MMLF

Maja Machine Learning Framework

Release 1.0

Jan Hendrik Metzen, Mark Edgington

September 23, 2011

CONTENTS

1 Tutorials **3**
1.1 Installation 3
1.2 Quick start (command line interface) 4
1.3 Quick start (graphical user interface) 7
1.4 Writing an agent 14
1.5 Writing an environment 21

2 Learn more about... **27**
2.1 Existing Agents and Environments 27
2.2 Experiments 29
2.3 Logging 35
2.4 State and Action Spaces 39
2.5 Viewers 41

3 API-documentation **49**
3.1 MMLF package interface 49
3.2 Agents 51
3.3 Environments 58
3.4 Observables 66
3.5 Resources 68
3.6 Framework 88

4 Indices and Tables **97**

Python Module Index **99**

Index **101**

The Maja Machine Learning Framework (MMLF) (download here) is a general framework for problems in the domain of Reinforcement Learning (RL) written in python. It provides a set of RL related algorithms and a set of benchmark domains. Furthermore it is easily extensible and allows to automate benchmarking of different agents. Among the RL algorithms are TD(lambda), CMA-ES, Fitted R-Max, Monte-Carlo learning, the DYNA-TD and the actor-critic architecture. MMLF contains different variants of the maze-world and pole-balancing problem class as well as the mountain-car testbed and the pinball maze domain.

A certain scenario which is studied is called a "world". An example of such a scenario would be a robot that tries to find its way through a maze. In RL, the world is typically decomposed into the "agent(s)" and the "environment". In the example, the robot would be the agent and the maze would be the environment. The MMLF adopts this view since it provides a natural modularization, which allows to write general agents capable of learning and to test them in a multitude of environments. All learning (optimization of behavior) is usually done within an agent while simulation of physics and other kinds of dynamics are performed within an environment.

Agents and enviroments are only loosely coupled. In order to allow agents and environments to interact, the MMLF uses an "interaction server" which controls how the information between these two types of entities are exchanged. For example, an environment might inform an agent about its current state and which actions the agent can perform, while the agent might inform the environment which action he actually performed. This communication via the interaction server is completely transparent for agents and environments, the only constraint they have to fulfill is that they must offer certain methods which are called from the interaction server.

Download MMLF

MMLF is available at http://sourceforge.net/projects/mmlf/

Getting support

Contact the MMLF support mailing list.

Documentation

CHAPTER

ONE

TUTORIALS

1.1 Installation

For installing of the most recent version of the MMLF, please download the tar.gz file from https://sourceforge.net/projects/mmlf/. With this file, the MMLF can either be installed locally or globally (see below).

The MMLF has been tested on Python v2.6 and v2.7 and requires that the python packages numpy (tested on version 1.3.0), scipy (v0.7.0), matplotlib (v1.0), and pyyaml (v3.0.9) are installed. If you want to use the graphical user interface, the PyQt4 (v4.7.2) package is required additional. Specific configurations of agents may require additional packages (e.g., scikits.ann and lwpr); however, most of the MMLF should be usable when the core dependencies are fulfilled. The easiest way to install the dependencies depends on your operating system. Under Windows you may use the python(x,y) distribution. Under Mac OSX, you may use MacPorts, and under Linux, your packaging tool will most likely ship these packages out-of-the-box.

1.1.1 Local Installation

This kind of installation is most useful for developers that don't have access to the MMLF revision control system. Installing MMLF locally means to simply extract the tar.gz archive. The resulting directory consists of the mmlf package, this documentation, the configuration directory, and two python-scripts: `run_mmlf` (for running MMLF in a non-graphical way) and `mmlf_gui` (for starting a GUI which allows to configure and run the MMLF). The additional script setup.py is required for installing the MMLF globally. Running the MMLF via the `run_mlf` and `mmlf_gui` scripts is possible without requiring any further adjustments (see *Quick Start Tutorial*). If the MMLF should be made available as a module to other python code, please add the directory into which you extracted the MMLF to your pythonpath. It will then be possible to use `import mmlf` in a python script or interactive interpreter session. Any changes made in the code in the mmlf directory are directly active (changes to config files should be made in the `$HOME/.mmlf` directory however – see the note below).

1.1.2 Global Installation

This kind of installation is most useful for people who don't want to modify or extend the MMLF, but only wish to use it. A global installation allows the MMLF to be installed in a way such that all users of a system can use it. In order to install globally, extract the tar.gz archive and run `python setup.py install`. This command may require superuser privileges, and uses the Python Distribution Utilities (distutils) to install the MMLF into the site-packages/dist-packages directory of your python distribution. You may have to install setuptools before. After installing, you can run the mmlf via the `run_mmlf` (for running MMLF in a non-graphical way) and `mmlf_gui` (for starting a GUI which allows to configure and run the MMLF) scripts. See *Quick Start Tutorial* for more details. Furthermore, you can now also import and use the `mmlf` module in your own python scripts by `import mmlf` (see *MMLF python package interface*).

Note: Under Windows, you have to call `python mmlf_gui` in the `Scripts` subdirectory of your python directory (e.g. `C:\Python26\Scripts`)

Note: Performing a global installation copies MMLF config files into `/etc/mmlf` (unix-like OSes) or `$APPDATA/mmlf` (Windows). However, the data in these directories only defines some default world setups, and **is not meant for modification**. If you wish to modify a world's setup, modify the configuration files in the `$HOME/.mmlf` directory that is created after the first start of the MMLF. For more information on that, see the *Quick Start Tutorial*

See Also:

Tutorial ***Quick start (command line interface)*** Learn how to use the MMLF with a non-grapical user interface

Tutorial ***Quick start (graphical user interface)*** Learn how to use the MMLF's graphical user interface

1.2 Quick start (command line interface)

This tutorial explains how you can let an agent learn in a certain environment using the command line interface. It assumes that you already *installed* the MMLF successfully.

Lets assume you just want to test the TD Lambda agent in the Mountain Car environment. Starting this is essentially a one-liner at the command line:

```
run_mmlf --config mountain_car/world_td_lambda_exploration.yaml
```

or for unix users with a local MMLF installation:

```
./run_mmlf --config mountain_car/world_td_lambda_exploration.yaml
```

This will start the MMLF and execute the world defined in the `world_td_lambda_exploration.yaml` file.

Note: If this is your very first run of the MMLF, the MMLF will create the so-called "rw-directory" for your user. This rw-directory is essentially the place where the MMLF stores the configurations of all worlds, the log files, etc. By default this directory is `$HOME/.mmlf` (under MS Windows, `$USERPROFILE\.` is used for the `$HOME` directory). If you want to change the configuration of a world, the rw-directory is the place to do it (**not** `/etc/mmlf`). If you want to use a different directory as the rw-directory, you can specify this with the option `--rwpath`. For instance, `run_mmlf --config mountain_car/world_td_lambda_exploration.yaml --rwpath /home/someuser/Temp/mmlfrw` would use the directory `/home/someuser/Temp/mmlfrw` as the rw-directory. Note that **the MMLF does not remember this path** – this directory must be specified every time the MMLF is invoked.

Once the MMLF rw-directory is created, the world will be started. Some information is printed out, such as information received by the agent about state and action space from the environment. Then, the agent starts to perform in the environment and the environment prints out some information about how the agent performs:

```
'2011-02-15 11:12:39,700 FrameworkLog         INFO     Using MMLF RW area /home/jmetzen/.mmlf
'2011-02-15 11:12:39,700 FrameworkLog         INFO     Loading world from config file mountain_car/wo
'2011-02-15 11:12:40,642 AgentLog             INFO     TDLambdaAgent got new state-space:
        StateSpace:
                position        : ContinuousDimension(LimitType: soft, Limits: (-1.200,0.600))
                velocity        : ContinuousDimension(LimitType: soft, Limits: (-0.070,0.070))
'2011-02-15 11:12:40,646 AgentLog             INFO     TDLambdaAgent got new action-space:
        ActionSpace:
                thrust          : DiscreteDimension(Values: ['left', 'right', 'none'])
'2011-02-15 11:12:59,130 EnvironmentLog       INFO     No goal reached but 500 steps expired!
'2011-02-15 11:13:13,181 EnvironmentLog       INFO     No goal reached but 500 steps expired!
'2011-02-15 11:13:22,174 EnvironmentLog       INFO     No goal reached but 500 steps expired!
'2011-02-15 11:13:28,678 EnvironmentLog       INFO     Goal reached after 202 steps!
'2011-02-15 11:13:28,797 EnvironmentLog       INFO     Goal reached after 6 steps!
'2011-02-15 11:13:31,143 EnvironmentLog       INFO     Goal reached after 137 steps!

....
```

This shows that the agent wasn't able to reach the goal during the first episodes but over time it finds its way to the goal more frequently and faster. You can observe the performance of the agent for some time and see if its performance improves. You can stop the world by pressing Ctrl-C.

Once you have stopped the learning, you can take a look in the MMLF RW area (the one created during your first run of the MMLF). There are now two subdirectories: config and logs. The logs directory contains information about the run you just conducted. Among other things, the length of the episodes is stored in a separated log file that can be used for later analysis of the agents performance. To learn more about this, you can take a look at the *Logging* page. In this tutorial, we only focus on the config directory.

The config directory contains configuration files for all worlds contained in the MMLF. Lets start with the world configuration file we just used to start our first MMLF run, which is located in `config/mountain_car`. The `world_td_lambda_exploration.yaml` file contains the following:

```
worldPackage : mountain_car
environment:
    moduleName : "mcar_env"
    configDict:
        maxStepsPerEpisode : 500
        accelerationFactor : 0.001
        maxGoalVelocity : 0.07
        positionNoise : 0.0
        velocityNoise : 0.0
agent:
    moduleName : "td_lambda_agent"
    configDict:
        update_rule: SARSA
        gamma : 1.0
        epsilon : 0.01
        lambda : 0.95
        minTraceValue : 0.5
        stateDimensionResolution : 9
        actionDimensionResolution : 7
        function_approximator :
            name : 'CMAC'
            number_of_tilings : 10
            learning_rate : 0.5
            update_rule : 'exaggerator'
            default : 0.0
monitor:
    policyLogFrequency : 10
```

This file specifies where the python-modules for the agent and the environment are located and what parameters to use for the agent and environment. Furthermore, it specifies which information a module called "Monitor" will store periodically in the log directory (see *Monitor* for more details on that). The config directory contains several world specification files, for instance `world_dps.yaml` in the `mountain_car` directory:

```
worldPackage : mountain_car
environment:
    moduleName : "mcar_env"
    configDict:
        maxStepsPerEpisode : 500
        accelerationFactor : 0.001
        maxGoalVelocity : 0.07
        positionNoise : 0.0
        velocityNoise : 0.0
agent:
    moduleName : "dps_agent"
    configDict:
        policy_search :
            method: 'fixed_parametrization'
            policy:
                type: 'linear'
                numOfDuplications: 1
                bias: True
            optimizer:
                name: 'evolution_strategy'
                sigma:  1.0
                populationSize : 5
                evalsPerIndividual: 10
                numChildren: 10
monitor:
    policyLogFrequency : 10
    functionOverStateSpaceLogging:
        active : True
        logFrequency : 250
        stateDims : None
        rasterPoints : 50
```

As you can see, the environments in the two configuration files are identical but the agents are different. This world can be started with a similar command as the first one, namely using

```
run_mmlf  --config mountain_car/world_dps.yaml
```

This will start a different learning agent (one using a Direct Policy Search algorithm for learning) and let it learn the mountain car task.

If you are more interested in further experiments with the td_lambda_agent, you simply modify and use the `world_td_lambda_exploration.yaml` file, editing the agent part of it. The interesting part of this configuration file is the agent's `configDict` dictionary, which contains the parameter values that are used by the agent. For instance, we see that the agent in `world_td_lambda_exploration.yaml` uses a discount factor gamma of 1.0 and follows an epsilon-greedy policy with epsilon=0.01 (for those are unfamiliar with the concepts behind how this agent works, check out the excellent (and **free**) book by Sutton and Barto). You can now modify the parameters and see how the learning performance is influenced. For instance, set epsilon to `0.0` to get an agent that always acts greedily and store the file as `world_td_lambda_no_exploration.yaml`. To start the world, simply run

```
run_mmlf  --config mountain_car/world_td_lambda_exploration.yaml
```

If you want to run a specific world only for a certain number of episodes (say 100), you can give an additional parameter at the command line:

```
run_mmlf  --config mountain_car/world_td_lambda_exploration.yaml --episodes 100
```

You can now use the basic features of the MMLF. Starting other worlds is done similarly, for instance

```
run_mmlf  --config single_pole_balancing/world_dps.yaml
```

will start the single-pole-balancing scenario with the DPS agent enabled.

See Also:

Tutorial ***Quick start (graphical user interface)*** Learn how to use the MMLF's graphical user interface

Learn more about ***Existing agents*** **and** ***Existing environments*** Get an overview over the agents and environments that are shipped with the MMLF

Tutorial ***Writing an agent*** Learn how to write your own MMLF agent

Tutorial ***Writing an environment*** Learn how to write your own MMLF environment

Learn more about ***Experiments*** Learn how to do a serious benchmarking and statistical comparison of the performance of different agents/environments

1.3 Quick start (graphical user interface)

This tutorial explains how you can use the MMLF's graphical user interface. It assumes that you already *installed* the MMLF successfully. It might be helpful to read the *Quick start (command line interface)* tutorial first in order to have some understanding of what is going on "under the hood".

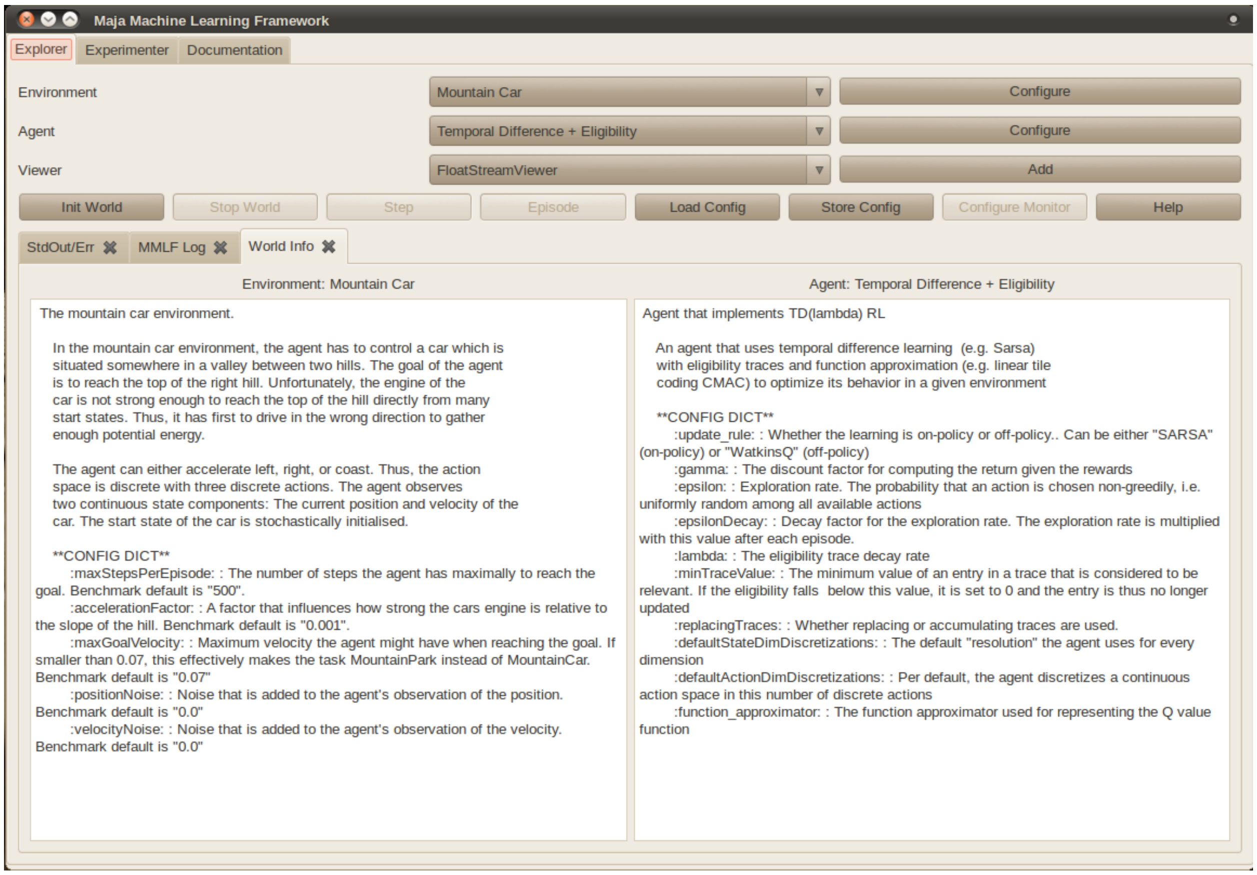
The MMLF's GUI can be started from the command line with the command

```
mmlf_gui
```

or for unix-based local installations:

```
./mmlf_gui
```

This should create a window that looks like this:



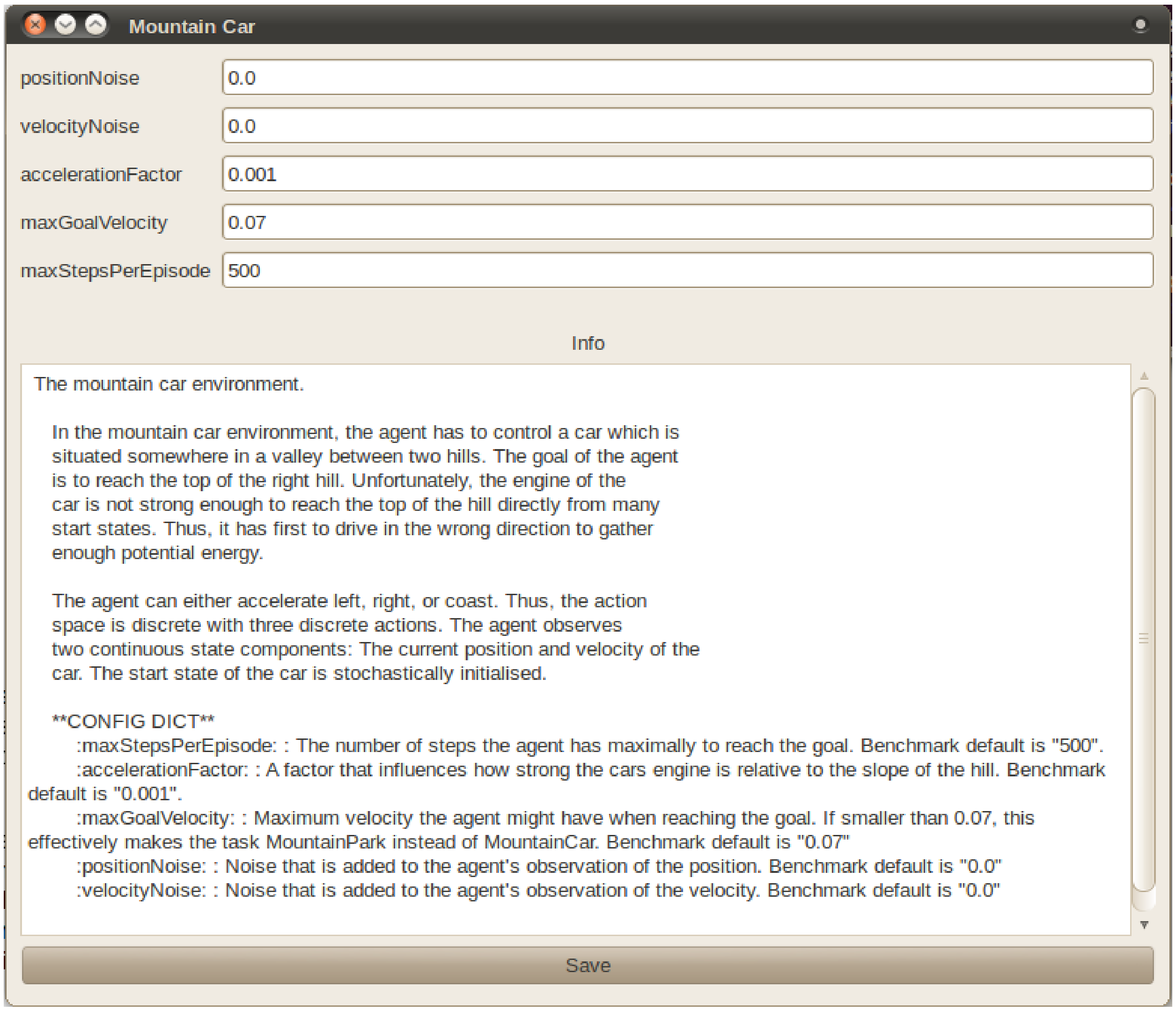
```
The mountain car environment.

    In the mountain car environment, the agent has to control a car which is
    situated somewhere in a valley between two hills. The goal of the agent
    is to reach the top of the right hill. Unfortunately, the engine of the
    car is not strong enough to reach the top of the hill directly from many
    start states. Thus, it has first to drive in the wrong direction to gather
    enough potential energy.

    The agent can either accelerate left, right, or coast. Thus, the action
    space is discrete with three discrete actions. The agent observes
    two continuous state components: The current position and velocity of the
    car. The start state of the car is stochastically initialised.

    **CONFIG DICT**
        :maxStepsPerEpisode: : The number of steps the agent has maximally to reach the
goal. Benchmark default is "500".
        :accelerationFactor: : A factor that influences how strong the cars engine is relative to
the slope of the hill. Benchmark default is "0.001".
        :maxGoalVelocity: : Maximum velocity the agent might have when reaching the goal. If
smaller than 0.07, this effectively makes the task MountainPark instead of MountainCar.
Benchmark default is "0.07"
        :positionNoise: : Noise that is added to the agent's observation of the position.
Benchmark default is "0.0"
        :velocityNoise: : Noise that is added to the agent's observation of the velocity.
Benchmark default is "0.0"
```

```
Agent that implements TD(lambda) RL

    An agent that uses temporal difference learning  (e.g. Sarsa)
    with eligibility traces and function approximation (e.g. linear tile
    coding CMAC) to optimize its behavior in a given environment

    **CONFIG DICT**
        :update_rule: : Whether the learning is on-policy or off-policy.. Can be either "SARSA"
(on-policy) or "WatkinsQ" (off-policy)
        :gamma: : The discount factor for computing the return given the rewards
        :epsilon: : Exploration rate. The probability that an action is chosen non-greedily, i.e.
uniformly random among all available actions
        :epsilonDecay: : Decay factor for the exploration rate. The exploration rate is multiplied
with this value after each episode.
        :lambda: : The eligibility trace decay rate
        :minTraceValue: : The minimum value of an entry in a trace that is considered to be
relevant. If the eligibility falls  below this value, it is set to 0 and the entry is thus no longer
updated
        :replacingTraces: : Whether replacing or accumulating traces are used.
        :defaultStateDimDiscretizations: : The default "resolution" the agent uses for every
dimension
        :defaultActionDimDiscretizations: : Per default, the agent discretizes a continuous
action space in this number of discrete actions
        :function_approximator: : The function approximator used for representing the Q value
function
```

The main window consists of three tabs: the "Explorer", the "Experimenter", and the "Documentation". The last tab displays the documentation you're now reading. The other two tabs will be explained in more detail in this tutorial.

1.3.1 Explorer

The explorer's main purpose is to investigate the behaviour of a specific agent in a specific environment. It provides different kinds of visualizations (depending on the world) of what is going on. In the explorer tab, the environment and the agent that should be used in the world can be selected from combo boxes. The selected agents and environments can be configured by pressing the configure button. This creates a popup window like the following:



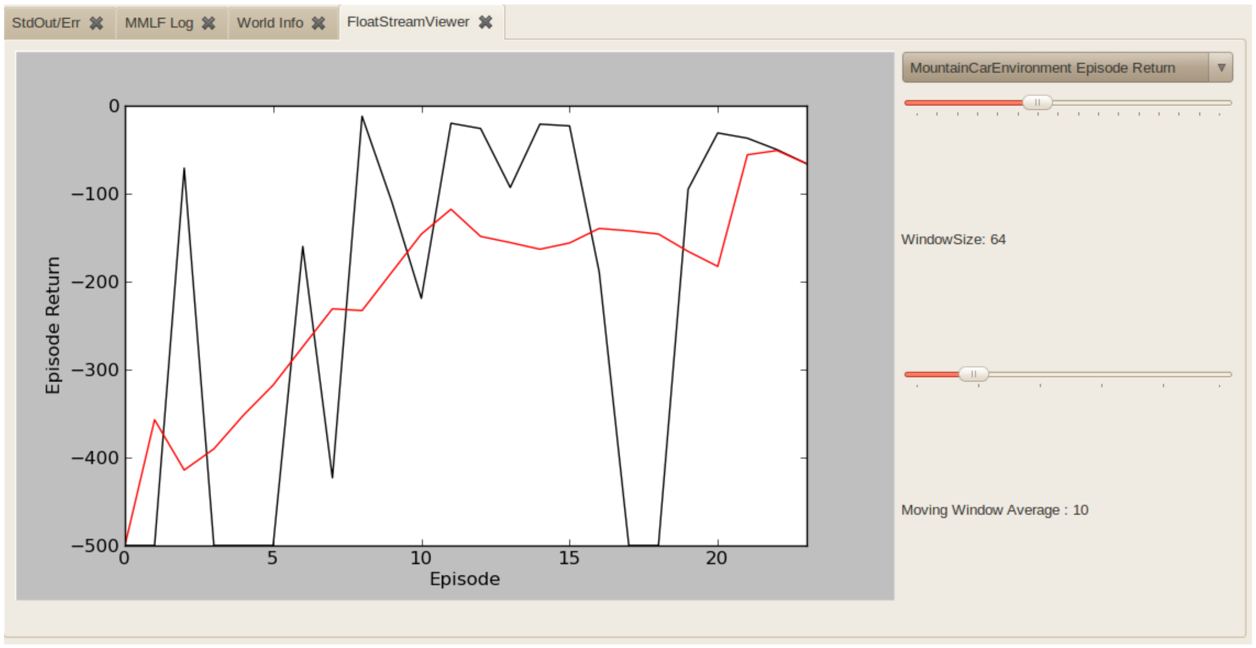
positionNoise	0.0
velocityNoise	0.0
accelerationFactor	0.001
maxGoalVelocity	0.07
maxStepsPerEpisode	500

```
The mountain car environment.

   In the mountain car environment, the agent has to control a car which is
   situated somewhere in a valley between two hills. The goal of the agent
   is to reach the top of the right hill. Unfortunately, the engine of the
   car is not strong enough to reach the top of the hill directly from many
   start states. Thus, it has first to drive in the wrong direction to gather
   enough potential energy.

   The agent can either accelerate left, right, or coast. Thus, the action
   space is discrete with three discrete actions. The agent observes
   two continuous state components: The current position and velocity of the
   car. The start state of the car is stochastically initialised.

   **CONFIG DICT**
       :maxStepsPerEpisode: : The number of steps the agent has maximally to reach the goal. Benchmark default is "500".
       :accelerationFactor: : A factor that influences how strong the cars engine is relative to the slope of the hill. Benchmark
default is "0.001".
       :maxGoalVelocity: : Maximum velocity the agent might have when reaching the goal. If smaller than 0.07, this
effectively makes the task MountainPark instead of MountainCar. Benchmark default is "0.07"
       :positionNoise: : Noise that is added to the agent's observation of the position. Benchmark default is "0.0"
       :velocityNoise: : Noise that is added to the agent's observation of the velocity. Benchmark default is "0.0"
```

In this popup, the agent's and environment's parameters can be modified and stored by pressing "Save". Furthermore, help on a specific parameter is provided as a tooltip of the respective edit field. An alternative to manually configuring agent and environment is to load a world with predefined agent and environment using the "Load Config" button. Accordingly, "Store Config" allows to store a manual configuration of a world such that it can be easily reloaded later on.

Back in the explorer tab, the selected agent and environment can be loaded by pressing "Init World". Now, the *Monitor* which controls the information that are automatically stored during running a world in the MMLF can be configured by pressing "Configure Monitor". Once this is done, we can start the configured world. One step in this world can be performed by pressing "Step", one single episode by pressing "Episode", and infinitely many by pressing "Start World". This indefinite run can be stopped by pressing "Pause World" and resumed by pressing "Resume World". By pressing "Stop World", the execution of the particular world is irrevocably terminated and a new world could be loaded using "Init World".

The tab "StdOut/Err" shows the output of the program to standard output and standard error. Some more detailed information about the currently running experiment can be obtained via the text output in the "MMLF Log" Tab. Additionally, the "World Info" tab shows some information about the currently selected agent and environment. Furthermore, so called *viewers* can be added that visualize the progress in a graphical manner. In all environments, the so called "FloatStreamViewer" is available that allows to monitor the development of a real-valued metric over time. This viewer looks like the following:

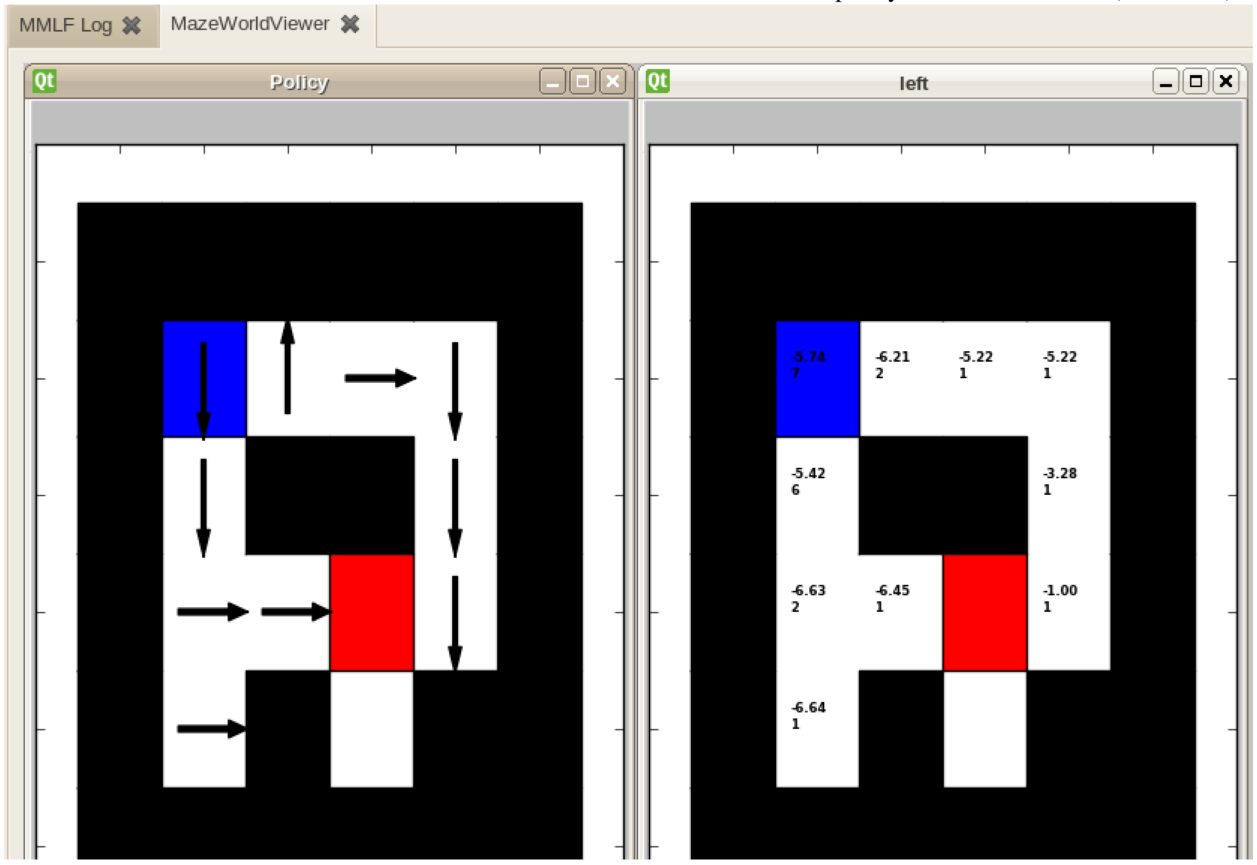


WindowSize: 64

Moving Window Average : 10

This viewer shows the change of the metric over time as well as its moving window average (in red). The metric can be selected via the combobox and the range of shown window as well as the length of the moving window average window can be specified.

For several worlds, additional viewers become available after loading a world using “Init World”. For instance, in the maze2d_environment, an additional viewer is available that shows the current policy and value function (see below):



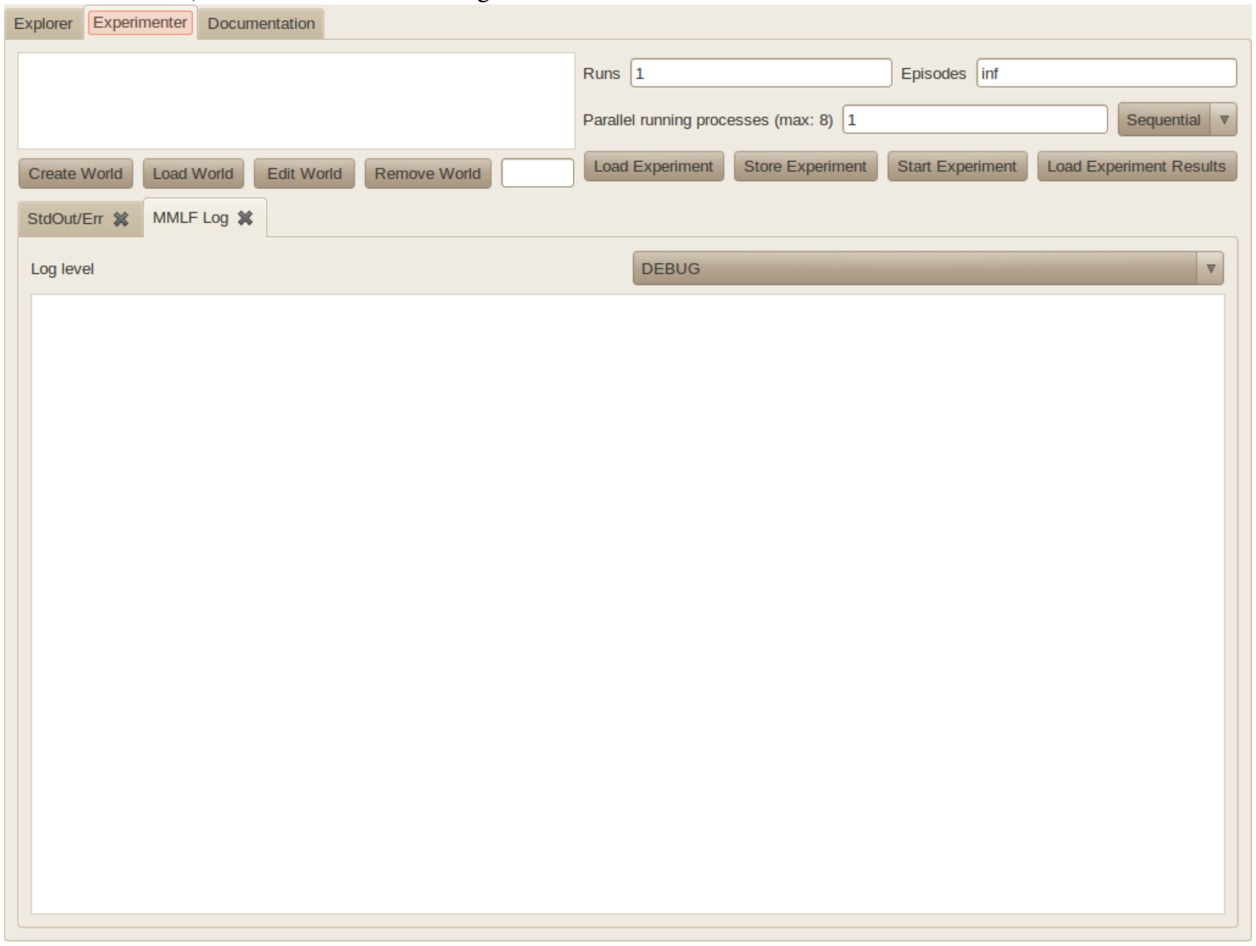
For an overview over all availabe viewers, please take a look at *Viewers*.

Note: Adding viewers might slow down the MMLF (the learning of the agents) considerably since updating the viewers with a high frequency might consume most of the CPU time. However, by closing the viewers, the MMLF should run with its former speed again. Thus, one can use viewers to introspect the current state of the world and close them after that.

A good way of getting to know the MMLF is to load different world configurations shipped with the MMLF (using "Load Config"), run these worlds, and visualize whats going on with different viewers.

1.3.2 Experimenter

The "Experimenter" is meant to be used when one wants to compare the learning performance in different settings (for example different agents/agent parametrizations in the same environment or the same agent in slightly different environments etc.). It looks like the following:



Runs 1 Episodes inf

Parallel running processes (max: 8) 1 Sequential

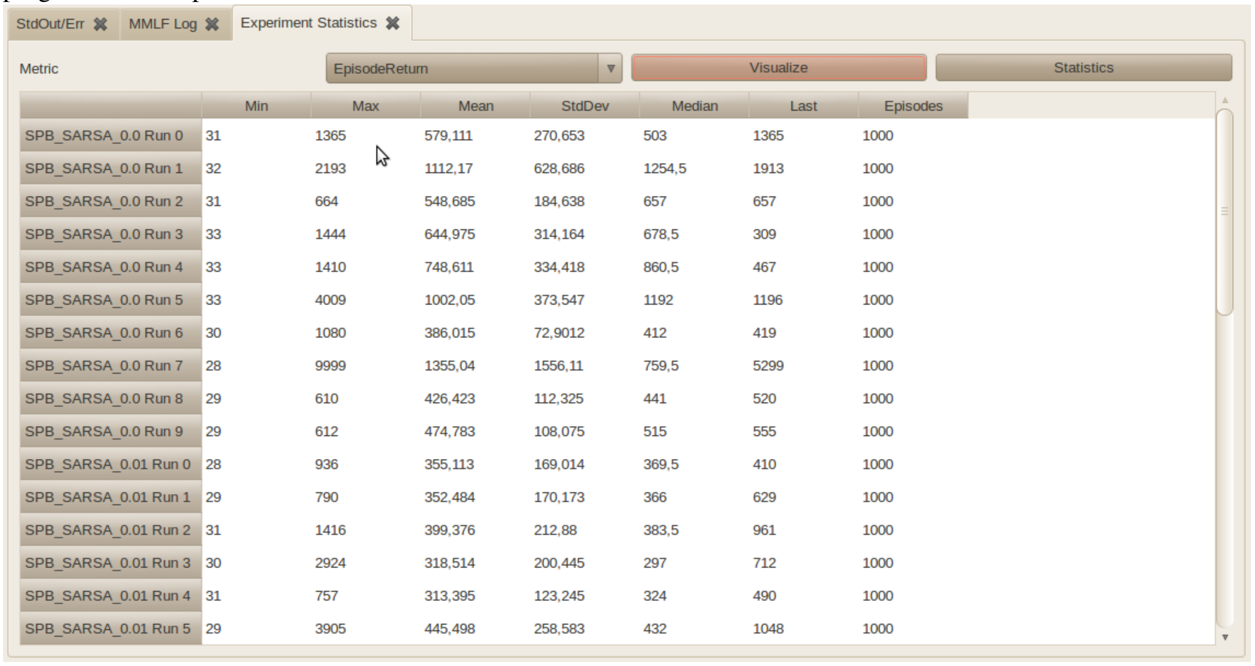
Log level DEBUG

The "Create World" button launches a window in which agent and environment of a world can be configured in the same way as in the Explorer. Once this is done, "Save" adds this world to the list of worlds in the upper left part of the Explorer tab. Alternatively, one can also load a stored world configuration using "Load World". An arbitrary world can be modified later on by selecting it in the world list and pressing "Edit World", can be removed from the world list by pressig "Remove world", and can get assigned a more meaningful name by selecting it and editing the name in the text field right of the "Remove world" button. Alternatively, instead of manually adding and editing worlds, one can also load a whole experiment configuration using "Load Experiment". Accordingly, configured experiments can be stored using "Store experiment".

Furthermore, one can specify how many independent runs of each world are conducted (the more often the more reliable become the performance estimates) by editing the entry in text field right of "Runs" and how many episodes each run should take (text field "Episodes"). In addition, one can select whether the independent runs of the world should be conducted sequentially (one after the other) or concurrently (in a separate OS-process each). By editing the text field "Parallel running processes", one can choose how many runs are conducted in parallel (this is fixed to 1 for sequential execution). The maximally allowed number of parallel processes is the number of (virtual) cores in the machine.

Note: A word of warning: the concurrent execution of world processes is still in an experimental state and may behave strangely under certain conditions (for instance it might not shutdown correctly and keep some zombie processes). Furthermore, Windows OSes do not support concurrent execution of worlds.

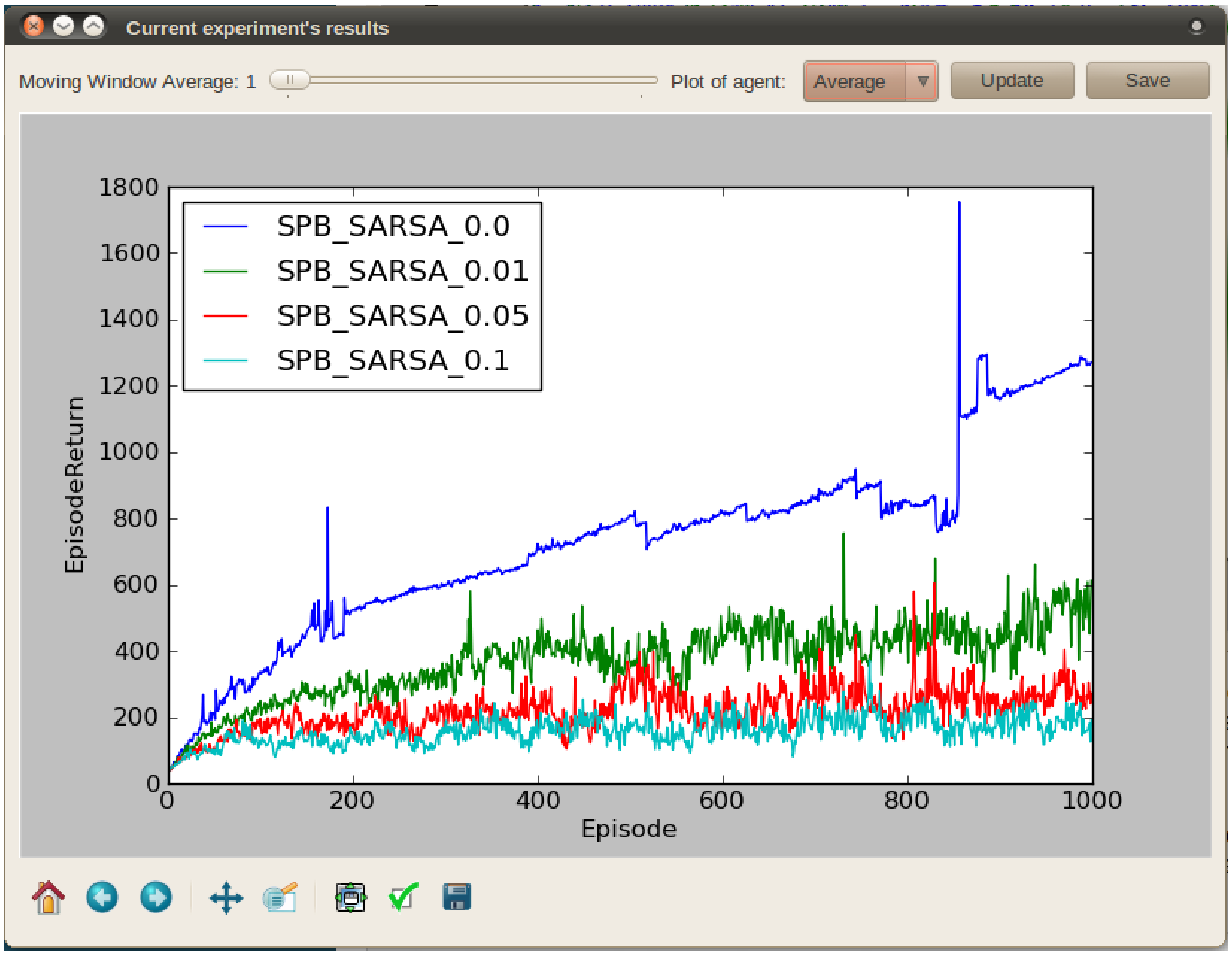
By pressing "Start Experiment", the experiment is started and a new tab "Experiment Statistics" is added in which the progress of the experiment can be monitored in real time:



Metric: EpisodeReturn

	Min	Max	Mean	StdDev	Median	Last	Episodes
SPB_SARSA_0.0 Run 0	31	1365	579,111	270,653	503	1365	1000
SPB_SARSA_0.0 Run 1	32	2193	1112,17	628,686	1254,5	1913	1000
SPB_SARSA_0.0 Run 2	31	664	548,685	184,638	657	657	1000
SPB_SARSA_0.0 Run 3	33	1444	644,975	314,164	678,5	309	1000
SPB_SARSA_0.0 Run 4	33	1410	748,611	334,418	860,5	467	1000
SPB_SARSA_0.0 Run 5	33	4009	1002,05	373,547	1192	1196	1000
SPB_SARSA_0.0 Run 6	30	1080	386,015	72,9012	412	419	1000
SPB_SARSA_0.0 Run 7	28	9999	1355,04	1556,11	759,5	5299	1000
SPB_SARSA_0.0 Run 8	29	610	426,423	112,325	441	520	1000
SPB_SARSA_0.0 Run 9	29	612	474,783	108,075	515	555	1000
SPB_SARSA_0.01 Run 0	28	936	355,113	169,014	369,5	410	1000
SPB_SARSA_0.01 Run 1	29	790	352,484	170,173	366	629	1000
SPB_SARSA_0.01 Run 2	31	1416	399,376	212,88	383,5	961	1000
SPB_SARSA_0.01 Run 3	30	2924	318,514	200,445	297	712	1000
SPB_SARSA_0.01 Run 4	31	757	313,395	123,245	324	490	1000
SPB_SARSA_0.01 Run 5	29	3905	445,498	258,583	432	1048	1000

In the top line, the metric that should be displayed can be selected. The metric "Episode Return" is always available; it shows the accumulated reward per Episode. Below this, a table is shown which presents some statistics (min, max, mean, median etc.) of the chosen metric for the different runs of the worlds. The results of the experiment can be analyzed for statistical significance by pressing the "Statistics" (see *Evaluating experiments*). By pressing "Visualize", these results can also be displayed in a graphical manner:



In this visualizations, one can see the development of the selected metric over time for the two agents. One can select whether one wants to see the average over all runs conducted for one agent or each of these runs separately. Furthermore, one can specify the length of the moving window average. The plot is not updated automatically, but only when one presses "Update" or when one changes the selections. One can save the generaed plot to a file by pressing "Save".

The "Experiment Statistics" tab can also be restored for an experiment conducted earlier by loading the results of this experiment into the Experimenter. This can be done by pressing "Load Experiment Results". This opens a file selection dialog in which the root directory of the particular experiment in the RW area must be selected.

Note: It may happen that different experiments share the same root directory (namely, when both experiments use the same environment). In this case, the Experimenter cannot distinguish these experiments and interprets them as a single experiment. In order to avoid that, please copy the results of an experiment to a unique directory manually.

See Also:

Tutorial *Quick start (command line interface)* Learn how to use the MMLF with a non-grapical user interface

Learn more about *Existing agents* **and** *Existing environments* Get an overview over the agents and environments that are shipped with the MMLF

Tutorial *Writing an agent* Learn how to write your own agent

Tutorial *Writing an environment* Learn how to write your own MMLF environment

Learn more about *Experiments* Learn how to do a serious benchmarking and statistical comparison of the performance of different agents/environments

1.4 Writing an agent

This tuturial will explain how you can write your own learning agent for the MMLF.

Note: Writing an agent is easier with a local installation of the MMLF (see *Installation Tutorial*).

See Also:

Get an overview over the existing agents in *Existing agents*

1.4.1 Learning about the basic structure of MMLF agents

For the start, please take a look into the agents subdirectory of the MMLF and open the random_agent.py in the python editor of your choice. The *RandomAgent* is a quite simple and straightforward agent which demonstrates well the inner life of an agent (though an intelligent agent might choose his actions differently ;-)).

What you can learn from the agent is the following:

- Each agent has to be a subclass of AgentBase
- Each agent class must have a static attribute DEFAULT_CONFIG_DICT, which contains the parameters that are available for customizing the agent's behaviour and their default values.
- The __init__ method gets passed additional arguments (`*args`) and keyword arguments (`**kwargs`). These MUST be passed on to the superclass' constructor using `super(RandomAgent, self).__init__(*args, **kwargs)`
- Each agent must have an AgentInfo attribute that specifies in which kinds of environments it can be used, which communication protocol it supports etc.
- The setStateSpace(self, stateSpace) method is called to inform the agent about the structure of the *state space*. A default implementation of this method is contained in the AgentBase class; if this implementation is sufficient the method need not be implemented again.
- The setActionSpace(self, actionSpace) method is called to inform the agent about the structure of the *action space*. A default implementation of this method is contained in the AgentBase class; if this implementation is sufficient the method need not be implemented again.
- The setState(self, state) method is called to inform the agent about the current state of the environment. To correctly interpret the state, the agent has to use the definition of the *state space*.
- The getAction(self) method is called to ask the agent for the action he wants to perform. The agent should store its decision in a dictionary which maps action dimension name to the chosen value for this dimension. For instance: {"gasPedalForce": "extreme", "steeringWheelAngle": 30} (see page *action space*). This action dictionary must be converted to an ActionTaken object via the method _generateActionObject(actionDictionary) of AgentBase.
- The giveReward(self, reward) method is called to reward the agent for its last action(s). The passed reward is a float value. The agent can treat this reward in different ways, e.g. accumulate it, use it directly for policy optimization etc.
- The nextEpisodeStarted(self) method is called whenever one epsiode is over and the next one is started, i.e. when the environment is reset. In this method, you should finish all calculations which make only sense during one episode (such as accumulating the reward obtained during one episode).

- In each agent module, the module-level attribute AgentClass needs set to the class that inherits from AgentBase. This assignment is located usually at the end of the module: AgentClass = RandomAgent
- Furthermore, the module-level attribute AgentName should be set to the name of the agent, e.g. AgentName = "Random". This name is used for instance in the GUI.
- The agent can send messages to the logger by calling "self.agentLog.info(message)"

1.4.2 Writing a new MMLF agent

Let's write a new agent. This agent should execute actions independent of the state in a round-robin like manner, i.e. when there are three available actions a1, a2, a3, the agent should choose actions in this sequence: a1,a2,a3,a1,a2,a3,... Obviously this is not a very clever approach, but for the tutorial it should suffice.

In order to implement a new agent that chooses actions in a round-robin like manner, you have to do the following:

1. Go into the agents subdirectory of the MMLF and create a copy of the random_agent.py (let's call this copy example_agent.py).
2. Open example_agent.py and rename the agent class from RandomAgent to ExampleAgent. Replace every occurrence of RandomAgent by ExampleAgent.
3. Set "DEFAULT_CONFIG_DICT = { }", since the agent does not have any configuration options.
4. Set "continuousAction = False", since the round-robin action selection is only possible for a finite (non-continuous) action set.
5. Add the following lines add the end of "setActionSpace":

```
# We can only deal with one-dimensional action spaces
assert self.actionSpace.getNumberOfDimensions() == 1
# Get a list of all actions this agent might take
self.actions = self.actionSpace.getActionList()
# Get name of action dimension
self.actionDimensionName = self.actionSpace.getDimensionNames()[0]
# Create an iterator that iterates in a round-robin manner over available actions
self.nextActionIterator = __import__("itertools").cycle(self.actions)
```

6. Reimplement the method "getAction()" as follows:

```
def getAction(self, **kwargs):
    """ Request the next action the agent want to execute """
    # Get next action  from iterator
    # We are only interested in the value of the first (and only) dimension,
    # thus the "0"
    nextAction = self.nextActionIterator.next()[0]
    # Create a dictionary that maps dimension name to chosen action
    actionDictionary = {self.actionDimensionName : nextAction}
    # Generate mmlf.framework.protocol.ActionTaken object
    return self._generateActionObject(actionDictionary)
```

7. Remove superfluous methods "setStateSpace()", "setState()", "giveReward", and "nextEpisodeStarted()". They can use the default implementation of the AgentBase class
8. Set thet AgentClass module attribute appropriately: "AgentClass = ExampleAgent"
9. Set the AgentName to something meaningful: "AgentName = "RoundRobin""
10. Do not forget to update the comments and the documentation of your new module!

That's it! Your agent module should now look like shown *here*. You can test it in the GUI by selecting "RoundRobin" as agent.

1.4.3 RandomAgent

```
# Maja Machine Learning Framework
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published
# by the Free Software Foundation; either version 3 of the License, or
# (at your option) any later version.

# This program is distributed in the hope that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, see <http://www.gnu.org/licenses/>.

# Author: Jan Hendrik Metzen  (jhm@informatik.uni-bremen.de)
# Created: 2007/07/23

""" MMLF agent that acts randomly

This module defines a simple agent that can interact with an environment.
It chooses all available actions with the same probability.

This module deals also as an example of how to implement an MMLF agent.
"""

__author__ = "Jan Hendrik Metzen"
__copyright__ = "Copyright 2011, University Bremen, AG Robotics"
__credits__ = ['Mark Edgington']
__license__ = "GPLv3"
__version__ = "1.0"
__maintainer__ = "Jan Hendrik Metzen"
__email__ = "jhm@informatik.uni-bremen.de"

from collections import defaultdict

import mmlf.framework.protocol

from mmlf.agents.agent_base import AgentBase

# Each agent has to inherit directly or indirectly from AgentBase
class RandomAgent(AgentBase):
    """ Agent that chooses uniformly randomly among the available actions. """

    # Add default configuration for this agent to this static dict
    # This specific parameter controls after how many steps we send information
    # regarding the accumulated reward to the logger.
    DEFAULT_CONFIG_DICT = {'Reward_log_frequency' : 100}

    def __init__(self, *args, **kwargs):
        # Create the agent info
        self.agentInfo = \
            mmlf.framework.protocol.AgentInfo(# Which communication protocol
                                                 # version can the agent handle?
                                                 versionNumber = "0.3",
                                                 # Name of the agent (can be
                                                 # chosen arbitrarily)
                                                 agentName= "Random",
                                                 # Can the agent be used in
                                                 # environments with continuous
                                                 # state spaces?
                                                 continuousState = True,
                                                 # Can the agent be used in
                                                 # environments with continuous
                                                 # action spaces?
                                                 continuousAction = True,
                                                 # Can the agent be used in
                                                 # environments with discrete
                                                 # action spaces?
                                                 discreteAction = True,
                                                 # Can the agent be used in
                                                 # non-episodic environments
                                                 nonEpisodicCapable = True)

        # Calls constructor of base class
        # After this call, the agent has an attribute "self.configDict",
        # The values of this dict are evaluated, i.e. instead of '100' (string),
        # the key 'Reward log frequency' will have the same value 100 (int).
        super(RandomAgent, self).__init__(*args, **kwargs)

        # The superclass AgentBase implements the methods setStateSpace() and
        # setActionSpace() which set the attributes stateSpace and actionSpace
        # They can be overwritten if the agent has to modify these spaces
        # for some reason
        self.stateSpace = None
        self.actionSpace = None

        # The agent keeps track of all rewards it obtained in an episode
        # The rewardDict implements a mapping from the episode index to a list
        # of all rewards it obtained in this episode
        self.rewardDict = defaultdict(list)

    ######################  BEGIN COMMAND-HANDLING METHODS ##############################

    def setStateSpace(self, stateSpace):
        """ Informs the agent about the state space of the environment

        More information about state spaces can be found in
        :ref:`state_and_action_spaces`
        """
        # We delegate to the superclass, which does the following:
        # self.stateSpace = stateSpace
        # We need not implement this method for this, but it is given in order
        # to show what is going on...
        super(RandomAgent, self).setStateSpace(stateSpace)


    def setActionSpace(self, actionSpace):
        """ Informs the agent about the action space of the environment
        More information about action spaces can be found in
        :ref:`state_and_action_spaces`
        """
        # We delegate to the superclass, which does the following:
        # self.actionSpace = actionSpace
        # We need not implement this method for this, but it is given in order
        # to show what is going on...
        super(RandomAgent, self).setActionSpace(actionSpace)

    def setState(self, state):
        """ Informs the agent of the environment's current state

        More information about (valid) states can be found in
        :ref:`state_and_action_spaces`
        """
        # We delegate to the superclass, which does the following:
        #     self.state = self.stateSpace.parseStateDict(state) # Parse state dict
        #     self.state.scale(0, 1) # Scale state such that each dimension falls into the bin (0,1)
        #     self.stepCounter += 1 # Count how many steps have passed

        # We need not implement this method for this, but it is given in order
        # to show what is going on...
        super(RandomAgent, self).setState(state)


    def getAction(self):
        """ Request the next action the agent want to execute """
        # Each action of the agent corresponds to one step
        action = self._chooseRandomAction()

        # Call super class method since this updates some internal information
        # (self.lastState, self.lastAction, self.reward, self.state, self.action)
        super(RandomAgent, self).getAction()

        return action

    def giveReward(self, reward):
        """ Provides a reward to the agent """
        self.rewardDict[self.episodeCounter].append(reward) # remember reward
        # Send message about the accumulated reward every
        # self.configDict['Reward log frequency'] episodes to logger
        if self.stepCounter % self.configDict['Reward_log_frequency'] == 0:
            self.agentLog.info("Reward accumulated after %s steps in episode %s: %s"
                               % (self.stepCounter, self.episodeCounter,
                                  sum(self.rewardDict[self.episodeCounter])))

    def nextEpisodeStarted(self):
        """ Informs the agent that a new episode has started."""
        # We delegate to the superclass, which does the following:
        #     self.episodeCounter += 1
        #     self.stepCounter = 0
        super(RandomAgent, self).nextEpisodeStarted()

    #######################  END COMMAND-HANDLING METHODS ##############################

    def _chooseRandomAction(self):
        "Chooses an action randomly from the action space"
        assert self.actionSpace, "Error: Action requested before actionSpace "\
                                 "was specified"

        # We sample a random action from the action space
        # This returns a dictionary with a mapping from action dimension name
        # to the sample value.
        # For instance: {"gasPedalForce": "extreme", "steeringWheelAngle": 30}
        actionDictionary = self.actionSpace.sampleRandomAction()

        # The action dictionary has to be converted into an
        # mmlf.framework.protocol.ActionTaken object.
        # This is done using the _generateActionObject method
        # of the superclass
        return self._generateActionObject(actionDictionary)

# Each module that implements an agent must have a module-level attribute
# "AgentClass" that is set to the class that inherits from Agentbase
AgentClass = RandomAgent
# Furthermore, the name of the agent has to be assigned to "AgentName". This
# name is used in the GUI.
AgentName = "Random"
```

1.4.4 ExampleAgent

```
# Maja Machine Learning Framework
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published
# by the Free Software Foundation; either version 3 of the License, or
# (at your option) any later version.

# This program is distributed in the hope that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, see <http://www.gnu.org/licenses/>.

# Author: Jan Hendrik Metzen  (jhm@informatik.uni-bremen.de)
# Created: 2007/07/23

""" MMLF agent that chooses actions in a round-robin manner.

This agent's sole purpose is to give an example of how to write an agent.
It should not be used for any actual learning.
"""

__author__ = "Jan Hendrik Metzen"
__copyright__ = "Copyright 2011, University Bremen, AG Robotics"
__credits__ = ['Mark Edgington']
__license__ = "GPLv3"
__version__ = "1.0"
__maintainer__ = "Jan Hendrik Metzen"
__email__ = "jhm@informatik.uni-bremen.de"
```

```
import mmlf.framework.protocol

from mmlf.agents.agent_base import AgentBase

# Each agent has to inherit directly or indirectly from AgentBase
class ExampleAgent(AgentBase):
    """ MMLF agent that chooses actions in a round-robin manner. """

    DEFAULT_CONFIG_DICT = {}

    def __init__(self, *args, **kwargs):

        # Create the agent info
        self.agentInfo = \
            mmlf.framework.protocol.AgentInfo(# Which communication protocol
                                              # version can the agent handle?
                                              versionNumber = "0.3",
                                              # Name of the agent (can be
                                              # chosen arbitrarily)
                                              agentName= "Round Robin",
                                              # Can the agent be used in
                                              # environment with contiuous
                                              # state spaces?
                                              continuousState = True,
                                              # Can the agent be used in
                                              # environment with continuous
                                              # action spaces?
                                              continuousAction = True,
                                              # Can the agent be used in
                                              # environment with discrete
                                              # action spaces?
                                              discreteAction = True,
                                              # Can the agent be used in
                                              # non-episodic environments
                                              nonEpisodicCapable = True)

        # Calls constructor of base class
        # After this call, the agent has an attribute "self.configDict",
        # that contains the information from config['configDict'].
        # The values of this dict are evaluated, i.e. instead of '100' (string),
        # the key 'Reward log frequency' will have the same value 100 (int).
        super(ExampleAgent, self).__init__(*args, **kwargs)

        # The superclass AgentBase implements the methods setStateSpace() and
        # setActionSpace() which set the attributes stateSpace and actionSpace
        # They can be overwritten if the agent has to modify these spaces
        # for some reason
        self.stateSpace = None
        self.actionSpace = None

        # The agent keeps track of the sum of all rewards it obtained
        self.rewardValue = 0

    ######################  BEGIN COMMAND-HANDLING METHODS ##############################

    def setActionSpace(self, actionSpace):
        """ Informs the agent about the action space of the environment
        More information about action spaces can be found in
        :ref:`state_and_action_spaces`
        """
        super(ExampleAgent, self).setActionSpace(actionSpace)

        # We can only deal with one-dimensional action spaces
        assert self.actionSpace.getNumberOfDimensions() == 1

        # Get a list of all actions this agent might take
        self.actions = self.actionSpace.getActionList()
        # Get name of action dimension
        self.actionDimensionName = self.actionSpace.getDimensionNames()[0]
        # Create an iterator that iterates in a round-robin manner over available actions
        self.nextActionIterator = __import__("itertools").cycle(self.actions)

    def getAction(self):
        """ Request the next action the agent want to execute """
        # Get next action  from iterator
        # We are only interested in the value of the first (and only) dimension,
        # thus the "0"
        nextAction = self.nextActionIterator.next()[0]
        # Create a dictionary that maps dimension name to chosen action
        actionDictionary = {self.actionDimensionName : nextAction}

        # Call super class method since this updates some internal information
        # (self.lastState, self.lastAction, self.reward, self.state, self.action)
        super(ExampleAgent, self).getAction()

        # Generate mmlf.framework.protocol.ActionTaken object
        return self._generateActionObject(actionDictionary)

# Each module that implements an agent must have a module-level attribute
# "AgentClass" that is set to the class that implements the AgentBase superclass
AgentClass = ExampleAgent
# Furthermore, the name of the agent has to be assigned to "AgentName". This
# name is used in the GUI.
AgentName = "RoundRobin"
```

1.5 Writing an environment

This tuturial will explain how you can implement your own environment for the MMLF.

Note: Implementing a new environment is easier with a local installation of the MMLF (see *Installation Tutorial*).

See Also:

Get an overview over the existing environments in *Existing environments*

1.5.1 Learning about the basic structure of MMLF environments

To begin, please take a look into the worlds/linear_markov_chain/environments subdirectory of the MMLF and open the linear_markov_chain_environment.py in the python editor of your choice. The *Linear Markov Chain* is a quite simple and straightforward environment which demonstrates well the inner life of an environment.

What you can learn from the environment is the following:

- Each environment has to be a subclass of SingleAgentEnvironment
- Each environment class must have a static attribute DEFAULT_CONFIG_DICT, which contains the parameters that are available for customizing the environment and their default values.
- The __init__ method gets passed additional arguments (`*args`) and keyword arguments (`**kwargs`). These MUST be passed on to the superclass' constructor using `super(SingleAgentEnvironment, self).__init__(useGUI, *args, **kwargs)`
- Each environment must have an EnvironmentInfo attribute that specifies which communication protocol the environment supports, which capabilities agents must have that can be used in this environment etc.
- The __init__ method defines *state space* and *action space* of the environment as well as its initial state. In the most simple form, these spaces are defined as dicts that map dimension name onto a pair specifying whether the dimension has discrete or continuous values and which values may occur (so-called 'old-style' spaces).
- The evaluateAction(self, actionObject) method is called to compute the effect of an action chosen by the agent onto the environment. The state transition is computed, and whether an episode has finished (i.e. whether a terminal state has been reached) is checked. Depending on this, the reward is computed. A dictionary containing the immediate reward, the terminal state (if one is reached; otherwise None), the current state (possibly the initial state of the next episode if the episode has been terminated), and a boolean indicating whether a new episodes starts is returned.
- In each environment module, the module-level attribute EnvironmentClass needs set to the class that inherits from SingleAgentEnvironment. This assignment is located usually at the end of the module: EnvironmentClass = LinearMarkovChainEnvironment
- Furthermore, the module-level attribute EnvironmentName should be set to the name of the environment, e.g. EnvironmentName = "Linear Markov Chain". This name is used for instance in the GUI.
- The environment can send messages to the logger by calling "self.environmentLog.info(message)"

1.5.2 Writing a new MMLF environment

For writing a new MMLF environment, the following steps must be executed:

1. Go into the worlds subdirectory of the MMLF and create a new world directory (e.g. example_world). Make this subdirectory a python package by adding an empty __init__.py file. Create a subdirectory "environments" in the world directory. In this "environments" subdirectory, create again an empty __init__.py file and a file that contains the actual python environment module (e.g. example_environment.py)
2. Open the example_environment.py file. In this file, you have to implement a subclass of SingleAgentEnvironment. Lets call this subclass ExampleEnvironment.
3. The environment class must have a class-attribute DEFAULT_CONFIG_DICT, which is a dictionary that contains the parameters that are available for customizing the environment and their default values. These parameters can be later on configured, e.g., in the MMLF GUI. Each parameter that can customize the behaviour of your environment should be contained in this dictionary. If your environment has no parameters, you can simply set "DEFAULT_CONFIG_DICT = {}"
4. In the __init__ method of the class, you have to specify EnvironmentInfo. Adapt this object such that reflects the demands your environment poses onto agents that can be used in it.
5. State- and ActionSpace must be defined. These can be either defined by defining each of their dimensions explicitly (see *State and Action Spaces*) and adding them to the spaces or by defining the spaces directly the "old-style" way. Such an old-style definition is a dictionary mapping the dimension names to a shorthand definition of them:

```
{"column": ("discrete", [0,1,2,3]),
"row": ("discrete", [3,4,5])}
```

This defines a space with two discrete dimensions with the names “column” and “row”. The “column” dimension can take on the values 0,1,2, and 3 and the “row” dimension the values 3,4, and 5.

6. The *getInitialState* method must be implemented: This method is used for sampling a start state at the beginning of each episode. This state is currently NOT an MMLF state object but a dictionary which maps dimension name to dimension value. This may change in future releases of the MMLF.
7. The *evaluateAction* method is the place where the actual dynamics of the environment are implemented. It gets as parameter the actionObject chosen by the agent. This actionObject is a dictionary mapping the action space dimensions onto the values chosen by the agent for the respective dimension. Thus, via “actionObject[’force’]”, one could access the force the agent has chosen (let force be an action space dimension). The implementation of the method depends on your environment; important is that a state transition has to happen and a reward must be computed. The method must thus return a dictionary containing the immediate reward, the terminal state (if one is reached; otherwise None), the current state (possibly the initial state of the next episode if the episode has been terminated), and a boolean indicating whether a new episodes starts is.
8. Create an module attribute EnvironmentClass and assign the environment class to it: “EnvironmentClass = ExampleEnvironment”
9. Create an module attribute EnvironmentName and assign the environment’s name to it: “Environment-Name = “Example”“

1.5.3 LinearMarkovChainEnvironment

```
# Maja Machine Learning Framework
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published
# by the Free Software Foundation; either version 3 of the License, or
# (at your option) any later version.

# This program is distributed in the hope that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, see <http://www.gnu.org/licenses/>.

# Author: Jan Hendrik Metzen  (jhm@informatik.uni-bremen.de)
# Created: 2011/04/05
""" A linear markov chain environment. """

__author__ = "Jan Hendrik Metzen"
__copyright__ = "Copyright 2011, University Bremen, AG Robotics"
__credits__ = ['Mark Edgington']
__license__ = "GPLv3"
__version__ = "1.0"
__maintainer__ = "Jan Hendrik Metzen"
__email__ = "jhm@informatik.uni-bremen.de"

from copy import deepcopy
```

```python
from mmlf.framework.spaces import StateSpace, ActionSpace
from mmlf.framework.protocol import EnvironmentInfo
from mmlf.environments.single_agent_environment import SingleAgentEnvironment

# Each environment has to inherit directly or indirectly from SingleAgentEnvironment
class LinearMarkovChainEnvironment(SingleAgentEnvironment):
    """ A linear markov chain.

    The agent starts in the middle of this linear markov chain. He can either
    move right or left. The chain is not stochastic, i.e. when the agent
    wants to move right, the state is decreased with probability 1 by 1.
    When the agent wants to move left, the state is increased with probability 1
    by 1 accordingly.

    .. versionadded:: 0.9.10
       Added LinearMarkovChain environment

    **CONFIG DICT**
        :length: : The number of states of the linear markov chain

    """

    # Add default configuration for this environment to this static dict
    # This specific parameter controls how long the linear markov chain is
    # (i.e. how many states there are)
    DEFAULT_CONFIG_DICT = {"length" : 21}

    def __init__(self, useGUI, *args, **kwargs):
        # Create the environment info
        self.environmentInfo = \
            EnvironmentInfo(# Which communication protocol version can the
                            # environment handle?
                            versionNumber="0.3",
                            # Name of the environment (can be chosen arbitrarily)
                            environmentName="LinearMarkovChain",
                            # Is the action space of this environment discrete?
                            discreteActionSpace=True,
                            # Is the environment episodic?
                            episodic=True,
                            # Is the state space of environment continuous?
                            continuousStateSpace=False,
                            # Is the action space of environment continuous?
                            continuousActionSpace=False,
                            # Is the environment stochastic?
                            stochastic=False)

        # Calls constructor of base class
        # After this call, the environment has an attribute "self.configDict",
        # The values of this dict are evaluated, i.e. instead of '100' (string),
        # the key 'length' will have the same value 100 (int).
        super(LinearMarkovChainEnvironment, self).__init__(useGUI=useGUI, *args, **kwargs)

        # The state space of the linear markov chain
        oldStyleStateSpace =  {"field": ("discrete", range(self.configDict["length"]))}

        self.stateSpace = StateSpace()
        self.stateSpace.addOldStyleSpace(oldStyleStateSpace, limitType="soft")
        # The action space of the linear markov chain
        oldStyleActionSpace =  {"action": ("discrete", ["left", "right"])}

        self.actionSpace = ActionSpace()
        self.actionSpace.addOldStyleSpace(oldStyleActionSpace, limitType="soft")

        # The initial state of the environment
        self.initialState =  {"field": self.configDict["length"] / 2}
        # The current state is initially set to the initial state
        self.currentState = deepcopy(self.initialState)

    ########################## Interface Functions #####################################
    def getInitialState(self):
        """ Returns the initial state of the environment """
        self.environmentLog.debug("Episode starts in state '%s'."
                                    % (self.initialState['field']))
        return self.initialState

    def evaluateAction(self, actionObject):
        """ Execute an agent's action in the environment.

        Take an actionObject containing the action of an agent, and evaluate
        this action, calculating the next state, and the reward the agent
        should receive for having taken this action.

        Additionally, decide whether the episode should continue,
        or end after the reward has been  issued to the agent.

        This method returns a dictionary with the following keys:
           :rewardValue: : An integer or float representing the agent's reward.
                           If rewardValue == None, then no reward is given to the agent.
           :startNewEpisode: : True if the agent's action has caused an episode
                               to get finished.
           :nextState: : A State object which contains the state the environment
                         takes on after executing the action. This might be the
                         initial state of the next episode if a new episode
                         has just started (startNewEpisode == True)
           :terminalState: : A State object which contains the terminal state
                             of the environment in the last episode if a new
                             episode has just started (startNewEpisode == True).
                             Otherwise None.
        """
        action = actionObject['action']
        previousState = self.currentState['field']

        # Change state of environment deterministically
        if action == 'left':
            self.currentState['field'] -= 1
        else:
            self.currentState['field'] += 1

        self.environmentLog.debug("Agent chose action '%s' which caused a transition from '%s' to '%
                                    % (action, previousState, self.currentState['field']))

        #Check if the episode is finished (i.e. the goal is reached)
        episodeFinished = self._checkEpisodeFinished()

        terminalState = self.currentState if episodeFinished else None
        if episodeFinished:
            self.episodeLengthObservable.addValue(self.episodeCounter,
                                                  self.stepCounter + 1)
            self.returnObservable.addValue(self.episodeCounter,
                                           -self.stepCounter)
            self.environmentLog.debug("Terminal state '%s' reached."
                                            % self.currentState['field'])
            self.environmentLog.info("Episode %s lasted for %s steps."
                                     % (self.episodeCounter, self.stepCounter  + 1))

            reward = 10 if self.currentState['field'] != 0 else -10

            self.stepCounter = 0
            self.episodeCounter += 1

            # Reset the simulation to the initial state (always the same)
            self.currentState = deepcopy(self.initialState)
        else:
            reward = -1
            self.stepCounter += 1

        resultsDict = {"reward" : reward,
                       "terminalState" : terminalState,
                       "nextState" : self.currentState,
                       "startNewEpisode" : episodeFinished}
        return resultsDict

    def _checkEpisodeFinished(self):
        """ Checks whether the episode is finished.

        An episode is finished whenever the leftmost or rightmost state of the
        chain is reached.
        """
        return self.currentState['field'] in [0, self.configDict['length']-1]

# Each module that implements an environment must have a module-level attribute
# "EnvironmentClass" that is set to the class that inherits from SingleAgentEnvironment
EnvironmentClass = LinearMarkovChainEnvironment
# Furthermore, the name of the environment has to be assigned to "EnvironmentName".
# This  name is used in the GUI.
EnvironmentName = "Linear Markov Chain"
```

CHAPTER

TWO

LEARN MORE ABOUT...

2.1 Existing Agents and Environments

2.1.1 Existing agents

Note: This list of agents has been automatically generated from the source code

Package that contains all available MMLF agents.

A list of all agents:

- Agent '*Model-based Direct Policy Search*' from module mbdps_agent implemented in class MBDPS_Agent.

  An agent that uses the state-action-reward-successor_state transitions to learn a model of the environment. It performs direct policy search (similar to the direct policy search agent using a black-box optimization algorithm to optimize the parameters of a parameterized policy) in the model in order to optimize a criterion defined by a fitness function. This fitness function can be e.g. the estimated accumulated reward obtained by this policy in the model environment. In order to enforce exploration, the model is wrapped for an RMax-like behavior so that it returns the reward RMax for all states that have not been sufficiently explored. RMax should be an upper bound to the actual achievable return in order to enforce optimism in the face of uncertainty.

- Agent '*Monte-Carlo*' from module monte_carlo_agent implemented in class MonteCarloAgent.

  An agent which uses Monte Carlo policy evaluation to optimize its behavior in a given environment.

- Agent '*Dyna TD*' from module dyna_td_agent implemented in class DynaTDAgent.

  Dyna-TD uses temporal difference learning along with learning a model of the environment and doing planning in it.

- Agent '*Temporal Difference + Eligibility*' from module td_lambda_agent implemented in class TDLambdaAgent.

  An agent that uses temporal difference learning (e.g. Sarsa) with eligibility traces and function approximation (e.g. linear tile coding CMAC) to optimize its behavior in a given environment

- Agent '*Policy Replay*' from module policy_replay_agent implemented in class PolicyReplayAgent.

  Agent which loads a stored policy and follows it without improving it.

- Agent '*Random*' from module random_agent implemented in class RandomAgent.

- Agent ‘*Actor Critic*‘ from module actor_critic_agent implemented in class ActorCriticAgent.

    This agent learns based on the actor critic architecture. It uses standard TD(lambda) to learn the value function of the critic. For this reason, it subclasses TDLambdaAgent. The main difference to TD(lambda) is the means for action selection. Instead of deriving an epsilon-greedy policy from its Q-function, it learns an explicit stochastic policy. To this end, it maintains preferences for each action in each state. These preferences are updated after each action execution according to the following rule:

- Agent ‘*RoundRobin*‘ from module example_agent implemented in class ExampleAgent.
- Agent ‘*Direct Policy Search*‘ from module dps_agent implemented in class DPS_Agent.

    This agent uses a black-box optimization algorithm to optimize the parameters of a parametrized policy such that the accumulated (undiscounted) reward of the the policy is maximized.

- Agent ‘*Fitted-RMax*‘ from module fitted_r_max_agent implemented in class FittedRMaxAgent.

    Fitted R-Max is a model-based RL algorithm that uses the RMax heuristic for exploration control, uses a fitted function approximator (even though this can be configured differently), and uses Dynamic Programming (boosted by prioritized sweeping) for deriving a value function from the model. Fitted R-Max learns usually very sample-efficient (meaning that a good policy is learned with only a few interactions with the environment) but requires a huge amount of computational resources.

See Also:

Tutorial ***Writing an agent*** Learn how to write your own MMLF agent

2.1.2 Existing environments

Note: This list of environments has been automatically generated from the source code

Package that contains all available MMLF world environments.

A list of all environments:

- Environment ‘*Maze Cliff*‘ from module maze_cliff_environment implemented in class MazeCliffEnvironment.

    In this maze, there are two alternative ways from the start to the goal state: one short way which leads along a dangerous cliff and one long but secure way. If the agent happens to step into the maze, it will get a huge negative reward (configurable via *cliffPenalty*) and is reset into the start state. Per default, the maze is deterministic, i.e. the agent always moves in the direction it chooses. However, the parameter *stochasticity* allows to control the stochasticity of the environment. For instance, when stochasticity is set to 0.01, the the agent performs a random move instead of the chosen one with probability 0.01.

- Environment ‘*Pinball 2D*‘ from module pinball_maze_environment implemented in class PinballMazeEnvironment.

    The pinball maze environment class.

- Environment ‘*Linear Markov Chain*‘ from module linear_markov_chain_environment implemented in class LinearMarkovChainEnvironment.

    The agent starts in the middle of this linear markov chain. He can either move right or left. The chain is not stochastic, i.e. when the agent wants to move right, the state is decreased with probability 1 by 1. When the agent wants to move left, the state is increased with probability 1 by 1 accordingly.

- Environment '*Maze 2D*' from module maze2d_environment implemented in class Maze2dEnvironment.

  A 2d maze world, in which the agent is situated at each moment in time in a certain field (specified by its (row,column) coordinate) and can move either upwards, downwards, left or right. The structure of the maze can be configured via a text-based config file.

- Environment '*Double Pole Balancing*' from module double_pole_balancing_environment implemented in class DoublePoleBalancingEnvironment.

  In the double pole balancing environment, the task of the agent is to control a cart such that two poles which are mounted on the cart stay in a nearly vertical position (to balance them). At the same time, the cart has to stay in a confined region.

- Environment '*Partial Observable Double Pole Balancing*' from module po_double_pole_balancing_environment implemented in class PODoublePoleBalancingEnvironment.

  In the partially observable double pole balancing environment, the task of the agent is to control a cart such that two poles which are mounted on the cart stay in a nearly vertical position (to balance them). At the same time, the cart has to stay in a confined region. In contrast to the fully observable double pole balancing environment, the agent only observes the current position of cart and the two poles but not their velocities. This renders the problem to be not markovian.

- Environment '*Mountain Car*' from module mcar_env implemented in class MountainCarEnvironment.

  In the mountain car environment, the agent has to control a car which is situated somewhere in a valley between two hills. The goal of the agent is to reach the top of the right hill. Unfortunately, the engine of the car is not strong enough to reach the top of the hill directly from many start states. Thus, it has first to drive in the wrong direction to gather enough potential energy.

- Environment '*Single Pole Balancing*' from module single_pole_balancing_environment implemented in class SinglePoleBalancingEnvironment.

  In the single pole balancing environment, the task of the agent is to control a cart such that a pole which is mounted on the cart stays in a nearly vertical position (to balance it). At the same time, the cart has to stay in a confined region.

- Environment '*Seventeen and Four*' from module seventeen_and_four implemented in class SeventeenAndFourEnvironment.

  This environment implements a simplified form of the card game seventeen & four, in which the agent takes the role of the player and plays against a hard-coded dealer.

See Also:

Tutorial ***Writing an environment*** Learn how to write your own MMLF environment

2.2 Experiments

With "experiment", we refer to the systematic analysis of a RL method, the influence of its parameters onto its performance, and/or the comparison of different RL methods. Note: Even though we focus in this text onto the agent component of the world, similar experiments could also be conducted for comparing the performance of an agent in different environments.

2.2.1 Conducting experiments

Experiments can be either be conducted using the *command line interface* or using the *MMLF Experimenter*. For this example, download this example experiment configuration here, and extract it into your rw-directory (this should result in a directory ~/.mmlf/test_experiment). Now, you can execute the experiment using:

```
run_mmlf --experiment test_experiment
```

Note that you have to use the `--experiment` argument instead of `--config`, and the path of the experiment *directory* must be relative to the rw-directory. For this particular experiment, this will conduct 5 runs a 100 episodes for 3 worlds. Alternatively, you could also load the experiment into the MMLF experimenter using the “Load Experiment” button and execute it using the “Start Experiment” button. In this case, you can monitor the progress in the GUI online. See the *MMLF Experimenter* for more details. A further option for conducting experiments is to script them (see *Scripting experiments*).

The experiment configuration (in the test_experiment directory) consists of the definition of several worlds (one for each yaml-file in the worlds subdirectory of test_experiment directory) and the configuration of the experiment itself in experiment_config.yaml:

```
concurrency: Sequential
episodesPerRun: '100'
parallelProcesses: '1'
runsPerWorld: '5'
```

This configuration defines whether the experiment should be executed sequentially (“concurrency: Sequential”, i.e. only one run at a time) or concurrently (“concurrency: Concurrent”, i.e. several runs at the same time), how many episodes per world run should be conducted (“episodesPerRun”), how many runs are executed in parallel for concurrent execution order (“parallelProcesses”), and how many runs (repetitions) should be conducted for each world (“runsPerWorld”). Note that “parallelProcesses” is ignored if the execution order is sequential.

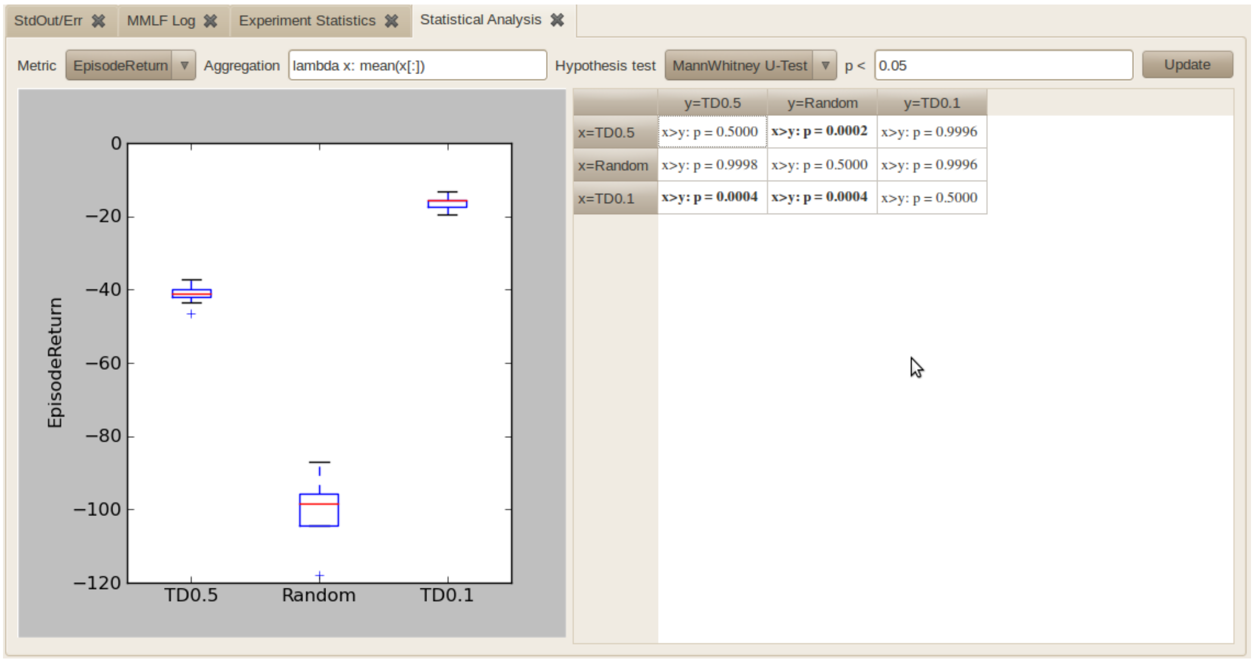
In this example experiment, three different agents are compared in the linear markov chain environment: an agent acting randomly, an TD-based agent with epsilon=0.5, and an TD-based agent with epsilon=0.1. Each of this agents acts for 100 episodes and gets 5 independent runs. By looking at the logs/linear_markov_chain directory, you may see that three novel results directories have been created (“849418770030302468” etc., see *Logging* for an explanation why the directories are named this way.) For convenience, you may rename these directories to more meaningful names once the experiment is finished (e.g. “Random”, “TD0.1”, and “TD0.5”). You can identify which directory belongs to which agent by looking into the world.yaml file in these directories.

2.2.2 Evaluating experiments

Once an experiment is finished, you can analyze it using the MMLF Experimenter. If you already conducted the experiment using the MMLF experimenter, a tab with the title “Experiment Statistics” has already been created which shows the results of the experiment. Otherwise, you can load the experiment's results by pressing “Load Experiment Results” in the MMLF Experimenter. This opens a file selection dialog in which the root directory of the particular experiment in the RW area must be selected (logs/linear_markov_chain for the example above).

Note: It may happen that different experiments share the same root directory (namely, when both experiments use the same environment). In this case, the Experimenter cannot distinguish these experiments and interprets them as a single experiment. In order to avoid that, please copy the results of an experiment to a unique directory manually.

The “Experiment Statistics” tab shows you various statistics for a selected metric and allows to plot these results by pressing “Visualize”. For a more detailed explanation of the “Experiment Statistics” tab, we refer to the *MMLF Experimenter*. For now, the interesting part is the “Statistical Analysis” tab that is opened by pressing “Statistics” in the “Experiment Statistics” tab:



Metric: EpisodeReturn | Aggregation: lambda x: mean(x[:]) | Hypothesis test: MannWhitney U-Test | p < 0.05

	y=TD0.5	y=Random	y=TD0.1
x=TD0.5	x>y: p = 0.5000	**x>y: p = 0.0002**	x>y: p = 0.9996
x=Random	x>y: p = 0.9998	x>y: p = 0.5000	x>y: p = 0.9996
x=TD0.1	**x>y: p = 0.0004**	**x>y: p = 0.0004**	x>y: p = 0.5000

This tab shows you a boxplot based visualization of the average performance of the different setups and also a comparison base

- The "metric" based on which the different setups should be compared (e.g. "EpisodeReturn" or "Episode-Length")
- How the different values the metric takes on during a run should be combined into a single scalar value ("aggregated"). This is accomplished by using a python lambda function like "lambda x: mean(x[:])". In this expression, the sequence of values the metric takes on during a run is stored in x (typically one value per episode). One can now select just a subset of this values using slicing (e.g. x[0:20] would just retain the first 20 values while x[:] retains all values) and choose how this value range is aggregated (mean, median, max, min are available). For instance, to compare based on the minimal performance obtained during the first 30 episodes, use "lambda x: min(x[0:30])".
- Which statistical hypothesis test is conducted (Student t-test or MannWhitney u-test)
- Which level is considered to be significant (typically p < 0.05)

By pressing "Update", the graphic and the significance table are updated. In the table, pairs of setups for which one setup obtained significantly better results than the other are shown bold (in the example TD0.1>TD0.5>Random)

Note: Please take care when comparing many setups: No correction for multiple testing is performed.

2.2.3 Scripting experiments

If a large number of agent/environments combinations should be compared it may become inconvenient to create for each combination a separate world file or to configure this world in the MMLF GUI. In such a situation it may be more convenient to write a python script that create the world configurations on the fly, execute the experiment and consolidates the log files. These python scripts use the *MMLF python package interface*. For example, the following script (download `here`) lets the TD(lambda) agent learn in the mountain car environment for six different values of the exploration ratio epsilon:

```
# Maja Machine Learning Framework
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published
# by the Free Software Foundation; either version 3 of the License, or
# (at your option) any later version.

# This program is distributed in the hope that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, see <http://www.gnu.org/licenses/>.

import os
import shutil
import copy
import yaml

# The mmlf must be importable (global installation or the directory into which
# the MMLF has been extracted must be in python search path).
import mmlf
# Initialize logging and rw-area
mmlf.setupConsoleLogging()
mmlf.initializeRWArea()

# Template for world configuration
# The specific values for epsilon will be changed later, the rest remains
# identical for all worlds
worldConfTemplateYaml = \
"""
worldPackage : mountain_car
environment:
    moduleName : "mcar_env"
    configDict:
        maxStepsPerEpisode : 500
        accelerationFactor : 0.001
        maxGoalVelocity : 0.07
        positionNoise : 0.0
        velocityNoise : 0.0
agent:
    moduleName : "td_lambda_agent"
    configDict:
        gamma : 1.0
        epsilon : 0.0
        lambda : 0.95
        minTraceValue : 0.5
        stateDimensionResolution : 9
        actionDimensionResolution : 7
        function_approximator :
            name : 'CMAC'
            number_of_tilings : 10
            learning_rate : 0.5
            update_rule : 'exaggerator'
            default : 0.0
monitor:
    policyLogFrequency : 250
```

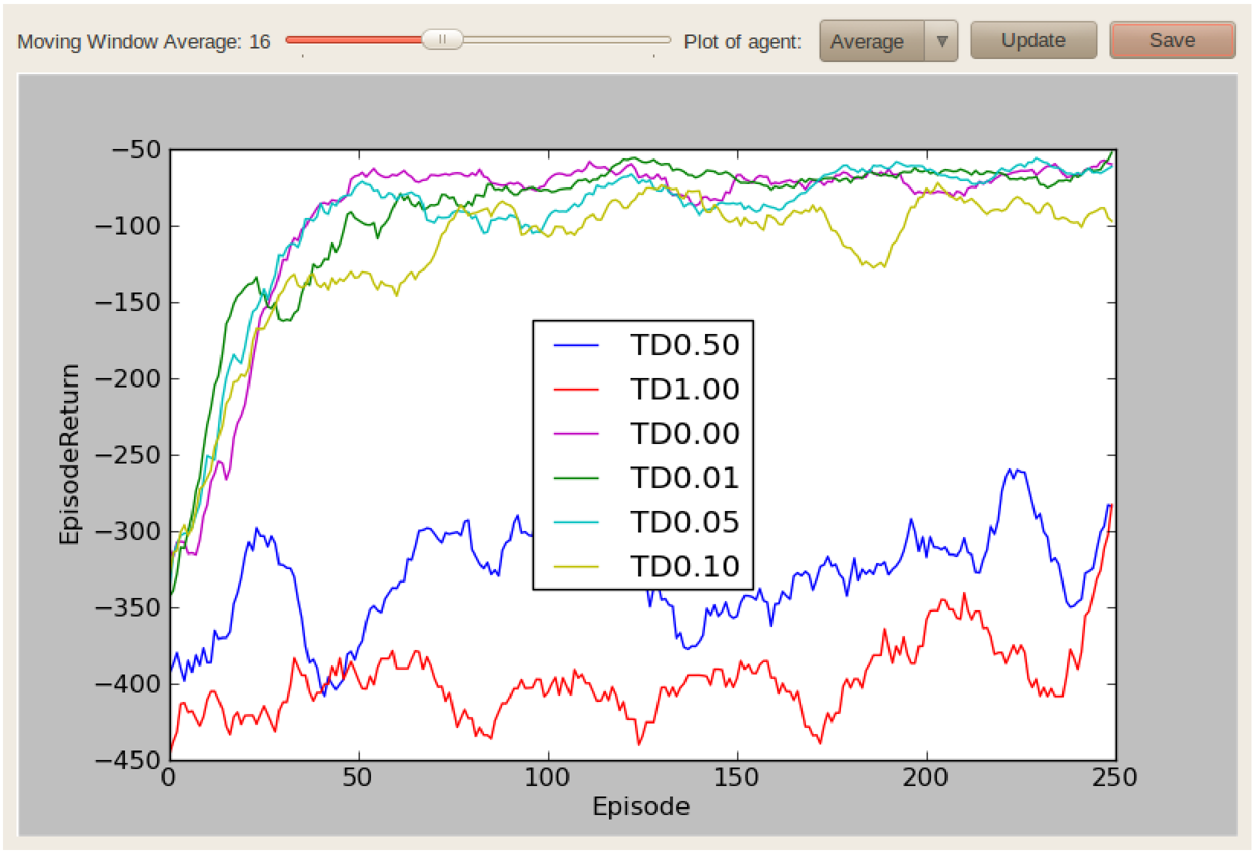
```
"""

worldConfTemplate = yaml.load(worldConfTemplateYaml)

# Create experiment configuration
# Experiment is conducted concurrently with 8 parallel processes (if enough
# cores are available). Each run consists of 250 episodes and 5 runs are
# conducted per world
experimentConf = {"concurrency": "Concurrent",
                  "episodesPerRun": '250',
                  "parallelProcesses": '8',
                  "runsPerWorld": '5',
                  "worlds" : dict()}

# Change exploration ratio of worlds
for epsilon in [0.0, 0.01, 0.05, 0.1, 0.5, 1.0]:
    worldConf = copy.deepcopy(worldConfTemplate)
    worldConf['agent']['configDict']['epsilon'] = epsilon
    experimentConf["worlds"]["TD%.2f"%epsilon] = worldConf

# Run the experiment (this may take some time)
mmlf.runExperiment(experimentConf)

# Give directories more meaningful names
for worldName, worldConf in experimentConf["worlds"].items():
    # Determine log directory into which the results of a specific world
    # have been stored. This is copy-pasted from world.py
    envConfig = worldConf['environment']
    agentConfig = worldConf['agent']
    confStr = "%s%s%s%s" % (envConfig["moduleName"],
                            sorted(zip(envConfig["configDict"].keys(),
                                       envConfig["configDict"].values())),
                            agentConfig["moduleName"],
                            sorted(zip(agentConfig["configDict"].keys(),
                                       agentConfig["configDict"].values())))

    directoryName = str(abs(hash(confStr)))

    # Rename directories
    shutil.move(os.sep.join([mmlf.getRWPath(), "logs", worldConf['worldPackage'], directoryName]),
                os.sep.join([mmlf.getRWPath(), "logs", worldConf['worldPackage'], worldName]))
```

You may execute this script with the following command (but be aware that this experiment may take some hours, in particular on slower machines):

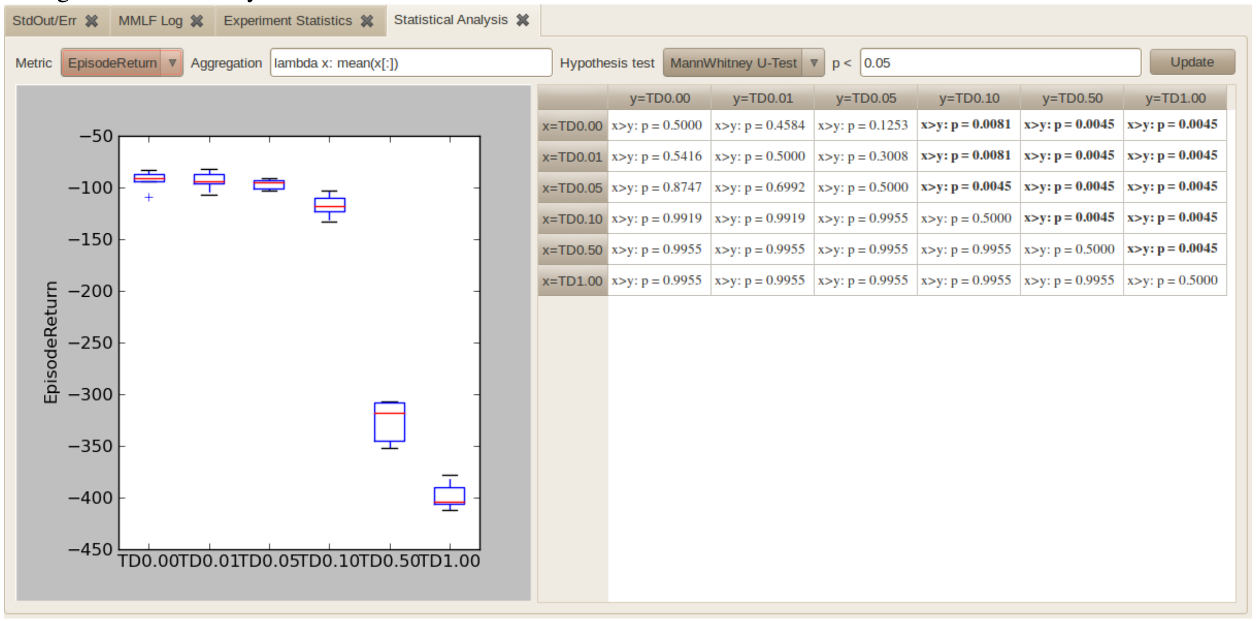
```
python scripted_experiment.py
```

Instead of executing the experiment, you can also download the results that have been obtained in one execution `here`. Now, you may load the results into the MMLF Experimenter (using “Load Experiment Results”) and analyze them.

For instance, plotting the average performance of an agent after applying a moving window average of length 16:



Doing a statistical analysis of the results:



Metric: EpisodeReturn
Aggregation: lambda x: mean(x[:])
Hypothesis test: MannWhitney U-Test
p < 0.05

	y=TD0.00	y=TD0.01	y=TD0.05	y=TD0.10	y=TD0.50	y=TD1.00
x=TD0.00	x>y: p = 0.5000	x>y: p = 0.4584	x>y: p = 0.1253	**x>y: p = 0.0081**	**x>y: p = 0.0045**	**x>y: p = 0.0045**
x=TD0.01	x>y: p = 0.5416	x>y: p = 0.5000	x>y: p = 0.3008	**x>y: p = 0.0081**	**x>y: p = 0.0045**	**x>y: p = 0.0045**
x=TD0.05	x>y: p = 0.8747	x>y: p = 0.6992	x>y: p = 0.5000	**x>y: p = 0.0045**	**x>y: p = 0.0045**	**x>y: p = 0.0045**
x=TD0.10	x>y: p = 0.9919	x>y: p = 0.9919	x>y: p = 0.9955	x>y: p = 0.5000	**x>y: p = 0.0045**	**x>y: p = 0.0045**
x=TD0.50	x>y: p = 0.9955	x>y: p = 0.9955	x>y: p = 0.9955	x>y: p = 0.9955	x>y: p = 0.5000	**x>y: p = 0.0045**
x=TD1.00	x>y: p = 0.9955	x>y: p = 0.9955	x>y: p = 0.9955	x>y: p = 0.9955	x>y: p = 0.9955	x>y: p = 0.5000

2.3 Logging

2.3.1 Basic logging

During execution of a world (a "run"), the MMLF stores several information about the agent's performance during learning. This information is stored into a subdirectory of the MMLF RW area (the one created during your first run of the MMLF, typically ~/.mmlf). For each run, a new subdirectory is created, for instance for a run in the maze2d environment, this subdirectory might have the path "maze2d/1854587605103575641/20110415_12_19_46_57_4547690801476827566". The first component of this path is the world package name, the second component the hash value of the joint agent and environment configuration, and the last component consists of the start time of the run and the thread and process id of this run. While this path appears to be overly complex at first glance, it has the following advantages: Several runs of the same world (the same configuration) are automatically stored in the same subdirectory of the world log directory such that it is easier to group many runs of the MMLF into groups of runs with the same world configuration. Since this grouping is done based on the hash value of the actual configuration, one can not mistakenly group together run with different parameter values. Furthermore, using start time and thread and process id as name of the run directory ensures that each run gets assigned automatically a unique directory even when several runs are conducted concurrently.

The following information are stored:

- The world configuration file is stored into the matching subdirectory (maze2d/1854587605103575641 in the example)
- Performance measurements of the agent (e.g. reward obtained per episode, episode length) are typically stored by the environment. This measurements go into a subdirectory with the environment class' name ("Maze2dEnvironment" for instance) and go into files with the ending (".fso"). These files are structured as follows:

  ```
  0        1620
  1        1188
  2        1555
  ...
  ```

  The first number in a line corresponds to a time stamp (typically the episode or step index), while the second one correponds to the agent's performance at this time (for instance the reward accrued during this episode). These files can then later be plotted and analysed using the MMLF experimenter (see *Evaluating experiments*). Which performance measures are stored depends on which "FloatStreamObservables" are defined in the environment (see *Writing an environment*). For FloatStreamObservables, one fso file is created. The name of this fso file corresponds to the observables internal name.

- Plots of internal components of the agent like policies, models, value function etc. This information is typically stored by the agent into a subdirectory named according to the agent's name (e.g. TDLambdaAgent). The plots go into a further subdirectory of this agent subdirectory named according to the agent's components (e.g. greedypolicy, optimalvaluefunction etc.). In this directory, pdf-files with the name "episode_00009.pdf", episode_00019.pdf", ... are generated, where "episode_00009.pdf" contains a plot of the component after the 9th episodes. Which internal components are plotted and with which frequency depends on the monitor's configuration (see below). The generated graphics are similar to those generated by the viewers (see *Viewers*).
- The policy followed by the agent at a certain time. This policy is stored in a serialized form (pickled). It goes into the subdirectory "policy" of the agent log directory. The files "policy_0", "policy_1", ... contain the policy folloewd by the agent at the end of episode 0, 1,...

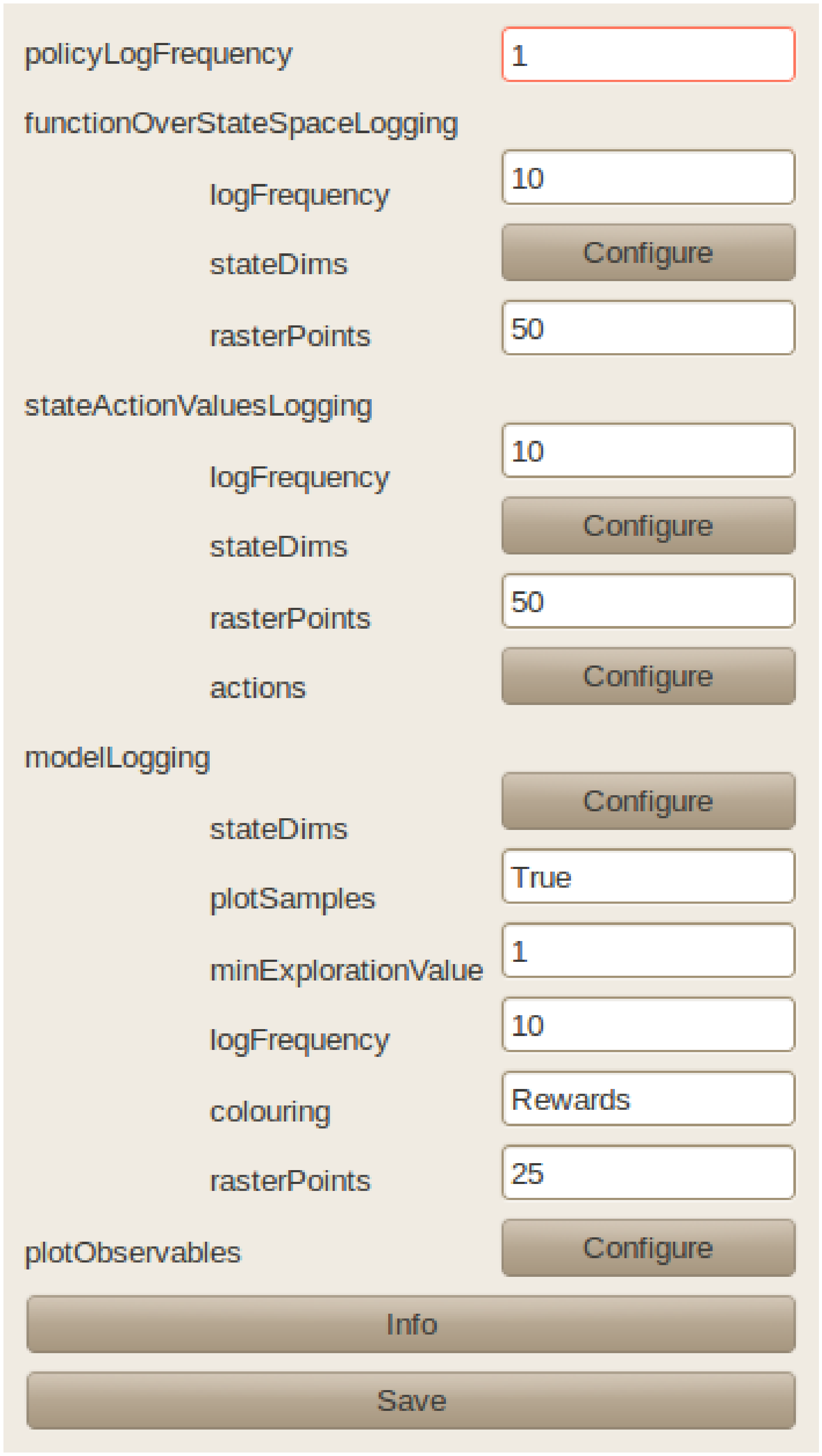
2.3.2 Monitor

The Monitor controls which information is automatically stored during a run of the MMLF. This information is stored in the RV

- performance measures of an agent
- pickling of an agent's policy
- creating periodically plots of selected StateActionValuesObservable
- creating periodically plots of selected FunctionOverStateSpaceObservable
- creating periodically plots of selected ModelObservable

The monitor can either be configured in the MMLF Explorer GUI (see image below) or in the world configuration file when running MMLF in a non-graphical way:

```
monitor:
 plotObservables: All
 policyLogFrequency: 1
 functionOverStateSpaceLogging:
   active : True
   logFrequency: 10
   rasterPoints: 50
   stateDims: null
 modelLogging:
   active : True
   colouring: Rewards
   logFrequency: 10
   minExplorationValue: 1
   plotSamples: true
   rasterPoints: 25
   stateDims: null
 stateActionValuesLogging:
   active : True
   actions: null
   logFrequency: 10
   rasterPoints: 50
   stateDims: null
```



policyLogFrequency	1
functionOverStateSpaceLogging	
logFrequency	10
stateDims	Configure
rasterPoints	50
stateActionValuesLogging	
logFrequency	10
stateDims	Configure
rasterPoints	50
actions	Configure
modelLogging	
stateDims	Configure
plotSamples	True
minExplorationValue	1
logFrequency	10
colouring	Rewards
rasterPoints	25
plotObservables	Configure

See the automatically generated documentation of the Monitor class below for an explanation of the different parameters. When specifying a world using a world configuration file, one may skip the functionOverStateSpaceLogging, modelLogging, and stateActionValuesLogging blocks. This causes the MMLF to not store any plots for the corresponding observables.

Note: If a world configuration file is loaded in which one of the blocks functionOverStateSpaceLogging, modelLogging, and stateActionValuesLogging is skipped, it is not yet possible to add it via the GUI configuration dialog for the monitor.

---

class `framework.monitor.`**`Monitor`**(*world*, *configDict*)

Monitor of the MMLF.

The monitor supervises the execution of a world within the MMLF and stores certain selected information periodically. It always stores the values a FloatStreamObservable takes on into a file with the suffix "fso". For other observables (FunctionOverStateSpaceObservable, StateActionValuesObservable, ModelObservable) a plot is generated and stored into files if this is specified in the monitor's config dict (using functionOverStateSpaceLogging, stateActionValuesLogging, modelLogging).

CONFIG DICT

policyLogFrequency : Frequency of storing the agent's policy in a serialized version to a file. The policy is stored in the file policy_x in the subdirectory "policy" of the agent's log directory where x is the episodes' numbers.

plotObservables : The names of the observables that should be stored to a file. If "All", all observables are stored. Defaults to "All" (also if plotObservables is not specified in a config file).

stateActionValuesLogging : Configuration of periodically plotting StateActionValuesObservables. Examples for StateActionValuesObservable are state-action value functions or stochastic policies. The plots are stored in the file episode_x.pdf in a subdirectory of the agent's log directory with the observable's name where x is the episodes' numbers.

functionOverStateSpaceLogging : Configuration of periodically plotting FunctionOverStateSpaceObservables. Examples for FunctionOverStateSpaceObservable are state value functions or deterministic policies. The plots are stored in the file episode_x.pdf in a subdirectory of the agent's log directory with the observable's name where x is the episodes' numbers.

modelLogging : Configuration of periodically plotting ModelObservables. Examples for ModelObservables are models. The plots are stored in the file episode_x.pdf in a subdirectory of the agent's log directory with the observable's name where x is the episodes' numbers.

active : Whether the respective kind of logging is activated

logFrequency : Frequency (in episodes) of creating a plot based on the respective observable and storing it to a file.

stateDims : The state space dimensions that are varied in the plot. All other state space dimensions are kept constant. If None, the stateDims are automatically deduced. This is only possible under specific conditions (2d state space)

actions : The actions for which separate plots are created for each StateActionValuesObservable.

rasterPoints : The resolution (rasterPoint*rasterPoint) of the plot in continuous domain.

colouring : The background colouring of a model plot. Can be either "Rewards" or "Exploration". If "Rewards", the reward predicted by the model is used for as background, while for "Exploration", each state-action pair tried at least minExplorationValue times is coloured with one colour, the others with an other colour.

plotSamples : If true, the state-action pairs that have been observed are plotted into the model-plot. Otherwise the model's predictions.

minExplorationValue : If the colouring is "Exploration", each state-action pair tried at least minExplorationValue times is coloured with one colour, the others with an other colour.

2.4 State and Action Spaces

Modules for state and action spaces

State and action spaces define the range of possible states the agent might perceive and the actions that are available to the agent. These spaces are dict-like objects that map dimension names (the dict keys) to dimension objects that contain information about this dimension. The number of items in this dict-like structure is the dimensionality of the (state/action) space.

2.4.1 Single dimension of a space

class `framework.spaces.`**`Dimension`**(*dimensionName*, *dimensionType*, *dimensionValues*, *limitType=None*)

A single dimension of a (state or action) space

A dimension is either continuous or discrete. A "discrete" dimension might take on only a finite, discrete number of values.

For instance, consider a dimension of a state space describing the color of a fruit. This dimension take on the values "red", "blue", or "green". In contrast, consider a second "continuous" dimension, e.g. the weight of a fruit. This weight might be somewhere between 0g and 1000g. If we allow any arbitrary weight (not only full gramms), the dimension is truly continuous.

This properties of a dimension can be checked using the method:

- *isDiscrete* : Returns whether the respective dimension is discrete
- *isContinuous* : Returns whether the respective dimension is continuous
- ***getValueRanges*** [Returns the allowed values a continuous! dimension] might take on.
- ***getValues*** [Returns the allowed values a discrete! dimension] might take on.

2.4.2 Base class for state and action spaces

class `framework.spaces.`**`Space`**

Base class for state and action spaces.

Class which represents the state space of an environment or the action space of an agent.

This is essentially a dictionary whose keys are the names of the dimensions, and whose values are *Dimension* objects.

`addContinuousDimension`(*dimensionName*, *dimensionValues*, *limitType='soft'*)

Add the named continuous dimension to the space.

dimensionValues is a list of (rangeStart, rangeEnd) 2-tuples which define the valid ranges of this dimension. (i.e. [(0, 50), (75.5, 82)])

If limitType is set to "hard", then the agent is responsible to check that the limits are not exceeded. When it is set to "soft", then the agent should not expect that all the values of this dimension will be strictly within the bounds of the specified ranges, but that the ranges serve as an approximate values of where the values will be (i.e. as [mean-std.dev., mean+std.dev] instead of [absolute min. value, absolute max. value])

addDiscreteDimension (*dimensionName*, *dimensionValues*)
Add the named continuous dimension to the space.

dimensionValues is a list of strings representing possible discrete states of this dimension. (i.e. ["red", "green", "blue"])

addOldStyleSpace (*oldStyleSpace*, *limitType='soft'*)
Takes an old-style (using the old format) space dictionary, and adds its dimensions to this object.

getDimensionNames ()
Return the names of the space dimensions

getDimensions ()
Return the names of the space dimensions

getNumberOfDimensions ()
Returns how many dimensions this space has

hasContinuousDimensions ()
Return whether this space has continuous dimensions

hasDiscreteDimensions ()
Return whether this space has discrete dimensions

2.4.3 State spaces

class framework.spaces.**StateSpace**
Specialization of Space for state spaces.

For instance, a state space could be defined as follows:

```
{ "color": Dimension(dimensionType = "discrete",
                     dimensionValues = ["red","green", "blue"]),
  "weight": Dimension(dimensionType = "continuous",
                      dimensionValues = [(0,1000)]) }
```

This state space has two dimensions ("color" and "weight"), a discrete and a continuous one. The discrete dimension "color" can take on three values ("red","green", or "blue") and the continuous dimension "weight" any value between 0 and 1000.

A valid state of the state space defined above would be:

```
s1 = {"color": "red", "weight": 300}
```

Invalid states (s2 since the color is invalid and s3 since its weight is too large):

```
s2 = {"color": "yellow", "weight": 300}
s3 = {"color": "red", "weight": 1300}
```

The class provides additional methods for checking if a certain state is valid according to this state space (*isValidState*) and to scale a state such that it lies within a certain interval (*scaleState*).

2.4.4 Action spaces

class framework.spaces.**ActionSpace**
Specialization of Space for action spaces.

For instance, an action space could be defined as follows:

```
{ "gasPedalForce": ("discrete", ["low", "medium", "floored"]),
  "steeringWheelAngle": ("continuous", [(-120,120)]) }
```

This action space has two dimensions ("gasPedalForce" and "steeringWheelAngle"), a discrete and a continuous one. The discrete dimension "gasPedalForce" can take on three values ("low","medium", or "floored") and the continuous dimension "steeringWheelAngle" any value between -120 and 120.

A valid action according to this action space would be:

```
a1 = {"gasPedalForce": "low", "steeringWheelAngle": -50}
```

Invalid actions (a2 since the gasPedalForce is invalid and s3 since its steeringWheelAngle is too small):

```
a2 = {"gasPedalForce": "extreme", "steeringWheelAngle": 30}
a3 = {"gasPedalForce": "medium", "steeringWheelAngle": -150}
```

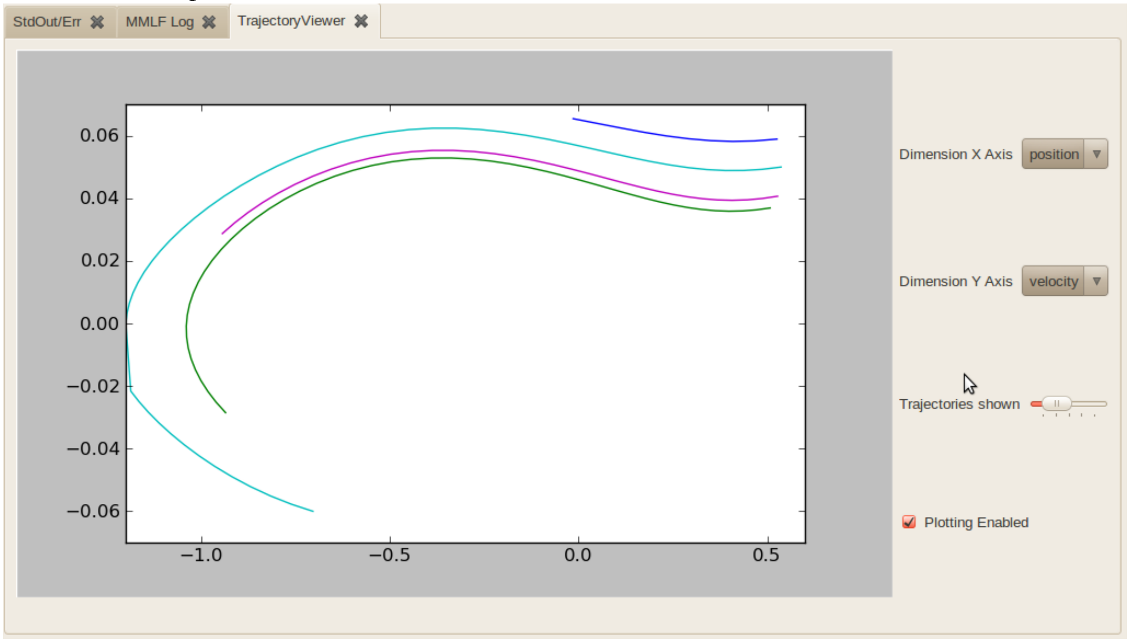
The class provides additional methods for discretizing an action space (*discretizedActionSpace*) and to return a list of all available actions (*getActionList*).

2.5 Viewers

Viewers can be used within the MMLF Explorer GUI to monitor online the progress of a learning agent while he is interacting with the environment.

2.5.1 General-purpose

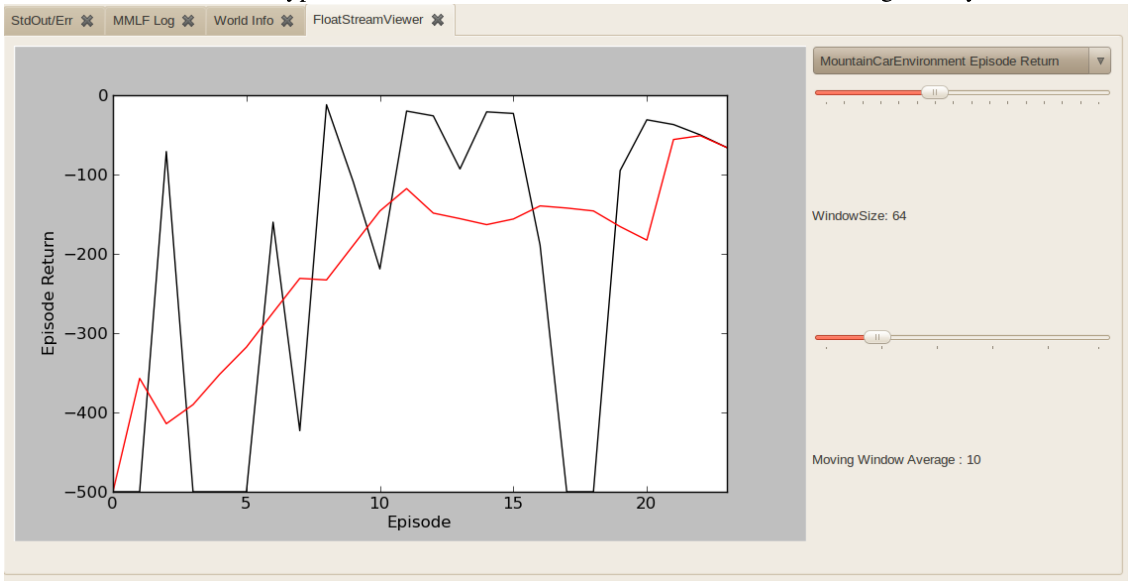
- **TrajectoryViewer** The TrajectoryViewer allows to visualize trajectories (the pathes an agent has taken through the state space in an episode). For environments with more than two dimensions, one can specify which two state dimensions are shown. The other dimensions are ignored. Onecan also choose how many trajectories (i.e. episodes) are shown at a time.



Dimension X Axis: position

Dimension Y Axis: velocity

- **FloatStreamViewer** The FloatStreamViewer allows to show how a scalar metric (like the return per episode, the episode lentgth etc.) change over time. It shows the development of this metric as well as a smoothed moving-window-average of the metric. It allows to configure which window of time (e.g. how many episodes) are shown at a time and over how many values the moving-window-average is computed. Any observable of type FloatStreamObservable defined in environment or agent may be visualized.

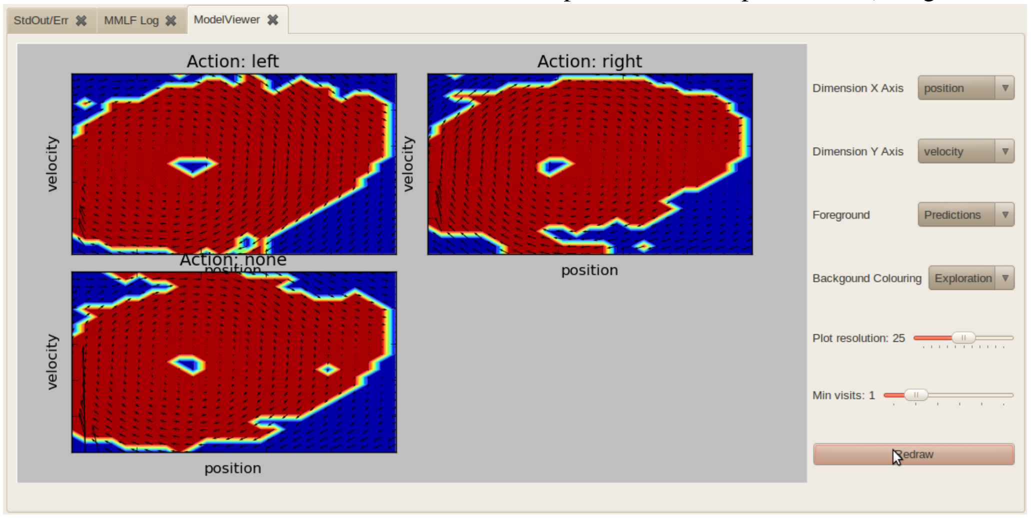


MountainCarEnvironment Episode Return

WindowSize: 64

Moving Window Average : 10

- **ModelViewer** The ModelViewer is available when using an agent which internally learns a model of its environment. It currently supports only continuous state spaces and is most useful when the state space is two-dimensional (e.g. in mountain-car). For higher dimensional state spaces, one has to select two dimensions ("Dimension X axis" and "Dimension Y axis") which are plotted. For the remaining dimensions, a default value is assumed. One can either plot the model's predictions (Foreground=Predictions):



Dimension X Axis: position

Dimension Y Axis: velocity

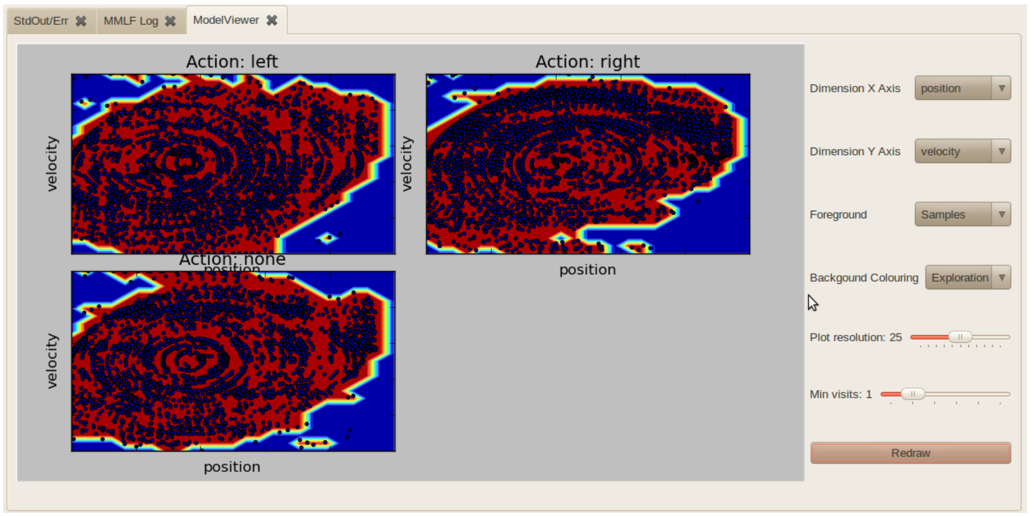
Foreground: Predictions

Backgound Colouring: Exploration

Plot resolution: 25

Min visits: 1

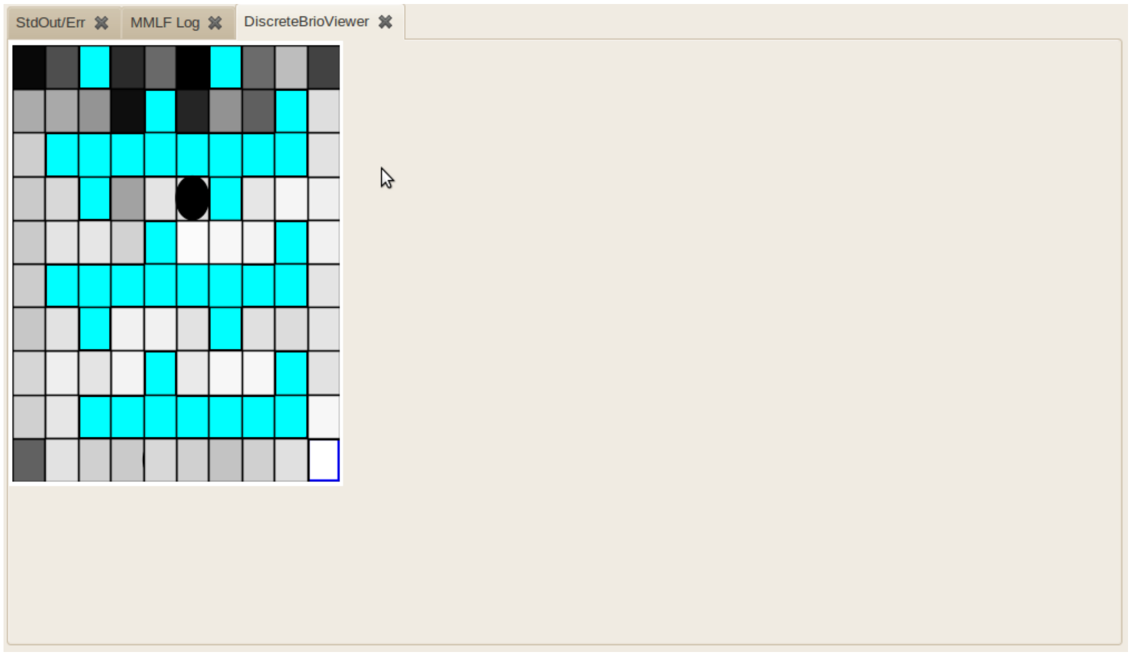
or plot which state-action combinations have been explored by the agent (blue dots, Foreground=Samples):



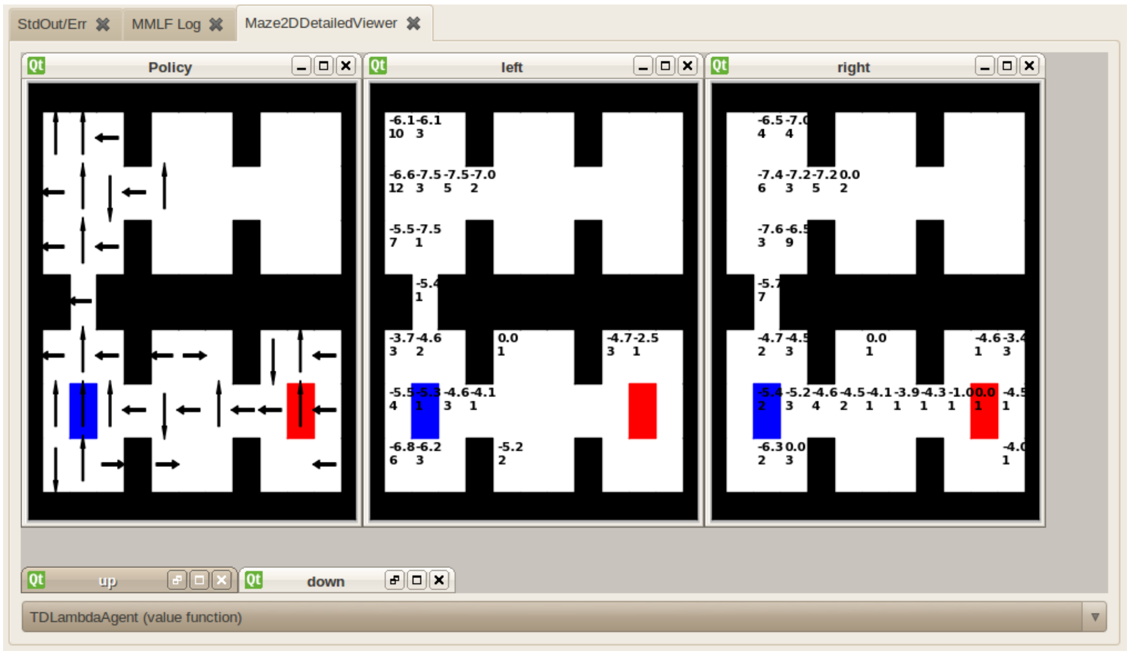
Dimension X Axis: position
Dimension Y Axis: velocity
Foreground: Samples
Backgound Colouring: Exploration
Plot resolution: 25
Min visits: 1

In the background, one can either plot the expected immediate reward for a state-action combination (color-coded) or the exploration value (how "explored" a certain region of the state-action space is). For the latter, one can define a threshold "Min visits" which specifies how many visits of a state-action pair are required to be considered as explored. Since the state space is continuous, visits are usually distributed over a local neighborhood. Furthermore, "Plot resolution" defines the resolution N of the 2d grid (N*N nodes) which is layed over the 2d state space. The higher the resolution, the slower the plotting becomes.

2.5.2 Environment specific

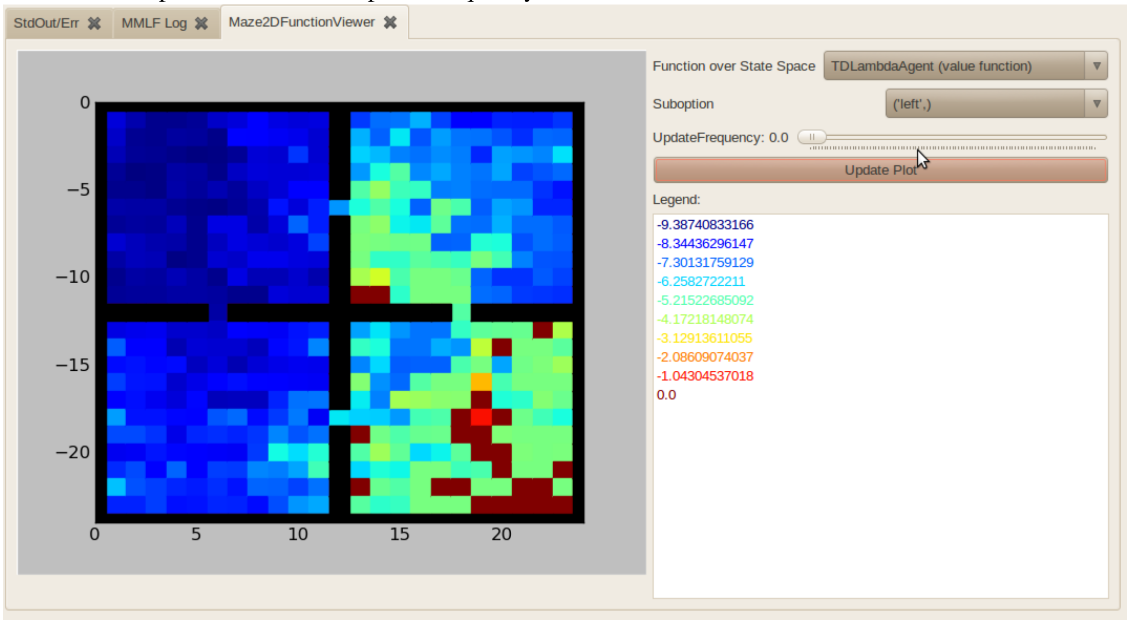


- **DiscreteBrioViewer**
- **Maze2DDetailedViewer** Allows to visualize the policy as well as the Q-function for all available actions in the Maze2D environment. In the Q-function viewer, a separate window is shown for each action. For each state, two values are given: The upper is the current Q-value and the lower one is the number of visits of the states. Note: The number of visits is the number of visits since adding this viewer; thus, it only equals the total number of visits if the viewer is created before starting the world.



TDLambdaAgent (value function)

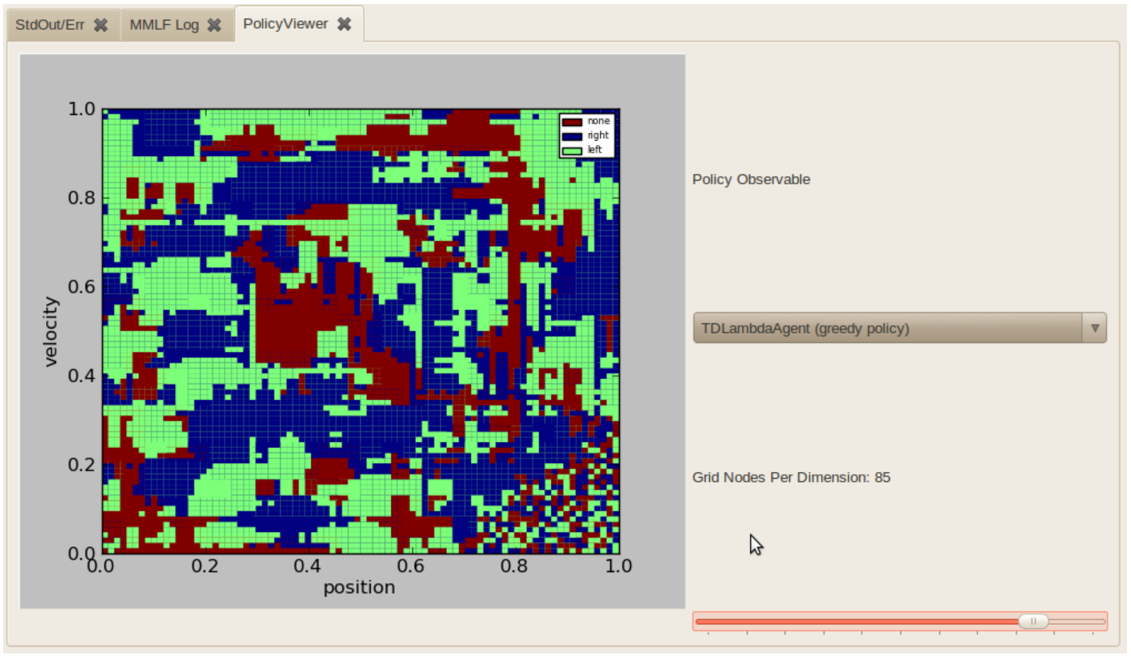
- **Maze2DFunctionViewer** Allows to visualize any FunctionOverStateSpaceObservable or StateActionValuesObservable defined in an agent that is used in combination with the Maze2D environment. For the StateActionValuesObservable, the action to be shown must be selected (called suboption in the GUI). This viewer can either ba updated automatically with a certain frequence, or only manually by pressing “Update Plot” when UpdateFrequency is 0.



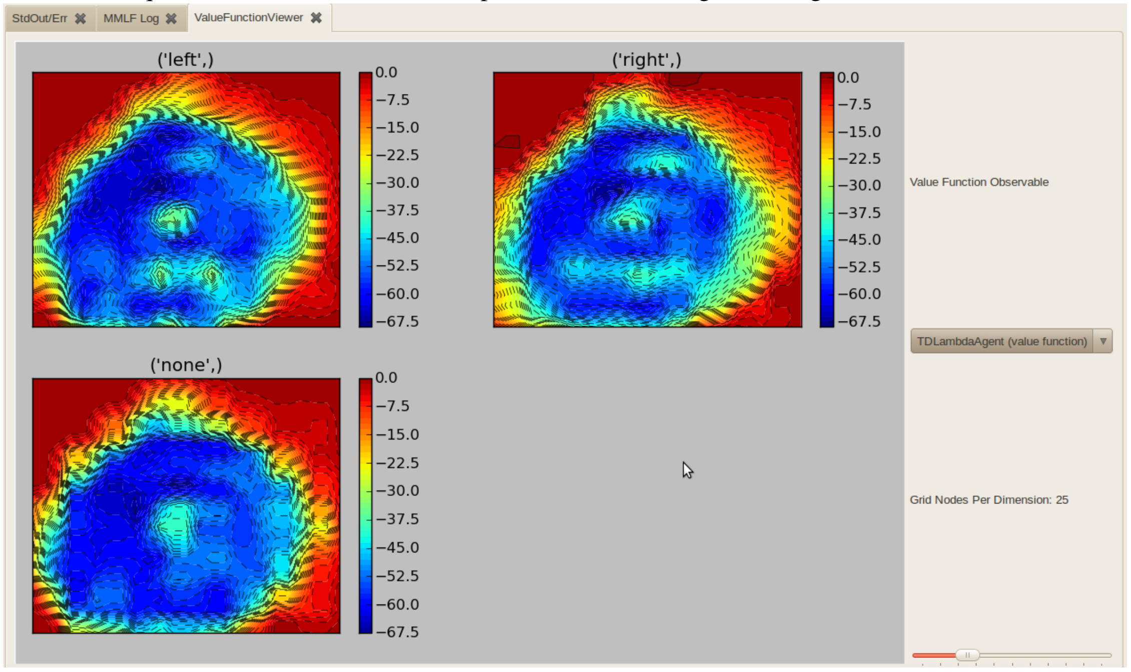
Function over State Space: TDLambdaAgent (value function)
Suboption: ('left',)
UpdateFrequency: 0.0

Legend:
-9.38740833166
-8.34436296147
-7.30131759129
-6.2582722211
-5.21522685092
-4.17218148074
-3.12913611055
-2.08609074037
-1.04304537018
0.0

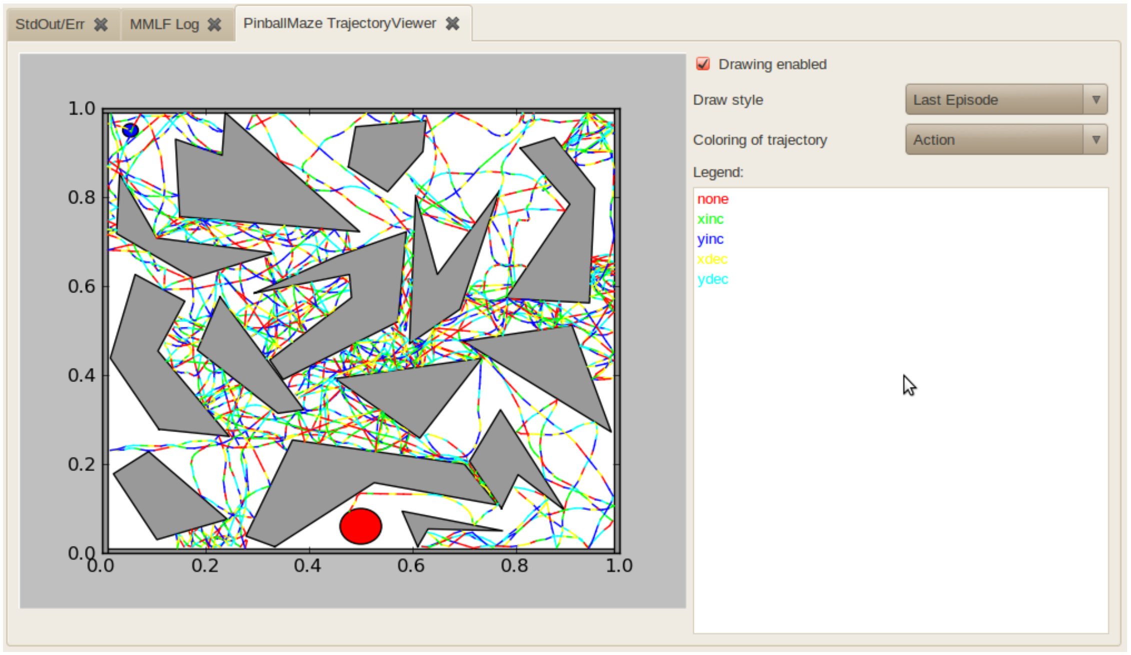
- **MountainCarPolicyViewer** Displays the policy (actually any FunctionOverStateSpaceObservable) over the two dimensional mountain car state space. The policy is evaluated for a grid of N*N equidistand states over the state space. N can be configured using “Grid Nodes Per Dimension”.



- **MountainCarValueFunctionViewer** Displays the value function (actually any StateActionValuesObservable) over the two dimensional mountain car state space. The value function is evaluated for a grid of N*N equidistand states over the state space. N can be configured using "Grid Nodes Per Dimension".


StdOut/Err
MMLF Log
ValueFunctionViewer
('left',)
('right',)
('none',)
Value Function Observable
TDLambdaAgent (value function)
Grid Nodes Per Dimension: 25

- **PinballMazeTrajectoryViewer** Displays the path of the ball through the pinball maze. Drawing can either be "Last Episode", which would show the trajectory taken in the last completed episode, "Current Position", which only shows the current position of the ball, or "Online (All)" which shows the whole trajectory taken by the agent in all episodes since drawin has been started. The trajectory can be coloured either according to the taken action, the obtained reward or the value of the value function.



Drawing enabled

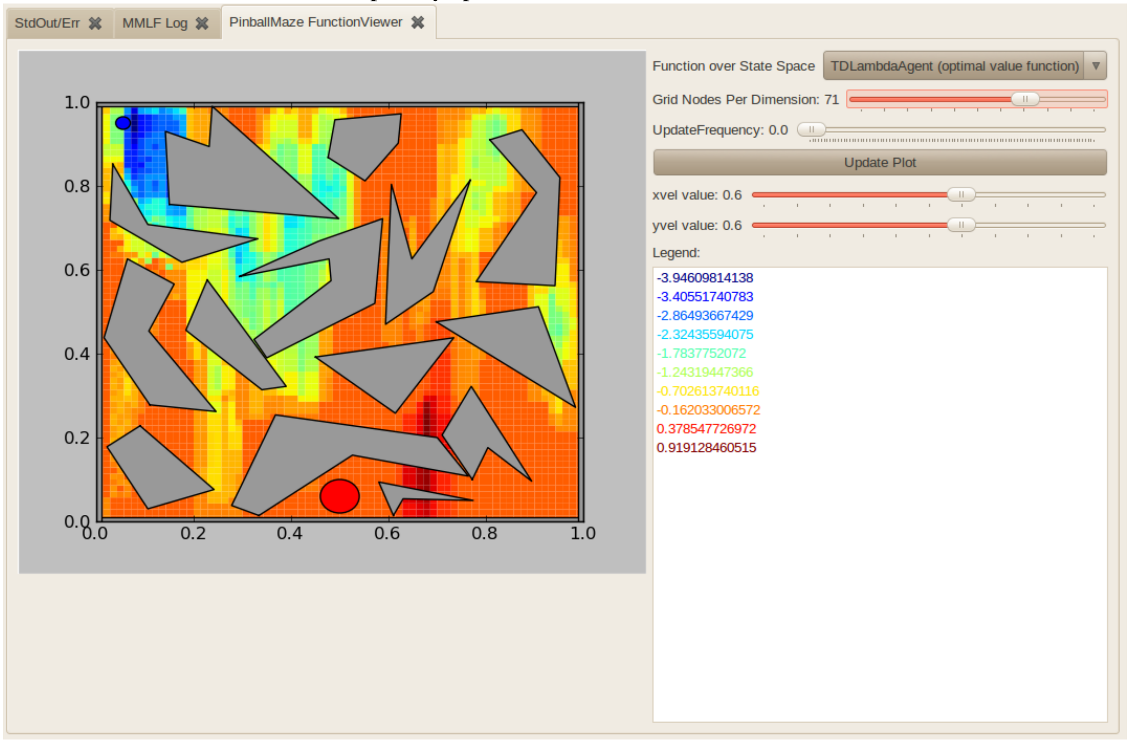
Draw style: Last Episode

Coloring of trajectory: Action

Legend:

none
xinc
yinc
xdec
ydec

- **PinballMazeFunctionViewer** Shows a two-dimensional slice of a FunctionOverStateSpaceObservable in the pinball domain. The two dimensions shown are the x and y position. The values for the two velocity dimensions must be explicitly specified.



Function over State Space: TDLambdaAgent (optimal value function)

Grid Nodes Per Dimension: 71

UpdateFrequency: 0.0

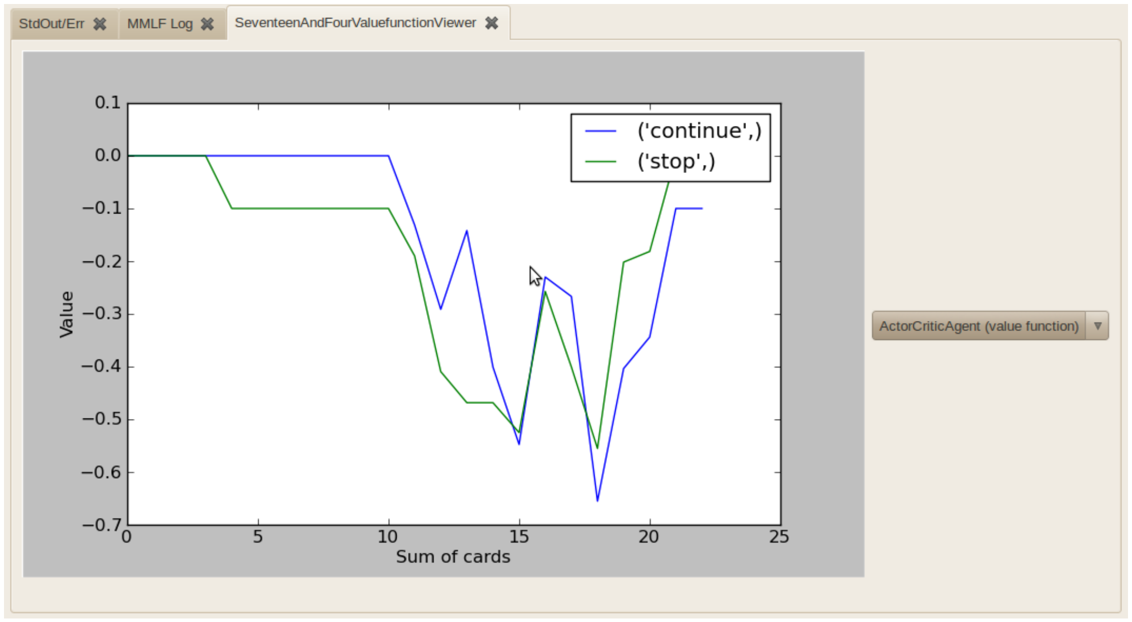
xvel value: 0.6

yvel value: 0.6

Legend:

-3.94609814138
-3.40551740783
-2.86493667429
-2.32435594075
-1.7837752072
-1.24319447366
-0.702613740116
-0.162033006572
0.378547726972
0.919128460515

- **SeventeenAndFourValueFunctionViewer** Visualizes an arbitrary StateActionValuesObservable (e.g. an agent's value function) in the Seventeen and Four domain.


StdOut/Err
MMLF Log
SeventeenAndFourValuefunctionViewer
('continue',)
('stop',)
Value
Sum of cards
0.1
0.0
−0.1
−0.2
−0.3
−0.4
−0.5
−0.6
−0.7
0
5
10
15
20
25
ActorCriticAgent (value function)

CHAPTER

THREE

API-DOCUMENTATION

3.1 MMLF package interface

Maja Machine Learning Framework

The Maja Machine Learning Framework (MMLF) is a general framework for problems in the domain of Reinforcement Learning (RL). It provides a set of RL related algorithms and a set of benchmark domains. Furthermore it is easily extensible and allows to automate benchmarking of different agents.

Among the RL algorithms are TD(lambda), DYNA-TD, CMA-ES, Fitted R-Max, and Monte-Carlo learning. MMLF contains different variants of the maze-world and pole-balancing problem class as well as the mountain-car testbed.

Further documentation is available under http://mmlf.sourceforge.net/

Contact the mailing list MMLF-support@lists.sourceforge.net if you have any questions.

exception `mmlf.`**`RWAreaInitializationFailedException`**

Exception raised if initialization of MMLF RW area failed.

`mmlf.`**`getRWPath`**`()`

Return the path of the MMLF RW area to be used.

This defaults to the path given by the environment variable $MMLF_RW_PATH if this variable exists. Otherwise, $HOME/.mmlf is used if the environment variable $HOME exists (typically under unix-like OS). Under Windows OS, $USERPROFILE/.mmlf is used instead. If None of these variable exists, an Exception of type RWAreaInitializationFailedException is raised.

`mmlf.`**`initializeRWArea`**`(`*rwPath=None*`)`

Initialize the RW area.

If *rwPath* is specified, the RW area located under that path is used. Otherwise, the standard RW area returned by *getRWPath()* is used.

If the required RW area does not exist, it is automatically created under the specified path. If config files in the RW area are missing, they are automatically restored.

`mmlf.`**`loadExperimentConfig`**`(`*experimentPath*`)`

Load the specification of an MMLF experiment.

Loads the specification of an MMLF experiment from the file `experiment_config.yaml` in *experimentPath*. Returns a dictionary containing the experiment's specification.

`mmlf.`**`loadWorld`**`(`*worldConfigObject*, *baseUserDir=None*, *useGUI=False*, *keepObservers=*[]`)`

Load the world specified in the *worldConfigObject*.

worldConfigObject must be a dictionary defining the configuration of a world. This dictionary must have the following keys:

> **worldPackage** : The name of the subpackage of `mmlf/world` in which the world is located.
>
> **environment** : Definition of the environment. Must be a dictionary with the keys "moduleName" (the python module in which the environment is defined; looked up in the subdirectory `environments` of the world package) and "configDict" (the config dictionary of the environment, must have the same key-values as the environment's DEFAULT_CONFIG_DICT).
>
> **agents** : Definition of the agent. Must be a dictionary with the keys "moduleName" (the python module in which the agent is defined; looked up in `mmlf/agents` or in the subdirectory `agents` of the world package) and "configDict" (the config dictionary of the agent, must have the same key-values as the agent's DEFAULT_CONFIG_DICT).
>
> **monitor** : The configuration dictionary of the MMLF monitor.

You may take a look at the *Learn more about Experiments/Scripting experiments* section of the MMLF documentation for an example.

baseUserDir must be a MMLF BaseUserDirectory object (which represent the MMLF RW area internally). If none, a new BaseUserDirectory object is created.

useGUI indicates whether we are running a graphical user interface. If True, this causes the agent and environment of the world to create Viewers (and thus requires that PyQt4 is installed).

Per default, loadWorld() removes all existing observer objects since these stem from prior runs. Of observables should be kept, these can be passed via the parameter *keepObservers* (a list).

`mmlf.`**`loadWorldFromConfigFile`**`(`*configPath*, *useGUI=False*`)`

Load the world specified in the *configPath*.

This method loads the MMLF world specified in the YAML-file under *configPath*. *configPath* can either be a relative path (in which case it is looked up in the config directory of MMLF RW area or an absolute path. The created MMLF World object is returned.

useGUI indicates whether we are running a graphical user interface. If True, this causes the agent and environment of the world to create Viewers (and thus requires that PyQt4 is installed).

`mmlf.`**`runExperiment`**`(`*experimentConf*`)`

Execute the experiment specified in experimentConf.

experimentConf must be a dictionary defining the specification of the experiment. This dictionary must have the following keys:

> **concurrency** : "Concurrent" if several runs of the experiment should be executed at the same time, "Sequential" if one run after the other should be executed.
>
> **parallelProcesses** : If concurrent execution is requested, this parameter specifies how many runs can be run in parallel. This value is upper-bounded by the number of cores.
>
> **episodesPerRun** : Integer defining how many episodes each run of a world should last.
>
> **runsPerWorld** : How many independent runs should be executed for each world.
>
> **worlds** : A dictionary mapping world name to a world configuration dictionary. See *loadWorld* for more details about world configuration dictionaries.

You may take a look at the *Learn more about Experiments/Scripting experiments* section of the MMLF documentation for an example.

`mmlf.`**`setupConsoleLogging`**`(`**args*, ***kwargs*`)`

Initializes the logging of messages to the console.

`mmlf.`**`setupFileLogging`**(*logdir*, **args*, ***kwargs*)
Initializes the logging of messages into a log file in *logdir*.

3.2 Agents

Package that contains all available MMLF agents.

A list of all agents:

- Agent '*Model-based Direct Policy Search*' from module mbdps_agent implemented in class MBDPS_Agent.

  An agent that uses the state-action-reward-successor_state transitions to learn a model of the environment. It performs direct policy search (similar to the direct policy search agent using a black-box optimization algorithm to optimize the parameters of a parameterized policy) in the model in order to optimize a criterion defined by a fitness function. This fitness function can be e.g. the estimated accumulated reward obtained by this policy in the model environment. In order to enforce exploration, the model is wrapped for an RMax-like behavior so that it returns the reward RMax for all states that have not been sufficiently explored. RMax should be an upper bound to the actual achievable return in order to enforce optimism in the face of uncertainty.

- Agent '*Monte-Carlo*' from module monte_carlo_agent implemented in class MonteCarloAgent.

  An agent which uses Monte Carlo policy evaluation to optimize its behavior in a given environment.

- Agent '*Dyna TD*' from module dyna_td_agent implemented in class DynaTDAgent.

  Dyna-TD uses temporal difference learning along with learning a model of the environment and doing planning in it.

- Agent '*Temporal Difference + Eligibility*' from module td_lambda_agent implemented in class TDLambdaAgent.

  An agent that uses temporal difference learning (e.g. Sarsa) with eligibility traces and function approximation (e.g. linear tile coding CMAC) to optimize its behavior in a given environment

- Agent '*Policy Replay*' from module policy_replay_agent implemented in class PolicyReplayAgent.

  Agent which loads a stored policy and follows it without improving it.

- Agent '*Random*' from module random_agent implemented in class RandomAgent.

- Agent '*Actor Critic*' from module actor_critic_agent implemented in class ActorCriticAgent.

  This agent learns based on the actor critic architecture. It uses standard TD(lambda) to learn the value function of the critic. For this reason, it subclasses TDLambdaAgent. The main difference to TD(lambda) is the means for action selection. Instead of deriving an epsilon-greedy policy from its Q-function, it learns an explicit stochastic policy. To this end, it maintains preferences for each action in each state. These preferences are updated after each action execution according to the following rule:

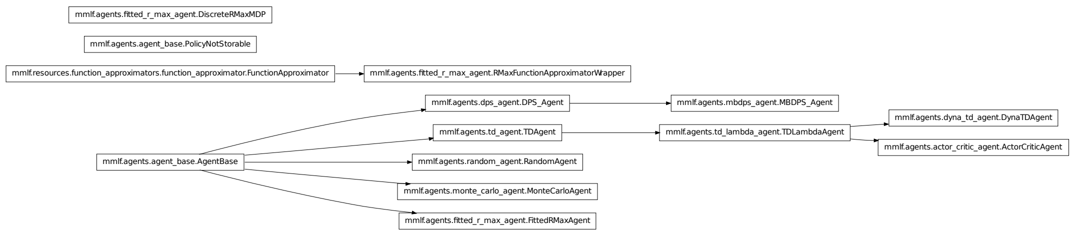
- Agent '*RoundRobin*' from module example_agent implemented in class ExampleAgent.

- Agent '*Direct Policy Search*' from module dps_agent implemented in class DPS_Agent.

  This agent uses a black-box optimization algorithm to optimize the parameters of a parametrized policy such that the accumulated (undiscounted) reward of the the policy is maximized.

- Agent '*Fitted-RMax*' from module fitted_r_max_agent implemented in class FittedRMaxAgent.

Fitted R-Max is a model-based RL algorithm that uses the RMax heuristic for exploration control, uses a fitted function approximator (even though this can be configured differently), and uses Dynamic Programming (boosted by prioritized sweeping) for deriving a value function from the model. Fitted R-Max learns usually very sample-efficient (meaning that a good policy is learned with only a few interactions with the environment) but requires a huge amount of computational resources.


mmlf.agents.fitted_r_max_agent.DiscreteRMaxMDP
mmlf.agents.agent_base.PolicyNotStorable
mmlf.resources.function_approximators.function_approximator.FunctionApproximator
mmlf.agents.fitted_r_max_agent.RMaxFunctionApproximatorWrapper
mmlf.agents.dps_agent.DPS_Agent
mmlf.agents.mbdps_agent.MBDPS_Agent
mmlf.agents.dyna_td_agent.DynaTDAgent
mmlf.agents.td_agent.TDAgent
mmlf.agents.td_lambda_agent.TDLambdaAgent
mmlf.agents.actor_critic_agent.ActorCriticAgent
mmlf.agents.agent_base.AgentBase
mmlf.agents.random_agent.RandomAgent
mmlf.agents.monte_carlo_agent.MonteCarloAgent
mmlf.agents.fitted_r_max_agent.FittedRMaxAgent

3.2.1 Agent base-class

Module for MMLF interface for agents

This module contains the AgentBase class that specifies the interface that all MMLF agents have to implement.

class `agents.agent_base.`**`AgentBase`**(*config*, *baseUserDir*, **args*, ***kwargs*)

MMLF interface for agents

Each agent that should be used in the MMLF needs to be derived from this class and implements the following methods:

Interface Methods

setStateSpace : Informs the agent of the environment's state space

setActionSpace : Informs the agent of the environment's action space

setState : Informs the agent of the environment's current state

giveReward : Provides a reward to the agent

getAction : Request the next action the agent want to execute

nextEpisodeStarted : Informs the agent that the current episode has terminated and a new one has started.

`getAction`()

Request the next action the agent want to execute

`getGreedyPolicy`()

Returns the optimal, greedy policy the agent has found so far

`giveReward`(*reward*)

Provides a reward to the agent

`nextEpisodeStarted`()

Informs the agent that a new episode has started.

`setActionSpace`(*actionSpace*)

Informs the agent about the action space of the environment

More information about action spaces can be found in *State and Action Spaces*

`setState` (*state*, *normalizeState=True*)

Informs the agent of the environment's current state.

If *normalizeState* is True, each state dimension is scaled to the value range(0,1). More information about (valid) states can be found in *State and Action Spaces*

`setStateSpace` (*stateSpace*)

Informs the agent about the state space of the environment

More information about state spaces can be found in *State and Action Spaces*

`storePolicy` (*filePath*, *optimal=True*)

Stores the agent's policy in the given file by pickling it.

Pickles the agent's policy and stores it in the file *filePath*. If the agent is based on value functions, a value function policy wrapper is used to obtain a policy object which is then stored.

If optimal==True, the agent stores the best policy it has found so far, if optimal==False, the agent stores its current (exploitation) policy.

3.2.2 Actor-critic

Agent that learns based on the actor-critic architecture.

This module contains an agent that learns based on the actor critic architecture. It uses standard TD(lambda) to learn the value function of the critic and updates the preferences of the actor based on the TD error.

class `agents.actor_critic_agent.`**`ActorCriticAgent`** (**args*, ***kwargs*)

Agent that learns based on the actor-critic architecture.

This agent learns based on the actor critic architecture. It uses standard TD(lambda) to learn the value function of the critic. For this reason, it subclasses TDLambdaAgent. The main difference to TD(lambda) is the means for action selection. Instead of deriving an epsilon-greedy policy from its Q-function, it learns an explicit stochastic policy. To this end, it maintains preferences for each action in each state. These preferences are updated after each action execution according to the following rule:

$$p(s,a) = p(s,a) + \delta, \tag{3.1}$$

where delta is the TD error

$$\delta = r + \gamma V(s') - V(s) \tag{3.2}$$

Action selection is based on a Gibbs softmax distribution:

$$\pi(s,a) = \frac{exp(\tau^{-1}p(s,a))}{\sum_{b \in A} exp(\tau^{-1}p(s,b))} \tag{3.3}$$

where tau is a temperature parameter.

Note that even though preferences are stored in a function approximator such that in principle, action preferences could be generalized over the state space, continuous state spaces are not yet supported. New in version 0.9.9: Added Actor-Critic agent

CONFIG DICT

gamma : The discount factor for computing the return given the rewards

lambda : The eligibility trace decay rate

tau : Temperature parameter used in the Gibbs softmax distribution for action selection

minTraceValue : The minimum value of an entry in a trace that is considered to be relevant. If the eligibility falls below this value, it is set to 0 and the entry is thus no longer updated

update_rule : Whether the learning is on-policy or off-policy.. Can be either "SARSA" (on-policy) or "WatkinsQ" (off-policy)

stateDimensionResolution : The default "resolution" the agent uses for every state dimension. Can be either an int (same resolution for each dimension) or a dict mapping dimension name to its resolution.

actionDimensionResolution : Per default, the agent discretizes a continuous action space in this number of discrete actions.

function_approximator : The function approximator used for representing the Q value function

preferences_approximator The function approximator used for representing the action preferences (i.e. the policy)

3.2.3 Direct Policy Search (DPS)

Agent that performs direct search in the policy space to find a good policy

This agent uses a black-box optimization algorithm to optimize the parameters of a parametrized policy such that the accumulated (undiscounted) reward of the the policy is maximized.

class `agents.dps_agent.`**`DPS_Agent`**`(`**args*, ***kwargs*`)`

Agent that performs direct search in the policy space to find a good policy

This agent uses a black-box optimization algorithm to optimize the parameters of a parametrized policy such that the accumulated (undiscounted) reward of the the policy is maximized.

CONFIG DICT

policy_search : The method used for search of an optimal policy in the policy space. Defines policy parametrization and internally used black box optimization algorithm.

3.2.4 Dyna TD

The Dyna-TD agent module

This module contains the Dyna-TD agent class. It uses temporal difference learning along with learning a model of the environment and is based on the Dyna architecture.

class `agents.dyna_td_agent.`**`DynaTDAgent`**`(`**args*, ***kwargs*`)`

Agent that learns based on the DYNA architecture.

Dyna-TD uses temporal difference learning along with learning a model of the environment and doing planning in it.

CONFIG DICT

gamma : The discount factor for computing the return given the rewards

epsilon : Exploration rate. The probability that an action is chosen non-greedily, i.e. uniformly random among all available actions

lambda : The eligibility trace decay rate

minTraceValue : The minimum value of an entry in a trace that is considered to be relevant. If the eligibility falls below this value, it is set to 0 and the entry is thus no longer updated

update_rule : Whether the learning is on-policy or off-policy.. Can be either "SARSA" (on-policy) or "WatkinsQ" (off-policy)

stateDimensionResolution : The default "resolution" the agent uses for every state dimension. Can be either an int (same resolution for each dimension) or a dict mapping dimension name to its resolution.

actionDimensionResolution : Per default, the agent discretizes a continuous action space in this number of discrete actions

planner : The algorithm used for planning, i.e. for optimizing the policy based on a learned model

model : The algorithm used for learning a model of the environment

function_approximator : The function approximator used for representing the Q value function

3.2.5 Fitted R-Max

Fitted R-Max agent

Fitted R-Max is a model-based RL algorithm that uses the RMax heuristic for exploration control, uses a fitted function approximator (even though this can be configured differently), and uses Dynamic Programming (boosted by prioritized sweeping) for deriving a value function from the model. Fitted R-Max learns usually very sample-efficient (meaning that a good policy is learned with only a few interactions with the environment) but requires a huge amount of computational resources.

class `agents.fitted_r_max_agent.`**`FittedRMaxAgent`**(**args*, ***kwargs*)

Fitted R-Max agent

Fitted R-Max is a model-based RL algorithm that uses the RMax heuristic for exploration control, uses a fitted function approximator (even though this can be configured differently), and uses Dynamic Programming (boosted by prioritized sweeping) for deriving a value function from the model. Fitted R-Max learns usually very sample-efficient (meaning that a good policy is learned with only a few interactions with the environment) but requires a huge amount of computational resources.

See Also:

Nicholas K. Jong and Peter Stone, "Model-based function approximation in reinforcement learning", in "Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems" Honolulu, Hawaii: ACM, 2007, 1-8, http://portal.acm.org/citation.cfm?id=1329125.1329242.

CONFIG DICT

gamma : The discount factor for computing the return given the rewards#

min_exploration_value : The agent explores in a state until the given exploration value (approx. number of exploratory actions in proximity of state action pair) is reached for all actions

RMax : An upper bound on the achievable return an agent can obtain in a single episode

planner : The algorithm used for planning, i.e. for optimizing the policy based on a learned model

model : The algorithm used for learning a model of the environment

function_approximator : The function approximator used for representing the Q value function

actionDimensionResolution : Per default, the agent discretizes a continuous action space in this number of discrete actions

3.2.6 Model-based Direct Policy Search (MBDPS)

The Model-based Direct Policy Search agent

This module contains an agent that uses the state-action-reward-successor_state transitions to learn a model of the environment. It performs than direct policy search (similar to the direct policy search agent using a black-box optimization algorithm to optimize the parameters of a parameterized policy) in the model in order to optimize a criterion defined by a fitness function. This fitness function can be e.g. the estimated accumulated reward obtained by this policy in the model environment. In order to enforce exploration, the model is wrapped for an RMax-like behavior so that it returns the reward RMax for all states that have not been sufficiently explored.

class `agents.mbdps_agent.`**`MBDPS_Agent`**(**args*, ***kwargs*)

The Model-based Direct Policy Search agent

An agent that uses the state-action-reward-successor_state transitions to learn a model of the environment. It performs direct policy search (similar to the direct policy search agent using a black-box optimization algorithm to optimize the parameters of a parameterized policy) in the model in order to optimize a criterion defined by a fitness function. This fitness function can be e.g. the estimated accumulated reward obtained by this policy in the model environment. In order to enforce exploration, the model is wrapped for an RMax-like behavior so that it returns the reward RMax for all states that have not been sufficiently explored. RMax should be an upper bound to the actual achievable return in order to enforce optimism in the face of uncertainty.

CONFIG DICT

gamma : The discount factor for computing the return given the rewards

planning_episodes : The number internally simulated episodes that are performed in one planning step

policy_search : The method used for search of an optimal policy in the policy space. Defines policy parametrization and internally used black box optimization algorithm.

model : The algorithm used for learning a model of the environment

3.2.7 Monte-Carlo

Monte-Carlo learning agent

This module defines an agent which uses Monte Carlo policy evaluation to optimize its behavior in a given environment

class `agents.monte_carlo_agent.`**`MonteCarloAgent`**(**args*, ***kwargs*)

Agent that learns based on monte-carlo samples of the Q-function

An agent which uses Monte Carlo policy evaluation to optimize its behavior in a given environment.

CONFIG DICT

gamma : The discount factor for computing the return given the rewards

epsilon : Exploration rate. The probability that an action is chosen non-greedily, i.e. uniformly random among all available actions

visit : Whether first (“first”) or every visit (“every”) is used in Monte-Carlo updates

defaultQ : The initially assumed Q-value for each state-action pair. Allows to control initial exploration due to optimistic initialization

3.2.8 Option

3.2.9 Random

MMLF agent that acts randomly

This module defines a simple agent that can interact with an environment. It chooses all available actions with the same probability.

This module deals also as an example of how to implement an MMLF agent.

class `agents.random_agent.`**`RandomAgent`**`(`**args*, ***kwargs*`)`

Agent that chooses uniformly randomly among the available actions.

3.2.10 Temporal Difference Learning

TD(0)

Agent based on temporal difference learning

This module defines a base agent for all kind of agents based on temporal difference learning. Most of these agents can reuse most methods of this agents and have to modify only small parts.

Note: The TDAgent cannot be instantiated by itself, it is a abstract base class!

class `agents.td_agent.`**`TDAgent`**`(`**args*, ***kwargs*`)`

A base agent for all kind of agents based on temporal difference learning. Most of these agents can reuse most methods of this agents and have to modify only small parts

Note: The TDAgent cannot be instantiated by itself, it is a abstract base class!

TD(lambda) - Eligibility Traces

Agent based on temporal difference learning with eligibility traces

This module defines an agent that uses temporal difference learning (e.g. Sarsa) with eligibility traces and function approximation (e.g. linear tile coding CMAC) to optimize its behavior in a given environment

class `agents.td_lambda_agent.`**`TDLambdaAgent`**`(`**args*, ***kwargs*`)`

Agent that implements TD(lambda) RL

An agent that uses temporal difference learning (e.g. Sarsa) with eligibility traces and function approximation (e.g. linear tile coding CMAC) to optimize its behavior in a given environment

CONFIG DICT

update_rule : Whether the learning is on-policy or off-policy.. Can be either "SARSA" (on-policy) or "WatkinsQ" (off-policy)

gamma : The discount factor for computing the return given the rewards

epsilon : Exploration rate. The probability that an action is chosen non-greedily, i.e. uniformly random among all available actions

epsilonDecay : Decay factor for the exploration rate. The exploration rate is multiplied with this value after each episode.

lambda : The eligibility trace decay rate

minTraceValue : The minimum value of an entry in a trace that is considered to be relevant. If the eligibility falls below this value, it is set to 0 and the entry is thus no longer updated

replacingTraces : Whether replacing or accumulating traces are used.

stateDimensionResolution : The default "resolution" the agent uses for every state dimension. Can be either an int (same resolution for each dimension) or a dict mapping dimension name to its resolution.

actionDimensionResolution : Per default, the agent discretizes a continuous action space in this number of discrete actions

function_approximator : The function approximator used for representing the Q value function

3.3 Environments

Package that contains all available MMLF world environments.

A list of all environments:

- Environment '*Maze Cliff*' from module maze_cliff_environment implemented in class MazeCliffEnvironment.

  In this maze, there are two alternative ways from the start to the goal state: one short way which leads along a dangerous cliff and one long but secure way. If the agent happens to step into the maze, it will get a huge negative reward (configurable via *cliffPenalty*) and is reset into the start state. Per default, the maze is deterministic, i.e. the agent always moves in the direction it chooses. However, the parameter *stochasticity* allows to control the stochasticity of the environment. For instance, when stochasticity is set to 0.01, the the agent performs a random move instead of the chosen one with probability 0.01.

- Environment '*Pinball 2D*' from module pinball_maze_environment implemented in class PinballMazeEnvironment.

  The pinball maze environment class.

- Environment '*Linear Markov Chain*' from module linear_markov_chain_environment implemented in class LinearMarkovChainEnvironment.

  The agent starts in the middle of this linear markov chain. He can either move right or left. The chain is not stochastic, i.e. when the agent wants to move right, the state is decreased with probability 1 by 1. When the agent wants to move left, the state is increased with probability 1 by 1 accordingly.

- Environment '*Maze 2D*' from module maze2d_environment implemented in class Maze2dEnvironment.

  A 2d maze world, in which the agent is situated at each moment in time in a certain field (specified by its (row,column) coordinate) and can move either upwards, downwards, left or right. The structure of the maze can be configured via a text-based config file.

- Environment '*Double Pole Balancing*' from module double_pole_balancing_environment implemented in class DoublePoleBalancingEnvironment.

  In the double pole balancing environment, the task of the agent is to control a cart such that two poles which are mounted on the cart stay in a nearly vertical position (to balance them). At the same time, the cart has to stay in a confined region.

- Environment '*Partial Observable Double Pole Balancing*' from module po_double_pole_balancing_environment implemented in class PODoublePoleBalancingEnvironment.

In the partially observable double pole balancing environment, the task of the agent is to control a cart such that two poles which are mounted on the cart stay in a nearly vertical position (to balance them). At the same time, the cart has to stay in a confined region. In contrast to the fully observable double pole balancing environment, the agent only observes the current position of cart and the two poles but not their velocities. This renders the problem to be not markovian.

- Environment '*Mountain Car*' from module mcar_env implemented in class MountainCarEnvironment.

  In the mountain car environment, the agent has to control a car which is situated somewhere in a valley between two hills. The goal of the agent is to reach the top of the right hill. Unfortunately, the engine of the car is not strong enough to reach the top of the hill directly from many start states. Thus, it has first to drive in the wrong direction to gather enough potential energy.

- Environment '*Single Pole Balancing*' from module single_pole_balancing_environment implemented in class SinglePoleBalancingEnvironment.

  In the single pole balancing environment, the task of the agent is to control a cart such that a pole which is mounted on the cart stays in a nearly vertical position (to balance it). At the same time, the cart has to stay in a confined region.

- Environment '*Seventeen and Four*' from module seventeen_and_four implemented in class SeventeenAndFourEnvironment.

  This environment implements a simplified form of the card game seventeen & four, in which the agent takes the role of the player and plays against a hard-coded dealer.

3.3.1 Single-agent environment base-class

Abstract base classes for environments in the MMLF.

This module defines abstract base classes from which environments in the MMLF must be derived.

The following environment base classes are defined:

SingleAgentEnvironment : Base class for environments in which a single agent can act.

class `environments.single_agent_environment.`**`SingleAgentEnvironment`**(*config*, *baseUserDir*, *useGUI*)

MMLF interface for environments with a simgle agent

Each environment that should be used in the MMLF needs to be derived from this class and implement the following methods:

Interface Methods

getInitialState Returns the initial state of the environment

getStateSpace Returns the state space of the environment

getActionSpace Returns the action space of the environment

evaluateAction(actionObject) Evaluate the action defined in *actionObject*

`actionTaken`(*action*)

Executes an action chosen by the agent.

Causes a state transition of the environment based on the specific action chosen by the agent. Depending on the successor state, the agent is rewarded, informed about the end of an episodes and/or provided with the next state.

Parameters

action : A dictionary that specifies for each dimension of the action space the value the agent has chosen for the dimension.

`evaluateAction`(*actionObject*)

Execute an agent's action in the environment.

Take an actionObject containing the action of an agent, and evaluate this action, calculating the next state, and the reward the agent should receive for having taken this action.

Additionally, decide whether the episode should continue, or end after the reward has been issued to the agent.

This method returns a dictionary with the following keys:

rewardValue : An integer or float representing the agent's reward. If rewardValue == None, then no reward is given to the agent.

startNewEpisode : True if the agent's action has caused an episode to get finished.

nextState : A State object which contains the state the environment takes on after executing the action. This might be the initial state of the next episode if a new episode has just started (startNewEpisode == True)

terminalState : A State object which contains the terminal state of the environment in the last episode if a new episode has just started (startNewEpisode == True). Otherwise None.

`getActionSpace`()

Return the action space of this environment.

More information about action spaces can be found in *State and Action Spaces*

`getInitialState`()

Returns the initial state of the environment

More information about (valid) states can be found in *State and Action Spaces*

`getStateSpace`()

Returns the state space of this environment.

More information about state spaces can be found in *State and Action Spaces*

`getWish`()

Query the next command object for the interaction server.

`giveAgentInfo`(*agentInfo*)

Check whether an agent is compatible with this environment.

The parameter *agentInfo* contains informations whether an agent is suited for continuous state and/or action spaces, episodic domains etc. It is checked whether the agent has the correct capabilties for this environment. If not, an `ImproperAgentException` exception is thrown.

Parameters

agentInfo : A dictionary-like object of type `GiveAgentInfo` that contains information regarding the agent's capabilities.

`plotStateSpaceStructure`(*axis*)

Plot structure of state space into given axis.

Just a helper function for viewers and graphic logging.

`stop`()

Method which is called when the environment should be stopped

class `environments.single_agent_environment.`**`ImproperAgentException`**

Exception thrown if an improper agent is added to a world.

3.3.2 Fully-observable Double Pole Balancing

Module that implements the double pole balancing environment and its dynamics.

class `worlds.double_pole_balancing.environments.double_pole_balancing_environment.`**`DoublePoleE`**

The double pole balancing environment

In the double pole balancing environment, the task of the agent is to control a cart such that two poles which are mounted on the cart stay in a nearly vertical position (to balance them). At the same time, the cart has to stay in a confined region.

The agent can apply in every time step a force between -10N and 10N in order to accelerate the car. Thus the action space is one-dimensional and continuous. The state consists of the cart's current position and velocity as well as the poles' angles and angular velocities. Thus, the state space is six-dimensional and continuous.

The config dict of the environment expects the following parameters:

CONFIG DICT

GRAVITY : The gravity force. Benchmark default "-9.8".

MASSCART : The mass of the cart. Benchmark default "1.0".

TAU : The time step between two commands of the agent. Benchmark default "0.02"

MASSPOLE_1 : The mass of pole 1. Benchmark default "0.1"

MASSPOLE_2 : The mass of pole 2. Benchmark default "0.01"

LENGTH_1 : The length of pole 1. Benchmark default "0.5"

LENGTH_2 : The length of pole 2. Benchmark default "0.05"

MUP : Coefficient of friction of the poles' hinges. Benchmark default "0.000002"

MUC : Coefficient that controls friction. Benchmark default "0.0005"

INITIALPOLEANGULARPOSITION1 : Initial angle of pole 1. Benchmark default "4.0"

MAXCARTPOSITION : The maximal distance the cart is allowed to move away from its start position. Benchmark default "2.4"

MAXPOLEANGULARPOSITION1 : Maximal angle pole 1 is allowed to take on. Benchmark default "36.0"

MAXPOLEANGULARPOSITION2 : Maximal angle pole 2 is allowed to take on. Benchmark default "36.0"

MAXSTEPS : The number of steps the agent must balance the poles. Benchmark default "100000"

3.3.3 Linear Markov Chain

A linear markov chain environment.

class `worlds.linear_markov_chain.environments.linear_markov_chain_environment.`**`LinearMarkovCha`**

A linear markov chain.

The agent starts in the middle of this linear markov chain. He can either move right or left. The chain is not stochastic, i.e. when the agent wants to move right, the state is decreased with probability 1 by 1. When the agent wants to move left, the state is increased with probability 1 by 1 accordingly. New in version 0.9.10: Added LinearMarkovChain environment

CONFIG DICT

length : The number of states of the linear markov chain

3.3.4 Maze 2D

Two-dimensional maze world environment.

class `worlds.maze2d.environments.maze2d_environment.`**`Maze2dEnvironment`**(*useGUI*, **args*, ***kwargs*)

The two-dimensional maze environment for an agent without orientation.

A 2d maze world, in which the agent is situated at each moment in time in a certain field (specified by its (row,column) coordinate) and can move either upwards, downwards, left or right. The structure of the maze can be configured via a text-based config file.

CONFIG DICT

episodesUntilDoorChange : Episodes that the door will remain in their initial state. After this number of episodes, the door state is inverted.

MAZE : Name of the config file, where the maze is defined. These files are located in folder 'worlds/maze2d'

3.3.5 Maze Cliff

This module contains an implementation of the maze 2d dynamics which can be used in a world of the mmlf.framework.

2008-03-08, Jan Hendrik Metzen (jhm@informatik.uni-bremen.de)

class `worlds.maze_cliff.environments.maze_cliff_environment.`**`MazeCliffEnvironment`**(*useGUI*, **args*, ***kwargs*)

The two-dimensional maze cliff environment.

In this maze, there are two alternative ways from the start to the goal state: one short way which leads along a dangerous cliff and one long but secure way. If the agent happens to step into the maze, it will get a huge negative reward (configurable via *cliffPenalty*) and is reset into the start state. Per default, the maze is deterministic, i.e. the agent always moves in the direction it chooses. However, the parameter *stochasticity* allows to control the stochasticity of the environment. For instance, when stochasticity is set to 0.01, the the agent performs a random move instead of the chosen one with probability 0.01.

The maze structure is as follows where "S" is the start state, "G" the goal state and "C" is a cliff field: ********** * * * * * * *SCCCCCCCCCCG* **********

CONFIG DICT

cliffPenalty : The reward an agent obtains when stepping into the cliff area

stochasticity : The stochasticity of the state transition matrix. With probability 1-*stochasticity* the desired transition is made, otherwise a random transition

3.3.6 Mountain Car

Module that implements the mountain car environment and its dynamics.

class `worlds.mountain_car.environments.mcar_env.`**`MountainCarEnvironment`**(*config*, *useGUI*, **args*, ***kwargs*)

The mountain car environment.

In the mountain car environment, the agent has to control a car which is situated somewhere in a valley between two hills. The goal of the agent is to reach the top of the right hill. Unfortunately, the engine of the car is not strong enough to reach the top of the hill directly from many start states. Thus, it has first to drive in the wrong direction to gather enough potential energy.

The agent can either accelerate left, right, or coast. Thus, the action space is discrete with three discrete actions. The agent observes two continuous state components: The current position and velocity of the car. The start state of the car is stochastically initialised.

CONFIG DICT

maxStepsPerEpisode : The number of steps the agent has maximally to reach the goal. Benchmark default is “500”.

accelerationFactor : A factor that influences how strong the cars engine is relative to the slope of the hill. Benchmark default is “0.001”.

maxGoalVelocity : Maximum velocity the agent might have when reaching the goal. If smaller than 0.07, this effectively makes the task MountainPark instead of MountainCar. Benchmark default is “0.07”

positionNoise : Noise that is added to the agent’s observation of the position. Benchmark default is “0.0”

velocityNoise : Noise that is added to the agent’s observation of the velocity. Benchmark default is “0.0”

3.3.7 Partially-observable Double Pole Balancing

Module that implements the partially observable double pole balancing.

class `worlds.po_double_pole_balancing.environments.po_double_pole_balancing_environment.`**`PODou`**

The partially observable double pole balancing environment

In the partially observable double pole balancing environment, the task of the agent is to control a cart such that two poles which are mounted on the cart stay in a nearly vertical position (to balance them). At the same time, the cart has to stay in a confined region. In contrast to the fully observable double pole balancing environment, the agent only observes the current position of cart and the two poles but not their velocities. This renders the problem to be not markovian.

The agent can apply in every time step a force between -10N and 10N in order to accelerate the car. Thus the action space is one-dimensional and continuous. The state consists of the cart’s current position and velocity as well as the poles’ angles and angular velocities. Thus, the state space is six-dimensional and continuous.

The config dict of the environment expects the following parameters:

CONFIG DICT

GRAVITY : The gravity force. Benchmark default "-9.8".

MASSCART : The mass of the cart. Benchmark default "1.0".

TAU : The time step between two commands of the agent. Benchmark default "0.02"

MASSPOLE_1 : The mass of pole 1. Benchmark default "0.1"

MASSPOLE_2 : The mass of pole 2. Benchmark default "0.01"

LENGTH_1 : The length of pole 1. Benchmark default "0.5"

LENGTH_2 : The length of pole 2. Benchmark default "0.05"

MUP : Coefficient of friction of the poles' hinges. Benchmark default "0.000002"

MUC : Coefficient that controls friction. Benchmark default "0.0005"

INITIALPOLEANGULARPOSITION1 : Initial angle of pole 1. Benchmark default "4.0"

MAXCARTPOSITION : The maximal distance the cart is allowed to move away from its start position. Benchmark default "2.4"

MAXPOLEANGULARPOSITION1 : Maximal angle pole 1 is allowed to take on. Benchmark default "36.0"

MAXPOLEANGULARPOSITION2 : Maximal angle pole 2 is allowed to take on. Benchmark default "36.0"

MAXSTEPS : The number of steps the agent must balance the poles. Benchmark default "100000"

3.3.8 Pinball

Module that contains the the pinball maze environment.

class `worlds.pinball_maze.environments.pinball_maze_environment.`**`PinballMazeEnvironment`**(*useGUI*, **args*, ***kwarg*

The pinball maze environment class.

The pinball maze environment class.

See Also:

George Konidaris and Andrew G Barto "Skill Discovery in Continuous Reinforcement Learning Domains using Skill Chaining" in "Advances in Neural Information Processing Systems", 2009

New in version 0.9.9.

CONFIG DICT

DRAG : Factor that slows the ball each time step (multiplied to velocity after each step)

NOISE : gaussian noise with MU_POS for position [x,y] and MU_VEL for velocity [xdot,ydot]; as simplification the covariance matrix is just a unit matrix multiplied with SIGMA

THRUST_PENALTY : Reward the agent gains each time it accelerates the ball

STEP_PENALTY : Reward the agent gains each time step it not thrusts or terminates

END_EPISODE_REWARD : Reward the agent gains if the ball reaches the goal

SUBSTEPS : number of dynamic steps of the environment between each of the agent's actions

MAZE : Name of the config file, where the maze is defined. These files are located in folder 'worlds/pinball_maze'

3.3.9 Seventeen and Four

Module that contains the seventeen & four environment

This module contains a simplified implementation of the card game seventeen & four, in which the agent takes the role of the player and plays against a hard-coded dealer.

class `worlds.seventeen_and_four.environments.seventeen_and_four.`**`SeventeenAndFourEnvironment`**(*u*
*
*

The seventeen & four environment

This environment implements a simplified form of the card game seventeen & four, in which the agent takes the role of the player and plays against a hard-coded dealer.

The player starts initially with two randomly drawn card with values of 2,3,4,7,8,9,10 or 11. The goal is get a set of cards whose sum is as close as possible to 21. The agent can stick with two cards or draw arbitrarily many cards sequentially. If the sum of cards becomes greater than 21, the agent looses and gets a reward of -1. If the agent stops with cards less valued than 22, a hard-coded dealer policy starts playing against the agent. This dealer draws card until it has either equal/more points than the agent or more than 21. In the first case, the dealer wins and the agent gets a reward of -1, otherwise the player wins and gets a reward of 0.

3.3.10 Single Pole Balancing

Module that implements the single pole balancing environment and its dynamics.

class `worlds.single_pole_balancing.environments.single_pole_balancing_environment.`**`SinglePoleB`**

The single pole balancing environment.

In the single pole balancing environment, the task of the agent is to control a cart such that a pole which is mounted on the cart stays in a nearly vertical position (to balance it). At the same time, the cart has to stay in a confined region.

The agent can apply in every time step a force between -2N and 2N in order to accelerate the car. Thus the action space is one-dimensional and continuous. The state consists of the cart's current position and velocity as well as the pole's angle and angular velocity. Thus, the state space is four-dimensional and continuous.

CONFIG DICT

GRAVITY : The gravity force. Benchmark default "-9.8"

MASSCART : The mass of the cart. Benchmark default "1.0"

MASSPOLE : The mass of the pole. Benchmark default "0.1"

TOTAL_MASS : The total mass (pole + cart). Benchmark default "1.1"

LENGTH : The length of the pole. Benchmark default "0.5"

POLEMASS_LENGTH : The center of mass of the pole. Benchmark default "0.05"

TAU : The time step between two commands of the agent. Benchmark default "0.02"

MAXCARTPOSITION : The maximal distance the cart is allowed to move away from its start position. Benchmark default "7.5"

MAXPOLEANGULARPOSITION : Maximal angle the pole is allowed to take on. Benchmark default "0.7"

MAXSTEPS : The number of steps the agent must balance the poles. Benchmark default "100000"

3.4 Observables

Module for that implements various kinds of observables

Currently implemented observables are

- FloatStreamObservable
- TrajectoryObservable
- StateActionValuesObservable
- FunctionOverStateSpaceObservable
- ModelObservable

class `framework.observables.`**`AllObservables`**

The set of observables is itself observable, too

`addObservable`(*observable*)

Add a new observable to the set of all observables.

`getAllObservablesOfType`(*type*)

Return all observables of a given type.

`getObservable`(*observableName*, *type*)

Return all observable of a given type with given observableName.

`removeObservable`(*observable*)

Remove an observable from the set of all observables.

class `framework.observables.`**`FloatStreamObservable`**(*title*, *time_dimension_name*, *value_name*)

An observable class that handles a stream of floats

`addValue`(*time*, *value*)

Add a new value for the given point in time to the observable.

class `framework.observables.`**`FunctionOverStateSpaceObservable`**(*title*, *discreteValues*)

Observable class for observing functions over the state space.

This class implements an observable for observing (one-dimensional) functions defined over the state space. This function can either implement an mapping f: S -> R (real numbers, *discreteValues*=False) or a mapping f: S -> C (some discrete finite set of values, *discreteValues*=True). An example for the first kind of function would be the optimal value function V(s), an example for the second case a deterministic policy pi(s), with C being the action space.

`plot`(*function*, *fig*, *stateSpace*, *actionSpace*, *plotStateDims=None*, *rasterPoints=100*)

Creates a graphical representation of a FunctionOverStateSpace.

Creates a plot of *policy* in the 2D subspace of the state space spanned by *stateIndex1* and *stateIndex2*.

updateFunction (*function*)
: Inform observable that the encapsulated function has changed.

class `framework.observables.`**ModelObservable** (*title*)
: Observable class for observing models.

  This class implements an observable for observing MMLF models. Whenever a new state transition, a new start state or a new terminal state is added to the model, this observable notifies all observers of this change.

  plot (*model*, *fig*, *stateSpace*, *colouring*, *plotSamples*, *minExplorationValue*, *plotStateDims*, *dimValues*)
  : Does the actual plotting (either for viewer or for log-graphic).

  updateModel (*model*)
  : Inform observable that the encapsulated model has changed.

class `framework.observables.`**Observable** (*title*)
: Base class for all MMLf observables.

  addObserver (*observer*)
  : Add an observer that gets informed of all changes of this observable.

  removeObserver (*observer*)
  : Remove an observer of this observable.

class `framework.observables.`**StateActionValuesObservable** (*title*)
: Observable class for real valued functions over the state action space.

  This might be for instance a Q-function or the eligibility traces or a stochastic policy.

  plot (*function*, *actions*, *fig*, *stateSpace*, *plotStateDims=None*, *plotActions=None*, *rasterPoints=100*)
  : Plots the q-Function for the case of a 2-dim subspace of the state space.

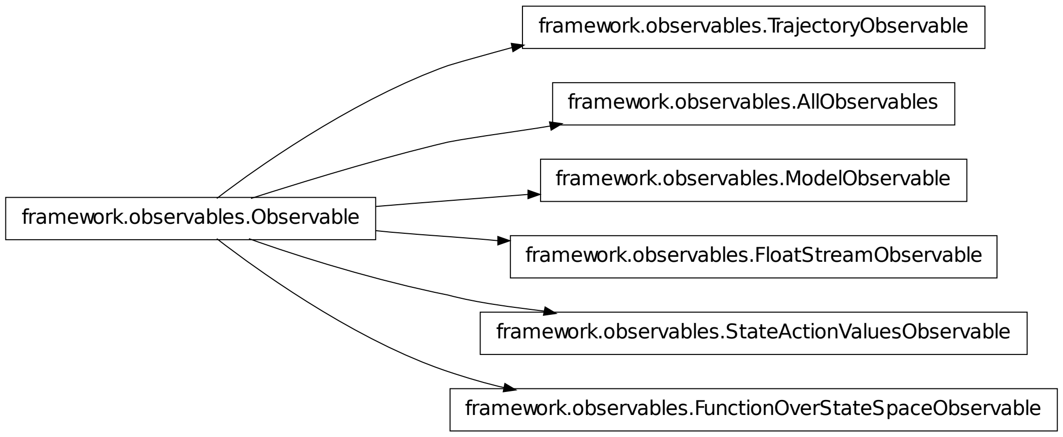
    plotStateDims :The 2 dims that should be plotted plotActions : The action that should be plotted rasterPoints : How many raster points per dimension

  updateValues (*valueAccessFunction*, *actions*)
  : Inform observable that the encapsulated function has changed.

class `framework.observables.`**TrajectoryObservable** (*title*)
: An observable class that can monitor an agent's trajectory

  addTransition (*state*, *action*, *reward*, *succState*, *episodeTerminated=False*)
  : Add a new transition to the observable.

    Adds the transition from *state* to *succState* for chosen action *action* and obtained reward *reward* to the observable. If *episodeTerminated*, succState is a terminal state of the environment.


framework.observables.Observable
framework.observables.TrajectoryObservable
framework.observables.AllObservables
framework.observables.ModelObservable
framework.observables.FloatStreamObservable
framework.observables.StateActionValuesObservable
framework.observables.FunctionOverStateSpaceObservable

3.5 Resources

3.5.1 Function Approximators

MMLF function approximator interface

This module defines the interface for function approximators that can be used with temporal difference learning methods.

The following methods must be implemented by each function approximator:

- *computeQ(state, action)*: Compute the Q value of the given state-action pair
- ***train()*: Train the function approximator on the trainSet consisting of** state-action pairs and the desired Q-value for these pairs.

class `resources.function_approximators.function_approximator.`**`FunctionApproximator`**(*stateSpace*, **args*, ***kwargs*)

The function approximator interface.

Each function approximator must specify two methods: * computeQ * train

`computeOptimalAction`(*state*)
Compute the action with maximal Q-value for the given state

`computeQ`(*state*, *action*)
Computes the Q-value of the given state, action pair

It is assumed that a state is a n-dimensional vector where n is the dimensionality of the state space. Furthermore, the states must have been scaled externally so that the value of each dimension falls into the bin [0,1]. action must be one of the actions given to the constructor

`computeV`(*state*)
Computes the V-value of the given state

static `create` (*faSpec*, *stateSpace*, *actions*)
Factory method that creates function approximator based on spec-dictionary.

static `getFunctionApproximatorDict` ()
Returns dict that contains a mapping from FA name to FA class.

`train` (*trainingSet*)
Trains the function approximator using the given training set.

trainingSet is a dictionary containing training data where the key is the respective (state, action) pair whose Q-value should be updated and the dict-value is this desired Q-value.

Cerebellar Model Articulation Controller

The Cerebellar Model Articulation Controller (CMAC) function approximator.

class `resources.function_approximators.cmac.`**`CMAC`** (*stateSpace*, *actions*, *number_of_tilings*, *learning_rate*, *default*, ***kwargs*)
The Cerebellar Model Articulation Controller (CMAC) function approximator.

CONFIG DICT

number_of_tilings : The number of independent tilings that are used in each tile coding

default : The default value that an entry stored in the function approximator has initially

learning_rate : The learning rate used internally in the updates

K-Nearest Neighbors

Function approximator based on k-Nearest-Neighbor interpolation.

class `resources.function_approximators.knn.`**`KNNFunctionApproximator`** (*stateSpace*, *actions*, *k=10*, *b_X=0.10000000000000001*, ***kwargs*)
Function approximator based on k-Nearest-Neighbor interpolation

A function approximator that stores the a given set of (state, action) -> Q-Value samples. The sample set is split into subsets, one for each action (thus, a discrete, finite set of actions is assumed). When the Q-Value of a state-action is queried, the *k* states most similar to the query state are extracted (under the constraint that the query action is the action applied in these states). The Q-Value of the query state-action is computed as weighted linear combination of the *k* extracted samples, where the weighting is based on the distance between the respective state and the query state. The weight of a sample is computed as exp(-(distance/b_x)**2), where *b_x* is an parameter that influences the generalization breadth. Smaller values of *b_X* correspond to increased weight of more similar states.

CONFIG DICT

k : The number of neighbors considered in k-Nearest Neighbors

b_X : The width of the gaussian weighting function. Smaller values of b_X correspond to increased weight of more similar states.

Linear Combination

The linear combination function approximator

This module defines the linear combination function approximator. It computes the Q-value as the dot product of the feature vector and a weight vector. It's main application area are discrete worlds; however given appropriate features it can also be used in continuous world.

class `resources.function_approximators.linear_combination.`**`LinearCombination`**(*stateSpace*, *actions*, *learning_rate=1.0*, ***kwargs*)

The linear combination function approximator.

This class implements the function approximator interface. It computes the Q-value as the dot product of the feature vector and a weight vector. It's main application area are discrete worlds; however given appropriate features it can also be used in continuous world. At the moment, it ignores the planned action since it is assummed that it is used in combination with minmax tree search.

CONFIG DICT

learning_rate : The learning rate used internally in the updates.

Multi-layer Perceptron (MLP)

This module defines a multi layer perceptron (MLP) function approximator.

class `resources.function_approximators.mlp.`**`MLP`**(*stateSpace*, *actions*, ***kwargs*)

Multi-Layer Perceptron function approximator.

Multilinear Grid

The multilinear grid function approximator.

In this function approximator, the state space is spanned by a regular grid. For each action a separate grid is spanned. The value of a certain state is determined by computing the grid cell it lies in and multilinearly interpolating from the cell corners to the particular state.

class `resources.function_approximators.multilinear_grid.`**`MultilinearGrid`**(*stateSpace*, *actions*, *learning_rate*, *default*, ***kwargs*)

The multilinear grid function approximator.

In this function approximator, the state space is spanned by a regular grid. For each action a separate grid is spanned. The value of a certain state is determined by computing the grid cell it lies in and multilinearly interpolating from the cell corners to the particular state.

CONFIG DICT

default : The default value that an entry stored in the function approximator has initially.

learning_rate : The learning rate used internally in the updates.

QCON

This module defines the QCON function approximator

class `resources.function_approximators.qcon.`**`QCON`**(*stateSpace*, *actions*, *hidden*, *learning_rate*, ***kwargs*)

Function approximator based on the connectionist QCON architecture

This class implements the QCON architecture which consists of a connectionist Q-learning model where each action has a separate network. The feed-forward neural network are implemented using the python package ffnet

CONFIG DICT

hidden : The number of neurons in the hidden layer of the multi-layer perceptron

learning_rate : The learning rate used internally in the updates.

Radial Base Function

This module defines the Radial Base Function function approximator.

class `resources.function_approximators.rbf.`**`RBF_FA`**(*stateSpace*, *actions*, *learning_rate*, ***kwargs*)

The Radial Base Function function approximator

This class implements the function approximator interface by using the RBF function approximator. A RBF function approximator is composed of several radial base functions, one for each action.

CONFIG DICT

learning_rate : The learning rate used internally in the updates.

Tabular Storage

This module defines the tabular storage function approximator

The tabular storage function approximator can be used for discrete worlds. Actually it is not really a function approximator but stores the value function exactly.

class `resources.function_approximators.tabular_storage.`**`TabularStorage`**(*actions=None*, *default=0*, *learning_rate=1.0*, *stateSpace=None*, ***kwargs*)

Function approximator for small, discrete environments.

This class implements the function approximator interface. It does not really approximate but simply stores the values in a table. Thus, it should not be applied in environments with continuous states.

CONFIG DICT

default : The default value that an entry stored in the function approximator has initially.

learning_rate : The learning rate used internally in the updates.

3.5.2 Learning Algorithms

Eligibility Traces

Module for Eligibility Traces.

This module contains code for eligibility traces, which can be used in combination with temporal difference learning.

Temporal Difference

Module for temporal difference learning methods SARSA and Watkins Q.

This module contains the main algorithmic code for the temporal difference learning methods SARSA and Watkins Q.

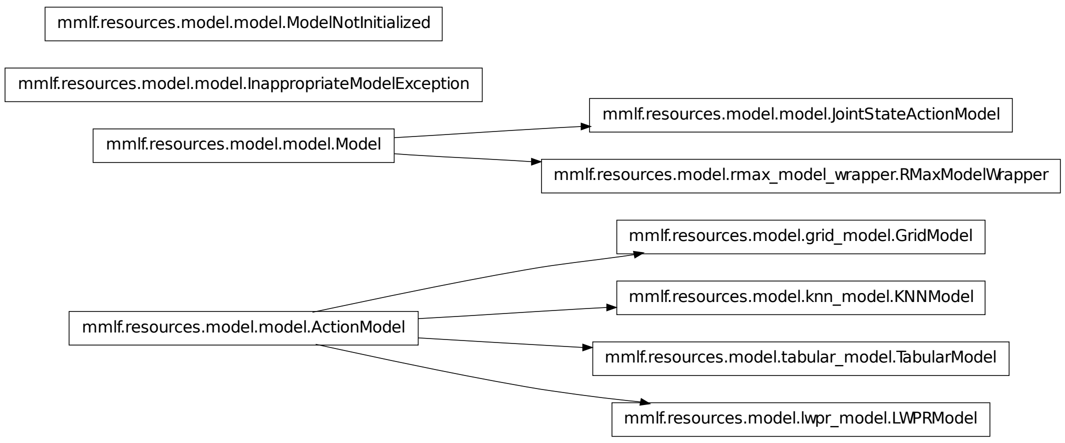
3.5.3 Models

Interfaces for MMLF models

This module contains the Model class that specifies the interface for models in the MMLF. The following methods must be implemented by each model class:

- addExperience
- sampleStateAction
- sampleSuccessorState
- getSuccessorDistribution
- getExpectedReward
- getExplorationValue

The standard way of implementing a model is to learn a model for each action separately. In order to simplify this task, the Model interface contains a standard implementation in the case that an ActionModelClass parameter is passed to constructor. This parameters must be a class that implements the interface ActionModel that is also contained in this module. If the ActionModelClass parameter is passed to the Model constructor, for each action one instance of it is constructed an all methods are per default forwarded to the respective method of the ActionModel.


mmlf.resources.model.model.ModelNotInitialized
mmlf.resources.model.model.InappropriateModelException
mmlf.resources.model.model.Model
mmlf.resources.model.model.JointStateActionModel
mmlf.resources.model.rmax_model_wrapper.RMaxModelWrapper
mmlf.resources.model.grid_model.GridModel
mmlf.resources.model.knn_model.KNNModel
mmlf.resources.model.model.ActionModel
mmlf.resources.model.tabular_model.TabularModel
mmlf.resources.model.lwpr_model.LWPRModel

class `resources.model.model.`**`Model`**(*agent*, *userDirObj*, *stateSpace*, *actions*, *ActionModelClass=None*, *cyclicDimensions=False*, ***kwargs*)

Interface for MMLF models

This class specifies the interface for models in the MMLF. The following methods must be implemented by each model:

- •addExperience
- •getSample
- •sampleSuccessorState
- •getSuccessorDistribution (optional)
- •getExpectedReward
- •getExplorationValue (optional)

Additional, this class already implements the following methods:

- addStartState
- addTerminalState
- drawStartState
- getNearestNeighbor
- getNearestNeighbors
- isTerminalState
- plot

The class contains a standard implementation in the case that an ActionModelClass parameter is passed to constructor. This parameters must be a class that implements the interface ActionModel that is also contained in this module. If the ActionModelClass parameter is passed to the Model constructor, for each action one instance of it is constructed an all methods are per default forwarded to the respective method of the ActionModel.

The constructor expects the following parameters:

- *agent* : The agent that is using this model
- *userDirObj* : The userDirObj object of the agent
- ***actions*** [A sequence of all allowed actions] (note: discrete action space assumed)
- ***ActionModelClass*** [A class implementing the ActionModel interface] If specified, for each available action a separate instance is created and all method calls are forwarded to the respective instance
- ***cyclicDimensions*** **If some (or all) of the dimensions of the state space** are cyclic (i.e. the value 1.0 and 0.0 are equivalent) this parameter should be set to True. Defaults to False.

`addExperience`(*state*, *action*, *succState*, *reward*)

Updates the model based on the given experience tuple

Update the model based on the given experience tuple consisting of a *state*, an *action* taken in this state, the resulting successor state *succState* and the obtained *reward*.

`addStartState`(*state*)

Add the given state to the set of start states

`addTerminalState`(*state*)

Add the given state to the set of terminal states

static create (*modelSpec*, *agent*, *stateSpace*, *actionSpace*, *userDirObj*)
Factory method that creates model-learners based on spec-dictionary.

drawStartState ()
Returns a random start state

getConfidence (*state*, *action*)
Return how confident the model is in its prediction for the given state action pair

getExpectedReward (*state*, *action*)
Returns expected reward when action is performed in state

getExplorationValue (*state*, *action*)
Returns how often the pair state-action has been explored

static getModelDict ()
Returns dict that contains a mapping from model name to model class.

getNearestNeighbor (*state*)
Returns the most similar known state to the given *state*

getNearestNeighbors (*state*, *k*, *b*)
Determines *k* most similar states to the given *state*

Determines *k* most similar states to the given *state*. Returns an iterator over (weight, neighbor), where weight is the guassian weigthed influence of the neighbor onto *state*. The weight is computed via exp(-dist/b**2)/sum_over_neighbors(exp(-dist_1/b**2)). Note that the weights sum to 1.

getPredecessorDistribution (*state*, *action*)
Returns an iterator that yields predecessor state probabilities

Return an iterator that yields the pairs of state along with their probabilities of being the predecessor state of *state* when *action* is performed by the agent. Note: This assumes a discrete (or discretized) state space since

otherwise this there will be infinitely many states with probability > 0.

getSample ()
Return a sample drawn randomly

Return a random sample (i.e. a state, action, reward, successor state 4-tuple).

getStates ()
Return all states that are contained in the example set or are terminal

getSuccessorDistribution (*state*, *action*)
Returns an iterator that yields successor state probabilities

Return an iterator that yields the pairs of state along with their probabilities of being the successor state of *state* when *action* is performed by the agent. Note: This assummes a discrete (or discretized) state space since

otherwise this there will be infinitely many states with probability > 0.

isTerminalState (*state*)
Returns an estimate of whether the given state is a terminal one

samplePredecessorState (*state*, *action*)
Return sample predecessor state of state-action.

Return a state drawn randomly from the predecessor distribution of *state-action*

sampleSuccessorState (*state*, *action*)
Return sample successor state of state-action.

Return a state drawn randomly from the successor distribution of *state-action*

class `resources.model.model.`**`JointStateActionModel`**(*agent*, *actionRange*, *actionSpace*, *ActionModelClass*, ***kwargs*)

Interface for models for continuous action spaces.

The *JointStateActionModel* subclasses *Model* and changes its default behaviour: Instead of forwarding every action to a separate ActionModel, state and action are concatenated and one ActionModel is used to learn the behaviour within this "State-Action-Space", i.e. a mapping from (state, action_1) -> (succState, action_2). Since action_2 depends on the policy, it cannot be learned by a model and is thus ignored. For training of the ActionModel, action_2 is set to action_1.

NOTE: Currently, only one action dimension is supported!

The constructor expects two parameter: * *actionRange* : A tuple indicating the minimal and maximal value of the

action space dimension.

- ***ActionModelClass*** [A class implementing the ActionModel interface. This] class is used to learn the State-Action Space dynamics.

`addExperience`(*state*, *action*, *succState*, *reward*)

Updates the model based on the given experience tuple

Update the model based on the given experience tuple consisting of a *state*, an *action* taken in this state, the resulting successor state *succState* and the obtained *reward*.

`getConfidence`(*state*, *action*)

Return how confident the model is in its prediction for the given state action pair

`getExpectedReward`(*state*, *action*)

Returns expected reward when action is performed in state

`getExplorationValue`(*state*, *action*)

Returns how often the pair state-action has been explored

`getPredecessorDistribution`(*state*, *action*)

Returns an iterator that yields predecessor state probabilities

Return an iterator that yields the pairs of state along with their probabilities of being the predecessor state of *state* when *action* is performed by the agent. Note: This assumes a discrete (or discretized) state space since

otherwise this there will be infinitely many states with probability > 0.

`getSample`()

Return a sample drawn randomly

Return a random sample (i.e. a state, action, reward, successor state 4-tuple).

`getSuccessorDistribution`(*state*, *action*)

Returns an iterator that yields successor state probabilities

Return an iterator that yields the pairs of state along with their probabilities of being the successor state of *state* when *action* is performed by the agent. Note: This assumes a discrete (or discretized) state space since

otherwise this there will be infinitely many states with probability > 0.

`samplePredecessorState`(*state*, *action*)

Return sample predecessor state of state-action.

Return a state drawn randomly from the predecessor distribution of *state-action*

sampleSuccessorState (*state*, *action*)
Return sample successor state of state-action.

Return a state drawn randomly from the successor distribution of *state-action*

class resources.model.model.**ActionModel** (*stateSpace*, **args*, ***kwargs*)
Interface for MMLF action models

addExperience (*state*, *succState*, *reward*)
Updates the action model based on the given experience tupel

getConfidence (*state*)
Return how confident the model is in its prediction for the given state

getExpectedReward (*state*)
Return the expected reward for the given state under this action

getExplorationValue (*state*)
Returns the exploration value for the given state

getPredecessorDistribution (*state*)
Iterates over pairs of predecessor states and their probabilities

getSuccessorDistribution (*state*)
Iterates over pairs of successor states and their probabilities

samplePredecessorState (*state*)
Sample a predecessor state for this state for this action-model

sampleState ()
Sample a state randomly from this action model

sampleSuccessorState (*state*)
Sample a successor state for this state for this action-model

Grid-based

Grid-based model based on a grid that spans the state space.

class resources.model.grid_model.**GridModel** (*stateSpace*, *nodesPerDim*, *activationRadius*, *b=None*, ***kwargs*)
Grid-based model based on a grid that spans the state space.

The state transition probabilities and reward probabilities are estimated only for the nodes of this grid. Experience samples are used to update the estimates of nearby grid nodes. Using a discrete grid has the advantage that there is only a finite amount of states for which probabilities needs to be estimated.

CONFIG DICT

nodesPerDim : The number of nodes of the grid in each dimensions. The total number of grid nodes is thus nodesPerDim**dims.

activationRadius : The radius of the region around the query state in which grid nodes are activated

b : Parameter controlling how fast activation decreases with distance from query state. If None, set to “activationRadius / sqrt(-log (0.01))”

K-Nearest Neighbors

An action model class based on KNN state transition modeling.

This model is based on the model proposed in: Nicholas K. Jong and Peter Stone, "Model-based function approximation in reinforcement learning", in Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems Honolulu, Hawaii: ACM, 2007, 1-8, http://portal.acm.org/citation.cfm?id=1329125.1329242.

class `resources.model.knn_model.`**`KNNModel`**(*stateSpace*, *k=10*, *b_Sa=0.10000000000000001*, ***kwargs*)

An action model class based on KNN state transition modeling.

This model learns the state successor (and predecessor) function using the "k-Nearest Neighbors" (KNN) regression learner. This learner learns a stochastic model, mapping each state s to the successor $s' = s + (s^'_{neighbor} - s_{neighbor})$ with probability exp(-(||s - s_{neighbor}||/b_Sa)^2)/ sum_{neighbor in knn(s)} exp(-(||s - s_{neighbor}||/b_Sa)^2). The reward function is learned using an KNN model, too.

CONFIG DICT

exampleSetSize : The maximum number of example transitions that is remembered in the example set. If the example set is full, old examples must be deleted or now new examples are accepted.

k : The number of neighbors considered in k-Nearest Neighbors

b_Sa : The width of the gaussian weighting function. Smaller values of *b_Sa* correspond to increased weight of more similar states

Locally Weighted Projection Regression (LWPR)

An action model class based on LWPR state transition modeling.

class `resources.model.lwpr_model.`**`LWPRModel`**(*stateSpace*, *exampleSetSize=1000*, *examplesPerModelUpdate=10000*, *init_d=25*, ***kwargs*)

An action model class based on LWPR state transition modeling.

This model learns the state successor (and predecessor) function using the "Locally Weighted Projection Regression" (LWPR) regression learner. This learner learns a deterministic model, i.e. each state is mapped onto the successor state the learner considers to be most likely (and thus not onto a probability distribution).

The reward function is learned using a nearest neighbor (NN) model. The reason that NN is used and not LWPR is that the reward function is usually non-smooth and every more sophisticated learning scheme might introduce additional unjustified bias.

This model stores a fixed number of example transitions in a so called example set and relearns the model whenever necessary (i.e. when predictions are requested and new examples have been addedsince the last learning). The model is relearned from scratch.

CONFIG DICT

exampleSetSize : The maximum number of example transitions that is remembered in the example set. If the example set is full, old examples must be deleted or now new examples are accepted.

examplesPerModelUpdate : The number of examples presented to LWPR before the learning is stopped

init_d : The init_d parameter for LWPR that controls the smoothness of the learned function. Smaller values correspond to more smooth functions.

R-Max Model Wrapper

A wrapper for models that changes them to have RMax like behavior

A wrapper that wraps a given *model* and changes its behavior to be RMax-like, i.e. return *RMax* instead of the reward predicted by *model* if the exploration value is below *minExplorationValue*. The implementation is based on the adapter pattern.

class `resources.model.rmax_model_wrapper.`**`RMaxModelWrapper`**(*model*, *RMax*, *minExplorationValue*, ***kwargs*)

A wrapper for models that changes them to have RMax like behavior

A wrapper that wraps a given *model* and changes its behavior to be RMax-like, i.e. return *RMax* instead of the reward predicted by *model* if the exploration value is below *minExplorationValue*. New in version 0.9.9.

CONFIG DICT

minExplorationValue : The agent explores in a state until the given exploration value (approx. number of exploratory actions in proximity of state action pair) is reached for all actions

RMax : An upper bound on the achievable return an agent can obtain in a single episode

model The actual model (this is only a wrapper arount the true model that implements optimism in the face of uncertainty)

Tabular Model

An action model class that is suited for discrete environments

This module contains a model that learns a distribution model for discrete environments.

class `resources.model.tabular_model.`**`TabularModel`**(*stateSpace*, ***kwargs*)

An action model class that is suited for discrete environments

This module contains a model that learns a distribution model for discrete environments. New in version 0.9.9.

CONFIG DICT

exampleSetSize : The maximum number of example transitions that is remembered in the example set. If the example set is full, old examples must be deleted or now new examples are accepted.

3.5.4 Optimization

Interface for MMLF black box optimization algorithms

MMLF optimizers must implement the following methods:

- getActiveIndividual
- getAllIndividuals
- tellFitness
- tellAllFitness
- getBestIndividuals

class `resources.optimization.optimizer.`**`Optimizer`**(*populationSize*, *evalsPerIndividual*)

Interface for MMLF black box optimization algorithms

MMLF optimizers must implement the following methods:

- **getActiveIndividual(): Returns the parameter vector of the currently** active individual
- getAllIndividuals(): Returns all individuals of the current population
- **tellFitness(fitnessSample): Gives one fitness sample for the currently** active individual
- **tellAllFitness(fitness, individuals): Gives fitness samples for all given** individuals
- **getBestIndividual(): Returns the parameter vector of the best individual** found so far

static create (*optimizerSpec*, *sampleInstance*)
Factory method that creates optimizer based on spec-dictionary.

getActiveIndividual ()
Returns the parameter vector of the currently active individual

getAllIndividuals ()
Returns all individuals of the current population

getBestIndividual ()
Returns the parameter vector of the best individual found so far

static getOptimizerDict ()
Returns dict that contains a mapping from optimizer name to optimizer class.

tellAllFitness (*individuals*, *fitness*)
Gives fitness samples for all given individuals

Gives fitness samples *fitness* for all given *individuals*. It is assumed that individuals is actually the whole population. A call of this method may trigger the creation of the next generation.

tellFitness (*fitnessSample*, *individual=None*)
Gives one fitness sample for the individual

Provides the fitness *fitnessSample* obtained in one evaluation of the given individual. In individual==None, then the fitnessSample is attributed to the currently active individual.

CMA-ES

The CMA-ES black-box optimization algorithm

Module that contains the covariance matrix adaptation - evolution strategy (CMA-ES) black-box optimization algorithm.

See also: Nikolaus Hansen and Andreas Ostermeier, "Completely derandomized self-adaptation in evolution strategies", Evolutionary Computation 9 (2001): 159–195.

class resources.optimization.cmaes_optimizer.**CMAESOptimizer** (*sampleInstance=None*, *evalsPerIndividual=1*, *initialMean=None*, *sigma=1.0*, ***kwargs*)

The CMA-ES black-box optimization algorithm

Optimizer that uses CMA-ES for black-box optimization

Optional parameters:

- **sampleInstance: A method which returns a valid parameter vector. This** method can always return the same instance. However, it is more common to provide a method that returns stochastically sampled different individuals. The returned individuals might for instance be used to generate an initial population.

CONFIG DICT

evalsPerIndividual : The number of fitness values that the optimizer expects for each individual.

numParameters : The number of parameters (dimensionality of parameter vector). If None, an initial parameter vector must be specified. Otherwise the initial parameters are drawn randomly.

initialMean : The initial parameter vector.

sigma0 : The initial standard deviation of the search distribution.

Evolution Strategy

A black-box optimization algorithm based on evolution strategies.

class `resources.optimization.evolution_strategy.`**`ESOptimizer`**(*sampleInstance*, *sigma=1.0*, *populationSize=5*, *evalsPerIndividual=1*, *numChildren=10*, ***kwargs*)

A black-box optimization algorithm based on evolution strategies

This black-box optimization algorithm is based on a simple evolution strategies (ES). The mu+lambda ES maintains a set of mu potential parents and generates for each generation a set of lambda children. These children are obtained by mutating randomly sampled parents. This mutation is accomplished by adding normally distributed 0-mean random values to the individuals.

After the children have been evaluated, parents and children are merged in a set and the mu fittest individuals in this set form the new parent set. The mutation standard deviation is adjusted based on Rechenbergs 1/5 rule.

Mandatory parameters:

- **sampleInstance: A method which returns a valid parameter vector. This** method can always return the same instance. However, it is more common to provide a method that returns stochastically sampled different individuals. The returned individuals might for instance be used to generate an initial population.

CONFIG DICT

sigma : The initial standard deviation of the the mutation operator. sigma can be either a single value (meaning that each dimension of the parameter vector will be equally strong mutated) or a vector with the same shape as the parameter vector.

populationSize : The number of parents that survive at the end of a generation (i.e. mu)

evalsPerIndividual : The number of fitness values that the optimizer expects for each individual

numChildren : The number of children evaluated within a generation (i.e. lambda).

Random Search

A black-box optimization algorithm based on random search.

class `resources.optimization.random_search.`**`RandomSearchOptimizer`**(*sampleInstance*, *populationSize=10*, *evalsPerIndividual=1*, ***kwargs*)

The random search optimization algorithm

Optimizer that uses a random search heuristic for black-box optimization. This optimization algorithm does not search the search space "intelligent" but just randomly samples candidate solutions uniformly from the search space and test these. It is mainly useful as a benchmark for more sophisticated search strategies.

Mandatory parameters:

- **sampleInstance: A method which returns a valid parameter vector. This** method can always return the same instance. However, it is more common to provide a method that returns stochastically sampled different individuals. The returned individuals might for instance be used to generate an initial population.

CONFIG DICT

populationSize : The number of individuals evaluated in parallel. Since the search distribution does not change over time, this can be set to any arbitrary value between 1 and maxEvals for this optimizer.

evalsPerIndividual : The number of fitness values that the optimizer expects for each individual.

3.5.5 Planner

Abstract base class for planners

This module contains the abstract base class "Planner" for planning algorithms, i.e. algorithms for computing the optimal state-action value function (and thus the optimal policy) for a given model (i.e. state transition and expected reward function). Subclasses of Planner must implement the "plan" method.

class `resources.planner.planner.`**`Planner`**(*stateSpace*, *functionApproximator*, *gamma*, *actions*, ***kwargs*)

Abstract base class for planners

This module contains the abstract base class "Planner" for planning algorithms, i.e. algorithms for computing the optimal state-action value function (and thus the optimal policy) for a given model (i.e. state transition and expected reward function). Subclasses of Planner must implement the "plan" method. New in version 0.9.9.

static **`create`**(*plannerSpec*, *stateSpace*, *functionApproximator*, *gamma*, *actions*, *epsilon=0.0*)

Factory method that creates planner based on spec-dictionary.

static **`getPlannerDict`**()

Returns dict that contains a mapping from planner name to planner class.

`setStates`(*states*)

Sets the discrete *states* on which dynamic programming is performed.

Remove Q-Values of state-action pairs that are no longer required. NOTE: Setting states is only meaningful for discrete state sets,

where the TabularStorage function approximator is used.

MBDPS Planner

A planner module which is used in the MBDPS agent.

class `resources.planner.mbdps_planner.`**`MBDPSPlanner`**(*gamma*, *planningEpisodes*, *evalsPerIndividual*, *fitnessFunction=<function estimatePolicyOutcome at 0x5488aa0>*, ***kwargs*)

A planner module which is used in the MBDPS agent.

The planner uses a policy search method to search in the space of policies. Policies are evaluated based on the return the achieve in trajectory sampled from a supplied model. A policy's fitness is set to the average return it obtains in several episodes starting from potentially different start states.

Parameters:

- ***fitnessFunction*** **The fitness function used to evaluate the policy.** Defaults to the module's function *estimatePolicyOutcome*.

New in version 0.9.9.

CONFIG DICT

gamma : The discounting factor.

planningEpisodes : The number of simulated episodes that can be conducted before the planning is stopped.

evalsPerIndividual : The number episodes each policy is evaluated.

Prioritized sweeping

Planning based on prioritized sweeping.

class `resources.planner.prioritized_sweeping.`**`PrioritizedSweepingPlanner`**(*stateSpace*, *functionApproximator*, *gamma*, *actions*, *epsilon*, *minSweepDelta*, *updatesPerStep*, ***kwargs*)

Planning based on prioritized sweeping

A planner based on the prioritized sweeping algorithm that allows to compute the optimal state-action value function (and thus the optimal policy) for a given distribution model (i.e. state transition and expected reward function). It is assumed that the MDP is finite and that the available actions are defined explicitly.

The following parameters must be passed to the constructor:

- *stateSpace* The state space of the agent (must be finite)
- ***functionApproximator*** **The function approximator which handles storing** the Q-Function
- *gamma* The discount factor of the MDP
- *actions* The actions available to the agent

New in version 0.9.9.

CONFIG DICT

minSweepDelta : The minimal TD error that is applied during prioritized sweeping. If no change larger than minSweepDelta remains, the sweep is stopped.

updatesPerStep : The maximal number of updates that can be performed in one sweep.

Trajectory sampling

Planning based on trajectory sampling.

This module contains a planner based on the trajectory sampling algorithm that allows to compute an improved state-action value function (and thus policy) for a given sample model. The MDP's state space need not be discrete and finite but it is assumed that there is only a finite number of actions that are defined explicitly.

class `resources.planner.trajectory_sampling.`**`TrajectorySamplingPlanner`**(*stateSpace*, *functionApproximator*, *gamma*, *actions*, *epsilon*, *maxTrajectoryLength*, *updatesPerStep*, *onPolicy*, ***kwargs*)

Planning based on trajectory sampling.

A planner based on the trajectory sampling algorithm that allows to compute an improved state-action value function (and thus policy) for a given sample model. The MDP's state space need not be discrete and finite but it is assumed that there is only a finite number of actions that are defined explicitly.

The following parameters must be passed to the constructor:

- *stateSpace* The state space of the agent
- ***functionApproximator* The function approximator which handles storing** the Q-Function
- *gamma* The discount factor of the MDP
- *actions* The actions available to the agent

New in version 0.9.9.

CONFIG DICT

maxTrajectoryLength : The maximal length of a trajectory before a new trajectory is started.

updatesPerStep : The maximal number of updates that can be performed in one planning call.

onPolicy : Whether the trajectory is sampled from the on-policy distribution.

Value iteration

Planning based on value iteration

This module contains a planner based on the value iteration algorithm that allows to compute the optimal state-action value function (and thus the optimal policy) for a given distribution model (i.e. state transition and expected reward function). It is assumed that the MDP is finite and that the available actions are defined explicitly.

class `resources.planner.value_iteration.`**`ValueIterationPlanner`**(*stateSpace*, *functionApproximator*, *gamma*, *actions*, *epsilon*, *minimalBellmanError*, *maxIterations*, ***kwargs*)

Planning based on value iteration

This module contains a planner based on the value iteration algorithm that allows to compute the optimal state-action value function (and thus the optimal policy) for a given distribution model (i.e. state transition and expected reward function). It is assumed that the MDP is finite and that the available actions are defined explicitly.

The following parameters must be passed to the constructor:

- *stateSpace* The state space of the agent (must be finite)
- ***functionApproximator* The function approximator which handles storing** the Q-Function
- *gamma* The discount factor of the MDP
- *actions* The actions available to the agent

New in version 0.9.9.

CONFIG DICT

minimalBellmanError : The minimal bellman error (sum of TD errors over all state-action pairs) that enforces another iteration. If the bellman error falls below this threshold, value iteration is stopped.

maxIterations : The maximum number of iterations before value iteration is stopped.

3.5.6 Policies

Interface for MMLF policies

A policy is a mapping from state to action. Different learning algorithms of the MMLF have different representation of policies, for example neural networks which represents the policy directly or polcies based on value functions.

This module encapsulates these details so that a stored policy can be loaded directly.

MMLF policies must implement the following methods:

- evaluate
- getParameters
- setParameters

class `resources.policies.policy.`**`Policy`**(**args*, ***kwargs*)

Interface for MMLF policies

MMLF policies must implement the following methods:

- evaluate(state): Evaluates the deterministic policy for the given state
- getParameters(): Returns the parameters of this policy
- setParameters(parameters): Sets the parameters of the policy to the given parameters

static **`create`**(*policySpec*, *numStateDims*, *actionSpace*)
Factory method that creates policy based on spec-dictionary.

`evaluate`(*state*)
Evaluates the deterministic policy for the given state

`getParameters`()
Returns the parameters of this policy

static **`getPolicyDict`**()
Returns dict that contains a mapping from policy name to policy class.

`setParameters`(*parameters*)
Sets the parameters of the policy to the given parameters

Linear Policy

Linear Policies for discrete and continuous action spaces

This module contains classes that represent linear policies for discrete and continuous action spaces.

class `resources.policies.linear_policy.`**`LinearDiscreteActionPolicy`**(*inputDims*, *actionSpace*, *bias=True*, *numOfDuplications=1*, ***kwargs*)

Linear policy for discrete action spaces

Class for linear policies on discrete action space using a 1-of-n encoding, i.e. pi(s) = argmax_{a_j} sum_{i=0}^n w_ij s_i

For each discrete action, *numOfDuplications* outputs in the 1-of-n encoding are created, i.e. n=numOfActions*numOfDuplications. This allows to represent more complex policies.

Expected parameters:

- *inputDims*: The number of input (state) dimensions
- *actionSpace*: The action space which determines which actions are allowed.

CONFIG DICT

bias : Determines whether or not an additional bias (state dimension always equals to 1) is added

numOfDuplications : Determines how many outputs there are for each discrete action in the 1-of-n encoding.

class `resources.policies.linear_policy.`**`LinearContinuousActionPolicy`**(*inputDims*, *actionSpace*, *bias=True*, ***kwargs*)

Linear policy for continuous action spaces

Class for linear policies on continuous action space , i.e. pi(s) = [sum_{i=0}^n w_i0 s_i, dots, sum_{i=0}^n w_ij s_i]

Expected parameters:

- *inputDims*: The number of input (state) dimensions

- ***actionSpace*: The action space which determines which actions are allowed.** It is currently only possible to use this policy with one-dimensional action space with contiguous value ranges

CONFIG DICT

bias : Determines whether or not an additional bias (state dimension always equals to 1) is added

Multi-layer Perceptron Policy

Policies for discrete and continuous action spaces based on an MLP

This module contains classes that represent policies for discrete and continuous action spaces that are based on an multi-layer perceptron representation.

class `resources.policies.mlp_policy.`**`MLPPolicy`**(*inputDims*, *actionSpace*, *hiddenUnits=5*, *bias=True*, *independentOutputs=False*, ***kwargs*)

Policy based on a MLP representation for disc. and cont. action spaces

The MLP are based on the ffnet module. It can be specified how many *hiddenUnits* should be contained in the MLP and whether the neurons should get an addition *bias* input. If *independentOutputs* is set to True, for each output, the hidden layer is cloned, i.e. different outputs do not share common neurons in the network. The *actionSpace* defines which actions the agent can choose.

If the action space has no continuous actions, the finite set of (n) possible action selections is determined. Action selection is based on a 1-of-n encoding, meaning that for each available action the MLP has one output. The action whose corresponding network output has maximal activation is chosen.

If the action space has continuous actions, it is assumed that the action space is one-dimensional. The MLP hase one output falling in the range [0,1]. This output is scaled to the allowed action range to yield the action. Currently, it is assumed that continuous action spaces are one-dimensional and contiguous.

CONFIG DICT

bias : Determines whether or not an additional bias (state dimension always equals to 1) is added

hiddenUnits : Determines hte number of neurons in the hidden layer of the multi-layer perceptron

independentOutputs : If True, for each output the hidden layer is cloned, i.e. different outputs do not share common neurons in the network

Value Function Policy

Policies that are represented using value function

This module contains a class that wraps function approximators such that they can be used directly as policies (i.e. implement the policy interface)

class `resources.policies.value_function_policy.`**`ValueFunctionPolicy`**(*valueFunction*, *actions*)

Class for policies that are represented using value function

This class wraps a function approximator such that it can be used directly as policy (i.e. implement the policy interface)

3.5.7 Policy Search Methods

Interface for MMLF policy search methods

MMLF policy search methods must implement the following methods:

- getActivePolicy
- getBestPolicy
- tellFitness

class `resources.policy_search.policy_search.`**`PolicySearch`**

Interface for MMLF policy search methods

MMLF optimizers must implement the following methods:

- getActivePolicy(): Returns the currently active policy
- getBestPolicy(): Returns the best policy found so far
- **tellFitness(fitnessSample): Gives one fitness sample for the currently** active policy

static **`create`**(*policySearchSpec*, *stateSpace*, *actionSpace*, *stateDims*)

Factory method that creates policy search method based on spec.

`getActivePolicy`()

Returns the currently active policy

`getBestPolicy`()

Returns the best policy found so far

static **`getPolicySearchDict`**()

Returns dict that contains a mapping from PS name to PS class.

`reset`()

Resets the policy search method

`tellFitness`(*fitnessSample*)

Gives one fitness sample for the currently active policy

Provides the fitness *fitnessSample* obtained in one evaluation of the currently active policy.

Fixed Parametrization

Policy search method that optimizes the parameters of a fixed policy class

The module contains a policy search method that optimizes the parameters of an externally predefined, parametrized policy class. It can use an arbitrary black-box optimization algorithm for this purpose.

class `resources.policy_search.fixed_parametrization.`**`FixedParametrization`**(*policy*, *optimizer*)

Policy search method that optimizes the parameters of a policy

The module contains a policy search method that optimizes the parameters of an externally predefined policy.

CONFIG DICT

policy : A parametrized class of policies

optimizer : An arbitrary black-box optimizer

3.5.8 Skill Discovery

MMLF skill discovery interface

This module defines the interface for skill discovery methods

The following methods must be implemented by each skill discovery method:

- ***tellStateTransition(self, state, action, reward, succState)*:** Inform skill discovery of a new transition and search for new skills.
- ***episodeFinished()*:** Inform skill discovery that an episode has terminated.

class `resources.skill_discovery.skill_discovery.`**`SkillDiscovery`**(*optionCreationFct*, *stateSpace*, **args*, ***kwargs*)

The skill discovery interface. New in version 0.9.9.

static **`create`**(*skillDiscoverySpec*, *optionCreationFct*, *stateSpace*)

Factory method that creates skill discovery based on spec-dictionary.

`episodeFinished`(*terminalState*)

Inform skill discovery method that the current episode terminated.

static **`getSkillDiscoveryDict`**()

Returns dict that contains a mapping from skill discovery name to class.

`tellStateTransition`(*state*, *action*, *reward*, *succState*)

Inform skill discovery method about a new state transition.

The agent has transitioned from *state* to *succState* after executing *action* and obtaining the reward *reward*.

Returns a pair consisting of the list of discovered options and whether a new option was discovered based on this state transition.

Predefined Skills

Local Graph Partitioning

3.6 Framework

Package that contains the core components of the MMLF.

3.6.1 World

Module containing a class that represents a RL world in the MMLF.

class `framework.world.`**`World`**(*worldConfigObject*, *baseUserDir*, *useGUI*)

Represents a world consisting of one agent and environment

The world is created based on a configuration dictionary (worldConfigObject). and provides methods for creating agent and environment based on this specification and for running the world a given number of steps or epsiodes.

`agentPollMethod`(*commandObject*)

Let the agent execute the given command object.

Valid command objects are found in the mmlf.framework.protocol module.

`createMonitor`(*monitorConf=None*)

Create monitor based on monitorConf.

`environmentPollMethod`(*commandObject*)

Let the environment execute the given command object.

Valid command objects are found in the mmlf.framework.protocol module.

`executeEpisode`(*monitorConf=None*)
Executes one episode in the current world.

`executeSteps`(*n=1*, *monitorConf=None*)
Executes n steps of the current world.

`getAgent`()
Returns the world's agent

`getEnvironment`()
Returns the world's environment

`loadAgent`(*agentConfig*, *useGUI*)
Create agent based on agent config dict.

`loadEnvironment`(*envConfig*, *useGUI*)
Create environment based on environment config dict.

`run`(*numOfEpisodes=inf*, *monitorConf=None*)
Start the execution of the current world.

Let the world run for *numOfEpisodes* episodes.

`setWorldPackageName`(*worldPackageName*)
Set the name of the python package from which the world should be taken.

`stop`()
Halt the execution of the current world.

3.6.2 Spaces

Modules for state and action spaces

State and action spaces define the range of possible states the agent might perceive and the actions that are available to the agent. These spaces are dict-like objects that map dimension names (the dict keys) to dimension objects that contain information about this dimension. The number of items in this dict-like structure is the dimensionality of the (state/action) space.

class `framework.spaces.`**`Dimension`**(*dimensionName*, *dimensionType*, *dimensionValues*, *limitType=None*)
A single dimension of a (state or action) space

A dimension is either continuous or discrete. A "discrete" dimension might take on only a finite, discrete number of values.

For instance, consider a dimension of a state space describing the color of a fruit. This dimension take on the values "red", "blue", or "green". In contrast, consider a second "continuous" dimension, e.g. the weight of a fruit. This weight might be somewhere between 0g and 1000g. If we allow any arbitrary weight (not only full gramms), the dimension is truly continuous.

This properties of a dimension can be checked using the method:

- *isDiscrete* : Returns whether the respective dimension is discrete
- *isContinuous* : Returns whether the respective dimension is continuous
- ***getValueRanges*** [Returns the allowed values a continuous! dimension] might take on.
- ***getValues*** [Returns the allowed values a discrete! dimension] might take on.

`getValueRanges`()
Returns the ranges the value of this dimension can take on

getValues()
Returns a list of all possible values of this dimension

isContinuous()
Returns whether this is a continuous dimension

isDiscrete()
Returns whether this is a discrete dimension

class `framework.spaces.`**Space**
Base class for state and action spaces.

Class which represents the state space of an environment or the action space of an agent.

This is essentially a dictionary whose keys are the names of the dimensions, and whose values are *Dimension* objects.

addContinuousDimension(*dimensionName*, *dimensionValues*, *limitType='soft'*)
Add the named continuous dimension to the space.

dimensionValues is a list of (rangeStart, rangeEnd) 2-tuples which define the valid ranges of this dimension. (i.e. [(0, 50), (75.5, 82)])

If limitType is set to "hard", then the agent is responsible to check that the limits are not exceeded. When it is set to "soft", then the agent should not expect that all the values of this dimension will be strictly within the bounds of the specified ranges, but that the ranges serve as an approximate values of where the values will be (i.e. as [mean-std.dev., mean+std.dev] instead of [absolute min. value, absolute max. value])

addDiscreteDimension(*dimensionName*, *dimensionValues*)
Add the named continuous dimension to the space.

dimensionValues is a list of strings representing possible discrete states of this dimension. (i.e. ["red", "green", "blue"])

addOldStyleSpace(*oldStyleSpace*, *limitType='soft'*)
Takes an old-style (using the old format) space dictionary, and adds its dimensions to this object.

getDimensionNames()
Return the names of the space dimensions

getDimensions()
Return the names of the space dimensions

getNumberOfDimensions()
Returns how many dimensions this space has

hasContinuousDimensions()
Return whether this space has continuous dimensions

hasDiscreteDimensions()
Return whether this space has discrete dimensions

class `framework.spaces.`**StateSpace**
Specialization of Space for state spaces.

For instance, a state space could be defined as follows:

```
{ "color": Dimension(dimensionType = "discrete",
                     dimensionValues = ["red","green", "blue"]),
  "weight": Dimension(dimensionType = "continuous",
                      dimensionValues = [(0,1000)]) }
```

This state space has two dimensions ("color" and "weight"), a discrete and a continuous one. The discrete dimension "color" can take on three values ("red","green", or "blue") and the continuous dimension "weight" any value between 0 and 1000.

A valid state of the state space defined above would be:

```
s1 = {"color": "red", "weight": 300}
```

Invalid states (s2 since the color is invalid and s3 since its weight is too large):

```
s2 = {"color": "yellow", "weight": 300}
s3 = {"color": "red", "weight": 1300}
```

The class provides additional methods for checking if a certain state is valid according to this state space (*isValidState*) and to scale a state such that it lies within a certain interval (*scaleState*).

getStateList()

Returns a list of all possible states.

Even if this state space has more than one dimension, it returns a one dimensional list that contains all possible states.

This is achieved by creating the crossproduct of the values of all dimensions. It requires that all dimensions are discrete.

class framework.spaces.**ActionSpace**

Specialization of Space for action spaces.

For instance, an action space could be defined as follows:

```
{ "gasPedalForce": ("discrete", ["low", "medium", "floored"]),
  "steeringWheelAngle": ("continuous", [(-120,120)]) }
```

This action space has two dimensions ("gasPedalForce" and "steeringWheelAngle"), a discrete and a continuous one. The discrete dimension "gasPedalForce" can take on three values ("low","medium", or "floored") and the continuous dimension "steeringWheelAngle" any value between -120 and 120.

A valid action according to this action space would be:

```
a1 = {"gasPedalForce": "low", "steeringWheelAngle": -50}
```

Invalid actions (a2 since the gasPedalForce is invalid and s3 since its steeringWheelAngle is too small):

```
a2 = {"gasPedalForce": "extreme", "steeringWheelAngle": 30}
a3 = {"gasPedalForce": "medium", "steeringWheelAngle": -150}
```

The class provides additional methods for discretizing an action space (*discretizedActionSpace*) and to return a list of all available actions (*getActionList*).

chopContinuousAction(*action*)

Chop a continuous action into the range of allowed values.

discretizedActionSpace(*discreteActionsPerDimension*)

Return a discretized version of this action space

Returns a discretized version of this action space. Every continuous action space dimension is discretized into *discreteActionsPerDimension* actions.

getActionList()

Returns a list of all allowed action an agent might take.

Even if this action space has more than one dimension, it returns a one dimensional list that contains all allowed action combinations.

This is achieved by creating the crossproduct of the values of all dimensions. It requires that all dimensions are discrete.

sampleRandomAction()
Returns a randomly sampled, valid action from the action space.

3.6.3 State

Module that contains the main State class of the MMLF

The MMLF uses a state class that is derived from numpy.array. In contrast to numpy.array, MMLF states can be hashed and thus be used as keys for dictionaries and set. This is realized by calling "hashlib.sha1(self).hexdigest()" In order to improve performance, State classes cache their hash value. Because of this, it is necessary to consider a state object to be constant, i.e. not changeable (except for calling "scale").

Each state object stores its state dimension definition. Furthermore, it implements a method "scale(self, minValue = 0, maxValue = 1):" which scales each dimension into the range (minValue, maxValue).

class `framework.state.`**State**
State class for the MMLF.

Based on numpy arrays, but can be used as dictionary key

hasDimension(*dimension*)
Returns whether this state has the dimension *dimension*

isContinuous()
Return whether this state is in a continuous state space

scale(*minValue=0*, *maxValue=1*)
Scale state such that it falls into the range (minValue, maxValue)

Scale the state so that (for each dimension) the range specified in this state space is scaled to the interval (minValue, maxValue)

3.6.4 Monitor

Monitoring of the MMLF based on logging performance metrics and graphics.

class `framework.monitor.`**Monitor**(*world*, *configDict*)
Monitor of the MMLF.

The monitor supervises the execution of a world within the MMLF and stores certain selected information periodically. It always stores the values a FloatStreamObservable takes on into a file with the suffix "fso". For other observables (FunctionOverStateSpaceObservable, StateActionValuesObservable, ModelObservable) a plot is generated and stored into files if this is specified in the monitor's config dict (using functionOverStateSpaceLogging, stateActionValuesLogging, modelLogging).

CONFIG DICT

policyLogFrequency : Frequency of storing the agent's policy in a serialized version to a file. The policy is stored in the file policy_x in the subdirectory "policy" of the agent's log directory where x is the episodes' numbers.

plotObservables : The names of the observables that should be stored to a file. If "All", all observables are stored. Defaults to "All" (also if plotObservables is not specified in a config file).

stateActionValuesLogging : Configuration of periodically plotting StateActionValuesObservables. Examples for StateActionValuesObservable are state-action value functions or stochastic policies. The plots are stored in the file episode_x.pdf in a subdirectory of the agent's log directory with the observable's name where x is the episodes' numbers.

functionOverStateSpaceLogging : Configuration of periodically plotting FunctionOverStateSpaceObservables. Examples for FunctionOverStateSpaceObservable are state value functions or deterministic policies. The plots are stored in the file episode_x.pdf in a subdirectory of the agent's log directory with the observable's name where x is the episodes' numbers.

modelLogging : Configuration of periodically plotting ModelObservables. Examples for ModelObservables are models. The plots are stored in the file episode_x.pdf in a subdirectory of the agent's log directory with the observable's name where x is the episodes' numbers.

active : Whether the respective kind of logging is activated

logFrequency : Frequency (in episodes) of creating a plot based on the respective observable and storing it to a file.

stateDims : The state space dimensions that are varied in the plot. All other state space dimensions are kept constant. If None, the stateDims are automatically deduced. This is only possible under specific conditions (2d state space)

actions : The actions for which separate plots are created for each StateActionValuesObservable.

rasterPoints : The resolution (rasterPoint*rasterPoint) of the plot in continuous domain.

colouring : The background colouring of a model plot. Can be either "Rewards" or "Exploration". If "Rewards", the reward predicted by the model is used for as background, while for "Exploration", each state-action pair tried at least minExplorationValue times is coloured with one colour, the others with an other colour.

plotSamples : If true, the state-action pairs that have been observed are plotted into the model-plot. Otherwise the model's predictions.

minExplorationValue : If the colouring is "Exploration", each state-action pair tried at least minExplorationValue times is coloured with one colour, the others with an other colour.

`notifyEndOfEpisode``()`
Notify monitor that an episode in the world has terminated.

3.6.5 Experiment

Classes and functions used for MMLF experiments.

class `framework.experiment.`**`RetryQueue`**`(`*maxsize=0*`)`
Queue which will retry if interrupted with EINTR.

class `framework.experiment.`**`WorldRunner`**`(`*worldConfigObject*, *numberOfEpisodes*, *exceptionOccurredSignal=None*`)`
Class which encapsulates running of a world in an MMLF experiment.

This class contains code for supporting the distributed, concurrent execution of worlds in an MMLF experiment. One instance of the class WorldRunner should be created for every run of a world. By calling this instance, the world is executed. This may happen in a separate thread or process since the instances of WorldRunner can be passed between threads and processes.

Parameters

worldConfigObject : The world configuration (in a dictionary)

numberOfEpisodes : The number of episodes that should be conducted in the world

exceptionOccurredSignal : An optional PyQt signal to which exceptions can be send

`framework.experiment.`**`runExperiment`**(*experimentConfig*, *observableQueue*, *updateQueue*, *exceptionOccurredSignal*, *useGUI=True*)

Handles the execution of an experiment.

This function handles the concurrent or sequential execution of an experiment.

Parameters

experimentConfig : The experiment configuration (in a dictionary)

observableQueue : A multiprocessing.Queue. Used for informing the main process (e.g. the GUI) of observables created in the world runs.

updateQueue : A multiprocessing.Queue. Used for informing the main process (e.g. the GUI) of changes in observables.

exceptionOccurredSignal : An optional PyQt signal to which exceptions can be send

3.6.6 Filesystem

class `framework.filesystem.`**`BaseUserDirectory`**(*basePath='/home/jmetzen/.mmlf'*)

Represents a user's base-directory (typically $HOME/.mmlf)

In this directory, the user can define worlds, YAML configuration files, and the interaction server logs information while running a world.

`addAbsolutePath`(*absDirPath*, *pathRef*, *force=False*)

Add an absolute path to self.pathDict. This is so that other commands can use this path via the pathRef shortcut.

The force argument allows an existing pathRef to be overridden. Normally this is not recommended, unless the pathRef is known not to be used globally.

`appendStringToPath`(*pathRef*, *stringToAppend=None*)

Modify the absolute path stored under pathRef, appending a "_<timestamp>" string to the end of the path (where <timestamp> will be something like 20070929_12_59_59, which represents 12:59:59 on 2007-09-29). If _<timestamp> has already been appended to the end of the path, it is replaced with a current timestamp.

If stringToAppend is provided, then a timestamp is *NOT* added, and instead, only the specified string is appended.

`createPath`(*pathList*, *refName=None*, *baseRef=None*, *force=False*)

Attept to create the path specified in pathList. pathList is a list which contains a series of directory names. For example, it might be ['abc','def','ghi'], which would cause this method to create all the necessary directories such that the following path exists:

$HOME/.mmlf/abc/def/ghi

refName is a quick reference name which can be used to refer to the path. If specified, the full path will be added to a dictionary, in which the key is the value of refName (a string, for example).

If the baseRef argument is provided, it will be looked up and used as the base directory to which the path defined in pathList will be appended. For example, if there already exists a 'logdir' refName in self.pathDict, then writing createPath(['a','b','c'], baseRef='logdir') might create (depending on how 'logdir' is defined) the directories needed for the path "$HOME/.mmlf/logs/a/b/c/".

The force argument allows an existing refName to be overridden. Normally this is not recommended, unless the refName is known not to be used globally.

fileExists(*pathRef*, *fileName*)
Check to see if the file (located in the path referred to by pathRef) exists.

getAbsolutePath(*pathRef*, *fileName=None*)
Return the absolute path to file fileName located in the directory referred to by pathRef. If no filename is specified, then only the path corresponding to pathRef is returned.

getFileAsText(*pathRef*, *fileName*)
Get the contents of the file fileName located in the directory referred to by pathRef, and return it as a string.

getFileObj(*pathRef*, *fileName*, *fileOpenMode='rb'*)
Get a file handle object for the fileName specified, assuming that the file is located in the directory referred to by pathRef (in self.pathDict). It is the programmer's responsibility to close this file object later on after it has been created..

setBasePath(*absolutePath=None*)
Set (and create if necessary) the base user-directory path. If a path is specified (as an absolute path string), then it is used. Otherwise, the default path is used.

If for some reason this fails, then a writeable temporary directory is chosen.

setTime(*timeStr=None*)
Sets the time string of baseUserDir (which is used to distinguish the logs of different runs).

If no timeString is given, it defaults to the current time

class `framework.filesystem.`**LogFile**(*baseUserDir*, *logFileName*, *baseRef=None*)
Represents a logfile, to which data can be written.

addText(*text*)
add the specified text to the logfile. Note, no newline is added to the logfile, so this must be included in the text string if it is desired.

3.6.7 Interaction Server

InteractionServer that handles the communication between environment and agent.

class `framework.interaction_server.`**InteractionServer**(*world*, *monitor*, *initialize=True*)
InteractionServer that handles the communication between environment and agent.

It controls the interaction between the environment and the agents, and allows direct control of various aspects of this interaction.

loopInitialize()
Inform environment about agent by giving its agentInfo.

loopIteration()
Perform one iteration of the interaction server loop.

Call the environmentPollMethod once and based on the result, decide if and how to call the agentPollMethod

run(*numOfEpisodes*)
Run an MMLF world for *numOfEpisodes* epsiodes.

stop()
Stop the execution of a world.

3.6.8 Protocol

class `framework.protocol.`**`Message`**`(`***kwargs*`)`
A base class which represents a command or response sent between the interaction server and agents/environment.

class `framework.protocol.`**`GiveAgentInfo`**`(`*agentInfo*`)`
Gives the Environment all information about the agents which the agents report on themselves.

AgentInfoList is a list of AgentInfo() objects. Each agent must make available such an object instance named "agentInfo". i.e. one can access an agents info by looking at agentObj.agentInfo

class `framework.protocol.`**`GetWish`**
Get the next wish/command of the Environment

The environment's transact method should return a single message object in response to receiving this message object via transact().

class `framework.protocol.`**`ActionTaken`**`(`*action=None*`)`
Communicate to the environment (from the interaction server) the action taken by an agent.

This message must be communicated via Env.transact() right after the 'wish' is requested by the environment. In other words, there should be no other message communicated via transact() before this message is communicated.

class `framework.protocol.`**`SetStateSpace`**`(`*stateSpace*`)`
Set the state space of one or more agents

class `framework.protocol.`**`SetState`**`(`*state*`)`
Set the state of one or more agents

class `framework.protocol.`**`SetActionSpace`**`(`*actionSpace*`)`
Set the action space of one or more agents

class `framework.protocol.`**`GetAction`**`(`*extendedTestingIsActive=False*`)`
Request (demand) an action from one or more agents.

If this request is issued from the Environment, then it will receive a response via a subsequent GiveResponse command.

class `framework.protocol.`**`GiveReward`**`(`*reward*`)`
Give a reward to one or more agents

class `framework.protocol.`**`NextEpisodeStarted`**
Communicate to an agent that a new episode has begun

class `framework.protocol.`**`AgentInfo`**`(`***kwargs*`)`
A class which represents the capabilities of an agent.

class `framework.protocol.`**`EnvironmentInfo`**`(`***kwargs*`)`
A class which represents the configuration of an environment.

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

a

agents, 51
agents.actor_critic_agent, 53
agents.agent_base, 52
agents.dps_agent, 54
agents.dyna_td_agent, 54
agents.fitted_r_max_agent, 55
agents.mbdps_agent, 56
agents.monte_carlo_agent, 56
agents.random_agent, 57
agents.td_agent, 57
agents.td_lambda_agent, 57

e

environments.single_agent_environment, 59

f

framework, 88
framework.experiment, 93
framework.filesystem, 94
framework.interaction_server, 95
framework.monitor, 92
framework.observables, 66
framework.protocol, 96
framework.spaces, 39
framework.state, 92
framework.world, 88

m

mmlf, 49

r

resources.function_approximators.cmac, 69
resources.function_approximators.function_approximator, 68
resources.function_approximators.knn, 69
resources.function_approximators.linear_combination, 69
resources.function_approximators.mlp, 70
resources.function_approximators.multilinear_grid, 70
resources.function_approximators.qcon, 71
resources.function_approximators.rbf, 71
resources.function_approximators.tabular_storage, 71
resources.learning_algorithms.eligibility_traces, 72
resources.learning_algorithms.temporal_difference, 72
resources.model.grid_model, 76
resources.model.knn_model, 77
resources.model.lwpr_model, 77
resources.model.model, 72
resources.model.rmax_model_wrapper, 78
resources.model.tabular_model, 78
resources.optimization.cmaes_optimizer, 79
resources.optimization.evolution_strategy, 80
resources.optimization.optimizer, 78
resources.optimization.random_search, 80
resources.planner.mbdps_planner, 81
resources.planner.planner, 81
resources.planner.prioritized_sweeping, 82
resources.planner.trajectory_sampling, 83
resources.planner.value_iteration, 83
resources.policies.linear_policy, 85
resources.policies.mlp_policy, 86
resources.policies.policy, 84
resources.policies.value_function_policy, 86
resources.policy_search.fixed_parametrization, 87

resources.policy_search.policy_search, 86
resources.skill_discovery.skill_discovery, 87

W

worlds, 28
worlds.double_pole_balancing.environments.double_pole_balancing_environment, 61
worlds.linear_markov_chain.environments.linear_markov_chain_environment, 61
worlds.maze2d.environments.maze2d_environment, 62
worlds.maze_cliff.environments.maze_cliff_environment, 62
worlds.mountain_car.environments.mcar_env, 63
worlds.pinball_maze.environments.pinball_maze_environment, 64
worlds.po_double_pole_balancing.environments.po_double_pole_balancing_environment, 63
worlds.seventeen_and_four.environments.seventeen_and_four, 65
worlds.single_pole_balancing.environments.single_pole_balancing_environment, 65

INDEX

A

ActionModel (class in resources.model.model), 76
ActionSpace (class in framework.spaces), 40, 91
ActionTaken (class in framework.protocol), 96
actionTaken() (environments.single_agent_environment.SingleAgentEnvironment method), 59
ActorCriticAgent (class in agents.actor_critic_agent), 53
addAbsolutePath() (framework.filesystem.BaseUserDirectory method), 94
addContinuousDimension() (framework.spaces.Space method), 39, 90
addDiscreteDimension() (framework.spaces.Space method), 40, 90
addExperience() (resources.model.model.ActionModel method), 76
addExperience() (resources.model.model.JointStateActionModel method), 75
addExperience() (resources.model.model.Model method), 73
addObservable() (framework.observables.AllObservables method), 66
addObserver() (framework.observables.Observable method), 67
addOldStyleSpace() (framework.spaces.Space method), 40, 90
addStartState() (resources.model.model.Model method), 73
addTerminalState() (resources.model.model.Model method), 73
addText() (framework.filesystem.LogFile method), 95
addTransition() (framework.observables.TrajectoryObservable method), 67
addValue() (framework.observables.FloatStreamObservable method), 66
AgentBase (class in agents.agent_base), 52
AgentInfo (class in framework.protocol), 96
agentPollMethod() (framework.world.World method), 88
agents (module), 27, 51
agents.actor_critic_agent (module), 53
agents.agent_base (module), 52
agents.dps_agent (module), 54
agents.dyna_td_agent (module), 54
agents.fitted_r_max_agent (module), 55
agents.mbdps_agent (module), 56
agents.monte_carlo_agent (module), 56
agents.random_agent (module), 57
agents.td_agent (module), 57
agents.td_lambda_agent (module), 57
AllObservables (class in framework.observables), 66
appendStringToPath() (framework.filesystem.BaseUserDirectory method), 94

B

BaseUserDirectory (class in framework.filesystem), 94

C

chopContinuousAction() (framework.spaces.ActionSpace method), 91
CMAC (class in resources.function_approximators.cmac), 69
CMAESOptimizer (class in resources.optimization.cmaes_optimizer), 79
computeOptimalAction() (resources.function_approximators.function_approximator.Function
method), 68
computeQ() (resources.function_approximators.function_approximator.Fun
method), 68
computeV() (resources.function_approximators.function_approximator.Fun
method), 68
create() (resources.function_approximators.function_approximator.Functio
static method), 68
create() (resources.model.model.Model static method), 73
create() (resources.optimization.optimizer.Optimizer static method), 79
create() (resources.planner.planner.Planner static method), 81
create() (resources.policies.policy.Policy static method), 84

create() (resources.policy_search.policy_search.PolicySearch static method), 87
create() (resources.skill_discovery.skill_discovery.SkillDiscovery static method), 88
createMonitor() (framework.world.World method), 88
createPath() (framework.filesystem.BaseUserDirectory method), 94

D

Dimension (class in framework.spaces), 39, 89
discretizedActionSpace() (framework.spaces.ActionSpace method), 91
DoublePoleBalancingEnvironment (class in worlds.double_pole_balancing.environments.double_pole_balancing_environment), 61
DPS_Agent (class in agents.dps_agent), 54
drawStartState() (resources.model.model.Model method), 74
DynaTDAgent (class in agents.dyna_td_agent), 54

E

EnvironmentInfo (class in framework.protocol), 96
environmentPollMethod() (framework.world.World method), 88
environments.single_agent_environment (module), 59
episodeFinished() (resources.skill_discovery.skill_discovery.SkillDiscovery method), 88
ESOptimizer (class in resources.optimization.evolution_strategy), 80
evaluate() (resources.policies.policy.Policy method), 85
evaluateAction() (environments.single_agent_environment.SingleAgentEnvironment method), 60
executeEpisode() (framework.world.World method), 89
executeSteps() (framework.world.World method), 89

F

fileExists() (framework.filesystem.BaseUserDirectory method), 94
FittedRMaxAgent (class in agents.fitted_r_max_agent), 55
FixedParametrization (class in resources.policy_search.fixed_parametrization), 87
FloatStreamObservable (class in framework.observables), 66
framework (module), 88
framework.experiment (module), 93
framework.filesystem (module), 94
framework.interaction_server (module), 95
framework.monitor (module), 92
framework.observables (module), 66
framework.protocol (module), 96
framework.spaces (module), 39, 89
framework.state (module), 92
framework.world (module), 88
FunctionApproximator (class in resources.function_approximators.function_approximator), 68
FunctionOverStateSpaceObservable (class in framework.observables), 66

G

getAbsolutePath() (framework.filesystem.BaseUserDirectory method), 95
GetAction (class in framework.protocol), 96
getAction() (agents.agent_base.AgentBase method), 52
getActionList() (framework.spaces.ActionSpace method), 91
getActionSpace() (environments.single_agent_environment.SingleAgentEnvironment method), 60
getActiveIndividual() (resources.optimization.optimizer.Optimizer method), 79
getActivePolicy() (resources.policy_search.policy_search.PolicySearch method), 87
getAllAgents() (framework.world.World method), 89
getAllIndividuals() (resources.optimization.optimizer.Optimizer method), 79
getAllObservablesOfType() (framework.observables.AllObservables method), 66
getBestIndividual() (resources.optimization.optimizer.Optimizer method), 79
getBestPolicy() (resources.policy_search.policy_search.PolicySearch method), 87
getConfidence() (resources.model.model.ActionModel method), 76
getConfidence() (resources.model.model.JointStateActionModel method), 75
getConfidence() (resources.model.model.Model method), 74
getDimensionNames() (framework.spaces.Space method), 40, 90
getDimensions() (framework.spaces.Space method), 40, 90
getEnvironment() (framework.world.World method), 89
getExpectedReward() (resources.model.model.ActionModel method), 76
getExpectedReward() (resources.model.model.JointStateActionModel method), 75

getExpectedReward() (resources.model.model.Model method), 74
getExplorationValue() (resources.model.model.ActionModel method), 76
getExplorationValue() (resources.model.model.JointStateActionModel method), 75
getExplorationValue() (resources.model.model.Model method), 74
getFileAsText() (framework.filesystem.BaseUserDirectory method), 95
getFileObj() (framework.filesystem.BaseUserDirectory method), 95
getFunctionApproximatorDict() (resources.function_approximators.function_approximator.FunctionApproximator static method), 69
getGreedyPolicy() (agents.agent_base.AgentBase method), 52
getInitialState() (environments.single_agent_environment.SingleAgentEnvironment method), 60
getModelDict() (resources.model.model.Model static method), 74
getNearestNeighbor() (resources.model.model.Model method), 74
getNearestNeighbors() (resources.model.model.Model method), 74
getNumberOfDimensions() (framework.spaces.Space method), 40, 90
getObservable() (framework.observables.AllObservables method), 66
getOptimizerDict() (resources.optimization.optimizer.Optimizer static method), 79
getParameters() (resources.policies.policy.Policy method), 85
getPlannerDict() (resources.planner.planner.Planner static method), 81
getPolicyDict() (resources.policies.policy.Policy static method), 85
getPolicySearchDict() (resources.policy_search.policy_search.PolicySearch static method), 87
getPredecessorDistribution() (resources.model.model.ActionModel method), 76
getPredecessorDistribution() (resources.model.model.JointStateActionModel method), 75
getPredecessorDistribution() (resources.model.model.Model method), 74
getRWPath() (in module mmlf), 49
getSample() (resources.model.model.JointStateActionModel method), 75
getSample() (resources.model.model.Model method), 74
getSkillDiscoveryDict() (resources.skill_discovery.skill_discovery.SkillDiscovery static method), 88
getStateList() (framework.spaces.StateSpace method), 91
getStates() (resources.model.model.Model method), 74
getStateSpace() (environments.single_agent_environment.SingleAgentEnvironment method), 60
getSuccessorDistribution() (resources.model.model.ActionModel method), 76
getSuccessorDistribution() (resources.model.model.JointStateActionModel method), 75
getSuccessorDistribution() (resources.model.model.Model method), 74
getValueRanges() (framework.spaces.Dimension method), 89
getValues() (framework.spaces.Dimension method), 89
GetWish (class in framework.protocol), 96
getWish() (environments.single_agent_environment.SingleAgentEnvironme method), 60
GiveAgentInfo (class in framework.protocol), 96
giveAgentInfo() (environments.single_agent_environment.SingleAgentEnvironment method), 60
GiveReward (class in framework.protocol), 96
giveReward() (agents.agent_base.AgentBase method), 52
GridModel (class in resources.model.grid_model), 76

H

hasContinuousDimensions() (framework.spaces.Space method), 40, 90
hasDimension() (framework.state.State method), 92
hasDiscreteDimensions() (framework.spaces.Space method), 40, 90

I

ImproperAgentException (class in environments.single_agent_environment), 60
initializeRWArea() (in module mmlf), 49
InteractionServer (class in framework.interaction_server), 95
isContinuous() (framework.spaces.Dimension method), 90
isContinuous() (framework.state.State method), 92
isDiscrete() (framework.spaces.Dimension method), 90
isTerminalState() (resources.model.model.Model method), 74

J

JointStateActionModel (class in resources.model.model), 75

K

KNNFunctionApproximator (class in resources.function_approximators.knn), 69
KNNModel (class in resources.model.knn_model), 77

L

LinearCombination (class in resources.function_approximators.linear_combination), 70
LinearContinuousActionPolicy (class in resources.policies.linear_policy), 85
LinearDiscreteActionPolicy (class in resources.policies.linear_policy), 85
LinearMarkovChainEnvironment (class in worlds.linear_markov_chain.environments.linear_markov_chain_environment), 61
loadAgent() (framework.world.World method), 89
loadEnvironment() (framework.world.World method), 89
loadExperimentConfig() (in module mmlf), 49
loadWorld() (in module mmlf), 49
loadWorldFromConfigFile() (in module mmlf), 50
LogFile (class in framework.filesystem), 95
loopInitialize() (framework.interaction_server.InteractionServer method), 95
loopIteration() (framework.interaction_server.InteractionServer method), 95
LWPRModel (class in resources.model.lwpr_model), 77

M

Maze2dEnvironment (class in worlds.maze2d.environments.maze2d_environment), 62
MazeCliffEnvironment (class in worlds.maze_cliff.environments.maze_cliff_environment), 62
MBDPS_Agent (class in agents.mbdps_agent), 56
MBDPSPlanner (class in resources.planner.mbdps_planner), 81
Message (class in framework.protocol), 96
MLP (class in resources.function_approximators.mlp), 70
MLPPolicy (class in resources.policies.mlp_policy), 86
mmlf (module), 49
Model (class in resources.model.model), 72
ModelObservable (class in framework.observables), 67
Monitor (class in framework.monitor), 38, 92
MonteCarloAgent (class in agents.monte_carlo_agent), 56
MountainCarEnvironment (class in worlds.mountain_car.environments.mcar_env), 63
MultilinearGrid (class in resources.function_approximators.multilinear_grid), 70

N

NextEpisodeStarted (class in framework.protocol), 96
nextEpisodeStarted() (agents.agent_base.AgentBase method), 52
notifyEndOfEpisode() (framework.monitor.Monitor method), 93

O

Observable (class in framework.observables), 67
Optimizer (class in resources.optimization.optimizer), 78

P

PinballMazeEnvironment (class in worlds.pinball_maze.environments.pinball_maze_environment), 64
Planner (class in resources.planner.planner), 81
plot() (framework.observables.FunctionOverStateSpaceObservable method), 66
plot() (framework.observables.ModelObservable method), 67
plot() (framework.observables.StateActionValuesObservable method), 67
plotStateSpaceStructure() (environments.single_agent_environment.SingleAgentEnvironment method), 60
PODoublePoleBalancingEnvironment (class in worlds.po_double_pole_balancing.environments.po_double_pole
63
Policy (class in resources.policies.policy), 84
PolicySearch (class in resources.policy_search.policy_search), 87
PrioritizedSweepingPlanner (class in resources.planner.prioritized_sweeping), 82

Q

QCON (class in resources.function_approximators.qcon), 71

R

RandomAgent (class in agents.random_agent), 57
RandomSearchOptimizer (class in resources.optimization.random_search), 80
RBF_FA (class in resources.function_approximators.rbf), 71
removeObservable() (framework.observables.AllObservables method), 66
removeObserver() (framework.observables.Observable method), 67

reset() (resources.policy_search.policy_search.PolicySearch method), 87
resources.function_approximators.cmac (module), 69
resources.function_approximators.function_approximator (module), 68
resources.function_approximators.knn (module), 69
resources.function_approximators.linear_combination (module), 69
resources.function_approximators.mlp (module), 70
resources.function_approximators.multilinear_grid (module), 70
resources.function_approximators.qcon (module), 71
resources.function_approximators.rbf (module), 71
resources.function_approximators.tabular_storage (module), 71
resources.learning_algorithms.eligibility_traces (module), 72
resources.learning_algorithms.temporal_difference (module), 72
resources.model.grid_model (module), 76
resources.model.knn_model (module), 77
resources.model.lwpr_model (module), 77
resources.model.model (module), 72
resources.model.rmax_model_wrapper (module), 78
resources.model.tabular_model (module), 78
resources.optimization.cmaes_optimizer (module), 79
resources.optimization.evolution_strategy (module), 80
resources.optimization.optimizer (module), 78
resources.optimization.random_search (module), 80
resources.planner.mbdps_planner (module), 81
resources.planner.planner (module), 81
resources.planner.prioritized_sweeping (module), 82
resources.planner.trajectory_sampling (module), 83
resources.planner.value_iteration (module), 83
resources.policies.linear_policy (module), 85
resources.policies.mlp_policy (module), 86
resources.policies.policy (module), 84
resources.policies.value_function_policy (module), 86
resources.policy_search.fixed_parametrization (module), 87
resources.policy_search.policy_search (module), 86
resources.skill_discovery.skill_discovery (module), 87
RetryQueue (class in framework.experiment), 93
RMaxModelWrapper (class in resources.model.rmax_model_wrapper), 78
run() (framework.interaction_server.InteractionServer method), 95
run() (framework.world.World method), 89
runExperiment() (in module framework.experiment), 94
runExperiment() (in module mmlf), 50
RWAreaInitializationFailedException, 49

S

samplePredecessorState() (resources.model.model.ActionModel method), 76
samplePredecessorState() (resources.model.model.JointStateActionModel method), 75
samplePredecessorState() (resources.model.model.Model method), 74
sampleRandomAction() (framework.spaces.ActionSpace method), 92
sampleState() (resources.model.model.ActionModel method), 76
sampleSuccessorState() (resources.model.model.ActionModel method), 76
sampleSuccessorState() (resources.model.model.JointStateActionModel method), 76
sampleSuccessorState() (resources.model.model.Model method), 74
scale() (framework.state.State method), 92
SetActionSpace (class in framework.protocol), 96
setActionSpace() (agents.agent_base.AgentBase method), 52
setBasePath() (framework.filesystem.BaseUserDirectory method), 95
setParameters() (resources.policies.policy.Policy method), 85
SetState (class in framework.protocol), 96
setState() (agents.agent_base.AgentBase method), 53
setStates() (resources.planner.planner.Planner method), 81
SetStateSpace (class in framework.protocol), 96
setStateSpace() (agents.agent_base.AgentBase method), 53
setTime() (framework.filesystem.BaseUserDirectory method), 95
setupConsoleLogging() (in module mmlf), 50
setupFileLogging() (in module mmlf), 50
setWorldPackageName() (framework.world.World method), 89
SeventeenAndFourEnvironment (class in worlds.seventeen_and_four.environments.seventeen_and_four), 65
SingleAgentEnvironment (class in environments.single_agent_environment), 59
SinglePoleBalancingEnvironment (class in worlds.single_pole_balancing.environments.single_pole_balancin
65
SkillDiscovery (class in resources.skill_discovery.skill_discovery), 88
Space (class in framework.spaces), 39, 90

State (class in framework.state), 92
StateActionValuesObservable (class in framework.observables), 67
StateSpace (class in framework.spaces), 40, 90
stop() (environments.single_agent_environment.SingleAgentEnvironment method), 60
stop() (framework.interaction_server.InteractionServer method), 95
stop() (framework.world.World method), 89
storePolicy() (agents.agent_base.AgentBase method), 53

T

TabularModel (class in resources.model.tabular_model), 78
TabularStorage (class in resources.function_approximators.tabular_storage), 71
TDAgent (class in agents.td_agent), 57
TDLambdaAgent (class in agents.td_lambda_agent), 57
tellAllFitness() (resources.optimization.optimizer.Optimizer method), 79
tellFitness() (resources.optimization.optimizer.Optimizer method), 79
tellFitness() (resources.policy_search.policy_search.PolicySearch method), 87
tellStateTransition() (resources.skill_discovery.skill_discovery.SkillDiscovery method), 88
train() (resources.function_approximators.function_approximator.FunctionApproximator method), 69
TrajectoryObservable (class in framework.observables), 67
TrajectorySamplingPlanner (class in resources.planner.trajectory_sampling), 83

U

updateFunction() (framework.observables.FunctionOverStateSpaceObservable method), 66
updateModel() (framework.observables.ModelObservable method), 67
updateValues() (framework.observables.StateActionValuesObservable method), 67

V

ValueFunctionPolicy (class in resources.policies.value_function_policy), 86
ValueIterationPlanner (class in resources.planner.value_iteration), 84

W

World (class in framework.world), 88
WorldRunner (class in framework.experiment), 93
worlds (module), 28, 58
worlds.double_pole_balancing.environments.double_pole_balancing_enviro (module), 61
worlds.linear_markov_chain.environments.linear_markov_chain_environme (module), 61
worlds.maze2d.environments.maze2d_environment (module), 62
worlds.maze_cliff.environments.maze_cliff_environment (module), 62
worlds.mountain_car.environments.mcar_env (module), 63
worlds.pinball_maze.environments.pinball_maze_environment (module), 64
worlds.po_double_pole_balancing.environments.po_double_pole_balancing (module), 63
worlds.seventeen_and_four.environments.seventeen_and_four (module), 65
worlds.single_pole_balancing.environments.single_pole_balancing_environ (module), 65