IBOAT RL Documentation

Release 1.0.0

Tristan Karch

Feb 19, 2018

CONTENTS

CHAPTER

ONE

A BRIEF CONTEXT

This project presents **Reinforcement Learning** as a solution to control systems with a **large hysteresis**. We consider an autonomous sailing robot (IBOAT) which sails upwind. In this configuration, the wingsail is almost aligned with the upcoming wind. It thus operates like a classical wing to push the boat forward. If the angle of attack of the wind coming on the wingsail is too great, the flow around the wing detaches leading to a **marked decrease of the boat's speed**.

Hysteresis such as stall are hard to model. We therefore proposes an **end-to-end controller** which learns the stall behavior and builds a policy that avoids it. Learning is performed on a simplified transition model representing the stochastic environment and the dynamic of the boat.

Learning is performed on two types of simulators, A **proof of concept** is first carried out on a simplified simulator of the boat coded in Python. The second phase of the project consist of trying to control a **more realisitic** model of the boat. For this purpose we use a dynamic library which is derived using the Code Generation tools in Simulink. The executable C are then feeded to Python using the "ctypes" library.

On this page, you will find the documentation of the simplified simulator of the boat as well as the documentation of the reinforcement learning tools. Each package contains tutorials to better understand how the code can be used

CHAPTER

TWO

REQUIREMENTS

The project depends on the following extensions :

1. NumPy for the data structures (http://www.numpy.org)
2. Matplotlib for the visualisation (https://matplotlib.org)
3. Keras for the convolutional neural network models (https://keras.io)

CHAPTER

THREE

LIBRARIES

There are two dynamic libraries available to simulate the realistic model of the boat :

1. `libBoatModel.so` for Linux users
2. `libBoatModel.dylib` for Mac users

One has to change the extension of the library in the file `Simulator_realistic.py` depending on its OS. We also provide a guideline to generate such libraries from a simulink (see `this file` for more information).

CHAPTER

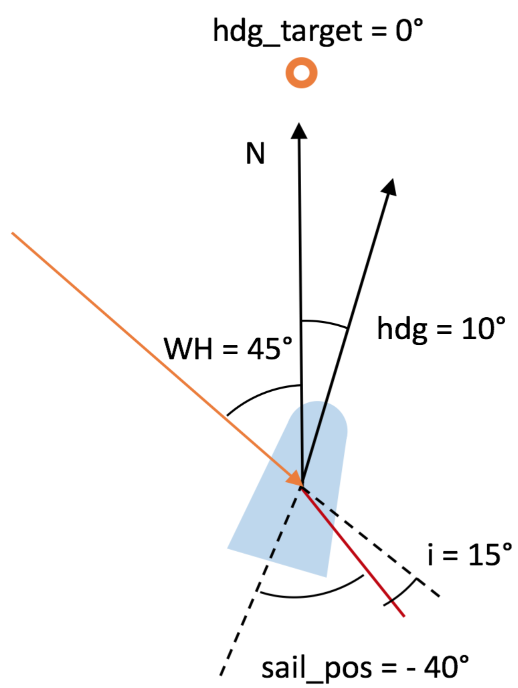
FOUR

CONTENTS

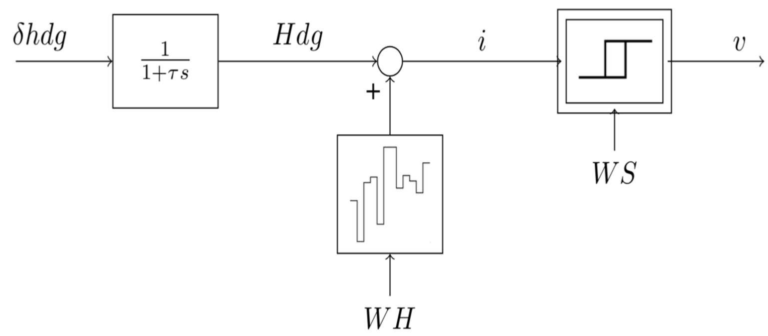
4.1 Package Simulator

This package contains all the classes required to build a simulation for the learning. In this small paragraph, the physic of the simulator is described so that the reader can better understand the implementation.

We need the boat to be in a configuration when it sails upwind so that the flow around the sail is attached and the sail works as a wing. To generate the configuration we first assume that the boat as a target heading *hdg_target* = 0. The boat as a certain heading *hdg* with respect to the north and faces an upcoming wind of heading *WH*. To lower the number of parameters at stake we consider that the wind has a **constant speed** of 15 knts. The sail is oriented with respect to the boat heading with an angle *sail_pos* = -40°. The angle of attack of the wind on the sail is therefore equal to *i* = *hdg* + *WH* + *sail_pos*. This angle equation can be well understood thanks to the following image.



The action taken to change the angle of attack are changes of boat heading *delta_hdg*. We therefore assume that *sail_pos* is constant and equal to -40°. The wind heading is fixed to *WH* = 45°. Finally, there is a delay between the command and the change of heading of τ = 0.5 seconds. The simulator can be represented with the following block diagram. It contains a delay and an hysteresis block that are variables of the simulator class.



4.1.1 Simulator

class `Simulator.`**`Simulator`**(*duration*, *time_step*)

Bases: `object`

Simulator object : It simulates a simplified dynamic of the boat with the config.

Variables

- **`time_step`** (*`float`*) – time step of the simulator, corresponds to the frequency of data acquisition.
- **`size`** (*`float`*) – size of the simulation.
- **`delay`** (*`int`*) – delay between the heading command and its activation.
- **`sail_pos`** (*`float`*) – position of the windsail [rad].
- **`hdg_target`** (*`float`*) – target heading towards which we want the boat to sail.
- **`hdg`** (*`np.array()`*) – array of size **size** that stores the heading of the boat [rad].
- **`vmg`** (*`np.array()`*) – array of size **size** that stores the velocity made good.
- **`hyst`** (`Hysteresis`) – Memory of the flow state during the simulations.

Raises **`ValueError`** – if the size of the simulation is zero or less.

`computeNewValues`(*delta_hdg*, *WH*)

Increment the boat headind and compute the corresponding boat velocity. This method uses the hysteresis function `Hysteresis.calculateSpeed()`. to calculate the velocity.

Parameters

- **`delta_hdg`** – increment of heading.
- **`WH`** – Heading of the wind on the wingsail.

Returns the heading and velocities value over the simulated time.

`getHdg`(*k*)

`getLength`()

`getTimeStep`()

`incrementDelayHdg`(*k*, *delta_hdg*)

incrementHdg(*k*, *delta_hdg*)
Increment the heading. :param k: index to increment. :param delta_hdg: value of the increment of heading. :return:

plot()

updateHdg(*k*, *inc*)
Change the value of the heading at index k. :param k: index to update. :param inc: :return:

updateVMG(*k*, *vmg*)

Parameters

- **k** – index to update
- **vmg** – value of the velocity to update

Returns

Warning: Be careful, the delay is expressed has an offset of index. the delay in s is equal to delay*time_step

4.1.2 Hysteresis

class Hysteresis.**Hysteresis**
Class that remembers the state of the flow and that computes the velocity for a given angle of attack of the wind on the wingsail. :ivar float e: state of the flow (0 if attached and 1 if detached)

calculateSpeed(*i*)
Calculate the velocity from angle of attack.

Parameters **i** (*float*) – angle of attack

Returns v - Boat velocity

Return type float

copy()

Returns a deepcopy of the object

reset()
Reset the memory of the flow.

4.1.3 Environment

class environment.**wind**(*mean*, *std*, *samples*)
Generetate the wind samples of the environment. The wind intesity is assumed constant and equal to 15 knots

Variables

- **mean** (*float*) – mean direction of the wind in [rad]
- **std** (*float*) – standard deviation of the wind direction in [rad]
- **samples** (*float*) – number of samples to generate

generateGust(*Delta_WH*)
Generates a Gust i.e. an important change of wind heading. :param Delta_WH: Magnitude of the gust. :return: The vector of wind heading corresponding to the gust.

generateWind()
Generates the wind samples :return: np.array of wind samples

4.1.4 Markov Decision Process (MDP)

class mdp.**ContinuousMDP**(*duration_history*, *duration_simulation*, *delta_t*, *LOWER_BOUND*, *UPPER_BOUND*)
Markov Decision process modelization of the transition Based on Realistic Simulator of Iboat autonomous sailboat provided by Simulink Compatible with continuous action

Variables

- **history_duration** (*float*) – Duration of the memory.
- **simulation_duration** (*float*) – Duration of the memory.
- **size** (*int*) – size of the first dimension of the state.
- **dt** (*float*) – time step between each value of the state.
- **s** (*np.array()*) – state containing the history of angles of attacks and velocities.
- **idx_memory** (*range*) – indices corresponding to the values shared by two successive states.
- **simulator** (Simulator) – Simulator used to compute new values after a transition.
- **reward** (*float*) – reward associated with a transition.
- **discount** (*float*) – discount factor.
- **action** (*float*) – action for transition.

computeState(*action*, *WH*)

copy()

extractSimulationData()

initializeMDP(*hdg0*, *WH*)

initializeState(*state*)

transition(*action*, *WH*)

class mdp.**MDP**(*duration_history*, *duration_simulation*, *delta_t*)
Markov Decision process modelization of the transition

Variables

- **history_duration** (*float*) – Duration of the memory.
- **simulation_duration** (*float*) – Duration of the memory.
- **size** (*int*) – size of the first dimension of the state.
- **dt** (*float*) – time step between each value of the state.
- **s** (*np.array()*) – state containing the history of angles of attacks and velocities.
- **idx_memory** (*range*) – indices corresponding to the values shared by two successive states.
- **simulator** (Simulator) – Simulator used to compute new values after a transition.
- **reward** (*float*) – reward associated with a transition.

- **discount** (*float*) – discount factor.
- **action** (*float*) – action for transition.

computeState (*action*, *WH*)
Computes the mdp state when an action is applied. :param action: :param WH: :return:

copy ()
Copy the MDP object

Returns Deep copy of the object.

Return type *MDP*

extractSimulationData ()

initializeMDP (*hdg0*, *WH*)
Initialization of the Markov Decicison Process.

Parameters

- **hdg0** (*float*) – initial heading of the boat.
- **np.array()** (*WH*) – Vector of wind heading.

Returns s initialized state

Return type np.array()

policy (*i_treshold*)

transition (*action*, *WH*)

Parameters

- **action** – action to make (either luff or bear off)
- **WH** – Wind heading provided by the environment during the transition.

Returns The state and reward

Note: The class variable simulation_duration defines the frequency of action taking. The reward is the average of the new velocities computed after each transition.

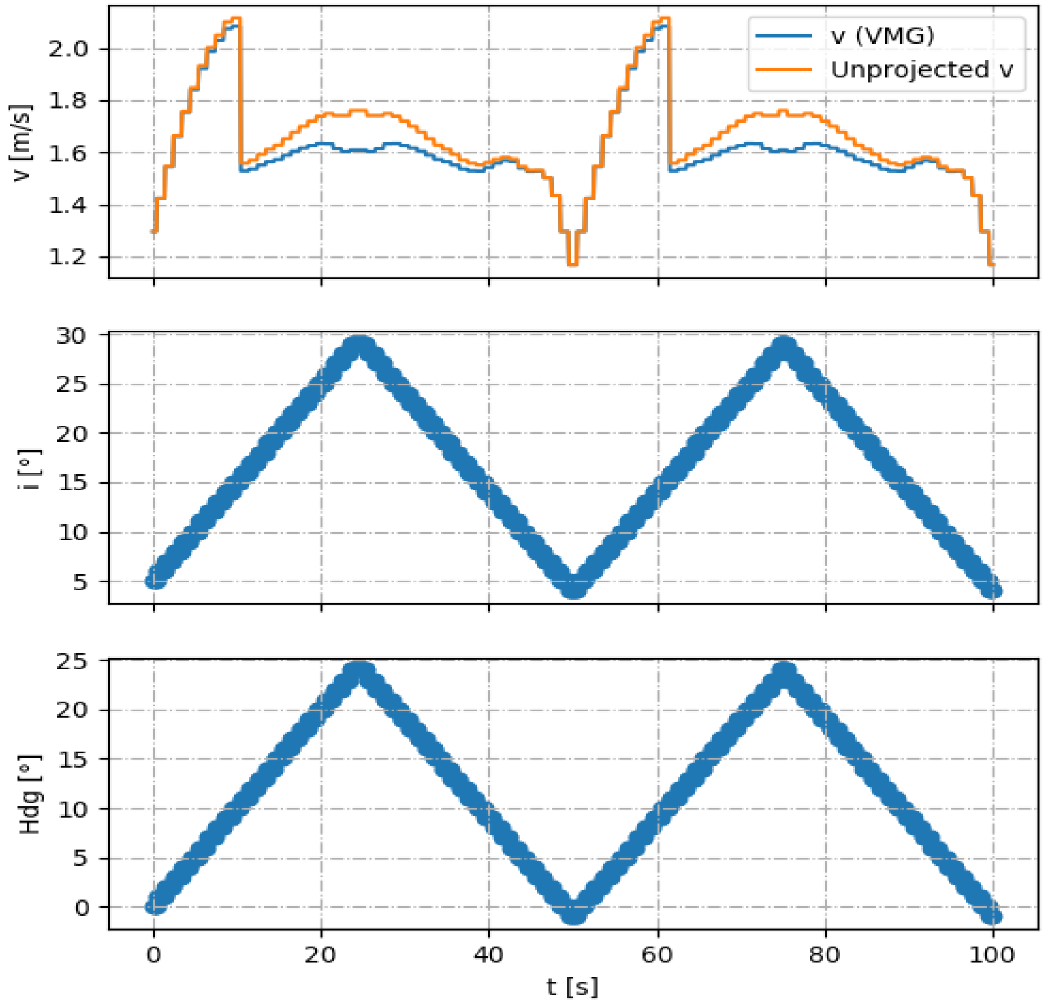
4.1.5 Tutorial

To visualize how a simulation can be generated we provide a file MDPmain.py that creates a simulation where the heading is first increase and then decrease.

```
import mdp
import numpy as np

TORAD = math.pi / 180

history_duration = 3
mdp_step = 1
time_step = 0.1
SP = -40 * TORAD
mdp = mdp.MDP(history_duration, mdp_step, time_step)

mean = 45 * TORAD
std = 0 * TORAD
```

```
wind_samples = 10
WH = np.random.uniform(mean - std, mean + std, size=10)

hdg0 = 0 * TORAD * np.ones(wind_samples)
state = mdp.initializeMDP(hdg0, WH)

SIMULATION_TIME = 100

i = np.ones(0)
vmg = np.ones(0)
wind_heading = np.ones(0)

for time in range(SIMULATION_TIME):
    print('t = {0} s'.format(time))
    action = 0
    WH = np.random.uniform(mean - std, mean + std, size=wind_samples)
    if time < SIMULATION_TIME / 4:
        action = 0
    elif time < SIMULATION_TIME / 2:
        action = 1
    elif time < 3 * SIMULATION_TIME / 4:
        action = 0
    else:
        action = 1

    next_state, reward = mdp.transition(action, WH)
    next_state = state
    i = np.concatenate([i, mdp.extractSimulationData()[0, :]])
    vmg = np.concatenate([vmg, mdp.extractSimulationData()[1, :]])
    wind_heading = np.concatenate([wind_heading, WH])

time_vec = np.linspace(0, SIMULATION_TIME, int((SIMULATION_TIME) / time_step))
hdg = i - wind_heading - SP
```

This results in the following value for the velocity, angle of attack and heading.



4.2 Package Realistic Simulator

This package contains all the classes required to build a realistic simulation of the boat. It uses the dynamic library `libBoatModel.so` which implements the accurate dynamic of the boat. It is coded in C++ in order to speed up the calculation process and hence the learning.

4.2.1 Realisitic Simulator

class `Simulator_realistic.`**`DW_Model_2_EXPORT_Discrete_T`**

Bases: `_ctypes.Structure`

`DiscreteFilter_states`

Structure/Union member

DiscreteTimeIntegrator1_DSTATE
Structure/Union member

DiscreteTimeIntegrator_DSTATE
Structure/Union member

DiscreteTimeIntegrator_DSTATE_p
Structure/Union member

DiscreteTimeIntegrator_IC_LOADI
Structure/Union member

DiscreteTimeIntegrator_IC_LOA_m
Structure/Union member

Discretemotormodel1_states
Structure/Union member

Discretemotormodel_states
Structure/Union member

FixPtUnitDelay1_DSTATE
Structure/Union member

FixPtUnitDelay2_DSTATE
Structure/Union member

G3_PreviousInput
Structure/Union member

PrevY
Structure/Union member

PrevY_d
Structure/Union member

incidencemeasure_DSTATE
Structure/Union member

speedmeasure_DSTATE
Structure/Union member

class `Simulator_realistic.`**RT_MODEL_Model_2_EXPORT_Discr_T**
Bases: `_ctypes.Structure`

dwork
Structure/Union member

class `Simulator_realistic.`**Simulator_realistic**(*simulation_duration*, *hdg0*, *speed0*)
Bases: `object`

step(*delta_hdg*, *truewindheading*, *truewindheading_std*, *duration*, *precision*)

terminate()

> **Warning:** Be careful to use the libBoatModel library corresponding to your OS (Linux or Mac)

4.2.2 Realistic MDP

class `mdp_realistic.`**`mdp_realistic`**(*mdp_duration*, *hdg0*, *speed0*, *history_duration*, *mdp_step*, *delta_t*)

Bases: `object`

`computeState`(*action*, *truewindheading*, *truewindheading_std*)

Computes the mdp state when an action is applied. :param action: :param WH: :return:

`copy`()

Copy the MDP object

Returns Deep copy of the object.

Return type *MDP*

`extractSimulationData`()

`initializeMDP`(*truewindheading*, *truewindheading_std*)

`transition`(*action*, *truewindheading*, *truewindheading_std*)

4.3 Package RL

4.3.1 Policy Learner

class `policyLearning.`**`PolicyLearner`**(*state_size*, *action_size*, *batch_size*)

Bases: `object`

The aim of this class is to learn the Q-value of the action defined by a policy.

Variables

- **`state_size`** (*`int`*) – shape of the input (for convolutionnal layers).
- **`action_size`** (*`int`*) – number of action output by the network.
- **`memory`** (*`deque`*) – last-in first-out list of the batch.
- **`gamma`** (*`float`*) – discount factor.
- **`epsilon`** (*`float`*) – exploration rate.
- **`epsilon_min`** (*`float`*) – smallest exploration rate that we want to converge to.
- **`epsilon_decay`** (*`float`*) – decay factor that we apply after each replay.
- **`learning_rate`** (*`float`*) – the learning rate of the NN.
- **`model`** (*`keras.model`*) – NN, i.e the model containing the weight of the value estimator.

`act`(*state*)

Calculate the action that yields the maximum Q-value.

Parameters **`state`** – state in which we want to chose an action.

Returns the greedy action.

`actDeterministicallyUnderPolicy`(*state*)

Policy that reattaches when the angle of attack goes higher than 16 degree

Parameters **`state`** (*`np.array`*) – state for which we want to know the policy action.

Returns the policy action.

actRandomly()

actUnderPolicy(*state*)
Does the same as `actDeterministicallyUnderPolicy()` instead that the returned action is sometime taken randomly.

evaluate(*state*)
Evaluate the Q-value of the two actions in a given state using the neural network.

Parameters **state** (*np.array*) – state that we want to evaluate.

Returns The actions values as a vector.

evaluateNextAction(*stall*)
Evaluate the next action without updating the stall state in order to use it during the experience replay :param np.array state: state for which we want to know the policy action. :return: the policy action.

get_stall()

init_stall(*mean*, *mdp*)

Parameters

- **mean** –
- **mdp** –

Returns

load(*name*)
Load the weight of the network saved in the file into :ivar model :param name: name of the file containing the weights to load

remember(*state*, *action*, *reward*, *next_state*, *stall*)
Remember a transition defined by an action *action* taken from a state *state* yielding a transition to a next state *next_state* and a reward *reward*. [s, a ,r, s']

Parameters

- **state** (*np.array*) – initial state (s).
- **action** (*int*) – action (a).
- **reward** (*float*) – reward received from transition (r).
- **next_state** (*np.array*) – final state (s').
- **stall** (*int*) – flow state in the final state (s').

replay(*batch_size*)
Perform the learning on a the experience replay memory.

Parameters **batch_size** – number of samples used in the experience replay memory for the fit.

Returns the average loss over the replay batch.

save(*name*)
Save the weights of the newtork :param name: Name of the file where the weights are saved

4.3.2 DQN

class dqn.**DQNAgent**(*state_size*, *action_size*)
Bases: `object`

DQN agent

Variables

- **state_size** (*np.shape()*) – shape of the input.
- **action_size** (*int*) – number of actions.
- **memory** (*deque()*) – memory as a list.
- **gamma** (*float*) – Discount rate.
- **epsilon** (*float*) – exploration rate.
- **epsilon_min** (*float*) – minimum exploration rate.
- **epsilon_decay** (*float*) – decay of the exploration rate.
- **learning_rate** (*float*) – initial learning rate for the gradient descent
- **model** (*keras.model*) – neural network model

act (*state*)
Act ϵ-greedy with respect to the actual Q-value output by the network. :param state: State from which we want to use the network to compute the action to take. :return: a random action with probability ϵ or the greedy action with probability 1-ϵ.

actDeterministically (*state*)
Predicts the action with the highest q-value at a given state. :param state: state from which we want to know the action to make. :return:

load (*name*)
Load the weights for a defined architecture. :param name: Name of the source file.

loadModel (*name*)
Load the an architecture from source file. :param name: Name of the source file. :return:

remember (*state, action, reward, next_state*)
Remember a transition defined by an action *action* taken from a state *state* yielding a transition to a next state *next_state* and a reward *reward*. [s, a ,r, s']

Parameters

- **state** (*np.array*) – initial state (s).
- **action** (*int*) – action (a).
- **reward** (*float*) – reward received from transition (r).
- **next_state** (*np.array*) – final state (s').

replay (*batch_size*)
Core of the algorithm Q update according to the current weight of the network. :param int batch_size: Batch size for the batch gradient descent. :return: the loss after the batch gradient descent.

save (*name*)
Save the weights for a defined architecture. :param name: Name of the output file

saveModel (*name*)
Save the model's weight and architecture. :param name: Name of the output file

4.3.3 DDPG

class DDPG.**DDPGAgent** (*state_size, action_size, lower_bound, upper_bound, sess*)
Bases: `object`

The aim of this class is to learn an optimal policy via an actor-critic structure with 2 separated Convolutional Neural Networks It uses the Deep Deterministic Policy Gradient to update tha actor network. This model deals with a continuous space of actions on the rudder, chosen between lower_bound and upper_bound.

Parameters

- **state_size** (*int*) – length of the state input (for convolutionnal layers).
- **action_size** (*int*) – number of continuous action output by the network.
- **lower_bound** (*float*) – minimum value for rudder action.
- **upper_bound** (*float*) – maximum value for rudder action.
- **sess** (*tensorflow.session*) – initialized tensorflow session within which the agent will be trained.

Variables

- **memory** (*deque*) – last-in first-out list of the batch buffer.
- **gamma** (*float*) – discount factor.
- **epsilon** (*float*) – exploration rate.
- **epsilon_min** (*float*) – smallest exploration rate that we want to converge to.
- **epsilon_decay** (*float*) – decay factor that we apply after each replay.
- **actor_learning_rate** (*float*) – the learning rate of the NN of actor.
- **critic_learning_rate** (*float*) – the learning rate of the NN of critic.
- ***update_target*** (*float*) – update factor of the Neural Networks for each fit to target
- **network** (*DDPGNetworks.Network*) – tensorflow model which defines actor and critic convolutional neural networks features

act (*state*)

Calculate the action given by the Actor network's current weights

Parameters **state** – state in which we want to chose an action.

Returns the greedy action according to actor network

act_epsilon_greedy (*state*)

With probability epsilon, returns a random action between bounds With probability 1 - epsilon, returns the action given by the Actor network's current weights

Parameters **state** – state in which we want to chose an action.

Returns a random action or the action given by actor

evaluate (*state, action*)

Evaluate the Q-value of a state-action pair using the critic neural network.

Parameters

- **state** (*np.array*) – state that we want to evaluate.
- **action** (*float*) – action that we want to evaluate (has to be between permitted bounds)

Returns The continuous action value.

load (*name*)

Load the weights of the 2 networks saved in the file into :ivar network :param name: name of the file containing the weights to load

noise_decay(*e*)
Applies decay on noisy epsilon-greedy actions

Parameters **e** – current episode playing during learning

remember(*state, action, reward, next_state*)
Remember a transition defined by an action *action* taken from a state *state* yielding a transition to a next state *next_state* and a reward *reward*. [s, a ,r, s']

Parameters

- **state** (*np.array*) – initial state (s).
- **action** (*int*) – action (a).
- **reward** (*float*) – reward received from transition (r).
- **next_state** (*np.array*) – final state (s').

replay(*batch_size*)
Performs an update of both actor and critic networks on a minibatch chosen among the experience replay memory.

Parameters **batch_size** – number of samples used in the experience replay memory for the fit.

Returns the average losses for actor and critic over the replay batch.

save(*name*)
Save the weights of both of the networks into a .ckpt tensorflow session file :param name: Name of the file where the weights are saved

update_target = None
Definition of the neural networks

4.3.4 Tutorial

```
history_duration = 3  # Duration of state history [s]
mdp_step = 1  # Step between each state transition [s]
time_step = 0.1  # time step [s] <-> 10Hz frequency of data acquisition
mdp = MDP(history_duration, mdp_step, time_step)

mean = 45 * TORAD
std = 0 * TORAD
wind_samples = 10
WH = np.random.uniform(mean - std, mean + std, size=10)

hdg0=0*np.ones(10)
mdp.initializeMDP(hdg0,WH)

hdg0_rand_vec=(-4,0,2,4,6,8,18,20,21,22,24)

action_size = 2
policy_angle = 18
agent = PolicyLearner(mdp.size, action_size, policy_angle)
#agent.load("policy_learning_i18_test_long_history")
batch_size = 120

EPISODES = 500
```

```
for e in range(EPISODES):
    WH = w.generateWind()
    hdg0_rand = random.choice(hdg0_rand_vec) * TORAD
    hdg0 = hdg0_rand * np.ones(10)

    mdp.simulator.hyst.reset()

    #  We reinitialize the memory of the flow
    state = mdp.initializeMDP(hdg0, WH)
    loss_sim_list = []
    for time in range(80):
        WH = w.generateWind()
        action = agent.act(state)
        next_state, reward = mdp.transition(action, WH)
        agent.remember(state, action, reward, next_state)  # store the transition +␣
↪the state flow in the
        # final state !!
        state = next_state
        if len(agent.memory) >= batch_size:
            loss_sim_list.append(agent.replay(batch_size))
            print("time: {}, Loss = {}".format(time, loss_sim_list[-1]))
            print("i : {}".format(mdp.s[0, -1] / TORAD))
        # For data visualisation
    loss_over_simulation_time = np.sum(np.array([los    s_sim_list])[0]) / len(np.
↪array([loss_sim_list])[0])
    loss_of_episode.append(loss_over_simulation_time)
```

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

H

I

L

M

N

P

R

S

T

U

W