Q-learning and Deep Q networks (DQN)

Vincent François-Lavet

Outline

Motivation for value-based reinforcement learning

The Bellman operator
- Dynamic programming
- Q-learning

Q-learning with deep learning as a function approximator

A few variants of DQN

Discussion of a parallel with neurosciences
- How to discount deep RL

Conclusions

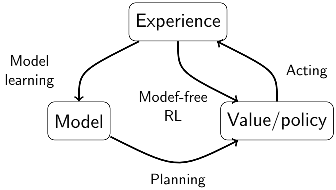
Project

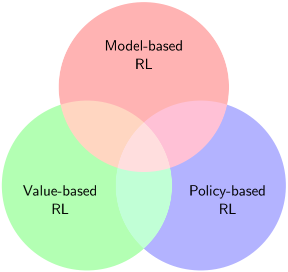
Motivation for value-based reinforcement learning

Overview of deep RL

In general, a reinforcement learning (RL) agent may include one or more of the following components:

- a representation of a value function that provides a prediction of how good is each state or each couple state/action,
- a direct representation of the policy $\pi(x)$ or $\pi(x, a)$, or
- a model of the environment in conjunction with a planning algorithm.





Deep learning has brought its generalization capabilities to RL.

The Bellman operator

Value based methods: recall

In an MDP $(\mathcal{X}, \mathcal{A}, T, R, \gamma)$, the expected return $V^\pi(x) : \mathcal{X} \to \mathbb{R}$ ($\pi \in \Pi$, e.g., $\mathcal{X} \to \mathcal{A}$) is defined such that

$$V^\pi(x) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid x_t = x, \pi\right],$$

with $\gamma \in [0, 1)$.

Value based methods: recall

In addition to the V-value function, the Q-value function $Q^{\pi}(x, a) : \mathcal{X} \times A \to \mathbb{R}$ is defined as follows:

$$Q^{\pi}(x, a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid x_t = x, a_t = a, \pi\right].$$

The particularity of the Q-value function as compared to the V-value function is that the optimal policy can be obtained directly from $Q^*(x, a)$:

$$\pi^*(x) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q^*(x, a).$$

Value based methods

The Bellman equation that is at the core of reinforcement learning makes use of the fact that the Q-function can be written in a recursive form:

$$
\begin{aligned}
Q^{\pi}(x, a) &= \mathbb{E}\left[\sum\nolimits_{k=0}^{\infty} \gamma^k r_{t+k} \mid x_t = x, a_t = a, \pi\right] \\
&= \mathbb{E}\left[r_t + \sum\nolimits_{k=1}^{\infty} \gamma^k r_{t+k} \mid x_t = x, a_t = a, \pi\right] \\
&= \mathbb{E}\left[r_t + \gamma Q^{\pi}(x_{t+1}, a' \sim \pi) \mid x_t = x, a_t = a, \pi\right]
\end{aligned}
$$

In particular:

$$
\begin{aligned}
Q^{*}(x, a) &= \mathbb{E}\left[r_t + \gamma Q^{*}(x_{t+1}, a' \sim \pi^{*}) \mid x_t = x, a_t = a, \pi^{*}\right] \\
&= \mathbb{E}\left[r_t + \gamma \max_{a' \in \mathcal{A}} Q^{*}(x_{t+1}, a') \mid x_t = x, a_t = a, \pi^{*}\right]
\end{aligned}
$$

Value-based method: Q-learning with one entry for every state-action pair

To obtain Q^*, you can:

1. Solve the system of equations (if you know T and R),
2. Initialize the Q-values and repeatedly apply "Bellman iterations" until you find the fixed point (if you know T and R) $\rightarrow$ the dynamic programming case, or
3. Use reinforcement learning to perform the Bellman iterations from data (trials and errors in the environment).

Dynamic programming

Value-based method: Q-learning with one entry for every state-action pair

In order to learn the optimal Q-value function, the Q-learning algorithm makes use of the Bellman equation for the Q-value function whose unique solution is $Q^*(x, a)$:

$$Q^*(x, a) = (\mathcal{B}Q^*)(x, a),$$

where $\mathcal{B}$ is the **Bellman operator** mapping any function $K : \mathcal{X} \times \mathcal{A} \rightarrow \mathbb{R}$ into another function $\mathcal{X} \times \mathcal{A} \rightarrow \mathbb{R}$ and is defined as follows:

$$(\mathcal{B}K)(x, a) = \sum_{x' \in S} T(x, a, x') \left(R(x, a, x') + \gamma \max_{a' \in \mathcal{A}} K(x', a') \right).$$

The chain problem

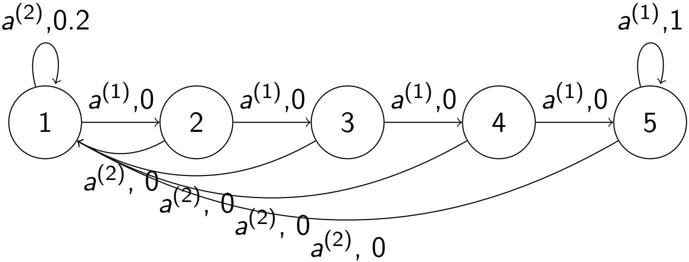


Figure: The chain environment ($\gamma = 0.9$)

The chain problem: tabular Q-values with dynamic programming

state/action	$a^{(1)}$	$a^{(2)}$
1	0	0.2
2	0	0
3	0	0
4	0	0
5	1	0

state/action	$a^{(1)}$	$a^{(2)}$
1	0	0.38
2	0	0.18
3	0	0.18
4	0.9	0.18
5	1.9	0.18

...

state/action	$a^{(1)}$	$a^{(2)}$
1	?	?
2	?	?
3	?	?
4	?	?
5	?	?

Table: Update of the tabular Q-values starting from an initialization to 0.

The resulting policy is to choose action $a^{(1)}$ in all states when the Bellman iterations have converged to its fixed point.

Value-based method: Q-learning (dynamic programming)

Value function $V = \max_a Q(x, a)$ | Resulting policy $\pi = \text{argmax}_a Q(x, a)$

i=0

0	0	1
0	0	0
0	0	0

←↑→↓	←↑→↓	R=1
←↑→↓	←↑→↓	←↑→↓
←↑→↓	←↑→↓	←↑→↓

i=1

0	0.9	1
0	0	0.9
0	0	0

←↑→↓	→	R=1
←↑→↓	←↑→↓	↑
←↑→↓	←↑→↓	←↑→↓

i=2

0.81	0.9	1
0	0.81	0.9
0	0	0.81

→	→	R=1
←↑→↓	↑→	↑
←↑→↓	←↑→↓	↑

Figure: Grid-world MDP with $\gamma = 0.9$, and where we assume that after obtaining R=1, we end up in terminal state (i.e. all following rewards=0)

Q-learning

Value-based method: Q-learning

As opposed to dynamic programming that assumes access to the knowledge a priori of the MDP, RL makes use of learning through trials and errors.

Algorithm 1 Pseudocode for the Q-learning algorithm in the tabular setting

procedure GET_Q_VALUES(node x)
  Initialize Q(x,a) arbitrarily
  for each episode **do**
    Initialize x
    for each step in episode **do**
      Choose action a given x using policy derived from Q (e.g., ϵ greedy)
      Take action a, observe r, x'
      $Q(x, a) \leftarrow Q(x, a) + \alpha[r + \gamma \max_{a'} Q(x', a') - Q(x, a)]$
  return $Q(\cdot, \cdot)$

Convergence Q-learning

Theorem: Given a finite MDP, the Q-learning algorithm given by the update rule

$$Q(x_t, a_t) \leftarrow Q(x_t, a_t) + \alpha_t[r_t + \gamma \max_{a' \in \mathcal{A}} Q_t(x_{t+1}, a') - Q_t(x_t, a_t)],$$

converges w.p.1 to the optimal Q-function as long as
$\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$. and
The exploration policy π is such that
$P_\pi[a_t = a | x_t = x] > 0, \forall(x, a)$.

More details: "Convergence of Q-learning: a simple proof", Francisco S. Melo

Example 1: Mountain car

A car tries to reach the top of the hill but the engine is not strong enough.

- State: position and velocity
- Action: accelerate forward, accelerate backward, coast.
- Goal: get the car to the top of the hill (e.g., reward $= 1$ at the top).

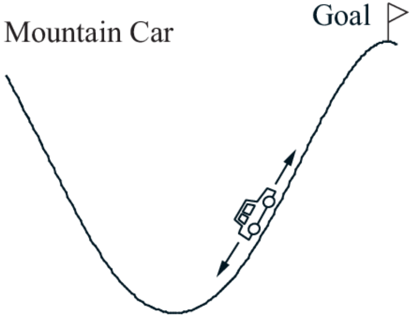


Figure: Mountain car

Example 1: Mountain car

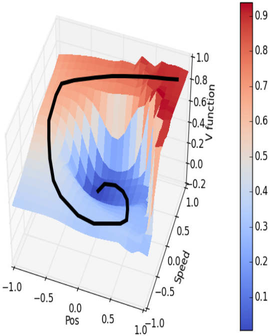


Figure: Application to the mountain car domain: $V = \max_a Q(x, a)$, where the state space has been discretized finely.

Example 1: Mountain car

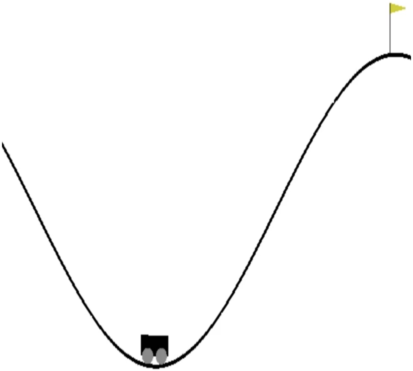


Figure: Mountain car optimal policy

Q-learning with deep learning as a function approximator

Why function approximators?

A tabular approach with discretization fails due to the curse of dimensionality when the number of (initially continuous) dimensions for the state $\gtrapprox 10$ or for a large number of states.

When do we need function approximators?

- large and/or continuous state space $\rightarrow$ DQN
- (large and/or continuous action space) $\rightarrow$ next week we'll see the continuous action space

Q-learning with function approximator

To deal with continuous state and/or action space, we can represent value function with function approximators and parameters θ:

$$Q(x, a; \theta) \approx Q(x, a)$$

The parameters θ are updated such that:

$$\theta := \theta + \alpha \frac{d}{d\theta} \left(Q(x, a; \theta) - Y_k^Q \right)^2$$

with

$$Y_k^Q = r + \gamma \max_{a' \in \mathcal{A}} Q(x', a'; \theta_k).$$

With deep learning, the update usually uses a mini-batch (e.g., 32 elements) of tuples $< x, a, r, x' >$.

DQN algorithm

For Deep Q-Learning, we can represent value function by deep Q-network with weights θ (instabilities !). In the DQN algorithm:

- Replay memory
- Target network

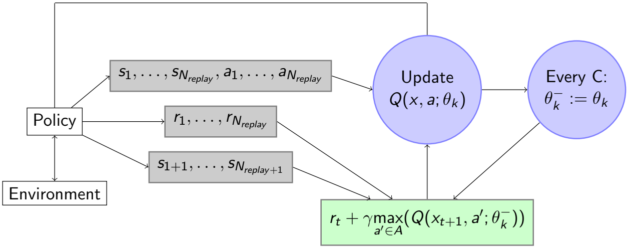


Figure: Sketch of the DQN algorithm. $Q(x, a; \theta_k)$ is initialized to random values (close to 0) everywhere on its domain and the replay memory is initially empty; the target Q-network parameters θ_k^- are only updated every C iterations with the Q-network parameters θ_k and are held fixed between updates; the update uses a mini-batch (e.g., 32 elements) of tuples $< x, a, r, x' >$ taken randomly in the replay memory.

Visualization of Q-values in mountain car

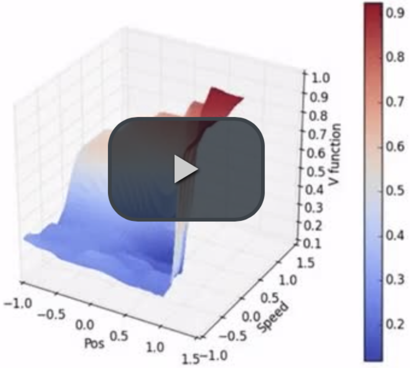


Figure: DQN for mountain car

Example 2: toy example in finance

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:

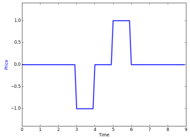


Figure: Price signal

- State: current price and price at the last five time steps+ whether the agent has one item of the good. (This problem becomes very complex without function approximator)
- Action: buy, sell, do nothing.
- Goal: get as much $$$ as possible.

Example using the DeeR library

```
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
```

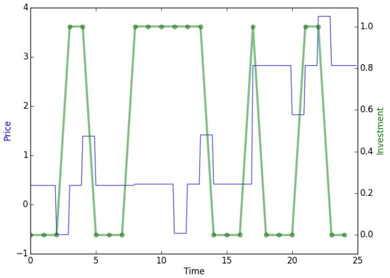
You can then launch "run_toy_env_simple.py" in the folder "examples/toy_env/".

Example: run_toy_env_simple.py

```
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 important elements.
agent.attach(bc.VerboseController())

# During training epochs, we want to train the agent after every action it takes.
agent.attach(bc.TrainerController())

# We also want to interleave a "test epoch" between each training epoch.
agent.attach(bc.InterleavedTestEpochController(epoch_length=500))

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

Example: run_toy_env_simple.py

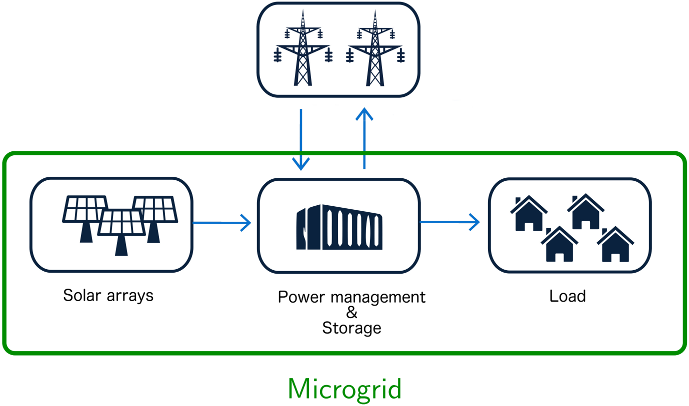
Every 10 epochs, a graph is saved in the "toy_env" folder:



In this graph, you can see that the agent has successfully learned to take advantage of the price pattern. 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.

Real-world application of deep RL: the microgrid benchmark

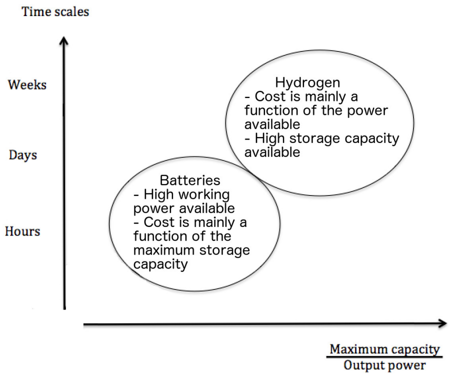
A microgrid is an electrical system that includes multiple loads and distributed energy resources that can be operated in parallel with the broader utility grid or as an electrical island.



Microgrids and storage

There exist opportunities with microgrids featuring:

- A short term storage capacity (typically batteries),
- A long term storage capacity (e.g., hydrogen).



Structure of the Q-network

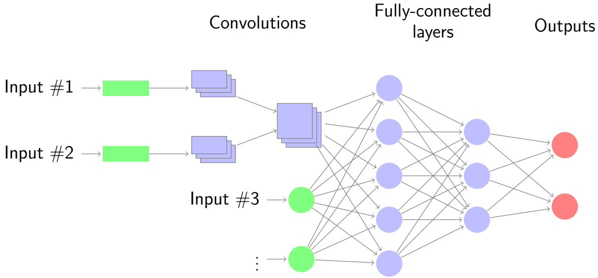


Figure: Sketch of the structure of the neural network architecture. The neural network processes the time series using a set of convolutional layers. The output of the convolutions and the other inputs are followed by fully-connected layers and the ouput layer. Architectures based on LSTMs instead of convolutions obtain similar results.

Results

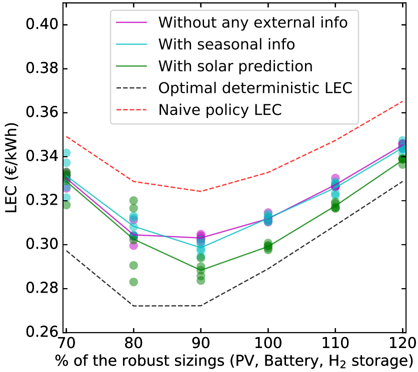


Figure: LEC on the test data function of the sizings of the microgrid.

A few variants of DQN

Distributional DQN

Another approach is to aim for a richer representation through a value distribution, i.e. the distribution of possible cumulative returns.
The value distribution Z^π is a mapping from state-action pairs to distributions of returns when following policy π. It has an expectation equal to Q^π:

$$Q^\pi(x, a) = \mathbb{E}Z^\pi(x, a).$$

This random return is also described by a recursive equation, but one of a distributional nature:

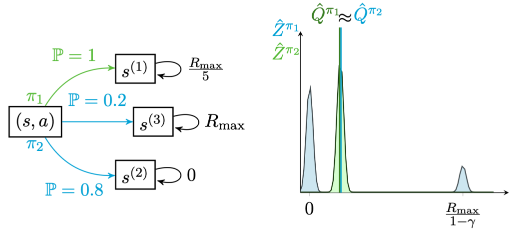
$$Z^\pi(x, a) = R(x, a, X') + \gamma Z^\pi(X', A'),$$

where we use capital letters to emphasize the random nature of the next state-action pair (X', A') and $A' \sim \pi(\cdot|X')$.

Distributional DQN

It has been shown that such a distributional Bellman equation can be used in practice, with deep learning as the function approximator. This approach has the following advantages:

- It is possible to implement risk-aware behavior.
- It leads to more performant learning in practice. One of the main elements is that the distributional perspective naturally provides a richer set of training signals than a scalar value function $Q(x, a)$ (effect of *auxiliary tasks*).



(a) Example MDP.

(b) Sketch (in an idealized version) of the estimate of resulting value distribution $\hat{Z}^{\pi_1}$ and $\hat{Z}^{\pi_2}$ as well as the estimate of the Q-values $\hat{Q}^{\pi_1}, \hat{Q}^{\pi_2}$.

Multi-step learning

In DQN, the target value used is estimated based on its own value estimate at the next time-step. For that reason, the learning algorithm is said to *bootstrap* as it recursively uses its own value estimates.

Such a variant in the case of DQN can be obtained by using the n-step target value given by:

$$Y_k^{Q,n} = \sum_{t=0}^{n-1} \gamma^t r_t + \gamma^n \max_{a' \in A} Q(x_n, a'; \theta_k)$$

where $(x_0, a_0, r_0, \cdots, s_{n-1}, a_{n-1}, r_{n-1}, s_n)$ is any trajectory of $n+1$ time steps with $s = s_0$ and $a = a_0$.
Warning: Online data is required for convergence without bias (or other specific techniques)

Discussion of a parallel with neurosciences

How to discount deep RL

Motivations

Effect of the discount factor in an online setting.

- *Empirical studies of cognitive mechanisms in delay of gratification*: The capacity to wait longer for the preferred rewards seems to develop markedly only at about ages 3-4 ("marshmallow experiment").

Increasing discount factor (using the DQN aglorithm)

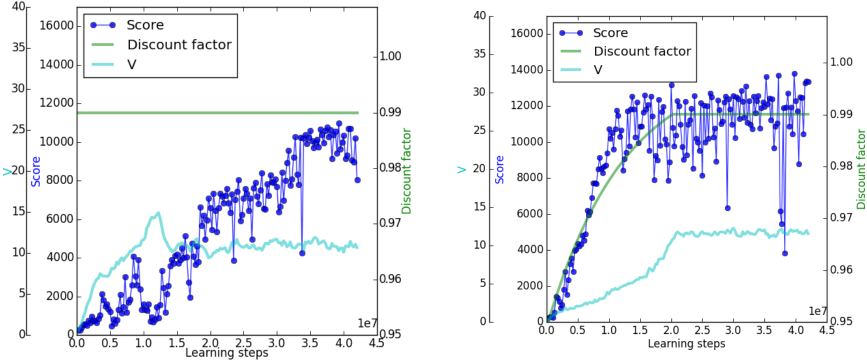


Figure: Illustration for the game q-bert of a discount factor γ held fixed on the right and an adaptive discount factor on the right.

Conclusions

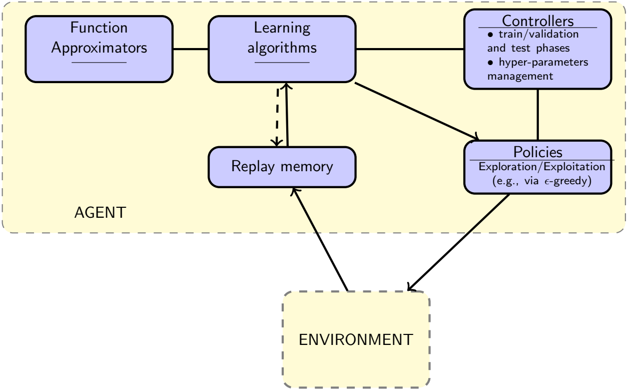
Summary of the lecture

- Introduction to Q-learning in the tabular case and with deep learning (DQN).
- Toy examples and real-world examples
- Brief discussion on the role of the discount factor and some relations to neuroscience

Further ressources (optional)

- Watkins, Christopher JCH, and Peter Dayan. "Q-learning." Machine learning 8, no. 3-4 (1992): 279-292.
- Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves et al. "Human-level control through deep reinforcement learning." nature 518, no. 7540 (2015): 529-533.

Further ressources



Implementation : https://github.com/VinF/deer

Questions?

Project

Project

You consider the chain environment made up of 5 discrete states and 2 discrete actions, where you get a reward of 0.2 on one end of the chain and 1 at the other end (see illustration below).

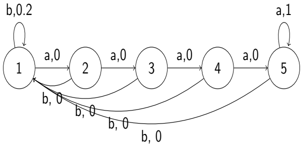


Figure: The chain environment ($\gamma = 0.9$). Initial state is state 1.

In part 1, you work in the tabular context:

- Solve using tabular Q-learning and ϵ-greedy. Provide the optimal Q-values, discuss the learning rate α and ϵ (4 points)
- Increase the size of the chain to 10 states while keeping the rewards at both end of the chain. Discuss the new results, in particular ϵ (2 points).

Project

In part 2 (4 points), you can either solve the chain problem using function approximators for $\gamma = 0.9$ and 10 states.

- Provide illustrations of the solutions of your optimal Q-values (2 points)
- Discuss the hyper-parameters and the convergence (2 points)

If you don't have much previous experience with function approximators such as deep learning, you can go for the following part 2 (4points):

- Study the effect of having the discount factor close to 0 or close to 1 (what happens to the optimal solution? What happens with the convergence?)

Deadline : 24th of December (try to aim for one week earlier!)

Example: run_toy_env_simple.py

If you start from
https://github.com/VinF/deer/blob/master/examples/toy_env,
You must modify Toy_env.py and run_toy_env_simple.py.

- You must code the MDP transition (and the reward) in the method act (you don't need to use *rng*)

```
def act(self, action):
    ...
```

- Your state is simply defined as one scalar (without history).

```
def inputDimensions(self):
        return [(1,)]
```

- Your have two actions

```
def nActions(self):
    return 2
```

Example: run_toy_env_simple.py

- You never have terminal states:

```
def inTerminalState(self):
    return False
```

- The function "observe" provides the encoded representation of the state

```
def observe(self):
    return np.array(self._last_ponctual_observation)
```

Questions?