IN104
Solving labyrinth with Reinforcement learning

Florence Carton

April 10, 2018

Contents

1 Introduction

The aim of this tutorial is to give basis of Reinforcement Learning to achieve the project IN104 at ENSTA ParisTech.
The goal of this project is to perform one or several algorithms that are able to find a path through a labyrinth.

Reinforcement learning is used when we have a clear idea of what we want, but not exactly how we can achieve it. For example if I play chess, I know I want to checkmate my opponent, but I don't know exactly which action to perform every time it is my turn.

2 RL : Definitions and Notions

2.1 Definitions

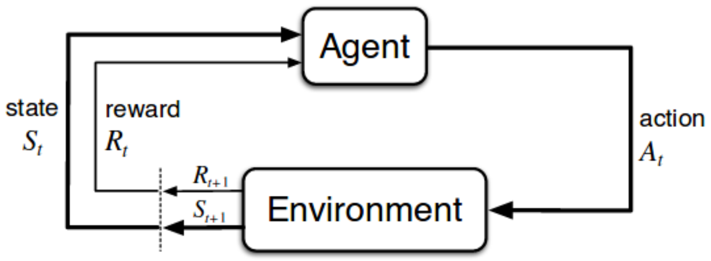

Figure 1: RL diagram from [1]

In Reinforcement Learning approach, an agent interacts with an environment. The environment produces a state s_t at each timestep t, and when receiving the current state s_t, the agent reacts with an action a_t. The agent acts according to a policy $\pi(a_t|s_t)$, which represents the probability to take an action a when being in state s (in a deterministic environment, $\pi(s) = a$). After the agent has taken the action a_t, the environment provides a new state s_{t+1} alongside a reward r_t, which indicates how good the new state is.
The goal of the agent is to maximize the cumulative rewards : $max \sum r_t$ (the rewards are often discounted to avoid exploding sums). The sum of cumulative (often discounted) reward is called the return R_t :

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{1}$$

We also define V-function and Q-function as follow :

$$V_\pi(s) = \mathbb{E}_\pi[R_t|S_t = s] \tag{2}$$

$$Q_\pi(s,a) = \mathbb{E}_\pi[R_t|S_t = s, A_t = a] \tag{3}$$

The V-function is the expected return from a given state, and the Q-function is the expected return from a given state-action pair. These functions evaluate then how a good a state (respectively a state-action pair) is when following a given policy π.

2.1.1 Policy

Policy π = how the agent behaves.
$\pi(a|s) = Prob(action = a|state = s)$ (in deterministic case $\pi(s) = a$)
$\boxed{\text{What we want is the optimal policy}}$

2.1.2 State-action values (= Q values)

As explained earlier, Q-values defines 'how good' is a pair state-action. For instance, in a grildworld, if the agent is in a given square, it needs to know if it must go up, down, left or right.

state\actions	up	down	right	left
state s_1	10	12	0	3
state s_2	3	20	4	1
...	...	...	...	...
state s_n	...	...	...	...

Figure 2: Q table

$\boxed{\text{Optimal policy } \pi^*(s) = argmax_a Q(a, s)}$

Objective of the project : Fill this Q table for different environments - with different algorithms (as once we have the Q table, we know how the agent must behave to get the maximum return.)

2.2 MDP

In Reinforcement Learning, the agent makes a decision according to a state it receives. The state signal is assumed to have the Markov property, i.e. to be only dependent on the previous state. Therefore the transition probability is a function only of the previous state and action : $p(s_{t+1}|s_t, a_t)$. The Reinforcement Learning problem is then a Markov Decision Process (MDP) defined by a 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$:

- $\mathcal{S}$: set of states
- $\mathcal{A}$: set of actions
- $\mathcal{P}$: transition function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$
- $\mathcal{R}$: reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathbb{R} \to [0, 1]$ or in deterministic cases $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$
- γ is the discount factor (trade-off between present and future reward that indicates that rewards matter more)

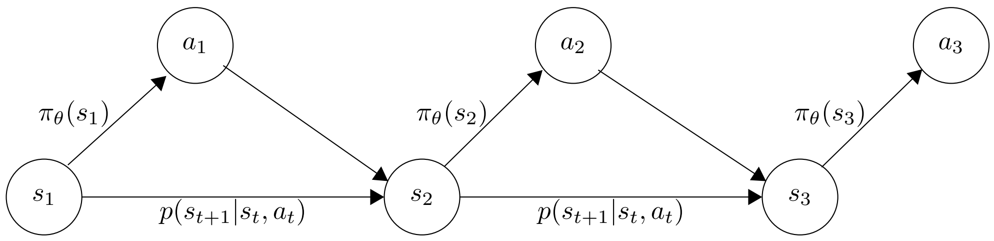

Figure 3: MDP in RL

3 Algorithms

We present here 3 algorithms that belongs to Q-learning category. The goal of Q-learning is to fill a Q-table.

We only care here about Model-free algorithms, i.e. the transition function is not known.

3.1 Exploration vs Exploitation

Exploration versus Exploitation is a very famous dilemma in Reinforcement Learning. Exploitation on one side, aims at making the best so far, i.e. with current knowledge, and exploration consists in testing other solutions and then gather information, as an unexplored decision could be better that the other already tested. Both exploration and exploitation are crucial in RL, and the challenge lies in the trade-off between these two notions. It seems reasonable to explore a lot at the beginning of the algorithms, when all paths have not been explored yet, and explore less and less when the algorithms improves. A common approaches to face the E-E dilemma is ϵ-greedy, which consists in picking the best action (= greedy action) with probability $1 - \epsilon$ and a random action with probability ϵ (ϵ from 0.01 to 0.1 are common choices).

3.2 Monte-Carlo - Q learning

Monte-Carlo Methods are model-free algorithms (= that don't know the transition function), that learn from experience. Monte Carlo methods also learn from complete episodes. The basis idea of these algorithms is to use the mean values, i.e. Q(s,a) = mean return starting from state s and performing action a.

Algorithm 1 First-visit Monte-Carlo policy evaluation from [1]

```
 1: Initialize, for all $s \in \mathcal{S}, a \in \mathcal{A}$ :
        $Q(s,a) \leftarrow$ arbitrary
        $Returns(s,a) \leftarrow$ empty list for all $s \in \mathcal{S}$
        $\pi(a|s) \leftarrow \epsilon$-greedy policy
 2: repeatforever
 3:     Generate an episode using $\pi$
 4:
 5:     for each state $(s,a)$ appearing in the episode : do
 6:         $R \leftarrow$ return following the first occurrence of $s$
 7:         Append $R$ to $Returns(s,a)$
 8:         $Q(s,a) \leftarrow average(Returns(s,a))$
 9:     end for
10:
```

3.3 TD learning

Contrary to Monte-Carlo methods, TD-learning algorithms don't wait until the end of the episode to estimate the value, they can update values during the episode, which presents the advantage of being faster. Here are shown 2 algorithms of TD-learning : Q-learning (which is off-policy) and SARSA (which is on-policy), both for q-value computing.

3.3.1 Q -learning

Algorithm 2 Q-learning from [1]

```
1: Initialize Q(s,a) arbitrarily, $\forall s \in \mathcal{S}, a \in \mathcal{A}$, and $Q(terminal\_state, \cdot)$
2: repeat(for each episode)
3:     Initialize s
4:     repeat(for each step of the episode)
5:         choose action a from s using policy derived from Q (e.g. $\epsilon$-greedy)
6:         take action a, observe reward r and next state s'
7:         $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_a Q(s',a) - Q(s,a)]$
8:         $s \leftarrow s'$
9:     until s is terminal
```

Q-learning is off-policy as Q-values are updates using a greedy policy (the max in the update line), whereas they are updated using the current policy on SARSA.

3.3.2 SARSA

Sarsa is another algorithm to learn the Q-values. The name Sarsa comes from the quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ needed for the algorithm.

Algorithm 3 SARSA from [1]

```
1: Initialize Q(s,a) arbitrarily, $\forall s \in \mathcal{S}, a \in \mathcal{A}$, and $Q(terminal\_state, \cdot)$
2: repeat(for each episode)
3:     Initialize s
4:     choose action a from s using policy derived from Q (e.g. $\epsilon$-greedy)
5:     repeat(for each step of the episode)
6:         take action a, observe reward r and next state s'
7:         choose action a' from s' using policy derived from Q (e.g. $\epsilon$-greedy)
8:         $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
9:         $s \leftarrow s'$ ; $a \leftarrow a'$
10:    until s is terminal
```

As SARSA is on-policy, its convergence properties depend on the policy used, whereas it in not the case for Q-learning, which is independent of the policy being followed.

4 Work to do

4.1 Structure du projet

main.py : fonction principale : pour lancer les codes
params.py : fichier contenant les paramètres par défaut
environements
|– Environment.py : contient la classe de base Environment
|– EnvironmentGrid1D : environement simple 1D
|– load_env.py : contient une fonction pour charger l'environement demandé
agents
|– Agent.py : contient la classe de base Agent
|– AgentRandom.py : contient un agent aléatoire
|– load_agent.py : fonction pour charger un agent

4.2 Moyen d'évaluation

4.2.1 Objectifs du projet

Implémenter un ou plusieurs algorithmes d'apprentissage par renforcement pour résoudre le déplacement dans un labyrinthe.
Les environements doivent être écrit en respectant le formalisme d'Openai Gym[1]
Pour aller plus loin :

- Rendu graphique
- Faire décroitre le taux d'exploration ϵ
- résoudre l'environement Cartpole d'Openai Gym

4.2.2 Rapport

- Description de l'algo utilisé
- Problèmes rencontrés
- Résultats
- ...

4.2.3 Clarté du code

- Code (très) bien commenté
- Noms de variables et de fonctions cohérents
- Readme : description du projet + comment lancer le projet
- Git : commits réguliers et bien commentés

4.2.4 Présentation

10 minutes présentation, 10 min questions

References

[1] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* Vol. 1. 1. MIT press Cambridge, 1998, p. 445. arXiv: `1603.02199`. URL: `http://www.incompleteideas.net/book/bookdraft2017nov5.pdf`.

[1] `https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py`