ReLe Documentation

Release

Matteo Pirotta, Davide Tateo

February 28, 2017

Contents

This is the documentation for ReLe , last updated February 28, 2017.

CHAPTER 1

Introduction

ReLe is a C++ library for REinforcement LEarning developped and maintained by the Politecnico di Milano. Its main features and design principles are:

- Template
- C++11

ReLe is licensed under the GPLV3 license; see LICENSE in the source distribution for details.

CHAPTER 2

Contents

Warning: Please be advised that the reference documentation discussing ReLe internals is currently incomplete. Please refer to the previous sections and the ReLe header files for the nitty gritty details.

API Reference

All functions and classes provided by the C++ Format library reside in namespace `ReLe`.

Library Core

Basic concepts

class `ReLe::`**`Action`**

Abstract class for all actions

Subclassed by *ReLe::DenseAction*, *ReLe::FiniteAction*

Public Functions

virtual std::string **`to_str`**`()` **const**

writes the action as string. Should be overridden

virtual int **`serializedSize`**`()`

Getter. Should be overridden.

Return the size of the action if serialized as a csv string

virtual **`~Action`**`()`

Destructor

Public Static Functions

static *Action* **`generate`**`(`std::vector<std::string>::iterator *&begin*, std::vector<std::string>::iterator *&end*`)`

factory method that transform a set of strings into an action.

Return the action build from strings

Parameters

- `begin`: the iterator of the first element in the range
- `end`: the iterator to the end of the range

class `ReLe::FiniteAction`

Finite action class. Actions of this types are described by a finite subset of unsigned integers.

Inherits from *ReLe::Action*

Public Functions

`FiniteAction`(unsigned int *n* = 0)

Constructor

Parameters

- `n`: the action number

`operator unsigned int&`()

This operator is used to convert the action class to an integer

Return a reference to the action number

`operator const unsigned int&`() **const**

This operator is used to convert the action class to an integer

Return a const reference to the action number

unsigned int **`getActionN`**() **const**

Getter.

Return the action number

void **`setActionN`**(unsigned int *actionN*)

Setter.

Parameters

- `actionN`: the action number to be set

virtual std::string **`to_str`**() **const**

writes the action as string. Should be overridden

int **`serializedSize`**()

Getter. Should be overridden.

Return the size of the action if serialized as a csv string

virtual **`~FiniteAction`**()

Destructor.

Public Static Functions

static *FiniteAction* **`generate`**(std::vector<std::string>::iterator &*begin*, std::vector<std::string>::iterator &*end*)

factory method that transform a set of strings into an action.

Return the action build from strings

Parameters

- `begin`: the iterator of the first element in the range
- `end`: the iterator to the end of the range

static std::vector<*FiniteAction*> **generate**(size_t *actionN*)
factory method that builds the first N actions

Return all the actions from 0 to actionN-1

Parameters

- `actionN`: the number of actions to be build

class `ReLe::DenseAction`
Dense action class. This class represent an action u, such that $u \in \mathbb{R}^{\kappa}$, where n is the dimensionality of action space

Inherits from *ReLe::Action*, vec

Public Functions

DenseAction()
Constructor. Builds an action with zero dimensions

DenseAction(std::size_t *size*)
Constructor.

Parameters

- `size`: the action dimensionality

DenseAction(arma::vec &*other*)
Constructor. Initialize the action from an armadillo vector

Parameters

- `other`: the vector to copy as action

virtual std::string **to_str**() **const**
writes the action as string. Should be overridden

int **serializedSize**()
Getter. Should be overridden.

Return the size of the action if serialized as a csv string

virtual ~DenseAction()
Destructor.

virtual bool **isAlmostEqual**(**const** arma::vec &*other*, double *epsilon* = 1e-6) **const**
Check whether two actions are almost equals

Return whether all elements differences are below the tolerance

Parameters

- `other`: the action to chek againts
- `epsilon`: the tolerance for each element

Public Static Functions

static *DenseAction* **generate**(std::vector<std::string>::iterator &*begin*, std::vector<std::string>::iterator &*end*)
factory method that transform a set of strings into an action.

Return the action build from strings

Parameters

- `begin`: the iterator of the first element in the range
- `end`: the iterator to the end of the range

class `ReLe::State`
Abstract class for all states

Subclassed by *ReLe::DenseState*, *ReLe::FiniteState*

Public Functions

State()
Constructor.

bool **isAbsorbing**() **const**
Getter.

Return whether this state is an absorbing one

void **setAbsorbing**(bool *absorbing* = true)
Setter.

Parameters

- `absorbing`: whether to set this state as an absorbing state or not.

virtual std::string **to_str**() **const**
writes the state as string. Should be overridden

virtual int **serializedSize**()
Getter. Should be overridden.

Return the size of the state if serialized as a csv string

class `ReLe::FiniteState`
Finite state class. States of this types are described by a finite subset of unsigned integers.

Inherits from *ReLe::State*

Public Functions

FiniteState(unsigned int *n* = 0)
Constructor

Parameters

- `n`: the state number

operator size_t&()
This operator is used to convert the state class to an integer

Return a reference to the state number

operator const size_t&() const
This operator is used to convert the state class to an integer

Return a const reference to the state number

std::size_t **getStateN**() **const**
Getter.

Return the action number

void **setStateN**(std::size_t *stateN*)
Setter.

Parameters

- stateN: the action number to be set

virtual std::string **to_str**() **const**
writes the state as string. Should be overridden

int **serializedSize**()
Getter. Should be overridden.

Return the size of the state if serialized as a csv string

virtual ~FiniteState()
Destructor.

class ReLe::**DenseState**
Dense action class. This class represent a state x, such that $x \in \mathbb{R}^{\times}$, where n is the dimensionality of state space

Inherits from *ReLe::State*, vec

Public Functions

DenseState()
Constructor. Builds a state with zero dimensions

DenseState(std::size_t *size*)
Constructor.

Parameters

- size: the state dimensionality

virtual std::string **to_str**() **const**
writes the state as string. Should be overridden

int **serializedSize**()
Getter. Should be overridden.

Return the size of the state if serialized as a csv string

virtual ~DenseState()
Destructor.

struct ReLe::**EnvironmentSettings**
Basic struct to describe an environment

Subclassed by ReLe::AcrobotSettings, ReLe::CarOnHillSettings, ReLe::DamSettings, ReLe::MultiHeatSettings, ReLe::NLSSettings, ReLe::PortfolioSettings, ReLe::SegwaySettings, ReLe::SwingUpSettings, ReLe::UnicyclePolarSettings, ReLe::UWVSettings, ReLe::WaterResourcesSettings

Public Functions

void **writeToStream**(std::ostream &*out*) **const**
Writes the struct to stream

void **readFromStream**(std::istream &*in*)
Reads the struct from stream

Public Members

bool **isFiniteHorizon**
Finite horizon flag.

bool **isAverageReward**
Average reward flag.

bool **isEpisodic**
Episodic environment flag.

double **gamma**
Discount factor.

unsigned int **horizon**
horizon of mdp

size_t **statesNumber**
number of finite states of the mdp. Should be zero for continuos state spaces

unsigned int **actionsNumber**
number of finite actions of the mdp. Should be zero for continuos action spaces

unsigned int **stateDimensionality**
number of dimensions of the state space states of the mdp. Should be one for finite state spaces

unsigned int **actionDimensionality**
number of dimensions of the state space states of the mdp. Should be one for finite state spaces

unsigned int **rewardDimensionality**
number of dimensions of the reward function

arma::vec **max_obj**
vector of maximum value of each dimension of the reward function

class ReLe::**AgentOutputData**
The basic interface to log data from Agents. In contains some common data and basic methods to process generic data.

Subclassed by ReLe::BlackBoxOutputData< BlackBoxPolicyIndividual >, ReLe::BlackBoxOutputData< PGPEPolicyIndividual >, ReLe::AbstractREPSOutputData, ReLe::BlackBoxOutputData< IndividualsClass >, *ReLe::FiniteTDOutput*, *ReLe::FQIOutput*, ReLe::GradientIndividual, *ReLe::LinearTDOutput*, ReLe::LSPIOutput

Public Functions

AgentOutputData(bool *final* = false)
Constructor

Parameters

- `final`: whether the data logged comes from the end of a run of the algorithm

virtual void **writeData**(std::ostream &*os*) = 0
Basic method to write plain data.

Parameters

- `os`: output stream in which the data should be logged

virtual void **writeDecoratedData**(std::ostream &*os*) = 0
Basic method to write decorated data, e.g. for printing on screen.

Parameters

- `os`: output stream in which the data should be logged

virtual ~AgentOutputData()
Destructor

bool **isFinal**() **const**
Getter.

Return if the data logged comes from the end of the run or not.

unsigned int **getStep**() **const**
Getter.

Return the step at which the data was logged.

void **setStep**(unsigned int *step*)
Setter. Sets the data step number. Should be used only by the *Core*.

template <class *ActionC*>
struct `ReLe::action_type`
This trait is used to get the appropriate raw types for action classes You need to specialize this traits to get the correct support for new action types

template <class *StateC*>
struct `ReLe::state_type`
This trait is used to get the appropriate raw types for state classes You need to specialize this traits to get the correct support for new state types

Trajectories

template <class *ActionC*, class *StateC*>
struct `ReLe::Transition`
This struct represent a single transition of the environment (the *SARSA* tuple). Implements some convenience setters and serialization functions

Public Functions

void **init**(**const** StateC &*x*)
Setter.

Parameters

- `x`: the initial state

void **update**(**const** ActionC &*u*, **const** StateC &*xn*, **const** Reward &*r*)
Setter.

Parameters

- `u`: the action performed
- `xn`: the state reached after the performed action
- `r`: the reward achieved

void **printHeader**(std::ostream &*os*)
This method prints the transition header, that consist of tree numbers, that represent the number of comma separated values needed to represent the state, the action and the reward.

void **print**(std::ostream &*os*)
Print the first part of the transition, with non final/non absorbing flag. This function is meaningful when multiple transitions are serialized

void **printLast**(std::ostream &*os*)
Print the last part of the transition, with final flag ad appropriate absorbing flag. This function is meaningful when multiple transitions are serialized

Public Members

StateC **x**
The initial state of the transition.

ActionC **u**
The performed action.

StateC **xn**
The state reached after performing the action.

Reward **r**
The reward achieved in the transition.

template <class *ActionC*, class *StateC*>
class `ReLe::Episode`
This class is the collection of multiple transition of a single episode The transition stored in this class should be a meaningful sequence, i.e. the next state of the previous transition should be the initial state of the subsequent one. This class contains some utility functions to perform common operations over an episode.

Inherits from std::vector< Transition< ActionC, StateC > >

Public Functions

arma::mat **computefeatureExpectation**(Features &*phi*, double *gamma* = 1)
This method can be used to compute episode features expectation over transitions.

Return a matrix of features expectation, with size phi.rows() ×phi.cols()

Parameters

- `phi`: the features $\phi(x, u, x_n)$ to be used
- `gamma`: the discount factor for the features expectations

arma::vec **getEpisodeReward**(double *gamma*)
This method returns the episode expected reward

Return the expected reward using gamma as discount factor

Parameters

- `gamma`: the discount factor to be used

unsigned int **`getRewardSize`**`()`
Getter.

Return the reward size

void **`printHeader`**`(std::ostream &os)`
Print the trasition header

See *Transition::printHeader*

void **`print`**`(std::ostream &os)`
Print the dataset to stream

void **`printDecorated`**`(std::ostream &os)`
Writes the decorated verion of the episode (with repetitions and extra tokens)

template <class *ActionC*, class *StateC*>
class `ReLe::Dataset`
This class represents a dataset, a set of episodes. This class contains some utility functions to perform common operations over a dataset.

Inherits from std::vector< Episode< ActionC, StateC > >

Public Functions

arma::mat **`computefeatureExpectation`**`(Features &phi, double gamma = 1)`
Computes the mean features expectations over the episodes

Return a matrix of features expectation, with size phi.rows() ×phi.cols()

Parameters

- `phi`: the features $\phi(x, u, x_n)$ to be used
- `gamma`: the discount factor for the features expectations

arma::mat **`computeEpisodeFeatureExpectation`**`(Features &phi, double gamma = 1)`
Computes the features expectations over the episodes

Return a matrix of features expectation, with size (phi.rows() ∗phi.cols()) ×this->size()

Parameters

- `phi`: the features $\phi(x, u, x_n)$ to be used
- `gamma`: the discount factor for the features expectations

unsigned int **`getTransitionsNumber`**`() const`
Getter.

Return the total number of transitions contained in this dataset

unsigned int **`getRewardSize`**`()`
Getter.

Return the reward size of this dataset

arma::mat **`getEpisodesReward`**`(double gamma)`
Computes the episode discounted reward

Return a matrix of the discounted reward, with size this->*getRewardSize()* ×this->size()

Parameters

- `gamma`: the discount factor to be used

arma::vec **getMeanReward**(double *gamma*)
Computes the mean discounted reward.

Return a vector of the mean discounted reward, with size this->*getRewardSize()*

Parameters

- `gamma`: the discount factor to be used

arma::mat **rewardAsMatrix**()
Get all transitions rewards as matrix

Return a matrix of the transition rewards, with size this->*getRewardSize()* ×this->*getTransitionsNumber()*

arma::mat **featuresAsMatrix**(Features &*phi*)
Get all features over transitions as matrix

Return a matrix of the transition features, with size (phi.rows() ∗phi.cols()) ×this->*getTransitionsNumber()*

Parameters

- `phi`: the features $\phi(x, u, x_n)$ to be used

void **addData**(*Dataset*<ActionC, StateC> &*data*)
This method joins two datasets

Parameters

- `data`: the new dataset to join to the prevoious one.

void **setData**(*Dataset*<ActionC, StateC> &*data*)
Setter.

Parameters

- `data`: the new dataset to be set

unsigned int **getEpisodesNumber**()
Getter.

Return the number of episodes in this dataset

unsigned int **getEpisodeMaxLength**()
Getter.

Return the maximum episode length

void **writeToStream**(std::ostream &*os*)
This method write an episode to an output stream

Parameters

- `os`: the output stream

void **readFromStream**(std::istream &*is*)
This method read an episode from an input stream

Parameters

- `is`: the input stream

void **printDecorated**(std::ostream &*os*)
Writes the decorated verion of the dataset (with repetitions and extra tokens)

Basic Interfaces

template <class *ActionC*, class *StateC*>
class `ReLe::Environment`
The environment is the basic interface

Subclassed by ReLe::ParametricRewardMDP< ActionC, StateC >

Public Functions

`Environment`()
Constructor

`Environment`(*EnvironmentSettings* **settings*)
Constructor

Parameters

- `settings`: a pointer to the environment settings

virtual void **`step`**(**const** ActionC &*action*, StateC &*nextState*, Reward &*reward*) = 0
This function is called to execute an action on the environment. Must be implemented.

Parameters

- `action`: the action to be executed at this time step
- `nextState`: the state reached after performing the action
- `reward`: the reward achieved by performing the last action on the environment

virtual void **`getInitialState`**(StateC &*state*) = 0
This function is called to get the initial environment state. Must be implemented.

Parameters

- `state`: the initial state

const *EnvironmentSettings* &**`getSettings`**() **const**
Getter.

Return a const reference to the environment settings

void **`setHorizon`**(unsigned int *h*)
Setter.

Parameters

- `h`: the new environment horizon

virtual ~`Environment`()
Destructor.

Protected Functions

EnvironmentSettings &**`getWritableSettings`**()
Getter. To be used in derived classes.

Return a reference to the environment settings

Protected Attributes

EnvironmentSettings ***settings**
the environment settings

template <class *ActionC*, class *StateC*>
class ReLe::**Agent**

The *Agent* is the basic interface of all online agents. All online algorithms should extend this abstract class. The *Agent* interface provides all the methods that can be used to interact with an environment through the *Core* class. It includes methods to run the learning over an environment and to test the learned policy.

Subclassed by ReLe::BlackBoxAlgorithm< ActionC, StateC, EMOutputData >, ReLe::BlackBoxAlgorithm< ActionC, StateC, NESIterationStats >, ReLe::BlackBoxAlgorithm< ActionC, StateC, PGPEIterationStats >, ReLe::BlackBoxAlgorithm< ActionC, StateC, REPSOutputData >, ReLe::AbstractPolicyGradientAlgorithm< ActionC, StateC >, ReLe::BlackBoxAlgorithm< ActionC, StateC, AgentOutputC >, *ReLe::PolicyEvalAgent< ActionC, StateC >*

Public Functions

virtual void **initTestEpisode**()
This method is called at the beginning of each test episode. by default does nothing, but can be overloaded

virtual void **initEpisode**(**const** StateC &*state*, ActionC &*action*) = 0
This method is called at the beginning of each learning episode. Must be implemented. Normally this method contains the algorithm initialization.

Parameters

- `state`: the initial environment state
- `action`: the action selected by the agent in the initial state

virtual void **sampleAction**(**const** StateC &*state*, ActionC &*action*) = 0
This method is used to sample an action in test episodes. Must be implemented. Normally, this method is trivial, as it just sample an action from a policy.

Parameters

- `state`: the current environment state
- `action`: the action selected by the agent in the current state

virtual void **step**(**const** Reward &*reward*, **const** StateC &*nextState*, ActionC &*action*) = 0
This method is used during each learning step. Must be implemented. Normally this method contains the learning algorithm, for step-based agents, or data collection for episode-based agents.

Parameters

- `reward`: the reward achieved in the previous learning step.
- `nextState`: the state reached after the previous learning state i.e. the current state
- `action`: the action selected by the agent in the current state

virtual void **endEpisode**(**const** Reward &*reward*) = 0
This method is called if an episode ends in a terminal state. Must be implemented. Normally this method contains the learning algorithm.

Parameters

- `reward`: the reward achieved after reaching the terminal state.

virtual void **endEpisode**() = 0
This method is called if an episode ends after reaching the maximum number of iterations. Must be implemented. Normally this method contains the learning algorithm.

virtual *AgentOutputData* ***getAgentOutputData**()
This method is used to log agent step informations. Can be overloaded to return information that can be processed by the logger. By default a null pointer is returned, which means that no data will be logged.

Return the data to be logged from the agent at the current step.

virtual *AgentOutputData* ***getAgentOutputDataEnd**()
This method is used to log agent informations at episode end. Can be overloaded to return information that can be processed by the logger. By default a null pointer is returned, which means that no data will be logged.

Return the data to be logged from the agent at the episode end.

bool **isTerminalConditionReached**()
This method is called before each learning step and return if the terminal condition has been reached. Terminal condition can be implemented by setting the *Agent::terminalCond* member.

Return whether the terminal condition has been reached.

void **setTask**(**const** *EnvironmentSettings* &*task*)
This method sets the agent task, i.e. the environment properties. This method also calls *Agent::init()*

Parameters

- `task`: the task properties of the environment

Protected Functions

virtual void **init**()
This method is called after the agent task has been set. By default does nothing, but can be overloaded with agent initialization, e.g. Q table allocation.

Protected Attributes

EnvironmentSettings **task**
The task that the agent will perform.

TerminalCondition ***terminalCond**
The terminal condition of the agent.

template <class *ActionC*, class *StateC*>
class `ReLe::BatchAgent`
The *BatchAgent* is the basic interface of all batch agents. All batch algorithms should extend this abstract class. The *BatchAgent* interface provides all the methods that can be used to interact with an MDP through the *BatchCore* class. It includes methods to run the learning over a dataset and log the progress of the algorithm.

Subclassed by ReLe::AbstractOffPolicyGradientAlgorithm< ActionC, StateC >, ReLe::MBPGA< ActionC, StateC >, ReLe::OffPolicyGradientAlgorithm< ActionC, StateC >

Public Functions

virtual void **init**(*Dataset*<ActionC, StateC> &*data*) = 0
This method setup the dataset to be used in the learning process. Must be implemented. Is called by the

BatchCore as the first step of the learning process.

Parameters

- `data`: the dataset to be used for learning

virtual void **`step`**`()` = 0
This method implement a step of the learning process trough the dataset. Must be implemented. Is called by the *BatchCore* until the algorithm converges or the maximum number of iteration is reached.

virtual *AgentOutputData* ***`getAgentOutputData`**`()`
This method is used to log agent step informations. Can be overloaded to return information that can be processed by the logger. By default a null pointer is returned, which means that no data will be logged.

Return the data to be logged from the agent at the current step.

virtual *AgentOutputData* ***`getAgentOutputDataEnd`**`()`
This method is used to log agent informations at episode end. Can be overloaded to return information that can be processed by the logger. By default a null pointer is returned, which means that no data will be logged.

Return the data to be logged from the agent at the episode end.

virtual *Policy*<ActionC, StateC> ***`getPolicy`**`()` = 0
Getter.

Return the policy learned by the agent

virtual bool **`hasConverged`**`()`
This method returns whether the algorithm has converged or not

Return the value of the flag converged

void **`setTask`**`(`**const** *EnvironmentSettings* *&task*`)`
This method sets the agent task, i.e. the environment properties. This method also calls *Agent::init()*

Parameters

- `task`: the task properties of the environment

Protected Functions

virtual void **`init`**`()`
This method is called after the agent task has been set. By default does nothing, but can be overloaded with agent initialization, e.g. Q table allocation.

Protected Attributes

EnvironmentSettings **`task`**
The task from which the data comes.

bool **`converged`**
flag to signal convergence of the algorithm

template <class *ActionC*, class *StateC*>
class `ReLe::`**`Core`**
This class implements both learning and testing of an agent over an environment. This class is able to run the agent on the environment, while logging both the environment and agent data. Both experiment parameters and logger are configurable. The core takes in account environment terminal states and agent termination conditions. Also gives to the agent the environment settings, calling *Agent::setTask*

Public Functions

Core(*Environment*<ActionC, StateC> &*environment*, *Agent*<ActionC, StateC> &*agent*)
Constructor.

Parameters

- `environment`: the environment used for the experiment
- `agent`: the agent used for the experiment

CoreSettings &**getSettings**()
Getter. Used to set the core parameters. Example:

```
core.getSettings().loggerStrategy = new PrintStrategy<FiniteAction, FiniteState>(false);
core.getSettings().episodeLength = 100;
core.getSettings().episodeN = 1000;
core.getSettings().testEpisodeN = 200;
```

Return a reference to the core settings.

void **runEpisode**()
This method runs a single learning episode.

void **runEpisodes**()
This method runs the learning episodes specified in the settings.

void **runTestEpisode**()
This method runs a single learning episode.

void **runTestEpisodes**()
This method runs the test episodes specified in the settings.

arma::vec **runEvaluation**()
This method runs the test episodes specified in the settings and computes the expected return of the agent w.r.t. the environment

struct CoreSettings
This struct stores the core parameters.

Public Members

LoggerStrategy<ActionC, StateC> ***loggerStrategy**
The logger strategy, or a null pointer if no data should be logged.

CoreCallback ***episodeCallback**
A callback to be called at each episode end.

unsigned int **episodeLength**
The length of episodes.

unsigned int **episodeN**
The number of learning episodes.

unsigned int **testEpisodeN**
The number of testing episodes.

template <**class** ActionC, **class** StateC>

Core<ActionC, StateC> ReLe::**buildCore**(*Environment*<ActionC, StateC> &*environment*, *Agent*<ActionC, StateC> &*agent*)

This function can be used to get a core instance from an agent and an environment, reducing boilerplate code: Example:

```
auto&& core = buildCore(environment, agent);
core.run();
```

template <class *ActionC*, class *StateC*>
class ReLe::**BatchOnlyCore**

This class can be used to run a batch agent over a dataset. This class handles batch agent termination flag and logging. The maximum number f iterations and the logging strategy can be specified in the settings.

Public Functions

BatchOnlyCore(*EnvironmentSettings* *task*, *Dataset*<ActionC, StateC> *data*, *BatchAgent*<ActionC, StateC> &*batchAgent*)

Constructor.

Parameters

- data: the dataset used for batch learning
- batchAgent: a batch learning agent

BatchOnlyCoreSettings &**getSettings**()

Getter.

Return the settings of the core.

void **run**()

Run the batch iterations over the dataset specified in the settings.

Parameters

- envSettings: settings of the environment

struct BatchOnlyCoreSettings

This struct contains the core settings

Public Members

BatchAgentLogger<ActionC, StateC> ***logger**

The logger for agent data.

unsigned int **maxBatchIterations**

The maximum number of iteration of the algorithm over the dataset.

template <**class** ActionC, **class** StateC>
BatchOnlyCore<ActionC, StateC> ReLe::**buildBatchOnlyCore**(*EnvironmentSettings* *task*, *Dataset*<ActionC, StateC> *data*, *BatchAgent*<ActionC, StateC> &*batchAgent*)

This function can be used to get a *BatchOnlyCore* instance from an agent and an environment, reducing boilerplate code: Example:

```
auto&& batchOnlyCore = buildBatchOnlyCore(environment, agent);
core.run();
```

template <class *ActionC*, class *StateC*>

class ReLe::**BatchCore**

This class is an extension of *BatchOnlyCore*, that takes care not only to run the batch algorithm, but also to generate the dataset and test the learned policy over the environment.

Public Functions

BatchCore(*Environment*<ActionC, StateC> &*environment*, *BatchAgent*<ActionC, StateC> &*batchAgent*)

Constructor.

Parameters

- environment: the environment used by this experiment
- batchAgent: the batch learning agent

BatchCoreSettings &**getSettings**()

Getter.

Return the core settings.

void **run**(*Policy*<ActionC, StateC> &*policy*)

This method is used to generate a dataset from the environment and run the batch learning algorithm on it.

void **run**(unsigned int *iterations*)

This method is used to iteratively generate a dataset from the environment using agent policy and then running the batch learning algorithm on it.

Dataset<ActionC, StateC> **runTest**()

This method generates a dataset using the agent learned policy

Return the generated dataset

struct BatchCoreSettings

This struct contains the core settings

Public Members

BatchDatasetLogger<ActionC, StateC> ***datasetLogger**

The logger for the dataset.

BatchAgentLogger<ActionC, StateC> ***agentLogger**

The logger for agent data.

unsigned int **episodeLength**

The episode lenght.

unsigned int **nEpisodes**

The number of episodes to run.

unsigned int **maxBatchIterations**

The maximum number of algorithm iterations.

template <**class** ActionC, **class** StateC>
BatchCore<ActionC, StateC> ReLe::**buildBatchCore**(*Environment*<ActionC, StateC> &*mdp*, *BatchAgent*<ActionC, StateC> &*batchAgent*)

This function can be used to get a *BatchCore* instance from an agent and an environment, reducing boilerplate code: Example:

```
    auto&& batchCore = buildBatchCore(environment, agent);
    core.run();
```

template <class *ActionC*, class *StateC*>
class ReLe::**Solver**

A solver is the abstract interface for exact or approximate solvers for a specific family of environments. Differently from agents, it does not implement a logger interface, and doesn't need to interface with a core. Often a solver is declared as friend of the class of MDPs that can solve, to access it's internal state.

Subclassed by ReLe::IRLSolver< ActionC, StateC >, ReLe::IRLSolver< ActionC, StateC, FeaturesInputC >

Public Functions

Solver()
Default Constructor

virtual void **solve**() = 0
This method run the computation of the optimal policy for the MDP. Must be implemented.

virtual *Dataset*<ActionC, StateC> **test**() = 0
This method runs the policy computed by *Solver::solve()* over the MDP. Must be implemented. Normally the implementation is simply wrapper for *Solver::test(Environment<ActionC, StateC>& env, Policy<ActionC, StateC>& pi)*

Return the set of trajectories sampled with the optimal policy from the MDP

virtual *Policy*<ActionC, StateC> &**getPolicy**() = 0
Getter

Return the set the optimal policy from the MDP

void **setTestParams**(unsigned int *testEpisodes*, unsigned int *testEpisodeLength*)
Set the episodes number and the episode maximum length

See *Solver::test()*

Parameters

- testEpisodes: the number of test episodes to run
- testEpisodeLength: the maximum length for each test episode

virtual **~Solver**()
Destructor

Protected Functions

virtual *Dataset*<ActionC, StateC> **test**(*Environment*<ActionC, StateC> &*env*, *Policy*<ActionC, StateC> &*pi*)
This method implements the low level test using the core.

Basic Environments

class ReLe::**FiniteMDP**

This class implements a finite MDP, that is an MDP with both finite actions and states. A finite MDP can be described by the tuple $MDP = \langle \mathcal{X}, \mathcal{U}, \mathcal{P}_u(.,..), \mathcal{R}_u(.,.), \gamma \rangle$ where $\mathcal{X}$ is the set of states, $\mathcal{U}$ is the set of actions, $\mathcal{P}_u : \mathcal{U} \times \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is the transition function, $\mathcal{R}_u : \mathcal{U} \times \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is the reward function. This class also

support gaussian reward functions, so it's possible to specify the variance of the reward. A finite MDP can also be solved exactly (or approximately) by a dynamic programming solver.

Inherits from *ReLe::Environment< FiniteAction, FiniteState >*

Public Functions

FiniteMDP(arma::cube *P*, arma::cube *R*, arma::cube *Rsigma*, bool *isFiniteHorizon*, double *gamma* = 1.0, unsigned int *horizon* = 0)

Constructor

Parameters

- `P`: the transition function, a $\mathcal{U} \times \mathcal{X} \times \mathcal{X}$ cube
- `R`: the reward function a $\mathcal{U} \times \mathcal{X} \times \mathcal{X}$ cube
- `Rsigma`: the reward function variance in each state a $\mathcal{U} \times \mathcal{X} \times \mathcal{X}$ cube
- `isFiniteHorizon`: if the mdp has a finite horizon
- `gamma`: the discount factor
- `horizon`: the mdp horizon

FiniteMDP(arma::cube *P*, arma::cube *R*, arma::cube *Rsigma*, *EnvironmentSettings* **settings*)

Constructor

Parameters

- `P`: the transition function, a $\mathcal{U} \times \mathcal{X} \times \mathcal{X}$ cube
- `R`: the reward function a $\mathcal{U} \times \mathcal{X} \times \mathcal{X}$ cube
- `Rsigma`: the reward function variance in each state a $\mathcal{U} \times \mathcal{X} \times \mathcal{X}$ cube
- `settings`: the environment settings

virtual void **step**(**const** *FiniteAction* &*action*, *FiniteState* &*nextState*, Reward &*reward*) **See** *Environment::step*

virtual void **getInitialState**(*FiniteState* &*state*) **See** *Environment::getInitialState*

class `ReLe::DenseMDP`

This class is the abstract interface for all MDPs that have continuous state space and finite actions.

Inherits from *ReLe::Environment< FiniteAction, DenseState >*

Subclassed by ReLe::Acrobot, ReLe::CarOnHill, *ReLe::DeepSeaTreasure*, *ReLe::DiscreteActionSwingUp*, *ReLe::MountainCar*, *ReLe::MultiHeat*, *ReLe::Portfolio*, *ReLe::TaxiFuel*, *ReLe::UnderwaterVehicle*

Public Functions

DenseMDP(*EnvironmentSettings* **settings*)

Constructor.

Parameters

- `settings`: the pointer to the environment settings

DenseMDP(std::size_t *stateSize*, unsigned int *actionN*, size_t *rewardSize*, bool *isFiniteHorizon*, bool *isEpisodic*, double *gamma* = 1.0, unsigned int *horizon* = 0)
Constructor.

Parameters

- `stateSize`: the dimensionality of the state space
- `actionN`: the number of actions
- `rewardSize`: the dimensionality of the reward function
- `isFiniteHorizon`: if the MDP has finite horizon
- `isEpisodic`: if the task is episodic
- `gamma`: the MDP discount factor
- `horizon`: the MDP horizon

virtual ~DenseMDP()
Destructor.

class `ReLe::ContinuousMDP`
This class is the abstract interface for all MDPs that have continuous state and action spaces.

Inherits from *ReLe::Environment< DenseAction, DenseState >*

Subclassed by *ReLe::Dam*, *ReLe::GaussianRewardMDP*, *ReLe::LQR*, *ReLe::NLS*, *ReLe::Pursuer*, *ReLe::Rocky*, *ReLe::Segway*, *ReLe::ShipSteering*, *ReLe::UnicyclePolar*, ReLe::WaterResources

Public Functions

ContinuousMDP()
Defualt Constructor.

ContinuousMDP(*EnvironmentSettings* **settings*)
Constructor.

Parameters

- `settings`: the pointer to the environment settings

ContinuousMDP(std::size_t *stateSize*, std::size_t *actionSize*, std::size_t *rewardSize*, bool *isFiniteHorizon*, bool *isEpisodic*, double *gamma* = 1.0, unsigned int *horizon* = 0)
Constructor.

Parameters

- `stateSize`: the dimensionality of the state space
- `actionSize`: the dimensionality of the action space
- `rewardSize`: the dimensionality of the reward function
- `isFiniteHorizon`: if the MDP has finite horizon
- `isEpisodic`: if the task is episodic
- `gamma`: the MDP discount factor
- `horizon`: the MDP horizon

Core loggers

template <class *ActionC*, class *StateC*>
class `ReLe::Logger`

This class implement the logger for the *ReLe::Core* class. this class is used to log both agent output data and environment trajectories into a dataset. The behavior of this class depends on the logger strategy selected. If no strategy is specified (or if the strategy is set to a null pointer) no logging will be performed.

Public Functions

Logger()
Constructor.

void **log**(StateC &*x*)
Method used to log the initial state

Parameters

- `x`: the initial environment state

void **log**(ActionC &*u*, StateC &*xn*, Reward &*r*)
Method used to log a state transition

Parameters

- `u`: the action performed by the agent
- `xn`: the state reached after the agent's action
- `r`: the reward achieved by the agent

void **log**(*AgentOutputData* **data*, unsigned int *step*)
Method used to log agent output data

Parameters

- `data`: the agent output data
- `step`: the current agent step

void **printStatistics**()
This method performs the logger operations

void **setStrategy**(*LoggerStrategy*<ActionC, StateC> **strategy*)
Setter.

Parameters

- `strategy`: the selected logger strategy to be performed

template <class *ActionC*, class *StateC*>
class `ReLe::LoggerStrategy`

This class implements the basic logger strategy interface. All logger strategies should implement this interface.

Subclassed by *ReLe::CollectorStrategy< ActionC, StateC >*, *ReLe::EvaluateStrategy< ActionC, StateC >*, *ReLe::PrintStrategy< ActionC, StateC >*, *ReLe::WriteStrategy< ActionC, StateC >*

Public Functions

virtual void **processData**(*Episode*<ActionC, StateC> &*samples*) = 0
This method describes how an episode should be processed. Must be implemented.

virtual void **processData**(std::vector<*AgentOutputData* *> &*data*) = 0
This method describes how the agent data should be processed. Must be implemented.

virtual ~LoggerStrategy()
Destructor.

Protected Functions

void **cleanAgentOutputData**(std::vector<*AgentOutputData* *> &*data*)
This method can be used to clean the agent data vector

template <**class** ActionC, **class** StateC>
class ReLe::**PrintStrategy**
This strategy can be used to print information to the console

Inherits from *ReLe::LoggerStrategy< ActionC, StateC >*

Public Functions

PrintStrategy(bool *logTransitions* = true, bool *logAgent* = true)
Constructor.

Parameters

- `logTransitions`: if the environment transitions should be printed on the console
- `logAgent`: if agent output data should be printed on the console

void **processData**(*Episode*<ActionC, StateC> &*samples*) **See** *LoggerStrategy::processData(Episode<ActionC, StateC>& samples)*

void **processData**(std::vector<*AgentOutputData* *> &*outputData*) **See** *LoggerStrategy::processData(std::vector<AgentOutputData*>& outputData)*

template <**class** ActionC, **class** StateC>
class ReLe::**WriteStrategy**
This strategy can be used to save logged informations to a file.

Inherits from *ReLe::LoggerStrategy< ActionC, StateC >*

Public Types

enum outType
enum used to select wheather to log only transitions, only agent data or both.

Values:

TRANS

AGENT

ALL

Public Functions

WriteStrategy(**const** std::string &*path*, *outType* *outputType* = ALL, bool *clean* = false)
Constructor

Parameters

- `path`: the path where to log the data
- `outputType`: what information should be logged
- `clean`: if the existing files should be overwritten up or not

`WriteStrategy` (**const** std::string &*transitionPath*, **const** std::string &*agentDataPath*)
Constructor

Parameters

- `transitionPath`: where the transitions will be logged
- `agentDataPath`: where the agent output data will be logged

void **`processData`** (*Episode*<ActionC, StateC> &*samples*) **See** *LoggerStrategy::processData(Episode<ActionC, StateC>& samples)*

void **`processData`** (std::vector<*AgentOutputData* *> &*outputData*) **See** *LoggerStrategy::processData(std::vector<AgentOutputData*>& outputData)*

template <**class** ActionC, **class** StateC>
class `ReLe::EvaluateStrategy`
This strategy can be used to evaluate the performances of an aget w.r.t. an environment

Inherits from *ReLe::LoggerStrategy< ActionC, StateC >*

Public Functions

`EvaluateStrategy` (double *gamma*)
Constructor

Parameters

- `gamma`: the discount factor for this environment

void **`processData`** (*Episode*<ActionC, StateC> &*samples*) **See** *LoggerStrategy::processData(Episode<ActionC, StateC>& samples)*

void **`processData`** (std::vector<*AgentOutputData* *> &*outputData*) **See** *LoggerStrategy::processData(std::vector<AgentOutputData*>& outputData)*

Public Members

std::vector<arma::vec> **`Jvec`**
A vector containing the returns of all episodes.

template <class *ActionC*, class *StateC*>
class `ReLe::CollectorStrategy`
This class simply collects the trajectories of all episodes into a *ReLe::Dataset*

Inherits from *ReLe::LoggerStrategy< ActionC, StateC >*

Public Functions

virtual void **`processData`** (*Episode*<ActionC, StateC> &*samples*) **See** *LoggerStrategy::processData(Episode<ActionC, StateC>& samples)*

virtual void **processData**(std::vector<*AgentOutputData* *> &*data*) **See** *LoggerStrategy::processData(std::vector<AgentOutputData*>& outputData)*

virtual ~CollectorStrategy()
Destructor.

Public Members

Dataset<ActionC, StateC> **data**
the collected trajectories

Batch loggers

template <class *ActionC*, class *StateC*>
class ReLe::**BatchAgentLogger**
This class is the default interface for *ReLe::BatchAgent* data logger. All batch agent loggers must extend this class.

Subclassed by *ReLe::BatchAgentPrintLogger< ActionC, StateC >*

Public Functions

void **log**(*AgentOutputData* **outputData*, unsigned int *step*)
This function is called automatically from *ReLe::BatchCore* for logging agent data.

Parameters

- `outputData`: the agent output data
- `step`: the current batch training step

virtual ~BatchAgentLogger()
Destructor.

Protected Functions

virtual void **processData**(*AgentOutputData* **outputData*) = 0
Abstract function called by the default log implementation. Should be overridden. This function implements the logging operations.

template <**class** ActionC, **class** StateC>
class ReLe::**BatchAgentPrintLogger**
This logger prints agent information to the console, calling *AgentOutputData::writeDecoratedData*.

Inherits from *ReLe::BatchAgentLogger< ActionC, StateC >*

template <class *ActionC*, class *StateC*>
class ReLe::**BatchDatasetLogger**
This class is the default interface for logging a dataset generated by *ReLe::BatchCore*. Implement this interface to add different dataset logging capabilities.

Subclassed by *ReLe::CollectBatchDatasetLogger< ActionC, StateC >*, *ReLe::WriteBatchDatasetLogger< ActionC, StateC >*

Public Functions

virtual ~BatchDatasetLogger()
Destructor.

virtual void **log**(*Dataset*<ActionC, StateC> &*data*) = 0
This function is called automatically from *ReLe::BatchCore* for logging the dataset. Should be overridden.

Parameters

- `data`: the dataset to be logged.

template <**class** ActionC, **class** StateC>
class `ReLe::CollectBatchDatasetLogger`
This logger simply stores the dataset into a variable, to be used later in the code

Inherits from *ReLe::BatchDatasetLogger< ActionC, StateC >*

Public Functions

void **log**(*Dataset*<ActionC, StateC> &*data*) **See** *BatchDatasetLogger::log*

Public Members

Dataset<ActionC, StateC> **data**
the logged dataset

template <**class** ActionC, **class** StateC>
class `ReLe::WriteBatchDatasetLogger`
This logger writes the dataset into a file.

Inherits from *ReLe::BatchDatasetLogger< ActionC, StateC >*

Public Functions

WriteBatchDatasetLogger(std::string *fileName*)
Constructor.

Parameters

- `fileName`: the absolute path where the dataset will be stored

void **log**(*Dataset*<ActionC, StateC> &*data*) **See** *BatchDatasetLogger::log*

Reward Transformation

class `ReLe::RewardTransformation`
Basic interface for reward function scalarization.

Subclassed by *ReLe::IndexRT*, *ReLe::WeightedSumRT*

Public Functions

virtual double **operator()** (**const** Reward &*r*) = 0
Evaluation method

Return the scalarization of the reward vector

Parameters

- r: the reward vector

class ReLe::**IndexRT**
This class implement the single objective scalarization, i.e. the scalarization is done by selecting a single value of the reward function.

Inherits from *ReLe::RewardTransformation*

Public Functions

IndexRT (unsigned int *idx*)
Constructor.

Parameters

- idx: the index of the reward element to choose

virtual double **operator()** (**const** Reward &*r*)
Evaluation method

Return the scalarization of the reward vector

Parameters

- r: the reward vector

class ReLe::**WeightedSumRT**
This class implement the weighted sum scalarization, i.e. the scalarization is a weighted sum of all elements in the reward vector.

Inherits from *ReLe::RewardTransformation*

Public Functions

virtual double **operator()** (**const** Reward &*r*)
Evaluation method

Return the scalarization of the reward vector

Parameters

- r: the reward vector

Basic Utilities

template <class *ActionC*, class *StateC*>
class ReLe::**PolicyEvalAgent**
This class implements a fake agent, that cannot be used for learning but only for evaluating a policy over an MDP. This class implements the agent interface, however the methods that implement learning should never be called.

Inherits from *ReLe::Agent< ActionC, StateC >*

Subclassed by *ReLe::PolicyEvalDistribution< ActionC, StateC >*

Public Functions

PolicyEvalAgent(*Policy*<ActionC, StateC> &*policy*)
Constructor

Parameters

- `policy`: the policy to be used in evaluation

virtual void **initTestEpisode**()
This method is called at the beginning of each test episode. by default does nothing, but can be overloaded

virtual void **initEpisode**(**const** StateC &*state*, ActionC &*action*)
This method should never be called. Throws an exception if used, as policy eval agent cannot be used for learning.

void **sampleAction**(**const** StateC &*state*, ActionC &*action*)
This method is used to sample an action in test episodes. Must be implemented. Normally, this method is trivial, as it just sample an action from a policy.

Parameters

- `state`: the current environment state
- `action`: the action selected by the agent in the current state

void **step**(**const** Reward &*reward*, **const** StateC &*nextState*, ActionC &*action*)
This method should never be called. Throws an exception if used, as policy eval agent cannot be used for learning.

void **endEpisode**(**const** Reward &*reward*)
This method should never be called. Throws an exception if used, as policy eval agent cannot be used for learning.

void **endEpisode**()
This method should never be called. Throws an exception if used, as policy eval agent cannot be used for learning.

template <**class** ActionC, **class** StateC>
class `ReLe::`**PolicyEvalDistribution**
This class implements a fake agent, that cannot be used for learning but only for evaluating a distribution of parametric policies over an MDP.

Inherits from *ReLe::PolicyEvalAgent< ActionC, StateC >*

Public Functions

PolicyEvalDistribution(*Distribution* &*dist*, *ParametricPolicy*<ActionC, StateC> &*policy*, unsigned int *episodesPerPolicy* = 1)
Constructor

Parameters

- `dist`: the distribution of the parameters of the policies
- `policy`: the family of parametric policies to be used

virtual void **initTestEpisode**()
This method is called at the beginning of each test episode. by default does nothing, but can be overloaded

arma::mat **getParams**()
This method can be used to return the parameters used during the test runs

Return a matrix paramsN × episodeN

Function Approximators

Basic interfaces

template <class *InputC*>
class ReLe::**BasisFunction_**
This interface represents a basis function. A basis function can be see as a mapping $\psi(i) \rightarrow \mathbb{R}$ with $i \in \mathcal{D}$. The template definition allows for generic domains $\mathcal{D}$. A set of basis function can be used as a set of features, that can be used for function approximation.

Subclassed by *ReLe::AffineFunction*, *ReLe::IdentityBasis_< InputC >*, *ReLe::InfiniteNorm*, *ReLe::InverseBasis_< InputC >*, *ReLe::NormBasis*, *ReLe::SubspaceBasis*

Public Functions

virtual double **operator()**(**const** InputC &*input*) = 0
Evaluates the basis function in input.

Return the value of the basis function at the input

Parameters

- input: the input data

virtual void **writeOnStream**(std::ostream &*out*) = 0
Writes the basis function to stream.

virtual void **readFromStream**(std::istream &*in*) = 0
Reads the basis function from stream.

virtual ~BasisFunction_()
Destructor.

Friends

std::ostream &**operator<<**(std::ostream &*out*, BasisFunction_<InputC> &*bf*)
Writes the basis function to stream.

std::istream &**operator>>**(std::istream &*in*, BasisFunction_<InputC> &*bf*)
Reads the basis function from stream.

template <class *InputC*>
class ReLe::**Tiles_**
This class is a common interface for tiles. Tiles are a way to divide an input space in multiple portions. In other words, tiles implements a discretization over the input space. Formally this class implements a mapping from an input data to a positive number, which represent the corresponding tile. One or a set of tilings can be used as a set of features for approximators.

Subclassed by *ReLe::BasicTiles*

Public Functions

virtual unsigned int **operator()**(**const** InputC &*input*) = 0
Find the index of the corresponding input tile.

virtual unsigned int **size**() = 0
Getter.

Return the total number of tiles

virtual void **writeOnStream**(std::ostream &*out*) = 0
Writes the basis function to stream

virtual void **readFromStream**(std::istream &*in*) = 0
Read the basis function from stream

virtual ~Tiles_()
Destructor.

Friends

std::ostream &**operator<<**(std::ostream &*out*, Tiles_<InputC> &*bf*)
Writes the basis function to stream

std::istream &**operator>>**(std::istream &*in*, Tiles_<InputC> &*bf*)
Read the basis function from stream

template <class *InputC*, bool *denseOutput* = true>
class ReLe::**Features_**
This interface represent a generic set of features. Features are a mapping $\phi(i) \rightarrow \mathbb{R}^n \times \mathbb{R}^m$ with $i \in \mathcal{D}$. The template definition allows for generic domains $\mathcal{D}$. Optionally, the features matrix returned can be a sparse matrix, this is done by setting to false the optional template parameter denseOutput.

Subclassed by *ReLe::TilesCoder_< InputC, denseOutput >*

Public Functions

virtual ~Features_()
Destructor.

virtual return_type **operator()**(**const** InputC &*input*) **const** = 0
Evaluate the features in input

Return the evaluated features

Parameters

- `input`: the input data

template <**class** Input1, **class** Input2>
return_type **operator()**(**const** Input1 &*input1*, **const** Input2 &*input2*) **const**
Overloading of the evaluation method, simply vectorizes the inputs and then evaluates the feature with the default 1 input method

Return the evaluated features

Parameters

- `input1`: the first input data

- `input2`: the second input data

template <**class** Input1, **class** Input2, **class** Input3>
return_type **`operator()`**(**const** Input1 &*input1*, **const** Input2 &*input2*, **const** Input3 &*input3*) **const**
Overloading of the evaluation method, simply vectorizes the inputs and then evaluates the feature with the default 1 input method

Return the evaluated features

Parameters

- `input1`: the first input data
- `input2`: the second input data
- `input3`: the third input data

virtual size_t **`rows`**() **const** = 0
Getter.

Return the number of rows of the output feature matrix

virtual size_t **`cols`**() **const** = 0
Getter.

Return the number of columns of the output feature matrix

template <class *InputC*, class *OutputC*, bool *denseOutput* = true>
class `ReLe::Regressor_`
This is the default interface for function approximators. We suppose that function approximation works applying an arbitrary function over a set of features of the input data. Formally a regressor is a function $f(i) \rightarrow \mathcal{D}_{output}^{n}$ with $i \in \mathcal{D}_{input}$ Function approximation can work with any type of input data and features over data (dense, sparse).

Subclassed by *ReLe::BatchRegressor_< InputC, OutputC, true >*, *ReLe::BatchRegressor_< InputC, OutputC, denseOutput >*, *ReLe::UnsupervisedBatchRegressor_< InputC, OutputC, denseOutput >*

Public Types

typedef InputC **`InputType`**
type of input of the regressor

typedef OutputC **`OutputType`**
type of the output of the regressor

Public Functions

`Regressor_`(*Features_*<InputC, denseOutput> &*phi*, unsigned int *output* = 1)
Constructor.

Parameters

- `phi`: the features used by the approximator
- `output`: the dimensionality of the output vector

virtual OutputC **`operator()`**(**const** InputC &*input*) = 0
Evaluates the function at the input.

Return the value of the function at input

Parameters

- `input`: the input data

template <**class** Input1, **class** Input2>
OutputC **operator()**(**const** Input1 &*input1*, **const** Input2 &*input2*)
Overloading of the evaluation method, simply vectorizes the inputs and then evaluates the regressor with the default 1 input method

Return the value of the function

Parameters

- `input1`: the first input data
- `input2`: the second input data

template <**class** Input1, **class** Input2, **class** Input3>
OutputC **operator()**(**const** Input1 &*input1*, **const** Input2 &*input2*, **const** Input3 &*input3*)
Overloading of the evaluation method, simply vectorizes the inputs and then evaluates the regressor with the default 1 input method

Return the value of the function

Parameters

- `input1`: the first input data
- `input2`: the second input data
- `input3`: the third input data

int **getOutputSize**()
Getter.

Return the dimensionality of the output.

*Features_*<InputC, denseOutput> &**getFeatures**()
Return the features used by the regressor.

const *Features_*<InputC, denseOutput> &**getFeatures**() **const**
Return the features used by the regressor.

virtual **~Regressor_**()
Destructor.

Public Static Attributes

const bool **isDense** = denseOutput
whether the features are dense or sparse

template <class *InputC*, class *OutputC*, bool *denseOutput* = true>
class `ReLe::Ensemble_`
Inherits from *ReLe::BatchRegressor_< InputC, OutputC, denseOutput >*

Subclassed by *ReLe::DoubleFQIEnsemble*

Public Functions

virtual OutputC **operator()**(**const** InputC &*input*)
Evaluates the function at the input.

Return the value of the function at input

Parameters

- `input`: the input data

virtual void **`trainFeatures`**(**const** *BatchData_*<OutputC, denseOutput> &*dataset*)
This method implements the low level training of a dataset from a set of already computed features.

Parameters

- `dataset`: the set of computed features.

virtual double **`computeJFeatures`**(**const** *BatchData_*<OutputC, denseOutput> &*dataset*)
This method is used to compute the performance of the features dataset w.r.t. the current learned regressor.

Return the value of the objective function

Parameters

- `dataset`: the features dataset

template <class *InputC*, bool *denseOutput* = true>
class `ReLe::ParametricRegressor_`
This is the default interface for function approximators that can be described through parameters. This interface assumes that the parametrization should be differentiable.

Inherits from *ReLe::Regressor_< InputC, arma::vec, denseOutput >*

Subclassed by ReLe::LinearApproximator_< InputC, denseFeatures >, ReLe::LinearApproximator_< InputC, true >, ReLe::FFNeuralNetwork_< InputC, denseOutput >, ReLe::LinearApproximator_< InputC, denseOutput >

Public Functions

`ParametricRegressor_`(*Features_*<InputC, denseOutput> &*phi*, unsigned int *output* = 1)
Constructor.

Parameters

- `phi`: the features used by the approximator
- `output`: the dimensionality of the output vector

virtual void **`setParameters`**(**const** arma::vec &*params*) = 0
Setter.

Parameters

- `params`: the parameters vector

virtual arma::vec **`getParameters`**() **const** = 0
Getter.

Return the parameters vector

virtual unsigned int **`getParametersSize`**() **const** = 0
Getter.

Return the length of parameters vector

virtual arma::vec **`diff`**(**const** InputC &*input*) = 0
Compute the derivative of the represented function w.r.t. the parameters, at input.

template <**class** Input1, **class** Input2>

arma::vec **diff**(**const** Input1 *&input1*, **const** Input2 *&input2*)
Overloading of the differentiation method, simply vectorizes the inputs and then evaluates the derivative with the default 1 input method

Return the value of the derivative

Parameters

- `input1`: the first input data
- `input2`: the second input data

template <**class** Input1, **class** Input2, **class** Input3>
arma::vec **diff**(**const** Input1 *&input1*, **const** Input2 *&input2*, **const** Input3 *&input3*)
Overloading of the differentiation method, simply vectorizes the inputs and then evaluates the derivative with the default 1 input method

Return the value of the derivative

Parameters

- `input1`: the first input data
- `input2`: the second input data
- `input3`: the third input data

virtual **~ParametricRegressor_**()
Destructor.

template <class *InputC*, class *OutputC*, bool *denseOutput* = true>
class `ReLe::BatchRegressor_`
This is the default interface for regressors that uses supervised learning for learning the target function. It contains functions to train the regressor from raw data or from already computed features, and performance evaluation methods as well.

Inherits from *ReLe::Regressor_< InputC, OutputC, denseOutput >*

Subclassed by *ReLe::Ensemble_< InputC, OutputC, denseOutput >*, ReLe::RegressionTree< InputC, OutputC, denseOutput >

Public Functions

BatchRegressor_(*Features_*<InputC, denseOutput> *&phi*, unsigned int *output* = 1)
Constructor.

Parameters

- `phi`: the features used by the approximator
- `output`: the dimensionality of the output vector

virtual void **train**(**const** *BatchDataRaw_*<InputC, OutputC> *&dataset*)
This method is used to train raw input data

Parameters

- `dataset`: the input dataset

double **computeJ**(**const** *BatchDataRaw_*<InputC, arma::vec> *&dataset*)
This method is used to compute the performance of an input dataset w.r.t. the current learned regressor.

Return the value of the objective function

Parameters

- `dataset`: the input dataset

virtual void **`trainFeatures`**(**const** *BatchData_*<OutputC, denseOutput> &*dataset*) = 0
This method implements the low level training of a dataset from a set of already computed features.

Parameters

- `dataset`: the set of computed features.

virtual double **`computeJFeatures`**(**const** *BatchData_*<OutputC, denseOutput> &*dataset*) = 0
This method is used to compute the performance of the features dataset w.r.t. the current learned regressor.

Return the value of the objective function

Parameters

- `dataset`: the features dataset

virtual ~`BatchRegressor_`()
Destructor.

template <class *InputC*, class *OutputC*, bool *denseOutput* = true>
class `ReLe::UnsupervisedBatchRegressor_`
This is the default interface for regressors that uses unsupervised learning for learning the target function. It contains functions to train the regressor from raw data or from already computed feature.

Inherits from *ReLe::Regressor_< InputC, OutputC, denseOutput >*

Public Functions

`UnsupervisedBatchRegressor_`(*Features_*<InputC, denseOutput> &*phi*, unsigned int *output* = 1)
Constructor.

Parameters

- `phi`: the features used by the approximator
- `output`: the dimensionality of the output vector

virtual void **`train`**(**const** std::vector<InputC> &*dataset*)
This method is used to compute the performance of an input dataset w.r.t. the current learned regressor.

Return the value of the objective function

Parameters

- `dataset`: the input dataset

virtual void **`trainFeatures`**(**const** FeaturesCollection &*features*) = 0
This method implements the low level training of a dataset from a set of already computed features.

Parameters

- `dataset`: the set of computed features.

virtual ~`UnsupervisedBatchRegressor_`()
Destructor.

Basis functions

class `ReLe::AffineFunction`
This class implements affine transformation functions for an input vector.

Inherits from *ReLe::BasisFunction_< InputC >*

Public Functions

`AffineFunction`(BasisFunction **bfs*, arma::mat *A*)
Constructor.

Parameters

- `bfs`: basis function
- `A`: the matrix for the affine transformation

void **`writeOnStream`**(std::ostream &*out*)
Writes the basis function to stream.

void **`readFromStream`**(std::istream &*in*)
Reads the basis function from stream.

Public Static Functions

static BasisFunctions **`generate`**(BasisFunctions &*basis*, arma::mat &*A*)
Return the basis functions for the affine transformation.

Return the generated basis functions

Parameters

- `basis`: basis function
- `A`: the matrix for the affine transformation

class `ReLe::GaussianRbf`
This class implements Gaussian Radial Basis Functions.

Inherits from *ReLe::BasisFunction_< arma::vec >*

Public Functions

`GaussianRbf`(double *center*, double *width*, bool *useSquareRoot* = false)
Constructor.

Parameters

- `center`: the center of the Gaussian radial function
- `width`: the width of the Gaussian radial function
- `useSquareRoot`: specify whether to use square root of the exponential term of the Gaussian radial function

`GaussianRbf`(arma::vec *center*, double *width*, bool *useSquareRoot* = false)
Constructor.

Parameters

- `center`: vector of centers of the Gaussian radial function
- `width`: the width of the Gaussian radial function
- `useSquareRoot`: specify whether to use square root of the exponential term of the Gaussian radial function

GaussianRbf(arma::vec *center*, arma::vec *width*, bool *useSquareRoot* = false)
Constructor.

Parameters

- `center`: vector of centers of the Gaussian radial function
- `width`: vector of widths of the Gaussian radial function
- `useSquareRoot`: specify whether to use square root of the exponential term of the Gaussian radial function

virtual ~GaussianRbf()
Destructor.

double **operator()**(**const** arma::vec &*input*)
Evaluates the basis function in input.

Return the value of the basis function at the input

Parameters

- `input`: the input data

arma::vec &**getCenter**()
Getter.

Return center of the Gaussian radial function

arma::vec &**getWidth**()
Getter.

Return width of the Gaussian radial function

unsigned int **getSize**()
Getter.

Return the number of centers

virtual void **writeOnStream**(std::ostream &*out*)
Writes the basis function to stream.

virtual void **readFromStream**(std::istream &*in*)
Reads the basis function from stream.

Public Static Functions

static BasisFunctions **generate**(arma::vec &*numb_centers*, arma::mat &*range*)
Return the Gaussian Radial Basis Functions with given number of centers and intervals.

Return the generated basis functions

Parameters

- `numb_centers`: vector of number of centers
- `range`: interval matrix

static BasisFunctions **generate**(unsigned int *n_centers*, std::initializer_list<double> *l*)
Return the Gaussian Radial Basis Functions with given number of centers and intervals.

Return the generated basis functions

Parameters

- `n_centers`: number of centers
- `l`: list of intervals

static BasisFunctions **generate**(std::initializer_list<unsigned int> *n_centers*, std::initializer_list<double> *l*)
Return the Gaussian Radial Basis Functions with given number of centers and intervals.

Return the generated basis functions

Parameters

- `n_centers`: list of number of centers
- `l`: list of intervals

static BasisFunctions **generate**(arma::mat &*centers*, arma::mat &*widths*)
Return the Gaussian Radial Basis Functions with given centers and widths.

Return the generated basis functions

Parameters

- `centers`: vector of centers of the Gaussian radial functions
- `widths`: vector of widths of the Gaussian radial functions

class `ReLe::AndConditionBasisFunction`
This class implements functions to attach a target feature(s) to a group of given basis functions. This is done by repeating the group of basis functions a number of time equal to the number of possible values of the target feature(s) and multiplying all elements of each group by the respective value of the target feature(s) and setting the others to zero. In particular, this can be useful with finite action spaces using the possible values of the action as the target feature.

Inherits from *ReLe::BasisFunction_< arma::vec >*

Public Functions

AndConditionBasisFunction(BasisFunction **bfs*, **const** std::vector<unsigned int> &*idxs*, **const** std::vector<double> &*condition_vals*)
Constructor.

Parameters

- `bfs`: the basis functions to which the target features are attached
- `idxs`: vector of indexes of the target features
- `condition_vals`: vector of the number of possible values of the target features

AndConditionBasisFunction(BasisFunction **bfs*, std::initializer_list<unsigned int> *idxs*, std::initializer_list<double> *condition_vals*)
Constructor.

Parameters

- `bfs`: the basis functions to which the target features are attached
- `idxs`: initializer list of indexes of the target features

- `condition_vals`: initializer list of the number of possible values of the target features

AndConditionBasisFunction(BasisFunction **bfs*, unsigned int *idx*, double *condition_vals*)
Constructor.

Parameters

- `bfs`: the basis functions to which the target feature is attached
- `idx`: the index of the target feature
- `condition_vals`: the number of possible values of the target feature

double **operator()**(**const** arma::vec &*input*)
Evaluates the basis function in input.

Return the value of the basis function at the input

Parameters

- `input`: the input data

void **writeOnStream**(std::ostream &*out*)
Writes the basis function to stream.

void **readFromStream**(std::istream &*in*)
Reads the basis function from stream.

Public Static Functions

static BasisFunctions **generate**(BasisFunctions &*basis*, unsigned int *index*, unsigned int *value*)
Generate the basis functions.

Return the basis functions

Parameters

- `basis`: the basis functions to which the target feature is attached
- `index`: the index of the target feature
- `value`: the number of possible values of the target feature

static BasisFunctions **generate**(BasisFunctions &*basis*, std::vector<unsigned int> *indexes*, std::vector<unsigned int> *valuesVector*)
Generate the basis functions.

Return the basis functions

Parameters

- `basis`: the basis functions to which the target features are attached
- `indexes`: vector of indexes of the target features
- `valuesVector`: vector of the number of possible values of the target features

class `ReLe::SubspaceBasis`
This class implements functions to apply the given basis functions to a subset of the input.

Inherits from *ReLe::BasisFunction_< InputC >*

Public Functions

SubspaceBasis(BasisFunction **basis*, **const** arma::span &*span*)
Constructor.

Parameters

- `basis`: basis function
- `span`: subset selection of the input

SubspaceBasis(BasisFunction **basis*, std::vector<arma::span> &*spanVector*)
Constructor.

Parameters

- `basis`: basis function
- `spanVector`: vector of subsets selection of the input

~SubspaceBasis()
Destructor.

void **writeOnStream**(std::ostream &*out*)
Writes the basis function to stream.

void **readFromStream**(std::istream &*in*)
Reads the basis function from stream.

Public Static Functions

static BasisFunctions **generate**(BasisFunctions &*basisVector*, std::vector<arma::span> &*spanVector*)
Return the basis functions for the selected subsets.

Parameters

- `basisVector`: vector of basis functions
- `spanVector`: vector of subsets selection of the input

static BasisFunctions **generate**(BasisFunctions &*basisVector*, arma::span *span*)
Return the basis functions for the selected subset.

Parameters

- `basisVector`: vector of basis functions
- `span`: subset selection of the input

class `ReLe::ModularBasis`
This class is the interface to build basis functions that transform the input into the corresponding value in the given range.

Inherits from *ReLe::BasisFunction_< arma::vec >*

Subclassed by *ReLe::ModularDifference*, *ReLe::ModularDivision*, *ReLe::ModularProduct*, *ReLe::ModularSum*

Public Functions

ModularBasis(unsigned int *index1*, unsigned int *index2*, **const** *ModularRange* &*range*)
Constructor.

Parameters

- `index1`: index of the first element of the input
- `index2`: index of the second element of the input
- `range`: range to be used to transform the input

class `ReLe::ModularSum`

This class builds basis functions to transform the sum of the elements of the input at the given indexes into the corresponding value in the given range.

Inherits from *ReLe::ModularBasis*

Public Functions

ModularSum(unsigned int *index1*, unsigned int *index2*, **const** *ModularRange* &*range*)

Constructor.

Parameters

- `index1`: index of the first element of the input
- `index2`: index of the second element of the input
- `range`: range to be used to transform the input

double **operator()**(**const** arma::vec &*input*)

Evaluates the basis function in input.

Return the value of the basis function at the input

Parameters

- `input`: the input data

void **writeOnStream**(std::ostream &*out*)

Writes the basis function to stream.

void **readFromStream**(std::istream &*in*)

Reads the basis function from stream.

class `ReLe::ModularDifference`

This class builds basis functions to transform the difference of the elements of the input at the given indexes into the corresponding value in the given range.

Inherits from *ReLe::ModularBasis*

Public Functions

ModularDifference(unsigned int *index1*, unsigned int *index2*, **const** *ModularRange* &*range*)

Constructor.

Parameters

- `index1`: index of the first element of the input
- `index2`: index of the second element of the input
- `range`: range to be used to transform the input

double **operator()**(**const** arma::vec &*input*)

Evaluates the basis function in input.

Return the value of the basis function at the input

Parameters

- `input`: the input data

void **writeOnStream**(std::ostream &*out*)
Writes the basis function to stream.

void **readFromStream**(std::istream &*in*)
Reads the basis function from stream.

class `ReLe::ModularProduct`
This class builds basis functions to transform the product of the elements of the input at the given indexes into the corresponding value in the given range.

Inherits from *ReLe::ModularBasis*

Public Functions

ModularProduct(unsigned int *index1*, unsigned int *index2*, **const** *ModularRange* &*range*)
Constructor.

Parameters

- `index1`: index of the first element of the input
- `index2`: index of the second element of the input
- `range`: range to be used to transform the input

double **operator()**(**const** arma::vec &*input*)
Evaluates the basis function in input.

Return the value of the basis function at the input

Parameters

- `input`: the input data

void **writeOnStream**(std::ostream &*out*)
Writes the basis function to stream.

void **readFromStream**(std::istream &*in*)
Reads the basis function from stream.

class `ReLe::ModularDivision`
This class builds basis functions to transform the division of the elements of the input at the given indexes into the corresponding value in the given range.

Inherits from *ReLe::ModularBasis*

Public Functions

ModularDivision(unsigned int *index1*, unsigned int *index2*, **const** *ModularRange* &*range*)
Constructor.

Parameters

- `index1`: index of the first element of the input
- `index2`: index of the second element of the input

- `range`: range to be used to transform the input

double **operator()** (**const** arma::vec &*input*)
Evaluates the basis function in input.

Return the value of the basis function at the input

Parameters

- `input`: the input data

void **writeOnStream** (std::ostream &*out*)
Writes the basis function to stream.

void **readFromStream** (std::istream &*in*)
Reads the basis function from stream.

class `ReLe::QuadraticBasis`
This class implements functions to make basis functions in quadratic form.

Inherits from *ReLe::BasisFunction_< arma::vec >*

Public Functions

QuadraticBasis (arma::mat &*Q*, arma::span = arma::span::all)
Constructor.

Parameters

- `Q`: matrix of the quadratic form
- `span`: indexes of the elements of the input to consider

QuadraticBasis (std::vector<arma::mat> &*Q*)
Constructor.

Parameters

- `Q`: vector of matrices of the quadratic forms

QuadraticBasis (std::vector<arma::mat> &*Q*, std::vector<arma::span> *span*)
Constructor.

Parameters

- `Q`: vector of matrices of the quadratic forms
- `span`: vector of indexes of the elements of the input to consider

virtual double **operator()** (**const** arma::vec &*input*)
Evaluates the basis function in input.

Return the value of the basis function at the input

Parameters

- `input`: the input data

virtual void **writeOnStream** (std::ostream &*out*)
Writes the basis function to stream.

virtual void **readFromStream** (std::istream &*in*)
Reads the basis function from stream.

template <class *InputC*>

class ReLe::**IdentityBasis_**
This template class implements functions to build basis functions that replicates the input.

Inherits from *ReLe::BasisFunction_< InputC >*

Public Functions

IdentityBasis_(unsigned int *index*)
Constructor.

Parameters

- index: the index of the element in the input

class ReLe::**IdentityBasis**
This class implements functions to build basis functions that replicates the input vector.

Inherits from *ReLe::IdentityBasis_< arma::vec >*

Public Functions

IdentityBasis(unsigned int *index*)
Constructor.

Parameters

- index: the index of the element in the input

virtual ~IdentityBasis()
Destructor.

double **operator()**(**const** arma::vec &*input*)
Evaluates the basis function in input.

Return the value of the basis function at the input

Parameters

- input: the input data

virtual void **writeOnStream**(std::ostream &*out*)
Writes the basis function to stream.

virtual void **readFromStream**(std::istream &*in*)
Reads the basis function from stream.

Public Static Functions

static BasisFunctions **generate**(unsigned int *input_size*)
Return the input in the form of a Basis Functions.

Return the generated basis functions

Parameters

- input_size: the size of the input vector

class ReLe::**FiniteIdentityBasis**
This class implements functions to build basis functions that replicates the input vector with finite values.

Inherits from *ReLe::IdentityBasis_< size_t >*

Public Functions

FiniteIdentityBasis(unsigned int *index*)
Constructor.

Parameters

- `index`: the index of the element in the input

virtual ~FiniteIdentityBasis()
Destructor.

double **operator()**(**const** size_t &*input*)
Evaluates the basis function in input.

Return the value of the basis function at the input

Parameters

- `input`: the input data

virtual void **writeOnStream**(std::ostream &*out*)
Writes the basis function to stream.

virtual void **readFromStream**(std::istream &*in*)
Reads the basis function from stream.

Public Static Functions

static BasisFunctions_<size_t> **generate**(unsigned int *stateN*)
Return the value associated to the given input.

Return the generated basis functions

Parameters

- `stateN`: the input value

class `ReLe::VectorFiniteIdentityBasis`
This class implements functions to build basis functions that associate a value to each finite value in the input vector.

Inherits from *ReLe::IdentityBasis_< arma::vec >*

Public Functions

VectorFiniteIdentityBasis(unsigned int *index*, double *value*)
Constructor.

Parameters

- `index`: the index of the element in the input
- `value`: the value to associate at the input element in the given index

virtual ~VectorFiniteIdentityBasis()
Destructor.

double **operator()**(**const** arma::vec &*input*)
Evaluates the basis function in input.

Return the value of the basis function at the input

Parameters

- `input`: the input data

virtual void **writeOnStream**(std::ostream &*out*)
Writes the basis function to stream.

virtual void **readFromStream**(std::istream &*in*)
Reads the basis function from stream.

Public Static Functions

static BasisFunctions **generate**(std::vector<unsigned int> *values*)
Return the values associated to each input.

Return the generated basis functions

Parameters

- `values`: the vector of values

static BasisFunctions **generate**(unsigned int *stateN*, unsigned int *values*)
Return the value associated to the given input.

Return the generated basis functions

Parameters

- `stateN`: the input value
- `values`: the vector of values

template <**class** InputC>
class `ReLe::`**InverseBasis_**
This class implements functions to build basis functions that, given an input value, return its inverse value.

Inherits from *ReLe::BasisFunction_< InputC >*

Public Functions

InverseBasis_(*BasisFunction_*<InputC> **basis*)
Constructor.

Parameters

- `basis`: basis function

~InverseBasis_()
Destructor.

double **operator()**(**const** InputC &*input*)
Evaluates the basis function in input.

Return the value of the basis function at the input

Parameters

- `input`: the input data

void **writeOnStream**(std::ostream &*out*)
Writes the basis function to stream.

void **readFromStream**(std::istream &*in*)
Reads the basis function from stream.

Public Static Functions

static BasisFunctions_<InputC> **generate**(BasisFunctions_<InputC> *basis*)
Return the inverse value associated to the given input.

Return the generated basis functions

Parameters

- `basis`: the basis function whose output value is inverted

class `ReLe::`**NormBasis**
This class implements function to return the p-norm of an input feature.

Inherits from *ReLe::BasisFunction_< InputC >*

Public Functions

NormBasis(unsigned int *p* = 2)
Constructor.

Parameters

- `p`: the p of the p-norm

virtual void **writeOnStream**(std::ostream &*out*)
Writes the basis function to stream.

virtual void **readFromStream**(std::istream &*in*)
Reads the basis function from stream.

class `ReLe::`**InfiniteNorm**
This class implements function to return the infinite norm of an input feature.

Inherits from *ReLe::BasisFunction_< InputC >*

Public Functions

InfiniteNorm(bool *max* = true)
Constructor.

Parameters

- `max`: flag to indicate wheter to use inf or -inf for the p of the p-norm

virtual void **writeOnStream**(std::ostream &*out*)
Writes the basis function to stream.

virtual void **readFromStream**(std::istream &*in*)
Reads the basis function from stream.

class `ReLe::`**PolynomialFunction**
This class implements functions to build polynomial basis functions.

Inherits from *ReLe::BasisFunction_< arma::vec >*

Public Functions

PolynomialFunction()
Constructor.

PolynomialFunction(std::vector<unsigned int> *dimension*, std::vector<unsigned int> *degree*)
Constructor.

Parameters

- `dimension`: vector of dimensions of the input vector
- `degree`: vector of degrees of the polynomial

PolynomialFunction(unsigned int *_dimension*, unsigned int *_degree*)
Constructor.

Parameters

- `_dimension`: the dimension of the input vector
- `_degree`: the degree of the polynomial

virtual ~PolynomialFunction()
Destructor.

double **operator()**(**const** arma::vec &*input*)
Evaluates the basis function in input.

Return the value of the basis function at the input

Parameters

- `input`: the input data

virtual void **writeOnStream**(std::ostream &*out*)
Writes the basis function to stream.

virtual void **readFromStream**(std::istream &*in*)
Reads the basis function from stream.

Public Static Functions

static BasisFunctions **generate**(unsigned int *degree*, unsigned int *input_size*)
Generate the basis functions.

Return the basis functions

Parameters

- `degree`: the degree of the polynomial
- `input_size`: the dimension of the input vector

Tiles

class `ReLe::BasicTiles`
This class implements the most simple type of tiling: a uniform grid tiling over the whole state space. As only a finite number of tiles is supported, the state space must be range limited.

Inherits from *ReLe::Tiles_< InputC >*

Subclassed by *ReLe::CenteredLogTiles*, *ReLe::LogTiles*, *ReLe::SelectiveTiles*

Public Functions

BasicTiles(**const** *Range* &*range*, unsigned int *tilesN*)
Constructor.

Parameters

- `range`: the range of the first state variable
- `tilesN`: the number of tiles to use for the first state variable

BasicTiles(**const** std::vector<*Range*> &*ranges*, **const** std::vector<unsigned int> &*tilesN*)
Constructor.

Parameters

- `ranges`: the range of the (first n) state variables
- `tilesN`: the number of tiles to use for each (of the first n) state variable

virtual unsigned int **size**()
Getter.

Return the total number of tiles

virtual void **writeOnStream**(std::ostream &*out*)
Writes the basis function to stream

virtual void **readFromStream**(std::istream &*in*)
Read the basis function from stream

class `ReLe::SelectiveTiles`
This class extends the *BasicTiles* class, allowing to choose over which state component tiling should be applied.

Inherits from *ReLe::BasicTiles*

Public Functions

SelectiveTiles(**const** std::vector<unsigned int> *stateComponents*, **const** std::vector<*Range*> &*ranges*, **const** std::vector<unsigned int> &*tilesN*)
Constructor.

Parameters

- `stateComponents`: the index of the state variables to use
- `ranges`: the range to use for each state variable
- `tilesN`: the number of tiles to use for each state variable

virtual void **writeOnStream**(std::ostream &*out*)
Writes the basis function to stream

virtual void **readFromStream**(std::istream &*in*)
Read the basis function from stream

class `ReLe::LogTiles`
This class implements the logarithmic spaced tiling. This type of tiling is a uniform tiling when considering the following transformed input space:

$$\hat{input} = \log(input - min + 1)$$

As only a finite number of tiles is supported, the state space must be range limited.

Inherits from *ReLe::BasicTiles*

Public Functions

LogTiles(**const** *Range* &*range*, unsigned int *tilesN*)
Constructor.

Parameters

- `range`: the range of the first state variable
- `tilesN`: the number of tiles to use for the first state variable

LogTiles(**const** std::vector<*Range*> &*ranges*, **const** std::vector<unsigned int> &*tilesN*)
Constructor.

Parameters

- `ranges`: the range of the (first n) state variables
- `tilesN`: the number of tiles to use for each (of the first n) state variable

virtual void **writeOnStream**(std::ostream &*out*)
Writes the basis function to stream

virtual void **readFromStream**(std::istream &*in*)
Read the basis function from stream

class `ReLe::CenteredLogTiles`
This class implements the centered logarithmic spaced tiling. This type of tiling is a uniform tiling when considering the following transformed input space:

$$\hat{input} = sign(input - center) \circ \log(|input - center| + 1)$$

As only a finite number of tiles is supported, the state space must be range limited.

Inherits from *ReLe::BasicTiles*

Public Functions

CenteredLogTiles(**const** *Range* &*range*, unsigned int *tilesN*)
Constructor.

Parameters

- `range`: the range of the first state variable
- `tilesN`: the number of tiles to use for the first state variable

CenteredLogTiles(**const** std::vector<*Range*> &*ranges*, **const** std::vector<unsigned int> &*tilesN*)
Constructor.

Parameters

- `ranges`: the range of the (first n) state variables
- `tilesN`: the number of tiles to use for each (of the first n) state variable

virtual void **writeOnStream**(std::ostream &*out*)
Writes the basis function to stream

virtual void **readFromStream**(std::istream &*in*)
Read the basis function from stream

Features types

template <**class** InputC>
class ReLe::**DenseFeatures_**
This class implements a dense features matrix. A dense features matrix is a feature matrix where all the elements are specified as a set of basis functions. The evaluation of this features class returns a dense matrix.

Inherits from *ReLe::Features_< InputC >*

Public Functions

DenseFeatures_(*BasisFunction_*<InputC> **basisFunction*)
Constructor. Construct a single feature matrix (a scalar).

Parameters

- `basisFunction`: the basis function rappresenting this feature.

DenseFeatures_(BasisFunctions_<InputC> &*basisVector*)
Constructor. Construct a feature vector.

Parameters

- `basisVector`: the set of basis functions to use

DenseFeatures_(BasisFunctions_<InputC> &*basisVector*, unsigned int *rows*, unsigned int *cols*)
Constructor. Construct a feature matrix.

Parameters

- `basisVector`: the set of basis functions to use
- `rows`: the number of rows of the feature matrix
- `cols`: the number of cols of the feature matrix

virtual ~DenseFeatures_()
Destructor. Destroys also all the given basis.

virtual arma::mat **operator()**(**const** InputC &*input*) **const**
Evaluate the features in input

Return the evaluated features

Parameters

- `input`: the input data

virtual size_t **rows**() **const**
Getter.

Return the number of rows of the output feature matrix

virtual size_t **cols**() **const**
Getter.

Return the number of columns of the output feature matrix

template <class InputC>
class ReLe::**SparseFeatures_**

This class implement a sparse features matrix. A sparse feature matrix is a matrix where some basis function are by default zero valued matrix. Despite the name, the evaluation of this features class returns a dense matrix.

Inherits from *ReLe::Features_< InputC >*

Public Functions

SparseFeatures_()

Constructor. Construct an empty set of sparse feature.

SparseFeatures_(BasisFunctions_<InputC> &*basis*, unsigned int *replicationN* = 1, bool *indipendent* = true)

Constructor. Constructs a set of sparse features from the given basis functions.

Parameters

- `basis`: the basis functions to use
- `replicationN`: the number of times the features should be replicated
- `indipendent`: if the features replicated should be padded by zeros, to avoid common parameters.

virtual arma::mat **operator()**(**const** InputC &*input*) **const**

Evaluate the features in input

Return the evaluated features

Parameters

- `input`: the input data

virtual size_t **rows**() **const**

Getter.

Return the number of rows of the output feature matrix

virtual size_t **cols**() **const**

Getter.

Return the number of columns of the output feature matrix

void **addBasis**(unsigned int *row*, unsigned int *col*, *BasisFunction_*<InputC> **bfs*)

Adds a single basis function

Parameters

- `row`: the row where to add the basis function
- `col`: the column where to add the basis function
- `bfs`: the basis function to add

void **setDiagonal**(BasisFunctions_<InputC> &*basis*)

Adds a set of basis function as diagonal features

Parameters

- `basis`: the vector of basis functions

virtual ~SparseFeatures_()

Destructor. Destroys also all the given basis.

template <**class** InputC, bool *denseOutput* = false>
class `ReLe::TilesCoder_`

A tile coder can be used to transform tiles into a set of features. A tile coder evaluate each tile set provided and transform it in a coherent set of binary features sparse matrix.

Inherits from *ReLe::Features_< InputC, denseOutput >*

Public Functions

`TilesCoder_`(*Tiles_*<InputC> **tiles*, unsigned int *outputs* = 1)

Constructor.

Parameters

- `tiles`: a set of tiles to be used
- `outputs`: the number of output features vectors

`TilesCoder_`(TilesVector_<InputC> &*tilesVector*, unsigned int *outputs* = 1)

Constructor.

Parameters

- `tilesVector`: a vector of multiple tilings to be used
- `outputs`: the number of output features vectors

virtual return_type **`operator()`**(**const** InputC &*input*) **const**

Evaluate the features in input

Return the evaluated features

Parameters

- `input`: the input data

virtual size_t **`rows`**() **const**

Getter.

Return the number of rows of the output feature matrix

virtual size_t **`cols`**() **const**

Getter.

Return the number of columns of the output feature matrix

virtual **`~TilesCoder_`**()

Destructor. Destroys also all the tiles passed to the coder.

Batch Dataset Utils

template <class *InputC*, class *OutputC*>
class `ReLe::BatchDataRaw_`

This class represents a dataset of raw input/output data, that can be used to train a ReLe::BatchRegressor

Public Functions

`BatchDataRaw_`()

Constructor.

BatchDataRaw_(std::vector<InputC> *inputs*, std::vector<OutputC> *outputs*)
Constructor.

Parameters

- `inputs`: the vector of input data
- `outputs`: the vector of output data, corresponding to the inputs

virtual *BatchDataRaw_*<InputC, OutputC> ***clone**() **const**
Create a copy of this object, containing an exact copy of the dataset.

Return a pointer to the copy

void **addSample**(**const** InputC &*input*, **const** OutputC &*output*)
Add a sample to the dataset.

Parameters

- `input`: the input to be added
- `output`: the corresponding output

virtual InputC **getInput**(unsigned int *index*) **const**
Getter.

Return the corresponding input

Parameters

- `index`: the index of the input in the dataset

virtual OutputC **getOutput**(unsigned int *index*) **const**
Getter.

Return the corresponding output

Parameters

- `index`: the index of the output in the dataset

virtual size_t **size**() **const**
Getter.

Return the number of input output couples in the dataset.

virtual **~BatchDataRaw_**()
Destructor.

template <class *OutputC*, bool *dense* = true>
class `ReLe::BatchData_`
This interface represents a dataset of input/output data, where the input is a set of precomputed features from the input dataset. An implementation of this interface can be used to train a ReLe::BatchRegressor using the low level method *BatchRegressor::trainFeatures*

Subclassed by *ReLe::BatchDataSimple_< OutputC, dense >*, *ReLe::MiniBatchData_< OutputC, dense >*

Public Types

typedef input_traits<dense>::column_type **features_type**
the type of the input features

typedef input_traits<dense>::type **FeaturesCollection**
the type of the set of input features

typedef output_traits<OutputC>::type **OutputCollection**
the type of the set of the outputs

Public Functions

BatchData_()
Constructor.

virtual *features_type* **getInput**(unsigned int *index*) **const** = 0
Getter.

Return the corresponding input feature vector

Parameters

- index: the index of the input feature vector in the dataset

virtual OutputC **getOutput**(unsigned int *index*) **const** = 0
Getter.

Return the corresponding output

Parameters

- index: the index of the output in the dataset

virtual size_t **size**() **const** = 0
Getter.

Return the number of input output couples in the dataset.

virtual size_t **featuresSize**() **const** = 0
Getter.

Return the dimensionality of the features vectors

virtual *OutputCollection* **getOutputs**() **const** = 0
Getter.

Return the output collection

virtual *FeaturesCollection* **getFeatures**() **const** = 0
Getter.

Return the input features.

virtual *BatchData_* ***clone**() **const** = 0
Create a copy of this object, containing an exact copy of the dataset.

Return a pointer to the copy

virtual *BatchData_* ***cloneSubset**(**const** arma::uvec &*indexes*) **const** = 0
Create a copy of a subste of the dataset.

Return a pointer to the copy of the subset of the dataset

Parameters

- indexes: the set of input features/outputs to be copied

virtual const *BatchData_* ***shuffle**() **const**
Creates a shuffled copy of the dataset.

Return a *ReLe::MiniBatchData_* object with shuffled indexes (by default)

virtual const std::vector<*MiniBatchData_*<OutputC, dense> *> **getMiniBatches**(unsigned int *miniBatchSize*) **const**

Split the dataset in minibatches of constant size.

Return a vector containing a set of pointers to the minibatches

Parameters

- `minibatchSize`: the size of all minibatches (except last)

virtual const std::vector<*MiniBatchData_*<OutputC, dense> *> **getNMiniBatches**(unsigned int *nMiniBatches*) **const**

Split the dataset in a set of N minibatches of equal length. The last one might have a different size.

Return a vector containing a set of pointers to the minibatches

Parameters

- `nMiniBatches`: the number of minibatches to use

OutputC **getMean**() **const**

Getter.

Return the mean output value of the dataset.

arma::mat **getVariance**() **const**

Getter.

Return the output variance of the dataset.

template <class *OutputC*, bool *dense* = true>
class `ReLe::MiniBatchData_`

Implementation of a minibatch of a dataset.

Inherits from *ReLe::BatchData_< OutputC, dense >*

Public Functions

MiniBatchData_(**const** *BatchData_*<OutputC, dense> **data*, **const** arma::uvec &*indexes*)

Construtcor.

Parameters

- `data`: the original dataset.
- `indexes`: the set of elements of the original dataset.

MiniBatchData_(**const** *BatchData_*<OutputC, dense> &*data*, **const** arma::uvec &*indexes*)

Construtcor.

Parameters

- `data`: the original dataset.
- `indexes`: the set of elements of the original dataset.

virtual *BatchData_*<OutputC, dense> ***clone**() **const**

Create a copy of this object, containing an exact copy of the dataset.

Return a pointer to the copy

virtual *BatchData_*<OutputC, dense> ***cloneSubset**(**const** arma::uvec &*indexes*) **const**

Create a copy of a subste of the dataset.

Return a pointer to the copy of the subset of the dataset

Parameters

- `indexes`: the set of input features/outputs to be copied

virtual const *BatchData_*<OutputC, dense> ***shuffle**() **const**
Creates a shuffled copy of the dataset.

Return a *ReLe::MiniBatchData_* object with shuffled indexes (by default)

virtual const std::vector<*MiniBatchData_*<OutputC, dense> *> **getMiniBatches**(unsigned int *miniBatchSize*) **const**
Split the dataset in minibatches of constant size.

Return a vector containing a set of pointers to the minibatches

Parameters

- `minibatchSize`: the size of all minibatches (except last)

virtual features_type **getInput**(unsigned int *index*) **const**
Getter.

Return the corresponding input feature vector

Parameters

- `index`: the index of the input feature vector in the dataset

virtual OutputC **getOutput**(unsigned int *index*) **const**
Getter.

Return the corresponding output

Parameters

- `index`: the index of the output in the dataset

virtual size_t **size**() **const**
Getter.

Return the number of input output couples in the dataset.

virtual size_t **featuresSize**() **const**
Getter.

Return the dimensionality of the features vectors

virtual OutputCollection **getOutputs**() **const**
Getter.

Return the output collection

virtual FeaturesCollection **getFeatures**() **const**
Getter.

Return the input features.

const arma::uvec **getIndexes**() **const**
Getter.

Return the indexes of the elements of the original dataset used by the minibatch

void **setIndexes**(**const** arma::uvec &*indexes*)
Setter.

Parameters

- `indexes`: the indexes of the element of the original dataset to be used

virtual ~MiniBatchData_()
Destructor.

Public Static Functions

static void **cleanMiniBatches**(std::vector<*MiniBatchData_*<OutputC, dense> *> *miniBatches*)
Method used to cleanup a vector of pointers to minibatches

Parameters

- `miniBatches`: the vector of pointers to clean

template <class *OutputC*, bool *dense* = true>
class `ReLe::BatchDataSimple_`
Simple implementation of the *ReLe::BatchData_* interface. Stores all input data in the memory.

Inherits from *ReLe::BatchData_< OutputC, dense >*

Public Functions

BatchDataSimple_(**const** FeaturesCollection &*features*, **const** OutputCollection &*outputs*)
Constructor.

Parameters

- `features`: the collection of input features
- `outputs`: the collection of the corresponding outputs

BatchDataSimple_(FeaturesCollection &&*features*, OutputCollection &&*outputs*)
Constructor.

Parameters

- `features`: the collection of input features
- `outputs`: the collection of the corresponding outputs

virtual size_t **size**() **const**
Getter.

Return the number of input output couples in the dataset.

virtual size_t **featuresSize**() **const**
Getter.

Return the dimensionality of the features vectors

virtual *BatchData_*<OutputC, dense> ***clone**() **const**
Create a copy of this object, containing an exact copy of the dataset.

Return a pointer to the copy

virtual *BatchData_*<OutputC, dense> ***cloneSubset**(**const** arma::uvec &*indexes*) **const**
Create a copy of a subste of the dataset.

Return a pointer to the copy of the subset of the dataset

Parameters

- `indexes`: the set of input features/outputs to be copied

virtual features_type **getInput**(unsigned int *index*) **const**
Getter.

Return the corresponding input feature vector

Parameters

- `index`: the index of the input feature vector in the dataset

virtual OutputC **getOutput**(unsigned int *index*) **const**
Getter.

Return the corresponding output

Parameters

- `index`: the index of the output in the dataset

virtual OutputCollection **getOutputs**() **const**
Getter.

Return the output collection

virtual FeaturesCollection **getFeatures**() **const**
Getter.

Return the input features.

virtual **~BatchDataSimple_**()
Destructor.

template <bool *dense* = true>
class ReLe**::Normalization**
This interface is used to implement a normalization algorithm over a dataset.

See *ReLe::normalizeDataset*

See *ReLe::normalizeDatasetFull*

Subclassed by *ReLe::NoNormalization< denseOutput >*, *ReLe::MinMaxNormalization< dense >*, *ReLe::NoNormalization< dense >*, *ReLe::ZscoreNormalization< dense >*

Public Functions

features_type **operator()**(**const** features_type &*features*)
Normalization operator.

Return the normalized features vector

See *normalize*

Parameters

- `features`: the features vector to be normalized

virtual features_type **normalize**(**const** features_type &*features*) **const** = 0
Compute normalization of a single input feature.

Return the normalized features vector

Parameters

- `features`: the features vector to be normalized

virtual features_type **restore**(**const** features_type &*features*) **const** = 0
Inverse normalization operation over an input feature.

Return the restored features vector

Parameters

- `features`: the features vector to be restored

virtual features_type **rescale**(**const** features_type &*features*) **const** = 0
This method is similar to restore, but, differently from *Normalization::restore*, it does not add the dataset mean, only the relative elements scales.

Return the rescaled features vector

virtual void **readData**(**const** collection_type &*dataset*) = 0
Read the whole dataset in order to compute the parameters for the normalization algorithm.

Parameters

- `dataset`: the dataset over which the normalization should be performed

virtual ~Normalization()
Destructor.

template <bool *dense* = true>
class `ReLe::NoNormalization`
This class implements a fake normaliztion algorithm. This algorithm doesn't perform any normalization.

Inherits from *ReLe::Normalization< dense >*

Public Functions

virtual features_type **normalize**(**const** features_type &*features*) **const**
Compute normalization of a single input feature.

Return the normalized features vector

Parameters

- `features`: the features vector to be normalized

virtual features_type **restore**(**const** features_type &*features*) **const**
Inverse normalization operation over an input feature.

Return the restored features vector

Parameters

- `features`: the features vector to be restored

virtual features_type **rescale**(**const** features_type &*features*) **const**
This method is similar to restore, but, differently from *Normalization::restore*, it does not add the dataset mean, only the relative elements scales.

Return the rescaled features vector

virtual void **readData**(**const** collection_type &*dataset*)
Read the whole dataset in order to compute the parameters for the normalization algorithm.

Parameters

- `dataset`: the dataset over which the normalization should be performed

virtual ~NoNormalization()
Destructor.

template <bool *dense* = true>
class ReLe::**MinMaxNormalization**
This class implements the normalization over an interval. Simply rescales the input features into a new range.

Inherits from *ReLe::Normalization< dense >*

Public Functions

MinMaxNormalization(double *minValue* = 0.0, double *maxValue* = 1.0)
Constructor.

Parameters

- minValue: the new minimum value of the dataset
- maxValue: the new maximum value for the dataset

virtual features_type **normalize**(**const** features_type &*features*) **const**
Compute normalization of a single input feature.

Return the normalized features vector

Parameters

- features: the features vector to be normalized

virtual features_type **restore**(**const** features_type &*features*) **const**
Inverse normalization operation over an input feature.

Return the restored features vector

Parameters

- features: the features vector to be restored

virtual features_type **rescale**(**const** features_type &*features*) **const**
This method is similar to restore, but, differently from *Normalization::restore*, it does not add the dataset mean, only the relative elements scales.

Return the rescaled features vector

virtual void **readData**(**const** collection_type &*dataset*)
Read the whole dataset in order to compute the parameters for the normalization algorithm.

Parameters

- dataset: the dataset over which the normalization should be performed

virtual ~MinMaxNormalization()
Destructor.

template <bool *dense* = true>
class ReLe::**ZscoreNormalization**
This class implements the z-score normalization. This type of normalization consisty of subtracting the dataset mean and dividing each element of the features vector by it's standard deviation.

Inherits from *ReLe::Normalization< dense >*

Public Functions

virtual features_type **normalize**(**const** features_type &*features*) **const**
Compute normalization of a single input feature.

Return the normalized features vector

Parameters

- `features`: the features vector to be normalized

virtual features_type **restore**(**const** features_type &*features*) **const**
Inverse normalization operation over an input feature.

Return the restored features vector

Parameters

- `features`: the features vector to be restored

virtual features_type **rescale**(**const** features_type &*features*) **const**
This method is similar to restore, but, differently from *Normalization::restore*, it does not add the dataset mean, only the relative elements scales.

Return the rescaled features vector

virtual void **readData**(**const** collection_type &*dataset*)
Read the whole dataset in order to compute the parameters for the normalization algorithm.

Parameters

- `dataset`: the dataset over which the normalization should be performed

template <**class** OutputC, bool *dense*>
*BatchDataSimple_*<OutputC, dense> ReLe::**normalizeDataset**(**const** *BatchData_*<OutputC, dense> &*dataset*, *Normalization*<dense> &*normalization*, bool *computeNormalization* = false)
This function can be used to create a normalized version of the dataset. Only input features are normalized.

Return the normalized dataset.

Parameters

- `dataset`: the dataset to be normalized.
- `normalization`: the normalization object to be used
- `computeNormalization`: if the normalization object should be initialized by computing the normalization parameters

template <bool *dense*>
*BatchDataSimple_*<arma::vec, dense> ReLe::**normalizeDatasetFull**(**const** *BatchData_*<arma::vec, dense> &*dataset*, *Normalization*<dense> &*featuresNormalization*, *Normalization*<true> &*outputNormalization*, bool *computeNormalization* = false)
This function can be used to create a normalized version of the dataset. Both input features and output are normalized.

Return the normalized dataset.

Parameters

- `dataset`: the dataset to be normalized.
- `featuresNormalization`: the normalization object to be used for input features
- `outputNormalization`: the normalization object to be used for outputs
- `computeNormalization`: if the normalization object should be initialized by computing the normalization parameters

Policy Representations

template <class *ActionC*, class *StateC*>
class `ReLe::Policy`

A policy provides a distribution over the action space in each state. Formally, it is defined as $\pi : \mathcal{X} \times \mathcal{U} \to [0, 1]$ where $\pi(u|x)$ denotes the probability of action u in state x. It is now clear that one basic function is represented by the possibility of evaluate the probability of an action is a state. The second functionality allows to draw a random action from the action distribution in a specified state. Particular policies are the deterministic policies where the action is deterministically choosen in each state. In this case we can say that $a = \pi(x)$.

Subclassed by ReLe::NonParametricPolicy< ActionC, StateC >, *ReLe::ParametricPolicy< ActionC, StateC >*, ReLe::StochasticDiscretePolicy< ActionC, StateC >

Public Functions

virtual *action_type*<ActionC>::type **operator()** (**typename** *state_type*<StateC>::const_type_ref *state*) = 0

Draw a random action from the distribution induced by the policy in a state x: $u \sim \pi(x)$.

Example:

```
DenseState x();
NormalPolicy policy;
DenseAction u = policy(x);
```

Return an action randomly drawn from the action distribution in the given state

Parameters

- `state`: the state where the policy must be evaluated

virtual double **operator()** (**typename** *state_type*<StateC>::const_type_ref *state*, **typename** *action_type*<ActionC>::const_type_ref *action*) = 0

Evaluate the density function in the provided (state,action)-pair. It computes the probability of an action in a given state: $p = \pi(x, u)$.

Example: *DenseState* x(); *DenseAction* u(); NormalPolicy policy; double pr = policy(x,u);

Return the probability of the (state,action)-pair

Parameters

- `state`: the state where the policy must be evaluated
- `action`: the action to evaluate

virtual std::string **getPolicyName** () = 0

Return a unique identifier of the policy type

Return a string storing the policy name

virtual hyperparameters_map **getPolicyHyperparameters**()
Map the name of the hyperparameters to their value and return such representation.

Return the hyperparameters of the policy

virtual std::string **printPolicy**() = 0
Generate a textual representation of the status of the policy. It is used to visualize the policy. Such representation must be human friendly.

Return The textual description of the policy

virtual *Policy*<ActionC, StateC> ***clone**() = 0
Generate an identical copy of the current policy A new object is created based on the status of the policy at the time of the call. If compared the returned policy is identical to the current policy but it is stored in a different area of the memory.

Return a clone of the current policy

template <class *ActionC*, class *StateC*>
class ReLe::**ParametricPolicy**
A parametric policy is a specialization of a policy where the representation is based on a set of parameters. This policy represents a specific instance of a family of policies. Let $\theta \in \Theta$ be a parameter vector. Then a parametric policy defines a parametric distribution over the action space, i.e.,

$$\pi(u|x,\theta) \qquad \forall x \in \mathcal{X}.$$

This means that the shape of the action distribution depends both on the state s and on the parameter vector θ.

Example:

```
Consider a normal distribution. A parametric normal policy is a
normal distribution where the mean and the standard deviation
are parametrized by a vector THETA.
```

Inherits from *ReLe::Policy< ActionC, StateC >*

Subclassed by *ReLe::DifferentiablePolicy< ActionC, StateC >*

Public Functions

virtual arma::vec **getParameters**() **const** = 0
Provide a way to access the parameters.

Return the vector of parameters θ

virtual const unsigned int **getParametersSize**() **const** = 0
Return the number of parameters

Return the length of the vector θ

virtual void **setParameters**(**const** arma::vec &*w*) = 0
Provide a way to modify the parameters of the policy by replacing the current parameter vector with the provided one.

Parameters

- w: the new policy parameters

template <class *ActionC*, class *StateC*>
class `ReLe::DifferentiablePolicy`

A differentiable policy is a specialization of a parametric policy that provides first- and second-order derivatives. The derivatives are computed w.r.t. the parameter vector and it is evaluated in the provided (state,action)-pair with the current policy parametrization. Let $\hat{x}, \hat{u}, \hat{\theta} \subseteq R^d$ be the provided state, action and the current policy representation, respectively. Then the first-order derivative of the policy is given by

$$\nabla_\theta \pi(\hat{u}|\hat{x}, \theta)\big|_{\hat{\theta}}.$$

Note that when the policy is deterministic the diff operator works on the deterministic function, i.e.,

$$\nabla_\theta \pi(u|\theta)\big|_{\hat{\theta}}.$$

Inherits from *ReLe::ParametricPolicy< ActionC, StateC >*

Subclassed by ReLe::GenericParametricMixturePolicy< ActionC, StateC >

Public Functions

virtual arma::vec **`diff`**(**typename** *state_type*<StateC>::const_type_ref *state*, **typename** *action_type*<ActionC>::const_type_ref *action*)

Compute the first-order derivative of the policy function which is evaluated in the provided state $\hat{x}$ and action $\hat{u}$ using the current parameter vector $\hat{\theta}$:

$$\nabla_\theta \pi(\hat{u}|\hat{x}, \theta)\big|_{\hat{\theta}} \in \mathbb{R}^d.$$

Return the first-order derivative

Parameters

- `state`: the state
- `action`: the action

virtual arma::vec **`difflog`**(**typename** *state_type*<StateC>::const_type_ref *state*, **typename** *action_type*<ActionC>::const_type_ref *action*) = 0

Compute the first-order derivative of the policy logarithm which is evaluated in the provided state $\hat{x}$ and action $\hat{u}$ using the current parameter vector $\hat{\theta}$:

$$\nabla_\theta \log \pi(\hat{u}|\hat{x}, \theta)\big|_{\hat{\theta}} \in \mathbb{R}^d.$$

Return the first-order derivative of the policy logarithm

Parameters

- `state`: the state
- `action`: the action

virtual arma::mat **`diff2log`**(**typename** *state_type*<StateC>::const_type_ref *state*, **typename** *action_type*<ActionC>::const_type_ref *action*) = 0

Compute the second-order derivative of the policy logarithm which is evaluated in the provided state $\hat{x}$ and action $\hat{u}$ using the current parameter vector $\hat{\theta}$:

$$H_\theta \log \pi(\hat{u}|\hat{x}, \theta)\big|_{\hat{\theta}} \in \mathbb{R}^{d \times d}.$$

Note that the Hessian matrix is given by

$$H(i,j) = \frac{\partial^2}{\partial\theta_i \partial_j} \log \pi(\hat{u}|\hat{x},\theta)\big|_{\hat{\theta}},$$

where i is the row and j is the column.

Return the hessian of the policy logarithm

Parameters

- `state`: the state
- `action`: the action

Normal Policies

class `ReLe::GenericMVNPolicy`

This class represents a multivariate Normal policy with fixed covariance matrix and generic parametric approximation of the mean value:

$$\pi^{\theta}(u|x) = \mathcal{N}\left(u; \mu(x,\theta), \Sigma\right), \qquad \forall x \in \mathbb{R}^{n_x}, u \in \mathbb{R}^{n_u},$$

where θ is a k-dimensional vector.

The parameters to be optimized are the one of the mean approximator, i.e., θ.

Example:

```
BasisFunctions basis = IdentityBasis::generate(2);
SparseFeatures phi;
phi.setDiagonal(basis);
arma::vec w = {1.0, 1.0};
LinearApproximator regressor(phi);
regressor.setParameters(w);
GenericMVNPolicy policy(regressor);

arma::vec state  = mvnrand({0.0, 0.0}, arma::diagmat(arma::vec({10.0, 10.0})));
arma::vec action = policy(state);
arma::vec diff = policy.difflog(state, action);
```

Inherits from *ReLe::DifferentiablePolicy< DenseAction, DenseState >*

Subclassed by *ReLe::GenericMVNDiagonalPolicy*, *ReLe::GenericMVNStateDependantStddevPolicy*

class `ReLe::GenericMVNDiagonalPolicy`

This class implements a generic multivariate normal policy with mean represented through a generic parametric regressor and diagonal covariance matrix:

$$\pi^{\theta}(u|x) = \mathcal{N}\left(u; \mu(x,\omega), diag(\sigma^2)\right), \qquad \forall x \in \mathbb{R}^{n_x}, u \in \mathbb{R}^{n_u},$$

where ω is a k-dimensional vector and σ is a d-dimensional vector. Note that the power operator is to be considered component-wise.

The parameters to be optimized are $\theta = [\omega, \sigma]$.

Inherits from *ReLe::GenericMVNPolicy*

class ReLe::**GenericMVNStateDependantStddevPolicy**

This class implements a multivariate normal policy with both mean and covariance represented through parametric functions. Let $f_\mu : \mathcal{X} \times \Omega \to n_u$ and $f_\Sigma : \mathcal{X} \times \Sigma \to n_u$. Then the policy class is defined as

$$\pi^\theta(u|x) = \mathcal{N}\left(u; f_\mu(x,\omega), diag(f_\Sigma(x,\sigma)^2)\right), \qquad \forall x \in \mathbb{R}^{n_x}, u \in \mathbb{R}^{n_u},$$

where ω is a k-dimensional vector and σ is a d-dimensional vector. Note that the power operator is to be considered component-wise.

The parameters to be optimized are $\theta = [\omega, \sigma]$.

Inherits from *ReLe::GenericMVNPolicy*

Solvers

class ReLe::**DynamicProgrammingAlgorithm**

This class implements the Dynamic Programming (DP) algorithm.

Inherits from *ReLe::Solver< FiniteAction, FiniteState >*

Subclassed by *ReLe::PolicyIteration*, *ReLe::ValueIteration*

Public Functions

DynamicProgrammingAlgorithm(*FiniteMDP &mdp*)

Constructor.

Parameters

- mdp: the mdp to solve

virtual *Policy*<*FiniteAction*, *FiniteState*> &**getPolicy**()

Getter

Return the set the optimal policy from the MDP

virtual *Dataset*<*FiniteAction*, *FiniteState*> **test**()

This method runs the policy computed by *Solver::solve()* over the MDP. Must be implemented. Normally the implementation is simply wrapper for *Solver::test(Environment<ActionC, StateC>& env, Policy<ActionC, StateC>& pi)*

Return the set of trajectories sampled with the optimal policy from the MDP

arma::vec **getValueFunction**()

Getter

Return the value function

class ReLe::**PolicyIteration**

This class implements the *Policy* Iteration algorithm.

Inherits from *ReLe::DynamicProgrammingAlgorithm*

Public Functions

PolicyIteration(*FiniteMDP &mdp*)

Constructor.

Parameters

- `mdp`: the mdp to solve

virtual void **`solve`**`()`
This method run the computation of the optimal policy for the MDP. Must be implemented.

virtual **`~PolicyIteration`**`()`
Destructor.

class `ReLe::`**`ValueIteration`**
This class implements the Value Iteration algorithm.

Inherits from *ReLe::DynamicProgrammingAlgorithm*

Public Functions

`ValueIteration``(`*FiniteMDP* *&mdp*, double *eps*`)`
Constructor.

Parameters

- `mdp`: the mdp to solve
- `eps`: threshold to be evaluated to stop the algorithm

virtual void **`solve`**`()`
This method run the computation of the optimal policy for the MDP. Must be implemented.

virtual **`~ValueIteration`**`()`
Destructor.

class `ReLe::`**`LQRExact`**
This class is not strictly a solver, but it can be used to calculate the exact expected return, the exact gradient, and the exact hessian of any *LQR* problem, and thus can be used by any optimization algorithm to find the optimal value for this kind of problem, even in the multidimensional reward setting.

Public Functions

`LQRExact``(`double *gamma*, arma::mat *A*, arma::mat *B*, std::vector<arma::mat> *Q*, std::vector<arma::mat> *R*, arma::vec *x0*`)`
Constructor.

Parameters

- `gamma`: the discout factor
- `A`: the state dynamics matrix
- `B`: the action dynamics matrix
- `Q`: a vector of weights matrixes for the state
- `R`: a vector of weights matrixes for the action
- `x0`: the initial state for the *LQR* problem

`LQRExact``(`*LQR* *&lqr*`)`
Constructor.

Parameters

- `lqr`: the *LQR* environment to be considered

arma::mat **solveRiccati**(**const** arma::vec &*k*, unsigned int *r* = 0)
Solves the Riccati equation for a given parameters vector.

Return the solution to the Riccati equation

Parameters

- `k`: the weights vector
- `r`: the reward index

arma::mat **riccatiRHS**(**const** arma::vec &*k*, **const** arma::mat &*P*, unsigned int *r* = 0)
Compute the right hand side of the Riccati equation for a given parameters vector.

Return the value of the right hand side of the Riccati equation

Parameters

- `k`: the weights vector
- `r`: the reward index

arma::vec **computeJ**(**const** arma::vec &*k*, **const** arma::mat &*Sigma*)
Compute the expected return under a normal policy.

Return the expected return vector

Parameters

- `k`: the weights vector
- `Sigma`: the covariance matrix

arma::mat **computeGradient**(**const** arma::vec &*k*, **const** arma::mat &*Sigma*, unsigned int *r* = 0)
Compute the gradient of the expected return under a normal policy, w.r.t the parameters k.

Return the gradient of the r component of the expected reward

Parameters

- `k`: the weights vector
- `Sigma`: the covariance matrix
- `r`: the reward index

arma::mat **computeJacobian**(**const** arma::vec &*k*, **const** arma::mat &*Sigma*)
Compute the jacobian of the expected return under a normal policy, w.r.t the parameters k.

Return the jacobian of the expected reward

Parameters

- `k`: the weights vector
- `Sigma`: the covariance matrix

arma::mat **computeHesian**(**const** arma::vec &*k*, **const** arma::mat &*Sigma*, unsigned int *r* = 0)
Compute the hessian of the expected return under a normal policy, w.r.t the parameters k.

Return the hessian of the expected return

Parameters

- `k`: the weights vector
- `Sigma`: the covariance matrix
- `r`: the reward index

class ReLe::**LQRsolver**

This class implements the Linear-Quadratic Regulator (*LQR*) solver.

Inherits from *ReLe::Solver< DenseAction, DenseState >*

Subclassed by ReLe::IRL_LQRSolver

Public Functions

LQRsolver(*LQR* &*lqr*, Features &*phi*, Type *type* = Type::MOO)

Constructor.

Parameters

- `lqr`: the Linear-Quadratic Regulator
- `phi`: features
- `type`: the type of Linear-Quadratic Regulator

virtual void **solve**()

This method run the computation of the optimal policy for the MDP. Must be implemented.

virtual *Dataset*<*DenseAction*, *DenseState*> **test**()

This method runs the policy computed by *Solver::solve()* over the MDP. Must be implemented. Normally the implementation is simply wrapper for *Solver::test(Environment<ActionC, StateC>& env, Policy<ActionC, StateC>& pi)*

Return the set of trajectories sampled with the optimal policy from the MDP

virtual *Policy*<*DenseAction*, *DenseState*> &**getPolicy**()

Getter

Return the set the optimal policy from the MDP

arma::mat **computeOptSolution**()

Compute the optimal solution of the problem.

Return the matrix with the optimal solution

void **setRewardIndex**(unsigned int *rewardIndex*)

Setter.

Parameters

- `rewardIndex`: the index of the reward to set

void **setRewardWeights**(arma::vec &*weights*)

Setter.

Parameters

- `weights`: the weights to be set

Distributions

class ReLe::**Distribution**

This is an interface for a generic distribution. A distribution is a multivariate statistical distribution that can change over time. This interface implements the operators to get the pdf of the distribution at a certain point, and to sample data according to the current state of the distribution.

Distribution can be used for describing high level policies, i.e. a distribution of parametric policies.

Subclassed by *ReLe::DifferentiableDistribution*, *ReLe::WishartBase*

Public Functions

Distribution(unsigned int *dim*)
Constructor.

Parameters

- `dim`: the number of variables of the distribution

virtual ~Distribution()
Destructor.

virtual arma::vec **operator()**() **const** = 0
Draw a point from the support of the distribution according to the probability defined by the distribution

Return a randomly generated point

virtual double **operator()**(**const** arma::vec &*point*) **const**
Return the probability of a point to be generated from the distribution.

Return the probability of the point

Parameters

- `point`: a point to be evaluated

virtual double **logPdf**(**const** arma::vec &*point*) **const** = 0
Return the logarithm of the probability of a point to be generated from the distribution.

Return the logarithm of the probability of the point

Parameters

- `point`: a point to be evaluated

unsigned int **getPointSize**() **const**
Getter.

Return The size of the support

virtual std::string **getDistributionName**() **const** = 0
Getter.

Return the name of the distribution

virtual arma::mat **getMean**() **const** = 0
Getter.

Return the distribution mean

virtual arma::mat **getCovariance**() **const** = 0
Getter.

Return the distribution covariance

virtual arma::mat **getMode**() **const** = 0
Getter.

Return the distribution mode

virtual void **wmle**(**const** arma::vec &*weights*, **const** arma::mat &*samples*) = 0
This method implements the weighted maximum likelihood estimate of the distribution, given a set of weighted samples.

Parameters

- `weights`: the weights for each sample.
- `samples`: the set of samples from the distribution

Protected Attributes

unsigned int **pointSize**
the number of variables of the distribution

class `ReLe::DifferentiableDistribution`
This class represents a generic parametrized distribution $x \sim D(\cdot|\rho)$ where $\rho \in \mathbb{R}^d$ is the parameter vector and $X \subseteq \mathbb{R}^n$ is the support space.

Inherits from *ReLe::Distribution*

Subclassed by *ReLe::ParametricNormal*

Public Functions

DifferentiableDistribution(unsigned int *dim*)
Constructor.

Parameters

- `dim`: the number of variables of the distribution

virtual unsigned int **getParametersSize**() **const** = 0
Getter.

Return The size of the parameters

virtual arma::vec **getParameters**() **const** = 0
Getter.

Return The parameters vector

virtual void **setParameters**(**const** arma::vec &*parameters*) = 0
Setter.

Parameters

- `parameters`: The new parameters of the distribution.

virtual void **update**(**const** arma::vec &*increment*) = 0
Update the internal parameters according to the given increment vector.

Parameters

- `increment`: a vector of increment value for each component

virtual arma::vec **difflog**(**const** arma::vec &*point*) **const** = 0
Compute the gradient of the logarithm of the distribution in the given point

Return the gradient vector

Parameters

- `point`: the point where the gradient is evaluated

virtual arma::mat **diff2log**(**const** arma::vec &*point*) **const** = 0
Compute the hessian ($d(d \log D)^T$) of the logarithm of the distribution in the given point.

Return the hessian matrix (out)

Parameters

- `point`: the point where the hessian is evaluated

virtual arma::vec **pointDifflog**(**const** arma::vec &*point*) **const** = 0
Compute the gradient of the logarithm of the distribution in the current params w.r.t. the given point. differently from difflog, the computed gradient is not the parameter's gradient, but the input gradient, i.e. how much the probability changes if the input changes.

Return the gradient vector

Parameters

- `point`: the point where the gradient is evaluated

class `ReLe::FisherInterface`
This interface can be implemented from a distribution that has a closed form fisher information matrix computation.

Subclassed by *ReLe::ParametricCholeskyNormal*, *ReLe::ParametricDiagonalNormal*, *ReLe::ParametricFullNormal*

Public Functions

virtual ~FisherInterface()
Destructor

virtual arma::sp_mat **FIM**() **const** = 0
Computes the fisher information matrix of the distribution.

virtual arma::sp_mat **inverseFIM**() **const** = 0
Computes the inverse of the fisher information matrix.

Normal Distributions

class `ReLe::ParametricNormal`
This class implements the Parametric Normal distribution.

$$x \in \mathbb{R}^n, x \sim \mathcal{N}(\mu, \Sigma)$$

This is the basic class of all normal distributions, by default only the mean is parametrized. This mean that the covariance matrix Σ is fixed.

Inherits from *ReLe::DifferentiableDistribution*

Subclassed by *ReLe::ParametricCholeskyNormal*, *ReLe::ParametricDiagonalNormal*, *ReLe::ParametricFullNormal*, *ReLe::ParametricLogisticNormal*

Public Functions

ParametricNormal(unsigned int *dim*)
Constructor.

Parameters

- `dim`: the number of variables of the distribution

ParametricNormal(**const** arma::vec &*params*, **const** arma::mat &*covariance*)
Constructor.

Parameters

- `params`: the parameters of the distribution
- `covariance`: the covariance matrix

virtual ~ParametricNormal()
Destructor.

virtual arma::vec **operator()**() **const**
Draw a point from the support of the distribution according to the probability defined by the distribution

Return a randomly generated point

virtual double **logPdf**(**const** arma::vec &*point*) **const**
Return the logarithm of the probability of a point to be generated from the distribution.

Return the logarithm of the probability of the point

Parameters

- `point`: a point to be evaluated

virtual std::string **getDistributionName**() **const**
Getter.

Return the name of the distribution

virtual void **wmle**(**const** arma::vec &*weights*, **const** arma::mat &*samples*)
This method implements the weighted maximum likelihood estimate of the distribution, given a set of weighted samples.

Parameters

- `weights`: the weights for each sample.
- `samples`: the set of samples from the distribution

unsigned int **getParametersSize**() **const**
Getter.

Return The size of the parameters

virtual arma::vec **getParameters**() **const**
Getter.

Return The parameters vector

virtual void **setParameters**(**const** arma::vec &*parameters*)
Setter.

Parameters

- `parameters`: The new parameters of the distribution.

virtual void **update**(**const** arma::vec &*increment*)
Update the internal parameters according to the given increment vector.

Parameters

- `increment`: a vector of increment value for each component

virtual arma::vec **difflog**(**const** arma::vec &*point*) **const**
Compute the gradient of the logarithm of the distribution in the given point

Return the gradient vector

Parameters

- `point`: the point where the gradient is evaluated

virtual arma::mat **diff2log**(**const** arma::vec &*point*) **const**
Compute the hessian ($d(d \log D)^T$) of the logarithm of the distribution in the given point.

Return the hessian matrix (out)

Parameters

- `point`: the point where the hessian is evaluated

virtual arma::vec **pointDifflog**(**const** arma::vec &*point*) **const**
Compute the gradient of the logarithm of the distribution in the current params w.r.t. the given point. differently from difflog, the computed gradient is not the parameter's gradient, but the input gradient, i.e. how much the probability changes if the input changes.

Return the gradient vector

Parameters

- `point`: the point where the gradient is evaluated

arma::mat **getMean**() **const**
Getter.

Return the distribution mean

arma::mat **getCovariance**() **const**
Getter.

Return the distribution covariance

arma::mat **getMode**() **const**
Getter.

Return the distribution mode

class `ReLe::ParametricDiagonalNormal`
This class represents a parametric Gaussian distribution with parameters ρ:

$$x \sim \mathcal{N}(\cdot|\rho).$$

The parameter vector ρ is then defined as follows:

$$\rho = [M, \Sigma]^T$$

where $M = [\mu_1, \ldots, \mu_n]$, $\Sigma = diag(\sigma_1, \ldots, \sigma_n)$ and n is the support dimension. As a consequence, the parameter dimension is $2 \cdot n$.

Given a parametrization ρ, the distribution is defined by the mean vector M and a diagonal covariance matrix Σ.

Inherits from *ReLe::ParametricNormal*, *ReLe::FisherInterface*

Public Functions

ParametricDiagonalNormal(**const** arma::vec &*mean*, **const** arma::vec &*covariance*)
Constructor.

Parameters

- `mean`: the initial value for the mean
- `covariance`: the initial covariance

virtual ~ParametricDiagonalNormal()
Destructor.

virtual std::string **getDistributionName**() **const**
Getter.

Return the name of the distribution

virtual void **wmle**(**const** arma::vec &*weights*, **const** arma::mat &*samples*)
This method implements the weighted maximum likelihood estimate of the distribution, given a set of weighted samples.

Parameters

- `weights`: the weights for each sample.
- `samples`: the set of samples from the distribution

arma::vec **difflog**(**const** arma::vec &*point*) **const**
Compute the gradient of the logarithm of the distribution in the given point

Return the gradient vector

Parameters

- `point`: the point where the gradient is evaluated

arma::mat **diff2log**(**const** arma::vec &*point*) **const**
Compute the hessian ($d(d \log D)^T$) of the logarithm of the distribution in the given point.

Return the hessian matrix (out)

Parameters

- `point`: the point where the hessian is evaluated

arma::sp_mat **FIM**() **const**
Computes the fisher information matrix of the distribution.

arma::sp_mat **inverseFIM**() **const**
Computes the inverse of the fisher information matrix.

unsigned int **getParametersSize**() **const**
Getter.

Return The size of the parameters

virtual arma::vec **getParameters**() **const**
Getter.

Return The parameters vector

virtual void **setParameters**(**const** arma::vec &*parameters*)
Setter.

Parameters

- `parameters`: The new parameters of the distribution.

virtual void **update**(**const** arma::vec &*increment*)
Update the internal parameters according to the given increment vector.

Parameters

- `increment`: a vector of increment value for each component

class `ReLe::ParametricLogisticNormal`
This class represents a parametric Gaussian distribution with parameters ρ:

$$x \sim \mathcal{N}(\cdot|\rho).$$

The parameter vector ρ is then defined as follows:

$$\rho = [M, \Omega]^T$$

where $M = [\mu_1, \dots, \mu_n]$, $\Omega = [\omega_1, \dots, \omega_n]$ and n is the support dimension. As a consequence, the parameter dimension is $2 \cdot n$.

Given a parametrization ρ, the distribution is defined by the mean vector M and a covariance matrix Σ. In order to reduce the number of parameters, we discard the cross-correlation terms in the covariance matrix: $\Sigma = diag(\sigma_1, \dots, \sigma_n)$. Moreover, in order to prevent the variance from becoming negative we exploit the parametrization presented by Kimura and Kobayashi (1998), where σ_i is represented by a logistic function parameterized by ω_i:

$$\sigma_i = \frac{\tau}{1 + e^{-\omega_i}}.$$

Inherits from *ReLe::ParametricNormal*

Public Functions

ParametricLogisticNormal(unsigned int *point_dim*, double *variance_asymptote*)
Constructor.

Parameters

- `point_dim`: the number of variables of the distribution
- `variance_asymptote`: the asymptotic value for the variance of each variable

ParametricLogisticNormal(**const** arma::vec &*mean*, **const** arma::vec &*logWeights*, double *variance_asymptote*)
Constructor.

Parameters

- `mean`: the initial mean value
- `logWeights`: the initial weights for the logistic function

- `variance_asymptote`: the asymptotic value for the variance of each variable

ParametricLogisticNormal(**const** arma::vec &*variance_asymptote*)
Constructor.

Parameters

- `variance_asymptote`: a vector of the asymptotic values for the variance of each variable

ParametricLogisticNormal(**const** arma::vec &*mean*, **const** arma::vec &*logWeights*, **const** arma::vec &*variance_asymptote*)
Constructor.

Parameters

- `mean`: the initial mean value
- `logWeights`: the initial weights for the logistic function
- `variance_asymptote`: a vector of the asymptotic values for the variance of each variable

virtual ~ParametricLogisticNormal()
Destructor.

virtual std::string **getDistributionName**() **const**
Getter.

Return the name of the distribution

virtual void **wmle**(**const** arma::vec &*weights*, **const** arma::mat &*samples*)
This method implements the weighted maximum likelihood estimate of the distribution, given a set of weighted samples.

Parameters

- `weights`: the weights for each sample.
- `samples`: the set of samples from the distribution

arma::vec **difflog**(**const** arma::vec &*point*) **const**
Compute the gradient of the logarithm of the distribution in the given point

Return the gradient vector

Parameters

- `point`: the point where the gradient is evaluated

arma::mat **diff2log**(**const** arma::vec &*point*) **const**
Compute the hessian ($d(d\log D)^T$) of the logarithm of the distribution in the given point.

Return the hessian matrix (out)

Parameters

- `point`: the point where the hessian is evaluated

unsigned int **getParametersSize**() **const**
Getter.

Return The size of the parameters

virtual arma::vec **getParameters**() **const**
Getter.

Return The parameters vector

virtual void **setParameters**(**const** arma::vec &*parameters*)
Setter.

Parameters

- `parameters`: The new parameters of the distribution.

virtual void **update**(**const** arma::vec &*increment*)
Update the internal parameters according to the given increment vector.

Parameters

- `increment`: a vector of increment value for each component

class `ReLe::ParametricCholeskyNormal`
This class represents a parametric Gaussian distribution with parameters ρ:

$$x \sim \mathcal{N}(\cdot|\rho).$$

The parameter vector ρ is then defined as follows:

$$\rho = [M, \Omega]^T$$

where $M = [\mu_1, \dots, \mu_n]$, $\Omega = [\omega_1, \dots, \omega_n]$ and n is the support dimension. As a consequence, the parameter dimension is $2 \cdot n$.

Given a parametrization ρ, the distribution is defined by the mean vector M and a covariance matrix Σ. In order to reduce the number of parameters and prevent the matrix become not positive definite, we parametrize the covariance matrix with the cholesky decomposition of the Covariance matrix, such that:

$$\Sigma = triangular(\Omega)^T triangular(\Omega)$$

Inherits from *ReLe::ParametricNormal*, *ReLe::FisherInterface*

Public Functions

ParametricCholeskyNormal(**const** arma::vec &*initial_mean*, **const** arma::mat &*initial_cholA*)
Constructor.

Parameters

- `initial_mean`: the initial mean parameters
- `initial_cholA`: the initial cholesky decomposition of the covariance matrix

virtual ~ParametricCholeskyNormal()
Destructor.

virtual std::string **getDistributionName**() **const**
Getter.

Return the name of the distribution

arma::vec **difflog**(**const** arma::vec &*point*) **const**
Compute the gradient of the logarithm of the distribution in the given point

Return the gradient vector

Parameters

- `point`: the point where the gradient is evaluated

arma::mat **diff2log**(**const** arma::vec &*point*) **const**
Compute the hessian ($d(d \log D)^T$) of the logarithm of the distribution in the given point.

Return the hessian matrix (out)

Parameters

- `point`: the point where the hessian is evaluated

arma::sp_mat **FIM**() **const**
Computes the fisher information matrix of the distribution.

arma::sp_mat **inverseFIM**() **const**
Computes the inverse of the fisher information matrix.

virtual void **wmle**(**const** arma::vec &*weights*, **const** arma::mat &*samples*)
This method implements the weighted maximum likelihood estimate of the distribution, given a set of weighted samples.

Parameters

- `weights`: the weights for each sample.
- `samples`: the set of samples from the distribution

unsigned int **getParametersSize**() **const**
Getter.

Return The size of the parameters

virtual arma::vec **getParameters**() **const**
Getter.

Return The parameters vector

virtual void **setParameters**(**const** arma::vec &*parameters*)
Setter.

Parameters

- `parameters`: The new parameters of the distribution.

virtual void **update**(**const** arma::vec &*increment*)
Update the internal parameters according to the given increment vector.

Parameters

- `increment`: a vector of increment value for each component

class `ReLe::ParametricFullNormal`
This class represents a parametric Gaussian distribution.

Differently from *ReLe::ParametricCholeskyNormal*, it uses a full parametrization of the covariance matrix, so the algorithm itself needs to maintain the positive definiteness of the parametrization.

Usually this class is used when the algorithm provide a full estimation of the covariance matrix or for weighted maximum likelihood.

Inherits from *ReLe::ParametricNormal*, *ReLe::FisherInterface*

Public Functions

ParametricFullNormal(**const** arma::vec &*initial_mean*, **const** arma::mat &*initial_cov*)
Constructor.

Parameters

- `initial_mean`: the initial mean parameters
- `initial_cov`: the initial covariance matrix

virtual ~ParametricFullNormal()
Destructor.

virtual std::string **getDistributionName**() **const**
Getter.

Return the name of the distribution

arma::vec **difflog**(**const** arma::vec &*point*) **const**
Compute the gradient of the logarithm of the distribution in the given point

Return the gradient vector

Parameters

- `point`: the point where the gradient is evaluated

arma::mat **diff2log**(**const** arma::vec &*point*) **const**
Compute the hessian ($d(d \log D)^T$) of the logarithm of the distribution in the given point.

Return the hessian matrix (out)

Parameters

- `point`: the point where the hessian is evaluated

arma::sp_mat **FIM**() **const**
Computes the fisher information matrix of the distribution.

arma::sp_mat **inverseFIM**() **const**
Computes the inverse of the fisher information matrix.

virtual void **wmle**(**const** arma::vec &*weights*, **const** arma::mat &*samples*)
This method implements the weighted maximum likelihood estimate of the distribution, given a set of weighted samples.

Parameters

- `weights`: the weights for each sample.
- `samples`: the set of samples from the distribution

unsigned int **getParametersSize**() **const**
Getter.

Return The size of the parameters

virtual arma::vec **getParameters**() **const**
Getter.

Return The parameters vector

virtual void **setParameters**(**const** arma::vec &*parameters*)
Setter.

Parameters

- `parameters`: The new parameters of the distribution.

virtual void **update**(**const** arma::vec &*increment*)
Update the internal parameters according to the given increment vector.

Parameters

- `increment`: a vector of increment value for each component

Wishart Distributions

class `ReLe::WishartBase`
This class is the base class for *Wishart* and Inverse-Wishart Distributions.

Inherits from *ReLe::Distribution*

Subclassed by *ReLe::InverseWishart*, *ReLe::Wishart*

Public Functions

WishartBase(unsigned int *p*)
Constructor.

Parameters

- `p`: the number of columns (and rows) of the sampled matrix

WishartBase(unsigned int *p*, unsigned int *nu*)
Constructor.

Parameters

- `p`: the number of columns (and rows) of the sampled matrix
- `nu`: the degrees of freedom of the distribution

unsigned int **getNu**() **const**
Getter.

Return the degrees of freedom of the distribution

virtual double **operator()**(**const** arma::vec &*point*) **const**
Return the probability of a point to be generated from the distribution.

Return the probability of the point

Parameters

- `point`: a point to be evaluated

virtual double **logPdf**(**const** arma::vec &*point*) **const** = 0
Return the logarithm of the probability of a point to be generated from the distribution.

Return the logarithm of the probability of the point

Parameters

- `point`: a point to be evaluated

class ReLe::**Wishart**

This class implements a *Wishart* distribution. This distribution is commonly used for precision matrix estimation.

Inherits from *ReLe::WishartBase*

Public Functions

Wishart(unsigned int *p*)

Constructor.

Parameters

- p: the number of rows and columns in the matrix

Wishart(unsigned int *p*, unsigned int *nu*)

Constructor.

Parameters

- p: the number of rows and columns in the matrix
- nu: he degrees of freedom of the wishart distribution

Wishart(unsigned int *nu*, **const** arma::mat &*V*)

Constructor.

Parameters

- nu: the degrees of freedom of the wishart distribution
- V: the covariance of the distribution

arma::mat **getV**() **const**

Getter.

Return the covariance matrix of the distribution

virtual **~Wishart**()

Destructor.

virtual arma::vec **operator()**() **const**

Draw a point from the support of the distribution according to the probability defined by the distribution

Return a randomly generated point

virtual double **logPdf**(**const** arma::vec &*point*) **const**

Return the logarithm of the probability of a point to be generated from the distribution.

Return the logarithm of the probability of the point

Parameters

- point: a point to be evaluated

virtual std::string **getDistributionName**() **const**

Getter.

Return the name of the distribution

arma::mat **getMean**() **const**

Getter.

Return the distribution mean

arma::mat **getCovariance**() **const**
Getter.

Return the distribution covariance

arma::mat **getMode**() **const**
Getter.

Return the distribution mode

virtual void **wmle**(**const** arma::vec &*weights*, **const** arma::mat &*samples*)
This method implements the weighted maximum likelihood estimate of the distribution, given a set of weighted samples.

Parameters

- `weights`: the weights for each sample.
- `samples`: the set of samples from the distribution

class `ReLe::InverseWishart`
This class implements a *Wishart* distribution. This distribution is commonly used for precision matrix estimation.

Inherits from *ReLe::WishartBase*

Public Functions

InverseWishart(unsigned int *p*)
Constructor.

Parameters

- `p`: the number of rows and columns in the matrix

InverseWishart(unsigned int *p*, unsigned int *nu*)
Constructor.

Parameters

- `p`: the number of rows and columns in the matrix
- `nu`: he degrees of freedom of the wishart distribution

InverseWishart(unsigned int *nu*, **const** arma::mat &*Psi*)
Constructor.

Parameters

- `nu`: the degrees of freedom of the wishart distribution
- `Psi`: the scale matrix of the distribution

arma::mat **getPsi**() **const**
Getter.

Return the scale matrix of the distribution

virtual **~InverseWishart**()
Destructor.

virtual arma::vec **operator()**() **const**
Draw a point from the support of the distribution according to the probability defined by the distribution

Return a randomly generated point

virtual double **logPdf**(**const** arma::vec &*point*) **const**
Return the logarithm of the probability of a point to be generated from the distribution.

Return the logarithm of the probability of the point

Parameters

- `point`: a point to be evaluated

virtual std::string **getDistributionName**() **const**
Getter.

Return the name of the distribution

arma::mat **getMean**() **const**
Getter.

Return the distribution mean

arma::mat **getCovariance**() **const**
Getter.

Return the distribution covariance

arma::mat **getMode**() **const**
Getter.

Return the distribution mode

virtual void **wmle**(**const** arma::vec &*weights*, **const** arma::mat &*samples*)
This method implements the weighted maximum likelihood estimate of the distribution, given a set of weighted samples.

Parameters

- `weights`: the weights for each sample.
- `samples`: the set of samples from the distribution

Environments

class `ReLe::Dam`
This class implements the *Dam* problem environment. The aim of this optimization problem is to decide the amount of water to release in order to satisfy conflicting objectives.

References

Castelletti, Pianosi, Restelli. A multiobjective reinforcement learning approach to water resources systems operation: Pareto frontier approximation in a single run. Water Resources Journal

Inherits from *ReLe::ContinuousMDP*

Public Functions

Dam()
Constructor.

Dam(DamSettings &*config*)
Constructor.

Parameters

- `config`: settings of the environment

virtual void **`step`**(**const** *DenseAction* &*action*, *DenseState* &*nextState*, Reward &*reward*) **See** *Environment::step*

virtual void **`getInitialState`**(*DenseState* &*state*) **See** *Environment::getInitialState*

void **`setCurrentState`**(**const** *DenseState* &*state*)
Set the current state.

Parameters

- `state`: current state

const DamSettings &**`getSettings`**() **const** **See** *Environment::getSettings*

class `ReLe::DeepSeaTreasure`
This class implements the Deep Sea Treasure problem. This task is a grid world modeling a submarine environment with multiple treasures with different values. The aim is to minimize the time to reach the treasures and maximize the values of reached treasures.

References

Van Moffaert, Nowe. Multi-Objective Reinforcement Learning using Sets of Pareto Dominating Policies

Inherits from *ReLe::DenseMDP*

Public Functions

`DeepSeaTreasure`()
Constructor.

virtual void **`step`**(**const** *FiniteAction* &*action*, *DenseState* &*nextState*, Reward &*reward*) **See** *Environment::step*

virtual void **`getInitialState`**(*DenseState* &*state*) **See** *Environment::getInitialState*

class `ReLe::GaussianRewardMDP`
This class implements a MDP with a Gaussian reward at each step.

Inherits from *ReLe::ContinuousMDP*

Public Functions

`GaussianRewardMDP`(unsigned int *dimension*, double *mu* = 0.0, double *sigma* = 1.0, double *gamma* = 0.9, unsigned int *horizon* = 50)
Constructor.

Parameters

- `dimension`: MDP dimension
- `mu`: mean of the reward Gaussian distribution
- `sigma`: standard deviation of the reward Gaussian distribution
- `gamma`: MDP discount factor

- `horizon`: MDP horizon

GaussianRewardMDP(arma::mat &*A*, arma::mat &*B*, arma::vec &*mu*, arma::mat &*sigma*, double *gamma* = 0.9, unsigned int *horizon* = 50)
Constructor.

Parameters

- `A`: initialization matrix
- `B`: initialization matrix
- `mu`: mean of the reward Gaussian distribution
- `sigma`: standard deviation of the reward Gaussian distribution
- `gamma`: MDP discount factor
- `horizon`: MDP horizon

virtual void **step**(**const** *DenseAction* &*action*, *DenseState* &*nextState*, Reward &*reward*) **See** *Environment::step*

virtual void **getInitialState**(*DenseState* &*state*) **See** *Environment::getInitialState*

class `ReLe::LQR`
This class implements a Linear-Quadratic Regulator. This task aims to minimize the undesired deviations from nominal values of some controller settings in control problems.

References

Parisi, Pirotta, Smacchia, Bascetta, Restelli. Policy gradient approaches for multi-objective sequential decision making. IJCNN 2014

Inherits from *ReLe::ContinuousMDP*

Public Functions

LQR(unsigned int *dimension*, unsigned int *reward_dimension*, S0Type *type* = FIXED, double *eps* = 0.1, double *gamma* = 0.9, unsigned int *horizon* = 50)
Constructor.

Parameters

- `dimension`: MDP dimension
- `reward_dimension`: reward dimension
- `eps`:
- `gamma`: MDP discount factor
- `horizon`: MDP horizon

LQR(arma::mat &*A*, arma::mat &*B*, std::vector<arma::mat> &*Q*, std::vector<arma::mat> &*R*, S0Type *type* = FIXED, double *gamma* = 0.9, unsigned int *horizon* = 50)
Constructor.

Parameters

- `A`: initialization matrix
- `B`: initialization matrix
- `Q`:

- `R`: reward matrix
- `gamma`: MDP discount factor
- `horizon`: MDP horizon

virtual void **`step`**(**const** *DenseAction* &*action*, *DenseState* &*nextState*, Reward &*reward*) **See** *Environment::step*

virtual void **`getInitialState`**(*DenseState* &*state*) **See** *Environment::getInitialState*

void **`setInitialState`**(arma::vec &*initialState*)
Setter. Set the initial state

Parameters

- `initialState`: initial state

class `ReLe::MountainCar`
This class implements the Mountain Car environment. In this problem a car is placed on a valley between two hills and has to reach the top of one of them. Unfortunately, it is not able to do so by only accelerating and, therefore, it has to accelerate in the opposite direction to use the slope of the other hill to acquire inertia that may allow it to reach the goal.

References

Sutton, Barto. Reinforcement Learning an introduction

Inherits from *ReLe::DenseMDP*

Public Types

enum `ConfigurationsLabel`
Type of configuration obtained from different experiments in respective articles.

Values:

`Sutton`

`Klein`

`Random`

Public Functions

`MountainCar`(*ConfigurationsLabel label* = Sutton, double *initialPosition* = -0.5, double *initialVelocity* = 0)
Constructor.

Parameters

- `label`: configuration type

virtual void **`step`**(**const** *FiniteAction* &*action*, *DenseState* &*nextState*, Reward &*reward*) **See** *Environment::step*

virtual void **`getInitialState`**(*DenseState* &*state*) **See** *Environment::getInitialState*

class ReLe::**MultiHeat**

This class implements the Multi Heat problem. The aim of this problem is to find the optimal heating policy according to environmental conditions and other criterion of optimality.

References

Pirotta, Manganini, Piroddi, Prandini, Restelli. A particle-based policy for the optimal control of Markov decision processes. CDC, 2014.

Inherits from *ReLe::DenseMDP*

Public Functions

MultiHeat()

Constructor.

MultiHeat(MultiHeatSettings &*config*)

Constructor.

Parameters

- config: the initial settings

virtual void **step**(**const** *FiniteAction* &*action*, *DenseState* &*nextState*, Reward &*reward*) **See** *Environment::step*

virtual void **getInitialState**(*DenseState* &*state*) **See** *Environment::getInitialState*

void **setCurrentState**(*DenseState* &*state*)

Set the current state.

Parameters

- state: the current state

class ReLe::**NLS**

This class implements the *NLS* problem. This problem is a two-dimensional MDP where the aim is to let a robot reach a goal state.

References

Vlassis, Toussaint, Kontes, Piperidis. Learning Model-free Robot Control by a Monte Carlo EM Algorithm

Inherits from *ReLe::ContinuousMDP*

Public Functions

NLS()

Constructor.

NLS(NLSSettings &*config*)

Constructor.

Parameters

- config: the initial settings

virtual void **step**(**const** *DenseAction* &*action*, *DenseState* &*nextState*, Reward &*reward*) **See** *Environment::step*

virtual void **getInitialState**(*DenseState* &*state*) **See** *Environment::getInitialState*

const NLSSettings &**getSettings**() **const** **See** *Environment::getSettings*

class ReLe::**Portfolio**

This class implements the *Portfolio* problem. The aim of this problem is to maximize the expected utility of a *Portfolio* in a financial market. For further information see here.

References

Di Castro, Tamar, Mannor. Policy gradients with variance related risk criteria. ICML 2012

Inherits from *ReLe::DenseMDP*

Public Functions

Portfolio()

Constructor.

Portfolio(PortfolioSettings &*config*)

Constructor.

Parameters

- `config`: the initial settings

virtual void **step**(**const** *FiniteAction* &*action*, *DenseState* &*nextState*, Reward &*reward*) **See** *Environment::step*

virtual void **getInitialState**(*DenseState* &*state*) **See** *Environment::getInitialState*

const PortfolioSettings &**getSettings**() **const** **See** *Environment::getSettings*

class ReLe::**Pursuer**

This class implements the *Pursuer* problem. The aim of this problem is to find the optimal policy to let multiple robots detect a mobile evader in an indoor environment.

Inherits from *ReLe::ContinuousMDP*

Public Functions

Pursuer()

Constructor

virtual void **step**(**const** *DenseAction* &*action*, *DenseState* &*nextState*, Reward &*reward*) **See** *Environment::step*

virtual void **getInitialState**(*DenseState* &*state*) **See** *Environment::getInitialState*

class ReLe::**Rocky**

This class implements an environment similar to the *Pursuer* problem. In this task, *Rocky* has to reach the chicken moving in a continuous MDP.

Inherits from *ReLe::ContinuousMDP*

Public Functions

Rocky()
Constructor.

virtual void **step**(**const** *DenseAction* &*action*, *DenseState* &*nextState*, Reward &*reward*) **See** *Environment::step*

virtual void **getInitialState**(*DenseState* &*state*) **See** *Environment::getInitialState*

class ReLe::**Segway**
This class implements a continuous MDP problem where a segway has to be controlled in order to balance it.

References

Tesi.

Inherits from *ReLe::ContinuousMDP*

Public Functions

Segway()
Constructor.

Segway(SegwaySettings &*config*)
Constructor.

Parameters

- config: the initial settings

virtual void **step**(**const** *DenseAction* &*action*, *DenseState* &*nextState*, Reward &*reward*) **See** *Environment::step*

virtual void **getInitialState**(*DenseState* &*state*) **See** *Environment::getInitialState*

const SegwaySettings &**getSettings**() **const** **See** *Environment::getSettings*

class ReLe::**ShipSteering**
This class implements the Ship Steering problem. The aim of this problem is to let a ship pass through a gate when starting from a random position and moving at constant speed. For further information see here.

References

Ghavamzadeh, Mahadevan. Hierarchical Policy Gradient Algorithms. ICML 2013

Inherits from *ReLe::ContinuousMDP*

Public Functions

ShipSteering(bool *small* = true)
Constructor.

Parameters

- small: field size

virtual void **step**(**const** *DenseAction* &*action*, *DenseState* &*nextState*, Reward &*reward*) **See** *Environment::step*

virtual void **getInitialState**(*DenseState* &*state*) **See** *Environment::getInitialState*

class ReLe::**DiscreteActionSwingUp**

This class implements a task where a pendulum has to be swinged controlling its rotation and trying to reach the maximum height without letting it fall.

References

Doya. Reinforcement Learning In Continuous Time and Space. Neural computation 12.1 (2000): 219-245.

Inherits from *ReLe::DenseMDP*

Public Functions

DiscreteActionSwingUp()

Constructor.

DiscreteActionSwingUp(SwingUpSettings &*config*)

Constructor.

Parameters

- `config`: the initial settings

void **step**(**const** *FiniteAction* &*action*, *DenseState* &*nextState*, Reward &*reward*) **See** *Environment::step*

void **getInitialState**(*DenseState* &*state*) **See** *Environment::getInitialState*

const SwingUpSettings &**getSettings**() **const** **See** *Environment::getSettings*

class ReLe::**TaxiFuel**

This class implements the Taxi Fuel problem. The aim of this problem is to find an optimal policy for a taxi cab in order to let it be able to transport passengers in the desired locations without running out of fuel.

References

Dietterich. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. JAIR

Inherits from *ReLe::DenseMDP*

Public Functions

TaxiFuel()

Constructor.

virtual void **step**(**const** *FiniteAction* &*action*, *DenseState* &*nextState*, Reward &*reward*) **See** *Environment::step*

virtual void **getInitialState**(*DenseState* &*state*) **See** *Environment::getInitialState*

std::vector<arma::vec2> **getLocations**()

Return the position of the special locations of the problem.

Return vector of positions

class `ReLe::UnderwaterVehicle`

This class implements the Underwater Vehicle problem. The task of this problem is to control the speed an underwater vehicle in a submarine environment modeling the complex dynamics of objects moving in fluids.

References

Hafner, Riedmiller. Reinforcement learning in feedback control. Challenges and benchmarks from technical process control. Machine Learning

Inherits from *ReLe::DenseMDP*

Public Functions

UnderwaterVehicle()

Constructor.

UnderwaterVehicle(UWVSettings &*config*)

Constructor.

Parameters

- `config`: the initial settings

virtual void **step**(**const** *FiniteAction* &*action*, *DenseState* &*nextState*, Reward &*reward*) **See** *Environment::step*

virtual void **getInitialState**(*DenseState* &*state*) **See** *Environment::getInitialState*

const UWVSettings &**getSettings**() **const** **See** *Environment::getSettings*

class `ReLe::UnicyclePolar`

This class implements the Unicycle problem. The aim of this problem is to control a unicycle in order to let it stay balanced.

$$\hat{e}(t) = [x(t) - x_g; y(t) - y_g; \theta(t) - \theta_g]^T$$

$$e(t) = \begin{bmatrix} e_x(t) \\ e_y(t) \\ e_\theta(t) \end{bmatrix} = \begin{bmatrix} cos\theta_g & sin\theta_g & 0 \\ -sin\theta_g & cos\theta_g & 0 \\ 0 & 0 & 1 \end{bmatrix} \hat{e}(t)$$

$$\rho = \sqrt{(e_x^2 + e_y^2)}$$

$$\gamma = atan2(e_y, e_x) - e_\theta + \pi$$

$$\delta = \gamma + e_\theta$$

Optimal control law:

$$v = k_1 \rho cos\gamma$$

$$w = k_2\gamma + k_1 \sin\gamma \cos\gamma(\gamma + k_3\delta)/\gamma$$

References

Master Thesis

Stabilized Feedback Control of Unicycle Mobile Robots

Inherits from *ReLe::ContinuousMDP*

Public Functions

UnicyclePolar()
Constructor.

UnicyclePolar(UnicyclePolarSettings &*config*)
Constructor.

Parameters

- `config`: the initial settings

virtual void **step**(**const** *DenseAction* &*action*, *DenseState* &*nextState*, Reward &*reward*) **See** *Environment::step*

virtual void **getInitialState**(*DenseState* &*state*) **See** *Environment::getInitialState*

const UnicyclePolarSettings &**getSettings**() **const** **See** *Environment::getSettings*

Generators

class `ReLe::FiniteGenerator`
This class contains function to generate finite MDP.

Subclassed by *ReLe::GridWorldGenerator*, *ReLe::SimpleChainGenerator*

Public Functions

FiniteMDP **getMDP**(double *gamma*)
Return the finite MDP with transition probabilities, reward and reward variance matrices.

void **printMatrices**()
Print transition probabilities, reward and reward variance matrices.

Protected Attributes

arma::cube **P**
Transition probability matrix.

arma::cube **R**
Reward matrix.

arma::cube **Rsigma**
Reward variance matrix.

size_t **stateN**
Number of states.

unsigned int **actionN**
Number of actions.

class ReLe::**SimpleChainGenerator**
This class contains function to generate a simple Markov chain.

Inherits from *ReLe::FiniteGenerator*

Public Functions

SimpleChainGenerator()
Constructor.

void **generate**(std::size_t *size*, std::size_t *goalState*)
Initialize the Markov chain.

Parameters

- size: the size of the Markov chain
- goalState: goal state index

void **setP**(double *p*)
Setter. Set probability of success of actions.

Parameters

- p: probability of success of actions

void **setRgoal**(double *rgoal*)
Setter. Set reward in case of reaching goal state.

Parameters

- rgoal: reward when reaching goal state

class ReLe::**GridWorldGenerator**
This class contains function to generate a grid world.

Inherits from *ReLe::FiniteGenerator*

Public Functions

GridWorldGenerator()
Constructor.

void **load**(**const** std::string &*path*)
Load a grid world from a text file.

Parameters

- path: path of the text file

void **setP**(double *p*)
Setter. Set probability of success of actions.

Parameters

- p: probability of success of actions

void **setRfall**(double *rfall*)
Setter. Set reward in case of falling out from the grid world.

Parameters

- `rfall`: reward in case of falling

void **setRgoal**(double *rgoal*)
Setter. Set reward in case of reaching goal state.

Parameters

- `rgoal`: reward when reaching goal state

void **setRstep**(double *rstep*)
Setter. Set the reward obtained at each step.

Parameters

- `rstep`: reward at each step

Algorithms

Step Rules

Learning Rate

template <**class** ActionC, **class** StateC>
class `ReLe::LearningRate_`
This class implement a learning rate, with eventually decay rules The learning rate can be action and state dependent

Subclassed by *ReLe::ConstantLearningRate_< ActionC, StateC >*, *ReLe::DecayingLearningRate_< ActionC, StateC >*

Public Functions

virtual double **operator()**(StateC *x*, ActionC *u*) = 0
Computes the learning rate

virtual void **reset**() = 0
Resets the learning rate to it's original value

virtual std::string **print**() = 0
Writes the learning rate as a string

virtual **~LearningRate_**()
Destructor.

template <**class** ActionC, **class** StateC>
class `ReLe::ConstantLearningRate_`
Implementation of a constant learning rate

Inherits from *ReLe::LearningRate_< ActionC, StateC >*

Public Functions

ConstantLearningRate_(double *alpha*)
Constructor.

Parameters

- `alpha`: the value for the learning rate

virtual double **`operator()`**(StateC *x*, ActionC *u*)
Computes the learning rate

virtual void **`reset`**()
Resets the learning rate to it's original value

virtual std::string **`print`**()
Writes the learning rate as a string

template <**class** ActionC, **class** StateC>
class `ReLe::DecayingLearningRate_`
Implementation of a simple decaying learning rate. The learning rate follows the rule:

$$\alpha(t) = \min\left(\alpha_{min}, \frac{\alpha(t_0)}{t^{\omega}}\right)$$

Inherits from *ReLe::LearningRate_< ActionC, StateC >*

Public Functions

`DecayingLearningRate_`(double *initialAlpha*, double *omega* = 1.0, double *minAlpha* = 0.0)
Constructor.

Parameters

- `initialAlpha`: the initial value for the learning rate
- `omega`: the exponent used to weight the time decay. $\omega \in (0, 1]$

virtual double **`operator()`**(StateC *x*, ActionC *u*)
Computes the learning rate

virtual void **`reset`**()
Resets the learning rate to it's original value

virtual std::string **`print`**()
Writes the learning rate as a string

Gradient Step

class `ReLe::GradientStep`
This interface implements a generic gradient step rule, i.e. a rule to select a parameter update, given the current parameters gradient (and optionally other information).

Subclassed by *ReLe::AdaptiveGradientStep*, *ReLe::ConstantGradientStep*, *ReLe::VectorialGradientStep*

Public Functions

virtual arma::vec **`operator()`**(**const** arma::vec &*gradient*) = 0
Computes the new parameters step assuming identity metric.

Return the delta to apply on the parameters

Parameters

- `gradient`: the actual gradient

virtual arma::vec **operator()**(**const** arma::vec &*gradient*, **const** arma::vec &*nat_gradient*) = 0
Computes the new parameters step using the gradient and the natural gradient.

Return the delta to apply on the parameters

Parameters

- `gradient`: the vanilla gradient
- `nat_gradient`: the natural gradient

virtual arma::vec **operator()**(**const** arma::vec &*gradient*, **const** arma::mat &*metric*, bool *inverse*) = 0
Computes the new parameters step assuming the given metric.

Return the delta to apply on the parameters

Parameters

- `gradient`: the gradient direction
- `metric`: a predefined space metric
- `inverse`: whether the metric parameter is the inverse (M^{-1}) one or not (M)

virtual void **reset**() = 0
This function is called in order to reset the internal state of the class

Protected Functions

arma::vec **computeGradientInMetric**(**const** arma::vec &*gradient*, **const** arma::mat &*metric*, bool *inverse*)
Default function for computing the product:

$$M^{-1}\nabla_\theta J$$

when $M = \mathcal{F}$, the Fisher Information Matrix, this function computes the natural gradient

Return the product $M^{-1}\nabla_\theta J$

Parameters

- `gradient`: the vanilla gradient
- `metric`: a predefined space metric whether the metric parameter is the inverse (M^{-1}) one or not (M)

class `ReLe::ConstantGradientStep`
Basic step rule. The step is very simple:

$$\Delta\theta = \alpha\nabla_\theta J$$

Inherits from *ReLe::GradientStep*

Public Functions

ConstantGradientStep(double *alpha*)
Constructor.

Parameters

- `alpha`: the constant factor to multiply to the gradient.

virtual arma::vec **`operator()`**(**const** arma::vec &*gradient*)
Computes the new parameters step assuming identity metric.

Return the delta to apply on the parameters

Parameters

- `gradient`: the actual gradient

virtual arma::vec **`operator()`**(**const** arma::vec &*gradient*, **const** arma::vec &*nat_gradient*)
Computes the new parameters step using the gradient and the natural gradient.

Return the delta to apply on the parameters

Parameters

- `gradient`: the vanilla gradient
- `nat_gradient`: the natural gradient

virtual arma::vec **`operator()`**(**const** arma::vec &*gradient*, **const** arma::mat &*metric*, bool *inverse*)
Computes the new parameters step assuming the given metric.

Return the delta to apply on the parameters

Parameters

- `gradient`: the gradient direction
- `metric`: a predefined space metric
- `inverse`: whether the metric parameter is the inverse (M^{-1}) one or not (M)

void **`reset`**()
This function is called in order to reset the internal state of the class

class `ReLe::VectorialGradientStep`
A constant vectorial step rule. The step is very simple:

$$\Delta\theta = \alpha \odot \nabla_\theta J$$

with:

$$\alpha, \theta \in \mathbb{R}^n$$

Inherits from *ReLe::GradientStep*

Public Functions

`VectorialGradientStep`(**const** arma::vec &*alpha*)
Constructor.

Parameters

- `alpha`: the vector of factors to multiply to each component of the gradient.

virtual arma::vec **`operator()`**(**const** arma::vec &*gradient*)
Computes the new parameters step assuming identity metric.

Return the delta to apply on the parameters

Parameters

- `gradient`: the actual gradient

virtual arma::vec **`operator()`**(**const** arma::vec &*gradient*, **const** arma::vec &*nat_gradient*)
Computes the new parameters step using the gradient and the natural gradient.

Return the delta to apply on the parameters

Parameters

- `gradient`: the vanilla gradient
- `nat_gradient`: the natural gradient

virtual arma::vec **`operator()`**(**const** arma::vec &*gradient*, **const** arma::mat &*metric*, bool *inverse*)
Computes the new parameters step assuming the given metric.

Return the delta to apply on the parameters

Parameters

- `gradient`: the gradient direction
- `metric`: a predefined space metric
- `inverse`: whether the metric parameter is the inverse (M^{-1}) one or not (M)

void **`reset`**()
This function is called in order to reset the internal state of the class

class `ReLe::AdaptiveGradientStep`
This class implements a basic adaptative gradient step. Instead of moving of a step proportional to the gradient, takes a step limited by a given metric. If no metric is given, the identity matrix is used.

The step rule is:

$$\Delta\theta = \underset{\Delta\vartheta}{argmax}\Delta\vartheta^t\nabla_\theta J$$

$$s.t. : \Delta\vartheta^T M\Delta\vartheta \leq \varepsilon$$

References

Neumann. Lecture Notes

Inherits from *ReLe::GradientStep*

Public Functions

`AdaptiveGradientStep`(double *eps*)
Constructor.

Parameters

- `eps`: the maximum allowed size for the step.

virtual arma::vec **`operator()`**(**const** arma::vec &*gradient*)
Computes the new parameters step assuming identity metric.

Return the delta to apply on the parameters

Parameters

- `gradient`: the actual gradient

virtual arma::vec **operator()** (**const** arma::vec &*gradient*, **const** arma::vec &*nat_gradient*)
Computes the new parameters step using the gradient and the natural gradient.

Return the delta to apply on the parameters

Parameters

- `gradient`: the vanilla gradient
- `nat_gradient`: the natural gradient

virtual arma::vec **operator()** (**const** arma::vec &*gradient*, **const** arma::mat &*metric*, bool *inverse*)
Computes the new parameters step assuming the given metric.

Return the delta to apply on the parameters

Parameters

- `gradient`: the gradient direction
- `metric`: a predefined space metric
- `inverse`: whether the metric parameter is the inverse (M^{-1}) one or not (M)

void **reset** ()
This function is called in order to reset the internal state of the class

Batch

class `ReLe::FQI`
This class implements the Fitted Q-Iteration algorithm. This algorithm is an off-policy batch algorithm that works with finite action spaces. It exploits the Bellman operator to build a dataset of Q-values from which a regressor is trained.

References

Ernst, Geurts, Wehenkel. Tree-Based Batch Mode Reinforcement Learning

Inherits from ReLe::BatchTDAgent< DenseState >

Subclassed by *ReLe::DoubleFQI*

Public Functions

FQI (BatchRegressor &*QRegressor*, double *epsilon*)
Constructor.

Parameters

- `QRegressor`: the regressor
- `nStates`: the number of states
- `nActions`: the number of actions
- `epsilon`: coefficient used to check whether to stop the training

virtual void **init** (*Dataset*<*FiniteAction*, *DenseState*> &*data*)
This method setup the dataset to be used in the learning process. Must be implemented. Is called by the *BatchCore* as the first step of the learning process.

Parameters

- `data`: the dataset to be used for learning

virtual void **step**()
This method implement a step of the learning process trough the dataset. Must be implemented. Is called by the *BatchCore* until the algorithm converges or the maximum number of iteration is reached.

virtual *AgentOutputData* ***getAgentOutputData**()
This method is used to log agent step informations. Can be overloaded to return information that can be processed by the logger. By default a null pointer is returned, which means that no data will be logged.

Return the data to be logged from the agent at the current step.

virtual *AgentOutputData* ***getAgentOutputDataEnd**()
This method is used to log agent informations at episode end. Can be overloaded to return information that can be processed by the logger. By default a null pointer is returned, which means that no data will be logged.

Return the data to be logged from the agent at the episode end.

class ReLe::**DoubleFQI**
This class implements a version of Fitted Q-iteration (*FQI*) that exploits the Double Estimator, as done in Double Q-Learning, using two regressors. One of the regressor is used to select the action with the highest action-value for the first regressor and the other is used to compute the value of the selected actions. Being a modified version of Fitted Q-Iteration, this algorithms deals only with finite action spaces.

Inherits from *ReLe::FQI*

Public Functions

DoubleFQI(BatchRegressor &*QRegressorA*, BatchRegressor &*QRegressorB*, double *epsilon*, bool *shuffle* = false)
Constructor.

Parameters

- QRegressorA: the first regressor
- QRegressorB: the second regressor
- nStates: the number of states
- nActions: the number of actions
- epsilon: coefficient used to check whether to stop the training
- shuffle: if true, each regressor takes a different half of the dataset at each iteration

virtual void **step**()
This method implement a step of the learning process trough the dataset. Must be implemented. Is called by the *BatchCore* until the algorithm converges or the maximum number of iteration is reached.

class ReLe::**DoubleFQIEnsemble**
This class implements an ensemble of regressors to be used for the modified version of Fitted Q-Iteration algorithm that uses the Double Estimator.

Inherits from *ReLe::Ensemble_< InputC, OutputC, denseOutput >*

Public Functions

DoubleFQIEnsemble(BatchRegressor &*QRegressorA*, BatchRegressor &*QRegressorB*)
Constructor.

Parameters

- `QRegressorA`: the first regressor of the ensemble
- `QRegressorB`: the second regressor of the ensemble

Warning: doxygenclass: Cannot find class "ReLe::W_FQI" in doxygen xml output for project "ReLe" from directory: doxyxml/

class `ReLe::LSPI`

This class implements the Least-Squares *Policy* Iteration (*LSPI*) algorithm. This algorithm is an off-policy batch algorithm that exploits the action-values approximation done by the LSTDQ algorithm to form an approximate policy-iteration algorithm.

References

Lagoudakis, Parr. Least-Squares Policy Iteration

Inherits from ReLe::BatchTDAgent< DenseState >

Public Functions

LSPI(LinearApproximator &*Q*, double *epsilon*)

Constructor.

Parameters

- `phi`: the features to be used for approximation
- `epsilon`: coefficient used to check whether to stop the training

virtual void **init**(*Dataset*<*FiniteAction*, *DenseState*> &*data*)

This method setup the dataset to be used in the learning process. Must be implemented. Is called by the *BatchCore* as the first step of the learning process.

Parameters

- `data`: the dataset to be used for learning

virtual void **step**()

This method implement a step of the learning process trough the dataset. Must be implemented. Is called by the *BatchCore* until the algorithm converges or the maximum number of iteration is reached.

virtual *AgentOutputData* ***getAgentOutputData**()

This method is used to log agent step informations. Can be overloaded to return information that can be processed by the logger. By default a null pointer is returned, which means that no data will be logged.

Return the data to be logged from the agent at the current step.

virtual *AgentOutputData* ***getAgentOutputDataEnd**()

This method is used to log agent informations at episode end. Can be overloaded to return information that can be processed by the logger. By default a null pointer is returned, which means that no data will be logged.

Return the data to be logged from the agent at the episode end.

Temporal Difference

class `ReLe::FiniteTD`

This interface is the basic interface for all Finite TD algorithms. A finite TD algorithm is a Q-table based algorithm, this means that both action and state space are finite.

Inherits from *ReLe::Agent< FiniteAction, FiniteState >*

Subclassed by *ReLe::Q_Learning*, *ReLe::R_Learning*, *ReLe::SARSA*, *ReLe::SARSA_lambda*

Public Functions

FiniteTD (ActionValuePolicy<*FiniteState*> &*policy*, LearningRate &*alpha*)
Constructor.

Parameters

- `policy`: the policy to be used by the algorithm
- `alpha`: the learning rate to be used by the algorithm

virtual void **endEpisode** ()
This method is called if an episode ends after reaching the maximum number of iterations. Must be implemented. Normally this method contains the learning algorithm.

virtual *AgentOutputData* ***getAgentOutputDataEnd** ()
This method is used to log agent informations at episode end. Can be overloaded to return information that can be processed by the logger. By default a null pointer is returned, which means that no data will be logged.

Return the data to be logged from the agent at the episode end.

void **resetLearningRate** ()
This method resets the algorithm learning rate.

Protected Functions

virtual void **init** ()
Implementation of the init method. Initializes the Q-table and setup the policy to use the Q-Table.

Protected Attributes

arma::mat **Q**
Action-value function.

size_t **x**
previous state

unsigned int **u**
previous action

LearningRate &**alpha**
learning rate

ActionValuePolicy<*FiniteState*> &**policy**
algorithm policy

class `ReLe::LinearTD`
This interface is the basic interface for all linear TD algorithms. A linear TD algorithm is an algorithm using finite action spaces and dense state spaces where the approximation of the action-values is performed with linear approximation.

Inherits from *ReLe::Agent< FiniteAction, DenseState >*

Subclassed by ReLe::DenseSARSA, *ReLe::LinearGradientSARSA*

Public Functions

LinearTD (Features &*phi*, ActionValuePolicy<*DenseState*> &*policy*, LearningRateDense &*alpha*)
Constructor.

Parameters

- `phi`: the features to be used for linear approximation of the state space
- `policy`: the policy to be used by the algorithm
- `alpha`: the learning rate to be used by the algorithm

virtual void **endEpisode** ()
This method is called if an episode ends after reaching the maximum number of iterations. Must be implemented. Normally this method contains the learning algorithm.

virtual *AgentOutputData* ***getAgentOutputDataEnd** ()
This method is used to log agent informations at episode end. Can be overloaded to return information that can be processed by the logger. By default a null pointer is returned, which means that no data will be logged.

Return the data to be logged from the agent at the episode end.

void **resetLearningRate** ()
This method resets the algorithm learning rate.

Protected Functions

virtual void **init** ()
Implementation of the init method. Initializes the Q-function approximator, and setup the policy to use the Q-Table.

Protected Attributes

LinearApproximator **Q**
Linear approximated action-value function.

DenseState **x**
previous state

unsigned int **u**
previous action

LearningRateDense &**alpha**
learning rate

ActionValuePolicy<*DenseState*> &**policy**
algorithm policy

class `ReLe::SARSA`
This class implements the *SARSA* algorithm. This algorithm is an on-policy temporal difference algorithm. Can only work on Finite MDP, i.e. with finite action and state space.

References

Rummery, Niranjan. On-line Q-learning using connectionist systems

Sutton, Barto. Reinforcement Learning: An Introduction (chapter 6.4)

Inherits from *ReLe::FiniteTD*

Public Functions

SARSA(ActionValuePolicy<*FiniteState*> &*policy*, LearningRate &*alpha*)
Constructor.

Parameters

- `policy`: the policy to be used by the algorithm
- `alpha`: the learning rate to be used by the algorithm

virtual void **initEpisode**(**const** *FiniteState* &*state*, *FiniteAction* &*action*)
This method is called at the beginning of each learning episode. Must be implemented. Normally this method contains the algorithm initialization.

Parameters

- `state`: the initial environment state
- `action`: the action selected by the agent in the initial state

virtual void **sampleAction**(**const** *FiniteState* &*state*, *FiniteAction* &*action*)
This method is used to sample an action in test episodes. Must be implemented. Normally, this method is trivial, as it just sample an action from a policy.

Parameters

- `state`: the current environment state
- `action`: the action selected by the agent in the current state

virtual void **step**(**const** Reward &*reward*, **const** *FiniteState* &*nextState*, *FiniteAction* &*action*)
This method is used during each learning step. Must be implemented. Normally this method contains the learning algorithm, for step-based agents, or data collection for episode-based agents.

Parameters

- `reward`: the reward achieved in the previous learning step.
- `nextState`: the state reached after the previous learning state i.e. the current state
- `action`: the action selected by the agent in the current state

virtual void **endEpisode**(**const** Reward &*reward*)
This method is called if an episode ends in a terminal state. Must be implemented. Normally this method contains the learning algorithm.

Parameters

- `reward`: the reward achieved after reaching the terminal state.

class `ReLe::SARSA_lambda`
This class implements the *SARSA*(λ) algorithm. Differently from the *SARSA* algorithm, this can use eligibility trace. With $\lambda = 0$, this algorithm is equivalent to plain *SARSA*, but is less efficient, as it stores the eligibility vector. This algorithm is an on-policy temporal difference algorithm. Can only work on Finite MDP, i.e. with finite action and state space.

References

Rummery, Niranjan. On-line Q-learning using connectionist systems

Sutton, Barto. Reinforcement Learning: An Introduction (chapter 7.5)

Inherits from *ReLe::FiniteTD*

Public Functions

SARSA_lambda (ActionValuePolicy<*FiniteState*> &*policy*, LearningRate &*alpha*, bool *accumulating*)
Constructor.

Parameters

- `policy`: the policy to be used by the algorithm
- `alpha`: the learning rate to be used by the algorithm
- `accumulating`: whether to use accumulating trace or replacing ones.

virtual void **initEpisode** (**const** *FiniteState* &*state*, *FiniteAction* &*action*)
This method is called at the beginning of each learning episode. Must be implemented. Normally this method contains the algorithm initialization.

Parameters

- `state`: the initial environment state
- `action`: the action selected by the agent in the initial state

virtual void **sampleAction** (**const** *FiniteState* &*state*, *FiniteAction* &*action*)
This method is used to sample an action in test episodes. Must be implemented. Normally, this method is trivial, as it just sample an action from a policy.

Parameters

- `state`: the current environment state
- `action`: the action selected by the agent in the current state

virtual void **step** (**const** Reward &*reward*, **const** *FiniteState* &*nextState*, *FiniteAction* &*action*)
This method is used during each learning step. Must be implemented. Normally this method contains the learning algorithm, for step-based agents, or data collection for episode-based agents.

Parameters

- `reward`: the reward achieved in the previous learning step.
- `nextState`: the state reached after the previous learning state i.e. the current state
- `action`: the action selected by the agent in the current state

virtual void **endEpisode** (**const** Reward &*reward*)
This method is called if an episode ends in a terminal state. Must be implemented. Normally this method contains the learning algorithm.

Parameters

- `reward`: the reward achieved after reaching the terminal state.

class `ReLe::Q_Learning`
This class implements the tabular Q-learning algorithm. This algorithm is an off-policy temporal difference algorithm. Can only work on finite MDP, i.e. with both finite action and state space.

References

Watkins, Dayan. Q-learning

Inherits from *ReLe::FiniteTD*

Subclassed by *ReLe::DoubleQ_Learning*, *ReLe::WQ_Learning*

Public Functions

virtual void **initEpisode**(**const** *FiniteState* &*state*, *FiniteAction* &*action*)
This method is called at the beginning of each learning episode. Must be implemented. Normally this method contains the algorithm initialization.

Parameters

- `state`: the initial environment state
- `action`: the action selected by the agent in the initial state

virtual void **sampleAction**(**const** *FiniteState* &*state*, *FiniteAction* &*action*)
This method is used to sample an action in test episodes. Must be implemented. Normally, this method is trivial, as it just sample an action from a policy.

Parameters

- `state`: the current environment state
- `action`: the action selected by the agent in the current state

virtual void **step**(**const** Reward &*reward*, **const** *FiniteState* &*nextState*, *FiniteAction* &*action*)
This method is used during each learning step. Must be implemented. Normally this method contains the learning algorithm, for step-based agents, or data collection for episode-based agents.

Parameters

- `reward`: the reward achieved in the previous learning step.
- `nextState`: the state reached after the previous learning state i.e. the current state
- `action`: the action selected by the agent in the current state

virtual void **endEpisode**(**const** Reward &*reward*)
This method is called if an episode ends in a terminal state. Must be implemented. Normally this method contains the learning algorithm.

Parameters

- `reward`: the reward achieved after reaching the terminal state.

class `ReLe::DoubleQ_Learning`
This class implements the Double Q-Learning algorithm. This algorithm is an off-policy temporal difference algorithm that tries to solve the well-known overestimation problem suffered by Q-Learning. More precisely, this algorithm stores two Q-tables and uses one of them to select the action that maximizes the Q-value, but uses the value of that action stored in the other Q-Table. This technique has been proved to have negative bias (as opposed to Q-Learning that has a positive bias) which can help to avoid incremental approximation error caused by overestimations. It can only work on finite MDP, i.e. with both finite action and state space.

References

Van Hasselt. Double Q-Learning

Inherits from *ReLe::Q_Learning*

Public Functions

virtual void **initEpisode**(**const** *FiniteState* &*state*, *FiniteAction* &*action*)
This method is called at the beginning of each learning episode. Must be implemented. Normally this method contains the algorithm initialization.

Parameters

- `state`: the initial environment state
- `action`: the action selected by the agent in the initial state

virtual void **sampleAction**(**const** *FiniteState* &*state*, *FiniteAction* &*action*)
This method is used to sample an action in test episodes. Must be implemented. Normally, this method is trivial, as it just sample an action from a policy.

Parameters

- `state`: the current environment state
- `action`: the action selected by the agent in the current state

virtual void **step**(**const** Reward &*reward*, **const** *FiniteState* &*nextState*, *FiniteAction* &*action*)
This method is used during each learning step. Must be implemented. Normally this method contains the learning algorithm, for step-based agents, or data collection for episode-based agents.

Parameters

- `reward`: the reward achieved in the previous learning step.
- `nextState`: the state reached after the previous learning state i.e. the current state
- `action`: the action selected by the agent in the current state

virtual void **endEpisode**(**const** Reward &*reward*)
This method is called if an episode ends in a terminal state. Must be implemented. Normally this method contains the learning algorithm.

Parameters

- `reward`: the reward achieved after reaching the terminal state.

class `ReLe::WQ_Learning`
This class implements the Weighted Q-Learning algorithm. This algorithm is an off-policy temporal difference algorithm that, as Double Q-Learning, tries to solve the well-known overestimation problem suffered by Q-Learning. This algorithm computes an estimate of the maximum action-value approximating it as weighted sum of action-values where the weights are the probabilities of the respective action-value to be the maximum. While Q-Learning gives the best results when there is a single action-value that is clearly the maximum and Double Q-Learning gives the best results when all action-values have almost the same value, Weighted Q-Learning has been proved to work well in intermediate cases giving, in general, better approximation than the other two approaches.

References

...

Inherits from *ReLe::Q_Learning*

Public Functions

virtual void **initEpisode**(**const** *FiniteState* &*state*, *FiniteAction* &*action*)
This method is called at the beginning of each learning episode. Must be implemented. Normally this method contains the algorithm initialization.

Parameters

- `state`: the initial environment state
- `action`: the action selected by the agent in the initial state

virtual void **sampleAction**(**const** *FiniteState* &*state*, *FiniteAction* &*action*)
This method is used to sample an action in test episodes. Must be implemented. Normally, this method is trivial, as it just sample an action from a policy.

Parameters

- `state`: the current environment state
- `action`: the action selected by the agent in the current state

virtual void **step**(**const** Reward &*reward*, **const** *FiniteState* &*nextState*, *FiniteAction* &*action*)
This method is used during each learning step. Must be implemented. Normally this method contains the learning algorithm, for step-based agents, or data collection for episode-based agents.

Parameters

- `reward`: the reward achieved in the previous learning step.
- `nextState`: the state reached after the previous learning state i.e. the current state
- `action`: the action selected by the agent in the current state

virtual void **endEpisode**(**const** Reward &*reward*)
This method is called if an episode ends in a terminal state. Must be implemented. Normally this method contains the learning algorithm.

Parameters

- `reward`: the reward achieved after reaching the terminal state.

class `ReLe::R_Learning`
This class implements the tabular R-learning algorithm. This algorithm is an off-policy temporal difference algorithm. Differently from Q-Learning, can learn properly the Q-function in the average reward case. Can only work on finite MDP, i.e. with both finite action and state space.

References

Schwartz. A reinforcement learning method for maximizing undiscounted rewards

Sutton, Barto. Reinforcement Learning: An Introduction (chapter 6.7)

Inherits from *ReLe::FiniteTD*

Public Functions

R_Learning(ActionValuePolicy<*FiniteState*> &*policy*, LearningRate &*alpha*, LearningRate &*beta*)
Constructor.

Parameters

- `policy`: the policy to be used by the algorithm
- `alpha`: the Q-function learning rate to be used by the algorithm
- `beta`: the average expected reward learning rate to be used by the algorithm

virtual void **initEpisode**(**const** *FiniteState* &*state*, *FiniteAction* &*action*)
This method is called at the beginning of each learning episode. Must be implemented. Normally this method contains the algorithm initialization.

Parameters

- `state`: the initial environment state
- `action`: the action selected by the agent in the initial state

virtual void **sampleAction**(**const** *FiniteState* &*state*, *FiniteAction* &*action*)
This method is used to sample an action in test episodes. Must be implemented. Normally, this method is trivial, as it just sample an action from a policy.

Parameters

- `state`: the current environment state
- `action`: the action selected by the agent in the current state

virtual void **step**(**const** Reward &*reward*, **const** *FiniteState* &*nextState*, *FiniteAction* &*action*)
This method is used during each learning step. Must be implemented. Normally this method contains the learning algorithm, for step-based agents, or data collection for episode-based agents.

Parameters

- `reward`: the reward achieved in the previous learning step.
- `nextState`: the state reached after the previous learning state i.e. the current state
- `action`: the action selected by the agent in the current state

virtual void **endEpisode**(**const** Reward &*reward*)
This method is called if an episode ends in a terminal state. Must be implemented. Normally this method contains the learning algorithm.

Parameters

- `reward`: the reward achieved after reaching the terminal state.

virtual *AgentOutputData* ***getAgentOutputDataEnd**()
This method is used to log agent informations at episode end. Can be overloaded to return information that can be processed by the logger. By default a null pointer is returned, which means that no data will be logged.

Return the data to be logged from the agent at the episode end.

class `ReLe::LinearGradientSARSA`
This class implements the linear *SARSA* algorithm. This algorithm is an on-policy temporal difference algorithm. Can only work on Dense MDP, i.e. with finite action and dense state space.

References

Seijen, Sutton. True Online TD(lambda)

Inherits from *ReLe::LinearTD*

Public Functions

LinearGradientSARSA(Features &*phi*, ActionValuePolicy<*DenseState*> &*policy*, LearningRate-Dense &*alpha*)
Constructor.

Parameters

- `phi`: the features to be used for linear approximation of the state space
- `policy`: the policy to be used by the algorithm
- `alpha`: the learning rate to be used by the algorithm

virtual void **initEpisode**(**const** *DenseState* &*state*, *FiniteAction* &*action*)
This method is called at the beginning of each learning episode. Must be implemented. Normally this method contains the algorithm initialization.

Parameters

- `state`: the initial environment state
- `action`: the action selected by the agent in the initial state

virtual void **`sampleAction`**(**const** *DenseState* &*state*, *FiniteAction* &*action*)
This method is used to sample an action in test episodes. Must be implemented. Normally, this method is trivial, as it just sample an action from a policy.

Parameters

- `state`: the current environment state
- `action`: the action selected by the agent in the current state

virtual void **`step`**(**const** Reward &*reward*, **const** *DenseState* &*nextState*, *FiniteAction* &*action*)
This method is used during each learning step. Must be implemented. Normally this method contains the learning algorithm, for step-based agents, or data collection for episode-based agents.

Parameters

- `reward`: the reward achieved in the previous learning step.
- `nextState`: the state reached after the previous learning state i.e. the current state
- `action`: the action selected by the agent in the current state

virtual void **`endEpisode`**(**const** Reward &*reward*)
This method is called if an episode ends in a terminal state. Must be implemented. Normally this method contains the learning algorithm.

Parameters

- `reward`: the reward achieved after reaching the terminal state.

Output Data

class `ReLe::FiniteTDOutput`
This class implements the output data for all Finite TD algorithms. All Finite TD algorithms should use, or extend this class

Inherits from *ReLe::AgentOutputData*

Subclassed by *ReLe::R_LearningOutput*

Public Functions

`FiniteTDOutput`(double *gamma*, **const** std::string &*alpha*, **const** std::string &*policyName*, **const** hyperparameters_map &*policyHPar*, **const** arma::mat &*Q*)
Constructor.

Parameters

- `gamma`: the discount factor
- `alpha`: a string describing the learning rate
- `policyName`: the name of the policy used
- `policyHPar`: the map of the hyperparameters of the policy
- `Q`: the Q-table

virtual void **writeData**(std::ostream &*os*)
Basic method to write plain data.

Parameters

- `os`: output stream in which the data should be logged

virtual void **writeDecoratedData**(std::ostream &*os*)
Basic method to write decorated data, e.g. for printing on screen.

Parameters

- `os`: output stream in which the data should be logged

class `ReLe::LinearTDOutput`
This class implements the output data for linear approximated state TD algorithms. All linear linear approximated state TD algorithms should use, or extend this class

Inherits from *ReLe::AgentOutputData*

Public Functions

LinearTDOutput(double *gamma*, **const** std::string &*alpha*, **const** std::string &*policyName*, **const** hyperparameters_map &*policyHPar*, **const** arma::vec *Qw*)
Constructor.

Parameters

- `gamma`: the discount factor
- `alpha`: a string describing the learning rate
- `policyName`: the name of the policy used
- `policyHPar`: the map of the hyperparameters of the policy
- `Qw`: the weights of the linear approximator of the Q-Function

virtual void **writeData**(std::ostream &*os*)
Basic method to write plain data.

Parameters

- `os`: output stream in which the data should be logged

virtual void **writeDecoratedData**(std::ostream &*os*)
Basic method to write decorated data, e.g. for printing on screen.

Parameters

- `os`: output stream in which the data should be logged

class `ReLe::R_LearningOutput`
This class implements the output data for the *ReLe::R_Learning* algorithm.

Inherits from *ReLe::FiniteTDOutput*

Public Functions

R_LearningOutput(**const** std::string &*alpha*, **const** std::string &*beta*, **const** std::string &*policyName*, **const** hyperparameters_map &*policyHPar*, **const** arma::mat &*Q*, double *ro*)
Constructor.

Parameters

- `alpha`: the Q-function learning rate
- `beta`: the average expected reward learning rate
- `policyName`: the name of the policy used
- `policyHPar`: the map of the hyperparameters of the policy
- `Q`: the Q-table
- `ro`: the average expected reward

virtual void **`writeData`**(std::ostream &*os*)
Basic method to write plain data.

Parameters

- `os`: output stream in which the data should be logged

virtual void **`writeDecoratedData`**(std::ostream &*os*)
Basic method to write decorated data, e.g. for printing on screen.

Parameters

- `os`: output stream in which the data should be logged

class `ReLe::FQIOutput`
This class implements the output data for all Fitted Q-Iteration algorithms. All version of Fitted Q-Iteration algorithms should use, or extend this class

Inherits from *ReLe::AgentOutputData*

Public Functions

`FQIOutput`(bool *isFinal*, double *gamma*, double *delta*, Regressor &*QRegressor*)
Constructor.

Parameters

- `isFinal`: whether the data logged comes from the end of a run of the algorithm
- `gamma`: the discount factor
- `QRegressor`: the regressor

virtual void **`writeData`**(std::ostream &*os*)
Basic method to write plain data.

Parameters

- `os`: output stream in which the data should be logged

virtual void **`writeDecoratedData`**(std::ostream &*os*)
Basic method to write decorated data, e.g. for printing on screen.

Parameters

- `os`: output stream in which the data should be logged

Batch

Policy Search

Optimization

class ReLe::**Optimization**
This class contains some utilities function for optimization with nlopt.

Public Static Functions

static double **oneSumConstraint**(unsigned int *n*, **const** double *x, double *grad, void *data)
This function implements the one sum constraint, to be used as nonlinear constraint in nlopt

static double **oneSumConstraintIndex**(unsigned int *n*, **const** double *x, double *grad, void *data)
This function implements the one sum constraint, to be used as nonlinear constraint in nlopt This version allow to specify a index vector for wich the constraint is valid

template <**class** Class, bool *print* = false>
static double **objFunctionWrapper**(unsigned int *n*, **const** double *x, double *grad, void *o)
This class implements a simple objective function wrapper, in order to use easily an objective function that uses armadillo vectors instead of std::vectors. Also this template provides the necessary cast to use a method as objective function. Class using this method should implement the method `double objFunction(const arma::vec& x, arma::vec& dx)` Optionally another template parameter can be defined to enable debug print (parameters, derivative, objective function). Example:

```
class Example
{
public:
        double objFunction(const arma::vec& x, arma::vec& dx);
        void runOptimization();

};

...

void Example::runOptimization()
{
        ...

        nlopt::opt optimizator(optAlg, effective_dim);
        optimizator.set_min_objective(Optimization::objFunctionWrapper<Example, true> , this

        ...
}
```

class ReLe::**Simplex**
This class implements the simplex one sum constraint for optimization. Given a parameter space $\theta_{simplex} \in \mathbb{R}^{n-1}$ this class reconstructs the full parametrization $\theta \in \mathbb{R}^n$ by appling the one sum constraint. This class can also compute the derivative of the reduced parametrization. Also, this class support non active parameters in the parameters vector, in order to compute the simplex constraint on a subset of the original parametrization.

Public Functions

Simplex (unsigned int *size*)
Constructor.

Parameters

- `size`: the dimension of the full parameter space

arma::vec **reconstruct** (**const** arma::vec &*xSimplex*)
Given a set of parameters in the simplex, reconstruct the full parametrization

Return a vector to the full parametrization

Parameters

- `xSimplex`: an armadillo vector of the simplex parameters

arma::vec **reconstruct** (**const** std::vector<double> *xSimplex*)
Given a set of parameters in the simplex, reconstruct the full parametrization

Return a vector to the full parametrization

Parameters

- `xSimplex`: an std::vector of the simplex parameters

arma::vec **getCenter** ()
returns the center of the simplex.

Return a vector to the full parametrization

arma::mat **diff** (**const** arma::mat &*df*)
Compute the derivative of the function w.r.t. the simplex

Return the composite derivative matrix

Parameters

- `df`: the derivative w.r.t. the full parametrization

unsigned int **getFeatureIndex** (unsigned int *i*)
Getter.

Return the nth active parameter index

unsigned int **getEffectiveDim** ()
Getter.

Return the simplex dimension (considering only active features)

void **setActiveFeatures** (arma::uvec &*active*)
Setter.

Parameters

- `active`: the active features

Feature Selection

class `ReLe::FeatureSelectionAlgorithm`
This class is the basic interface for features selection algorithm. Features selection is the task of computing, from a set of features ϕ a new set of features $\bar{\phi}$, such that: $\exists T : \bar{\phi} = T(\phi)$

Subclassed by *ReLe::LinearFeatureSelectionAlgorithm*

Public Functions

virtual void **createFeatures**(**const** arma::mat &*features*) = 0
This method computes the new features from the initial ones

Parameters

- `features`: the initial features

virtual arma::mat **getNewFeatures**() = 0
Getter.

Return the new features, computed by the algorithm.

class `ReLe::LinearFeatureSelectionAlgorithm`
This class is the basic interface for linear features selection algorithms, i.e. a set of algorithms where the features selection functon is a linear function: $\bar{\phi} = T\phi$

Inherits from *ReLe::FeatureSelectionAlgorithm*

Subclassed by *ReLe::PrincipalComponentAnalysis*, *ReLe::PrincipalFeatureAnalysis*

Public Functions

virtual arma::mat **getTransformation**() = 0
Getter.

Return the linear features transformation.

class `ReLe::PrincipalComponentAnalysis`
This class implements Principal Component Analysis (PCA). PCA computes a linear combination of a set of features to reduce data dimensionality.

Inherits from *ReLe::LinearFeatureSelectionAlgorithm*

Public Functions

PrincipalComponentAnalysis(double *varMin*, bool *useCorrelation* = true)
Constructor.

Parameters

- `varMin`: the minimum amount of features to be used
- `useCorrelation`: if to use correlation or covariance matrix as selection criterion

virtual void **createFeatures**(**const** arma::mat &*features*)
This method computes the new features from the initial ones

Parameters

- `features`: the initial features

virtual arma::mat **getTransformation**()
Getter.

Return the linear features transformation.

virtual arma::mat **getNewFeatures**()
Getter.

Return the new features, computed by the algorithm.

class ReLe::**PrincipalFeatureAnalysis**

This class implements the Principal Feature Analysis (PFA). This method search the most promising features index to reduce dimensionality, thus maintaining the initial features, instead of creating new ones.

Inherits from *ReLe::LinearFeatureSelectionAlgorithm*

Public Functions

PrincipalFeatureAnalysis(double *varMin*, bool *useCorrelation* = true)

Constructor.

Parameters

- varMin: the minimum variability to retain from data
- useCorrelation: if to use correlation or covariance matrix as selection criterion

virtual void **createFeatures**(**const** arma::mat &*features*)

This method computes the new features from the initial ones

Parameters

- features: the initial features

virtual arma::mat **getTransformation**()

Getter.

Return the linear features transformation.

virtual arma::mat **getNewFeatures**()

Getter.

Return the new features, computed by the algorithm.

arma::uvec **getIndexes**()

Getter.

Return the indexes of the most promising features

Utils

Armadillo Extensions

Linear Algebra

arma::mat ReLe::**null**(**const** arma::mat &*A*, double *tol* = -1)

Null Space.

Return an orthonormal basis (Z) for the null space of A obtained from Singular Value Decomposition (SVD). That is, A * Z has negligible elements, Z.n_cols is the nullity of A and Z.t() * Z = I.

Parameters

- A: the input matrix
- tol: tolerance used in the rank test. If tol < 0, a default tolerance is used

arma::uvec ReLe::**rref**(**const** arma::mat &*X*, arma::mat &*A*, double *tol* = -1)

Reduced row echelon form (Gauss-Jordan elimination).

Return a vector idxs such that:

1. r = length(idxs) is this algorithm's idea of the rank of X.
2. A(:, idxs) is a basis for the range of X.
3. A(1:r, idxs) is the r-by-r identity matrix. In addition the reduced row echelon form of X using Gauss-Jordan elimination with partial pivoting is returned through A.

Parameters

- `X`: matrix to be reduced
- `A`: reduced row echelon form
- `tol`: tolerance used in the rank test. If tol < 0, a default tolerance of (max(size(A)) * eps * norm(A, inf)) tests for negligible column elements.

Warning: doxygenfunction: Cannot find function "ReLe::wrapTo2Pi" in doxygen xml output for project "ReLe" from directory: doxyxml/

Warning: doxygenfunction: Cannot find function "ReLe::wrapTo2Pi" in doxygen xml output for project "ReLe" from directory: doxyxml/

Warning: doxygenfunction: Cannot find function "ReLe::wrapToPi" in doxygen xml output for project "ReLe" from directory: doxyxml/

Warning: doxygenfunction: Cannot find function "ReLe::wrapToPi" in doxygen xml output for project "ReLe" from directory: doxyxml/

void `ReLe::meshgrid`(**const** arma::vec &*x*, **const** arma::vec &*y*, arma::mat &*xx*, arma::mat &*yy*)

Rectangular grid in 2-D space. meshgrid(xgv, ygv, X, Y) replicates the grid vectors xgv and ygv to produce a full grid stored in X and Y. This grid is represented by the output coordinate arrays X and Y. The output coordinate arrays X and Y contain copies of the grid vectors xgv and ygv respectively. The sizes of the output arrays are determined by the length of the grid vectors. For grid vectors xgv and ygv of length M and N respectively, X and Y will have N rows and M columns.

Parameters

- `x`: Grid vector specifying a series of grid point coordinates in the x direction
- `y`: Grid vector specifying a series of grid point coordinates in the y direction
- `xx`: Output matrix that specifies the full grid components in the x direction
- `yy`: Output matrix that specifies the full grid components in the y direction

arma::sp_mat `ReLe::blockdiagonal`(**const** std::vector<arma::mat> &*diag_blocks*)

Generate a block diagonal matrix from the input arguments. Note that the provided order is used to define the diagonal.

Return a sparse block diagonal matrix

Parameters

- `diag_blocks`: a set of matrices

arma::sp_mat `ReLe::blockdiagonal`(**const** std::vector<arma::mat> &*diag_blocks*, int *rows*, int *cols*)

Generate a block diagonal matrix from the input arguments. This implementation is more efficient when the size of the resulting matrix is known by the caller. This function avoid the computation of such information from the given matrix vector.

Return a sparse block diagonal matrix

Parameters

- `diag_blocks`:
- `rows`: number of rows of the resulting matrix
- `cols`: number of cols of the resulting matrix

arma::vec `ReLe::range`(arma::mat &*X*, unsigned int *dim* = 0)
range(X) returns the difference between the maximum and the minimum of a sample. For vectors, range(x) is the range of the elements. For matrices, range(X) is a row vector containing the range of each column of X.

Return the difference between the maximum and the minimum

Parameters

- `X`: the vector of matrix to be analyzed
- `dim`: the range is computed along dimension dim of X

void `ReLe::vecToTriangular`(**const** arma::vec &*vector*, arma::mat &*triangular*)
Generate a lower triangular matrix from the input vector. This function avoids the computation of matrix size from the given matrix vector.

Parameters

- `vector`: vector of elements of the triangular matrix, must be of size (dim^2 - dim) / 2
- `triangular`: resulting triangular matrix, must be of size (dim, dim)

void `ReLe::triangularToVec`(**const** arma::mat &*triangular*, arma::vec &*vector*)
Generate a vector from the input lower triangular matrix. This function avoid the computation of matrix size from the given vector.

Parameters

- `triangular`: resulting triangular matrix, must be of size (dim, dim)
- `vector`: vector of nonzero elements of the triangular matrix, must be of size (dim^2 - dim) / 2

Warning: doxygenfunction: Cannot find function "ReLe::safeChol" in doxygen xml output for project "ReLe" from directory: doxyxml/

Distributions

double `ReLe::mvnpdf`(**const** arma::vec &*x*, **const** arma::vec &*mean*, **const** arma::mat &*cov*)
Calculates the multivariate Gaussian probability density function.

Return Probability density of x given the distribution

Parameters

- `x`: Observation
- `mean`: Mean of multivariate Gaussian
- `cov`: Covariance of multivariate Gaussian

double ReLe::**mvnpdfFast**(**const** arma::vec &*x*, **const** arma::vec &*mean*, **const** arma::mat &*inverse_cov*, **const** double &*det*)

Calculates the multivariate Gaussian probability density function.

Differently from *ReLe::mvnpdf*, needs the determinant and the inverse matrix for fast pdf computation.

Return Probability density of x given the distribution

Parameters

- `x`: Observation
- `mean`: Mean of multivariate Gaussian
- `inverse_cov`: Inverse of the covariance of multivariate Gaussian
- `det`: The determinant of the covariance matrix

double ReLe::**mvnpdf**(**const** arma::vec &*x*, **const** arma::vec &*mean*, **const** arma::mat &*cov*, arma::vec &*g_mean*)

Calculates the multivariate Gaussian probability density function and also the gradients of the mean w.r.t. the input value x.

Return Probability density of x given the distribution

Parameters

- `x`: Observation
- `mean`: Mean of multivariate Gaussian
- `cov`: Covariance of multivariate Gaussian
- `g_mean`: the gradient w.r.t. the mean vector

double ReLe::**mvnpdf**(**const** arma::vec &*x*, **const** arma::vec &*mean*, **const** arma::mat &*cholCov*, arma::vec &*g_mean*, arma::mat &*g_cholSigma*)

Calculates the multivariate Gaussian probability density function and also the gradients of the mean and cholesky decomposition of covariance w.r.t. the input value x.

Return Probability density of x given the distribution

Parameters

- `x`: Observation
- `mean`: Mean of multivariate Gaussian
- `cholCov`: Cholesky decomposition of covariance of multivariate Gaussian.
- `g_mean`: the gradient w.r.t. the mean vector
- `g_cholSigma`: the gradient w.r.t the cholesky decomposition of covariance matrix

double ReLe::**mvnpdfFast**(**const** arma::vec &*x*, **const** arma::vec &*mean*, **const** arma::mat &*inverse_cov*, **const** double &*det*, arma::vec &*g_mean*, arma::vec &*g_cov*)

Calculates the multivariate Gaussian probability density function and also the gradients of the mean and variance w.r.t. the input value x.

Differently from *ReLe::mvnpdf*, needs the determinant and the inverse matrix for fast pdf computation.

Return Probability density of x given the distribution

Parameters

- `x`: Observation
- `mean`: Mean of multivariate Gaussian
- `inverse_cov`: the inverse of the covariance matrix
- `det`: the determinant of the covariance matrix
- `g_mean`: the gradient w.r.t. the mean vector
- `g_cov`: the gradient w.r.t. the covariance matrix

void `ReLe::mvnpdf`(**const** arma::mat &*x*, **const** arma::vec &*mean*, **const** arma::mat &*cov*, arma::vec &*probabilities*)

Calculates the multivariate Gaussian probability density function for each data point (column) in the given matrix, with respect to the given mean and variance.

Parameters

- `x`: List of observations.
- `mean`: Mean of multivariate Gaussian.
- `cov`: Covariance of multivariate Gaussian.
- `probabilities`: Output probabilities for each input observation.

arma::mat `ReLe::mvnrand`(int *n*, **const** arma::vec &*mu*, **const** arma::mat &*sigma*)

Samples n points from a gaussian multivariate distribution.

Return a matrix with n columns, representing the sampled points

Parameters

- `n`: number of points to sample
- `mu`: mean of the multivariate gaussian
- `sigma`: covariance matrix of the multivariate gaussian

arma::vec `ReLe::mvnrand`(**const** arma::vec &*mu*, **const** arma::mat &*sigma*)

Samples a points from a gaussian multivariate distribution.

Return the sampled point

Parameters

- `mu`: mean of the multivariate gaussian
- `sigma`: covariance matrix of the multivariate gaussian

arma::vec `ReLe::mvnrandFast`(**const** arma::vec &*mu*, **const** arma::mat &*CholSigma*)

Samples a points from a gaussian multivariate distribution.

Differently from *ReLe::mvnrand*, needs the cholesky decomposition of the covariance matrix for fast sampling.

Return the sampled point

Parameters

- `mu`: mean of the multivariate gaussian
- `CholSigma`: covariance matrix of the multivariate gaussian

Other Utils

class `ReLe::ConsoleManager`
This class implements some basics functions to manage the progress status of a running algorithm.

Public Functions

`ConsoleManager`(unsigned int *max*, unsigned int *step*, bool *percentage* = false)
Constructor.

Parameters

- `max`: max value of the progress
- `step`: step size of the progress
- `percentage`: indicates whether to show the progress in percentage values or not

void **`printProgress`**(unsigned int *progress*)
Print progress.

Parameters

- `progress`: the current progress value

void **`printInfo`**(**const** std::string &*info*)
Print info.

Parameters

- `info`: the info to be printed

class `ReLe::CSVutils`
This class offers some helpful functions to manage CSV files.

Public Static Functions

static bool **`readCSVLine`**(std::istream &*is*, std::vector<std::string> &*tokens*)
Read a line from a CSV file.

Return a bool value indicating whether the read has been successful or not

Parameters

- `is`: the CSV file to read
- `tokens`: vector of elements contained in the CSV file

static void **`matrixToCSV`**(**const** arma::mat &*M*, std::ostream &*os*)
Print a CSV file from an armadillo matrix.

Parameters

- `M`: matrix to print in the file
- `os`: the file where the matrix is printed

static void **`vectorToCSV`**(**const** arma::vec &*v*, std::ostream &*os*)
Print a CSV file from an armadillo vector.

Parameters

- `v`: vector to print in the file

- `os`: the file where the vector is printed

template <class T>
static void **vectorToCSV**(**const** std::vector<T> &*v*, std::ostream &*os*)
Template function to print a CSV file from an armadillo matrix.

Parameters

- `v`: vector to print in the file
- `os`: the file where the vector is printed

class `ReLe::FileManager`
This class has some useful functions to manage I/O of data files.

Public Functions

FileManager(**const** std::string &*environment*, **const** std::string &*algorithm*)
Constructor. It creates the path where the file will be saved or loaded.

Parameters

- `environment`: name of the environment in which the algorithm is executed
- `algorithm`: name of the algorithm to be performed

FileManager(**const** std::string &*testName*)
Constructor. It creates the path where the file will be saved or loaded.

Parameters

- `testName`: name of the test to be performed

void **createDir**()
Creates the directory at the path given by the constructor.

void **cleanDir**()
Cleans the directory at the path given by the constructor parameters

std::string **addPath**(**const** std::string &*fileName*)
Add the name of the file to the path created in the constructor.

Return the path string

Parameters

- `fileName`: name of the file

std::string **addPath**(**const** std::string &*prefix*, **const** std::string &*fileName*)
Add the name of the file to the path created in the constructor.

Return the path string

Parameters

- `prefix`: the prefix before the name of the file which is separated by '_'
- `fileName`: name of the file

class `ReLe::RngGenerators`
This class implements function to generate Random Number Generators.

Public Functions

RngGenerators(unsigned int *seed*)
Constructor. This function initializes the Random Number Generator of std and armadillo libraries and the Random Number Generator of the class attribute with the provided seed.

Parameters

- `seed`: the seed to be set

void **seed**(unsigned int *seed*)
Modifies the Random Number Generator of std and armadillo libraries and the Random Number Generator of the class attribute with the provided seed.

Parameters

- `seed`: the seed to be set

Public Members

std::mt19937 **gen**
Random Number Generator variable based on Mersenne Twister Algorithm.

class `ReLe::RandomGenerator`
This class has some useful function to sample random numbers from different probability distributions.

Public Static Functions

static uint32_t **randu32**()
Generate a random unsigned integer number of 32 bits.

Return the random unsigned integer number

static double **sampleNormal**()
Sample values from a normal distribution with mean = 0 and std = 1.

Return the random number sampled from the normal distribution

static double **sampleNormal**(double *m*, double *sigma*)
Sample values from a normal distribution with provided mean and standard deviation.

Return the random number sampled from the normal distribution

Parameters

- `m`: the mean of the normal distribution
- `sigma`: the standard deviation of the normal distribution

static double **sampleUniform**(**const** double *lo*, **const** double *hi*)
Sample values from a uniform distribution with provided lower and higher values where the higher one is excluded from the range.

Return the random number sampled from the uniform distribution

Parameters

- `lo`: the lower value of the uniform distribution
- `hi`: the upper value of the uniform distribution

static double **sampleUniformHigh**(**const** double *lo*, **const** double *hi*)
Sample values from a uniform distribution with provided lower and higher values where the lower one is excluded from the range.

Return the random number sampled from the uniform distribution

Parameters

- `lo`: the lower value of the uniform distribution
- `hi`: the upper value of the uniform distribution

static std::size_t **sampleUniformInt**(**const** int *lo*, **const** int *hi*)
Sample integer values from a uniform distribution with provided lower and higher values where the lower one is excluded from the range.

Return the integer random number sampled from the uniform distribution

Parameters

- `lo`: the lower value of the uniform distribution
- `hi`: the upper value of the uniform distribution

static std::size_t **sampleDiscrete**(std::vector<double> &*prob*)
Sample a random integer from 0 to n where n is the number of elements of the given probability vector which indicates the probability of each number to be sampled.

Return the sampled number

Parameters

- `prob`: vector of probabilities

template <class Iterator**>**
static std::size_t **sampleDiscrete**(Iterator *begin*, Iterator *end*)
Template function to sample a random integer from 0 to n where n is the number of elements of the given probability vector (of arbitrary type) which indicates the probability of each number to be sampled.

Return the sampled number

Parameters

- `begin`: the first element of the probabilities vector
- `end`: the last element of the probabilities vector

static bool **sampleEvent**(double *prob*)
Sample the outcome of an event (false or true) from a uniform distribution with given probability.

Return a bool value representing the happening of an event

Parameters

- `prob`: the probability of the event

static void **seed**(unsigned int *seed*)
Set the seed of the Random Number Generator with a given seed.

Parameters

- `seed`: the seed to be set

class `ReLe::Range`
Simple real-valued range. It contains an upper and lower bound.

Subclassed by *ReLe::ModularRange*

Public Functions

Range()
Constructor. Initialize to an empty set (where lo > hi).

Range(**const** double *point*)
Constructor. Initialize a range to enclose only the given point (lo = point, hi = point).

Parameters

- `point`: point that this range will enclose

Range(**const** double *lo*, **const** double *hi*)
Initialize to specified range.

Parameters

- `lo`: the lower bound of the range
- `hi`: the upper bound of the range

double **lo**() **const**
Get the lower bound.

Return the lower bound value

double &**lo**()
Get a reference to the lower bound.

Return a reference to the lower bound

double **hi**() **const**
Get the upper bound.

Return the upper bound value

double &**hi**()
Get a reference to the upper bound.

Return a reference to the upper bound

double **width**() **const**
Get the span of the range (hi - lo).

Return the width of the range

double **mid**() **const**
Get the midpoint of this range.

Return the mid point value

virtual double **bound**(**const** double &*value*) **const**
Check whether a value is inside the range.

Return the lower bound in case the value is lower than it, the upper bound in case the value is greater than it, the value in case the value is inside the range

Parameters

- `value`: the value to check

Range &**operator|=**(**const** *Range* &*rhs*)
Expand this range to include another range.

Return a reference to the expanded range

Parameters

- `rhs`: range to include

Range **operator|**(**const** *Range* &*rhs*) **const**
Expand this range to include another range.

Return the expanded range

Parameters

- `rhs`: range to include

Range &**operator&=**(**const** *Range* &*rhs*)
Shrink this range to make it overlap to another range; this makes an empty set if there is no overlapping.

Return a reference to the overlapped range

Parameters

- `rhs`: other range

Range **operator&**(**const** *Range* &*rhs*) **const**
Shrink this range to make it overlap to another range; this makes an empty set if there is no overlapping.

Return the overlapped range

Parameters

- `rhs`: other range

Range &**operator+=**(**const** double *d*)
Add offset to the bounds by the given double.

Return a reference to the translated range

Parameters

- `d`: offset

Range **operator+**(**const** double *d*) **const**
Add offset to the bounds by the given double.

Return the translated range

Parameters

- `d`: offset

Range &**operator*=**(**const** double *d*)
Scale the bounds by the given double.

Return a reference to the scaled range

Parameters

- `d`: scaling factor

Range **operator***(**const** double *d*) **const**
Scale the bounds by the given double.

Return the scaled range

Parameters

- `d`: scaling factor

bool **operator==**(**const** *Range* &*rhs*) **const**
Compare with another range for strict equality.

Return true if the ranges are equal

Parameters

- `rhs`: other range

bool **operator!=**(**const** *Range* &*rhs*) **const**
Compare with another range for strict inequality. param rhs other range

Return true if the ranges are not equal

bool **operator<**(**const** *Range* &*rhs*) **const**
Compare with another range. For *Range* objects x and y, x < y means that x is strictly less than y and does not overlap at all.

Return true if the range is strictly less than the other

Parameters

- `rhs`: other range

bool **operator>**(**const** *Range* &*rhs*) **const**
Compare with another range. For *Range* objects x and y, x > y means that x is strictly greater than y and does not overlap at all.

Return true if the range is strictly greater than the other

Parameters

- `rhs`: other range

virtual bool **contains**(**const** double *d*) **const**
Determine if a point is contained within the range.

Return true if the point is within the range

Parameters

- `d`: the point to check.

bool **contains**(**const** *Range* &*r*) **const**
Determine if another range overlaps completely with this one.

Return true if ranges overlap completely

Parameters

- `r`: other range

std::string **toString**() **const**
Return a string representation of an object.

Return the string indicating lower and upper bound of the range

Friends

Range **operator+**(**const** double *d*, **const** Range &*r*)
Translates the bounds of another range by a given double and creates a new range from them.

Return a new translated range

Parameters

- `d`: offset
- `r`: range whose bound values are to be translated

Range **operator*** (**const** double *d*, **const** Range &*r*)
Scale the bounds of another range by a given double and creates a new range from them.

Return a new scaled range

Parameters

- `d`: scaling factor
- `r`: range whose bound values are to be scaled

std::ostream &**operator<<** (std::ostream &*out*, **const** Range &*range*)
Print a string representation of an object to an upstream variable.

Return the output object

Parameters

- `out`: the output where to print the string
- `range`: the range to be printed

class `ReLe::ModularRange`
In some cases (e.g. angle values), ranges repeats over and over and one may want to convert a number into the corresponding number within the given range. This class is useful to do so. For instance, given a range of [-180°, 180°] and an angle of 186°, the corresponding number within the range would be -174°.

Inherits from *ReLe::Range*

Subclassed by *ReLe::Range2Pi*, *ReLe::RangePi*

Public Functions

ModularRange (**const** double *lo*, **const** double *hi*)
Constructor.

Parameters

- `lo`: the lower bound of the range
- `hi`: the upper bound of the range

virtual bool **contains** (**const** double *d*) **const**
Determine if a point has a regular value (i.e. different from infinity or NaN).

Return true if the point has a regular value

Parameters

- `d`: the point to check

virtual double **bound** (**const** double &*value*) **const**
Given a number, it returns the corresponding number within the range.

Return the converted number

Parameters

- `value`: the number to be converted

class `ReLe::NumericalGradient`
This class implements functions to compute the numerical gradient of functions. The real gradient is approximated with the well-known incremental formula (with a small value of ϵ), i.e.,

$$\frac{\partial J}{\partial \theta} \approx \frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon}.$$

Public Static Functions

template <**class** F>
static arma::mat **compute**(F *J*, arma::vec *theta*, unsigned int *size* = 1)
Numerical gradient computation for generic functor.

Return the numerical gradient

Parameters

- `J`: the function to be derived
- `theta`: parameters vector
- `size`: the dimensionality of the output

static arma::mat **compute**(ParametricRegressor &*regressor*, arma::vec *theta*, arma::vec &*input*)
Numerical gradient computation for regression functions.

Return the numerical gradient

Parameters

- `regressor`: regression function to be derived
- `theta`: parameters vector
- `input`: input vector of the regressor

template <**class** ActionC, **class** StateC>
static arma::mat **compute**(*DifferentiablePolicy*<ActionC, StateC> &*policy*, arma::vec *theta*, **typename** *state_type*<StateC>::const_type_ref *state*, **typename** *action_type*<ActionC>::const_type_ref *action*)
Numerical gradient computation for policy functions.

Return the numerical gradient

Parameters

- `policy`: policy function to be derived
- `theta`: parameters vector
- `state`: vector of states
- `action`: vector of actions

class `ReLe::Range2Pi`
This class is used to manage fixed angle ranges from 0 to 2 * pi radians (0° to 360° in degrees).

Inherits from *ReLe::ModularRange*

Public Functions

Range2Pi()
Constructor. Create the range between 0 and 2 * pi radians.

Public Static Functions

static double **wrap**(double *value*)
Given a number, it returns the corresponding number within the range.

Return the converted number

Parameters

- `value`: the number to be converted

class `ReLe::RangePi`

This class is used to manage fixed angle range from 0 to pi radians (0° to 180° in degrees).

Inherits from *ReLe::ModularRange*

Public Functions

`RangePi``()`

Constructor. Create the range between 0 and pi radians.

Public Static Functions

static double **`wrap`**(double *value*)

Given a number, it returns the corresponding number within the range.

Return the converted number

Parameters

- `value`: the number to be converted

Tutorials

Online methods

Learn a simple MDP

In this tutorial we will use ReLe to run the Q-Learning algorithm on a simple MDP.

We will use as environment a simple chain below:

The action success probability is p=0.8, if an action fails the agent stays in the same state.

First of all we will create a Finite MDP using a generator:

```
1    //Create the MDP
2    SimpleChainGenerator generator;
3    generator.generate(5, 2);
4    FiniteMDP mdp = generator.getMDP(0.9);
```

Now we have to create a Q-Learning agent. The Q-Learning agent needs a policy specification, and the learning rate. For this simple environment we can use an ϵ-greedy policy and a costant learning rate

```
1    //Create the agent
2    e_Greedy policy;
3    ConstantLearningRate alpha(0.2);
4    Q_Learning agent(policy, alpha);
```

Finally we create a core to run our agent on the mdp. In this simple example we can just run a single episode. We will use a *ReLe::PrintStrategy* to print the results on the console.

```
    //Setup the experiment
    Core<FiniteAction, FiniteState> core(mdp, agent);
    core.getSettings().episodeLength = 10000;
    bool logTransition = false;
    bool logAgent = true;
    core.getSettings().loggerStrategy =
                new PrintStrategy<FiniteAction, FiniteState>(logTransition,
                                                             logAgent);

    //Run the learning
    cout << "starting episode" << endl;
    core.runEpisode();
```

After running the code you should see on the console two section:

- Statistics, in which is described the initial state and the state frequencies
- Agent data at episode end, in which is printed agent data
  - Parameters: discount factor, learning rate, ϵ of the ϵ greedy policy
  - Action Value function
  - Policy

The output shoudl be similar to this:

```
starting episode


--- statistics ---

- Initial State
x(t0) = [3]
- State Statistics
0: 0.0165
1: 0.2389
2: 0.4696
3: 0.2574
4: 0.0176

--- Agent data at episode end ---
Using e-Greedy policy

- Parameters
gamma: 0.9
alpha: 0.200000
eps: 0.15
- Action-value function
Q(0, 0) = 3.89557
Q(0, 1) = 3.13971
Q(1, 0) = 4.30185
Q(1, 1) = 3.62914
Q(2, 0) = 3.58693
Q(2, 1) = 3.7851
Q(3, 0) = 3.56217
Q(3, 1) = 4.3094
Q(4, 0) = 2.89102
Q(4, 1) = 3.84717
- Policy
policy(0) = 0
```

```
policy(1) = 0
policy(2) = 1
policy(3) = 1
policy(4) = 1
```

Please, note that this mdp has two optimal policies, as in the goal state (state n 2), the two action has the same expected return; and Q-Learning can find two different optimal policies.

The complete code is the following:

```
#include "rele/core/FiniteMDP.h"
#include "rele/algorithms/td/SARSA.h"
#include "rele/algorithms/td/Q-Learning.h"
#include "rele/core/Core.h"

#include "rele/policy/q_policy/e_Greedy.h"
#include "rele/policy/q_policy/Boltzmann.h"

#include "rele/generators/SimpleChainGenerator.h"

#include "rele/approximators/basis/IdentityBasis.h"

#include <iostream>

using namespace std;
using namespace ReLe;

int main(int argc, char *argv[])
{
    //Create the MDP
    SimpleChainGenerator generator;
    generator.generate(5, 2);
    FiniteMDP mdp = generator.getMDP(0.9);

    //Create the agent
    e_Greedy policy;
    ConstantLearningRate alpha(0.2);
    Q_Learning agent(policy, alpha);

    //Setup the experiment
    Core<FiniteAction, FiniteState> core(mdp, agent);
    core.getSettings().episodeLength = 10000;
    bool logTransition = false;
    bool logAgent = true;
    core.getSettings().loggerStrategy =
                new PrintStrategy<FiniteAction, FiniteState>(logTransition,
                                                             logAgent);

    //Run the learning
    cout << "starting episode" << endl;
    core.runEpisode();
    delete core.getSettings().loggerStrategy;

    return 0;
}
```

Batch methods

Inverse Reinforcement Learning

O

R