Reinforcement Learning

Guillaume Charpiat

Course at CentraleSupelec 2016-2018

Contents

Chapter 1

Introduction

1.1 Overview of the contents of this course

This course comprises lessons as well as exercises. All information, content and references are available online at `https://www.lri.fr/~gcharpia/machinelearningcourse/`. This document is the summary of (a part of) the lessons only.

1.2 Overview of this document

We first present, in chapters 3-4, classical reinforcement learning in the discrete setting, to introduce basic concepts. We extend them to the continuous domain in chapter 5, and have a quick word about the particular case of robotics in chapter 6. Reinforcement problems in robotics are indeed particulary complex, due to the continuous nature of most state variables, to the large number of possible actions, and, consequently, to the size of the domain of possible stategies to explore. Chapter 7 is about 2-player adversarial games, such as chess or go.

Before that, let us study the simplistic yet insightful case of *bandits*, where actions do not modify the environment, in order to explain the dilemma between exploration and exploitation.

Chapter 2

Bandits

2.1 Reinforcement learning framework

2.1.1 General setting

The *Agent* performs an *action* on the *Environment* and gets a *reward* that depends on the Environment. This action leads to a new *state*. The set of possible actions may depend on the state, and their consequences as well. Rewards and new states may be stochastic.

2.1.2 The particular case of bandits

In the restricted case of *bandits*, the environment is static (no environment basically), *e.g.* there is only one state.

2.1.3 Example

Ex: Medicine trial. One would like to make statistics on medicines A, B, C to know which one is the best. You also want to minimize the number of people dying in the process.

Possible actions: choose A, B or C
Rewards: -1 if people died, +0.5 if they survived with after-effects, +1 if they healed
Goal: maximize sum of rewards
Issue: you do not know *yet* which medecine is the best, and have to make trials for this

2.2 Strategies

2.2.1 Exploration vs. exploitation

There is a dilemna between *exploration* and *exploitation*. *Exploring* means testing each possibility (*bandit arm*) many times, to make sure that our estimation of the average reward for that arm is precise enough. On the opposite, *exploiting* means picking always the arm that seems to be the best. One cannot fully explore and fully exploit at the same time.

2.2.2 Examples of strategies

- Greedy: you would try a few times each of them (same number each) and then you say: "I am done exploring" then you exploit (you take always the best arm in the previous step). Issues: How many times is "a few times" ?

- To be better you need to keep exploring all the time: ε-greedy: At each time-step you flip a coin. With probability ε you pick randomly uniformly any arm, otherwise (probability $1-\varepsilon$) you take the best arm so far. At initialization you either chose a high or low estimate for each arm (high if you want to force exploration at the beginning). Issue: this never stops exploring even when estimates have converged.

- Softmax: Denote by $Q(a)$ the average reward for the action a. Consider probabilities of taking action a:

$$p(a) = \frac{\exp(\frac{Q(a)}{\tau})}{\sum_{a'} \exp(\frac{Q(a')}{\tau})}$$

  If τ is close to 0, it tends to be close to the greedy strategy.
  If τ tends to infinity: you pick every arm with equi-probability.
  Issue: for a given reasonable fixed τ: same issue as ε-greedy.

2.3 Modeling and optimization criterion

Denote by K the number of arms to pull (number of possible actions). Each time you pull one arm k, there is a distribution of possible rewards D_k with its value in $[0,1]$ (or any other bounded space). At time t we chose an arm $I_t \in \{1,..,K\}$ and get the reward x_t which follows the distribution D_{I_t} (x_t are iid [independent, identically distributed]).

We define μ_k, the average reward for arm k, and μ_*, the average reward of the best arm:

$$\mu_k = \mathbb{E}[x_k] \quad \text{and} \quad \mu_* = \max_k(\mu_k)$$

The expectation $\mathbb{E}$ is here taken over the stochasticity of the environment. The *regret* for T trials is defined as:

$$\begin{aligned} R_T &= T\mu_* - \sum_{t=1}^{T} x_t \\ &= \sum_t (\text{best-reward-on-average} - \text{current-reward}) \end{aligned}$$

It expresses how far a strategy is from the optimal choice (consisting in picking always the best arm, from the beginning).

We have, for a given strategy, on average over the stochasticity of the environment and of the strategy if it is stochastic as well:

$$\begin{aligned} \mathbb{E}[R_T] &= T\mu_* - \sum_t \mathop{\mathbb{E}}_{I_t}[\mu_{I_t}] \\ &= \sum_k \mathbb{E}\left[T_k\right](\mu_* - \mu_k) \\ &= \sum_k \mathbb{E}\left[T_k\right]\delta_k \end{aligned}$$

where T_k is the number of times you have selected arm k, and $\delta_k = \mu_* - \mu_k \geqslant 0$.

2.4 Example of an optimal strategy: UCB

Multi-armed bandits are one of the only setups in reinforcement learning where one can obtain mathematical guarantees about the training results for some algorithms, such as UCB.

2.4.1 UCB algorithm

UCB (Upper Confidence Bound) consists in the following stragegy: at each time step t, for each arm k, compute:

$$B(k) = \widehat{\mu_k} + \sqrt{\frac{2\log t}{t_k}}$$

with t_k the number of times you have picked arm k until now, and $\widehat{\mu_k}$ the average reward so far for arm k. Now, select:

$$I_t = \text{argmax}_k B_{(t)}(k)$$

This selects the arm with the highest optimistic average (estimation of the average reward + possible error margin due to the fact that empirical averages can differ from real expectations because of sampling effects).

2.4.2 UCB Bound

You can skip this section if you're not fond of maths.

The strange-looking error bound $\sqrt{\frac{2\log t}{t_k}}$ comes from the Chernoff-Hoeffding inequality, which reads, for any $\varepsilon > 0$:

$$p\left(\widehat{\mu_k} - \mu_k \;\geq\; \varepsilon\right) \;=\; p\left(\frac{1}{t_k}\sum_{\substack{t \text{ for which}\\ \text{arm } k\\ \text{was chosen}} } x_t - \mu_k \;\geq\; \varepsilon\right) \;\leqslant\; \exp(-2t_k\varepsilon^2)$$

For the suitable choice of $\varepsilon = \sqrt{\frac{2\log t}{t_k}}$, we get

$$p\left(\widehat{\mu_k} + \sqrt{\frac{2\log t}{t_k}} \leqslant \mu_k\right) \;\leqslant\; \exp(-4\log(t)) \;=\; \frac{1}{t^4}$$

which states that the probability of mistaking tends to 0 very fast with the total number t of trials.

Lemma For a arm k which is not optimal:

$$\mathbb{E}[T_k] \leqslant 8\frac{\log T}{\delta_k^2} + \frac{\pi^2}{3}$$

[proof next page] This means that the number of times one mistakenly chooses arm k instead of the best arm is bounded linearly by $\log T$, which is a pretty low amount of errors. In particular, this means that the error rate, $\frac{\mathbb{E}[T_k]}{T}$, tends to 0 with increasing T.

Concerning the regret, we thus obtain:

$$\begin{aligned}\mathbb{E}[R_T] &= \sum_k \delta_k\,\mathbb{E}[T_k]\\ &\leqslant \sum_{k\neq k^*} 8\frac{\log T}{\delta_k} + K\frac{\pi^2}{3}\end{aligned}$$

So, the regret is also similarly bounded. One could argue that δ_k could be very small and make this quantity high. This is actually not a problem because Cauchy-Schwartz

yields:

$$\begin{aligned}\mathbb{E}[R_T] &= \sum_k \delta_k \, \mathbb{E}[T_k] \\ &\leqslant \sum_k \sqrt{\delta_k^2 \, \mathbb{E}[T_k]} \sqrt{\mathbb{E}[T_k]} \\ &\leqslant \sqrt{\sum_k \delta_k^2 \, \mathbb{E}[T_k]} \sqrt{\sum_k \mathbb{E}[T_k]} \\ &\leqslant \sqrt{8KT(\log T + \frac{\pi^2}{3})}\end{aligned}$$

Proof of the lemma Suppose at time t, for all arms k:

$$\mu_k - \sqrt{\frac{2\log t}{t_k}} \leqslant \widehat{\mu_k} \leqslant \mu_k + \sqrt{\frac{2\log t}{t_k}}$$

Consider an arm k not optimal, while k_* would be the best one. If, at time t, arm k is picked, then :

$$\begin{gathered} B_t(k) \geqslant B_t(k_*) \\ \widehat{\mu_k} + \sqrt{\frac{2\log t}{t_k}} \geqslant \widehat{\mu_{k_*}} + \sqrt{\frac{2\log t}{t_*}} \\ \mu_k + 2\sqrt{\frac{2\log t}{t_k}} \geqslant \widehat{\mu_k} + \sqrt{\frac{2\log t}{t_k}} \geqslant \widehat{\mu_{k_*}} + \sqrt{\frac{2\log t}{t_*}} \geqslant \mu_* \\ 2\sqrt{\frac{2\log t}{t_k}} \geqslant \mu_* - \mu_k \\ t_k \leqslant 8\frac{\log t}{\delta_k^2}\end{gathered}$$

Now we need to sum over times t from 1 to T. $\forall u \in \mathbb{N}$,

$$\begin{aligned}T_k &\leqslant u + \sum_{t=u+1}^{T} \mathbb{1}_{I_t=k \ \wedge \ T_k(t)\geqslant u} \\ &\leqslant u + \sum_{t=u+1}^{T} \mathbb{1}_{\exists s,s_*, \text{ s.t. } u<s\leqslant t \ \wedge \ 1\leqslant s_*\leqslant t \ \wedge \ B_{t,s}(k)\geqslant B_{t,s_*}(k_*)}\end{aligned}$$

Finally, consider $u = \frac{8\log T}{\delta_k^2}$:

$$\begin{aligned}\mathbb{E}[T_k] &\leqslant \frac{8\log T}{\delta_k^2} + \sum_{t=u+1}^{T} \Big(\sum_{s=u+1}^{t} t^{-4} + \sum_{s=1}^{t} t^{-4} \Big) \\ &\leqslant \frac{8\log T}{\delta_k^2} + \frac{\pi^2}{3}\end{aligned}$$

This last summation of $\frac{1}{t^4}$ is the reason why we needed an error probability decreasing to 0 as fast as $\frac{1}{t^4}$. A slower asymptotic rate would prevent convergence of the sum. Hence the choice of $\varepsilon = \sqrt{\frac{2\log t}{t_k}}$ for Chernoff-Hoeffding inequality.

2.5 Performance Bounds

2.5.1 How good can a strategy be?

One can prove infimum bounds on optimal strategies, that is, there exists no strategy that on average can perform better than such bound. Thanks to such studies [Lai et Robbins 1985], one can know whether a given strategy is *optimal*, *i.e.* one cannot expect better regret. The bound is based on the difference between the distribution D_k of rewards for arm k and the one D_* for the best arm:

$$\begin{aligned} \limsup_T \frac{\mathbb{E}[T_k]}{\log T} &\geqslant \frac{1}{KL(D_k||D*)} \\ \mathbb{E}[T_k] &= \Omega(\log T) \\ \mathbb{E}[R_T] &= \Omega(\log T) \end{aligned}$$

which means that one can expect neither to do fewer than $\log T$ mistakes, nor to obtain a regret less than $\log T$ (up to constant factors).

At the higher level of considering not just one particular bandit task but all possible bandit tasks: for any algorithm, there exists a bandit task on which the regret will be at least $\sqrt{KT}$ [Cesa-Bianchi and Lugos 2006]:

$$\inf_{\text{algo}} \sup_{\text{task}} \mathbb{E}[R_T] = \Omega(\sqrt{KT})$$

2.5.2 KL-UCB

This is a variant of UCB exploiting the information theory bounds above. Consider:

$$B_t(k) = \sup_D \left\{ \mathbb{E}[D],\ (\widehat{\mu_k} - \mathbb{E}[D])^2 \leqslant \frac{f(t)}{T_k(t)} \right\}$$

for some appropriate function $f(t)$ such as $2\log t$, and where D is any distribution modeling the true reward distribution of arm k, consequently supposed to be close to the distribution D_k of observed rewards. The term $(\widehat{\mu_k} - \mathbb{E}[D])^2$ is better replaced with $KL(D_k||D)$.

2.6 Softmax strategy and Time

2.6.1 Variation on the Softmax strategy

Let $Q_t(k)$ be the total reward of arm k until now (not the average, but the sum!). Set:

$$p_t(k) \propto \exp\left(Q_t(k)\right) = \frac{\exp\left(Q_t(k)\right)}{\sum_{k'} \exp\left(Q_t(k')\right)}$$

This can be seen as considering a variable rate $\tau = \frac{1}{t}$. It progressively moves from exploration to exploitation. It also has a Bayesian interpretation as $p(k|\text{all past rewards})$.

Consider also the ε-soft variant, with a small additive uniform noise:

$$p(k) = (1-\gamma)p_t(k) + \gamma\frac{1}{K}$$

If choosing $\gamma = \frac{K \log K}{(e-1)T}$, then we have the following guarantee:

$$\mathbb{E}[R_T] \leqslant O\left(\sqrt{KT \log K}\right)$$

2.6.2 Combining experts' advice for time series prediction

A bit farther from reinforcement learning, let's consider the task of time series prediction. We suppose we are given a panel of "experts" (*i.e.*, algorithms), each of which emits predictions at each time step, under the form of a probability distribution over possibilities.

A first goal would be to find the best expert (based on experience, which one yields the most accurate predictions?).

However a linear combination of all experts' predicted distributions can do better than listening to the best single expert only. Regret is then redefined with respect to the new maximum expected gain.

Among classical solutions lies an adaptation of the softmax strategy above (based on total reward).

Chapter 3

Learning dynamics

3.1 Definitions

An **Agent** interacts with an **Environment**. The **State** is given both by the internal configuration of the agent and the environment. Generally speaking a state is a compact description of the world. You can say it is an a-priori where you put all information available for the decision to be taken. You can also put history in that. The difference between the agent and the environment is usually defined by what you can control and what you cannot.

At each step, the agent takes an **action**, go to a new **state** and gain a **reward**.

When you are in a particular state you choose an action based on a **policy**. We try to find the best policy to get maximum reward.

Example 1 Let us consider a Robot with an arm. The position of the arm represents the state of the robot.

An agent may not know the full state of the world. It may have only a partial view.

Example 2 Let us consider a Robot moving. The agent is the controller, the part making decision. It is not the hardware.

Example 3 Let us consider a chess game. If you specify sub-goals, for example gaining points when taking adverse pieces, you might end up taking the queen and losing the game: subgoals are dangerous.

The goal is to maximize rewards on the long run.

Let us consider a finite horizon T (finite number of turns in a game). At time t you want to maximize:

$$\mathop{\mathbb{E}}_{e,\pi}\left[\sum_{i=t+1}^{T} r_i\right]$$

where π is the policy, a_t, s_t, r_t are respectively action, state and reward at time t and e is the environment.

In practice there might not be a specified finite number of turns but there usually exist terminal states (states that end the game).

You might want to put less weight on rewards you will gain in a very far future:

$$\mathop{\mathbb{E}}_{e,\pi}\left[\sum_{k=0}^{\infty}\gamma^k r_{t+1+k}\right]$$

where γ, the discount rate, is such that $0 \leqslant \gamma \leqslant 1$.

We define the return R by:

$$R = \sum_{k=0}^{\infty}\gamma^k r_{t+1+k}$$

3.1.1 Markov Decision Process (MDP)

To chose an action you might want to evaluate the quantity:

$$p = p(s_{t+1}, r_{t+1}|s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, ...)$$

We make the hypothesis of a Markovian environment (a way to do that is to include all history in the state $s'_t = (s_t, s_{t-1}, r_{t-1}, s_{t-2}, \dots)$) so that we have $p = p(s_{t+1}, r_{t+1}|s_t, a_t)$.

In a Markov Decision Process, everything is described by only two quantities:

- $p(s_{t+1}|s_t, a_t)$: this describes the dynamics (how we are moving in the space of states)
- $\mathbb{E}_e[r_{t+1}|s_t, a_t, s_{t+1}]$: the average reward (full distribution not required)

This can be described as a graph (like a Markov chain).

3.1.2 Value functions: V and Q

Let us suppose that an environment follows a Markov Decision Process and that we know all the state and reward graph.

Policy π is a function such that $\pi(state, action)$ is the probability of taking that action when in that state. Of course $\sum_{\substack{\text{possible}\\ \text{actions } a}} \pi(state, a) = 1$.

Let us assume you are in the state s_t, you chose action a_t according to the policy distribution, and a new state and reward is sampled from the environment.

A *state value function* V is a function such that $V(state)$ is the expected return when you start from that state, and follow a fixed policy π:

$$V^\pi(s) = \mathop{\mathbb{E}}_{\pi,e}\left[R_t|s_t = s\right] = \mathop{\mathbb{E}}_{e,\pi}\left[\sum_{k=0}^{\infty}\gamma^k r_{t+1+k}|s_t = s\right]$$

with s_T the terminal state. We have $V^\pi(s_T) = 0$.

A *state-action value function* Q is a function such that $Q(state, action)$ is the expected return when you are in that state and you take that action:

$$Q^\pi(s,a) = \mathop{\mathbb{E}}_{\pi,e}[R_t|s_t, a_t]$$

Consider any policy π and any state s. At each time step we choose an action a_t which depends on the policy π and the current state s_t. Depending on the action a_t the current state s_t and the environment e, we get a new state s_{t+1} and a reward r_{t+1}. We denote $\mathbb{E}_{e,\pi}$ the expectation taken on all possible actions, states and rewards.

$$\begin{aligned} V^\pi(s) &= \mathop{\mathbb{E}}_{\pi,e}[R_t|s_t = s] \\ &= \mathop{\mathbb{E}}_{e,\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+1+k}|s_t = s\right] \\ &= \mathop{\mathbb{E}}_{e,\pi}\left[r_{t+1} + \sum_{k=0}^{\infty} \gamma^k r_{t+2+k}|s_t = s\right] \\ &= \mathop{\mathbb{E}}_{a_t\sim\pi, s_{t+1}\sim e}[r_{t+1} + \gamma V^\pi(s_{t+1})] \end{aligned}$$

3.1.3 Optimal Policies

Let us suppose we have a Markov decision process. There is a partial order over policies: we say $\pi \geqslant \pi' \iff \forall s, V^\pi(s) \geqslant V^{\pi'}(s)$.

Properties

- Existence of Optimal policy:

$$\exists \pi^*, \forall \pi, \pi^* \geqslant \pi$$

- All possible π^* have the same value functions:

$$\forall s, Q^*(s,a) = \max_\pi Q^\pi(s,a)$$

Bellman optimality equation

$$V^*(s) = \max_a \sum_{s'} p(s'|s,a)(R^a_{s,s'} + \gamma V^*(s'))$$

$R^a_{s,s'}$ is the average reward when taking action a in state s and arriving in s' and $p(s'|s,a)$ depends only on the environment.

Similarly:

$$Q^*(s,a) = \sum_{s'} p(s'|s,a)(R^a_{s,s'} + \gamma \max_{a'} Q^*(s',a'))$$

Optimizing the policy When you have a policy π you want to find the value function V^π (this is called policy evaluation)

First step: Estimating the value function An iterative approach works as follow. Let us start with any value function V_k.

$$\begin{aligned} V_{k+1}(s) &= \mathop{\mathbb{E}}_{\pi,e}\left[r_{t+1} + \gamma V_k(s_{t+1})|s_t = s\right] \\ &= \sum_a \pi(s,a) \sum_{s'} p(s'|s,a)(R^a_{s,s'} + \gamma V_k(s')) \end{aligned}$$

I am in state s_t. I need to take an action a_t. I arrive in a new state (s_{t+1}, r_{t+1}) with some probabilities. My estimation of r_{t+1} is exact but the estimation of $V_k(s_{t+1})$ is the one I assumed before (at iteration k). This converges to the real value function since $\gamma < 1$. Making use of $V_k(s_{t+1})$ to re-estimate $V_{k+1}(s_t)$ is named bootstrapping.

The proof of convergence is given using

$$\begin{aligned} |(V_{k+1} - V^\pi)(s)| &= |\sum_a \pi(s,a) \sum_{s'} p(s'|s,a)\left[R^a_{s,s'} + \gamma V_k(s')) - R^a_{s,s'} - \gamma V^\pi(s')\right]| \\ &\leqslant \sum_a \pi(s,a) \sum_{s'} p(s'|s,a)\gamma |V_k(s') - V^\pi(s')| \\ &\leqslant \sum_a \pi(s,a) \sum_{s'} p(s'|s,a)\gamma \|V_k - V^\pi\|_\infty \\ &\leqslant \gamma \|V_k - V^\pi\|_\infty \end{aligned}$$

We define the infinite norm by $\|V_{k+1} - V^\pi\|_\infty = \max_s |(V_{k+1} - V^\pi)(s)|$ and we find

$$||V_{k+1} - V^\pi||_\infty \leqslant \gamma ||V_k - V^\pi||_\infty$$

which gives convergence at an exponential rate:

$$||V_k - V^\pi||_\infty \leqslant \gamma^k ||V_0 - V^\pi||_\infty.$$

There exists an in-place version where updates for s are done in series rather than in parallel.

Policy improvement We have some policy π, we describe how good it is with V^π and Q^π. Define greedy policy π' which consists in taking the best action according to the previous value function:

$$\pi'(s) = \text{argmax}_a Q^\pi(s,a)$$

Then $\pi'(s) \geqslant \pi(s)$.

If no improvement the policy is optimal already. Doing **policy evaluation** and **policy improvement** repetitively is called **policy iteration**.

Let us introduce T the Bellman operator such that $T(\pi) = \pi'$ is the result of the policy improvement algorithm on π.

The optimal policy $\pi*'$ satisfies $T\pi* = \pi*$ which we can write $B\pi' = 0$ with $B = T - Id$.

The optimization we do is equivalent to a Newton Method-like step to find a 0 of B:

$$\pi_{k+1} = \pi_k - \left(\frac{dB}{d\pi}\right)^{-1} B\left(\pi_k\right)$$

A method called **truncated policy evaluation** works as follow. You don't wait for convergence, you just apply one step of policy evaluation before doing policy improvement. Still same guaranties of convergence. The proof is the same.

This can be generalized to any number of steps, the same guaranties always apply. With this dynamic programming algorithm, finding optimal policy is polynomial in the number of actions m and states s. Without this trick (naive evaluation of all policies) we would get exponential complexity $O(m^n)$.

3.2 Monte Carlo (MC) methods

The value function $V^\pi(s)$ is the average return in state s.

When running simulations, we are playing episodes = runs of the game. Averaging the empirical value function for a particular state seen during many episodes (one episode = one sample) might be a good evaluation of the value function.

Whether you consider only the first visit or every visit to the state, your estimate is always going to converge to $V^\pi(s)$ with precision $\frac{1}{\sqrt{n_{\text{samples}}}}$.

We have a fixed policy π you sample using Monte Carlo Q and you update your policy using the greedy policy improvement presented in last section.

3.2.1 On-Policy MC-control

We sample according to the policy we want to use. An additional hypothesis is that you have ε-soft policy with $\varepsilon > 0$, to be sure exploration is done. This means $\forall s, a\ \pi(s,a) > \varepsilon$ (ε-greedy is one example of such policy).

3.2.2 Off-Policy evaluation

We consider 2 policies simultaneously, for different purposes: one for exploration and one for exploitation. We can evaluate a policy π based on experiments with another one π'. It requires that $\pi(s,a) > 0 \implies \pi'(s,a) > 0$. Consider the first visit to a state

s. Let us denote p_i the likelihood of future state, reward and action for policy π and let us denote p'_i the likelihood of future state, reward and action for policy π'.

$$p_i(s) = \prod_{k=t}^{T-1} \pi(s_k, a_k) p(s_{t+1}|a_k, s_k)$$

$$p'_i(s) = \prod_{k=t}^{T-1} \pi'(s_k, a_k) p(s_{t+1}|a_k, s_k)$$

Instead of averaging like $V(s) = 1/n \sum_i R_i(s)$, we consider

$$\begin{aligned} V(s) &= \frac{1}{Z} \sum_i \frac{p_i(s)}{p'_i(s)} R_i(s) \\ &= \frac{1}{Z} \sum_i \frac{\pi(s_k, a_k)}{\pi'(s_k, a_k)} R_i(s) \end{aligned}$$

The behaviour policy is different than the estimation policy. The behaviour policy is used for sampling (running experiments). The estimation policy is the policy you really want to use in the end (the one you want to improve).

$\forall s, a$
$Q(s, a) \leftarrow random()$
$N(s, a) \leftarrow 0$
$D(s, a) \leftarrow 0$
$\pi \leftarrow random()$
repeat
generate an episode $s_0, a_0, r_0, s_1, a_1, r_1, ...$ with any soft policy π'
take the tail ($t > \tau$) so that all $a_{t>\tau}$ follows $\pi(s_t)$
$\forall (s, a)$ in that tail
$w \leftarrow \prod_{k=1}^{T-1} \frac{1}{\pi'(s_k, a_k)}$
$N(s, a) += wR_t$
$D(s, a) += w$
$Q(s, a) = \frac{N}{D}(s, a)$
$\forall s, \pi(s) = \text{argmax}_a Q(s, a)$
until enough iteration are made

On the good size of MC methods, there is no need to estimate environment bootstrap or probabilities. However, many explorations might be needed.

3.2.3 Temporal difference

Immediate MC on reward (not returns).

$$V(s_t) += \alpha \left([r_{t+1} + \gamma V(s_{t+1})] - V(s_t) \right)$$

with $\alpha \in [0, 1]$

This is bootstraping (using $V(s_{t+1})$).

Proved to converge to V^π if α is small enough or $\alpha_t \to 0$ and $\sum \alpha_t \to \infty$

You can also do this in batch mode: You run a number of episodes, average, and then update (faster convergence).

3.2.4 On-policy TD control: SARSA

We have a policy π we want to estimate the value $Q(s, a)$ (same as before replacing V with Q)

$$Q(s_t, a_t) += \alpha \left([r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})] - Q(s_t, a_t) \right)$$

Update for each 5-uple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ hence the name SARSA.

π derived from Q needs to be soft e. g. ε-greedy.

3.2.5 Off-policy TD control: Q-learning

[Watkins, 1989] Breakthrough at that time because it is working better than other approaches.

- Exploitation

$$Q_t(s_t, a_t) += \alpha \left(\left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) \right] - Q(s_t, a_t) \right)$$

- Exploration: Choose an action a using a soft policy derived from Q e.g ε-greedy (for softness).

Distinguishing the exploitation policy from the exploration one solves the *cliff example* issue that on-policy methods (such as SARSA) suffer from.

Chapter 4

Eligibility traces

4.1 n-step prediction

We have different ways of estimating the return:

- Monte Carlo (run a full episode and get the sum of all rewards):

$$R_t = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{T-t-1} r_T$$

- Temporal difference TD(0) (bootstrapping at t+1):

$$R_t^{(1)} = r_{t+1} + \gamma V_t(s_{t+1})$$

We can interpolate between these two methods (start running an episode and bootstrap at time t+n):

- n-step return

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n})$$

How to choose the right value of n? Don't chose, mix all possibilities:

- Temporal difference TD(λ) (λ with $\lambda \in [0, 1]$)

$$R_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)}$$

- Note, if terminal state reached in finite time: Temporal difference (λ with $\lambda \in [0, 1]$ and $T \leqslant \infty$)

$$R_t^\lambda = (1-\lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} R_t^{(n)} + \lambda^{T-t-1} R_t$$

  with R_t the complete return. When λ goes to 1, R_t^λ goes to R_t. When λ goes to 0, R_t^λ goes to $R_t^{(0)}$ which corresponds to Temporal difference (0)

4.2 n-step backup

Update of the value function for state s_t:

$$\delta(V_t(s_t)) = \alpha[R_t^{(n)} - V_t(s_t)]$$

$$V_{t+1}(s_t) = V_s(s_t) + \alpha[R_t^{(n)} - V_t(s_t)]$$

4.3 λ-return algorithm

$$\delta(V_t(s_t)) = \alpha[R_t^{\lambda} - V_t(s_t)]$$

only for one s_t.
R_t^{λ} is the target.

4.4 TD(λ) Backward view

Eligibility trace $e_t(s) \geqslant 0$ depends on the state and the time defined by:

$$\begin{aligned} e_t(s) &:= \gamma\lambda e_{t-1}(s) && \text{if } s \neq s_t \\ &:= \gamma\lambda e_{t-1}(s) + 1 && \text{otherwise} \end{aligned}$$

$\gamma\lambda < 1$ so it goes to 0 if you stay in the same state $s_t \neq s$.
For each particular state: it "jumps" by +1 if the state comes, and otherwise decreases exponentially. The eligibility $e_t(s)$ is the accumulation of fading traces. λ is the decreasing speed of the trace (i.e. of the influence from the past).

TD (temporal difference) TD for prediction given the policy π:

$$\delta_t := [r_{t+1} + \gamma V_t(s_{t+1})] - V_t(s_t)$$

Defined only for the current state s_t.

For any state s,

$$\Delta V_t(s) := \alpha\delta_t e_t(s)$$

α is the update speed,
δ_t is the quantity update I want to add,
e_t is the trace. The trace is the credit to state s for what happens now at s_t.

One can show that this algorithm computes the right total update:

$$\sum_t \Delta V_t(s) = \alpha(R_t^{alpha}(s) - V_t(s))$$

Proof Very easy (TODO).

Particular case $TD(\lambda = 1)$ is the same as MonteCarlo method but with updates at each time for all states instead of waiting for the end of the game.

4.5 SARSA-λ

On-policy TD-control using Q

For all states s and action a:

- Define $e_t(s, a)$
- update of $Q_{t+1}(s, a) = Q_t(s, a) + \alpha\delta_t e_t(s, a)$

$$\delta_t = [r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1})] - Q_t(s_t, a_t)$$

$$e_t(s, a) = \begin{cases} \gamma\lambda e_{t-1}(s, a) & \text{if } (s, a) \neq (s_t, a_t) \\ \gamma\lambda e_{t-1}(s, a) + 1 & \text{otherwise} \end{cases}$$

On-policy: you update the policy according to an ε-soft policy based on Q (i.e ε-greedy).

4.6 $Q(\lambda)$

Update is the following:

$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha\delta_t e_t(s, a)$$

with $\delta_t = [r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a')] - Q_t(s_t, a_t)$
and $e_t(s, a) = \mathbb{1}_{s=s_t \wedge a=a_t} + \mathbb{1}_{Q_{t-1}(s_t,a_t)=\max_a Q_{t-1}(s_t,a)}\gamma\lambda e_{t-1}(s, a)$.
Transmission only if optimal action is chosen.

Until now we have estimated only a value for a state $V(s)$ one value for each state s. It is a huge vector in $\mathbb{R}^{|S|}$ with $|S|$ the number of states. This can lead to huge dimensions problem. You might want to consider a much smaller set of states using a parameterization of the problem.

4.7 Parametrized familly $V_\theta(s)$

You compute $f_\theta(s)$ with f_θ a function depending on parameters θ (like a neural network) and you do the learning on $f_\theta(s)$, i.e. by learning the right value of θ.

An update $\delta\theta$ of the vector of parameters θ leads to an update of the value function:

$$V_{\theta+\delta\theta}(s) = V_\theta(s) + \frac{dV(s)}{d\theta}\delta\theta.$$

Reciprocally, given a desired target value δV, if $\frac{dV(s)}{d\theta}$ is invertible, the parameters θ should be updated as: $\delta\theta = (\frac{dV(s)}{d\theta})^{-1}(\delta V(s))$.

The target values δV can be:

- $DP : \mathbb{E}_{\pi,e}\left[r_{t+1} + \gamma V_t(s_{t+1})|s_t = s\right]$
- $MC : R_t$
- $TD(0) : r_{t+1} + \gamma V_t(s_{t+1})$
- $TD(\lambda)$: see above

Policy evaluation $\pi \mapsto V^\pi$

$V_{\theta_t}(s)$ does not cover the full space of states so we cannot always find a θ such that V_θ would be exactly V^π for all s.

How can we say that a θ is better than another one ? You can define a criterion (i.e Mean Square Error):

$$\begin{aligned} MSE(\theta_t) &= \sum_s p(s)[V^\pi(s) - V_{\theta_t}(s)]^2 \\ &= \mathop{\mathbb{E}}_{\pi,e}\left[(V^\pi - V_{\theta_t})^2\right] \end{aligned}$$

We can estimate θ using gradient descent:

$$\theta \ -= \ \tau \nabla_\theta MSE(V_\theta)$$

which yields:

$$\theta_{t+1} = \theta_t \ + \ 2\tau \left(V^\pi(s_t) - V_{\theta_t}(s_t)\right) \nabla_\theta V_{\theta_t}(s_t)$$

θ_{t+1} is a vector, the new values of the parameters.
$\nabla_\theta V_{\theta_t}(s_t)$ is a vector of same dimension as θ_t.
$V^\pi(s_t)$ is the target (a real number) (see possible target values above), and $V_{\theta_t}(s_t)$ is the current value. The value of the target $V^\pi(s_t)$ might be unknown, in which case one can replace it with an approximation v_t of it.

If the approximation v_t of the target $V^\pi(s_t)$ is an unbiased estimator, i.e. if $\mathbb{E}_{\pi,e}[v_t] = V^\pi(s_t)$, then θ_t converges to a local optimum provided α_t goes to 0 and $\sum_t \alpha_t$ goes to infinity when t goes to infinity.

If $v_t = R_t^\lambda$ it is bootstraping (this is biased).

Approximation $v_t = R_t^\lambda$:

$$\theta_{t+1} = \theta_t + \alpha \left[(r_{t+1} + \gamma V_{\theta_t}(s_{t+1})) - V_{\theta_t}(s_{t+1})\right] e_t$$

where
$e_t = \gamma\lambda e_t + \nabla_\theta V_{\theta_t}(s_t)$
$e_0 = 0$

Example of a parameterized family: the linear case

$$V_{\theta_t}(s) \;=\; <\theta_t|\phi(s)> \;=\; \sum_i \theta_t^i \phi_i(s)$$

where $\phi(s)$ is a representation of state s (features). Note that $\phi(s)$ and θ are vectors (of same dimension). $V_{\theta_t}(s)$ is a linear combination of the features $\phi_i(s)$ with weights θ^i.

Gradient descent on θ: $\nabla_\theta V_{\theta_t} = \phi(s)$

In the case of a convex problem (such as MSE) with an unbiased estimate of $V^\pi(s)$: we will reach the globally optimum parameters.

In the $TD(\lambda)$ case, the estimate is biased, and we get only the following convergence guarantee:

$$MSE(\theta_\infty) \leqslant \frac{1-\gamma\lambda}{1-\gamma} MSE(\theta^*).$$

Which features $\phi(s)$?

- Best case: Modeling.
- Rich enough features (kernels, neural networks)

Control with function approximation:

Gradient descent on parameters:

$$\theta_{t+1} = \theta_t + \alpha(v_t - Q_{\theta_t}(s_t, a_t))\nabla_{\theta_t} Q_{\theta_t}(s_t, a_t),$$

where v_t is a real number, it is the target. Many choices are possible, for example $r_{t+1} + \gamma Q_{\theta_t}(s_{t+1}, a_{t+1})$ (bootstrap).
$\nabla_\theta Q_{\theta_t}(s_t, a_t)$ is the gradient vector.
We can also use traces: $e_t = \gamma\lambda e_{t-1} + \nabla_\theta Q_{\theta_t}(s_t, a_t)$.
With these particular settings and with an ε-soft policy derived from Q_{θ_t}, one gets the "function approximation" version of SARSA(λ).
Without bootstrap: one obtains a real gradient descent on an energy (so you have good properties).

With bootstrap: one gets only near-optimal guaranties. If on-policy: not an issue, if off-policy it can even diverge.

The Mean Square Error as a criterion to optimize is also a matter of choice. Example of alternative to MSE:

$$\sum_s p(s) \mathop{\mathbb{E}}_{\pi,e} \left[\left((r_{t+1} + \gamma V_{\theta_t}(s_{t+1})) - V_{\theta_t}(s) \right)^2 \right].$$

The difference is that θ_t appears twice (and both occurrences will appear in the computation of the gradient).

Chapter 5

Policy Gradient

5.1 New framework: Optimize policy directly

We estimate directly the policy $\pi(a|s,\theta)$ itself. θ is our parametrization of the problem.

Example: Softmax:

$$\pi(a|s,\theta) = \frac{\exp(h(s,a,\theta))}{\sum_b \exp(h(s,b,\theta))}$$

with h any pre-defined reasonable function expressing how good it is to take action a when in state s, depending on parameters θ. This softmax trick is a generic way of turning any function into a probability distribution.

5.2 Policy Gradient Theorem

The goal is to maximize some criterion E (for Energy):

$$E(\theta) = v_{\pi_\theta}(s_0)$$

For example: $v_{\pi_\theta}(s_0)$ can be the expected return when starting in state s_0 under policy π_θ, or the average reward over time (if infinite duration).

Let us state the policy gradient theorem:

$$\nabla_\theta E = \sum_s d_\pi(s) \left(\sum_a Q_{\pi_\theta}(s,a) \nabla_\theta \pi(a|s,\theta) \right)$$

where $d_\pi(s)$ is the probability to be in state s when following π_θ for a long while (stationary distribution hypothesis).

If an action-independent quantity $q(s)$ is added to $Q_{\pi_\theta}(s,a)$, it does not change $\nabla_\theta E$.

Proof: it would add, for each s, a term $q(s) \sum_a \nabla_\theta \pi(a|s,\theta)$, with $\sum_a \nabla_\theta \pi(a|s,\theta) = \nabla_\theta \sum_a \pi(a|s,\theta) = \nabla_\theta 1 = 0$.

5.3 REINFORCE algorithm

REINFORCE is a Monte Carlo policy gradient.

We have

$$\begin{aligned}\nabla_\theta E &= \mathop{\mathbb{E}}_{s_t\sim\pi_\theta,e}\left[\gamma^t \sum_{a_t} Q_{\pi_\theta}(s_t,a_t)\nabla_\theta \pi(a_t|s_t,\theta)\right] \\ &= \mathop{\mathbb{E}}_{a_t,s_t\sim\pi_\theta,e}\left[\gamma^t Q_{\pi_\theta}(s_t,a_t)\frac{\nabla_\theta \pi(a_t|\theta,s_t)}{\pi(a_t|s_t,\theta)}\right] \\ &= \mathop{\mathbb{E}}_{s,a,R\sim\pi_\theta,e}\left[\gamma^t R_t \nabla_\theta \log \pi(a_t|s_t,\theta)\right]\end{aligned}$$

R_t is the return from time t. The algorithm is simply the following:

$$\theta_{t+1} = \theta_t + \alpha\gamma^t R_t \nabla_\theta \log \pi_\theta(a_t|s_t,\theta).$$

Note that it does not require the estimation of a value function V or Q, but only directly the parameters of the policy π_θ.

5.4 Actor Critic

As said before we can any quantity to R as long as it does not depend on the action. So we can replace R_t by $R_t - V(s_t)$ which balances the amplitudes and reduces the variance. In this case we need to estimate V, and for this we can parameterize $V = V_{\theta_2}$ with an other parameter θ_2 and perform this optimization with respect to θ_2 as in the previous courses.

Actor-critic is this process of comparing the return (or expected return, or reward depending on the settings) to the ones you would get performing another action in the same state (subtracting the average is actually such a comparison).

If you use Actor-critic and bootstrap you get:

$$\theta_{t+1} \;=\; \theta_t \;+\; \alpha\left(\left[r_{t+1} + \gamma V(s_{t+1,\theta_2})\right] - V(s_t,\theta_2)\right)\; \nabla_\theta \log \pi(a_t|s_t,\theta).$$

$\left[r_{t+1} + \gamma V(s_{t+1,\theta_2})\right]$ is the (Monte Carlo) estimate of the return R_t and $V(s_t,\theta_2)$ is the actor-critic comparison.

The parameters θ_2 of the value function V also need to be estimated, as follows (as in the previous lesson):

$$\theta_2 := \theta_2 + \alpha_2\big(r_{t+1} + \gamma V(s_{t+1,\theta_2}) - V(s_t,\theta_2)\big)\nabla_{\theta_2} V.$$

Note the term between parentheses is the same. Let's name it δ_t.

5.5 Version with eligibility traces

Set

$$e_{t+1} = \lambda e_t + I \nabla_\theta \log \pi_\theta(a|s)$$

with I defined such as

$$\begin{cases} I = 1 & \text{if } t = 0 \\ I_{t+1} = \gamma I_t & \text{otherwise.} \end{cases}$$

Then, implement the parameter evolution:

$$\theta_{t+1} = \theta_t + \alpha \delta_t e_t$$

with δ_t defined as

$$\delta_t = \big(r_{t+1} + \gamma V(s_{t+1,\theta_2}) - V(s_t, \theta_2)\big).$$

For the value function, updates are defined similarily as :

$$e_2 = \lambda e_2 + I \nabla_{\theta_2} V_{\theta_2}(s)$$

and

$$\theta_2 = \theta_2 + \alpha_2 \delta_t e_2.$$

5.5.1 Different Goal

In case of a very long game you might prefer to estimate the average reward $E(\theta)$ is the average reward.

It is the same reasoning but subtracting at each time step the average reward $\overline{r}$:

$$\delta = [(r_{t+1} - \overline{r}) + V_{\theta_2}(s_{t+1})] - V_{\theta_2}(s_t)$$

which corresponds to considering the return $R - T\overline{r}$ in order to avoid divergence issues.

5.6 An example from the literature: DeepMind's DQN (Deep Q-Network): Atari Games

This is an example for the previous lesson, not of policy gradient.

Q_{θ_t} is a neural network with parameters θ_t. The target value for Q_{θ_t} at time $t+1$ is $\mathbb{E}[r + \gamma \max_a Q_{\theta_t}]$. The loss to minimize is:

$$L(\theta_{t+1}) = \mathbb{E}\left[\big(\text{target} - Q_{\theta_{t+1}}\big)^2\right]$$

using gradient descent.

Q_{θ_t}: Neural network with 3 layers:

- conv
- conv
- fully connected

Input: 4 frames of size 84×84 pixels
Output: 18 actions.

$$\theta_{t+1} = \theta_t + \alpha[r_{t+1} + \gamma \max_a Q_\theta(s_{t+1}, a) - Q_\theta(s_t, a_t)] \nabla_\theta Q_\theta(s_t, a_t)$$

Chapter 6

Robotics

6.1 Continuity

In robotics, states and actions are usually continuous.
For instance for a robotic arm, a state is typically of the form $s = (\theta, \dot{\theta})$, where θ is a list of angles, and $\dot{\theta} = \frac{d\theta}{dt}$ the associated angular velocities.
Actions are then typically forces that can be applied, *i.e.* $\ddot{\theta}$.

High dimensional space $\implies$ too huge space to explore (curse of dimensionality). Function approximation is required, in order not to learn independently for every possible state, but transfer knowledge between similar states. Yet it is generally not sufficient alone to tackle high dimensional spaces.

Good representation can help to decrease the dimension of the space ; a relevant representation makes the problem easier.

Hierarchical representation: of trajectories, e.g., or of the task, decomposed in sub-tasks (each of which is easy to solve).

Examples:

- hand-designed discretization of the state space in regions of interest
- or learned (regions)
- meta-actions: one meta-action = a given sequence of movements (ex: move 5m ahead)
- combining action primitives
- considering clusters sharing properties
- local models (different models for different state space regions)

Get help from:

- imitation learning: show a few examples of correct trajectories to the robot, for a few cases. Use them as an exploration basis, *e.g.* linear combinations of primitives + noise
- good policy initialization $\implies$ faster to learn
- or make use of pre-programmed policies
- incorporate physical constraints in the policy $\implies$ smaller space to explore
- hierachical RL: decompose in sub-tasks
- or increase task difficulty progressively
- encourage to explore new promising tracks: reward for novelty, curiosity, etc. (but useful ones only, i.e. that bring the most information)

6.2 Time

Algorithm should run real-time, and time needs to be discretized.

Sensing and motor delays $\implies$ better be taken into account (by learning, e.g.)

Examples:

- pre-structured policy, with motor primitives using physics equations

6.3 Virtual world

Learning in a virtual world : much faster than real robot movements (+ all hardware issues, experiment setting...)

$\implies$ reality gap : needs to adjust the virtual world model to the real one

Examples: temperature impacts robot dynamics; light conditions; sensing uncertainty...; small i.i.d. errors explode when taking derivatives...

$\implies$ loop : planning in the virtual world / testing it the real world
+ need to adapt / be robust (e.g., notice temperature is wrong, and adapt)

To help to be robust:

- model the distribution of possible worlds, or consider stochastic virtual worlds

- model the uncertainty (about the space / action / dynamics) and consider it when updating policies

To better compare policies:

- run them on the exact same example (with same random seed)

6.4 Goal specification

Sparse reward $\implies$ difficult optimization (especially in a such huge space)

$\implies$ more informative rewards (and more frequent), while making sure they will not lead to undesirable behaviors (as in the boat race game with too high side bonuses).

Example: reward the proximity to the solution. For instance, for a bilboquet, distance to the target ; or $\ddot{\theta}^2 + \|\text{desired location} - \text{actual location}\|^2$.

6.5 Examples of approaches

Continuous $\implies$ function approximation $\implies$ policy gradient (with actor-critic), or deep Q-learning

In combination with a learned model of the world,
and with examples of solutions in a few cases.

6.6 Links

Reinforcement Learning in Robotics: A Survey, by Kober, Bagnell & Peters,
`http://www.ias.tu-darmstadt.de/uploads/Publications/Kober_IJRR_2013.pdf`

Playground (virtual environments for RL training): OpenAI Gym
`https://gym.openai.com/`

To go further: state of the art: workshop at NeurIPS 2019
`http://www.robot-learning.ml/2019/`

Chapter 7

Monte Carlo Tree Search (MCTS)

7.1 Adversarial games and classical approaches

Setting: adversarial game where each one plays in turn.
Question: best strategy, to maximize chances of winning (or score)?

Minimax approach You alternate the best action that you can take and the best action the adversary can take (which is the worst for you).

In practice you cannot go very far in the tree (of possible future actions) with that approach. When you have to stop (descending the tree, i.e. at leaves of the tree), you have to rely on a hand-made estimate of your probability to win.

$\alpha - \beta$ **pruning** Branch and bound to discard branches that cannot lead to a better score.

7.2 Upper confidence Tree

Use bandits, one bandit per action. This was done in CrazyStone (a go player). We take the arm i that maximizes the following quantity:

$$\overline{r_i} + \sqrt{\frac{\log n}{T_i(n)} F_i}$$

where $F_i = \min(1/4, \sigma_i^2 + \sqrt{\frac{2 \log n}{T_i(n)}})$ is an optimistic estimation of the variance of the rewards for arm i, after n time steps including $T_i(n)$ times pulling that arm.

At some point you reach the end of the game. The good thing is that bandits give you a policy without having to estimate the probability to win.

7.3 MOGO: Monte Carlo Tree Search (MCTS)

Bandits each time yo play. But instead of developing the full tree (of future actions), when I am in a new state (never visited before, i.e. without a bandit already there to tell which action to take), I play many random games and take the average score. Thus you get an estimate of the state, action value and create a new bandit node for that action. You remember only the nodes seen more than twice.

7.4 Alpha-GO

See the Alpha-Go paper in Nature: `https://storage.googleapis.com/deepmind-media/alphago/AlphaGoNaturePaper.pdf` mixing MCTS with neural netwoks, and see Alpha-Zero, a different approach learning everything from scratch (and not pre-trained to mimick experts).

Appendix

Acknowledgments

This LaTeX document is initially based on Hugo Richard's notes of my course at CentraleSupelec in 2018.

Links and references

The web version of this course (including videos and references) is available at:

- `https://www.lri.fr/~gcharpia/machinelearningcourse/`.

The main reference of the field is the book:

- *Reinforcement Learning: An Introduction* by Richard S. Sutton & Andrew G. Barto.

More references, in particular for more advanced mathematics on the topic:

- the book *Statistical Learning and Sequential Prediction* by Alexander Rakhlin & Karthik Sridharan,
- the course *Reinforcement Learning* by Rémi Munos.

Research articles and review papers for specific topics are indicated for each chapter on the web page.