RLPy Documentation

Release 1.3.6

Alborz Geramifard, Robert H. Klein, Christoph Dann, Jonathan Ho

April 22, 2016

Contents

RLPy is a framework to conduct sequential decision making experiments. The current focus of this project lies on value-function-based reinforcement learning, specifically using linear function approximators. The project is distributed under the 3-Clause BSD License.

CHAPTER 1

Important Links

- Official source code repository: http://github.com/rlpy/rlpy
- Bitbucket mirror: http://bitbucket.org/rlpy/rlpy
- Documentation: http://rlpy.readthedocs.org
- Issue Tracker: https://github.com/rlpy/rlpy/issues
- Download latest version: https://pypi.python.org/pypi/rlpy
- Mailing list: rlpy@mit.edu (Subscribe here)

CHAPTER 2

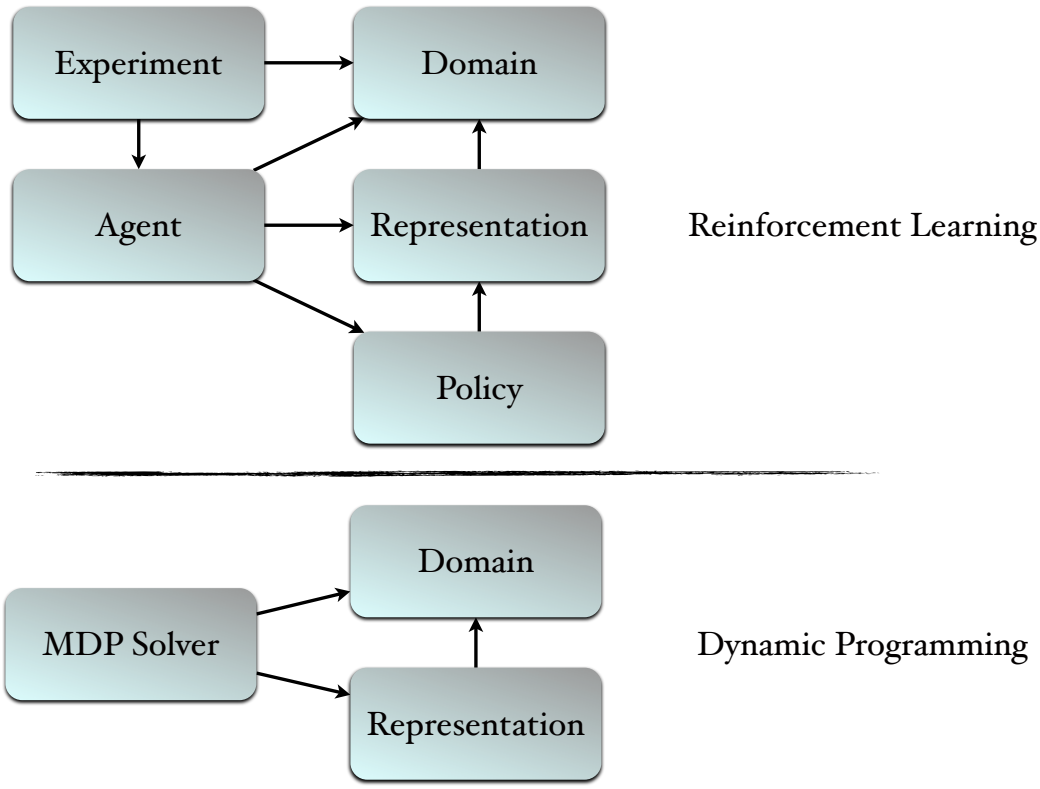
Documentation

2.1 Overview

2.1.1 Vision

RLPy is a framework for conducting sequential decision making experiments that involve value-function based approaches. It provides a modular toolbox, where various components can be linked together to create experiments.

2.1.2 The Big Picture



Reinforcement Learning (RL)

Setting up an RL experiment requires selecting the following 4 key components:

1. *Agent*: This is the box where learning happens. It is often done by changing the weight vector corresponding to the features.
2. *Policy*: This box is responsible to generate actions based on the current states. The action selection mechanism often dependends on the estimated value function.
3. *Representation*: In this framework, we assume the use of linear function approximators to represent the value function. This box realizes the underlying representation used for capturing the value function. Note that the features used for approximation can be non-linear.
4. *Domain*: This box is an MDP that we are interested to solve.

The *Experiment* class works as a glue that connect all these pieces together.

Dynamic Programming

If the full model of the MDP is known, Dynamic Programming techniques can be used to solve the MDP. To setup a DP experiment the following 3 components have to be set:

1. *MDP Solver*: Dynamic programming algorithm
2. *Representation*: Same as the RL case. Notice that the Value Iteration and Policy Iteration techniques can be only coupled with the tabular representation.
3. *Domain*: Same as the RL case.

Note: Each of the components mentioned here has several realizations in RLPy, yet this website provides guidance only on the main abstract classes, namely: *Agent*, *MDP Solver*, *Representation*, *Policy*, *Domain* and *Experiment*

See also:

The *tutorial page* provides simple 10-15 minutes examples on how various experiments can be setup and used.n

2.2 Acknowledgements

The project was partially funded by **ONR** and **AFOSR** grants.

2.3 Citing RLPy

If you use RLPy to conduct your research, please cite

Alborz Geramifard, Robert H Klein, Christoph Dann, William Dabney and Jonathan P How, RLPy: The Reinforcement Learning Library for Education and Research, 2013. http://acl.mit.edu/RLPy, April, 2013

Bibtex:

```
@ONLINE{RLPy,
author = {Alborz Geramifard and Robert H Klein and Christoph Dann and
William Dabney and Jonathan P How},
title = {{RLPy: The Reinforcement Learning Library for Education and Research}},
month = April,
```

```
year = {2013},
howpublished = {\url{http://acl.mit.edu/RLPy}},
}
```

2.4 Staying Connected

Feel free to join the rlpy list, rlpy@mit.edu, by clicking here. This list is intended for open discussion about questions, potential improvements, etc.

2.5 Installation

If you have a running Python distribution on your system, the external dependencies of RLPy are most likly installed already. Try to install rlpy with distutils as discribed in the following sections. If you do not have a Python distribution installed or run into trouble during installation, have a look at the Dependencies section below.

2.5.1 Stable Version

RLPy is available on Pypi. The latest stable version can be installed directly with pip:

```
pip install -U rlpy
```

This command downloads the latest release of RLPy and installs it into the default package location of your python distribution. Only the library itself is installed. If you like to have the documentation and example scripts, have a look at how to install the development version below.

For more information of the disutils package systems, have a look at the documentation.

If you are using MacOS make sure you have the latest version of Xcode using:

```
xcode-select --install
```

Alternatively, you can download RLPy manually, extract the package and execute the installer by hand:

```
python setup.py install
```

All RLPy packages are now successfully installed in Python's site-package directory.

Note: Only the Python files necessary to use the toolbox are installed. Documentation and example scripts how to use the toolbox are not installed by pip. They are, however, included in the package, which can be downloaded from https://pypi.python.org/pypi/rlpy. We recommend to download the archive and extract the *examples* folder to get examples of how to use the toolbox. Please also check out the tutorial at *Getting Started*.

2.5.2 Development Version

The current development version of RLPy can be installed in distutils editable mode with:

```
pip install -e git+https://github.com/rlpy/rlpy.git#egg=rlpy
```

This command clones the RLPy repository into the directory *src/rlpy*, compiles all C-extensions and tells the Python distribution where to find RLPy by creating a *.egg-link* file in the default package directory.

Alternatively, you can clone the RLPy directory manually by:

```
git clone https://github.com/rlpy/rlpy.git RLPy
```

and make your Python distribution aware of RLPy by:

```
python setup.py develop
```

Note: If you install rlpy directly from the development repository, you need cython to build the cython extensions. You can get the latest version of cython by: "pip install cython -U"

2.6 Dependencies

We recommend using the Anaconda Python distribution. This software package comes with a current version of Python and many libraries necessary for scientific computing. It simplifies installing and updating Python libraries significantly on Windows, MacOS and Linux. Please follow the original installation instructions of Anaconda.

RLPy requires the following software besides Python:

Tk as a backend for matplotlib and for visualizations of some domains.

Graphviz (optional) for creating the graphical ouput of the code profiling tool.

If you are using the Anaconda Python distribution, you can install Tk by executing:

```
conda install tk
```

In addition, RLPy requires Python 2.7 to run. We do not support Python 3 at the moment since most scientific libraries still require Python 2.

2.7 Getting Started

This tutorial covers the most common type of experiment in reinforcement learning: the control experiment. An agent is supposed to find a good policy while interacting with the domain.

Note: If you don't use the developer verion of rlpy but installed the toolbox via pip you can get the example scripts referenced in this tutorial as follows: Download the latest RLPy package from https://pypi.python.org/pypi/rlpy and extract the *examples* folder from the archive. In this folder you find several examples of how to use RLPy.

2.7.1 First Run

Begin by looking at the file *examples/tutorial/gridworld.py*:

```
1 #!/usr/bin/env python
2 """
3 Getting Started Tutorial for RLPy
4 ================================
5
```

```
"""
This file contains a very basic example of a RL experiment:
A simple Grid-World.
"""
__author__ = "Robert H. Klein"
from rlpy.Domains import GridWorld
from rlpy.Agents import Q_Learning
from rlpy.Representations import Tabular
from rlpy.Policies import eGreedy
from rlpy.Experiments import Experiment
import os


def make_experiment(exp_id=1, path="./Results/Tutorial/gridworld-qlearning"):
    """
    Each file specifying an experimental setup should contain a
    make_experiment function which returns an instance of the Experiment
    class with everything set up.

    @param id: number used to seed the random number generators
    @param path: output directory where logs and results are stored
    """
    opt = {}
    opt["exp_id"] = exp_id
    opt["path"] = path

    # Domain:
    maze = os.path.join(GridWorld.default_map_dir, '4x5.txt')
    domain = GridWorld(maze, noise=0.3)
    opt["domain"] = domain

    # Representation
    representation = Tabular(domain, discretization=20)

    # Policy
    policy = eGreedy(representation, epsilon=0.2)

    # Agent
    opt["agent"] = Q_Learning(representation=representation, policy=policy,
                        discount_factor=domain.discount_factor,
                        initial_learn_rate=0.1,
                        learn_rate_decay_mode="boyan", boyan_N0=100,
                        lambda_=0.)
    opt["checks_per_policy"] = 100
    opt["max_steps"] = 2000
    opt["num_policy_checks"] = 10
    experiment = Experiment(**opt)
    return experiment

if __name__ == '__main__':
    experiment = make_experiment(1)
    experiment.run(visualize_steps=False,  # should each learning step be shown?
                   visualize_learning=True,  # show policy / value function?
                   visualize_performance=1)  # show performance runs?
    experiment.plot()
    experiment.save()
```

The file is an example for a reinforcement learning experiment. The main components of such an experiment are the **domain**, *GridWorld* in this case, the **agent** (*Q_Learning*), which uses the **policy** *eGreedy* and the value function

representation *Tabular*. The **experiment** *Experiment* is in charge of the execution of the experiment by handling the interaction between the agent and the domain as well as storing the results on disk (see also *Overview*).

The function *make_experiment* gets an id, which specifies the random seeds and a path where the results are stored. It returns an instance of an *Experiment* which is ready to run. In line 53, such an experiment is created and then executed in line 54 by calling its *run* method. The three parameters of *run* control the graphical output. The result are plotted in line 57 and subsequently stored in line 58.

You can run the file by executing it with the ipython shell from the rlpy root directory:

```
ipython examples/tutorial/gridworld.py
```

Tip: We recommend using the IPython shell. Compared to the standard interpreter it provides color output and better help functions. It is more comportable to work with in general. See the Ipython homepage for details.

Note: If you want to use the standard python shell make sure the rlpy root directory is in the python seach path for modules. You can for example use:

```
PYTHONPATH=. python examples/tutorial/gridworld.py
```

Tip: You can also use the IPython shell interactively and then run the script from within the shell. To do this, first start the interactive python shell with:

```
ipython
```

and then inside the ipython shell execute:

```
%run examples/tutorial/gridworld.py
```

This will not terminate the interpreter after running the file and allows you to inspect the objects interactively afterwards (you can exit the shell with CTRL + D).

2.7.2 What Happens During a Control Experiment

During an experiment, the agent performs a series of episodes, each of which consists of a series of steps. Over the course of its lifetime, the agent performs a total of *max_steps* learning steps, each of which consists of:

1. The agent choses an action given its (exploration) policy
2. The domain transitions to a new state
3. The agent observes the old and new state of the domain as well as the reward for this transition and improves its policy based on this new information

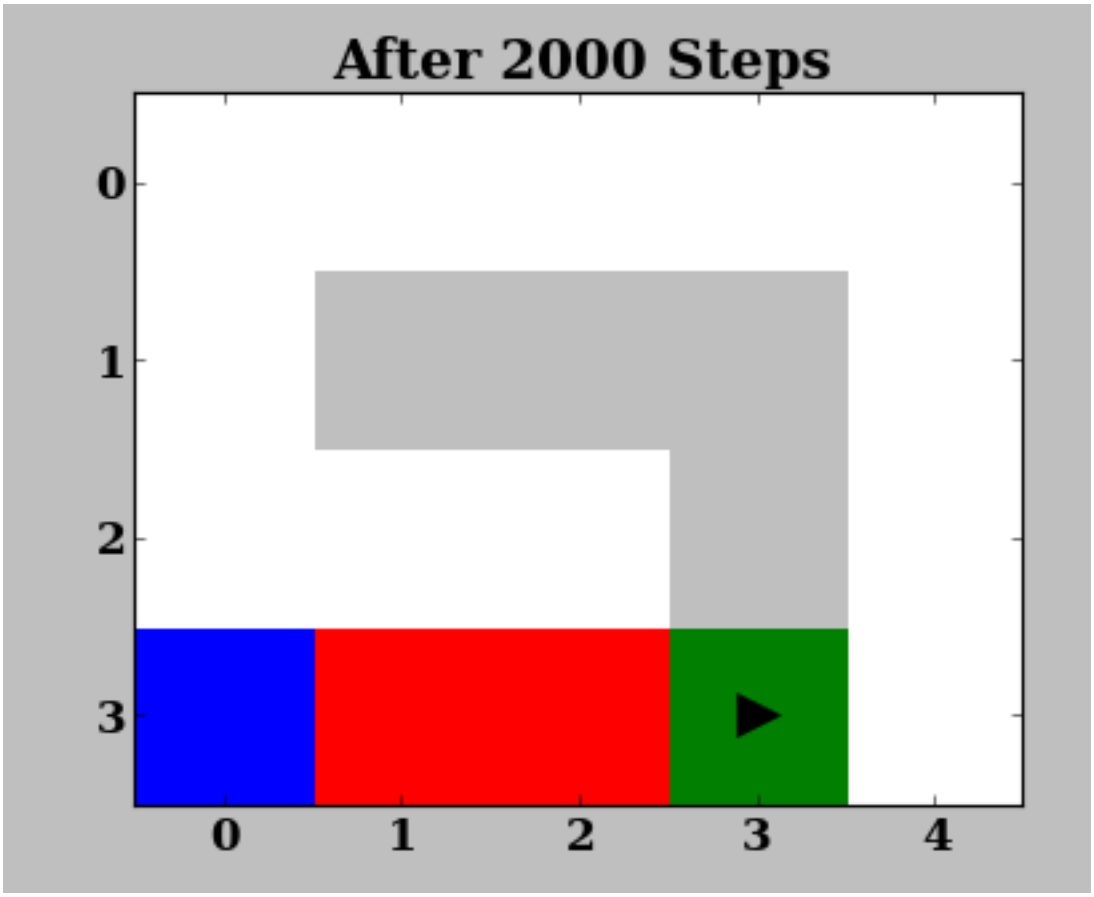
To track the performance of the agent, the quality of its current policy is assessed *num_policy_checks* times during the experiment at uniformly spaced intervals (and one more time right at the beginning). At each policy check, the agent is allowed to interact with the domain in what are called **performance runs**, with *checks_per_policy* runs occuring in each. (Using these multiple samples helps smooth the resulting performanace.) During performance runs, the agent does not do any exploration but always chooses actions optimal with respect to its value function. Thus, each step in a performance run consists of:

1. The agent choses an action it thinks is optimal (e.g. greedy w.r.t. its value function estimate)
2. The domain transitions to a new state

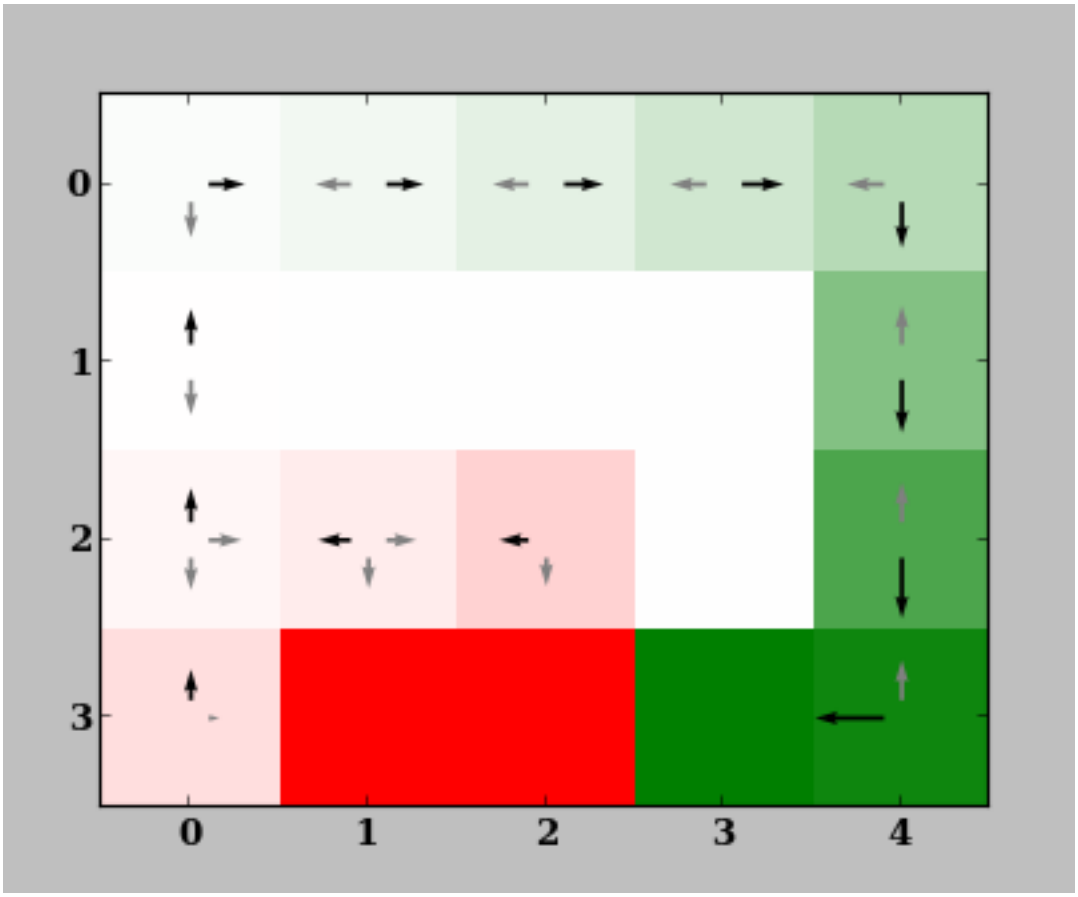
Note: No learning happens during performance runs. The total return for each episode of performance runs is averaged to obtain a quality measure of the agent's policy.

2.7.3 Graphical Output

While running the experiment you should see two windows, one showing the domain:



and one showing the value function:



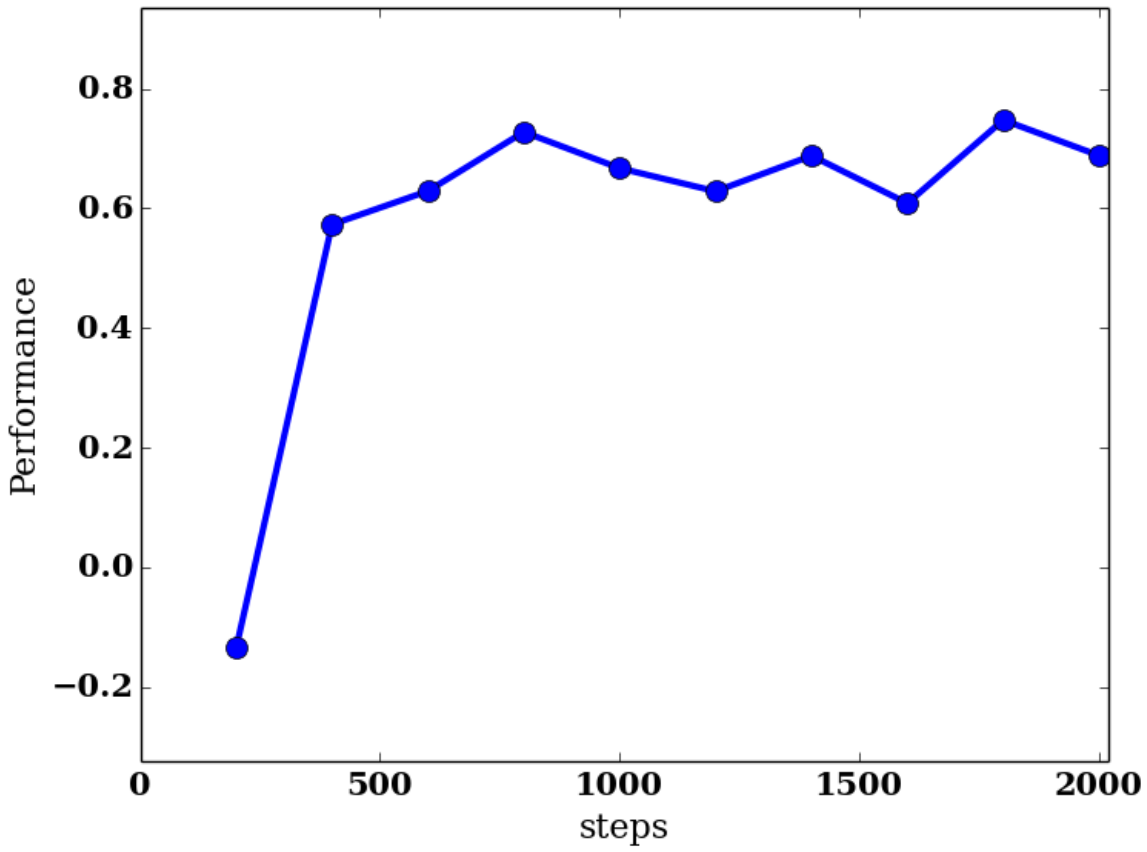
The Domain window is a visual representation of the domain (here, *GridWorld*) and is useful in quickly judging or demonstrating the performance of an agent. In this domain, the agent (triangle) has to move from the start (blue) to the goal (green) location in the shortest distance possible, while avoiding the pits (red). The agent receives -0.001 reward every step. When it reaches the goal or a pit, it obtains rewards of +1.0 or and the episode is terminated.

The value function window shows the value function and the resulting policy. It is shown because *visualize_learning=True*. Notice how the policy gradually converges to the optimal, direct route which avoids pits. After successive iterations, the agent learns the high (green) value of being in states that lie along the optimal path, even though they offer no immediate reward. It also learns the low (red) value of unimportant / undesirable states.

The set of possible actions in each grid is highlighted by arrows, where the size of arrows correspond to the state-action value function $Q(s, a)$. The best action is shown in black. If the agent has not learned the optimal policy in some grid cells, it has not explored enough to learn the correct action. (This often happens in Row 2, Column 1 of this example, where the correct action is *left*.) The agent likely still performs well though, since such states do not lie along the optimal route from the initial state *s0*; they are only rarely reached either because of ϵ-greedy policy which choses random actions with probability $\epsilon = 0.2$, or noise in the domain which takes a random action despite the one commanded.

Most domains in RLPy have a visualization like *GridWorld* and often also a graphical presentation of the policy or value function.

At the end of the experiment another window called *Performance* pops up and shows a plot of the average return during each policy assessment.



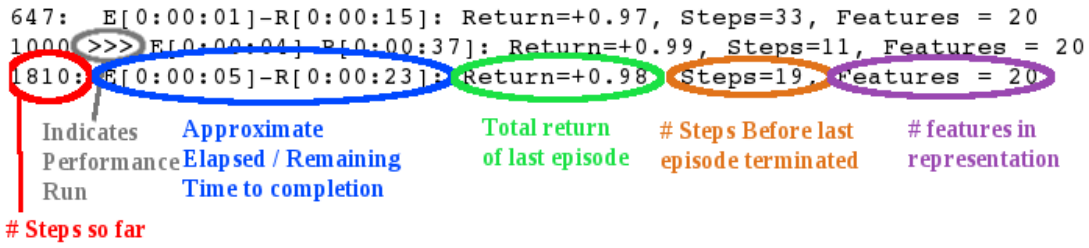
As we can see the agent learns after about 500 steps to obtain on average a reward of 0.7. The theoretically optimal reward for a single run is 0.99. However, the noise in the domain causes the agent to take the commanded action only 70% of the time (see the domain initialization in line 32); thus the total reward is correspondingly lower on average. In fact, the policy learned by the agent after 500 steps is the optimal one.

2.7.4 Console Outputs

During execution of *examples/tutorial/gridworld.py*, you should see in the console window output similar to the following:

```
647: E[0:00:01]-R[0:00:15]: Return=+0.97, Steps=33, Features = 20
1000 >>> E[0:00:04]-R[0:00:37]: Return=+0.99, Steps=11, Features = 20
1810: E[0:00:05]-R[0:00:23]: Return=+0.98, Steps=19, Features = 20
```

Each part has a specific meaning:



Lines with >>> are the averaged results of a policy assessment. Results of policy assessments are always shown. The outcome of learning episodes is shown only every second. You might therefore see no output for learning episodes if your computer is fast enough to do all learning steps between two policy assessments in less than one second.

Note: Throughout these experiments, if you see error messages similar to: `rlpy/Tools/transformations.py:1886: UserWarning: failed to import module _transformations` you may safely ignore them. They merely reflect that configuration does not support all features of rlpy.

2.7.5 A Slightly More Challenging Domain: Inverted Pole Balancing

We will now look at how to run experiments in batch and how to analyze and compare the performance of different methods on the same task. To this end, we compare different value function representations on the Cart-Pole Balancing task with an infinite track. The task is to keep a pole balanced upright. The pole is mounted on a cart which we can either push to the left or right.

The experimental setup is specified in *examples/tutorial/infTrackCartPole_tabular.py* with a tabular representation and in *examples/tutorial/infTrackCartPole_rbfs.py* with radial basis functions (RBFs). The content of *infTrackCartPole_rbfs.py* is

```
"""
Cart-pole balancing with randomly positioned radial basis functions
"""
from rlpy.Domains import InfCartPoleBalance
from rlpy.Agents import Q_Learning
from rlpy.Representations import *
from rlpy.Policies import eGreedy
from rlpy.Experiments import Experiment
import numpy as np
from hyperopt import hp

param_space = {
    "num_rbfs": hp.qloguniform("num_rbfs", np.log(1e1), np.log(1e4), 1),
    'resolution': hp.quniform("resolution", 3, 30, 1),
    'lambda_': hp.uniform("lambda_", 0., 1.),
    'boyan_N0': hp.loguniform("boyan_N0", np.log(1e1), np.log(1e5)),
    'initial_learn_rate': hp.loguniform("initial_learn_rate", np.log(5e-2), np.log(1))}


def make_experiment(
        exp_id=1, path="./Results/Temp/{domain}/{agent}/{representation}/",
        boyan_N0=753,
        initial_learn_rate=.7,
        resolution=25.,
        num_rbfs=206.,
```

```
        lambda_=0.75):
    opt = {}
    opt["exp_id"] = exp_id
    opt["path"] = path
    opt["max_steps"] = 5000
    opt["num_policy_checks"] = 10
    opt["checks_per_policy"] = 10

    domain = InfCartPoleBalance(episodeCap=1000)
    opt["domain"] = domain
    representation = RBF(domain, num_rbfs=int(num_rbfs),
                         resolution_max=resolution, resolution_min=resolution,
                         const_feature=False, normalize=True, seed=exp_id)
    policy = eGreedy(representation, epsilon=0.1)
    opt["agent"] = Q_Learning(
        policy, representation, discount_factor=domain.discount_factor,
        lambda_=lambda_, initial_learn_rate=initial_learn_rate,
        learn_rate_decay_mode="boyan", boyan_N0=boyan_N0)
    experiment = Experiment(**opt)
    return experiment

if __name__ == '__main__':
    experiment = make_experiment(1)
    experiment.run_from_commandline()
    experiment.plot()
```

Again, as the first GridWorld example, the main content of the file is a *make_experiment* function which takes an id, a path and some more optional parameters and returns an `Experiment.Experiment` instance. This is the standard format of an RLPy experiment description and will allow us to run it in parallel on several cores on one computer or even on a computing cluster with numerous machines.

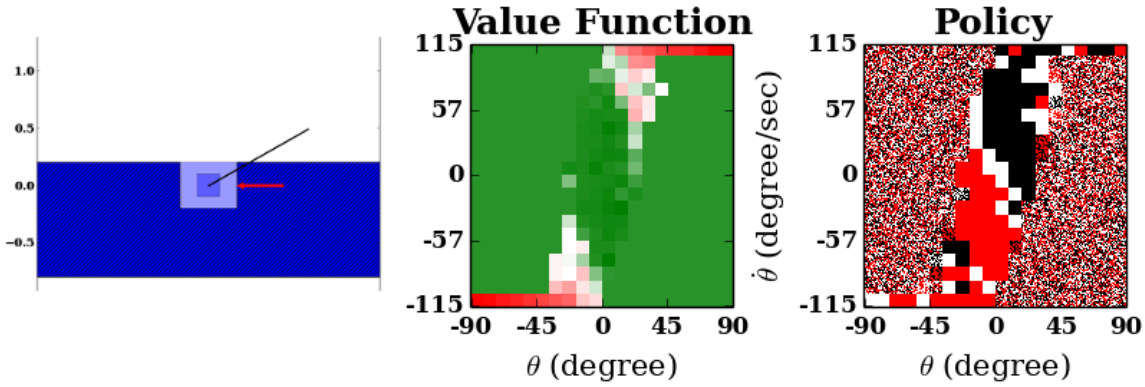
The content of *infTrackCartPole_tabular.py* is very similar but differs in the definition of the representation parameter of the agent. Compared to our first example, the experiment is now executed by calling its `Experiments.Experiment.run_from_commandline()` method. This is a wrapper around `Experiments.Experiment.run()` and allows to specify the options for visualization during the execution with command line arguments. You can for example run:

```
ipython examples/tutorial/infTrackCartPole_tabular.py -- -l -p
```

from the command line to run the experiment with visualization of the performance runs steps, policy and value function.

Note: The —— is only necessary, when executing a script directly at start-up of IPython. If our use the standard python interpreter or execute the file from within IPython with *%run* you can omit the ——.

Note: As learning occurs, execution may appear to slow down; this is merely because as the agent learns, it is able to balance the pendulum for a greater number of steps, and so each episode takes longer.



The value function (center), which plots pendulum angular rate against its angle, demonstrates the highly undesirable states of a steeply inclined pendulum (near the horizontal) with high angular velocity in the direction in which it is falling. The policy (right) initially appears random, but converges to the shape shown, with distinct black (counter-clockwise torque action) and red (clockwise action) regions in the first and third quadrants respectively, and a white stripe along the major diagonal between. This makes intuitive sense; if the pendulum is left of center and/or moving counterclockwise (third quadrant), for example, a corrective clockwise torque action should certainly be applied. The white stripe in between shows that no torque should be applied to a balanced pendulum with no angular velocity, or if it lies off-center but has angular velocity towards the balance point.

If you pass no command line arguments, no visualization is shown and only the performance graph at the end is produced. For an explanation of each command line argument type:

```
ipython examples/tutorial/infTrackCartPole_tabular.py -h
```

When we run the experiment with the tabular representation, we see that the pendulum can be balanced sometimes, but not reliably.

In order to properly assess the quality of the learning algorithm using this representation, we need to average over several independent learning sequences. This means we need to execute the experiment with different seeds.

2.7.6 Running Experiments in Batch

The module `Tools.run` provides several functions that are helpful for running experiments. The most important one is `Tools.run.run()`.

It allows us to run a specific experimental setup specified by a *make_experiment* function in a file with multiple seeds in parallel. For details see `Tools.run.run()`.

You find in *examples/tutorial/run_infTrackCartPole_batch.py* a short script with the following content:

```
from rlpy.Tools.run import run
run("examples/tutorial/infTrackCartPole_rbfs.py", "./Results/Tutorial/InfTrackCartPole/RBFs",
    ids=range(10), parallelization="joblib")

run("examples/tutorial/infTrackCartPole_tabular.py", "./Results/Tutorial/InfTrackCartPole/Tabular",
    ids=range(10), parallelization="joblib")
```

This script first runs the infinite track cartpole experiment with radial basis functions ten times with seeds 1 to 10. Subsequently the same is done for the experiment with tabular representation. Since we specified *parallelization=joblib*, the joblib library is used to run the experiment in parallel on all but one core of your computer. You can execute this script with:

```
ipython examples/tutorial/run_infTrackCartPole_batch.py
```

Note: This might take a few minutes depending on your hardware, and you may see minimal output during this time.

2.7.7 Analyzing Results

Running experiments via `Tools.run.run()` automatically saves the results to the specified path. If we run an `Experiments.Experiment` instance directly, we can store the results on disc with the `Experiments.Experiment.save()` method. The outcomes are then stored in the directory that is passed during initialization. The filename has the format *XXX-results.json* where *XXX* is the id / seed of the experiment. The results are stored in the JSON format that look for example like:

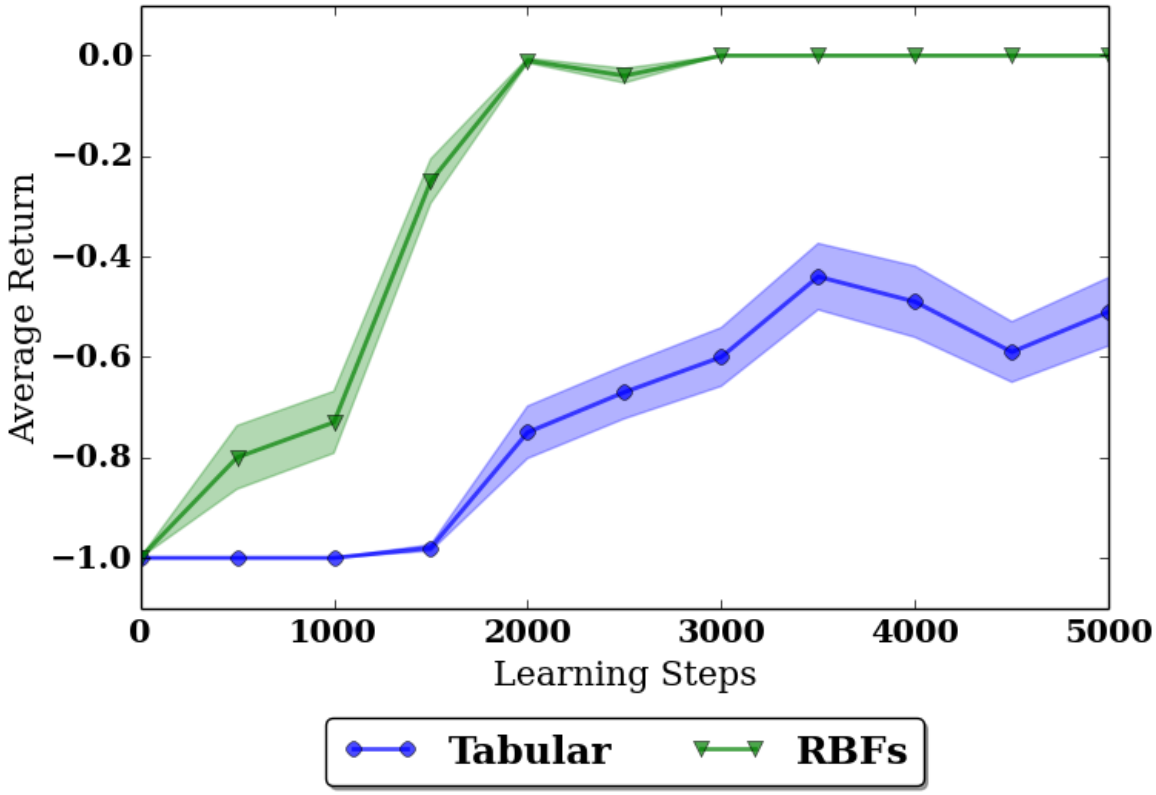
```
{"learning_steps": [0, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000],
 "terminated": [1.0, 1.0, 1.0, 1.0, 0.9, 0.8, 0.3, 0.3, 0.0, 0.7, 0.0],
 "return": [-1.0, -1.0, -1.0, -1.0, -0.9, -0.8, -0.3, -0.3, 0.0, -0.7, 0.0],
 "learning_time": [0, 0.31999999999999995, 0.6799999999999998, 1.0099999999999998, 1.559999999999999
 "num_features": [400, 400, 400, 400, 400, 400, 400, 400, 400, 400, 400],
 "learning_episode": [0, 45, 71, 85, 99, 104, 110, 121, 136, 144, 152],
 "discounted_return": [-0.6646429809896579, -0.529605466143065, -0.09102296558580342, -0.20856188627
 "seed": 1,
 "steps": [9.0, 14.1, 116.2, 49.3, 355.5, 524.2, 807.1, 822.4, 1000.0, 481.0, 1000.0]}
```

The measurements of each assessment of the learned policy is stored sequentially under the corresponding name. The module `Tools.results` provides a library of functions and classes that simplify the analysis and visualization of results. See the the api documentation for details.

To see the different effect of RBFs and tabular representation on the performance of the algorithm, we will plot their average return for each policy assessment. The script saved in *examples/tutorial/plot_result.py* shows us how:

```
1 import rlpy.Tools.results as rt
2
3 paths = {"RBFs": "./Results/Tutorial/InfTrackCartPole/RBFs",
4          "Tabular": "./Results/Tutorial/InfTrackCartPole/Tabular"}
5
6 merger = rt.MultiExperimentResults(paths)
7 fig = merger.plot_avg_sem("learning_steps", "return")
8 rt.save_figure(fig, "./Results/Tutorial/plot.pdf")
```

First, we specify the results we specify the directories where the results are stored and give them a label, here *RBFs* and *Tabular*. Then we create an instance of `Tools.results.MultiExperimentResults` which loads all corresponding results an let us analyze and transform them. In line 7, we plot the average return of each method over the number learning steps done so far. Finally, the plot is saved in *./Results/Tutorial/plot.pdf* in the lossless pdf format. When we run the script, we get the following plot



The shaded areas in the plot indicate the standard error of the sampling mean. We see that with radial basis functions the agent is able to perform perfectly after 2000 learning steps, but with the tabular representation, it stays at a level of -0.4 return per episode. Since the value function only matters around the center (zero angle, zero velocity), radial basis functions can capture the necessary form there much more easily and therefore speed up the learning process.

2.7.8 Tuning Hyperparameters

The behavior of each component of an agent can be drastically modified by its parameters (or hyperparameters, in contrast to the parameters of the value function that are learned). The module `Tools.hypersearch` provides tools for optimizing these parameters to get the best out of the algorithms.

We first need to specify what the hyperparameters for a specific experimental setup are and what values they can possibly take. We therefore again look at part of *examples/tutorial/infTrackCartPole_rbfs.py*

```
1
2  param_space = {
3      "num_rbfs": hp.qloguniform("num_rbfs", np.log(1e1), np.log(1e4), 1),
4      'resolution': hp.quniform("resolution", 3, 30, 1),
5      'lambda_': hp.uniform("lambda_", 0., 1.),
6      'boyan_N0': hp.loguniform("boyan_N0", np.log(1e1), np.log(1e5)),
7      'initial_learn_rate': hp.loguniform("initial_learn_rate", np.log(5e-2), np.log(1))}
8
9
10 def make_experiment(
11         exp_id=1, path="./Results/Temp/{domain}/{agent}/{representation}/",
```

```
        boyan_N0=753,
        initial_learn_rate=.7,
        resolution=25.,
        num_rbfs=206.,
        lambda_=0.75):
    opt = {}
    opt["exp_id"] = exp_id
    opt["path"] = path
    opt["max_steps"] = 5000
```

The variable *param_space* contains the definition of the space of hyperparameters we are considering. As the *make_experiment* function, the variable needs to have exactly this name. For details on how this definition has to look like we refer to the documentation of hyperopt, the package we are using for optimizing hyperparameters.

For each hyperparameter (in this example *num_rbfs*, *resolution*, *lambda_*, *boyan_N0* and *initial_alpha*), the *make_experiment* function has to have an optional argument with the same name.

The script saved in *examples/tutorial/run_parametersearch.py* shows us how to perform a quick search good parameters

```
from rlpy.Tools.hypersearch import find_hyperparameters
best, trials = find_hyperparameters(
    "examples/tutorial/infTrackCartPole_rbfs.py",
    "./Results/Tutorial/InfTrackCartPole/RBFs_hypersearch",
    max_evals=10, parallelization="joblib",
    trials_per_point=5)
print best
```

Warning: Running this script might take a while (approx. 5-30 min)

The `Tools.hypersearch.find_hyperparameters()` function is the most important tools for finding good parameters. For details on how to use it see its api documentation.

During the optimization, the results of several an entire experimental run need to be compressed into one target value. The parameter *objective* controls which quantity to optimize. In this example, it is *maximize the reward*. We could just take the return of the policy assessment with the most observations (the final policy). However, this can lead to artifacts and causes all hyperparameters that yield the same final performance to be considered equally good, no matter how fast they reach this performance. Therefore, the target value is computed as described below.

The target value is the weighted average over all measurements of the desired quantity (e.g., the average return during each policy assessment). The weights increase quadratically with the observation number, i.e., the return achieved in the first policy assessment has weight 1, the second weight 2, 9, 16, ... This weighting scheme ensures makes the final performance most important but also takes into account previous ones and therefore makes sure that the convergence speed is reflected in the optimized value. This weighting scheme has shown to be very robust in practice.

When we run the search, we obtain the following result:

> **{'initial_alpha': 0.3414408997566183,** 'resolution': 21.0, 'num_rbfs': 6988.0, 'lambda_': 0.38802888678400627, 'boyan_N0': 5781.602341902433}

Note: This parameters are not optimal. To obtain better ones, the number of evaluations need to be increased to 50 - 100. Also, *trials_per_point=10* makes the search more reliable. Be aware that 100 evaluations with 10 trials each result in 1000 experiment runs, which can take a very long time.

We can for example save these values by setting the default values in *make_experiment* accordingly.

2.7.9 What to do next?

In this introduction, we have seen how to

- run a single experiment with visualizations for getting an intuition of a domain and an agent
- run experiments in batch in parallel on multiple cores
- analyze and create plot the results of experiments
- optimize hyperparameters.

We covered the basic tasks of working with rlpy. You can see more examples of experiments in the *examples* directory. If you want to implement a new algorithm or problem, have a look at the api documentation. Contributions to rlpy of each flavor are always welcome!

2.7.10 Staying Connected

Feel free to join the rlpy list, rlpy@mit.edu by clicking here. This list is intended for open discussion about questions, potential improvements, etc.

> The only real mistake is the one from which we learn nothing.
>
> —John Powell

2.8 Creating a New Agent

This tutorial describes the standard RLPy `Agent` interface, and illustrates a brief example of creating a new learning agent.

The Agent receives observations from the Domain and updates the Representation accordingly.

In a typical Experiment, the Agent interacts with the Domain in discrete timesteps. At each Experiment timestep the Agent receives some observations from the Domain which it uses to update the value function Representation of the Domain (ie, on each call to its `learn()` function). The Policy is used to select an action to perform. This process (observe, update, act) repeats until some goal or fail state, determined by the Domain, is reached. At this point the Experiment determines whether the agent starts over or has its current policy tested (without any exploration).

Note: You may want to review the namespace / inheritance / scoping rules in Python.

2.8.1 Requirements

- Each learning agent must be a subclass of `Agent` and call the `__init__()` function of the Agent superclass.
- Accordingly, each Agent must be instantiated with a Representation, Policy, and Domain in the `__init__()` function
- Any randomization that occurs at object construction *MUST* occur in the `init_randomization()` function, which can be called by `__init__()`.
- Any random calls should use `self.random_state`, not `random()` or `np.random()`, as this will ensure consistent seeded results during experiments.
- After your agent is complete, you should define a unit test to ensure future revisions do not alter behavior. See rlpy/tests for some examples.

REQUIRED Instance Variables

—

REQUIRED Functions

`learn()` - called on every timestep (see documentation)

Note: The Agent *MUST* call the (inherited) `episodeTerminated()` function after learning if the transition led to a terminal state (ie, `learn()` will return `isTerminal=True`)

Note: The `learn()` function *MUST* call the `pre_discover()` function at its beginning, and `post_discover()` at its end. This allows adaptive representations to add new features (no effect on fixed ones).

2.8.2 Additional Information

- As always, the agent can log messages using `self.logger.info(<str>)`, see the Python `logger` documentation
- You should log values assigned to custom parameters when `__init__()` is called.
- See `Agent` for functions provided by the superclass.

2.8.3 Example: Creating the `SARSA0` Agent

In this example, we will create the standard SARSA learning agent (without eligibility traces (ie the λ parameter= 0 always)). This algorithm first computes the Temporal Difference Error, essentially the difference between the prediction under the current value function and what was actually observed (see e.g. Sutton and Barto's *Reinforcement Learning* (1998) or Wikipedia). It then updates the representation by summing the current function with this TD error, weighted by a factor called the *learning rate*.

1. Create a new file in the current working directory, `SARSA0.py`. Add the header block at the top:

```
__copyright__ = "Copyright 2013, RLPy http://www.acl.mit.edu/RLPy"
__credits__ = ["Alborz Geramifard", "Robert H. Klein", "Christoph Dann",
               "William Dabney", "Jonathan P. How"]
__license__ = "BSD 3-Clause"
__author__ = "Ray N. Forcement"

from rlpy.Agents.Agent import Agent, DescentAlgorithm
import numpy
```

2. Declare the class, create needed members variables (here a learning rate), described above) and write a docstring description:

```
class SARSA0(DescentAlgorithm, Agent):
    """
    Standard SARSA algorithm without eligibility trace (ie lambda=0)
    """
```

3. Copy the __init__ declaration from `Agent` and `DescentAlgorithm` in `Agent.py`, and add needed parameters (here the initial_learn_rate) and log them. (kwargs is a catch-all for initialization parameters.) Then call the superclass constructor:

```
    def __init__(self, policy, representation, discount_factor, initial_learn_rate=0.1, **kwargs):
        super(SARSA0,self).__init__(policy=policy,
         representation=representation, discount_factor=discount_factor, **kwargs)
        self.logger.info("Initial learning rate:\t\t%0.2f" % initial_learn_rate)
```

4. Copy the learn() declaration and implement accordingly. Here, compute the td-error, and use it to update the value function estimate (by adjusting feature weights):

```
    def learn(self, s, p_actions, a, r, ns, np_actions, na, terminal):

        # The previous state could never be terminal
        # (otherwise the episode would have already terminated)
        prevStateTerminal = False

        # MUST call this at start of learn()
        self.representation.pre_discover(s, prevStateTerminal, a, ns, terminal)

        # Compute feature function values and next action to be taken

        discount_factor = self.discount_factor # 'gamma' in literature
        feat_weights    = self.representation.weight_vec # Value function, expressed as feature weig
        features_s      = self.representation.phi(s, prevStateTerminal) # active feats in state
        features        = self.representation.phi_sa(s, prevStateTerminal, a, features_s) # active f
        features_prime_s= self.representation.phi(ns, terminal)
        features_prime  = self.representation.phi_sa(ns, terminal, na, features_prime_s)
        nnz             = count_nonzero(features_s)  # Number of non-zero elements

        # Compute td-error
        td_error            = r + np.dot(discount_factor * features_prime - features, feat_weights)

        # Update value function (or if TD-learning diverges, take no action)
        if nnz > 0:
            feat_weights_old = feat_weights.copy()
            feat_weights               += self.learn_rate * td_error
            if not np.all(np.isfinite(feat_weights)):
                feat_weights = feat_weights_old
                print "WARNING: TD-Learning diverged, theta reached infinity!"

        # MUST call this at end of learn() - add new features to representation as required.
        expanded = self.representation.post_discover(s, False, a, td_error, features_s)

        # MUST call this at end of learn() - handle episode termination cleanup as required.
        if terminal:
            self.episodeTerminated()
```

Note: You can and should define helper functions in your agents as needed, and arrange class hierarchy. (See eg TDControlAgent.py)

That's it! Now test the agent by creating a simple settings file on the domain of your choice. An example experiment is given below:

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

```
Agent Tutorial for RLPy
================================

Assumes you have created the SARSA0.py agent according to the tutorial and
placed it in the current working directory.
Tests the agent on the GridWorld domain.
"""
__author__ = "Robert H. Klein"
from rlpy.Domains import GridWorld
from SARSA0 import SARSA0
from rlpy.Representations import Tabular
from rlpy.Policies import eGreedy
from rlpy.Experiments import Experiment
import os


def make_experiment(exp_id=1, path="./Results/Tutorial/gridworld-sarsa0"):
    """
    Each file specifying an experimental setup should contain a
    make_experiment function which returns an instance of the Experiment
    class with everything set up.

    @param id: number used to seed the random number generators
    @param path: output directory where logs and results are stored
    """
    opt = {}
    opt["exp_id"] = exp_id
    opt["path"] = path

    ## Domain:
    maze = '4x5.txt'
    domain = GridWorld(maze, noise=0.3)
    opt["domain"] = domain

    ## Representation
    # discretization only needed for continuous state spaces, discarded otherwise
    representation  = Tabular(domain, discretization=20)

    ## Policy
    policy = eGreedy(representation, epsilon=0.2)

    ## Agent
    opt["agent"] = SARSA0(representation=representation, policy=policy,
                    discount_factor=domain.discount_factor,
                        initial_learn_rate=0.1)
    opt["checks_per_policy"] = 100
    opt["max_steps"] = 2000
    opt["num_policy_checks"] = 10
    experiment = Experiment(**opt)
    return experiment

if __name__ == '__main__':
    experiment = make_experiment(1)
    experiment.run(visualize_steps=False,  # should each learning step be shown?
                    visualize_learning=True,  # show policy / value function?
                    visualize_performance=1)  # show performance runs?
    experiment.plot()
    experiment.save()
```

2.8.4 What to do next?

In this Agent tutorial, we have seen how to

- Write a learning agent that inherits from the RLPy base `Agent` class
- Add the agent to RLPy and test it

Adding your component to RLPy

If you would like to add your component to RLPy, we recommend developing on the development version (see *Development Version*). Please use the following header at the top of each file:

```
__copyright__ = "Copyright 2013, RLPy http://www.acl.mit.edu/RLPy"
__credits__ = ["Alborz Geramifard", "Robert H. Klein", "Christoph Dann",
               "William Dabney", "Jonathan P. How"]
__license__ = "BSD 3-Clause"
__author__ = "Tim Beaver"
```

- Fill in the appropriate `__author__` name and `__credits__` as needed. Note that RLPy requires the BSD 3-Clause license.
- If you installed RLPy in a writeable directory, the className of the new agent can be added to the `__init__.py` file in the `Agents/` directory. (This allows other files to import the new agent).
- If available, please include a link or reference to the publication associated with this implementation (and note differences, if any).

If you would like to add your new agent to the RLPy project, we recommend you branch the project and create a pull request to the RLPy repository.

You can also email the community list `rlpy@mit.edu` for comments or questions. To subscribe click here.

2.9 Creating a New Representation

This tutorial describes the standard RLPy `Representation` interface, and illustrates a brief example of creating a new value function representation.

The Representation is the approximation of the value function associated with a `Domain`, usually in some lower-dimensional feature space.

The Agent receives observations from the Domain on each step and calls its `learn()` function, which is responsible for updating the Representation accordingly. Agents can later query the Representation for the value of being in a state *V(s)* or the value of taking an action in a particular state (known as the Q-function, *Q(s,a)*).

Note: At present, it is assumed that the Linear Function approximator family of representations is being used.

Note: You may want to review the namespace / inheritance / scoping rules in Python.

2.9.1 Requirements

- Each Representation must be a subclass of `Representation` and call the `__init__()` function of the Representation superclass.

- Accordingly, each Representation must be instantiated with and a Domain in the `__init__()` function. Note that an optional `discretization` parameter may be used by discrete Representations attempting to represent a value function over a continuous space. It is ignored for discrete dimensions.
- Any randomization that occurs at object construction *MUST* occur in the `init_randomization()` function, which can be called by `__init__()`.
- Any random calls should use `self.random_state`, not `random()` or `np.random()`, as this will ensure consistent seeded results during experiments.
- After your Representation is complete, you should define a unit test to ensure future revisions do not alter behavior. See rlpy/tests/test_representations for some examples.

REQUIRED Instance Variables

The new Representation *MUST* set the variables *BEFORE* calling the superclass `__init__()` function:

1. `self.isDynamic` - bool: True if this Representation can add or remove features during execution
2. `self.features_num` - int: The (initial) number of features in the representation

REQUIRED Functions

The new Representation *MUST* define two functions:

1. `phi_nonTerminal()`, (see linked documentation), which returns a vector of feature function values associated with a particular state.
2. `featureType()`, (see linked documentation), which returns the data type of the underlying feature functions (eg "float" or "bool").

SPECIAL Functions

Representations whose feature functions may change over the course of execution (termed **adaptive** or **dynamic** Representations) should override one or both functions below as needed. Note that `self.isDynamic` should = `True`.

1. `pre_discover()`
2. `post_discover()`

2.9.2 Additional Information

- As always, the Representation can log messages using `self.logger.info(<str>)`, see Python `logger` doc.
- You should log values assigned to custom parameters when `__init__()` is called.
- See `Representation` for functions provided by the superclass, especially before defining helper functions which might be redundant.

2.9.3 Example: Creating the `IncrementalTabular` Representation

In this example we will recreate the simple `IncrementalTabular` Representation, which merely creates a binary feature function $f_d()$ that is associated with each discrete state `d` we have encountered so far. $f_d(s) = 1$ when $d{=}s$, 0 elsewhere, ie, the vector of feature functions evaluated at s will have all zero elements except one. Note that this is identical to the `Tabular` Representation, except that feature functions are only created as needed, not instantiated for every single state at the outset. Though simple, neither the `Tabular` nor `IncrementalTabular` representations generalize to nearby states in the domain, and can be intractable to use on large domains (as there are as many feature functions as there are states in the entire space). Continuous dimensions of `s` (assumed to be bounded in this Representation) are discretized.

1. Create a new file in your current working directory, `IncrTabularTut.py`. Add the header block at the top:

```
__copyright__ = "Copyright 2013, RLPy http://www.acl.mit.edu/RLPy"
__credits__ = ["Alborz Geramifard", "Robert H. Klein", "Christoph Dann",
               "William Dabney", "Jonathan P. How"]
__license__ = "BSD 3-Clause"
__author__ = "Ray N. Forcement"

from rlpy.Representations.Representation import Representation
import numpy as np
from copy import deepcopy
```

2. Declare the class, create needed members variables (here an optional hash table to lookup feature function values previously computed), and write a docstring description:

```
class IncrTabularTut(Representation):
    """
    Tutorial representation: identical to IncrementalTabular

    """
    hash = None
```

3. Copy the __init__ declaration from `Representation.py`, add needed parameters (here none), and log them. Assign self.features_num and self.isDynamic, then call the superclass constructor:

```
def __init__(self, domain, discretization=20):
    self.hash           = {}
    self.features_num   = 0
    self.isDynamic      = True
    super(IncrTabularTut, self).__init__(domain, discretization)
```

4. Copy the `phi_nonTerminal()` function declaration and implement it accordingly to return the vector of feature function values for a given state. Here, lookup feature function values using self.hashState(s) provided by the parent class. Note here that self.hash should always contain hash_id if `pre_discover()` is called as required:

```
def phi_nonTerminal(self, s):
    hash_id = self.hashState(s)
    id  = self.hash.get(hash_id)
    F_s = np.zeros(self.features_num, bool)
    if id is not None:
        F_s[id] = 1
    return F_s
```

5. Copy the `featureType()` function declaration and implement it accordingly to return the datatype returned by each feature function. Here, feature functions are binary, so the datatype is boolean:

```
    def featureType(self):
        return bool
```

6. Override parent functions as necessary; here we require a `pre_discover()` function to populate the hash table for each new encountered state:

```
    def pre_discover(self, s, terminal, a, sn, terminaln):
        return self._add_state(s) + self._add_state(sn)
```

7. Finally, define any needed helper functions:

```
    def _add_state(self, s):
        hash_id = self.hashState(s)
        id  = self.hash.get(hash_id)
        if id is None:
            #New State
            self.features_num += 1
            #New id = feature_num - 1
            id = self.features_num - 1
            self.hash[hash_id] = id
            #Add a new element to the feature weight vector
            self.addNewWeight()
            return 1
        return 0

    def __deepcopy__(self, memo):
        new_copy = IncrementalTabular(self.domain, self.discretization)
        new_copy.hash = deepcopy(self.hash)
        return new_copy
```

That's it! Now test your Representation by creating a simple settings file on the domain of your choice. An example experiment is given below:

```
#!/usr/bin/env python
"""
Representation Tutorial for RLPy
================================

Assumes you have created the IncrTabularTut.py agent according to the tutorial and
placed it in the current working directory.
Tests the Representation on the GridWorld domain usin SARSA
"""
__author__ = "Robert H. Klein"
from rlpy.Domains import GridWorld
from rlpy.Agents import SARSA
from IncrTabularTut import IncrTabularTut
from rlpy.Policies import eGreedy
from rlpy.Experiments import Experiment
import os


def make_experiment(exp_id=1, path="./Results/Tutorial/gridworld-IncrTabularTut"):
    """
    Each file specifying an experimental setup should contain a
    make_experiment function which returns an instance of the Experiment
    class with everything set up.

    @param id: number used to seed the random number generators
    @param path: output directory where logs and results are stored
```

```
    """
    opt = {}
    opt["exp_id"] = exp_id
    opt["path"] = path

    ## Domain:
    maze = '4x5.txt'
    domain = GridWorld(maze, noise=0.3)
    opt["domain"] = domain

    ## Representation
    # discretization only needed for continuous state spaces, discarded otherwise
    representation  = IncrTabularTut(domain)

    ## Policy
    policy = eGreedy(representation, epsilon=0.2)

    ## Agent
    opt["agent"] = SARSA(representation=representation, policy=policy,
                   discount_factor=domain.discount_factor,
                        initial_learn_rate=0.1)
    opt["checks_per_policy"] = 100
    opt["max_steps"] = 2000
    opt["num_policy_checks"] = 10
    experiment = Experiment(**opt)
    return experiment

if __name__ == '__main__':
    experiment = make_experiment(1)
    experiment.run(visualize_steps=False,  # should each learning step be shown?
                   visualize_learning=True,  # show policy / value function?
                   visualize_performance=1)  # show performance runs?
    experiment.plot()
    experiment.save()
```

2.9.4 What to do next?

In this Representation tutorial, we have seen how to

- Write an adaptive Representation that inherits from the RLPy base `Representation` class
- Add the Representation to RLPy and test it

Adding your component to RLPy

If you would like to add your component to RLPy, we recommend developing on the development version (see *Development Version*). Please use the following header at the top of each file:

```
__copyright__ = "Copyright 2013, RLPy http://www.acl.mit.edu/RLPy"
__credits__ = ["Alborz Geramifard", "Robert H. Klein", "Christoph Dann",
               "William Dabney", "Jonathan P. How"]
__license__ = "BSD 3-Clause"
__author__ = "Tim Beaver"
```

- Fill in the appropriate `__author__` name and `__credits__` as needed. Note that RLPy requires the BSD 3-Clause license.

- If you installed RLPy in a writeable directory, the className of the new representation can be added to the `__init__.py` file in the `Representations/` directory. (This allows other files to import the new representation).
- If available, please include a link or reference to the publication associated with this implementation (and note differences, if any).

If you would like to add your new representation to the RLPy project, we recommend you branch the project and create a pull request to the RLPy repository.

You can also email the community list `rlpy@mit.edu` for comments or questions. To subscribe click here.

2.10 Creating a New Domain

This tutorial describes the standard RLPy `Domain` interface, and illustrates a brief example of creating a new problem domain.

The Domain controls the environment in which the `Agent` resides as well as the reward function the Agent is subject to.

The Agent interacts with the Domain in discrete timesteps called *episodes* (see `step()`). At each step, the Agent informs the Domain what indexed action it wants to perform. The Domain then calculates the effects this action has on the environment and updates its internal state accordingly. It also returns the new state (*ns*) to the agent, along with a reward/penalty, (*r*) and whether or not the episode is over (*terminal*), in which case the agent is reset to its initial state.

This process repeats until the Domain determines that the Agent has either completed its goal or failed. The `Experiment` controls this cycle.

Because Agents are designed to be agnostic to the Domain that they are acting within and the problem they are trying to solve, the Domain needs to completely describe everything related to the task. Therefore, the Domain must not only define the observations that the Agent receives, but also the states it can be in, the actions that it can perform, and the relationships between the three.

Warning: While each dimension of the state *s* is either *continuous* or *discrete*, discrete dimensions are assume to take nonnegative **integer** values (ie, the index of the discrete state).

Note: You may want to review the namespace / inheritance / scoping rules in Python.

2.10.1 Requirements

- Each Domain must be a subclass of `Domain` and call the `__init__()` function of the Domain superclass.
- Any randomization that occurs at object construction *MUST* occur in the `init_randomization()` function, which can be called by `__init__()`.
- Any random calls should use `self.random_state`, not `random()` or `np.random()`, as this will ensure consistent seeded results during experiments.
- After your agent is complete, you should define a unit test to ensure future revisions do not alter behavior. See rlpy/tests/test_domains for some examples.

REQUIRED Instance Variables

The new Domain *MUST* set these variables *BEFORE* calling the superclass `__init__()` function:

1. `self.statespace_limits` - Bounds on each dimension of the state space. Each row corresponds to one dimension and has two elements [min, max]. Used for discretization of continuous dimensions.
2. `self.continuous_dims` - array of integers; each element is the index (eg, row in `statespace_limits` above) of a continuous-valued dimension. This array is empty if all states are discrete.
3. `self.DimNames` - array of strings, a name corresponding to each dimension (eg one for each row in `statespace_limits` above)
4. `self.episodeCap` - integer, maximum number of steps before an episode terminated (even if not in a terminal state).
5. `actions_num` - integer, the total number of possible actions (ie, the size of the action space). This number **MUST** be a finite integer - continuous action spaces are not currently supported.
6. `discount_factor` - float, the discount factor (gamma in literature) by which rewards are reduced.

REQUIRED Functions

1. `s0()`, (see linked documentation), which returns a (possibly random) state in the domain, to be used at the start of an *episode*.
2. `step()`, (see linked documentation), which returns the tuple `(r,ns,terminal, pa)` that results from taking action *a* from the current state (internal to the Domain).
   - *r* is the reward obtained during the transition
   - *ns* is the new state after the transition
   - *terminal*, a boolean, is true if the new state *ns* is a terminal one to end the episode
   - *pa*, an array of possible actions to take from the new state *ns*.

SPECIAL Functions

In many cases, the Domain will also override the functions:

1. `isTerminal()` - returns a boolean whether or not the current (internal) state is terminal. Default is always return False.
2. `possibleActions()` - returns an array of possible action indices, which often depend on the current state. Default is to enumerate **every** possible action, regardless of current state.

OPTIONAL Functions

Optionally, define / override the following functions, used for visualization:

1. `showDomain()` - Visualization of domain based on current internal state and an action, *a*. Often the header will include an optional argument *s* to display instead of the current internal state. RLPy frequently uses matplotlib to accomplish this - see the example below.
2. `showLearning()` - Visualization of the "learning" obtained so far on this domain, usually a value function plot and policy plot. See the introductory tutorial for an example on `GridWorld`

XX expectedStep(), XX

2.10.2 Additional Information

- As always, the Domain can log messages using `self.logger.info(<str>)`, see Python `logger` doc.
- You should log values assigned to custom parameters when `__init__()` is called.
- See `Domain` for functions provided by the superclass, especially before defining helper functions which might be redundant.

2.10.3 Example: Creating the `ChainMDP` Domain

In this example we will recreate the simple `ChainMDP` Domain, which consists of *n* states that can only transition to *n-1* or *n+1*: `s0 <-> s1 <-> ... <-> sn` n The goal is to reach state `sn` from `s0`, after which the episode terminates. The agent can select from two actions: left [0] and right [1] (it never remains in same state). But the transitions are noisy, and the opposite of the desired action is taken instead with some probability. Note that the optimal policy is to always go right.

1. Create a new file in your current working directory, `ChainMDPTut.py`. Add the header block at the top:

```
__copyright__ = "Copyright 2013, RLPy http://www.acl.mit.edu/RLPy"
__credits__ = ["Alborz Geramifard", "Robert H. Klein", "Christoph Dann",
               "William Dabney", "Jonathan P. How"]
__license__ = "BSD 3-Clause"
__author__ = "Ray N. Forcement"

from rlpy.Tools import plt, mpatches, fromAtoB
from rlpy.Domains.Domain import Domain
import numpy as np
```

2. Declare the class, create needed members variables (here several objects to be used for visualization and a few domain reward parameters), and write a docstring description:

```
class ChainMDPTut(Domain):
    """
    Tutorial Domain - nearly identical to ChainMDP.py
    """
    #: Reward for each timestep spent in the goal region
    GOAL_REWARD = 0
    #: Reward for each timestep
    STEP_REWARD = -1
    #: Set by the domain = min(100,rows*cols)
    episodeCap = 0
    # Used for graphical normalization
    MAX_RETURN  = 1
    # Used for graphical normalization
    MIN_RETURN  = 0
    # Used for graphical shifting of arrows
    SHIFT       = .3
    #:Used for graphical radius of states
    RADIUS      = .5
    # Stores the graphical pathes for states so that we can later change their colors
    circles     = None
    #: Number of states in the chain
    chainSize   = 0
    # Y values used for drawing circles
    Y           = 1
```

3. Copy the __init__ declaration from `Domain.py`, add needed parameters (here the number of states in the chain, `chainSize`), and log them. Assign `self.statespace_limits`,

`self.episodeCap`, `self.continuous_dims`, `self.DimNames`, `self.actions_num`, and `self.discount_factor`. Then call the superclass constructor:

```
def __init__(self, chainSize=2):
    """
    :param chainSize: Number of states \'n\' in the chain.
    """
    self.chainSize          = chainSize
    self.start              = 0
    self.goal               = chainSize - 1
    self.statespace_limits  = np.array([[0,chainSize-1]])
    self.episodeCap         = 2*chainSize
    self.continuous_dims    = []
    self.DimNames           = ['State']
    self.actions_num        = 2
    self.discount_factor    = 0.9
    super(ChainMDPTut,self).__init__()
```

4. Copy the `step()` and function declaration and implement it accordingly to return the tuple (r,ns,isTerminal,possibleActions), and similarly for `s0()`. We want the agent to always start at state *[0]* to begin, and only achieves reward and terminates when *s = [n-1]*:

```
def step(self,a):
    s = self.state[0]
    if a == 0: #left
        ns = max(0,s-1)
    if a == 1: #right
        ns = min(self.chainSize-1,s+1)
    self.state = np.array([ns])

    terminal = self.isTerminal()
    r = self.GOAL_REWARD if terminal else self.STEP_REWARD
    return r, ns, terminal, self.possibleActions()

def s0(self):
    self.state = np.array([0])
    return self.state, self.isTerminal(), self.possibleActions()
```

5. In accordance with the above termination condition, override the `isTerminal()` function by copying its declaration from `Domain.py`:

```
def isTerminal(self):
    s = self.state
    return (s[0] == self.chainSize - 1)
```

6. For debugging convenience, demonstration, and entertainment, create a domain visualization by overriding the default (which is to do nothing). With matplotlib, generally this involves first performing a check to see if the figure object needs to be created (and adding objects accordingly), otherwise merely updating existing plot objects based on the current `self.state` and action *a*:

```
def showDomain(self, a = 0):
    #Draw the environment
    s = self.state
    s = s[0]
    if self.circles is None: # We need to draw the figure for the first time
       fig = plt.figure(1, (self.chainSize*2, 2))
       ax = fig.add_axes([0, 0, 1, 1], frameon=False, aspect=1.)
       ax.set_xlim(0, self.chainSize*2)
       ax.set_ylim(0, 2)
```

```
            ax.add_patch(mpatches.Circle((1+2*(self.chainSize-1), self.Y), self.RADIUS*1.1, fc="w"))
            ax.xaxis.set_visible(False)
            ax.yaxis.set_visible(False)
            self.circles = [mpatches.Circle((1+2*i, self.Y), self.RADIUS, fc="w") for i in np.arange(
            for i in np.arange(self.chainSize):
                ax.add_patch(self.circles[i])
                if i != self.chainSize-1:
                     fromAtoB(1+2*i+self.SHIFT,self.Y+self.SHIFT,1+2*(i+1)-self.SHIFT, self.Y+self.SH
                     if i != self.chainSize-2: fromAtoB(1+2*(i+1)-self.SHIFT,self.Y-self.SHIFT,1+2*i+
                fromAtoB(.75,self.Y-1.5*self.SHIFT,.75,self.Y+1.5*self.SHIFT,'r',connectionstyle='arc
                plt.show()

        [p.set_facecolor('w') for p in self.circles]
        self.circles[s].set_facecolor('k')
        plt.draw()
```

Note: When first creating a matplotlib figure, you must call pl.show(); when updating the figure on subsequent steps, use pl.draw().

That's it! Now test it by creating a simple settings file on the domain of your choice. An example experiment is given below:

```
#!/usr/bin/env python
"""
Domain Tutorial for RLPy
================================

Assumes you have created the ChainMDPTut.py domain according to the
tutorial and placed it in the current working directory.
Tests the agent using SARSA with a tabular representation.
"""
__author__ = "Robert H. Klein"
from rlpy.Agents import SARSA
from rlpy.Representations import Tabular
from rlpy.Policies import eGreedy
from rlpy.Experiments import Experiment
from ChainMDPTut import ChainMDPTut
import os
import logging


def make_experiment(exp_id=1, path="./Results/Tutorial/ChainMDPTut-SARSA"):
    """
    Each file specifying an experimental setup should contain a
    make_experiment function which returns an instance of the Experiment
    class with everything set up.

    @param id: number used to seed the random number generators
    @param path: output directory where logs and results are stored
    """
    opt = {}
    opt["exp_id"] = exp_id
    opt["path"] = path

    ## Domain:
    chainSize = 50
```

```
    domain = ChainMDPTut(chainSize=chainSize)
    opt["domain"] = domain

    ## Representation
    # discretization only needed for continuous state spaces, discarded otherwise
    representation  = Tabular(domain)

    ## Policy
    policy = eGreedy(representation, epsilon=0.2)

    ## Agent
    opt["agent"] = SARSA(policy=policy, representation=representation, discount_factor=domain.discou
    opt["checks_per_policy"] = 100
    opt["max_steps"] = 2000
    opt["num_policy_checks"] = 10
    experiment = Experiment(**opt)
    return experiment

if __name__ == '__main__':
    experiment = make_experiment(1)
    experiment.run(visualize_steps=False,  # should each learning step be shown?
                   visualize_learning=True,  # show policy / value function?
                   visualize_performance=1)  # show performance runs?
    experiment.plot()
    experiment.save()
```

2.10.4 What to do next?

In this Domain tutorial, we have seen how to

- Write a Domain that inherits from the RLPy base `Domain` class
- Override several base functions
- Create a visualization
- Add the Domain to RLPy and test it

Adding your component to RLPy

If you would like to add your component to RLPy, we recommend developing on the development version (see *Development Version*). Please use the following header template at the top of each file:

```
__copyright__ = "Copyright 2013, RLPy http://www.acl.mit.edu/RLPy"
__credits__ = ["Alborz Geramifard", "Robert H. Klein", "Christoph Dann",
               "William Dabney", "Jonathan P. How"]
__license__ = "BSD 3-Clause"
__author__ = "Tim Beaver"
```

Fill in the appropriate `__author__` name and `__credits__` as needed. Note that RLPy requires the BSD 3-Clause license.

- If you installed RLPy in a writeable directory, the className of the new domain can be added to the `__init__.py` file in the `Domains/` directory. (This allows other files to import the new domain).
- If available, please include a link or reference to the publication associated with this implementation (and note differences, if any).

If you would like to add your new domain to the RLPy project, we recommend you branch the project and create a pull request to the RLPy repository.

You can also email the community list `rlpy@mit.edu` for comments or questions. To subscribe click here.

2.11 Creating a New Policy

This tutorial describes the standard RLPy `Policy` interface, and illustrates a brief example of creating a new problem domain.

The Policy determines the discrete action that an `Agent` will take given its current value function `Representation`.

The Agent learns about the `Domain` as the two interact. At each step, the Agent passes information about its current state to the Policy; the Policy uses this to decide what discrete action the Agent should perform next (see `pi()`) n

Warning: While each dimension of the state *s* is either *continuous* or *discrete*, discrete dimensions are assume to take nonnegative **integer** values (ie, the index of the discrete state).

Note: You may want to review the namespace / inheritance / scoping rules in Python.

2.11.1 Requirements

- Each Policy must be a subclass of `Policy` and call the `__init__()` function of the Policy superclass.
- Any randomization that occurs at object construction *MUST* occur in the `init_randomization()` function, which can be called by `__init__()`.
- Any random calls should use `self.random_state`, not `random()` or `np.random()`, as this will ensure consistent seeded results during experiments.
- After your Policy is complete, you should define a unit test to ensure future revisions do not alter behavior. See rlpy/tests for some examples.

REQUIRED Instance Variables

—

REQUIRED Functions

1. `pi()` - accepts the current state *s*, whether or not *s* is *terminal*, and an array of possible actions indices *p_actions* and returns an action index for the Agent to take.

SPECIAL Functions

Policies which have an explicit exploratory component (eg epsilon-greedy) **MUST** override the functions below to prevent exploratory behavior when evaluating the policy (which would skew results)

1. `turnOffExploration()`
2. `turnOnExploration()`

2.11.2 Additional Information

- As always, the Policy can log messages using `self.logger.info(<str>)`, see Python `logger` documentation.
- You should log values assigned to custom parameters when `__init__()` is called.
- See `Policy` for functions provided by the superclass.

2.11.3 Example: Creating the `Epsilon-Greedy` Policy

In this example we will recreate the `eGreedy` Policy. From a given state, it selects the action with the highest expected value (greedy with respect to value function), but with some probability `epsilon`, takes a random action instead. This explicitly balances the exploration/exploitation tradeoff, and ensures that in the limit of infinite samples, the agent will have explored the entire domain.

1. Create a new file in your current working directory, `eGreedyTut.py`. Add the header block at the top:

```
__copyright__ = "Copyright 2013, RLPy http://www.acl.mit.edu/RLPy"
__credits__ = ["Alborz Geramifard", "Robert H. Klein", "Christoph Dann",
               "William Dabney", "Jonathan P. How"]
__license__ = "BSD 3-Clause"
__author__ = "Ray N. Forcement"

from rlpy.Policies.Policy import Policy
import numpy as np
```

2. Declare the class, create needed members variables, and write a docstring description. See the role of member variables in comments:

```
class eGreedyTut(Policy):
    """
    From the tutorial in policy creation.  Identical to eGreedy.py.
    """

    # Probability of selecting a random action instead of greedy
    epsilon          = None
    # Temporarily stores value of ``epsilon`` when exploration disabled
    old_epsilon      = None
    # bool, used to avoid random selection among actions with the same values
    forcedDeterministicAmongBestActions = None
```

3. Copy the `__init__()` declaration from `Policy.py` and add needed parameters. In the function body, assign them and log them. Then call the superclass constructor. Here the parameters are the probability of selecting a random action, `epsilon`, and how to handle the case where multiple best actions exist, ie with the same value, `forcedDeterministicAmongBestActions`:

```
def __init__(self,representation, epsilon = .1,
    forcedDeterministicAmongBestActions = False, seed=1):
     self.epsilon = epsilon
     self.forcedDeterministicAmongBestActions = forcedDeterministicAmongBestActions
     super(eGreedyTut,self).__init__(representation)
```

4. Copy the `pi()` declaration from `Policy.py` and implement it to return an action index for any given state and possible action inputs. Here, with probability epsilon, take a random action among the possible. Otherwise, pick an action with the highest expected value (depending on `self.forcedDeterministicAmongBestActions`, either pick randomly from among the best actions or always select the one with lowest index:

```
    def pi(self,s, terminal, p_actions):
        coin = self.random_state.rand()
        #print "coin=",coin
        if coin < self.epsilon:
            return self.random_state.choice(p_actions)
        else:
            b_actions = self.representation.bestActions(s, terminal, p_actions)
            if self.forcedDeterministicAmongBestActions:
                return b_actions[0]
            else:
                return self.random_state.choice(b_actions)
```

5. Because this policy has an exploratory component, we must override the `turnOffExploration()` and `turnOnExploration()` functions, so that when evaluating the policy's performance the exploratory component may be automatically disabled so as not to influence results:

```
    def turnOffExploration(self):
        self.old_epsilon = self.epsilon
        self.epsilon = 0
    def turnOnExploration(self):
        self.epsilon = self.old_epsilon
```

Warning: If you fail to define `turnOffExploration()` and `turnOnExploration()` for functions with exploratory components, measured algorithm performance will be worse, since exploratory actions by definition are suboptimal based on the current model.

That's it! Now test your new Policy by creating a simple settings file on the domain of your choice. An example experiment is given below:

```
#!/usr/bin/env python
"""
Policy Tutorial for RLPy
================================

Assumes you have created the eGreedyTut.py agent according to the tutorial and
placed it in the current working directory.
Tests the policy on the GridWorld domain, with the policy and value function
visualized.
"""
__author__ = "Robert H. Klein"
from rlpy.Domains import GridWorld
from rlpy.Agents import SARSA
from rlpy.Representations import Tabular
from eGreedyTut import eGreedyTut
from rlpy.Experiments import Experiment
import os


def make_experiment(exp_id=1, path="./Results/Tutorial/gridworld-eGreedyTut"):
    """
    Each file specifying an experimental setup should contain a
    make_experiment function which returns an instance of the Experiment
    class with everything set up.

    @param id: number used to seed the random number generators
    @param path: output directory where logs and results are stored
    """
    opt = {}
```

```
    opt["exp_id"] = exp_id
    opt["path"] = path

    ## Domain:
    maze = '4x5.txt'
    domain = GridWorld(maze, noise=0.3)
    opt["domain"] = domain

    ## Representation
    # discretization only needed for continuous state spaces, discarded otherwise
    representation  = Tabular(domain, discretization=20)

    ## Policy
    policy = eGreedyTut(representation, epsilon=0.2)

    ## Agent
    opt["agent"] = SARSA(representation=representation, policy=policy,
                   discount_factor=domain.discount_factor,
                        initial_learn_rate=0.1)
    opt["checks_per_policy"] = 100
    opt["max_steps"] = 2000
    opt["num_policy_checks"] = 10
    experiment = Experiment(**opt)
    return experiment

if __name__ == '__main__':
    experiment = make_experiment(1)
    experiment.run(visualize_steps=False,  # should each learning step be shown?
                   visualize_learning=True,  # show policy / value function?
                   visualize_performance=1)  # show performance runs?
    experiment.plot()
    experiment.save()
```

2.11.4 What to do next?

In this Policy tutorial, we have seen how to

- Write a Policy that inherits from the RLPy base `Policy` class
- Override several base functions, including those that manage exploration/exploitation
- Add the Policy to RLPy and test it

Adding your component to RLPy

If you would like to add your component to RLPy, we recommend developing on the development version (see *Development Version*). Please use the following header at the top of each file:

```
__copyright__ = "Copyright 2013, RLPy http://www.acl.mit.edu/RLPy"
__credits__ = ["Alborz Geramifard", "Robert H. Klein", "Christoph Dann",
               "William Dabney", "Jonathan P. How"]
__license__ = "BSD 3-Clause"
__author__ = "Tim Beaver"
```

- Fill in the appropriate `__author__` name and `__credits__` as needed. Note that RLPy requires the BSD 3-Clause license.

- If you installed RLPy in a writeable directory, the className of the new policy can be added to the `__init__.py` file in the `Policies/` directory. (This allows other files to import the new policy).
- If available, please include a link or reference to the publication associated with this implementation (and note differences, if any).

If you would like to add your new policy to the RLPy project, we recommend you branch the project and create a pull request to the RLPy repository.

You can also email the community list `rlpy@mit.edu` for comments or questions. To subscribe click here.

2.12 Creating a Unit Test

This tutorial briefly describes how to create a unit test for your new module.

2.13 Python Nose

RLPy uses nose to perform unit tests. The syntax is:

```
nosetests <directory or file to test>
```

If a directory is supplied, `nose` attempts to recursively locate all files that it thinks contain tests. Include the word **test** in your filename and `nose` will search the file for any methods that look like tests; ie, again, include **test** in your method name and nose will execute it. Note for example, the tabular domain can be tested by running:

```
nosetests rlpy/tests/test_representations/test_Tabular.py
```

And that all representations (that have tests defined) can be tested by running:

```
nosetests rlpy/tests/test_representations/
```

And that in fact all modules with tests can be tested by running:

```
nosetests rlpy/tests/
```

Warning: The last command may take several minutes to run.

2.14 Unit Test Guidelines

There are no technical requirements for the unit tests; they should ensure correct behavior, which is unique to each class.

In general, each method should start with *test* and contain a series of `assert` statements that verify correct behavior.

Test:

- preconditions
- postconditions
- consistency when using a seeded random number generator
- etc.

Pay special attention to corner cases, such as when a Representation has not yet received any data or when an episode is reset.

2.15 Example: Tabular

Open `rlpy/tests/test_representations/test_Tabular.py` Observe the filename includes the word *test*, as does each method name.

Many, many tests are possible, but the author identified the most pertinent ones as:

```
* Ensure appropriate number of cells are created
* Ensure the correct binary feature is activated for a particular state
* Ensure correct discretization in continuous state spaces
```

These have each been given a test as shown in the file.

The integrity of the module is always tested with a statement of the form `assert <module quantity> == <expected/known quantity>`.

For example, the code:

```
mapname=os.path.join(mapDir, "4x5.txt") # expect 4*5 = 20 states
domain = GridWorld(mapname=mapname)
rep = Tabular(domain, discretization=100)
assert rep.features_num == 20
rep = Tabular(domain, discretization=5)
assert rep.features_num == 20
```

creates a 4x5 GridWorld and ensures that Tabular creates the correct number of features (20), and additionally tests that this holds true even when bogus `discretization` parameters are passed in.

2.15.1 What to do next?

In this unit testing tutorial, we have seen how to

- Write a successful unit test
- Use `nosetests` to run the tests

You should write unit tests for any new modules you create. Feel free to modify / extend existing unit tests as well - there is always more to test!

2.16 Frequently Asked Questions (FAQ)

2.16.1 How do I use the framework?

You can have a look at *Getting Started* or the *examples* directory where you find many ready-to-run examples of reinforcement learning experiments.

2.16.2 What does each line of output mean?

See documentation in the Getting Started section of the *Getting Started*.

88825: E[0:01:23]-R[0:00:10]: Return=-1.00, Steps=56, Features = 174

Field	Meaning
88825	steps of learning
E[0:01:23]	Elapsed time (s)
R[0:00:10]	Remaining time (s)
Return=-1.00	Sum of rewards for the last episode
Steps=56	Number of steps for the last episode
Features = 174	Number of Features used for the last episode

2.16.3 My code is slow, how can I improve its speed?

You can use the `rlpy.Tools.run.run_profiled()` function which takes a *make_experiment* function and generates a pictorial profile of the resulting running time in pdf format (see api doc for details on where to find this files). Each node represents proportional time for finishing the function, proportional time spent within the function, and number of times it has been called. Nodes are color coded based on their time. You want to spend your time boosting the running time of nodes with the highest proportional time spent within them shown in parentheses. As an example you can look at `Profiling/Example.pdf`

2.16.4 My project does not work. Do I need to install packages?

Please see the *Install page*.

2.16.5 I used to plot my figures based on number of episodes, why do you prefer steps?

The use of episode numbers does not provide accurate plots as the number of samples can vary within each episode. The use of steps gurantees that all methods saw exactly the same amount of data before being tested.

2.17 The RLPy API

2.17.1 Base Classes

Experiment

Agent

Policy

Representation

MDP Solvers

Domain

2.17.2 Domains

Acrobot with Euler Integration

Acrobot with Runge-Kutta Integration

Bicycle Balancing

BlocksWorld

Linear Chain MDP

Fifty-State Chain MDP

FlipBoard

GridWorld

HIV Treatment

Helicopter Hovering

Intruder Monitoring

Mountain Car

Persistent Search and Track Mission

Pacman

Pinball

PuddleWorld

RCCar

System Administrator

Swimmer

Finite Track CartPole: Balance Task

CHAPTER 3

Indices and tables

- genindex
- search