deer Documentation

Release 0.3

deer contributors

February 26, 2017

Contents

DeeR (Deep Reinforcement) is a python library to train an agent how to behave in a given environment so as to maximize a cumulative sum of rewards (see *What is deep reinforcement learning?*).

Here are key advantages of the library:

- You have access within a single library to techniques such as Double Q-learning, prioritized Experience Replay, Deep deterministic policy gradient (DDPG), etc.
- This package provides a general framework where observations are made up of any number of elements : scalars, vectors and frames. The belief state on which the agent is based to build the Q function or the policy is made up of any length history of each element provided in the observation.
- You can easily add up a validation phase that allows to stop the training process before overfitting. This possibility is useful when the environment is dependent on scarce data (e.g. limited time series).

In addition, the framework is made in such a way that it is easy to

- build any environment
- modify any part of the learning process
- use your favorite python-based framework to code your own neural network architecture. The provided neural network architectures are based on Keras (or pure Theano) but you may easily use others.

It is a work in progress and input is welcome. Please submit any contribution via pull request.

CHAPTER 1

What is new

Version 0.3

- Integration of different exploration/exploitation policies and possibility to easily built your own (see *Policies*)
- Integration of DDPG for continuous action spaces (see *Actor-critic learning*)
- *Naming convention for this project* and some interfaces have been updated. This may cause broken backward compatibility. In that case, make the changes to the new convention by looking at the API in this documentation or by looking at the current version of the examples.
- Additional automated tests

Version 0.2

- Standalone python package (you can simply do `pip install deer`)
- Integration of new examples environments : *The pendulum on a cart*, *PLE environment* and *Gym environment*
- Double Q-learning and prioritized Experience Replay
- Augmented documentation
- First automated tests

Future extensions:

- Add planning (e.g. MCTS based when deterministic environment)
- Several agents interacting in the same environment
- ...

CHAPTER 2

User Guide

Installation

Dependencies

This framework is tested to work under Python 2.7, and Python 3.5. It should also work with Python 3.3 and 3.4.

The required dependencies are NumPy >= 1.10, joblib >= 0.9. You also need theano >= 0.7 (lasagne is optional) or you can write your own neural network using your favorite framework.

For running some of the examples, Matplotlib >= 1.1.1 is required. You also sometimes need to install specific dependencies (e.g. for the atari games, you need to install ALE >= 0.4).

User install instructions

You can install the framework with pip:

```
pip install deer
```

For the bleeding edge version (recommanded), you can simply use

```
pip install git+git://github.com/VINF/deer.git@master
```

Developer install instructions

As a developer, you can set you up with the bleeding-edge version of DeeR with:

```
git clone -b master https://github.com/VinF/deer.git
```

Assuming you already have a python environment with `pip`, you can automatically install all the dependencies (except specific dependencies that you may need for some examples) with:

```
pip install -r requirements.txt
```

And you can install the framework as a package using the mode `develop` so that you can make modifications and test without having to re-install the package.

```
python setup.py develop
```

Tutorial

What is deep reinforcement learning?

Deep reinforcement learning is the combination of two fields:

- *Reinforcement learning (RL)* is a theory that allows an agent to learn a startegy so as to maximize a sum of cumulated (delayed) rewards from any given environment. If you are not familiar with RL, you can get up to speed easily with by Sutton and Barto.
- *Deep learning* is a branch of machine learning for regression and classification. It is particularly well suited to model high-level abstractions in data by using multiple processing layers composed of multiple non-linear transformations.

This combination allows to learn complex tasks such as playing ATARI games from high-dimensional sensory inputs. For more informations, you can refer to one of the main papers in the domain : .

How can I get started?

First, make sure you have installed the package properly by following the steps descibed in *Installation*.

The general idea of this framework is that you need to instantiate an agent (along with a q-network) and an environment. In order to perform an experiment, you also need to attach to the agent some controllers for controlling the training and the various parameters of your agent.

The environment should be built specifically for any specific task while q-networks, the DQN agent and many controllers are provided within this package.

The best to get started is to have a look at the *Examples* and in particular the two first environments that are simple to understand:

- *Toy environment with time series*
- *The pendulum on a cart*

If you find something that is not yet implemented and if you wish to contribute, you can check the section *Development*.

Any Question?

You can raise questions about the DeeR project on github :

Examples

You can find these examples at the . For each example at least two files are provided:

- A launcher file (whose name usually starts by `run_`).
- An environnement file (whose name usually ends by `_env`).

The launcher file performs different actions:

- It instantiates the environment, the agent (along with a q-network).
- It binds controllers to the agent
- it finally runs the experiment

Examples are better than precepts and the best is to get started with the following examples (with the simplest examples listed first)

Toy environment with time series

Description of the environement

This environment simulates the possibility of buying or selling a good. The agent can either have one unit or zero unit of that good. At each transaction with the market, the agent obtains a reward equivalent to the price of the good when selling it and the opposite when buying. In addition, a penalty of 0.5 (negative reward) is added for each transaction.

The price pattern is made by repeating the following signal plus a random constant between 0 and 3:

Let's now see how this environement is built by looking into the file `Toy_env.py` in . It is important to note that any environment derives from the base class *Environment* and you can refer to it in order to understand the required methods and their usage.

How to run

A minimalist way of running this example can be found in the file `run_toy_env_simple.py` in .

- First, we need to import the agent, the Q-network, the environement and some controllers

```
import numpy as np

from deer.agent import NeuralAgent
from deer.q_networks.q_net_theano import MyQNetwork
from Toy_env import MyEnv as Toy_env
import deer.experiment.base_controllers as bc
```

- Then we instantiate the different elements as follows:

```
if __name__ == "__main__":

    rng = np.random.RandomState(123456)

    # --- Instantiate environment ---
    env = Toy_env(rng)

    # --- Instantiate qnetwork ---
    qnetwork = MyQNetwork(
        environment=env,
        random_state=rng)

    # --- Instantiate agent ---
    agent = NeuralAgent(
        env,
        qnetwork,
        random_state=rng)

    # --- Bind controllers to the agent ---
    # Before every training epoch, we want to print a summary of the agent's epsilon, discount and
    # learning rate as well as the training epoch number.
    agent.attach(bc.VerboseController())

    # During training epochs, we want to train the agent after every action it takes.
    # Plus, we also want to display after each training episode (!= than after every training) the a
```

```
    # residual and the average of the V values obtained during the last episode.
    agent.attach(bc.TrainerController())

    # All previous controllers control the agent during the epochs it goes through. However, we want
    # "test epoch" between each training epoch. We do not want these test epoch to interfere with the
    # agent. Therefore, we will disable these controllers for the whole duration of the test epochs
    # way, using the controllersToDisable argument of the InterleavedTestEpochController. The value
    # is a list of the indexes of all controllers to disable, their index reflecting in which order
    agent.attach(bc.InterleavedTestEpochController(
        epoch_length=500,
        controllers_to_disable=[0, 1]))

    # --- Run the experiment ---
    agent.run(n_epochs=100, epoch_length=1000)
```

Results

Navigate to the folder `examples/toy_env/` in a terminal window. The example can then be run by using

```
python run_toy_env_simple.py
```

You can also choose the full version of the launcher that specifies the hyperparameters for better performance.

```
python run_toy_env.py
```

Every 10 epochs, a graph is saved in the 'toy_env' folder. You can then see that kind of behaviour for the test policy at the end of the training:

In this graph, you can see that the agent has successfully learned to take advantage of the price pattern to buy when it is low and to sell when it is high. This example is of course easy due to the fact that the patterns are very systematic which allows the agent to successfuly learn it. It is important to note that the results shown are made on a validation set that is different from the training and we can see that learning generalizes well. For instance, the action of buying at time step 7 and 16 is the expected result because in average this will allow to make profit since the agent has no information on the future.

Using Convolutions VS LSTM's

So far, the neural network was build by using a convolutional architecture as follows:

The neural nework processes time series thanks to a set of convolutions layers. The output of the convolutions as well as the other inputs are followed by fully connected layers and the ouput layer.

When working with deep reinforcement learning, it is also possible to work with LSTM's (see for instance this)

If you want to use LSTM's architecture, you can import the following libraries

```
from deer.q_networks.q_net_keras import MyQNetwork
from deer.q_networks.NN_keras_LSTM import NN as NN_keras
```

and then instanciate the qnetwork by specifying the 'neural_network' as follows:

```
qnetwork = MyQNetwork(
    env,
    neural_network=NN_keras)
```

The pendulum on a cart

Description

The environment simulates the behavior of an inverted pendulum. The theoretical system with its equations are as described in :

- A cart of mass M that can move horizontally;
- A pole of mass m and length l attached to the cart, with θ in $[0, -\pi]$ for the lefthand plane, and $[0, \pi]$ for the righthand side. We are supposing that the cart is moving on a rail and the pole can go under it.

The goal of the agent is to balance the pole above its supporting cart ($\theta = 0$), by displacing the cart left or right - thus, 2 actions are possible. To do so, the environment communicates to the agent:

- A vector (position, speed, angle, angular speed);
- The reward associated to the action chosen by the agent.

Results

In a terminal windown go to the folder `examples/pendulum`. The example can then be run with

```
python run_pendulum.py
```

Here are the outputs of the agent after respectively 20 and 70 learning epochs, with 1000 steps in each. We clearly see the final success of the agent in controlling the inverted pendulum.

Note: a MP4 is generated every *PERIOD_BTW_SUMMARY_PERFS* epochs and you need the [FFmpeg](https://www.ffmpeg.org/) library to do so. If you do not want to install this library or to generate the videos, just set *PERIOD_BTW_SUMMARY_PERFS* = *-1*.

Details on the implementation

The main focus in the environment is to implement *act(self, action)* which specifies how the cart-pole system behaves in response to an input action. So first, we transcript the physical laws that rule the motion of the pole and the cart. The simulation timestep of the agent is $\Delta_t = 0.02$ second. But we discretize this value even further in *act(self, action)*, in order to obtain dynamics that are closer to the exact differential equations. Secondly, we chose the reward function as the sum of :

- $-|\theta|$ such that the agent receives 0 when the pole is standing up, and a negative reward proportional to the angle otherwise.
- $-\frac{|x|}{2}$ such that the agent receives a negative reward when it is far from $x = 0$.

Gym environment

Some examples are also provided with the .

Here is the resulting policy for the mountain car example:

Here is the resulting policy for the pendulum example:

Two storage devices environment

Description of the environement

This example simulates the operation of a realistic micro-grid (such as a smart home for instance) that is not connected to the main utility grid (off-grid) and that is provided with PV panels, batteries and hydrogen storage. The battery has the advantage that it is not limited in instaneous power that it can provide or store. The hydrogen storage has the advantage that is can store very large quantity of energy.

```
python run_MG_two_storage_devices
```

This example uses the environment defined in MG_two_storage_devices_env.py. The agent can either choose to store in the long term storage or take energy out of it while the short term storage handle at best the lack or surplus of energy by discharging itself or charging itself respectively. Whenever the short term storage is empty and cannot handle the net demand a penalty (negative reward) is obtained equal to the value of loss load set to 2euro/kWh.

The state of the agent is made up of an history of two to four punctual observations:

- Charging state of the short term storage (0 is empty, 1 is full)
- Production and consumption (0 is no production or consumption, 1 is maximal production or consumption)
- (Distance to equinox)
- (Predictions of future production : average of the production for the next 24 hours and 48 hours)

Two actions are possible for the agent:

- Action 0 corresponds to discharging the long-term storage
- Action 1 corresponds to charging the long-term storage

More information can be found in the paper to be published : Deep Reinforcement Learning Solutions for Energy Microgrids Management, Vincent François-Lavet, David Taralla, Damien Ernst, Raphael Fonteneau

Annex to the paper

PV production and consumption profiles

Solar irradiance varies throughout the year depending on the seasons, and it also varies throughout the day depending on the weather and the position of the sun in the sky relative to the PV panels. The main distinction between these profiles is the difference between summer and winter PV production. In particular, production varies with a factor 1:5 between winter and summer as can be seen from the measurements of PV panels production for a residential customer located in Belgium in the figures below.

Fig. 2.1: Total energy produced per month

Fig. 2.2: Typical production in winter

Fig. 2.3: Typical production in summer

A simple residential consumption profile is considered with a daily average consumption of 18kWh (see figure below).

Fig. 2.4: Representative residential consumption profile

Main microgrid parameters

Table 2.1: Data used for the PV panels

cost	c^{PV}	$1euro/W_p$
Efficiency	η^{PV}	18%
Life time	L^{PV}	$20years$

Table 2.2: Data used for the $LiFePO_4$ battery

cost	c^B	$500euro/kWh$
discharge efficiency	η_0^B	90%
charge efficiency	ζ_0^B	90%
Maximum instantaneous power	P^B	$> 10kW$
Life time	L^B	$20years$

Table 2.3: Data used for the Hydrogen storage device

cost	c^{H_2}	$14euro/W_p$
discharge efficiency	$\eta_0^{H_2}$	65%
charge efficiency	$\zeta_0^{H_2}$	65%
Life time	L^{H_2}	$20years$

Table 2.4: Data used for reward function

cost endured per kWh not supplied within the microgrid	k	$2euro/kWh$
revenue/cost per kWh of hydrogen produced/used	k^{H_2}	$0.1euro/kWh$

ALE environment

This environment is an interface with the that simulates any ATARI game. The hyper-parameters used in the example provided aim to simulate as closely as possible the following paper : Mnih, Volodymyr, et al. “Human-level control through deep reinforcement learning.” Nature 518.7540 (2015): 529-533.

Some changes are still necessary to obtain the same performances.

PLE environment

This environment is an interface with the .

Development

DeeR is a work in progress and contributions are welcome via pull request.

For more information, you can check out this link : .

You should also make sure that you install the repository approriately for development (see *Developer install instructions*).

Guidelines for this project

Here are a few guidelines for this project.

- Simplicity: Be easy to use but also easy to understand when one digs into the code. Any additional code should be justified by the usefulness of the feature.
- Modularity: The user should be able to easily use its own code with any part of the deer framework (probably at the exception of the core of agent.py that is coded in a very general way).

These guidelines come of course in addition to all good practices for open source development.

Naming convention for this project

- All classes and methods have word boundaries using medial capitalization. Classes are written with UpperCamelCase and methods are written with lowerCamelCase respectively. Example: “two words” is rendered as “TwoWords” for the UpperCamelCase (classes) and “twoWords” for the lowerCamelCase (methods).
- All attributes and variables have words separated by underscores. Example: “two words” is rendered as “two_words”
- If a variable is intended to be ‘private’, it is prefixed by an underscore.

CHAPTER 3

API reference

If you are looking for information on a specific function, class or method, this API is for you.

Agent

This module contains classes used to define the standard behavior of the agent. It relies on the controllers, the chosen training/test policy and the learning algorithm to specify its behavior in the environment.

NeuralAgent(environment, q_network[, ...])	The NeuralAgent class wraps a deep Q-network for training and testing in a given e
DataSet(env[, random_state, max_size, ...])	A replay memory consisting of circular buffers for observations, actions, rewards ar

Detailed description

class deer.agent.**NeuralAgent**(*environment*, *q_network*, *replay_memory_size=1000000*, *replay_start_size=None*, *batch_size=32*, *random_state=<mtrand.RandomState object>*, *exp_priority=0*, *train_policy=None*, *test_policy=None*, *only_full_history=True*)

The NeuralAgent class wraps a deep Q-network for training and testing in a given environment.

Attach controllers to it in order to conduct an experiment (when to train the agent, when to test,...).

Parameters **environment** : object from class Environment

The environment in which the agent interacts

q_network : object from class QNetwork

The q_network associated to the agent

replay_memory_size : int

Size of the replay memory. Default : 1000000

replay_start_size : int

Number of observations (=number of time steps taken) in the replay memory before starting learning. Default: minimum possible according to environment.inputDimensions().

batch_size : int

Number of tuples taken into account for each iteration of gradient descent. Default : 32

random_state : numpy random number generator

Default : random seed.

exp_priority : float

The exponent that determines how much prioritization is used, default is 0 (uniform priority). One may check out Schaul et al. (2016) - Prioritized Experience Replay.

train_policy : object from class Policy

Policy followed when in training mode (mode -1)

test_policy : object from class Policy

Policy followed when in other modes than training (validation and test modes)

only_full_history : boolean

Whether we wish to train the neural network only on full histories or we wish to fill with zeroes the observations before the beginning of the episode

Methods

attach(controller)	
avgBellmanResidual()	Returns the average training loss on the epoch
avgEpisodeVValue()	Returns the average V value on the episode (on time steps where a non-random act
bestAction()	Returns the best Action
detach(controllerIdx)	
discountFactor()	Get the discount factor
dumpNetwork(fname[, nEpoch])	Dump the network
learningRate()	Get the learning rate
mode()	
overrideNextAction(action)	Possibility to override the chosen action.
resumeTrainingMode()	
run(n_epochs, epoch_length)	This function encapsulates the whole process of the learning.
setControllersActive(toDisable, active)	Activate controller
setDiscountFactor(df)	Set the discount factor
setLearningRate(lr)	Set the learning rate for the gradient descent
setNetwork(fname[, nEpoch])	Set values into the network
startMode(mode, epochLength)	
summarizeTestPerformance()	
totalRewardOverLastTest()	Returns the average sum of rewards per episode and the number of episode
train()	This function selects a random batch of data (with self._dataset.randomBatch) and

avgBellmanResidual()
Returns the average training loss on the epoch

avgEpisodeVValue()
Returns the average V value on the episode (on time steps where a non-random action has been taken)

bestAction()
Returns the best Action

discountFactor()
Get the discount factor

dumpNetwork(*fname*, *nEpoch=-1*)
Dump the network

Parameters **fname** : string

Name of the file where the network will be dumped

nEpoch : int

Epoch number (Optional)

learningRate()
Get the learning rate

overrideNextAction(*action*)
Possibility to override the chosen action. This possibility should be used on the signal OnActionChosen.

run(*n_epochs*, *epoch_length*)
This function encapsulates the whole process of the learning. It starts by calling the controllers method "onStart", Then it runs a given number of epochs where an epoch is made up of one or many episodes (called with agent._runEpisode) and where an epoch ends up after the number of steps reaches the argument "epoch_length". It ends up by calling the controllers method "end".

Parameters **n_epochs** : number of epochs

int

epoch_length : maximum number of steps for a given epoch

int

setControllersActive(*toDisable*, *active*)
Activate controller

setDiscountFactor(*df*)
Set the discount factor

setLearningRate(*lr*)
Set the learning rate for the gradient descent

setNetwork(*fname*, *nEpoch=-1*)
Set values into the network

Parameters **fname** : string

Name of the file where the values are

nEpoch : int

Epoch number (Optional)

totalRewardOverLastTest()
Returns the average sum of rewards per episode and the number of episode

train()
This function selects a random batch of data (with self._dataset.randomBatch) and performs a Q-learning iteration (with self._network.train).

class deer.agent.**DataSet**(*env*, *random_state=None*, *max_size=1000*, *use_priority=False*, *only_full_history=True*)
A replay memory consisting of circular buffers for observations, actions, rewards and terminals.

Methods

actions()	Get all actions currently in the replay memory, ordered by time where they were
addSample(obs, action, reward, is_terminal, ...)	Store a (observation[for all subjects], action, reward, is_terminal) in the dataset
observations()	Get all observations currently in the replay memory, ordered by time where they
randomBatch(size, use_priority)	Return corresponding states, actions, rewards, terminal status, and next_states f
rewards()	Get all rewards currently in the replay memory, ordered by time where they were
terminals()	Get all terminals currently in the replay memory, ordered by time where they w
updatePriorities(priorities, rndValidIndices)	

actions()
Get all actions currently in the replay memory, ordered by time where they were taken.

addSample(*obs*, *action*, *reward*, *is_terminal*, *priority*)
Store a (observation[for all subjects], action, reward, is_terminal) in the dataset. Parameters ———— obs : ndarray

An ndarray(dtype='object') where obs[s] corresponds to the observation made on subject s before the agent took action [action].

action [int] The action taken after having observed [obs].

reward [float] The reward associated to taking this [action].

is_terminal [bool] Tells whether [action] lead to a terminal state (i.e. corresponded to a terminal transition).

priority [float] The priority to be associated with the sample

observations()
Get all observations currently in the replay memory, ordered by time where they were observed.

observations[s][i] corresponds to the observation made on subject s before the agent took actions()[i].

randomBatch(*size*, *use_priority*)
Return corresponding states, actions, rewards, terminal status, and next_states for size randomly chosen transitions. Note that if terminal[i] == True, then next_states[s][i] == np.zeros_like(states[s][i]) for each subject s.

Parameters **size** : int

Number of transitions to return.

Returns

———-

states : ndarray

An ndarray(size=number_of_subjects, dtype='object), where states[s] is a 2+D matrix of dimensions size x s.memorySize x "shape of a given observation for this subject". States were taken randomly in the data with the only constraint that they are complete regarding the histories for each observed subject.

actions : ndarray

An ndarray(size=number_of_subjects, dtype='int32') where actions[i] is the action taken after having observed states[:][i].

rewards : ndarray

An ndarray(size=number_of_subjects, dtype='float32') where rewards[i] is the reward obtained for taking actions[i-1].

next_states : ndarray

Same structure than states, but next_states[s][i] is guaranteed to be the information concerning the state following the one described by states[s][i] for each subject s.

terminals : ndarray

An ndarray(size=number_of_subjects, dtype='bool') where terminals[i] is True if actions[i] lead to terminal states and False otherwise

Throws

——-

SliceError If a batch of this size could not be built based on current data set (not enough data or all trajectories are too short).

rewards()
Get all rewards currently in the replay memory, ordered by time where they were received.

terminals()
Get all terminals currently in the replay memory, ordered by time where they were observed.

terminals[i] is True if actions()[i] lead to a terminal state (i.e. corresponded to a terminal transition), and False otherwise.

updatePriorities(*priorities*, *rndValidIndices*)

Controller

This file defines the base Controller class and some presets controllers that you can use for controlling the training and the various parameters of your agents.

Controllers can be attached to an agent using the agent's `attach(Controller)` method. The order in which controllers are attached matters. Indeed, if controllers C1, C2 and C3 were attached in this order and C1 and C3 both listen to the onEpisodeEnd signal, the onEpisodeEnd() method of C1 will be called *before* the onEpisodeEnd() method of C3, whenever an episode ends.

Controller()	A base controller that does nothing when receiving the various signals emitte
LearningRateController([...])	A controller that modifies the learning rate periodically upon epochs end.
EpsilonController([initial_e, e_decays, ...])	A controller that modifies the probability "epsilon" of taking a random action
DiscountFactorController([...])	A controller that modifies the q-network discount periodically.
TrainerController([evaluate_on, ...])	A controller that makes the agent train on its current database periodically.
InterleavedTestEpochController([id, ...])	A controller that interleaves a test epoch between training epochs of the agent
FindBestController([validationID, testID, ...])	A controller that finds the neural net performing at best in validation mode (i.

Detailed description

class deer.experiment.base_controllers.**Controller**
A base controller that does nothing when receiving the various signals emitted by an agent. This class should be the base class of any controller you would want to define.

Methods

onActionChosen(agent, action)	Called whenever the agent has chosen an action.
onActionTaken(agent)	Called whenever the agent has taken an action on its environment.
onEnd(agent)	Called when the agent has finished processing all its epochs, just before returni
onEpisodeEnd(agent, terminal_reached, reward)	Called whenever the agent ends an episode, just after this episode ended and be
onEpochEnd(agent)	Called whenever the agent ends an epoch, just after the last episode of this epoc
onStart(agent)	Called when the agent is going to start working (before anything else).
setActive(active)	Activate or deactivate this controller.

onActionChosen(*agent*, *action*)
Called whenever the agent has chosen an action.

This occurs after the agent state was updated with the new observation it made, but before it applied this action on the environment and before the total reward is updated.

onActionTaken(*agent*)
Called whenever the agent has taken an action on its environment.

This occurs after the agent applied this action on the environment and before terminality is evaluated. This is called only once, even in the case where the agent skip frames by taking the same action multiple times. In other words, this occurs just before the next observation of the environment.

onEnd(*agent*)
Called when the agent has finished processing all its epochs, just before returning from its run() method.

onEpisodeEnd(*agent*, *terminal_reached*, *reward*)
Called whenever the agent ends an episode, just after this episode ended and before any onEpochEnd() signal could be sent.

Parameters **agent** : NeuralAgent

The agent firing the event

terminal_reached : bool

Whether the episode ended because a terminal transition occured. This could be False if the episode was stopped because its step budget was exhausted.

reward : float

The reward obtained on the last transition performed in this episode.

onEpochEnd(*agent*)
Called whenever the agent ends an epoch, just after the last episode of this epoch was ended and after any onEpisodeEnd() signal was processed.

Parameters **agent** : NeuralAgent

The agent firing the event

onStart(*agent*)
Called when the agent is going to start working (before anything else).

This corresponds to the moment where the agent's run() method is called.

Parameters **agent** : NeuralAgent

The agent firing the event

setActive(*active*)
Activate or deactivate this controller.

A controller should not react to any signal it receives as long as it is deactivated. For instance, if a controller maintains a counter on how many episodes it has seen, this counter should not be updated when this controller is disabled.

class deer.experiment.base_controllers.**LearningRateController**(*initial_learning_rate=0.005*, *learning_rate_decay=1.0*, *periodicity=1*)

Bases: *deer.experiment.base_controllers.Controller*

A controller that modifies the learning rate periodically upon epochs end.

Parameters **initial_learning_rate** : float

The learning rate upon agent start

learning_rate_decay : float

The factor by which the previous learning rate is multiplied every [periodicity] epochs.

periodicity : int

How many epochs are necessary before an update of the learning rate occurs

Methods

onActionChosen(agent, action)	Called whenever the agent has chosen an action.
onActionTaken(agent)	Called whenever the agent has taken an action on its environment.
onEnd(agent)	Called when the agent has finished processing all its epochs, just before returni
onEpisodeEnd(agent, terminal_reached, reward)	Called whenever the agent ends an episode, just after this episode ended and be
onEpochEnd(agent)	
onStart(agent)	
setActive(active)	Activate or deactivate this controller.

class deer.experiment.base_controllers.**EpsilonController**(*initial_e=1.0*, *e_decays=10000*, *e_min=0.1*, *evaluate_on='action'*, *periodicity=1*, *reset_every='none'*)

Bases: *deer.experiment.base_controllers.Controller*

A controller that modifies the probability "epsilon" of taking a random action periodically.

Parameters **initial_e** : float

Start epsilon

e_decays : int

How many updates are necessary for epsilon to reach eMin

e_min : float

End epsilon

evaluate_on : str

After what type of event epsilon shoud be updated periodically. Possible values: 'action', 'episode', 'epoch'.

periodicity : int

How many [evaluateOn] are necessary before an update of epsilon occurs

reset_every : str

After what type of event epsilon should be reset to its initial value. Possible values: 'none', 'episode', 'epoch'.

Methods

onActionChosen(agent, action)	
onActionTaken(agent)	Called whenever the agent has taken an action on its environment.
onEnd(agent)	Called when the agent has finished processing all its epochs, just before returni
onEpisodeEnd(agent, terminal_reached, reward)	
onEpochEnd(agent)	
onStart(agent)	
setActive(active)	Activate or deactivate this controller.

class deer.experiment.base_controllers.**DiscountFactorController**(*initial_discount_factor=0.9*, *discount_factor_growth=1.0*, *discount_factor_max=0.99*, *periodicity=1*)

Bases: *deer.experiment.base_controllers.Controller*

A controller that modifies the q-network discount periodically. More informations in : Francois-Lavet Vincent et al. (2015) - How to Discount Deep Reinforcement Learning: Towards New Dynamic Strategies (http://arxiv.org/abs/1512.02011).

Parameters **initial_discount_factor** : float

Start discount

discount_factor_growth : float

The factor by which the previous discount is multiplied every [periodicity] epochs.

discount_factor_max : float

Maximum reachable discount

periodicity : int

How many training epochs are necessary before an update of the discount occurs

Methods

onActionChosen(agent, action)	Called whenever the agent has chosen an action.
onActionTaken(agent)	Called whenever the agent has taken an action on its environment.
onEnd(agent)	Called when the agent has finished processing all its epochs, just before returni
onEpisodeEnd(agent, terminal_reached, reward)	Called whenever the agent ends an episode, just after this episode ended and be
onEpochEnd(agent)	
onStart(agent)	
setActive(active)	Activate or deactivate this controller.

class deer.experiment.base_controllers.**TrainerController**(*evaluate_on='action'*, *periodicity=1*, *show_episode_avg_V_value=True*, *show_avg_Bellman_residual=True*)

Bases: *deer.experiment.base_controllers.Controller*

A controller that makes the agent train on its current database periodically.

Parameters **evaluate_on** : str

After what type of event the agent shoud be trained periodically. Possible values: 'action', 'episode', 'epoch'. The first training will occur after the first occurence of [evaluateOn].

periodicity : int

How many [evaluateOn] are necessary before a training occurs _show_avg_Bellman_residual [bool] - Whether to show an informative message after each episode end (and after a training if [evaluateOn] is 'episode') about the average bellman residual of this episode

show_episode_avg_V_value : bool

Whether to show an informative message after each episode end (and after a training if [evaluateOn] is 'episode') about the average V value of this episode

Methods

onActionChosen(agent, action)	Called whenever the agent has chosen an action.
onActionTaken(agent)	
onEnd(agent)	Called when the agent has finished processing all its epochs, just before returni
onEpisodeEnd(agent, terminal_reached, reward)	
onEpochEnd(agent)	
onStart(agent)	
setActive(active)	Activate or deactivate this controller.

class deer.experiment.base_controllers.**InterleavedTestEpochController**(*id=0*, *epoch_length=500*, *controllers_to_disable=[]*, *periodicity=2*, *show_score=True*, *summarize_every=10*)

Bases: *deer.experiment.base_controllers.Controller*

A controller that interleaves a test epoch between training epochs of the agent.

Parameters **id** : int

The identifier (>= 0) of the mode each test epoch triggered by this controller will belong to. Can be used to discriminate between datasets in your Environment subclass (this is the argument that will be given to your environment's reset() method when starting the test epoch).

epoch_length : float

The total number of transitions that will occur during a test epoch. This means that this epoch could feature several episodes if a terminal transition is reached before this budget is exhausted.

controllers_to_disable : list of int

A list of controllers to disable when this controller wants to start a test epoch. These same controllers will be reactivated after this controller has finished dealing with its test epoch.

periodicity : int

How many epochs are necessary before a test epoch is ran (these controller's epochs included: "1 test epoch on [periodicity] epochs"). Minimum value: 2.

show_score : bool

Whether to print an informative message on stdout at the end of each test epoch, about the total reward obtained in the course of the test epoch.

summarize_every : int

How many of this controller's test epochs are necessary before the attached agent's summarizeTestPerformance() method is called. Give a value <= 0 for "never". If > 0, the first call will occur just after the first test epoch.

Methods

`onActionChosen`(agent, action)	Called whenever the agent has chosen an action.
`onActionTaken`(agent)	Called whenever the agent has taken an action on its environment.
`onEnd`(agent)	Called when the agent has finished processing all its epochs, just before returni
`onEpisodeEnd`(agent, terminal_reached, reward)	Called whenever the agent ends an episode, just after this episode ended and be
`onEpochEnd`(agent)	
`onStart`(agent)	
`setActive`(active)	Activate or deactivate this controller.

class `deer.experiment.base_controllers.`**`FindBestController`**(*validationID=0*, *testID=None*, *unique_fname='nnet'*)

Bases: *deer.experiment.base_controllers.Controller*

A controller that finds the neural net performing at best in validation mode (i.e. for mode = [validationID]) and computes the associated generalization score in test mode (i.e. for mode = [testID], and this only if [testID] is different from None). This controller should never be disabled by InterleavedTestControllers as it is meant to work in conjunction with them.

At each epoch end where this controller is active, it will look at the current mode the agent is in.

If the mode matches [validationID], it will take the total reward of the agent on this epoch and compare it to its current best score. If it is better, it will ask the agent to dump its current nnet on disk and update its current best score. In all cases, it saves the validation score obtained in a vector.

If the mode matches [testID], it saves the test (= generalization) score in another vector. Note that if [testID] is None, no test mode score are ever recorded.

At the end of the experiment (onEnd), if active, this controller will print information about the epoch at which the best neural net was found together with its generalization score, this last information shown only if [testID] is different from None. Finally it will dump a dictionnary containing the data of the plots ({n: number of epochs

elapsed, ts: test scores, vs: validation scores}). Note that if [testID] is None, the value dumped for the 'ts' key is [].

Parameters **validationID** : int

See synopsis

testID : int

See synopsis

unique_fname : str

A unique filename (basename for score and network dumps).

Methods

`onActionChosen`(agent, action)	Called whenever the agent has chosen an action.
`onActionTaken`(agent)	Called whenever the agent has taken an action on its environment.
`onEnd`(agent)	
`onEpisodeEnd`(agent, terminal_reached, reward)	Called whenever the agent ends an episode, just after this episode ended and be
`onEpochEnd`(agent)	
`onStart`(agent)	Called when the agent is going to start working (before anything else).
`setActive`(active)	Activate or deactivate this controller.

Environment

Detailed description

class `deer.base_classes.`**Environment**

All your Environment classes should inherit this interface.

The environment defines the dynamics and the reward signal that the agent observes when interacting with it.

An agent sees at any time-step from the environment a collection of observable elements. Observing the environment at time t thus corresponds to obtaining a punctual observation for each of these elements. According to the control problem to solve, it might be useful for the agent to not only take action based on the current punctual observations but rather on a collection of the last punctual observations. In this framework, it's the environment that defines the number of each punctual observation to be considered.

Different "modes" are used in this framework to allow the environment to have different dynamics and/or reward signal. For instance, in training mode, only a part of the dynamics may be available so that it is possible to see how well the agent generalizes to a slightly different one.

Methods

`act`(action)	Applies the agent action [action] on the environment.
`end`()	Optional hook called at the end of all epochs
`inTerminalState`()	Tells whether the environment reached a terminal state after the last transition (i.e.
`inputDimensions`()	Gets the shape of the input space for this environment.
`nActions`()	Gets the number of different actions that can be taken on this environment.
`observationType`(subject)	Gets the most inner type (np.uint8, np.float32, ...) of [subject].

Table 3.12 – continued from previous page

observe()	Gets a list of punctual observations on all subjects composing this environment.
reset(mode)	Resets the environment and put it in mode [mode].
summarizePerformance(test_data_set)	Optional hook that can be used to show a summary of the performance of the agent on

act (*action*)

Applies the agent action [action] on the environment.

Parameters **action** : int

The action selected by the agent to operate on the environment. Should be an identifier included between 0 included and nActions() excluded.

end ()

Optional hook called at the end of all epochs

inTerminalState ()

Tells whether the environment reached a terminal state after the last transition (i.e. the last transition that occured was terminal).

As the majority of control tasks considered have no end (a continuous control should be operated), by default this returns always False. But in the context of a video game for instance, terminal states can occurs and these cases this method should be overriden.

Returns **isTerminal** : bool

inputDimensions ()

Gets the shape of the input space for this environment.

This returns a list whose length is the number of subjects observed on the environment. Each element of the list is a tuple whose content and size depends on the type of data observed: the first integer is always the history size (or batch size) for observing this subject and the rest describes the shape of a single observation on this subject: - () or (1,) means each observation on this subject is a single number and the history size is 1 (= no history) - (N,) means each observation on this subject is a single number and the history size is N - (N, M) means each observation on this subject is a vector of length M and the history size is N - (N, M1, M2) means each observation on this subject is a matrix with M1 rows and M2 columns and the history size is N

nActions ()

Gets the number of different actions that can be taken on this environment. It can be either an integer in the case of a finite discrete number of actions or it can be a list of couples [min_action_value,max_action_value] for a continuous action space

observationType (*subject*)

Gets the most inner type (np.uint8, np.float32, ...) of [subject].

Parameters **subject** : int

The subject

observe ()

Gets a list of punctual observations on all subjects composing this environment.

This returns a list where element i is a punctual observation on subject i. You will notice that the history of observations on this subject is not returned; only the very last observation. Each element is thus either a number, vector or matrix and not a succession of numbers, vectors and matrices.

See the documentation of batchDimensions() for more information about the shape of the observations according to their mathematical representation (number, vector or matrix).

reset(*mode*)
Resets the environment and put it in mode [mode]. This function is called when beginning every new episode.

The [mode] can be used to discriminate for instance between an agent which is training or trying to get a validation or generalization score. The mode the environment is in should always be redefined by resetting the environment using this method, meaning that the mode should be preserved until the next call to reset().

Parameters **mode** : int

The mode to put the environment into. Mode "-1" is reserved and always means "training".

summarizePerformance(*test_data_set*)
Optional hook that can be used to show a summary of the performance of the agent on the environment in the current mode.

Parameters **test_data_set** : agent.DataSet

The dataset maintained by the agent in the current mode, which contains observations, actions taken and rewards obtained, as well as wether each transition was terminal or not. Refer to the documentation of agent.DataSet for more information.

Learning algorithms

Q-learning

deer.base_classes.QNetwork(environment, ...)	All the Q-networks and actor-critic networks should inherit this inter
deer.q_networks.q_net_theano.MyQNetwork(...)	Deep Q-learning network using Theano
deer.q_networks.q_net_keras.MyQNetwork(...)	Deep Q-learning network using Keras (with any backend)

Actor-critic learning

deer.q_networks.AC_net_keras.MyACNetwork(...)	Deep Q-learning network using Keras with Tensorflow backend

Detailed description

class deer.base_classes.**QNetwork**(*environment*, *batch_size*)
All the Q-networks and actor-critic networks should inherit this interface.

Parameters **environment** : object from class Environment

The environment linked to the Q-network

batch_size : int

Number of tuples taken into account for each iteration of gradient descent

Methods

chooseBestAction

Continued on next page

Table 3.15 – continued from previous page

discountFactor
learningRate
qValues
setDiscountFactor
setLearningRate
train

chooseBestAction(*state*)
Get the best action for a belief state

discountFactor()
Getting the discount factor

learningRate()
Getting the learning rate

qValues(*state*)
Get the q value for one belief state

setDiscountFactor(*df*)
Setting the discount factor

Parameters **df** : float

The discount factor that has to bet set

setLearningRate(*lr*)
Setting the learning rate

Parameters **lr** : float

The learning rate that has to bet set

train(*states*, *actions*, *rewards*, *nextStates*, *terminals*)
This method performs the Bellman iteration for one batch of tuples.

class deer.q_networks.q_net_theano.**MyQNetwork**(*environment*, *rho=0.9*, *rms_epsilon=0.0001*, *momentum=0*, *clip_delta=0*, *freeze_interval=1000*, *batch_size=32*, *update_rule='rmsprop'*, *random_state=<mtrand.RandomState object>*, *double_Q=False*, *neural_network=<class 'deer.q_networks.NN_theano.NN'>*)

Bases: deer.base_classes.QNetwork.QNetwork

Deep Q-learning network using Theano

Parameters **environment** : object from class Environment

rho : float

Parameter for rmsprop. Default : 0.9

rms_epsilon : float

Parameter for rmsprop. Default : 0.0001

momentum : float

Not implemented.

clip_delta : float

If > 0, the squared loss is linear past the clip point which keeps the gradient constant. Default : 0

freeze_interval : int

Period during which the target network is freezed and after which the target network is updated. Default : 1000

batch_size : int

Number of tuples taken into account for each iteration of gradient descent. Default : 32

update_rule: str

{sgd,rmsprop}. Default : rmsprop

random_state : numpy random number generator

Default : random seed.

double_Q : bool

Activate or not the DoubleQ learning : not implemented yet. Default : False More informations in : Hado van Hasselt et al. (2015) - Deep Reinforcement Learning with Double Q-learning.

neural_network : object

default is deer.qnetworks.NN_theano

Methods

`chooseBestAction`
`discountFactor`
`getAllParams`
`learningRate`
`qValues`
`setAllParams`
`setDiscountFactor`
`setLearningRate`
`train`

`chooseBestAction`(*state*)
Get the best action for a belief state

Returns **The best action** : int

`qValues`(*state_val*)
Get the q values for one belief state

Returns The q values for the provided belief state

`train`(*states_val*, *actions_val*, *rewards_val*, *next_states_val*, *terminals_val*)
Train one batch.

1. Set shared variable in states_shared, next_states_shared, actions_shared, rewards_shared, terminals_shared
2. perform batch training

Parameters **states_val** : list of batch_size * [list of max_num_elements* [list of k * [element 2D,1D or scalar]])

actions_val : b x 1 numpy array of integers

rewards_val : b x 1 numpy array

next_states_val : list of batch_size * [list of max_num_elements* [list of k * [element 2D,1D or scalar]])

terminals_val : b x 1 numpy boolean array

Returns Average loss of the batch training (RMSE)

Individual (square) losses for each tuple

class deer.q_networks.AC_net_keras.**MyACNetwork**(*environment*, *rho=0.9*, *rms_epsilon=0.0001*, *momentum=0*, *clip_delta=0*, *freeze_interval=1000*, *batch_size=32*, *update_rule='rmsprop'*, *random_state=<mtrand.RandomState object>*, *double_Q=False*, *neural_network_critic=<class 'deer.q_networks.NN_keras.NN'>*, *neural_network_actor=<class 'deer.q_networks.NN_keras.NN'>*)

Bases: deer.base_classes.QNetwork.QNetwork

Deep Q-learning network using Keras with Tensorflow backend

Parameters **environment** : object from class Environment

rho : float

Parameter for rmsprop. Default : 0.9

rms_epsilon : float

Parameter for rmsprop. Default : 0.0001

momentum : float

Default : 0

clip_delta : float

Not implemented.

freeze_interval : int

Period during which the target network is freezed and after which the target network is updated. Default : 1000

batch_size : int

Number of tuples taken into account for each iteration of gradient descent. Default : 32

update_rule: str

{sgd,rmsprop}. Default : rmsprop

batch_accumulator : str

{sum,mean}. Default : sum

random_state : numpy random number generator

double_Q : bool, optional

Activate or not the double_Q learning. More informations in : Hado van Hasselt et al. (2015) - Deep Reinforcement Learning with Double Q-learning.

neural_network : object, optional

default is deer.qnetworks.NN_keras

Methods

chooseBestAction
clip_action
discountFactor
getAllParams
gradients
learningRate
qValues
setAllParams
setDiscountFactor
setLearningRate
train

chooseBestAction (*state*)
Get the best action for a belief state

Returns **best_action** : float

estim_value : float

train (*states_val*, *actions_val*, *rewards_val*, *next_states_val*, *terminals_val*)
Train one batch.

1.Set shared variable in states_shared, next_states_shared, actions_shared, rewards_shared, terminals_shared

2.perform batch training

Parameters **states_val** : list of batch_size * [list of max_num_elements* [list of k * [element 2D,1D or scalar]])

actions_val : b x 1 numpy array of objects (lists of floats)

rewards_val : b x 1 numpy array

next_states_val : list of batch_size * [list of max_num_elements* [list of k * [element 2D,1D or scalar]])

terminals_val : b x 1 numpy boolean array (currently ignored)

Returns Average loss of the batch training

Individual losses for each tuple

class deer.q_networks.q_net_keras.**MyQNetwork**(*environment*, *rho=0.9*, *rms_epsilon=0.0001*, *momentum=0*, *clip_delta=0*, *freeze_interval=1000*, *batch_size=32*, *update_rule='rmsprop'*, *random_state=<mtrand.RandomState object>*, *double_Q=False*, *neural_network=<class 'deer.q_networks.NN_keras.NN'>*)

Bases: deer.base_classes.QNetwork.QNetwork

Deep Q-learning network using Keras (with any backend)

Parameters **environment** : object from class Environment

rho : float

Parameter for rmsprop. Default : 0.9

rms_epsilon : float

Parameter for rmsprop. Default : 0.0001

momentum : float

Default : 0

clip_delta : float

Not implemented.

freeze_interval : int

Period during which the target network is freezed and after which the target network is updated. Default : 1000

batch_size : int

Number of tuples taken into account for each iteration of gradient descent. Default : 32

update_rule: str

{sgd,rmsprop}. Default : rmsprop

random_state : numpy random number generator

double_Q : bool, optional

Activate or not the double_Q learning. More informations in : Hado van Hasselt et al. (2015) - Deep Reinforcement Learning with Double Q-learning.

neural_network : object, optional

default is deer.qnetworks.NN_keras

Methods

chooseBestAction
discountFactor
getAllParams
learningRate
qValues
setAllParams
setDiscountFactor

Continued on next page

Table 3.18 – continued from previous page

`setLearningRate`
`train`

chooseBestAction (*state*)
Get the best action for a belief state

Returns **The best action** : int

qValues (*state_val*)
Get the q values for one belief state

Returns The q values for the provided belief state

train (*states_val*, *actions_val*, *rewards_val*, *next_states_val*, *terminals_val*)
Train one batch.

1.Set shared variable in states_shared, next_states_shared, actions_shared, rewards_shared, terminals_shared

2.perform batch training

Parameters **states_val** : list of batch_size * [list of max_num_elements* [list of k * [element 2D,1D or scalar]])

actions_val : b x 1 numpy array of integers

rewards_val : b x 1 numpy array

next_states_val : list of batch_size * [list of max_num_elements* [list of k * [element 2D,1D or scalar]])

terminals_val : b x 1 numpy boolean array

Returns Average loss of the batch training (RMSE)

Individual (square) losses for each tuple

Policies

deer.base_classes.Policy(q_network, ...)	Abstract class for all policies.
deer.policies.EpsilonGreedyPolicy(q_network, ...)	The policy acts greedily with probability $1-\epsilon$ and acts randomly
deer.policies.LongerExplorationPolicy(...[, ...])	Simple alternative to ϵ-greedy that can explore more efficiently fo

Detailed description

class deer.base_classes.**Policy** (*q_network*, *n_actions*, *random_state*)
Abstract class for all policies. A policy takes observations as input, and outputs an action.

Parameters **q_network** : object from class QNetwork

n_actions : int or list

Definition of the action space provided by Environment.nActions()

random_state : numpy random number generator

Methods

action
bestAction
randomAction

action(*state*)
Main method of the Policy class. It can be called by agent.py, given a state, and should return a valid action w.r.t. the environment given to the constructor.

bestAction(*state*)
Returns the best Action for the given state. This is an additional encapsulation for q-network.

randomAction()
Returns a random action

class deer.policies.**EpsilonGreedyPolicy**(*q_network*, *n_actions*, *random_state*, *epsilon*)
Bases: deer.base_classes.Policy.Policy

The policy acts greedily with probability $1 - \epsilon$ and acts randomly otherwise. It is now used as a default policy for the neural agent.

Parameters **epsilon** : float

Proportion of random steps

Methods

action
bestAction
epsilon
randomAction
setEpsilon

epsilon()
Get the epsilon for ϵ-greedy exploration

setEpsilon(*e*)
Set the epsilon used for ϵ-greedy exploration

class deer.policies.**LongerExplorationPolicy**(*q_network*, *n_actions*, *random_state*, *epsilon*, *length=10*)
Bases: deer.base_classes.Policy.Policy

Simple alternative to ϵ-greedy that can explore more efficiently for a broad class of realistic problems.

Parameters **epsilon** : float

Proportion of random steps

length : int

Length of the exploration sequences that will be considered

Methods

action
bestAction
epsilon
randomAction
sampleUniformActionSequence
setEpsilon

epsilon()
Get the epsilon

setEpsilon(*e*)
Set the epsilon

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

d

P

Q

R

S

T

U