Reinforcement learning

Master 2

Université Paris Dauphine - PSL

2021-2025

(c) Gabriel Turinici DO NOT DISTRIBUTE

DISCLAIMER

Warning

What follows is a "cultural" presentation of the reinforcement learning theory. We present the theory as found in the literature but in particular there is no guarantee on the performance of these procedures in the REAL WORLD applications, be it in medicine, finance, marketing, etc.
Nothing of what follows is an invitation to use one approach or another in a professional or personal framework, the reader is encouraged to use her/his common sense and critical views.
In particular, the treatment of heavy tailed distributions can and do play an important role that cannot be neglected in practice.
If in need of a particular advice on a specific application contact me directly.

Outline

1. AI and deep learning : introduction
2. Building blocks of strategies : the Multi-armed bandit
   - Presentation
   - Basic strategy: ϵ-greedy
   - MAB: optimism
   - MAB: Confidence intervals
   - MAB: policy gradient algorithms
3. Formalization: (Finite) Markov decision processes (MDP), value function and Bellman equation
4. Bellman : policy evaluation, improvement, value iteration
5. Monte Carlo approaches
6. Reinforcement learning frameworks : Open AI Gym
7. Temporal difference learning
8. Learning strategies : Q-learning, double Q-learning, SARSA
   - Q-Learning
   - Double / Dual Q-Learning
   - SARSA and variants

Reminders : types of "learning"

Left : supervised learning e.g. classification on CIFAR10 labels. (source: Tensorflow Sept 23); **Middle** : generative learning (source Wikipedia Sept 2023) ; **Right** : reinforcement learning, credits : https://www.youtube.com/watch?v=Qi1HGSYbjDQ and https://www.youtube.com/watch?v=VMp6pq6_QjI.

- We will focus on reinforcement learning and often on "deep reinforcement learning".

Reminders : types of "learning"

- Supervised learning : e.g. classification: the labels are given i.e. we know the value function;
- Unsupervised learning : e.g. generative : no labels, only an objective e.g. clustering or generate objects similar to a given set
- reinforcement learning : e.g. game play : based on the interaction with the environment; any action executed within an environment; a signal is received that indicates whether the action has been positive or negative. The good actions are **reinforced** encouraged and bad actions are "punished"; note that in the beginning good/bad is not always defined (e.g. 0.5 is good ?)

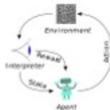


Figure: The typical framing of a Reinforcement Learning (RL) scenario: an agent takes actions in an environment, which is interpreted into a reward and a representation of the state, which are fed back into the agent. Source: Wikipedia

Goal : obtain high scores by accumulating rewards

Question : how to learn i.e. how to choose the next action to take ?

Reinforcement learning : general characteristics

- this course proposes a computational approach to learning through the interaction with the environment
  The goal of reinforcement learning is to know what to do in a given context to maximize the reward signal.
- often the actions influence future environments
- the learner needs to DISCOVER, by trial and error, the best actions that give best rewards
- actions influence the reward, the next situation, environment and can have LONG TERM CONSEQUENCES
- future reward may come with huge delays (e.g. chess, go ...) so it can be interesting to play in the long term and exchange a small immediate reward for a larger one in the future

Reinforcement learning : general characteristics

- sometimes it is DIFFICULT TO KNOW whether an action is good or bad (long term consequences)
- time is important (not i.i.d !)
- the environment can be UNCERTAIN and next state not deterministic function of the action and present state.
- often the agent needs to adapt the strategy to current environment, not play some optimal strategy given in advance.

Outline

1. AI and deep learning : introduction
2. Building blocks of strategies : the Multi-armed bandit
   - Presentation
   - Basic strategy: ϵ-greedy
   - MAB: optimism
   - MAB: Confidence intervals
   - MAB: policy gradient algorithms
3. Formalization: (Finite) Markov decision processes (MDP), value function and Bellman equation
4. Bellman : policy evaluation, improvement, value iteration
5. Monte Carlo approaches
6. Reinforcement learning frameworks : Open AI Gym
7. Temporal difference learning
8. Learning strategies : Q-learning, double Q-learning, SARSA
   - Q-Learning
   - Double / Dual Q-Learning
   - SARSA and variants

Multi-armed bandit

- the problem is to allocate limited resources (time, money, turns etc.) among terms of a given list. Goal is to maximize expected rewards.
- Name: from slot-machines (one-armed bandit); example of goal maximize return over $n = 1000$ steps.

References : [1, 2] etc.

Multi-armed bandit

- k-armed bandit : has k options to choose from
- Other situations: choice among medical treatments, for a series of patients
- rewards information: each action has a random reward with a given but unknown mean;
- the means will be called "values" of the arms.
- Notations t : turn or time; R_t : reward at step t, A_t : action at step t, $\mathcal{A}$: set of possible actions
- value function $q_* : \mathcal{A} \to \mathbb{R}$ is unknown; in particular $q_*(a) := \mathbb{E}[R_t | A_t = a]$. (note : here "*" stands for the "true" or "optimal" or "most precise")

Multi-armed bandit

- Main question : HOW TO CHOOSE among the k options ???
- Idea: use some estimation Q for the value function q_*.
- the estimation will evolve in time $Q = Q_t$, we hope $Q_t \simeq q_*$
- TWO MAIN CHOICES : exploitation or exploration
- exploitation example: use action a which maximizes $Q_t(a)$: NAME= "greedy" . Exploitation is related to short term reward maximization.
- exploration example : choose a random choice. Exploration can improve the long term reward but is "costly" in the short term.

Multi-armed bandit

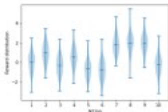
How to construct the estimate Q_t for q_* ?
First proposal : action-value methods: use actions to estimate the values

$$Q_t(a) = \frac{\text{sum of rewards when action a was taken before t}}{\text{number of times a was taken before t}} = \frac{\sum_{\ell=1}^{t} R_\ell \mathbb{1}_{A_\ell = a}}{\sum_{\ell=1}^{t} \mathbb{1}_{A_\ell = a}}$$

By the law of large numbers we expect $Q_t \to q_*$ when t is large.
Greedy action selection : $A_t = \arg\max_a Q_t(a)$
Variant : mostly greedy BUT with ϵ probability select action at random (to explore !!)
Such a combined strategy is called ϵ-greedy.

Multi-armed bandit : empirical evaluation

Take $k = 10$ with q_* chosen from a standard normal (once for all, at the beginning).
The behavior of the reward is also a normal centered in the (unknown to the player) value $q_*(a)$: see figure below.



Total steps $T = 1000$; total runs $= M = 2000$.

Multi-armed bandit : empirical results for the ϵ-greedy strategy

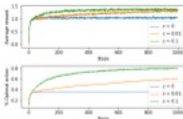


Figure: **Top :** average reward of "ϵ-greedy action-value strategy for the k-armed example. Data is averaged over M runs. **Bottom :** the number of times the actions taken were indeed optimal.
Implementation detail remark: the initial values of the actions are taken to be all zero. This means that for $\epsilon = 0$ some exploration may be done beyond the first step but this is very limited.

Multi-armed bandit : computational considerations: incremental computations

Question: how to compute efficiently Q_t ?
Idea : incremental.
If rewards R_1 ... R_n have been obtained for some action a (chosen $n-1$ times) : $Q_n = \frac{R_1+...+R_{n-1}}{n-1}$ then the $n+1$ time :
$Q_{n+1} = \frac{R_1+...+R_n}{n} = \frac{(n-1)Q_n+R_n}{n}$ thus $Q_{n+1} = Q_n + \frac{1}{n}[R_n - Q_n]$
This idea appears very often
$NewEstimation = OldEstimation + \alpha_t \cdot [Target - OldEstimation]$
Vocabulary:

- "Target - OldEstimation" is called an "error"
- $\alpha = 1/n$ (here) is called the step size.

Multi-armed bandit : computational considerations: incremental computations

Question: what if the rewards are changing in time ?
Idea: give more weight to recent values in the estimation, i.e. replace the polynomial decay $\alpha = 1/n$ by some exponential decay (like in "exponentially moving averages").
Formula $Q_{n+1} = Q_n + \alpha[R_n - Q_n]$
All terms formula: $Q_{n+1} = (1-\alpha)^n Q_1 + \sum_{\ell=1}^{n} \alpha(1-\alpha)^{n-\ell} R_\ell$
Question: what properties for α_n ?
Answer : $\sum_\ell \alpha_\ell = \infty$ and $\sum_\ell \alpha_\ell^2 < \infty$.
Comments: need to both have limited variance and also erase any error in the initial steps; cf. also SGD convergence

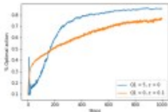
Multi-armed bandit : Python implementation pseudocode

Pseudocode : MAB

- initialize k (no. of arms), T (no. of time steps), M (no. of realizations), ϵ, vector of rewards (shape $T \times M$)
- iterate over realizations:
  - initialize (random from $\mathcal{N}(0,1)$) the true averages $q_*(a)$, $a = 1, ..., k$
  - initialize $N(a)$ (as vector) to 0 and $Q(a) = 0$
  - iterate over time steps
    - draw uniformly $x \in [0,1]$;
    - if $x < \epsilon$ then next action A is at random in the k list of actions
    - if $x \geq \epsilon$ then next action $A = \arg\max_a Q(a)$ breaking ties at random
    - update $N(A) = N(A) + 1$
    - add reward R to the reward list $R = q_*(A) + \mathcal{N}(0,1)$
    - update $Q(A) = Q(A) + \frac{1}{N(A)}[R - Q(A)]$
- plot mean over realization of rewards R (eventually w/r to $\mathbb{E}[\max_a q_*(a)]$)

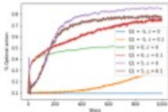
Multi-armed bandit : optimism

The initial values Q_1 have been selected at some arbitrary value, here 0. What if we select them to a different level ?
E.g. "optimistic" initial values 5; then, even if the full-greedy strategy will explore more and be more successful.



Multi-armed bandit : optimism

TODO (implementations) : compare optimism in two settings: $Q_1 < 0$ vs $Q_1 > 0$



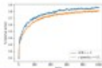
Multi-armed bandit : optimize estimations by confidence intervals

Idea: a 'bad arm' should not be tried many times. How can one know it is 'bad' ? If we are "confident" that it is bad .... use confidence intervals !

- since we are maximizing, it is maybe a good idea to use the upper bound of the confidence interval: the UCB method.
- in practice choose A that maximizes $Q_t(a) + c\sqrt{\frac{\ln(t)}{N_t(a)}}$ (recall $\sigma = 1$) with c a constant e.g. $c = 2$ (related to Hoeffding inequality / subgaussian variables, see [2])

TODO : implementation (compare with LCB !)

Remarks: works well, but need attention in unsteady settings; difficult to generalize well to large state spaces (k).



Multi-armed bandit : gradient algorithms

We next formalize the choice of an action a as a "preference" $H_t(a)$ which in technical terms gives rise to a probability law through a softmax function $P(A_t = a) = \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}} =: \pi_t(a)$

The object $\pi_t(a)$ (probability to take action a) is initialized with $H_1 = 0$.

Idea: stochastic gradient (ascent) on H_t :

$H_{t+1}(A_t) := H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t))$

For $a \neq A_t$: $H_{t+1}(a) := H_t(a) - \alpha(R_t - \bar{R}_t)\pi_t(a)$

The baseline $\bar{R}_t$ = average of rewards up to time t (not including time t) computed incrementally (cf. non stationary settings).

Idea: increase probability of action A_t when giving rewards higher than the average, decrease if not.

For non selected actions : move in opposite direction.

The baseline ensures quick adaptation and makes algorithm faster.

Multi-armed bandit : theoretical insights into gradient algorithms

- Perspective: stochastic optimization approach (e.g. like Stochastic Gradient Descent [1]) to maximize the expected reward $\mathcal{R} = \mathbb{E}[R_t] = \sum_b q_*(b)\pi_t(b)$ w/r to H_t which define π_t.
- softmax derivation rule : $\nabla_{H_t(a)}\pi_t(b) = \pi_t(b)(\mathbb{1}_{b=a} - \pi_t(a))$
- Recall: SGD uses a non-biased version of the gradient, possibly involving some random variable here A_t
- $\nabla_{H_t(a)}\mathcal{R} = \nabla_{H_t(a)}\left(\sum_b q_*(b)\pi_t(b)\right) = \sum_b q_*(b)\pi_t(b)(\mathbb{1}_{b=a} - \pi_t(a))$ $= \mathbb{E}_{A_t}[q_*(A_t)(\mathbb{1}_{A_t=a} - \pi_t(a))]$
- $R_t(\mathbb{1}_{a=A_t} - \pi_t(a))$ = unbiased estimator for $\nabla_{H_t(a)}\mathcal{R}$ because $q_*(A_t) = \mathbb{E}[R_t|A_t]$; we use it in the SGD update of H_{t+1}.

---

[1] Gabriel Turinici. The convergence of the Stochastic Gradient Descent (SGD) : a self-contained proof. arXiv:2103.14350 [cs, math, stat], March 2021

Multi-armed bandit : theoretical insights into gradient algorithms

- Next idea: minimize $\mathcal{R}$ or $\mathcal{R} - c$ is the same (cst. c independent of A_t)
- $\mathcal{R} - c = \sum_b (q_*(b) - c)\pi_t(b)$
- $\nabla_{H_t(a)}(\mathcal{R} - c) = ... = \mathbb{E}_{A_t}[(q_*(A_t) - c)(\mathbb{1}_{A_t=a} - \pi_t(a))]$
- Choice for c ? Idea: consistency "if we are already in the solution move the least possible" : $c = \bar{R}_t$ (baseline) (e.g. take a situation with 2 or 3 actions having same q_*); can be seen as variance reduction technique [4, 5]. Other interpretation : consider that the rewards for arms not chosen at t to be $\bar{R}_t$ and not 0 as would be with $c = 0$ (same formula with the case when one can obtain the value of all rewards simultaneously).
- Final update formula $H_{t+1}(a) = H_t(a) + \alpha(R_t - \bar{R}_t)(\mathbb{1}_{a=A_t} - \pi_t(a))$ as expected.

Multi-armed bandit : additional optimal baseline considerations

Alternative view: cf. the SGD convergence proof [3], the optimal value of the constant c will minimize $\mathbb{E}_{A_t}[\|\nabla_{H_t}\mathcal{R}(A_t, H_t)\|^2]$; we obtain, after some computations, that (we denote $\bar{R}_t = \mathbb{E}_{A_t}[q_*(A_t)] = \sum_a \pi(a)q_*(a)$):

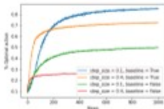
$$c_{optimum} = \frac{(1+\|\pi\|^2)\bar{R}_t - 2\sum_a \pi^2(a)q_*(a)}{1-\|\pi\|^2}$$

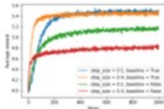
This cannot be computed online because requires knowledge of too many quantities; but at start, the distribution π is uniform over the k choices and the formula reduces to

$$c_{optimum} \simeq \frac{(1+1/k)\bar{R}_t - 2\bar{R}_t/k}{1-1/k} = \bar{R}_t.$$

Multi-armed bandit : empirical effect of using baseline

Impact of using or not a baseline (the true values $q_* \sim \mathcal{N}(4, 1)$):





Practice

Correct chatGPT code on www to implement policy gradient MAB with baseline.

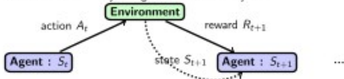
Related algo: REINFORCE which iterates between computing an episode under a given π and updating π (step by step in time) using rewards stored.

Outline

1. AI and deep learning : introduction
2. Building blocks of strategies : the Multi-armed bandit
   - Presentation
   - Basic strategy: ϵ-greedy
   - MAB: optimism
   - MAB: Confidence intervals
   - MAB: policy gradient algorithms
3. **Formalization: (Finite) Markov decision processes (MDP), value function and Bellman equation**
4. Bellman : policy evaluation, improvement, value iteration
5. Monte Carlo approaches
6. Reinforcement learning frameworks : Open AI Gym
7. Temporal difference learning
8. Learning strategies : Q-learning, double Q-learning, SARSA
   - Q-Learning
   - Double / Dual Q-Learning
   - SARSA and variants

Finite Markov Decision Processes (MDP)

Idea: formalize the cycle Agent-Environment cycle :



Object sequencing : $S_0, A_0, R_1, S_1, A_1, \ldots$
Important object $p(s', r|s, a) := \mathbb{P}[S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a]$
Environment : states $S_t \in \mathcal{S}$; actions $A_t \in \mathcal{A}$ ...
Note: not all states $s \in \mathcal{S}$ allow all actions $a \in \mathcal{A}$.

Finite Markov Decision Processes (MDP)

Definition

A Markov decision process (MDP) is a 4-tuple $(\mathcal{S}, \mathcal{A}, \mathbb{P}, R)$, where:
$\mathcal{S}$ is a set of states called the state space,
$\mathcal{A}$ is a set of actions called the action space ($\mathcal{A}_s$ are actions available from state $s \in \mathcal{S}$);
$p(s', r|s, a)$ is the probability to go from state s to state s' with reward r after having chosen action a. We denote $\mathbb{P}$ the whole set of probabilities p.
$R(s, a, s')$ is the immediate reward (or expected immediate reward) received from the transition from s to s' due to action a.
A policy function π is a (potentially probabilistic) mapping from state space $\mathcal{S}$ to action space $\mathcal{A}$.

One can write $\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$
The state and action spaces may be finite or infinite, for example $\mathbb{N}$, $\mathbb{R}$.
Sometimes one includes γ (discount rate) into the MDP definition.

MDP: remarks

- for any $s \in \mathcal{S}$ and $a \in \mathcal{A}_s$: $p(\cdot, \cdot | s, a)$ is a probability law
- when there is no ambiguity we only write $\mathcal{A}$ instead of $\mathcal{A}_s$
- Goals are not the same as rewards ... sometimes we need to encode the rewards to be coherent with our goals, e.g. robot, car parking, etc.
- there may be episodes or infinite horizon games
- return $G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ where γ is the discount rate
- important remark $G_t = R_{t+1} + \gamma G_{t+1}$
- Example : pole balancing; goal: keep angle below some threshold
- state value function for policy π

$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{k+t+1} \middle| S_t = s\right]$

- the action-value function for policy π

$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{k+t+1} \middle| S_t = s, A_t = a\right]$

Coherence condition : Bellman equation

Theorem (Bellman equation for state value function and action-state value function)

The following relation hold

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \tag{1}$$

or, in developped form

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]. \tag{2}$$

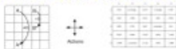
Similar considerations for state-action value function q :

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a]. \tag{3}$$

Proof: use $v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$ and $q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$.

Bellman equation : gridworld example

Example : gridworld [1, chapter 3] Rules : the cells of the grid correspond to the states of the environment. At each cell, four actions are possible: north, south, east, and west, which deterministically cause the agent to move one cell in the respective direction on the grid. Actions that would take the agent off the grid leave its location unchanged, but also result in a reward of -1. Other actions result in a reward of 0, except those that move the agent out of the special states A and B. From state A, all four actions yield a reward of $+10$ and take the agent to A'. From state B, all actions yield a reward of $+5$ and take the agent to B'.



Left figure : environment, center: actions, right: state value function for the uniform policy ($\gamma = 0.9$)

Attention, value function takes into account the future, e.g. value of A is less than 10 and value of B more than 5; some values are negative

Bellman equation : gridworld example

Example : gridworld [1, chapter 3]
Left figure : environment, center: actions, right: state value function for the uniform policy ($\gamma = 0.9$)

Exercice: check state value Bellman equation
$v_\pi(s) = ... = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$ for some states.
state at (1, 1) value $3.31 = 0.9 * (1/4) * (8.79 + 1.52 + 3.31 + 3.31) - 1/4 - 1/4$;
state at (1, 3) of value $4.43 = 0.9 * 1/4(5.82 + 2.25 + 8.79 + 4.43) - 1/4$;
state at (1, 2) of value $8.79 = 10 + 0.9 * (-1.35)$

Exercice 2 : add a constant to all rewards; how does v changes ?

Important remark : Bellman equation can be solved through a linear system.

Bellman optimality equation

Theorem (Bellman's principle of optimality; R.E. Bellman, Dynamic Programming, 1957))

An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision

Ordering of policies $\pi \geq \pi'$ iff $v_\pi(s) \geq v_{\pi'}(s), \forall s \in \mathcal{S}$.
Optimal policy : a policy better than or equal to anyone else.
optimal state value function $v_*(s) = \max_\pi v_\pi(s)$ $\forall s \in \mathcal{S}$
optimal action-value function $q_*(s,a) = \max_\pi q_\pi(s,a)$ $\forall s \in \mathcal{S}, a \in \mathcal{A}_s$.
Rq: $q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a]$ $\forall s \in \mathcal{S}, a \in \mathcal{A}_s$.

Bellman optimality equation

Theorem (optimal Bellman equation)

The optimal value function and state-action value function satisfy

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}_s} \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_{a \in \mathcal{A}_s} \sum_{s', r} p(s', r | s, a)[r + \gamma v_*(s')] \end{aligned} \tag{4}$$

$$\begin{aligned} q_*(s, a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a' \in \mathcal{A}_{S_{t+1}}} q_*(S_{t+1}, a') | S_t = s, A_t = a] \\ &= \sum_{s', r} p(s', r | s, a)[r + \gamma \max_{a'} q_*(s', a')] \end{aligned} \tag{5}$$

Rq: the equations have no π in them !

Bellman optimality equation

Theorem (existence for optimal policy)

There exists at least a optimal policy. All optimal policies share the same optimal state value function v_* and optimal action-state value function q_*.

Proof idea: recall $v_*(s) = \max_{a \in \mathcal{A}_s} q_*(s, a)$

- define the Bellman operator (update operator)

$v \mapsto v' = B(v): \ v'(s) = \max_{a \in \mathcal{A}_s} \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) | S_t = s, A_t = a]$

- prove B is a contraction, fixed point is an optimum

Bellman optimality equation

Example : gridworld [1, chapter 3]

	1	2	3	4	5
1	21.98	24.42 (A)	21.98	19.42 (B)	17.48
2	19.78	21.98	19.78	17.8	16.02
3	17.8	19.78	17.8	16.02 (B')	14.42
4	16.02	17.8	16.02	14.42	12.98
5	14.42	16.02 (A')	14.42	12.98	11.68

	1	2	3	4	5
1	[illegible]	[illegible] (A)	[illegible]	[illegible] (B)	[illegible]
2	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
3	[illegible]	[illegible]	[illegible]	[illegible] (B')	[illegible]
4	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
5	[illegible]	[illegible] (A')	[illegible]	[illegible]	[illegible]

Left figure : optimal value function, right: optimal policy ($\gamma = 0.9$)
Exercice : check that the state value function above satisfies the optimality equation $v_*(s) = \max_{a \in \mathcal{A}_s} q_*(s, a)$

Bellman optimality equation

Important remark: having v_* or q_* makes choosing actions easy (q_* is

	1	2	3	4	5
1	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
2	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
3	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
4	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
5	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

	1	2	3	4	5
1	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
2	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
3	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
4	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
5	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

easier!)

How to find v_* / q_* / π_* ?

Outline

1. AI and deep learning : introduction
2. Building blocks of strategies : the Multi-armed bandit
   - Presentation
   - Basic strategy: ϵ-greedy
   - MAB: optimism
   - MAB: Confidence intervals
   - MAB: policy gradient algorithms
3. Formalization: (Finite) Markov decision processes (MDP), value function and Bellman equation
4. **Bellman : policy evaluation, improvement, value iteration**
5. Monte Carlo approaches
6. Reinforcement learning frameworks : Open AI Gym
7. Temporal difference learning
8. Learning strategies : Q-learning, double Q-learning, SARSA
   - Q-Learning
   - Double / Dual Q-Learning
   - SARSA and variants

Bellman : policy evaluation (prediction)

Goal: estimate v_π when π is given, fixed (known).
Recall $v_\pi(s) = \sum_{a \in \mathcal{A}_s} \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$

- Memory consuming idea : solve linear system (deterministic setting)

- in practice: iterative procedures :

Policy evaluation

for all $s \in \mathcal{S}$: $v_{k+1}(s) = \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s]$
or equivalently
$v_{k+1}(s) = \sum_{a \in \mathcal{A}_s} \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_k(s')]$

One can prove (in general) that $v_k \to v_\pi$ for $k \to \infty$.

Bellman : policy evaluation (prediction)

Faster version : replace new value immediately upon computation

Pseudocode : fast policy evaluation

- input : π, error threshold θ
- initialize V (size of V = size of $\mathcal{S}$)
- repeat
  - set max error $e = 0$
  - for any $s \in \mathcal{S}$
    - store old value $v = V(s)$
    - assign $V(s) = \sum_{a \in \mathcal{A}_s} \pi(a|s) \sum_{s', r} p(s', r|s, a)[r + \gamma V(s')]$
    - $e = \max\{e, |v - V(s)|\}$
- until $e < \theta$

Bellman : policy improvement

How to improve the policy π for which we know v_π ?
For any $s \in \mathcal{S}$ compare $v_\pi(s)$ with $q_\pi(s,a)$ for any $a \in \mathcal{A}_s$. If larger than $v_\pi(s)$ then one should take action a.
Greedy policy : $\pi'(s) = \arg\max_{a \in \mathcal{A}_s} q_\pi(s,a)$
Replacing π with π' (or any other better than π) is called policy improvement.
When several improvements are executed one after another, we obtain a policy iteration, which is a iterative two-steps procedure: policy evaluation, policy improvement, policy evaluation, ...
In the policy improvement step we loop over all states to see whether improvement exists at least in one state.

Bellman : value iteration for estimating v_*

To find v_* (keyword : value iteration) we can apply the Bellman operator as many times as needed to convergence.
Note : does not need to update whole vector of values, one at a time is enough (stochastic gradient style)
Recall $v_*(s) = \max_{a \in \mathcal{A}(s)} \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')]$

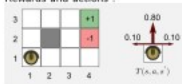
Pseudocode : value iteration

- initialize V (size of V = size of $\mathcal{S}$), error threshold θ
- repeat
  - error $e = 0$
  - for any $s \in \mathcal{S}$:
    - assign $V(s) = \max_{a \in \mathcal{A}_s} \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
    - update error $e = \max\{e, \text{absolute change in V}\}$
- until $e \leq \theta$

Value iteration example

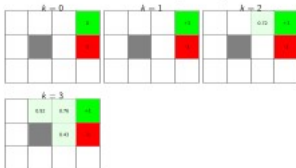
Example [6, chap 17] : robot on a 3×4 grid, reward $+1$ at $(4,3)$ (terminal state), -1 at $(4,2)$ (terminal state), obstacle at $(2,2)$; start in $(1,1)$. proba $1-e$: action as planned; proba $e/2$ skids to left of action, $e/2$ skids to right of action; $\gamma = 0.9$, $e = 0.2$, rewards $r = 0$ elsewhere.

Rewards and actions :



Value iteration example

Iterations : from bottom left, x (y?) dimension first; NOTE: final states are taken to have value equal to their reward. THIS IS A CHOICE (on next slide the other convention is given too)!



Value iteration example

At convergence we obtain the optimal value function v_*. It can be used to obtain the optimal strategy π_*.

$k = \infty$, final states value=reward

0.64	0.74	0.85	+1.0
0.57		0.57	-1.0
0.49	0.43	0.48	0.28

$k = \infty$; final states value=0

0.72	0.83	0.94	0
0.63		0.64	0
0.55	0.48	0.53	0.31

Practice

Implement value iteration.

Outline

1. AI and deep learning : introduction
2. Building blocks of strategies : the Multi-armed bandit
   - Presentation
   - Basic strategy: ϵ-greedy
   - MAB: optimism
   - MAB: Confidence intervals
   - MAB: policy gradient algorithms
3. Formalization: (Finite) Markov decision processes (MDP), value function and Bellman equation
4. Bellman : policy evaluation, improvement, value iteration
5. **Monte Carlo approaches**
6. Reinforcement learning frameworks : Open AI Gym
7. Temporal difference learning
8. Learning strategies : Q-learning, double Q-learning, SARSA
   - Q-Learning
   - Double / Dual Q-Learning
   - SARSA and variants

Monte Carlo approaches

- Monte Carlo approaches: use just repeated experience
- Value prediction i.e., compute V_π for given π: simulate many experiences according to π, then measure reward starting from any encountered state, compute average and output this as V_π
- Action-state values : similar approach

- Monte Carlo control i.e., policy improvement : replace $\pi(s)$ by $\arg\max_a q(s,a)$

- Problems not all state-action pairs are visited, convergence may be slow, ...

Outline

1. AI and deep learning : introduction
2. Building blocks of strategies : the Multi-armed bandit
   - Presentation
   - Basic strategy: ϵ-greedy
   - MAB: optimism
   - MAB: Confidence intervals
   - MAB: policy gradient algorithms
3. Formalization: (Finite) Markov decision processes (MDP), value function and Bellman equation
4. Bellman : policy evaluation, improvement, value iteration
5. Monte Carlo approaches
6. Reinforcement learning frameworks : Open AI Gym
7. Temporal difference learning
8. Learning strategies : Q-learning, double Q-learning, SARSA
   - Q-Learning
   - Double / Dual Q-Learning
   - SARSA and variants

Open AI Gym

A library for RL, contains some environments (e.g., games), helper functions and algo.

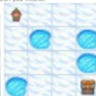
- Methods : gym.make(); env.reset(); env.step(action): takes a step, returns (state,reward,terminated,trucated, info)
Other methods : env.seed(), env.render(), env.close()

- Attributes : `gym.Env.action_space`, `gym.Env.observation_space`, `gym.Env.reward_range`, `gym.Env.metadata`, `gym.Env.spec`,

- gym.space class: `env.action_space.sample()`, `env.action_space.contains(action)`,
- list gym environments : `gym.envs.registry.values()`,

Open AI Gym: Frozen lake

Description (cf [7, 8]) Winter is here. You and your friends were tossing around a frisbee at the park when you made a wild throw that left the frisbee out in the middle of the lake. The water is mostly frozen, but there are a few holes where the ice has melted. If you step into one of those holes, you'll fall into the freezing water. At this time, there's an international frisbee shortage, so it's absolutely imperative that you navigate across the lake and retrieve the disc. However, the ice is slippery, so you won't always move in the direction you intend.



Credits : left : `https://www.pandotrip.com/wp-content/uploads/2015/11/Chagnaqtis-Photo-by-Matthieu-Paley2-980x711.jpg` ; right : [7].

Open AI Gym

Practice

Try and play, at random, one game in gym: "FrozenLake-v1", "CartPole-v0", "MountainCar-v0", ...

```
import numpy as np
import gymnasium as gym #to load the FrozenLake Environment

#this command creates the FrozenLake environment using "gym"
env = gym.make("FrozenLake-v1")

#test each one
env.reward_range
env.metadata
env.spec
env.action_space.contains(2)
```

Open AI Gym

Practice

```
state=env.reset()#initialize
#create table
action_size = env.action_space.n#how many actions there are
state_size = env.observation_space.n#how many states
qtable = np.zeros((state_size, action_size))
# variant : initialize uniformly over reward space
print(qtable)

#choose one of those :
action = np.argmax(qtable[0,:])#here state=0
action = env.action_space.sample()
# Take the action (a) and observe the outcome state(s')
# and reward (r)
new_state, reward, term, trunc, info = env.step(action)
```

Open AI Gym

Practice

```
# install gymnasium[toy-text,atari,accept-rom-license],
# with jupyter, spyder : install pygame for nicer display
# gymnasium  as of 03/2024 : use render_mode="human"
# or render_mode="rgb_array"
# env = gym.make("FrozenLake-v1",is_slippery=True)

# to play a game use repeatedly :
# new_state, reward, term,trunc, info = env.step(action)
# choose action among 0,1,2,3, then look at state with :
# env.render()
```

Practice

Cheating : use the Bellman iterations to compute the V/Q tables using the code provided `value_function_frozen_lake_Bellman_iter.py`.

Outline

1. AI and deep learning : introduction
2. Building blocks of strategies : the Multi-armed bandit
   - Presentation
   - Basic strategy: ϵ-greedy
   - MAB: optimism
   - MAB: Confidence intervals
   - MAB: policy gradient algorithms
3. Formalization: (Finite) Markov decision processes (MDP), value function and Bellman equation
4. Bellman : policy evaluation, improvement, value iteration
5. Monte Carlo approaches
6. Reinforcement learning frameworks : Open AI Gym
7. **Temporal difference learning**
8. Learning strategies : Q-learning, double Q-learning, SARSA
   - Q-Learning
   - Double / Dual Q-Learning
   - SARSA and variants

Time-difference learning

- Idea: combine Monte Carlo with dynamic programming (Bellman equation), update information on value function V or state-action value Q on the fly ...

- Can also be seen as a version of SGD ...

- Can be more reactive then Monte Carlo because updates the q/v tables on the fly, e.g. as in online GPS-like update ETA as compared to a posteriori updates

- in $TD(0)$ we update current state based on difference between the current estimate and the estimate of the next state plus the reward; in $TD(1)$ we use all previous states and for $\lambda \in [0, 1]$ state weight decays each time by a factor λ.
- Examples follow ...

Outline

1. AI and deep learning : introduction
2. Building blocks of strategies : the Multi-armed bandit
   - Presentation
   - Basic strategy: ϵ-greedy
   - MAB: optimism
   - MAB: Confidence intervals
   - MAB: policy gradient algorithms
3. Formalization: (Finite) Markov decision processes (MDP), value function and Bellman equation
4. Bellman : policy evaluation, improvement, value iteration
5. Monte Carlo approaches
6. Reinforcement learning frameworks : Open AI Gym
7. Temporal difference learning
8. Learning strategies : Q-learning, double Q-learning, SARSA
   - Q-Learning
   - Double / Dual Q-Learning
   - SARSA and variants

Q-Learning

- Off-policy learning algorithm, meaning that the policy followed is not changing, e.g. a robot with a hardwired policy that we want to improve. The order of states presented to us is NOT our choice, but we can still learn the optimal state-action q_* function by using the max over the actions when updating the candidate for q_*.
- If no policy is given one can use ϵ-greedy for instance (or any fixed one).
- Advantage of off-policy : can use replays from old episodes / games...
- Dis-advantage: has to compute the max over actions...
- Rq : as an "off-policy" algorithm, the Q-values are updated based on the rewards received for the action taken, which is not necessarily the optimal action (according to the current Q-values). But it will still converge to the optimal policy which is generally different from the policy used to take actions during training.

Q-Learning

Q-learning (Watkins 1989)

$$Q^{new}(S_t, A_t) \leftarrow \underbrace{Q(S_t, A_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\bigg(\underbrace{\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(S_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(S_t, A_t)}_{\text{old value}} \bigg)}^{\text{temporal difference}}$$

[from Wikipedia with adaptations] $0 < \alpha \leq 1$

Note: it the policy is not given in advance (true "off-policy") it can be chosen by default e.g. ϵ-greedy.

Q-Learning

Q-learning pseudocode

- read input parameters α, $\epsilon > 0$
- initialize all $Q(s, a)$ (all admissible) arbitrary except for terminal values set to 0.
- Loop over each episode
  - initialize S
  - loop over each step of the current episode
    - choose A according to the policy, here ϵ-greedy (can also be a given policy)
    - take action A, observe R, S'
    - update $Q(S, A) = Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
    - replace S by S'
  - until S is terminal

Practice

Implement Q-learning on Frozen-Lake/ Taxi; adjust hyper-parameters.

q = 2D array of shape : (env.observation_space.n,env.action_space.n).

Double / Dual Q-Learning

- Idea: eliminate some bias coming from the maximization step which is biased: the "max" has a different distribution as the initial distribution e.g. for X, Y independent $\mathcal{N}(0,1)$:

$$\mathbb{E}[\max(X,Y)] = \mathbb{E}[X\mathbb{1}_{X\geq Y} + Y\mathbb{1}_{Y\geq X}] = \mathbb{E}[X\mathbb{1}_{X\geq Y} - Y\mathbb{1}_{X\geq Y}] =$$
$$\mathbb{E}[(X-Y)\mathbb{1}_{X-Y\geq 0}] \overset{Z=X-Y\sim\mathcal{N}(0,2)}{=\!=\!=\!=} \mathbb{E}[Z\mathbb{1}_{Z\geq 0}] = 2/\sqrt{\pi} > 0.$$

Thus $\mathbb{E}[\max(X,Y)] > \max(\mathbb{E}[X], \mathbb{E}[Y])$ (biased !!)

- Example: suppose one measures the height of some trees which in fact have all same height, $= H$. BUT measurement induces an error e (symmetric); then $\max(H+e) > H$!
- To avoid this bias use two measurements : one to decide which is the tallest tree an independent one to decide its height.

Double Q-Learning

- Solution: double / dual Q-learning: extract max using another Q-table

Double Q-learning (van Hasselt 2011)

$$Q^2_{t+1}(S_t, A_t) = Q^2_t(S_t, A_t) + \alpha_t(S_t, A_t)\left(R_{t+1} + \gamma\, Q^1_t\left(S_{t+1}, \arg\max_a Q^2_t(S_{t+1}, a)\right) - Q^2_t(S_t, A_t)\right).$$

$$Q^1_{t+1}(S_t, A_t) = Q^1_t(S_t, A_t) + \alpha_t(S_t, A_t)\left(R_{t+1} + \gamma\, Q^2_t\left(S_{t+1}, \arg\max_a Q^1_t(S_{t+1}, a)\right) - Q^1_t(S_t, A_t)\right)$$

[from Wikipedia with adaptations] $0 < \alpha \leq 1$

Double Q-Learning

Double Q-learning pseudocode

- read input parameters α, $\epsilon > 0$
- initialize $Q^1(\cdot,\cdot)$ and $Q^2(\cdot,\cdot)$ as before
- Loop over each episode
  - initialize S
  - loop over each step of the current episode
    - choose A according to the policy, here ϵ-greedy (can also be a given policy) in $Q^1 + Q^2$
    - take action A, observe R, S'
    - set $U = 1$ or $U = 2$ (50%/50%), denote V the other
    - $Q^U(S,A) = Q^U(S,A) + \alpha[R + \gamma Q^V(S', \arg\max_a Q^U(S',a)) - Q^U(S,A)]$
    - replace S by S'
  - until S is terminal

Practice

Implement Double Q-learning on Frozen-Lake.

Deep learning strategies : SARSA and variants

Bellman iterations: iterate : loop over states, compute max (loop over actions)
Q-learning : iterate : game play with loop over actions
To minimize cost: eliminate the need to loop over actions (in the 'max')
On-policy : the strategy is updated as we progress towards the optimal q_*.
May require less evaluations than the standard Q-learning.

SARSA for estimating q_*

$$Q^{new}(S_t, A_t) \leftarrow \underbrace{Q(S_t, A_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Bigg(\underbrace{\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{Q(S_{t+1}, A_{t+1})}_{\text{estimate of future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(S_t, A_t)}_{\text{old value}} \Bigg)}^{\text{temporal difference}}$$

[from[1] and Wikipedia with adaptations] $0 < \alpha \leq 1$

SARSA pseudocode

- read input parameters α, $\epsilon > 0$
- initialize all $Q(s, a)$ (all admissible) arbitrary except for terminal values set to 0.
- Loop over each episode
  - initialize S
  - choose A from S according to Q (e.g., ϵ-greedy)
  - loop over each step of the current episode
    - take action A, observe R, S'
    - choose A' from S' according to Q (e.g., ϵ-greedy)
    - update $Q(S, A) = Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
    - replace S by S', A by A'
  - until S is terminal

Practice

Implement SARSA on Frozen-Lake using previous (Q-learning) implementation.

Expected SARSA

Expected SARSA for estimating q_* [from[1, 9] and Wikipedia with adaptations]

$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\right]$
$0 < \alpha \leq 1$

Value function style : denote $V_\pi(S_{t+1}) := \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a)$
Update : $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma V_\pi(S_{t+1}) - Q(S_t, A_t)\right]$

Expected SARSA can be used both off and on-policy : in both cases calculation is done as an expectation value but policy π can be derived from Q (e.g., ϵ-greedy) or not.

Expected SARSA : precise computations for π=Q-ϵ-greedy :
$V_\pi(S) = \epsilon \cdot numpy.mean(Q[S,:]) + (1-\epsilon) \cdot numpy.max(Q[S,:])$.

Outline

1. AI and deep learning : introduction
2. Building blocks of strategies : the Multi-armed bandit
   - Presentation
   - Basic strategy: ϵ-greedy
   - MAB: optimism
   - MAB: Confidence intervals
   - MAB: policy gradient algorithms
3. Formalization: (Finite) Markov decision processes (MDP), value function and Bellman equation
4. Bellman : policy evaluation, improvement, value iteration
5. Monte Carlo approaches
6. Reinforcement learning frameworks : Open AI Gym
7. Temporal difference learning
8. Learning strategies : Q-learning, double Q-learning, SARSA
   - Q-Learning
   - Double / Dual Q-Learning
   - SARSA and variants

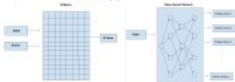
Deep Q-Learning

- can adapt better to new settings; can treat larger state spaces. High impact paper : [10] that used same architecture to play several Atari games.

Practice

Load "Breakout" Atari game, "play" some moves.

- Idea: replace the Q table by a mapping constructed, e.g. with a deep neural network.
- There are several variants, often used is that the network constructs the map $s \mapsto Q(s,a)_{a \in \mathcal{A}(s)}$ as a vector map from $\mathcal{S}$ to $\mathbb{R}^{|\mathcal{A}|}$.



Deep Q-Learning

- Recall : Q-learning = off-policy (see Q-learning slides) with updates :
$$Q^{new}(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \Big)$$
- Loss : squared Bellman error loss:
$\Big(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \Big)^2$; here
$y_t = R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$ is treated as a fixed "label" (no differential computed).
- SGD-like updates : ϕ network parameters $Q = Q_\phi$; notation i for (S, A) :
- Algorithm : take some action a_i to observe s_i', r_i from s_i, a_i
$$\phi \leftarrow \phi - \alpha \partial_\phi (Q_\phi(s_i, a_i) - y_i)^2 = \phi - \alpha \partial_\phi Q_\phi(s_i, a_i)(Q_\phi(s_i, a_i) - y_i)$$
$$\phi \leftarrow \phi - \alpha \partial_\phi Q_\phi(s_i, a_i)(Q_\phi(s_i, a_i) - [r_i + \gamma \max_a Q_\phi(s_i', a_i)])$$
- Problems for SGD : there is ϕ also in y_i ; the s_i, a_i are not iid

Deep Q-Learning

$\phi \leftarrow \phi - \alpha \partial_\phi Q_\phi(s_i, a_i)(Q_\phi(s_i, a_i) - [r_i + \gamma \max_a Q_\phi(s'_i, a_i)])$

- Problem for SGD : there is ϕ also in y_i
- Answer: the target map is given by a non-trainable copy of the neural network updated every "K" iterations, i.e. use $\max_a Q_{\phi'}$ instead of $\max_a Q_\phi$ with $\phi' = \phi$ every K iterations. $Q_{\phi'}$ = "target network"

- Problem for SGD : the s_i, a_i are not iid
- Answer: experience replay : a buffer of transitions (offline policy) $(s_i, a_i, s'_i, r_i)_{i=1}^{B}$ then sample from this batch to compute gradient. Buffer is updated regularly with new samples (FIFO).

- other concepts : eligibility traces (... later), $TD(\lambda)$

Deep Q-Learning

Deep Q-learning with Experience Replay and target refresh

- read input parameters $\epsilon > 0$, α (learning rate), replay memory capacity $N_{\mathcal{M}}$, memory refresh size M_r, target refresh T_r, batch size $B > M_r$
- initialize $Q(\cdot, \cdot)$ at random, initialize network, target network, reset environment etc.
- Initialize replay memory $\mathcal{M}$, fill up to capacity $N_{\mathcal{M}}$ (run env)
- Loop until maximum iterations reached
  - add to replay buffer M_r transitions
    - choose A according to the policy, here ϵ-greedy
    - take action A, observe R, S'
    - store transition S, A, R, S' in $\mathcal{M}$ (FIFO-style, max capacity=$N_{\mathcal{M}}$)
  - sample B transitions from $\mathcal{M}$
  - perform a step of stochastic optimization of the quadratic Bellman error loss (use the target network) on the sampled batch
  - if iteration number = multiple of T_r update target network

Deep Q-Learning

Practice

Using "Atari Breakout" practice the game through a gymnasium interface. Same for Pong.

Practice

Run the policy gradients DQN "Pong" (Karpathy style) implementation given.

Practice

Run the DQN "Atari Breakout" implementation given.
- change the network structure by adding FC or Conv2D layers; which one is more efficient ?
- test updating more often or less often the target network
- test putting more images in the observation object
- test with another Atari game

Policy gradient on Pong (Karpathy style)

- Policy Gradient methods are often better than DQN

- Reminder : PG optimizes expected reward, is a Markov decision process; the strategy is given in the form of a probability law π and actions are drawn from this law.

- PG Pong implementation: the NN will parameterize the actions law from an input which is the difference of two consecutive game images (to obtain the information on ball movement cf. also Breakout); output is the probability to go up.

- NN architecture: 2 FC layers : input 80×80 to $\mathbb{R}^{200}$, ReLU activated + FC $\mathbb{R}^{200}$ to $\mathbb{R}$ sigmoid activated($1/(1 + exp(-x))$); the output will be interpreted as the probability to select "up" action (=2 in Atari/Pong); for instance the forward pass is $h = W_1 x$, $logp = W_2 h$, $p = sigmoid(logp)$. Implementation in pure python of the fdw and bkwd pass. Optimizer = RMS prop.

Policy gradient on Pong (Karpathy style)

- note: there some preprocessing of the frames to eliminate useless parts and reduce from $210 \times 160 \times 3$ to 80×80

- credit assignment (was a given action "good" or "bad" ?) is difficult here; rewards are calculated from future (until end of game) by discounting with formula : $reward = \sum \gamma^k R_{t+k}$.

TODO

- run notebook from my site to test packages
- load a converged result from same site (adapt name of the package)
- play a game selecting actions with NN and rendering result

- Recommended reading: Karpathy's blog 2016 [11], with code link.

- Also useful: policy gradient (PG) variants like TRPO (trust region PG), PPO

Policy gradient on Pong (Karpathy style)

- some technical details 2-dim PG: if $\mathbb{P}(\text{"} action = up \text{"}) = \frac{1}{1+e^{-\lambda}} = \sigma$ and rewards are r_1 and r_2 in the two states, the total reward will be $\sigma r_1 + (1-\sigma) r_2$; its derivative with respect to λ is $\sigma(1-\sigma)(r_1 - r_2)$. We can write :

$$\begin{aligned}
&\sigma(1-\sigma)(r_1 - r_2) \\
&= \mathbb{P}(\text{"}action = up\text{"})(1-\sigma) r_1 - \mathbb{P}(\text{"}action = down\text{"})\sigma r_2 \\
&= \mathbb{P}(\text{"}action = up\text{"})(1-\sigma) R_{action=up} \\
&- \mathbb{P}(\text{"}action = down\text{"})\sigma R_{action=down} = \mathbb{E}_\pi[(1_{action=up} - \sigma) \cdot R] \quad (6)
\end{aligned}$$

and is thus in practice an unbiased estimation of the gradient is $(1_{action=up} - \sigma) \cdot reward$. Lines that implement this part in the code `y = 1 if action == 2 else 0` then `dlogps.append(y - aprob)` then after the computation of (discounted) rewards in variable "discounted_epr" : `epdlogp *= discounted_epr`

Outline

1. AI and deep learning : introduction
2. Building blocks of strategies : the Multi-armed bandit
   - Presentation
   - Basic strategy: ϵ-greedy
   - MAB: optimism
   - MAB: Confidence intervals
   - MAB: policy gradient algorithms
3. Formalization: (Finite) Markov decision processes (MDP), value function and Bellman equation
4. Bellman : policy evaluation, improvement, value iteration
5. Monte Carlo approaches
6. Reinforcement learning frameworks : Open AI Gym
7. Temporal difference learning
8. Learning strategies : Q-learning, double Q-learning, SARSA
   - Q-Learning
   - Double / Dual Q-Learning
   - SARSA and variants

Actor-Critic and variants

- a type of Temporal Difference(TD) approach
- two networks:
  - Actor: chooses actions, responsible for finding the good policy $\pi(\cdot)$ improves it using policy gradient,
  - Critic : computes the value function $V(\cdot)$
- similar in spirit to Generative Adversarial Networks (GANs)

- has several versions: Actor-Critic (AC), Advantage Actor-Critic (A2C), Asynchronous Advantage Actor-Critic (A3C)

- for instance Asynchronous Advantage Actor-Critic (A3C) is on-policy algorithm, can handle continuous state/action spaces, approximates the 'advantage' $A(s, a) = Q(s, a) - V(s)$, uses simultaneously multiple agents ('asynchronous') to interact with copies of the same env. Each agent contributes to the centralized version of the networks and loads this set of parameters periodically.

Actor-Critic : suite

Practice

Adapt the Tensorflow tutorial on "Actor-Critic" and run it (use the 'Cartpole' environment).
Here: actor is a NN with 2 ouputs (the 'logits' that will then pass a softmax to get the strategy); the critic has one outputs (the value).
Training is done once per episode after computation of the rewards.
Site : `https://www.tensorflow.org/tutorials/reinforcement_learning/actor_critic`
Test with another environment.

Practice

"Simple" implementation: use DQN code to implement algo from [1, page 332 "One-step Actor-Critic (episodic)].
Other implementations : see book [12].

Other subjects RL

- Sometimes the value function is a distribution not a value cf. QR-DQN [13] algorithm

- Note: beyond that there are simple stochastic search algorithms (cf. cross-entropy method (CEM) https://en.wikipedia.org/wiki/Cross-entropy_method)

References I

[1] Richard S. Sutton and Andrew G. Barto.
Reinforcement learning. An introduction.
Adapt. Comput. Mach. Learn. Cambridge, MA: MIT Press, 2nd expanded and updated edition edition, 2018.
http://incompleteideas.net/book/the-book-2nd.html.

[2] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer.
Finite-time Analysis of the Multiarmed Bandit Problem.
Machine Learning, 47(2):235–256, May 2002.

[3] Gabriel Turinici.
The convergence of the Stochastic Gradient Descent (SGD) : a self-contained proof.
arXiv:2103.14350 [cs, math, stat], March 2021.

References II

[4] Lex Weaver and Nigel Tao.
The optimal reward baseline for gradient-based reinforcement learning, 2013.
arxiv:1301.2315.

[5] Evan Greensmith, Peter L. Bartlett, and Jonathan Baxter.
Variance reduction techniques for gradient estimates in reinforcement learning.
J. Mach. Learn. Res., 5:1471–1530, dec 2004.

[6] Stuart Jonathan Russell and Peter Norvig.
Artificial Intelligence: A Modern Approach.
Pearson, 2021.

[7] Frozen Lake - Gym Documentation.
https://www.gymlibrary.ml/environments/toy_text/frozen_lake/.

References III

[8] Deeplizzard www page.
https://deeplizard.com/learn/video/HGeI30uATws.

[9] Harm van Seijen, Hado van Hasselt, Shimon Whiteson, and Marco Wiering.
A theoretical and empirical analysis of Expected Sarsa.
In 2009 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning, pages 177–184, March 2009.
ISSN: 2325-1867.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller.
Playing atari with deep reinforcement learning, 2013.

[11] Deep Reinforcement Learning: Pong from Pixels.
https://karpathy.github.io/2016/05/31/rl/ Code on Gitbuh : karpathy/pg-pong.py.

References IV

[12] L. Graesser and W.L. Keng.
Foundations of Deep Reinforcement Learning.
Addison-Wesley Data & Analytics Series. Pearson Education, 2019.

[13] Will Dabney, Mark Rowland, Marc G. Bellemare, and RÃ©mi Munos.
Distributional reinforcement learning with quantile regression, 2017.