def init_opt(self):
    obs_var = self.env.observation_space.new_tensor_variable(
        'obs',
        extra_dims=1,
    )
    action_var = self.env.action_space.new_tensor_variable(
        'action',
        extra_dims=1,
    )
    advantage_var = TT.vector('advantage')
    dist = self.policy.distribution
```

(continues on next page)

(continued from previous page)

```
    old_dist_info_vars = {
        k: TT.matrix('old_%s' % k)
        for k in dist.dist_info_keys
        }
    old_dist_info_vars_list = [old_dist_info_vars[k] for k in dist.dist_info_keys]

    state_info_vars = {
        k: ext.new_tensor(
            k,
            ndim=2 + is_recurrent,
            dtype=theano.config.floatX
        ) for k in self.policy.state_info_keys
    }
    state_info_vars_list = [state_info_vars[k] for k in self.policy.state_info_keys]

    dist_info_vars = self.policy.dist_info_sym(obs_var, state_info_vars)
    logli = dist.log_likelihood_sym(action_var, dist_info_vars)

    # formulate as a minimization problem
    # The gradient of the surrogate objective is the policy gradient
    surr_obj = - TT.mean(logli * advantage_var)

    input_list = [obs_var, action_var, advantage_var] + state_info_vars_list

    self.optimizer.update_opt(surr_obj, target=self.policy, inputs=input_list)
```

The code is very similar to what we implemented in the basic version. Note that we use an optimizer, which in this case would be an instance of `garage.optimizers.first_order_optimizer.FirstOrderOptimizer`.

Here's how we might implement `optimize_policy`:

```
def optimize_policy(self, itr, policy, samples_data, opt_info):
    inputs = ext.extract(
        samples_data,
        "observations", "actions", "advantages"
    )
    agent_infos = samples_data["agent_infos"]
    state_info_list = [agent_infos[k] for k in self.policy.state_info_keys]
    inputs += tuple(state_info_list)
    self.optimizer.optimize(inputs)
```

1.5.2 Parallel Sampling

The `garage.parallel_sampler` module takes care of parallelizing the sampling process and aggregating the collected trajectory data. It is used by the `BatchPolopt` class like below:

```
# At the beginning of training, we need to register the environment and the policy
# onto the parallel_sampler
parallel_sampler.populate_task(self.env, self.policy)

# ...

# Within each iteration, we just need to update the policy parameters to
# each worker
cur_params = self.policy.get_param_values()
```

(continues on next page)

(continued from previous page)

```
paths = parallel_sampler.request_samples(
    policy_params=cur_params,
    max_samples=self.batch_size,
    max_path_length=self.max_path_length,
)
```

The returned `paths` is a list of dictionaries with keys `rewards`, `observations`, `actions`, `env_infos`, and `agent_infos`. The latter two, `env_infos` and `agent_infos` are in turn dictionaries, whose values are numpy arrays of the environment and agent (policy) information per time step stacked together. `agent_infos` will contain at least information that would be returned by calling `policy.dist_info()`. For a gaussian distribution with diagonal variance, this would be the means and the logarithm of the standard deviations.

After collecting the trajectories, the `process_samples` method in the `BatchPolopt` class computes the empirical returns and advantages by using the baseline specified through command line arguments (we'll talk about this below). Then it trains the baseline using the collected data, and concatenates all rewards, observations, etc. together to form a single huge tensor, just as we did for the basic algorithm implementation.

One different semantics from the basic implementation is that, rather than collecting a fixed number of trajectories with potentially different number of steps per trajectory (if the environment implements a termination condition), we specify a desired total number of samples (i.e. time steps) per iteration. The number of actual samples collected will be around this number, although sometimes slightly larger, to make sure that all trajectories are run until either the horizon or the termination condition is met.

1.6 Running jobs on EC2

garage comes with support for running jobs on EC2 cluster. Here are the steps to set it up:

1. Create an AWS account at https://aws.amazon.com/. You need to supply a valid credit card (or set up consolidated billing to link your account to a shared payer account) before running experiments.
2. After signing up, go to the Security Credentials console, click on the "Access Keys" tab, and create a new access key. If prompted with "You are accessing the security credentials page for your AWS account. The account credentials provide unlimited access to your AWS resources," click "Continue to Security Credentials". Save the downloaded `root_key.csv` file somewhere.
3. Set up environment variables. On Linux / macOS, edit your `~/.bash_profile` and add the following environment variables:

```
export AWS_ACCESS_KEY="(your access key)"
export AWS_ACCESS_SECRET="(your access secret)"
export GARAGE_S3_BUCKET="(think of a bucket name)"
```

For `GARAGE_S3_BUCKET`, come up with a name for the S3 bucket used to store your experiment data. See here for rules for bucket naming. You don't have to manually create the bucket as this will be handled by the script later. It should be of sufficient length to be unique.

4. Install the AWS command line interface. Set it up using the credentials you just downloaded by running `aws configure`. Alternatively, you can edit the file at `~/.aws/credentials` (create the folder / file if it does not exist) and set its content to the following:

```
[default]
aws_access_key_id = (your access key)
aws_secret_access_key = (your access secret)
```

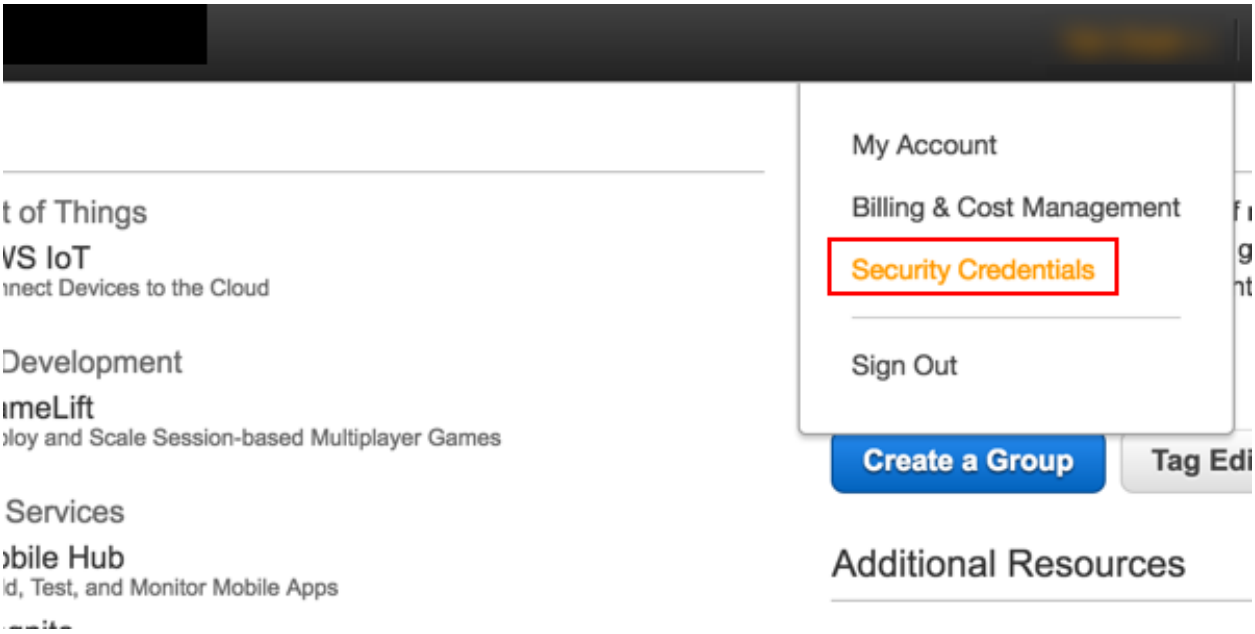

Fig. 1: Click on the Security Credentials tab.

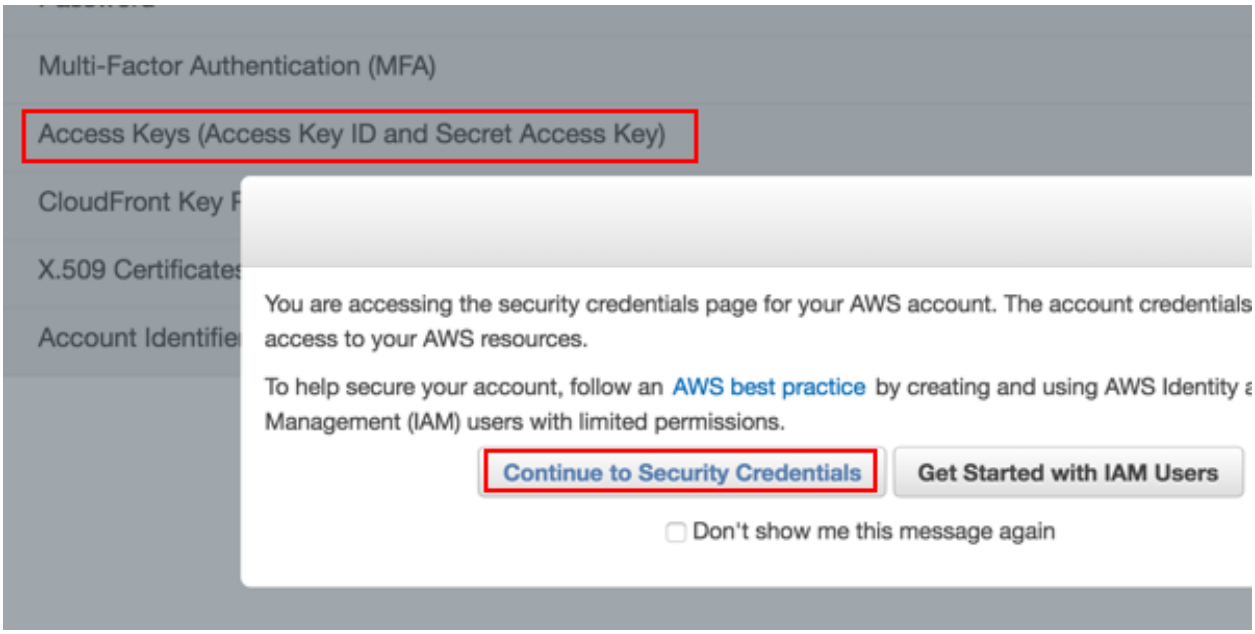

Fig. 2: Click "Continue to Security Credentials" if prompted. Then, click the Acces Keys tab.

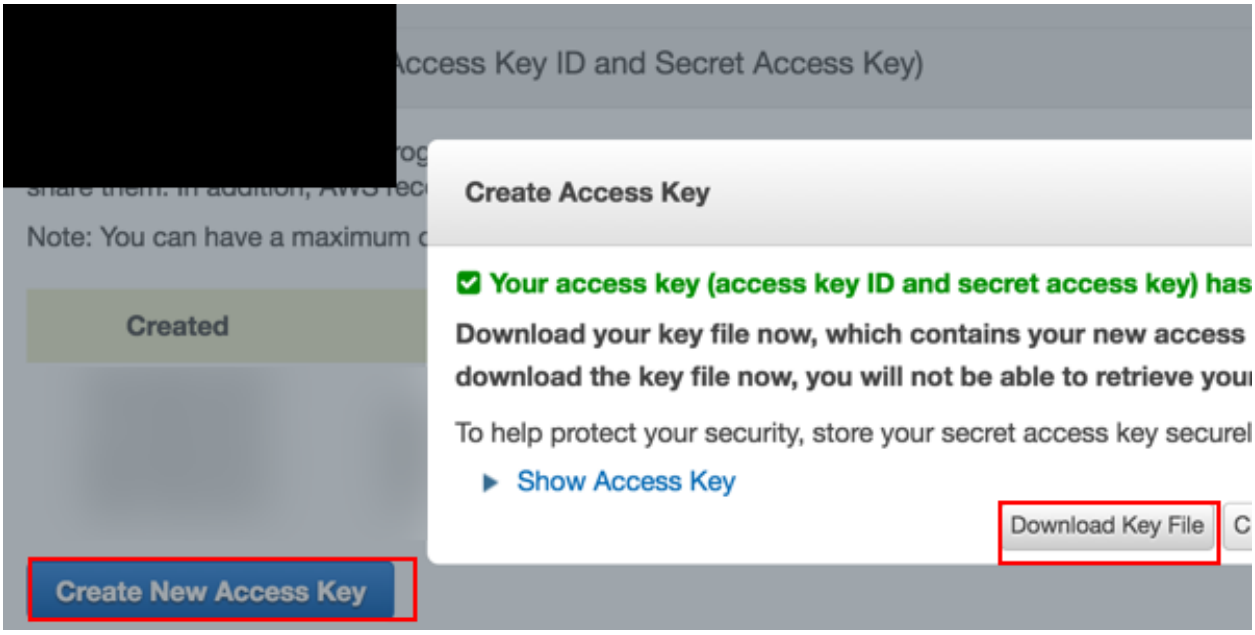

Fig. 3: Click "Create New Access Key". Then download the key file.

Note that there should not be quotes around the keys / secrets!

5. Open a new terminal to make sure that the environment variables configured are in effect. Also make sure you are using the Py3 version of the environment (`source activate garage`). Then, run the setup script:

```
python scripts/setup_ec2_for_garage.py
```

6. If the script runs fine, you've fully set up garage for use on EC2! Try running `examples/cluster_demo.py` to launch a single experiment on EC2. If it succeeds, you can then comment out the last line `sys.exit()` to launch the whole set of 15 experiments, each on an individual machine running in parallel. You can sign in to the EC2 panel to view spot instance requests status or running instances.

7. While the experiments are running (or when they are finished), use `python scripts/sync_s3.py first-exp` to download stats collected by the experiments. You can then run `python garage/viskit/frontend.py data/s3/first-exp` and navigate to http://localhost:5000 to view the results.

CHAPTER 2

Citing garage

If you use garage for academic research, you are highly encouraged to cite the following paper:

- Yan Duan, Xi Chen, Rein Houthooft, John Schulman, Pieter Abbeel. "Benchmarking Deep Reinforcement Learning for Continuous Control. *Proceedings of the 33rd International Conference on Machine Learning (ICML), 2016.*

CHAPTER 3

Indices and tables

- genindex
- modindex
- search